Verification Continuum™

Synopsys

Identify® Debugger for
Microsemi Edition User Guide

December 2019

SYNOPSYS®

Copyright Notice and Proprietary Information

© 2019 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

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

December 2019

Contents

Chapter 1: Using the Debugger

Chapter 2: Connecting to the Target System

CHAPTER 1

Using the Debugger

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:

Invoking and Configuring the Debugger

Configuring and invoking the debugger is described in the following sections:

- Invoking the Debugger, on page 8
- Reviewing the Instrumentation Settings (Debugger Information), on page 10
- Configuring the Debugger, on page 11

Invoking the Debugger

Before you can open a design in the debugger, the design must have been created with the instrumentor (only the instrumentor can configure a design for debugging) and synthesized. The debugger can be launched directly from a synthesis project or opened directly from a Windows or Linux prompt. Invoking the debugger includes:

- Invocation from Synplify Tool, on page 8
- Operating System based Invocation, on page 8

Invocation from Synplify Tool

If you are using a Synopsys FPGA synthesis tool, invoke the debugger directly from the graphical user interface as follows:

- From Synplify Pro 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 Synplify, 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 (see Debugger Windows and Panels, on page 24).

Operating System based Invocation

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

- 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 debugger from a Linux system:

- run identify_debugger from the /bin directory of the installation path

The debugger window opens. To display the project in the debugger window, 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.
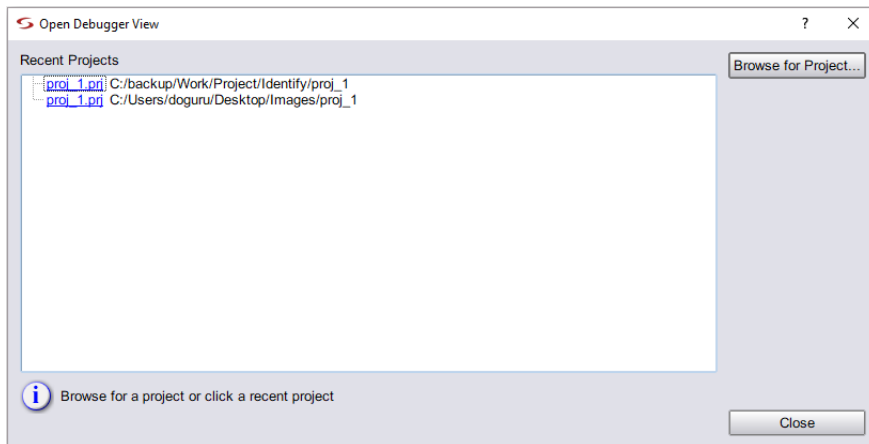
| Function	Description
Recent Projects	Displays a list of recent projects; clicking a debug.prj link opens the project.
Browse for Project	Opens a browser to select the location of a different debug.prj file.

- Click Close to navigate to the Debugger window, select File->Open from the main menu and, in the Open File dialog box, navigate to the project directory and open the corresponding project (prj) file.

While opening a project from either the tool or system invocation, the working directory is automatically set from the corresponding project file. If the project was saved with encrypted original sources, you are prompted to enter the original password used to encrypt the files. This password is then used to read any encrypted files.

Reviewing the Instrumentation Settings (Debugger Information)

The instrumentation settings are displayed in the Debugger Information window. Because these configuration settings are inherited from the instrumentor and used to construct the IICE, you cannot change these settings in the debugger. To view the debugger information, Select Debugger->Debugger Information or click ⓘ from the toolbar. The Debugger Information window pops up.
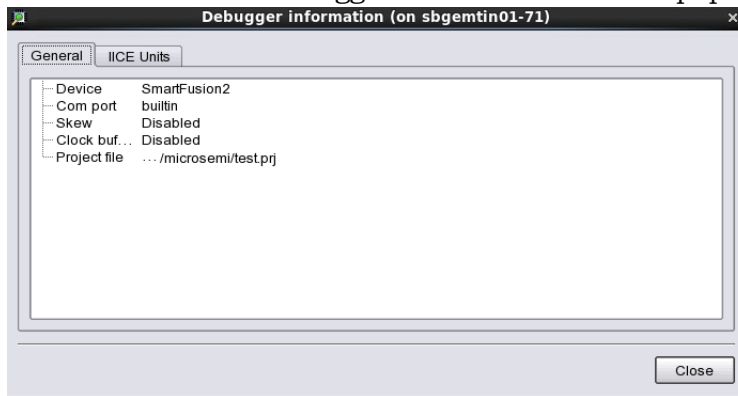
Configuring the Debugger

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. To configure the debugger settings, select Debugger->Setup debugger or click 🖉 from the toolbar. The Setup and Preferences window is displayed.
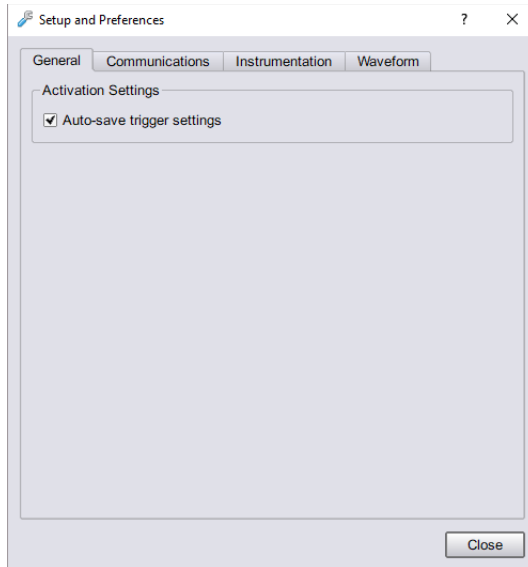
Note: Make sure that the Device Family setting is correct; an incorrect setting requires the entire instrumentation flow to be repeated before the design can be debugged.

General

By default, when you exit the debugger tool without explicitly saving an activation, the active activation is automatically saved to the last_run.adc file if the Auto-save trigger settings check box is selected. This file is automatically loaded the next time when the same project is loaded. By default, this checkbox is selected. See Autosaving Current Activation, on page 56.
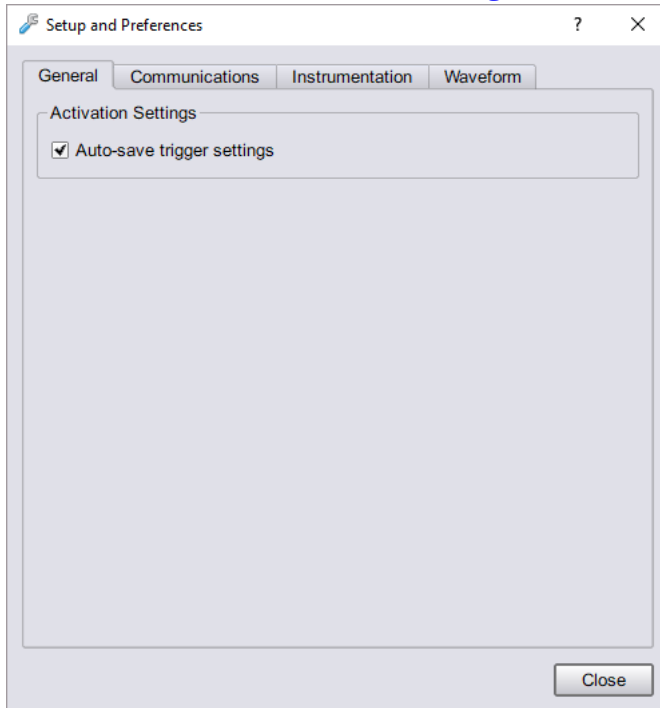
Communications

In the Communications tab, set the cable type, client server, and communication chains.

Cable Settings

The cable type and port specification communication settings can be set or changed. Clicking the Cable Settings button brings up the Cable Settings dialog box.
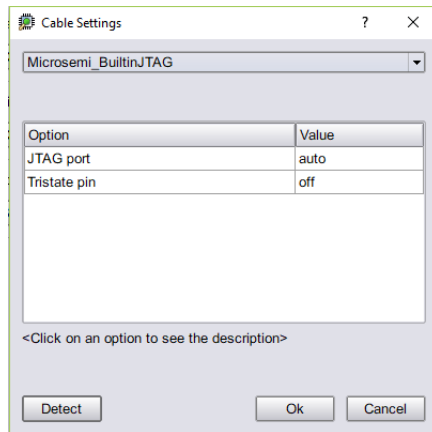
There is a list of possible vendor cable-type settings available from the drop-down menu, see *Cable Type, on page 80.* Set the Cable type value according to the type of cable you are using to connect to the programmable device (see *Debugger Communications Settings,* on page 80).

Adjust the port setting based on the port where the communication cable is connected. Most often, lpt1 is the correct setting for parallel ports.

Configure Client Server

Set up the connection between the debugger and the instrumented device through the cable by clicking 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 Remote Client-Server Configuration, on page 86).

The available configure client-server settings in the dialog box are defined in the following table:

Setting	Function
Cable	The type of interface cable (see *Cable Type, on page 80*).
Use Client/Server	Check box for enabling client-server communications when the cable type is USB-based UMRBus (limited to HAPS-70 systems).
Server Address	The address of the server. The address localhost (or 127.0.0.1) is used when the debugger is run on the same machine connected to the FPGA device. The server address is set to the name or tcp/ip4 address of the machine connected to the FPGA device/board when the debugger is run from a different machine.
Port	The port number of the server. For all Microsemi cable types, the default port number is 58015. Change the server port setting when there is a conflict with another tool on the machine.
Log file	The name of the log file.

Setting	Function
Start	Server control button for starting the server in stand-alone mode. Activating the button adds a start entry to the log file.
Stop	Server control button for stopping the server in stand-alone mode. Activating the button adds a stop entry to the log file.
Server Status	Adds a start/stop entry to the log file.

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

Show Chain

The JTAG chain settings are viewed by clicking the Show Chain button. Normally, the JTAG chain settings for the devices are automatically extracted from the devices available in the JTAG chain. When the chain settings cannot be determined, they must be created and/or edited using the chain command in the console window. Refer to the FPGA Command Reference User Guide. 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. By clicking Auto-Detect, the tool automatically identifies the programmable devices that are daisy chained on the hardware.
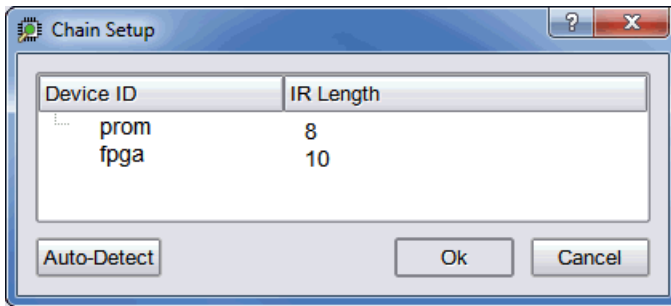
Comm Check

Runs a hardware check to verify board and cable integrity. The results of the check are displayed in the Tcl Script window.

Instrumentation

As per the IICE settings in the Instrumentation tool, you can perform the instrumentation in the Debugger tool.
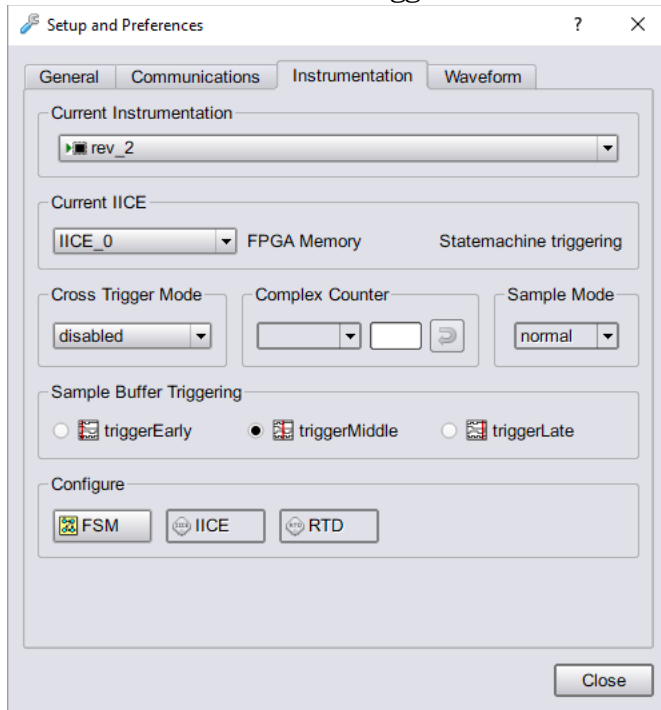
- Select the required project to instrument from the Current Instrumentation drop-down list in case of multiple projects.

- Select the required IICE from the Current IICE drop-down list. Based on the selected IICE's settings in the Instrumentation tool, you can perform the triggering here.

- Cross triggering for an IICE unit is enabled in the instrumentor by selecting the Allow cross-triggering in IICE check box on the IICE Options tab for the local IICE unit. The cross-trigger mode is selected from the drop-down menu in the debugger as shown below.

— The drop-down menu selections are as follows:

Menu Selection	Function
disabled	No triggers accepted from external IICE units (event trigger can only originate from local IICE unit)
any	Event trigger from local IICE unit occurs when an event at any IICE unit, including the local IICE unit, occurs
all	Event trigger from local IICE unit occurs when all events, irrespective of order, occur at all IICE units including the local IICE unit
After IICE	Event trigger from local IICE unit occurs only after the event at selected external IICE unit has occurred (external IICE units are individually listed)

— If the drop-down menu does not display, 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.

• 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:

— The counter target value can be set to any value in the counter's range.

— The counter has four modes: events, cycles, watchdog, and pulsewidth.

— The counter target value and the counter mode can be set directly from the debugger GUI.
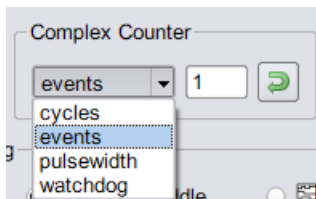
The following table provides a general description of the trigger behavior for the various complex counter modes. Each mode is described in more detail in individual subsections, and examples are included showing how the modes are used. In both the table and subsection descriptions, the counter target 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. See events Mode, on page 72.
cycles	stop sampling on 1st trigger event	stop sampling n cycles after the 1st trigger event. See cycles Mode, on page 72.
watchdog	illegal	stop sampling if the trigger condition is not met for n consecutive cycles. See watchdog Mode, on page 73.
pulsewidth	illegal	stop sampling the first time the trigger condition is met for n consecutive cycles. See pulsewidth Mode, on page 73.

- 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*.
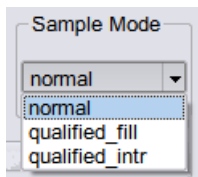
- FSM- Clicking the FSM button displays the Configure State Machine dialog box for the selected IICE.
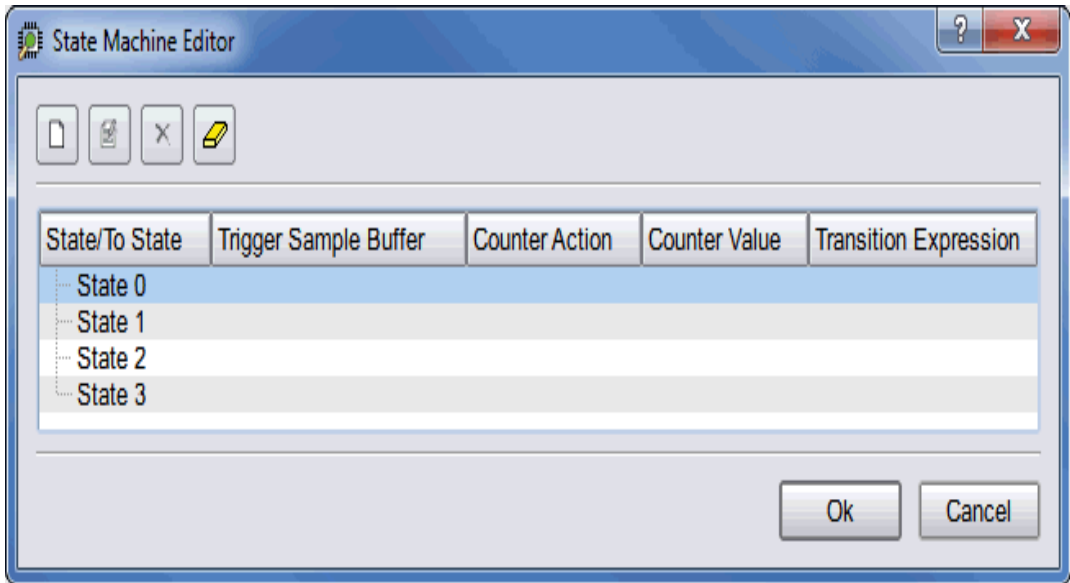
- Each state is defined in an individual entry field based on the number of states defined in the instrumentor. For each entry, you can add, edit or remove transitions from that state using the transition editing icons in the upper left corner of the dialog box. The Add a new transition icon allows you to define or redefine the state machine in the center

panel. Each transition includes either one or two actions and a condition.
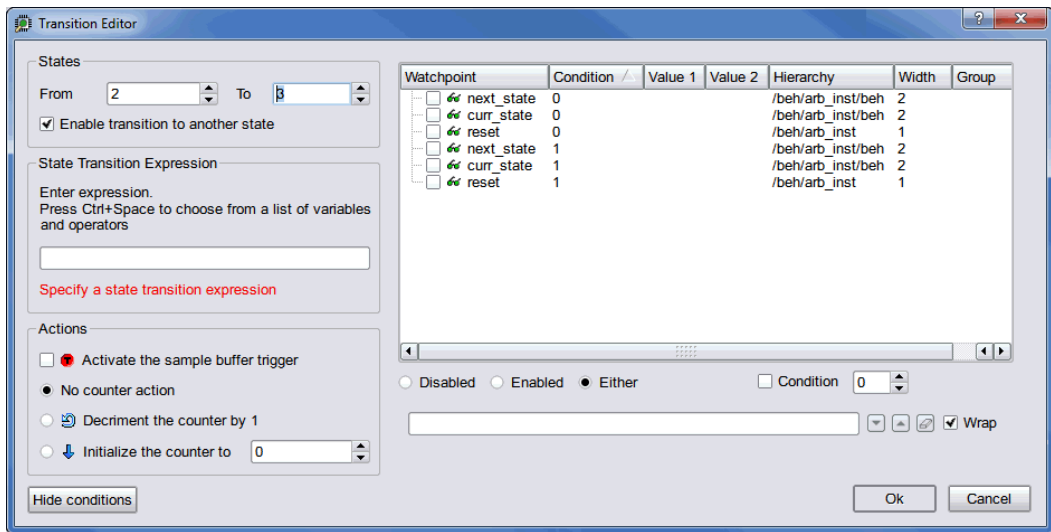
– From the dialog box, enter the state-transition expression in the corresponding field. The conditions in the following table are available for defining state transition expressions.

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 Conditions, on page 60)

– Note that you can view the corresponding state-machine commands in the Tcl Script 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$
```

- IICE - Clicking the IICE displays the Enhanced IICE Settings dialog box.
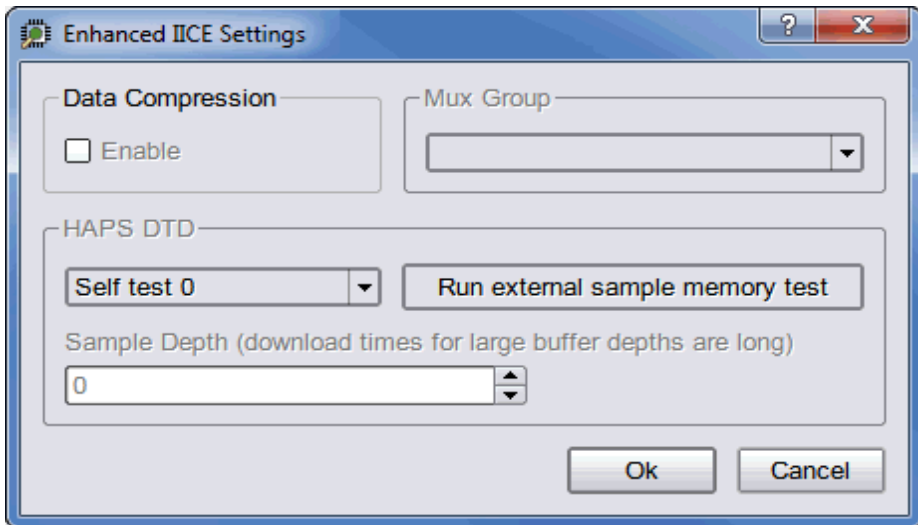
- From the Mux Group drop-down menu select the group number to be active for the debug session.
- The signals group command can be used to assign groups from the console window (see *signals* in the *Debug Environment Reference Manual*).

Waveform

The waveform display control displays the sampled data in a waveform. By default, this feature uses the GTKWave waveform viewer. Provision for using other popular waveform viewers that support VCD and FSDB data is included. Alternately, you can interface your own waveform viewer by writing

a customized script to access your waveform viewer from the debugger. For details, see the application note, "Interfacing Your Waveform Viewer with the Debugger" on the Synopsys website.

```
┌─────────────────────────────────────────────────────────────┐
│ General │ Waveform │                                          │
│ ┌─Default Waveform Viewer──────────────────────────────────┐ │
│ │                                                          │ │
│ │  [GTKWave          ▼]  [custom_waveformviewer         ]  │ │
│ │                                                          │ │
│ └──────────────────────────────────────────────────────────┘ │
│ ┌─Waveform Display─────────────────────────────────────────┐ │
│ │                                                          │ │
│ │  Period (ns)            [20      ▲]    [ Set Default ]   │ │
│ │                                ▼                         │ │
│ │  ☐ Show Equivalent signals in waveform                  │ │
│ │  ☑ Show cycle in waveform                               │ │
│ │  ☑ Show sample clock in waveform                        │ │
│ │  ☐ Separate data with 'X' in waveform for  qualified    │ │
│ │     triggering/sampling                                 │ │
│ │  Reverse the Unmapped LL signal Bus (return after the   │ │
│ │     change)                                             │ │
│ │  [0 - descending for SRS and SRM (if design view disabled) ▼] │
│ │  Enum representation format    [native      ▼]          │ │
│ │  Waveform range dialog threshold  [8000000  ▲]          │ │
│ │                                            ▼            │ │
│ └──────────────────────────────────────────────────────────┘ │
└─────────────────────────────────────────────────────────────┘
```
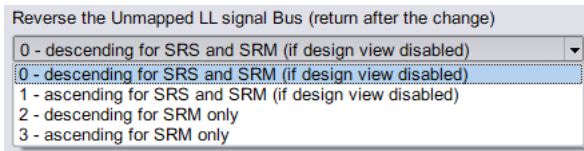
Note: The Synopsys DVE and FSDB waveform viewers are available on Linux platforms only.

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

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

- Enter the Maximum number of FSDB files to generate.

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

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

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

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

- 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
```

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

- Set the threshold range for the waveform dialog.

Debugger Windows and Panels

The Graphical User Interface for the debugger includes all of the required user controls, a hierarchy browser in the center, and the following panels for debugger-specific functions:

- Run Panel and Run Button, on page 25

- Search Panel, on page 26

- Hierarchy Browser, on page 28

- RTL Panel, on page 29

- Watchpoints Panel, on page 30

- Breakpoints Panel, on page 31

- Sampled Signals Panel, on page 34

- TCL Window, on page 35
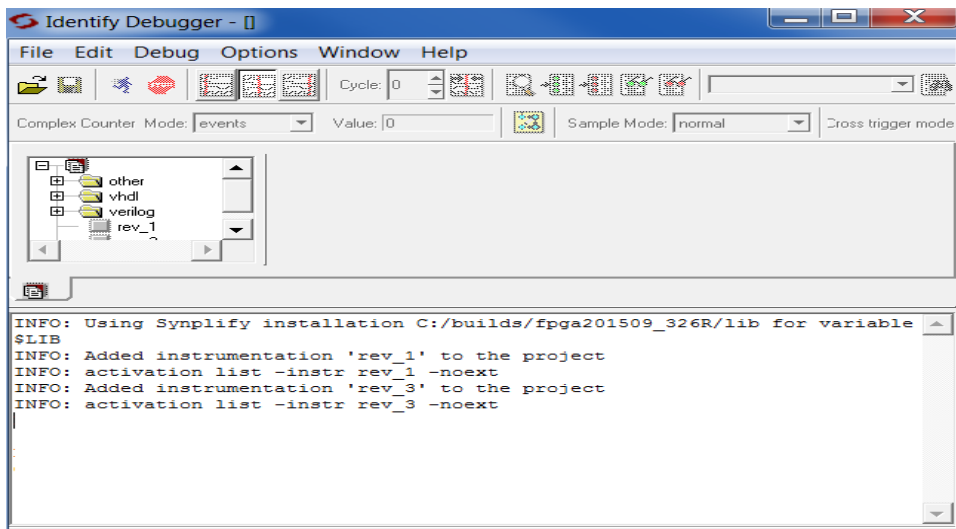
In this section, each of these areas and their uses are described. The following discussions assume that:

- an HDL design has been loaded into the instrumentor and instrumented

- the design has been synthesized in the synthesis tool

- the synthesized output netlist has been placed and routed by the place and route tool

- the resultant bit file has been used to program the FPGA with the instrumented design

- the board containing the programmed FPGA is cabled to the host for analysis by the debugger

User can toggle between the control/search, browser, and view panels.

- ⊞ Toggle Control/Search Panel - Click to view or hide the Search panel.

- ⊞Toggle Browser Panel - Click to view or hide the Browser (hierarchy) panel.

- ⊞Toggle View Panel - Click to view or hide the View (RTL, Watchpoints, Breakpoints, and Sampled Signals tabs) panel.

- ⊞Show All Panels - Click to view all panels.

Run Panel and Run Button

The Run panel reports the results of a run according to the watchpoints and breakpoints set.
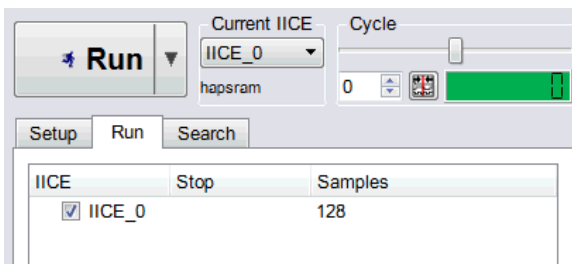

The Run panel includes six fields:

- IICE – displays the IICE unit or units selected for the run. Clicking an adjacent checkbox enables the IICE unit.

- Stop – displays a Stop sign icon while the run is in progress; clicking the icon prematurely stops the run.

- Samples – displays the number of samples collected when the run is completed.

- Type – displays type of the IICE.

- Buffer – displays the memory selected in the Instrumentor.

- Status Info – indicates the current IICE status.

A run is initiated by either clicking the Run button directly above the Run panel or by selecting Debugger->Run from the menu bar. The Run button drop-down menu determines the unit or units selected for the run. In case of multiple IICEs, you can run a single or multiple IICEs.
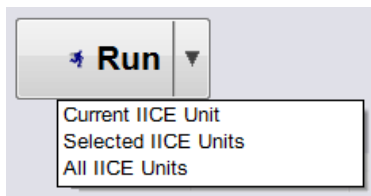

When initiated, the command sends watchpoint and breakpoint activations to the IICE, waits for the trigger event to occur, receives the captured sampled data back from the IICE when the trigger occurs, and then updates the watchpoint/breakpoint data in the RTL panel.

If data compression is to be used on the sample data, refer to Sampled Data Compression, on page 48. 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 or written out to a file (see the debugger waveform and write vcd command descriptions in the *Debug Environment Reference Manual*.

Search Panel

The Search panel is a general utility to search for signals, breakpoints, conditions and/or instances. The panel includes an area for specifying the objects to find and an area for displaying the results of the search.
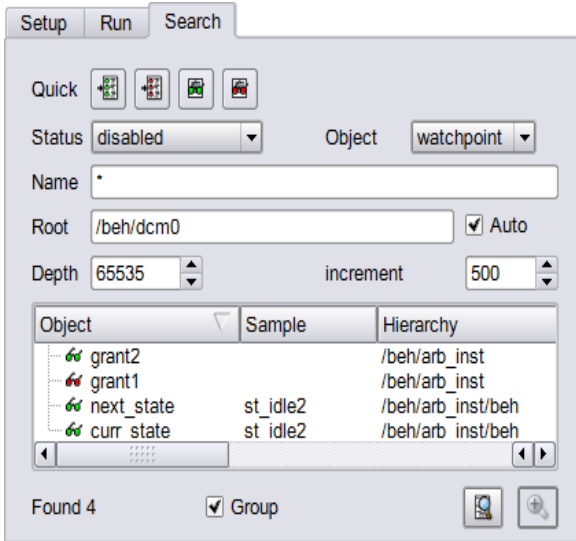
The search criteria in the upper section of the panel includes these options:

- Quick – icons that preset the conditions for instrumented or non-instrumented watchpoints or breakpoints.

  - Search for disabled breakpoints

    –To display the disabled (inactive) breakpoints, click the Search for disabled breakpoints icon.

  - Search for enabled breakpoints

    –To display the enabled (active) breakpoints, click the Search for enabled breakpoints icon.

  - Search for disabled watchpoints

    –To display the disabled (inactive) watchpoints, click the Search for disabled watchpoints icon.

  - Search for enabled watchpoints

    –To display the enabled (active) watchpoints, click the Search for enabled watchpoints icon.

- Status – specifies the status of the object to be found from the drop-down menu. The values can be enabled, disabled, sample_clock, sample_only, or "*" (any). The default is "*" (any). Status will have readback as drop down if the loaded project is of readback type.

- Object – specifies the type of object to search for from the drop-down menu: breakpoint, instance, signal or "*" (any). The default is "*" (any).

- Auto root – when checked, fills in the root hierarchy.

- Name – specifies a name, or partial name to search for in the design. Wild cards are allowed in the name. The default is "*" (any).

- Root – specifies the location in the design hierarchy to begin the recursive search. Root ("/") is the default setting.

- Depth – specifies the hierarchical depth of the instrumented signals to be searched.

- Increment – specifies the maximum number of signals to be listed upon successful search. Click the icon 🔲 to add more results to the list.

- Group – indicates to expand search to all groups when checked.

The search results in the lower section of the panel show each object found along with its hierarchical location. In addition, for breakpoints and watchpoints, the results section includes the corresponding icon (watchpoint or breakpoint) that indicates the instrumentation status of the qualified signal or breakpoint.

Hierarchy Browser

The hierarchy browser shows a graphical representation of the design hierarchy. At the top of the browser is the ROOT node. The ROOT node represents the top-level entity or module of your design. For VHDL designs, the first-level below the ROOT is the architecture of the top-level entity. The level below the top-level architecture for VHDL designs, or below the ROOT for Verilog designs, shows the entities or modules instantiated at the top-level.

Clicking on a + sign opens the entity/module instance so that the hierarchy below that instance can be viewed. Lower levels of the browser represent instantiations, case statements, if statements, functional operators, and other statements.
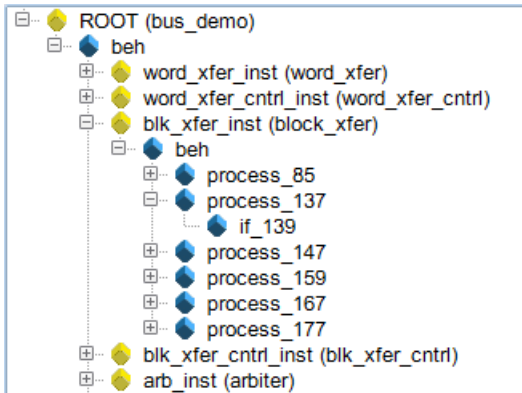
Single clicking on any element in the hierarchy browser causes the associated HDL code to be displayed in the RTL panel.

RTL Panel

The RTL panel displays the HDL source code annotated with signals and breakpoints that were previously instrumented.

Note: Signals and breakpoints that were not enabled in the instrumentor are not displayed in the debugger.

Signals that can be selected for setting watchpoints are underlined, colored in blue text, and have goggles   (or   ) icons next to them. Breakpoints that can be activated have small green circular icons in the left margin to the left of the line number.

```
32  entity arbiter is
33    port (
34      ⏰ clk       : in std_logic;
35      clk_sys : in std_logic;
36      👓 reset '1'   : in std_logic;
37      r1        : in std_logic;
38      r2        : in std_logic;
39      👓 grant1  : out std_logic;
40      👓 grant2  : out std_logic
41      );
42  end arbiter;
43
44  architecture beh of arbiter is
45    type states is ( st_idle1, st_idle2, st_grant1, st_grant2);
46    signal 👓 curr_state st_idle1, 👓 next_state st_idle2: states;
47    signal req1, req2, req11, req22, grant11, grant22 : std_logic;
48
49  begin
50    process(⏰ clk, 👓 reset '1')
```

Selecting a 👓 watchpoint or 🅿 icon next to a signal (or the signal itself) allows you to select the Watchpoint Setup dialog box from the popup menu. This dialog box specifies a watchpoint expression for the signal. See Setting a Watchpoint Expression, on page 40.

Selecting the green breakpoint icon to the left of the source line number causes that breakpoint to become armed when the run command is executed. See Run Command, on page 47.

Watchpoints Panel

The watchpoint panel lists all of the active and inactive watchpoints in the design. The icons of active watchpoints are highlighted in red with an adjacent marked check box. Inactive watch points are green regardless of their type. Values for the active watchpoints are listed. You can list all the watchpoints hierarchically based on a specific Root option.
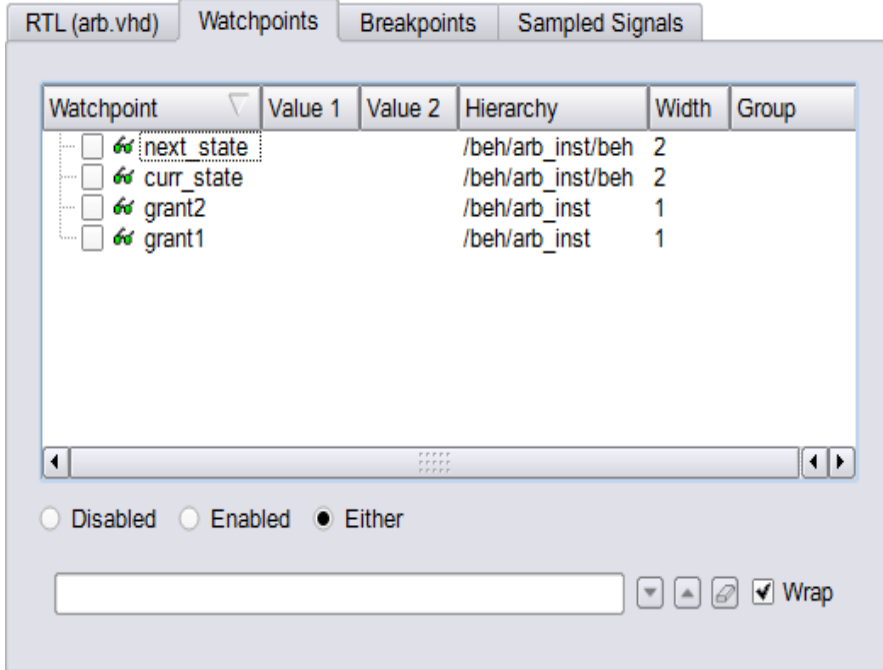
A watchpoint creates a trigger that is determined by the state of a signal in the design. The watchpoint can trigger either on the value of a signal or on a transition of a signal from one value to another.

Breakpoints Panel
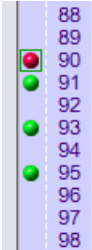
The breakpoints panel lists all of the active and inactive breakpoints in the design. The icons of active breakpoints are highlighted in red with an adjacent marked check box. Inactive watch points are green regardless of their type. Values for the active watchpoints are listed. You can list all the breakpoints hierarchically based on a specific Root option.
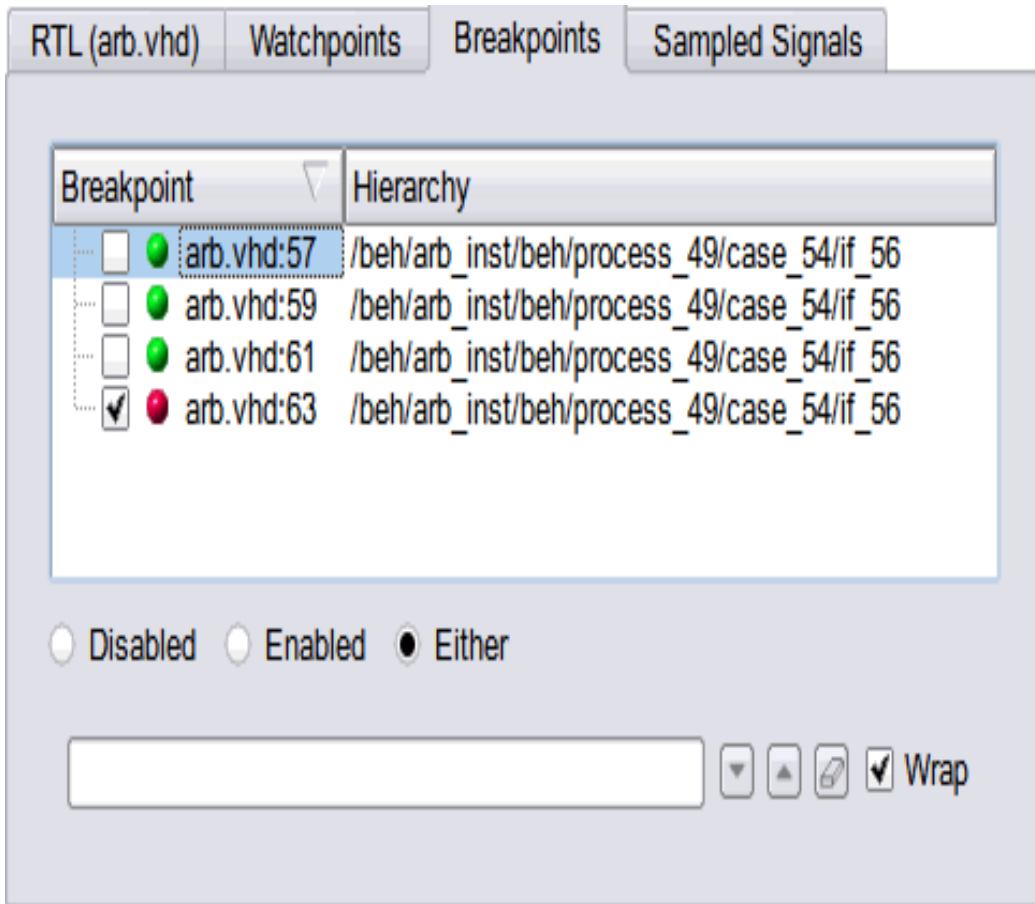
Breakpoints are a way to easily create a trigger that is determined by the flow of control in the design.

In both Verilog and VHDL, the flow of control in a design is primarily determined by if, else, and case statements. The control state of these statements is determined by their controlling HDL conditional expressions. Breakpoints provide a simple way to trigger when the conditional expressions of one or more if, else, or case statements have particular values.

The example below shows a VHDL code fragment and its associated breakpoints.

The four breakpoints correspond to these control flow equations:

- Breakpoint at line number 102:

  ```
  (op_code = "0100")
  ```

- Breakpoint at line number 104:

  ```
  (op_code = "0100") and (cc = '1')
  ```

- Breakpoint at line number 106:

  ```
  (op_code = "0100") and (cc = '1') and (result = zero)
  ```

- Breakpoint at line number 108:

  ```
  (op_code = "0100") and (cc = '1') and (result != zero)
  ```

Multiple Activated Breakpoints and Watchpoints

How breakpoints and watchpoints operate individually is described in the
Identify Instrumentor User Guide. Activated breakpoints and watchpoints also
interact with each other in a very specific way.

Multiple Activated Breakpoints

Each breakpoint is implemented as logic that watches for a particular event
in the design. When an instrumented design has more than one activated
breakpoint, the breakpoint events are ORed together. This effectively allows
the breakpoints to operate independently – only one activated breakpoint
must trigger in order to cause the sampling buffer to acquire its sample.

Multiple Activated Watchpoints

Each watchpoint is implemented as logic that watches for a specific event consisting of a bit pattern or transition on a specific set of signals. When an instrumented design has more than one activated watchpoint, the watchpoint events are ANDed together. This effectively causes the watchpoints to be dependent on each other – all activated watchpoint events must occur concurrently to cause the sampling buffer to acquire its sample.

For example, if watchpoint 1 implements (count == 23) and watchpoint 2 implements (ack == '1'), then activating these watchpoints together effectively creates a new watchpoint: (count == 23) && (ack == '1').

Combining Activated Breakpoints and Activated Watchpoints

When an instrumented design has one or more activated breakpoints and one or more activated watchpoints, the result of the OR of the breakpoint events and the result of the AND of the watchpoint events is ANDed together. The result of this AND operation is called the Master Trigger Signal. This ANDing effectively causes the breakpoints and watchpoints to be dependent on each other – one activated breakpoint and all activated watchpoint events must occur concurrently to cause the sampling buffer to acquire its sample.

As a result, a Master Trigger Signal event can be constructed that operates like a conditional breakpoint. For example, activating a breakpoint and the two watchpoints from the previous example produces a conditional breakpoint: (breakpoint event) && (count== 23) && (ack == '1').

Sampled Signals Panel

The Sampled Signals panel lists all of the sampled signals and their values when the sample trigger occurred. You can list all the sampled signals hierarchically based on a specific Root option.
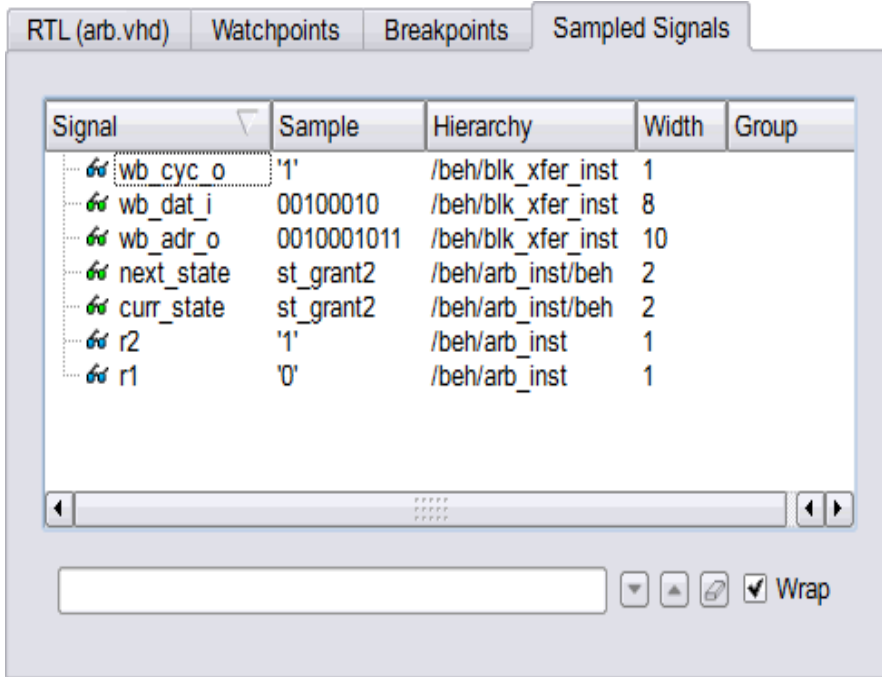
The sampling block is basically a large memory used to store all the sampled signals. During an active debugging session, the sampled signals are continually being stored in the sample block. When the sample block receives an event from the Master Trigger Signal event logic or the complex counter logic, the sampling block stops writing new data to the buffer and holds its contents. Eventually, the contents of the sample block are uploaded to the debugger for display and formatting.

TCL Window

TCL window is divided into Tcl console (Information) and Tcl shell windows. The Tcl console window displays status of the commands that have been executed, including those executed by menu selections and button clicks. The Tcl shell window allows you to type debugger commands and to view the results of command execution in the console window.

```
run -iice {IICE_0}
write vcd {C:/designs/bus_demo_1703/runtime_dir1/arb/debug/IICE_0.vcd} -iice IICE_0 -gtkwave
C:/designs/bus_demo_1703/runtime_dir1/arb/debug/IICE_0
Trigger detected, downloading samples...
%
```

TCL Script

To capture all text written to the Tcl shell window, use the log console command, and to capture all commands executed in the Tcl Script window, use the transcript command. To clear the text from the console window, use the clear command (see the *Debug Environment Reference Manual* for descriptions and syntax.

Debugger GUI Operations

In addition to the panels previously described, the graphical interface includes functions and icons as well as the top-level menu bar.

Debugger Main Menu

Clicking Debugger in the top menu bar displays a drop-down menu with the following selections:
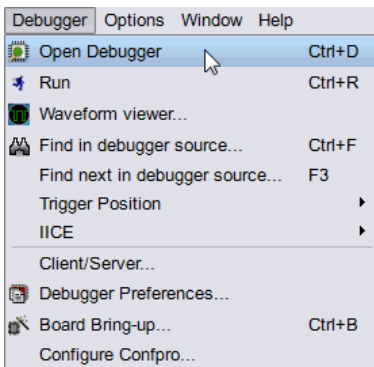
Run

Selecting Run initiates a run sequence by sending watchpoint and breakpoint activations to the IICE and waiting for a trigger. See Run Panel and Run Button, on page 25 for more information.

Waveform Viewer

The Waveform selection is enabled following a run sequence and displays the resulting waveform data in the specified waveform viewer. Clicking the Waveform viewer icon is equivalent to this selection. For more information, see Waveform Display, on page 77.

Find in debugger source

The Find dialog box locates text strings in the RTL source code. The dialog box is only valid for the RTL panel.

Find next in debugger source

Selecting Find next in debugger source locates the next occurrence of the specified search pattern.

Trigger Position

Sets the position of the trigger within the sample buffer. For more information on trigger position, see Sample Buffer Trigger Position, on page 49.

Configure State Machine

The Configure State Machine window is used to configure the state transitions. See Instrumentation, on page 16.

IICE

Brings up a submenu to allow the user to configure the Regular IICE, RTD IICE, and the Readback IICE settings.
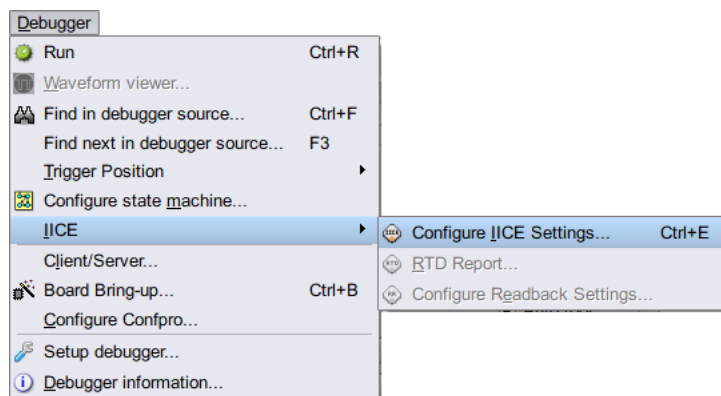
Configure IICE Settings brings up the Enhanced IICE Settings dialog box for enabling data compression (see Sampled Data Compression, on page 48), selecting a multiplexed group (see Selecting Multiplexed Instrumentation Sets, on page 44), and running a DTD self test (equivalent to clicking the Configure IICE Settings icon).
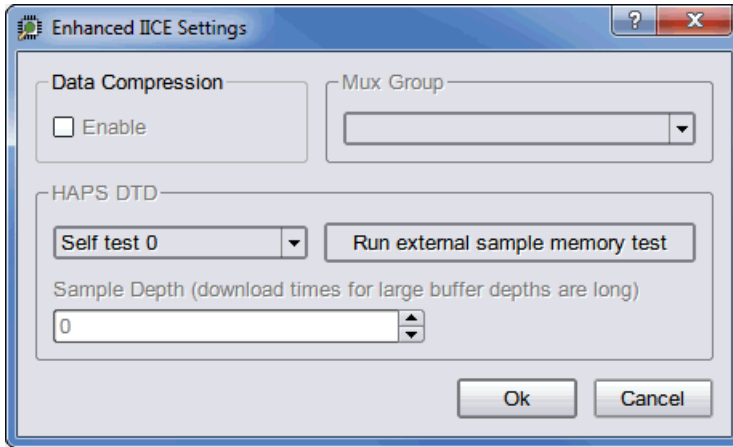
The Configure Readback Settings selections are not supported in the current version of the debugger.

Client/Server

Brings up the Configure Client/Server dialog box to set up the connection between the debugger and the instrumented device through the cable. For detailed information, see *Debugger Configuration,* on page 83.

Setup Debugger

Brings up the Setup and Preferences dialog box (equivalent to clicking the Setup debugger icon). For information, see Configuring the Debugger, on page 11.

Debugger Information

The instrumentation settings are displayed in the Debugger Information window (see Reviewing the Instrumentation Settings (Debugger Information), on page 10).

Basic Debugger Operations

Basic debugger operations include:

Activating/Deactivating an Instrumentation

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

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 it one or two HDL constant expressions. When a watched signal changes to the value of its watchpoint expression, a trigger event occurs.

A watchpoint is set on a signal by clicking-and-holding on the signal or the watchpoint icon next to the signal and then selecting the Set trigger expressions menu item to bring up the Watchpoint Setup dialog box.

A watchpoint is set on a partial bus signal by clicking-and-holding on the signal or the "P" icon next to the signal, selecting the partial bus group from the list displayed, and then selecting the Set Trigger Expressions menu item to bring up the Watchpoint Setup dialog box.

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.

To create a value watchpoint, assign a single, constant expression to the watchpoint. A value watchpoint triggers when the watched signal value equals the expression. The following trigger options are pre-defined:

- Trigger low - Triggers when the signal is low.

- Trigger high - Triggers when the signal is high.

- Trigger don't care - Triggers when the signal is unknown.

- Trigger 0->1 - Triggers when the signal is 0 to 1.

- Trigger 1->0 - Triggers when the signal is 1 to 0.

- Custom trigger - To set the specific trigger values.

In the example below, the signal is a 4-bit signal, and the watchpoint expression is set to "0010" (binary). Any legal VHDL or Verilog (as appropriate) constant expression is accepted.
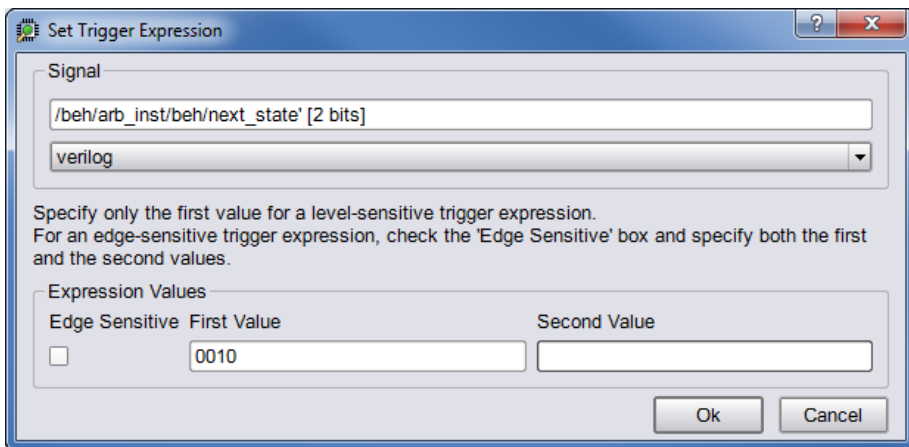
To create a transition watchpoint, assign a second constant expression to the watchpoint. A transition watchpoint triggers when the watched signal value is equal to the first expression during a clock period and the value is equal to the second expression during the next clock period. In the example below, the transition being defined is a transition from "0010" to "1011."
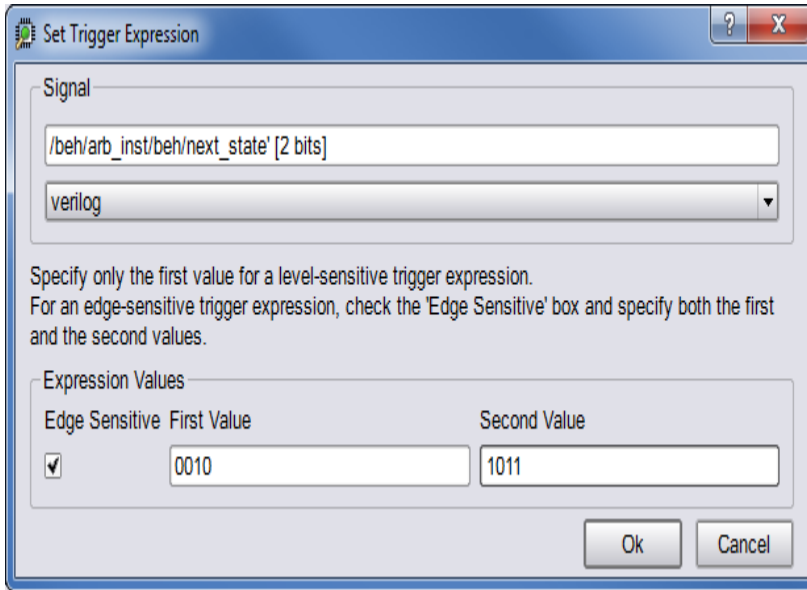
The VHDL or Verilog expressions that are entered in the Watchpoint Setup dialog box can also contain "X" values. The "X" values allow the value of some bits of the watched signal to be ignored (effectively, "X" values are don't-care values). For example, the above value watchpoint expression can be specified as "X010" which causes the watchpoint to trigger only on the values of the three right-most bits.

Hexadecimal values can additionally be entered as watchpoint values using the following syntax:

> **x"***hexValue***"**

> As shown, a hexadecimal value is introduced with an x character and the value must be enclosed in quotation marks.

Clicking OK on the Watchpoint Setup dialog box activates the watchpoint (the watchpoint or "P" icon changes to red) which is then armed in the hardware the next time the Run button is pressed.

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.
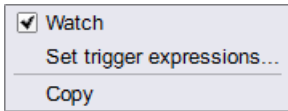
The Watch menu selection will have a check mark to indicate that the watchpoint is activated. Click on the Watch menu selection to toggle the check mark and deactivate the watchpoint.

Reactivating a Watchpoint

To reactivate an inactive watchpoint, 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. 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.

Active breakpoint (red)

Inactive breakpoint (green)
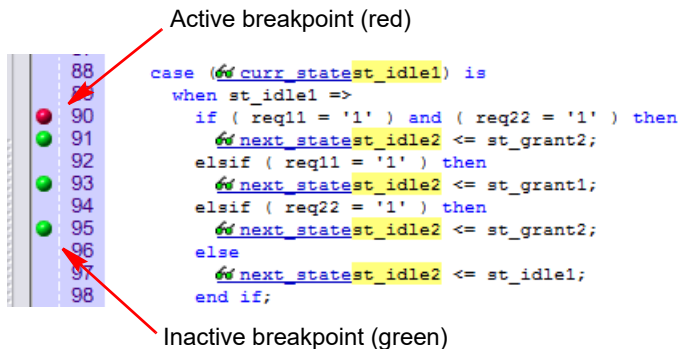
Selecting Multiplexed Instrumentation Sets

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 *Multiplexed Groups* in the *Identify Instrumentor User Guide*).

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.
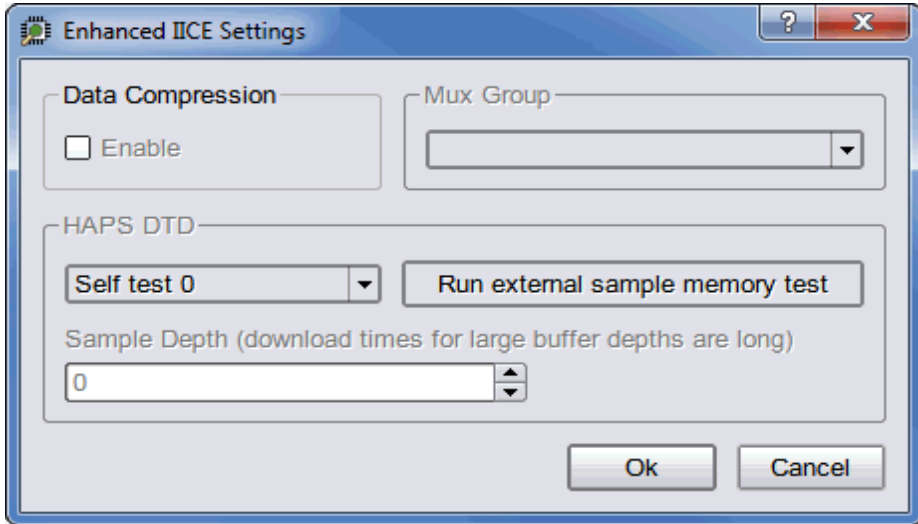
2. Use the drop-down menu in the Mux Group section to select the group number to be active for the debug session.

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 debugging 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 instru-mented instances when the breakpoint or signal is selected for activa-tion/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 breakpoint are instrumented.

To activate/deactivate instances of the "val" signal, select the watchpoint icon next to the signal. A list will pop-up with the two instrumented instances of the signal val available for activation/deactivation:

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

Either of these instances is activated/deactivated by clicking on the appropriate line in the list box to bring up the watchpoint menu shown in the following figure.

```
 5  entity repeated_unit is
 6     port(
 7          clr : in std_logic;
 8          val : out std_logic;
 9          clk            /rtl/cnt_inst0/val ▶    ✔ Sample
10        );              /rtl/cnt_inst1/val ▶      Watch
11  end repeated_unit;
12
13  architecture rtl of repeated_unit is                Set trigger expressions ...
14  begin
15
16     val <= clk and clr;
```
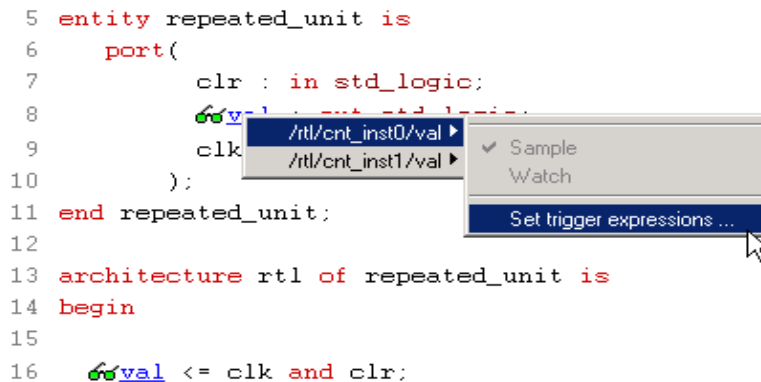
The color of the watchpoint icon is determined as follows:

- If no instances of the signal are activated, the watchpoint icon is green in color.

- If some, but not all, instances of the signal are activated, the watchpoint icon is yellow in color.

- If all instances are activated, the watchpoint icon is red in color.

For related information on folded hierarchies, see *Sampling Signal in a Folded Hierarchy* in the *Identify Instrumentor User Guide* and Displaying Data from Folded Signals, on page 52.

Activating/Deactivating a Folded Breakpoint

To activate/deactivate instances of the breakpoint on line 24, select the icon next to line number 24. A list will pop up with the two instrumented instances of the breakpoint available for activation/deactivation:

```
/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.

The color of the breakpoint icon is determined as follows:

- If no instances of the breakpoint are activated, the breakpoint icon is green.

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

- If all instances are activated, the breakpoint icon is red.

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 . To execute the Run command for multiple IICE units, enable the individual IICE units in the Run panel by checking their corresponding boxes, and then 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 source code window. 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.

Activated and triggered breakpoint   Sampled data (in yellow)

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).

Data compression is enabled from the project view by clicking the IICE icon to display the Enhanced Settings for IICE Unit dialog box and clicking the Enable check box in the Data Compression section or from the command prompt by entering 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 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
```

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, the exact relationship between the trigger event and the termination of the sampling can be controlled by the user. Currently, the debugger supports the following trigger positions relative to the sample buffer:

- Early

- Middle

- Late

Determining the correct setting for the trigger position is up to the user. For example, if the design behavior of interest usually occurs after a particular trigger event, set the trigger position to "early."

The trigger position can be changed without requiring the design to be re-instrumented or recompiled. A new trigger position setting takes effect the next time the Run command is executed.

Early Position

The sample buffer trigger position can be set to "early" so that the majority of the samples occurs after the trigger event. To set the trigger position to "early," use the Debugger->Trigger Position->Set trigger early in sample buffer menu selection or click on the Set trigger position to early in the sample buffer icon.

Middle Position

The sample buffer trigger position defaults to "middle" so that there is an equal number of samples before and after the trigger event. To set the trigger position to "middle," use the Debugger->Trigger Position->Set trigger to middle of sample buffer menu selection or click on the Set trigger position to the middle of the sample buffer icon.

Late Position

The sample buffer trigger position can be set to "late" so that the majority of the samples occurs before the trigger event. To set the trigger position to "late," use the Debugger->Trigger Position->Set trigger late in sample buffer menu selection or click on the Set trigger position to late in the sample buffer icon.

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 Cycle text field. You can manually change the cycle number by typing a number in the entry field. Also, the up and down arrows to the right of the cycle number increment or decrement the cycle number for each click.
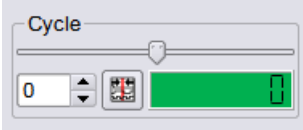
To reset the cycle number to the default position (the zero time position), use the Debug->Cycle->home menu selection or click on the Goto trigger event in sample history icon.

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.
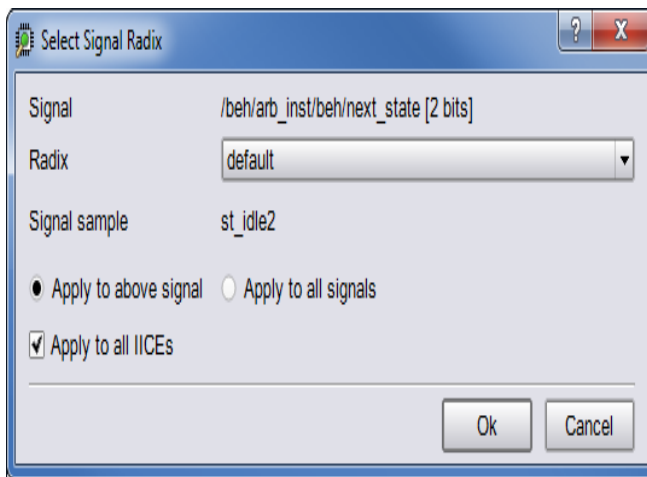
2. Select the desired radix from the Radix drop-down menu.

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 information 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:

Indicator of folded data   Data values for instances of folded signal **val**

For related information on folded hierarchies, see *Sampling Signals in a Folded Hierarchy* in the *Identify Instrumentor User Guide* and Activating/Deactivating Folded Instrumentation, on page 45.

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.

Composite bus value   Data values of partial buses

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.
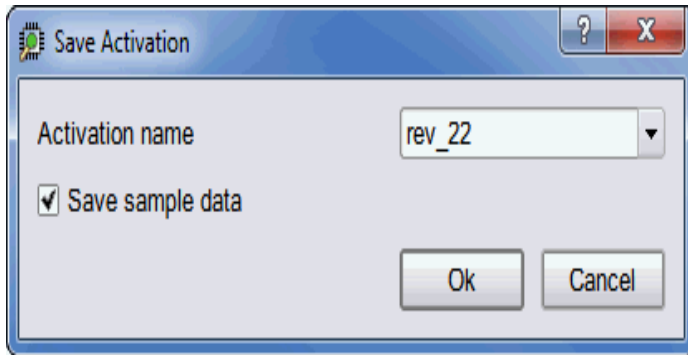
4. Enter (or select) an activation name in the adjacent field. Selecting an existing activation from the drop-down menu overwrites the selected activation.

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.
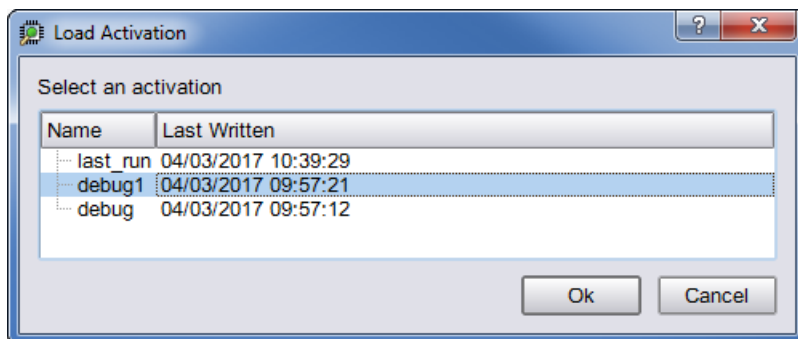
2. Click on the desired activation and select Load activation.

Autosaving Current Activation

By default, when you exit the debugger without explicitly saving an activation, 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 explicitly 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->Generaltab).

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 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). See Instrumentation, on page 16, for more information.

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 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, meaning 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, user can define complex trigger conditions using the advance triggering techniques. For instance, the state machine based triggering enables the user 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.

- State Machine based Triggering, on page 58

- Qualified Sampling, on page 69

- Always-Armed based Triggering, on page 70

- Sampled Data Compression, on page 70

- Cross-Triggering, on page 71

- Complex Counter, on page 72

- Remote Triggering, on page 73

State Machine based Triggering

State-machine triggering allows you to pre-instrument a variable-sized state machine that can be used to specify an ultimately flexible trigger condition. The state-machine triggering feature allows the creation of counter-based state machines from sequences of trigger conditions to create very effective triggers. 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.

Using State Machine Triggering in the Debugger

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.

- • **watch enable -condition** (*triggerCondition*|**all**) *signalName value1* [*value2* ...]

- • **watch disable -condition** (*triggerCondition*|**all**) *signalName*

- • **watch info** [**-raw**] *signalName*

The parameter *triggerCondition* is a list value conforming to the Tcl language. Examples are: 1, "1 2 3", {2 3}, or [list 1 2 3], quotes, braces, and brackets included, respectively. Alternatively, the keyword all can be specified to apply the setting to all trigger conditions.

The debugger watch info command reports status information about the signal. This information is returned in machine-processable form if the optional parameter -raw is specified.

Similarly for the debugger stop command:

- **stop enable -condition** (*triggerCondition*|**all**) *breakpoint*

- **stop disable -condition** (*triggerCondition*|**all**) *breakpoint*

- **stop info** [**-raw**] *breakpoint*

The semantics of the parameters are identical to the above descriptions.

The statemachine Command

During instrumentation, the number of states was previously defined using the -triggerstates option of the instrumentor iice controller command. Now, at debug time, you can define what happens in each state and transition depending on the pattern matches computed by the trigger conditions.

The debugger statemachine command is used to configure the trigger state machine with the desired behavior. This is very similar to the "advanced" trigger mode offered by many logic analyzers. As it is very easy to introduce errors in the process of specifying the state machine, special caution is appropriate. Also, a state-machine editor is available in the debugger user interface to facilitate state-machine development and understanding (see State-Machine Editor, on page 65). It is also important to note that the initial state for each run is always state 0 and that not all of the available states need to be defined.

The syntax forms of the debugger statemachine command are:

- **statemachine addtrans -from** *state* [**-to** *state*] [**-cond "**equation|**ti** *triggerInID***"**] [**-cntval** *integer*] [**-cnten**] [**-trigger**]

- **statemachine clear (-all**|*state* [*state* ...]**)**

- **statemachine info** [**-raw**] **(-all**|*state* [*state* ...]**)**

Subcommand statemachine addtrans

The debugger addtrans subcommand defines the transitions between the states. The options are as follows:

- **-from** *state* – specifies the state this transition is exiting from.

- **-to** *state* – specifies the state this transition goes to. If this is not given, it defaults to the state given in the -from option.

- **-cond "***equation***|ti***triggerInID***"** – specifies the condition or external trigger input under which the transition is to be taken. The default is "true" (i.e., the transition is taken regardless of input data; see below for more details).

- **-cntval** *integer* – specifies that if this transition is taken, the counter is loaded with the given value. Only valid when a counter is instrumented.

- **-cnten** – when this flag is given, the counter is decremented by 1 during this transition. Only valid when a counter is instrumented.

- **-trigger** – when this flag is given, a trigger occurs during this transition.

The order in which the transitions are added is important. In each state, the first transition condition that matches the current data is taken and any subsequent transitions in the list that match the current data are ignored.

Conditions

The conditions are specified using Boolean expressions comprised of variables and operators. The available variables are:

- **c0,... c***n*, where *n* is the number of trigger conditions instrumented. These variables represent the output bit of the respective trigger condition.

- **ti***triggerInID* – the ID (0 thru 7) of an external trigger input.

- **cntnull** – true whenever the counter is equal to 0 (only available when a counter is instrumented).

- *iiceID* – variable used with cross triggering to define the source IICE units to be included in the equation for the destination IICE trigger.

Operators are:

- Negation: not, !, ~

- AND operators: and, &&, &

- OR operators: or, ||, |

- XOR operators: xor, ^

- NOR operators: nor, ~|

- NAND operators: nand, ~&

- XNOR operators: xnor, ~^

- Equivalence operators: ==, =

- Constants: `0`, `false`, `OFF`, `1`, `true`, `ON`

Parentheses '(', ')' are recommended whenever the operator precedence is in question. Use the debugger statemachine info command to verify the conditions specified.

For example, valid expression examples are:

```
"c0 and c1"
"!(c1 or c2) and c3"
"c0 or ti4" (condition c0 or external trigger ID ti4)
```

Other Subcommands

The debugger statemachine clear command deletes all transitions from the states given in the argument, or from all states if the argument -all is specified.

The debugger statemachine info command prints the current state machine settings for the states given in the argument, or for the entire state machine, if the option -all is specified. If the option -raw is given, the information is returned in a machine-processible form.

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.

Convenience Functions

There are a number of convenience functions to set up complex triggers available in the file *InstallDir*/share/contrib/syn_trigger_utils.tcl which is loaded into the debugger at startup:

- **st_events** *condition integer* – Sets up the state machine to mimic counter mode events of the simple triggering mode as described above. The argument *condition* is a boolean equation setting up the condition, and *integer* is the counter value.

- **st_watchdog** *condition integer* – Same as st_events for watchdog mode.

- **st_cycles** *condition integer* – Same as above for cycles mode.

- **st_pulsewidth** *condition integer* – Same as above for pulsewidth mode.

- **st_B_after_A** *conditionA conditionB* [*integer***:=1**] – Sets up a trigger mode to trigger if *conditionB* becomes true anytime after *conditionA* became true. The optional *integer* argument defaults to 1 and denotes how many times *conditionB* must become true in order to trigger.

- **st_B_after_A_before_C** *conditionA conditionB conditionC* [*integer***:=1**] – Sets up a trigger mode to trigger if *conditionB* becomes true after *conditionA* becomes true, but without an intervening *conditionC* becoming true (same as the second example above). The optional *integer* argument defaults to 1 and denotes how many times *conditionB* must become true without seeing *conditionC* in order to trigger.

- **st_snapshot_fill** *condition* [*integer*] – Uses qualified sampling to sample data until sample buffer is full. The argument *condition* is a boolean equation defining the trigger condition, and *integer* is the number of samples to take with each occurrence of the trigger (default 1).

- **st_snapshot_intr** *condition* [*integer*] – Uses qualified sampling to sample data until manually interrupted by an debugger stop command. The argument *condition* is a boolean equation defining the trigger condition and *integer* is the number of samples to take with each occurrence of the trigger (default 1).

Refer to the file syn_trigger_utils.tcl mentioned above for the implementation of these trigger modes using the debugger statemachine command. Users can add their own convenience functions by following the examples in this file.

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. The IICE being triggered is referred to as the "destination" IICE; the other IICE units are referred to as the "source" IICE units.

Multiple IICE designs allow triggering and sampling of signals from different clock domains. With an asynchronous design, a separate IICE unit can be assigned to each clock domain, triggers can be set on signals within each IICE unit, and then the IICE units scheduled to trigger each other on a user-defined sequence using cross triggering. In this configuration, each IICE unit is independent and can have unique IICE parameter settings including sample depth, sample/trigger options, and sample clock and clock edges.

Cross triggering is supported in all three IICE controller configurations (simple, complex counter, and state-machine triggering) and all three configurations make use of state machines.

Cross triggering is enabled in the instrumentor (cross triggering can be selectively disabled in the debugger). To enable a destination IICE unit to accept a trigger from a source IICE unit, enter the following command in the instrumentor console window (by default, cross triggering is disabled):

```
iice controller -crosstrigger 1
```

For cross triggering to function correctly, the destination and the contributing source IICE units must be instrumented by selecting breakpoints and watchpoints. Concurrently run these units either by selecting the individual IICE units and clicking the RUN button in the debugger project view or by entering one of the following commands in the debugger console window:

```
run -iice all
```

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

When simple- or complex-counter triggering is selected in the destination IICE controller, the following debugger cross-trigger commands are available:

- The following debugger command causes the destination IICE to trigger normally (the triggers from source IICE units are ignored).

```
iice controller -crosstriggermode DISABLED
```

- The following debugger command causes the destination IICE to trigger when any source IICE triggers or on its own internal trigger.

```
iice controller -crosstriggermode ANY
```

- The following debugger command causes the destination IICE to trigger when all source IICE units and the destination IICE unit have triggered in any order.

```
iice controller -crosstriggermode ALL
```

- The following debugger commands cause the destination IICE to trigger after the source IICE unit triggers coincident with the next destination IICE internal trigger.

```
iice controller -crosstriggermode after -crosstriggeriice iiceID
iice controller -crosstriggermode after -crosstriggeriice all
```

The first debugger command uses a single source IICE unit (*iiceID*), and the second debugger command requires all source IICE units to trigger.

When state-machine triggering is selected, the state machine must be specified with at least three states (three states are required for certain triggering conditions, for example, when the destination IICE is in Cycles mode and you want to configure the destination IICE to trigger after another (source) IICE.

With state-machine triggering, the following debugger statemachine command sequences are available in the debugger console window:

- The following debugger command sequence is equivalent to disabling cross triggering. The destination IICE triggers on its own internal trigger condition (c0).

```
statemachine clear -all
statemachine addtrans -from 0 -cond "c0" -trigger
```

- In the following debugger command sequence, the destination IICE waits for *iiceID* to trigger and then triggers on its own internal trigger condition (c0). This sequence implements the "after *iiceID*" functionality of the simple- and complex-counter triggering modes.

```
statemachine clear -all
statemachine addtrans -from 0 -to 1 -cond "iiceID"
statemachine addtrans -from 1 -to 0 -cond "c0" -trigger
```

- In the following debugger command sequence, the destination IICE
  triggers when the last running IICE triggers.

  ```
  statemachine clear -all
  statemachine addtrans -from 0 -cond "c0 and iiceID and iiceID1
      and iiceID2" -trigger
  statemachine addtrans -from 0 -to 1 -cond "c0"
  statemachine addtrans -from 1 -to 0 -cond "iiceID and iiceID1
      and iiceID2" -trigger
  ```

- In the following debugger command sequence, the destination IICE
  waits for all the other running source IICE units to trigger and then
  triggers on its own internal trigger condition (c0).

  ```
  statemachine clear -all
  statemachine addtrans -from 0 -to 1 -cond "iiceID and iiceID1
      and iiceID2"
  statemachine addtrans -from 1 -cond "c0" -trigger"
  ```

The incorporation of a counter in the state-machine configuration is similar
to the use of a counter in non-cross trigger mode for a state machine.

State-Machine Editor

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.

To bring up the state-machine editor in the debugger, click the
Configure state machine icon in the debugger toolbar. Note that the icon
will be disabled if state-machine triggering was not selected in the
instrumentor when the design was instrumented and that an error message
will be generated if more than 10 states are defined. Clicking the icon
displays the Configure State Machine dialog box for the selected IICE. See
Instrumentation, on page 16.
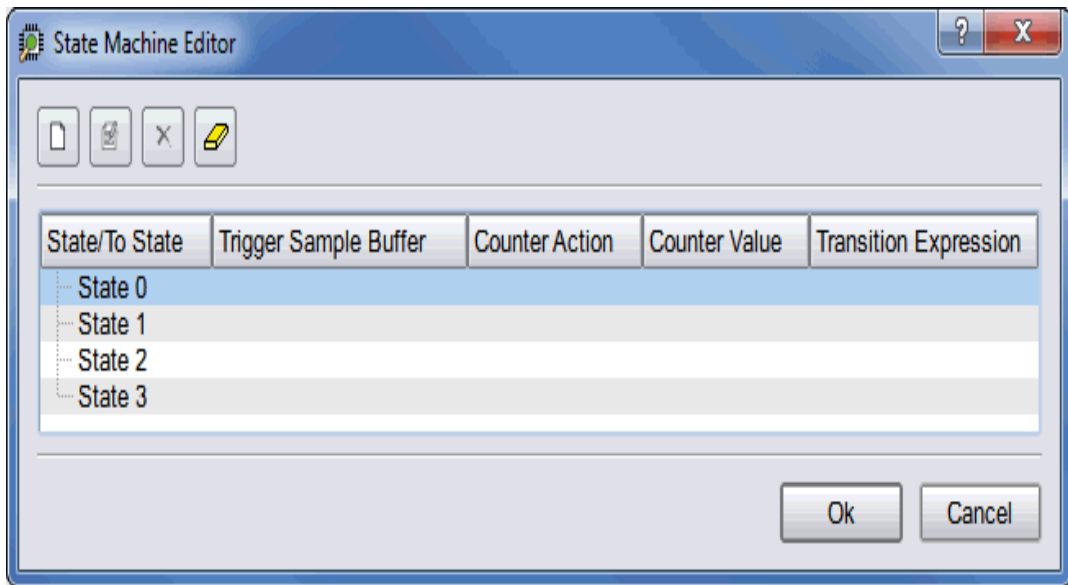
State Machine Examples

The state-machine triggering feature allows the creation of counter-based state machines from sequences of trigger conditions to create very effective triggers. 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.

Building a Complex State-machine Trigger

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. A common design configuration is to trigger when a specific sequence of events occurs which, in turn, causes data collection to stop and the sample data to be downloaded by the corresponding debugger executable from the FPGA. 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 done:

- In the instrumentor graphical user interface, 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.
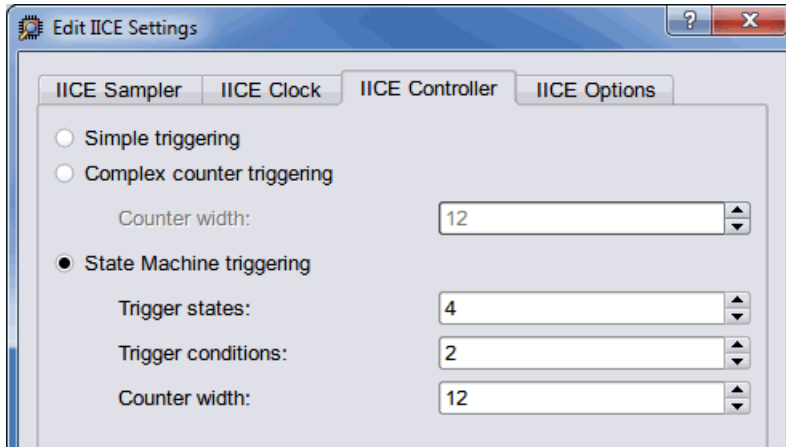
2. Build the state machine trigger from the debugger console window. The
   following debugger command sequence is an example.

   ```
   statemachine addtrans -from 0 -to 1 -cond c0 -cntval 7 -trigger
   statemachine addtrans -from 1 -to 0 -cond "cntnull"
   statemachine addtrans -from 1 -to 1 -cnten -trigger
   ```

   Note that in the last debugger statemachine command, the -to 1 can be
   omitted (unnecessary because there is no change in state) and that
   because the -from states are the same in the second and third
   commands, execution falls through to the third command when the
   second condition is not true.

3. Once the state-machine trigger is created, use the debugger statemachine
   info -all command to display and review the state-machine transitions.

The debugger state-machine and state-machine transition editors allow:

- Graphical entry of state machines

- Editing of state transitions and trigger events

- Conditions to be combined with each other or with a counter

- Counter mode selection of up, down, or initialized to any value

State-machine Triggering with Tcl Commands

The IICE can be configured using Tcl commands entered from both the instrumentor and debugger console windows. Some of the example commands are as follows:

- To delete the state transitions from each IICE, use the following debugger command:

  ```
  statemachine clear -iice all
  ```

- To enable complex counter triggering, use the following instrumentor command:

  ```
  iice controller complex
  ```

- To set the counter width, use the following instrumentor command:

  ```
  iice controller -counterwidth 8
  ```

- To configure an IICE for state-machine triggering, use the following instrumentor command sequence:

  ```
  iice controller -iice IICE statemachine
  iice controller -iice IICE -counterwidth 4
  iice controller -iice IICE -triggerconditions 2
  iice controller -iice IICE -triggerstates 2
  ```

  In addition to state-machine triggering, the above instrumentor commands set the number of trigger conditions to 2 and the number of trigger states to 2.

- To enable cross triggering, use the following instrumentor command:

  ```
  iice controller -crosstrigger 1
  ```

- Similarly, to configure the sample depth, use the following instrumentor command:

  ```
  iice sampler -depth 2048
  ```

Note that the only option for buffer type is internal_memory.

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. As a prerequisite in the instrumentor GUI:

   – From the Edit IICE Settings dialog box, select the IICE Controller tab, click the State Machine triggering radio button, and enter a value in the Counter width field to define the width of the sample buffer.

   – Select the IICE Sampler tab and enable the Allow qualified sampling check box.

2. Clicking Setup debugger icon or selecting from Debugger -> Setup Debugger, open the Setup and Preferences dialog box. 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*.
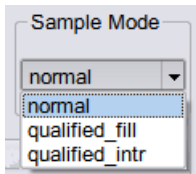
3. From the debugger GUI, click on the Configure state machine icon or click the FSM button from the Debugger -> Setup debugger -> Instrumentation tab and define the state-machine trigger event.

When you click Run, the sample buffer begins accumulating data when the trigger event occurs and stops accumulating data after the specified number of samples is reached.

You can also perform qualified sampling using equivalent debugger Tcl commands. The following debugger example command sequence samples the data every *N* cycles beginning with the first trigger event.

```
iice sampler -samplemode qualified_fill
statemachine clear -iice IICE -all
statemachine addtrans -iice IICE -from 0 -to 1
   -cond "true" -cntval 0
statemachine addtrans -iice IICE -from 1 -to 2
   -cond "c0" -cntval 15 -trigger
statemachine addtrans -iice IICE -from 2 -to 2
   -cond "! cntnull" -cnten
statemachine addtrans -iice IICE -from 2 -to 2
   -cond "cntnull" -cntval 15 -trigger
```

Always-Armed based Triggering

The Allow always-armed triggering check box, when checked in the instru-
mentor, 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.

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

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).

Data compression is enabled from the project view by clicking the IICE icon to
display the Enhanced Settings for IICE Unit dialog box and clicking the Enable
check box in the Data Compression section or from the command prompt by
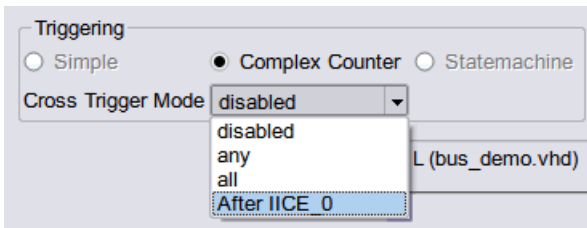entering the following command:

```
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 or always-armed sampling.

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).

Cross triggering for an IICE unit is enabled in the instrumentor by selecting the Allow cross-triggering in IICE check box on the IICE Options tab for the local IICE unit. The cross-trigger mode is selected from the drop-down menu in the debugger as shown below.

The drop-down menu selections are as follows:

Menu Selection	Function
disabled	No triggers accepted from external IICE units (event trigger can only originate from local IICE unit)
any	Event trigger from local IICE unit occurs when an event at any IICE unit, including the local IICE unit, occurs
all	Event trigger from local IICE unit occurs when all events, irrespective of order, occur at all IICE units including the local IICE unit
After IICE	Event trigger from local IICE unit occurs only after the event at selected external IICE unit has occurred (external IICE units are individually listed)

If the drop-down menu does not display, 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.

Complex 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 Edit IICE Settings dialog box or using the instrumentor iice controller command.

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.

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. See Instrumentation, on page 16.

events Mode

In the events mode, the number of times the Master Trigger Signal logic produces an event is counted. When the nth Master Trigger Signal event occurs, the complex counter sends a trigger event to the sample block. For example, this mode could be used to trigger on the 12278th time a collision was detected in a bus arbiter.

cycles Mode

In the cycles mode, the complex counter sends a trigger event to the sample block on the nth cycle after the first Master Trigger Signal event is received. The clock cycles counted are from the clock defined for sampling. For example, this mode could be used to observe the behavior of a design 2,000,000 cycles after it is reset.

watchdog Mode

In the watchdog mode, the counter sends a trigger event to the sample block if no Master Trigger Signal events have been received for n cycles. For example, if an event is expected to occur regularly, such as a memory refresh cycle, this mode triggers when the expected event fails to occur.

pulsewidth Mode

In the pulsewidth mode, the complex counter sends a trigger event to the sample block if the Master Trigger Signal logic has produced an event during each of the most recent n consecutive cycles. For example, this mode can be used to detect when a request signal is held high for more than n cycles thereby detecting when the request has not been serviced within a specified interval.

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.

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.

You can selectively set the remote trigger to:

- trigger all IICEs in all debugger executables
- trigger all IICEs in a specific debugger executable
- trigger a specific IICE in a specific debugger executable

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.

Remote triggering is a scripting application. The IICE/debugger targets are defined by the debugger remote_trigger command (see the command description in the *Debug Environment Reference Manual*).

As an example, the debugger scripting sequence

```
run ; remote_trigger -pid 12
```

waits for the trigger condition in the active IICE and then sends a trigger to all
IICE units in the debugger executable identified by process ID 12.

Importing External Triggers

An import external trigger capability can be used with trigger signals origi-
nating from on-board logic external to the FPGA or from an IICE module in a
second FPGA. For information on using this feature with state-machine
triggering, see the *Importing External Triggers* application note available on
the Synopsys website.

Verdi-Identify Flow

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

1. Instrument the design with the essential signal list (see *Instrumenting
   the Verdi Signal Database* in the *Identify Instrumentor User Guide*).

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.

Debugging on a Different Machine

It is not unusual for the instrumentation phase and the debugging phase to 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 the project file (*projectName*.prj) and the following files to the lab machine:

- Implementation folder (for example, rev_1); it is not necessary to copy the contents of the folder

- syn.db file

- instr.db file

- orig_sources files

Because the instrumentor/debugger 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 project, 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 implementation directory so that when you copy the implementation 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 is set before compiling the instrumented design by selecting Options->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 project file (*projectName*.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 (for information on file encryption, see *Including Original HDL Source* in the *Identify Instrumentor User Guide*).


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.
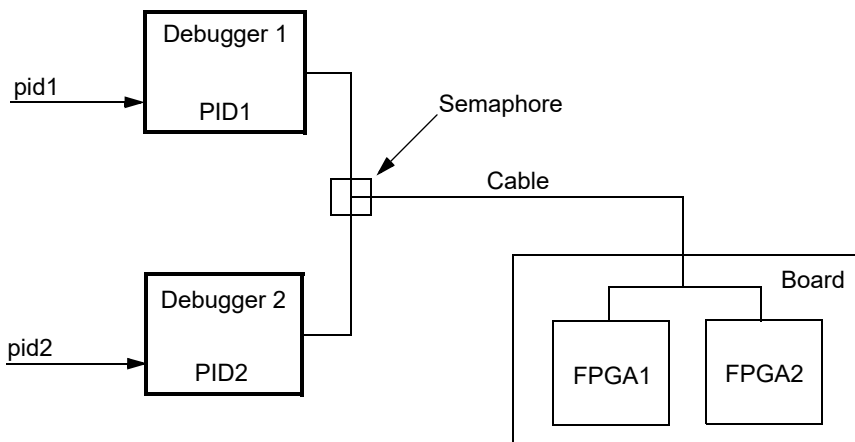
Waveform Display

The waveform display control displays the sampled data in a waveform. For information about setting up the waveform, see Waveform, on page 21.

After running the debugger, the selected waveform viewer is displayed by selecting Debugger->Waveform viewer from the menu or by clicking the Waveform viewer icon in the toolbar. All sampled signals in the design are included in the waveform display. Two additional signals are added to the top of the display when enabled by their corresponding check boxes. The first signal, identify_cycle, reflects the trigger location in the sample buffer. The second signal, identify_sampleclock, is a reference that shows every clock edge. The following figure shows a typical waveform view with the identify_cycle and identify_sampleclock signals enabled (highlighted in the figure).
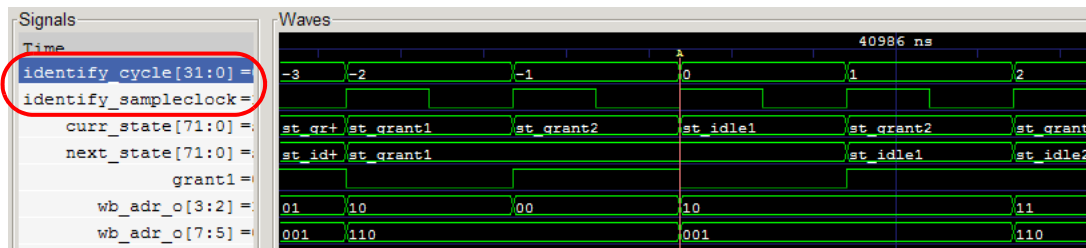
If you select a waveform viewer from the Waveform preference dialog box that is not installed, an error message is displayed when you attempt to invoke the viewer. To install the waveform viewer:

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

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

3. 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.

To invoke the viewer after running the debugger, select Debugger->Waveform viewer or click on the Waveform viewer icon.

Generating the Fast Signal Database

The debugger is used to generate the fast signal database (FSDB) for the Verdi platform and for display by 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 39 in the Identify Instrumentor User Guide).

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 VerdinWave 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 into the Verdi platform.

SYNOPSYS®

Silicon to Software™

C H A P T E R   2

Connecting to the Target System

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

- Basic Communication Connection, on page 80
- JTAG Communication Interface, on page 91

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, on page 80

- Debugger Configuration, on page 83

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.

Cable Type

The cable type is selected from the Cable Settings dialog box in the Setup and Preferences window of the debugger. A drop-down menu allows cable selection based on the specified technology.
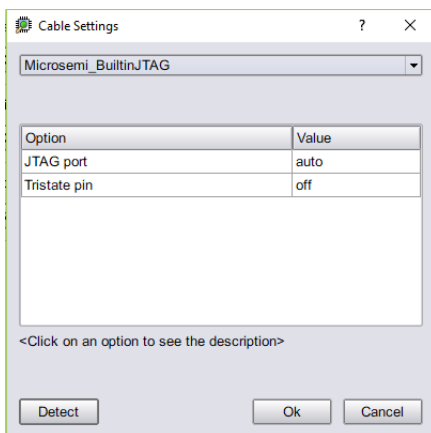
The following table lists the correspondence between cable-type setting and the available vendor cables supported by the Identify debugger.

Cable Type Settings	Compatible Hardware Cables
Microsemi_BuiltinJTAG	Microsemi FlashPro, FlashProLite, or FlashPro3
JTAGTech3710	JTAGTech3710
Catapult_EJ1	Standard Ethernet cable in an IP network

If you are using the command interface, set the com command's cabletype option to Microsemi_BuiltinJTAG, JTAGTech3710, Catapult_EJ1, or demo according to the cable being used.

Byteblaster Cable Setting

To configure a ByteBlaster cable, click the Cable Settings button to display the Cable Settings dialog box and select the appropriate port from the Value drop-down menu (see following figure).
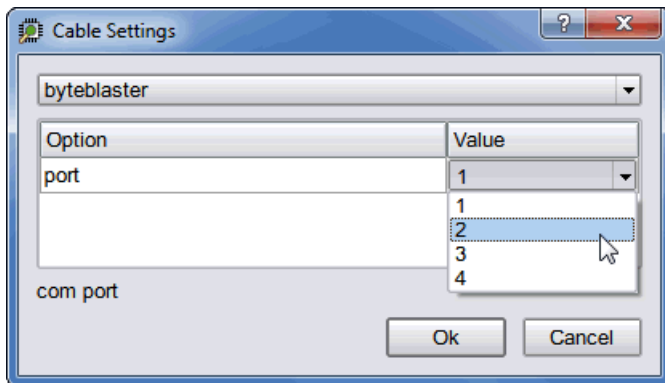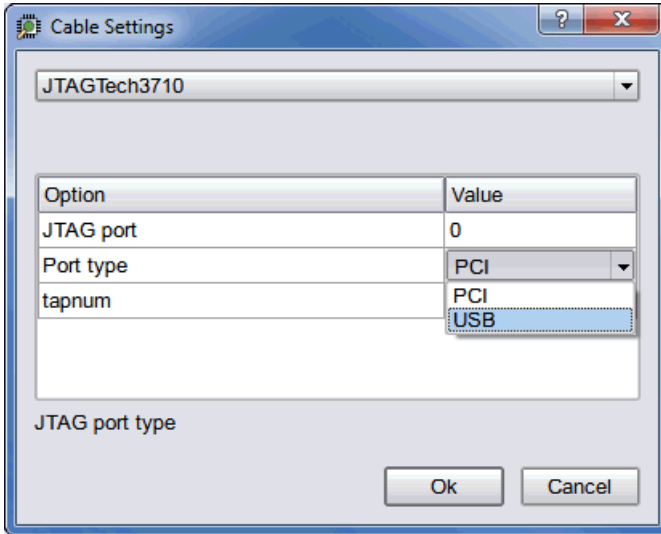

If you are using the command interface, set the com command's cableoptions byteblaster_port option to 1 (lpt1), 2 (lpt2), 3 (lpt3), or 4 (lpt4). Different computers have their lpt ports defined for different address ranges so the port you use depends on how your computer is configured.

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, you can use the setsys set command variable lpt_address to set the hardware port address (for example, setsys set lpt_address 0x0378 defines port lpt1).

JTAGTech3710 Cable Settings

To configure a JTAGTech3710 cable, click the Port Settings button to display the Configure Port Settings dialog box (see following figure) and enter the corresponding parameters (type, port, and tap number). If you are using the command interface, use the com command's cableoptions option to set the cable-specific parameters – JTAGTech_type (takes values PCI and USB; default is PCI), JTAGTech_port (takes values 0, 1, 2, ...; default value is 0), and JTAGTech_tapnum (takes values 1, 2, 3, or 4; default is 1).

Microsemi Actel_BuiltinJTAG cable Settings

To configure a Microsemi FlashPro, FlashProLite, or FlashPro3 cable, simply select the Microsemi_BuiltinJTAG setting from the Cable type drop-down menu. If you are using the command interface, you can additionally use the com command's cableoptions option to set the tristate pin parameter (see the com command cableoptions option in the *Debug Environment Reference Manual* for the parameter syntax).

Catapult EJ-1 Settings

To configure a Catapult EJ-1 cable, select the Catapult_EJ1 setting from the Cable type drop-down menu. Click the Port Settings button to display the Configure Port Settings dialog box and enter the host IP address.
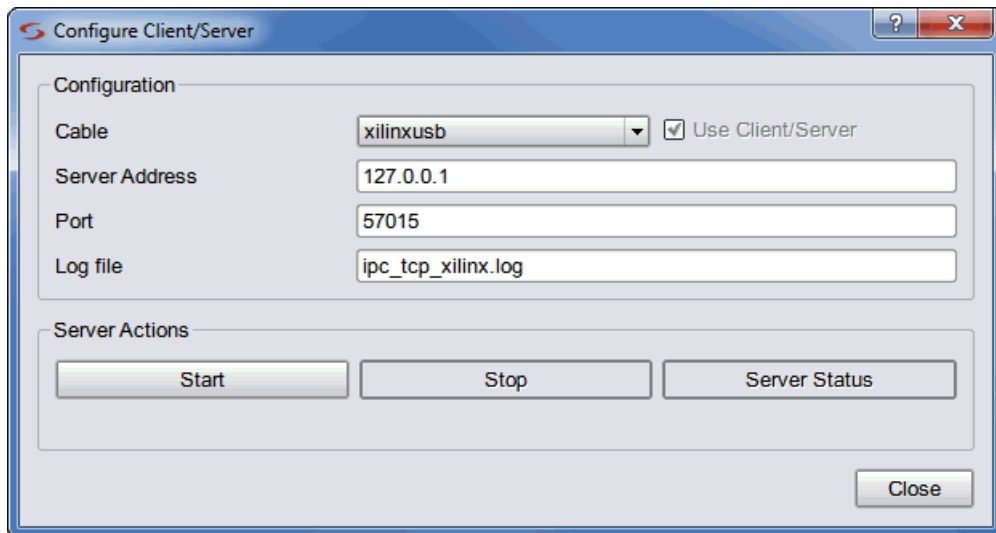
Debugger Configuration

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 described in Debugger Communications Settings, on page 80, the following additional requirements must be met to ensure proper communications.

Local Client-Server Configuration

The following figure shows a typical local server configuration.
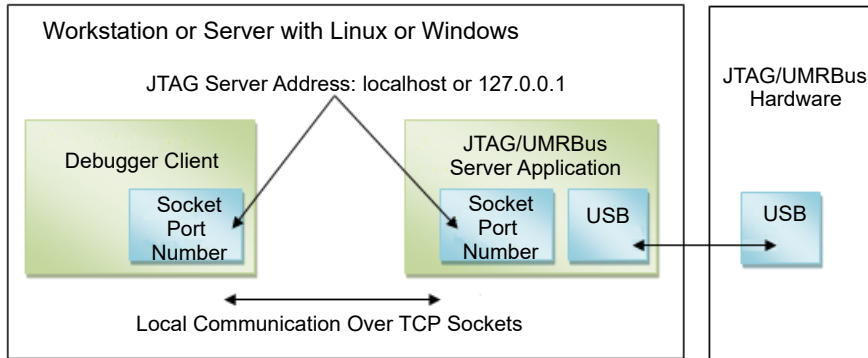
The client-server configuration is set from a dialog box available by selecting Configure Client Server in the Communications tab from the Setup and Preferences window. The default settings are usually correct for most configurations and require changing only when the default server port address is already 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 Remote Client-Server Configuration, on page 86).

The available configure client-server settings in the dialog box are defined in the following table:

| Setting	Function
Cable	The type of interface cable (see *Cable Type, on page 80*).
Use Client/Server	Check box for enabling client-server communications when the cable type is USB-based UMRBus (limited to HAPS-70 systems).
Server Address	The address of the server. The address localhost (or 127.0.0.1) is used when the debugger is run on the same machine connected to the FPGA device. The server address is set to the name or tcp/ip4 address of the machine connected to the FPGA device/board when the debugger is run from a different machine.
Port	The port number of the server. For all Microsemi cable types, the default port number is 58015. Change the server port setting when there is a conflict with another tool on the machine.
Log file	The name of the log file.
Start	Server control button for starting the server in stand-alone mode. Activating the button adds a start entry to the log file.
Stop	Server control button for stopping the server in stand-alone mode. Activating the button adds a stop entry to the log file.
Server Status	Adds a start/stop entry to the log file.

To establish a local client-server connection:

1. Start the debugger and open the Configure Client/Server settings dialog box. Select the cable type from the Cable drop-down menu and make sure that the server address is either 127.0.0.1 or localhost.

2. 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. Known ports for system components range from 0 through 1023, and registered ports for software components range from 1024 through 49151. The dynamic and/or private ports range from 49152 through 65535. If possible, use a port address within this range where there is usually ample room.

3. For a local client-server connection, click the OK button when satisfied with the server address and port values. Do not use the Start button as this creates a *standalone* server which must then be manually stopped with the Stop button.

4. 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.

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

Remote Client-Server Configuration

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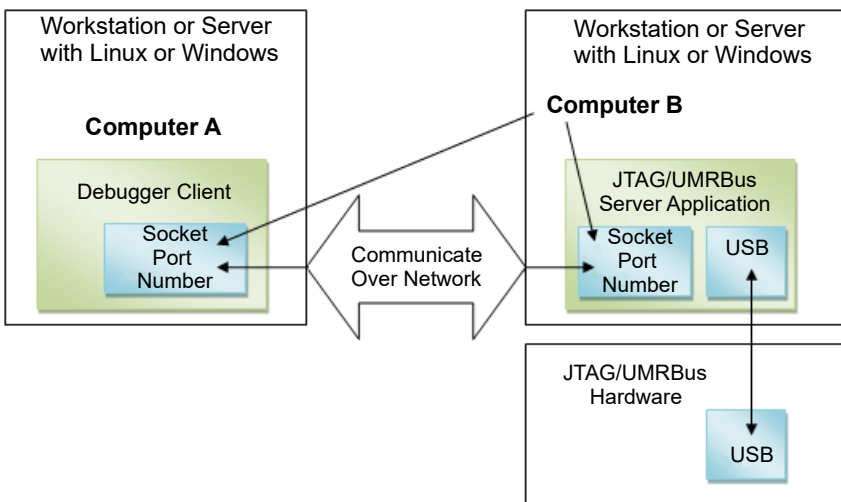
The Identify debugger uses a client-server architecture to communicate with the device. Client-server architecture lets you work remotely with the Identify debugger using Ethernet as the backbone for the client-server communication.

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). You use the Configure

Client/Server dialog box described in the previous section to set up both the client and server machines so that you can remotely debug the design. 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) of the design 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 corresponding 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.

   – Start the server in stand-alone mode by clicking the Start button in the dialog box. 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:

jtag_server set -addr {*hostName*/*IP_address***} -port {***serverPort***} -logf {***logFileName***}**

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

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 instru-mented 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 Microsemi device system either from the debugger GUI or from the command line. Both methods are described below.

  1. Start the debugger on the HAPS 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.

5. 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

There are two connections: cable-to-board and cable-to-host. The latest cable
types use a USB connector to interface with the host and require a USB
driver to be installed (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, you can use the setsys set command variable
lpt_address to set the hardware port address (for example, setsys set lpt_address
0x0378 defines port lpt1).

The cable-to-board connection requires a type of mating connector or inter-
face pod to connect to the board containing the device. If you are using
parallel JTAG cables, see JTAG Hardware in Instrumented Designs, on
page 92.

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 95).

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 following figure), or multiple devices can be connected in a serial chain.
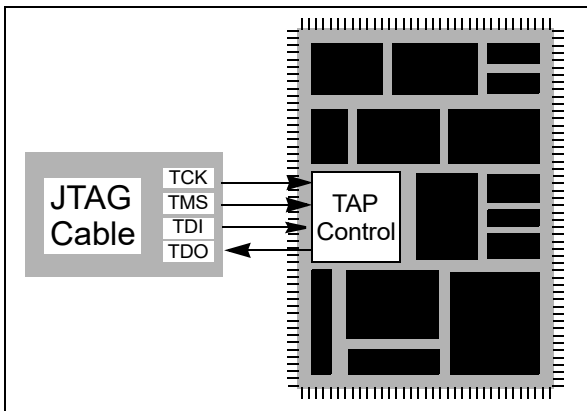
The following topics are included in this section:

- JTAG Communication Block, on page 91
- JTAG Hardware in Instrumented Designs, on page 92
- JTAG Communication Debugging, on page 101

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). See Chapter 2, *Connecting to the Target System*, for more information on the JTAG controller.

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. Unfortunately, not all devices 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 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 also can 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 following figure).
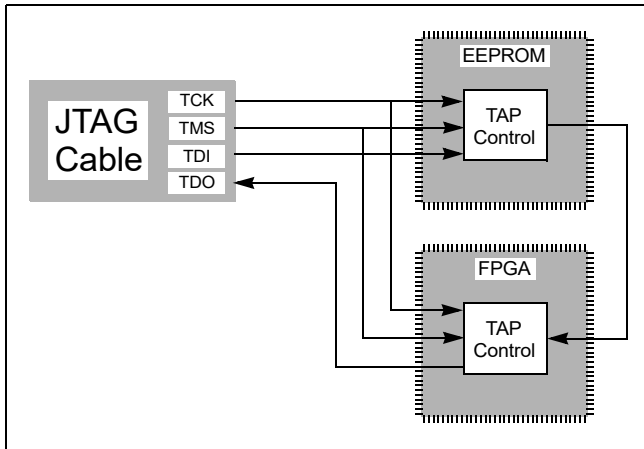
This configuration is well suited to the debugger and works just like any other serially connected chain.

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 *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 simple guides:

- 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.
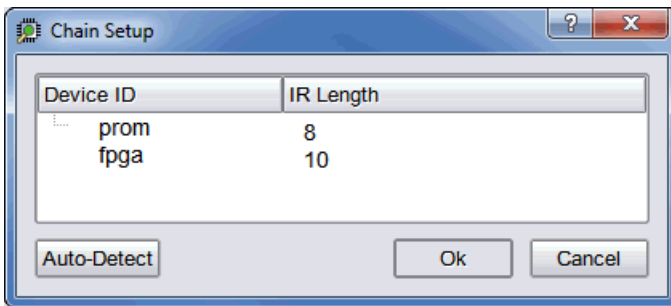
To enable the debugger to properly communicate with the target device, the device chain must be configured correctly. If, for some reason, the JTAG chain cannot be successfully configured, you must manually specify the chain through a series of chain instructions entered in the console window.

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 Microsemi 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.

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, perform the following steps:

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

   use -forced 1.

2. Define the new cable option. For example:

   com cableoptions Microsemi_BuiltinJTAG_port usb32388

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

Adding Multiple FlashPro Programmers through GUI

This features enables the user to select multi FlashPro devices to debug respective projects for Microsemi devices using identify debugger.

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 and click Cable Settings.tab.

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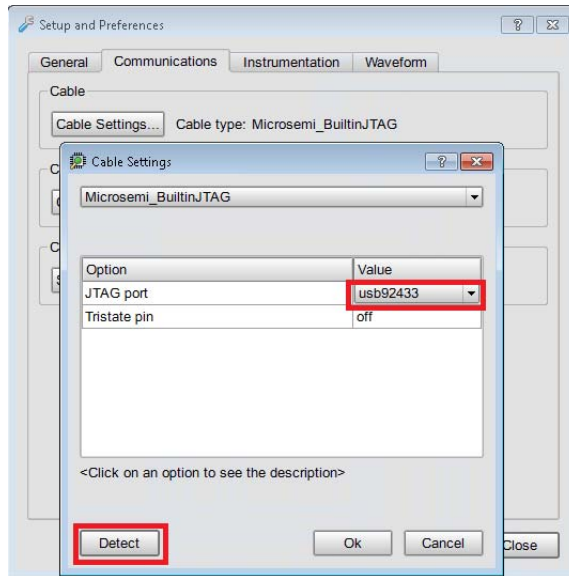
6. Select Microsemi_BuiltinJTAG from the Cable Type drop-down list.

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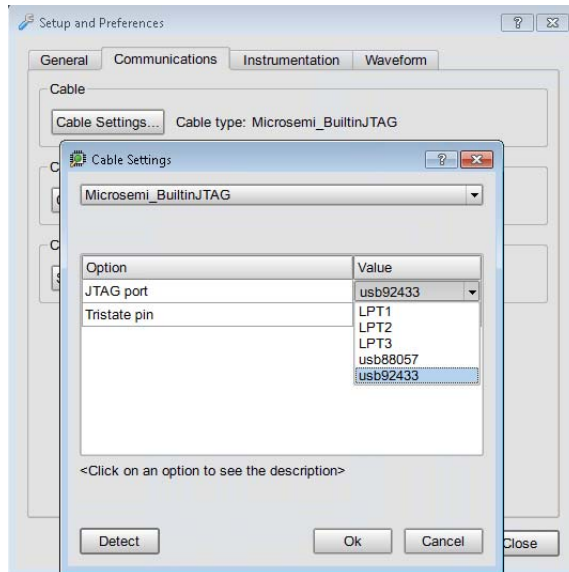
8. Perform the regular identify debugger steps to debug the selected project.

9. Follow the same steps to debug the another project.

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.

Below is a list of communication related error messages with some additional explanations.

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. Please check the cable connection.
  It is likely that the debugger is unable to communicate with the instrumented chip. This error is usually a cable connection problem, or the cable type is not set correctly.

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

- ERROR: Communication is returning incorrect IR data. Please 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 connection 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 integrity 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 instrumented 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 instrumentation 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. Please 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). Please review your chain configuration.

- ERROR: Communication with device number XX is not correct. Please 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.

Index