### Introduction

Debugging today's FPGA designs can be a daunting task. As your product requirements continue to increase in complexity, the time you spend on design verification continues to rise. To get your product to market as quickly as possible, you must minimize design verification time. To help alleviate the time-to-market pressure, a set of verification tools that are powerful and easy to use are required.

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

This section provides a quick overview on the tools available in the on-chip debugging suite and discusses the criteria for selecting the best tool for your design.

### On-Chip Debugging Ecosystem

Table IV–1 summarizes the tools in the In-System verification tool suite that are covered in this section of the Quartus II Handbook.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
<th>Typical Circumstances of Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SignalTap® II Logic Analyzer</strong></td>
<td>This embedded logic analyzer uses FPGA resources to sample test nodes and outputs the information to the Quartus II software for display and analysis.</td>
<td>You have spare on-chip memory and you want functional verification of your design running in hardware.</td>
</tr>
<tr>
<td><strong>SignalProbe</strong></td>
<td>This tool incrementally routes internal signals to I/O pins while preserving results from your last place-and-routed design.</td>
<td>You have spare I/O pins and you would like to check the operation of a small set of control pins using either an external logic analyzer or an oscilloscope.</td>
</tr>
<tr>
<td><strong>Logic Analyzer Interface (LAI)</strong></td>
<td>This tool multiplexes a larger set of signals to a smaller number of spare I/O pins. LAI allows you to select which signals are switched onto the I/O pins over a JTAG connection.</td>
<td>You have limited on-chip memory, and have a large set of internal data busses that you would like to verify using an external logic analyzer. Logic analyzer vendors, such as Tektronics and Agilent, provide integration with the tool to improve the usability.</td>
</tr>
<tr>
<td><strong>In-System Memory Content Editor</strong></td>
<td>This tool displays and allows you to edit on-chip memory.</td>
<td>You would like to view and edit the contents of the either the instruction cache or data cache of a Nios® II processor application.</td>
</tr>
</tbody>
</table>
With the exception of SignalProbe, each of the on-chip debugging tools uses the JTAG port to control and read back data from debugging logic and signals under test. The JTAG resource is shared among all of the on-chip debugging tools. The Quartus II software compiles logic into your design automatically to distinguish between data and control information and each of the debugging logic blocks when the JTAG resource is required. This arbitration logic, also known as the System-Level Debugging (SLD) infrastructure, is shown in the design hierarchy of your compiled project as `sld_hub:sld_hub_inst`. The SLD logic allows you to instantiate multiple debugging blocks into your design and run them simultaneously.

To maximize debugging closure, the Quartus II software allows you to use a combination of the debugging tools in tandem to fully exercise and analyze the logic under test. All of the tools described in Table IV–1 have basic analysis features built in; that is, all of the tools enable you to read back information collected from the design nodes that are connected to the debugging logic. Out of the set of debugging tools, the SignalTap II Logic Analyzer, the LAI, and the SignalProbe feature are general-purpose debugging tools optimized for probing signals in your RTL netlist. In-System Sources and Probes, the Virtual JTAG Interface, and In-System Memory content editor, in addition to being able to read back data from the debugging points, allow you to input values into your design during runtime. Taken together, the set of on-chip debugging tools form a debugging ecosystem. The set of tools can generate a stimulus to and solicit a response from the logic under test, providing a complete debugging solution (Figure IV–1).

### Table IV–1. Available Tools in the In-System Verification Tools Suite (Part 2 of 2)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
<th>Typical Circumstances of Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-System Sources and Probes</td>
<td>This feature provides an easy way to drive and sample logic values to and from internal nodes using the JTAG interface.</td>
<td>You want to prototype a front panel with virtual buttons for your FPGA design.</td>
</tr>
<tr>
<td>Virtual JTAG Interface</td>
<td>This megafuntion opens up the JTAG interface so that you can develop your own custom applications.</td>
<td>You want to generate a large set of test vectors and send them to your device over the JTAG port to functionally verify your design running in hardware.</td>
</tr>
</tbody>
</table>
The tools in the toolchain offer different advantages and different trade-offs. To understand the selection criteria between the different tools, the following sections analyze the tools according to their typical applications.

The first section, “Analysis Tools for RTL Nodes”, compares the SignalTap II Logic Analyzer, SignalProbe, and the LAI. These three tools are logically grouped since they are intended for debugging nodes from your RTL netlist at system speed.

The next section, “Stimulus-Capable Tools” on page 14–8, compares the In-System Memory Content Editor, Virtual JTAG Interface Megafunction, and In-System Sources and Probes. These tools are logically grouped since they offer the ability to both read and write transactions through the JTAG port.

### Analysis Tools for RTL Nodes

The SignalTap II Embedded Logic Analyzer, the SignalProbe feature, and the LAI are designed specifically for probing and debugging RTL signals at system speed. They are general-purpose analysis tools that enable you to tap and analyze any routable node from the FPGA or CPLD. These three tools satisfy a range of requirements. If you have spare logic and memory resources, the SignalTap II Logic Analyzer is useful for providing fast functional verification of your design running on actual hardware.

On the other hand, if logic and memory resources are tight and you require the large sample depths associated with external logic analyzers, both the LAI and the SignalProbe feature make it easy to view internal design signals using external equipment.
The most important selection criteria for these three tools are the available resources remaining on your device after implementing your design and the number of spare pins available. It is worthwhile to evaluate your preferred debugging option early on in the design planning process to ensure that your board, your Quartus II project, and your design are all set up to support the appropriate options. Planning early can reduce time spent during debugging and eliminate the necessary late changes to accommodate your preferred debugging methodologies. The following two sections provide information to assist you in choosing the appropriate tool by comparing the tools according to their resource usage and their pin consumption.

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

**Resource Usage**

Any debugging tool that requires the use of a JTAG connection requires the SLD infrastructure logic mentioned earlier, for communication with the JTAG interface and arbitration between any instantiated debugging modules. This overhead logic uses around 200 LEs, a small fraction of the resources available in any of the supported devices. The overhead logic is shared between all available debugging modules in your design. Both the SignalTap II Logic Analyzer and the LAI use a JTAG connection.

SignalProbe requires very few on-chip resources. Because it requires no JTAG connection, SignalProbe uses no logic or memory resources—it uses only routing resources to route an internal signal to a debugging test point.

The LAI requires a small amount of logic to implement the multiplexing function between the signals under test, in addition to the SLD infrastructure logic. Because no data samples are stored on the chip, the LAI uses no memory resources.

The SignalTap II Logic Analyzer requires both logic and memory resources. The number of logic resources used depends on the number of signals tapped and the complexity of the trigger logic. However, the amount of Logic Resources that the SignalTap II Logic Analyzer uses is typically a small percentage of most designs. A baseline configuration consisting of the SLD arbitration logic and a single node with basic triggering logic contains approximately 300–400 logic elements (LEs). Each additional node you add to the baseline configuration adds about 11 LEs. Compared with logic resources, memory resources are a more important factor to consider for your design. Memory usage can be significant and depends on how you configure your SignalTap II Logic Analyzer instance to capture data and the sample depth that your design requires for debugging. For the SignalTap II Logic Analyzer, there is the added benefit of requiring no external equipment, as all of the triggering logic and storage is on the chip.

Figure IV–2 shows a conceptual graph of the resource usage of the three analysis tools relative to each other.
The resource estimation feature for the SignalTap II Logic Analyzer and the LAI allows you to quickly judge if enough on-chip resources are available before compiling the tool with your design. Figure IV–3 shows the resource estimation feature for the SignalTap II Logic Analyzer and the Logic Analyzer Interface.

**Figure IV–2. Resource Usage per Debugging Tool (Note 1)**

![Resource Usage per Debugging Tool](image)

**Note to Figure IV–2:**
(1) Though resource usage is highly dependent on the design, this graph provides a rough guideline for tool selection.

**Figure IV–3. Resource Estimator**

<table>
<thead>
<tr>
<th>Instance</th>
<th>Status</th>
<th>LEs: 652</th>
<th>Memory: 52/288</th>
<th>M512/M512E: 0/512</th>
<th>M4K/M4K: 128/60</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignalTap II</td>
<td>Not running</td>
<td>652 cells</td>
<td>52/288 bits</td>
<td>0 blocks</td>
<td>Can't Fit 128 blocks</td>
</tr>
</tbody>
</table>

**Pin Usage**

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

The ratio of the number of pins used to the number of signals tapped for the LAI is many-to-one. It can map up to 256 signals to each debugging pin, depending on available routing resources. The control of the active signals that are mapped to the spare I/O pins is performed via the JTAG port. The LAI is ideal for routing data buses to a set of test pins for analysis.

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

The SignalTap II Embedded Logic Analyzer, the SignalProbe feature, and the LAI tools can be added to your existing design with minimal effects. With the node finder, you can find signals to route to a debugging module without making any changes to your HDL. SignalProbe inserts signals directly from your post-fit database. The SignalTap II Logic Analyzer and LAI support inserting signals from both pre-synthesis and post-fit netlists. All three tools allow you to find and configure your debugging setup quickly. In addition, the Quartus II incremental compilation feature and the Quartus II incremental routing feature allow for a fast turnaround time for your programming file, increasing productivity and enabling fast debugging closure.

Both LAI and the SignalTap II Logic Analyzer support incremental compilation. With incremental compilation, you can add a SignalTap II Logic Analyzer instance or an LAI instance incrementally into your placed-and-routed design. This has the benefit of both preserving your timing and area optimizations from your existing design, and decreasing the overall compilation time when any changes are necessary during the debugging process. With incremental compilation, you can save up to 70% compile time of a full compilation.

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

As another productivity enhancement, all tools in the on-chip debugging tool set support scripting via the `quartus_stp` Tcl package. For the SignalTap II Logic Analyzer and the LAI, scripting enables user-defined automation for data collection while debugging in the lab.

In addition, the JTAG server allows you to debug a design that is running on a device attached to a PC in a remote location. This allows you to set up your hardware in the lab environment, download any new `.sof` files, and perform any analysis from your desktop.

Table IV–2 compares common debugging features between these tools and provides suggestions about which is the best tool to use for a given feature.

### Table IV–2. Suggested On-Chip Debugging Tools for Common Debugging Features (Part 1 of 2) (Note 1)

<table>
<thead>
<tr>
<th>Feature</th>
<th>SignalProbe</th>
<th>Logic Analyzer Interface (LAI)</th>
<th>SignalTap II Logic Analyzer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Sample Depth</td>
<td>N/A</td>
<td>✓</td>
<td>—</td>
<td>An external logic analyzer used with the LAI has a bigger buffer to store more captured data than the SignalTap II Logic Analyzer. No data is captured or stored with SignalProbe.</td>
</tr>
<tr>
<td>Ease in Debugging Timing Issue</td>
<td>✓</td>
<td>✓</td>
<td>—</td>
<td>External equipment, such as oscilloscopes and Mixed Signal Oscilloscopes (MSOs), can be used with either LAI or SignalProbe used with the LAI to provide you with access to timing mode, enabling you to debug combined streams of data.</td>
</tr>
<tr>
<td>Feature</td>
<td>SignalProbe</td>
<td>Logic Analyzer Interface (LAI)</td>
<td>SignalTap II Logic Analyzer</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-------------</td>
<td>--------------------------------</td>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Minimal Effect on Logic Design</td>
<td>✓</td>
<td>✓ (2)</td>
<td>✓ (2)</td>
<td>The LAI adds minimal logic to a design, requiring fewer device resources. The SignalTap II Logic Analyzer has little effect on the design, as it is set as a separate design partition. SignalProbe incrementally routes nodes to pins, not affecting the design at all.</td>
</tr>
<tr>
<td>Short Compile and Recompile Time</td>
<td>✓</td>
<td>✓ (2)</td>
<td>✓ (2)</td>
<td>SignalProbe attaches incrementally routed signals to previously reserved pins, requiring very little recompilation time to make changes to source signal selections. The SignalTap II Logic Analyzer and the LAI can take advantage of incremental compilation to refit their own design partitions to decrease recompilation time.</td>
</tr>
<tr>
<td>Triggering Capability</td>
<td>N/A</td>
<td>N/A</td>
<td>✓</td>
<td>The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to commercial logic analyzers.</td>
</tr>
<tr>
<td>I/O Usage</td>
<td>—</td>
<td>—</td>
<td>✓</td>
<td>No additional output pins are required with the SignalTap II Logic Analyzer. Both the LAI and SignalProbe require I/O pin assignments.</td>
</tr>
<tr>
<td>Acquisition Speed</td>
<td>N/A</td>
<td>—</td>
<td>✓</td>
<td>The SignalTap II Logic Analyzer can acquire data at speeds of over 200 MHz. The same acquisition speeds are obtainable with an external logic analyzer used with the LAI, but signal integrity issues may limit this.</td>
</tr>
<tr>
<td>No JTAG Connection Required</td>
<td>✓</td>
<td>—</td>
<td>—</td>
<td>An FPGA design with the SignalTap II Logic Analyzer or the LAI requires an active JTAG connection to a host running the Quartus II software. SignalProbe does not require a host for debugging purposes.</td>
</tr>
<tr>
<td>No External Equipment Required</td>
<td>—</td>
<td>—</td>
<td>✓</td>
<td>The SignalTap II Logic Analyzer logic is completely internal to the programmed FPGA device. No extra equipment is required other than a JTAG connection from a host running the Quartus II software or the stand-alone SignalTap II software. SignalProbe and the LAI require the use of external debugging equipment, such as multimeters, oscilloscopes, or logic analyzers.</td>
</tr>
</tbody>
</table>

Notes to Table IV–2:

(1) ✓ indicates the recommended tools for the feature.
— indicates that while the tool is available for that feature, that tool may not give the best results.
N/A indicates that the feature is not applicable for the selected tool.

(2) When used with incremental compilation.
Stimulus-Capable Tools

The In-System Memory Content Editor, the In-System Sources and Probes, and the Virtual JTAG interface each enable you to use the JTAG interface as a general-purpose communication port. Though all three tools can be used to achieve the same results, there are some considerations that make one tool easier to use in certain applications than others. In-System Sources and Probes is ideal for toggling control signals. The In-System Memory Content Editor is useful for inputting large sets of test data. Finally, the Virtual JTAG megafunction is well suited for more advanced users who want to develop their own customized JTAG solution.

In-System Sources and Probes

In-System Sources and Probes is an easy way to access JTAG resources to both read and write to your design. You can start by instantiating a megafunction into your HDL. The megafunction contains source ports and probe ports for driving values into and sampling values from the signals that are connected to the ports, respectively. Transaction details of the JTAG interface are abstracted away by the megafunction. During runtime, a GUI displays each source and probe port by instance and allows you to read from each probe port and drive to each source port. The GUI makes this tool ideal for toggling a set of control signals during the debugging process.

A good application of In-System Sources and Probes is to use the GUI as a replacement for the push buttons and LEDs used during the development phase of a project. Furthermore, In-System Sources and Probes supports a set of scripting commands for reading and writing using `quartus_stp`. When used with the Tk toolkit, you can build your own graphical interfaces—a feature that is ideal for building a virtual front panel during the prototyping phase of the design.

In-System Memory Content Editor

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

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

The Virtual JTAG Interface megafunction provides the finest level of granularity for manipulating the JTAG resource. This megafunction allows you to build your own JTAG scan chain by exposing all of the JTAG control signals and configuring your JTAG Instruction Registers (IRs) and JTAG Data Registers (DRs). During runtime, you control the IR/DR chain through a Tcl API. This feature is meant for users who have a thorough understanding of the JTAG interface and want precise control over the number and type of resources used.

Conclusion

The Quartus II on-chip debugging tool suite allows you to reach debugging closure quickly by providing you a set of powerful analysis tools and a set of tools that open up the JTAG port as a general purpose communication interface. The Quartus II software further broadens the scope of applications by giving you a comprehensive Tcl/Tk API. With the Tcl/Tk API, you cannot only increase the level of automation for all of the analysis tools, but you can also build virtual front panel applications quickly early in the prototyping phase.

In addition, all of the on-chip debugging tools have a tight integration with the rest of the productivity features within the Quartus II software. The incremental compile and incremental routing features enable a fast turnaround time for programming file generation. The cross-probing feature allows you to find and identify nodes quickly. The SignalTap II Logic Analyzer, when used with the TimeQuest Timing Analyzer, is a best-in-class timing verification suite that allows fast functional and timing verification.

This section contains the detailed usage for each of the On-Chip Debugging tools. This section contains the following chapters:

- Chapter 14, Quick Design Debugging Using SignalProbe
- Chapter 15, Design Debugging Using the SignalTap II Embedded Logic Analyzer
- Chapter 16, In-System Debugging Using External Logic Analyzers
- Chapter 17, In-System Updating of Memory and Constants
- Chapter 18, Design Debugging Using In-System Sources and Probes
This chapter provides detailed instructions about how to use SignalProbe to quickly debug your design.

Introduction

Hardware verification can be a lengthy and expensive process. The SignalProbe incremental routing feature helps reduce the hardware verification process and time-to-market for system-on-a-programmable-chip (SOPC) designs.

Easy access to internal device signals is important in the design or debugging process. The SignalProbe feature makes design verification more efficient by routing internal signals to I/O pins quickly without affecting the design. When you start with a fully routed design, you can select and route signals for debugging to either previously reserved or currently unused I/O pins.

The SignalProbe feature is fully functional with Arria® GX, Stratix® series, Cyclone® series, and MAX® II, device families.

If you are using the SignalProbe feature to debug your Stratix series, Cyclone series, or MAX II device, refer to “Debugging Using the SignalProbe Feature”.

The Quartus® II software provides a portfolio of on-chip debugging solutions. For an overview and comparison of all of the tools available in the Quartus II software on-chip debugging tool suite, refer to Section V. In-System Design Debugging in volume 3 of the Quartus II Handbook.

Debugging Using the SignalProbe Feature

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

This section describes the SignalProbe process for the Stratix series, Cyclone series, and MAX II device families.

You can reserve pins for SignalProbe and assign I/O standards before or after a full compilation. Each SignalProbe-source to SignalProbe-pin connection is implemented as an ECO change that is applied to your netlist after a full compilation.

To route the internal signals to the device’s reserved pins for SignalProbe, perform the following tasks:

2. Perform a Full Compilation, described on page 14–3.
3. Assign a SignalProbe Source, described on page 14–3.
4. Add Registers to the Pipeline Path to SignalProbe Pin, described on page 14–4.
5. Perform a SignalProbe Compilation, described on page 14–5.
6. Analyze the Results of the SignalProbe Compilation, described on page 14–5.

**Reserve the SignalProbe Pins**

SignalProbe pins can be reserved before or after compiling your design. Reserving SignalProbe pins before a compilation is optional. You can also reserve any unused I/Os of the device for SignalProbe pins after compilation. Assigning sources is a simple process after reserving SignalProbe pins. The sources for SignalProbe pins are the internal nodes and registers in the post-compilation netlist that you want to probe.

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

To reserve an available package pin as a SignalProbe pin using the SignalProbe Pins dialog box, perform the following steps:

1. On the Tools menu, click SignalProbe Pins. The SignalProbe Pins dialog box appears (Figure 14–1). The Pin name and I/O Standard appear as the only fields that are editable if place-and-route or fitting have not been performed.

   ![Figure 14–1. Reserving a SignalProbe Pin in the SignalProbe Pins Dialog Box](image)

2. In the Current and potential SignalProbe pins list, click a pin from the Number column and type your SignalProbe pin name in the Pin name box.

3. Select an I/O standard from the I/O standard list.

4. To add a new SignalProbe pin, click Add. To edit or change a previously reserved pin for SignalProbe, click Change. (Figure 14–1 shows how to use the dialog box to edit a previously reserved pin; if you were adding a new SignalProbe pin, the Add button appears instead of the Change button.)
5. Click OK.

**Perform a Full Compilation**

You must complete a full compilation to generate an internal netlist containing a list of internal nodes to probe to a SignalProbe outpin.

To perform a full compilation, on the Processing menu, click **Start Compilation**.

**Assign a SignalProbe Source**

A SignalProbe source can be any combinational node, register, or pin in your post-compilation netlist. To find a SignalProbe source, in the Node Finder, use the SignalProbe filter to remove all sources that cannot be probed. You might not be able to find a particular internal node because the node can be optimized away during synthesis, or the node cannot be routed to the SignalProbe pin, as it is untappable. For example, internal nodes and registers within the Gigabit transceivers cannot be probed because there are no physical routes to the pins available.

To probe virtual I/O pins generated in low-level partitions in an incremental compilation flow, select the source of the logic that feeds the Virtual Pin as your SignalProbe source pin.

To assign a SignalProbe source to your SignalProbe reserved pin, perform the following steps:

1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears (Figure 14–1 on page 14–2).
2. If a SignalProbe reserved pin is shown, in the **Current and potential SignalProbe pins** list, click the pin. Alternately, you can click an available pin number in the **Current and potential SignalProbe pins** list and type a new SignalProbe pin name in the **Pin name** box.
3. In the **Source** box, specify the source name. Click the browse button. The **Node Finder** dialog box appears.
4. When you open the **Node Finder** dialog box from the **SignalProbe Pins** dialog box, **SignalProbe** is selected by default in the **Filter** list. To show a set of nodes that can be probed in the **Nodes Found** list, click **List**.
5. In the **Nodes Found** list, select your source node and click the > button. The selected node appears in the **Selected Nodes** list.
6. Click **OK**.
7. After a source is selected, the **SignalProbe enable** option is turned on. Click **Change** or **Add** to accept the changes.

Because SignalProbe pins are implemented and routed as ECOs, turning the **SignalProbe enable** option on or off is the same as selecting **Apply Selected Change** or **Restore Selected Change** in the Change Manager window. (If the Change Manager window is not visible at the bottom of your screen, on the View menu, point to **Utility Windows** and click **Change Manager**.)
For more information about the Change Manager for the Chip Planner and Resource Property Editor, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

### Add Registers to the Pipeline Path to SignalProbe Pin

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

Figure 14–2 shows a single register between the SignalProbe source Reg_b_1 and SignalProbe SignalProbe_Output_2 output pin added to synchronize the data between the two SignalProbe output pins.

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

![Figure 14–2. Synchronizing SignalProbe Outputs with a SignalProbe Register](image)

To pipeline an existing SignalProbe connection, perform the following steps:

1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears.
2. Select a SignalProbe pin and in the Clock dialog box, type the clock name used to drive your registers, or click the browse button to use the Node Finder to select your clock source.
3. In the Registers dialog box, specify the number of registers you want to add in between the SignalProbe source and the SignalProbe output.
4. Click Change.
5. Click OK.
In addition to clock input for pipeline registers, you can also specify a reset signal pin for pipeline registers. To specify a reset pin for pipeline registers, use the Tcl command `make_sp`, as described in “Scripting Support” on page 14–11.

### Perform a SignalProbe Compilation

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

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

### Analyze the Results of the SignalProbe Compilation

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

The status of each SignalProbe pin appears in the Change Manager window (Figure 14–3). (If the Change Manager window is not visible at the bottom of your GUI, from the View menu, point to **Utility Windows** and click **Change Manager**.)

![Figure 14–3. Change Manager Window with SignalProbe Pins](image)

For more information about how to use the Change Manager, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

To view the timing results of each successfully routed SignalProbe pin, on the Processing menu, point to **Start** and click **Start Timing Analysis**.

### SignalProbe ECO Flows

SignalProbe pins are implemented using the same flow as other post-compilation changes made as ECOs. The following section describes SignalProbe ECO flows with and without the Quartus II incremental compilation feature.
SignalProbe ECO Flow with Quartus II Incremental Compilation

The incremental compilation feature is turned on by default. The top-level design is automatically set to a design partition when the incremental compilation feature is on. A design partition during incremental compilation can have different netlist types. (Netlist types can be set to source HDL, post synthesis, or post-fit.) The netlist type indicates whether that partition should be resynthesized or refit during Quartus II incremental compilation. Incremental compilation saves you time and preserves the placement of unchanged partitions in your design if small changes must be made to some partitions late in the design cycle.

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

The behavior of SignalProbe pins during an incremental compilation depends on the Netlist Type setting. When the top-level partition netlist type is set to post-fit, SignalProbe ECOs are retained if the partition being probed is preserved when you recompile the design.

SignalProbe connections always link the partition being probed with the top-level partition. As such, a SignalProbe connection might change the preservation attributes in a lower-level partition. This is known as partition linking. When partition linking occurs, all partitions that become linked share the attribute for the preservation level that is the strictest among all of the affected partitions. As a result, when you tap any partitions that are not post-fit and the top level is set to a netlist type of post-fit, your SignalProbe connection is preserved.

The behavior is different in the case that your top-level partition netlist type is set to post-synthesis and you have no other lower-level partitions defined. In this case, the partition with the strictest preservation type is set to post-synthesis. If you create SignalProbe ECOs and recompile the design, your SignalProbe ECOs are not retained and a warning message appears in the Messages window. The warning indicates that ECO modifications are discarded; however, all of the ECO information is retained in the Change Manager. In this case, apply SignalProbe ECOs from the Change Manager and perform the Check and Save All Netlist Changes step, as described in “SignalProbe ECO Flow Without Quartus Incremental Compilation” on page 14–6.

SignalProbe ECO Flow Without Quartus Incremental Compilation

If you do not use the Quartus II incremental compilation feature and you implement SignalProbe pins after the initial compilation of your design, SignalProbe ECOs are not retained during recompilation. However, all of the SignalProbe ECOs remain in the Change Manager.

To apply a SignalProbe ECO, right-click in the Change Manager and select Apply Selected Change (Figure 14–4). (If the Change Manager window is not visible at the bottom of your screen, from the View menu, point to Utility Windows and click Change Manager.)
Alternately, you can use the **SignalProbe Pins** dialog box to enable the ECOs, as shown in Figure 14–5. This has the same effect as applying the SignalProbe ECOs within the Change Manager.

**Figure 14–5.** Enabling ECOs in the SignalProbe Pins Dialog Box

After applying the selected SignalProbe ECO, either right-click anywhere in the Change Manager and select **Check and Save All Netlist Changes** (Figure 14–6), or, on the Processing menu, point to **Start** and click **Start Check and Save All Netlist Changes** to perform the ECO compilation.
Common Questions About the SignalProbe Feature

The following are answers to common questions about the SignalProbe feature.

**Why Did I Get the Following Error Message, **"**Error: There are No Enabled SignalProbes to Process**"?**

This error message is generated when a SignalProbe compilation was attempted with either no SignalProbe pins to route, or with all SignalProbe pins disabled.

This might occur if you perform a SignalProbe compilation after a full compilation. For example, when a full compilation is performed, all SignalProbe pins are disabled. You can create or re-enable your SignalProbe pins in the SignalProbe Pins dialog box.

**How Can I Retain My SignalProbe ECOs During Re-Compilation of My Design?**

To retain your existing ECOs during recompilation of your design, you must use Quartus II incremental compilation. To learn more about the flow, refer to “SignalProbe ECO Flow with Quartus II Incremental Compilation” on page 14–6.

**Why Did My SignalProbe Source Disappear in the Change Manager?**

The SignalProbe source information for each SignalProbe connection is stored in the project database (db directory). SignalProbe pins are post-compilation changes to your netlist and are interpreted as ECOs. These changes are stored in the project db and if the project database is removed, the SignalProbe source information is lost and does not appear in the SignalProbe Pins dialog box. To restore your SignalProbe pins after the design compilation step, source the signalprobe_qsf.tcl script located in your project directory.

To restore your SignalProbe source information after compilation, type the following command from a command-line prompt:

```
quartus_cdb -t signalprobe_qsf.tcl
```

Before typing this command, you must close your design project. When the command finishes, you can open your design project again. The Change Manager shows the sources for SignalProbe pins.

**What is an ECO and Where Can I Find More Information about ECOs?**

ECOs are late design cycle changes made to your design that do not alter functionality and timing.
For more information about ECOs and using the Change Manager, refer to the Engineering Change Management with the Chip Planner chapter in volume 2 of the Quartus II Handbook.

**How Do I Migrate My Previous SignalProbe Assignments in the Quartus II Software Version 5.1 and Earlier to Version 6.0 and Later?**

In earlier versions of the Quartus II software, SignalProbe pins were stored in the Quartus II Settings File (.qsf). These assignments are automatically converted into ECO changes when you open the SignalProbe dialog box or when you start a SignalProbe compilation in the Quartus II software versions 6.0 and higher.

For example, the SignalProbe source assignment from a .qsf file is removed and added to the Change Manager as an ECO after the SignalProbe dialog box is opened, or when you perform a SignalProbe compilation. Example 14–1 shows SignalProbe assignments in the .qsf file. Example 14–2 shows the same assignments after opening the SignalProbe Pins dialog box.

**Example 14–1. SignalProbe Assignments in the Quartus II Settings File**

```plaintext
set_location_assignment PIN_C22 -to my_signalprobe_pin
set_instance_assignment -name RESERVE_PIN "AS SIGNALPROBE OUTPUT" -to my_signalprobe_pin
set_instance_assignment -name IO_STANDARD LVTTL -to my_signalprobe_pin
set_instance_assignment -name SIGNALPROBE_ENABLE ON -to my_signalprobe_pin
set_instance_assignment -name SIGNALPROBE_SOURCE inst5[0] -to my_signalprobe_pin
```

**Example 14–2. SignalProbe Assignments in the Quartus II Settings File after Opening the SignalProbe Pins Dialog Box**

```plaintext
set_location_assignment PIN_C22 -to my_signalprobe_pin
set_instance_assignment -name RESERVE_PIN "AS SIGNALPROBE OUTPUT" -to my_signalprobe_pin
set_instance_assignment -name IO_STANDARD LVTTL -to my_signalprobe_pin
set_instance_assignment -name SIGNALPROBE_ENABLE ON -to my_signalprobe_pin
```

**What are all the Changes for the SignalProbe Feature between the Quartus II Software Version 5.1 and Earlier, and Version 6.0 and Later?**

The following list highlights the changes that affect users of the SignalProbe feature in the Quartus II software versions 5.1 and earlier. This applies to Stratix series, Cyclone series, and MAX II device families.

For more information about the changes that pertain to each release of the Quartus II software, refer to the Release Notes on the Altera website (www.altera.com).

- In Quartus II software versions 5.1 and earlier, the SignalProbe Pins dialog box was accessed on the Assignments menu. To access it with the Quartus II software version 6.0 and later, on the Tools menu, click SignalProbe Pins.
- A full compilation is required before making SignalProbe connections. However, you can still reserve pins before compilation for later use by SignalProbe. You can reserve pins by creating a SignalProbe in the SignalProbe dialog box without specifying a source. This is the same behavior as in the Quartus II software version 5.1.
To route the SignalProbe pins, you must perform a SignalProbe compilation after a full compilation. The **Automatically route SignalProbe signals during compilations** and **Modify latest fitting results during SignalProbe compilation** options are no longer supported.

After subsequent compiles, full or incremental, existing SignalProbe pins are disabled and are not present in the post-compilation netlist. To add them back, enable the SignalProbe pins and perform a SignalProbe compilation.

SignalProbe pins are not controlled via assignments in the `.qsf` file because they are now ECOs. Existing `.qsf` files automatically convert to ECOs when a SignalProbe compilation is performed or when the **SignalProbe** dialog box is opened.

The Tcl interface for creating SignalProbe pins has improved and is a part of the Chip Planner package `::quartus::chip_editor`. Refer to “Scripting Support” on page 14–11.

Previously, the `quartus_fit --signalprobe` command was used to perform a SignalProbe compilation. This is not supported in the Quartus II software version 6.0 and later, and is replaced by the improved Tcl interface and the `check_netlist_and_save` Tcl command.

The SignalProbe timing report generated after a successful SignalProbe compilation is not available in the Quartus II software version 6.0 and later. You can view the timing results of your SignalProbe pins in the SignalProbe Fitting Results, under the Fitter report, or in the tco results page of the Timing report.

You cannot make SignalProbe pins in the Assignment Editor. Use the **SignalProbe Pins** dialog box to make and edit your SignalProbe pins.

### Why Can't I Reserve a SignalProbe Pin?

If you cannot reserve a SignalProbe pin in the Quartus II software, it is likely that one of the following is true:

- You have selected multiple pins.
- A compile is running in the background. Wait until the compilation is complete before reserving the pin.
- You have the Quartus II Web Edition software, in which the SignalProbe feature is not enabled by default. You must turn on TalkBack to enable the SignalProbe feature in the Quartus II Web Edition software.
- You have not set the pin reserve type to **As Signal Probe Output**. To reserve a pin, on the Assignments menu, in the **Assign Pins** dialog box, select **As SignalProbe Output**.
- The pin is reserved from a previous compilation. During a compilation, the Quartus II software reserves each pin on the targeted device. If you end the Quartus II process during a compilation, for example, with the **Windows Task Manager End Process** command or the UNIX `kill` command, perform a full recompilation before reserving pins as SignalProbe outputs.
- The pin does not support the SignalProbe feature. Select another pin.
- The current family does not support the SignalProbe feature.
Scripting Support

Running procedures and make settings using a Tcl script are described in this chapter. 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 *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 all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Make a SignalProbe Pin

To make a SignalProbe pin, type the following command:

```
-loc <loc> -pin_name <pin name> [-regs <regs>] [-reset <reset>] \
-src_name <source name>
```

Delete a SignalProbe Pin

To delete a SignalProbe pin, type the following command:

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

Enable a SignalProbe Pin

To enable a SignalProbe pin, type the following command:

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

Disable a SignalProbe Pin

To disable a SignalProbe pin, type the following command:

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

Perform a SignalProbe Compilation

To perform a SignalProbe compilation, type the following command:

```
check_netlist_and_save
```

Migrate Previous SignalProbe Pins to the Quartus II Software Versions 6.0 and Later

To migrate previous SignalProbe pins to the Quartus II software versions 6.0 and later, type the following command:

```
convert_signal_probes
```

Script Example

Example 14–3 shows a script that creates a SignalProbe pin called `sp1` and connects the `sp1` pin to source node `reg1` in a project that was already compiled.
Adding SignalProbe Sources

A SignalProbe source is a signal in the post-compilation design database with a possible route to an output pin. To assign a SignalProbe source to a SignalProbe pin or an unused output pin, perform the following steps:

1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears.
2. In the **Current and potential SignalProbe pins** list, select the SignalProbe pin to which you want to add a SignalProbe source.
3. Click **Browse** and select a SignalProbe source.
4. Click **OK**.
5. In the **Assign SignalProbe Pins** dialog box, if a source has not been assigned to the SignalProbe pin, click **Add**. If a SignalProbe pin has been already assigned, click **Change**.
6. Click **OK**.

The **Node Finder** dialog box appears with the SignalProbe filter selected (Figure 14–7). Click **List** to view all of the available SignalProbe sources. If you cannot find a specific node with the SignalProbe filter, the node has either been removed by the Quartus II software during optimization or placed in the device where there are no possible routes to a pin.

**Figure 14–7.** Available SignalProbe Sources in the Node Finder

---

<table>
<thead>
<tr>
<th>Node Finder</th>
<th>Filter</th>
<th>SignalProbe</th>
<th>Customize</th>
<th>List</th>
<th>OK</th>
<th>Cancel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Finder</td>
<td>Filter</td>
<td>SignalProbe</td>
<td>Customize</td>
<td>List</td>
<td>OK</td>
<td>Cancel</td>
</tr>
<tr>
<td>Nodes Found</td>
<td>Name</td>
<td>Value</td>
<td>Filter</td>
<td>SignalProbe</td>
<td>Customize</td>
<td>List</td>
</tr>
<tr>
<td>Name</td>
<td>Value</td>
<td>Filter</td>
<td>SignalProbe</td>
<td>Customize</td>
<td>List</td>
<td>OK</td>
</tr>
<tr>
<td>Name</td>
<td>Value</td>
<td>Filter</td>
<td>SignalProbe</td>
<td>Customize</td>
<td>List</td>
<td>OK</td>
</tr>
<tr>
<td>Name</td>
<td>Value</td>
<td>Filter</td>
<td>SignalProbe</td>
<td>Customize</td>
<td>List</td>
<td>OK</td>
</tr>
</tbody>
</table>

---

When the source of the SignalProbe pin is added or changed, the SignalProbe pin is automatically enabled. To disable a SignalProbe pin, turn off SignalProbe enable.
Performing a SignalProbe Compilation

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

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

To run the SignalProbe compilation automatically after a full compilation, on the Tools menu, click **SignalProbe Pins**. In the **SignalProbe Pins** dialog box, turn on **Automatically route SignalProbe signals during compilation**.

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

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

To enable or disable each SignalProbe pin, in the **SignalProbe Pins** dialog box, turn the **SignalProbe enable** option on or off.

Running SignalProbe with Smart Compilation

Running a smart compilation reduces compilation time by running only necessary modules during compilation. However, a full compilation is required if any design files, Analysis and Synthesis settings, or Fitter settings have changed.

To turn on smart compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings** and turn on **Use Smart compilation**.

If you run a SignalProbe compilation with smart compilation turned on, and there are changes to a design file or settings related to the Analysis and Synthesis or Fitter modules, the following message is displayed:

**Error: Can't perform SignalProbe compilation because design requires a full compilation.**

You should turn smart compilation on, which allows you to work with the latest settings and design files.

Understanding the Results of a SignalProbe Compilation

After a SignalProbe compilation, the results appear in two sections of the compilation report file. The fitting results and status (Table 14–1) of each SignalProbe pin is displayed in the **SignalProbe Fitting Result** screen in the Fitter section of the Compilation Report (Figure 14–8).

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routed</td>
<td>Connected and routed successfully</td>
</tr>
<tr>
<td>Not Routed</td>
<td>Not enabled</td>
</tr>
</tbody>
</table>
The timing results of each successfully routed SignalProbe pin is displayed in the SignalProbe source to output delays screen in the Timing Analysis section of the Compilation Report (Figure 14–9).

![Figure 14–8. SignalProbe Fitting Results Page in the Compilation Report Window](image)

**Table 14–1. Status Values (Part 2 of 2)**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failed to Route</td>
<td>Failed routing during last SignalProbe compilation</td>
</tr>
<tr>
<td>Need to Compile</td>
<td>Assignment changed since last SignalProbe compilation</td>
</tr>
</tbody>
</table>

![Figure 14–9. SignalProbe Source to Output Delays Page in the Compilation Report Window](image)

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

**Analyzing SignalProbe Routing Failures**

The SignalProbe can begin compilation; however, one of the following reasons can prevent complete compilation:

- **Route unavailable**—the SignalProbe compilation failed to find a route from the SignalProbe source to the SignalProbe pin because of routing congestion
- **Invalid or nonexistent SignalProbe source**—you entered a SignalProbe source that does not exist or is invalid
Unusable output pin—the output pin selected is found to be unusable

Routing failures can occur if the SignalProbe pin’s I/O standard conflicts with other I/O standards in the same I/O bank.

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

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

SignalProbe Scripting Support

Running procedures and making settings using a Tcl script are described in this chapter. 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 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. Refer to the Quartus II Settings File Reference Manual for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the Command-Line Scripting chapter in volume 2 of the Quartus II Handbook.

Reserving SignalProbe Pins

To reserve a SignalProbe pin, type the commands shown in Example 14–4.

**Example 14–4.** Reserving a SignalProbe Pin

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

Valid locations are pin location names, such as Pin_A3.

For more information about reserving SignalProbe pins, refer to “Reserve the SignalProbe Pins” on page 14–2.
Adding SignalProbe Sources

Use the following Tcl commands to add SignalProbe sources. For more information about adding SignalProbe sources, refer to “Adding SignalProbe Sources” on page 14–12.

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

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

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

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

Assigning I/O Standards

To assign an I/O standard to a pin, type the following Tcl command:

```
set_instance_assignment -name IO_STANDARD <I/O standard> -to \<SignalProbe pin name>
```

For a list of valid I/O standards, refer to the I/O Standards general description in the Quartus II Help.

Adding Registers for Pipelining

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

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

Run SignalProbe Automatically

To run SignalProbe automatically after a full compile, type the following Tcl command:

```
set_global_assignment -name SIGNALPROBE_DURING_NORMAL_COMPILATION ON
```

For more information about running SignalProbe automatically, refer to “Performing a SignalProbe Compilation” on page 14–13.

Run SignalProbe Manually

To run SignalProbe manually with a Tcl command or the quartus_fit command, type the following at a command prompt.

```
execute_flow -signalprobe
```

The execute_flow command is in the flow package. At a command prompt, type the following command:

```
quartus_fit <project name> --signalprobe
```

For more information about running SignalProbe manually, refer to “Performing a SignalProbe Compilation” on page 14–13.
Enable or Disable All SignalProbe Routing

Use the Tcl command in Example 14–5 to turn on or turn off SignalProbe routing. When using this command, to turn SignalProbe routing on, specify ON. To turn SignalProbe routing off, specify OFF.

Example 14–5. Turning SignalProbe On or Off with Tcl Commands

```tcl
set spe [get_all_assignments -name SIGNALPROBE_ENABLE] \
foreach_in_collection asgn $spe {
    set signalprobe_pin_name [lindex $asgn 2]
    set_instance_assignment -name SIGNALPROBE_ENABLE -to \
    $signalprobe_pin_name <ON|OFF> }
```

For more information about enabling or disabling SignalProbe routing, refer to page 14–13.

Running SignalProbe with Smart Compilation

To turn on Smart Compilation, type the following Tcl command:

```tcl
set_global_assignment -name SMART_RECOMPILE
```

For more information, refer to “Running SignalProbe with Smart Compilation” on page 14–13.

Allow SignalProbe to Modify Fitting Results

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

```tcl
set_global_assignment -name SIGNALPROBE_ALLOW_OVERUSE
```

For more information, refer to “Analyzing SignalProbe Routing Failures” on page 14–14.

Conclusion

Using the SignalProbe feature can significantly reduce the time required compared to a full recompilation. Use the SignalProbe feature for quick access to internal design signals to perform system-level debugging.

Referenced Documents

This chapter references the following documents:

- Command-Line Scripting chapter in volume 2 of the Quartus II Handbook
- Engineering Change Management with the Chip Planner chapter in volume 2 of the Quartus II Handbook
- Release Notes on the Altera website (www.altera.com)
- Section V. In-System Design Debugging in volume 3 of the Quartus II Handbook
- Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook
- Quartus II Settings File Reference Manual
Tcl Scripting chapter in volume 2 of the Quartus II Handbook

Document Revision History

Table 14–2 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>■ Removed all references and procedures for APEX devices.</td>
<td>Updated for the Quartus II software version 9.1 release.</td>
</tr>
<tr>
<td></td>
<td>■ Style changes.</td>
<td></td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>■ Removed the “Generate the Programming File” section</td>
<td>Updated for the Quartus II software version 9.0 release.</td>
</tr>
<tr>
<td></td>
<td>■ Removed unnecessary screenshots</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Minor editorial updates</td>
<td></td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>■ Modified description for preserving SignalProbe connections when using Incremental Compilation</td>
<td>Updated for the Quartus II software version 8.1 release.</td>
</tr>
<tr>
<td></td>
<td>■ Added plausible scenarios where SignalProbe connections are not reserved in the design</td>
<td></td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>■ Added “Arria GX” to the list of supported devices</td>
<td>Organizational changes for the Quartus II software version 8.0 release.</td>
</tr>
<tr>
<td></td>
<td>■ Removed the “On-Chip Debugging Tool Comparison” and replaced with a reference to the Section V Overview on page 13–1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Added hyperlinks to referenced documents throughout the chapter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Minor editorial updates</td>
<td></td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.
15. Design Debugging Using the SignalTap II Embedded Logic Analyzer

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.
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>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple logic analyzers in a single device</td>
<td>Captures data from multiple clock domains in a design at the same time.</td>
</tr>
<tr>
<td>Multiple logic analyzers in multiple devices in a single JTAG chain</td>
<td>Simultaneously captures data from multiple devices in a JTAG chain.</td>
</tr>
<tr>
<td>Plug-In Support</td>
<td>Easily specifies nodes, triggers, and signal mnemonics for IP, such as the Nios® II embedded processor.</td>
</tr>
<tr>
<td>Up to 10 basic or advanced trigger conditions for each analyzer instance</td>
<td>Enables more complex data capture commands to be sent to the logic analyzer, providing greater accuracy and problem isolation.</td>
</tr>
<tr>
<td>Power-Up Trigger</td>
<td>Captures signal data for triggers that occur after device programming but before manually starting the logic analyzer.</td>
</tr>
<tr>
<td>State-based Triggering Flow</td>
<td>Enables you to organize your triggering conditions to precisely define what your embedded logic analyzer will capture.</td>
</tr>
<tr>
<td>Incremental compilation</td>
<td>Modifies the SignalTap II Embedded Logic Analyzer monitored signals and triggers without performing a full compilation, saving time.</td>
</tr>
<tr>
<td>Flexible buffer acquisition modes</td>
<td>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.</td>
</tr>
<tr>
<td>MATLAB integration with included MEX function</td>
<td>Collects the SignalTap II Embedded Logic Analyzer captured data into a MATLAB integer matrix.</td>
</tr>
<tr>
<td>Up to 2,048 channels per logic analyzer instance</td>
<td>Samples many signals and wide bus structures.</td>
</tr>
<tr>
<td>Up to 128K samples in each device</td>
<td>Captures a large sample set for each channel.</td>
</tr>
<tr>
<td>Fast clock frequencies</td>
<td>Synchronous sampling of data nodes using the same clock tree driving the logic under test.</td>
</tr>
<tr>
<td>Resource usage estimator</td>
<td>Provides estimate of logic and memory device resources used by SignalTap II Embedded Logic Analyzer configurations.</td>
</tr>
<tr>
<td>No additional cost</td>
<td>The SignalTap II Embedded Logic Analyzer is included with a Quartus II subscription and with the Quartus II Web Edition (with TalkBack enabled).</td>
</tr>
<tr>
<td>Compatibility with other on-chip debugging utilities</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
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
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).
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>
<thead>
<tr>
<th>Signals (Width)</th>
<th>Samples (Depth)</th>
<th>256</th>
<th>512</th>
<th>2,048</th>
<th>8,192</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>&lt; 1</td>
<td>1</td>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>8</td>
<td>32</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>32</td>
<td>128</td>
<td>512</td>
<td></td>
</tr>
</tbody>
</table>

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:

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).
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.
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.

   ![Figure 15–6. Select Embedded Logic Analyzer Parameters](image)

   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.
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>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acq_data_in</td>
<td>Input</td>
<td>No</td>
<td>This set of signals represents signals that are monitored in the SignalTap II Embedded Logic Analyzer.</td>
</tr>
<tr>
<td>acq_trigger_in</td>
<td>Input</td>
<td>No</td>
<td>This set of signals represents signals that are used to trigger the analyzer.</td>
</tr>
</tbody>
</table>
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 megafuction 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

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acq_clk</td>
<td>Input</td>
<td>Yes</td>
<td>This port represents the sampling clock that the SignalTap II Embedded Logic Analyzer uses to capture data.</td>
</tr>
<tr>
<td>trigger_in</td>
<td>Input</td>
<td>No</td>
<td>This signal is used to trigger the SignalTap II Embedded Logic Analyzer.</td>
</tr>
<tr>
<td>trigger_out</td>
<td>Output</td>
<td>No</td>
<td>This signal is enabled when the trigger event occurs.</td>
</tr>
<tr>
<td>storage_enable</td>
<td>Input</td>
<td>No</td>
<td>This signal is used to enable a write transaction into the acquisition buffer.</td>
</tr>
</tbody>
</table>
Configure the SignalTap II Embedded Logic Analyzer

Configure the SignalTap II Embedded Logic Analyzer

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 \texttt{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 \textit{Analyzing Designs with Quartus II Netlist Viewers} chapter in volume 1 of the \textit{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 (“\texttt{\textasciitilde}”) 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**

![Figure 15–8. Finding Data Signals Using the Technology Map Viewer](image)

**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.
Configure the SignalTap II Embedded Logic Analyzer

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 megafuction**—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.

   ![Image](https://example.com/image.png)

   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.

   ![Image](https://example.com/image.png)

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.

   ![Image](https://example.com/image.png)

   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

<table>
<thead>
<tr>
<th>Type</th>
<th>Alias</th>
<th>Name</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMD</td>
<td>state</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

(a) Non-segmented Buffer

(b) Segmented Buffer

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.
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)
Configure the SignalTap II Embedded Logic Analyzer

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

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

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

(1) StorageQualifier 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.
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 control 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>
<thead>
<tr>
<th>Name of Operator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less Than</td>
<td>Comparison</td>
</tr>
<tr>
<td>Less Than or Equal To</td>
<td>Comparison</td>
</tr>
<tr>
<td>Equality</td>
<td>Comparison</td>
</tr>
<tr>
<td>Inequality</td>
<td>Comparison</td>
</tr>
<tr>
<td>Greater Than</td>
<td>Comparison</td>
</tr>
<tr>
<td>Greater Than or Equal To</td>
<td>Comparison</td>
</tr>
<tr>
<td>Logical NOT</td>
<td>Logical</td>
</tr>
<tr>
<td>Logical AND</td>
<td>Logical</td>
</tr>
<tr>
<td>Logical OR</td>
<td>Logical</td>
</tr>
<tr>
<td>Logical XOR</td>
<td>Logical</td>
</tr>
<tr>
<td>Reduction AND</td>
<td>Reduction</td>
</tr>
<tr>
<td>Reduction OR</td>
<td>Reduction</td>
</tr>
<tr>
<td>Reduction XOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>Left Shift</td>
<td>Shift</td>
</tr>
<tr>
<td>Right Shift</td>
<td>Shift</td>
</tr>
<tr>
<td>Bitwise Complement</td>
<td>Bitwise</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>Bitwise</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td>Bitwise</td>
</tr>
<tr>
<td>Bitwise XOR</td>
<td>Bitwise</td>
</tr>
<tr>
<td>Edge and Level Detector</td>
<td>Signal Detection</td>
</tr>
</tbody>
</table>

**Table 15–4. Advanced Triggering Operators (Note 1)**

**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 workspace 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

<table>
<thead>
<tr>
<th>Advanced Trigger Condition Editor</th>
<th>Level 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result: <code>outa</code> = <code>outb</code></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of Advanced Trigger Condition Editor](image)

- 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

| Result: `outa` = `outb` & `enable` |

![Diagram of Advanced Trigger Condition Editor](image)
Define Triggers

- 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

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

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.
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.
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.
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>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination of goto action</td>
<td>Allows you to modify the destination of the state transition at runtime.</td>
</tr>
<tr>
<td>Comparison values</td>
<td>Allows comparison values in Boolean expressions to be modifiable at runtime. In addition, it allows the segment_trigger and trigger action post-fill count argument to be modifiable at runtime.</td>
</tr>
<tr>
<td>Comparison operators</td>
<td>Allows comparison operators in Boolean expressions to be modifiable at runtime.</td>
</tr>
<tr>
<td>Logical operators</td>
<td>Allows the logical operators in Boolean expressions to be modifiable at runtime.</td>
</tr>
</tbody>
</table>

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)

\[
\text{state <State_label>:}<action_list>
\]

\[
\text{if( <Boolean_expression> )<action_list>}
\]

\[
[\text{else if ( <boolean_expression> )}<action_list>] (1)
\]

\[
\text{[else}<action_list>]
\]

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.

\[
\text{state <state_label>:} \text{ begins the description of the actions evaluated when this state is reached.}
\]

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>NOT operator</td>
<td>expr1</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND operator</td>
<td>expr1 &amp;&amp; expr2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&lt;identifier&gt; &gt; &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or Equal to</td>
<td>&lt;identifier&gt; &gt;= &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>==</td>
<td>Equals</td>
<td>&lt;identifier&gt; == &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>!=</td>
<td>Does not equal</td>
<td>&lt;identifier&gt; != &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>&lt;identifier&gt; &lt;= &lt;numerical_value&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>&lt;identifier&gt; &lt; &lt;numerical_value&gt;</td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>increment</td>
<td>Increments a counter resource by 1</td>
<td><code>increment &lt;counter_identifier&gt;;</code></td>
</tr>
<tr>
<td>decrement</td>
<td>Decrements a counter resource by 1</td>
<td><code>decrement &lt;counter_identifier&gt;;</code></td>
</tr>
</tbody>
</table>
Define Triggers

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>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigger</td>
<td>Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.</td>
<td><code>trigger &lt;post-fill_count&gt;%;</code></td>
</tr>
<tr>
<td>segment_trigger</td>
<td>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.</td>
<td><code>segment_trigger &lt;post-fill_count&gt;%;</code></td>
</tr>
</tbody>
</table>

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.

When using 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

```plaintext
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.
The logic analyzer will be able to finish the acquisition and display the contents of the waveform if it can successfully finish the post-fill acquisition samples before Condition 3 occurs. In this specific case, the capture ends if the post-fill count value is less than \( m \).

If the post-fill count value specified in Trigger Flow description 1 is greater than \( m \) samples, the buffer pauses acquisition indefinitely, provided there is no recurrence of Condition 1 to trigger the logic analyzer to start capturing data again. The SignalTap II 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 and Figure 15–36 show a real data acquisition of the scenario described in Figure 15–33. Figure 15–35 illustrates a scenario where the data capture finishes successfully. It uses a buffer with a sample depth of 64, \( m = n = 10 \), and the post-fill count value = 5. Figure 15–36 illustrates a scenario where the logic analyzer pauses indefinitely even after a trigger condition occurs due to a `stop_store` condition. This scenario uses a sample depth of 64, with \( m = n = 10 \) and post-fill count = 15.

**Figure 15–35.** Storage Qualification with Post-Fill Count Value Less than \( m \) (Acquisition successfully completes)
The combination of using counters, Boolean and relational operators in conjunction with the `start_store` and `stop_store` commands can give a clock-cycle level of resolution to controlling the samples that 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.

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

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).
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.

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.
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 \( f_{\text{MAX}} \) 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 \( f_{\text{MAX}} \) 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

![JTAG Chain Diagram](image)

### 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.
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

<table>
<thead>
<tr>
<th>Runtime Reconfigurable Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Trigger Conditions and Basic Storage Qualifier Conditions</td>
<td>All signals that have the trigger check box enabled can be changed to any basic trigger condition value without recompiling.</td>
</tr>
<tr>
<td>Advanced Trigger Conditions and Advanced Storage Qualifier</td>
<td>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.</td>
</tr>
<tr>
<td>Switching between a storage-qualified and a continuous acquisition</td>
<td>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.</td>
</tr>
<tr>
<td>State-based trigger flow parameters</td>
<td>Refer to Table 15–5 for a list of Reconfigurable State-based trigger flow options.</td>
</tr>
</tbody>
</table>

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”

```vhdl
state ST1:
if ( condition1 && (c1 <= m) ) // each "segment" triggers on condition
   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
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
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

<table>
<thead>
<tr>
<th>Message</th>
<th>Message Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not running</td>
<td>The SignalTap II Logic Analyzer is not running. There is no connection to a device or the device is not configured.</td>
</tr>
<tr>
<td>(Power-Up Trigger) Waiting for clock (1)</td>
<td>The SignalTap II Logic Analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.</td>
</tr>
<tr>
<td>Acquiring (Power-Up) pre-trigger data (1)</td>
<td>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.</td>
</tr>
<tr>
<td>Trigger In conditions met</td>
<td>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.</td>
</tr>
<tr>
<td>Waiting for (Power-up) trigger (1)</td>
<td>The SignalTap II Logic Analyzer is now waiting for the trigger event to occur.</td>
</tr>
<tr>
<td>Trigger level &lt;x&gt; met</td>
<td>The condition of trigger condition x has occurred. The SignalTap II Embedded Logic Analyzer is waiting for the condition specified in condition x+1 to occur.</td>
</tr>
<tr>
<td>Acquiring (power-up) post-trigger data (1)</td>
<td>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.</td>
</tr>
<tr>
<td>Offload acquired (Power-Up) data (1)</td>
<td>Data is being transmitted to the Quartus II software through the JTAG chain.</td>
</tr>
<tr>
<td>Ready to acquire</td>
<td>The SignalTap II Embedded Logic Analyzer is waiting for the user to arm the analyzer.</td>
</tr>
</tbody>
</table>

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.
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 15–47. Captured SignalTap II Data](image)
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.

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

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

```matlab
gaddpath <Quartus install directory>\win
```

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

```matlab
alt_signaltap_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:

```matlab
stp = alt_signaltap_run(''<stp filename>>','[,''
signaled']|'unsigned')
\n''\n'\n'signaled']|'unsigned')
\n'signal set name'
'my_signalset'
'my_trigger'

<table>
<thead>
<tr>
<th>Option</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed</td>
<td>'signed'</td>
<td>The <code>signed</code> 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 <code>unsigned</code> option keeps the data as an unsigned integer. The default is <code>signed</code>.</td>
</tr>
<tr>
<td>unsigned</td>
<td>'unsigned'</td>
<td></td>
</tr>
<tr>
<td><code>&lt;instance name&gt;</code></td>
<td>'auto_signaltap_0'</td>
<td>Specify a SignalTap II instance if more than one instance is defined. The default is the first instance in the .stp file, <code>auto_signaltap_0</code>.</td>
</tr>
<tr>
<td><code>&lt;signal set name&gt;</code></td>
<td>'my_signalset'</td>
<td>Specify the signal set and trigger from the SignalTap II data log if multiple configurations are present in the .stp file. The default is the active signal set and trigger in the file.</td>
</tr>
<tr>
<td><code>&lt;trigger name&gt;</code></td>
<td>'my_trigger'</td>
<td></td>
</tr>
</tbody>
</table>

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:

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

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

```matlab
alt_signaltap_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).
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.
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.
Example 15–6 illustrates how to compile a design with the SignalTap II Embedded Logic Analyzer at the command line.

```
quartus_stp filtref --stp_file stp1.stp --enable
quartus_map filtref --source=filtref.bdf --family=CYCLONE
quartus_fit filtref --part=EP1C12Q240C6 --fmax=80MHz --tsu=8ns
quartus_asm filtref
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>
<thead>
<tr>
<th>Command</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open_session</td>
<td>-name &lt;stp_filename&gt;</td>
<td>Opens the specified .stp file. All captured data is stored in this file.</td>
</tr>
<tr>
<td>run</td>
<td>-instance &lt;instance_name&gt;</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>-signal_set &lt;signal_set&gt;</td>
<td>(optional)</td>
</tr>
<tr>
<td></td>
<td>-trigger &lt;trigger_name&gt;</td>
<td>(optional)</td>
</tr>
<tr>
<td></td>
<td>-data_log &lt;data_log_name&gt;</td>
<td>(optional)</td>
</tr>
<tr>
<td></td>
<td>-timeout &lt;seconds&gt;</td>
<td>(optional)</td>
</tr>
<tr>
<td>run_multiple_start</td>
<td>None</td>
<td>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.</td>
</tr>
<tr>
<td>run_multiple_end</td>
<td>None</td>
<td>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.</td>
</tr>
<tr>
<td>stop</td>
<td>None</td>
<td>Stops data acquisition.</td>
</tr>
<tr>
<td>close_session</td>
<td>None</td>
<td>Closes the currently open .stp file. You cannot run the analyzer after the .stp file is closed.</td>
</tr>
</tbody>
</table>

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

When the script is completed, open the .stp 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.

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

Each segment acts as a non-segmented buffer that continuously updates the memory contents with the signal values. The last acquisition before stopping the buffer is displayed on the Data tab as the last sample number in the affected segment. The trigger position in the affected segment is then defined by \( N - \text{post count fill} \), 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.**

```plaintext
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*
Table 15–15 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>No change to content.</td>
<td>—</td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>Updated Table 15–1</td>
<td>Updated for the Quartus II software version 9.0 release.</td>
</tr>
<tr>
<td></td>
<td>Updated “Using Incremental Compilation with the SignalTap II Embedded Logic Analyzer” on page 15–55</td>
<td>Updated for the Quartus II software version 9.0 release.</td>
</tr>
<tr>
<td></td>
<td>Added new Figure 15–42</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Made minor editorial updates</td>
<td></td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Updated for the Quartus II software version 8.1 release:</td>
<td>Updated for the Quartus II software version 8.1 release.</td>
</tr>
<tr>
<td></td>
<td>Added new section “Using the Storage Qualifier Feature” on page 14–25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Added description of start_store and stop_store commands in section “Trigger Condition Flow Control” on page 14–36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Added new section “Runtime Reconfigurable Options” on page 14–63</td>
<td></td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>Updated for the Quartus II software version 8.0:</td>
<td>Updated for the Quartus II software version 8.0 release.</td>
</tr>
<tr>
<td></td>
<td>Added “Debugging Finite State machines” on page 14-24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>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</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Added “Capturing Data Using Segmented Buffers” on page 14–16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Added hyperlinks to referenced documents throughout the chapter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Minor editorial updates</td>
<td></td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.
The Quartus II Logic Analyzer Interface (LAI) allows you to use an external logic analyzer and a minimal number of FPGA I/O pins to examine the behavior of internal signals while your design is running at full speed on your FPGA.

Introduction

The LAI connects a large set of internal device signals to a small number of output pins. You can connect these output pins to an external logic analyzer for debugging purposes. In the Quartus II LAI, the internal signals are grouped together, distributed to a user-configurable multiplexer, and then output to available I/O pins on your FPGA. Instead of having a one-to-one relationship between internal signals and output pins, the Quartus II LAI enables you to map many internal signals to a smaller number of output pins. The exact number of internal signals that you can map to an output pin varies based on the multiplexer settings in the Quartus II LAI.

This chapter details the following topics:

- “Choosing a Logic Analyzer”
- “Debugging Your Design Using the Logic Analyzer Interface” on page 16–3
- “Advanced Features” on page 16–10

This chapter’s use of the term “logic analyzer” includes both logic analyzers and oscilloscopes equipped with digital channels, commonly referred to as mixed signal analyzers or MSOs.

Choosing a Logic Analyzer

The Quartus II software offers the following two general purpose on-chip debugging tools for debugging a large set of RTL signals from your design:

- The SignalTap® II Logic Analyzer
- An external logic analyzer, which connects to internal signals in your FPGA by using the Quartus II LAI

Table 16–1 describes the advantages of each debugging technology.
The Quartus II software offers a portfolio of on-chip debugging tools. For an overview and comparison of all tools available in the Quartus II software on-chip debugging tool suite, refer to Section V. In-System Debugging in volume 3 of the Quartus II Handbook.

### Required Components

You must have the following components to perform analysis using the Quartus II LAI:

- The Quartus II software starting with version 5.1 and later
- The device under test
- An external logic analyzer
- An Altera® communications cable
- A cable to connect the FPGA to the external logic analyzer

Figure 16–1 shows the LAI and the hardware setup.

<table>
<thead>
<tr>
<th>Feature and Description</th>
<th>Logic Analyzer Interface</th>
<th>SignalTap II Logic Analyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Depth</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>Debugging Timing Issues</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>Performance</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>Triggering Capability</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Use of Output Pins</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>Acquisition Speed</td>
<td>—</td>
<td>✓</td>
</tr>
</tbody>
</table>

You have access to a wider sample depth with an external logic analyzer. In the SignalTap II Logic Analyzer, the maximum sample depth is set to 128 Kb, which is a device constraint. However, with an external logic analyzer, there are no device constraints, providing you a wider sample depth.

Using an external logic analyzer provides you with access to a “timing” mode, which enables you to debug combined streams of data.

You frequently have limited routing resources available to place and route when you use the SignalTap II Logic Analyzer with your design. An external logic analyzer adds minimal logic, which removes resource limits on place-and-route.

The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to external logic analyzers.

Using the SignalTap II Logic Analyzer, no additional output pins are required. Using an external logic analyzer requires the use of additional output pins.

With the SignalTap II Logic Analyzer, you can acquire data at speeds of over 200 MHz. You can achieve the same acquisition speeds with an external logic analyzer; however, you must consider signal integrity issues.
You can use the Quartus II Logic Analyzer Interface (LAI) with the following FPGA device families:

- Arria® GX
- Stratix® series
- Cyclone® series
- MAX® II

FPGA Device Support

Debugging Your Design Using the Logic Analyzer Interface

Figure 16–2 shows the steps you must follow to debug your design with the Quartus II LAI.
Creating an LAI File

The Logic Analyzer Interface (.lai) file defines the interface that builds a connection between internal FPGA signals and your external logic analyzer. Figure 16–3 shows an example of an .lai editor.

To define the Quartus II LAI, you can create a new .lai file or use an existing .lai file.
Creating a New Logic Analyzer Interface File
To create a new .lai file, perform the following steps:

1. In the Quartus II software, on the File menu, click New. The New dialog box displays.
2. Click the Other Files tab.
3. Select Logic Analyzer Interface File.
4. Click OK. The LAI editor appears. The file name is assigned by the Quartus II software (refer to Figure 16–3 on page 16–4). When you save the file, you will be prompted for a file name. Refer to “Saving the External Analyzer Interface File” on page 16–5.

Opening an Existing External Analyzer Interface File
To open an existing .lai file, on the Tools menu, click Logic Analyzer Interface Editor. If no .lai file is enabled for the current project, the editor automatically creates a new .lai file. If an .lai file is currently enabled for the project, that file opens when you select the Logic Analyzer Interface Editor. Alternatively, on the File menu, click Open, and select the .lai file you want to open.

Saving the External Analyzer Interface File
To save the .lai file, perform the following steps:

1. In the Quartus II software, on the File menu, click Save As. The Save As dialog box appears.
2. In the File name box, enter the desired file name.
3. Click Save.

Configuring the Logic Analyzer Interface File Core Parameters
After you have created the .lai file, you must configure the .lai file core parameters.
To configure the .lai file core parameters, from the Setup View list, select Core Parameters. Refer to Figure 16–4.

Figure 16–4. Logic Analyzer Interface File Core Parameters
Table 16–2 lists the .laif file core parameters.

Table 16–2. Logic Analyzer Interface File Core Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin Count</td>
<td>The Pin Count parameter signifies the number of pins you want dedicated to your LAI. The pins must be connected to a debug header on your board. Within the FPGA, each pin is mapped to a user-configurable number of internal signals. The Pin Count parameter can range from 1 to 255 pins.</td>
</tr>
<tr>
<td>Bank Count</td>
<td>The Bank Count parameter signifies the number of internal signals that you want to map to each pin. For example, a Bank Count of 8 implies that you will connect eight internal signals to each pin. The Bank Count parameter can range from 1 to 255 banks.</td>
</tr>
<tr>
<td>Output/Capture Mode</td>
<td>The Output/Capture Mode parameter signifies the type of acquisition you perform. There are two options that you can select:</td>
</tr>
<tr>
<td></td>
<td>Combinational/Timing—This acquisition uses your external logic analyzer’s internal clock to determine when to sample data. Because Combinational/Timing acquisition samples data asynchronously to your FPGA, you must determine the sample frequency you should use to debug and verify your system. This mode is effective if you want to measure timing information, such as channel-to-channel skew. For more information about the sampling frequency and the speeds at which it can run, refer to the data sheet for your external logic analyzer.</td>
</tr>
<tr>
<td></td>
<td>Registered/State—This acquisition uses a signal from your system under test to determine when to sample. Because Registered/State acquisition samples data synchronously with your FPGA, it provides you with a functional view of your FPGA while it is running. This mode is effective when you verify the functionality of your design.</td>
</tr>
<tr>
<td>Clock</td>
<td>The Clock parameter is available only when Output/Capture Mode is set to Registered State. You must specify the sample clock in the Core Parameters view. The sample clock can be any signal in your design. However, for best results, Altera recommends that you use a clock with an operating frequency fast enough to sample the data you would like to acquire.</td>
</tr>
<tr>
<td>Power-Up State</td>
<td>The Power-Up State parameter specifies the power-up state of the pins you have designated for use with the LAI. You have the option of selecting tri-stated for all pins, or selecting a particular bank that you have enabled.</td>
</tr>
</tbody>
</table>

Mapping the Logic Analyzer Interface File Pins to Available I/O Pins

To configure the .laif file I/O pins parameters, select Pins in the Setup View list (Figure 16–5).

Figure 16–5. Logic Analyzer Interface File Pins Parameters

To assign pin locations for the LAI, double-click the Location column next to the reserved pins in the Name column, and the Pin Planner opens.
For information about how to use the Pin Planner, refer to the Pin Planner section in the I/O Management chapter in volume 2 of the Quartus II Handbook.

Mapping Internal Signals to the Logic Analyzer Interface Banks

After you have specified the number of banks to use in the Core Parameters settings page, you must assign internal signals for each bank in the LAI. Click the Setup View arrow and select Bank n or All Banks (Figure 16–6).

**Figure 16–6.** Logic Analyzer Interface Bank Parameters

<table>
<thead>
<tr>
<th>Bank Index</th>
<th>Pin Index</th>
<th>Type</th>
<th>Alias</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>State Clock</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To view all of your bank connections, click Setup View and select All Banks (Figure 16–7).

**Figure 16–7.** Logic Analyzer Interface All Bank Parameters

Using the Node Finder

Before making bank assignments, on the View menu, point to Utility Windows and click Node Finder. Find the signals that you want to acquire, then drag and drop the signals from the Node Finder dialog box into the bank Setup View. When adding signals, use SignalTap II: pre-synthesis for non-incrementally routed instances and SignalTap II: post-fitting for incrementally routed instances.
As you continue to make assignments in the bank Setup View, the schematic of your LAI in the Logical View of your .lai file begins to reflect your assignments (Figure 16–8).

**Figure 16–8.** A Logical View of the Logic Analyzer Interface Schematic

Continue making assignments for each bank in the Setup View until you have added all of the internal signals for which you wish to acquire data.

You can right-click to switch between the LAI schematic and the LAI Setup view.

**Enabling the Logic Analyzer Interface Before Compiling Your Quartus II Project**

Compile your project after you have completed the following steps:

- Configure your LAI parameters
- Map the LAI pins to available I/O pins
- Map the internal signals to the LAI banks

**Compiling Your Quartus II Project**

Before compiling your project, you must enable the LAI. Perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Logic Analyzer Interface**. The **Logic Analyzer Interface** appears.
3. Turn on **Enable Logic Analyzer Interface**.
4. Click **Logic Analyzer Interface file name** and specify the full path name to your .lai file.

After you have specified the name of your .lai file, you must compile your project. To compile your project, on the Processing menu, click **Start Compilation**.

To ensure that the LAI is properly compiled with your project, expand the entity hierarchy in the Project Navigator. (To display the Project Navigator, on the View menu, point to **Utility Windows** and click **Project Navigator**.) If the LAI is compiled with your design, the sld_hub and sld_multitap entities are shown in the project navigator (Figure 16–9).
Chapter 16: In-System Debugging Using External Logic Analyzers

Debugging Your Design Using the Logic Analyzer Interface

Programming Your FPGA Using the Logic Analyzer Interface

After compilation completes, you must configure your FPGA before using the LAI. To configure a device for use with the LAI, perform the following steps:

1. Open the .lai file (Figure 16–10).
2. Under the JTAG Chain Configuration pane, under Hardware, select your hardware communications device. You may have to click Setup to configure your hardware.
3. Click Scan Chain, which interrogates the JTAG chain for devices contained on the chain.
4. Under Device, select the FPGA device you want to download to your design.
5. If desired, turn on Incremental Compilation.
6. Save the .lai file.
7. Click the Program Device icon to program the device.

Using the Logic Analyzer Interface with Multiple Devices

You can use the LAI with multiple devices in your JTAG chain. Your JTAG chain can also consist of devices that do not support the LAI or non-Altera, JTAG-compliant devices. To use the LAI in more than one FPGA, create an LAI and configure an .lai file for each FPGA that you want to analyze. To perform multi-FPGA analysis, perform the following steps:
1. Open the Quartus II software.
2. Create, configure, and compile an .lai file for each design.
3. Open one .lai file at a time.

    ![Image](image)

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

5. Click the Program Device icon to program the device.
6. Control each .lai file independently.

### Configuring Banks in the Logic Analyzer Interface File

When you have programmed your FPGA, you can control which bank is mapped to the reserved .lai file output pins. To control which bank is mapped, in the schematic in the logical view, right-click the bank and click Connect Bank (Figure 16–11).

### Acquiring Data on Your Logic Analyzer

To acquire data on your logic analyzer, you must establish a connection between your device and the external logic analyzer.

For more information about this process and for guidelines about how to establish connections between debugging headers and logic analyzers, refer to the documentation for your logic analyzer.

### Advanced Features

This section describes the following advanced features:

- Using the Logic Analyzer Interface with Incremental Compilation
- Creating Multiple Logic Analyzer Interface Instances in One FPGA
Using the Logic Analyzer Interface with Incremental Compilation

Using the LAI with Incremental Compilation enables you to preserve the synthesis and fitting of your original design, and add the LAI to your design without recompiling your original source code.

To use the LAI with Incremental Compilation, perform the following steps:

1. Start the Quartus II software.
2. Enable design partitions. To enable partitions, perform the following steps:
   a. On the Assignments menu, click Design Partitions Window.
   b. Click Incremental Compilation under Compilation Process Settings in the Settings dialog box, and turn on Full incremental compilation.
   c. To create design partitions for the entities in your design, in the Partition Name column, double-click the cell that contains the <<new>> text. In the Create New Partitions dialog box, click any instance in the hierarchy to designate the instance as a design partition, and then click OK.
   d. Set the Netlist Type to Post-fit.
   e. On the Processing menu, click Start Compilation.
3. Enable LAI Incremental Compilation by performing the following steps:
   a. In your .lai file, under Instance Manager, click Incremental Compilation.
      ❗️ When you enable Incremental Compilation, all existing presynthesis signals are converted into post-fitting signals. Only post-fitting signals can be used with the LAI with Incremental Compilation.
   b. Add Post-Fitting nodes to your .lai file.
   c. On the Processing menu, click Start Compilation.

Creating Multiple Logic Analyzer Interface Instances in One FPGA

The LAI includes support for multiple interfaces in one FPGA. This feature is particularly useful when you want to build LAI configurations that contain different settings. For example, you can build one LAI instance to perform Registered/State analysis and build another instance that performs Combinational/Timing analysis on the same set of signals.

Another example would be performing Registered/State analysis on portions of your design that are in different clock domains.

To create multiple LAs, on the Edit menu, click Create Instance. Alternatively, you can right-click the Instance Manager window and click Create Instance (Figure 16–12).
Conclusion

As the FPGA industry continues to make technological advancements, outdated debugging methodologies must be replaced with new technologies that maximize productivity. The LAI feature enables you to connect many internal signals within your FPGA to an external logic analyzer with the use of a small number of I/O pins. This new technology in the Quartus II software enables you to use feature-rich external logic analyzers to debug your FPGA design, ultimately enabling you to deliver your product in the shortest amount of time.

Referenced Documents

This chapter references the following documents:

- Section V. In-System Debugging in volume 3 of the Quartus II Handbook
- I/O Management chapter in volume 2 of the Quartus II Handbook

Document Revision History

Table 16–3 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>Removed references to APEX devices.</td>
<td>Updated for the Quartus II software version 9.1 release.</td>
</tr>
<tr>
<td></td>
<td>Editorial updates.</td>
<td></td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>Minor editorial updates.</td>
<td>Updated for the Quartus II software version 9.0 release.</td>
</tr>
<tr>
<td></td>
<td>Removed Figures 15–4, 15–5, and 15–11 from 8.1 version.</td>
<td></td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
<td>Updated for the Quartus II software version 8.1 release.</td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>Updated device support list on page 15–3</td>
<td>Updated for the Quartus II software version 8.0 release.</td>
</tr>
<tr>
<td></td>
<td>Added links to referenced documents throughout the chapter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Added “Referenced Documents”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Added reference to Section V. In-System Debugging</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Minor editorial updates</td>
<td></td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.
17. In-System Updating of Memory and Constants

This chapter explains how to use the Quartus II In-System Memory Content Editor as part of your FPGA design and verification flow.

Introduction

The In-System Memory Content Editor allows you to view and update memories and constants with the JTAG port connection.

The In-System Memory Content Editor allows access to dense and complex FPGA designs. When you program devices, you have read and write access to the memories and constants through the Joint Test Action Group (JTAG) interface. You can then identify, test, and resolve issues with your design by testing changes to memory contents in the FPGA while your design is running.

Overview

This chapter contains the following sections:

- “Device Megafuinction Support” on page 17–2
- “Updating Memory and Constants in Your Design” on page 17–2
- “Creating In-System Modifiable Memories and Constants” on page 17–3
- “Running the In-System Memory Content Editor” on page 17–3

When you use the In-System Memory Content Editor in conjunction with the SignalTap® II Embedded Logic Analyzer, you can more easily view and debug your design in the hardware lab.

For more information about the SignalTap II Embedded Logic Analyzer, refer to the Design Debugging Using the SignalTap II Embedded Logic Analyzer chapter in volume 3 of the Quartus II Handbook.

The ability to read data from memories and constants allows you to quickly identify the source of problems. The write capability allows you to bypass functional issues by writing expected data. For example, if a parity bit in your memory is incorrect, you can use the In-System Content Editor to write the correct parity bit values into your RAM, allowing your system to continue functioning. You can also intentionally write incorrect parity bit values into your RAM to check the error handling functionality of your design.

The Quartus II software offers a variety of on-chip debugging tools. For an overview and comparison of all tools available in the Quartus II software on-chip debugging tool suite, refer to Section V. In-System Debugging in volume 3 of the Quartus II Handbook.
Device Megafunction Support

The following tables list the devices and types of memories and constants that are currently supported by the Quartus II software. Table 17–1 lists the types of memory supported by the MegaWizard™ Plug-In Manager and the In-System Memory Content Editor.

Table 17–1. MegaWizard Plug-In Manager Support

<table>
<thead>
<tr>
<th>Installed Plug-Ins Category</th>
<th>Megafuction Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gates</td>
<td>LPM_CONSTANT</td>
</tr>
<tr>
<td>Memory Compiler</td>
<td>RAM: 1-PORT, ROM: 1-PORT</td>
</tr>
<tr>
<td>Storage</td>
<td>ALTSYNCRAM, LPM_RAM_DQ, LPM_ROM</td>
</tr>
</tbody>
</table>

Table 17–2 lists support for in-system updating of memory and constants for the Stratix® series, Arria® GX, and Cyclone® series device families.

Table 17–2. Supported Megafunctions

<table>
<thead>
<tr>
<th>Megafuction</th>
<th>M512 Blocks</th>
<th>M4K Blocks</th>
<th>MegaRAM Blocks</th>
<th>Cyclone Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPM_CONSTANT</td>
<td>Read/Write</td>
<td>Read/Write</td>
<td>Read/Write</td>
<td>Read/Write</td>
</tr>
<tr>
<td>LPM_ROM</td>
<td>Write</td>
<td>Read/Write</td>
<td>N/A</td>
<td>Read/Write</td>
</tr>
<tr>
<td>LPM_RAM_DQ</td>
<td>N/A</td>
<td>Read/Write</td>
<td>Read/Write</td>
<td>Read/Write</td>
</tr>
<tr>
<td>ALTSYNCRAM (ROM)</td>
<td>Write</td>
<td>Read/Write</td>
<td>N/A</td>
<td>Read/Write</td>
</tr>
<tr>
<td>ALTSYNCRAM (Single-Port RAM Mode) (1)</td>
<td>N/A</td>
<td>Read/Write</td>
<td>Read/Write</td>
<td>Read/Write</td>
</tr>
</tbody>
</table>

Note to Table 17–2:

(1) Only write-only mode is applicable for this single-port RAM. In read-only mode, use LPM_ROM instead of LPM_RAM_DQ.

Updating Memory and Constants in Your Design

To use the In-System Updating of Memory and Constants feature, perform the following steps:

1. Identify the memories and constants that you want to access.
2. Edit the memories and constants to be run-time modifiable.
3. Perform a full compilation.
4. Program your device.
5. Launch the In-System Memory Content Editor.
Creating In-System Modifiable Memories and Constants

When you specify that a memory or constant is run-time modifiable, the Quartus II software changes the default implementation. A single-port RAM is converted to dual-port RAM, and a constant is implemented in registers instead of look-up tables (LUTs). These changes enable run-time modification without changing the functionality of your design. For a list of run-time modificable megafuctions, refer to Table 17–1.

To enable your memory or constant to be modifiable, perform the following steps:

1. On the Tools menu, click MegaWizard Plug-In Manager.
2. If you are creating a new megafunction, select Create a new custom megafuction variation. If you have an existing megafunction, select Edit an existing custom megafuction variation.
3. To customize the megafuction based on the characteristics required by your design, turn on Allow In-System Memory Content Editor to capture and update content independently of the system clock, and type a value in the Instance ID text box. These parameters can be found on the last page of the wizard for megafuctions that support in-system updating.

   The Instance ID is a four-character string value used to distinguish the megafuction from other in-system memories and constants.

4. Click Finish.
5. On the Processing menu, click Start Compilation.

If you instantiate a memory or constant megafuction directly with ports and parameters in VHDL or Verilog HDL, add or modify the lpm_hint parameter as follows:

In VHDL code, add the following:

```
lpm_hint => "ENABLE_RUNTIME_MOD = YES,
           INSTANCE_NAME = <instantiation name>";
```

In Verilog HDL code, add the following:

```
defparam <megafunction instance name>.lpm_hint =
   "ENABLE_RUNTIME_MOD = YES,
   INSTANCE_NAME = <instantiation name>";
```

Running the In-System Memory Content Editor

The In-System Memory Content Editor has three separate panes: the Instance Manager, the JTAG Chain Configuration, and the Hex Editor (Figure 17–1).
Running the In-System Memory Content Editor

The **Instance Manager** pane displays all available run-time modifiable memories and constants in your FPGA device. The **JTAG Chain Configuration** pane allows you to program your FPGA and select the Altera® device in the chain to update.

Using the In-System Memory Content Editor does not require that you open a project. The In-System Memory Content Editor retrieves all instances of run-time configurable memories and constants by scanning the JTAG chain and sending a query to the specific device selected in the **JTAG Chain Configuration** pane.

If you have more than one device with in-system configurable memories or constants in a JTAG chain, you can launch multiple In-System Memory Content Editors within the Quartus II software to access the memories and constants in each of the devices. Each In-System Memory Content Editor can access the in-system memories and constants in a single device.

**Instance Manager**

When you scan the JTAG chain to update the **Instance Manager** pane, you can view a list of all run-time modifiable memories and constants in the design. The **Instance Manager** pane displays the Index, Instance, Status, Width, Depth, Type, and Mode of each element in the list.

You can read and write to in-system memory with the **Instance Manager** pane, as shown in Figure 17–2.
Running the In-System Memory Content Editor

The following buttons are provided in the **Instance Manager** pane:

- **Read data from In-System Memory**—Reads the data from the device independently of the system clock and displays the data in the **Hex Editor** pane
- **Continuously Read Data from In-System Memory**—Continuously reads the data asynchronously from the device and displays the data in the **Hex Editor** pane
- **Stop In-System Memory Analysis**—Stops the current read or write operation
- **Write Data to In-System Memory**—Asynchronously writes data present in the **Hex Editor** pane to the device

In addition to the buttons available in the **Instance Manager** pane, you can read and write data by selecting commands from the Processing menu, or the right button pop-up menu in the **Instance Manager** pane or **Hex Editor** pane.

The status of each instance is also displayed beside each entry in the **Instance Manager** pane. The status indicates if the instance is **Not running**, **Offloading data**, or **Updating data**. The health monitor provides information about the status of the editor.

The Quartus II software assigns a different index number to each in-system memory and constant to distinguish between multiple instances of the same memory or constant function. View the **In-System Memory Content Editor Settings** section of the Compilation Report to match an index with the corresponding instance ID (Figure 17–3).
Running the In-System Memory Content Editor

Editing Data Displayed in the Hex Editor Pane

You can edit data read from your in-system memories and constants displayed in the Hex Editor pane by typing values directly into the editor or by importing memory files.

To modify the data displayed in the Hex Editor pane, click a location in the editor and type or paste in the new data. The new data appears in blue, indicating modified data that has not been written into the FPGA. On the Edit menu, choose Value, and click Fill with 0's, Fill with 1's, Fill with Random Values, or Custom Fills to update a block of data that you have selected.

Importing and Exporting Memory Files

The In-System Memory Content Editor allows you to import and export data values for memories that have the In-System Updating feature enabled. Importing from a data file enables you to quickly load an entire memory image. Exporting to a data file enables you to save the contents of the memory for future use.

To import a file to memory with the In-System Memory Content Editor, select the memory or constant that you want to target from the Instance Manager pane. On the Edit menu, click Import Data from File, and specify the data file that you want to load to the targeted memory or constant. You can only import a memory file that is in either a Hexadecimal (Intel-Format) File (.hex) or Memory Initialization File (.mif) format.

Similarly, to export the contents of memory to a file with the In-System Memory Content Editor, select the memory or constant that you want to target from the Instance Manager pane. From the Edit menu, click Export Data from File, and specify the file name to which you want to save the data. You can export data to a .hex, .mif, Verilog Value Change Dump file (.vcd), or RAM Initialization File (.rif) format.

Figure 17–3. Compilation Report In-System Memory Content Editor Settings Section
Viewing Memories and Constants in the Hex Editor Pane

For each instance of an in-system memory or constant, the **Hex Editor** pane displays data in hexadecimal representation and ASCII characters (if the word size is a multiple of 8 bits). The arrangement of the hexadecimal numbers depends on the dimensions of the memory. For example, if the word width is 16 bits, the **Hex Editor** pane displays data in columns of words that contain columns of bytes (Figure 17–4).

Unprintable ASCII characters are represented by a period ( . ). The color of the data changes as you perform reads and writes. Data displayed in black indicates the data in the **Hex Editor** pane was the same as the data read from the device. If the data in the **Hex Editor** pane changes color to red, the data previously shown in the **Hex Editor** pane was different from the data read from the device.

As you analyze the data, you can use the cursor and the status bar to quickly identify the exact location in memory. The status bar is located at the bottom of the In-System Memory Content Editor and displays the selected instance name, word position, and bit offset.

The bit offset is the bit position of the cursor within the word. In the following example, a word is set to 8 bits.

With the cursor in the position shown in Figure 17–5, the word location is 0x0000 and the bit position is 0x0007.

**Figure 17–5.** Hex Editor Cursor Positioned at Bit 0x0007
With the cursor in the position shown in Figure 17–6, the word location remains 0x0000, but the bit position is 0x0003.

**Figure 17–6.** Hex Editor Cursor Positioned at Bit 0×0003

### Scripting Support

The In-System Memory Content Editor supports reading and writing of memory contents via a Tcl script or Tcl commands entered at a command prompt. For detailed information about scripting command options, refer to the Quartus 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.

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*.

The commonly used commands for the In-System Memory Content Editor are as follows:

- **Reading from memory:**
  
  `read_content_from_memory`

  `-content_in_hex`

  `-instance_index <instance index>`

  `-start_address <starting address>`

  `-word_count <word count>`

- **Writing to memory:**

  `write_content_to_memory`

- **Save memory contents to file:**

  `save_content_from_memory_to_file`

- **Update memory contents from File:**

  `update_content_to_memory_from_file`

  For descriptions about the command options and scripting examples, refer to the Tcl API Help Browser and the *Quartus II Scripting Reference Manual*.

### Programming the Device with the In-System Memory Content Editor

If you make changes to your design, you can program the device from within the In-System Memory Content Editor. To program the device, perform the following steps:

1. On the Tools menu, click **In-System Memory Content Editor**.
2. In the JTAG Chain Configuration pane of the In-System Memory Content Editor, select the SRAM object file (.sof) that includes the modifiable memories and constants.

3. Click Scan Chain.

4. In the Device list, select the device you want to program.

5. Click Program Device.

Example: Using the In-System Memory Content Editor with the SignalTap II Embedded Logic Analyzer

The following scenario describes how you can use the In-System Updating of Memory and Constants feature with the SignalTap II Embedded Logic Analyzer to efficiently debug your design in-system. You can use the In-System Content Editor and the SignalTap II Embedded Logic Analyzer simultaneously with the JTAG communication interface.

After completing your FPGA design, you find that the characteristics of your FIR filter design are not as expected.

1. To locate the source of the problem, change all your FIR filter coefficients to be in-system modifiable and instantiate the SignalTap II Embedded Logic Analyzer.

2. Using the SignalTap II Embedded Logic Analyzer to tap and trigger on internal design nodes, you find the FIR filter to be functioning outside of the expected cutoff frequency.

3. Using the In-System Memory Content Editor, you check the correctness of the FIR filter coefficients. Upon reading each coefficient, you discover that one of the coefficients is incorrect.

4. Because your coefficients are in-system modifiable, you update the coefficients with the correct data with the In-System Memory Content Editor.

In this scenario, you can quickly locate the source of the problem using both the In-System Memory Content Editor and the SignalTap II Embedded Logic Analyzer. You can also verify the functionality of your device by changing the coefficient values before modifying the design source files.

You can also modify the coefficients with the In-System Memory Content Editor to vary the characteristics of the FIR filter, for example, filter attenuation, transition bandwidth, cut-off frequency, and windowing function.

Conclusion

The In-System Updating of Memory and Constants feature provides access into a device for efficient debugging in a hardware lab. You can use In-System Updating of Memory and Constants with the SignalTap II Embedded Logic Analyzer to maximize the visibility into an Altera FPGA. By increasing visibility and access to internal logic of the device, you can identify and resolve problems with your design more easily.
Referenced Documents

This chapter references the following documents:

- Command-Line Scripting chapter in volume 2 of the Quartus II Handbook
- Design Debugging Using the SignalTap II Embedded Logic Analyzer chapter in volume 3 of the Quartus II Handbook
- Quartus II Scripting Reference Manual
- Tcl Scripting chapter in volume 2 of the Quartus II Handbook

Document Revision History

Table 17–3 shows the revision history of this chapter.

<table>
<thead>
<tr>
<th>Date and Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>■ Delete references to APEX devices. ■ Style changes.</td>
<td>Updated for the Quartus II software 9.1 release.</td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>No change to content.</td>
<td>Updated for the Quartus II software version 9.0 release.</td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
<td>Updated for the Quartus II software version 8.1 release.</td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>■ Added reference to Section V. In-System Debugging in volume 3 of the Quartus II Handbook on page 16-1. ■ Removed references to the Mercury device, as it is now considered to be a “Mature” device ■ Added links to referenced documents throughout document ■ Minor editorial updates</td>
<td>Updated for the Quartus II software version 8.0 release.</td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.
This chapter provides detailed instructions about how to use the In-System Sources and Probes Editor and Tcl scripting in the Quartus® II software to debug your design.

**Introduction**

Traditional debugging techniques often involve using an external pattern generator to exercise the logic and a logic analyzer to study the output waveforms during run time. The SignalTap® II Logic Analyzer and SignalProbe allow you to read or “tap” internal logic signals during run time as a way to debug your logic design. You can make the debugging cycle more efficient when you can drive any internal signal manually within your design, which allows you to perform the following actions:

- Force the occurrence of trigger conditions set up in the SignalTap II Logic Analyzer
- Create simple test vectors to exercise your design without using external test equipment
- Dynamically control run time control signals with the JTAG chain

The In-System Sources and Probes Editor in the Quartus II software extends the portfolio of verification tools, and allows you to easily control any internal signal and provides you with a completely dynamic debugging environment. Coupled with either the SignalTap II Logic Analyzer or SignalProbe, the In-System Sources and Probes Editor gives you a powerful debugging environment in which to generate stimuli and solicit responses from your logic design.

The Virtual JTAG Megafunction and the In-System Memory Content Editor also give you the capability to drive virtual inputs into your design. The Quartus II software offers a variety of on-chip debugging tools. For an overview and comparison of all the tools available in the Quartus II software on-chip debugging tool suite, refer to *Section V. In-System Debugging* in volume 3 of the *Quartus II Handbook*.

**Overview**

This chapter includes the following topics:

- “Design Flow Using the In-System Sources and Probes Editor” on page 18–3
- “Running the In-System Sources and Probes Editor” on page 18–6
- “Tcl interface for the In-System Sources and Probes Editor” on page 18–9
- “Design Example: Dynamic PLL Reconfiguration” on page 18–12

The In-System Sources and Probes Editor consists of the ALTSOURCE_PROBE megafunction and an interface to control the ALTSOURCE_PROBE megafunction instances during run time. Each ALTSOURCE_PROBE megafunction instance provides you with source output ports and probe input ports, where source ports drive selected signals and probe ports sample selected signals. When you compile
your design, the ALTSOURCE_PROBE megafunction sets up a register chain to either drive or sample the selected nodes in your logic design. During run time, the In-System Sources and Probes Editor uses a JTAG connection to shift data to and from the ALTSOURCE_PROBE megafunction instances. Figure 18–1 shows a block diagram of the components that make up the In-System Sources and Probes Editor.

Figure 18–1. In-System Sources and Probes Editor Block Diagram

The ALTSOURCE_PROBE megafunction hides the detailed transactions between the JTAG controller and the registers instrumented in your design to give you a basic building block for stimulating and probing your design. Additionally, the In-System Sources and Probes Editor provides single-cycle samples and single-cycle writes to selected logic nodes. You can use this feature to input simple virtual stimuli and to capture the current value on instrumented nodes. Because the In-System Sources and Probes Editor gives you access to logic nodes in your design, you can toggle the inputs of low-level components during the debugging process. If used in conjunction with the SignalTap II Logic Analyzer, you can force trigger conditions to help isolate your problem and shorten your debugging process.

The In-System Sources and Probes Editor allows you to easily implement control signals in your design as virtual stimuli. This feature can be especially helpful for prototyping your design, such as in the following operations:

- Creating virtual push buttons
- Creating a virtual front panel to interface with your design
- Emulating external sensor data
- Monitoring and changing run time constants on the fly
The In-System Sources and Probes Editor supports Tcl commands that interface with all your ALTSOURCE_PROBE megafunction instances to increase the level of automation.

**Hardware and Software Requirements**

The following components are required to use the In-System Sources and Probes Editor:

- Quartus II software

  or

- Quartus II Web Edition (with the TalkBack feature turned on)
- Download Cable (USB-Blaster™ download cable or ByteBlaster™ cable)
- Altera® development kit or user design board with a JTAG connection to device under test

The In-System Sources and Probes Editor supports the following device families:

- Arria® GX
- Stratix® series
- HardCopy® II
- Cyclone® series
- MAX® II

**Design Flow Using the In-System Sources and Probes Editor**

The In-System Sources and Probes Editor supports an RTL flow. Signals that you want to view in the In-System Sources and Probes editor are connected to an instance of the ALTSOURCE_PROBE megafunction. After you compile the design, you can control each ALTSOURCE_PROBE instance via the In-System Sources and Probes Editor pane or via a Tcl interface. The complete design flow is shown in Figure 18–2.
To use the In-System Sources and Probes Editor in your design, you must first instantiate the ALTSOURCE_PROBE megafunction variation file. You can configure the ALTSOURCE_PROBE megafunction with the MegaWizard™ Plug-In Manager. Each source or probe port can be up to 256 bits. You can have up to 128 instances of the ALTSOURCE_PROBE megafunction in your design.

To configure the ALTSOURCE_PROBE megafunction, performing the following steps:

1. On the Tools menu, click **MegaWizard Plug-In Manager**.
2. Select **Create a new custom megafunction variation**.
3. Click **Next**.

**Figure 18–2.** FPGA Design Flow Using the In-System Sources and Probes Editor
4. On page 2a of the MegaWizard Plug-In Manager, make the following selections:
   a. In the Installed Plug-Ins list, expand the JTAG-accessible Extensions folder and select In-System Sources and Probes.
      - Verify that the currently selected device family matches the device you are targeting.
   b. Select an output file type and enter the name of the ALTSOURCE_PROBE megafunction. You can choose AHDL (.tdf), VHDL (.vhd), or Verilog HDL (.v) as the output file type.

5. Click Next.

6. On page 3 of the MegaWizard Plug-In Manager, make the following selections:
   a. Under Do you want to specify an Instance Index?, turn on Yes.
   b. Specify the Instance ID of this instance.
   c. Specify the width of the probe port. The width can be from 1 bit to 256 bits.
   d. Specify the width of the source port. The width can be from 1 bit to 256 bits.

7. On page 3 of the MegaWizard Plug-In Manager, you can click Advanced Options and specify other options, including the following:
   - What is the initial value of the source port, in hexadecimal?—Allows you to specify the initial value driven on the source port at run time.
   - Write data to the source port synchronously to the source clock—Allows you to synchronize your source port write transactions with the clock domain of your choice.
   - Create an enable signal for the registered source port—When turned on, creates a clock enable input for the synchronization registers. You can turn on this option only when the Write data to the source port synchronously to the source clock option is turned on.

The In-System Sources and Probes Editor does not support simulation. You must remove the ALTSOURCE_PROBE megafunction instantiation before you create a simulation netlist.

**Instantiating the ALTSOURCE_PROBE Megafunction**

The MegaWizard Plug-In Manager produces the necessary variation file and the instantiation template based on your inputs to the MegaWizard. Use the template to instantiate the ALTSOURCE_PROBE megafunction variation file in your design. The port information is shown in Table 18–1.
Compiling the Design

When you compile your design with the In-System Sources and Probes megafunction instantiated, an instance of the ALTSOURCE_PROBE instance and SLD_HUB megafunsions are added to your compilation hierarchy automatically. These instances provide communication between the JTAG controller and your instrumented logic.

You can modify the number of connections to your design by editing the ALTSOURCE_PROBE megafunction. To open the design instance you want to modify in the MegaWizard Plug-In Manager, double-click the instance in the Project Navigator. You can then modify the connections in the HDL source file. You must recompile your design after you make changes.

You can use the Quartus II incremental compilation feature to reduce compilation time. Incremental compilation allows you to organize your design into logical partitions. During recompilation of a design, incremental compilation preserves the compilation results and performance of unchanged partitions and reduces design iteration time by compiling only modified design partitions.

For more information about 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.

Running the In-System Sources and Probes Editor

The In-System Sources and Probes Editor gives you control over all ALTSOURCE_PROBE megafunsion instances within your design. The editor allows you to view all available run time controllable instances of the ALTSOURCE_PROBE megafunction in your design, provides a push-button interface to drive all your source nodes, and provides a logging feature to store your probe and source data.

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Required?</th>
<th>Direction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>probe[]</td>
<td>No</td>
<td>Input</td>
<td>The outputs from your design.</td>
</tr>
<tr>
<td>source_clk</td>
<td>No</td>
<td>Input</td>
<td>Source Data is written synchronously to this clock. This input is required if you turn on Source Clock in the Advanced Options box in the MegaWizard Plug-In Manager.</td>
</tr>
<tr>
<td>source_ena</td>
<td>No</td>
<td>Input</td>
<td>Clock enable signal for source_clk. This input is required if specified in the Advanced Options box in the MegaWizard Plug-In Manager.</td>
</tr>
<tr>
<td>source[]</td>
<td>No</td>
<td>Output</td>
<td>Used to drive inputs to user design.</td>
</tr>
</tbody>
</table>
To run the In-System Sources and Probes Editor, on the Tools menu, click **In-System Sources and Probes Editor**.

The In-System Sources and Probes Editor contains three panes:

- **JTAG Chain Configuration**—Allows you to specify programming hardware, device, and file settings that the In-System Sources and Probes Editor uses to program and acquire data from a device.
- **Instance Manager**—Displays information about the instances generated when you compile a design, and allows you to control data that the In-System Sources and Probes Editor acquires.
- **In-System Sources and Probes Editor**—Logs all data read from the selected instance and allows you to modify source data that is written to your device.

When you use the In-System Sources and Probes Editor, you do not need to open a Quartus II software project. The In-System Sources and Probes Editor retrieves all instances of the ALTSOURCE_PROBE megafunction by scanning the JTAG chain and sending a query to the device selected in the **JTAG Chain Configuration** pane. You can also use a previously saved configuration to run the In-System Sources and Probes Editor.

Each **In-System Sources and Probes Editor** pane can access the ALTSOURCE_PROBE megafunction instances in a single device. If you have more than one device containing megafunction instances in a JTAG chain, you can launch multiple **In-System Sources and Probes Editor** panes to access the megafunction instances in each device.

### Programming Your Device With JTAG Chain Configuration

After you compile your project, you must configure your FPGA before you use the In-System Sources and Probes Editor. To configure a device to use with the In-System Sources and Probes Editor, perform the following steps:

1. Open the In-System Sources and Probes Editor.
2. In the **JTAG Chain Configuration** pane, point to **Hardware**, and then select the hardware communications device. You may be prompted to configure your hardware; in this case, click **Setup**.
3. From the **Device** list, select the FPGA device to which you want to download the design (the device may be automatically detected). You may need to click **Scan Chain** to detect your target device.
4. In the **JTAG Chain Configuration** pane, click to browse for the SRAM Object File (.sof) that includes the In-System Sources and Probes instance or instances. (The .sof may be automatically detected).
5. Click **Program Device** to program the target device.

### Instance Manager

The **Instance Manager** pane provides a list of all ALTSOURCE_PROBE instances in the design and allows you to configure how data is acquired from or written to those instances.
The following buttons and sub-panes are provided in the Instance Manager pane:

- **Read Probe Data**—Samples the probe data in the selected instance and displays the probe data in the In-System Sources and Probes Editor pane.

- **Continuously Read Probe Data**—Continuously samples the probe data of the selected instance and displays the probe data in the In-System Sources and Probes Editor pane; you can modify the sample rate via the Probe read interval setting.

- **Stop Continuously Reading Probe Data**—Cancels continuous sampling of the probe of the selected instance.

- **Write Source Data**—Writes data to all source nodes of the selected instance.

- **Probe Read Interval**—Displays the sample interval of all the In-System Sources and Probe instances in your design; you can modify the sample interval by clicking Manual.

- **Event Log**—Controls the event log in the In-System Sources and Probes Editor pane.

- **Write Source Data**—Allows you to manually or continuously write data to the system.

  The status of each instance is also displayed beside each entry in the Instance Manager pane. The status indicates if the instance is Not running Offloading data, Updating data, or if an Unexpected JTAG communication error occurs. This status indicator provides information about the sources and probes instances in your design.

### In-System Sources and Probes Editor Pane

The In-System Sources and Probes Editor pane allows you to view data from all sources and probes in your design. The data is organized according to the index number of the instance. The editor provides an easy way to manage your signals, and allows you to rename signals or group them into buses. All data collected from in-system source and probe nodes is recorded in the event log and you can view the data as a timing diagram.

### Reading Probe Data

You can read data by selecting the ALTSOURCE_PROBE instance in the Instance Manager pane and clicking Read Probe Data. This action produces a single sample of the probe data and updates the data column of the selected index in the In-System Sources and Probes Editor pane. You can save the data to an event log by turning on the Save data to event log option in the Instance Manager pane.

If you want to sample data from your probe instance continuously, in the Instance Manager pane, click the instance you want to read, and then click Continuously read probe data. While reading, the status of the active instance shows Unloading. You can read continuously from multiple instances.

You can access read data with the shortcut menus in the Instance Manager pane.
To adjust the probe read interval, in the **Instance Manager** pane, turn on the **Manual** option in the **Probe read interval** sub-pane, and specify the sample rate in the text field next to the **Manual** option. The maximum sample rate depends on your computer setup. The actual sample rate is shown in the **Current interval** box. You can adjust the event log window buffer size in the **Maximum Size** box.

**Writing Data**

To modify the source data you want to write into the ALTSOURCE_PROBE instance, click the name field of the signal you want to change. For buses of signals, you can double-click the data field and type the value you want to drive out to the ALTSOURCE_PROBE instance. The In-System Sources and Probes Editor stores the modified source data values in a temporary buffer. Modified values that are not written out to the ALTSOURCE_PROBE instances appear in red. To update the ALTSOURCE_PROBE instance, highlight the instance in the **Instance Manager** pane and click **Write source data**. The **Write source data** function is also available via the shortcut menus in the **Instance Manager** pane.

The In-System Sources and Probes Editor provides the option to continuously update each ALTSOURCE_PROBE instance. Continuous updating allows any modifications you make to the source data buffer to also write immediately to the ALTSOURCE_PROBE instances. To continuously update the ALTSOURCE_PROBE instances, change the **Write source data** field from **Manually** to **Continuously**.

**Organizing Data**

The **In-System Sources and Probes Editor** pane allows you to group signals into buses, and also allows you to modify the display options of the data buffer.

To create a group of signals, select the node names you want to group, right-click and select **Group**. You can modify the display format in the Bus Display Format and the Bus Bit order shortcut menus.

The **In-System Sources and Probes Editor** pane allows you to rename any signal. To rename a signal, double-click the name of the signal and type the new name.

The event log contains a record of the most recent samples. The buffer size is adjustable up to 128k samples. The time stamp for each sample is logged and is displayed above the event log of the active instance as you move your pointer over the data samples.

You can save the changes that you make and the recorded data to a Sources and Probes File (*.spf). To save changes, on the File menu, click **Save**. The file contains all the modifications you made to the signal groups, as well as the current data event log.

---

**Tcl interface for the In-System Sources and Probes Editor**

To support automation, the In-System Sources and Probes Editor supports the procedures described in this chapter in the form of Tcl commands. The Tcl package for the In-System Sources and Probes Editor is included by default when you run `quartus_stp`.

The Tcl interface for the In-System Sources and Probes Editor provides a powerful platform to help you debug your design. The Tcl interface is especially helpful for debugging designs that require toggling multiple sets of control inputs. You can combine multiple commands with a Tcl script to define a custom command set.
For more information about Tcl scripting, refer to the Tcl Scripting chapter in volume 2 of the Quartus II Handbook. For more information about settings and constraints in the Quartus II software, refer to the Quartus II Settings File Manual. For more information about command-line scripting, refer to the Command-Line Scripting chapter in volume 2 of the Quartus II Handbook.

Table 18–2 shows the Tcl commands you can use instead of the In-System Sources and Probes Editor.

<table>
<thead>
<tr>
<th>Command</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_insystem_source_probe</td>
<td>-device_name &lt;device name&gt;</td>
<td>Opens a handle to a device with the specified hardware. Call this command before starting any transactions.</td>
</tr>
<tr>
<td></td>
<td>-hardware_name &lt;hardware name&gt;</td>
<td></td>
</tr>
<tr>
<td>get_insystem_source_probe_instance_info</td>
<td>-device_name &lt;device name&gt;</td>
<td>Returns a list of all ALTSOURCE_PROBE instances in your design. Each record returned is in the following format: {&lt;instance_index&gt;, &lt;source width&gt;, &lt;probe width&gt;, &lt;instance name&gt;}</td>
</tr>
<tr>
<td></td>
<td>-hardware_name &lt;hardware name&gt;</td>
<td></td>
</tr>
<tr>
<td>read_probe_data</td>
<td>-instance_index &lt;instance_index&gt;</td>
<td>Retrieves the current value of the probe. A string is returned that specifies the status of each probe, with the MSB as the left-most bit.</td>
</tr>
<tr>
<td></td>
<td>-value_in_hex (optional)</td>
<td></td>
</tr>
<tr>
<td>read_source_data</td>
<td>-instance_index &lt;instance_index&gt;</td>
<td>Retrieves the current value of the sources. A string is returned that specifies the status of each source, with the MSB as the left-most bit.</td>
</tr>
<tr>
<td></td>
<td>-value_in_hex (optional)</td>
<td></td>
</tr>
<tr>
<td>write_source_data</td>
<td>-instance_index &lt;instance_index&gt;</td>
<td>Sets the value of the sources. A binary string is sent to the source ports, with the MSB as the left-most bit.</td>
</tr>
<tr>
<td></td>
<td>-value &lt;value&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-value_in_hex (optional)</td>
<td></td>
</tr>
<tr>
<td>end_interactive_probe</td>
<td>None</td>
<td>Releases the JTAG chain. Issue this command when all transactions are finished.</td>
</tr>
</tbody>
</table>

Example 18–1 shows an excerpt from a Tcl script with procedures that control the ALTSOURCE_PROBE instances of the design as shown in Figure 18–3. The example design contains a DCFIFO with ALTSOURCE_PROBE instances to read from and write to the DCFIFO. A set of control muxes are added to the design to control the flow of data to the DCFIFO between the input pins and the ALTSOURCE_PROBE instances. A pulse generator is added to the read request and write request control lines to guarantee a single sample read or write. The ALTSOURCE_PROBE instances, when used with the script in Example 18–1, provide visibility into the contents of the FIFO by performing single sample write and read operations and reporting the state of the full and empty status flags.

Use the Tcl script in debugging situations to either empty or preload the FIFO in your design. For example, you can use this feature to preload the FIFO to match a trigger condition you have set up within the SignalTap II Logic Analyzer.
**Figure 18–3.** A DCFIFO Example Design Controlled by the Tcl Script in Example 18–1

![Diagram of a DCFIFO example design controlled by a Tcl script](image)

**Example 18–1.** Tcl Script Procedures for Reading and Writing to the DCFIFO in Figure 18–3 (Part 1 of 2)

```tcl
## Setup USB hardware - assumes only USB Blaster is installed and
## an FPGA is the only device in the JTAG chain

set usb [lindex [get_hardware_names] 0]
set device_name [lindex [get_device_names -hardware_name $usb] 0]

## write procedure : argument value is integer
proc write {value} {
    global device_name usb
    variable full
    start_insystem_source_probe -device_name $device_name
    write_clock
    #write value
    set full [read_probe_data -instance_index 0]
    if {$full == 1} {end_insystem_source_probe
    return "Write Buffer Full"
}
```
Example 18–1. Tcl Script Procedures for Reading and Writing to the DCFIFO in Figure 18–3 (Part 2 of 2)

```tcl
##toggle select line, drive value onto port, toggle enable
##bits 7:0 of instance 0 is S_data[7:0]; bit 8 = S_write_req;
##bit 9 = Source_write_sel
##int2bits is custom procedure that returns a bitstring from an integer argument

write_source_data -instance_index 0 -value /[int2bits [expr 0x200 | $value]]
write_source_data -instance_index 0 -value [int2bits [expr 0x300 | $value]]

##clear transaction
write_source_data -instance_index 0 -value 0
end_insystem_source_probe
}

proc read {} {
    global device_name usb
    variable empty
    start_insystem_source_probe -device_name $device_name -hardware_name $usb

    ##read empty flag : probe port[7:0] reads FIFO output; bit 8 reads empty_flag
    set empty [read_probe_data -instance_index 1]
    if {[regexp {1........} $empty]} { end_insystem_source_probe
        return "FIFO empty"
    }

    ## toggle select line for read transaction
    ## Source_read_sel = bit 0; s_read_reg = bit 1
    ## pulse read enable on DC FIFO
    write_source_data -instance_index 1 -value 0x1 -value_in_hex
    write_source_data -instance_index 1 -value 0x3 -value_in_hex

    set x [read_probe_data -instance_index 1 ]
    end_insystem_source_probe
    return $x
}
```

**Design Example: Dynamic PLL Reconfiguration**

The In-System Sources and Probes Editor can help you create a virtual front panel during the prototyping phase of your design. You can create relatively simple, high functioning designs of in a short amount of time. The following PLL reconfiguration example demonstrates how to use the In-System Sources and Probes Editor to provide a GUI to dynamically reconfigure a Stratix PLL.
Stratix PLLs allow you to dynamically update PLL coefficients during run time. Each enhanced PLL within the Stratix device contains a register chain that allows you to modify the pre-scale counters \((m\) and \(n\) values), output divide counters, and delay counters. In addition, the ALTPLL_RECONFIG megafuction provides an easy interface to access the register chain counters. The ALTPLL_RECONFIG megafuction provides a cache that contains all modifiable PLL parameters. After you update all the PLL parameters in the cache, the ALTPLL_RECONFIG megafuction drives the PLL register chain to update the PLL with the updated parameters. Figure 18–4 shows a Stratix-enhanced PLL with reconfigurable coefficients.

Stratix II and Stratix III devices also allow you to dynamically reconfigure PLL parameters. For more information about these families, refer to the appropriate data sheet. For more information about dynamic PLL reconfiguration, refer to AN 282: Implementing PLL Reconfiguration in Stratix & Stratix GX Devices or AN 367: Implementing PLL Reconfiguration in Stratix II Devices.

The following design example uses an ALTSOURCE_PROBE instance to update the PLL parameters in the ALTPLL_RECONFIG megafuction cache. The ALTPLL_RECONFIG megafuction connects to an enhanced PLL in a Stratix FPGA to drive the register chain containing the PLL reconfigurable coefficients. This design example uses a Tcl/Tk script to generate a GUI where you can enter in new \(m\) and \(n\) values for the enhanced PLL. The Tcl script extracts the \(m\) and \(n\) values from the GUI,
shifts the values out to the ALTSOURCE_PROBE instances to update the values in the ALTPLL_RECONFIG megafunction cache, and asserts the reconfiguration signal on the ALTPLL_RECONFIG megafunction. The reconfiguration signal on the ALTPLL_RECONFIG megafunction starts the register chain transaction to update all PLL reconfigurable coefficients. A block diagram of a design example is shown in Figure 18–5. The Tk GUI is shown in Figure 18–6.

**Figure 18–5.** Block Diagram of Dynamic PLL Reconfiguration Design Example

![Block Diagram of Dynamic PLL Reconfiguration Design Example](image)

**Figure 18–6.** Interactive PLL Reconfiguration GUI Created with Tk and In-System Sources and Probes Tcl Package

![Interactive PLL Reconfiguration GUI](image)

This design example was created using a Nios® II Development Kit, Stratix Edition. The file `sourceprobe_DE_dynamic_pll.zip` contains all the necessary files for running this design example, including the following:

- **Readme.txt**—A text file that describes the files contained in the design example and provides instructions about running the Tk GUI shown in Figure 18–6.

- **Interactive_Reconfig.qar**—The archived Quartus II project for this design example.

Download the `sourceprobe_DE_dynamic_pll.zip` file from the Literature: Quartus II Handbook page of the Altera website.
Conclusion

The In-System Sources and Probes Editor provides stimuli and receives responses from the target design during run time. With the simple and intuitive interface, you can add virtual inputs to your design during run time without using external equipment. When used in conjunction with the SignalTap II Logic Analyzer, you can use the In-System Sources and Probes Editor to obtain greater control of the signals in your design, and thus help shorten the verification cycle.

Referenced Documents

This chapter references the following documents:

- Command-Line Scripting chapter in volume 2 of the Quartus II Handbook
- Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook
- Quartus II Settings File Manual
- Section V. In-System Debugging in volume 3 of the Quartus II Handbook
- Tcl Scripting chapter in volume 2 of the Quartus II Handbook

Document Revision History

Table 18–3 shows the revision history for this chapter.

Table 18–3. Document Revision History

<table>
<thead>
<tr>
<th>Date and Document Version</th>
<th>Changes Made</th>
<th>Summary of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2009 v9.1.0</td>
<td>Removed references to obsolete devices. Style changes.</td>
<td>Updated for the Quartus II software version 9.1 release.</td>
</tr>
<tr>
<td>March 2009 v9.0.0</td>
<td>No change to content.</td>
<td></td>
</tr>
<tr>
<td>November 2008 v8.1.0</td>
<td>Changed to 8-1/2 x 11 page size. No change to content.</td>
<td>Updated for the Quartus II software version 8.1 release.</td>
</tr>
<tr>
<td>May 2008 v8.0.0</td>
<td>Documented that this feature does not support simulation on page 17–5</td>
<td>Updated for the Quartus II software version 8.0 release.</td>
</tr>
<tr>
<td></td>
<td>Updated Figure 17–8 for Interactive PLL reconfiguration manager</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Added hyperlinks to referenced documents throughout the chapter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Minor editorial updates</td>
<td></td>
</tr>
</tbody>
</table>

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