Design Debugging Using the SignalTap II Logic Analyzer 13

2014.06.30

QII53009 Subscribe Send Feedback

About the SignalTap II Logic Analyzer

Altera provides the SignalTap® II Logic Analyzer to help with design debugging. This logic analyzer allows you to examine the behavior of internal signals, without using extra I/O pins, while the design is running at full speed on an FPGA device.

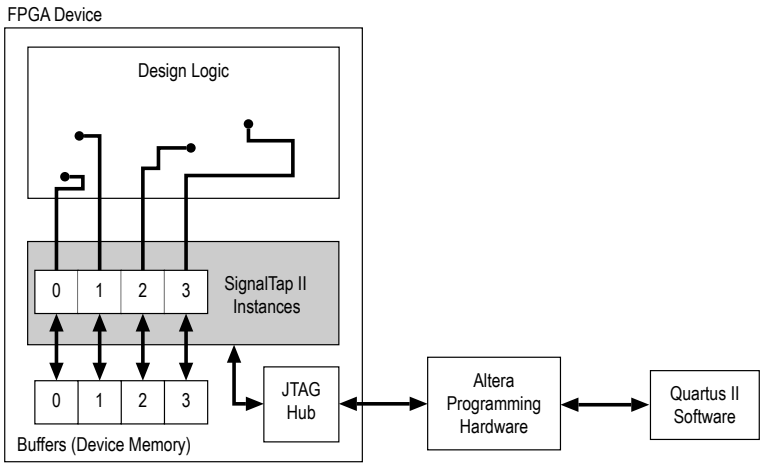
The SignalTap II Logic Analyzer is scalable, easy to use, and available as a stand-alone package or included with the Quartus® II software subscription. This logic analyzer helps debug an FPGA design by probing the state of the internal signals in the design without the use of external equipment. Defining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

The SignalTap II Logic Analyzer is a next-generation, system-level debugging tool that captures and displays real-time signal behavior in a system-on-a-programmable-chip (SOPC) or any FPGA design. The SignalTap II Logic Analyzer supports the highest number of channels, largest sample depth, and fastest clock speeds of any logic analyzer in the programmable logic market.

© 2014 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, ENPIRION, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

ISO 9001:2008 Registered

Figure 13-1: SignalTap II Logic Analyzer Block Diagram



Note to figure :

1. This diagram assumes that you compiled the SignalTap II Logic Analyzer with the design as a separate design partition using the Quartus II incremental compilation feature. This is the default setting for new projects in the Quartus II software. If incremental compilation is disabled or not used, the SignalTap II logic is integrated 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 SignalTap II 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 SignalTap II Logic Analyzer, knowledge of the Quartus II incremental compilation feature is helpful.

Related Information

Quartus II Incremental Compilation for Hierarchical and Team-Based Design documentation

Hardware and Software Requirements

You need the following components to perform logic analysis with the SignalTap II Logic Analyzer:

- Software:
  - Quartus II design software
    - or
  - Quartus II Web Edition (with the TalkBack feature enabled)
    - or
  - SignalTap II Logic Analyzer standalone software and standalone Programmer software.
- Download/upload cable
- Altera® development kit or your design board with JTAG connection to device under test

Note: The Quartus II software Web Edition does not support the SignalTap II Logic Analyzer with the incremental compilation feature.

The memory blocks of the device store captured data and transfers the data to the Quartus II software waveform display with a JTAG communication cable, such as EthernetBlaster or USB-Blaster™.

Table 13-1: SignalTap II Logic Analyzer Features and Benefits

Feature	Benefit
Toolbar with commonly used menu items.	Single-click operation of commonly used menu items. Hover over the icons to see tool tips.
Multiple logic analyzers in a single device	Captures data from multiple clock domains in a design at the same time.
Multiple logic analyzers in multiple devices in a single JTAG chain	Simultaneously captures data from multiple devices in a JTAG chain.
Plug-In Support	Easily specifies nodes, triggers, and signal mnemonics for IP, such as the Nios® II processor.
Up to 10 basic or advanced trigger conditions for each analyzer instance	Enables sending more complex data capture commands to the logic analyzer, providing greater accuracy and problem isolation.
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 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	Modifies the SignalTap II Logic Analyzer monitored signals and triggers without performing a full compilation, saving time.
Incremental Route with Rapid Recompile	Manually allocate trigger input, data input, storage filter input node count, and perform a full compilation to include the SignalTap II Logic Analyzer in your design, then you can selectively connect, disconnect, and swap to different nodes in your design. Use Rapid Recompile to perform incremental routing and gain a 2-4x speedup over the initial full compilation.
Flexible buffer acquisition modes	The buffer acquisition control allows you to precisely control the data that is written into the acquisition buffer. Both segmented buffers and non-segmented buffers with storage qualification allow you to discard data samples that are not relevant to the debugging of your design.
MATLAB integration with included MEX function	Collects the SignalTap II Logic Analyzer captured data into a MATLAB integer matrix.

Feature	Benefit
Up to 2,048 channels per logic analyzer instance	Samples many signals and wide bus structures.
Up to 128K samples in each device	Captures a large sample set for each channel.
Fast clock frequencies	Synchronous sampling of data nodes using the same clock tree driving the logic under test.
Resource usage estimator	Provides estimate of logic and memory device resources used by SignalTap II Logic Analyzer configurations.
No additional cost	The SignalTap II Logic Analyzer is included with a Quartus II subscription and with the Quartus II Web Edition (with TalkBack enabled).
Compatibility with other on-chip debugging utilities	You can use the SignalTap II Logic Analyzer in tandem with any JTAG-based on-chip debugging tool, such as an In-System Memory Content editor, allowing you to change signal values in real-time while you are running an analysis with the SignalTap II Logic Analyzer.

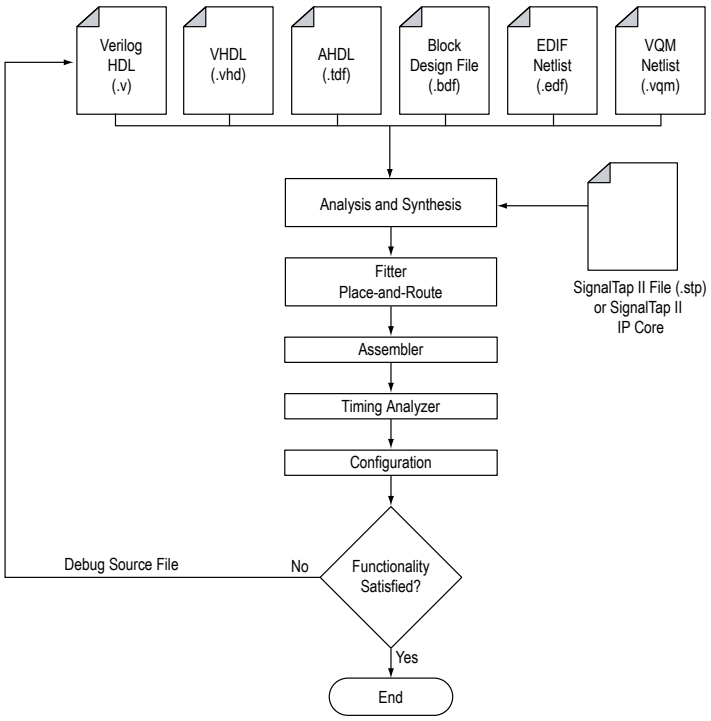
Related Information

System Debugging Tools Overview documentation
Overview and comparison of all tools available in the In-System Verification Tool set

Design Flow Using the SignalTap II Logic Analyzer

Figure 13-2 shows a typical overall FPGA design flow using the SignalTap II Logic Analyzer. A SignalTap II file (**.stp**) is added to and enabled in your project, or a SignalTap II IP core, created with the IP Catalog, is instantiated in your design. Figure 13-2 shows the flow of operations from initially adding the SignalTap II Logic Analyzer to your design to final device configuration, testing, and debugging.

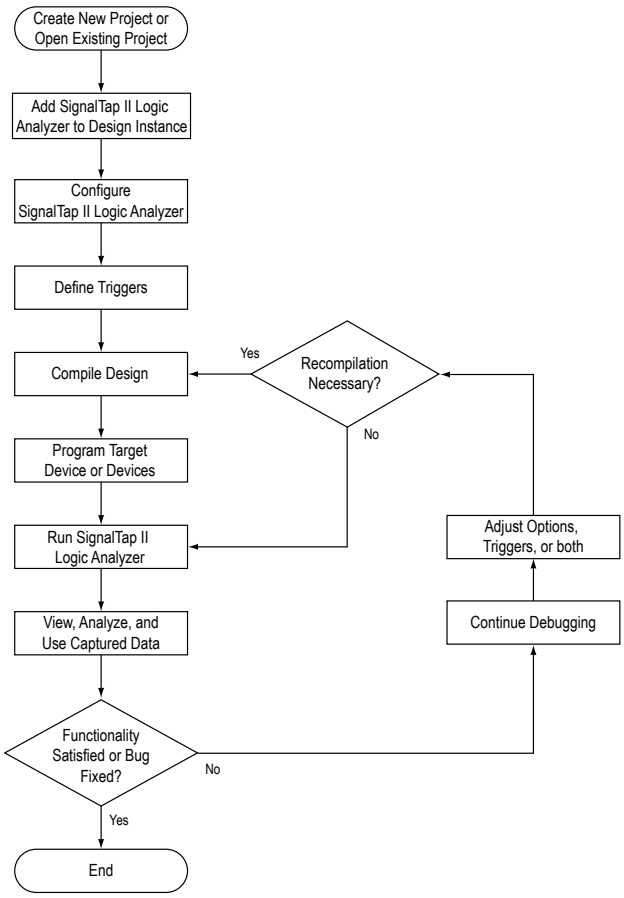
Figure 13-2: SignalTap II FPGA Design and Debugging Flow



SignalTap II Logic Analyzer Task Flow

To use the SignalTap II Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer. **Figure 13-3** shows a typical flow of the tasks you complete to debug your design.

Figure 13-3: SignalTap II Logic Analyzer Task Flow



Add the SignalTap II Logic Analyzer to Your Design

Create an **.stp** or create a parameterized HDL instance representation of the logic analyzer using the IP Catalog. 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.

Related Information

Setting Up the SignalTap II Logic Analyzer online help

Configure the SignalTap II Logic Analyzer

After you add the SignalTap II Logic Analyzer to your design, configure the logic analyzer to monitor the signals you want. You can manually add signals or use a plug-in, such as the Nios II processor plug-in, to quickly add entire sets of associated signals for a particular intellectual property (IP). You can also specify settings for the data capture buffer, such as its size, the method in which data is captured and stored, and the device memory type to use for the buffer in devices that support memory type selection.

Related Information

Creating a Power-Up Trigger on page 13-43

Define Trigger Conditions

The SignalTap II Logic Analyzer captures data continuously while the logic analyzer is running. To capture and store specific signal data, set up triggers that tell the logic analyzer under what conditions to stop capturing data. The SignalTap II Logic Analyzer allows you to define trigger conditions that range from very simple, such as the rising edge of a single signal, to very complex, involving groups of signals, extra logic, and multiple conditions. Power-Up Triggers allow you to capture data from trigger events occurring immediately after the device enters user-mode after configuration.

Compile the Design

With the **.stp** configured and trigger conditions defined, compile your project as usual to include the logic analyzer in your design. Because you may need to change monitored signal nodes or adjust trigger settings frequently during debugging, Altera recommends that you use the incremental compilation feature built into the SignalTap II Logic Analyzer, along with Quartus II incremental compilation, to reduce recompile times. You can also use Incremental Route with Rapid Recompile to reduce recompile times.

Related Information

Compiling a Design that Contains a SignalTap II Logic Analyzer online help

Program the Target Device or Devices

When you debug a design with the SignalTap II Logic Analyzer, you can program a target device directly from the **.stp** without using the Quartus II Programmer. You can also program multiple devices with different designs and simultaneously debug them.

Note: The SignalTap II Logic Analyzer supports all current Altera FPGA device families including Arria®, Cyclone®, and Stratix® devices.

Related Information

- **Managing Multiple SignalTap II Files and Configurations** on page 13-23
- **Running the SignalTap II Logic Analyzer online help**

Run the SignalTap II Logic Analyzer

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

Related Information

Analyzing Data in the SignalTap II Logic Analyzer online help

View, Analyze, and Use Captured Data

After you have captured data and read it into the **.stp**, that data is available for analysis and debugging. Set up mnemonic tables, either manually or with a plug-in, to simplify reading and interpreting the captured signal data. To speed up debugging, use the **Locate** feature in the **SignalTap II node** list to find the locations

of problem nodes in other tools in the Quartus II software. Save the captured data for later analysis, or convert the data to other formats for sharing and further study.

Embedding Multiple Analyzers in One FPGA

The SignalTap II Logic Analyzer Editor includes support for adding multiple logic analyzers by creating instances in the **.stp**. You can create a unique logic analyzer for each clock domain in the design.

Monitoring FPGA Resources Used by the SignalTap II Logic Analyzer

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

You can see resource usage of each logic analyzer instance and total resources used in the columns of the **Instance Manager** pane of the SignalTap II Logic Analyzer Editor. Use this feature when you know that your design is running low on resources.

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

Using the IP Catalog to Create Your Logic Analyzer

You can create a SignalTap II Logic Analyzer instance by using the IP Catalog. The IP Catalog generates an HDL file that you instantiate in your design.

Note: The State-based trigger flow, the state machine debugging feature, and the storage qualification feature are not supported when using the IP Catalog to create the logic analyzer.

Configure the SignalTap II Logic Analyzer

There are many ways to configure instances of the SignalTap II Logic Analyzer. Some of the settings are similar to those found on traditional external logic analyzers. Other settings are unique to the SignalTap II Logic Analyzer because of the requirements for configuring a logic analyzer. All settings allow you to configure the logic analyzer the way you want to help debug your design.

Note: Some settings can only be adjusted when you are viewing Run-Time Trigger conditions instead of Power-Up Trigger conditions.

Assigning an Acquisition Clock

Assign a clock signal to control the acquisition of data by the SignalTap II Logic Analyzer. The logic analyzer samples data on every positive (rising) edge of the acquisition clock. The logic analyzer does not support sampling on the negative (falling) edge of the acquisition clock. You can use any signal in your design as the acquisition clock. However, for best results, Altera recommends that you use a global, non-gated clock synchronous to the signals under test for data acquisition. Using a gated clock as your acquisition clock can result in unexpected data that does not accurately reflect the behavior of your design. The Quartus II static timing analysis tools show the maximum acquisition clock frequency at which you can run your design. Refer to the Timing Analysis section of the Compilation Report to find the maximum frequency of the logic analyzer clock.

Note: Altera recommends that you exercise caution when using a recovered clock from a transceiver as an acquisition clock for the SignalTap II Logic Analyzer. Incorrect or unexpected behavior has been noted, particularly when a recovered clock from a transceiver is used as an acquisition clock with the power-up trigger feature.

If you do not assign an acquisition clock in the SignalTap II Logic Analyzer Editor, the Quartus II software automatically creates a clock pin called `auto_stp_external_clk`.

You must make a pin assignment to this pin independently from the design. Ensure that a clock signal in your design drives the acquisition clock.

Related Information

- **Working with Nodes in the SignalTap II Logic Analyzer online help**
  Information about assigning an acquisition clock
- **Managing Device I/O Pins documentation**
  Information about assigning signals to pins

Adding Signals to the SignalTap II File

While configuring the logic analyzer, add signals to the node list in the **.stp** to select which signals in your design you want to monitor. You can also select signals to define triggers. You can assign the following two types of signals to your **.stp** file:

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

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. Source file changes appear in the Node Finder after you perform an Analysis and Elaboration. On the Processing Menu, point to **Start** and click **Start Analysis & Elaboration**.

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

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

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

When adding pre-synthesis signals, make all connections to the SignalTap II Logic Analyzer before synthesis. Logic and routing resources are allocated during recompilation to make the connection as if a change in

your design files had been made. Pre-synthesis signal names for signals driving to and from IOEs coincide with the signal names assigned to the pin.

In the case of post-fit signals, connections that you make to the SignalTap II Logic Analyzer are the signal names from the actual atoms in your post-fit netlist. You can only make a connection if the signals are part of the existing post-fit netlist and existing routing resources are available from the signal of interest to the SignalTap II Logic Analyzer. In the case of post-fit output signals, tap the `COMBOUT` or `REGOUT` signal that drives the IOE block. For post-fit input signals, signals driving into the core logic coincide with the signal name assigned to the pin.

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

Related Information

- **Faster Compilations with Quartus II Incremental Compilation** on page 13-46
- **Incremental Compilation online help**
- **Analyzing Designs with Quartus II Netlist Viewers documentation**
  Information about cross-probing to source design files and other Quartus II windows

Signal Preservation

Many of the RTL signals are optimized during the process of synthesis and place-and-route. RTL signal names frequently may not appear in the post-fit netlist after optimizations. For example, the compilation process can add tildes ("~") to nets that fan-out from a node, making it difficult to decipher which signal nets they actually represent. These process results can cause problems when you use the incremental compilation flow with the SignalTap II Logic Analyzer. Because you can only add post-fitting signals to the SignalTap II 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. When the Quartus II software encounters these synthesis attributes, it does not perform any optimization on the specified signals, forcing them to continue to exist in the post-fit netlist. However, if you do this, you could see an increase in resource utilization or a decrease in timing performance. The two attributes you can use are:

- `keep`—Ensures that combinational signals are not removed
- `preserve`—Ensures that registers are not removed

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

If you use incremental compilation flow with the SignalTap II Logic Analyzer, pre-synthesis nodes may not be connected to the SignalTap II Logic Analyzer if the affected partition is of the post-fit type. A critical warning is issued for all pre-synthesis node names that are not found in the post-fit netlist.

Related Information

- **Quartus II Integrated Synthesis documentation**
  Information about using signal preservation attributes

- **Working with Nodes in the SignalTap II Logic Analyzer online help**

Assigning Data Signals Using the Technology Map Viewer

You can easily add post-fit signal names that you find in the Technology map viewer. To do so, launch the Technology map viewer (post-fitting) after compiling your design. When you find the desired node, copy the node to either the active **.stp** for your design or a new **.stp**

Node List Signal Use Options

When a signal is added to the node list, you can select options that specify how the signal is used with the logic analyzer. You can turn off the ability of a signal to trigger the analyzer by disabling the **Trigger Enable** option for that signal in the node list in the **.stp**. This option is useful when you want to see only the captured data for a signal and you are not using that signal as part of a trigger.

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

Related Information

Define Triggers on page 13-24

Disabling and Enabling a SignalTap II Instance

From the **Instance Manager** pane, you can disable and enable SignalTap II instances. Physically adding or removing instances requires recompilation after disabling and enabling a SignalTap II instance.

Untappable Signals

Not all of the post-fitting signals in your design are available in the **SignalTap II : post-fitting filter** in the **Node Finder** dialog box. The following signal types cannot be tapped:

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

Adding Signals with a Plug-In

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

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

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

- **Nios II Disassembly** (**Data** tab)—Displays disassembled code from the corresponding address.

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

1. Right-click in the node list. On the Add Nodes with Plug-In submenu, choose the plug-in you want to use, such as the included plug-in named **Nios II**.

   Note: If the IP for the selected plug-in does not exist in your design, a message informs you that you cannot use the selected plug-in.

2. The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design. Select the IP that contains the signals you want to monitor with the plug-in and click **OK**.
3. If all the signals in the plug-in are available, a dialog box might appear, depending on the plug-in selected, where you can specify options for the plug-in. With the Nios II plug-in, you can optionally select an **.elf** containing program symbols from your Nios II Integrated Development Environment (IDE) software design. Specify options for the selected plug-in as desired and click **OK**.

   Note: To make sure all the required signals are available, in the Quartus II **Analysis & Synthesis** settings, turn on **Create debugging nodes for IP cores**.

   All the signals included in the plug-in are added to the node list.

Related Information

- **Define Triggers** on page 13-24
- **View, Analyze, and Use Captured Data** on page 13-7

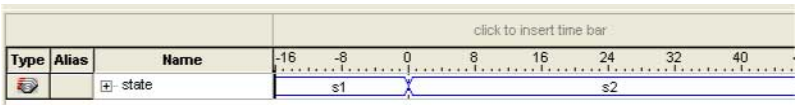
Adding Finite State Machine State Encoding Registers

Finding the signals to debug Finite State Machines (FSM) can be challenging. Finding nodes from the post-fit netlist may be impossible, as FSM encoding signals may be changed or optimized away during synthesis and place-and-route. If you can find all of the relevant nodes in the post-fit netlist or you used the nodes from the pre-synthesis netlist, an additional step is required to find and map FSM signal values to the state names that you specified in your HDL.

The SignalTap II Logic Analyzer GUI can detect FSMs in your compiled design. The SignalTap II Logic Analyzer configuration automatically tracks the FSM state signals as well as state encoding through the compilation process. Shortcut menu commands from the SignalTap II Logic Analyzer GUI allow you to add all of the FSM state signals to your logic analyzer with a single command. For each FSM added to your SignalTap II configuration, the FSM debugging feature adds a mnemonic table to map the signal values to the state enumeration that you provided in your source code. The mnemonic tables enable you to visualize state machine transitions in the waveform viewer. The FSM debugging feature supports adding FSM signals from both the pre-synthesis and post-fit netlists.

Figure 13-4: Decoded FSM Mnemonics

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



Related Information

- **Recommended HDL Coding Styles documentation**
  Coding guidelines for specifying FSM in Verilog and VHDL
- **Setting Up the SignalTap II Logic Analyzer online help**
  Information about adding FSM signals to the configuration file

Modifying and Restoring Mnemonic Tables for State Machines

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

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

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

Related Information

Creating Mnemonics for Bit Patterns on page 13-60

Additional Considerations

The SignalTap II configuration GUI recognizes state machines from your design only if you use Quartus II Integrated Synthesis (QIS). The state machine debugging feature is not able to track the FSM signals or state encoding if you use other EDA synthesis tools.

If you add post-fit FSM signals, the SignalTap II Logic Analyzer FSM debug feature may not track all optimization changes that are a part of the compilation process. If the following two specific optimizations are enabled, the SignalTap II FSM debug feature may not list mnemonic tables for state machines in the design:

- If you have physical synthesis turned on, state registers may be resource balanced (register retiming) to improve f_{MAX}. The FSM debug feature does not list post-fit FSM state registers if register retiming occurs.
- The FSM debugging feature does not list state signals that have been packed into RAM and DSP blocks during QIS or Fitter optimizations.

You can still use the FSM debugging feature to add pre-synthesis state signals.

Specifying the Sample Depth

The sample depth specifies the number of samples that are captured and stored for each signal in the captured data buffer. To specify the sample depth, select the desired number of samples to store in the **Sample Depth** list. The sample depth ranges from 0 to 128K.

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

Capturing Data to a Specific RAM Type

When you use the SignalTap II Logic Analyzer with some devices, you have the option to select the RAM type where acquisition data is stored. Once SignalTap II Logic Analyzer is allocated to a particular RAM block, the entire RAM block becomes a dedicated resource for the logic analyzer. RAM selection allows you to preserve a specific memory block for your design and allocate another portion of memory for SignalTap II Logic Analyzer data acquisition. For example, if your design has an application that requires a large block of memory resources, such a large instruction or data cache, you would choose to use MLAB, M512, or M4k blocks for data acquisition and leave the M9k blocks for the rest of your design.

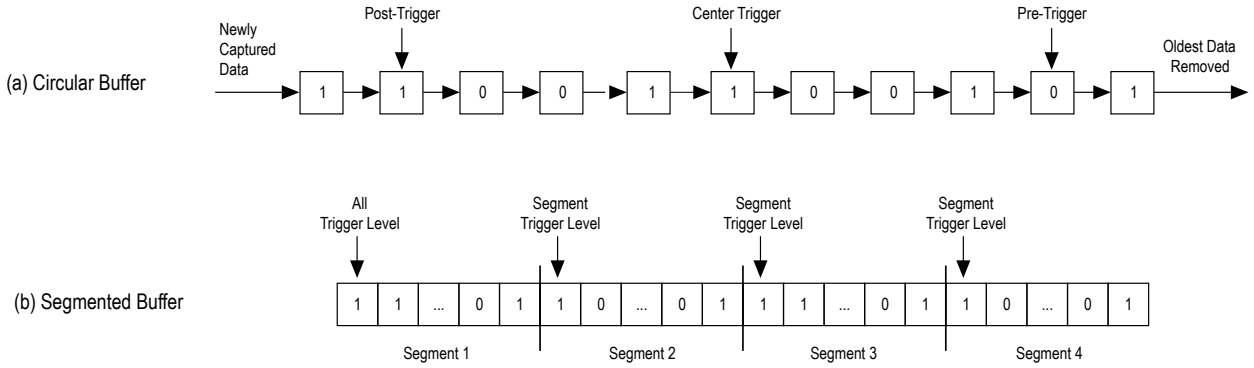
To select the RAM type to use for the SignalTap II Logic Analyzer buffer, select it from the RAM type list. Use this feature when the acquired data (as reported by the SignalTap II resource estimator) is not larger than the available memory of the memory type that you have selected in the FPGA.

Choosing the Buffer Acquisition Mode

The Buffer Acquisition Type Selection feature in the SignalTap II Logic Analyzer lets you choose how the captured data buffer is organized and can potentially reduce the amount of memory that is required for SignalTap II data acquisition. There are two types of acquisition buffer within the SignalTap II Logic Analyzer—a non-segmented buffer and a segmented buffer. With a non-segmented buffer, the SignalTap II Logic Analyzer treats entire memory space as a single FIFO, continuously filling the buffer until the logic analyzer reaches a defined set of trigger conditions. With a segmented buffer, the memory space is split into a number of separate buffers. Each buffer acts as a separate FIFO with its own set of trigger conditions. Only a single buffer is active during an acquisition. The SignalTap II Logic Analyzer advances to the next segment after the trigger condition or conditions for the active segment has been reached.

When using a non-segmented buffer, you can use the storage qualification feature to determine which samples are written into the acquisition buffer. Both the segmented buffers and the non-segmented buffer with the storage qualification feature help you maximize the use of the available memory space. **Figure 13-5** illustrates the differences between the two buffer types.

Figure 13-5: Buffer Type Comparison in the SignalTap II Logic Analyzer



Notes to figure :

1. Both non-segmented and segmented buffers can use a predefined trigger (Pre-Trigger, Center Trigger, Post-Trigger) position or define a custom trigger position using the **State-Based Triggering** tab. Refer to **Specifying the Trigger Position** for more details.
2. Each segment is treated like a FIFO, and behaves as the non-segmented buffer shown in (a).

Related Information

Using the Storage Qualifier Feature on page 13-16

Non-Segmented Buffer

The non-segmented buffer (also known as a circular buffer) shown in **Figure 13-5** (a) is the default buffer type used by the SignalTap II Logic Analyzer. While the logic analyzer is running, data is stored in the buffer until it fills up, at which point new data replaces the oldest data. This continues until a specified trigger event, consisting of a set of trigger conditions, occurs. When the trigger event happens, the logic analyzer continues to capture data after the trigger event until the buffer is full, based on the trigger position setting in the **Signal Configuration** pane or the **.stp**. To capture the majority of the data before the trigger occurs, select **Post trigger position** from the list. To capture the majority of the data after the trigger, select **Pre-trigger position**. To center the trigger position in the data, select **Center trigger position**. Alternatively, use the custom State-based triggering flow to define a custom trigger position within the capture buffer.

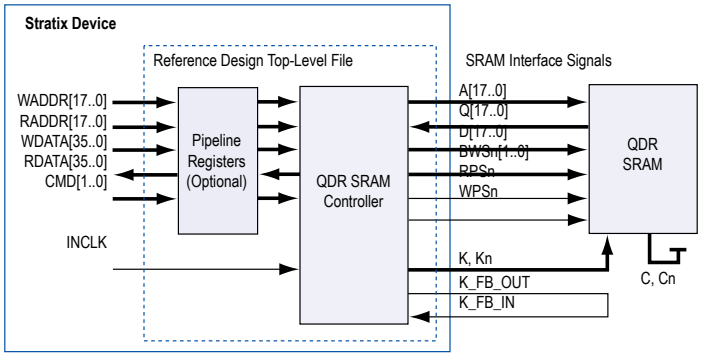
Related Information

Specifying the Trigger Position on page 13-42

Segmented Buffer

A segmented buffer allows you to debug systems that contain relatively infrequent recurring events. The acquisition memory is split into evenly sized segments, with a set of trigger conditions defined for each segment. Each segment acts as a non-segmented buffer. If you want to have separate trigger conditions for each of the segmented buffers, you must use the state-based trigger flow. **Figure 13-6** shows an example of a segmented buffer system. If you want to have separate trigger conditions for each of the segmented buffers, you must use the state-based trigger flow.

Figure 13-6: Example System that Generates Recurring Events



The SignalTap II Logic Analyzer verifies the functionality of the design shown in **Figure 13-6** to ensure that the correct data is written to the SRAM controller. Buffer acquisition in the SignalTap II Logic Analyzer allows you to monitor the `RDATA` port when `H'0F0F0F0F` is sent into the `RADDR` port. You can monitor multiple read transactions from the SRAM device without running the SignalTap II Logic Analyzer again. The buffer acquisition feature allows you to segment the memory so you can capture the same event multiple times without wasting allocated memory. The number of cycles that are captured depends on the number of segments specified under the **Data** settings.

To enable and configure buffer acquisition, select **Segmented** in the SignalTap II Logic Analyzer Editor and select the number of segments to use. In the example in **Figure 13-6**, selecting sixty-four 64-sample segments allows you to capture 64 read cycles when the `RADDR` signal is `H'0F0F0F0F`.

Related Information

Configuring the Trigger Flow in the SignalTap II Logic Analyzer online help
Information about buffer acquisition mode

Using the Storage Qualifier Feature

Both non-segmented and segmented buffers described in the previous section offer a snapshot in time of the data stream being analyzed. The default behavior for writing into acquisition memory with the SignalTap II Logic Analyzer is to sample data on every clock cycle. With a non-segmented buffer, there is one data window that represents a comprehensive snapshot of the datastream. Similarly, segmented buffers use several smaller sampling windows spread out over more time, with each sampling window representing a contiguous data set.

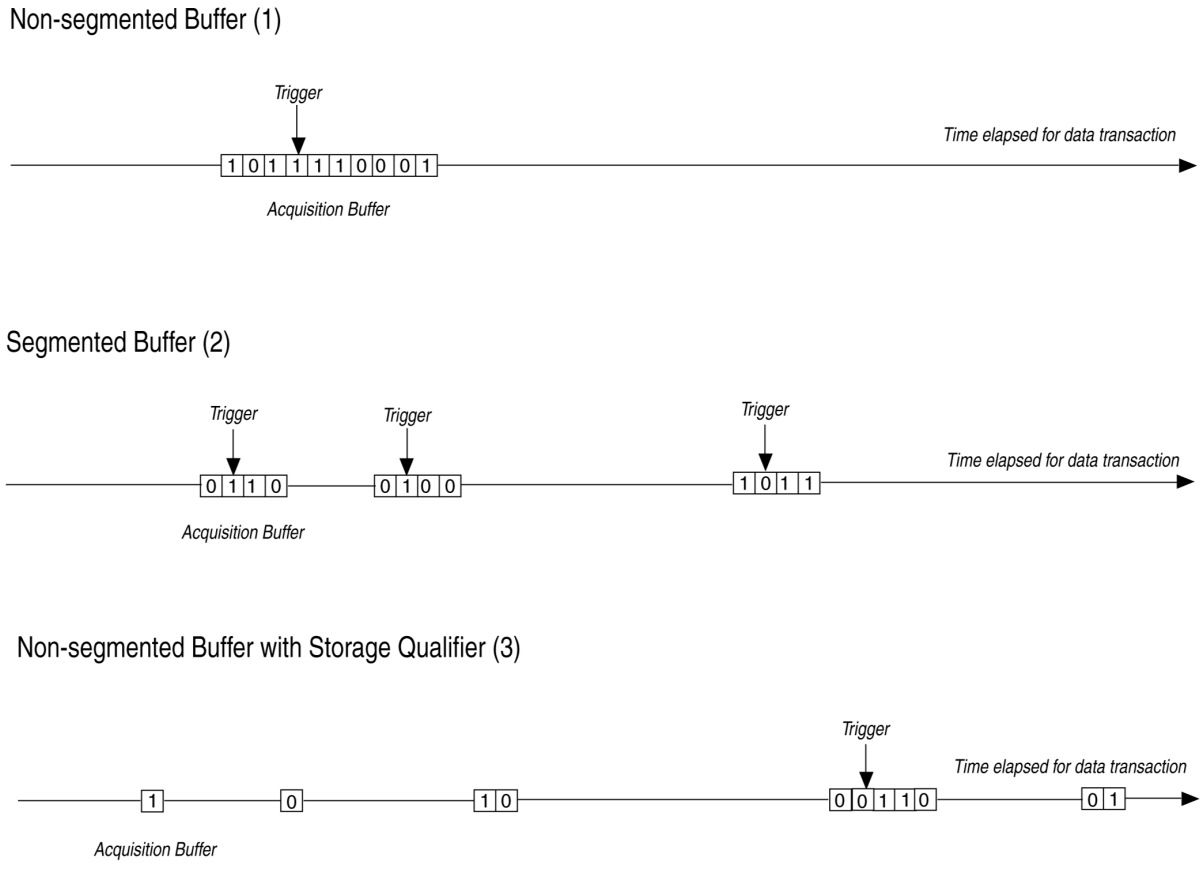
With carefully chosen trigger conditions and a generous sample depth for the acquisition buffer, analysis using segmented and non-segmented buffers captures a majority of functional errors in a chosen signal set. However, each data window can have a considerable amount of redundancy associated with it; for example, a capture of a data stream containing long periods of idle signals between data bursts. With default behavior using the SignalTap II Logic Analyzer, you cannot discard the redundant sample bits.

The Storage Qualification feature allows you to filter out individual samples not relevant to debugging the design. With this feature, a condition acts as a write enable to the buffer during each clock cycle of data acquisition. Through fine tuning the data that is actually stored in acquisition memory, the Storage Qualification feature allows for a more efficient use of acquisition memory in the specified number of samples over a longer period of analysis.

Use of the Storage Qualification feature is similar to an acquisition using a segmented buffer, in that you can create a discontinuity in the capture buffer. Because you can create a discontinuity between any two samples in the buffer, the Storage Qualification feature is equivalent to being able to create a customized segmented buffer in which the number and size of segment boundaries are adjustable.

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

Figure 13-7: Data Acquisition Using Different Modes of Controlling the Acquisition Buffer



Notes to figure :

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

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

- **Continuous**
- **Input port**
- **Transitional**

- **Conditional**
- **Start/Stop**
- **State-based**

Continuous (the default mode selected) turns the Storage Qualification feature off.

Each selected storage qualifier type is active when an acquisition starts. Upon the start of an acquisition, the SignalTap II Logic Analyzer examines each clock cycle and writes the data into the acquisition buffer based upon storage qualifier type and condition. The acquisition stops when a defined set of trigger conditions occur.

Note: Trigger conditions are evaluated independently of storage qualifier conditions. The SignalTap II Logic Analyzer evaluates the data stream for trigger conditions on every clock cycle after the acquisition begins.

The storage qualifier operates independently of the trigger conditions.

Related Information

Define Trigger Conditions on page 13-7

Input Port Mode

When using the Input port mode, the SignalTap II Logic Analyzer takes any signal from your design as an input. When the design is running, if the signal is high on the clock edge, the SignalTap II Logic Analyzer stores the data in the buffer. If the signal is low on the clock edge, the data sample is ignored. A pin is created and connected to this input port by default if no internal node is specified.

If you are using an **.stp** to create a SignalTap II Logic Analyzer instance, specify the storage qualifier signal using the input port field located on the **Setup** tab. You must specify this port for your project to compile.

If you use the IP Catalog flow, 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 13-8 shows a data pattern captured with a segmented buffer. **Figure 13-9** shows a capture of the same data pattern with the storage qualification feature enabled.

Figure 13-8: Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Input port mode)

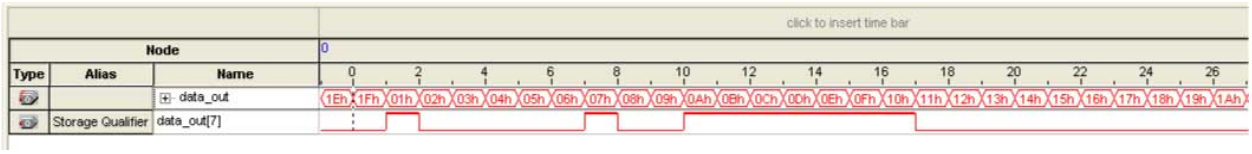
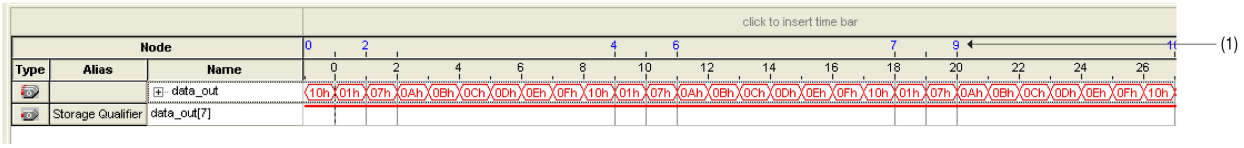


Figure 13-9: Data Acquisition of a Recurring Data Pattern Using an Input Signal as a Storage Qualifier



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

Transitional Mode

In Transitional mode, you choose a set of signals for inspection using the node list check boxes in the **Storage Qualifier** column. During acquisition, if any of the signals marked for inspection have changed since the previous clock cycle, new data is written to the acquisition buffer. If none of the signals marked have changed since the previous clock cycle, no data is stored. Figure 13-10 shows the transitional storage qualifier setup. Figure 13-11 and Figure 13-12 show captures of a data pattern in continuous capture mode and a data pattern using the Transitional mode for storage qualification.

Figure 13-10: Transitional Storage Qualifier Setup

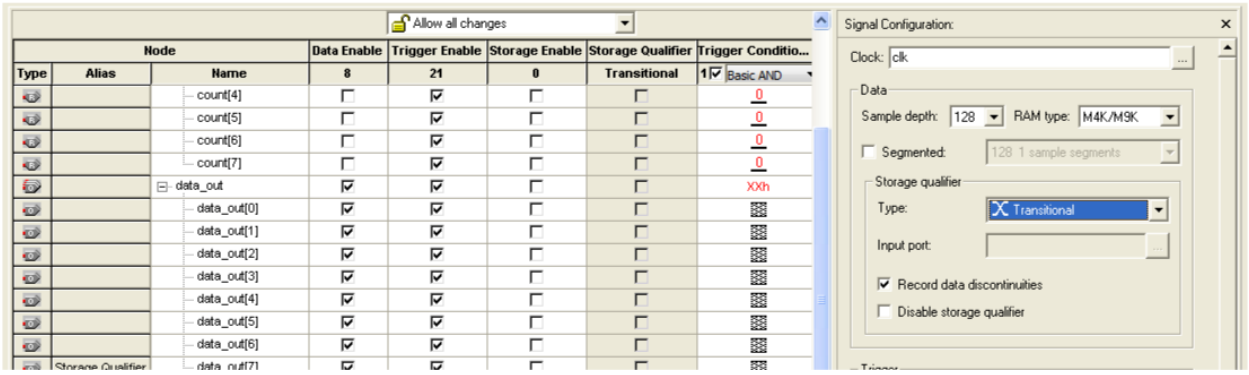


Figure 13-11: Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Transitional mode)

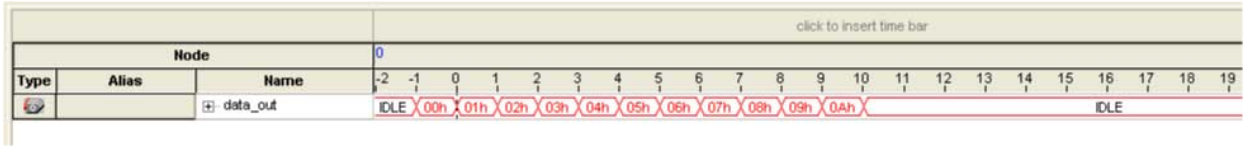
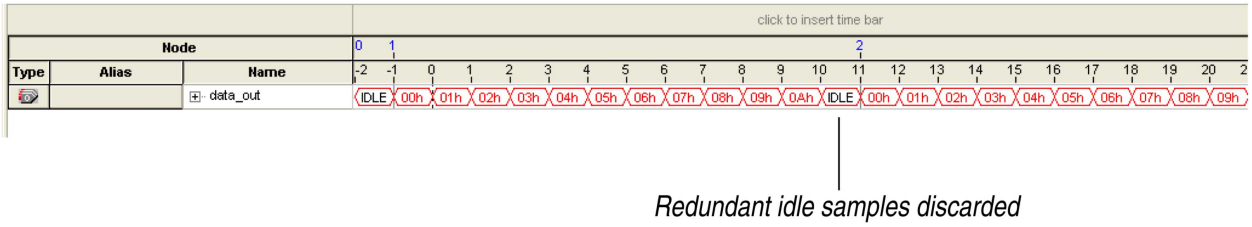


Figure 13-12: Data Acquisition of Recurring Data Pattern Using a Transitional Mode as a Storage Qualifier



Conditional Mode

In Conditional mode, the SignalTap II Logic Analyzer evaluates a combinational function of storage qualifier enabled signals within the node list to determine whether a sample is stored. The SignalTap II Logic Analyzer writes into the buffer during the clock cycles in which the condition you specify evaluates TRUE.

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

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

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

Any other combinational or relational operators that you may want to specify with the enabled signal set for storage qualification can be done with an advanced storage condition. Figure 13-13 details the conditional storage qualifier setup in the **.stp**.

You can specify storage qualification conditions similar to the manner in which trigger conditions are specified.

Figure 13-14 and Figure 13-15 show a data capture with continuous sampling, and the same data pattern using the conditional mode for analysis, respectively.

Figure 13-13: Conditional Storage Qualifier Setup

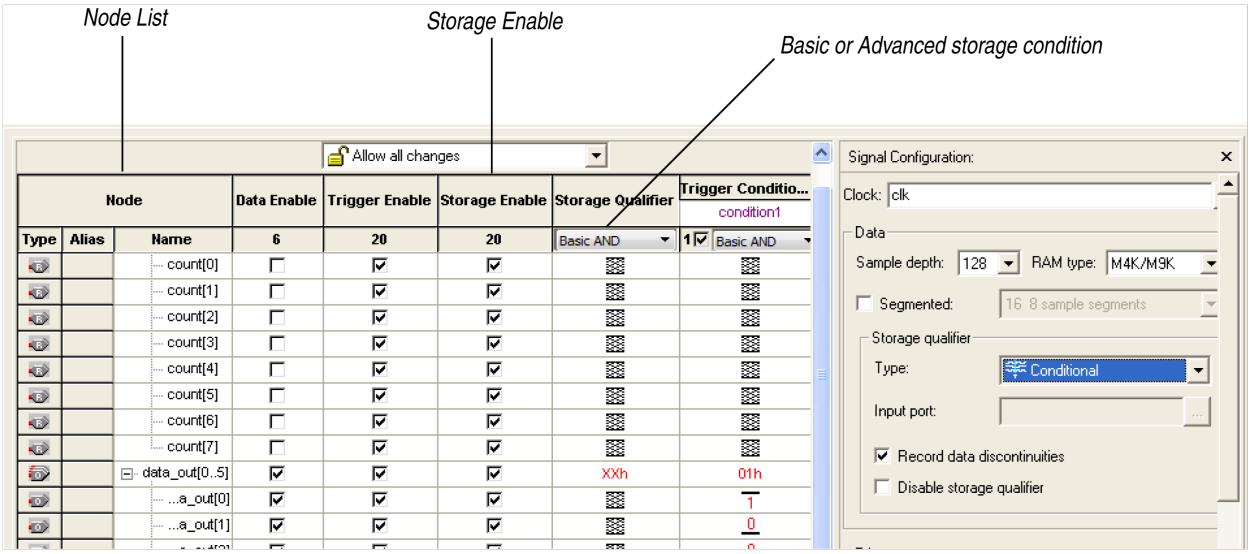
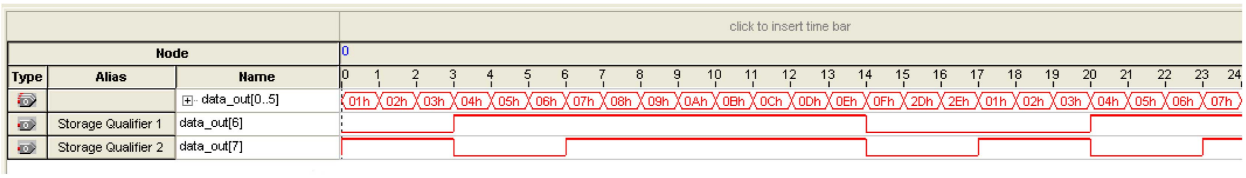
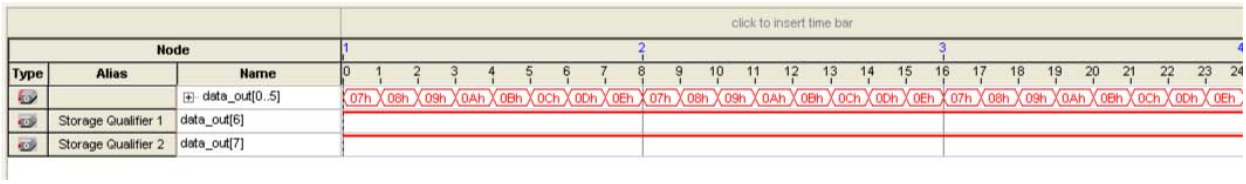


Figure 13-14: Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Conditional capture)



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

Figure 13-15: Data Acquisition of a Recurring Data Pattern in Conditional Capture Mode



Related Information

- **Creating Basic Trigger Conditions** on page 13-25
- **Creating Advanced Trigger Conditions** on page 13-26

Start/Stop Mode

The Start/Stop mode is similar to the Conditional mode for storage qualification. However, in this mode there are two sets of conditions, one for start and one for stop. If the start condition evaluates to TRUE, data

is stored in the buffer every clock cycle until the stop condition evaluates to TRUE, which then pauses the data capture. Additional start signals received after the data capture has started are ignored. If both start and stop evaluate to TRUE at the same time, a single cycle is captured.

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

Figure 13-16 shows the Start/Stop mode storage qualifier setup. Figure 13-17 and Figure 13-18 show captures data pattern in continuous capture mode and a data pattern in using the Start/Stop mode for storage qualification.

Figure 13-16: Start/Stop Mode Storage Qualifier Setup

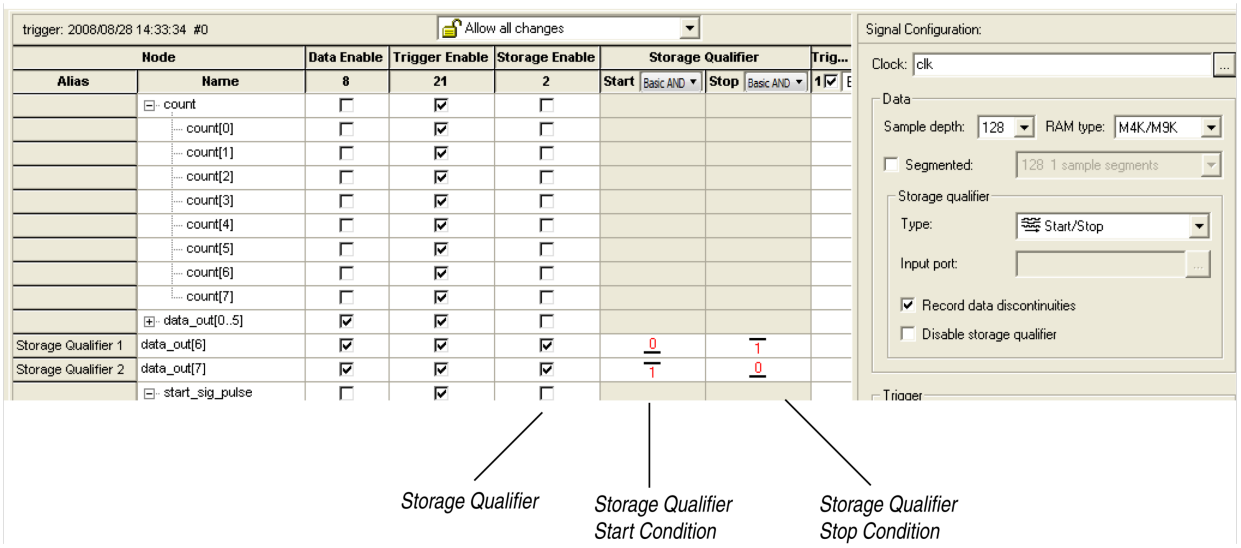


Figure 13-17: Data Acquisition of a Recurring Data Pattern in Continuous Mode (to illustrate Start/Stop mode)

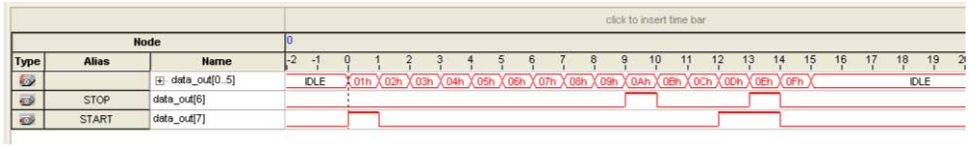
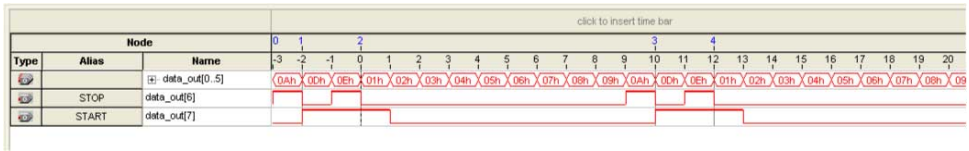


Figure 13-18: Data Acquisition of a Recurring Data Pattern with Start/Stop Storage Qualifier Enabled



State-Based

The State-based storage qualification mode is used with the State-based triggering flow. The state based triggering flow evaluates an if-else based language to define how data is written into the buffer. With the State-based trigger flow, you have command over boolean and relational operators to guide the execution flow for the target acquisition buffer. When the storage qualifier feature is enabled for the State-based flow, two additional commands are available, the `start_store` and `stop_store` commands. These commands operate similarly to the Start/Stop capture conditions described in the previous section. Upon the start of acquisition, data is not written into the buffer until a `start_store` action is performed. The `stop_store` command pauses the acquisition. If both `start_store` and `stop_store` actions are performed within the same clock cycle, a single sample is stored into the acquisition buffer.

Related Information

State-Based Triggering on page 13-32

Showing Data Discontinuities

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

Disable Storage Qualifier

You can turn off the storage qualifier quickly with the **Disable Storage Qualifier** option, and perform a continuous capture. This option is run-time reconfigurable; that is, the setting can be changed without recompiling the project. Changing storage qualifier mode from the **Type** field requires a recompilation of the project.

Related Information

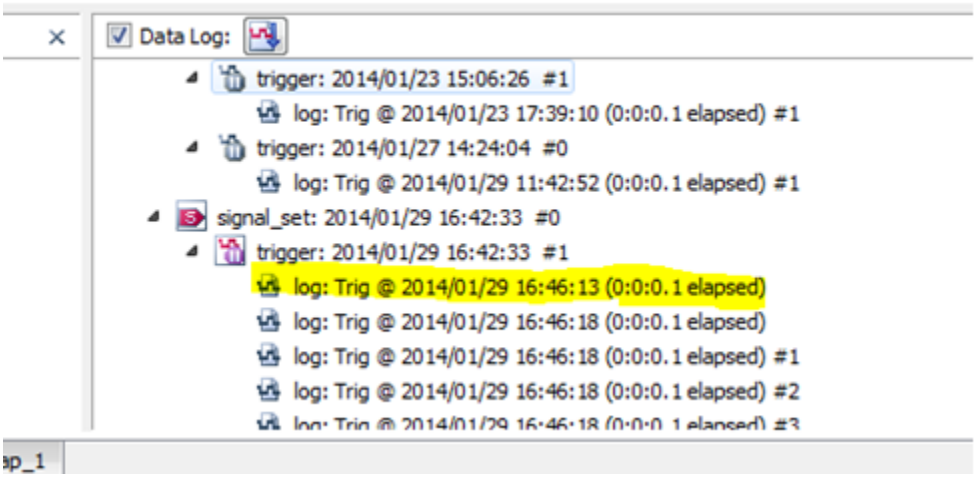
Runtime Reconfigurable Options on page 13-53

Managing Multiple SignalTap II Files and Configurations

You may have more than one **.stp** in one design. Each file potentially has a different group of monitored signals. These signal groups make it possible to debug different blocks in your design. In turn, each group of signals can also be used to define different sets of trigger conditions. Along with each **.stp**, there is also an associated programming file (SRAM Object File [**.sof**]). The settings in a selected SignalTap II file must match the SignalTap II logic design in the associated **.sof** for the logic analyzer to run properly when the device is programmed. Use the Data Log feature and the SOF Manager to manage all of the **.stp** files and their associated settings and programming files.

The Data Log allows you to store multiple SignalTap II configurations within a single **.stp**. **Figure 13-19** shows two signal set configurations with multiple trigger conditions in one **.stp**. To toggle between the active configurations, double-click on an entry in the Data Log. As you toggle between the different configurations, the signal list and trigger conditions change in the **Setup** tab of the **.stp**. The active configuration displayed in the **.stp** is indicated by the blue square around the signal specified in the Data Log. Enable the Data Log by clicking the check box next to **Data Log**. To store a configuration in the Data Log, on the Edit menu, click **Save to Data Log** or click the **Save to Data Log** icon at the top of the Data Log. The time stamping for the Data Log entries display the wall-clock time when SignalTap II triggered and the elapsed time from when acquisition started to when the device triggered.

Figure 13-19: Data Log



The SOF Manager allows you to embed multiple SOFs into one **.stp**. Embedding an SOF in an **.stp** lets you move the **.stp** to a different location, either on the same computer or across a network, without the need to include the associated **.sof** separately. To embed a new SOF in the **.stp**, right-click in the SOF Manager, and click **Attach SOF File** .

Figure 13-20: SOF Manager

As you switch between configurations in the Data Log, you can extract the SOF that is compatible with that particular configuration. You can use the programmer in the SignalTap II Logic Analyzer to download the new SOF to the FPGA, ensuring that the configuration of your **.stp** always matches the design programmed into the target device.

Define Triggers

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

Creating Basic Trigger Conditions

The simplest kind of trigger condition is a basic trigger. Select this from the list at the top of the **Trigger Conditions** column in the node list in the SignalTap II Logic Analyzer Editor. If you select the **Basic AND** or **Basic OR** trigger type, you must specify the trigger pattern for each signal you have added in the **.stp**. To specify the trigger pattern, right-click in the **Trigger Conditions** column and click the desired pattern. Set the trigger pattern to any of the following conditions:

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

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

For more information about creating and using mnemonic tables, refer to **View, Analyze, and Use Captured Data**, and to the Quartus II Help.

For signals added with certain plug-ins, you can create basic triggers easily using predefined mnemonic table entries. For example, with the Nios II plug-in, if you have specified an **.elf** from your Nios II IDE design, you can type the name of a function from your Nios II code. The logic analyzer triggers when the Nios II instruction address matches the address of the specified code function name.

Data capture stops and the data is stored in the buffer when the logical `AND` of all the signals for a given trigger condition evaluates to `TRUE`.

Using the Basic OR Triggering Condition with Nested Groups

When you specify a set of signals as a nested group (group of groups) with the **Basic OR** trigger type, an advanced trigger condition is generated. This advanced trigger condition sorts signals within groups to minimize the need to recompile your design. As long as the parent-child relationships of nodes are kept constant, the generated advanced trigger condition does not change. You can modify the sibling relationships of nodes and not need to recompile your design. The precedence of how this trigger condition is evaluated starts at the bottom-level with the leaf-groups first, then their resulting logic-1 or logic-0 value is used to compute the result of their parent group's logic value. Specifying a value of **TRUE** for a group sets that group's logical result to logic-1 and effectively eliminates all members beneath it from affecting the result of the group trigger. Specifying a value of **FALSE** for a group sets that group's logical result to logic-0 and effectively eliminates all members beneath it from affecting the result of the group trigger.

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

   Note: The OR and AND group trigger conditions are only selectable for groups with no groups as children (bottom-level groups).

Figure 13-21: Creating Nested Groups

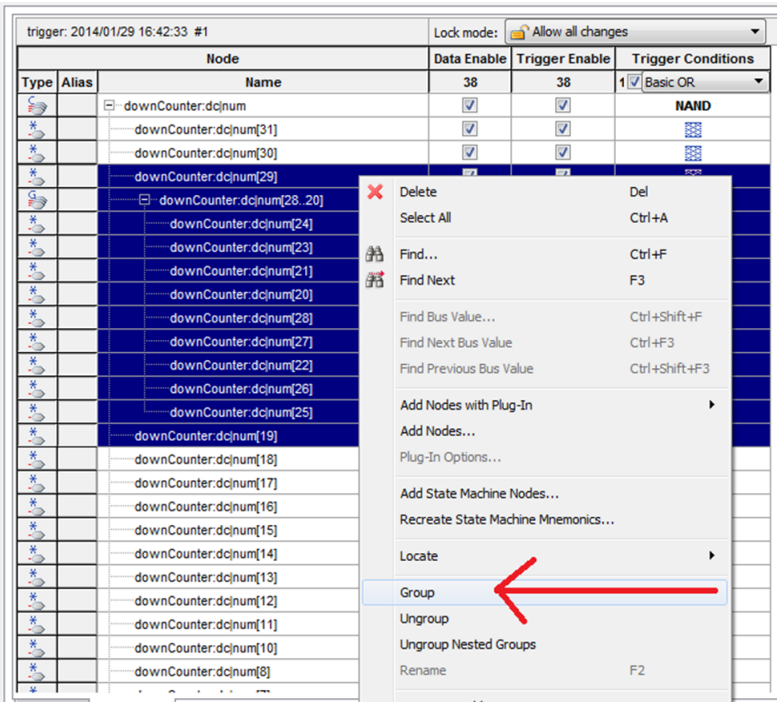
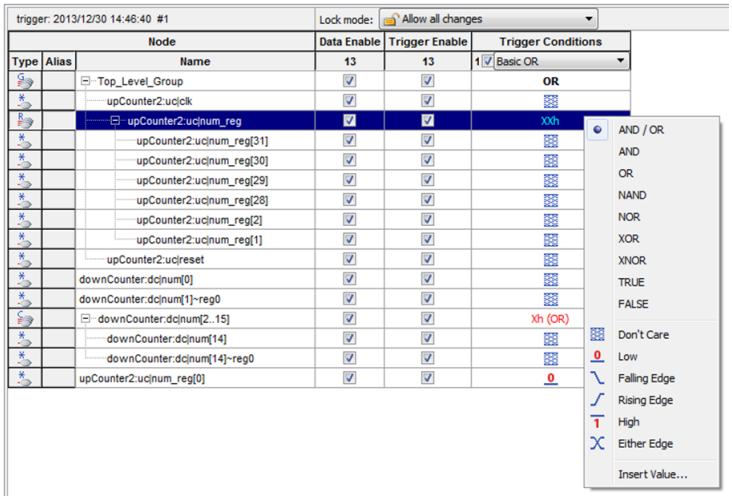


Figure 13-22: Applying Group Trigger Condition



Creating Advanced Trigger Conditions

With the basic triggering capabilities of the SignalTap II Logic Analyzer, you can build more complex triggers with extra logic that enables you to capture data when a combination of conditions exist. If you select the **Advanced** trigger type at the top of the **Trigger Conditions** column in the node list of the SignalTap II Logic Analyzer Editor, a new tab named **Advanced Trigger** appears where you can build a complex trigger

expression using a simple GUI. Drag-and-drop operators into the Advanced Trigger Configuration Editor window to build the complex trigger condition in an expression tree. To configure the operators' settings, double-click or right-click the operators that you have placed and select **Properties**.

Table 13-2: Advanced Triggering Operators

Name of Operator	Type
Less Than	Comparison
Less Than or Equal To	Comparison
Equality	Comparison
Inequality	Comparison
Greater Than	Comparison
Greater Than or Equal To	Comparison
Logical `NOT`	Logical
Logical `AND`	Logical
Logical `OR`	Logical
Logical `XOR`	Logical
Reduction `AND`	Reduction
Reduction `OR`	Reduction
Reduction `XOR`	Reduction
Left Shift	Shift
Right Shift	Shift
Bitwise Complement	Bitwise
Bitwise `AND`	Bitwise
Bitwise `OR`	Bitwise
Bitwise `XOR`	Bitwise
Edge and Level Detector	Signal Detection

Note to table :

1. For more information about each of these operators, refer to the Quartus II Help.

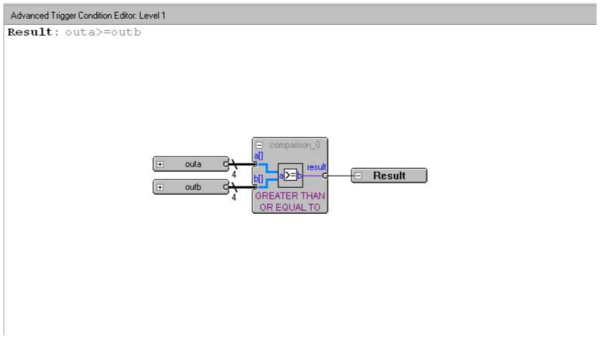
Adding many objects to the Advanced Trigger Condition Editor can make the work space cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the shortcut menu and select **Arrange All Objects**. You can also use the **Zoom-Out** command to fit more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

Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

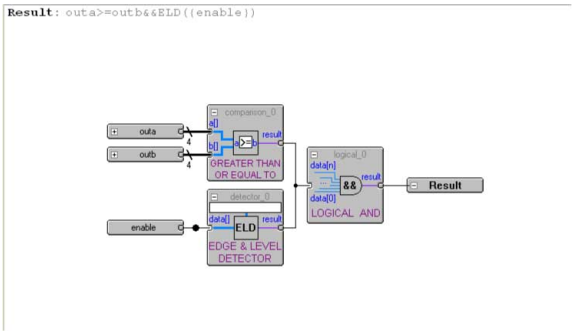
- Trigger when bus outa is greater than or equal to outb.

Figure 13-23: Bus outa Is Greater Than or Equal to Bus outb



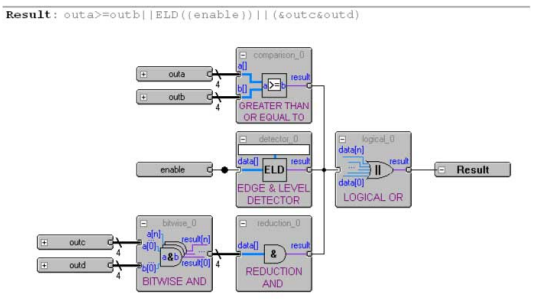
- Trigger when bus outa is greater than or equal to bus outb, and when the enable signal has a rising edge (Figure 13-24).

Figure 13-24: Enable Signal Has a Rising Edge



- 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 (Figure 13-25).

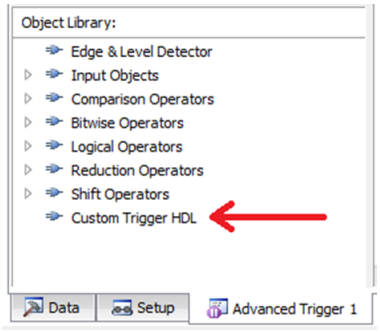
Figure 13-25: Bitwise AND Operation



Custom Trigger HDL Object

The Custom Trigger HDL object found in the **Advanced Trigger** editor allows you to create a customized trigger condition with your own HDL module in either Verilog or VHDL. You can use this object to simulate the behavior of your triggering logic to make sure that the logic itself is not faulty. You can tap specific instances of modules located anywhere in the hierarchy of your design without having to manually route all the necessary connections.

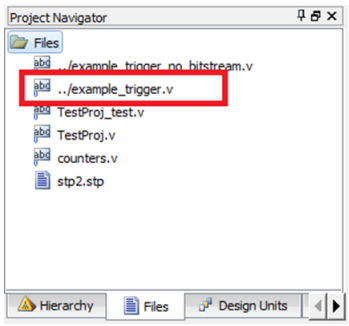
Figure 13-26: Object Library



Custom Trigger Flow

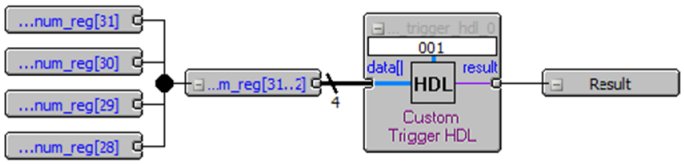
1. Select Advanced for a given trigger-level to make the Advanced Trigger editor active.
2. Prepare your Custom Trigger HDL module. You can either add a new source file to Quartus II that contains the trigger module or append the HDL for your trigger module to a source file already included in Quartus II **Files** under the **Project Navigator**.

Figure 13-27: Project Navigator



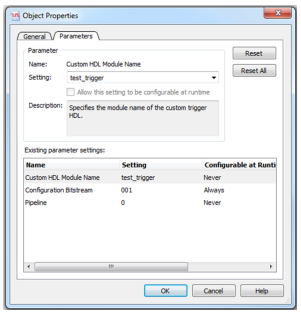
3. Implement the required inputs and outputs for your Custom Trigger HDL module, see **Table 13-3**.
4. Drag in your Custom Trigger HDL object and connect the object's data input bus and result output bit to the final trigger result.

Figure 13-28: Custom Trigger HDL Object



5. Right-click your Custom Trigger HDL object and configure the object's properties, see **Table 13-4**.

Figure 13-29: Configure Object Properties



6. Compile your design.
7. Acquire data with SignalTap II using your custom Trigger HDL object.

Table 13-3: Custom Trigger HDL Module Required Inputs and Outputs

Name	Description	Input/Output	Required/ Optional
`acq_clk`	Acquisition clock used by SignalTap II	Input	Required
`reset`	Reset signal used by SignalTap II when restarting a capture.	Input	Required
`data_in`	• Data input to be connected in the Advanced Trigger editor. • Data your module will use to trigger.	Input	Required
`pattern_in`	• Module's input for the configuration bitstream property. • Runtime configurable property that can be set from SignalTap II GUI to change the behavior of your trigger logic.	Input	Optional
`trigger_out`	Output signal of your module to be asserted when triggering conditions have been met.	Output	Required

Table 13-4: Custom Trigger HDL Module Properties

Property	Description
Custom HDL Module Name	Module name of your triggering logic i.e. module **trigger_foo** (input x, y ...);

Property	Description
Configuration Bitstream	• Allows you to create runtime-configurable trigger logic which can change its behavior based upon the value of the configuration bitstream. • Configuration bitstream property is interpreted as binary and should only contain the characters 1 and 0. The bit-width (number of 1s and 0s) should match the `pattern_in` bit width in Table 13-3. • A blank configuration bitstream implies that your module does not have a `pattern_in` input.
Pipeline	• Tells the advanced trigger editor how many stages of pipeline your triggering logic has. • If it takes three clock cycles after a triggering input is received for the trigger output to be asserted, you can denote a pipeline value of three.

Figure 13-30: Example of Verilog Trigger Using Configuration Bitstream

```
module test_trigger(input acq_clk, reset, input [3:0] data_in, input
[1:0] pattern_in, output reg trigger_out);

    always @ (pattern_in) begin
        case (pattern_in)
            2'b00:
                trigger_out = &data_in;
            2'b01:
                trigger_out = |data_in;
            2'b10:
                trigger_out = 1'b0;
            2'b11:
                trigger_out = 1'b1;
        endcase
    end

endmodule
```

Figure 13-31: Example of Verilog Trigger with No Configuration Bitstream

```
module test_trigger_no_bs(input acq_clk, reset, input [3:0]
data_in, output trigger_out);

    assign trigger_out = &data_in;

endmodule
```

Trigger Condition Flow Control

The SignalTap II Logic Analyzer offers multiple triggering conditions to give you precise control of the method in which data is captured into the acquisition buffers. Trigger Condition Flow allows you to define the relationship between a set of triggering conditions. The SignalTap II Logic Analyzer **Signal Configuration** pane offers two flow control mechanisms for organizing trigger conditions:

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

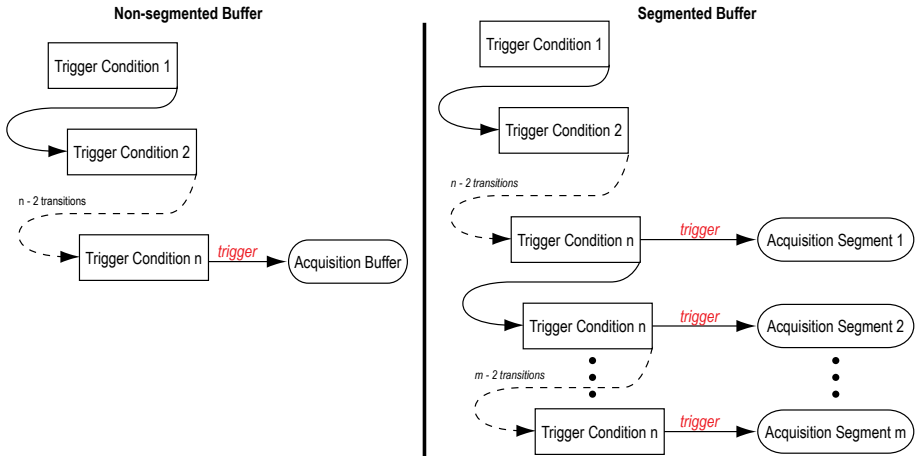
You can use sequential or state based triggering with either a segmented or a non-segmented buffer.

Sequential Triggering

Sequential triggering flow allows you to cascade up to 10 levels of triggering conditions. The SignalTap II Logic Analyzer sequentially evaluates each of the triggering conditions. When the last triggering condition evaluates to TRUE, the SignalTap II Logic Analyzer triggers the acquisition buffer. For segmented buffers, every acquisition segment after the first segment triggers on the last triggering condition that you have specified. Use the Simple Sequential Triggering feature with basic triggers, advanced triggers, or a mix of both. Figure 13-32 illustrates the simple sequential triggering flow for non-segmented and segmented buffers.

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

Figure 13-32: Sequential Triggering Flow



Notes to figure :

1. The acquisition buffer stops capture when all n triggering levels are satisfied, where $n \leq 10$.
2. An external trigger input, if defined, is evaluated before all other defined trigger conditions are evaluated.

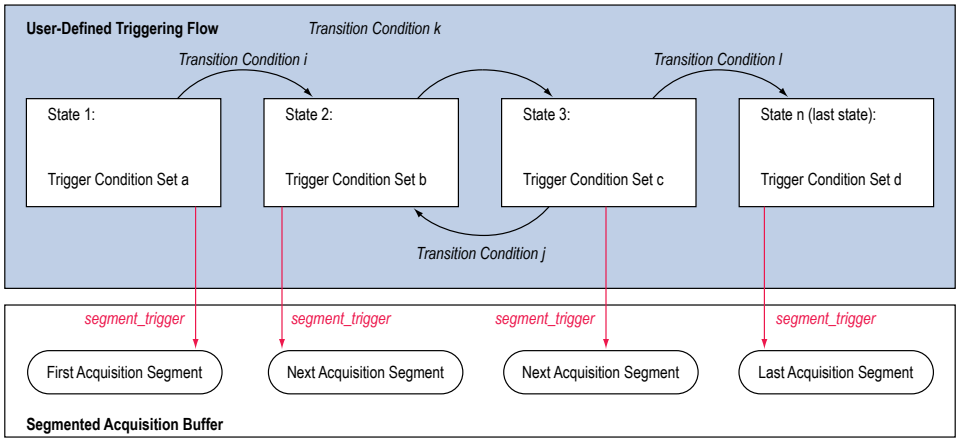
To configure the SignalTap II Logic Analyzer for Sequential triggering, in the SignalTap II editor on the **Trigger flow control** list, select **Sequential**. Select the desired number of trigger conditions from the **Trigger Conditions** list. After you select the desired number of trigger conditions, configure each trigger condition in the node list. To disable any trigger condition, turn on the trigger condition at the top of the column in the node list.

State-Based Triggering

Custom State-based triggering provides the most control over triggering condition arrangement. The State-Based Triggering flow allows you to describe the relationship between triggering conditions precisely, using an intuitive GUI and the SignalTap II Trigger Flow Description Language, a simple description language based upon conditional expressions. Tooltips within the custom triggering flow GUI allow you to describe your desired flow quickly. The custom State-based triggering flow allows for more efficient use of the space available in the acquisition buffer because only specific samples of interest are captured.

Events that trigger the acquisition buffer are organized by a state diagram that you define. All actions performed by the acquisition buffer are captured by the states and all transition conditions between the states are defined by the conditional expressions that you specify within each state.

Figure 13-33: State-Based Triggering Flow



Notes to figure:

1. You are allowed up to 20 different states.
2. An external trigger input, if defined, is evaluated before any conditions in the custom State-based triggering flow are evaluated.

Each state allows you to define a set of conditional expressions. Each conditional expression is a Boolean expression dependent on a combination of triggering conditions (configured within the **Setup** tab), counters, and status flags. Counters and status flags are resources provided by the SignalTap II Logic Analyzer custom-based triggering flow.

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

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

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

This SignalTap II custom State-based triggering flow allows you to capture a sequence of events that may not necessarily be contiguous in time; for example, capturing a communication transaction between two devices that includes a handshaking protocol containing a sequence of acknowledgements.

The **State-Based Trigger Flow** tab is the control interface for the custom state-based triggering flow. To enable this tab, select **State-based** on the **Trigger Flow Control** list. (Note that when **Trigger Flow Control** is specified as **Sequential**, the **State-Based Trigger Flow** tab is hidden.)

The **State-Based Trigger Flow** tab is partitioned into the following three panes:

- **State Diagram** pane
- **Resources** pane
- **State Machine** pane

State Diagram Pane

The **State Diagram** pane provides a graphical overview of the triggering flow that you define. It shows the number of states available and the state transitions between the states. You can adjust the number of available states by using the menu above the graphical overview.

State Machine Pane

The **State Machine** pane contains the text entry boxes where you can define the triggering flow and actions associated with each state. You can define the triggering flow using the SignalTap II Trigger Flow Description Language, a simple language based on "if-else" conditional statements. Tooltips appear when you move the mouse over the cursor, to guide command entry into the state boxes. The GUI provides a syntax check on your flow description in real-time and highlights any errors in the text flow.

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

Related Information

- **SignalTap II Trigger Flow Description Language** on page 13-35
- **SignalTap II Trigger Flow Description Language online help**

Resources Pane

The **Resources** pane allows you to declare Status Flags and Counters for use in the conditional expressions in the Custom Triggering Flow. Actions to decrement and increment counters or to set and clear status flags are performed within the triggering flow that you define.

You can specify up to 20 counters and 20 status flags. Counter and status flags values may be initialized by right-clicking the status flag or counter name after selecting a number of them from the respective pull-down list, and selecting **Set Initial Value.** To specify a counter width, right-click the counter name and select **Set Width**. Counters and flag values are updated dynamically after acquisition has started to assist in debugging your trigger flow specification.

The **configurable at runtime** options in the **Resources** pane allows you to configure the custom-flow control options that can be changed at runtime without requiring a recompilation.

Table 13-5: Runtime Reconfigurable Settings, State-Based Triggering Flow

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

Setting	Description
Comparison operators	Allows you to modify the operators in Boolean expressions at runtime.
Logical operators	Allows you to modify the logical operators in Boolean expressions at runtime.

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

1. On the **Setup** tab, select **Allow trigger condition changes only**.
2. Modify the Trigger Flow conditions in the **Custom Trigger Flow** tab.
3. Click the desired parameter in the text box and select a new parameter from the menu that appears.

Note: Trigger lock mode restricts changes to the configuration settings that have **configurable at runtime** specified. The runtime configurable settings for the **Custom Trigger Flow** tab are on by default. You may get some performance advantages by disabling some of the runtime configurable options.

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. Refer to **Incremental Route with Rapid Recompile** on page 13-48.

Related Information

- **Performance and Resource Considerations** on page 13-51
- **Runtime Reconfigurable Options** on page 13-53

SignalTap II Trigger Flow Description Language

The Trigger Flow Description Language is based on a list of conditional expressions per state to define a set of actions. Each line in the example shows a language format. Keywords are shown in bold. Non-terminals are delimited by "<>" and are further explained in the following sections. Optional arguments are delimited by "[]" .

```
state <State_label>:
<action_list>

if( <Boolean_expression> )
<action_list>
[else if ( <boolean_expression> )
<action_list>]
[else
<action_list>]
```

Note to example :

1. Multiple `else if` conditions are allowed.

The priority for evaluation of conditional statements is assigned from top to bottom. The *<boolean_expression>* in an `if` statement can contain a single event, or it can contain multiple event conditions. The `action_list` within an `if` or an `else if` clause must be delimited by the begin and end tokens when the action list contains multiple statements. When the boolean expression is evaluated `TRUE`, the logic analyzer analyzes all of the commands in the action list concurrently. The possible actions include:

- Triggering the acquisition buffer

- Manipulating a counter or status flag resource
- Defining a state transition

Related Information

Custom Triggering Flow Application Examples on page 13-67

State Labels

State labels are identifiers that can be used in the action `goto`.

`state` <*state_label*>: begins the description of the actions evaluated when this state is reached.

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

Boolean_expression

`Boolean_expression` is a collection of logical operators, relational operators, and their operands that evaluate into a Boolean result. Depending on the operator, the operand can be a reference to a trigger condition, a counter and a register, or a numeric value. Within an expression, parentheses can be used to group a set of operands.

Logical operators accept any boolean expression as an operand.

Table 13-6: Logical Operators

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

Relational operators are performed on counters or status flags. The comparison value, the right operator, must be a numerical value.

Table 13-7: Relational Operators

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

Notes to table :

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

Action_list

Action_list is a list of actions that can be performed when a state is reached and a condition is also satisfied. If more than one action is specified, they must be enclosed by `begin` and `end`. The actions can be categorized as resource manipulation actions, buffer control actions, and state transition actions. Each action is terminated by a semicolon (`;`).

Resource Manipulation Action

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

Table 13-8: Resource Manipulation Action

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

Buffer Control Action

Buffer control actions specify an action to control the acquisition buffer.

Table 13-9: Buffer Control Action

Action	Description	Syntax
`trigger`	Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.	`trigger <post-fill_count>;`
`segment_trigger`	Ends the acquisition of the current segment. The SignalTap II Logic Analyzer starts acquiring from the next segment on evaluating this command. If all segments are filled, the oldest segment is overwritten with the latest sample. The acquisition stops when a trigger action is evaluated. This action cannot be used in non-segmented acquisition mode.	`segment_trigger <post-fill_count>;`
`start_store`	Asserts the `write_enable` to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled.	`start_store`
`stop_store`	De-asserts the `write_enable` signal to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled.	`stop_store`

Both `trigger` and `segment_trigger` actions accept an optional post-fill count argument. If provided, the current acquisition acquires the number of samples provided by post-fill count and then stops acquisition. If no post-count value is specified, the trigger position for the affected buffer defaults to the trigger position specified in the **Setup** tab.

Note: In the case of `segment_trigger`, acquisition of the current buffer stops immediately if a subsequent triggering action is issued in the next state, regardless of whether or not the post-fill count has been satisfied for the current buffer. The remaining unfilled post-count acquisitions in the current buffer are discarded and displayed as grayed-out samples in the data window.

State Transition Action

The State Transition action specifies the next state in the custom state control flow. It is specified by the `goto` command. The syntax is as follows:

```
goto <state_label>;
```

Using the State-Based Storage Qualifier Feature

When you select State-based for the storage qualifier type, the `start_store` and `stop_store` actions are enabled in the State-based trigger flow. These commands, when used in conjunction with the expressions of the State-based trigger flow, give you maximum flexibility to control data written into the acquisition buffer.

Note: The `start_store` and `stop_store` commands can only be applied to a non-segmented buffer.

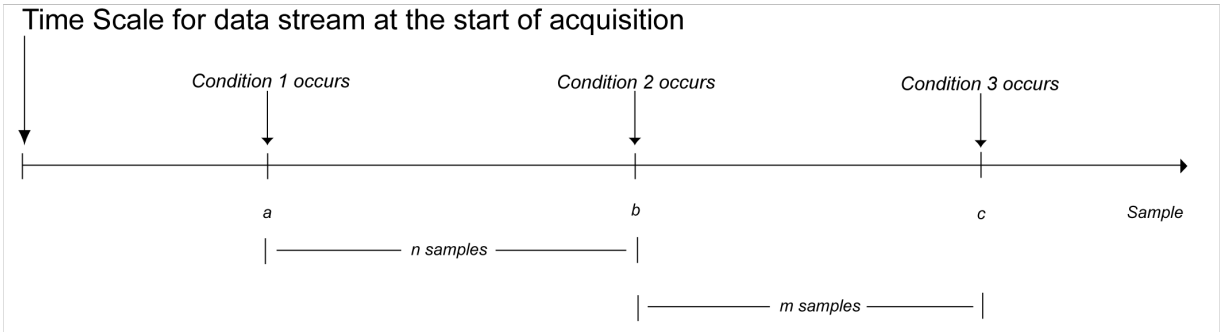
The `start_store` and `stop_store` commands function similar to the start and stop conditions when using the **start/stop** storage qualifier mode conditions. If storage qualification is enabled, the `start_store` command must be issued for SignalTap II to write data into the acquisition buffer. No data is acquired until the `start_store` command is performed. Also, a `trigger` command must be included as part of the trigger flow description. The `trigger` command is necessary to complete the acquisition and display the results on the waveform display.

The following example illustrates the behavior of the State-based trigger flow with the storage qualification commands.

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

Figure 13-34 shows a hypothetical scenario with three trigger conditions that happen at different times after you click **Start Analysis**. The trigger flow description in the example above , when applied to the scenario shown in Figure 13-34, illustrates the functionality of the storage qualification feature for the state-based trigger flow.

Figure 13-34: Capture Scenario for Storage Qualification with the State-Based Trigger Flow



In this example, the SignalTap II Logic Analyzer does not write into the acquisition buffer until sample a, when Condition 1 occurs. Once sample b is reached, the `trigger value` command is evaluated. The logic analyzer continues to write into the buffer to finish the acquisition. The trigger flow specifies a `stop_store` command at sample c, m samples after the trigger point occurs.

The logic analyzer finishes the acquisition and displays the contents of the waveform if it can successfully finish the post-fill acquisition samples before Condition 3 occurs. In this specific case, the capture ends if the post-fill count value is less than *m*.

If the post-fill count value specified in Trigger Flow description 1 is greater than m samples, the buffer pauses acquisition indefinitely, provided there is no recurrence of Condition 1 to trigger the logic analyzer to start capturing data again. The SignalTap II Logic Analyzer continues to evaluate the **stop_store** and **start_store** commands even after the **trigger** command is evaluated. If the acquisition has paused, you can click **Stop Analysis** to manually stop and force the acquisition to trigger. You can use counter values, flags, and the State diagram to help you perform the trigger flow. The counter values, flags, and the current state are updated in real-time during a data acquisition.

Figure 13-35 and **Figure 13-36** show a real data acquisition of the scenario. **Figure 13-35** illustrates a scenario where the data capture finishes successfully. It uses a buffer with a sample depth of 64, $m = n = 10$, and the post-fill count value = 5. **Figure 13-36** illustrates a scenario where the logic analyzer pauses indefinitely even after a trigger condition occurs due to a `stop_store` condition. This scenario uses a sample depth of 64, with $m = n = 10$ and post-fill count = 15.

Figure 13-35: Storage Qualification with Post-Fill Count Value Less than *m* (Acquisition Successfully Completes)

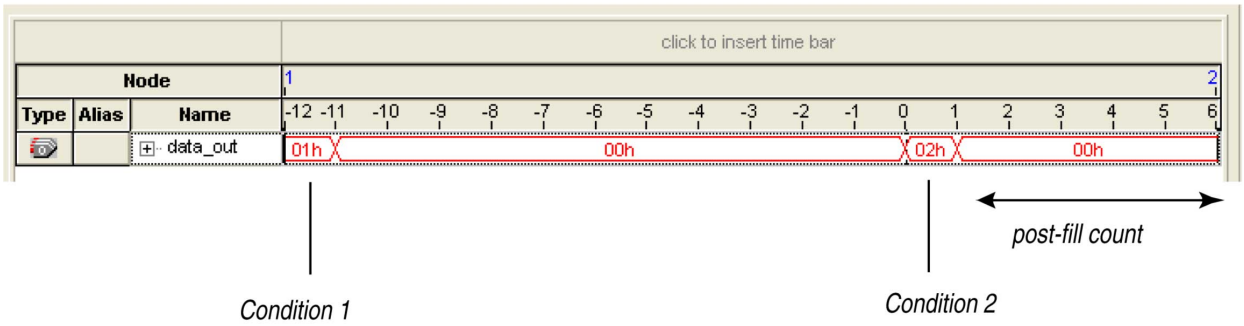


Figure 13-36: Storage Qualification with Post-Fill Count Value Greater than *m* (Acquisition Indefinitely Paused)

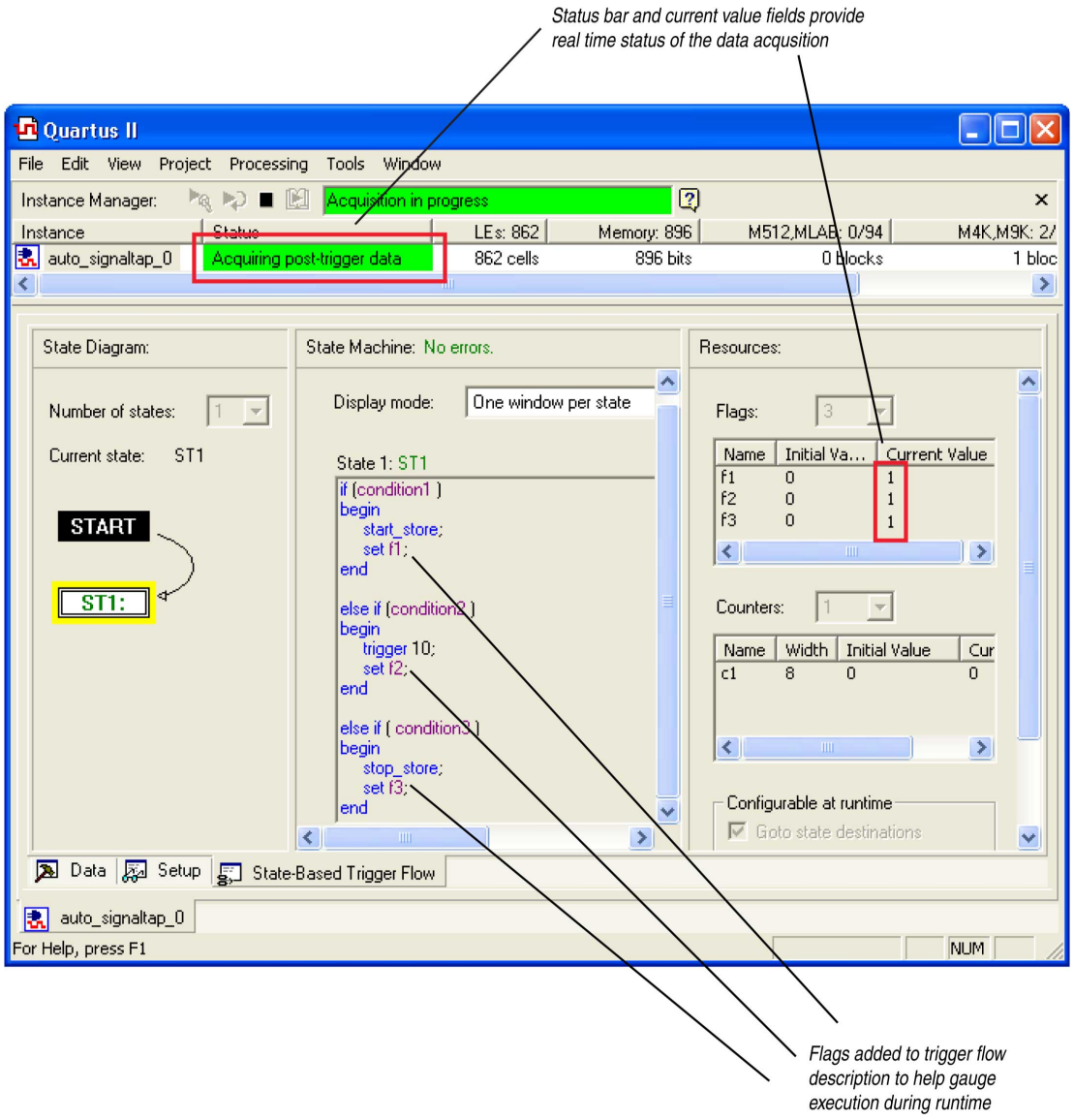
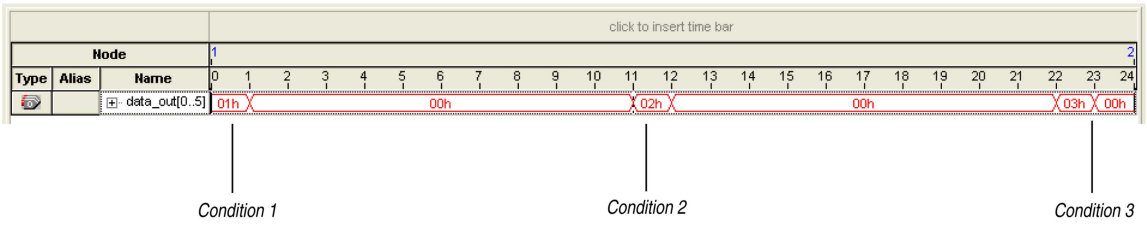


Figure 13-37: Waveform After Forcing the Analysis to Stop



The combination of using counters, Boolean and relational operators in conjunction with the `start_store` and `stop_store` commands can give a clock-cycle level of resolution to controlling the samples that are written into the acquisition buffer. The code example below shows a trigger flow description that skips three clock cycles of samples after hitting condition 1. **Figure 13-38** shows the data transaction on a continuous capture and **Figure 13-40** shows the data capture with the Trigger flow description applied, in the example below.

```
State 1: ST1
start_store
if ( condition1 )
begin
   stop_store;
   goto ST2;
end

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

Figure 13-38: Continuous Capture of Data Transaction for Example 2

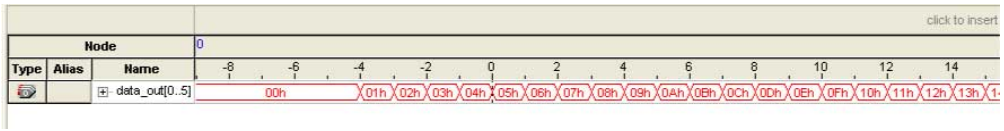
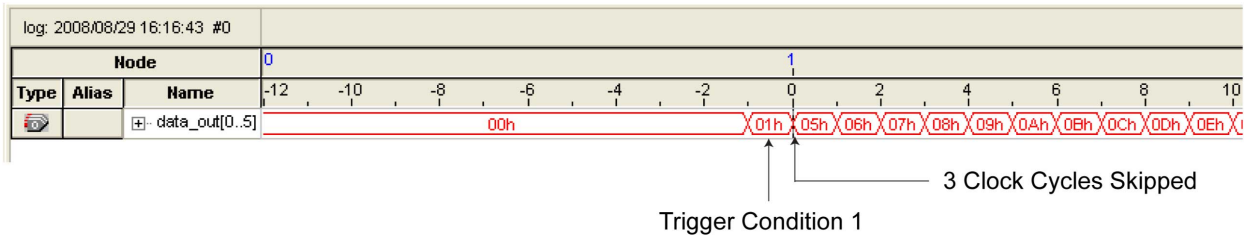


Figure 13-39: Capture of Data Transaction with Trigger Flow Description Applied



Specifying the Trigger Position

The SignalTap II Logic Analyzer allows you to specify the amount of data that is acquired before and after a trigger event. You can specify the trigger position independently between a Runtime and Power-Up Trigger. Select the desired ratio of pre-trigger data to post-trigger data by choosing one of the following ratios:

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

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

If you use the custom-state based triggering flow, you can specify a custom trigger position. The `segment_trigger` and trigger actions accept a post-fill count argument. The post-fill count specifies the number of samples to capture before stopping data acquisition for the non-segmented buffer or a data segment when using the `trigger` and `segment_trigger` commands, respectively. When the captured data is displayed in the SignalTap II data window, the trigger position appears as the number of post-count samples from the end of the acquisition segment or buffer.

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

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

For segmented buffers, the acquisition segments that have a post-count argument define use of the post-count setting. Segments that do not have a post-count setting default to the trigger position ratios defined in the **Setup** tab.

Related Information

State-Based Triggering on page 13-32

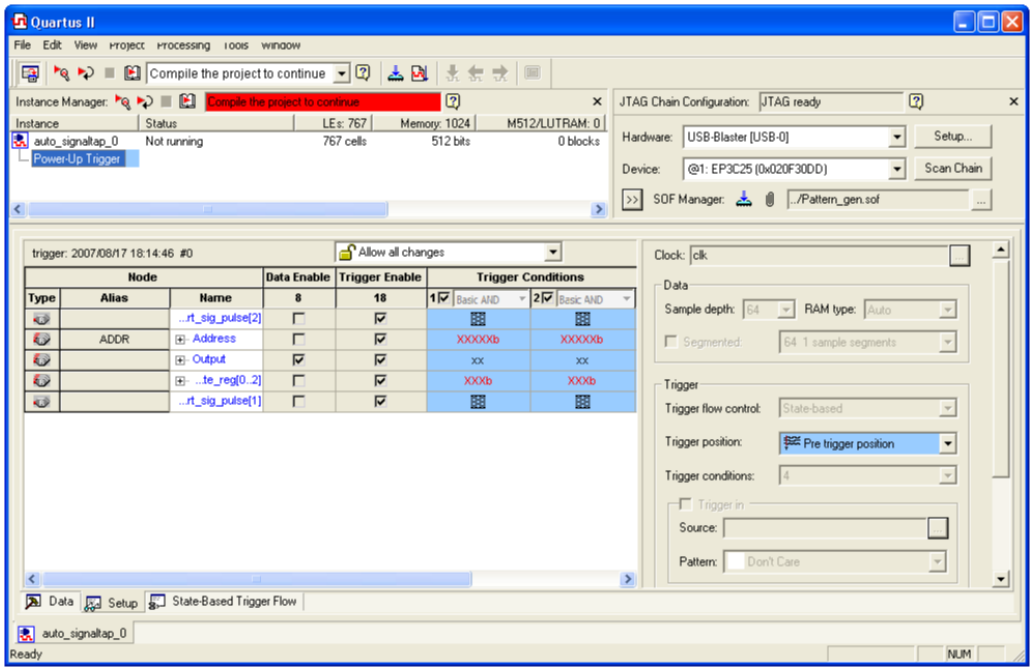
Creating a Power-Up Trigger

Typically, the SignalTap II Logic Analyzer is used to trigger on events that occur during normal device operation. You start an analysis manually once the target device is fully powered on and the JTAG connection for the device is available. However, there may be cases when you would like to capture trigger events that occur during device initialization, immediately after the FPGA is powered on or reset. With the SignalTap II Power-Up Trigger feature, you arm the SignalTap II Logic Analyzer and capture data immediately after device programming.

Enabling a Power-Up Trigger

You can add a different Power-Up Trigger to each logic analyzer instance in the **SignalTap II Instance Manager** pane. To enable the Power-Up Trigger for a logic analyzer instance, right-click the instance and click **Enable Power-Up Trigger**, or select the instance, and on the **Edit** menu, click **Enable Power-Up Trigger**. To disable a Power-Up Trigger, click **Disable Power-Up Trigger** in the same locations. Power-Up Trigger is shown as a child instance below the name of the selected instance with the default trigger conditions specified in the node list.

Figure 13-40: SignalTap II Logic Analyzer Editor with Power-Up Trigger Enabled



Managing and Configuring Power-Up and Runtime Trigger Conditions

When the Power-Up Trigger is enabled for a logic analyzer instance, you can create basic and advanced trigger conditions for the trigger as you do with a Run-Time Trigger. Power-Up Trigger conditions that you can adjust are color coded light blue, while Run-Time Trigger conditions you cannot adjust remain white. Since each instance now has two sets of trigger conditions—the Power-Up Trigger and the Run-Time Trigger—you can differentiate between the two with color coding. To switch between the trigger conditions of the Power-Up Trigger and the Run-Time Trigger, double-click the instance name or the Power-Up Trigger name in the **Instance Manager**.

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

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

While creating or making changes to the trigger conditions for the Run-Time Trigger or the Power-Up Trigger, you may want to copy these conditions to the other trigger. This enables you to look for the same trigger during both power-up and runtime. To do this, right-click the instance name or the Power-Up Trigger name in the **Instance Manager** and click **Duplicate Trigger**, or select the instance name or the Power-Up Trigger name and on the **Edit** menu, click **Duplicate Trigger**.

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

Related Information

Design Debugging Using In-System Sources and Probes documentation

Using External Triggers

You can create a trigger input that allows you to trigger the SignalTap II Logic Analyzer from an external source. The external trigger input behaves like trigger condition 1, is evaluated, and must be `TRUE` before any other configured trigger conditions are evaluated. The logic analyzer supplies a signal to trigger external devices or other SignalTap II Logic Analyzer instances. These features allow you to synchronize external logic analysis equipment with the internal logic analyzer. Power-Up Triggers can use the external triggers feature, but they must use the same source or target signal as their associated Run-Time Trigger.

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

Related Information

- **FPGA-Adaptive Software Debug and Performance Analysis white paper**
  Information about the ARM DS-5 debugging solution
- **Signal Configuration Pane online help**
  Information about setting up external triggers

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

An advanced feature of the SignalTap II 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`.

Compile the Design

When you add an **.stp** to your project, the SignalTap II Logic Analyzer becomes part of your design. You must compile your project to incorporate the SignalTap II logic and enable the JTAG connection you use to control the logic analyzer. When you are debugging with a traditional external logic analyzer, you must often make changes to the signals monitored as well as the trigger conditions. Because these adjustments require that you recompile your design when using the SignalTap II Logic Analyzer, use the SignalTap II Logic Analyzer feature along with incremental compilation in the Quartus II software to reduce recompilation time.

Related Information

Using the Incremental Compilation Design Flow online help

Faster Compilations with Quartus II Incremental Compilation

When you compile your design with an **.stp**, the `sld_signaltap` and `sld_hub` entities are automatically added to the compilation hierarchy. These two entities are the main components of the SignalTap II Logic Analyzer, providing the trigger logic and JTAG interface required for operation.

Incremental compilation enables you to preserve the synthesis and fitting results of your original design and add the SignalTap II Logic Analyzer to your design without recompiling your original source code. Incremental compilation is also useful when you want to modify the configuration of the **.stp**. For example, you can modify the buffer sample depth or memory type without performing a full compilation after the change is made. Only the SignalTap II Logic Analyzer, configured as its own design partition, must be recompiled to reflect the changes.

To use incremental compilation, first enable **Full Incremental Compilation** for your design if it is not already enabled, assign design partitions if necessary, and set the design partitions to the correct preservation levels. Incremental compilation is the default setting for new projects in the Quartus II software, so you can establish design partitions immediately in a new project. However, it is not necessary to create any design partitions to use the SignalTap II incremental compilation feature. When your design is set up to use full incremental compilation, the SignalTap II Logic Analyzer acts as its own separate design partition. You can begin taking advantage of incremental compilation by using the **SignalTap II: post-fitting filter** in the Node Finder to add signals for logic analysis.

Enabling Incremental Compilation for Your Design

When enabled for your design, the SignalTap II Logic Analyzer is always a separate partition. After the first compilation, you can use the SignalTap II Logic Analyzer to analyze signals from the post-fit netlist. If your partitions are designed correctly, subsequent compilations due to SignalTap II 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 wish to tap as **Post-fit**.

Related Information

Quartus II Incremental Compilation for Hierarchical and Team-Based Design documentation

Using Incremental Compilation with the SignalTap II Logic Analyzer

The SignalTap II Logic Analyzer is automatically configured to work with the incremental compilation flow. For all signals that you want to connect to the SignalTap II Logic Analyzer from the post-fit netlist, set the netlist type of the partition containing the desired signals to **Post-Fit** or **Post-Fit (Strict)** with a Fitter Preservation Level of **Placement and Routing** using the Design Partitions window. Use the **SignalTap II: post-fitting filter** in the **Node Finder** to add the signals of interest to your SignalTap II configuration file. If you want to add signals from the pre-synthesis netlist, set the netlist type to **Source File** and use the **SignalTap II: pre-synthesis filter** in the **Node Finder**. Do not use the netlist type **Post-Synthesis** with the SignalTap II Logic Analyzer.

Caution: Be sure to conform to the following guidelines when using post-fit and pre-synthesis nodes:

- Read all incremental compilation guidelines to ensure the proper partition of a project.
- To speed compile time, use only post-fit nodes for partitions specified as to 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 may be different 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 SignalTap II 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 **SignalTap II data** tab.

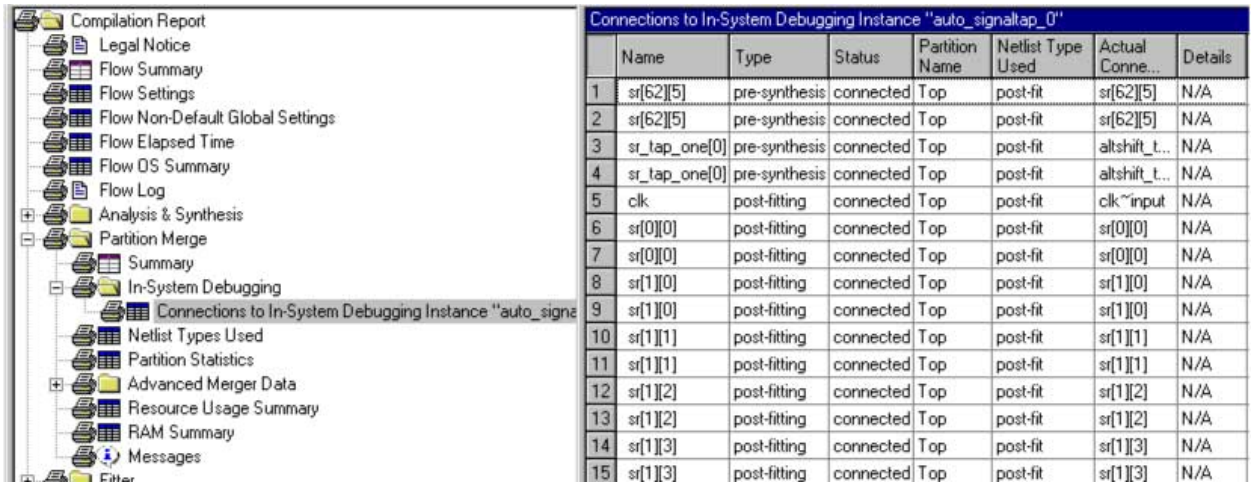
If you do use incremental compilation flow with the SignalTap II Logic Analyzer and source file changes are necessary, be aware that you may have to remove compiler-generated post-fit net names. Source code changes force the affected partition to go through resynthesis. During synthesis, the compiler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

Note: Altera 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 SignalTap II Logic Analyzer configuration.

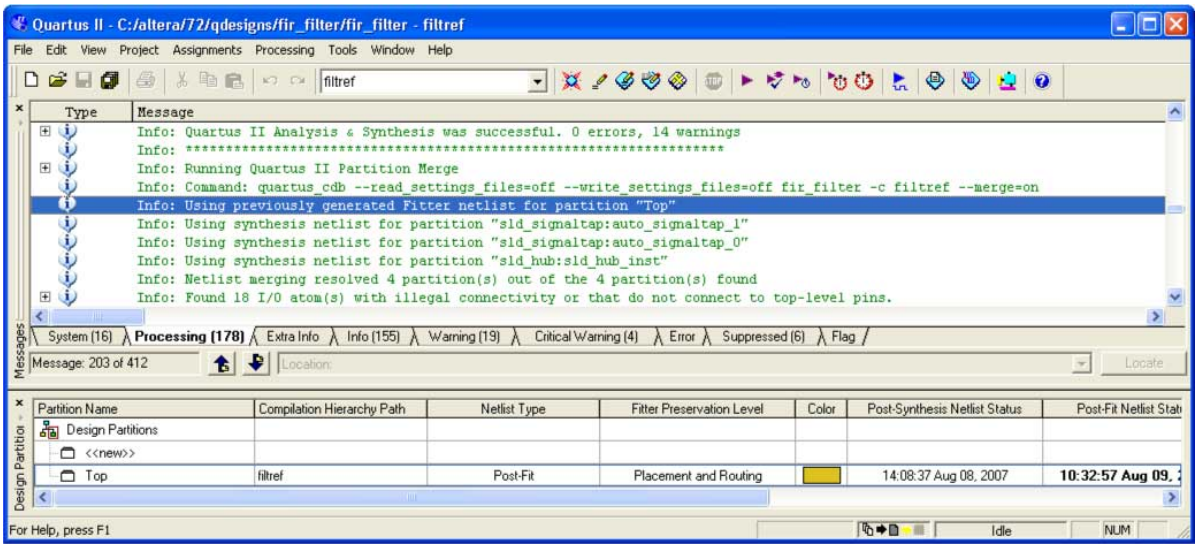
You can check the nodes that are connected to each SignalTap II instance using the In-System Debugging compilation reports. These reports list each node name you selected to connect to a SignalTap II instance, the netlist type used for the particular connection, and the actual node name used after compilation. If incremental compile is turned off, the In-System Debugging reports are located in the Analysis & Synthesis folder. If incremental compilation is turned on, this report is located in the Partition Merge folder.

Figure 13-41: Compilation Report Showing Connectivity to SignalTap II Instance



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

Figure 13-42: Compilation Report Messages



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

Preventing Changes Requiring Recompilation

You can configure the **.stp** to prevent changes that normally require recompilation. To do this, select a lock mode from above the node list in the **Setup** tab. To lock your configuration, choose to allow only trigger condition changes, regardless of whether you use incremental compilation.

Related Information

Setup Tab (SignalTap II Logic Analyzer) online help

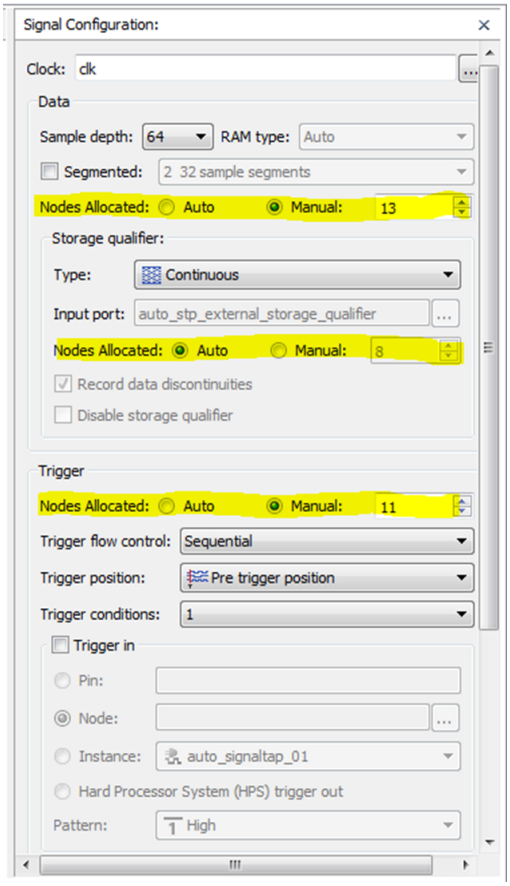
Incremental Route with Rapid Recompile

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.

Device support in Quartus II software v14.0 for Incremental Route with Rapid Recompile is limited to Arria V, Cyclone V, and Stratix V.

Incremental Route Flow

Figure 13-43: Manually Allocate Nodes

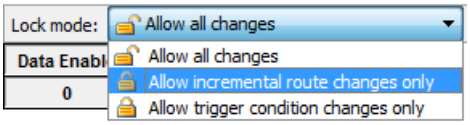


1. Open your design, and run **Analysis & Elaboration** (or a full compilation) to give node visibility in SignalTap II.
2. Add SignalTap II to your design and specify manual allocation for Trigger and Data (Storage Qualifier, if used) nodes in the SignalTap II **Signal Configuration** pane.

   Note: Selecting **Manual** allows you to control the number of nodes compiled into the design. This is critical for the Incremental Route flow. If you select **Auto**, the number of nodes compiled into the design will directly reflect the number of nodes (not including groups, which are not signals) currently in the **Setup** tab. If you then add a node, the number of nodes required on the device will not match what has already been compiled, and you will then need to perform a full compilation.

3. Specify the number of nodes that you estimate will be needed for the debugging process. You can increase the number of nodes later, but this will require more compilation time.
4. Connect nodes you are interested in tapping.
5. Run a full compilation, if you have not already done a full compile on your project. Otherwise, you can start incremental compile using Rapid Recompile.
6. Debug and determine additional signals of interest.
7. (Optional) Turn on **Allow incremental route changes only** lock-mode.

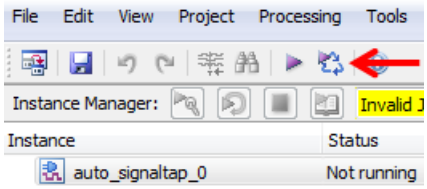
Figure 13-44: Incremental Route Lock-Mode



8. Add additional nodes in the SignalTap II **Setup** tab without exceeding the number of manually allocated nodes in step 2. Avoid making changes to non-runtime configurable settings.
9. Click the Rapid Recompile icon from the toolbar (or from the Processing menu, click **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.

Figure 13-45: Rapid Recompile Icon



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 the number/names of tapped nodes are changed.
- Use the Incremental Route lock-mode to avoid inadvertent changes requiring a full compilation.

Timing Preservation with the SignalTap II Logic Analyzer

In addition to verifying functionality, timing closure is one of the most crucial processes in successfully completing a design. When you compile a project with a SignalTap II Logic Analyzer without the use of incremental compilation, you add IP to your existing design. Therefore, you can affect the existing placement, routing, and timing of your design. To minimize the effect that the SignalTap II Logic Analyzer has on your design, Altera recommends that you use incremental compilation for your project. Incremental compilation is the default setting in new designs and can be easily enabled and configured in existing designs. With the SignalTap II Logic Analyzer instance in its own design partition, it has little to no affect on your design.

In addition to using the incremental compilation flow for your design, you can use the following techniques to help maintain timing:

- Avoid adding critical path signals to your **.stp**.

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

Related Information

Timing Closure and Optimization documentation

Performance and Resource Considerations

There is a necessary trade-off between the runtime flexibility of the SignalTap II Logic Analyzer, the timing performance of the SignalTap II Logic Analyzer, and resource usage. The SignalTap II Logic Analyzer allows you to select the runtime configurable parameters to balance the need for runtime flexibility, speed, and area. The default values have been chosen to provide maximum flexibility so you can complete debugging as quickly as possible; however, you can adjust these settings to determine whether there is a more optimal configuration for your design.

The following tips provide extra timing slack if you have determined that the SignalTap II logic is in your critical path, or to alleviate the resource requirements that the SignalTap II Logic Analyzer consumes if your design is resource-constrained.

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

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

If your design is resource constrained, follow these tips to reduce the amount of logic or memory used by the SignalTap II Logic Analyzer:

- **Disable runtime configurable options**—Disabling runtime configurability for advanced trigger conditions or runtime configurable options in the State-based triggering flow results in using fewer LEs.
- **Minimize the number of segments in the acquisition buffer**—You can reduce the number of logic resources used for the SignalTap II Logic Analyzer by limiting the number of segments in your sampling buffer to only those required.
- **Disable the Data Enable for signals that are used for triggering only**—By default, both the **data enable** and **trigger enable** options are selected for all signals. Turning off the **data enable** option for signals used as trigger inputs only saves on memory resources used by the SignalTap II Logic Analyzer.

Because performance results are design-dependent, try these options in different combinations until you achieve the desired balance between functionality, performance, and utilization.

Program the Target Device or Devices

After you compile your project, including the SignalTap II Logic Analyzer, configure the FPGA target device. When you are using the SignalTap II Logic Analyzer for debugging, configure the device from the **.stp** instead of the Quartus II Programmer. Because you configure from the **.stp**, you can open more than one **.stp** and program multiple devices to debug multiple designs simultaneously.

The settings in an **.stp** must be compatible with the programming **.sof** used to program the device. An **.stp** is considered compatible with an **.sof** when the settings for the logic analyzer, such as the size of the capture buffer and the signals selected for monitoring or triggering, match the way the target device is programmed. If the files are not compatible, you can still program the device, but you cannot run or control the logic analyzer from the SignalTap II Logic Analyzer Editor.

Note: When the SignalTap II Logic Analyzer detects incompatibility after analysis is started, a system error message is generated containing two CRC values, the expected value and the value retrieved from the **.stp** instance on the device. The CRC values are calculated based on all SignalTap II settings that affect the compilation.

To ensure programming compatibility, make sure to program your device with the latest **.sof** created from the most recent compilation. Checking whether or not a particular **.sof** is compatible with the current SignalTap II configuration is achieved quickly by attaching the **.sof** to the SOF manager.

Before starting a debugging session, do not make any changes to the **.stp** settings that would requires recompiling the project. You can check the SignalTap II status display at the top of the **Instance Manager** pane to verify whether a change you made requires recompiling the project, producing a new **.sof**. This gives you the opportunity to undo the change, so that you do not need to recompile your project. To prevent any such changes, select **Allow trigger condition changes only** to lock the **.stp**. The Incremental Route lock mode, **Allow incremental route changes only**, limits changes which will only require an Incremental Route using Rapid Recompile, and not a full compile.

Although the Quartus II project is not required when using an **.stp**, it is recommended. The project database contains information about the integrity of the current SignalTap II Logic Analyzer session. Without the project database, there is no way to verify that the current **.stp** matches the **.sof** that is downloaded to the device. If you have an **.stp** that does not match the **.sof**, incorrect data is captured in the SignalTap II Logic Analyzer.

Related Information

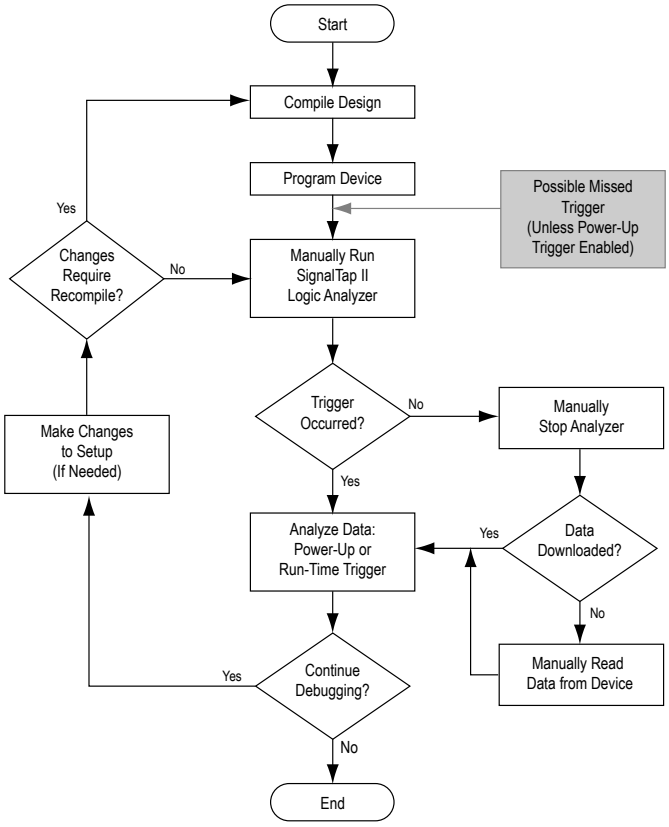
Running the SignalTap II Logic Analyzer online help

Run the SignalTap II Logic Analyzer

After the device is configured with your design that includes the SignalTap II Logic Analyzer, perform debugging operations in a manner similar to when you use an external logic analyzer. You initialize the logic analyzer by starting an analysis. When your trigger event occurs, the captured data is stored in the memory buffer on the device and then transferred to the **.stp** with the JTAG connection.

You can also perform the equivalent of a force trigger instruction that lets you view the captured data currently in the buffer without a trigger event occurring. **Figure 13-46** illustrates a flow that shows how you operate the SignalTap II Logic Analyzer. The flowchart indicates where Power-Up and Runtime Trigger events occur and when captured data from these events is available for analysis.

Figure 13-46: Power-Up and Runtime Trigger Events Flowchart



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

Related Information

- **Running the SignalTap II Logic Analyzer online help**
  Information on running the analyzer from the **Instance Manager** pane
- **Design Debugging Using In-System Sources and Probes documentation**

Runtime Reconfigurable Options

Certain settings in the **.stp** are changeable without recompiling your design when you use Runtime Trigger mode.

Table 13-10: Runtime Reconfigurable Features

Runtime Reconfigurable Setting	Description
Basic Trigger Conditions and Basic Storage Qualifier Conditions	All signals that have the Trigger condition turned on can be changed to any basic trigger condition value without recompiling.
Advanced Trigger Conditions and Advanced Storage Qualifier Conditions	Many operators include runtime configurable settings. For example, all comparison operators are runtime-configurable. Configurable settings are shown with a white background in the block representation.This runtime reconfigurable option is turned on in the **Object Properties** dialog box.
Switching between a storage-qualified and a continuous acquisition	Within any storage-qualified mode, you can switch to continuous capture mode without recompiling the design. To enable this feature, turn on **disable storage qualifier**.
State-based trigger flow parameters	Table 13-5 lists Reconfigurable State-based trigger flow options.

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

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

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

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

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

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

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

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

Note to example :

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

Figure 13-47 shows a segmented buffer described by the trigger flow in example above.

During runtime, the values m and n are runtime reconfigurable. By changing the m and n values in the preceding trigger flow description, you can dynamically adjust the segment boundaries without incurring a recompile.

Figure 13-47: Segmented Buffer Created with Storage Qualifier and State-Based Trigger

Note to figure :

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

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

The example below shows a modified description of the example above with an additional state inserted. You use this extra state to specify a different trigger condition that does not use the storage qualifier feature. You insert status flags into the conditional statements to control the execution of the trigger flow.

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

                                        //to enable.
begin
   start_store;
   increment c1;
   goto ST3:
end
else (c1 > m )
   start_store
   Trigger (n-1)
end
state ST3:
if ( c2 >= n)
begin
 reset c2;
```

```
 stop_store;
 goto ST1;
end
else (c2 < n)
begin
 increment c2;
 goto ST2;
end
```

SignalTap II Status Messages

Table 13-11 describes the text messages that might appear in the SignalTap II Status Indicator in the **Instance Manager** pane before, during, and after a data acquisition. Use these messages to monitor the state of the logic analyzer or what operation it is performing.

Table 13-11: Text Messages in the SignalTap II Status Indicator

Message	Message Description
`Not running`	The SignalTap II Logic Analyzer is not running. There is no connection to a device or the device is not configured.
`(Power-Up Trigger) Waiting for clock` (1)	The SignalTap II Logic Analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.
`Acquiring (Power-Up) pre-trigger data` (1)	The trigger condition has not been evaluated yet. A full buffer of data is collected if using the non-segmented buffer acquisition mode and storage qualifier type is continuous.
`Trigger In conditions met`	Trigger In condition has occurred. The SignalTap II Logic Analyzer is waiting for the condition of the first trigger condition to occur. This can appear if Trigger In is specified.
`Waiting for (Power-up) trigger` (1)	The SignalTap II Logic Analyzer is now waiting for the trigger event to occur.
`Trigger level <x>` met	The condition of trigger condition *x* has occurred. The SignalTap II Logic Analyzer is waiting for the condition specified in condition x + 1 to occur.
`Acquiring (power-up) post-trigger data` (1)	The entire trigger event has occurred. The SignalTap II Logic Analyzer is acquiring the post-trigger data. The amount of post-trigger data collected is you define between 12%, 50%, and 88% when the non-segmented buffer acquisition mode is selected.
`Offload acquired (Power-Up) data` (1)	Data is being transmitted to the Quartus II software through the JTAG chain.
`Ready to acquire`	The SignalTap II Logic Analyzer is waiting for you to initialize the analyzer.

Note to Table 13-11 :

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

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

View, Analyze, and Use Captured Data

Once a trigger event has occurred or you capture data manually, you can use the SignalTap II interface to examine the data, and use your findings to help debug your design.

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

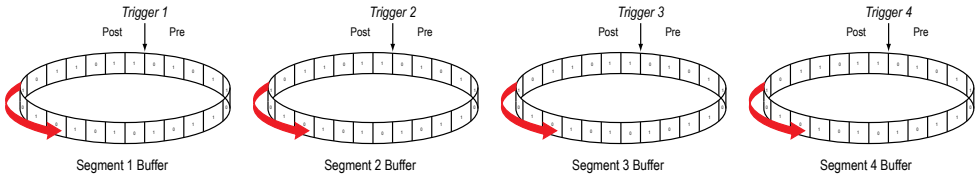
Related Information

Analyzing Data in the SignalTap II Logic Analyzer online help
Information about what you can do with captured data

Capturing Data Using Segmented Buffers

Segmented Acquisition buffers allow you to perform multiple captures with a separate trigger condition for each acquisition segment. This feature allows you to capture a recurring event or sequence of events that span over a long period time efficiently. Each acquisition segment acts as a non-segmented buffer, continuously capturing data when it is activated. When you run an analysis with the **segmented buffer** option enabled, the SignalTap II Logic Analyzer performs back-to-back data captures for each acquisition segment within your data buffer. The trigger flow, or the type and order in which the trigger conditions evaluate for each buffer, is defined by either the Sequential trigger flow control or the Custom State-based trigger flow control. **Figure 13-48** shows a segmented acquisition buffer with four segments represented as four separate non-segmented buffers.

Figure 13-48: Segmented Acquisition Buffer



The SignalTap II Logic Analyzer finishes an acquisition with a segment, and advances to the next segment to start a new acquisition. Depending on when a trigger condition occurs, it may affect the way the data capture appears in the waveform viewer. **Figure 13-48** illustrates the method in which data is captured. The Trigger markers in **Figure 13-48**—Trigger 1, Trigger 2, Trigger 3 and Trigger 4—refer to the evaluation of the `segment_trigger` and `trigger` commands in the Custom State-based trigger flow. If you use a sequential flow, the Trigger markers refer to trigger conditions specified within the **Setup** tab.

If the Segment 1 Buffer is the active segment and Trigger 1 occurs, the SignalTap II Logic Analyzer starts evaluating Trigger 2 immediately. Data Acquisition for Segment 2 buffer starts when either Segment Buffer 1 finishes its post-fill count, or when Trigger 2 evaluates as `TRUE`, whichever condition occurs first. Thus, trigger conditions associated with the next buffer in the data capture sequence can preempt the post-fill count of the current active buffer. This allows the SignalTap II Logic Analyzer to accurately capture all of the trigger conditions that have occurred. Samples that have not been used appear as a blank space in the waveform viewer.

Figure 13-49 shows an example of a capture using sequential flow control with the trigger condition for each segment specified as **Don't Care**. Each segment before the last captures only one sample, because the next trigger condition immediately preempts capture of the current buffer. The trigger position for all

segments is specified as pre-trigger (10% of the data is before the trigger condition and 90% of the data is after the trigger position). Because the last segment starts immediately with the trigger condition, the segment contains only post-trigger data. The three empty samples in the last segment are left over from the pre-trigger samples that the SignalTap II Logic Analyzer allocated to the buffer.

Figure 13-49: Segmented Capture with Preemption of Acquisition Segments

Note to **Figure 13-49** :

1. A segmented acquisition buffer using the sequential trigger flow with a trigger condition specified as Don't Care. All segments, with the exception of the last segment, capture only one sample because the next trigger condition preempts the current buffer from filling to completion.

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

Differences in Pre-fill Write Behavior Between Different Acquisition Modes

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

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

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

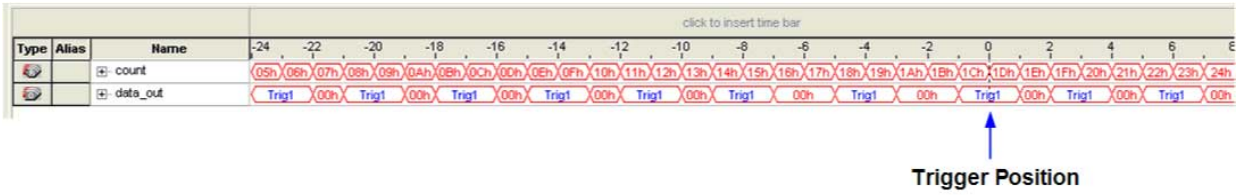
Filling the buffer provides you with as much data as possible within the capture window. The buffer gets pre-filled with data samples prior to evaluating the trigger condition. As such, SignalTap requires that the buffer be filled at least once before any data can be retrieved through the JTAG connection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

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

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

Figure 13-50 and **Figure 13-51** show the difference between a non-segmented buffer in continuous mode and a non-segmented buffer using a storage qualifier. The logic analyzer for the waveforms below is configured with a sample depth of 64 bits, with a trigger position specified as **Post trigger position**.

Figure 13-50: SignalTap II Logic Analyzer Continuous Data Capture

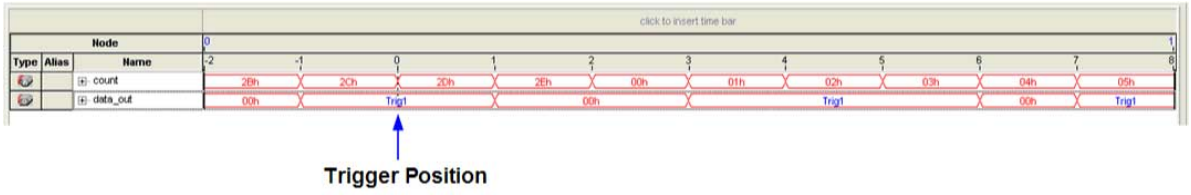


Note to **Figure 13-50** :

1. Continuous capture mode with post-trigger position.
2. Capture of a recurring pattern using a non-segmented buffer in continuous mode. The SignalTap II Logic Analyzer is configured with a basic trigger condition (shown in the figure as "Trig1") with a sample depth of 64 bits.

Notice in **Figure 13-50** that Trig1 occurs several times in the data buffer before the SignalTap II Logic Analyzer actually triggers. A full buffer's worth of data is captured before the logic analyzer evaluates any trigger conditions. After the trigger condition occurs, the logic analyzer continues acquisition until it captures eight additional samples (12% of the buffer, as defined by the "post-trigger" position).

Figure 13-51: SignalTap II Logic Analyzer Conditional Data Capture



Note to **Figure 13-51** :

1. Conditional capture, storage always enabled, post-fill count.
2. SignalTap II Logic Analyzer capture of a recurring pattern using a non-segmented buffer in conditional mode. The logic analyzer is configured with a basic trigger condition (shown in the figure as "Trig1"), with a sample depth of 64 bits. The “Trigger in” condition is specified as "Don't care", which means that every sample is captured.

Notice in Figure 13-51 that the logic analyzer triggers immediately. As in Figure 13-50, the logic analyzer completes the acquisition with eight samples, or 12% of 64, the sample capacity of the acquisition buffer.

Creating Mnemonics for Bit Patterns

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

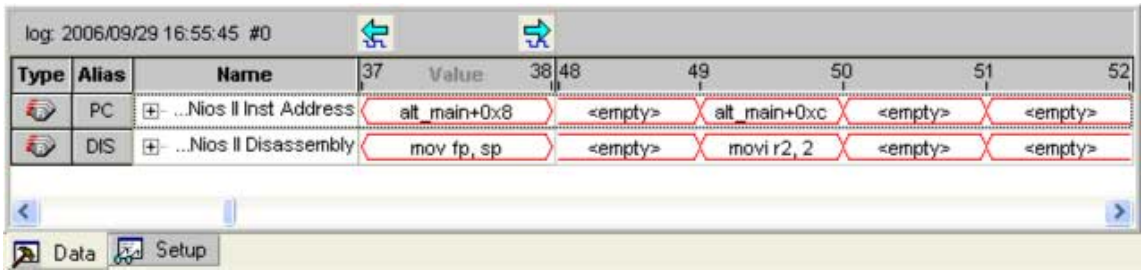
You use the labels you create in a table in different ways on the **Setup** and **Data** tabs. On the **Setup** tab, you can create basic triggers with meaningful names by right-clicking an entry in the **Trigger Conditions** column and selecting a label from the table you assigned to the signal group.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

Automatic Mnemonics with a Plug-In

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

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

Figure 13-52: Data Tab when the Nios II Plug-In is Used



Locating a Node in the Design

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

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

- Assignment Editor

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

Saving Captured Data

The data log shows the history of captured data and the triggers used to capture the data. The SignalTap II Logic Analyzer acquires data, stores it in a log, and displays it as waveforms. When the logic analyzer is in auto-run mode and a trigger event occurs more than once, captured data for each time the trigger occurred is stored as a separate entry in the data log. This allows you to review the captured data for each trigger event. The default name for a log is based on the time when the data was acquired. Altera recommends that you rename the data log with a more meaningful name.

The logs are organized in a hierarchical manner; similar logs of captured data are grouped together in trigger sets. To open the **Data Log** pane, on the View menu, select **Data Log**. To turn on data logging, turn on **Enable data log** in the **Data Log** (Figure 13-19). To recall and activate a data log for a given trigger set, double-click the name of the data log in the list. The time stamping for the Data Log entries display the wall-clock time when SignalTap II triggered and the elapsed time from when acquisition started to when the device triggered.

Related Information

Managing Multiple SignalTap II Files and Configurations on page 13-23
You can use the Data Log feature for organizing different sets of trigger conditions and different sets of signal configurations.

Exporting Captured Data to Other File Formats

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

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

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

Creating a SignalTap II List File

Captured data can also be viewed in an **.stp** list file. An **.stp** list file is a text file that lists all the data captured by the logic analyzer for a trigger event. Each row of the list file corresponds to one captured sample in the buffer. Columns correspond to the value of each of the captured signals or signal groups for that sample. If a mnemonic table was created for the captured data, the numerical values in the list are replaced with a matching entry from the table. This is especially useful with the use of a plug-in that includes instruction code disassembly. You can immediately see the order in which the instruction code was executed during the

same time period of the trigger event. To create an **.stp** list file in the Quartus II software, on the File menu, select **Create/Update** and click **Create SignalTap II List File**.

Other Features

The SignalTap II Logic Analyzer has other features that do not necessarily belong to a particular task in the task flow.

Using the SignalTap II MATLAB MEX Function to Capture Data

If you use MATLAB for DSP design, you can call the MATLAB MEX function `alt_signaltap_run`, built into the Quartus II software, to acquire data from the SignalTap II Logic Analyzer directly into a matrix in the MATLAB environment. If you use the MATLAB MEX function in a loop, you can perform as many acquisitions in the same amount of time as you can when using SignalTap II in the Quartus II software environment.

Note: The SignalTap II MATLAB MEX function is available in the Windows version and Linux version of the Quartus II software. It is compatible with MATLAB Release 14 Original Release Version 7 and later.

To set up the Quartus II software and the MATLAB environment to perform SignalTap II acquisitions, perform the following steps:

1. In the Quartus II software, create an **.stp** file.
2. In the node list in the **Data** tab of the SignalTap II Logic Analyzer Editor, organize the signals and groups of signals into the order in which you want them to appear in the MATLAB matrix. Each column of the imported matrix represents a single SignalTap II acquisition sample, while each row represents a signal or group of signals in the order they are organized in the **Data** tab.

   Note: Signal groups acquired from the SignalTap II Logic Analyzer and transferred into the MATLAB MEX function are limited to a width of 32 signals. If you want to use the MATLAB MEX function with a bus or signal group that contains more than 32 signals, split the group into smaller groups that do not exceed the 32-signal limit.

3. Save the **.stp** and compile your design. Program your device and run the SignalTap II Logic Analyzer to ensure your trigger conditions and signal acquisition work correctly.
4. In the MATLAB environment, add the Quartus II binary directory to your path with the following command:

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

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

   ```
   alt_signaltap_run
   ```

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

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

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

When capturing data you must assign a filename, for example, <*stp filename*> as a requirement of the MATLAB MEX function. Other MATLAB MEX function options are described in **Table 13-12**.

Table 13-12: SignalTap II MATLAB MEX Function Options

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

You can enable or disable verbose mode to see the status of the logic analyzer while it is acquiring data. To enable or disable verbose mode, use the following commands:

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

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

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

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

Using SignalTap II in a Lab Environment

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

Remote Debugging Using the SignalTap II Logic Analyzer

Debugging Using a Local PC and an Altera SoC

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

Related Information

Remote Hardware Debugging over TCP/IP for Altera SoC application note

Debugging Using a Local PC and a Remote PC

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

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

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

Equipment Setup

On the PC in the remote location, install the standalone version of the SignalTap II Logic Analyzer, included in the Quartus II standalone Programmer, or the full version of the Quartus II software. This remote computer must have Altera programming hardware connected, such as the EthernetBlaster or USB-Blaster.

On the local PC, install the full version of the Quartus II software. This local PC must be connected to the remote PC across a LAN with the TCP/IP protocol.

Related Information

Using the JTAG Server online help
Information about enabling remote access to a JTAG server

Using the SignalTap II Logic Analyzer in Devices with Configuration Bitstream Security

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

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

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

Backward Compatibility with Previous Versions of Quartus II Software

You can open an **.stp** created in a previous version in a current version of the Quartus II software. However, opening an **.stp** modifies it so that it cannot be opened in a previous version of the Quartus II software.

If you have a Quartus II project file from a previous version of the software, you may have to update the **.stp** configuration file to recompile the project. You can update the configuration file by opening the SignalTap II Logic Analyzer. If you need to update your configuration, a prompt appears asking if you would like to update the **.stp** to match the current version of the Quartus II software.

SignalTap II Command-Line Options

To compile your design with the SignalTap II Logic Analyzer using the command prompt, use the `quartus_stp` command. Table 13-13 shows the options that help you use the `quartus_stp` executable.

Table 13-13: SignalTap II Command-Line Options

Option	Usage	Description
`stp_file`	`quartus_stp --stp_file <stp_filename>`	Assigns the specified **.stp** to the `USE_SIGNALTAP_FILE` in the **.qsf**.
`enable`	`quartus_stp --enable`	Creates assignments to the specified **.stp** in the **.qsf** and changes `ENABLE_SIGNALTAP` to `ON`. The SignalTap II Logic Analyzer is included in your design the next time the project is compiled. If no **.stp** is specified in the **.qsf**, the `--stp_file` option must be used. If the `--enable` option is omitted, the current value of `ENABLE_SIGNALTAP` in the **.qsf** is used.
`disable`	`quartus_stp --disable`	Removes the **.stp** reference from the **.qsf** and changes `ENABLE_SIGNALTAP` to `OFF`. The SignalTap II Logic Analyzer is removed from the design database the next time you compile your design. If the `--disable` option is omitted, the current value of `ENABLE_SIGNALTAP` in the **.qsf** is used.
`create_signaltap_hdl_file`	`quartus_stp --create_signaltap_hdl_file`	Creates an **.stp** representing the SignalTap II instance. The file is based on the last compilation. You must use the `--stp_file` option to create an **.stp** properly. Analogous to the **Create SignalTap II File from Design Instance(s)** command in the Quartus II software.

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

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

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

The example below shows how to create a new **.stp** after building the SignalTap II Logic Analyzer instance with the IP Catalog.

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

Related Information

Command-Line Scripting documentation
Information about the other command line executables and options

SignalTap II Tcl Commands

The **quartus_stp** executable supports a Tcl interface that allows you to capture data without running the Quartus II GUI. You cannot execute SignalTap II Tcl commands from within the Tcl console in the Quartus II software. They must be executed from the command line with the **quartus_stp** executable. To execute a Tcl file that has SignalTap II Logic Analyzer Tcl commands, use the following command:

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

The example is an excerpt from a script you can use to continuously capture data. Once the trigger condition is met, the data is captured and stored in the data log.

```
#opens signaltap session
open_session -name stp1.stp
#start acquisition of instance auto_signaltap_0 and
#auto_signaltap_1 at the same time
#calling run_multiple_end will start all instances
#run after run_multiple_start call
run_multiple_start
run -instance auto_signaltap_0 -signal_set signal_set_1 -trigger \
trigger_1 -data_log log_1 -timeout 5
run -instance auto_signaltap_1 -signal_set signal_set_1 -trigger \
trigger_1 -data_log log_1 -timeout 5
run_multiple_end
#close signaltap session
close_session
```

When the script is completed, open the **.stp** that you used to capture data to examine the contents of the Data Log.

Related Information

::quartus::stp online help
Information about Tcl commands that you can use with the SignalTap II Logic Analyzer Tcl package

Design Example: Using SignalTap II Logic Analyzers

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

Related Information

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

Custom Triggering Flow Application Examples

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

Related Information

On-chip Debugging Design Examples website

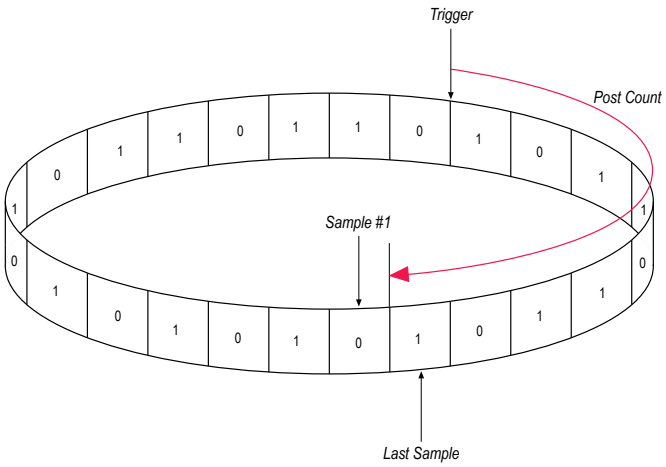
Design Example 1: Specifying a Custom Trigger Position

Actions to the acquisition buffer can accept an optional post-count argument. This post-count argument enables you to define a custom triggering position for each segment in the acquisition buffer. The example shows how to apply a trigger position to all segments in the acquisition buffer. The example describes a triggering flow for an acquisition buffer split into four segments. If each acquisition segment is 64 samples in depth, the trigger position for each buffer will be at sample #34. The acquisition stops after all four segments are filled once.

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

Each segment acts as a non-segmented buffer that continuously updates the memory contents with the signal values. The last acquisition before stopping the buffer is displayed on the **Data** tab as the last sample number in the affected segment. The trigger position in the affected segment is then defined by `N - post count fill`, where N is the number of samples per segment. **Figure 13-53** illustrates the triggering position.

Figure 13-53: Specifying a Custom Trigger Position



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

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

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

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

SignalTap II Scripting Support

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

```
quartus_sh --qhelp
```

Related Information

- **Tcl Scripting documentation**
- **Quartus II Tcl Scripting online help**

Document Revision History

Table 13-14: Document Revision History

Date	Version	Changes Made
June 2014	14.0.0	• DITA conversion. • Replaced MegaWizard Plug-In Manager and Megafunction content with IP Catalog and parameter editor content. • Added flows for custom trigger HDL object, Incremental Route with Rapid Recompile, and nested groups with Basic OR. • GUI changes: toolbar, drag to zoom, disable/enable instance, trigger log time-stamping.
November 2013	13.1.0	Removed HardCopy material. Added section on using cross-triggering with DS-5 tool and added link to white paper 01198. Added section on remote debugging an Altera SoC and added link to application note 693. Updated support for MEX function.
May 2013	13.0.0	• Added recommendation to use the state-based flow for segmented buffers with separate trigger conditions, information about Basic OR trigger condition, and hard processor system (HPS) external triggers. • Updated "Segmented Buffer" on page 13-17, Conditional Mode on page 13-21, Creating Basic Trigger Conditions on page 13-16, and Using External Triggers on page 13-48.
June 2012	12.0.0	Updated Figure 13–5 on page 13–16 and "Adding Signals to the SignalTap II File" on page 13–10.
November 2011	11.0.1	Template update. Minor editorial updates.
May 2011	11.0.0	Updated the requirement for the standalone SignalTap II software.
December 2010	10.0.1	Changed to new document template.
July 2010	10.0.0	• Add new acquisition buffer content to the "View, Analyze, and Use Captured Data" section. • Added script sample for generating hexadecimal CRC values in programmed devices. • Created cross references to Quartus II Help for duplicated procedural content.
November 2009	9.1.0	No change to content.
March 2009	9.0.0	• Updated Table 13–1 • Updated "Using Incremental Compilation with the SignalTap II Logic Analyzer" on page 13–45 • Added new Figure 13–33 • Made minor editorial updates

Date	Version	Changes Made
November 2008	8.1.0	Updated for the Quartus II software version 8.1 release: • Added new section "Using the Storage Qualifier Feature" on page 14–25 • Added description of `start_store` and `stop_store` commands in section "Trigger Condition Flow Control" on page 14–36 • Added new section "Runtime Reconfigurable Options" on page 14–63
May 2008	8.0.0	Updated for the Quartus II software version 8.0: • Added "Debugging Finite State machines" on page 14-24 • Documented various GUI usability enhancements, including improvements to the resource estimator, the bus find feature, and the dynamic display updates to the counter and flag resources in the State-based trigger flow control tab • Added "Capturing Data Using Segmented Buffers" on page 14–16 • Added hyperlinks to referenced documents throughout the chapter • Minor editorial updates

Related Information

Quartus II Handbook Archive

For previous versions of the Quartus II Handbook