Verification Continuum™

Identify®
Instrumentor and Debugger
for Microchip User Guide

June 2021

SYNOPSYS®

Copyright Notice and Proprietary Information

© 2021 Synopsys, Inc. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at
http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 East Middlefield Road
Mountain View, CA 94043
www.synopsys.com

June 2021

This section contains the following topics:

The Design Flow

Instrumenting the Design

Setting up and Running Debug

Debugging Using FPGA Memory

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

Contents

Chapter 3: Setting up and Running Debug

Chapter 4: Debugging Using FPGA Memory

CHAPTER 1

The Design Flow

The Synopsys® Identify® software includes functionality for instrumentation, and supports various debug schemes. This chapter introduces the Identify product and the FPGA synthesis tools with which it is integrated. See the following topics for details:

- About Design Verification, on page 10
- Identify Instrumentor and Debugger, on page 10
- Identify Design Flow, on page 11
- Launching the Instrumentor Tool, on page 13
- Launching the Identify Tool, on page 15

About Design Verification

As designs get larger and errors more expensive, verification is an increasingly important part of the design cycle, in terms of both time and necessity. There are various verification methodologies, from simulation to emulation to formal verification.

The Synopsys® Identify® product is a verification tool that offers a way to debug the HDL. HDL debugging ensures functional correctness that a post-synthesis debugger cannot guarantee. This is because synthesis optimizations cause gate-level netlists to significantly differ from the functional HDL description, making it hard to trace bugs back to the original HDL. The Identify software runs debug on hardware.

Identify Instrumentor and Debugger

The Identify tool set allows you to debug an operating FPGA directly in the HDL source code. Using the tool, you can verify your design in hardware as you would in simulation. Unlike simulation, it has in-system stimuli and is much faster.

The Identify tool consists of two sets of functionalities: instrumentation and debug.

- Instrumentation means marking the signals, usually before compiling the design. Designers and verification engineers navigate the design graphically and mark signals as probes or sample triggers directly in the HDL with which they are familiar. Instrumentation gives you the ability to monitor performance and diagnose errors when you get to the debug stage.

- Debugging is a way to verify the FPGA design by running it on hardware. The identify tool supports various debug strategies.

  You can use the hardware platforms from the Synopsys HAPS® prototyping products for debug. You can also use the Identify functionality with third-party hardwares or on your own boards. The Identify documentation does not cover these custom design flows.

Identify Design Flow

The Identify software can be run standalone, but is also integrated into a seamless development environment with these Synopsys FPGA synthesis tools: Synplify Pro® and Certify®. The following figure provides an overview of the use model.
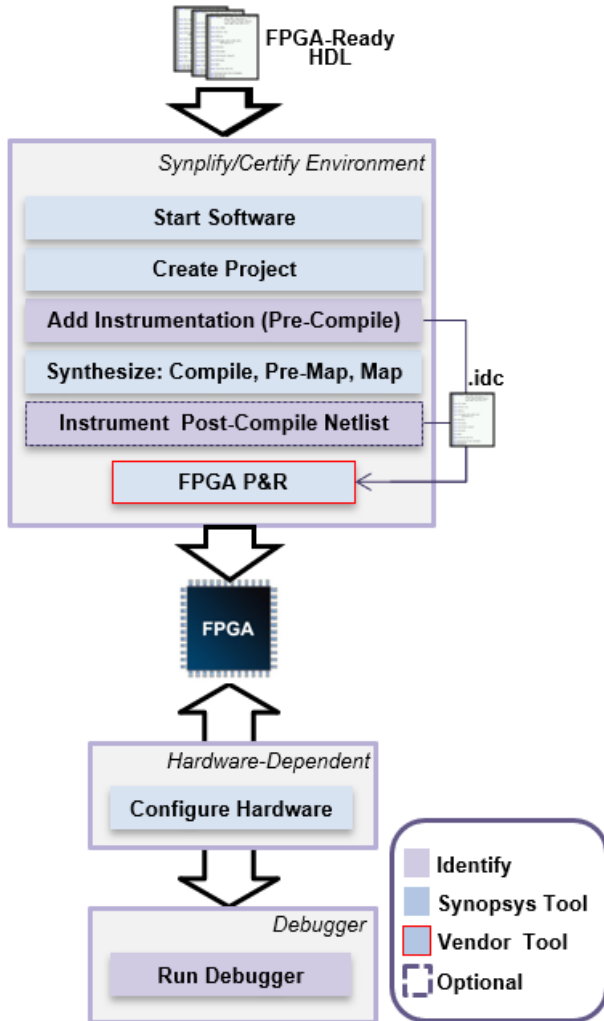
The identify design flow is described in the following steps.

1. Start the instrumentor, and specify the signals to be monitored.

   After instrumentation, the tool generates an instrumentation design constraints (idc) file that contains the instrumented signals. For information about starting the instrumentor and specifying signals, see *Launching the Instrumentor Tool,* on page 13 and *Adding Instrumentation,* on page 28, respectively.

2. Synthesize, place, and route the design as usual.

   If you need to iterate, use incremental runs. After synthesis, you can view the results in the HDL source code or in the waveform viewer.

3. Program the hardware with the instrumented design.

4. Run the debugger.

   Launch the debugger to analyze the design while it is running in the target system. The debugger interacts with the instrumented HDL design that is implemented on the target hardware system. You can activate the marked signals to cause trigger events on the target device, and the triggers capture signal data.

   For information about running the debugger, see *Launching the Identify Tool,* on page 15.

5. Analyze the captured data.

   The debugger transfers the data through a communications port to where it can be displayed in various formats.

Launching the Instrumentor Tool

User can launch the instrumentor in three different methods.

Launching from the Synthesis Tool GUI

To launch the Instrumentor from the synthesis tool:

- From the Synplify Pro GUI, highlight the Identify implementation and select Run > Identify Instrumentor from the menu bar or pop-up menu, or click the Identify Instrumentor icon in the menu bar.

- From the tool, select Run > Identify Instrumentor from the menu bar or click the Identify Instrumentor icon in the top menu bar.

   The identify instrumentor window is displayed, as shown below.
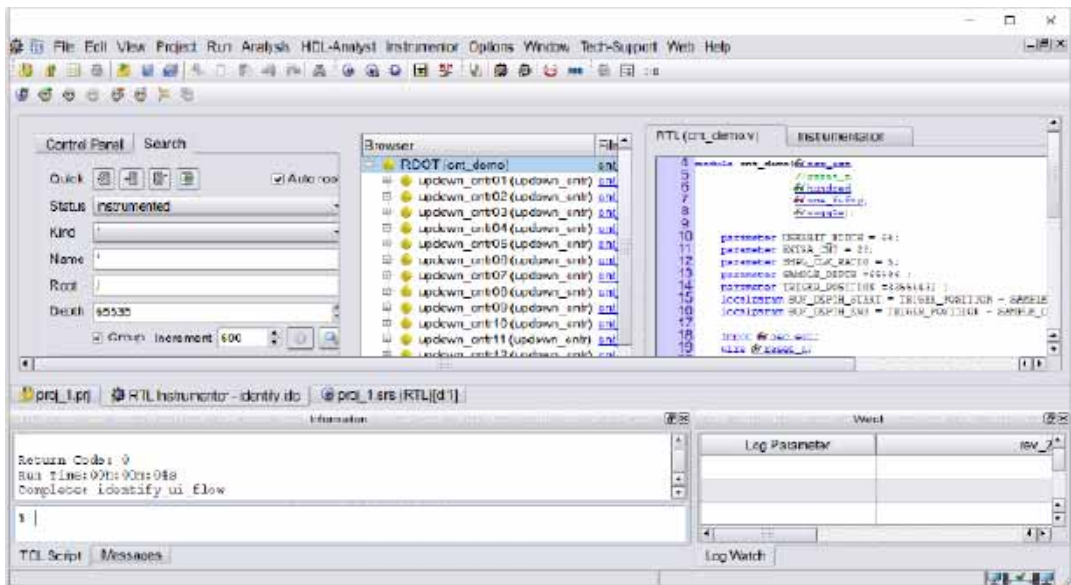
On launching the instrumentor tool, the design hierarchy and the RTL file content with all the potential instrumentation marked and available for selection are displayed.

Launching with a Tcl Command or in Batch Mode

The instrumentor tool can be launched in any of the three execution modes as outlined below.

To open the instrumentor in the GUI:

- `identify_instrumentor`

To run a Tcl startup file and open the instrumentor in the graphical user interface:

- `identify_instrumentor -f fileName.tcl`

To open the instrumentor in the shell and/or script mode:

- `identify_instrumentor_shell [-version]`

If the optional -version argument is included, the above command displays the software version without opening the instrumentor.

Invoking the Tool from the Operating System

The instrumentor runs on both the Windows and Linux platforms. To explicitly invoke the instrumentor from a Windows system, either:

- Double-click the Identify Instrumentor icon on the desktop

- Run identify_instrumentor.exe from the /bin directory of the installation path

To explicitly invoke the debugger from a Linux system:

- Run identify_instrumentor from the /bin directory of the installation path

Launching the Identify Tool

User can launch the Identify tool in three various methods.

- Launching from the Synthesis Tool GUI, on page 15

- Launching with a Tcl Command or in Batch Mode, on page 16

- Invoking the Tool from the Operating System, on page 16

Launching from the Synthesis Tool GUI

To open the Identify from the synthesis tool:

- From the tool, highlight the Identify implementation and select Run > Launch Identify Debugger from the menu bar or pop-up menu, or click the Launch Identify Debugger icon in the menu bar.

- From the tool, select Run > Launch Identify Debugger from the menu bar or click the Launch Identify Debugger icon in the top menu bar.

  The debugger window opens with the corresponding project displayed.
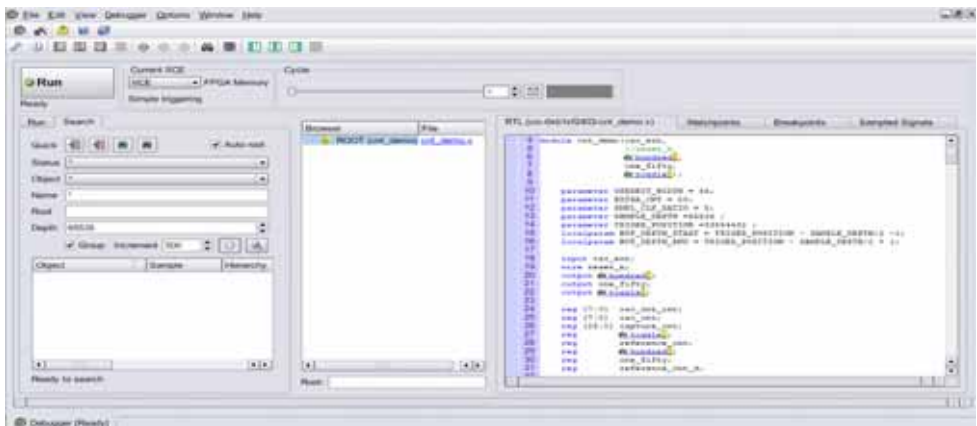
To open the project in the debugger tool, do either of the following:

- Click the Browse for Project button, navigate to the project directory and open the corresponding project (.prj) file.

- Click Close to navigate to the Debugger window, select File > Open Debugger Project from the main menu and, in the Open Project File dialog

box, click the Browse for Project.. button, navigate to the project directory and open the corresponding project (prj) file.
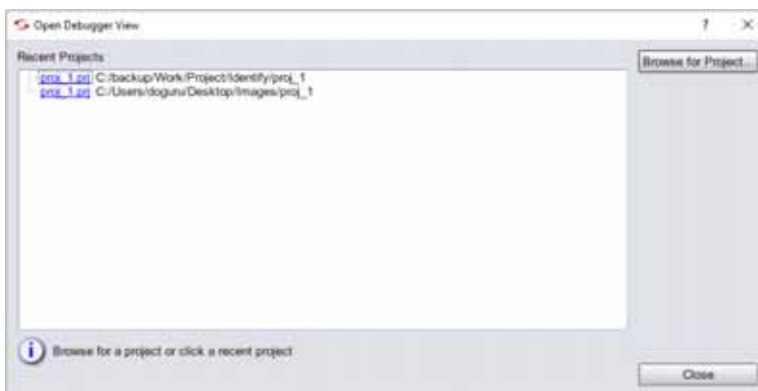
Launching with a Tcl Command or in Batch Mode

You can start the tool with a Tcl command, that can also be used in batch mode:

To open the debugger in the GUI.

- identify_debugger

To run the debugger from a Tcl startup file that opens the tool in the graphical user interface:

- identify_debugger -f *fileName*.tcl

To start the debugger in shell and/or script mode:

- identify_debugger_shell [-version]

If the optional -version argument is included, the above command reports the software version without opening the tool.

Invoking the Tool from the Operating System

The identify tool can be invoked on both the Windows and Linux platforms.

To explicitly invoke the tool from a Windows operating system, do one of the following:

- Double-click the Identify Debugger icon on the desktop
- Run identify_debugger.exe from the /bin directory of the installation path

To explicitly invoke the tool from a Linux operating system:

- Run identify_debugger from the /bin directory of the installation path.

CHAPTER 2

Instrumenting the Design

The first step to debug is to instrument the design, which prepares the design for debugging. Instrumentation consists of adding specific logic, which is used to run on-chip debugging, after programming the FPGA. It is best to instrument early and reserve the resources needed for instrumentation, instead of doing it after the design is in place.

Incorporating instrumentation as part of the initial phase reduces turnaround time. Instrumentation allows you to pipe clean the flow and identify logic, memory, timing, and other limitations early in the cycle.

See the following topics for details:

The Instrumentation Flow

To create an instrumented design, you must first complete the following steps:

1. Specify IICE parameters

2. Select signals to sample

3. Select breakpoints to instrument

4. Optionally, include the original HDL source

The following image illustrates the instrumentation flow.
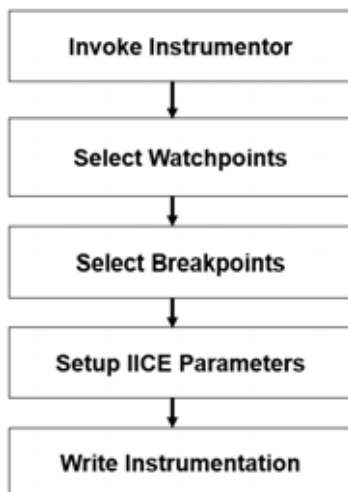
To include the original HDL source with the exported design files, select Instru-mentor > Instrumentor Preferences and enable the Save original source in instrumentation directory check box. If the original source is to be encrypted, additionally enable the Use encryption check box.
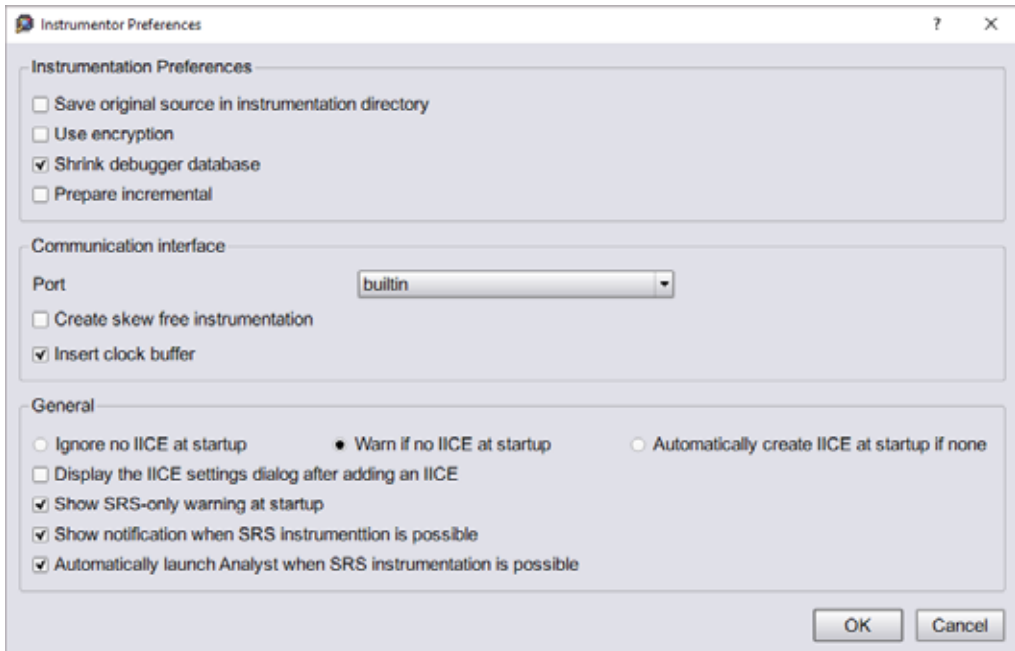
Finally, select the File > Save from the main menu to capture your instrumentation.

Saving the instrumentation generates an *instrumentation design constraints* (idc) file or IICE file and adds compiler pragmas in the form of constraint files to the design RTL for the instrumented signals and break points. This information is then used by the synthesis tool to incorporate the instrumentation logic (IICE and COMM blocks) into the synthesized netlist. If you include an encrypted HDL source (Use encryption box checked), you are first prompted to supply a password for the encryption. See the *Debug Environment Reference Manual*.

Planning Instrumentation and Debugging

Recommendations to instrument the design are provided below:

- Proactively plan for debug visibility
  - Plan for debug when you plan the clocks, resets, and I/Os. In particular, consider the clock structure and how it maps to the global clocks, Mixed-Mod Clock Manager (MMCM)s and Phase Lock Loop (PLL)s for clock control and synchronization, and reset synchronization and control.
  - Plan for debug with incremental logic changes to an existing design, as and when a new IP is added.

- Mark signals for debug
  - Create separate .idc files for each block
  - Use multiple IICEs to debug different clock domains
  - Group signals in each IICE using mux groups, and selectively debug

- Consider memory depth
  - For designs that require a high sample clock (for example, interface IP) consider using BRAM or real time debug (RTD). BRAM uses block RAM in the FPGA, so is best suited for shallow sample windows. Use RTD with an external logic analyzer, if you need a much larger sample window.

Instrumenting the Design

You can add instrumentation to a pre-compiled signal or to a netlist that has already been compiled, as shown below:
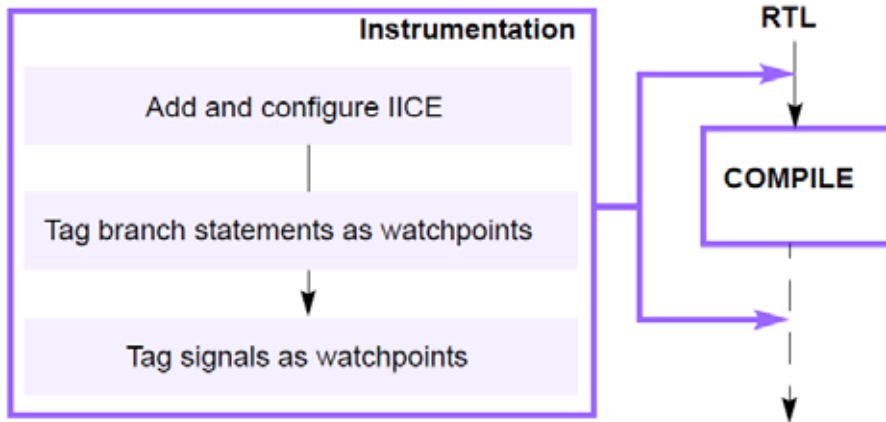
See the following topics:

Instrumenting Signals Before Compile

There are pros and cons to adding debug points to the design before compiling, as opposed to after compiling. Marking signals before compiling offers more visibility into the design and lets you implement complex trigger and muxing options. However, some logic, like generate statements, might not be visible, and the design and the available resources are probably not stable. Further, adding debug points at this stage modifies the HDL, because it requires that extra logic to be created for debugging.

The following procedure is an overview of how to add debug points to a design before it is compiled.

1. Set up the design.

   – Add the required source files to the design.

- Create a new Identify Implementation. See the *Synthesis FPGA User Guide* for details.

2. Click Identify Instrumentor or open the instrumentor GUI by typing the following command:

    identify_instrumentor

3. In the Instrumentor GUI, instrument the RTL design by setting watchpoints and breakpoints that you want to trigger and sample when you debug the design.

    For few guidelines on instrumenting the design and debug planning, see *Instrumenting a Netlist After Compile,* on page 25.
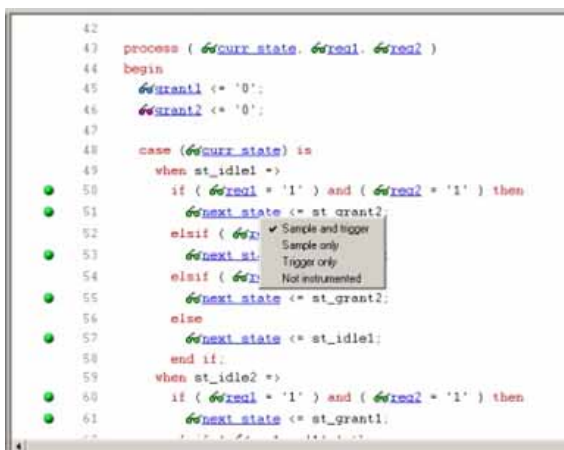
- Add Intelligent IICE and communication blocks for probe and communication logic to trigger and sample the design. You can add multiple IICEs to handle multiple clock domains. See *Adding IICE,* on page 41.

4. Save the design by clicking the Save/Save All button from the main menu.

    The tool writes out an idc file with information about the instrumented design. You can open this file on subsequent runs with the edit idc command.

    See the scripts provided with the tool for examples of simple scripts you can use to automate instrumentation tasks. Also refer to the example in *Adding Instrumentation,* on page 28.

5. Set pre-configure and pre-arm triggers.

6. Click the Run button to compile the instrumented design or use the following command.

   project -run compile

   The project -run compile command automatically includes information from the idc file when it compiles the design.

7. Analyze the instrumented design.

   From the GUI, click the RTL view icon to view the data schematic by clicking the RTL view icon (⊕) in the GUI or using the view schematic command, which allows you trace nodes back to the original RTL.
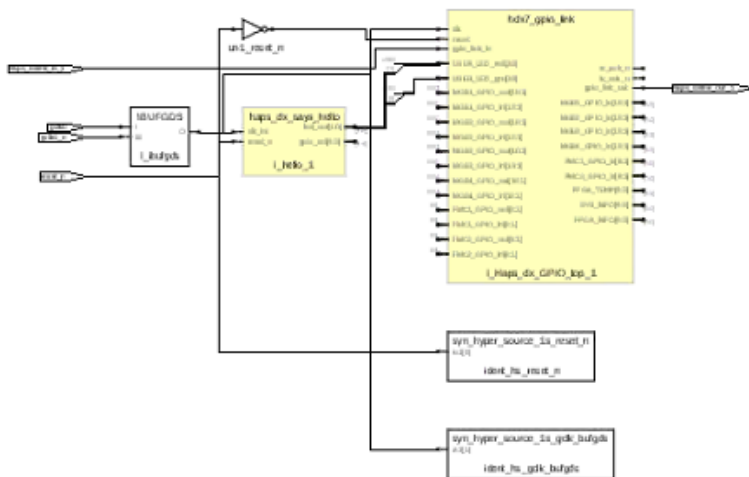
You can also instrument signals in the schematic. The schematic instrumentation can be in addition to the RTL-based instrumentation that was done earlier, or as an alternative to it.

See *Instrumenting a Netlist After Compile,* on page 25 for information on editing and adding instrumentation to a compiled netlist.

Instrumenting a Netlist After Compile

Typically, you instrument a design before it is compiled, as described in *Instrumenting Signals Before Compile,* on page 23. This section provides an overview of post-compile instrumentation, when you start with a compiled netlist rather than the RTL source files.

This approach offers slightly less visibility into the design than pre-compile instrumentation, but the design is at a more stable stage, with the RTL elaborated. You can use complex trigger and muxing options for the instrumentation. The downside to post-compile instrumentation is that some compiler optimizations might affect the observability and affect mapping to RTL. You could create a script to check post-compile signals against the RTL to instrument the design and flag mismatches.

With a compiled netlist, you can instrument the signals directly in the compiled netlist file, outside the instrumentor. This allows you to update instrumentation that was inserted previously. It also allows you to instrument signals within a parameterized module, which were unavailable for instrumentation before compilation.

1. Instrument a design and compile it by clicking the Run button or using the project -run compile command.

2. Click the RTL view icon to view the schematic of the compiled project.

3. Instrument the signals you want from the schematic.

   – Select the signal you want to instrument or update.

   – Right-click the signal, and set the type of instrumentation you want by selecting Instrumentor from the pop-up menu, and selecting the kind of sample or trigger instrumentation you want to use.
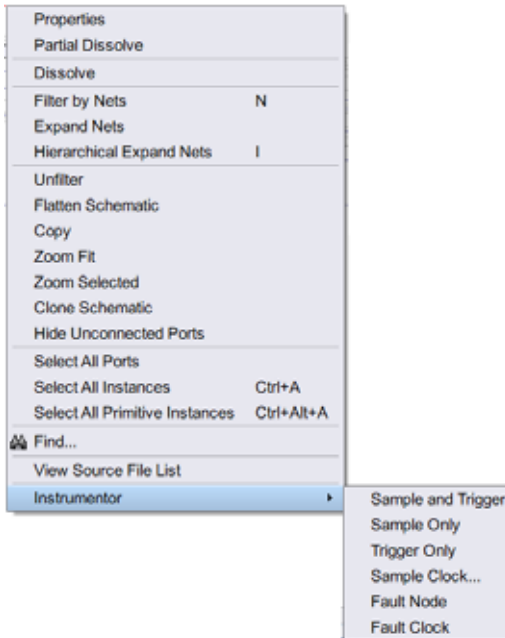
4. Add the instrumented signal to the idc file.

   –   Paste the signal string into the idc file. You can create a new idc file or
       update an existing one. If you are creating a new file, you must add
       the IICE definition shown on lines 2-4 in the figure below (iice new, iice
       controller and iice sampler commands for defining a new IICE,
       configuring the controller, and setting IICE sampler options
       respectively).

       The figure shows the signal on line 6 pasted into the idc file as a
       sample-only signal.

```
1 device jtagport builtin
2 iice new {IICE} -type regular
3 iice controller -iice {IICE} none
4 iice sampler -iice {IICE} -depth 128
5 iice clock -iice {IICE} -edge positive {/SRS/clk}
6 signals add -iice {IICE} -silent -sample {/SRS/d}
7 signals add -iice {IICE} -silent -sample -trigger {/SRS/q}
```

   –   Save the edited idc file.

5. Run pre-map. This will re-run the entire synthesis flow till map.

project -run synthesis

6. Continue with place and route of the design.

7. Set up the design and start the debugger.

Adding Instrumentation

The following sections describe basic procedures to add instrumentation.

Selecting Signals for Data Sampling

To select a signal to be sampled, follow these steps:

1. In the RTL tab, click the watchpoint icon next to the signal name.

2. Select the signals for sampling, triggering, or both from the pop-up menu.
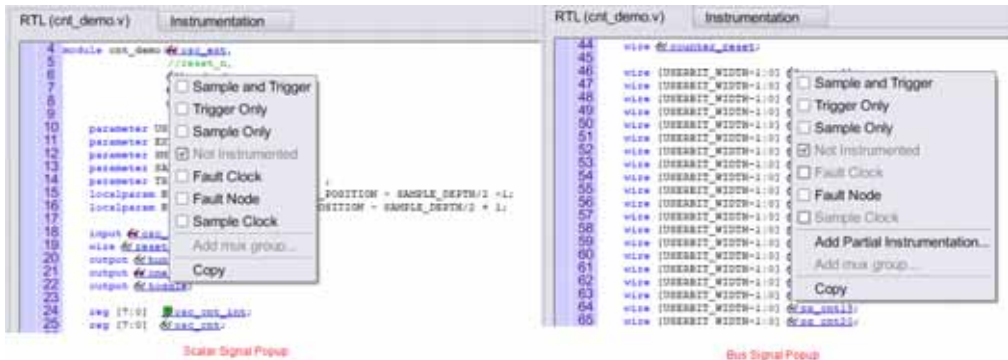
When the watchpoint icon is clear (unfilled), the signal has not been
instrumented. The colors of the filled icons are described in the following
table:

🍒	Red	Signal is enabled for triggering only
🍒	Green	Signal is enabled for both sampling and triggering
🍒	Blue	Signal is enabled for sampling only

You can use Find to recursively search for signals and then instrument
selected signals directly from the Find dialog box (see *Capturing
Commands from the Tcl Script Window,* on page 40).

– Never instrument the following input and output buffer signals, as
   they cause an error in the synthesis tool during subsequent mapping:
   input of IBUF or IBUFG, and output of OBUF or OBUFT. These
   signals either drive or are driven by user logic.

– To control the overhead for the trigger logic, always instrument
   signals that are not needed for triggering with the Sample only selection
   (the watchpoint icon is blue for sample-only signals).

– Specify a qualified clock signals as the sample clock (see the *Debug
   Environment Reference Manual*). You can also specify bus segments
   individually (see *Instrumenting Buses,* on page 30). In addition,
   signals specified as Sample and trigger or Sample only can be included in
   multiplexed groups as described in *Adding Multiplexed Groups,* on
   page 34.

The example below shows how signal grant1 is enabled for sample and
trigger.
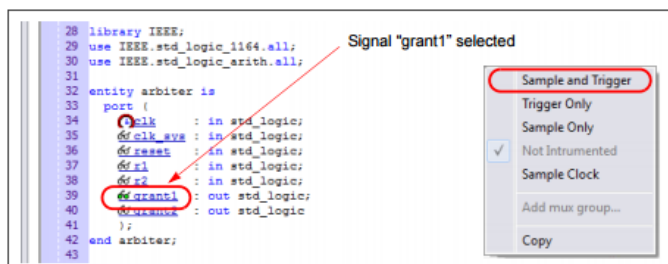
The TCL Script window at the bottom displays the Tcl command that implements the selection and the results of executing the command.

signals add -iice {IICE} -sample -trigger {/beh/arb_inst/grant1}

```
signals add -iice {IICE} -sample -trigger {/beh/arb_inst/grant1}
  added for sampling and triggering to iice: IICE
     Total instrumentation in bits: Sample Only 27, Trigger Only 5, Sample and trigger 13
     Group wise instrumentation in bits:
          Groupdefault:Sample Only 27, Sample and trigger 13
```

To disable a signal for sampling or triggering, select the signal from the RTL tab and then select Not instrumented from the popup menu; the watchpoint icon will again be clear (unfilled).

Instrumenting Buses

Entire buses, individual bits, or groups of bits of a bus can be individually instrumented.

- Instrumenting a Partial Bus, on page 30
- Instrumenting Single Bits of a Bus, on page 31
- Instrumenting Non-Contiguous Bits or Bit Ranges, on page 32
- Changing the Instrumentation Type, on page 32

Instrumenting a Partial Bus

To instrument a sequence (range) of bits of a bus:

1. Place the cursor over a bus that is not fully instrumented and select Add Partial Instrumentation. The following dialog box is displayed.
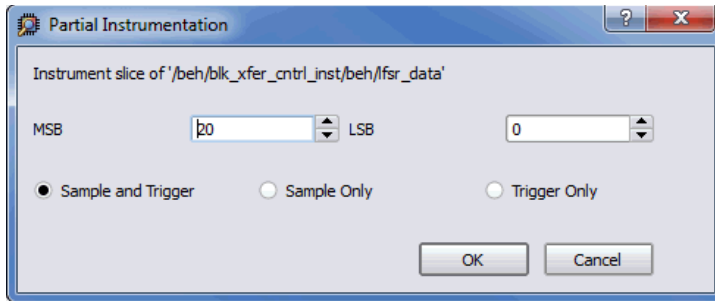
2. In the dialog box, enter the most- and least-significant bits in the MSB and LSB fields.

   Note that the bit range specified is contiguous; to instrument non-contiguous bit ranges, see the section, *Instrumenting Non-Contiguous Bits or Bit Ranges,* on page 32.

   When specifying the MSB and LSB values, the index order of the bus must be followed. For example, when defining a partial bus range for bus [63:0] (or "63 downto 0"), the MSB value must be greater than the LSB value. Similarly, for bus [0:63] (or "0 upto 63"), the MSB value must be less that the LSB value.

3. Select the type of instrumentation for the specified bit range from the radio buttons and click OK.

   When you click OK, a large letter "P" is displayed to the left of the bus name in place of the watchpoint icon. The color of this letter indicates if the partial bus is enabled for triggering only (red), for sampling only (blue), or for both sampling and triggering (green).

Instrumenting Single Bits of a Bus

To instrument a single bit of a bus:

   1. Enter the bit value in the MSB field of the Add partial bus dialog box.

   2. Leave the LSB field blank.

3. Select the instrumentation type.

Instrumenting Non-Contiguous Bits or Bit Ranges

To instrument non-contiguous bits or bit ranges:

1. Instrument the first bit range or bit. See *Instrumenting a Partial Bus,* on page 30.

2. Re-position the cursor over the bus, right-click and select Add partial instrumentation to redisplay the Add partial bus dialog box.

   The previously instrumented bit or bit range is now displayed.

/beh/blk_xfer_cntrl_inst/adr_o_7_4 ▸
Add Partial Instrumentation...
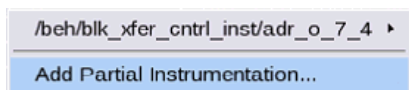
3. Specify the bit or bit range to be instrumented. See *Instrumenting a Partial Bus,* on page 30.

4. Select the type of instrumentation and click OK.

   If the type of instrumentation is different from the existing instrumentation, the letter P will be yellow to indicate a mixture of instrumentation types.

   Bits cannot overlap groups (a bit cannot be instrumented more than once).

Changing the Instrumentation Type

Use the procedure below to change the instrumentation type of a partial bus or to remove the instrumentation from a bit or bit range.

1. Right-click on the bit.

2. Highlight the bit range or bit to be changed and select the new instrumentation type from the menu.
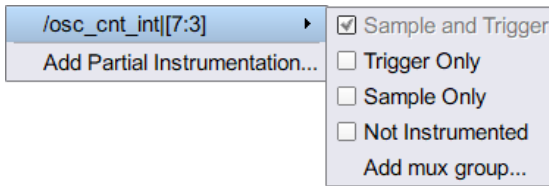
3. To remove instrumentation from a bit or bit range, select Not Instrumented.

Adding Partial Instrumentation

Partial instrumentation allows fields within a record or a structure to be individually instrumented.

1. Select a compatible signal for instrumentation, either on the RTL tab or through the Instrumentor Search dialog box.

   Partial instrumentation can only be added to a field or record one slice level down in the signal hierarchy.

2. Right-click on the signal and select Add Partial Instrumentation.

3. Enter the most- and least-significant bits in the MSB and LSB fields and select type of instrumentation.

   When instrumented, the signal has a �P icon in place of the watchpoint (glasses) icon to indicate that portions of the record are instrumented. The P icon is the same icon that is used to show partial instrumentation of a bus and uses a similar color coding:

Instance	Color
All instances sample only	Blue
All instances trigger only	Red
All instances sample and trigger	Green
All instances in any combination	Yellow

   The figure below shows the partial instrumentation icon on signal tt. The yellow color indicates that the individual fields (tt.r2 and tt.c2) are assigned different types of instrumentation.

```
31
32  signal Ptt: matrix_element1;
33  begin
34    Ptt.r2 <= 66r2;
35    Ptt.c2 <= 66c2;
36    Ptt.ele.r1 <= 66r1;
37    Ptt.ele.c1 <= 66c1;
38
```

The Search Panel also uses the partial instrumentation icon to show the state of instrumentation on fields of partially instrumented records (see *Capturing Commands from the Tcl Script Window,* on page 40).

Adding Multiplexed Groups

Only signals or buses that are instrumented as either Sample and Trigger or Sample only can be added to a multiplexed group.

1. To create multiplexed groups, right-click on each individual instrumented signal or bus and select Add mux group from the pop-up menu. You can also use the signals group command to assign groups from the console window (see *signals* in the *Debug Environment Reference Manual*).

   Command options allow more than one instrumented signal to be assigned in a single operation and allow the resultant group assignments to be displayed.

```
☑ Sample and Trigger
☐ Trigger Only
☐ Sample Only
☐ Not Instrumented
☐ Fault Clock
☐ Fault Node
☐ Sample Clock
  Add Partial Instrumentation...
  Add mux group...
  Copy
```
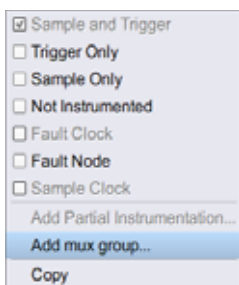
2. In the Add mux group dialog box displayed, select a corresponding group by checking the group number.

3. Click OK to assign to the signal or bus to that group.

A signal can be included in more than one group by checking additional group numbers.
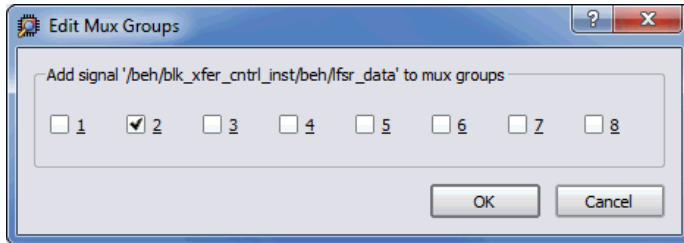
When assigning instrumented signals to groups:

- A maximum of eight groups can be defined; signals can be included in more than one group, but only one group can be active in the debugger at any one time.

- Signals instrumented as Sample Clock, Trigger only, or Partial Buses cannot be included in multiplexed groups.

For information on using multiplexed groups in the debugger, see *Using Mux Sets,* on page 119.

Sampling Signals in a Folded Hierarchy

When a design contains entities or modules that are instantiated more than once, it is termed to have a *folded* hierarchy. Folded hierarchies also occur when multiple instances are created within a generate loop. By definition, there will be more than one instance of every signal in a folded entity or module. To allow you to instrument a particular instance of a folded signal, the instrumentor automatically recognizes folded hierarchies and presents a choice of all possible instances of each signal within the hierarchy.

The choices are displayed in terms of an absolute signal path name originating at the top-level entity or module. The list of choices for a particular signal is accessed by clicking the watchpoint icon or corresponding signal.

To select the signals for folded hierarchy:

1. Identify the repeated unit entity.

2. Click the watchpoint icon or the signal name to get the list of instances of the signal.

3. Select one or all instances of the signal by selecting the signal instance and then sliding the cursor over to select the type of sampling to be instrumented.
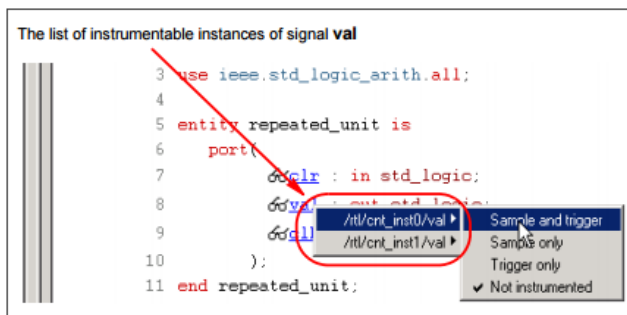
The color of the watchpoint icon is determined as follows:

- If no instances of the signal are selected, the watchpoint icon is clear.

- If all instances are defined for sampling, the color of the watchpoint icon is determined by the type of sampling specified.

Instance	Color
All instances sample only	Blue
All instances trigger only	Red
All instances sample and trigger	Green
All instances in any combination	Yellow

For example, see the *Debug Environment Reference Manual*.

To disable an instance of a signal that is currently defined for sampling:

1. Click on the watchpoint icon or signal.

2. Select the instance from the list displayed, and select Not instrumented.

For related information on folded hierarchies, see *Activating/Deactivating Folded Instrumentation,* on page 90 and *Displaying Data from Folded Signals,* on page 97.

Instrumenting the Verdi Signal Database

The instrumentor can import signals directly from the Verdi platform. After performing behavioral analysis and generating the essential signal database (ESDB), the essential signal list from the Verdi platform is brought directly into the instrumentor where the signals are instrumented. To bring in the essential signal list:

1. Load the project into the instrumentor.

2. Parse the essential signal list from the ESDB using the command:

   verdi getsignals *ESDBpath*

   In the above syntax, *ESDBpath* is the location where es.esdb++ is installed. For example:

   ```
   verdi getsignals path/es
   ```

3. Instrument the essential signal list using the command:

   verdi instrument

   The signals are automatically instrumented as sample and trigger.

4. Instrument the sample clock (a sample clock is required by the instrumentor).

5. Configure the IICE and instrument the design.

The instrumented design is then synthesized, placed and routed, and programmed into the FPGA. The debugger samples the data and generates the fast signal database (FSDB) which is then displayed in the Verdi nWave viewer.

Limitation

The instrumentation of all signals in the essential signal database may not be possible due to changes in the original signal names during optimization by the synthesis tool or differences in the signal naming conventions between the instrumentor and Verdi tools.

Selecting Breakpoints

⬤ Breakpoints are used to trigger data sampling. Only the breakpoints that are instrumented in the instrumentor can be enabled as triggers in the debugger.

To instrument a breakpoint in the instrumentor, click on the circular icon to the left of the line number. The color of the icon changes to green when enabled.
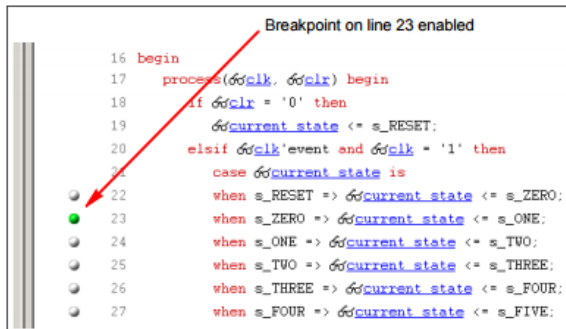
Once a breakpoint is instrumented, the instrumentor creates trigger logic that becomes active when the code region (in which the breakpoint resides) is active.

In the above example, the code region of the instrumented breakpoint is active if the variable current_state is state zero (s_ZERO) and the signal clr is not 0 when the clock event occurs.

Selecting Breakpoints in Folded Hierarchies

If a design contains entities or modules that are instantiated more than once, the design is termed to have *folded* hierarchy. By definition, there will be more than one instance of every breakpoint in a folded entity or module. To allow you to instrument a particular instance of a folded breakpoint, the instrumentor automatically detects folded hierarchy and presents a choice of all possible instances of each breakpoint.

The choices are displayed in terms of an absolute breakpoint path name originating at the top-level entity or module. The list of choices for a particular breakpoint is accessed by clicking on the breakpoint icon to the left of the line number.

To select the breakpoints in folded hierarchies:

1. Identify the repeated unit entity.

2. Click the breakpoint icon to get the list of instances of the breakpoint are available for sampling.

3. Select any or all of these breakpoints by clicking on the corresponding line entry in the list displayed.
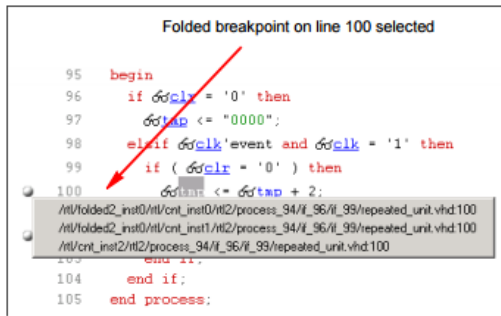
The color of the breakpoint icon is determined as follows:

- If no instances of the breakpoint are selected, the icon is clear in color.

- If some, but not all, instances of the breakpoint are selected, the icon is yellow.

- If all instances are selected, the icon is green.

For example, see the *Debug Environment Reference Manual*.

The lines in the list of breakpoint instances act to toggle the selection of an instance of the breakpoint. To disable an instance of a breakpoint that has been previously selected, simply select the appropriate line in the list box.

Configuring the IICE

If the IICE configuration parameters for the active IICE need to be changed, use the Edit IICE icon to change them. *Adding IICE,* on page 41, discusses how to set these parameters for both single- and multi-IICE configurations.

Synthesizing Instrumented Designs

When you save your instrumentation, the synthesis tool creates the set of files and subdirectories required by the debugger. These files and subdirectories are then exported from the database to an external directory location. This location can be local to your system (when running the debugger on the same machine) or the exported directory can be copied to a remote system using tar or file transfer protocol (FTP).

Capturing Commands from the Tcl Script Window

- To capture all text written to the console window, use the log console command (see the *Debug Environment Reference Manual*).

- To capture all commands executed in the console window use the transcript command (see the *Debug Environment Reference Manual*).

- To clear the text from the console window, use the clear command.

Working with IICE Files

This section describes how to configure one or more IICE units. IICE configu-
rations set in the instrumentor impact the operations available in the
debugger.

Adding IICE

Either of the following action opens the Add IICE dialog box to define the type
and name of the new IICE unit.

- Click Add IICE icon ![icon] on the instrumentor graphical window to define
  an additional IICE unit for the current design.

- Select Instrumentor > IICE > Add IICE from the menu bar.
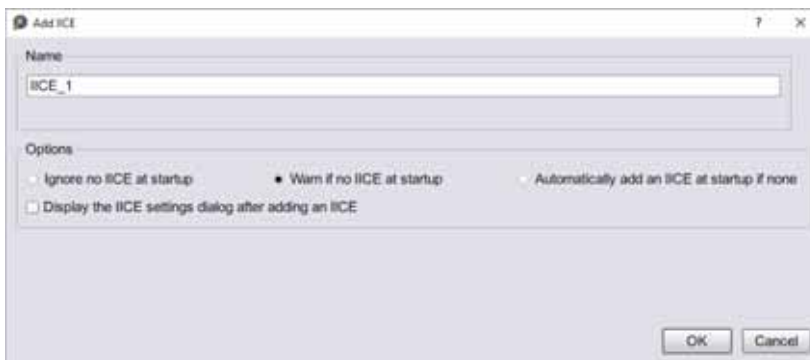
When you click OK, the HDL source code in the RTL window is redisplayed
without any signals instrumented. The Instrumentation window is cleared,
and the IICE selection reported in the status panel on the left is updated with
the name of the IICE unit. When creating a new IICE unit:

- Select Regular (the default) to add a normal IICE unit.

- Optionally enter a name for the IICE unit in the Name field. By default, the IICE name is formed by adding an _n suffix to IICE (for example, IICE_0, IICE_1, etc.).

Defining IICE/Editing IICE

The individual parameters for each IICE are defined on a series of tabs of the Edit IICE Settings dialog box.

1. Select the name of the target IICE unit appears in the Control Panel tab.
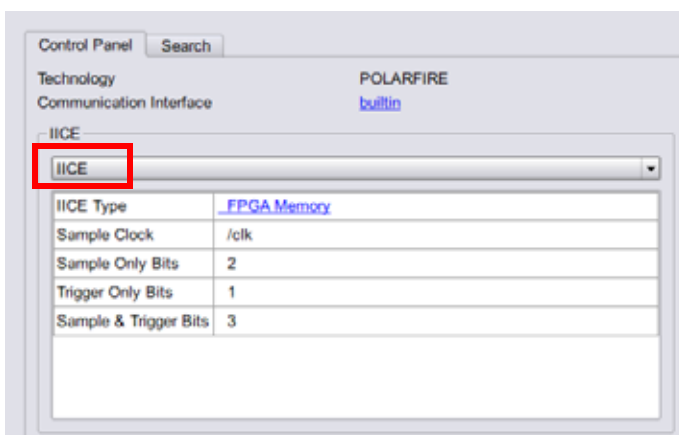
2. Click the Edit IICE icon in the top menu bar or click the entry for the IICE Type field in the Control Panel to display the Edit IICE Settings dialog box. For detailed information on IICE settings, see the *Debug Environment Reference Manual*.
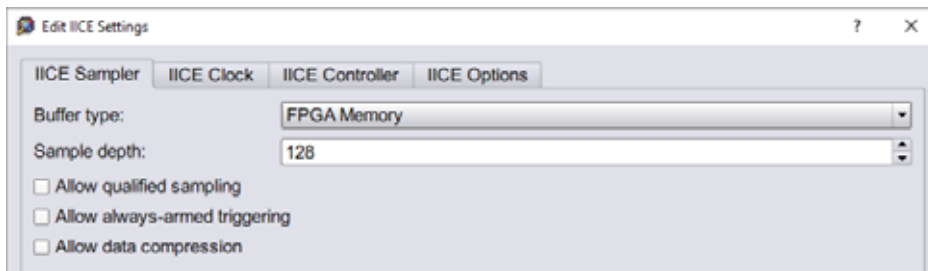
3. Select or define the required IICE settings in the following tabs:

   – IICE Sampler Tab – Select buffer type for the external memory configuration, define sample depth, and select allow qualified sampling, always-armed triggering, and data compression checkbox to perform the same in debugger.

   – IICE Clock Tab – Select the sample clock and clock edge.

   – IICE Controller Tab – Define complex counter trigger width specification and state machine triggering specifications.

   – IICE Options Tab – Set trigger-signal export and cross triggering.

4. Click Ok to save the defined settings.

For details description of the tabs, see *Debug Environment Reference Manual*.

Deleting an IICE Unit

To delete an IICE unit from the design:

1. From the Control Panel, select the specific IICE to delete.

2. Select Instrumentor > IICE > Delete IICE from the top menu or click on the Delete IICE icon in the instrumentor GUI.

3. Click OK in the confirmation dialog.

Generating an IICE File

After your HDL is successfully created, the instrumentor is used to define the specific signals to be monitored. Saving the instrumented design generates an *instrumentation design constraints* (idc) file or IICE file and adds constraint files to the HDL source for the instrumented signals and break points. The design is synthesized and then run through the remainder of the process. After the device is programmed with the debuggable design, the debugger is launched to debug the design while it is running in the target system. For information on using the debugger, see *Setting up and Running Debug,* on page 61.

The information required to instrument a design includes references to the HDL design source, the user-selected instrumentation, the settings used to create the IICE, and other system settings. Additionally, you can save the original design in either an encrypted or non-encrypted format, which is then used to reproduce the exact state of the design.

Instrumenting and Saving a Design

1. Set up the IICE. See *Adding IICE,* on page 41.

2. Define the instrumentation (selecting the signals for sampling, and setting breakpoints). See *Adding Instrumentation,* on page 28.

3. Save the instrumented design.

Saving a design generates an `idc` file and adds compiler pragmas in the form of constraint files to the design RTL for the instrumented signals and break points. This information is then used to incorporate the instrumentation logic (IICE and COMM blocks) into the synthesized netlist.

Multiple IICE Units

Multiple IICE units allow triggering and sampling of signals from different clock domains within a design. Each IICE unit is independent and can have unique IICE parameter settings including sample depth, sampling/triggering options, and sample clock and clock edge. During the subsequent debugging phase, individual or multiple IICE units can be armed. Each IICE is armed at the same or closest clock cycle. IICE will automatically download sample data when a trigger occurs, even while other IICE is still polling for triggers.

Adding Triggers

The triggering modes can be broadly classified as simple triggering mode and the complex triggering mode. The simple mode allows comparing signals to values (including don't cares) and then triggering when the signals match those values. This scheme can be enhanced by using breakpoints to denote branches in control logic. If a breakpoint is enabled, this particular branch must be active at the same time that the signals match their respective values. The overall trigger logic involves signals and breakpoints in the following way:

- Signals: All signals must match their respective comparison values in order to trigger.

- Breakpoints: All breakpoints are OR connected, which means that any one enabled breakpoint is enough to trigger.

- Signals and breakpoints are combined using AND, such that all signals must match their values AND at least one enabled breakpoint must occur.

In the advance triggering mode, you can define complex trigger conditions using the advance triggering techniques. For instance, the state machine based triggering enables you to trigger on a certain sequence of events like "trigger if pattern A occurs exactly five cycles after pattern B, but only if pattern C does not intervene."

By default, the instrumentor instruments the design according to the simple trigger mode. See the following for more information on how to make use of advance triggering techniques.

This section describes the usage of various triggering methods available in the debugger.

- Enabling State Machine based Triggering, on page 46

- Enabling Qualified Sampling, on page 46

- Enabling Always-Armed based Triggering, on page 47

- Enabling Always-Armed based Triggering, on page 47

- Enabling Sampled Data Compression, on page 47

- Enabling Complex-Counter Triggering, on page 47

- Enabling Import External Triggers, on page 48

- Enabling Export IICE Trigger Signal, on page 49

- Enabling Cross Triggering, on page 49

Enabling State Machine based Triggering

When building a complex, state-machine trigger, you specify the number of trigger states, the trigger conditions (which can be set dynamically in the debugger), and the counter width. You can enable state-machine triggering and specify the states through the user interface as outlined in the following steps:

1. Make sure that the following prerequisites are met:

   – In the instrumentor tool, select Instrumentor > IICE > Edit IICE from the menu bar or click the Edit IICE icon.

   – From the instrumentor Edit IICE Settings dialog box, select the IICE Controller tab, click the State Machine triggering radio button, and specify the number of trigger states, trigger conditions, and the counter width in the corresponding fields.

2. Create the state machine trigger in debugger. See *Creating State Machine Trigger,* on page 102.

Enabling Qualified Sampling

To create a complex trigger event to perform qualified sampling:

1. In the instrumentor tool, select Instrumentor > IICE > Edit IICE from the menu bar or click the Edit IICE icon.

2. From the Edit IICE Settings dialog box, select the IICE Sampler tab, click the Allow qualified sampling checkbox.

See also:

- *Always-Armed based Triggering,* on page 106

- *Debug Environment Reference Manual*

Enabling Always-Armed based Triggering

To enable the always-armed based triggering:

1. In the instrumentor tool, select Instrumentor > IICE > Edit IICE from the menu bar or click the Edit IICE icon.

2. From the Edit IICE Settings dialog box, select the IICE Sampler tab, click the Allow always-armed triggering checkbox.

See also:

- *Always-Armed based Triggering,* on page 106
- *Debug Environment Reference Manual*

Enabling Sampled Data Compression

A data compression mechanism is available to compress the sampled data to effectively increase the depth of the sample buffer without requiring any additional hardware resources.

To enable the data compression:

1. In the instrumentor tool, select Instrumentor > IICE > Edit IICE from the menu bar or click the Edit IICE icon.

2. From the Edit IICE Settings dialog box, select the IICE Sampler tab, click the Allow data compression checkbox or use the following command:

```
iice sampler –datacompression 1
```

See *Debug Environment Reference Manual*.

Enabling Complex-Counter Triggering

Complex-counter triggering augments the simple triggering by instrumenting a variable-width counter that can be used to create a more complex trigger function. Use the width setting to control the desired width of the counter. The complex counter connects the output of the breakpoint and watchpoint event logic to the sampling block and allows the user to implement complex triggering behavior.

Creating a Complex Counter

The counter is created, configured, and inserted into the HDL design during instrumentation using the instrumentor IICE Controller tab of the IICE Configuration dialog box or using the instrumentor iice controller command.

Complex counter for an IICE unit is enabled in the instrumentor.

1. Select Instrumentor > IICE > Edit IICE from the menu bar or click the Edit IICE icon.

2. From the Edit IICE Settings dialog box, select the IICE Controller tab.

3. Select the Complex counter triggering option and enter the Counter width.

   During configuration, the size of the counter is specified. For example, a 16-bit counter is the default. This default value produces a counter that ranges from 0 to 65535. Setting the counter size to zero during instrumentation configuration disables counter insertion.

See *Debugging with the Complex Counter,* on page 108, for details on how to debug with complex counter.

Enabling Import External Triggers

To enable this option:

1. Select Instrumentor > IICE > Edit IICE from the menu bar or click the Edit IICE icon.

2. From the Edit IICE Settings dialog box, select the IICE Options tab.

3. Enter the Import external trigger signals.

Note: When using external triggers, the pin assignments for the corresponding input ports must be defined in the synthesis or place and route tool.

Enabling Export IICE Trigger Signal

To enable this option:

1. Select Instrumentor > IICE > Edit IICE from the menu bar or click the Edit IICE icon.

2. From the Edit IICE Settings dialog box, select the IICE Options tab.

3. Select the Export IICE trigger signal check box.

Enabling Cross Triggering

Cross triggering allows the trigger from one IICE unit to be used to qualify a trigger on another IICE unit, even when the two IICE units are in different time domains. Cross triggering is available in both the simple triggering and complex counter triggering modes (state-machine triggering supports cross triggering by allowing the IICE unit IDs to be included in the state-machine equations).

To enable cross triggering for an IICE unit in the instrumentor:

1. Select Instrumentor > IICE > Edit IICE from the menu bar or click the Edit IICE icon.

2. From the Edit IICE Settings dialog box, select the IICE Options tab, click the Allow cross-triggering in IICE checkbox.

See *Selecting Cross Triggering Mode,* on page 107, for selecting cross-triggering modes.

Remote Triggering

Remote triggering allows one debugger executable to send a software trigger event to terminate data collection in the other debugger executables, effectively creating a remote stop button. It is a scripting application. The IICE or debugger targets are defined by the debugger remote_trigger command (see the command description in the *Debug Environment Reference Manual*).

You can selectively set the remote trigger to the following.

- Trigger all IICEs in all debugger executables
  remote_trigger [-all-iice iiceID]

- Trigger all IICEs in a specific debugger executable
  remote_trigger [-all-pid processID]

- Trigger a specific IICE in a specific debugger executable
  remote_trigger [-all|-pid processID|-iice iiceID]

A common design configuration is to trigger all FPGAs on a single board-level event; when that event occurs, data collection is stopped and the sample data is downloaded by the corresponding debugger executables for all FPGAs.

Selecting Buffer Type

The buffer type specifies the type of memory used to capture the on-chip signal data for debug. The type of memory you select depends on the hardware and your design requirements.

To use an external logic analyzer for debug, set up real-time debug with a new IICE (iice new command) instead of specifying a buffer type.

1. Set the buffer type from the GUI or the command line:

    – In the instrumentor tool, select the IICE icon, from the IICE Sampler tab, set the buffer type.

    – To use the command line, include this command in the idc file:

        iice sampler -iice {*iiceID* | all} *bufferType*

2. Specify the buffer type and click Ok.

Buffer Type	Command	Supported Hardware Platform
BRAM (Built-in memory)	iice sampler -iice *name* internal_memory	All
DDR3 (DDR3 memory)	iice sampler -iice *name* haps_dtd	HAPS-DX7: Onboard memory

– BRAM:
Instrumentor logic that uses distributed RAM blocks (part of the FPGA resources) to store sample data. You can use this for single-FPGA or multi-FPGA debug. See *Using BRAM for Debugging,* on page 118 for details.

– DDR3 Daughter board:
Instrumentor logic that uses external DDR3 memory to store sample data. Allows larger memory depth as compared to the BRAM based instrumentation.

Support Limitations

The debug environment fully supports the synthesizable subset of both Verilog and VHDL design languages. Designs with a mixture of VHDL and Verilog languages can be debugged – the software reads the design files in either language.

There are some limitations on which parts of a design can be instrumented by the instrumentor. However, in most cases you can always instrument all other parts of your design.

The instrumentation limitations are usually related to language features. These limitations are described in these sections.

- VHDL Instrumentation Limitations, on page 52
- Verilog Instrumentation Limitations, on page 54
- SystemVerilog Instrumentation Limitations, on page 57

VHDL Instrumentation Limitations

The synthesizable subsets of VHDL IIEEE 1076-1993 and IEEE 1076-1987 are supported in the current release of the debugger.

Design Hierarchy

Entities that are instantiated more than once are supported for instrumentation with the exception that signals that have type characteristics specified by unique generic parameters cannot be instrumented.

Subprograms

Subprograms such as VHDL procedures and functions cannot be instrumented. Signals and breakpoints within these specific subprograms cannot be selected for instrumentation.

Loops

Breakpoints within loops cannot be instrumented.

Generics

VHDL generic parameters are fully supported as long as the generic parameter values for the entire design are identical during both instrumentation and synthesis.

Transient Variables

Transient variables defined locally in VHDL processes cannot be instrumented.

Scalar Signal Syntax

The values of scalar signals of type std_logic must be enclosed in single quotes in both the GUI and the shell as shown in the following command:

```
watch enable -iice IICE -condition 0 /my_signal {'0'}
```

Entering a scalar signal either without quotes or in double quotes results in an error. Conversely, a vector signal must be entered without quotes as shown in the following command:

```
watch enable -iice IICE -condition 0 /my_bus {1010}
```

Examples of FF Coding with Breakpoints

Breakpoints inside flip-flop inferring processes can only be instrumented if they follow the coding styles outlined below:

For flip-flops with asynchronous reset:

```
process(clk, reset, ...) begin
   if reset = '0' then
      reset_statements;
   elsif clk'event and clk = '1' then
      synchronous_assignments;
   end if;
end process;
```

For flip-flops with synchronous reset or without reset:

```
process(clk, ...) begin
   if clk'event and clk = '1' then
      synchronous_assignments;
   end if;
end process;
```

Or:

```
process begin
   wait until clk'event and clk = '1'
      synchronous_assignments;
end process;
```

The reset polarity and clock-edge specifications above are only exemplary. The debug software has no restrictions with respect to the polarity of reset and clock. A coding style that uses wait statements must have only one wait statement and it must be at the top of the process.

Using any other coding style for flip-flop inferring processes will have the effect that no breakpoints can be instrumented inside the corresponding process. During design compilation, the instrumentor issues a warning when the code cannot be instrumented.

Verilog Instrumentation Limitations

The synthesizable subsets of Verilog HDL IEEE 1364-1995 and 1364-2001 are supported.

Subprograms

Subprograms such as Verilog functions and tasks cannot be instrumented. Signals and breakpoints within these specific subprograms cannot be selected for instrumentation.

Loops

Breakpoints within loops cannot be instrumented.

Parameters

Verilog HDL parameters are fully supported. However, the values of all the parameters throughout the entire design must be identical during instrumentation and synthesis.

Locally Declared Registers

Registers declared locally inside a named begin block cannot be instrumented and will not be offered for instrumentation. Only registers declared in the module scope and wires can be instrumented.

Verilog Include Files

There are no limitations on the instrumentation of 'include files that are referenced only once. When an 'include file is referenced multiple times as shown in the following example, the following limitations apply:

- If the keyword module or endmodule, or if the closing ')' of the module port list is located inside a multiply-included file, no constructs inside the corresponding module or its submodules can be instrumented.

- If significant portions of the body of an always block are located inside a multiply-included file, no breakpoints inside the corresponding always block can be instrumented.

If either situation is detected during design compilation, the instrumentor issues an appropriate warning message.

As an example, consider the following three files:

adder.v File

```
module adder (cout, sum, a, b, cin);
parameter size = 1;
output cout;
output [size-1:0] sum;
input cin;
input [size-1:0] a, b;
assign {cout, sum} = a + b + cin;
endmodule
```

adder8.v File

```
`include "adder.v"
module adder8 (cout, sum, a, b, cin);
output cout;
parameter my_size = 8;
output [my_size - 1: 0] sum;
input [my_size - 1: 0] a, b;
input cin;
adder #(my_size) my_adder (cout, sum, a, b, cin);
endmodule
```

adder16.v File

```
`include "adder.v"
module adder16 (cout, sum, a, b, cin);
output cout;
parameter my_size = 16;
output [my_size - 1: 0] sum;
input [my_size - 1: 0] a, b;
input cin;
adder #(my_size) my_adder (cout, sum, a, b, cin);
endmodule
```

There is a workaround for this limitation. Make a copy of the include file and change one particular include statement to refer to the copy. Signals and breakpoints that originate from the copied include file can now be instrumented.

Macro Definitions

The code inside macro definitions cannot be instrumented. If a macro definition contains important parts of some instrumentable code, that code also cannot be instrumented. For example, if a macro definition includes the case keyword and the controlling expression of a case statement, the case statement cannot be instrumented.

Always Blocks

Breakpoints inside a synchronous flip-flop inferring an always block can only be instrumented if the always block follows the coding styles outlined below:

For flip-flops with asynchronous reset:

```
always @(posedge clk or negedge reset) begin
   if(!reset) begin
      reset_statements;
      end
   else begin
      synchronous_assignments;
   end;
end;
```

For flip-flops with synchronous reset or without reset:

```
always @(posedge clk) begin
   synchronous_assignments;
end process;
```

The reset polarity and clock-edge specifications and the use of begin blocks above are only exemplary. The instrumentor has no restrictions with respect to these other than required by the language.

For other coding styles, the instrumentor issues a warning that the code is not instrumentable.

SystemVerilog Instrumentation Limitations

The synthesizable subsets of Verilog HDL IEEE 1364-2005 (SystemVerilog) are supported with the following exceptions.

Typedefs

Create names for type definitions that you use frequently in your code. SystemVerilog adds the ability to define new net and variable user-defined names for existing types using the typedef keyword. Only typedefs of supported types are supported.

Struct Construct

A structure data type represents collections of data types. These data types can be either standard data types (such as int, logic, or bit) or, they can be user-defined types (using SystemVerilog typedef). Signals of type structure can only be sampled and cannot be used for triggering; individual elements of a structure cannot be instrumented, and it is only possible to instrument (sample only) an entire structure. The following code segment illustrates these limitations:
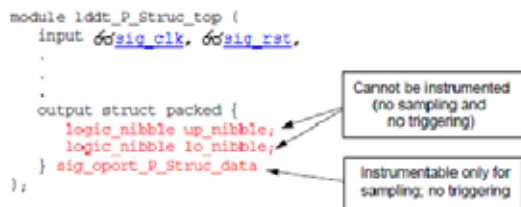
In the above code segment, port signal sig_oport_P_Struc_data is a packed structure consisting of two elements (up_nibble and lo_nibble) which are of a user-defined datatype. As elements of a structure, these elements cannot be instrumented. The signal sig_oport_P_Struc_data can be instrumented for sampling, but cannot be used for triggering (setting a watch point on the signal is not allowed). If this signal is instrumented for sample and trigger, the instrumentor allows only sampling and ignores triggering.

Union Construct

A union is a collection of different data types similar to a structure with the exception that members of the union share the same memory location. Trigger-expression settings for unions are either in the form of serialized bit vectors or hex/integers with the trigger bit width representing the maximum available bit width among all the union members. Trigger expressions using enum are not allowed.

The example below shows an acceptable sample code segment for a packed union; the trigger expression for union d1 can be defined as:

```
typedef union packed {
    shortint u1;
    logic signed [2:1][1:2][4:1] u2;
        struct packed {
            bit signed [1:2][1:2][2:1] st1;
            struct packed {
                byte unsigned st2;
            } u3_int;
        } u3;
        logic [1:2][0:7] u4;
        bit [1:16] u5;
} union_dt;
```

```
module top (
   input logic clk,
   input logic rst,
   input union_dt d1,
   output union_dt q1,
   ...
```

The maximum bit width of all elements is 16 which requires a serialized 16-bit vector to define the trigger. For example, to set st1 (2x2x2x1bit):

```
st1[1][1][2]=0
st1[1][1][1]=0
st1[1][2][2]=1
st1[1][2][1]=1
st1[2][1][2]=0
st1[2][1][1]=1
st1[2][2][2]=1
st1[2][2][1]=0
```

Similarly, to set st2:

```
(unsigned int) 200 = (bin) 11001000
```

The trigger expression is defined as:

```
16b'   00110110  11001000
       |  st1    |  st2   |
```

Arrays

Partial instrumentation of multi-dimensional arrays and multi-dimensional arrays of struct and unions are not permitted.

Interface

Interface and interface items are not supported for instrumentation and cannot be used for sampling or triggering. The following code segment illustrates this limitation:

```
interface ff_if (input logic clk, input logic rst,
   input logic din, output logic dout);
modport write (input clk, input rst, input din, output dout);
endinterface: ff_if

module top (input logic clk, input logic rst,
   input logic din, output logic dout) ;
```

```
      ff_if ff_if_top(.clk(clk), .rst(rst), .*);
      sff UUT (.ff_if_0(ff_if_top.write));
   endmodule
```

In the above code segment, the interface instantiation of interface ff_if is
ff_if_top which cannot be instrumented. Similarly, interface item modport write
cannot be instrumented.

Port Connections for Interfaces and Variables

Instrumentation of named port connections on instantiations to implicitly
instantiate ports is not supported.

Packages

Packages permit the sharing of language-defined data types, typedef
user-defined types, parameters, constants, function definitions, and task
definitions among one or more compilation units, modules, or interfaces.
Instrumentation within a package is not supported.

Concatenation Syntax

The concatenation syntax on an array watchpoint signal is not accepted by
the debugger. To illustrate, consider a signal declared as:

```
   bit [3:0] sig_bit_type;
```

To set a watchpoint on this signal, the accepted syntax in the debugger is:

```
   watch enable -iice IICE {/sig_bit_type} {4'b1001}
```

The 4-bit vector cannot be divided into smaller vectors and concatenated (as
accepted in SystemVerilog). For example, the below syntax is not accepted:

```
   watch enable -iice IICE {/sig_bit_type} {{2'b10,2'b01}}
```

CHAPTER 3

Setting up and Running Debug

Before a design can be debugged, the instrumentor is first used to define the specific signals to be monitored and then to generate an instrumentation design constraints (idc) file containing the instrumented signals and break points. The design is synthesized and the device is programmed with the debuggable design. The debugger is then launched to analyze the design while it is running in the target system.

The debugger enables HDL designs to be analyzed by interacting with the instrumented HDL design implemented in the target hardware system. You can activate breakpoints and watchpoints to cause trigger events within the IICE on the target device. These triggers cause signal data to be captured in the IICE. The data is then transferred to the debugger through a communications port where it can be displayed in a variety of formats. This chapter describes:

- Setting up the Hardware, on page 62

- Setting the Waveform Viewer, on page 84

- Debugger Operations, on page 87

- Configuring Triggering Modes, on page 101

- Debugging on a Different Machine, on page 114

- Simultaneous Debugging, on page 116

Setting up the Hardware

This section describes methods to connect the debugger to the target hardware system. The programmable device in the target system that contains the design to be debugged is usually placed on a printed circuit board along with a number of other support devices. The difficulty is that the boards differ greatly in the connections between their programmable devices, the other components, and the external connections of the boards.

This section outlines how to connect the debugger to most of the common board configurations and addresses the following topics:

Basic Communication Connection

The components that make up the debugging system are:

- The host machine running the debug environment with a loaded project.
- The communication cable connecting the host machine to the programmable device.
- The programmable device or devices loaded with the instrumented version of the design to be debugged.

The following topics are outlined in this section:

Debugger Communications Settings

Debugger communications settings are defined on the Setup tab and include selecting the cable type and setting the port parameters for the selected cable.

Selecting the Cable Type

1. From the Setup and Preferences window, select Cable Settings.

2. Select the desired technology from the drop-down menu.

3. Select the appropriate cable from the drop-down menu.
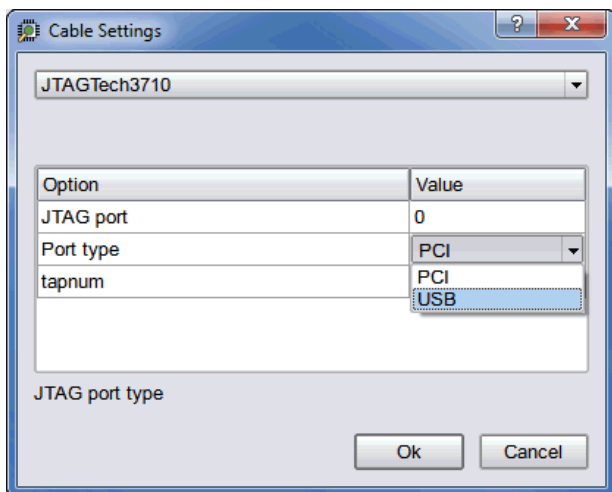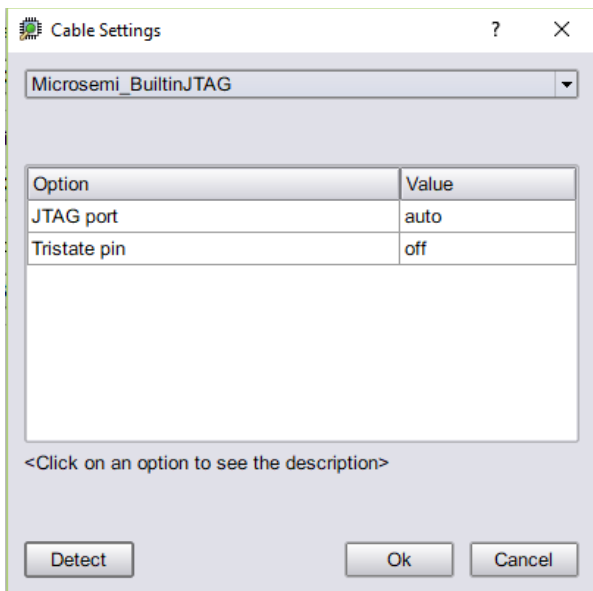
For more information on cable settings, see the *Debug Environment Reference Manual.*
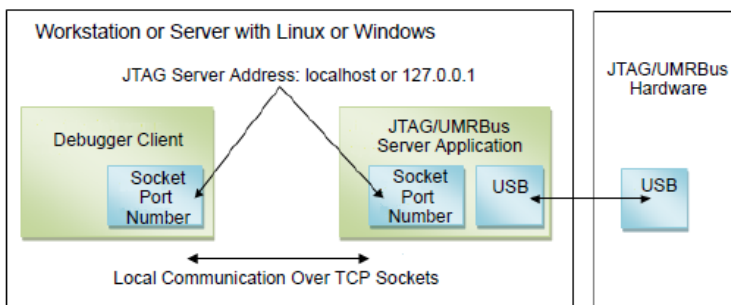
Configuring the Debugger

All parts of the debugging system must be configured correctly to make a successful connection between the debugger and the instrumented device through the cable.

In addition to selecting the cable type and port parameters, the following additional requirements must be met to ensure proper communications.

- Configuring the Local Client-Server, on page 64

- Configuring a Remote Client-Server, on page 66

- License Consumption, on page 68

- Communications Cable Connections, on page 69

- Project File, on page 70

- JTAG Chain Description, on page 70

- Device Family, on page 70

- Device Programming, on page 70
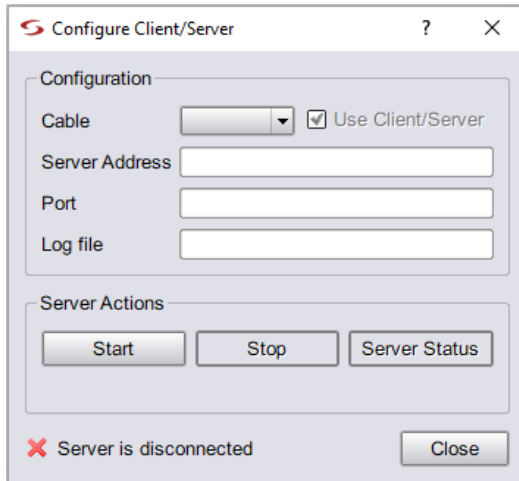
Configuring the Local Client-Server

The following figure shows a typical local server configuration.

To view the Client-Server configuration:

1. In the debugger tool, select Debugger > Setup debugger. The Setup and Preferences dialog box appears.

2. In the Communications tab, click the Configure Client/Server button.

The default settings are usually correct for most configurations and require changing only when the default server port address is already in use or when the debugger is being run from a remote machine that is not the same machine connected to the FPGA board/device (see *Config-uring a Remote Client-Server,* on page 66).

To establish a local client-server connection:

1. In the debugger tool, select Debugger > Setup debugger. The Setup and Preferences dialog box is displayed.

2. In the Communications tab, click the Configure Client/Server button.

3. Select the cable type from the Cable drop-down menu

4. Provide the server address as either 127.0.0.1 or localhost.

5. Use the default client-server port (59015), if available.

   If this port is already in use, list the port status with the netstat command and select an unused port. If possible, use a port address within the listed range where there is usually ample room.

| Ports	Port Range
Known ports for system components	0 through 1023
Registered ports for software components	1024 through 49151
Dynamic and/or private ports	49152 through 65535

6. Click the OK button.

Note: Do not use the Start button as this creates a *standalone* server which must then be manually stopped with the Stop button.

---

7. Start the debugger client-server session with a run or com check command after loading the project. The local client-server application ends automatically when the Identify debugger session ends.

8. Check the Cable type setting in the main page of the debugger after loading the project.

Configuring a Remote Client-Server

The figure below shows a client-server configuration for remote debugging.
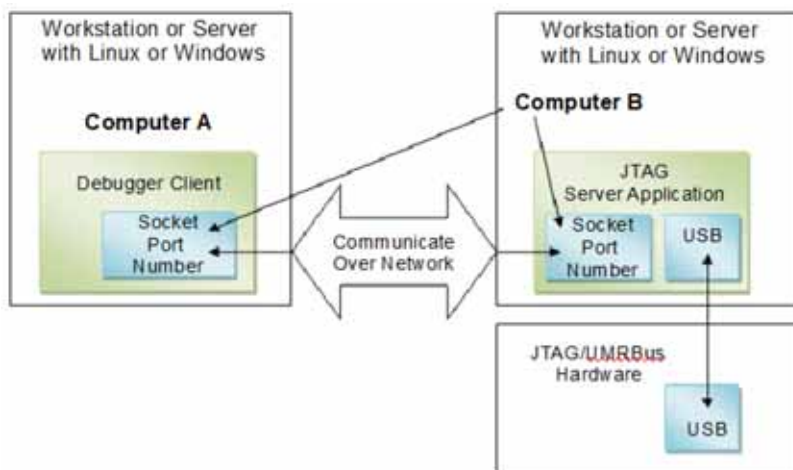
The Identify debugger uses a client-server architecture to communicate with the device. Client-server architecture lets you work remotely with the Identify debugger using Ethernet as the backbone for the client-server communication.

In the client-server architecture, the machine connected to the target device hardware (Computer B in the diagram) is termed the *server* and any machine on the same network that is used to launch the Identify debugger and connect to the server is termed the *client* (Computer A). Client-server communication uses the TCP/IP communication protocol over the network.

To establish a server connection for remote debugging:

1. Configure the target device with the design to be debugged.

2. To start the server on the machine connected to the target device, launch the Identify debugger, and then configure the server-side Identify debugger as described below:

   – Load the design project file (debug.prj) to be debugged.

   – In the debugger GUI, select Client/Server from the Debugger popup menu to display the Client/Server dialog box.

   – Specify the cable type, server address, port number, and log file name in the respective fields. Set the client/server port according to the selected cable type and enable the Use Client/Server check box. Configuring the client-server parameters does not start the server.

   – Click the Start button to start the server in standalone mode. Once started, close the dialog box by clicking OK to save any changed settings or simply click Cancel to close. With the server running, you can exit the debugger, but you must manually stop the server (click the Stop button) after your session ends.

   – If the server starts successfully, a green tick mark is shown. If the server cannot be started on the host machine, an error message is displayed.

3. To debug the design from a remote machine (client), launch the debugger on the client machine and load the design to be debugged. Then configure the client-side debugger as described below:

   – In the debugger GUI, select Client/Server from the Debugger popup menu.

   – Specify the server address, port number, and log file name in the Configure Client/Server dialog box. Use the ipconfig (Windows) or

/sbin/ifconfig (Linux) command to verify the name or tcp/ip4 address of the client. The port number must be the same as the port number used to configure the server.

Once started, close the dialog box by clicking OK to save any changed settings or simply click Cancel to close.

The following syntax shows the equivalent Tcl commands to configure the server:

server set [**-cabletype** *cabletype*] [**-addr** *hostName|IP_address*] [**-port** *serverPort*] [**-logf** *logfileName*] [**-usecs** *1|0*] [**-lastcabletype** *cableType*] [**-timeout** *timeOut*]

To view the existing server configuration settings, use the server get Tcl command.

Checking the Client-Server Communication

Check the client-server communication by running the com check command (click the Comm check button in the Setup panel). If the client-server communication cannot be established, an error message is displayed in the debugger.

The client-server architecture may not always work within a WLAN. Also, firewall restrictions as well as security software such as anti-virus or anti-spyware can also impact client-server communications.

Once the client-server communication is running properly, you can debug the design remotely.

License Consumption

If you start a debugger session on the server machine, then load an instrumented project, and run a communications check, the server does not start in standalone mode. With this method, you cannot terminate the debugger session, and two licenses are consumed.

You can start the acteljtag process in stand-alone mode on the server/host machine that interfaces to the HAPS hardware or on a Microchip device system either from the debugger GUI or from the command line. Both methods are described below.

Through GUI

1. Start the debugger on the system host.

2. Configure the client/server:
   - Select Debugger > Setup debugger > Communications tab > Configure Client Server.
   - Set the Cable Type.
   - Set the server address to the hostname of the machine (localhost or 127.0.0.1).
   - In the dialog box, specify the Port Number.
   - Click the Start button to start the acteljtag process, according to the cable type selected.

3. Close the debugger session.

   The server (acteljtag) continues to run in standalone mode, without consuming a debugger license.

4. Verify that the acteljtag process is running, using systems tools such as Task Manager or Process Explorer on Windows or ps, top, or htop on Linux.

Using Commands

As an alternative to the previous steps, start the process by running the appropriate command from the shell or command prompt.

      acteljtag -p *portNum* -l *logfile*

Use the - option with either of the commands to verify that the process is running. For example: umrbus -. For usage information about these commands, specify the -? option.

Communications Cable Connections

Two communication connections are available in the debugger.

- Cable-to-Host
- Cable-to-Board

Cable-to-Host

The latest cable types use a USB connector to interface with the host and require a USB driver to be installed. For details on installing the driver, see the installation procedures in the release notes.

A parallel port connection is also supported and requires the installation of a parallel-port driver.

When using a parallel port, make sure that the parallel port where the cable is connected corresponds to the lpt specified using the com port command. The Identify debugger uses the "standard" I/O port definitions: lpt1: 0x378-0x37B, lpt2: 0x278-0x27B, lpt3: 0x3BC-0x3BF, and lpt4: 0x288-0x28B if it cannot determine the proper definitions from the operating system. If the hardware address for your parallel port does not match the addresses for lpt1 through lpt4, use the setsys set command variable lpt_address to set the hardware port address (for example, setsys set lpt_address 0x0378 defines port lpt1).

Project File

Make sure that the project file you load into the debugger is the same one used to create the instrumented version of your design. The debugger detects any difference between the project and hardware versions when it first attempts to communicate with the device.

JTAG Chain Description

If you are using the builtin JTAG connection and the device to be debugged is part of a multi-device scan chain, the debugger first attempts to detect the devices in the scan chain. If auto-detection is unsuccessful, describe the device chain to the debugger using the chain command (see *Setting the JTAG Chain*, on page 74).

Device Family

If you are using the Identify instrumentor/Identify debugger tool set in stand-alone mode, make sure that the device family is correct for the type of programmable chip being used. If this is incorrect, you must go back and re-instrument your design using the proper device family.

Device Programming

Make sure that you program the device with the instrumented version of your design, not the original version.

JTAG Communication Interface

JTAG is a 4-wire communication protocol defined by the IEEE 1149.1 standard. The JTAG standard defines the names of the four connections as: TCK, TMS, TDI, and TDO.

The JTAG-compliant devices are connected to a host computer through a JTAG cable. Such devices can be connected directly to the cable (see the following figure), or multiple devices can be connected in a serial chain.
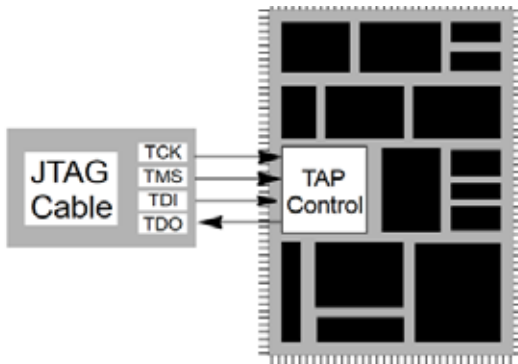
The following topics are included in this section:

- JTAG Communication Block, on page 71
- JTAG Hardware in Instrumented Designs, on page 71
- JTAG Communication Debugging, on page 80

JTAG Communication Block

The JTAG communication block can be implemented using either the built-in device-specific TAP controller (the builtin option) or using the debug environment implementation of the TAP controller (the soft option).

JTAG Hardware in Instrumented Designs

When the debug environment uses a JTAG connection to communicate with the instrumented design, the IICE must contain a TAP controller to implement the JTAG standard. The IICE JTAG connection currently can be implemented in one of two ways:

- The IICE can be configured (using the builtin option) to use the JTAG controller that is built into the programmable chip. This approach has the advantage that the built-in TAP controller already has hard-wired connections and four dedicated pins. Accordingly, employing the debug environment does not cost extra pins. In addition, the built-in TAP controller does not require any user logic resources because it usually is implemented in hard-wired logic on the chip. All devices do not have a usable built-in TAP controller.

- The IICE can be configured (using the soft JTAG port option) to include a complete, JTAG-compliant TAP controller. The TAP controller is connected to external signals by using four standard I/O pins on the programmable device. Any programmable device family can utilize this type of cable connection since it only requires four standard I/O pins.

The following sections provide more detail on these two JTAG communication options.
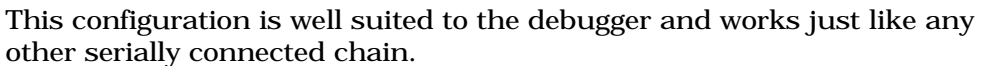
Using the Built-in JTAG Port

Some programmable device families employ a built-in TAP controller as a means for device configuration. In most cases, the IICE can also be configured to use this built-in TAP controller. Using this TAP controller saves the user logic necessary to implement the controller and also saves four I/O pins.

Using the built-in port is slightly more complicated than using the soft debug port because the built-in port usually has special board-level connections that facilitate the programming of the chip. Consequently, these programming connections must be understood to properly connect the JTAG cable to the board and to properly communicate with the IICE.

Boards with Direct JTAG Connections

HAPS boards and other boards that connect the built-in JTAG port directly to four header pins on the board allow the JTAG cable to simply be connected directly to the header pins. This configuration works for both directly connected devices and serially chained devices.

A common serial configuration is the combination of an EEPROM with a programmable device. This configuration allows you to either directly program the chip, or to program the EEPROM and then use the contents of the EEPROM to program the device via some other connection (see the following figure).

This configuration is well suited to the debugger and works just like any other serially connected chain.

Using the Synopsys Debug Port

By configuring the IICE using the soft JTAG port option, the design instrumentation includes a complete, JTAG-compliant TAP controller. The debugger connects the TAP controller to four top-level I/O connections to the design. The signal names for these connections are:

- identify_jtag_tck: The asynchronous clock signal

- identify_jtag_tms: The control signal

- identify_jtag_tdi: The serial data IN signal

- identify_jtag_tdo: The serial data OUT signal

Direct JTAG Connection

Commonly, the host computer is directly connected to the four JTAG signals on the programmable chip as follows:

- The four JTAG I/O signals on the programmable chip are connected to a header on the circuit board that contains the programmable chip.

- A standard JTAG cable is connected to the four pins on the circuit board header.

- The other end of the JTAG cable is connected to the host computer.

Serial JTAG Connection

A programmable chip using the Synopsys FPGA Debug Port can also be connected in a serial chain. To allow the debugger to communicate with the device, the configuration of the device chain must be successfully auto-detected or declared using the chain command (see the *Debug Environment Reference Manual*). The steps for making a serial cable connection are the same as a direct cable connection described above.

JTAG Clock Considerations

The JTAG clock signal syn_tck on the JTAG port drives many flip-flops in the instrumentation logic – the number depends on the instrumentation, but can be larger than 1000 flip-flops. Consequently, the clock signal on the programmable device must be able to drive large numbers of flip-flops and have low-skew properties. If the JTAG clock signal is not handled correctly, it is likely that the instrumentation will act erratically.

Most programmable devices have the ability to route such high-fanout signals using dedicated clock drivers and global clock distribution networks. Different devices use different methods of accomplishing this and have different names for this resource. Here are some guidelines:

- Some programmable devices have a number of dedicated clock I/O pins that drive internal clock distribution networks. In this case, be sure to connect the syn_tck signal to the chip using one of these clock I/O pins.

- Other programmable devices have clock buffers and clock distribution networks that can use any internal signal as a clock signal. For these technologies, the synthesis tool usually detects high-fanout signals and implements them with a clock buffer. In this case, it is important to make sure that the synthesis tool has worked correctly. If it does not put the syn_tck signal into a global buffer, it may be necessary to manually add a global buffer to this signal.

Setting the JTAG Chain

JTAG connections on an FPGA board usually chain devices together to form a serial chain of devices. This chain includes PROMs and other FPGA devices present on the board.

The debugger automatically detects the JTAG chain at the beginning of the debug session. You can review the JTAG chain settings by clicking the Show Chain button in the Setup panel.
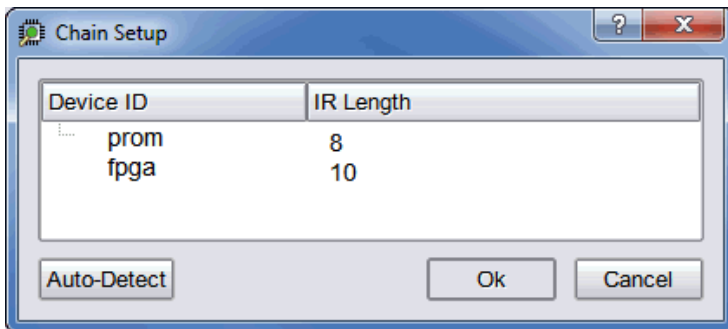
To enable the debugger to properly communicate with the target device, the device chain must be configured correctly. If, for some reason, the JTAG chain cannot be successfully configured, you must manually specify the chain through a series of chain instructions entered in the console window.

Configuring a device chain is very similar to the steps required to program the device with a JTAG programmer.

For the debugger, the devices in the chain must be known and specified. The following information is required to configure the device chain:

- The number of devices in the JTAG chain.

- The length of the JTAG instruction register for each device.

Instruction register length information is usually available in the bsd file for the particular device. Specifically, it is the Instruction_length attribute listed in the bsd file.

For the board used in developing this documentation, the following sequence of commands was used to specify a chain consisting of a PROM followed by the FPGA. The instruction length of the PROM is 8 while the instruction length of the FPGA is 5. Note that the chain select command identifies the instrumented device to the system. Identifying the instrumented device is essential when a board includes multiple FPGAs.

Note: The names PROM and FPGA have no meaning to the debugger – they simply are used for convenience. The two devices could be named device1 and device2, and the debugger would function exactly the same.

Again, the sequence of chain commands is specific to the JTAG chain on your board; these commands are the chain commands for the board used to develop this document – the board you use will most likely be different.

Type the following sequence in the console window of the debugger:

```
chain clear
chain add prom 8
chain add fpga 5
chain select fpga
chain info
```

The following figure shows the results of the above command sequence.

```
% chain clear
chain add prom 8
chain add fpga 5
chain select fpga
chain info
Position      Active      Chip ID      IR length
------------------------------------------------
1                         prom         8
2             X           fpga         5
================================================
%
```
TCL Script

Adding Microchip Soft JTAG TAP Controllers

This procedure describes how to select and set up a specific FlashPro programmer, when multiple FlashPro programmers are connected to a common host.

Note: Identify supports FlashPro4, FlashPro5 and FlashPro6 programmers on Windows and Linux platforms.

The com cableoptions option allows you to select one among the multiple FlashPro programmers connected to a common host:

```
com cableoptions Microsemi_BuiltinJTAG_port <string>
```

The string represents the FlashPro programmer's port name.

You can identify the port name and proceed to use the cable option as described below:

1. Start FlashPro.

2. Scan the programmers that are connected to the host and note down the port name (for example—usb32344).

3. Close FlashPro.

4. Start Identify debugger.

5. Define the cable type as:

    com cabletype Microsemi_BuiltinJTAG

6. Define the cable option using the FlashPro programmer port name that you identified in step 2. For example:
    com cableoptions Microsemi_BuiltinJTAG_port usb32344

Note: For FlashPro4 programmer ports, the port name must include the usb prefix, as shown in the example above. FlashPro5 ports on the other hand, must NOT include the prefix. For example: com cableoptions Microsemi_BuiltinJTAG_port S201R1NLS.

7. Check communication with the port using the com check command. If the check is successful, you can start the debugger and debug the design.

Note that you cannot change to a different port by just re-running step 6 with the new port's name. To select a different port, you need to stop the server and perform the following steps:

1. Stop the server using the server stop -forced 0 command. If this does not work, use -forced 1.

2. Define the new cable option. For example:

    com cableoption Microsemi_BuiltinJTAG_port usb32388

3. Run com check to check communication with the new port.

Adding Multiple FlashPro Programmers through GUI

This feature enables the user to select multi FlashPro devices to debug respective projects for Microchip devices using the identify tool.

The following steps describe how to select multiple FlashPro programmers in the Identify Debugger tool.+

1. Create two individual projects for two boards. For example, for boards with port number usb92433 and usb88057. The port numbers are provided to demonstrate the features.

   – Synplifyusb92433.prj

   – Synplifyusb88057.prj

2. Open project Synplifyusb92433.prj to debug the board with usb92433 JTAG port number.

3. From the Debugger menu, select Setup Debugger. The Setup and Preferences window is displayed.

4. In the Setup and Preferences dialog box click the Communications tab and click Cable Settings.

5. Click the Detect button (Ignore any warning or errors) and select the JTAG port values.
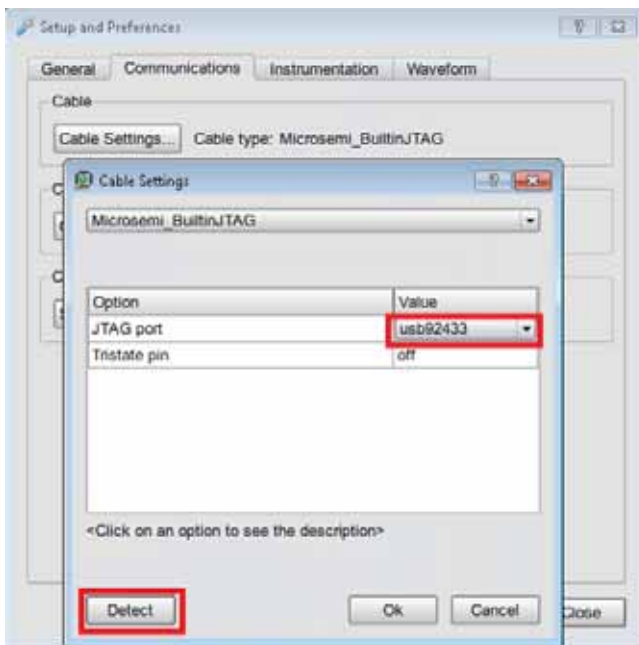
6. Select Microsemi_BuiltinJTAG from the Cable Type drop-down list.

7. Select respective JTAG port for the opened project from available JTAG ports and click Ok.
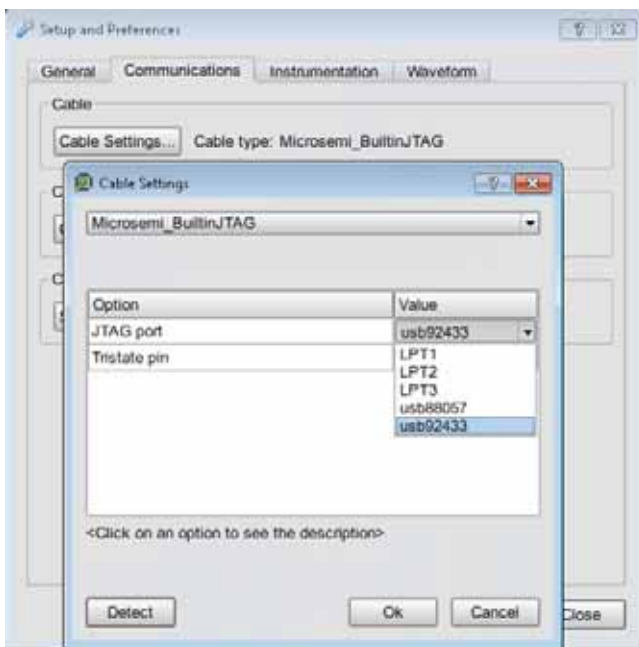
8. Perform the regular debugging steps to debug the selected project.

9. Follow the same steps to debug other projects.

JTAG Communication Debugging

The debugger performs a number of diagnostic communication tests. The first time the debugger connects to the on-chip TAP controller, it performs extensive communication tests. Later, every time the run function is executed, either by clicking the Run button or executing the run command, simpler and faster tests are executed.

A list of communication related error messages with some additional explanations are listed below.

Basic Communication Test

This test sends a pattern of ones and zeros to the chip and examines the return values

- ERROR: Communication is stuck at zero. Check the cable connection.
  It is likely that the debugger is unable to communicate with the instru-
  mented chip. This error is usually a cable connection problem, or the
  cable type is not set correctly.

- ERROR: Communication is stuck at one. Check the cable connection.
  This has the same reasons as a stuck-at-zero communication error.

- ERROR: Communication is returning incorrect IR data. Check the cable connection.
  If this error is received, then the previous two errors were NOT received
  as the communication is returning a mixture of ones and zeros.
  However, the data is not coherent and again the communication connec-
  tion is suspect.

- ERROR: Communication problem - Data sent is not the same as data received.
  This test verifies that the debugger can shift data into the instrumented
  chip and receive the same data back. If this error occurs, there is again a
  problem with your cable connection or the cable type setting is incorrect.
  Also, the JTAG chain may be experiencing noise immunity/signal integ-
  rity problems. As a troubleshooting step, select a reduced JTAG clock
  frequency by clicking Port settings in the debugger project window and
  selecting a lower clock frequency.

The last two errors can also be the result of a syn_tck signal that is not using a
high-fanout clock buffer resource, and thus may show large clock skew
properties. If you are using a parallel port, make sure that you have selected
the correct port.

On-chip Identification Register

The instrumentor adds hardware to implement an on-chip identification
register.

- ERROR: Cannot find valid instrumented design.
  The debugger cannot verify that the identification register on the instru-
  mented design is correct or even exists. This error usually means that
  the design on the programmable chip is not the instrumented version of
  the design.

- ERROR: Instrumented design on FPGA differs from design loaded into Identify
  Debugger.
  The debugger verified that the chip is instrumented but the instrumen-
  tation does not match the design that was loaded into the debugger.

JTAG Chain Tests

The debugger attempts to verify the device chain (as defined by the chain auto-detector or the chain command).

- ERROR: No hardware devices were found. Check the cable connection.
  No devices can be seen in the JTAG identification register chain. Probably a bad cable connection, or the cable type is incorrect.

- ERROR: The actual number of devices differs from the defined number: ACTUAL: XX DEFINED: YY
  The number of devices seen in the JTAG chain is XX, but the debugger was expecting the number to be YY (as was defined using the chain command). The chain description is incorrect.

- ERROR: The actual IR chain size differs from the defined size: ACTUAL: XX DEFINED: YY
  The total number of JTAG identification register bits is incorrect. The debugger measured the hardware to have XX bits, but was expecting YY bits (as was defined using the chain command). Review your chain configuration.

- ERROR: Communication with device number XX is not correct. Check your chain setup.
  If this error appears, the previous error does not appear. Thus, the total JTAG instruction register length is correct, but the size of the instruction register of device number XX is incorrect. It is likely that the order of your devices is incorrect. Review your chain settings.

Viewing JTAG Chain Settings

To view the UMRBus and JTAG chain settings, click the Show chain button in the Communication settings section of the design-view window. Normally, the chain settings for the devices are automatically extracted from the design. When the chain settings cannot be determined, they must be created and/or edited using the chain command in the console window.

The settings shown below are for a 2-device chain that has JTAG identification register lengths of 8 and 10 bits. In addition, the device named fpga has been enabled for debugging.
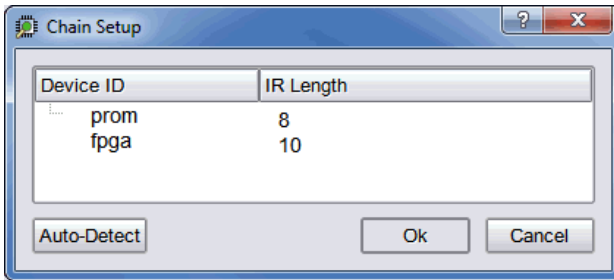
Setting the Waveform Viewer

The waveform display control displays the sampled data in a waveform. For details, see the application note, *Interfacing Your Waveform Viewer with the Debugger* on the Synopsys website.

This section describes the waveform settings and steps to install a waveform viewer if not installed earlier.

- Waveform Settings, on page 84

- Installing the Waveform Viewer, on page 86
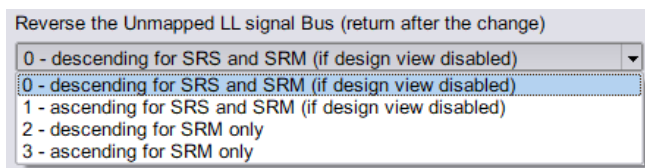
Waveform Settings

To define the waveform settings:

1. From the Debugger menu, select Setup debugger or click the Setup debugger icon on the toolbar. The Setup and Preferences dialog box is displayed.
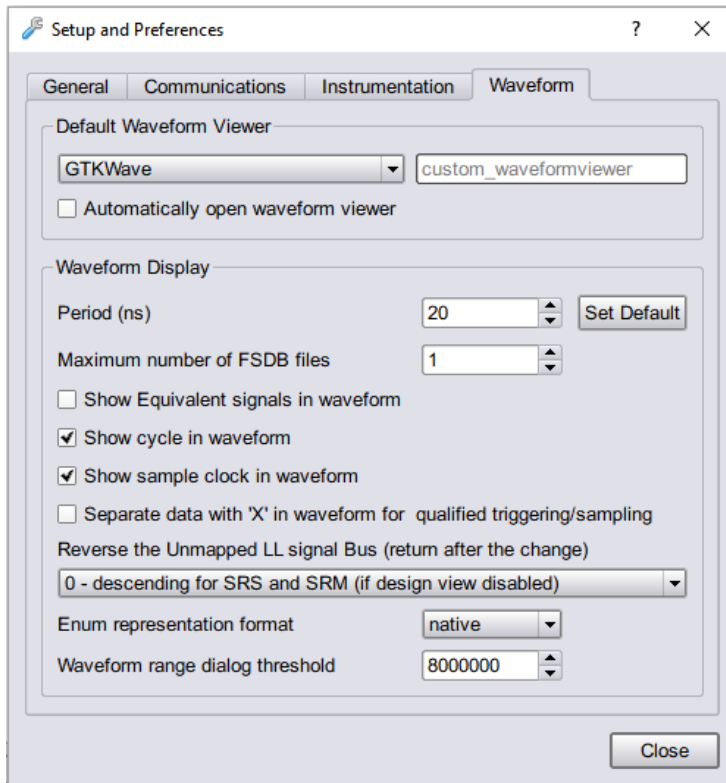
Note: The Synopsys DVE and FSDB waveform viewers are available on Linux platforms only.

2. Select the Automatically open waveform viewer checkbox, the waveform will be displayed in the selected default waveform viewer.

3. Set the Period (ns) for the waveform display and it is independent of the design speed.

4. Enter the Maximum number of FSDB files to generate.

5. Select the Show Equivalent signals in waveform checkbox to display the equivalent signals.

6. Select the Show cycle in waveform checkbox to display the clock cycles in the waveform.

7. Select the Show sample clock in waveform checkbox to display the
   instrumented sample clocks.

8. Select the Separate data with 'X' in waveform for qualified triggering/sampling
   checkbox to separate and display the data as specified.

9. Select the order of the unmapped LL signal bus from the drop-down list.

Reverse the Unmapped LL signal Bus (return after the change)

0 - descending for SRS and SRM (if design view disabled)	▼
0 - descending for SRS and SRM (if design view disabled)	
1 - ascending for SRS and SRM (if design view disabled)	
2 - descending for SRM only	
3 - ascending for SRM only	

10. Select Enum representation format from the drop-down list.

11. Set the threshold range for the waveform dialog.

Installing the Waveform Viewer

If you select a waveform viewer from the Waveform preference dialog box that is
not installed, an error message is displayed when you attempt to invoke the
viewer. To install the waveform viewer:

1. Open the Setup and Preferences dialog box (select Debugger > Setup
   debugger).

2. Select the desired waveform viewer from the drop-down menu.

Make sure that the selected simulator is installed on your machine and that
the path to the executable is set by your $PATH environment variable.

Debugger Operations

This section describes the following debugger operations:

Activating/Deactivating an Instrumentation

The trigger conditions used to control the sampling buffer comprise break-points, watchpoints, and counter settings. Activation and deactivation of breakpoints and watchpoints are discussed in this section.

Setting a Watchpoint Expression

Any signal that has been instrumented for triggering can be activated as a watchpoint in the debugger. A watchpoint is defined by assigning one or two HDL constant expressions to it. When a watched signal changes to the value of its watchpoint expression, a trigger event occurs.

1. Click-and-hold on the signal or the watchpoint icon 👓 next to the signal or click-and-hold on the signal or the P icon next to the signal for partial bus signal

2. Select Conditions > Triggering.

3. From the Set trigger expressions dialog box, enable the required condition and provide values.

4. Click Ok.

There are two forms of watchpoints: value and transition.

- A value watchpoint triggers when the watched signal attains a specific value.

- A transition watchpoint triggers when the watched signal has a specific value transition.

For information on the trigger conditions, see the *Debug Environment Reference Manual*.

Deactivating a Watchpoint

By default, a watchpoint that does not have a watchpoint expression is inactive. A watchpoint that has a watchpoint expression can be temporarily deactivated. A deactivated watchpoint retains the expression entered, but is not armed in the hardware and does not result in a trigger.

To deactivate a watchpoint:

- Click-and-hold on the signal or the associated watchpoint icon. The watchpoint pop-up menu appears.

To deactivate a partial-bus watchpoint:

- Click-and-hold on the signal or the associated "P" icon and select the bus segment from the list of segments displayed. The watchpoint popup menu appears.
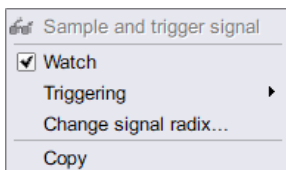
The Watch menu selection will have a check mark to indicate that the watchpoint is activated. Click on the Watch menu selection to toggle the check mark and deactivate the watchpoint.

Reactivating a Watchpoint

To reactivate an inactive watchpoint:

1. Click-and-hold on the signal or the associated watchpoint or "P" icon. Clicking the watchpoint icon redisplays the watchpoint pop-up menu

   Clicking the P icon, lists the partial bus segments; select the bus segment from the list displayed to display the watchpoint popup menu.

2. Click on the Watch menu selection to toggle the check mark and reactivate the watchpoint.

Activating a Breakpoint

Instrumented breakpoints are shown in the debugger as green icons in the left margin adjacent to the source-code line numbers. Green breakpoint icons are inactive breakpoints, and red breakpoint icons are active breakpoints. To activate a breakpoint, click on the icon to toggle it from green to red; to deactivate an active breakpoint, click on the breakpoint icon to toggle it from red to green.
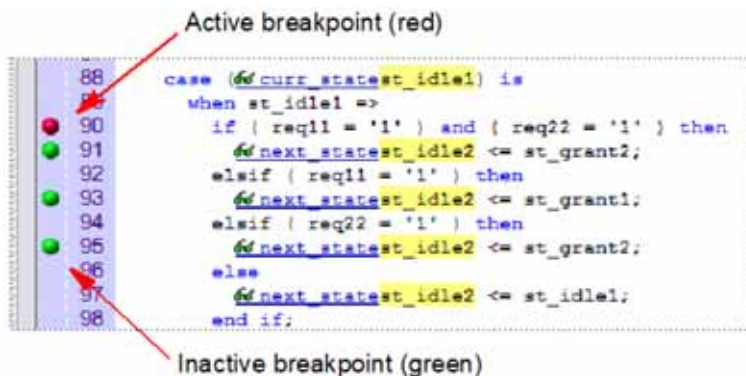
Selecting Multiplexed Instrumentation Sets

Multiplexed groups of instrumented signals defined in the instrumentor can be individually selected for activation in the debugger. For information on defining a multiplexed group in the instrumentor, see *Instrumenting the Design,* on page 19).

Using multiplexed groups can substantially reduce the amount of pattern memory required during debugging when all of the originally instrumented signals are not required to be loaded into memory at the same time.

To activate a predefined multiplexed group in the debugger:

1. Select Debugger > IICE > Configure IICE Settings or click the Configure IICE Settings icon to display the dialog box.
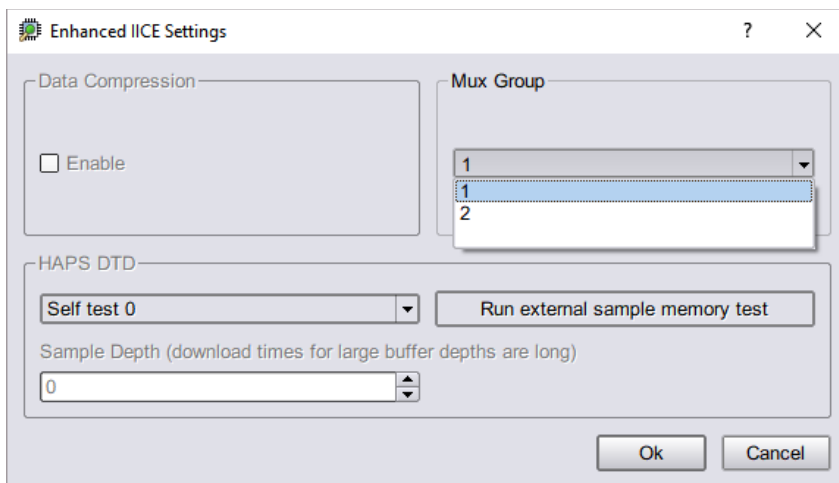
2. From the drop-down menu in the Mux Group section, select the group number to be active for the debug session.

The signals group command can be used to assign groups from the console window (see *signals* in the *Debug Environment Reference Manual*).

Activating/Deactivating Folded Instrumentation

If your design contains entities or modules that are instantiated more than once, the design is termed to have a "folded" hierarchy (folded hierarchies also occur when replicated instances are created within a generate loop). By definition, there will be more than one instance of every signal and break-point in a folded entity or module. During instrumentation, it is possible to instrument more than one instance of a signal or breakpoint.

When you debug an instrumented design with replicated instrumented instances of a breakpoint or signal, the debugger allows you to activate/deactivate each of these instrumented instances independently.

Independent selection is accomplished by displaying a list of the instrumented instances when the breakpoint or signal is selected for activation/deactivation.

Activating/Deactivating a Folded Watchpoint

The following example consists of two instances of the repeated_unit entity. The source code of repeated_unit is displayed. In this folded entity, multiple instances of the signal val and the breakpoints are instrumented.

To activate/deactivate instances of the val signal:

1. Select the watchpoint icon next to the signal.

   A list is displayed with two instrumented instances of the signal val, available for activation/deactivation:

   ```
   /rtl/cnt_inst0/val
   /rtl/cnt_inst1/val
   ```

2. Click on the appropriate line in the list box to bring up the watchpoint menu to activate/deactivate the folded watchpoint.
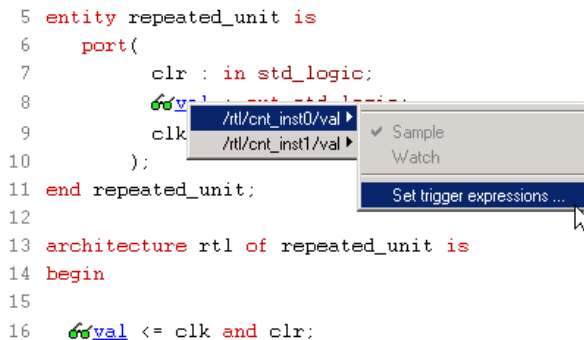
For related information on folded hierarchies, see *Sampling Signals in a Folded Hierarchy,* on page 35 and *Displaying Data from Folded Signals,* on page 97.

Activating/Deactivating a Folded Breakpoint

To activate/deactivate instances of the breakpoint, select the icon next to line number. A list will pop up with the two instrumented instances of the breakpoint available for activation/deactivation.

For example, to activate/deactivate an instance of a breakpoint on line 24, select the icon next to line number 24.

```
/rtl/inst0/rtl/process_18/if_20/if_23/repeated_unit.vhd:24
/rtl/inst1/rtl/process_18/if_20/if_23/repeated_unit.vhd:24
```

Either of these instances can be activated/deactivated by clicking on the appropriate line in the list box.

Run Command

The Run command sends watchpoint and breakpoint activations to the IICE, waits for the trigger to occur, receives data back from the IICE when the trigger occurs, and then displays the data in the source window.

To execute the Run command for the active IICE (or a single IICE):

- Select Debugger > Run from the menu or click the Run button.

If data compression is to be used on the sample data, see *Sampled Data Compression,* on page 93.

To execute the Run command for multiple IICE units

1. Enable the individual IICE units in the Run panel by checking their corresponding boxes.

2. Click Run button or select Debugger > Run from the menu.

After the Run command is executed, the sample of signal values at the trigger position is annotated to the HDL code in the RTL panel. This data can be displayed in a waveform viewer with the *waveform* command or written out to a file with the *write vcd/fsdb* command (see the corresponding command descriptions in the *Debug Environment Reference Manual*).

Note: In a multi-IICE environment, you can edit and run other IICEs while an IICE is running.

The following example shows a design with one breakpoint activated, the breakpoint triggered, and the sample data displayed. The small green arrow next to the activated breakpoint in the example indicates that this breakpoint was the actual breakpoint that triggered. Note that the green arrow is only present with simple triggering.
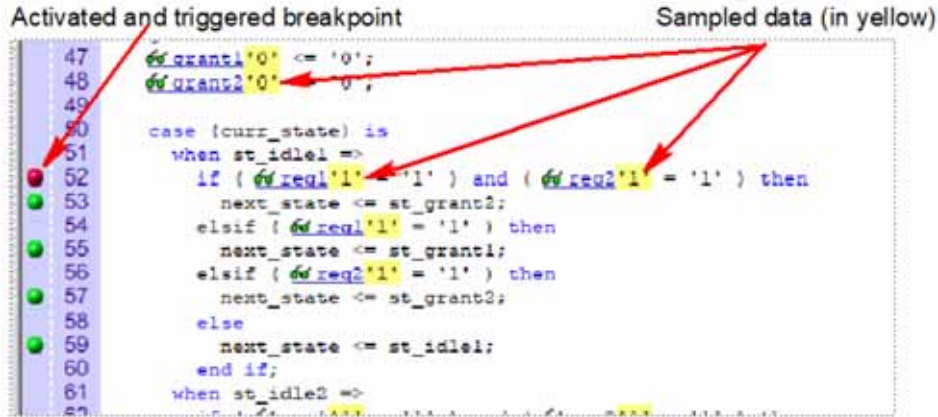
1.

Sampled Data Compression

A data compression mechanism is available to compress the sampled data to effectively increase the depth of the sample buffer without requiring any additional hardware resources. When enabled, data compression engine will ignore the sampled data that remains unchanged between the sampled cycles. A sample is automatically taken after 64 unchanging cycles.

To enable the data compression from the project view:

1. Select Debugger > Setup debugger > Instrumentation tab.

2. Click the IICE button to display the Enhanced Settings for IICE Unit dialog box

3. Click the Enable check box in the Data Compression section or enter the following command:

```
iice sampler -datacompression 1
```

Data compression must be set prior to executing the Run command and applies to all enabled IICE units. Data compression is not available when using state-machine triggering, or qualified sampling or always-armed sampling.

Sample Data Masking

A masking option is available with data compression to selectively mask individual bits or buses from being considered as changing values within the sample data. This option is only available through the Tcl interface using the following syntax:

iice sampler -enablemask 0 |1 [-msb *integer* **-lsb** *integer*] *signalName*

For example, the following command masks bits 0 through 3 of vector signal mybus[7:0] from consideration by the data compression mechanism:

```
iice sampler –enablemask 1 –msb 3 –lsb 0 mybus
```

Similarly, to reinstate the masked signals in the above example, use the command:

```
iice sampler –enablemask 0 –msb 3 –lsb 0 mybus
```

   4.

Sample Buffer Trigger Position

The purpose of the activated watchpoints and breakpoints is to cause a trigger event to occur. The trigger event causes sampling to terminate in a controlled fashion. Once sampling terminates, the data in the sample buffer is communicated to the debugger and then displayed in the GUI.

The sample buffer is continuously sampling the design signals. Consequently, you can control the exact relationship between the trigger event and the termination of the sampling. Currently, the debugger supports the following trigger positions relative to the sample buffer:

- Early

- Middle

- Late

You can determine the correct setting for the trigger position, as required. For example, if the design behavior of interest usually occurs after a particular trigger event, set the trigger position to "early." See the *Debug Environment Reference Manual*, for more information.

Sampled Data Display Controls

The sampled data display controls are used to navigate through the data values captured by the sample buffer. All sample buffer data is tagged with a cycle number based on when the data item was stored in the sample buffer relative to the trigger event. The data item stored at the trigger event time has cycle number 0, the data item stored one sample clock cycle *after* the trigger has cycle number 1, and the data item stored one sample clock cycle *before* the trigger has cycle number -1. The data display procedures allow you to retrieve data values for a specific cycle number.

The sampled data displayed in the debugger is controlled by the value given in the Cycle text field. You can manually change the cycle number by typing a number in the entry field. Or use the up and down arrows to the right of the cycle number increment or decrement the cycle number for each click.
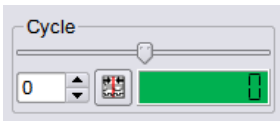
 To reset the cycle number to the default position (the zero time position), use the Debug > Cycle > Home menu selection or click on the Goto trigger event in sample history icon.

Radix

The radix of the sampled data displayed can be set to any of a number of different number bases. To change the radix of a sampled signal:

1. Right-click on the signal name or the watchpoint or P icon and select Change signal radix to display the following dialog box.
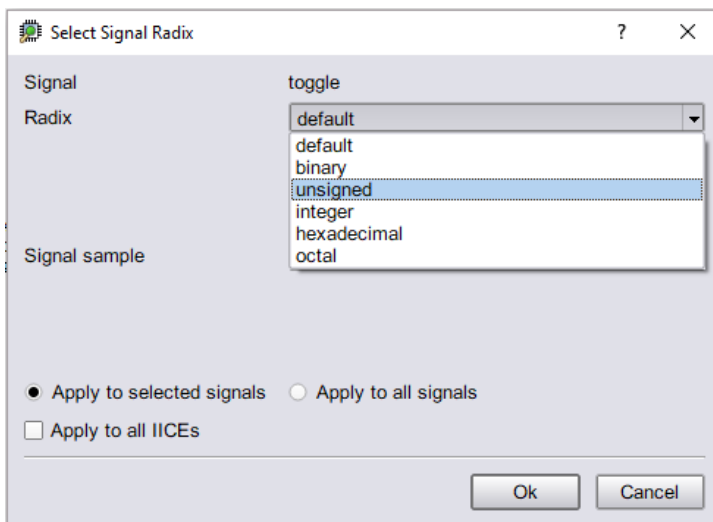
2. Select the desired radix from the Radix drop-down menu.

3. Click OK.

Note: You can change the radix before the data is sampled. The watch-
point signal value will appear in the specified radix when the
sampled data is displayed.

Selecting default resets the radix to its initial intended value. Note that the
radix value is maintained in the "activation database" and that this informa-
tion will be lost if you fail to save or reload your activation. Also, the radix set
on a signal is local to the debugger and is not propagated to any of the
waveform viewers.

Note: Changing the radix of a partial bus changes the radix for all bus
segments.

Displaying Data from Folded Signals

If your design contains entities or modules that are instantiated more than once, it is termed to have a "folded" hierarchy (folded hierarchies also occur when multiple instances are created within a generate loop). By definition, there will be more than one instance of every signal in a folded entity or module. During instrumentation, it is possible to instrument more than one instance of a signal.

When debugging an instrumented design with multiple instrumented instances of a signal, the debugger allows you to display the data values of each of these instrumented signals.

Because multiple data values cannot be displayed at the same location, a single data value is always displayed. For multiply instrumented signals, the debugger displays an ellipsis (…) to indicate that there are multiple values present. To display all of the instrumented values, click-and-hold on the ellipsis indicator.

The example below consists of a top-level entity called top and two instances of the repeated_unit entity. In the example, the source code of repeated_unit is displayed, and both of the lists of instances of the signal val have been instrumented. The two instances are /rtl/inst0/val and /rtl/inst1/val, and their data values are displayed in the pop-up window as shown in the following figure:
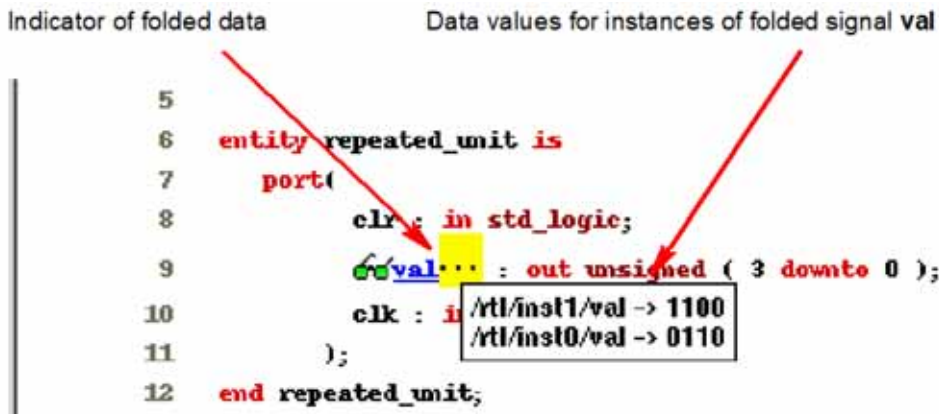
For related information on folded hierarchies, see *Sampling Signals in a Folded Hierarchy*, on page 35 and *Activating/Deactivating Folded Instrumentation*, on page 90.

Displaying Data for Partial Buses

When debugging designs with partially instrumented buses, the debugger displays the data values of each of the instrumented segments.

To display the instrumented values for the individual bus segments, position the cursor over the composite bus value. The individual partial bus values are displayed in a tooltip in the specified radix as shown in the following figure.
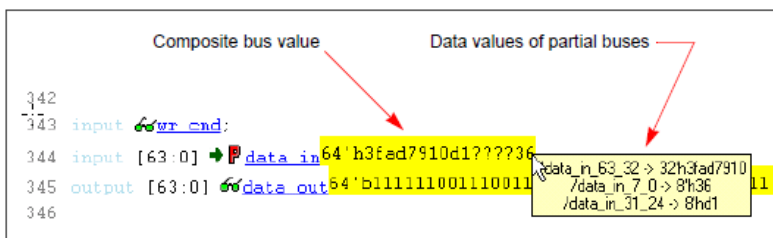
In the above figure, the question marks (?) in the composite bus value (64' h3fad7910d1????36) indicate that the corresponding segment (data_in [23:8]) has not been instrumented.

Displaying Data for Partial Instrumentation

In the debugger, the value for a fully instrumented record or structure is shown with a value for each field, in field order. The following figure shows instrumented signal sig_iport_P_Struc_instr. When displaying a partially instrumented bus, the value U is used for the uninstrumented slices. This same notation is used to show the data values for a partially instrumented record or structure (the value for each instrumented field is listed in field order, and an uninstrumented field value is shown as a U).

The Find dialog in the debugger shows a partially instrumented signal with the P icon. You can set the trigger expressions on the fields instrumented for triggering in the same manner as if the signal was fully instrumented (that is, select the signal, right-click to bring up the dialog, and select the option to set the trigger expression).

Saving and Loading Activations

The debugger includes a capture and replay function that allows you to save and load a set of enabled watchpoints and breakpoints referred to collectively as an activation. Each activation can additionally include the sample data set that was captured for a given trigger condition. Activations are stored in files with an adc extension in a project's instrumentation subdirectory.

Saving an Activation

An activation can be explicitly saved or saved on exit. To explicitly save an activation:

1. Enable the set of watchpoints and breakpoints for the activation.

2. If the sample data set is to be included, run the debugger to collect the sample data.

3. Select File > Save activations in the menu bar to bring up the following dialog box.
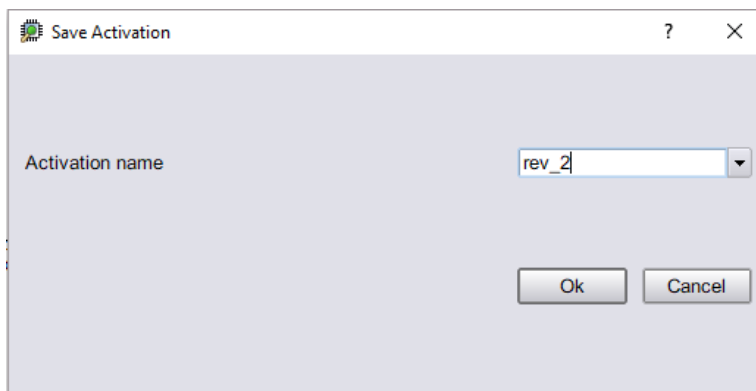
   4. Enter (or select) an activation name in the adjacent field. Selecting an
      existing activation from the drop-down menu overwrites the selected
      activation.

   5. Click OK to save the activation.

Loading an Activation

To load an existing activation:

   1. Expand (if necessary) the hierarchy to display the list of activations as
      shown in the following figure.
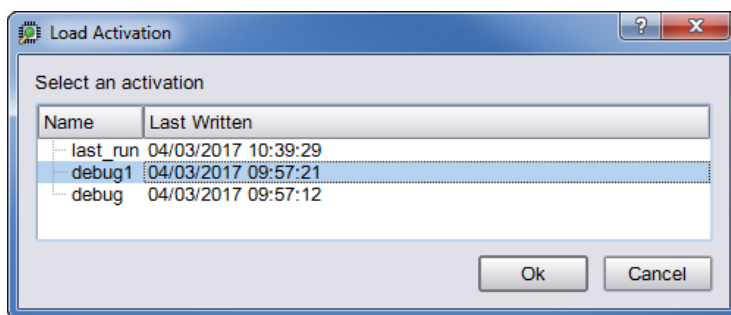
   2. Click on the desired activation and select Ok.

Autosaving Current Activation

By default, when you exit the debugger without explicitly saving an activa-
tion, the active activation is automatically saved to the last_run.adc file. This
file is automatically loaded the next time you open the project. By selecting
the Auto-save trigger settings check box in the General tab, the active activation
is automatically saved to the last_run.adc file.

Note: To save a specific activation, always use Save activations to explic-
     itly name the project file and prevent the data from overwriting
     the last_run.adc file.

To disable the auto-save feature, uncheck the Auto-save trigger settings check
box in the Setup and Preferences dialog box (select Debugger > Setup debugger >
General tab).

Configuring Triggering Modes

The triggering modes can be broadly classified as simple triggering mode and the complex triggering mode. The simple mode allows comparing signals to values (including don't cares) and then begins triggering when the signals match those values. This scheme can be enhanced by using breakpoints to denote branches in control logic. If a breakpoint is enabled, this particular branch must be active at the same time that the signals match their respective values. For more information on triggering modes, see *Adding Triggers, on page 45*.

- State Machine based Triggering, on page 101
- Qualified Sampling, on page 105
- Always-Armed based Triggering, on page 106
- Sampled Data Compression, on page 106
- Selecting Cross Triggering Mode, on page 107
- Debugging with the Complex Counter, on page 108
- Importing External Triggers, on page 109
- Exporting IICE Trigger Signal, on page 109

State Machine based Triggering

You can set up a state-machine trigger during instrumentation and then program the state machine dynamically during debug to create a complex, design-specific trigger.

This section describes:

- Creating State Machine Trigger, on page 102
- Defining State Machine from Macro Description, on page 104
- Cross Triggering with State Machines, on page 105

Creating State Machine Trigger

The debugger includes a graphical state-machine editor that is available
when state-machine triggering is enabled for the active IICE unit on the IICE
Controller tab in the instrumentor. See the *Debug Environment Reference
Manual* for more information.

1. Click the Configure state machine icon ![icon] in the debugger toolbar. Clicking
   the icon displays the Configure State Machine dialog box for the selected
   IICE.

Note: The Configure state machine icon will be disabled if state-machine
triggering was not selected in the instrumentor when the design
was instrumented and an error message will be generated if more
than 10 states are defined.
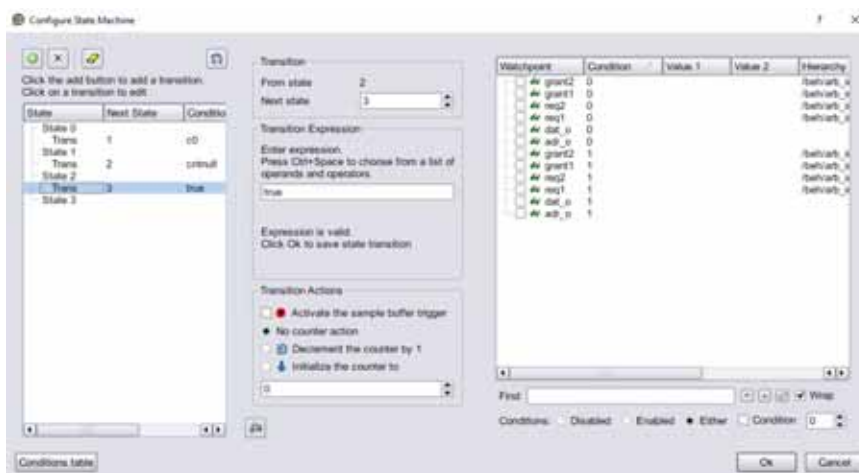
Each state is defined in an individual entry field based on the number of
states defined in the instrumentor. For each entry, you can add, edit, or
remove transitions from that state using the transition editing icons in
the upper left corner of the dialog box.

2. Click the Add a new transition icon to define or redefine the state machine.
   A panel is displayed on the right side to define the state machine.

   Each transition includes either one or two actions and a condition.
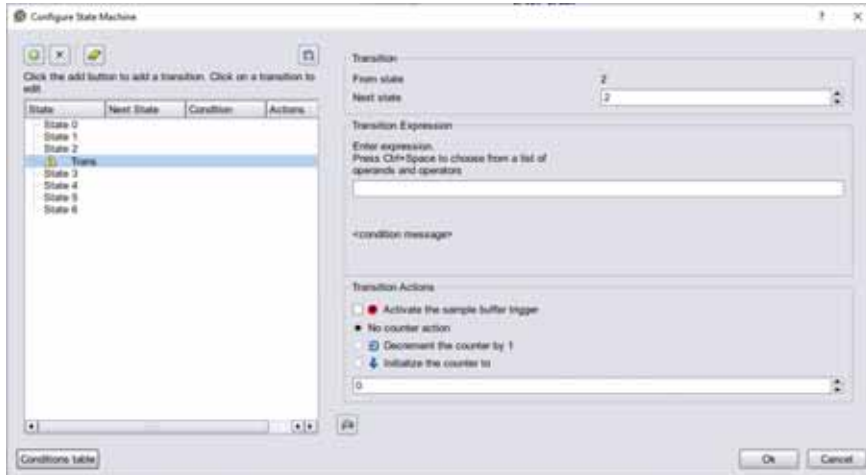
3. Enter the Transition Expression in the corresponding field.

   The conditions provided in the following table are available for defining state transition expressions.

Condition	Description
c0 ... **c***N*	References trigger event in active IICE unit
cntnull	True when counter is equal to 0 (available only when counter is instrumented)
iiceID	References trigger event from a second IICE unit for cross triggering (cross triggering must have been enabled when the design was instrumented)
ti *triggerInID*	References external trigger originating from an IICE module in another FPGA or on-board external logic
Boolean	Boolean operators used to define state-machine events (see the *Debug Environment Reference Manual*.)

4. Click OK in the initial Statemachine Editor dialog box when the state-machine triggering condition has been defined.

Note: You can view the corresponding state-machine commands in the Tcl window using the statemachine info -all command.

```
C:/tools/ident211_078R/bin$ statemachine info -all
State 0:
  if "c0" goto 1 -cntval 4
State 1:
  if "(c1 and cntnull)" goto 0 -trigger
  if "c1" goto 1 -cnten
State 2:
State 3:

C:/tools/ident211_078R/bin$
```

5. Select the Transition Actions and click Ok to save the defined triggering conditions.

Defining State Machine from Macro Description

You can define a state machine from macro settings.

1. Click the 🔧 Configure state machine from macro description icon.
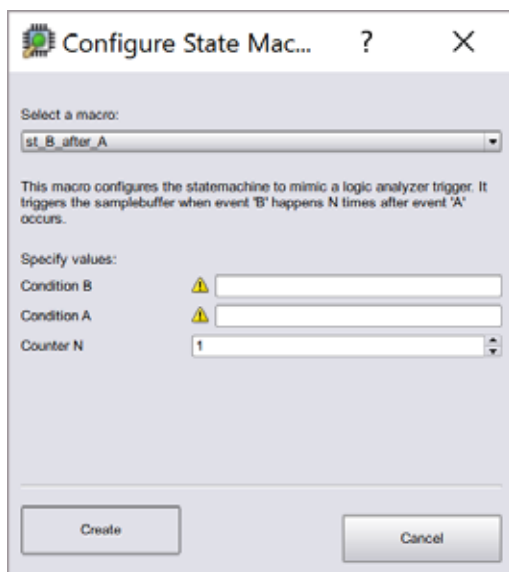
2. Select predefined macro from the Select a macro drop-down list.

3. Specify the required conditions and counters.

4. Click Create to set the defined macro settings to the state selected.

5. Click OK after all of the parameters are entered in the Configure State Machine dialog.

Cross Triggering with State Machines

Cross triggering allows a specific IICE unit to be triggered by one or more IICE units in combination with its own internal trigger conditions.

1. Ensure that cross-triggering option is enabled in the instrumentor. See
   Enabling Cross Triggering, on page 49.

   ```
   iice controller -crosstrigger 1
   ```

2. Click the Run button in the debugger project view or the following
   command in the debugger console window:

   ```
   run -iice {iiceID1 iiceID2 ... iiceIDn}
   ```

For more information on state machine triggering, see the *Debug Environment Reference Manual*.

Qualified Sampling

During qualified sampling, a single sample of all sampled signals is collected each time the trigger condition is true. When a trigger condition occurs, instead of filling the entire buffer, the IICE collects the single sample and then waits for the next trigger to acquire the next sample. The following example uses qualified sampling to examine the data for a given number of clock cycles. To create a complex trigger event to perform qualified sampling:

1. Click the Setup debugger icon or select from Debugger > Setup Debugger, to
   open the Setup and Preferences dialog box.

2. In the Instrumentation tab, select qualified_fill or qualified_intr from the
   Sample Mode drop-down list. For more information, see
   -qualified_sampling under the iice command description in the *Debug
   Environment Reference Manual*.
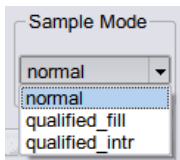
See the *Debug Environment Reference Manual,* for the example command sequence samples the data every *N* cycles beginning with the first trigger event.

Always-Armed based Triggering

The Allow always-armed triggering check box, when checked in the instrumentor, saves the sample buffer for the most recent trigger and waits for the next trigger or until interrupted. When always-armed sampling is enabled, a snapshot is taken each time the trigger condition becomes true.

To enable the always-armed based triggering, see *Enabling Always-Armed based Triggering,* on page 47.

With always-armed triggering, you always acquire the data associated with the last trigger condition prior to the interrupt. This mode is helpful when analyzing a design that uses a repeated pattern as a trigger (for example, bus cycles) and then randomly freezes. You can retrieve the data corresponding to the last time the repeated pattern occurred prior to freezing. Using always-armed sampling includes a minimal area and clock-speed penalty.

Sampled Data Compression

When enabled, data compression engine will ignore the sampled data that remains unchanged between the sampled cycles (a sample is automatically taken after 64 unchanging cycles).

To enable the data compression, see *Enabling Sampled Data Compression,* on page 47.

Data compression must be set prior to executing the Run command and applies to all enabled IICE units.

Note: Data compression is not available when using state-machine triggering, or qualified or always-armed sampling.

Sample Data Masking

A masking option is available with data compression to selectively mask individual bits or buses from being considered as changing values within the sample data. This option is only available through the Tcl interface using the following syntax:

> **iice sampler -enablemask 0|1** [**-msb** *integer* **-lsb** *integer*] *signalName*

For example, the following command masks bits 0 through 3 of vector signal mybus[7:0] from consideration by the data compression mechanism:

```
iice sampler -enablemask 1 -msb 3 -lsb 0 mybus
```

Similarly, to reinstate the masked signals in the above example, use the command:

```
iice sampler -enablemask 0 -msb 3 -lsb 0 mybus
```

Selecting Cross Triggering Mode

Cross triggering allows the trigger from one IICE unit to be used to qualify a trigger on another IICE unit, even when the two IICE units are in different clock domains. Cross triggering is available in both the simple triggering and complex counter triggering modes (state-machine triggering supports cross triggering by allowing the IICE unit IDs to be included in the state-machine equations).

1. In the debugger tool, from the Setup and Preference dialog box, select the Instrumentation tab.

2. Select the required Cross Trigger Mode from the drop-down list.
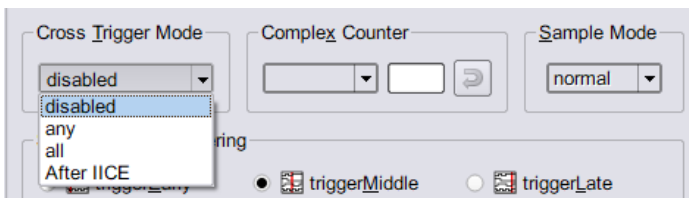
For the description of available options, see the *Debug Environment Reference Manual*.

If the cross-trigger mode drop-down is not enabled, make sure that Allow cross-triggering in IICE is enabled on the IICE Options tab of the instrumentor and that you have defined more than one IICE unit. See *Enabling Complex-Counter Triggering,* on page 47, for more information.

Debugging with the Complex Counter

The complex counter is used to produce complex triggering behavior. During the debugging of the design, the complex counter is set to zero on invocation of the debugger run command. Then, it counts events from the Master Trigger Signal event logic in a specific way depending on the counter mode.

Finally, the counter sends a trigger event to the sample block when a termination condition occurs. The form of the termination condition depends on the mode of operation of the counter and on the target value of the counter:

1. Select Debugger > Setup debugger > Instrumentation tab.

2. Select the counter mode from the drop-down list.

3. Enter the counter target value.
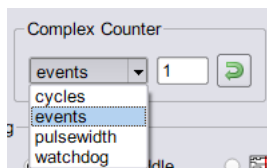
To enable the complex counter, see *Enabling Complex-Counter Triggering, on page 47*.

The following table provides a general description of the trigger behavior for the various complex counter modes. Each mode is described in more detail in individual subsections, and examples are included showing how the modes are used. In both the table and subsection descriptions, the counter target value setting is represented by the symbol n.

Counter mode	Target value = 0	Target value $n > 0$
events	Illegal	Stop sampling on the nth trigger event.
cycles	Stop sampling on 1st trigger event	Stop sampling n cycles after the 1st trigger event.
watchdog	Illegal	Stop sampling if the trigger condition is not met for n consecutive cycles.
pulsewidth	Illegal	Stop sampling the first time the trigger condition is met for n consecutive cycles.

For more description on counter modes, see the *Debug Environment Reference Manual*.

Disabling the Counter

According to the previous table, the counter can be disabled simply by setting its target value to 1 and its mode to events. Then, the complex counter will pass any received event from the Master Trigger Signal logic on to the sample block with no additional delay.

Importing External Triggers

An import external trigger capability can be used with trigger signals originating from on-board logic external to the FPGA or from an IICE module in a second FPGA. To enable, see *Enabling Import External Triggers,* on page 48.

For information on using this feature with state-machine triggering, see the *Importing External Triggers* application note available on the Synopsys website.

Exporting IICE Trigger Signal

Selecting this feature in the Instrumentor enables the master trigger signal of the IICE hardware to be exported to the top-level of the instrumented design. See *Enabling Export IICE Trigger Signal,* on page 49.

Verdi-Identify Flow

The debugger is used to generate the fast signal database (FSDB) in the Verdi platform and to display the results through the Verdi nWave viewer. To generate this database:

1. Instrument the design with the essential signal list (see *Instrumenting the Verdi Signal Database,* on page 37).

2. Run the instrumented design in the synthesis tool and load the project into the debugger.

3. Use the Debugger Preferences dialog box and make sure that Synopsys Verdi nWave is selected as the default waveform viewer.

4. Setup the trigger conditions and click the Run button to download the sample buffer.

5. Generate the fast signal database using the following command syntax:

   write fsdb -iice *iiceID* **-showequiv** *fsdbFilename*

6. Click the Open Waveform Display icon to view the samples in the nWave viewer.

The fast signal database file (*fsdbFilename*) can be imported directly back to the Verdi platform after debugging.

Debugging with the Waveform Viewer

You can use the Verdi nWave waveform viewer for data expansion and debugging, once you have generated the FSDB.

1. Launch the Verdi software from the shell.
   tb_run_verdi_fsdb fsdb *Filename*
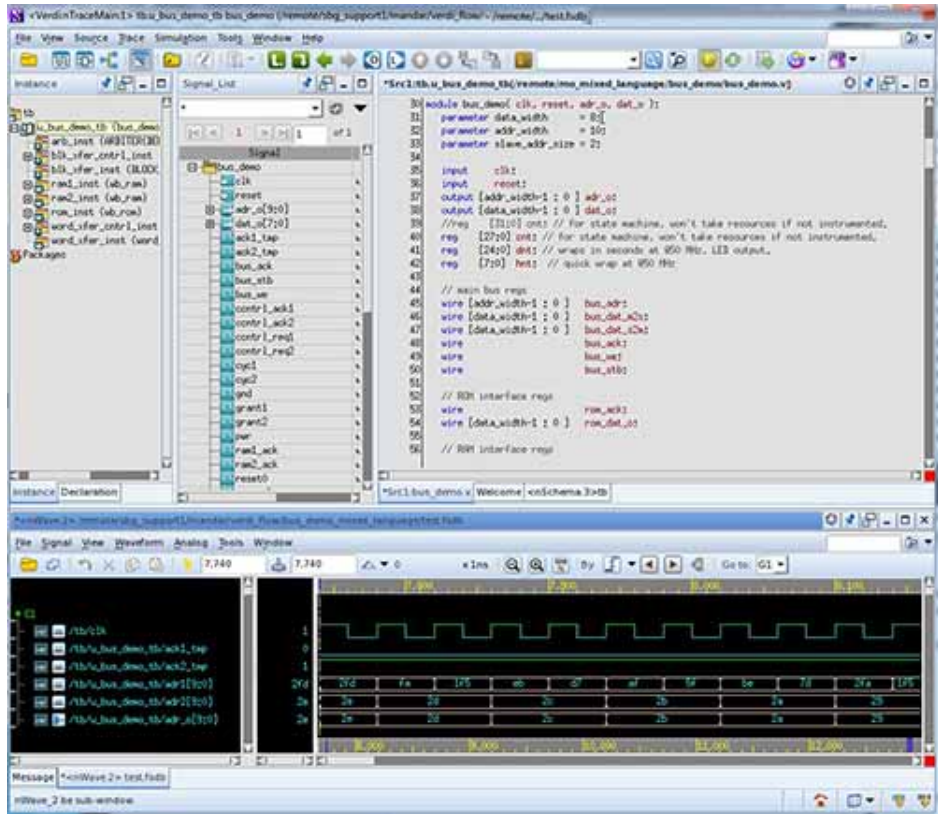
2. Enable data expansion by clicking the DE icon ( ) in the toolbar.

   The icon is green when enabled ( ) and yellow when disabled. The Data Expansion engine calculates combinational values on the fly, as they are requested.

3. Open the nWave viewer and view signals that are not in the FSDB with the Data Expansion engine.

– Click the New Waveform toolbar icon (⊞) in the main Verdi window

– Click the Get Signals icon (⤴) or select Signal > Get Signals in the nWave window.

– In the Get Signals hierarchy browser pane, click the + icon to expand items. The signals in bold are signals that are not in the FSDB, and which can be expanded (calculated on the fly) by the Data Expansion engine.

– Select the signals you want to view in the waveform viewer and click OK to add them to the waveform. Drag left to select several signals; make sure to include the top-level I/O port.

– In the main window, select Source > Active Annotation so that you can see the values overlaid on the source code.

The values for all signals are now annotated. The text in purple color are expanded signals which did not come from the FSDB file, but were calculated. This is an example of the results:

```
IICE Assignments Report

-----------------------------------------------------------------
      RTD type IICE 'IICE_1' Assignments Report
-----------------------------------------------------------------


                    Signal/breakpoint Assignments

-----------------------------------------------------------------
 Mictor Pin  | Fpga Pin |              Signal/Breakpoint
-----------------------------------------------------------------
mictor_clock_pinloc| Unknown  | mapped_nothing
-----------------------------------------------------------------
10_11_12.M1.D3e| AP11    | /beh/blk_xfer_inst/beh/cntrl_adr_tmp[7]
-----------------------------------------------------------------
10_11_12.M1.D4e| AP13    | /beh/blk_xfer_inst/beh/cntrl_adr_tmp[6]
-----------------------------------------------------------------
10_11_12.M1.D5e| AN13    | /beh/blk_xfer_inst/beh/cntrl_adr_tmp[5]
-----------------------------------------------------------------
10_11_12.M1.D6e| AN11    | /beh/blk_xfer_inst/beh/cntrl_adr_tmp[4]
-----------------------------------------------------------------
10_11_12.M1.D7e| AN12    | /beh/blk_xfer_inst/beh/cntrl_adr_tmp[3]
```

Debugging on a Different Machine

The instrumentation phase and the debugging phase can be performed on different machines. For example, the debug machine is often located in a hardware lab. When a different machine is used for debugging, you must copy or transfer the exported runtime directory from the database to the lab machine.

Since the tool set allows you to debug your design in the HDL, the debugger must have access to the original source files. Depending on the type of your network, the debugger may be able to access the original sources files directly from the lab machine. If this is not possible or if the two computers are not networked, you must also copy the original sources to the debug machine. If the debugger cannot locate the original source files, it will open the design, but an error will be generated for each missing file, and the corresponding source code will not be visible in the source viewer.

Copying the source files to the debug machine can be done in two ways:

- The instrumentor can automatically include the original source files in the exported runtime directory so that when you transfer the directory to the lab machine, the original sources files (in the orig_sources subdirectory) are included. The debugger automatically looks in this directory for any missing source files. This preference can be set before compiling the instrumented design by selecting Instrumentor > Instrumentation preference and making sure that Save original source in instrumentation directory is checked.

- The original source files can be manually copied to the lab machine or may already exist in a different location on this machine. In this case, it may be necessary to help locate the design files using the searchpath command. Simply call this command from the command line before loading the design file (debug.prj). The argument is a semi-colon-separated (Windows) or colon-separated (Linux) list of directories in which to find the original source files.

```
searchpath {d:/temp;c:/Documents and Settings/me/my_design/}
```

The debugger only displays files that match the CRC generated at the time of instrumentation.

Note: If there are security issues with having the original source files on the lab machine, the instrumentor can password-protect the original sources on the development machine for use with the debugger.

Simultaneous Debugging

When multiple debugger licenses are available, multiple FPGAs residing on a single, non-HAPS board can be debugged concurrently through a single cable. This capability is based on semaphores that allow more than one debugger to share the common port.
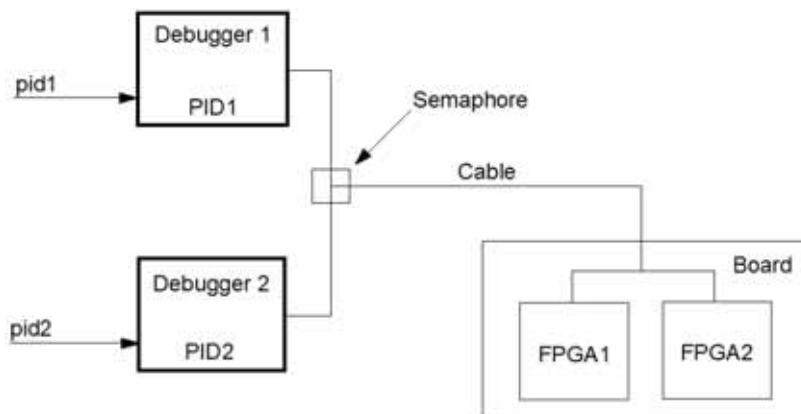
CHAPTER 4

Debugging Using FPGA Memory

This chapter describes debugging using FPGA memory and using mux sets:

- Using BRAM for Debugging, on page 118
- Using Mux Sets, on page 119
- Using State-Based Triggering, on page 120
- Debugging Script Example, on page 121

Using BRAM for Debugging

You can use the on-board BRAM blocks to store sample data for debugging operations. BRAM-based debug offers many advantages. It is a platform-independent, fast, method that does not consume I/O resources. It leverages existing system resources, so it does not require additional hardware resources. The performance limits and width are determined by the FPGA resources.

You can use it for any sampling frequency, but it is best suited to high-frequency debugging.

1. Instrument the design.

   – The design at the pre-instrument state (RTL-based instrumentation) or after compile (compiled database instrumentation).

   – Set the buffer type to internal memory:

         iice sampler iice internal_memory

   – Set simple triggering only.

2. Run through the synthesis implementation flow as usual.

   The tool uses four pins per slave IICE. It automatically inserts a cross-trigger network.

3. Run debug.

   The tool uses distributed RAM blocks to store sample data. You can view the results in a single waveform view (FSDB, VCD). It generates a balanced network where waveforms do not need to be post-processed.

Using Mux Sets

Multiplexed groups of instrumented signals defined in the instrumentor can be individually selected for activation in the debugger (for information on defining a multiplexed group in the instrumentor, see *Adding Multiplexed Groups,* on page 34).

Using multiplexed groups can substantially reduce the amount of pattern memory required during debugging when all of the originally instrumented signals are not required to be loaded into memory at the same time.

To activate a predefined multiplexed group in the debugger:

1. Select Debugger > IICE > Configure IICE Settings or click the Configure IICE Settings icon to display the dialog box.
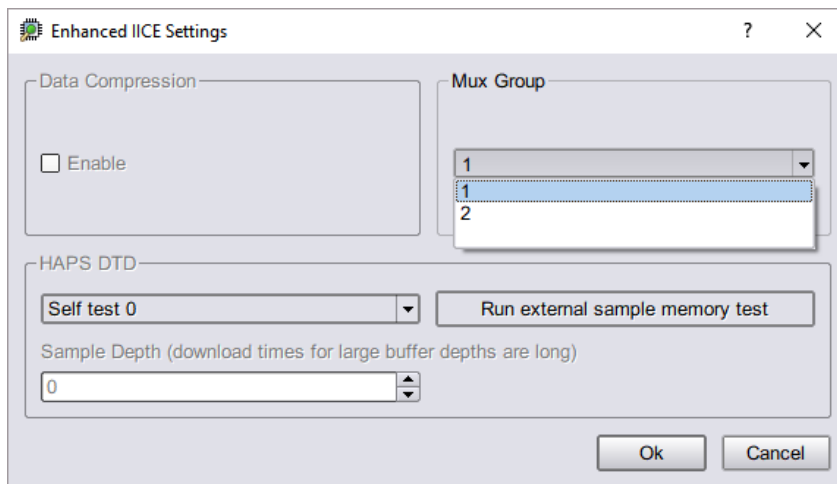
2. Use the drop-down menu in the Mux Group section to select the group number to be active for the debug session.

3. The signals group command can be used to assign groups from the console window (see the *signals* command in the *Debug Environment Reference Manual*).

Using State-Based Triggering

Perform the following steps in the debugger console window to setup a trigger in advanced triggering mode. These steps can be done in any order.

- Setup the values for the trigger conditions using the debugger watch and stop commands.

- Setup the trigger state machine behavior using the debugger statemachine command.

The watch command takes an additional parameter, -condition, specifying the trigger conditions that the given condition is intended for. This argument is available in simple mode as well, but as there is only one trigger condition in this case, the argument is redundant. See statemachine command in the *Debug Environment Reference Manual*.

State Machine Examples

To implement a trigger behavior that triggers when the pattern on condition 1 or condition 2 (c1 or c2) becomes true for the 10th time (a setting identical to counter mode events in the simple mode triggering), the following state machine can be used:

```
statemachine addtrans -from 0 -to 1 -cntval 9
statemachine addtrans -from 1 -cond "(c1 | c2) & cntnull" -trigger
statemachine addtrans -from 1 -cond "c1 or c2" -cnten
```

A trigger condition requiring pattern c2 to occur 10 times after pattern c1 has occurred, without pattern c3 occurring in between (commonly available in logic analyzers as "Pattern 1 followed by Pattern 2 before Pattern 3") can be achieved with the following state machine:

```
statemachine addtrans -from 0 -to 1 -cond c1 -cntval 9
statemachine addtrans -from 1 -cond "c2 & cntnull" -trigger
statemachine addtrans -from 1 -to 0 -cond c3
statemachine addtrans -from 1 -cond "c2" -cnten
```

These behaviors can be cascaded by moving on to the next behavior instead of triggering in the transition that has -trigger specified, as long as there are trigger conditions and states available.

Debugging Script Example

You can set up a script to instrument your design. This is one example of a
script that instruments the design without starting the GUI. For others, refer
to the debug examples included in the tool installation.

```
#Open the debugger project
set prj_path myPath/debug/debug.prj
project open $prj_path

#Set the focus to i250 debugger logic
iice current iice_dtd

#Set the sample depth, 20 signals instrumented
iice sampler -iice iice_dtd -sampledepth 500000

set keystroke Y
set i 0

while  {$keystroke eq Y} {
   #Time before running the Run command
   set sys_time_start [clock seconds]
   puts "Time before RUN command: [clock format $sys_time_start
      -format %H:%M:%S]"

   #Run the debugger
   run -iice iice_dtd -wait

   set sys_time_end [clock seconds]
   set sys_time_difference [expr  $sys_time_end - $sys_time_start]
   puts "Time after RUN command : [clock format $sys_time_end
      -format %H:%M:%S]"
   puts "Sample download time : $sys_time_difference seconds"

   puts "Downloading of samples to host computer is complete."
   puts "Writing VCD format file of sampled data."
   write vcd -iice iice_dtd -comment {Instrumentor-created
      VCD dump} -gtkwave -noequiv debug$i.vcd

   incr i

   puts "Continue Y/N"
   set keystroke [gets stdin]
   puts $keystroke

}
```

Index

A

activations
  auto-saving 114
  loading 114
  saving 113

B

blocks
  JTAG communication 83
boundary-scan registers 86
BRAM
  debug memory 136
  IICE buffer type 54
breakpoint icon
  color coding 37
breakpoints
  activating 101
  in folded hierarchy 36
  instance selection 37
  selecting 36
buffer types
  IICE 54
buffers
  instrumenting restrictions 27
buses
  instrumenting partial 28

C

cable type 70
cables
  connection 78
client-server configuration 73
communications settings 70
complex counter 51
  disabling 123
  modes 122
  size 51

Configure IICE dialog box 130
console window operations 43
cross triggering 52, 119, 121, 134
  enabling 119

D

data compression 51, 107, 120
  masking 107
DDR3
  memory card for debug 54
Debugger tool
  invoking 96
debugging
  on separate machines 132
  performance 136
  using BRAM 136
design flow 9
designs
  writing instrumented 18
dialog boxes
  Configure IICE 130
directories
  instrumentation 42

E

essential signal database 35

F

fast signal database 124
files
  idc 24
  IICE core 42
  last_run.adb 114
  project 47
folded hierarchy 33
folded signals 111