

Introduction

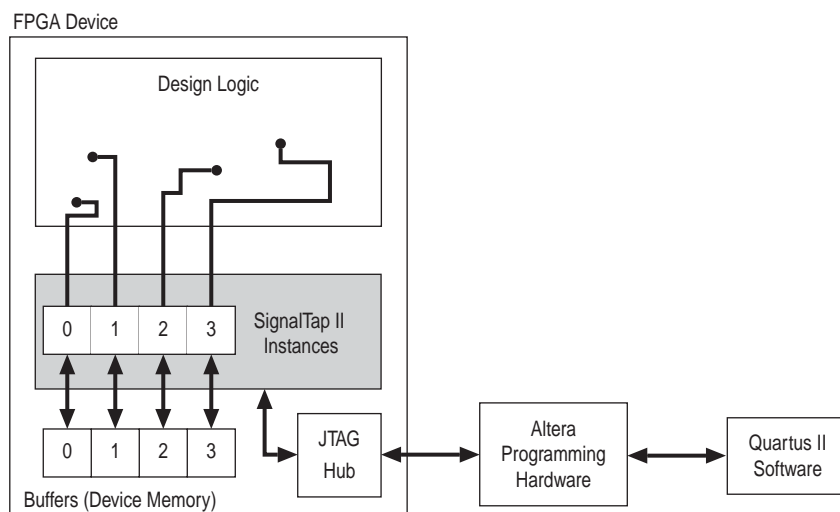
To help with the process of design debugging, Altera provides a solution that allows you to examine the behavior of internal signals, without using extra I/O pins, while the design is running at full speed on an FPGA device.

The SignalTap® II Embedded Logic Analyzer is scalable, easy to use, and is 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 Embedded 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 topics in this chapter include:

- “Design Flow Using the SignalTap II Embedded Logic Analyzer” on page 15–4
- “SignalTap II Embedded Logic Analyzer Task Flow” on page 15–4
- “Add the SignalTap II Embedded Logic Analyzer to Your Design” on page 15–6
- “Configure the SignalTap II Embedded Logic Analyzer” on page 15–14
- “Define Triggers” on page 15–33
- “Compile the Design” on page 15–53
- “Program the Target Device or Devices” on page 15–59
- “Run the SignalTap II Embedded Logic Analyzer” on page 15–60
- “View, Analyze, and Use Captured Data” on page 15–66
- “Other Features” on page 15–71
- “SignalTap II Scripting Support” on page 15–76
- “Design Example: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems” on page 15–79
- “Custom Triggering Flow Application Examples” on page 15–79

The SignalTap II Embedded 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 Embedded Logic Analyzer supports the highest number of channels, largest sample depth, and fastest clock speeds of any embedded logic analyzer in the programmable logic market. [Figure 15–1](#) shows a block diagram of the components that make up the SignalTap II Embedded Logic Analyzer.

Figure 15-1. SignalTap II Embedded Logic Analyzer Block Diagram (Note 1)**Note to Figure 15-1:**

- (1) This diagram assumes that the SignalTap II Embedded Logic Analyzer was compiled 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. For information about the use of incremental compilation with SignalTap II, refer to “[Faster Compilations with Quartus II Incremental Compilation](#)” on page 15-53.

This chapter is intended for any designer who wants to debug their FPGA design during normal device operation without the need for external lab equipment. Because the SignalTap II Embedded 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 Embedded Logic Analyzer, knowledge of the Quartus II incremental compilation feature is helpful.



For information about using the Quartus II incremental compilation feature, refer to the [Quartus II Incremental Compilation for Hierarchical and Team-Based Design](#) chapter in volume 1 of the *Quartus II Handbook*.

Hardware and Software Requirements

The following components are required to perform logic analysis with the SignalTap II Embedded Logic Analyzer:

- Quartus II design software

or

Quartus II Web Edition (with the TalkBack feature enabled)

or

SignalTap II Embedded Logic Analyzer standalone software

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



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

Captured data is stored in the device's memory blocks and transferred to the Quartus II software waveform display with a JTAG communication cable, such as EthernetBlaster or USB-Blaster™. [Table 15-1](#) summarizes some of the features and benefits of the SignalTap II Embedded Logic Analyzer.

Table 15-1. SignalTap II Features and Benefits

| Feature | Benefit |
|--|--|
| 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 embedded processor. |
| Up to 10 basic or advanced trigger conditions for each analyzer instance | Enables more complex data capture commands to be sent 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. |
| State-based Triggering Flow | Enables you to organize your triggering conditions to precisely define what your embedded logic analyzer will capture. |
| Incremental compilation | Modifies the SignalTap II Embedded Logic Analyzer monitored signals and triggers without performing a full compilation, saving time. |
| 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 debug of your design. |
| MATLAB integration with included MEX function | Collects the SignalTap II Embedded Logic Analyzer captured data into a MATLAB integer matrix. |
| Up to 2,048 channels per logic analyzer instance | Samples many signals and wide bus structures. |
| Up to 128K samples in each device | Captures a large sample set for each channel. |
| Fast clock frequencies | Synchronous sampling of data nodes using the same clock tree driving the logic under test. |
| Resource usage estimator | Provides estimate of logic and memory device resources used by SignalTap II Embedded Logic Analyzer configurations. |
| No additional cost | The SignalTap II Embedded 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 | The SignalTap II Embedded Logic Analyzer can be used in tandem with any JTAG based on-chip debugging tool, such as an in-system memory content editor. This ability to share the JTAG chain allows you to change signal values in real-time while you are running an analysis with the SignalTap II Embedded Logic Analyzer. |

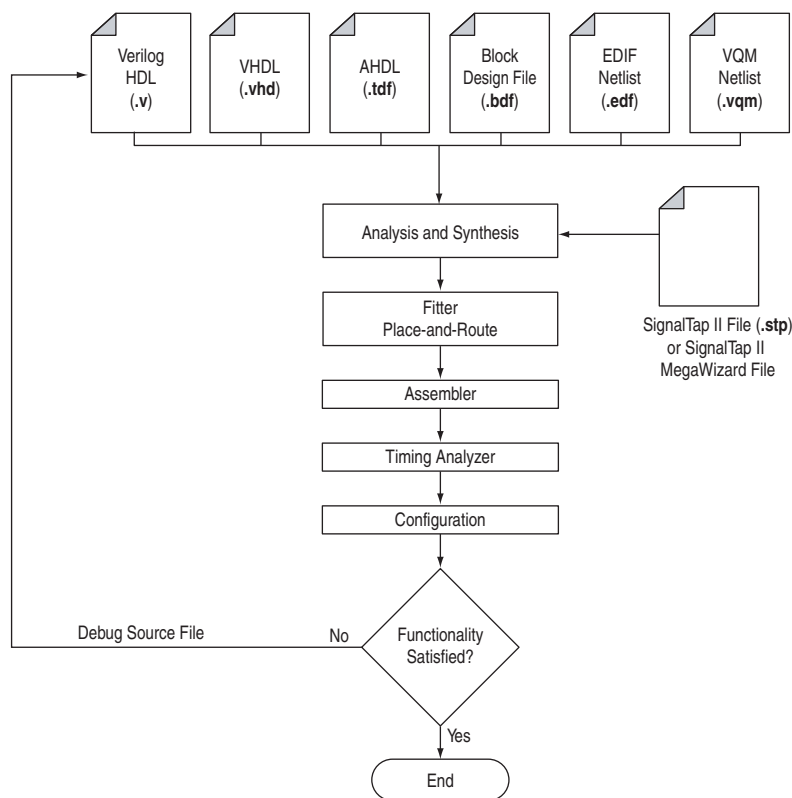


The Quartus II software offers a portfolio of on-chip debugging solutions. For an overview and comparison of all of the tools available in the In-System Verification Tool set, refer to [Section V. In-System Design Debugging](#).

Design Flow Using the SignalTap II Embedded Logic Analyzer

Figure 15-2 shows a typical overall FPGA design flow for using the SignalTap II Embedded Logic Analyzer in your design. A SignalTap II file (.stp) is added to and enabled in your project, or a SignalTap II HDL function, created with the MegaWizard™ Plug-In Manager, is instantiated in your design. The diagram shows the flow of operations from initially adding the SignalTap II Embedded Logic Analyzer to your design to final device configuration, testing, and debugging.

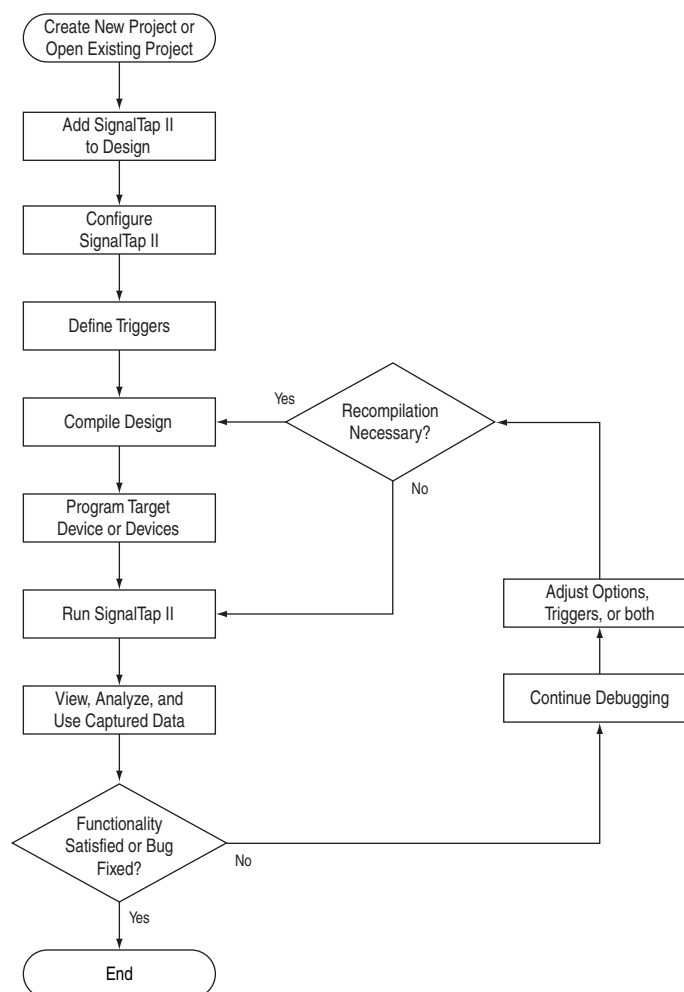
Figure 15-2. SignalTap II FPGA Design and Debugging Flow



SignalTap II Embedded Logic Analyzer Task Flow

To use the SignalTap II Embedded Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer. Figure 15-3 shows a typical flow of the tasks you complete to debug your design. Refer to the appropriate section of this chapter for more information about each of these tasks.

Figure 15-3. SignalTap II Embedded Logic Analyzer Task Flow



Add the SignalTap II Embedded Logic Analyzer to Your Design

Create an `.stp` file or create a parameterized HDL instance representation of the logic analyzer using the MegaWizard Plug-In Manager. If you want to monitor multiple clock domains simultaneously, add additional instances of the logic analyzer to your design, limited only by the available resources in your device.

Configure the SignalTap II Embedded Logic Analyzer

After the SignalTap II Embedded Logic Analyzer is added to your design, configure it to monitor the signals you want. You can manually add signals or use a plug-in, such as the Nios II embedded 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.

Define Trigger Conditions

The SignalTap II Embedded Logic Analyzer captures data continuously while it 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 Embedded Logic Analyzer lets you 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 give you the ability to capture data from trigger events occurring immediately after the device enters user-mode after configuration.

Compile the Design

With the **.stp** file 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 Embedded Logic Analyzer, along with Quartus II incremental compilation, to reduce recompile times.

Program the Target Device or Devices

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

Run the SignalTap II Embedded 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** file for analysis.

View, Analyze, and Use Captured Data

After you have captured data and read it into the **.stp** file, it is available for analysis and use in the debugging process. Either manually or with a plug-in, set up mnemonic tables to make it easier to read and interpret 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 it to other formats for sharing and further study.

Add the SignalTap II Embedded Logic Analyzer to Your Design

Because the SignalTap II Embedded Logic Analyzer is implemented in logic on your target device, it must be added to your FPGA design as another part of the design itself. There are two ways to generate the SignalTap II Embedded Logic Analyzer and add it to your design for debugging:

- Create an **.stp** file and use the SignalTap II Editor to configure the details of the logic analyzer

or

- Create and configure the **.stp** file with the MegaWizard Plug-In Manager and instantiate it in your design

Creating and Enabling a SignalTap II File

To create an embedded logic analyzer, use an existing **.stp** file or create a new file. After a file is created or selected, it must be enabled in the project where it is used.

Creating a SignalTap II File

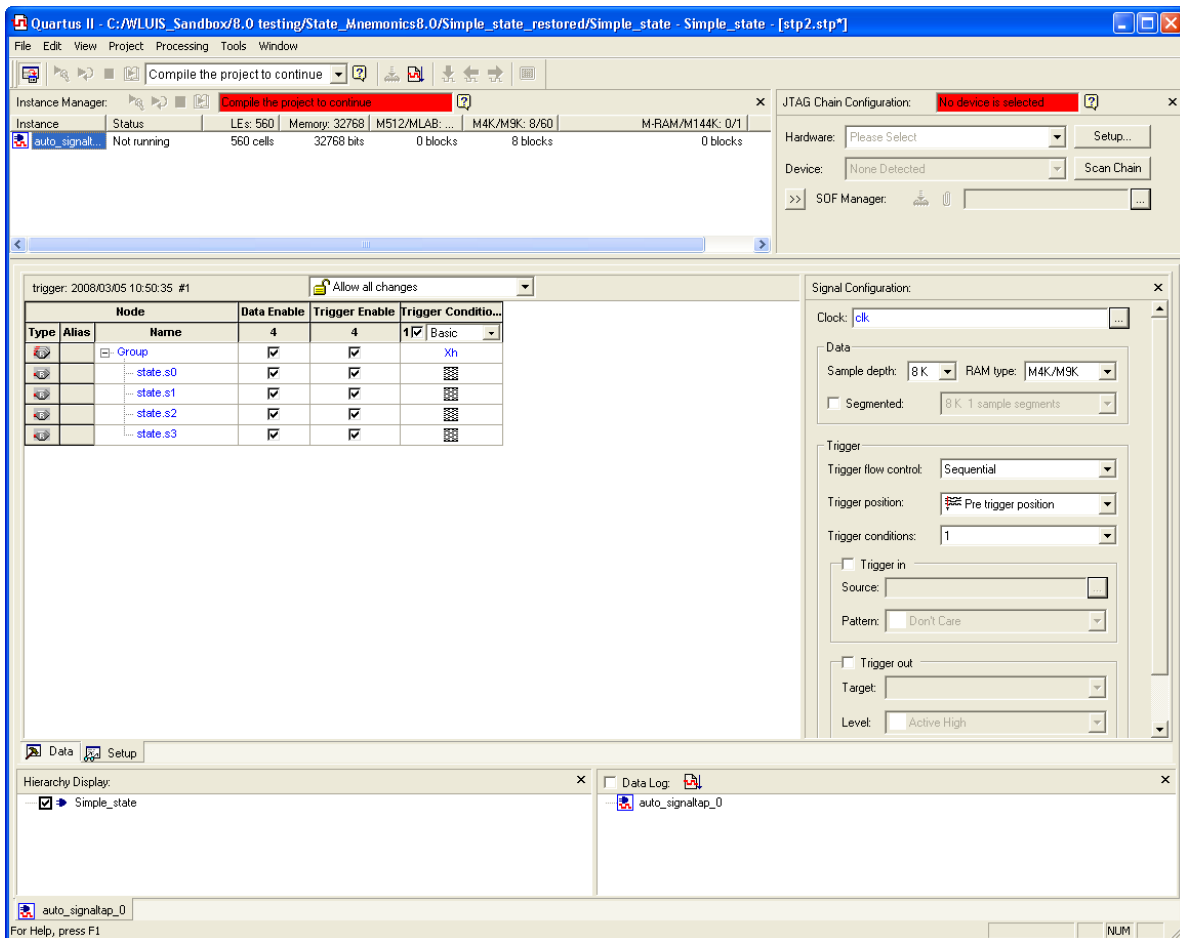
The **.stp** file contains the SignalTap II Embedded Logic Analyzer settings and the captured data for viewing and analysis. To create a new **.stp** file, perform the following steps:

1. On the File menu, click **New**.
2. In the **New** dialog box, click the **Other Files** tab and select **SignalTap II Logic Analyzer File**.
3. Click **OK**.

To open an existing **.stp** file already associated with your project, on the Tools menu, click **SignalTap II Logic Analyzer**. You can also use this method to create a new **.stp** file if no **.stp** file exists for the current project.

To open an existing file, on the File menu, click **Open** and select an **.stp** file (Figure 15-4).

Figure 15-4. SignalTap II Editor



Enabling and Disabling a SignalTap II File for the Current Project

Whenever you save a new **.stp** file, the Quartus II software asks you if you want to enable the file for the current project. However, you can add this file manually, change the selected **.stp** file, or completely disable the logic analyzer by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **SignalTap II Logic Analyzer**. The **SignalTap II Logic Analyzer** page appears.
3. Turn on **Enable SignalTap II Logic Analyzer**. Turn off this option to disable the logic analyzer, completely removing it from your design.
4. In the **SignalTap II File name** box, type the name of the **.stp** file you want to include with your design, or browse to and select a file name.
5. Click **OK**.

Embedding Multiple Analyzers in One FPGA

The SignalTap II Editor includes support for adding multiple logic analyzers using a single **.stp** file. This feature is well-suited for creating a unique logic analyzer for each clock domain in the design.

To create multiple analyzers, on the Edit menu, click **Create Instance**, or right-click in the Instance Manager window and click **Create Instance**.

You can configure each instance of the SignalTap II Embedded Logic Analyzer independently. The icon in the Instance Manager for the currently active instance that is available for configuration is highlighted by a blue box. To configure a different instance, double-click the icon or name of another instance in the Instance Manager.

Monitoring FPGA Resources Used by the SignalTap II Embedded Logic Analyzer

The SignalTap II Embedded 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 embedded 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 section of the SignalTap II 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.

Table 15-2 shows the SignalTap II Embedded Logic Analyzer M4K memory block resource usage for the listed devices per signal width and sample depth.

Table 15-2. SignalTap II Embedded Logic Analyzer M4K Block Utilization for Stratix® II, Stratix, Stratix GX, and Cyclone® Devices (Note 1)

| Signals (Width) | Samples (Depth) | | | |
|-----------------|-----------------|-----|-------|-------|
| | 256 | 512 | 2,048 | 8,192 |
| 8 | < 1 | 1 | 4 | 16 |
| 16 | 1 | 2 | 8 | 32 |
| 32 | 2 | 4 | 16 | 64 |
| 64 | 4 | 8 | 32 | 128 |
| 256 | 16 | 32 | 128 | 512 |

Note to Table 15-2:

- (1) When you configure a SignalTap II Embedded Logic Analyzer, the Instance Manager reports an estimate of the memory bits and logic elements required to implement the given configuration.

Using the MegaWizard Plug-In Manager to Create Your Embedded Logic Analyzer

You can create a SignalTap II Embedded Logic Analyzer instance by using the MegaWizard Plug-In Manager. The MegaWizard Plug-In Manager generates an HDL file that you instantiate in your design.



The State-based trigger flow, the state machine debugging feature, and the storage qualification feature are not supported when using the MegaWizard Plug-In Manager to create the embedded logic analyzer. These features are described in the following sections:

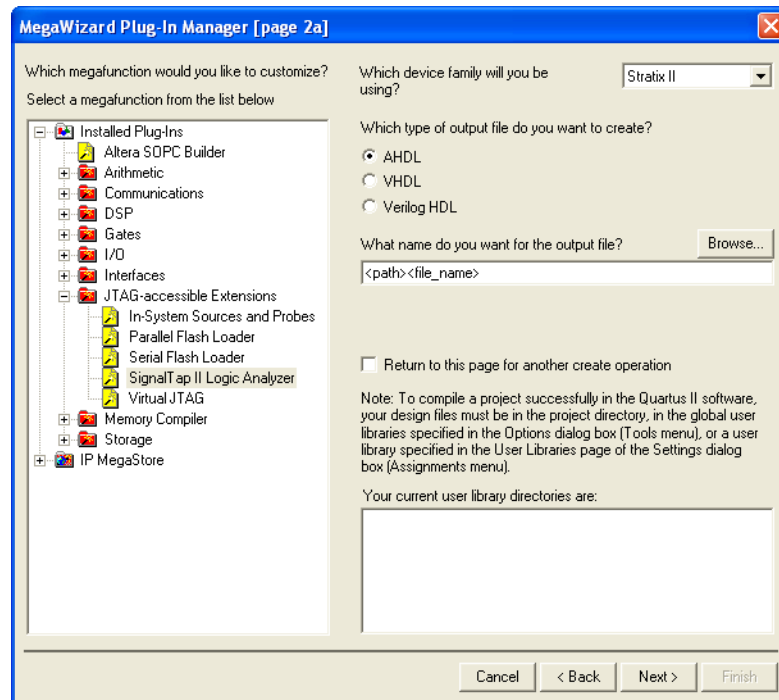
- “Adding Finite State Machine State Encoding Registers” on page 15-20
- “Using the Storage Qualifier Feature” on page 15-25
- “Custom State-Based Triggering” on page 15-38

Creating an HDL Representation Using the MegaWizard Plug-In Manager

The Quartus II software allows you to easily create your SignalTap II Embedded Logic Analyzer using the MegaWizard Plug-In Manager. To implement the SignalTap II megafunction, perform the following steps:

1. On the Tools menu, click **MegaWizard Plug-In Manager**. Page 1 of the **MegaWizard Plug-In Manager** appears.
2. Select **Create a new custom megafunction variation**.
3. Click **Next**.
4. In the **Installed Plug-Ins** list, expand the **JTAG-accessible Extensions** folder and select **SignalTap II Embedded Logic Analyzer**. Select an output file type and enter the desired name of the SignalTap II megafunction. You can choose AHDL (.tdf), VHDL (.vhd), or Verilog HDL (.v) as the output file type (Figure 15-5).

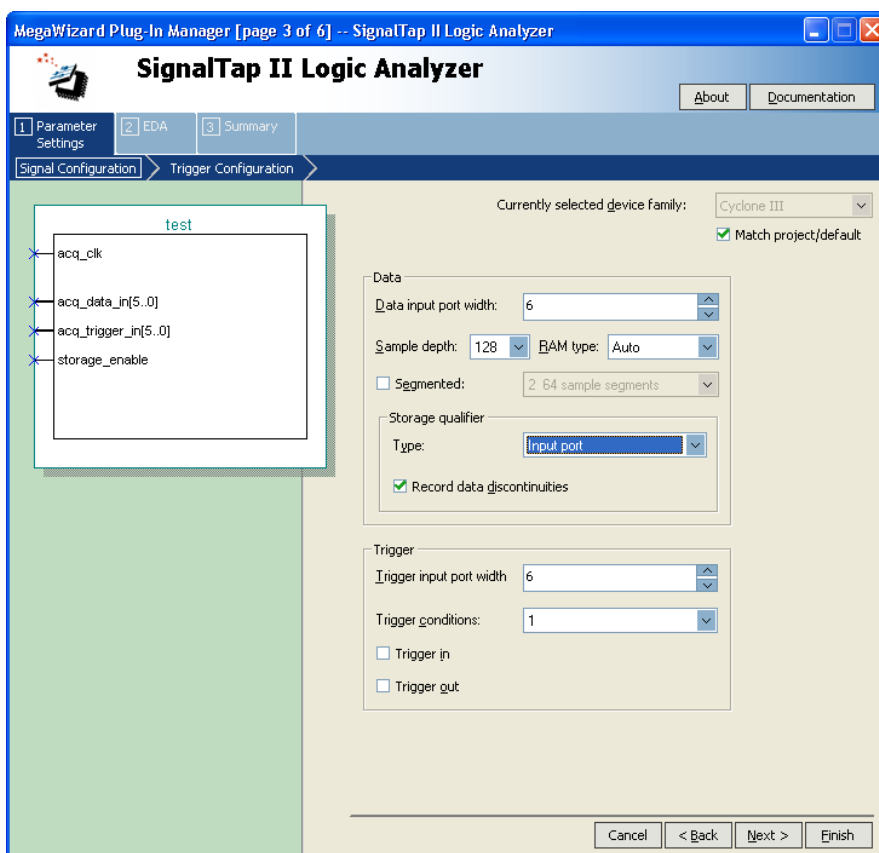
Figure 15-5. Creating the SignalTap II Embedded Logic Analyzer in the MegaWizard Plug-In Manager



5. Click **Next**.
6. Configure the analyzer by specifying the **Sample depth**, **RAM Type**, **Data input port width**, **Trigger levels**, **Trigger input port width**, whether to enable an external **Trigger in** or **Trigger out**, whether to enable the **Segmented** memory buffer option, and whether to enable the Storage Qualifier for non-segmented buffers (Figure 15-6).

For information about these settings, refer to “[Configure the SignalTap II Embedded Logic Analyzer](#)” on page 15-14 and “[Define Triggers](#)” on page 15-33.

Figure 15-6. Select Embedded Logic Analyzer Parameters

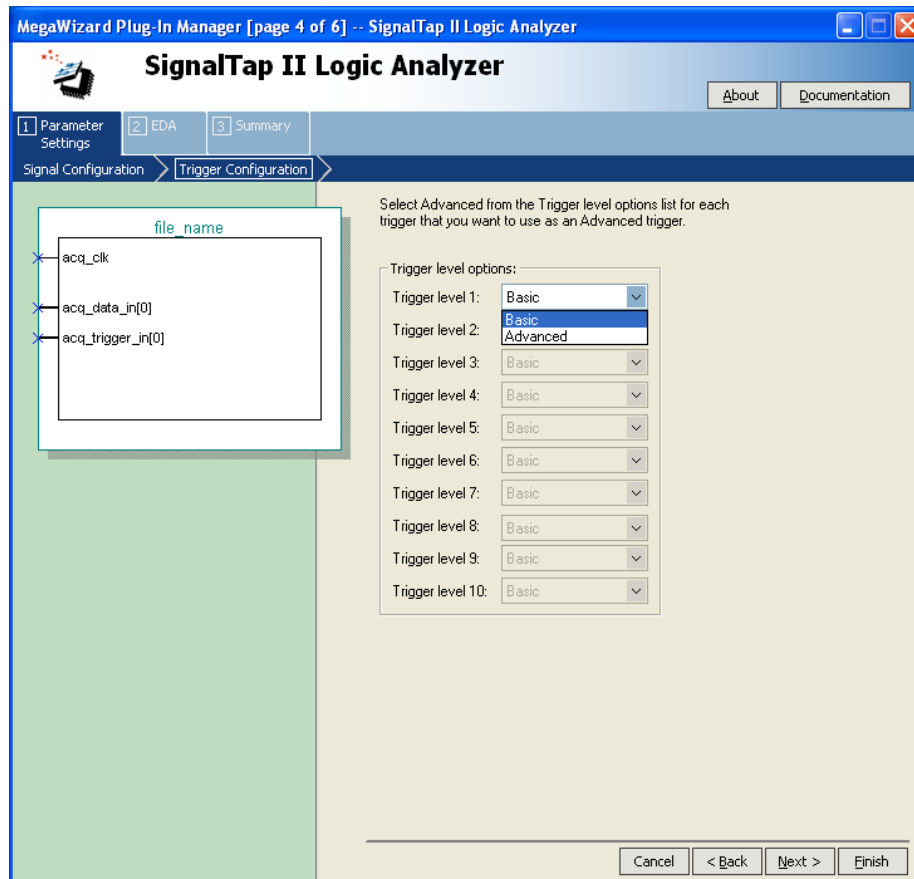


7. Click **Next**.
8. Set the **Trigger level** options by selecting **Basic** or **Advanced** (Figure 15-7). If you select **Advanced** for any trigger level, the next page of the MegaWizard Plug-In Manager displays the Advanced Trigger Condition Editor. You can configure an advanced trigger expression using the number of signals you specified for the trigger input port width.



You cannot define a Power-Up Trigger using the MegaWizard Plug-In Manager. Refer to “[Define Triggers](#)” on page 15-33 to learn how to do this using the .stp file.

Figure 15–7. MegaWizard Basic and Advanced Trigger Options



9. On the final page of the MegaWizard Plug-In Manager, select any additional files you want to create and click **Finish** to create an HDL representation of the SignalTap II Embedded Logic Analyzer.

For information about the configuration settings options in the MegaWizard Plug-In Manager, refer to [“Configure the SignalTap II Embedded Logic Analyzer” on page 15–14](#). For information about defining triggers, refer to [“Define Triggers” on page 15–33](#).

SignalTap II Megafunction Ports

Table 15–3 provides information about the SignalTap II megafunction ports.



For the most current information about the ports and parameters for this megafunction, refer to the latest version of the Quartus II Help.

Table 15–3. SignalTap II Megafunction Ports (Part 1 of 2)

| Port Name | Type | Required | Description |
|----------------|-------|----------|--|
| acq_data_in | Input | No | This set of signals represents signals that are monitored in the SignalTap II Embedded Logic Analyzer. |
| acq_trigger_in | Input | No | This set of signals represents signals that are used to trigger the analyzer. |

Table 15-3. SignalTap II Megafunction Ports (Part 2 of 2)

| Port Name | Type | Required | Description |
|----------------|--------|----------|---|
| acq_clk | Input | Yes | This port represents the sampling clock that the SignalTap II Embedded Logic Analyzer uses to capture data. |
| trigger_in | Input | No | This signal is used to trigger the SignalTap II Embedded Logic Analyzer. |
| trigger_out | Output | No | This signal is enabled when the trigger event occurs. |
| storage_enable | Input | No | This signal is used to enable a write transaction into the acquisition buffer. |

Instantiating the SignalTap II Embedded Logic Analyzer in Your HDL

Add the code from the files that are generated by the MegaWizard Plug-In Manager to your design, mapping the signals in your design to the appropriate SignalTap II megafunction ports. You can instantiate up to 127 analyzers in your design, or as many as physically fit in the FPGA. Once you have instantiated the **.stp** file in your HDL file, compile your Quartus II project to fit the logic analyzer in the target FPGA.

To capture and view the data, create an **.stp** file from your SignalTap II HDL output file. To do this, on the File menu, point to **Create/Update** and click **Create SignalTap II File from Design Instance(s)**.



If you make any changes to your design or the SignalTap II instance, recreate or update the **.stp** file using the **Create/Update** command. This ensures that the **.stp** file is always compatible with the SignalTap II instance in your design. If the **.stp** file is not compatible with the SignalTap II instance in your design, you may not be able to control the SignalTap II Embedded Logic Analyzer after it is programmed into your device.

For information about **.stp** file compatibility with programmed SignalTap II instances, refer to [“Program the Target Device or Devices” on page 15-59](#).

Configure the SignalTap II Embedded Logic Analyzer

The **.stp** file provides many options for configuring instances of the logic analyzer. Some of the settings are similar to those found on traditional external logic analyzers. Other settings are unique to the SignalTap II Embedded Logic Analyzer because of the requirements for configuring an embedded logic analyzer. All settings give you the ability to configure the logic analyzer the way you want to help debug your design.



Some settings can only be adjusted when you are viewing Run-Time Trigger conditions instead of Power-Up Trigger conditions. To learn about Power-Up Triggers and viewing different trigger conditions, refer to [“Creating a Power-Up Trigger” on page 15-49](#).

Assigning an Acquisition Clock

Assign a clock signal to control the acquisition of data by the SignalTap II Embedded 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.

To assign an acquisition clock, perform the following steps:

1. In the SignalTap II Logic Analyzer window, click the **Setup** tab.
2. In the **Signal Configuration** pane, next to the **Clock** field, click **Browse**. The **Node Finder** dialog box appears.
3. From the **Filter** list, select **SignalTap II: post-fitting**
or
SignalTap II: pre-synthesis.
4. In the **Named** field, type the exact name of a node that you want to use as your sample clock, or search for a node using a partial name and wildcard characters.
5. To start the node search, click **List**.
6. In the **Nodes Found** list, select the node that represents the design's global clock signal.
7. Add the selected node name to the **Selected Nodes** list by clicking ">" or by double-clicking the node name.
8. Click **OK**. The node is now specified as the acquisition clock in the SignalTap II Editor.

If you do not assign an acquisition clock in the SignalTap II 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.



For information about assigning signals to pins, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Adding Signals to the SignalTap II File

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

- **Pre-synthesis**—This signal 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.



If you are not using incremental compilation, add only pre-synthesis signals to your **.stp** file. Using pre-synthesis is particularly useful if you want to add a new node after you have made design changes. Source file changes appear in the Node Finder after an Analysis and Elaboration has been performed. 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, the signals shown in red text are invalid signals. Unless you are certain that these signals are valid, remove them from the .stp file for correct operation. The SignalTap II Status Indicator also indicates if an invalid node name exists in the .stp file.

As a general guideline, signals can be tapped 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, all connections made to the SignalTap II Embedded Logic Analyzer are made prior to synthesis. Logic and routing resources are allocated during recompilation to make the connection as if a change in your design files had been made. As such, 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 Embedded Logic Analyzer are the signal names from the actual atoms in your post-fit netlist. A connection can only be made 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 Embedded 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.



If you are tapping the signal from the atom that is driving an IOE, be aware that the signal may be inverted due to NOT-gate push back. 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 are also helpful in finding post-fit node names.



For information about cross-probing to source design file and other Quartus II windows, refer to the *Analyzing Designs with Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*.

For more information about the use of incremental compilation with the SignalTap II Embedded Logic Analyzer, refer to “*Faster Compilations with Quartus II Incremental Compilation*” on page 15-53.

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 are fanning out from a node, making it difficult to decipher which signal nets they actually represent. This can cause a problem when you use the incremental

compilation flow with the SignalTap II Embedded Logic Analyzer. Because only post-fitting signals can be added to the SignalTap II Embedded Logic Analyzer in partitions of type **post-fit**, RTL signals that you want to monitor may not be available, preventing their usage. 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



For more information about using these attributes, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

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 Embedded Logic Analyzer. This is often necessary when a plug-in is used to add a group of signals for a particular IP.

To prevent the Quartus II software from optimizing away debugging signals on IP cores, perform the following steps:

1. In the Quartus II GUI, on the Assignments menu, click **Settings**.
2. In the **Category** list, select **Analysis & Synthesis Settings**.
3. Turn on **Create debugging nodes** for IP cores to make these nodes available to the SignalTap II Embedded Logic Analyzer.

Assigning Data Signals Using the Node Finder

To assign data signals, perform the following steps:

1. Perform Analysis and Elaboration, Analysis and Synthesis, or fully compile your design.
2. In the SignalTap II Logic Analyzer window, click the **Setup** tab.
3. Double-click anywhere in the node list of the SignalTap II Editor to open the **Node Finder** dialog box.
4. In the **Fitter** list, select **SignalTap II: pre-synthesis** or **SignalTap II: post-fitting**. Only signals listed under one of these filters can be added to the **SignalTap II node** list. Signals cannot be selected from any other filters.



Altera recommends that you do not add a mix of pre-synthesis and post-fitting signals within the same partition. For more details, refer to “*Using Incremental Compilation with the SignalTap II Embedded Logic Analyzer*” on page 15-55.

If you use incremental compilation flow with the SignalTap II Embedded Logic Analyzer, pre-synthesis nodes may not be connected to the SignalTap II Embedded 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.

1. In the **Named** field, type a node name, or search for a particular node by entering a partial node name along with wildcard characters. To start the node name search, click **List**.
2. In the **Nodes Found** list, select the node or bus you want to add to the **.stp** file.
3. Add the selected node name(s) to the **Selected Nodes** list by clicking ">" or by double-clicking the node name(s).
4. To insert the selected nodes in the **.stp** file, click **OK**. With the default colors set for the SignalTap II Embedded Logic Analyzer, a pre-synthesis signal in the list is shown in black; a post-fitting signal is shown in blue.



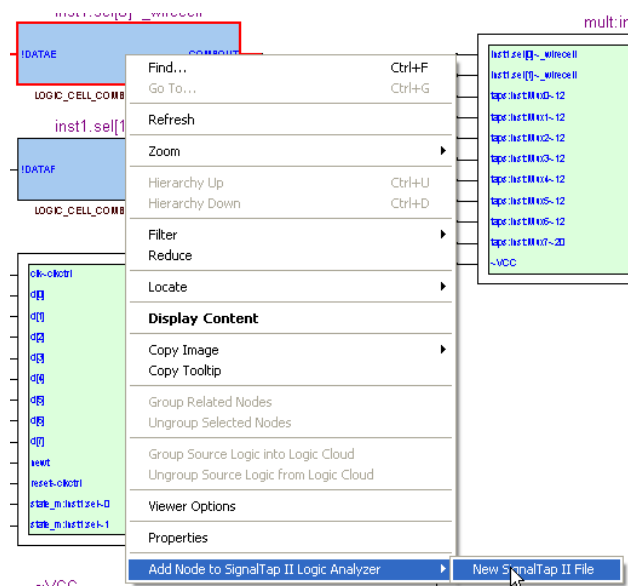
You can also drag and drop signals from the **Node Finder** dialog box into an **.stp** file.

Assigning Data Signals Using the Technology Map Viewer

Starting with Quartus II software version 8.0, 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** file for your design or a new **.stp** file.

Figure 15-8 shows the right-click menu for adding a node using the Technology map viewer.

Figure 15-8. Finding Data Signals Using the Technology Map Viewer



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** file. 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.

For information about using signals in the node list to create SignalTap II trigger conditions, refer to [“Define Triggers” on page 15-33](#).

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 megafunction**—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/DDR2 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 through the use of a plug-in. The SignalTap II Embedded Logic Analyzer comes with one plug-in already installed for the Nios II processor. Besides easy signal addition, plug-ins also provide a number of other 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.

For information about the other features plug-ins provided, refer to [“Define Triggers” on page 15-33](#) and [“View, Analyze, and Use Captured Data” on page 15-66](#).

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

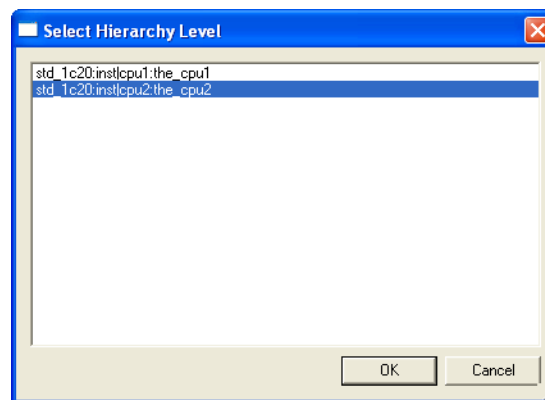
1. Right-click in the node list. On the Add Nodes with Plug-In submenu, click the name of the plug-in you want to use, such as the included plug-in named **Nios II**.



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

2. The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design (Figure 15-9). Select the IP that contains the signals you want to monitor with the plug-in and click **OK**.

Figure 15-9. IP Hierarchy Selection



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 set any available options for the plug-in. With the Nios II plug-in, you can optionally select an .elf file containing program symbols from your Nios II Integrated Development Environment (IDE) software design. Set options for the selected plug-in as desired and click **OK**.



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

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

Adding Finite State Machine State Encoding Registers

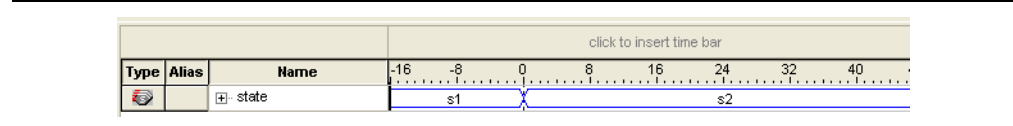
Finding the signals to debug Finite State Machines (FSM) can be challenging. Finding nodes from the post-fit netlist may be impossible, as FSM encoding signals may be changed or optimized away during synthesis and place-and-route. If you are able to find all of the relevant nodes in the post-fit netlist 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.

Beginning with Quartus II software version 8.0, the SignalTap II GUI can detect FSMs in your compiled design. The SignalTap II configuration automatically tracks the FSM state signals as well as state encoding through the compilation process. Right-click dialog boxes from the SignalTap II GUI allow you to add all of the FSM state signals to your embedded 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 easily. The FSM debugging feature supports adding FSM signals from both the pre-synthesis and post-fit netlists.

Figure 15-10 shows the waveform viewer with decoded signal values from a state machine added with the FSM debugging feature.

Figure 15-10. Decoded FSM Mnemonics




For coding guidelines for specifying FSM in Verilog and VHDL, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

To add pre-synthesis FSM signals to the configuration file, perform the following steps after running Analysis and Elaboration on your design:

1. Create a new **.stp** file or use an existing **.stp** file.

 Any **.stp** files that the MegaWizard Plug-In Manager creates from instantiations are not supported for this feature.

2. In the SignalTap II setup tab, right-click anywhere on the node list and select **Add State Machine Nodes**. The **Add State Machine Nodes** dialog box appears. This dialog box lists all the FSMs that have been found in your design.

 For the SignalTap II GUI to detect pre-synthesis state-machine signals, perform Analysis and Elaboration of your design.

3. From the Netlist pull-down menu, select **Pre-Synthesis**.
4. Select the desired FSM.
5. Click **OK**. This adds the FSM nodes to the configuration file. A mnemonic table is automatically applied to the FSM signal group.

To add post-fit FSM signals to the configuration file, perform the following steps after performing a full compile of your design:

1. Set the design partition of the FSM that you want to debug to post-fit.
2. Enable the **.stp** file for the Quartus II project using the **SignalTap II Embedded Logic Analyzer** page of the **Settings** dialog box. You can either create a new **.stp** file or use an existing **.stp** file.

 For the SignalTap II GUI to detect post-fit state-machine signals, perform a full compile of your design.

3. In the SignalTap II setup tab, right-click anywhere on the node list and select **Add State Machine Nodes**. The **Add State Machine Nodes** dialog box appears. This dialog box lists all the FSMs that have been found in your design.

4. From the Netlist pull-down menu, select **Post-Fit**.
5. Select the desired FSM.
6. Click **OK**. This adds the FSM nodes to the configuration file. A mnemonic table is automatically applied to the FSM signal group.

Modifying and Restoring Mnemonic Tables for State Machines

When you add FSM state signals via the FSM debugging feature, the SignalTap II 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. If you would like to restore a FSM mnemonic table to a new record, uncheck the **Overwrite existing mnemonic table** option in the **Recreate State Machine Mnemonics** dialog box.



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.

For more information about using Mnemonics, refer to “[Creating Mnemonics for Bit Patterns](#)” on page 15-69.

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 have used a third-party synthesis tool.

If you are adding post-fit FSM signals, the SignalTap II FSM debug feature may not be able to track all of the 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 are still able to 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 set 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

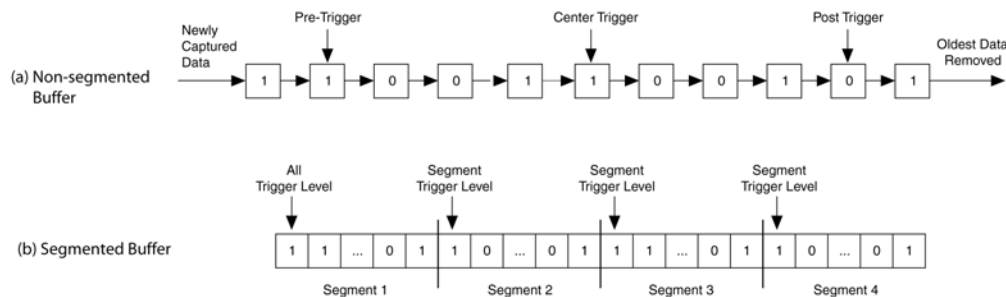
When you use the SignalTap II Embedded Logic Analyzer with some devices, you have the option to select the RAM type where acquisition data is stored. RAM selection allows you to preserve a specific memory block for your design and allocate another portion of memory for SignalTap II data acquisition. For example, if your design implements a large buffering application such as a system cache, it is ideal to place this application into M-RAM blocks so that the remaining M512 or M4K blocks are used for SignalTap II data acquisition.

To select the RAM type to use for the SignalTap II 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 Embedded 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 Embedded Logic Analyzer—a non-segmented buffer and a segmented buffer. With a non-segmented buffer, the SignalTap II Embedded Logic Analyzer treats entire memory space as a single FIFO, continuously filling the buffer until the embedded 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 Embedded 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 15-11](#) illustrates the differences between the two buffer types.

Figure 15-11. Buffer Type Comparison in the SignalTap II Embedded Logic Analyzer (Note 1)**Note to Figure 15-11:**

- (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](#)” on page 15-48 for more details.
- (2) Each segment is treated like a FIFO, and behaves as the non-segmented buffer shown in (a).

For more information about the storage qualification feature, refer to “[Using the Storage Qualifier Feature](#)” on page 15-25.

Non-Segmented Buffer

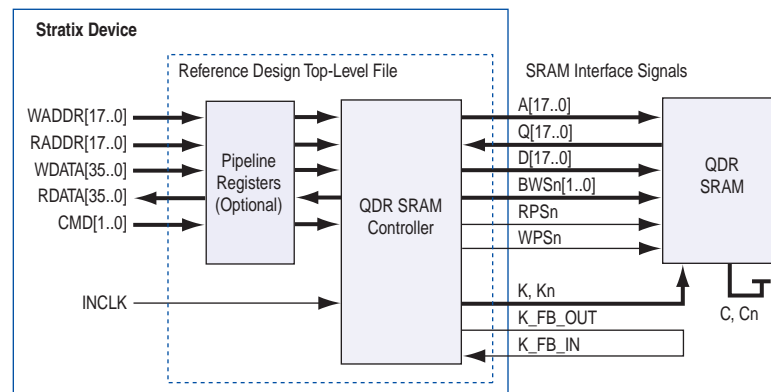
The non-segmented buffer (also known as a circular buffer) shown in [Figure 15-11 \(a\)](#) is the default buffer type used by the SignalTap II Embedded Logic Analyzer. While the logic analyzer is running, data is stored in the buffer until it fills up, at which point new data replaces the oldest data. This continues until a specified trigger event—that is, a set of trigger conditions—occurs. When this 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 in the **.stp** file. Select a setting from the list to choose whether to capture the majority of the data before (**Post trigger position**), after (**Pre-trigger position**) the trigger occurs, or to center the trigger position in the data (**Center trigger position**). Alternatively, use the custom State-based triggering flow to define a custom trigger position within the capture buffer.

For more information, refer to “[Specifying the Trigger Position](#)” on page 15-48.

Segmented Buffer

A segmented buffer makes it easier to debug systems that contain relatively infrequent recurring events. The acquisition memory is split into a set of evenly sized segments, with a set of trigger conditions defined for each segment. Each segment acts as a non-segmented buffer. [Figure 15-12](#) shows an example of this type of buffer system.

Figure 15-12. Example System that Generates Recurring Events



The SignalTap II Embedded Logic Analyzer verifies the functionality of the design shown in [Figure 15-12](#) to ensure that the correct data is written to the SRAM controller. Buffer acquisition in the SignalTap II Embedded 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 Embedded 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 Editor and select the number of segments to use. In the example, selecting sixty-four 64-sample segments allows you to capture 64 read cycles when the RADDR signal is `H' 0F0F0F0F`.



For more information about buffer acquisition mode, refer to *Setting the Buffer Acquisition Mode* in the Quartus II Help.

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 Embedded Logic Analyzer is to sample data on every clock cycle. With a non-segmented buffer, there is one data window that represents a contiguous snapshot of the datastream. Similarly, segmented buffers use several smaller sampling windows spread out over a larger time scale, 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 Embedded Logic Analyzer, there is no way to 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 each clock cycle during a 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 and covers a larger time scale.

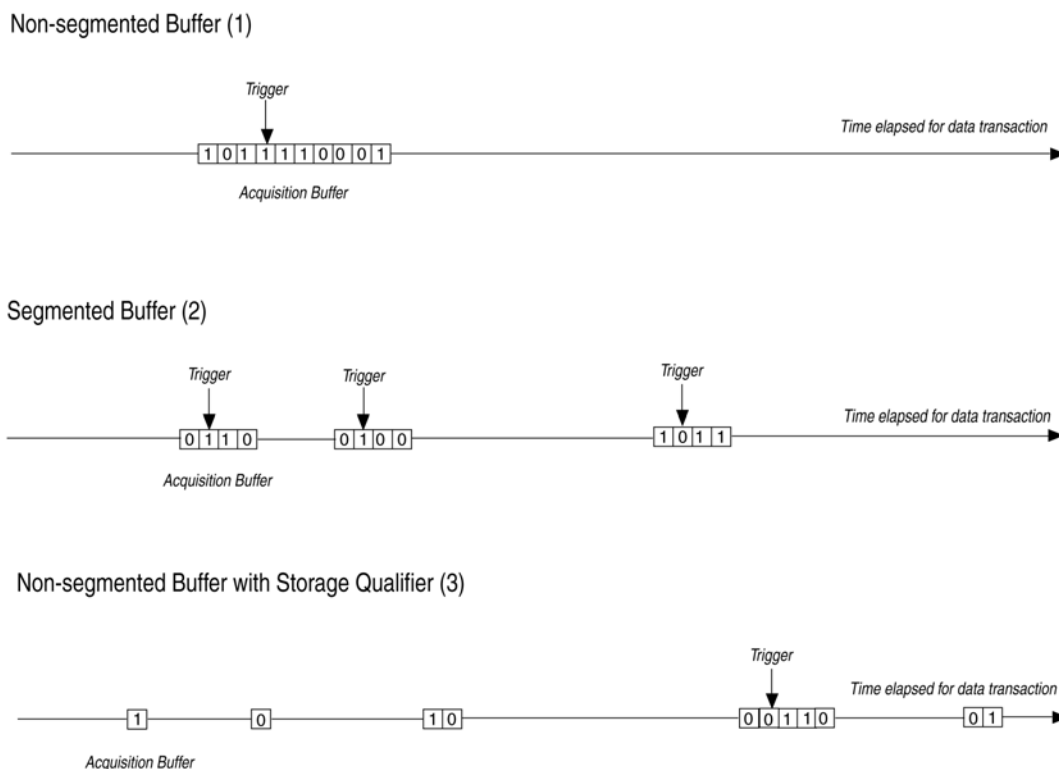
Use of the Storage Qualification feature is similar to an acquisition using a segmented buffer, in that you can create a discontinuity in the capture buffer. Because you can create a discontinuity between any two samples in the buffer, the Storage Qualification feature is equivalent to being able to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

Figure 15-13 illustrates three ways the SignalTap II Embedded Logic Analyzer writes into acquisition memory.



The Storage Qualification feature can only be used with a non-segmented buffer. The MegaWizard Plug-In Manager instantiated flow only supports the Input Port mode for the Storage Qualification feature.

Figure 15-13. Data Acquisition Using Different Modes of Controlling the Acquisition Buffer



Notes to Figure 15-13:

- (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 five 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 Embedded 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.



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

Trigger conditions are defined in “[Define Trigger Conditions](#)” on page 15-6.

The storage qualifier operates independently of the trigger conditions.

The following subsections describe each storage qualification mode from the acquisition buffer.

Input Port Mode

When using the Input port mode, the SignalTap II Embedded 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 Embedded 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 file to create a SignalTap II Embedded Logic Analyzer instance, specify the storage qualifier signal using the input port field located on the **Setup** tab. This port must be specified for your project to compile.

If you are using the MegaWizard Plug-In Manager flow, the storage qualification input port, if specified, will appear in the MegaWizard-generated instantiation template. This port can then be connected to a signal in your RTL.

[Figure 15-14](#) shows a data pattern captured with a segmented buffer. [Figure 15-15](#) shows a capture of the same data pattern with the storage qualification feature enabled.

Figure 15-14. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Input port mode)

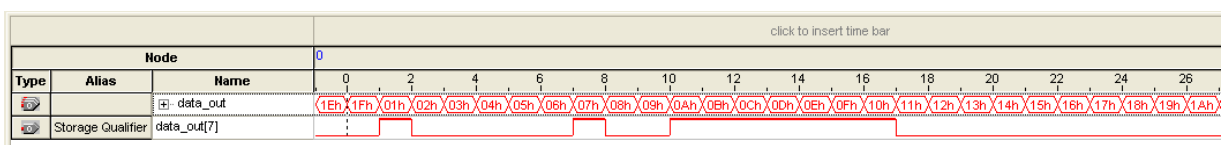
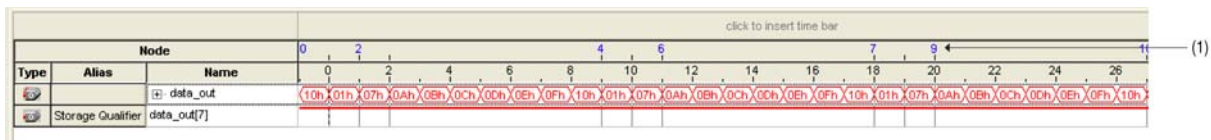
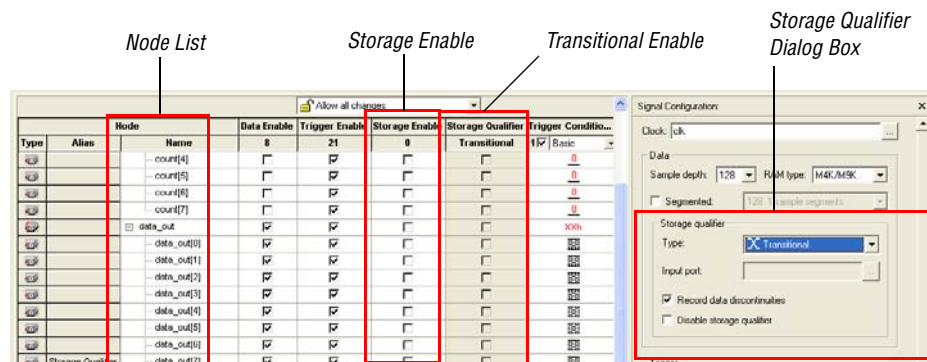
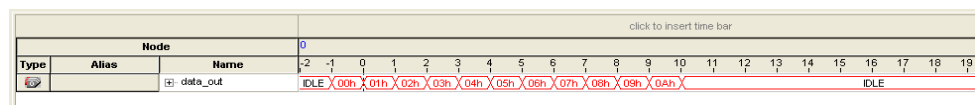
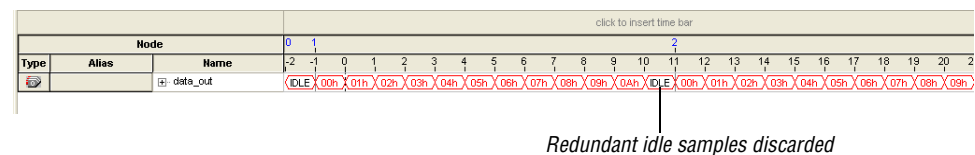


Figure 15-15. Data Acquisition of a Recurring Data Pattern Using an Input Signal as a Storage Qualifier

(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 15-16 shows the transitional storage qualifier setup. Figure 15-17 and Figure 15-18 show captures of a data pattern in continuous capture mode and a data pattern using the Transitional mode for storage qualification.

Figure 15-16. Transitional Storage Qualifier Setup**Figure 15-17.** Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Transitional mode)**Figure 15-18.** Data Acquisition of Recurring Data Pattern Using a Transitional Mode as a Storage Qualifier

Conditional Mode

In Conditional mode, the SignalTap II Embedded 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 Embedded Logic Analyzer writes into the buffer during the clock cycles in which the condition you specify evaluates TRUE.

There are two types of conditions that you can specify: basic and advanced. A basic storage condition matches each signal to one of the following:

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

If a Basic Storage condition is specified for more than one signal, the SignalTap II Embedded 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 15-19 details the conditional storage qualifier setup in the .stp file.

You can set up storage qualification conditions similar to the manner in which trigger conditions are set up. For details about basic and advanced trigger conditions, refer to the sections “Creating Basic Trigger Conditions” on page 15-33 and “Creating Advanced Trigger Conditions” on page 15-34. Figure 15-20 and Figure 15-21 show a data capture with continuous sampling, and the same data pattern using the conditional mode for analysis, respectively.

Figure 15-19. Conditional Storage Qualifier Setup

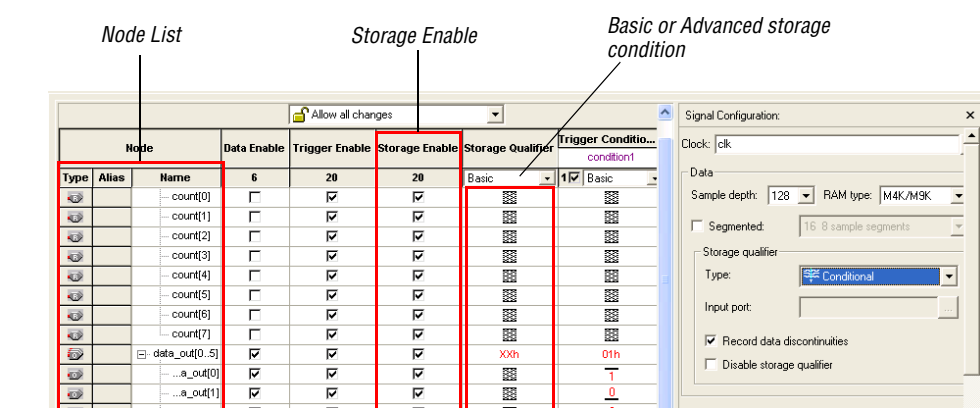
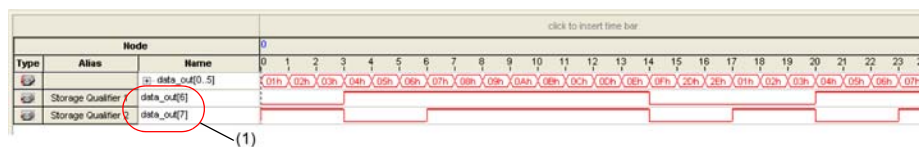
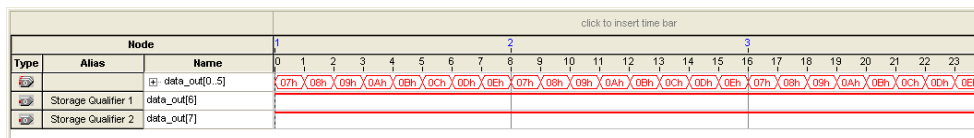


Figure 15-20. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Conditional capture)

(1) Storage Qualifier condition is set up to pause acquisition when the following occurs:

data_out[6] AND data_out[7] = True. Resultant capture with storage qualifier enabled is shown in Figure 14-21.

Figure 15-21. Data Acquisition of a Recurring Data Pattern in Conditional Capture Mode

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 begins to be 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.



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

Figure 15-22 shows the Start/Stop mode storage qualifier setup. Figure 15-23 and Figure 15-24 show captures data pattern in continuous capture mode and a data pattern in using the Start/Stop mode for storage qualification.

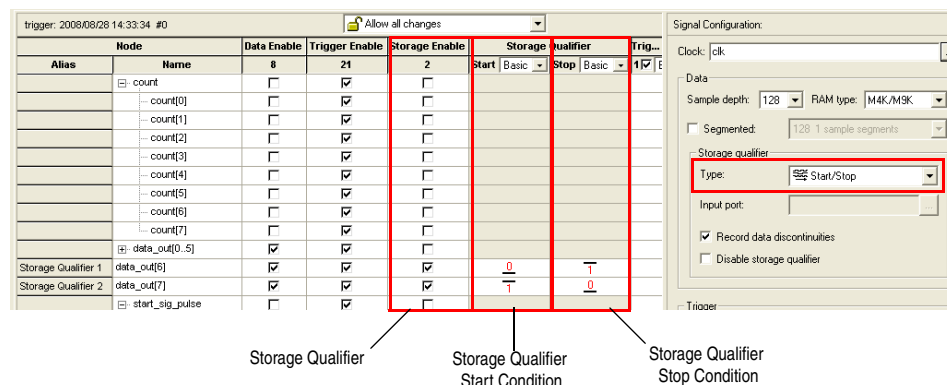
Figure 15-22. Start/Stop Mode Storage Qualifier Setup

Figure 15-23. Data Acquisition of a Recurring Data Pattern in Continuous Mode (to illustrate Start/Stop mode)

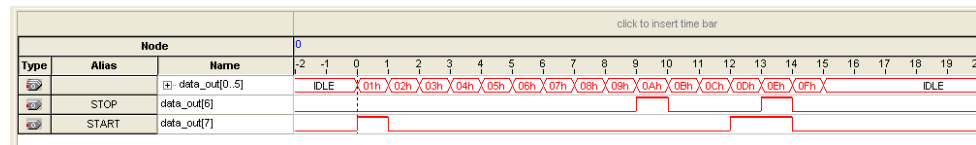
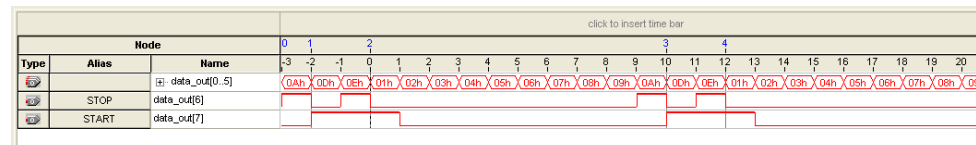


Figure 15-24. Data Acquisition of a Recurring Data Pattern with Start/Stop Storage Qualifier Enabled



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.

For more information about the State-based flow and storage qualification using the State-based trigger flow, refer to the section [“Custom State-Based Triggering” on page 15-38](#).

Showing Data Discontinuities

When you enable the check box option **Record data discontinuities**, the SignalTap II Embedded 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

The **Disable Storage Qualifier** check box allows you to turn off the storage qualifier quickly 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.



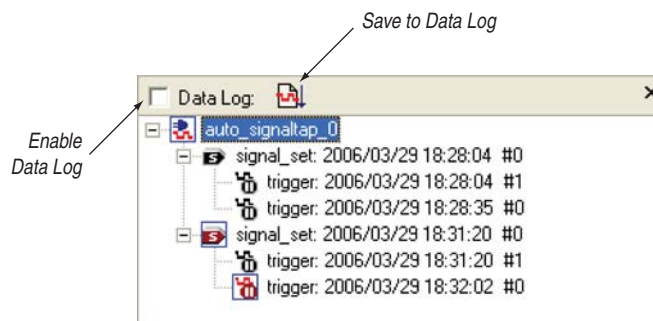
For a detailed explanation of Runtime Reconfigurable options available with the SignalTap II Embedded Logic Analyzer, and storage qualifier application examples using runtime reconfigurable options, refer to [“Runtime Reconfigurable Options” on page 15-63](#).

Managing Multiple SignalTap II Files and Configurations

In some cases you may have more than one **.stp** file 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** file, 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** file for the logic analyzer to run properly when the device is programmed. Managing all of the **.stp** files and their associated settings and programming files is a challenging task. To help you manage everything, use the Data Log feature and the SOF Manager.

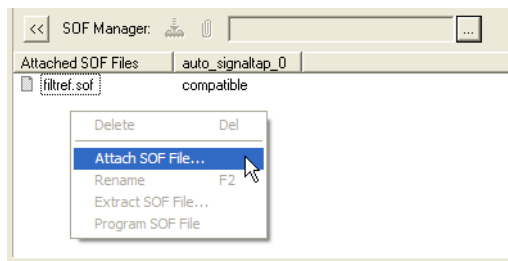
The Data Log allows you to store multiple SignalTap II configurations within a single **.stp** file. [Figure 15-25](#) shows two signal set configurations with multiple trigger conditions in one **.stp** file. To toggle between the active configurations, double-click on an entry in the Data Log. As you toggle between the different configurations, the signal list and trigger conditions change in the **Setup** tab of the **.stp** file. The active configuration displayed in the **.stp** file is indicated by the blue square around the signal set in the 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** button at the top of the Data Log.

Figure 15-25. Data Log



The SOF Manager allows you to embed multiple SOFs into one **.stp** file. Embedding an SOF in an **.stp** file lets you move the **.stp** file to a different location, either on the same computer or across a network, without the need to include the associated **.sof** as a separate file. To embed a new SOF in the **.stp** file, right-click in the SOF Manager, and click **Attach SOF File** ([Figure 15-26](#)).

Figure 15-26. SOF Manager



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

Define Triggers

When you start the SignalTap II Embedded Logic Analyzer, it samples activity continuously from the monitored signals. The SignalTap II Embedded Logic Analyzer “triggers”—that is, 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 set using the SignalTap II Embedded Logic Analyzer.

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 Editor. With the trigger type set to Basic, set the trigger pattern for each signal you have added in the **.stp** file. To set 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** file that have an associated mnemonic table, you can right-click and select an entry from the table to set pre-defined conditions for the trigger.

For more information about creating and using mnemonic tables, refer to [“View, Analyze, and Use Captured Data” on page 15-66](#), 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** file from your Nios II IDE design, you can type the name of a function from your Nios II code. The logic analyzer triggers when the Nios II instruction address matches the address of the 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.

Creating Advanced Trigger Conditions

With the SignalTap II Embedded Logic Analyzer's basic triggering capabilities, you can build more complex triggers utilizing extra logic that enables you to capture data when a particular combination of conditions exist. If you set the trigger type to **Advanced** at the top of the **Trigger Conditions** column in the node list of the SignalTap II Editor, a new tab named **Advanced Trigger** appears where you can build a complex trigger expression using a simple GUI. To build the complex trigger condition in an expression tree, drag-and-drop operators into the Advanced Trigger Configuration Editor window. To configure the operators' settings, double-click or right-click the operators that you have placed and select **Properties**. Table 15-4 lists the operators you can use.

Table 15-4. Advanced Triggering Operators (Note 1)

| 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 15-4:

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

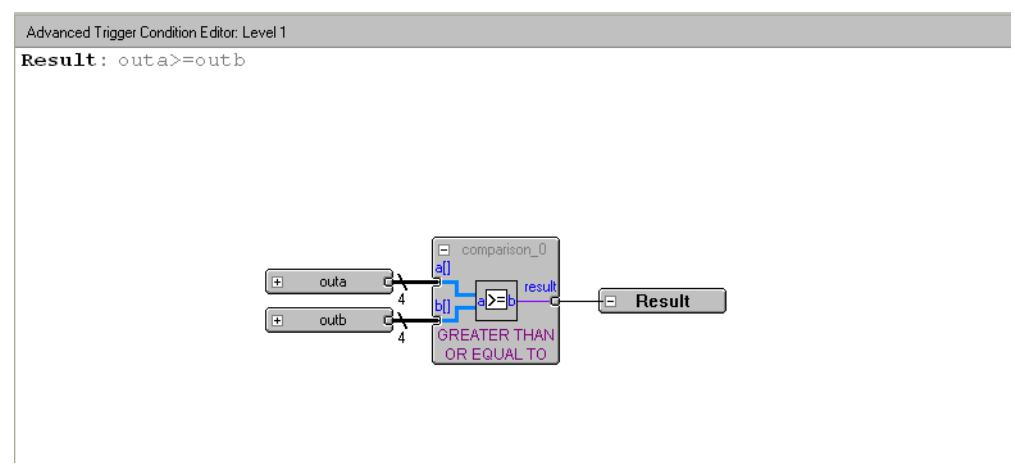
Adding many objects to the Advanced Trigger Condition Editor can make the work space cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the right-click menu and select **Arrange All Objects**. You can also use the **Zoom-Out** command to fit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

Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

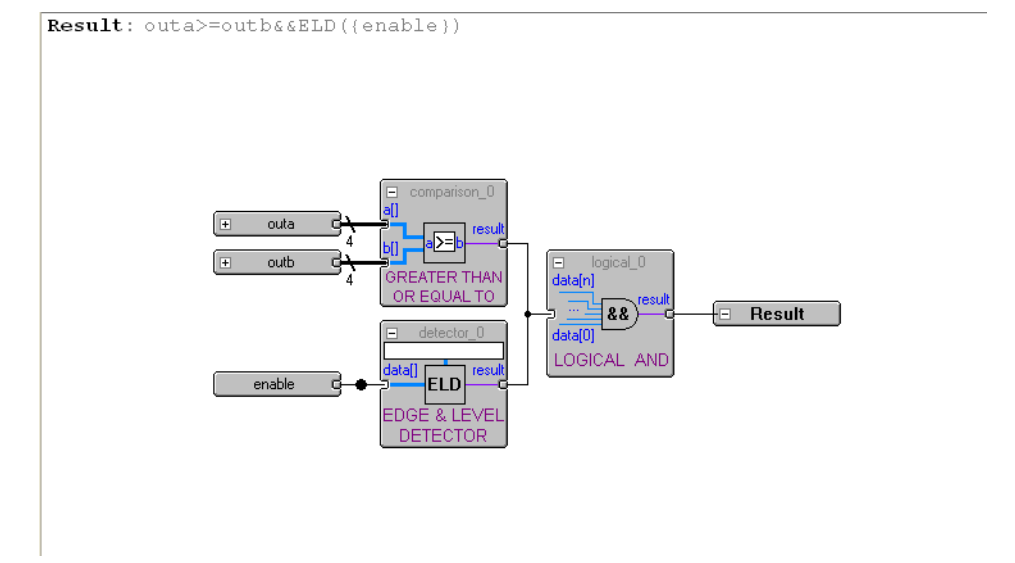
- Trigger when bus outa is greater than or equal to outb (Figure 15-27).

Figure 15-27. Bus outa is Greater Than or Equal to Bus outb



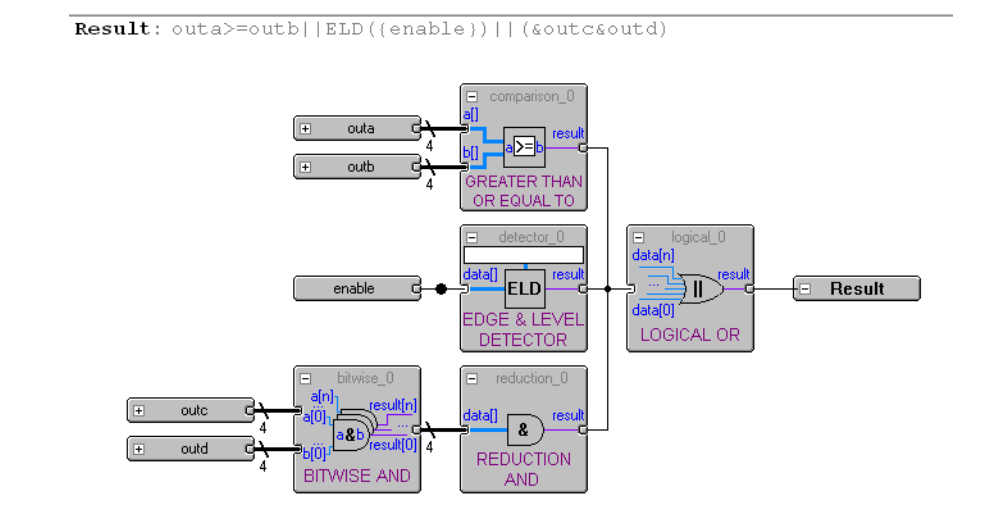
- Trigger when bus outa is greater than or equal to bus outb, and when the enable signal has a rising edge (Figure 15-28).

Figure 15-28. 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 15-29).

Figure 15-29. Bitwise AND Operation



Trigger Condition Flow Control

The SignalTap II Embedded Logic Analyzer offers multiple triggering conditions to give you precise control of the method 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 Embedded Logic Analyzer offers two flow control mechanisms for organizing trigger conditions:

- **Sequential Triggering**—This is 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.
- **Custom State-Based Triggering**—This flow 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 either method with either a segmented or a non-segmented buffer.

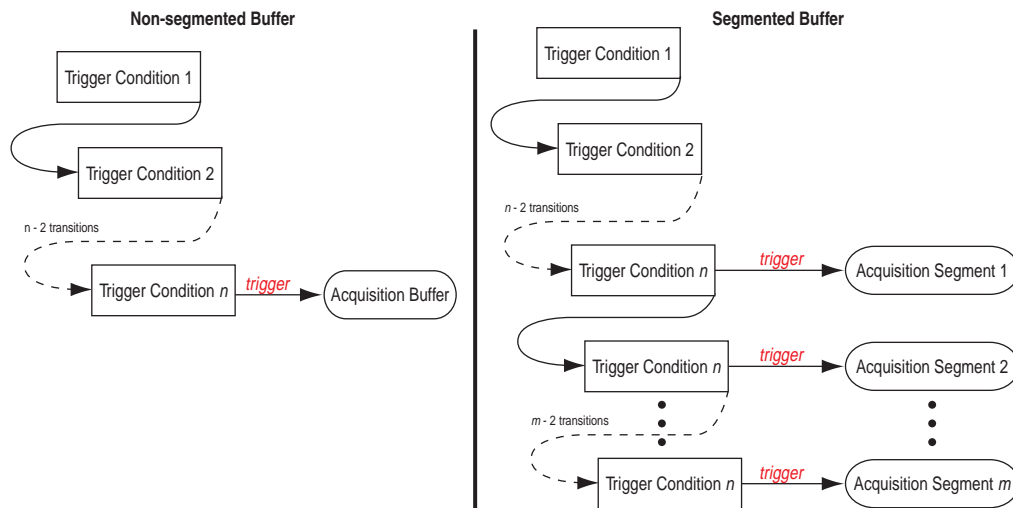
Sequential Triggering

Sequential triggering flow allows you to cascade up to 10 levels of triggering conditions. The SignalTap II Embedded Logic Analyzer sequentially evaluates each of the triggering conditions. When the last triggering condition evaluates to TRUE, the SignalTap II Embedded 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 15-30 illustrates the simple sequential triggering flow for non-segmented and segmented buffers.



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

Figure 15–30. Sequential Triggering Flow (Note 1), (2)

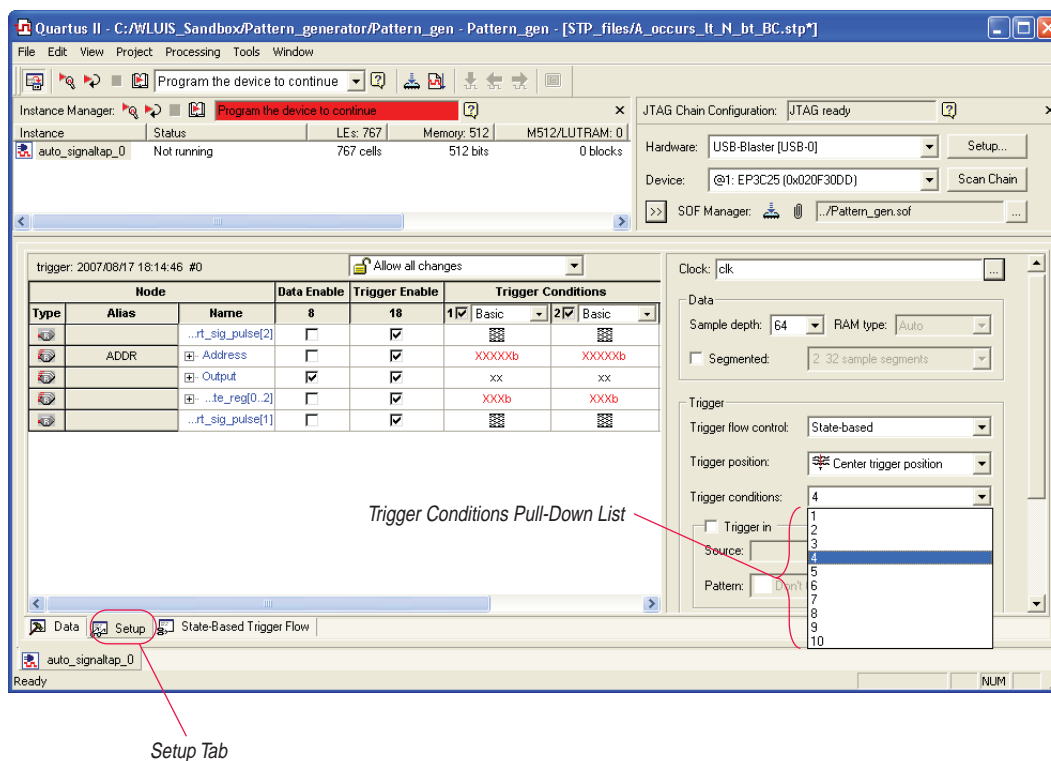


Notes to Figure 15–30:

- (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. For more information about external triggers, refer to “Using External Triggers” on page 15–51.

To configure the SignalTap II Embedded 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 by using the **Trigger Conditions** pull-down list. After you select the desired number of trigger conditions, configure each trigger condition in the node list. To disable any trigger condition, click the check box next to the trigger condition at the top of the column in the node list. Figure 15–31 shows the **Setup** tab for Sequential Triggering.

Figure 15-31. Setup Tab

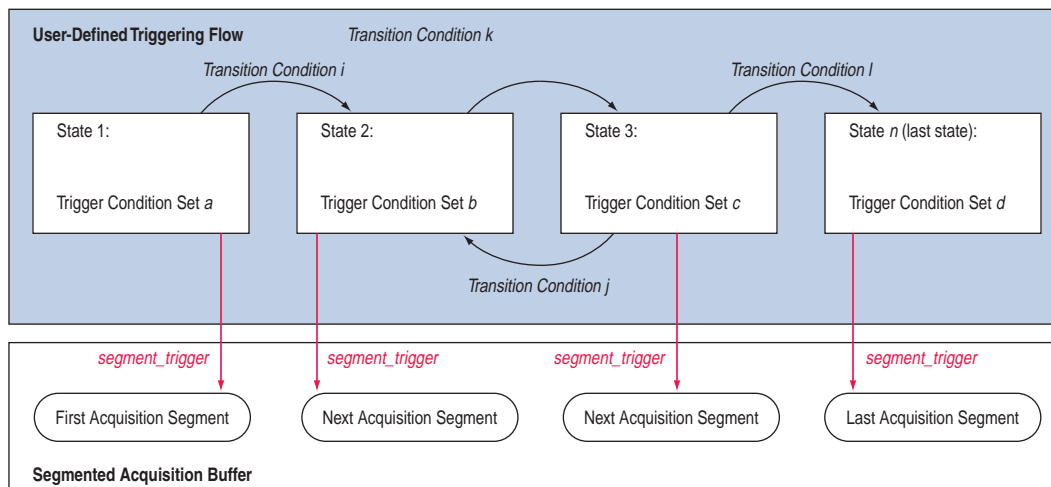


Custom State-Based Triggering

Custom State-based triggering gives you the most control of triggering condition arrangement. This flow gives you the ability 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.

Figure 15-32 illustrates the custom State-based triggering flow. Events that trigger the acquisition buffer are organized by a user-defined state diagram. 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 15-32. Custom State-Based Triggering Flow (Note 1), (2)



Notes to Figure 15-32:

- (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. For more information, refer to "Using External Triggers" on page 15-51.

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 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.

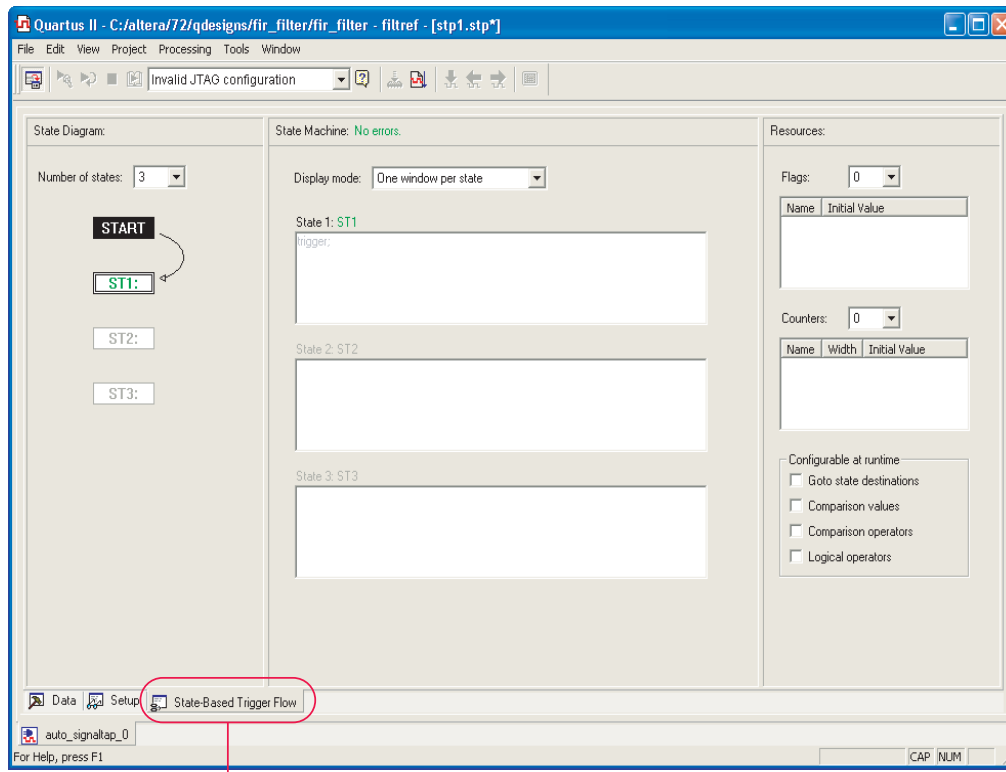
Trigger actions can apply to either a single segment of a segmented acquisition buffer or to the entire non-segmented acquisition buffer. Each trigger action provides you with an optional count that specifies the number of samples to be captured before stopping acquisition of the current segment. The 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, on the **Trigger Flow Control** pull-down list, select **State-based**. (Note that when the **Trigger Flow Control** option is set to **Sequential**, the **State-Based Trigger Flow** tab is hidden.)

Figure 15-33 shows the **State-Based Trigger Flow** tab.

Figure 15-33. State-Based Trigger Flow Tab*State-Based Trigger Flow Tab*

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 all of the states. You can adjust the number of available states by using the pull-down 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.

 For a full description of the SignalTap II Trigger Flow Description Language, refer to “SignalTap II Trigger Flow Description Language” on page 15-42. You can also refer to the Quartus II Help.

The State Machine description text boxes default to show one text box per state. You can optionally 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**.

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 set 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 set 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.

Runtime Reconfigurability—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 15-5 contains a description of options for the State-based trigger flow that can be reconfigured at runtime.


 For a broader discussion about all options that can be changed without incurring a recompile refer to “Runtime Reconfigurable Options” on page 15-63.

Table 15-5. 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 comparison values in Boolean expressions to be modifiable at runtime. In addition, it allows the <code>segment_trigger</code> and trigger action post-fill count argument to be modifiable at runtime. |
| Comparison operators | Allows comparison operators in Boolean expressions to be modifiable at runtime. |
| Logical operators | Allows the logical operators in Boolean expressions to be modifiable at runtime. |

You can restrict changes to your SignalTap configuration to include only the options that do not require a recompilation by using the pull-down menu above the trigger list in the **Setup** tab. The option **Allow trigger condition changes only** restricts changes to only the configuration settings that have the **configurable at runtime** set. With this option enabled, to modify Trigger Flow conditions in the **Custom Trigger Flow** tab, click the desired parameter in the text box and select a new parameter from the menu that appears.



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. For details about the effects of turning off the runtime modifiable options, refer to “[Performance and Resource Considerations](#)” on page 15-57.

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 [Example 15-1](#) 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 “[]” ([Example 15-1](#)).



Examples of Triggering Flow descriptions for common scenarios using the SignalTap II Custom Triggering Flow are provided in “[Custom Triggering Flow Application Examples](#)” on page 15-79.

Example 15-1. Trigger Flow Description Language Format (*Note 1*)

```
state <State_label>:
<action_list>

if( <Boolean_expression> )
<action_list>
[else if ( <boolean_expression> )
<action_list>] (1)
[else
<action_list>]
```

Note to Example 15-1:

(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` embedded 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

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. The supported logical operators are shown in [Table 15-6](#).

Table 15-6. 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. The supported relational operators are shown in [Table 15-7](#).

Table 15-7. Relational Operators

| Operator | Description | Syntax <i>(Note 1) (2)</i> |
|----------|--------------------------|-----------------------------------|
| > | 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 15-7](#):

- (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 15-8](#) shows the description and syntax of each action.

Table 15-8. Resource Manipulation Action (Part 1 of 2)

| Action | Description | Syntax |
|-----------|------------------------------------|---------------------------------|
| increment | Increments a counter resource by 1 | increment <counter_identifier>; |
| decrement | Decrements a counter resource by 1 | decrement <counter_identifier>; |

Table 15-8. Resource Manipulation Action (Part 2 of 2)

| Action | Description | Syntax |
|--------|--|--|
| reset | Resets counter resource to initial value | <code>reset <counter_identifier>;</code> |
| set | Sets a status Flag to 1 | <code>set <register_flag_identifier>;</code> |
| clear | Sets a status Flag to 0 | <code>clear <register_flag_identifier>;</code> |

Buffer Control Action

Buffer control actions specify an action to control the acquisition buffer. [Table 15-9](#) shows the description and syntax of each action.

Table 15-9. 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. | <code>trigger <post-fill_count>;</code> |
| segment_trigger | Ends the acquisition of the current segment. The SignalTap II Embedded 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. | <code>segment_trigger <post-fill_count>;</code> |
| start_store | Asserts the <code>write_enable</code> to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled. | <code>start_store</code> |
| stop_store | De-asserts the <code>write_enable</code> signal to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled. | <code>stop_store</code> |

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.



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.



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.

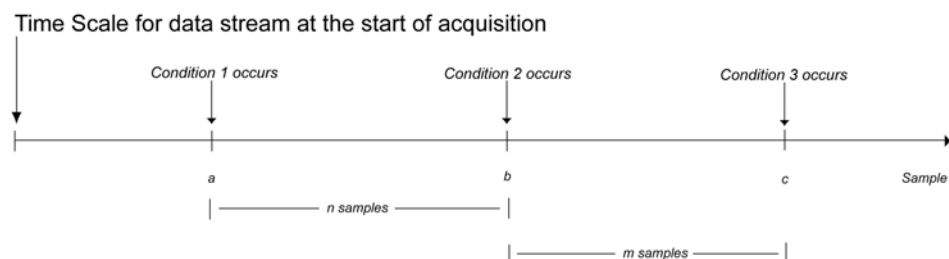
The following examples illustrate the behavior of the State-based trigger flow with the storage qualification commands.

Figure 15-34 shows a hypothetical scenario with three trigger conditions that happen at different points in time after the run analysis button is pushed. The trigger flow description in Example 15-2, when applied to the scenario shown in Figure 15-34, illustrates the functionality of the storage qualification feature for the state-based trigger flow.

Example 15-2. Trigger Flow Description 1

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

Figure 15-34. Capture Scenario for Storage Qualification with the State-Based Trigger Flow



In this example, the SignalTap II Embedded 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.

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 Embedded 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 manually stop and force the acquisition to trigger by using the **Stop Analysis** button. You can use counter values, flags, and the State diagram to help you gauge the execution of the trigger flow. The counter values, flags, and the current state are updated in real-time during a data acquisition.

Figure 15–35. Storage Qualification with Post-Fill Count Value Less than m (Acquisition successfully completes)

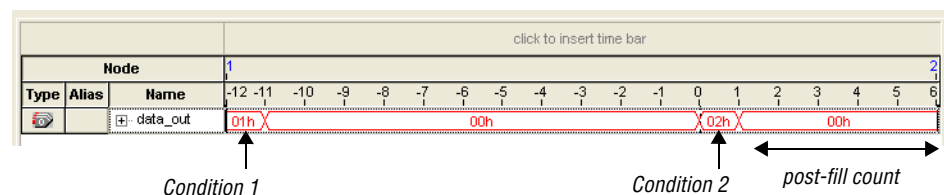
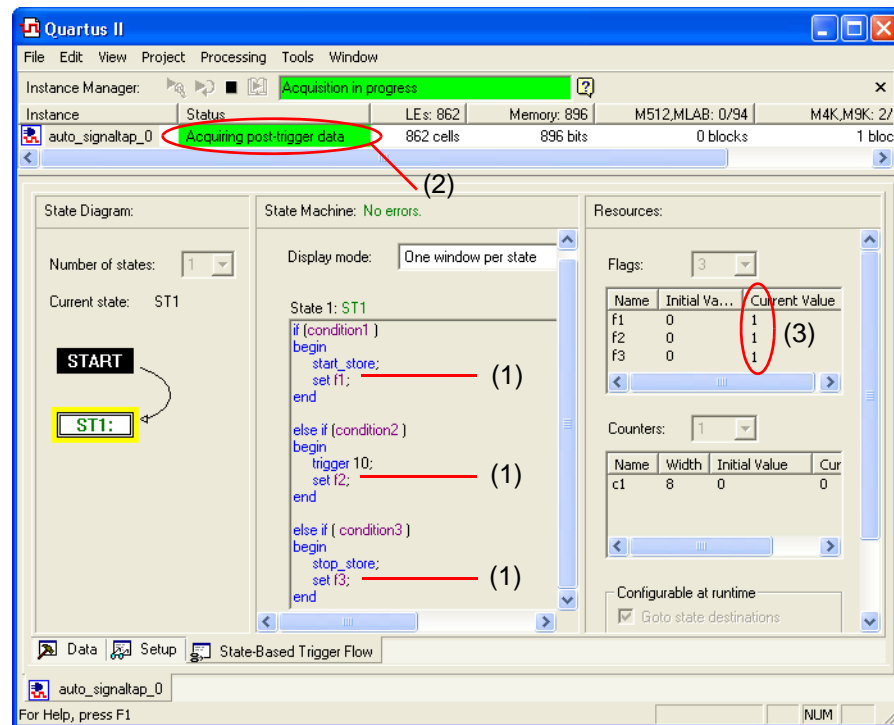
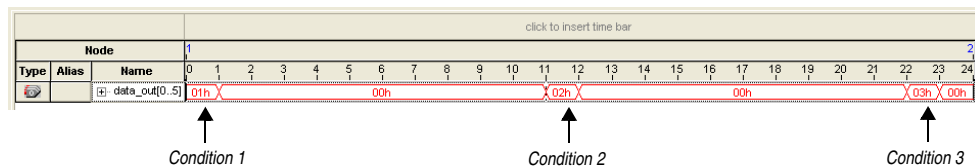


Figure 15-36. Storage Qualification with Post-Fill Count Value Greater than m (Acquisition indefinitely paused)



Waveform after forcing the analysis to stop



- (1) Flags added to trigger flow description to help gauge the execution during runtime.
- (2), (3) Status bar and current value fields update during an acquisition to provide real time status of the data acquisition.

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 get written into the acquisition buffer. **Example 15-3** shows a trigger flow description that skips three clock cycles of samples after hitting condition 1. **Figure 15-37** shows the data transaction on a continuous capture and **Figure 15-39** shows the data capture with the Trigger flow description in **Example 15-3** applied.

Example 15-3. Trigger Flow Description 2

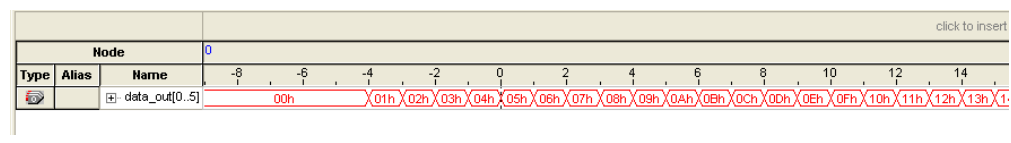
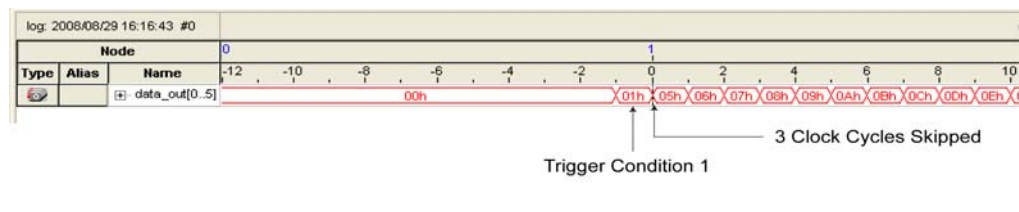
```

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 15-37. Continuous Capture of Data Transaction for Example 2**Figure 15-38.** Capture of Data Transaction with Trigger Flow Description Applied**Specifying the Trigger Position**

The SignalTap II Embedded Logic Analyzer allows you to specify the amount of data that is acquired before and after a trigger event. You can set 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**—This selection saves signal activity that occurred after the trigger (12% pre-trigger, 88% post-trigger).
- **Center**—This selection saves 50% pre-trigger and 50% post-trigger data.
- **Post**—This selection 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. Refer to [Equation 15-1](#):

Equation 15-1.

$$\text{Sample Number of Trigger Position} = (N - \text{Post-Fill Count})$$

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 defined use the post-count setting. Segments that do not have a post-count setting default to the trigger position ratios defined in the **Setup** tab.

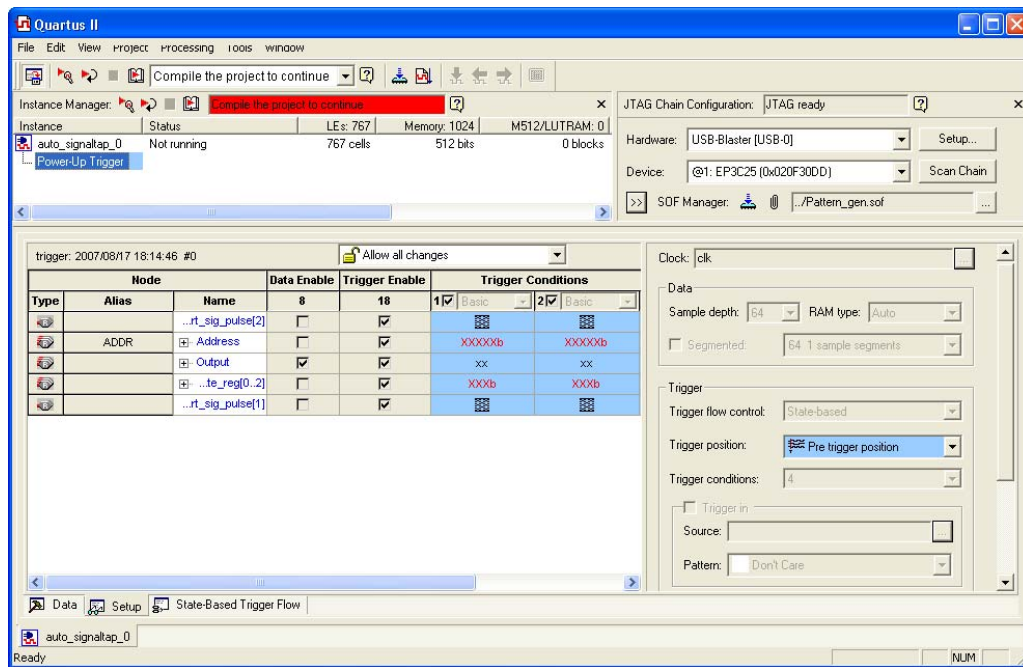
For more details about the custom State-based triggering flow, refer to [“Custom State-Based Triggering” on page 15-38](#) and [“Custom State-Based Triggering” on page 15-38](#).

Creating a Power-Up Trigger

Typically, the SignalTap II Embedded 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 device's JTAG connection 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 Embedded 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. 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 set in the node list. [Figure 15-39](#) shows the SignalTap II Editor when Power-Up Trigger is enabled.

Figure 15-39. SignalTap II 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 it in the same way you do with the regular trigger, also called the Run-Time Trigger. Power-Up Trigger conditions that you can adjust are color coded light blue, while Run-Time Trigger conditions 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. For these changes to be applied to the Power-Up Trigger conditions, first make the changes using the Runtime Trigger conditions.



Any change made to the Power-Up Trigger conditions requires that the SignalTap II Embedded Logic Analyzer be recompiled, 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**.

For information about running the SignalTap II Embedded Logic Analyzer instance with a Power-Up Trigger enabled, refer to [“Running with a Power-Up Trigger” on page 15-62](#).

Using External Triggers

You can create a trigger input that allows you to trigger the SignalTap II Embedded Logic Analyzer from an external source. The external trigger input behaves like trigger condition 1. It is evaluated and must be TRUE before any other configured trigger conditions are evaluated. The analyzer can also supply a signal to trigger external devices or other SignalTap II 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.

Trigger In

To use Trigger In, perform the following steps:

1. In the SignalTap II Editor, click the **Setup** tab.
2. If a Power-Up Trigger is enabled, ensure you are viewing the Runtime Trigger conditions.
3. In the **Signal Configuration** pane, turn on **Trigger In**.
4. In the **Pattern** list, select the condition you want to act as your trigger event. You can set this separately for Runtime or Power-Up Trigger.
5. Click Browse next to the **Source** field in the **Trigger In** pane ([Figure 15-41 on page 15-53](#)). The **Node Finder** dialog box appears.
6. In the **Node Finder** dialog box, select the signal (either an input pin or an internal signal) that you want to drive the Trigger In source and click **OK**.

If you type a new signal name in the **Source** field, you create a new node that you can assign to an input pin in the Pin Planner or Assignment editor. If you leave the **Source** field blank, a default name is entered in the form `auto_stp_trigger_in_<SignalTap instance number>`.

Trigger Out

To use Trigger Out, perform the following steps:

1. In the SignalTap II Editor, click the **Setup** tab.
2. If a Power-Up trigger is enabled, ensure you are viewing the Runtime Trigger conditions.
3. To signify that the trigger event is occurring, in the **Signal Configuration** pane, turn on **Trigger Out** (refer to [Figure 15-40 on page 15-52](#)).
4. In the **Level** list, select the condition you want. You can set this separately for a Run-Time or a Power-Up Trigger.

5. Type a new signal name in the **Target** field. A new node name is created that you must assign to an output pin in the Pin Planner or Assignment Editor.

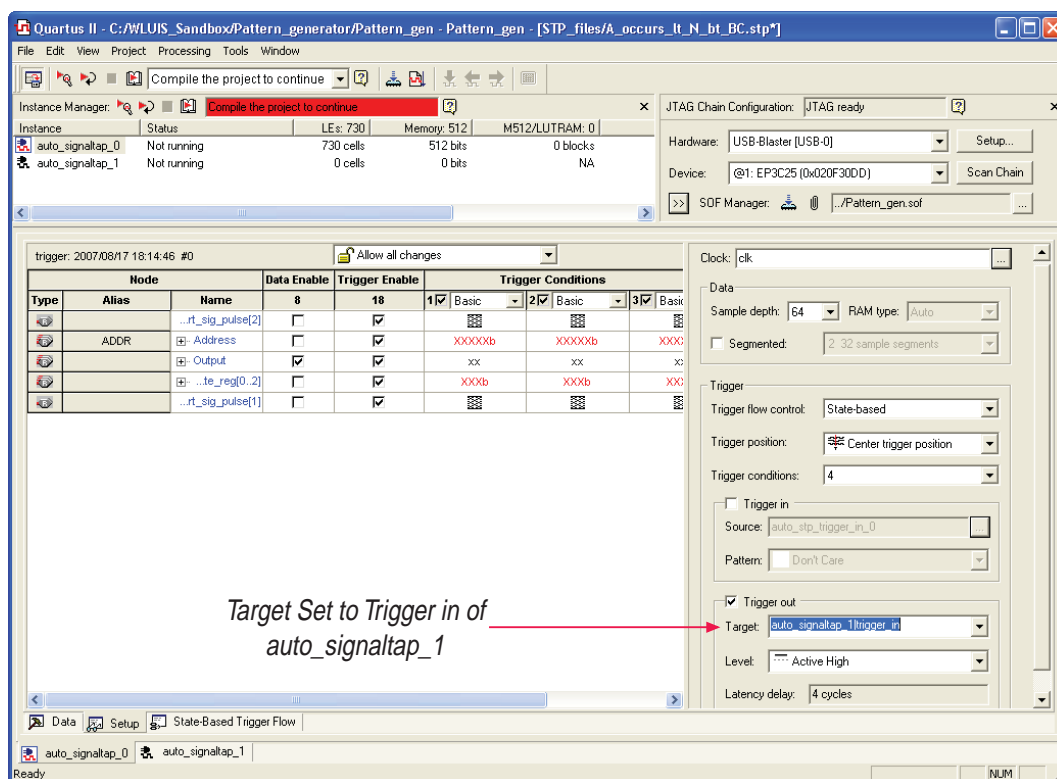
If you leave the **Target** field blank, a default name is entered in the form `auto_stp_trigger_out_<SignalTap instance number>`. When the logic analyzer triggers, a signal at the level you indicated is output on the pin you assigned to the new node.

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

An advanced feature of the SignalTap II Embedded 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.

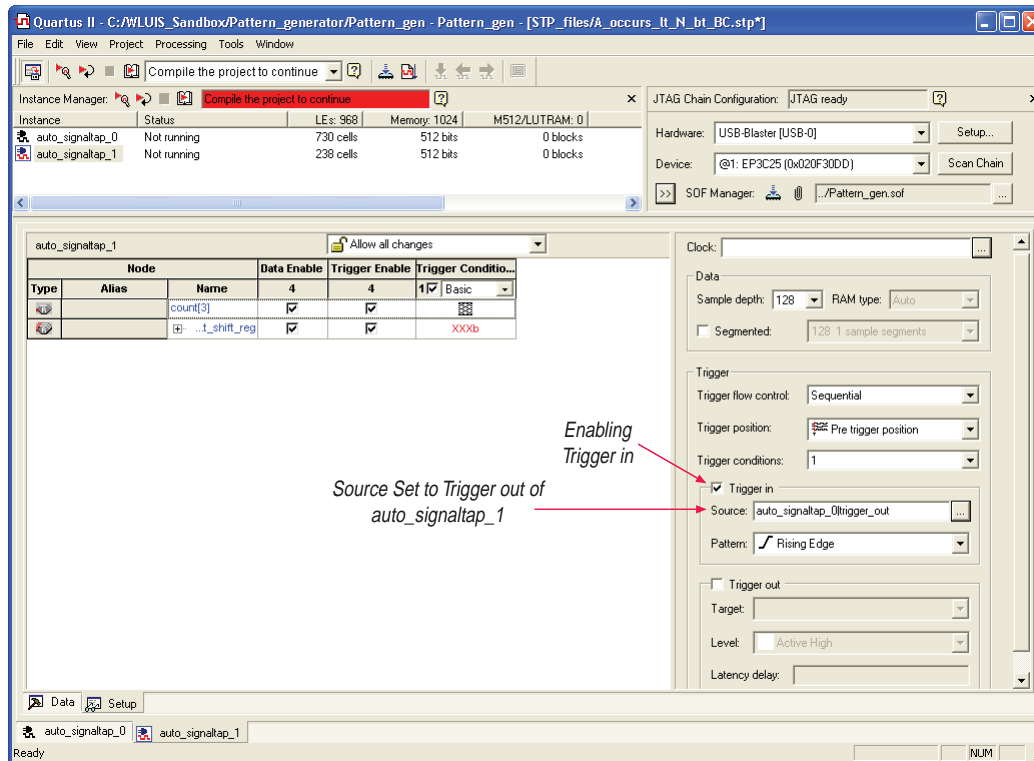
To perform this operation, first enable the **Trigger Out** of the source logic analyzer instance. On the Trigger Out **Target** list, select the targeted logic analyzer instance. For example, if the instance named `auto_signaltap_0` should trigger `auto_signaltap_1`, select `auto_signaltap_1 | trigger_in` from the list (Figure 15-40).

Figure 15-40. Configuring the Trigger Out Signal



- This automatically enables the Trigger In of the targeted logic analyzer instance and fills in the Trigger In **Source** field 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 Source field of `auto_signaltap_1` is automatically filled in with `auto_signaltap_0 | trigger_out` (Figure 15-41).

Figure 15-41. Configuring the Trigger In Signal



Compile the Design

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

Faster Compilations with Quartus II Incremental Compilation

To use incremental compilation with the SignalTap II Embedded Logic Analyzer, perform the following steps:

1. Enable **Full Incremental Compilation** for your design.
2. Assign design partitions.
3. Set partitions to the proper preservation levels.
4. Enable **SignalTap** for your design.
5. Add signals to **SignalTap** using the appropriate netlist filter in the node finder (either SignalTap II: pre-synthesis or SignalTap II: post-fitting).

When you compile your design with an **.stp** file, 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 Embedded 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 Embedded Logic Analyzer to your design without recompiling your original source code. This feature is also useful when you want to modify the configuration of the **.stp** file. 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 Embedded 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 Embedded 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

To enable incremental compilation if it is not already enabled, perform the following steps:

1. On the Assignments menu, click the **Design Partitions** window.
2. In the **Incremental Compilation** list, select **Full Incremental Compilation**.
3. Create user-defined partitions if desired and set the Netlist Type to **Post-fit** for all partitions.



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

4. On the Processing menu, click **Start Compilation**, or, on the toolbar, click **Start Compilation**.

Your project is fully compiled the first time, establishing the design partitions you have created. When enabled for your design, the SignalTap II Embedded Logic Analyzer is always a separate partition. After the first compilation, you can use the SignalTap II Embedded Logic Analyzer to analyze signals from the post-fit netlist. If your partitions are set correctly, subsequent compilations due to SignalTap II settings are able to take advantage of the shorter compilation times.



For more information about configuring and performing incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Using Incremental Compilation with the SignalTap II Embedded Logic Analyzer

The SignalTap II Embedded Logic Analyzer is automatically configured to work with the incremental compilation flow. For all signals that you want to connect to the SignalTap II Embedded 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 Embedded Logic Analyzer.



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 set 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 Embedded 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 compile flow with the SignalTap II Embedded 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.

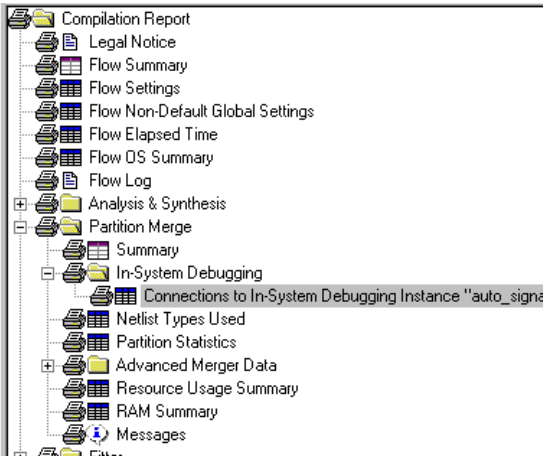


Altera recommends using only registered and user-input signals as debugging taps in your **.stp** file whenever possible.

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** file limits the changes you need to make to your SignalTap 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 compile is turned on, this report is located in the Partition Merge folder. Figure 15-42 shows an example of an In-System Debugging compilation report for a design using incremental compilation.

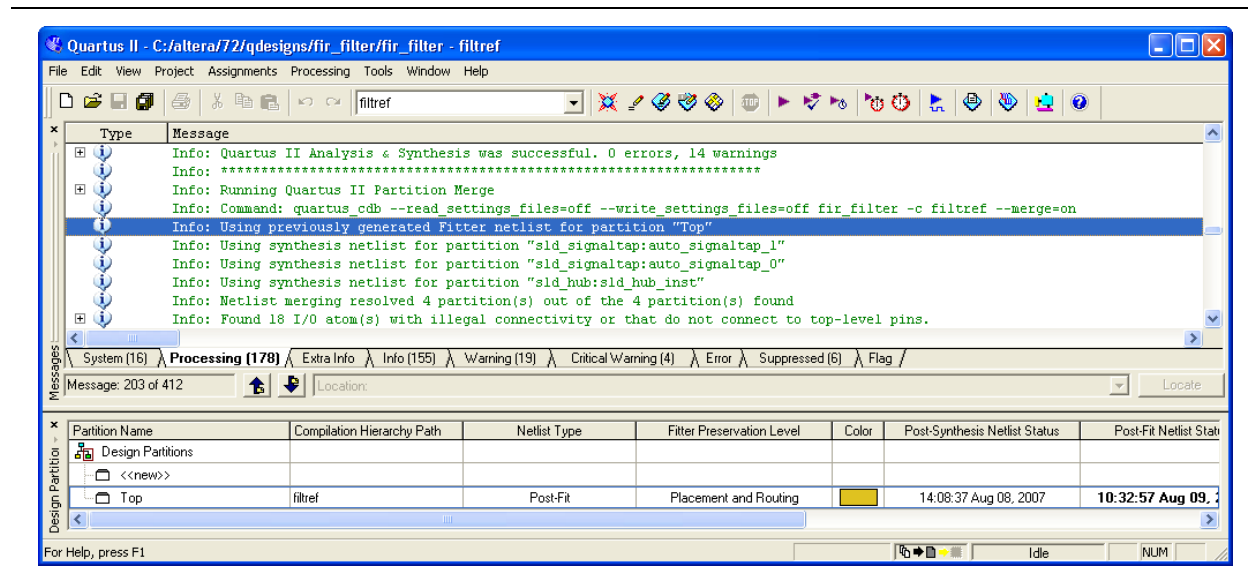
Figure 15-42. Compilation Report Showing Connectivity to SignalTap II Instance



| | Name | Type | Status | Partition Name | Netlist Type Used | Actual Connection | Details |
|----|---------------|---------------|-----------|----------------|-------------------|-------------------|---------|
| 1 | sr[62][5] | pre-synthesis | connected | Top | post-fit | sr[62][5] | N/A |
| 2 | sr[62][5] | pre-synthesis | connected | Top | post-fit | sr[62][5] | N/A |
| 3 | sr_tap_one[0] | pre-synthesis | connected | Top | post-fit | altshift_t... | N/A |
| 4 | sr_tap_one[0] | pre-synthesis | connected | Top | post-fit | altshift_t... | N/A |
| 5 | clk | post-fitting | connected | Top | post-fit | clk~input | N/A |
| 6 | sr[0][0] | post-fitting | connected | Top | post-fit | sr[0][0] | N/A |
| 7 | sr[0][0] | post-fitting | connected | Top | post-fit | sr[0][0] | N/A |
| 8 | sr[1][0] | post-fitting | connected | Top | post-fit | sr[1][0] | N/A |
| 9 | sr[1][0] | post-fitting | connected | Top | post-fit | sr[1][0] | N/A |
| 10 | sr[1][1] | post-fitting | connected | Top | post-fit | sr[1][1] | N/A |
| 11 | sr[1][1] | post-fitting | connected | Top | post-fit | sr[1][1] | N/A |
| 12 | sr[1][2] | post-fitting | connected | Top | post-fit | sr[1][2] | N/A |
| 13 | sr[1][2] | post-fitting | connected | Top | post-fit | sr[1][2] | N/A |
| 14 | sr[1][3] | post-fitting | connected | Top | post-fit | sr[1][3] | N/A |
| 15 | sr[1][3] | post-fitting | connected | Top | post-fit | sr[1][3] | N/A |

To verify that your original design was not modified, examine the messages in the **Partition Merge** section of the Compilation Report. Figure 15-43 shows an example of the messages displayed.

Figure 15-43. 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** file, only the SignalTap II design partition must be recompiled, reducing your recompilation time.

Preventing Changes Requiring Recompilation

You can configure the **.stp** file 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.



For more information about the use of lock modes, refer to the Quartus II Help.

Timing Preservation with the SignalTap II Embedded 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 Embedded 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 Embedded 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 Embedded Logic Analyzer 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** file.
- Minimize the number of combinational signals you add to your **.stp** file and add registers whenever possible.
- Specify an f_{MAX} constraint for each clock in your design.



For an example of timing preservation with the SignalTap II Embedded Logic Analyzer, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Performance and Resource Considerations

There is an inherent trade-off between runtime flexibility of the SignalTap II Embedded Logic Analyzer, timing performance of the SignalTap II Embedded Logic Analyzer, and resource usage. The SignalTap II Embedded 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 reach debugging closure as quickly as possible; however, you can adjust these settings to determine whether there is a more optimal configuration for your design.

The suggestions in this section provide some tips to 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 Embedded Logic Analyzer consumes if your design is resource-constrained.

If SignalTap II logic is part of your critical path, the following suggestions can help to speed up the performance of the SignalTap II Embedded Logic Analyzer:

- **Disable runtime configurable options**—Certain resources are allocated to accommodate for runtime flexibility. If you are using 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 of the signals that you add to the .stp file 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 to logic gate-based triggering condition, such as a basic trigger condition or a logical reduction operator in the advanced trigger tab, turn on the **Perform register retiming** option. This can help balance combinational logic across LABs.

If your design is resource constrained, the following suggestions can help to reduce the amount of logic or memory used by the SignalTap II Embedded 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 Embedded 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 Embedded 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.



For more information about area and timing optimization, refer the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Program the Target Device or Devices

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

The settings in an **.stp** file must be compatible with the programming **.sof** file used to program the device. An **.stp** file is considered compatible with an **.sof** file 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 will be 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 Editor.

To ensure programming compatibility, make sure to program your device with the latest **.sof** file created from the most recent compilation.

Before starting a debugging session, do not make any changes to the **.stp** file settings that would require the project to be recompiled. You can check the SignalTap II status display at the top of the Instance Manager to verify whether a change you made requires the design to be recompiled, producing a new **.sof** file. This gives you the opportunity to undo the change, so that a recompilation is not necessary. To prevent any such changes, enable lock mode in the **.stp** file.

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

Programming a Single Device

To configure a single device for use with the SignalTap II Embedded Logic Analyzer, perform the following steps:

1. In the **JTAG Chain Configuration** pane in the SignalTap II Editor, select the connection you use to communicate with the device from the **Hardware** list. If you need to add your communication cable to the list, click **Setup** to configure your connection.
2. In the **JTAG Chain Configuration** pane, click **Browse** and select the **.sof** file that includes the compatible SignalTap II Embedded Logic Analyzer.
3. Click **Scan Chain**. The Scan Chain operation enumerates all of the JTAG devices within your JTAG chain.
4. In the **Device** list, select the device to which you want to download the design. The device list shows an ordered list of all devices in the JTAG chain.

All of the devices are numbered sequentially according to their position in the JTAG chain, prefixed with the "@". For example: @1 : EP3C25 (0x020F30DD) lists a Cyclone III device as the first device in the chain with the JTAG ID code of 0x020F30DD.

5. Click the **Program Device** icon.

Programming Multiple Devices to Debug Multiple Designs

You can simultaneously debug multiple designs using one instance of the Quartus II software by performing the following steps:

1. Create, configure, and compile each project that includes an **.stp** file.
2. Open each **.stp** file.

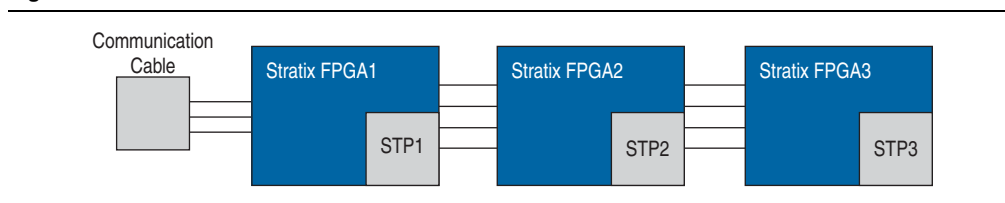


You do not have to open a Quartus II project to open an **.stp** file.

3. Use the **JTAG Chain Configuration** pane controls to select the target device in each **.stp** file.
4. Program each FPGA.
5. Run each analyzer independently.

Figure 15-44 shows a JTAG chain and its associated **.stp** files.

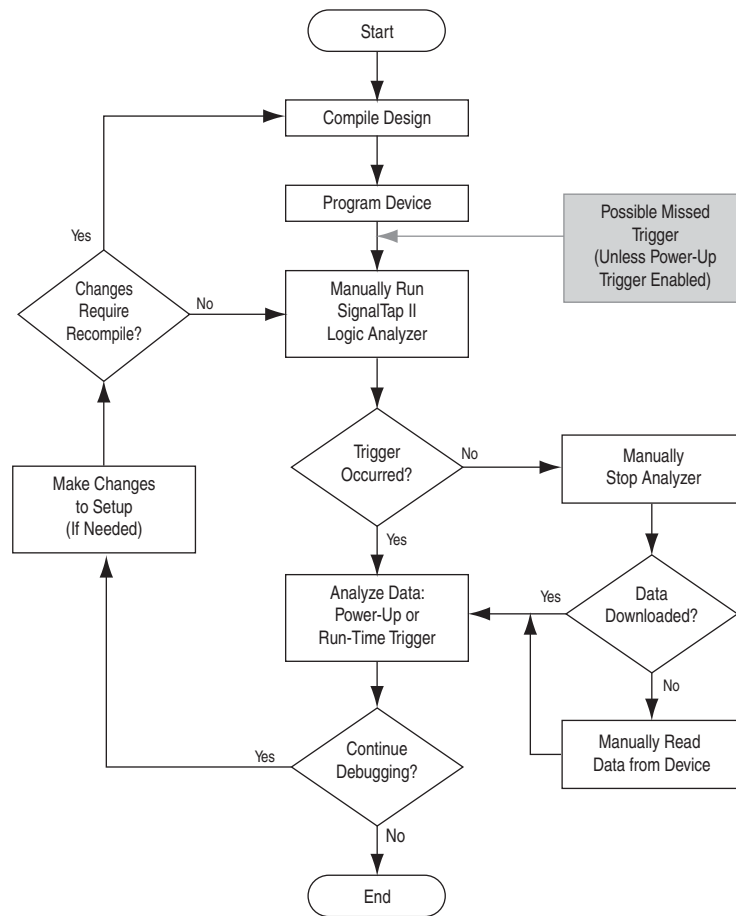
Figure 15-44. JTAG Chain



Run the SignalTap II Embedded Logic Analyzer

After the device is configured with your design that includes the SignalTap II Embedded Logic Analyzer, perform debugging operations in a manner similar to the use of an external logic analyzer. You “arm” 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** file over the JTAG connection. You can also perform the equivalent of a “force trigger” that lets you view the captured data currently in the buffer without a trigger event occurring. Figure 15-45 illustrates a flow that shows how you operate the SignalTap II Embedded 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 15-45. Power-Up and Runtime Trigger Events Flowchart



The SignalTap II toolbar in the Instance Manager has four options for running the analyzer:

- **Run Analysis**—The SignalTap II Embedded Logic Analyzer runs until the trigger event occurs. When the trigger event occurs, monitoring and data capture stops when the acquisition buffer is full.
- **AutoRun Analysis**—The SignalTap II Embedded Logic Analyzer continuously captures data until the **Stop Analysis** button is clicked, ignoring all trigger event conditions.
- **Stop Analysis**—SignalTap II analysis stops. The acquired data does not appear automatically if the trigger event has not occurred.
- **Read Data**—Captured data is displayed. This button is useful to view the acquired data, even if the trigger has not occurred.

Running with a Power-Up Trigger

If you have enabled and set up a Power-Up Trigger for an instance of the SignalTap II Embedded Logic Analyzer, the captured data may already be available for viewing if the trigger event occurred after device configuration. To download the captured data or to check if the Power-Up Trigger is still running, in the Instance Manager, click **Run Analysis**. If the Power-Up Trigger occurred, the logic analyzer immediately stops, and the captured data is downloaded from the device. The data can now be viewed on the **Data** tab of the SignalTap II Editor. If the Power-Up Trigger did not occur, no captured data is downloaded, and the logic analyzer continues to run. You can wait for the Power-Up Trigger event to occur, or, to stop the logic analyzer, click **Stop Analysis**.

Running with Runtime Triggers

You can arm and run the SignalTap II Embedded Logic Analyzer manually after device configuration to capture data samples based on the Runtime Trigger. You can do this immediately if there is no Power-Up Trigger enabled. If a Power-Up Trigger is enabled, you can do this after the Power-Up Trigger data is downloaded from the device or once the logic analyzer is stopped because the Power-Up Trigger event did not occur. Click **Run Analysis** in the SignalTap II Editor to start monitoring for the trigger event. You can start multiple SignalTap II instances at the same time by selecting all of the required instances before you click **Run Analysis** on the toolbar.

Unless the logic analyzer is stopped manually, data capture begins when the trigger event evaluates to TRUE. When this happens, the captured data is downloaded from the buffer. You can view the data in the **Data** tab of the SignalTap II Editor.

Performing a Force Trigger

Sometimes when you use an external logic analyzer or oscilloscope, you want to see the current state of signals without setting up or waiting for a trigger event to occur. This is referred to as a “force trigger” operation, because you are forcing the test equipment to capture data without regard to any set trigger conditions. With the SignalTap II Embedded Logic Analyzer, you can choose to run the analyzer and capture data immediately or run the analyzer and capture data when you want.

To run the analyzer and immediately capture data, disable the trigger conditions by turning off each **Trigger Condition** column in the node list. This operation does not require a recompilation. In the Instance Manager, click **Run Analysis**. The SignalTap II Embedded Logic Analyzer immediately triggers, captures, and downloads the data to the **Data** tab of the SignalTap II Editor. If the data does not download automatically, click **Read Data** in the Instance Manager.

If you want to choose when to capture data manually, it is not required that you disable the trigger conditions. To start the logic analyzer, click **Autorun Analysis**; to capture data, click **Stop Analysis**. If the data does not download to the **Data** tab of the SignalTap II Editor automatically, click **Read Data**.



You can also use In-System Sources and Probes in conjunction with the SignalTap II Embedded 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. For more information, refer to the *Design Debugging Using In-System Sources and Probes* chapter in volume 3 of the *Quartus II Handbook*.

Runtime Reconfigurable Options

Certain settings in the **.stp** file are changeable without incurring a recompilation when you are using Runtime Trigger mode. All Runtime Reconfigurable features are described in [Table 15-10](#).

Table 15-10. Runtime Reconfigurable Features

| Runtime Reconfigurable Setting | Description |
|---|---|
| Basic Trigger Conditions and Basic Storage Qualifier Conditions | All signals that have the trigger check box enabled 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 enabled through the Object Properties dialog box. |
| Switching between a storage-qualified and a continuous acquisition | Within any storage-qualified mode, you can switch between to continuous capture mode easily without recompiling the design. You enable this feature by selecting the check box for disable storage qualifier . |
| State-based trigger flow parameters | Refer to Table 15-5 for a list of 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. The trade-off is that there may be a slight impact to the performance and logic utilization of the SignalTap II IP core. Runtime re-configurability for Advanced Trigger Conditions and the State-based trigger flow parameters can be turned off, boosting performance and decreasing area utilization.

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

[Example 15-4](#) 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.

Example 15-4. Trigger Flow Description Providing Runtime Reconfigurable “Segments”

```

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

```

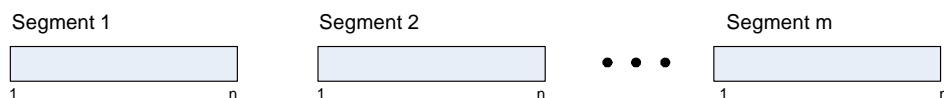
Note to Example 15-4:

(1) $m \times n$ must equal the sample depth to efficiently use the space in the sample buffer.

Figure 15-46 depicts a segmented buffer described by the trigger flow in Example 15-4.

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 15-46. Segmented Buffer Created with Storage Qualifier and State-Based Trigger (Note 1)

**Note to Figure 15-46:**

(1) Total sample depth is fixed, where $m \times n$ 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.

Example 15-5 shows a modified description of Example 15-4 with an additional state inserted. This extra state is used to specify a different trigger condition that does not use the storage qualifier feature. Status flags are inserted into the conditional statements to control the execution of the trigger flow.

Example 15-5. Modified Trigger Flow Description of Example 15-4 with Status Flags to Selectively Enable States

```
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 15-11 describes the text messages that might appear in the SignalTap II Status Indicator in the Instance Manager before, during, and after a data acquisition. Use these messages to know the state of the logic analyzer or what operation it is performing.

Table 15-11. 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 Embedded 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 <i>x</i> has occurred. The SignalTap II Embedded Logic Analyzer is waiting for the condition specified in condition <i>x + 1</i> to occur. |
| Acquiring (power-up) post-trigger data (1) | The entire trigger event has occurred. The SignalTap II Embedded Logic Analyzer is acquiring the post-trigger data. The amount of post-trigger data collected is user-defined 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 Embedded Logic Analyzer is waiting for the user to arm the analyzer. |

Note to Table 15-11:

- (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.



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

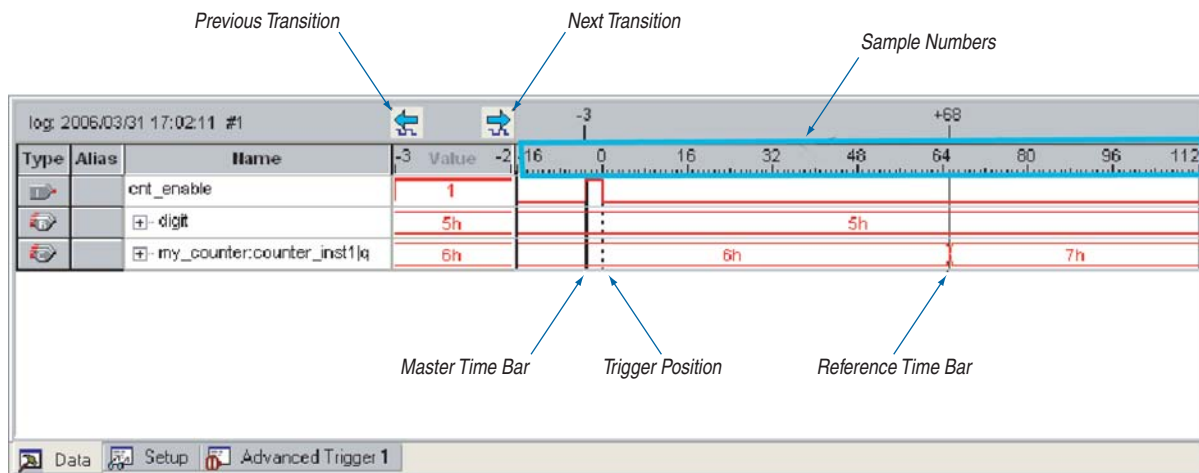
View, Analyze, and Use Captured Data

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.

Viewing Captured Data

You can view captured SignalTap II data in the **Data** tab of the **.stp** file (Figure 15-47). Each row of the **Data** tab displays the captured data for one signal or bus. Buses can be expanded to show the data for each individual signal on the bus. Click on the data waveforms to zoom in on the captured data samples; right-click to zoom out.

Figure 15-47. Captured SignalTap II Data



When viewing captured data, it is often useful to know the time interval between two events. Time bars enable you to see the number of clock cycles between two samples of captured data in your system. There are two types of time bars:

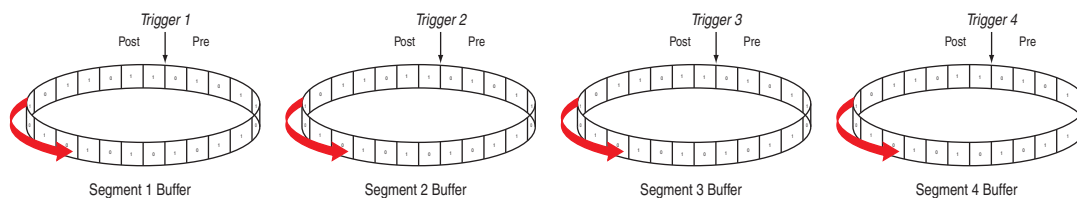
- **Master Time Bar**—The master time bar's label displays the absolute time of its location in bold. The master time bar is a thick black line in the **Data** tab. The captured data has only one master time bar.
- **Reference Time Bar**—The reference time bar's label displays the time relative to the master time bar. You can create an unlimited number of reference time bars.

To help you find a transition of signals relative to the master time bar location, use either the **Next Transition** or the **Previous Transition** button. This aligns the master time bar with the next or previous transition of a selected signal or group of selected signals. This feature is very useful when the sample depth is very large and the rate at which signals toggle is very low.

To find bus values within the waveform quickly, use the **Find bus value** option. After you select the bus, right-click and select **Find bus value**. A dialog box appears to enter search parameters.

Capturing Data Using Segmented Buffers

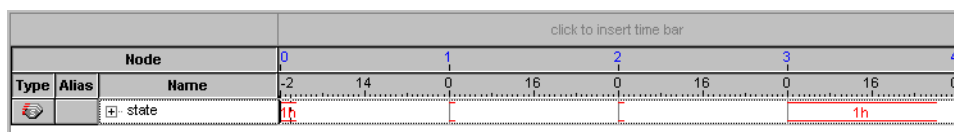
Segmented Acquisition buffers allow you to perform multiple captures with a separate trigger condition for each acquisition segment. This feature allows you to capture a recurring event or sequence of events that span over a long period time efficiently. Each acquisition segment acts as a non-segmented buffer, continuously capturing data when it is activated. When you run an analysis with the **segmented buffer** option enabled, the SignalTap II Embedded 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 15-48 shows a segmented acquisition buffer with four segments represented as four separate non-segmented buffers.

Figure 15-48. Segmented Acquisition Buffer

The SignalTap II Embedded Logic Analyzer finishes an acquisition with a segment, and advances to the next segment to start a new acquisition. Depending when on a trigger condition occurs, it may affect the way the data capture appears in the waveform viewer. Figure 15-48 illustrates the method data is captured. The Trigger markers in Figure 15-48—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 are using 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 Embedded 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 is necessary so the SignalTap II Embedded Logic Analyzer can capture accurately all of the trigger conditions that have occurred. Samples that have not been used appear as a blank space in the waveform viewer.

Figure 15-49 shows an example of a capture using sequential flow control with the trigger condition for each segment set to “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 set to 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 is the space left over from the pre-trigger samples that the SignalTap II Embedded Logic Analyzer allocated to the buffer.

Figure 15-49. Segmented Capture with Preemption of Acquisition Segments (Note 1)**Note to Figure 15-49:**

- (1) A segmented acquisition buffer using the sequential trigger flow with a trigger condition set to “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 that is specific to your debugging scenario, you can help maximize the use of the allocated buffer space.

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** file 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 it 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.

The labels you create in a table are used 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 any **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** file, mnemonic tables for the added signals are automatically created and assigned to the signals defined in the plug-in. If you ever need to enable these mnemonic tables manually, right-click on the name of the signal or signal group. On the **Bus Display Format** submenu, click the name of the mnemonic table that matches the plug-in.

As an example, the Nios II plug-in makes it easy to monitor your design's signal activity as code is executed. If you have set up the logic analyzer to trigger on a function name in your Nios II code based on data from an **.elf** file, 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 15-50](#). Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

Figure 15-50. Data Tab when the Nios II Plug-In is Used

| Type | Alias | Name | 37 | Value | 38 | 48 | 49 | 50 | 51 | 52 |
|------|-------------------------|--------------|---------|---------|---------|---------|--------------|---------|---------|---------|
| PC | ...Nios II Inst Address | alt_main+0x8 | <empty> | <empty> | <empty> | <empty> | alt_main+0xc | <empty> | <empty> | <empty> |
| DIS | ...Nios II Disassembly | mov fp, sp | <empty> | <empty> | <empty> | <empty> | movi r2, 2 | <empty> | <empty> | <empty> |

Locating a Node in the Design

When you find the source of an error in your design using the SignalTap II Embedded 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 Embedded Logic Analyzer in one of the Quartus II software tools or your design files, right-click on the signal in the **.stp** file, and click **Locate in <tool name>**.

You can locate a signal from the node list in any of the following locations:

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



For more information about using these tools, refer to each of the corresponding chapters in the *Quartus II Handbook*.

Saving Captured Data

The data log shows the history of captured data and the triggers used to capture the data. The 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 go back and 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. If the **Data Log** pane is closed, on the View menu, select **Data Log** to reopen it. To enable data logging, turn on **Enable data log** in the **Data Log** (Figure 15-25). To recall a data log for a given trigger set and make it active, double-click the name of the data log in the list.

The Data Log feature is useful for organizing different sets of trigger conditions and different sets of signal configurations. For more information, refer to “[Managing Multiple SignalTap II Files and Configurations](#)” on page 15-32.

Converting Captured Data to Other File Formats

You can export captured data in the following file formats, some of which can be used 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 SignalTap II Embedded Logic Analyzer’s captured data, 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, on the File menu, select **Create/Update** and click **Create SignalTap II List File**.

Other Features

The SignalTap II Embedded 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 Embedded Logic Analyzer directly into a matrix in the MATLAB environment. If you use the MEX function repeatedly in a loop, you can perform as many acquisitions as you can when using SignalTap II in the Quartus II software environment in the same amount of time.



The SignalTap II MATLAB MEX function is available only in the Windows 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 (refer to [“Creating and Enabling a SignalTap II File”](#) on page 15-7).
2. In the node list in the **Data** tab of the SignalTap II 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.



Signal groups acquired from the SignalTap II Embedded Logic Analyzer and transferred into the MATLAB environment with the MEX function are limited to a width of 32 signals. If you want to use the 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** file and compile your design. Program your device and run the SignalTap II Embedded Logic Analyzer to ensure your trigger conditions and signal acquisition are working 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_signalsnap_run ↵
```

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

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

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

When capturing data, `<stp filename>` is the name of the `.stp` file you want to use. This is required for using the MEX function. The other MEX function options are defined in [Table 15-12](#).

Table 15-12. 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_signalsnap_0' | Specify a SignalTap II instance if more than one instance is defined. The default is the first instance in the <code>.stp</code> file, <code>auto_signalsnap_0</code> . |
| <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 <code>.stp</code> file. 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_signalsnap_run('VERBOSE_ON'); ↵
alt_signalsnap_run('VERBOSE_OFF'); ↵
```

When you finish acquiring data, close the JTAG connection. Use the following command to close the connection:

```
alt_signalsnap_run('END_CONNECTION'); ↵
```



For more information about the use of 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 Embedded 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 stand-alone version of the SignalTap II Embedded Logic Analyzer is included with the Quartus II stand-alone Programmer and is available from the Downloads page of the Altera website (www.altera.com).

Remote Debugging Using the SignalTap II Embedded Logic Analyzer

You can use the SignalTap II Embedded 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 Embedded 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

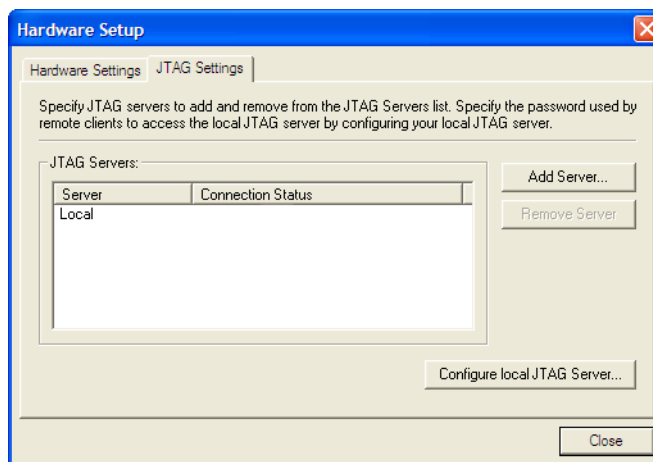
On the PC in the remote location, install the stand-alone version of the SignalTap II Embedded Logic Analyzer or the full version of the Quartus II software. This remote computer must have Altera programming hardware connected, such as the EthernetBlaster or USB-Blaster.

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.

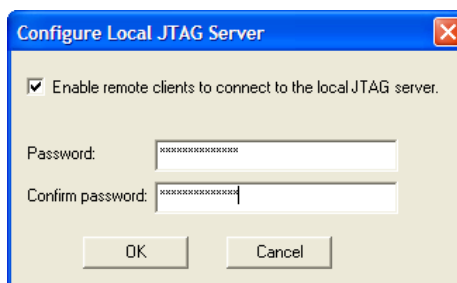
Software Setup on the Remote PC

To set up the software on the remote PC, perform the following steps:

1. In the Quartus II programmer, click **Hardware Setup**.
2. Click the **JTAG Settings** tab ([Figure 15-51](#)).

Figure 15-51. Configure JTAG on Remote PC

3. Click **Configure local JTAG Server**.
4. In the **Configure Local JTAG Server** dialog box (Figure 15-52), turn on **Enable remote clients to connect to the local JTAG server** and in the password box, type your password. In the **Confirm Password** box, type your password again and click **OK**.

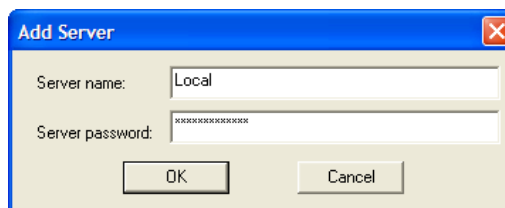
Figure 15-52. Configure Local JTAG Server on Remote

Software Setup on the Local PC

To set up your software on your local PC, perform the following steps:

1. Launch the Quartus II programmer.
2. Click **Hardware Setup**.
3. On the **JTAG** settings tab, click **Add server**.
4. In the **Add Server** dialog box (Figure 15-53), type the network name or IP address of the server you want to use and the password for the JTAG server that you created on the remote PC.

Figure 15-53. Add Server Dialog Box



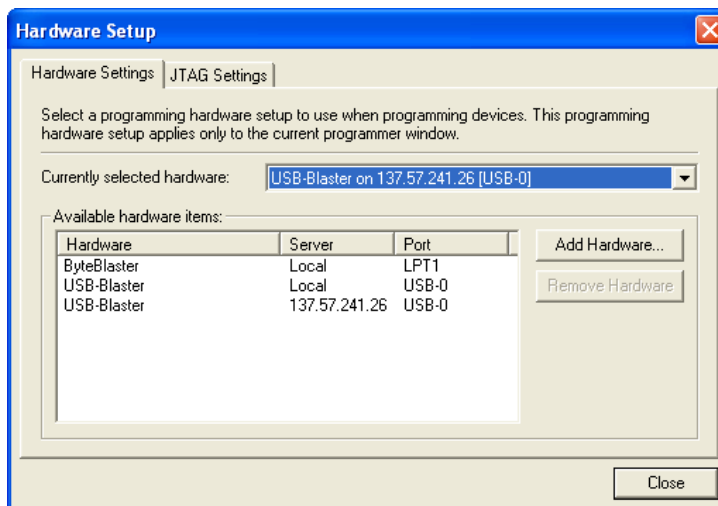
5. Click **OK**.

SignalTap II Setup on the Local PC

To connect to the hardware on the remote PC, perform the following steps:

1. Click the **Hardware Settings** tab and select the hardware on the remote PC (Figure 15-54).

Figure 15-54. Selecting Hardware on Remote PC



2. Click **Close**.

You can now control the logic analyzer on the device attached to the remote PC as if it was connected directly to the local PC.

Using the SignalTap II Embedded 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 Embedded 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 Embedded 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 Embedded Logic Analyzer. After the FPGA is configured with a SignalTap II Embedded Logic Analyzer instance in the design, when you open the SignalTap II Embedded Logic Analyzer window/GUI in the Quartus II software, you then scan the chain and it will be ready to acquire data over JTAG.

Backward Compatibility with Previous Versions of Quartus II Software

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

If you have a Quartus project file from a previous version of the software, you may have to update the STP configuration file if you wish to recompile the project. You can update the configuration file by invoking the SignalTap II GUI. If any updates to your configuration are necessary, a prompt will appear asking if you would like to update the .stp file to match the current version of the Quartus II software.

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 command at the command prompt:

```
quartus_sh --qhelp ↵
```



The *Quartus II Scripting Reference Manual* includes the same information in PDF format.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.



For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

SignalTap II Command-Line Options

To compile your design with the SignalTap II Embedded Logic Analyzer using the command prompt, use the `quartus_stp` command. [Table 15-13](#) shows the options that help you better understand how to use the `quartus_stp` executable.

Table 15-13. SignalTap II Command-Line Options

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

Example 15-6 illustrates how to compile a design with the SignalTap II Embedded Logic Analyzer at the command line.

Example 15-6.

```
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_tan filtref ↵
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 Embedded Logic Analyzer to compile properly into your design.

Example 15-7 shows how to create a new **.stp** file after building the SignalTap II Embedded Logic Analyzer instance with the MegaWizard Plug-In Manager.

Example 15-7.

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



For information about the other command line executables and options, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

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 GUI. They must be run from the command line with the **quartus_stp** executable. To run a Tcl file that has SignalTap II Tcl commands, use the following command:

```
quartus_stp -t <Tcl file> ↵
```

Table 15-14 shows the Tcl commands that you can use with SignalTap II Embedded Logic Analyzer.

Table 15-14. SignalTap II Tcl Commands

| Command | Argument | Description |
|--------------------|---|---|
| open_session | -name <stp_filename> | Opens the specified .stp file. All captured data is stored in this file. |
| run | -instance <instance_name> -signal_set <signal_set> (optional) -trigger <trigger_name> (optional) -data_log <data_log_name> (optional) -timeout <seconds> (optional) | Starts the analyzer. This command must be followed by all the required arguments to properly start the analyzer. You can optionally specify the name of the data log you want to create. If the Trigger condition is not met, you can specify a timeout value to stop the analyzer. |
| run_multiple_start | None | Defines the start of a set of run commands. Use this command when multiple instances of data acquisition are started simultaneously. Add this command before the set of run commands that specify data acquisition. You must use this command with the run_multiple_end command. If the run_multiple_end command is not included, the run commands do not execute. |
| run_multiple_end | None | Defines the end of a set of run commands. Use this command when multiple instances of data acquisition are started simultaneously. Add this command after the set of run commands. |
| stop | None | Stops data acquisition. |
| close_session | None | Closes the currently open .stp file. You cannot run the analyzer after the .stp file is closed. |



For more information about SignalTap II Tcl commands, refer to the Quartus II Help.

Example 15-8 is an excerpt from a script that is used to continuously capture data. Once the trigger condition is met, the data is captured and stored in the data log.

Example 15-8.

```
#opens signaltap session
open_session -name stpl.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** file that you used to capture data to examine the contents of the Data Log.

Design Example: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems

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 a button push. After a button is pushed, the processor initiates a DMA transfer, which you analyze using the SignalTap II Embedded Logic Analyzer.



For more information about this example and using the SignalTap II Embedded Logic Analyzer with SOPC builder systems, refer to [AN 323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems](#) and [AN 446: Debugging Nios II Systems with the SignalTap II Logic Analyzer](#).

Custom Triggering Flow Application Examples

The custom triggering flow in the SignalTap II Embedded 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 Embedded 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**.



For additional triggering flow design examples, refer to the [Quartus II On-Chip Debugging Design Examples](#) page for on-chip debugging.

Design Example 1: Specifying a Custom Trigger Position

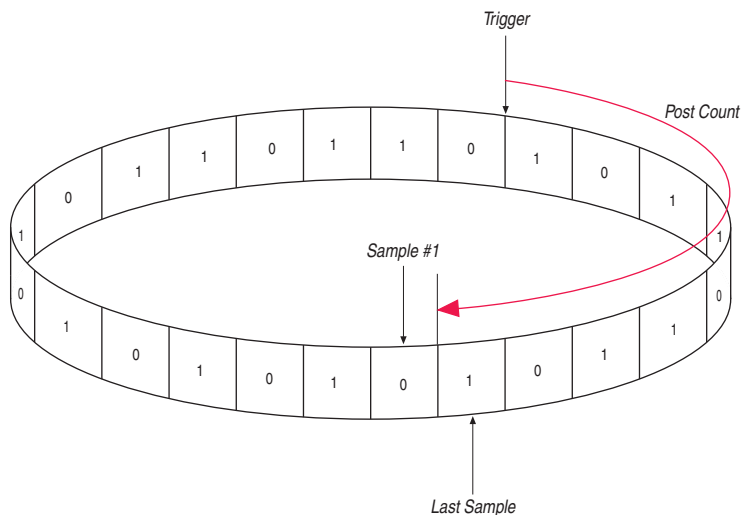
Actions to the acquisition buffer can accept an optional post-count argument. This post-count argument enables you to define a custom triggering position for each segment in the acquisition buffer. [Example 15-9](#) shows an example that applies a trigger position to all segments in the acquisition buffer. 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.

Example 15-9.

```
if (c1 == 3 && condition1)
    trigger 30;
else if ( condition1 )
    begin
        segment_trigger 30;
        increment c1;
    end
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 $N - \text{post count fill}$, where N is the number of samples per segment. [Figure 15-55](#) illustrates the triggering position.

Figure 15-55. 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. [Example 15-10](#) 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.

Example 15-10.

```
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;
```

Conclusion

As the FPGA industry continues to make technological advancements, outdated methodologies need to be replaced with new technologies that maximize productivity. The SignalTap II Embedded Logic Analyzer gives you the same benefits as a traditional logic analyzer, without the many shortcomings of a piece of dedicated test equipment. This version of the SignalTap II Embedded Logic Analyzer provides many new and innovative features that allow you to capture and analyze internal signals in your FPGA, allowing you to find the source of a design flaw in the shortest amount of time.

Referenced Documents

This chapter references the following documents:

- *AN 323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder System*
- *AN 446: Debugging Nios II Systems with the SignalTap II Embedded Logic Analyzer*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Design Debugging Using In-System Sources and Probes* chapter in volume 3 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*

- [Quartus II Handbook](#)
- [Quartus II Incremental Compilation for Hierarchical and Team-Based Design](#) chapter in volume 1 of the [Quartus II Handbook](#)
- [Quartus II Integrated Synthesis](#) chapter in volume 1 of the [Quartus II Handbook](#)
- [Quartus II Scripting Reference Manual](#)
- [Quartus II Settings File Reference Manual](#)
- [Quick Design Debugging Using SignalProbe](#) chapter in volume 3 of the [Quartus II Handbook](#)
- [Tcl Scripting](#) chapter in volume 2 of the [Quartus II Handbook](#)

Document Revision History

Table 15–15 shows the revision history for this chapter.

Table 15–15. Document Revision History

| Date and Document Version | Changes Made | Summary of Changes |
|---------------------------|--|--|
| November 2009 v9.1.0 | No change to content. | — |
| March 2009 v9.0.0 | <ul style="list-style-type: none"> ■ Updated Table 15–1 ■ Updated “Using Incremental Compilation with the SignalTap II Embedded Logic Analyzer” on page 15–55 ■ Added new Figure 15–42 ■ Made minor editorial updates | Updated for the Quartus II software version 9.0 release. |
| November 2008 v8.1.0 | Updated for the Quartus II software version 8.1 release: <ul style="list-style-type: none"> ■ Added new section “Using the Storage Qualifier Feature” on page 14–25 ■ Added description of <code>start_store</code> and <code>stop_store</code> commands in section “Trigger Condition Flow Control” on page 14–36 ■ Added new section “Runtime Reconfigurable Options” on page 14–63 | Updated for the Quartus II software version 8.1 release. |
| May 2008 v8.0.0 | Updated for the Quartus II software version 8.0: <ul style="list-style-type: none"> ■ 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 | Updated for the Quartus II software version 8.0 release. |



For previous versions of the [Quartus II Handbook](#), refer to the [Quartus II Handbook Archive](#).