

Identify Debugger User Guide

March 2015

<https://solvnet.synopsys.com>

SYNOPSYS®

Copyright Notice and Proprietary Information

© 2015 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.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only.

Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

“This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____.”

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.

Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, CODE V, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclipse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIMplus, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, Total-Recall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (sm)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A
March 2015

Contents

Chapter 1: Using the Debugger

Configuring and Invoking the Debugger	10
Reviewing the Instrumentation Settings	10
Changing the Communication Settings	10
Reviewing the JTAG Chain Settings	11
Saving the Debugged Design	12
Invoking the Debugger	12
Debugger Windows	13
IICE Instrumentation Window	14
Console Window	16
Project Window	17
Commands and Procedures	18
Opening and Saving Projects	18
Executing a Script File	19
Activating/Deactivating an Instrumentation	19
Selecting Multiplexed Instrumentation Sets	23
Activating/Deactivating Folded Instrumentation	24
Run Command	26
Sampled Data Compression	28
Sample Buffer Trigger Position	29
Sampled Data Display Controls	31
Saving and Loading Activations	35
Cross Triggering	36
Listing Watchpoints and Signals	38
HAPS Deep Trace Debug	40
Running Deep Trace Debug with DDR3 Memory	40
Viewing Captured Deep Trace Debug Samples	41
Hardware Configuration Verification	42
Debugging on a Different Machine	44
Simultaneous Debugging	45

Debugger-Analyst Integration	46
Waveform Display	51
Generating the Fast Signal Database	53
Logic Analyzer Interface Parameters	54
Logic Analyzer Scan Tab	54
Logic Analyzer Properties Tab	56
Logic Analyzer Submit Tab	56
IICE Assignments Report Tab	57

Chapter 2: Board Bring-up

Board Query	60
Board Bring-up	63
Setting Initial Values	63
ConfPro GUI	64
Board Configuration Tests	66
Utility Commands	70

Chapter 3: Incremental Flow

Requirements	75
Setting up the Original Design	77
Creating the Incremental Instrumentation	77
Redefining the Instrumented Signals	78
Generating the Bit File	78
Incremental Implementation Support with Distributed Instrumentation	79
Debugging the Incremental Version	79
Incremental Flow Limitations	79

Chapter 4: IICE Hardware Description

JTAG Communication Block	81
Breakpoint and Watchpoint Blocks	82
Breakpoints	82
Watchpoints	83
Multiple Activated Breakpoints and Watchpoints	83
Sampling Block	84
Complex Counter	85
Creating a Complex Counter	85
Debugging with the Complex Counter	85
Disabling the Counter	87

State Machine Triggering	88
Simple or Advanced Triggering	88
Advanced Triggering Mode	89
State-Machine Editor	99
State-Machine Examples	103

Chapter 5: Connecting to the Target System

Basic Communication Connection	112
Debugger Communications Settings	112
Debugger Configuration	118
JTAG Communication	124
JTAG Hardware in Instrumented Designs	126
Using the Built-in JTAG Port	126
Using the Synopsys Debug Port	128
Boards Without Direct Built-in JTAG Connections	130
Setting the JTAG Chain	132
JTAG Communication Debugging	134
Basic Communication Test	134
On-chip Identification Register	135
JTAG Chain Tests	135
UMRBus Communications Interface	136

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 breakpoints. 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:

- [Configuring and Invoking the Debugger](#), on page 10
- [Debugger Windows](#), on page 13
- [Commands and Procedures](#), on page 18
- [HAPS Deep Trace Debug](#), on page 40
- [Debugging on a Different Machine](#), on page 44
- [Simultaneous Debugging](#), on page 45
- [Debugger-Analyst Integration](#), on page 46
- [Waveform Display](#), on page 51
- [Logic Analyzer Interface Parameters](#), on page 54

Configuring and Invoking the Debugger

To configure a design for debugging, click the project tab to reopen the project window (reopening the project window shows the instrumentation and communication settings). Configuring and invoking the debugger is described in the following sections:

- [Reviewing the Instrumentation Settings](#), on page 10
- [Changing the Communication Settings](#), on page 10
- [Reviewing the JTAG Chain Settings](#), on page 11
- [Saving the Debugged Design](#), on page 12
- [Invoking the Debugger](#), on page 12

Reviewing the Instrumentation Settings

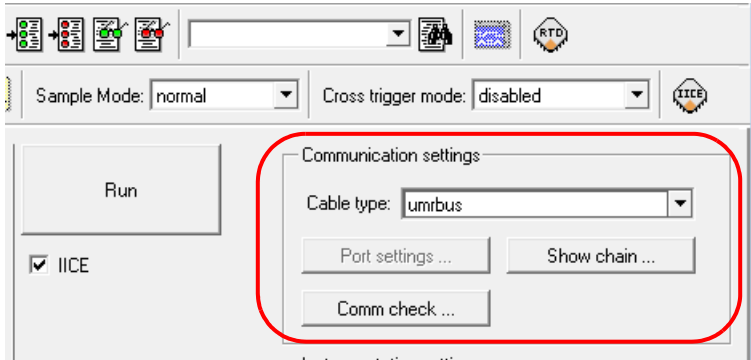
The instrumentation settings are displayed in the Instrumentation settings section of the project window. Because these configuration settings are inherited from the instrumentor and used to construct the IICE, you cannot change these settings in the debugger.

Changing the Communication Settings

The cable type and port specification communication settings can be set or changed from the project window.

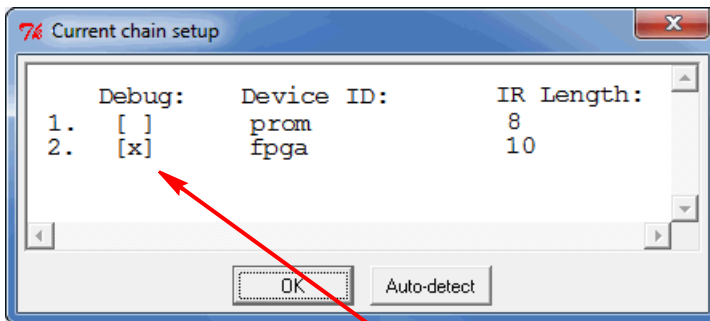
There is a list of possible vendor cable-type settings available from the Cable type drop-down menu. A umrbus setting is also available to setup UMRBus communications between the host and the HAPS[®] board system (see [UMRBus Communications Interface, on page 136](#)). Set Cable type value according to the type of cable you are using to connect to the programmable device.

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



Reviewing the JTAG Chain Settings

The JTAG chain settings are viewed by clicking the Show chain button in the Communication settings section of the project window. Normally, the JTAG chain settings for the devices are automatically extracted from the design. When the chain settings cannot be determined, they must be created and/or edited using the chain command in the console window. The settings shown below are for a 2-device chain that has JTAG identification register lengths of 8 and 10 bits. In addition, the device named “fpga” has been enabled for debugging.



“fpga” device enabled for debugging

Saving the Debugged Design

Saving a your design in the debugger saves the following additional information to the project definition file:

- IICE settings
- Instrumentations and activations



To save your design definition in the debugger, click the Save current activations icon or select File->Save activations from the menu.

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:

- [Synthesis Tool Launch](#), on page 12
- [Operating System Invocation](#), on page 13

Synthesis Tool Launch

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

- From Synplify Pro or Synplify Premier, highlight the Identify implementation and select Run->Launch Identify Debugger from the menu bar or popup menu, or click the Launch Identify Debugger icon in the top 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.
- From Certify, highlight the Identify implementation and select Tools->Launch Identify Debugger from the menu bar or popup menu, or click the Launch Identify Debugger icon in the top menu bar.

The debugger IICE instrumentation window opens with the corresponding project displayed (see [IICE Instrumentation Window, on page 14](#)).

Operating System 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 initial debugger project window opens. To display the instrumentation window, do either of the following:

- Click the Open existing project icon in the menu bar and, in the Open Project File dialog box, navigate to the project directory and open the corresponding project (`prj`) file.
- Select File->Open project from the main menu and, in the Open Project File dialog box, navigate to the project directory and open the corresponding project (`prj`) file.

The debugger instrumentation (IICE) window opens with the corresponding project displayed (see [Project Window, on page 17](#)).

Debugger Windows

The Graphical User Interface for the debugger has three major areas:

- [IICE Instrumentation Window](#), on page 14
- [Console Window](#), on page 16
- [Project Window](#), on page 17

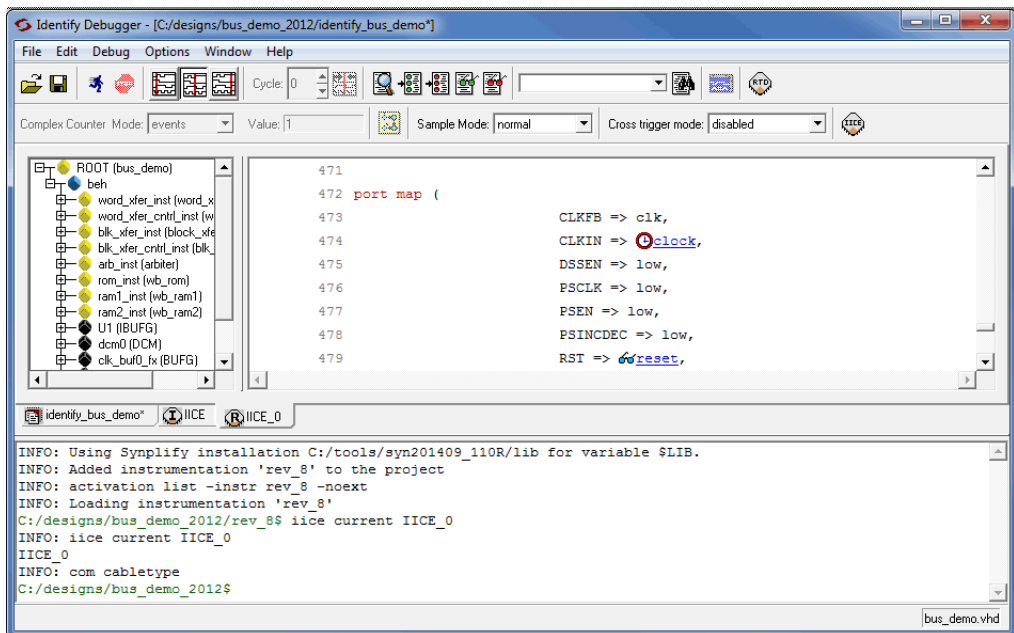
In this section, each of these areas and their uses are described. The following discussions assume that:

- 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 your host for analysis by the debugger

IICE Instrumentation Window

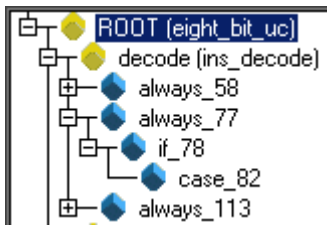
The instrumentation window in the debugger, like the instrumentation window in the instrumentor, includes a hierarchy browser on the left and the source code display on the right.



Hierarchy Browser

The hierarchy browser on the left shows a graphical representation of the design's 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 adjacent source code window.

Source Code Display

The source code display shows 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 small watchpoint (or “P”) 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.

```

44  begin
45      grant1 <= '0';
46      grant2 <= '0';
47
48      case (curr_state) is
49          when st_idle1 =>
50              if ( req1 = '1' ) and ( req2 = '1' ) then
51                  next_state <= st_grant2;
52              elsif ( req1 = '1' ) then
53                  next_state <= st_grant1;
54              elsif ( req2 = '1' ) then
55                  next_state <= st_grant2;
56              else

```

Selecting the watchpoint or “P” 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 is used to specify a watchpoint expression for the signal. See [Setting a Watchpoint Expression, on page 19](#).

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 26](#).

Console Window

The debugger console window displays commands that have been executed, including those executed by menu selections and button clicks. The console window also allows you to enter debugger commands and to view the results of command execution.

```

D:/DESIGNS/SYN_COUNTER$ project open -reapply {D:/Designs/syn_counter/syn_counter.bsp}
INFO: Changed working directory to "D:/Designs/syn_counter"
INFO: Loading design instrumentation version 4.0
INFO: Created Mon Jan 06 10:41:00 2003
INFO: User = garyl
INFO: Platform = windows
INFO: Machine Name = GARY2
INFO: Machine Type = intel
INFO: OS = Windows NT
INFO: OS version = 5.0
INFO: Using instrumentation in "D:/Designs/syn_counter/syn_syn_counter"
D:/DESIGNS/SYN_COUNTER$

```

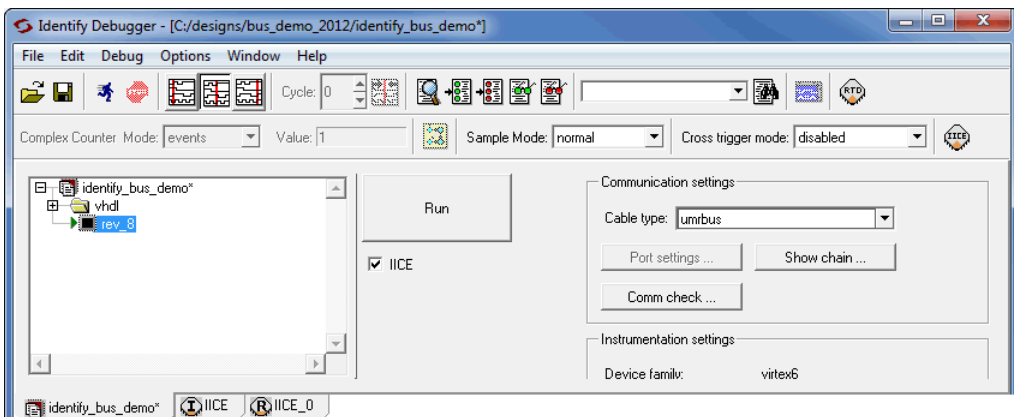

To capture all the text written to the console, use the log console command (see the *Reference Manual*). Alternately, you can click the right mouse button inside the console window and select Save Console Output from the menu. To capture all commands executed in the console window, use the transcript command (see the *Reference Manual*).

To clear the text in the console window, use the clear command or click the right mouse button inside the console window and select clear from the menu.

Project Window

An empty project window is displayed when you explicitly start up the debugger. The window is replaced by the instrumentation window when the synthesis project (prj) file is read into the debugger.

The project window is restored at any time by clicking its tab at the bottom of the window.



The project window displays the symbolic view of the project on the left and a Run button with a list of all of the available IICE units that can be debugged on the right.



Commands and Procedures

This section describes the typical operations performed in the debugger and includes the following topics:

- [Opening and Saving Projects](#), on page 18
- [Executing a Script File](#), on page 19
- [Activating/Deactivating an Instrumentation](#), on page 19
- [Selecting Multiplexed Instrumentation Sets](#), on page 23
- [Activating/Deactivating Folded Instrumentation](#), on page 24
- [Run Command](#), on page 26
- [Sampled Data Compression](#), on page 28
- [Sample Buffer Trigger Position](#), on page 29
- [Sampled Data Display Controls](#), on page 31
- [Saving and Loading Activations](#), on page 35
- [Cross Triggering](#), on page 36
- [Listing Watchpoints and Signals](#), on page 38

Opening and Saving Projects

The debugger commands to open and save projects are available as menu items and icons.

Function	Menu Bar Icon	Menu Command
Open existing project		File->Open project
Save current activations		File->Save activations

When opening a project:

- 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.

Executing a Script File

A script file contains Tcl commands and is a convenient way to capture a command sequence that you would like to repeat. To execute a script file, select the File->Execute Script menu selection and navigate to your script file location or use the source command (see [source](#), on page 83 in the *Reference Manual*).

Activating/Deactivating an Instrumentation

The trigger conditions used to control the sampling buffer comprise breakpoints, watchpoints, and counter settings (see [Chapter 4, IICE Hardware Description](#)). 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.

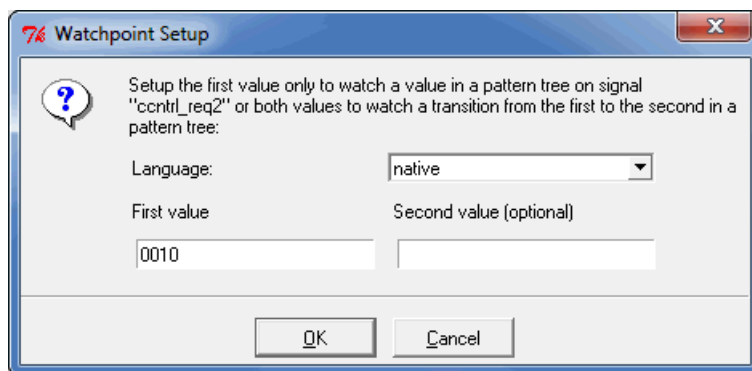


A watchpoint is set on a partial bus signal by clicking-and-holding on the signal or the “P” icon next to the signal, selecting the partial bus group from the list displayed, and then selecting the Set Trigger Expressions menu item to bring up the Watchpoint Setup dialog box.

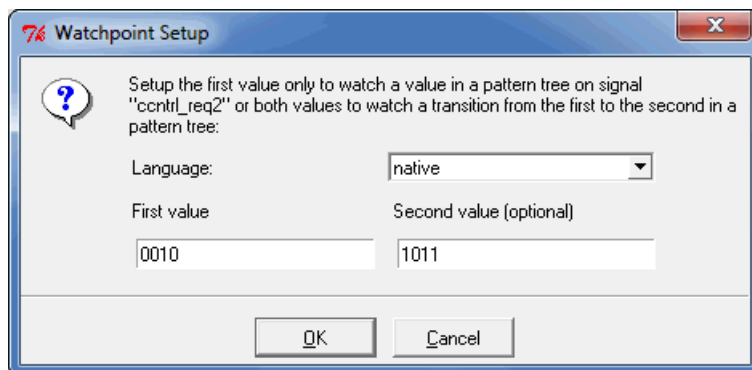
There are two forms of watchpoints: value and transition.

- 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. 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 two constant expressions 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. Similarly, you can include a hexadecimal entry in an equivalent Tcl command by literalizing the quote marks with back slashes as shown in the following example:

```
watch enable -iice IICE -condition 0 /structural/reg_fout x\"aa\"
```

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 popup 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.



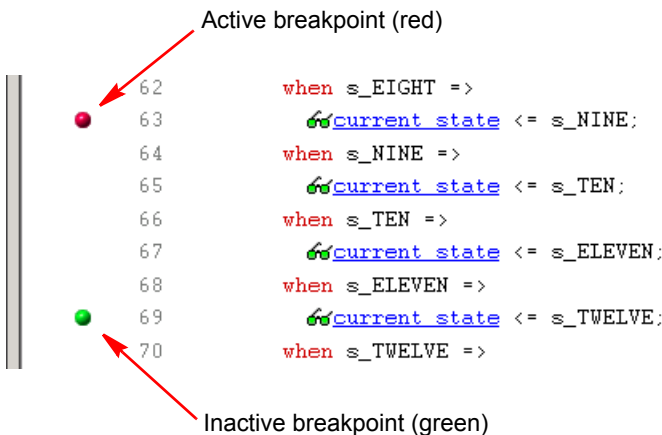
The Watch menu selection will have a check mark to indicate that the watchpoint is activated. Click on the Watch menu selection to toggle the check mark and deactivate the watchpoint.

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 redisplay the watchpoint popup 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.

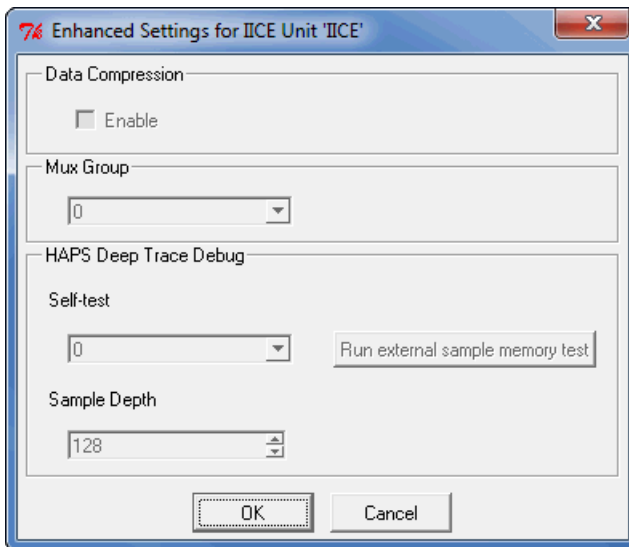
Selecting Multiplexed Instrumentation Sets

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](#), on page 55 in the *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. Click on the IICE icon in the top menu to display the Enhanced Settings for IICE Unit dialog box.



2. Use the drop-down menu in the Mux Group section to select the group number to be active for the debug session.
3. The signals group command can be used to assign groups from the console window (see [signals](#), on page 78 of the *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 multiple instances are created within a generate loop). By definition, there will be more than one instance of every signal and breakpoint 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 multiple instrumented instances of a breakpoint or signal, the debugger allows you to activate/deactivate each of these instrumented instances independently. Independent selection is accomplished by displaying a list of the instrumented instances when the breakpoint or signal is selected for activation/deactivation.

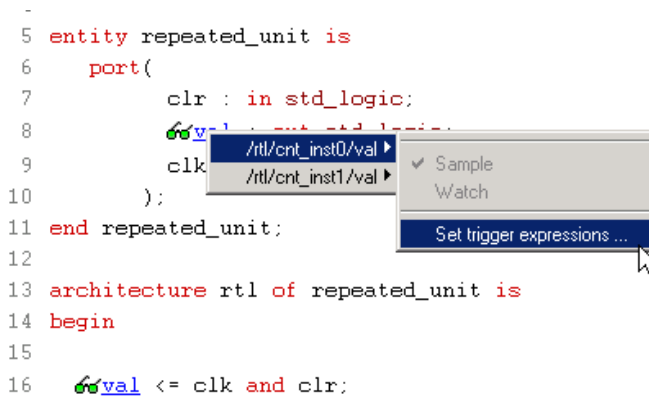
Activating/Deactivating a Folded Watchpoint

The following example consists of a top-level entity called `folded2` and 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 at line 24 (not shown) 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.



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 Signals in a Folded Hierarchy, on page 56](#) in the *Instrumentor User Guide* and [Displaying Data from Folded Signals, on page 32](#).

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:

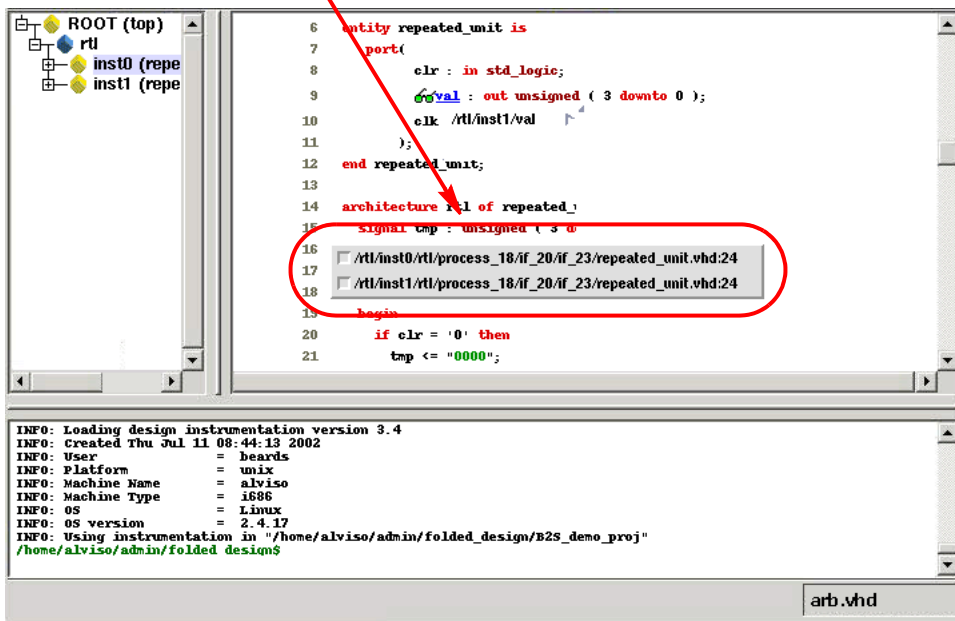
```

/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 list of instrumented instances



The color of the breakpoint icon is determined as follows:

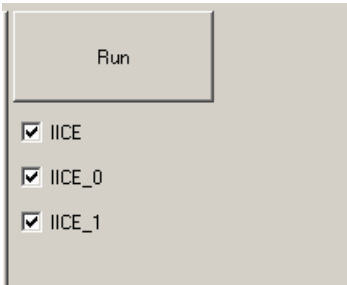
- If no instances of the breakpoint are activated, the breakpoint icon is green.
- If some, but not all, instances of the breakpoint are activated, the breakpoint icon is yellow.
- If all instances are activated, the breakpoint icon is red.

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.



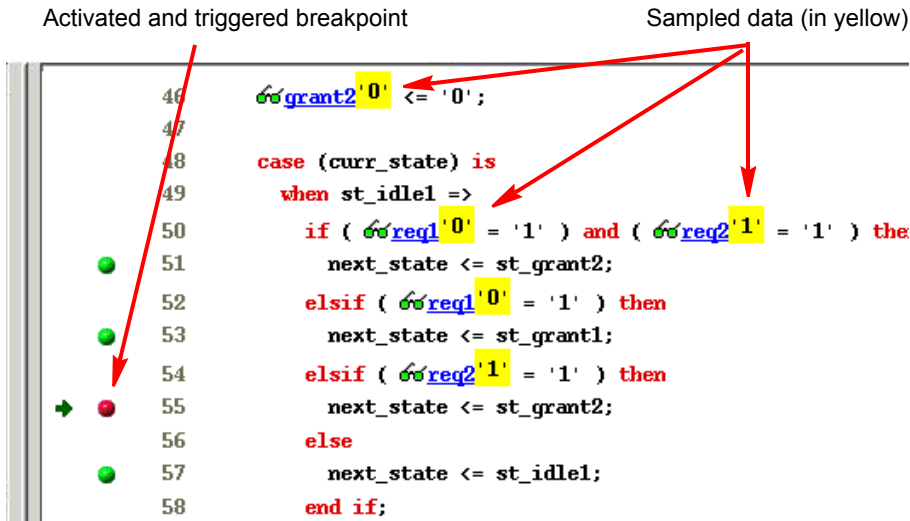
To execute the Run command for the active IICE (or a single IICE), select Debug->Run from the menu or click the Arm selected IICE(s) for triggering icon. If data compression is to be used on the sample data, see [Sampled Data Compression, on page 28](#). To execute the Run command for multiple IICE units, open the project window (click the project window tab), enable the individual IICE units by checking their corresponding boxes, and either click the large Run button or select Debug->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 (see the debugger [waveform](#) command) or written out to a file (see the debugger [write vcd](#) command).

Note: In a multi-IICE environment, you can edit and run other IICEs while an IICE is running. The icons within the individual IICE tabs indicate when an IICE is running (rotating arrow) and when an IICE has new sample data (green check mark).

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.



Stop Command



The Stop command sends control back to the debugger after you have armed the trigger, but before the trigger occurs. The Stop command can be executed by selecting Debug->Stop from the menu or by clicking the Stop debugging hardware icon.

Note: If you are running the IICE from the project window using the Run button and IICE check boxes (multi-IICE mode), you can stop a run by clicking the STOP icon adjacent to the check box.

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 is applied to the sampled data to temporarily remove any data that remains unchanged between 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 reinstrumented 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 Debug->Trigger Position->early 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 Debug->Trigger Position->middle menu selection or click on the Set trigger position to the middle of the sample buffer icon.

Late Position

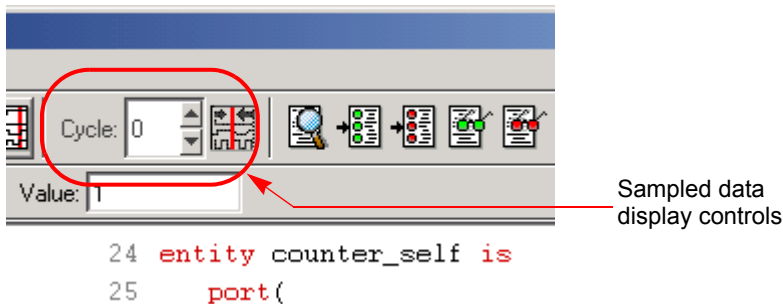


The sample buffer trigger position can be set to “late” so that the majority of the samples occurs before the trigger event. To set the trigger position to “late,” use the Debug->Trigger Position->late menu selection or click on the Set trigger position to late in the sample buffer icon.

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.

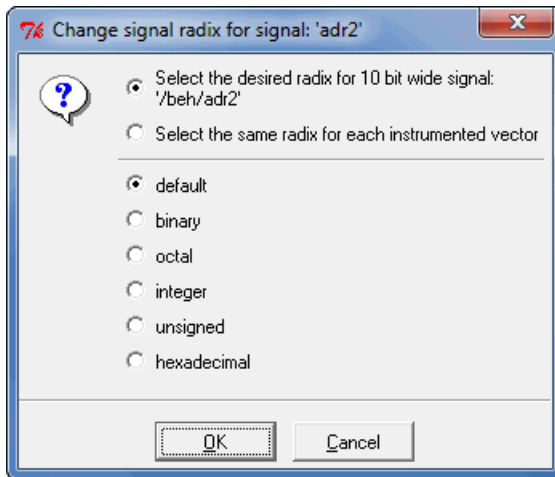


To reset the cycle number to the default position (the zero time position), use the Debug->Cycle->home menu selection or click on the Goto trigger event in sample history icon.

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. Click the corresponding radio button.
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.

Specifying **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.

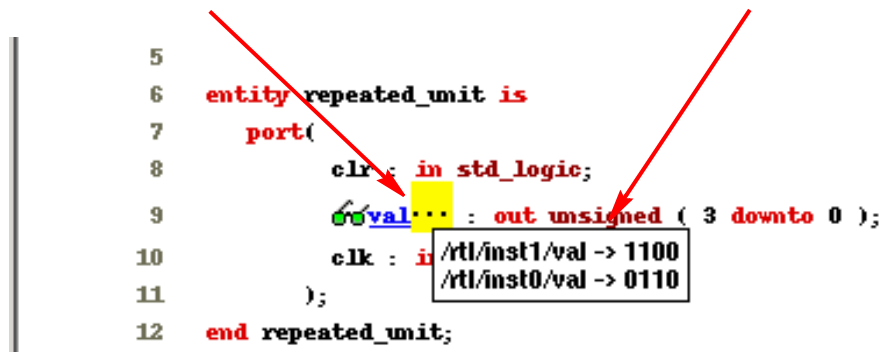
When debugging an instrumented design with multiple instrumented instances of a signal, the debugger allows you to display the data values of each of these instrumented signals.

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**

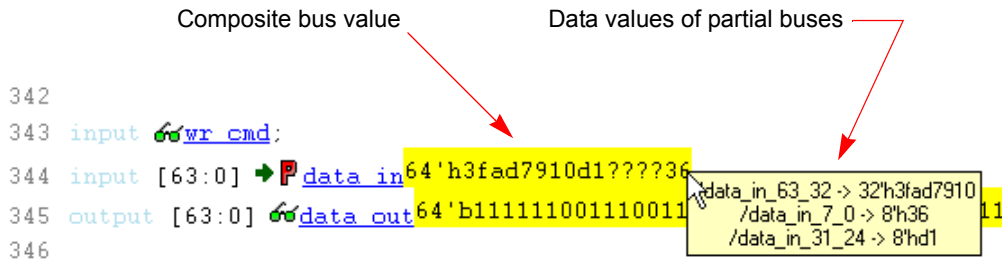


For related information on folded hierarchies, see [Sampling Signals in a Folded Hierarchy](#), on page 56 in the *Instrumentor User Guide* and [Activating/Deactivating Folded Instrumentation](#), on page 24.

Displaying Data for Partial Buses

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

To display the instrumented values for the individual bus segments, position the cursor over the composite bus value. The individual partial bus values are displayed in a tooltip in the specified radix as shown in the following figure.



In the above figure, the question marks (?) in the composite bus value (`64'h3fad7910d1????36`) indicate that the corresponding segment (`data_in [23:8]`) has not been instrumented.

Displaying Data for Partial Instrumentation

In the debugger, the value for a fully instrumented record or structure is shown with a value for each field, in field order. The following figure shows instrumented signal `sig_oport_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`).

```

10 module uddt_P_Struc_tbttop (
11   input 64'clk_in;
12   output type_Unsigned_P_Struc_data P_sig_oport_P_Struc_data
13 );
14
15 logic 64'tb_rst 1'b1;
16 shortint unsigned 64'rst_cnt 65535;
17
18 type_P_Struc_instr 64'sig_oport_P_Struc_instr CMP {{4'b0000} {4'b0010}}
19
20 always @ (posedge 64'clk_in) //rst generation

```

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).

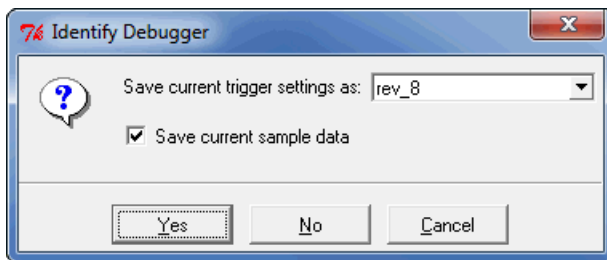
Saving and Loading Activations

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 adb 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 or click the Save current activations icon in the menu bar to bring up the following dialog box.



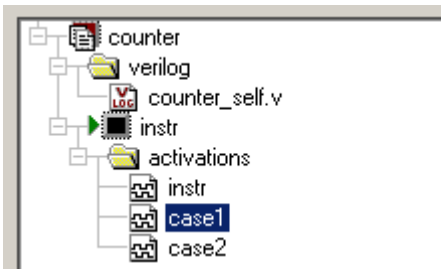
4. Enter (or select) an activation name in the Save current trigger settings as: field. Selecting an existing activation from the drop-down menu overwrites the selected activation.
5. To include the sample data set with the activation, enable the Save current sample data check box.

6. Click Yes to save the activation.

Loading an Activation

To load an existing activation:

1. Open the project view.
2. Expand (if necessary) the hierarchy to display the list of activations as shown in the following figure.



3. 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.adb` file. This file is automatically loaded the next time you open the project.

Note: To save a specific activation, always use Save current activations to explicitly name the file and prevent the data from overwriting the `last_run.adb` file.

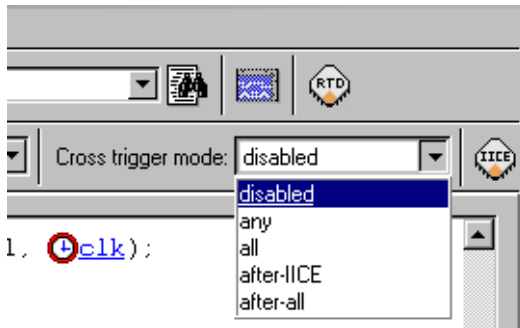
To disable the auto-save feature, uncheck the Auto-save trigger settings and sample results check box on the Debugger Preferences dialog box (select Options->Debugger preferences).

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 Controller tab for the local IICE unit. The cross-trigger mode is selected from the drop-down menu in the debugger as shown below.



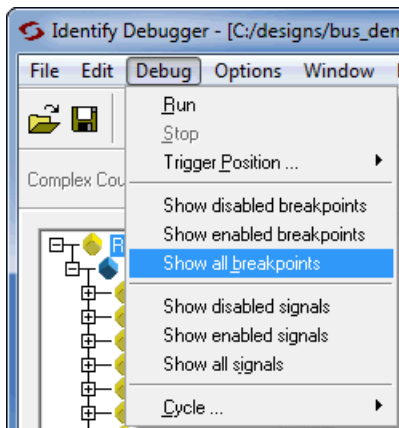
The drop-down menu selections are as follows:

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- <i>iceName</i>	Event trigger from local IICE unit occurs only after the event at selected external IICE unit <i>iceName</i> has occurred (external IICE units are individually listed)
after all	Event trigger from local IICE unit occurs after all events occur at all IICE units

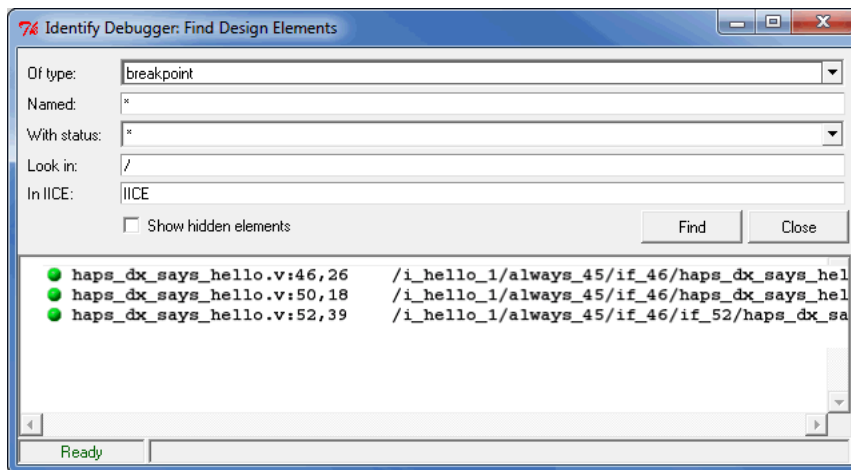
Note: If the drop-down menu does not display, make sure that Allow cross-triggering in IICE is enabled on the IICE Controller tab of the instrumentor and that you have defined more than one IICE unit.

Listing Watchpoints and Signals

To list categories of watchpoints and signals in the debugger, use the popup Debug menu selection and select the category from the list displayed.



The results are displayed in the Find Design Elements dialog box.



The show watchpoint and breakpoint icons in the menu bar display their corresponding values in the Find Design Elements dialog box as follows:

Show Disabled Breakpoints



To display the disabled (inactive) breakpoints, click the Show disabled breakpoints icon.

Show Enabled Breakpoints



To display the enabled (active) breakpoints, click the Show enabled breakpoints icon.

Show Disabled Watchpoints



To display the disabled (inactive) watchpoints, click the Show disabled watchpoints icon.

Show Enabled Watchpoints



To display the enabled (active) watchpoints, click the Show enabled watchpoints icon.

HAPS Deep Trace Debug

The HAPS Deep Trace Debug feature supports any of the following memory configurations:

- DDR3_SODIMM_HT3 (4GB single-rank daughter board)
- DDR3_SODIMM2R_HT3 (8GB dual-rank daughter board)
- HAPS SRAM_HT3 (720MB daughter board)
- HAPS SRAM_1x1_HTII (288MB daughter board for HAPS-60 systems)

Using any of these types of added memory provides a significantly deeper, signal-trace buffer.

With the HAPS deep trace debug mode, the flow remains unchanged. The only difference is in the configuration of the additional memory as the sample buffer using IICE parameters in the instrumentor (see [Chapter 3, HAPS Deep Trace Debug](#) in the *Instrumentor User Guide*).

When you debug the design, the debugger automatically calculates the sample depth and source clock based on the configuration settings supplied in the instrumentor. The configured sample depth can be varied dynamically from the minimum depth to the maximum configured depth.

Running Deep Trace Debug with DDR3 Memory

To maximize performance when using DDR3 memory, refer to the guidelines in the table below to determine the sample frequency based on the number of sample bits being instrumented. The maximum number of instrumented bits that can be sampled with DDR3 is 2042.

Instrumented Bits	Max Sample Frequency	Max Sample Depth (4GB single rank)	Max Sample Depth (8GB dual rank)
1 to 250	140 MHz	134,217,727	268,435,455
251 to 506	70 MHz	67,108,863	134,217,727
507 to 1018	35 MHz	33,554,431	67,108,863
1019 to 2042	17.5 MHz	16,777,215	33,554,431

Based on the number of signals instrumented, the tool automatically calculates the maximum buffer depth. The configured sample depth can be varied dynamically from the minimum depth to the maximum depth.

When the sample depth is set to a large value, the captured samples are first downloaded block-by-block. Once all of the blocks are downloaded, viewing of large samples in the waveform viewer is very time consuming and also the size of the VCD/FSDB dumps becomes extremely large (for a full buffer depth, the time to download all the sample blocks can be between 30 and 40 minutes and a full VCD dump can require several hours).

To reduce these times, use the waveform writer in the debugger to dump a specific range or *slice* of the VCD/FSDB waveform (see [Viewing Captured Deep Trace Debug Samples, on page 41](#)). In the debugger, click on the waveform display icon to bring up the pop-up window where you can specify the cycle range over which to view the waveform. The configured sample depth can be varied in the debugger, but cannot be greater than the depth set in the instrumentor.

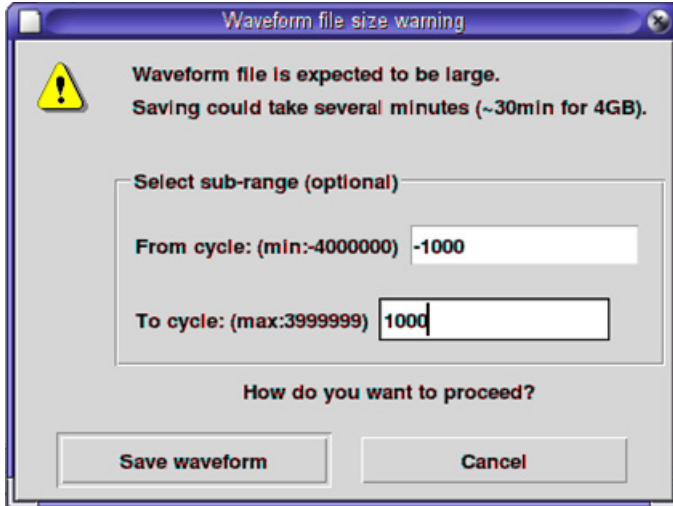
Also, using deep-trace debug on a Windows-based system with minimal resources can be extremely slow, especially when downloading large captured samples or when writing the corresponding VCD/FSDB waveform dumps. Increasing the memory capacity and processor speed of the host can significantly improve performance.

Viewing Captured Deep Trace Debug Samples

A large sample depth translates to large VCD/FSDB dump files. For these cases, the debugger includes the option of viewing or writing out a slice of the FSD or VCD waveform based the number of captured cycles.

To write out a slice of the waveform:

1. Launch the debugger with the exported runtime environment from the operating system (see [Invoking the Debugger, on page 12](#)).
2. In the debugger GUI, open the design definition file (debug.prj).
3. Click the Waveform Display icon. If the sample depth is set to more than 8000000, the tool displays a popup window.



4. In the pop-up window:
 - Specify the cycle range to view on either side of the trigger position. The following example shows a sub-range of -1000 to 1000 specified, although the complete VCD/FSDB extends from -4000000 to 3999999 on either side of the trigger position.
 - Click Save waveform at the bottom of the dialog box to save and view the specified sub-range. If you click the button without specifying a sub-range, the tool saves the entire waveform to IICE.vcd or IICE.fsdb. This could take some time, as it downloads the full buffer depth and all the sample blocks. A full VCD dump can take hours.
5. Alternatively, write out vcd or fsdb using the -range argument with the appropriate TCL command:

```
write vcd -range {fromCycle toCycle} filename.vcd  
write fsdb -range {fromCycle toCycle} filename.vcd
```

Hardware Configuration Verification

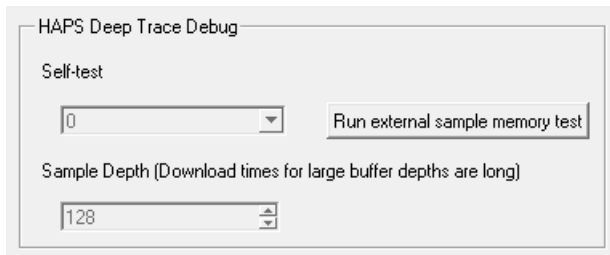
A self-test is available for verifying the deep trace debug hardware configuration. The self-test writes data patterns to the external memory and reads back the data pattern written to detect configuration errors, connectivity problems, and SRAM frequency mismatches.

The self test is normally executed:

- following the initial setup to verify the hardware configuration against the instrumentation
- during routine operations whenever a hardware problem is suspect
- whenever the physical configuration is modified (changing any of the IICE Sampler dialog box configuration settings such as relocating the SRAM daughter card to another connector)

To run the self-test from the debugger GUI:

1. Open the project view.
2. Click the IICE icon.
3. Select one of the two patterns (pattern 0 or pattern 1) from the Self-test drop-down menu.
4. Click the Run SRAM tests button.



Selecting 0 uses one test pattern, and selecting 1 uses another pattern. To ensure adequate testing, repeat the command using alternate pattern settings.

The self-test can also be run from the command line using the following syntax:

iice sampler -runselftest 1|0

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.

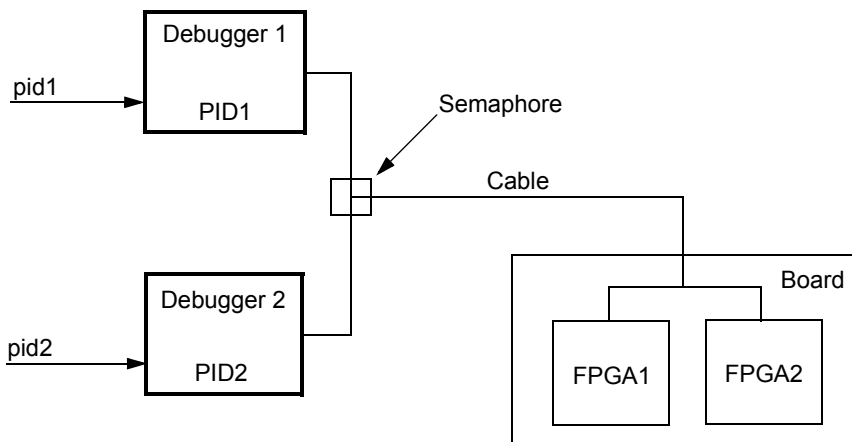
```
searchpath {d:/temp;c:/Documents and Settings/me/my_design/}
```

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, on page 67](#) in the *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.




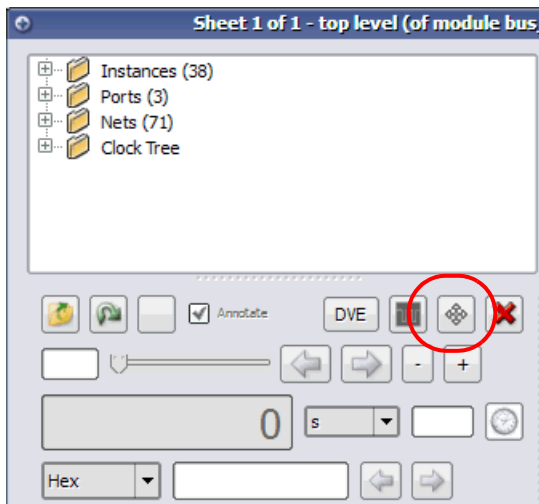
Debugger-Analyst Integration


The display of instrumented signals captured in a VCD file by the debugger is available within the HDL Analyst in Synplify Premier.

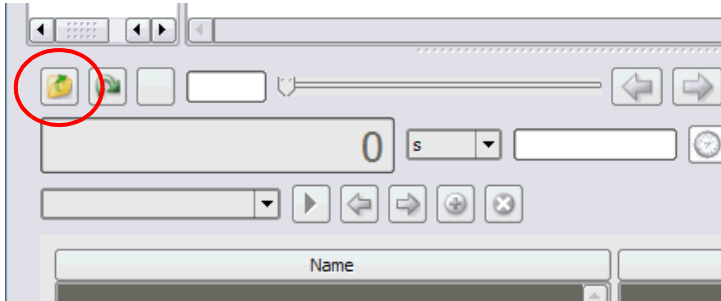
The following steps outline an abbreviated procedure for using an debugger-generated VCD file with the HDL Analyst. For a complete description of this feature, see VCD-Analyst Integration in the *Synopsys FPGA Synthesis User Guide*.

After generating a VCD file in the debugger and opening the HDL Analyst RTL view in Synplify Premier:

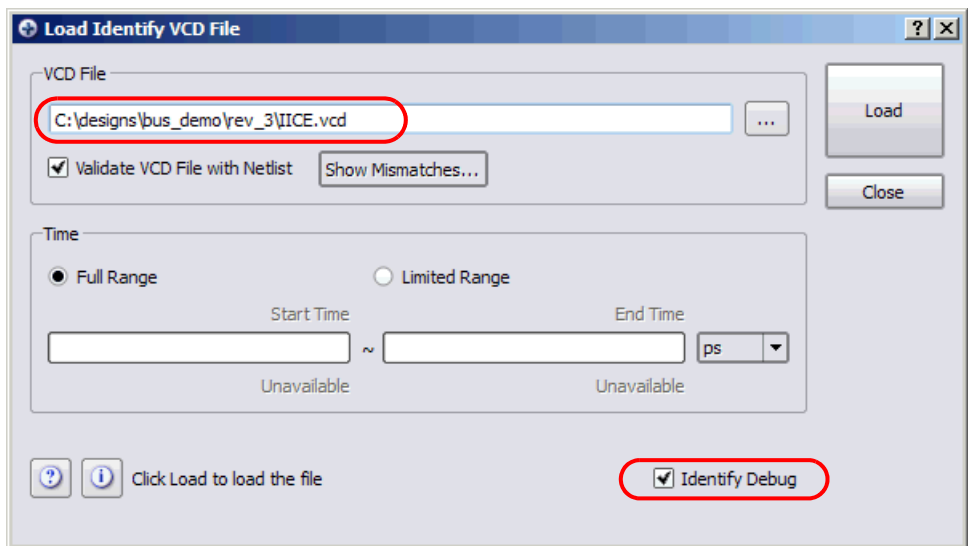
1. Click the VCD Panel icon () or select VCD->VCD Panel from the HDL-Analyst menu to display the VCD control panel.
2. If necessary, click the Move this panel to an alternate location button to relocate the VCD control panel under the RTL view.



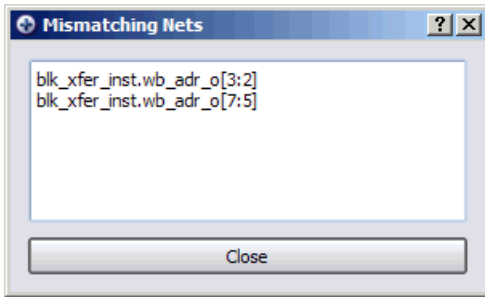
3. Click the Open a VCD File icon () or select VCD->Load VCD File from the HDL-Analyst menu to open the Load Identify VCD File dialog box.



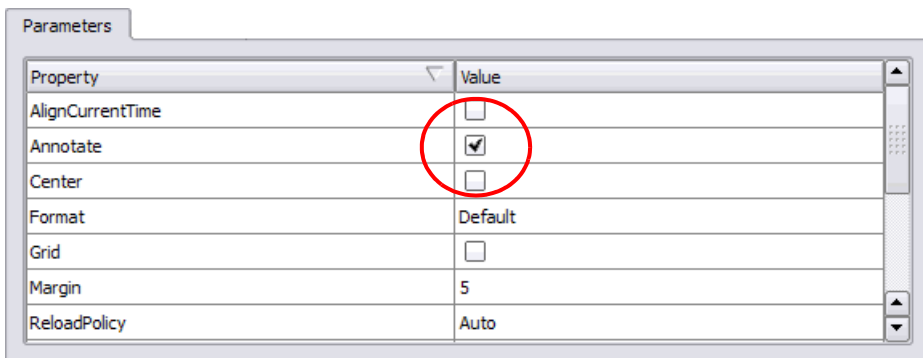
4. In the dialog box, enter the path to the vcd file generated by the debugger (use the browse ... button) and make sure that the Identify Debug box is checked. The Validate VCD File with Netlist check box, when enabled, checks for mismatches between the design netlist and the VCD file loaded.



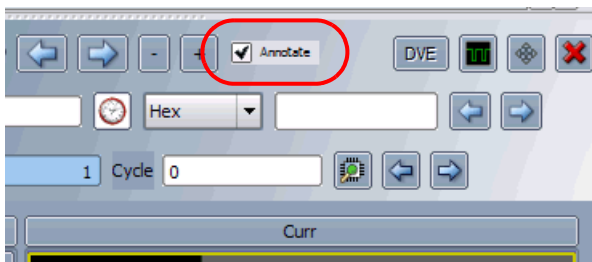
5. Click the Load button to load the VCD file and display the instrumented signals from the VCD file in the waveform viewer.
6. If Validate VCD File with Netlist is checked, click the Show Mismatches button to display any mismatched nets. Mismatches are reported in the Mismatching Nets dialog box.



7. Close the Load Identify VCD File dialog box.
8. To view values for the signals, select the desired signals in the waveform viewer and select HDL-Analyst->VCD->VCD Properties. On the Parameters tab, enable the Annotate check box.



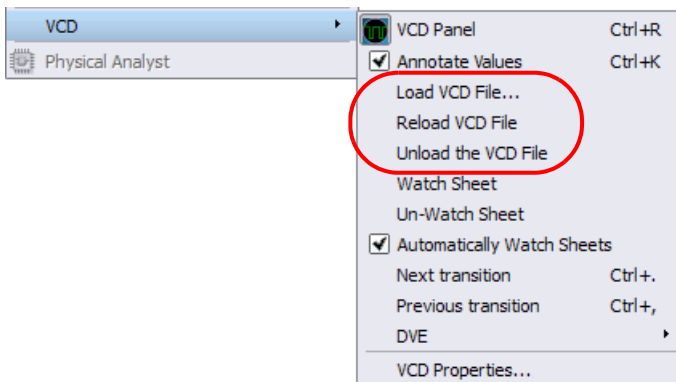
9. To annotate values on the waveform viewer to their respective HDL Analyst sheet, check the Annotate box on the control panel.





Select a particular signal on the control panel to highlight its corresponding signal in the RTL view. Signals are annotated with their previous, current, and next values.

Loading and Unloading VCD Files

You can load, re-load, or unload debugger VCD files from the HDL-Analyst->VCD menu.



- To load an debugger VCD file, select Load VCD File (or click the Open a VCD File icon () on the control panel).
- To re-load an debugger VCD file, select Reload VCD File (or click the Re-open the previously loaded VCD file icon () on the control panel).

When the debugger generates a revised VCD file, changes to the VCD file must be handled after it is loaded. The reload policy implemented provides the following options:

- Auto – automatically reload the VCD file (the default)
- Ask – prompt if the VCD file is to be reloaded
- Never – never reload the VCD file

The reload policy is set on the Parameters tab of the VCD Properties dialog box. When a VCD file is reloaded, the tool preserves information as much as possible such as the current time and watched signals.

- To unload a VCD file, select Unload the VCD File. This option frees up memory used by the debug data without having to close and re-open the HDL Analyst view.

VCD Control Panel Functions

The following additional functions are available from the VCD control panel:

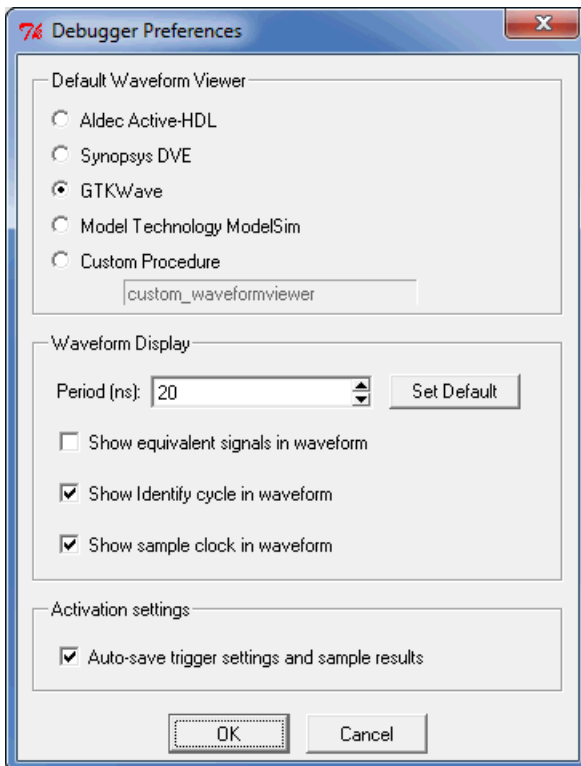
- Observing nets on a particular HDL Analyst sheet
- Changing the format of signals displayed in the viewer

These functions are described in detail in the *Synopsys FPGA Synthesis User Guide*.

Waveform Display

The waveform display control displays the sampled data in a waveform style. By default, this feature uses the Synopsys DVE waveform viewer. Provision for using other popular waveform viewers that support VCD 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 SolvNet website.

Viewer selection and setup are controlled by the Waveform Viewer Preferences dialog box. Selecting Options->Debugger preferences from the menu bar brings up the dialog box shown below.

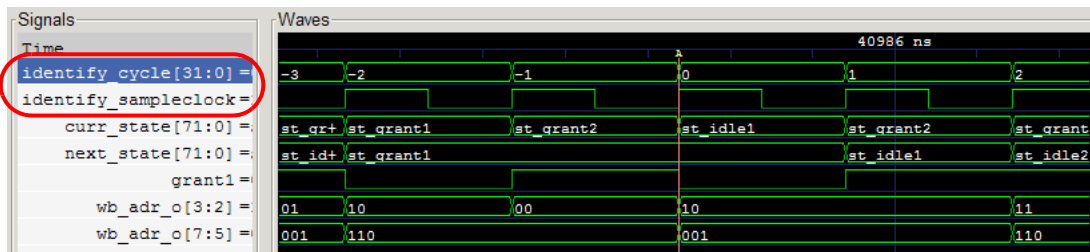


The Synopsys DVE waveform viewer is only available on Linux platforms. To use the included GTKWave viewer, click the GTKWave radio button in the Default Waveform Viewer section.

The Period field sets the period for the waveform display and is independent of the design speed.



After running the debugger, the selected waveform viewer is displayed by selecting Window->Waveform from the menu or by clicking the Open Waveform Display icon in the menu bar. 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 Debugger Preferences dialog box (select Options->Debugger preferences).
2. Select the desired waveform viewer by clicking the adjacent radio button and then click OK.
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 Window->Waveform or click on the Open Waveform Display 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 58](#) in the *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 into the Verdi platform.

Logic Analyzer Interface Parameters



The logic analyzer interface parameters for the real-time debug feature in the debugger are defined on the tabs of the RTD type IICE information dialog box. To display this dialog box, click on the RTD (RTD type IICE Information/Settings) icon in the top menu. The remainder of this section describes the individual logic analyzer tabs:

- [Logic Analyzer Scan Tab](#), on page 54
- [Logic Analyzer Properties Tab](#), on page 56
- [Logic Analyzer Submit Tab](#), on page 56
- [IICE Assignments Report Tab](#), on page 57

Logic Analyzer Scan Tab

The Logic Analyzer Scan tab defines:

- the logic analyzer type
- the TLA script program
- user name
- host name/IP address
- if pods are automatically assigned to Mictor connectors

Type of Logic Analyzer

Selects the type of logic analyzer from a drop-down menu. Current supported types are Agilent 16700 and 16900 series and Tektronix TLA series analyzers. The logic analyzer must be accessible on the local network.

TLA Script Program

Specifies the full path to the tlascript script file on the Tektronix logic analyzer. The default path is C:\Program Files\TLA 700\System\tlascript. If this location does not match the location expected by the Tektronix logic analyzer, change the location setting. The logic analyzer requires an rsh daemon to access the script file. To download and install the rsh daemon on the logic analyzer, see the web-site at <http://rshd.sourceforge.net>.

User Name

Identifies the user name on the analyzer (Tektronix only).

Host Name/IP Address

Specifies the host name or IP address for the debugger host.

Assign Pods automatically to Mictor connectors

When checked, automatically assigns pods to the Mictor connectors.

Scan Logic Analyzer

Clicking the Scan Logic Analyzer button scans the specified IP address and, if scanned successfully:

- opens a network connection with the given parameters
- retrieves the modules and pods information
- displays Logic Analyzer Properties and Logic Analyzer Submit tabs

Logic Analyzer Properties Tab

The Logic Analyzer Properties tab allows Mictor pin groups to be manually assigned to modules and pods using corresponding drop-down menus. Clicking the Assign Pods button updates the assignments.

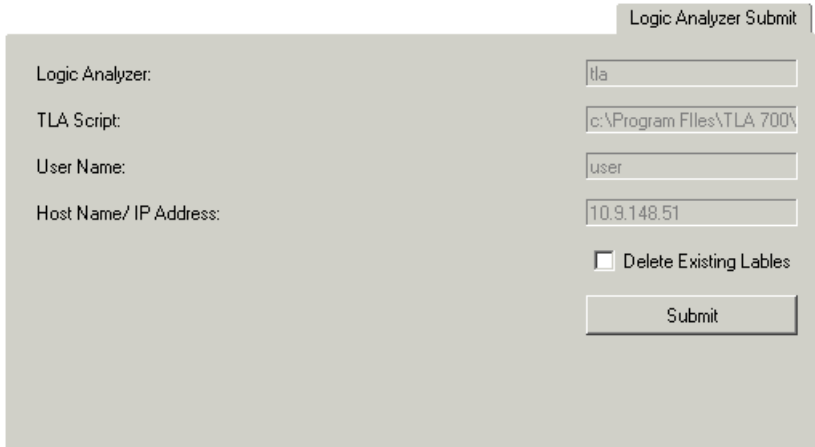
The screenshot shows the 'Logic Analyzer Properties' dialog box. It contains a table with three columns: 'MictorConnectorPinGroup', 'Module', and 'Pod'. The 'Pod' column has a dropdown menu open, displaying a list of pod names. A mouse cursor is pointing at the first item in the list, 'A0A1CK1'. Below the table is an 'Assign Pods' button.

MictorConnectorPinGroup	Module	Pod
3.M1.e	1	A0A1CK1
3.M1.o		A2A3CK0
3.M2.e		C0C1Q1
3.M2.o		C2C3CK3
3.M3.e		D0D1CK2
3.M3.o		D2D3Q0
3.M4.e		E0E1Q2
3.M4.o		E2E3Q3

Assign Pods

Logic Analyzer Submit Tab

The Logic Analyzer Submit tab submits signal/breakpoint names to the logic analyzer.



The image shows a dialog box titled "Logic Analyzer Submit". It contains four text input fields: "Logic Analyzer:" with the value "tla", "TLA Script:" with the value "c:\Program Files\TLA 700\...", "User Name:" with the value "user", and "Host Name/ IP Address:" with the value "10.9.148.51". Below these fields is a checkbox labeled "Delete Existing Lables" which is currently unchecked. At the bottom right of the dialog is a "Submit" button.

IICE Assignments Report Tab

When using the real-time debugging feature in the instrumentor (see [Real-time Debugging, on page 63](#) in the *Instrumentor User Guide*), the signal/breakpoint interface assignments to the Mictor connector are reported in the debugger on the IICE Assignments Report tab. Clicking the tab before assigning logic analyzer pods to the Mictor pin groups reports only the signal/breakpoint assignments. Clicking the tab after assigning logic analyzer pods to the Mictor pin groups includes the pods assignments in the report.

By default, the report is displayed on the screen (standard out). The report can be redirected to a file using the `iice assignmentsreport Tcl` command (see [iice, on page 51](#) in the *Reference Manual*).

IICE Assignments Report		
Pod Assignments		
Mictor Connector	LogicAnalyzer Module	LogicAnalyzer Pod
Signal/breakpoint Assignments		
Mictor Pin	Signal/Breakpoint	
3.M1.D0e	/adr_o[9]	
3.M1.D1e	/adr_o[8]	
3.M1.D2e	/adr_o[7]	
3.M1.D3e	/adr_o[6]	

CHAPTER 2

Board Bring-up

When using a HAPS system with the Certify multi-FPGA prototyping tool, special software is available to query the HAPS system and then to use this information to create a board file that describes that system configuration. Once configured, a set of board bring-up utilities are available to validate the configuration.

To assist in HAPS-60 and HAPS-70 series system board development, Certify software uses the `confpro` shell in the debugger to:

- query the actual board configuration to create an initial board (`vb`) file for the system including applicable daughter cards and cable interconnect, clock and reset configuration, and voltage region definitions
- perform verification of the newly-defined board including HAPS hardware checks, FPGA bus access verification, clock and reset checks, HSTDM performance, and a self-test

Before you can use the HAPS board query and bring-up utilities, the following software must be installed:

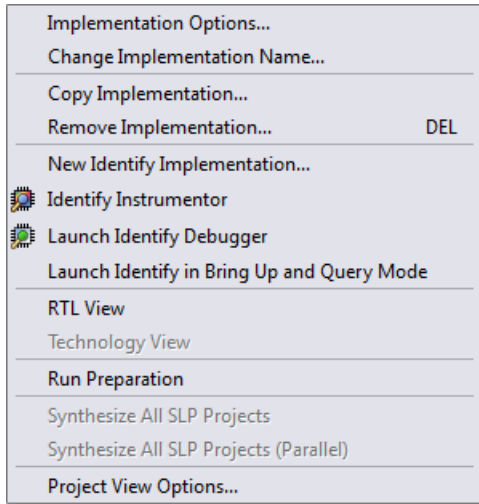
- Current version of the instrumentor/debugger tool set
- ConfPro GUI

For current versions and installation information, see the release notes.

Running the board bring-up utility launches a special debugger window where you can either use the debugger graphical interface or enter a series of `haps` utility and `board-test` commands at the command prompt to set board parameters and to run individual board tests to verify the board configuration.

Board Query

The board query utility is initiated from the Certify user interface by selecting Launch Identify in Bring Up and Query Mode from the Tools popup menu. Running this utility creates a Tcl script for generating a first approximation of the board file. This file replaces the original *skeleton* board file defined for the project.



The script uses a set of `board_system` Tcl commands to describe the system configuration in terms of boards, interconnect, clocks, resets, and voltage regions. For detailed information and command syntax, see *HAPS Board Systems* in Chapter 26, Board Description Files, in the *Certify User Guide* and the individual `board_system` command descriptions in the *Certify Command Reference*.

The following shows a set of auto-generated, board-query Tcl commands.

```
# ABFG TCL file from Hardware Configuration
board_system_create -haps -name conf_board
board_system_create -add HAPS64 -name FB1
board_system_create -add HAPS64 -name FB2

# Connections on Board FB1
board_system_create -interconnect -manual CON_CABLE40e
    -name CON_CABLE40e-13501 -connector {FB1.A6 FB1.B6}
board_system_create -interconnect -manual SRAM_1x1 HTII
    -name SRAM_1x1_HTII-PD-00792 -connector {FB1.A2}
```

```
# Connections on Board FB2
board_system_create -interconnect -manual SRAM_1x1_HTII
    -name SRAM_1x1_HTII-PD-00787 -connector {FB2.A3}
board_system_create -interconnect -manual SRAM_1x1_HTII
    -name SRAM_1x1_HTII-PD-00788 -connector {FB2.A6}

# Clocks and Resets
board_system_configure -voltage {FB1.V1a} 2.5
board_system_configure -clock {FB1.GCLK1} PLL1
board_system_configure -reset {FB1.RESET_A} 1

#Save Board
board_system_save -board conf_board.vb
```

The auto-generated Tcl file can be edited to explicitly describe additional cable interconnect. In the excerpt below, commands are manually inserted to describe the added interconnect.

```
# ABFG TCL file from Hardware Configuration
board_system_create -haps -name conf_board
board_system_create -add HAPS64 -name FB1
board_system_create -add HAPS64 -name FB2

# Connections on Board FB1
# Manual additions here to describe intra- and inter-board
  cable interconnect
board_system_create -interconnect -auto -width 24
    -devices {FB1.uA FB2.uB}
board_system_create -interconnect -manual CON_CABLE40e
    -name CON_CABLE40e-13501 -connector {FB1.A6 FB1.B6}
board_system_create -interconnect -manual SRAM_1x1_HTII
    -name SRAM_1x1_HTII-PD-00792 -connector {FB1.A2}

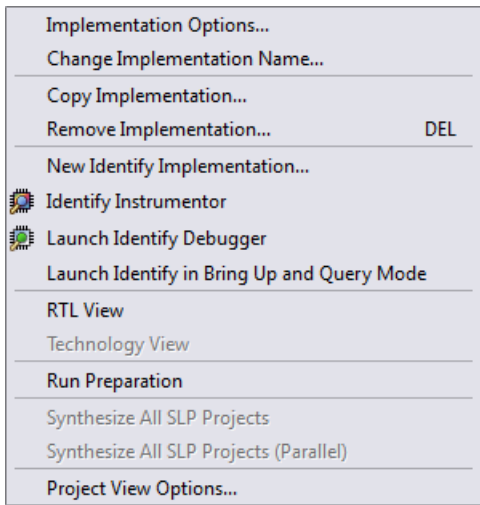
# Connections on Board FB2
# Manual additions here to describe intra- and inter-board cable
  interconnect
board_system_create -interconnect -auto -width 50
    -devices {FB1.uB FB2.uA}
board_system_create -interconnect -manual SRAM_1x1_HTII
    -name SRAM_1x1_HTII-PD-00787 -connector {FB2.A3}
board_system_create -interconnect -manual SRAM_1x1_HTII
    -name SRAM_1x1_HTII-PD-00788 -connector {FB2.A6}

# Clocks and Resets
board_system_configure -voltage {FB1.V1a} 2.5
board_system_configure -clock {FB1.GCLK1} PLL1
board_system_configure -reset {FB1.RESET_A} 1
```

```
#Save Board  
board_system_save -board new_conf_board.vb
```

Board Bring-up

The board bring-up utility is initiated from the Certify user interface by selecting Launch Identify in Bring-up and Query Mode from the Tools popup menu. To use this utility, an Identify implementation must be defined for the project and a HAPS-60 or HAPS-70 board (vb) file must be included in the defined project (an HDL design is not required, but an initial board file must be present).

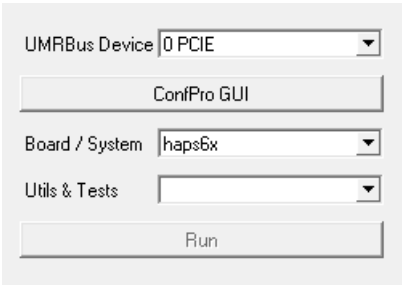


Running the board bring-up utility launches a special debugger window where you can either use the debugger graphical interface or enter a series of haps utility and board-test commands at the command prompt to set board parameters and to run individual board tests to verify the board configuration.

The debugger haps command syntax is described in the *Debug Environment Reference Manual*. The individual board tests are described in [Board Configuration Tests](#), on page 66.

Setting Initial Values

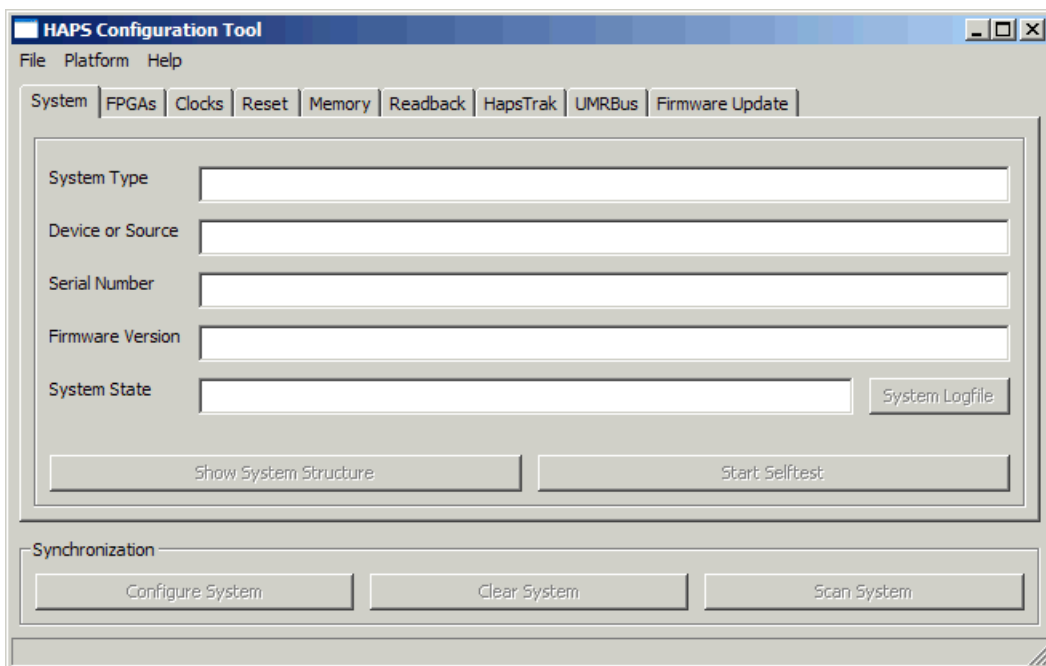
The following table describes the selections and fields in the debugger bring-up utility menu.



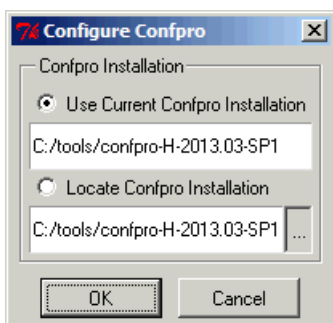
Function	Description
UMRBus Device	Selects the type and location of the UMRBus device. Eight PCIe and eight UMRBus devices can be selected from the drop-down menu.
ConfPro GUI	Selects the ConfPro GUI. If the location of the ConfPro GUI is not specified, you are prompted for the install location. For more information on the ConfPro GUI, see ConfPro GUI, on page 64 .
Board/System	Selects the type of board/system connected to the host. The allowed selections are available from the drop-down menu.
Utils & Tests	Selects the query/test to be run based on the type of board system selected. For more information on the available utilities and tests, see Board Configuration Tests, on page 66
Run	Runs the selected query/test to be executed.

ConfPro GUI

The ConfPro GUI is launched from the board bring-up utility in the debugger GUI. When you click the ConfPro button, the HAPS Configuration Tool menu shown below is displayed.



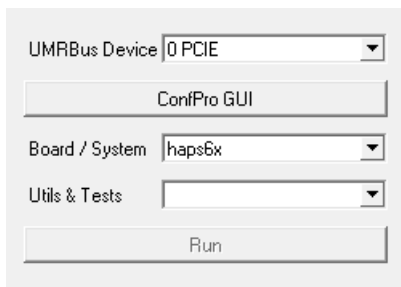
The HAPS Configuration Tool dialog box includes both a File and a Platform top-level menu as well as the top-level Help menu. Complete ConfPro GUI documentation is available by selecting Help->Contents from the top-level menu. The location of the ConfPro installation is specified by selecting Options->Configure Confpro from the debugger menu.



Board Configuration Tests

The following board tests are available from the graphical user interface and by individual Tcl commands. Before you can run any of the tests, make sure:

- confpro has been installed and configured
- the UMRBus PCI/USB device is selected
- a Board/System is selected

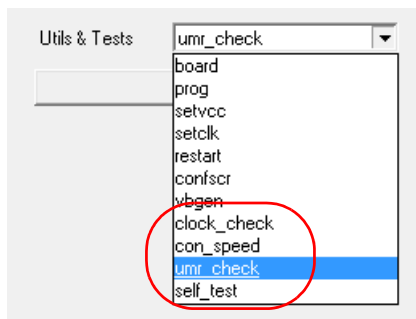


The corresponding Tcl command for device and board selection is

haps settings {PORT_NAME portName DEV_ID device BUS_NUM bus}

which specifies the HAPS port (PORT_NAME), device (DEV_ID), and bus (BUS_NUM) settings. For detailed information on the haps Tcl commands, see [haps, on page 39](#) in the *Reference Manual*.

Board tests are selected from the Utils & Tests drop-down menu.



The same board configuration tests can be run directly from the debugger command prompt without requiring the installation of the Certify prototyping tool.

umr_check Test

The umr_check test verifies the basic functionality of the UMRBus. The test passes if the basic UMRBus operation succeeds. The umr_check test:

- Downloads bin files through the UMRBus
- Reads status registers
- Writes control registers

The umr_check test is selected from the Utils & Tests drop-down menu and includes both FPGA ID and Frequency selections. The FPGA ID drop-down menu selects the FPGA to be tested; the Frequency field sets the GCLK1 clock frequency used to test the design. The test is executed by clicking the Run button.

The screenshot shows the ConfPro GUI interface for the umr_check test. On the left, there are three dropdown menus: 'UMRBus Device' set to '0 PCIE', 'Board / System' set to 'haps6x', and 'Utils & Tests' set to 'umr_check'. Below these are two buttons: 'ConfPro GUI' and 'Run'. On the right, there is a text box stating: 'The umr_check test verifies the basic functionality of the UMRBus. The test will pass if the UMRBus is functioning correctly.' Below this text are two input fields: 'FPGA ID' set to '1' and 'Frequency (Khz)' set to '140000'.

The Tcl syntax for running the umr_check test is:

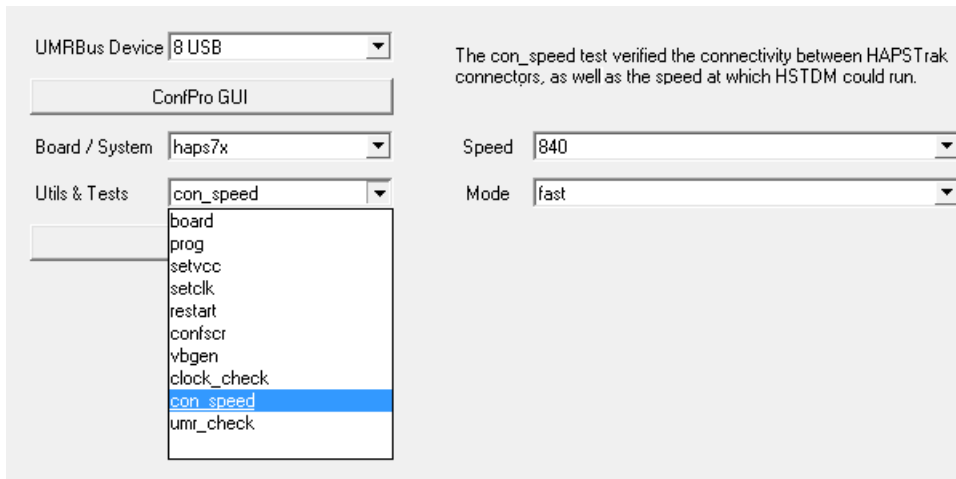
```
haps run umr_check fpgaID frequency
```

In the above syntax:

- *fpgaID* – indicates which FPGA device is to be tested. The default is 1, which is the first FPGA device on the first board.
- *frequency* – sets the frequency for GCLK1 which is used in the test of the design. The default frequency is 140Mhz.

con_speed Test

The con_speed test verifies the connectivity between HapsTrak[®] connectors as well as the speed at which HSTDM can run. The con_speed test is selected from the Utils & Tests drop-down menu and includes both a Speed and Mode selection. The test is executed by clicking the Run button.



The Speed selection sets the HSTDM verification rate (in Mbps). The ALL selection scans a range of speeds to determine the highest rate possible. Two Mode selections are available – fast and sweep. The fast mode (the default) verifies each channel at the specified speed; the sweep mode tests (sweeps) all channels at each speed and can require several hours to complete.

The con_speed test requires a connectivity.tcl file to describe the Certify board configuration. A connGen.tcl script is used to generate the connectivity.tcl file for the con_speed test based on the Certify board file. To run the connGen.tcl script from the Certify GUI:

1. Open the project (select File->Open Project)
2. If the project is not current, click the Run Preparation button
3. Make sure that the HSTDM check box on the Partitioning tab is checked
4. Fully partition the design
5. Source the connGen.tcl file:

```
source identify_install/lib/bringup_utils/swlib/connGen.tcl
```

To run the connGen.tcl script from the command line:

1. Open the project in Certify (project -load *projectPath*)
2. Run preparation (project -run compile)

3. Load the board file with HSTDm (`load_board/proto/board.srs -hstdm`)
4. Load the netlist (`load_netlist/proto/designNetlist.srs`)
5. Source the connGen.tcl file:

```
source identify_install/lib/bringup_utils/swlib/connGen.tcl
```

Sourcing the connGen.tcl file generates the connectivity.tcl file in the design directory. You must copy this file to your current directory before you can run the con_speed test.

To run the con_speed test, either click the Run button in the debugger GUI or use the following Tcl command syntax:

```
haps run con_speed speed {fast|sweep}
```

The con_speed test produces a log file (hapstest.log) in the current directory that reports:

- FPGA connectivity and banks
- Voltage regions
- FPGA configuration including clock and reset
- Transfer speed and results
- Training results for the banks and minimum eye width
- Hardware/firmware version and test results including any failed channels

HapsTrak 3 Connector Considerations

The HapsTrak 3 connectors at board locations 16, 17, 19, and 20 on a HAPS-70 series system are special connectors in that each of these connectors has only a single clock pair (normal connectors have four clock pairs). When testing the connectivity between any of these special connectors and a standard connector, the special connector is listed last in a connectPorts statement as shown below:

```
connectPorts mb1.B13 mb1.A16
```

When you use the connGen.tcl script in the Certify project directory to generate the connectivity.tcl file, the connGen.tcl script automatically orders the entries in the connectPorts statements.

clock_check Test

The `clock_check` test reports the clock frequency of each GCLK output to allow all of the GCLK frequencies to be verified, and also reports board status. To run the `clock_check` test, either click the Run button in the debugger GUI or use the following Tcl command syntax:

```
haps run clock_check
```

self_test Test

The `self_test` test replaces the traditional self test with an STB2 test card. The self test:

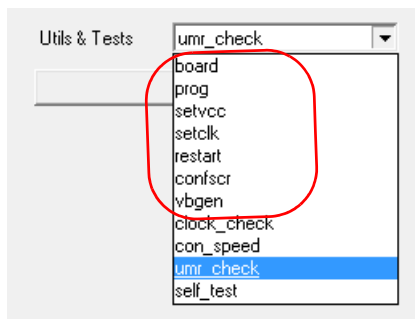
- Executes `haps run self_test` without an STB2 card inserted. The test should pass; failure indicates a short circuit at the connector.
- Executes `haps run self_test` with an STB2 card inserted. The test should pass; failure indicates an open circuit at the connector.

To run the `self_test` test, either click the Run button in the debugger GUI or use the following Tcl command syntax:

```
haps run self_test
```

Utility Commands

Utility commands, like the board test commands, are selected from the Utils & Tests drop-down menu.



This menu is board/system dependent and remains disabled until a board/system selection is made. Selecting a command from the drop-down menu displays a description of the selected command, and clicking the Run

button executes the command. The same utility commands can be run directly from the debugger command prompt without requiring the installation of the Certify prototyping tool.

board

Displays the board status to the console window. Status includes clock and voltage settings, reset condition, daughter card connections, firmware version, and board serial number. The equivalent Tcl command syntax is:

haps board

prog

Programs the FPGA specified in the FPGA ID field with the contents of the selected bin file. Click the Open button to browse to the bin file location. The FPGA ID selection ranges from 1 to 16.

The screenshot shows the 'ConfPro GUI' window. On the left, there are three dropdown menus: 'UMRBus Device' set to '0 PCIE', 'Board / System' set to 'haps6x', and 'Utils & Tests' set to 'prog'. Below these is a 'Run' button. On the right, there is a text label 'Program FPGA with a .bin file. 'devID' start from 1.' Below this is a 'BIN File' input field with an 'Open' button next to it. Below the 'BIN File' field is an 'FPGA ID' dropdown menu set to '1'.

The equivalent Tcl command syntax is:

haps prog binFile devID

setvcc

Sets the I/O voltage for the board regions. The voltage value and region are selected from the corresponding drop-down menus and differ with the board/system selected. Multiple regions can be selected using the Ctrl key.

UMRBus Device: 0 PCIE

ConfPro GUI

Board / System: haps6x

Utils & Tests: setvcc

Run

Set IO voltage. The default region is ALL.

Voltage: 1V5

Region:

- V1A
- V1B
- V1C
- V2A
- V2B
- V2C
- V3A

The equivalent Tcl command syntax is:

haps setvcc *voltage* [*region*]

setclk

Sets the frequency for the global input clock identified by the Clock name entry with the frequency specified in the Frequency field. The frequency value is in kHz unless specified otherwise.

UMRBus Device: 0 PCIE

ConfPro GUI

Board / System: haps6x

Utils & Tests: setclk

Run

Set frequency for an global input clock

Clock name: GCLK1

Frequency: 100MHz

The equivalent Tcl command syntax is:

haps setclk *clockName* *frequency*

restart

Restarts the board. The equivalent Tcl command syntax is:

haps restart

confscr

Runs confprosh Tcl scripts. For example, the confprosh command can be used to source a HAPS clock and voltage-region configuration script; the user could then run clock checks to verify the on-board clock configuration. The name of the script is entered in the TCL script field; use the Open button to browse to the script location.

The screenshot shows a GUI for the 'confscr' command. On the left, there are three dropdown menus: 'UMRBus Device' set to '0 PCIE', 'Board / System' set to 'haps6x', and 'Utils & Tests' set to 'confscr'. Below these is a 'Run' button. To the right of the dropdowns is a 'ConfPro GUI' button. Further right, the text 'Run confprosh tcl scripts' is displayed. Below this text is a text field for 'TCL script' and an 'Open' button.

The equivalent Tcl command syntax is:

```
haps confscr scriptFileName
```

vbgen

Queries the HAPS system and generates a corresponding Tcl file for Certify board file generation. The output Tcl file is written to the filename specified in the Output TCL file field. Clicking the Save button prompts for an alternate location to save the Tcl file (by default, the Tcl file is saved to the current working directory).

The screenshot shows a GUI for the 'vbgen' command. On the left, there are three dropdown menus: 'UMRBus Device' set to '0 PCIE', 'Board / System' set to 'haps6x', and 'Utils & Tests' set to 'vbgen'. Below these is a 'Run' button. To the right of the dropdowns is a 'ConfPro GUI' button. Further right, the text 'Query the board and create board-generation tcl file' is displayed. Below this text is a text field for 'Output TCL file' containing 'board_gen.tcl' and a 'Save' button.

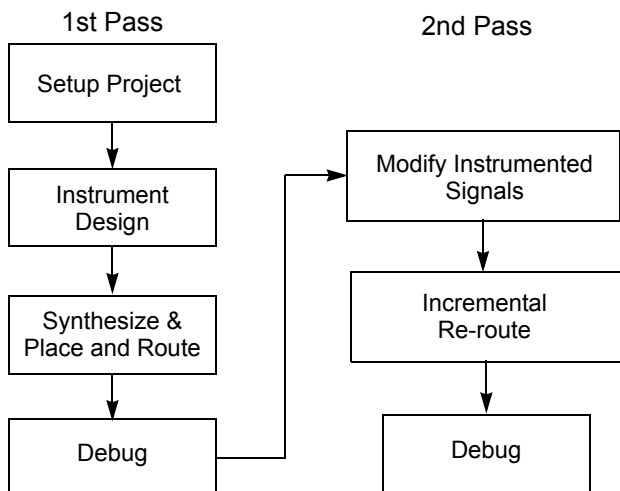
The equivalent Tcl command syntax is:

```
haps vbgen tclFile
```


CHAPTER 3

Incremental Flow

Incremental flow is a multi-pass flow available for the Xilinx technologies that allows you to make minor changes to the set of instrumented signals without needing to resynthesize and rerun place and route on your entire design. With the incremental flow, signals within the initial pass can be replaced with a set of different signals for a subsequent pass. The incremental flow is shown in the following figure.



Requirements

The incremental flow supported by the instrumentor/debugger tool set is available only when using a Xilinx Virtex-7 technology.

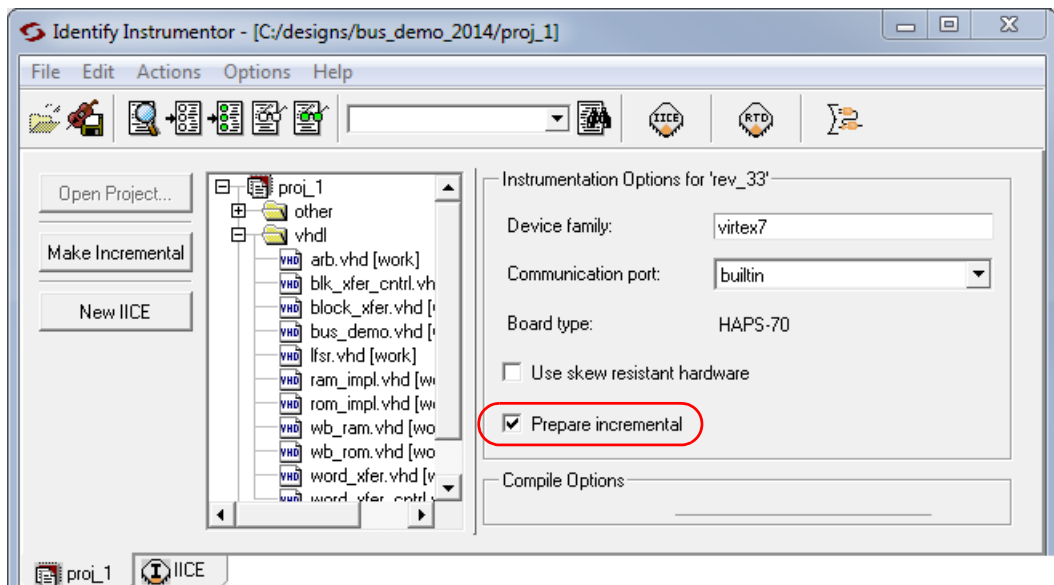
Note: Incremental flow is supported by the most recent version of the Xilinx Vivado software; see the release notes for the specific Vivado release version.

To work with Xilinx Virtex-7 devices, the Vivado place-and-route flow is required. To create an incremental implementation within this flow, a dcp file is required.

Incremental Flow Modes

The incremental flow can be used with or without enabling the prepare incremental function in the instrumentor.

- When the prepare incremental function is enabled (either by checking the Prepare incremental check box in the Project view or using the device prepare_incremental command with a value of 1), essentially any of the signals in the base (original) instrumentation can be re-instrumented.



- When the prepare incremental function is disabled, only a subset of the instrument signals in the base instrumentation can be reinstrumented.

Setting up the Original Design

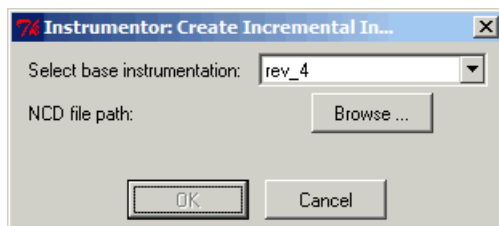
To use the incremental flow, you must set the following in the instrumentor before you instrument and place and route your original design:

1. Open the design in the instrumentor by clicking on the Identify implementation and selecting Launch Identify Instrumentor from the popup menu.
2. Select the incremental flow mode (see [Incremental Flow Modes](#), on page 76).
3. Instrument and save your original design and close the instrumentor.
4. From the synthesis tool:
 - right click on the Identify implementation and select Add Place & Route from the popup menu
 - enable the Xilinx P & R check box
 - click the Run button to run Xilinx place-and-route and generate the dcp file

Creating the Incremental Instrumentation

To make incremental changes to your original design:

1. From the synthesis tool, open the instrumentor by clicking on the original implementation and selecting Launch Identify Instrumentor from the popup menu.
2. Open the project view and click the Make Incremental button to display the Create Incremental Implementation dialog box.



3. In the dialog box, select the base (original) instrumentation and use the Browse button to select the path to the Xilinx dcp file.

Note: Make sure that you enter the path to the correct `ncd` or `dcp` file.

4. Click OK to close the dialog box.
5. At this point, a new incremental icon appears in the project window. This icon is labeled *incr_baseinstr* where *baseinstr* is the name of the initial instrumentation. Click on the icon to display the new instrumentation.

Redefining the Instrumented Signals

In the new instrumentation, most of the registered signals and most of the I/O pins will be available. The console window reports status of the three “buckets” (sample only, trigger only, and sample and trigger); for every new signal or bit that you add to a bucket, you must remove at least one existing signal or bit (the number of signals/bits in a bucket cannot exceed the original number).

Note: You cannot change the device or IICE configuration in the new instrumentation.

When you have finished removing and adding signals, save the new instrumentation. This action invokes the FPGA editor and runs incremental routing in the background and creates a new `ncd` or `dcp` file in the instrumentation directory. Use this file to generate your new bit file.

Generating the Bit File

When a new implementation is saved in the Vivado flow, the `dcp` files are updated with the modifications. At this point, both the original and incremental versions are available for debugging from their corresponding directories. Go to the appropriate directory and use the following command sequence to generate the desired bit file:

```
vivado -mode tcl  
  
read_checkpoint dcpFilename  
  
write_bitstream bitFilename
```

The first command opens Vivado in Tcl mode; the `read_checkpoint` command reads the dcp file, and the `write_bitstream` command writes out the bit file. With distributed instrumentation, this sequence must be repeated for each FPGA.

Incremental Implementation Support with Distributed Instrumentation

To use incremental instrumentation with distributed implementation, every FPGA must be synthesized. The basic distributed instrumentation flow with incremental instrumentation is:

1. Launch the instrumentor.
2. Create an incremental implementation for each FPGA by selecting the appropriate dcp file.
3. Re-instrument any desired signals noting which signals are partitioned into which FPGAs.
4. Save the instrumentation; the Vivado (for V7) incremental routing script is called separately for each individual FPGA and run in the background.

Debugging the Incremental Version

In the debugger, open the project, load the incremental instrumentation (the most recent instrumentation is loaded by default), and debug as normal.

Incremental Flow Limitations

The following limitations are present when using the incremental flow:

- The incremental flow is only available when using the Xilinx Virtex-7 technology.
- Removing the instrumentation from a signal assigned to a MUX group automatically deletes the signal from its assigned group. If the replacement signal is part of the incremental update, the signal must be assigned to the same MUX group as the deleted signal (the sampling logic in the IICE implementation is the only available resource for the new signal).

- A re-instrumented signal cannot exceed the bit-width of a removed signal.
- The real-time debugging feature, which provides scope or logic analyzer access to instrumented signals directly through a Mictor board interface connector installed on the HAPS board, cannot be used with the incremental flow.

CHAPTER 4

IICE Hardware Description

The instrumentor adds instrumentation logic to your HDL design that allows you to understand and debug design operation. There are some aspects of the instrumentation logic that are important to understand in order to use the debug environment tool set in the most effective way. In this chapter, the overall instrumentation logic is described briefly followed by descriptions of some of the more important features. A simplified functional breakdown of the instrumentation logic consists of:

- [JTAG Communication Block](#)
- [Breakpoint and Watchpoint Blocks](#)
- [Sampling Block](#)
- [Complex Counter](#)
- [State Machine Triggering](#)

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 5, *Connecting to the Target System*](#), for more information on the JTAG controller.

Breakpoint and Watchpoint Blocks

The following topics are described in this section:

- [Breakpoints](#)
- [Watchpoints](#), on page 83
- [Multiple Activated Breakpoints and Watchpoints](#), on page 83

Breakpoints

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.

```
99 process(op_code, cc, result) begin
100     case op_code is
101         when "0100" =>
102             result <= part_res;
103             if cc = '1' then
104                 c_flag <= carry;
105                 if result = zero then
106                     z_flag <= '1';
107                 else
108                     z_flag <= '0';
109                 end if;
110             end if;
```

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)
```

Watchpoints

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.

Multiple Activated Breakpoints and Watchpoints

How breakpoints and watchpoints operate individually is described in the *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').

Sampling Block

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.

Whenever possible, the sample block should use the built-in RAM blocks that are available in most programmable chips. Otherwise, implementing the sample buffer using individual storage elements will consume large amounts of the logic capacity of the chip. If you have no choice but to use individual storage elements, analyze how much logic you have available on your chip and adjust how many signals you sample and the depth of the sample buffer.

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 IICE Configuration 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

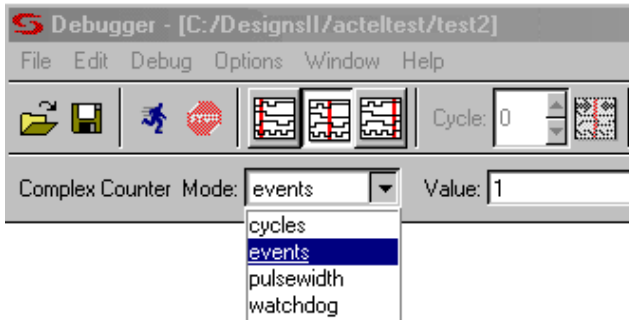
The complex counter is used to produce complex triggering behavior. During the debugging of the design, the complex counter is set to zero on invocation of the debugger run command. Then, it counts events from the Master Trigger Signal event logic in a specific way depending on the counter mode.

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

- 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 main menu.



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 n th trigger event.
cycles	stop sampling on 1st trigger event	stop sampling n cycles after the 1st trigger event.
watchdog	illegal	stop sampling if the trigger condition is not met for n consecutive cycles.
pulsewidth	illegal	stop sampling the first time the trigger condition is met for n consecutive cycles.

events Mode

In the events mode, the number of times the Master Trigger Signal logic produces an event is counted. When the n th 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 n th 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.

State Machine Triggering

This section describes the different methods of triggering available in the debugger. It explains the different choices available during instrumentation and the functionality these choices provide in the debugger as well as discussing the cost effects of the various types of instrumentation.

Simple or Advanced Triggering

There are two triggering modes available, the simple mode and the advanced 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.

The logic that implements breakpoint and signal triggering is referred to as trigger condition in the following text.

In the advanced trigger mode, multiple such trigger conditions are instrumented, and a runtime-programmable state machine is also instrumented to allow you to specify the temporal and logical behavior that combines these trigger conditions into a complex trigger function. For instance, this state machine enables you to trigger on a certain sequence of events like “trigger if pattern A occurs exactly five cycles after pattern B, but only if pattern C does not intervene.”

By default, the instrumentor instruments the design according to the simple trigger mode. See the following for more information on how to select the advanced trigger mode.

Advanced Triggering Mode

Setting up an instrumented design to enable advanced triggering is extremely easy. There are two iice controller command options available in the instrumentor that control the extent and cost of the instrumentation:

- **-triggerconditions** *integer* – The *integer* argument to this option defines how many trigger conditions are created. The range is from 1 to 16. All these trigger conditions are identical in terms of signals and breakpoints connected to them, but they can be programmed separately in the debugger.
- **-triggerstates** *integer* – The *integer* argument to this option defines how many states the trigger state machine will have. The range is 2 to 16; powers of 2 are preferable as other numbers limit functionality and do not provide any cost savings.

Similar to the simple-triggering mode, a counter can be instrumented to augment the functionality of the state machine. To instrument a counter, enter an iice controller -counterwidth option with an argument greater than 0 in the instrumentor console window.

Please refer to the following text to determine cost and consequences of these settings in the instrumentor.

Structural Implementation of State Machine Triggering

For each trigger condition c_i , a logic cone is implemented which evaluates the signals and the breakpoints connected to the trigger logic and culminates in a 1-bit result identical to the trigger condition in simple mode. All these 1-bit results are connected to the address inputs of a RAM table.

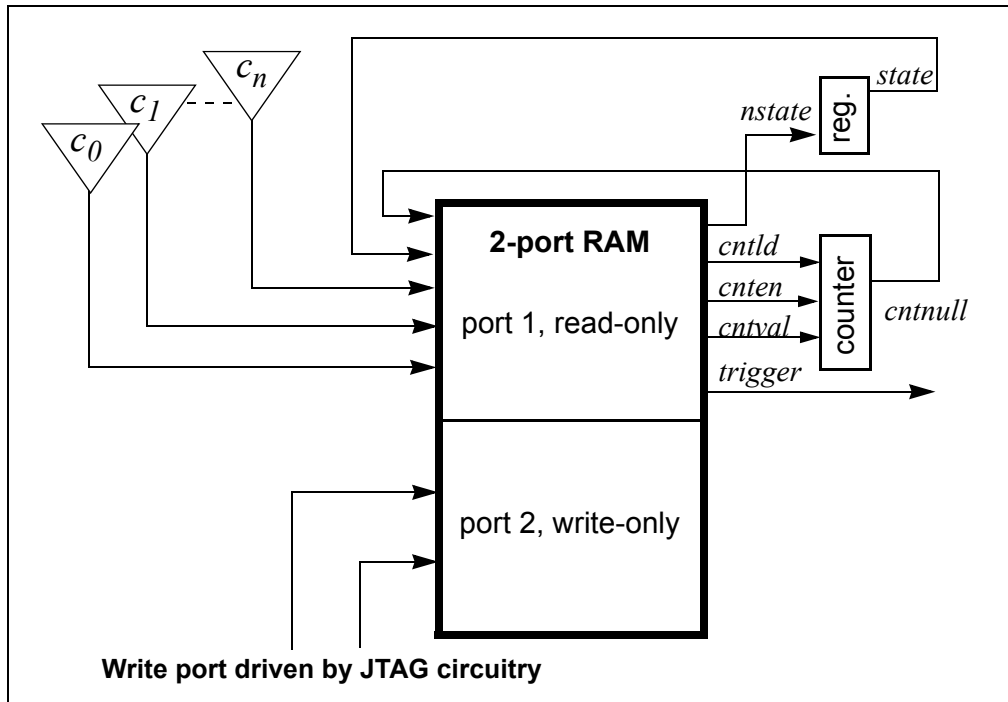
If a counter has been added to the instrumentation, the counter output is compared to constant 0, and the single-bit output of that comparison is also connected to the address inputs of the same RAM table.

The other address inputs are provided by the state register. The outputs of the RAM table are:

- the next-state value *nstate*
- the trigger signal *trigger* (causes the sample buffer to take a snapshot if high)
- the counter-enable signal *cnten* (if '1', counter is decremented by 1)

- the counter-load signal `cntld` (if '1', counter is loaded with `cntval`)
- the counter value `cntval` (only useful in conjunction with `cntld`)

The last three outputs are only present if a counter is instrumented. Please also refer to the figure below.



The implementation of the RAM table is identical to the implementation of the sample buffer (that is, the device `buffertype` setting selects the implementation of both the sample buffer and the state-machine RAM table).

Cost Estimation

The most critical setting with respect to cost is the number of trigger conditions, as each trigger condition results in an additional address bit on the RAM, and thus doubles the size of the RAM table with each bit. Next in importance is the counter width as this factor contributes directly to RAM table width and is especially significant in the context of FPGA RAM primitives that allow a trade-off of width for depth.

The block RAM on Xilinx Virtex-II, Virtex-II Pro, Virtex-4, and Spartan-6 devices includes 18k bits per block and a number of different possible configurations (Virtex-5, Virtex-6, and Virtex-7 devices include 36k bits per block). The following table provides some hints for good, trigger state-machine settings for the smaller, 18k-bit devices when using only a single block for the trigger-state machine.

Table 1: Xilinx Virtex-II, Virtex-II Pro, Virtex-4, and Spartan-6 devices

RAM size			With counter			Without counter	
Address	Depth	Data	Conditions	States	Counter	Conditions	States
9-bit	512	36-bit	5	8	30-bit	no useful setting	
9-bit	512	36-bit	6	4	31-bit	no useful setting	
10-bit	1024	18-bit	6	8	12-bit	no useful setting	
10-bit	1024	18-bit	7	4	13-bit	no useful setting	
11-bit	2048	9-bit	7	8	3-bit	7	16
12-bit	4096	4-bit	n/a	n/a	n/a	9	8

The actual instrumentation, however, is not limited to the values provided, nor is it limited to the use of a single block RAM (for example, it may be advantageous in a particular situation to trade away states for additional trigger conditions or for additional counter width). Any configuration can be automatically implemented, as long as it fits on the device with the remainder of the design.

Although RAM parameters are automatically determined by the instrumentor, this information should be monitored to make sure that no resources are wasted.

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-processible 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 99](#)). 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|titriggerInID*"] [**-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|titriggerInID*" – 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**, . . . **cn**, where *n* is the number of trigger conditions instrumented. These variables represent the output bit of the respective trigger condition.
- **titriggerInID** – 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).

Please 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"
```

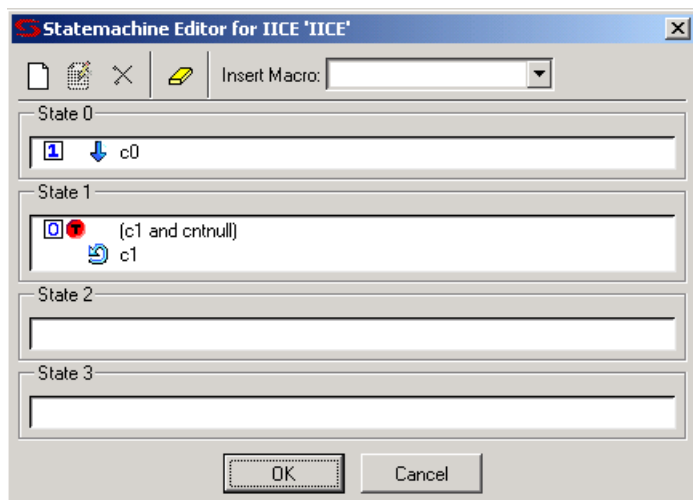
The incorporation of a counter in the state-machine configuration is similar to the use of a counter in non-cross trigger mode for a state machine.

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 Trigger icon in the debugger toolbar. Note that the icon will be grayed out if state-machine triggering was not enabled 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 Statemachine Editor dialog box for the selected IICE.



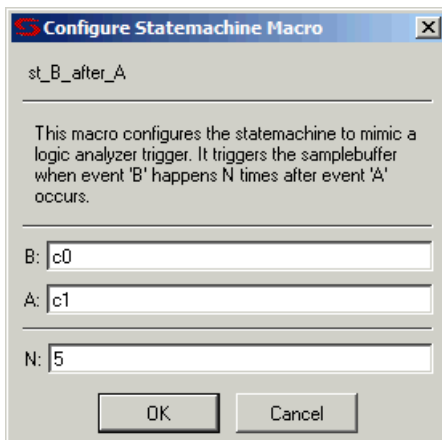
Each state is defined in an individual entry field. Within each entry, you can add multiple definitions for transitioning from that state. Each transition includes either one or two actions and a condition. The actions and conditions are defined in the following tables.

Action	Description
 Decrement Counter	Decrements counter when condition is true (mutually exclusive with Initialize Counter)
 Initialize Counter	Initializes counter to count specified by statemachine transition editor (mutually exclusive with Decrement Counter)
 Trigger Sample Buffer	Triggers sample buffer when condition is true
 Go to State	Transitions to specified state when condition is true

Condition	Description
c0 ... cN	References trigger event in active IICE unit
cnnull	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)
titriggerInID	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 93)

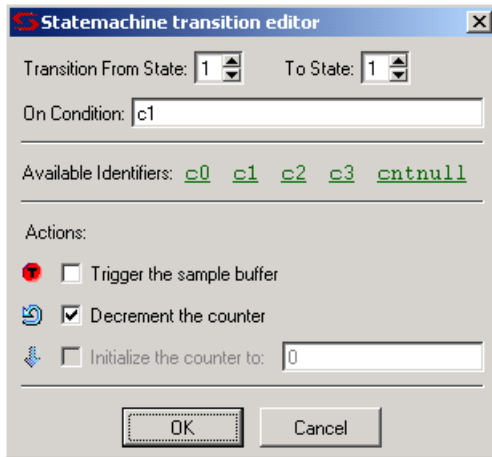
To use the dialog box:

- As an optional starting point, use Insert Macro to select predefined state-machine behaviors from the drop-down list. When a macro is selected, a corresponding Configure Statemachine Macro dialog box is displayed to set the parameters for the macro. The following figure shows the dialog box for the st_B_after_A macro.



Enter the required parameters into the dialog box. These parameters include events, Boolean functions, transition count, and IICE unit. Click OK after all of the parameters are entered.

- Use the Add new transition, Edit current transition, and Delete current transition icons as required. The Add new transition and Edit current transition icons bring up the Statemachine transition editor dialog box which allows transitions to be defined or redefined.



Click OK when the transition has been defined/redefined.

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

Note that you can view the corresponding state-machine commands in the debugger console 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$
```

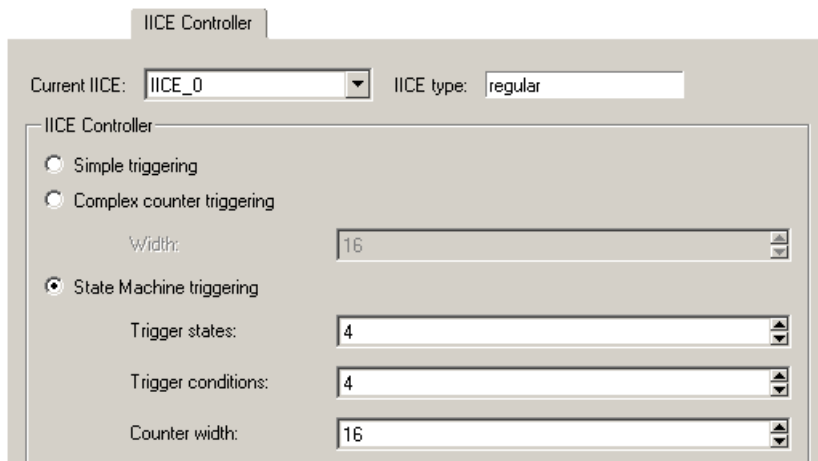
State-Machine Examples

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 Actions->Configure IICE from the top menu bar or click the IICE icon.
 - From the instrumentor Configure IICE 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.



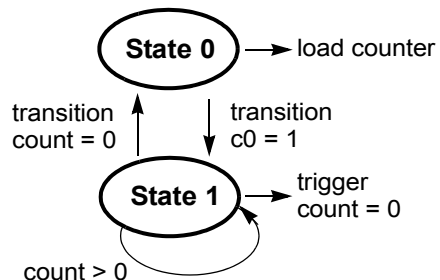
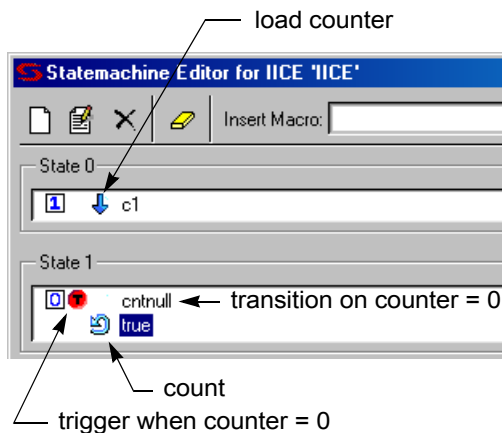
2. Build the state machine trigger from the debugger console window. The following debugger command sequence is an example.

```
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
```

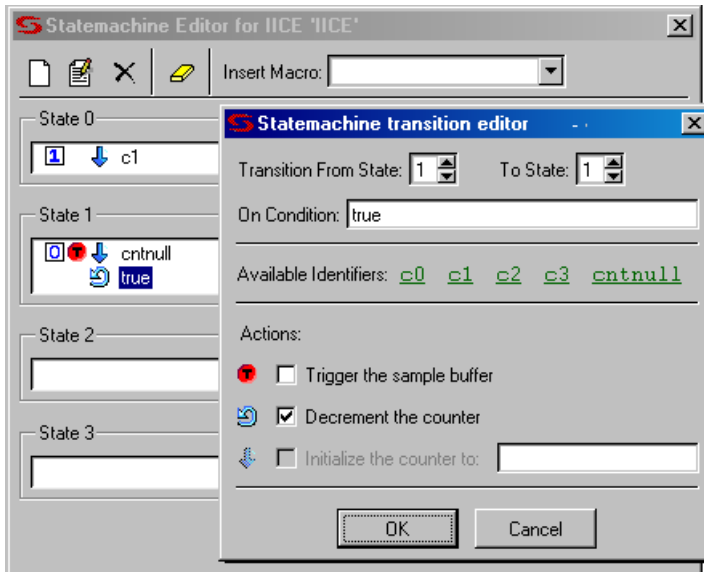
Note that in the last debugger `statemachine` command, the `-to 1` can be omitted (unnecessary because there is no change in state) and that because the `-from` states are the same in the second and third commands, execution falls through to the third command when the second condition is not true.

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

The state-machine editor in the debugger GUI can be used to define the state-machine trigger event described in step 3 as shown in the following figure.



The following figure shows the state-machine transition editor (click the Add new transition icon).



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:

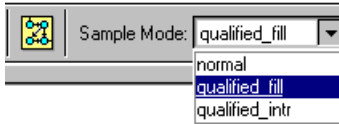
```
iice sampler -depth 2048
```

Note that the only option for buffer type is `internal_memory`.

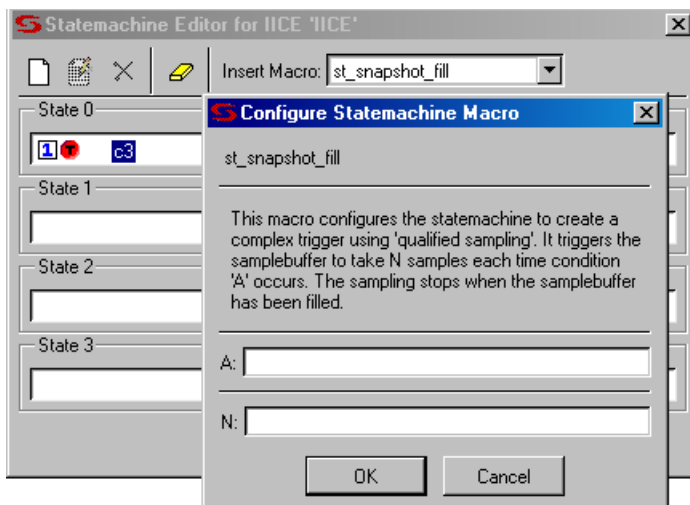
Qualified Sampling

During qualified sampling, data is sampled on every clock. 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 Configure IICE 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. From the debugger GUI, select `qualified_fill` from the Sample Mode drop-down menu.



3. From the debugger GUI, click on the adjacent Configure Statemachine Trigger icon and define the state-machine trigger event.
4. From the debugger GUI, select the `st_snapshot_fill` macro from the Insert Macro drop-down menu.



Enter the trigger event (the condition that will be the qualifying trigger) in field A, enter the number of samples to be accumulated in the sample buffer after the trigger event occurs in field N, and click OK to update the state-machine definition.

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

Note: If you use the debugger `st_snapshot_intr` macro in place of the `st_snapshot_fill` macro, the sample buffer is continually overwritten until manually interrupted by a stop command.

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 "c1 cntnull" -cnten
statemachine addtrans -iice IICE -from 2 -to 2
    -cond "cntnull" -cntval 15 -trigger
```

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 *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 originating 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 SolvNet.

CHAPTER 5

Connecting to the Target System

This chapter describes methods to connect the debugger to the target hardware system. The programmable device or devices in the target system that contain 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](#)
- [JTAG Communication](#)
- [JTAG Hardware in Instrumented Designs](#)
- [Using the Built-in JTAG Port](#)
- [Using the Synopsys Debug Port](#)
- [JTAG Communication Debugging](#)
- [UMRBus Communications Interface](#)

Basic Communication Connection

The components that make up the debugging system are:

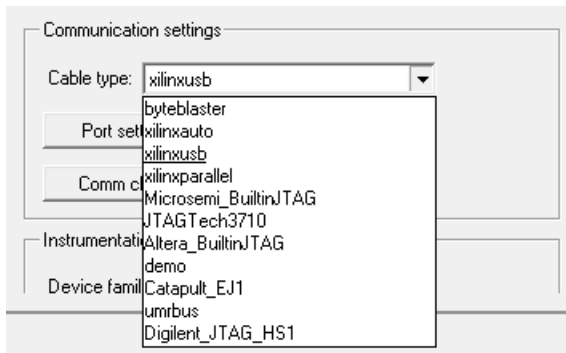
- The host machine running the debug environment with a loaded project.
- The communication cable connecting the host machine to the programmable device.
- The programmable device or devices loaded with the instrumented version of the design to be debugged.

Debugger Communications Settings

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

Cable Type

The cable type is selected from a drop-down menu in the Communications settings area of the debugger project window (see following figure).



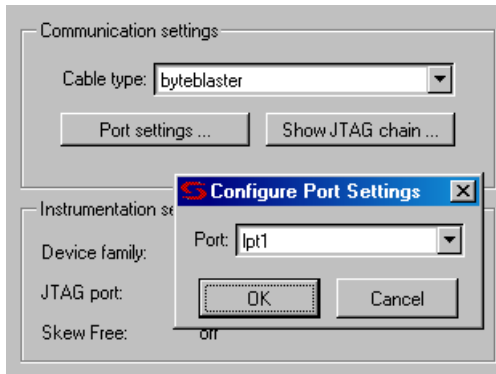
The following table lists the correspondence between cable-type setting and the supported cables in the debugger.

Cable Type Setting	Compatible Hardware Cables
byteblaster (soft JTAG port)	Altera ByteBlaster and ByteBlaster MV
xilinxauto	Auto selection of Xilinx USB or parallel cable
xilinxusb	Xilinx USB (Windows only)
xilinxparallel	Xilinx Parallel III and Xilinx Parallel IV
Microsemi_BuiltinJTAG	Microsemi FlashPro, FlashProLite, or FlashPro3
JTAGTech3710	JTAGTech3710
Altera_BuiltinJTAG (builtin JTAG port)	Altera MasterBlaster (parallel, serial, or USB) or Altera USBBlaster
Catapult_EJ1	Standard Ethernet cable in an IP network
Diligent_JTAG_HS1	Diligent cable for Xilinx FPGAs

If you are using the command interface, set the `com` command's `cabletype` option to `byteblaster`, `xilinxauto`, `xilinxusb`, `xilinxparallel`, `Microsemi_BuiltinJTAG`, `JTAGTech3710`, `Altera_BuiltinJTAG`, `Catapult_EJ1`, `Diligent_JTAG_HS1`, or `demo` according to the cable being used. Note that if you are using the Altera builtin JTAG port, any Altera cable type can be used (communications are controlled through the Quartus driver). If you are using the soft JTAG port, you must use either a ByteBlaster or ByteBlaster MV hardware cable.

Byteblaster Cable Setting

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

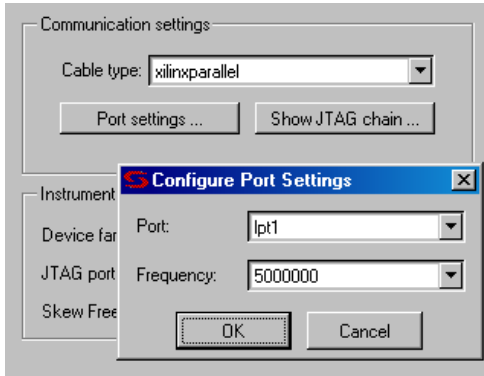


If you are using the command interface, set the `com` command's `cableoptions` `byteblaster_port` option to 1 (lpt1), 2 (lpt2), 3 (lpt3), or 4 (lpt4). Different computers have their lpt ports defined for different address ranges so the port you use depends on how your computer is configured.

The 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).

Xilinx Parallel Cable Settings

To configure a Xilinx parallel cable, click the Port Settings button to display the Configure Port Settings dialog box and select the appropriate port and communication speed frequency from the drop-down menu (see following figure).

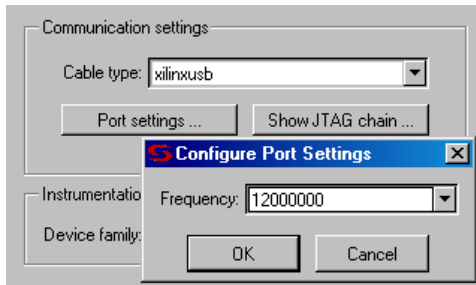


If you are using the command interface, set the com command's port cable options `xilinxparallel_port` option to 1 (lpt1), 2 (lpt2), 3 (lpt3), or 4 (lpt4) and set the `xilinxparallel_speed` option to 5000000 (5MHz), 2500000 (2.5 MHz), or 200000 (200kHz). Note that different computers have their lpt ports defined for different address ranges so the port you use depends on how your computer is configured.

The 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).

Xilinx USB Cable Setting

To configure a Xilinx USB cable, click the Port Settings button to display the Configure Port Settings dialog box and select the appropriate communication speed frequency from the drop-down menu (see following figure).

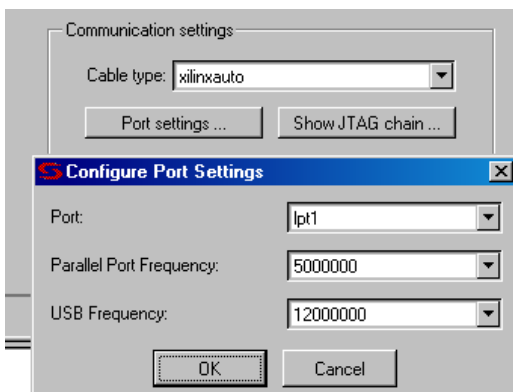


Note: The Xilinx USB cable is only supported on the Windows platform.

If you are using the command interface, set the `com` command's port cable options `xilinxusb_speed` option to 24000000 (24MHz), 12000000 (12 MHz), 6000000 (6 MHz), 3000000 (3 MHz), 1500000 (1.5MHz), or 750000 (750 kHz).

Xilinxauto Cable Settings

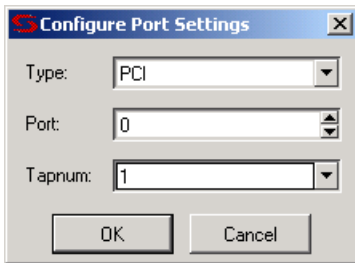
Selecting the Xilinxauto cable type allows the debugger to dynamically select the appropriate Xilinx cable (parallel or USB) for the hardware configuration. From the project window, click the Port Settings button to display the Configure Port Settings dialog box and select the appropriate parallel port and communication speed frequencies for both the parallel and USB cables from the drop-down menus (see following figure).



If you are using the command interface, set the com command's port cableoptions xilinxparallel_port, xilinxparallel_speed, and xilinxusb_speed options described previously in [Xilinx Parallel Cable Settings, on page 115](#) and [Xilinx USB Cable Setting, on page 115](#).

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 *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.

Diligent_JTAG_HS1 Settings

To configure a Diligent JTAG HS1 cable, select the Diligent_JTAG_HS1 setting from the Cable type drop-down menu. Click the Port Settings button to display the Configure Port Settings dialog box and select the appropriate communication frequency from the drop-down menu.

Demo Cable Settings

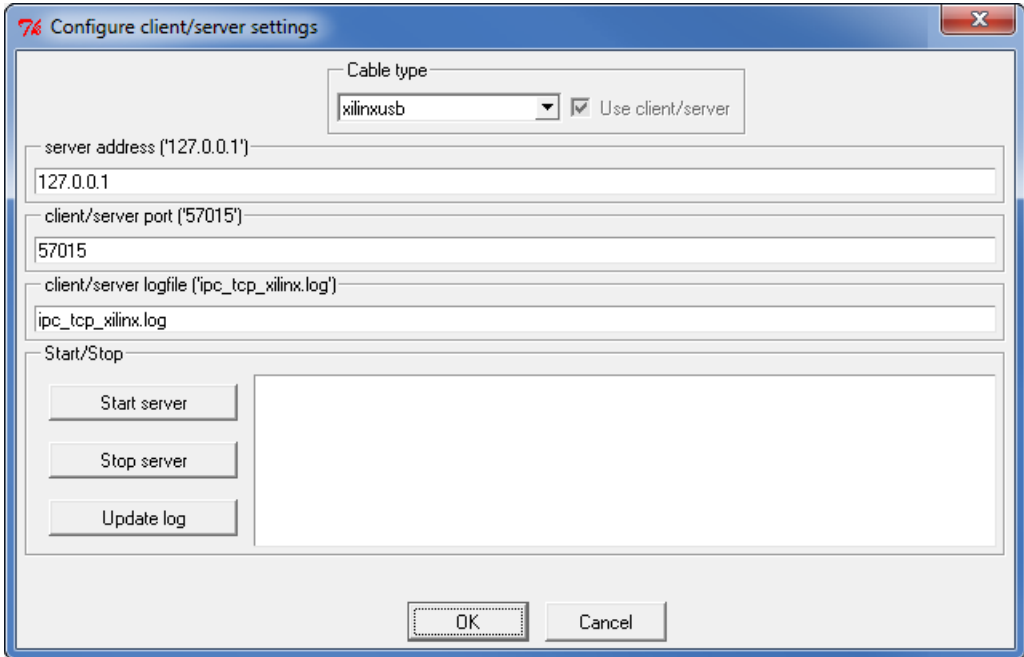
The Port Settings button is disabled when the demo cable is selected.

Debugger Configuration

All parts of the debugging system must be configured correctly to make a successful connection between the debugger and the instrumented device or devices through the cable. In addition to selecting the cable type and port parameters described in [Debugger Communications Settings, on page 112](#), the following additional requirements must be met to ensure proper communications.

Client-Server Configuration

The client-server configuration is set from a dialog box available by selecting Options->Configure client/server server in the debugger. 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 machine that is not the same machine connected to the FPGA board/device (see [Client-Server Configuration for Remote Debugging, on page 120](#)).



In the dialog box:

Cable type – the type of interface cable (see [Cable Type](#), on page 112).

server address – the address of the server. The address localhost is used when the debugger is run on the same machine connected to the FPGA device. The server address is set to the IP address of the machine connected to the FPGA device/board when the debugger is run from a different machine.

client/server port – the port number of the server. For all Xilinx cable types, the default port number is 57015. Change the server port setting when there is a conflict with another tool on the machine.

client/server logfile – the name of the log file.

Start/Stop – server control buttons for starting and stopping the server. The Update log button adds a start/stop entry to the log file.

Client-Server Configuration for Remote Debugging

The debugger uses a client-server architecture to communicate with FPGAs. Client-server architecture lets you work remotely with the debugger using Ethernet as the backbone for client-server communication. The debugger can be configured in either the client or server mode.

In the client-server architecture, the machine connected to the target FPGA board is termed the *server* and any machine on the same network that is used to launch the debugger and connect to the server is *termed* the client. You can use the Configure client/server settings dialog box described in the previous section to set the IP address or the host name of the server so that you can remotely debug the design. You can also specify the port for client-server communication. Client-server communication uses the TCP/IP communication protocol.

Client-Server Configuration

To establish a client-server connection:

1. Configure the target FPGA with the design to be debugged.
2. Start the server on the machine connected to the target FPGA board, launch the debugger, and then configure the server-side debugger as described below:
 - Load the project file (design) to be debugged.
 - In the debugger UI, select Configure client/server from the Options drop-down menu.
 - Specify the server address, port number, and log file name in the Configure client/server settings dialog box. The server address can be either the name of the host machine or its IP address. If you do not know the hostname or IP address, set it to localhost. Set the client/server port according to the selected cable type. Configuring the client-server parameters does not start the server.
 - To start the server, select the Start server button in the dialog box. Alternatively, you can run the com check command by selecting the Comm check button in the debugger project view. If the server starts successfully, you see the xilinxjtag process running in the task manager. 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 project to be debugged. Then configure the client-side debugger as described below:
 - In the debugger UI, select Configure client/server from the Options drop-down menu.
 - Specify the server address, port number, and log file name in the Configure client/server settings dialog box. The server address can be either the name of the host machine or its IP address. The port number must be the same as the port number used to configure the server.

The following is the syntax for the equivalent TCL command 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 by selecting the Comm check button in the debugger project view. If the client-server communication cannot be established, an error message is displayed in the debugger.

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

Parallel/USB Port Drivers

The parallel port or USB driver must be installed and operating (see the installation procedures in the release notes). Make sure the host machine on which you are running the debugger has the parallel port or USB driver installed. If you are using the Altera builtin JTAG, the bin directory for the Quartus software must be included in the users "path" variable.

Communication Cable Connections

The communication cable must be connected correctly. There are two connections:

- Cable-to-host – make sure that the parallel port you connect the cable to corresponds to the `lpt` specified using the `com port` command.

The 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`).

- Cable-to-board – the cable must be connected correctly to the board that contains the programmable device or devices to be debugged. The Altera ByteBlaster cable connects with a 10-pin connector to a special connector on the board. The Xilinx parallel cable (and similar cables) have six flying leads. The leads connect to the four JTAG signals and also to power and ground. Be sure to connect ALL six leads. When you instrumented your design, you selected a JTAG connection to use: `builtin` or `Synopsys debug port (soft)`. If you selected the `builtin` option, connect the cable to the same leads that you use for the JTAG based programming of the chip. If you selected the `Synopsys debug port (soft)`, four JTAG signals were added to the top level of your design. You must assign these signals to pins on the chip that are connected to accessible probe points on your board. Once this is complete, connect the four JTAG signals to the proper probe points, and make sure that you also connect the power and ground leads.

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 will detect 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.

Device Family

If you are using the instrumentor/debugger tool set in stand-alone mode, make sure that the device family (generic, APEX, Virtex-4, ProASIC, ...) 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.

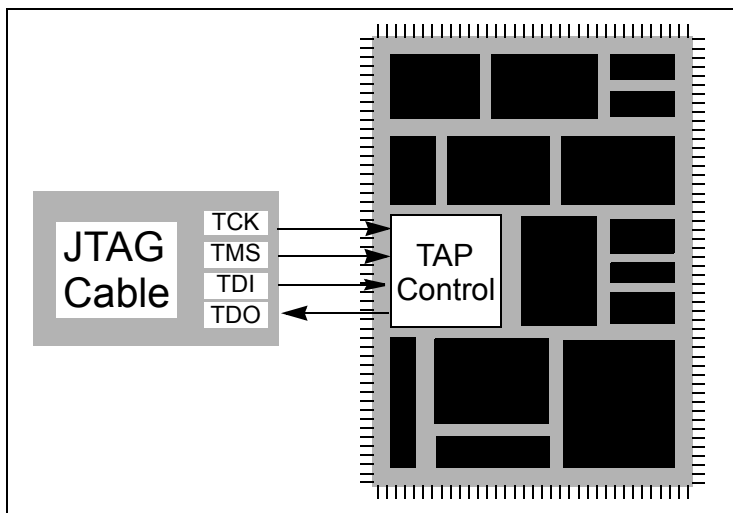
Chip Programming

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

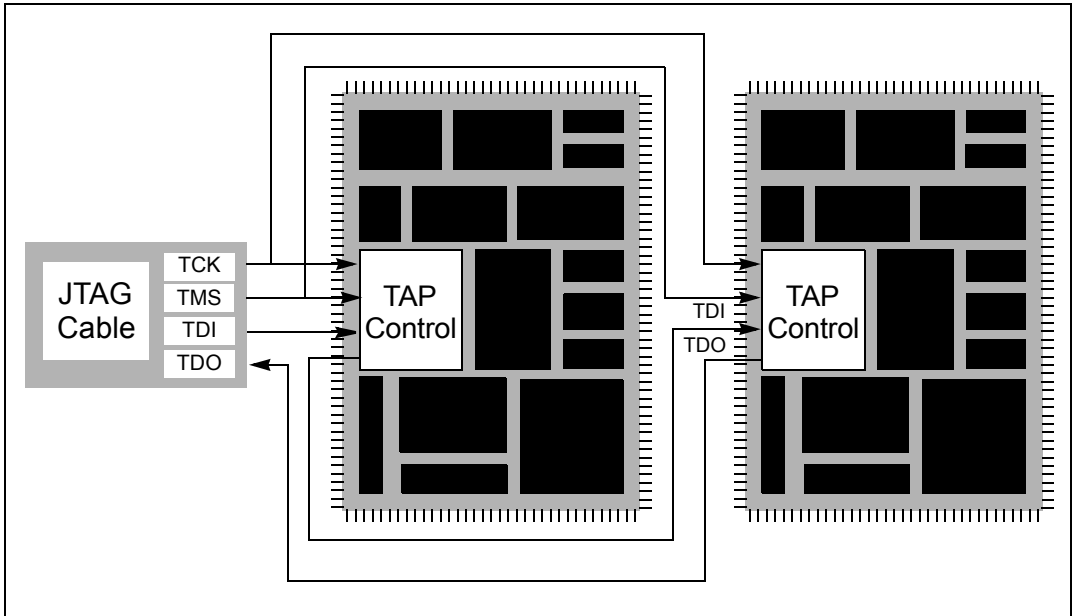
JTAG Communication

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 can be 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 as shown in the figure on the following page.



Notice in the second figure that the TCK and TMS connections are connected directly to both devices while the TDI and TDO connections route from one device to the other and loop back to the JTAG cable.



JTAG Hardware in Instrumented Designs

The debug environment uses a JTAG connection to communicate with the instrumented design. To do this, the IICE must contain a TAP controller that implements 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

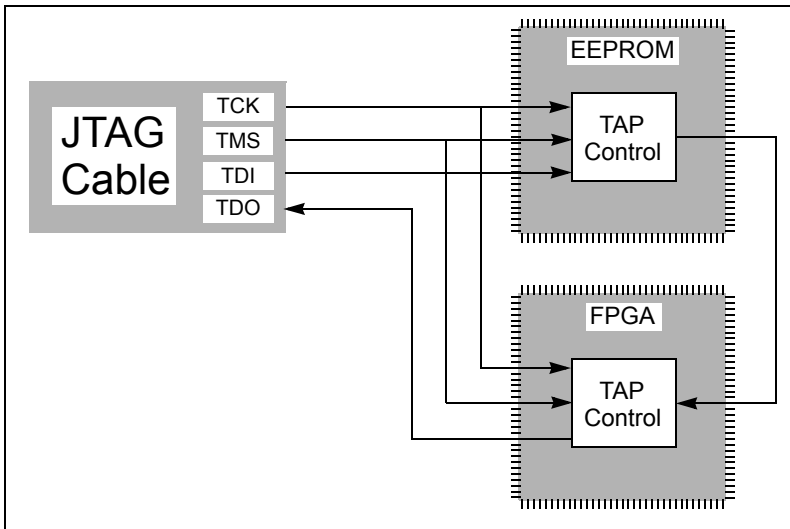
Some programmable device families employ a built-in TAP controller as a means for device configuration. In most cases, the IICE also can be configured to use this built-in TAP controller. Using this TAP controller saves the user logic necessary to implement the controller and also saves four I/O pins.

Using the built-in port is slightly more complicated than using the 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

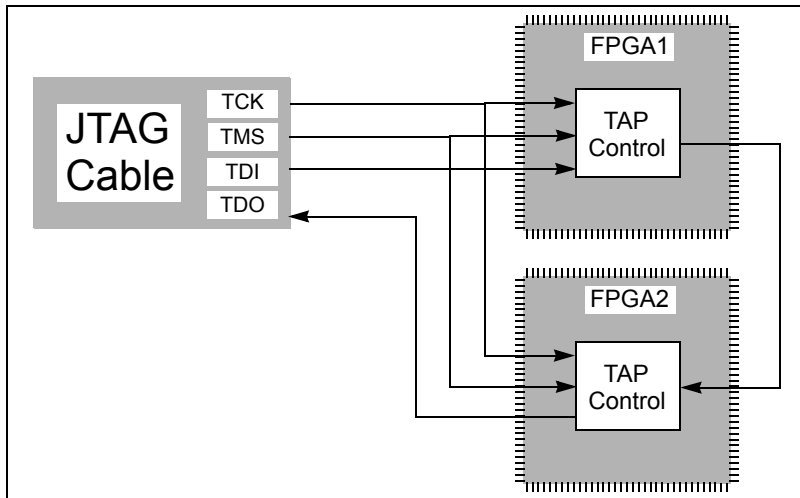
HAPS boards and other boards that connect the built-in JTAG port directly to four header pins on the board allow the JTAG cable to simply be connected directly to the header pins. This configuration works for both directly connected devices and serially chained devices.

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.

Similarly, when using the instrumentor/debugger tool set with a HAPS board in a multi-FPGA environment, the design is distributed among the FPGAs and the instrumented logic is included in one or more of the FPGAs. In this configuration, the IICE unit or units in each FPGA are individually accessed to provide the required debugging capabilities for their associated portion of the design logic.



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.

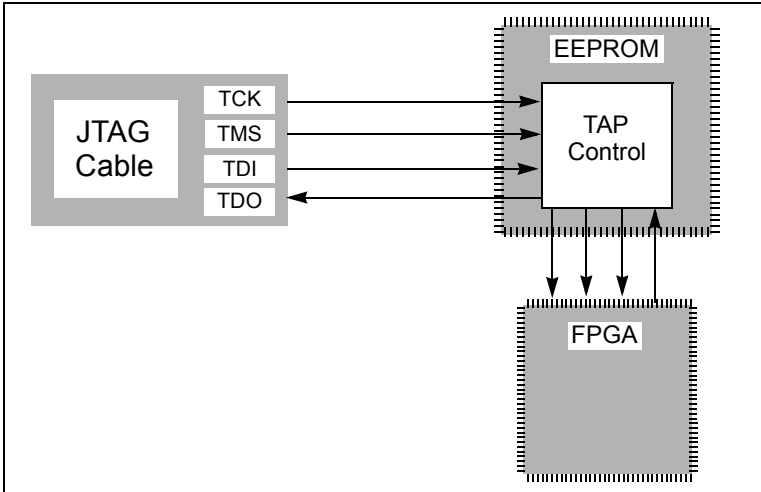
JTAG Registers

Xilinx-based designs allow JTAG boundary-scan registers to be user-defined through the BSCAN_VIRTEX* library macro. After configuring the Xilinx FPGA, these registers are accessible through the JTAG controller's TAP pins. The Virtex-4, Virtex-5, Virtex-6, and Virtex-7 devices include four boundary-scan registers designated USER1, USER2, USER3, and USER4; the other supported Xilinx devices include two registers designated USER1 and USER2.

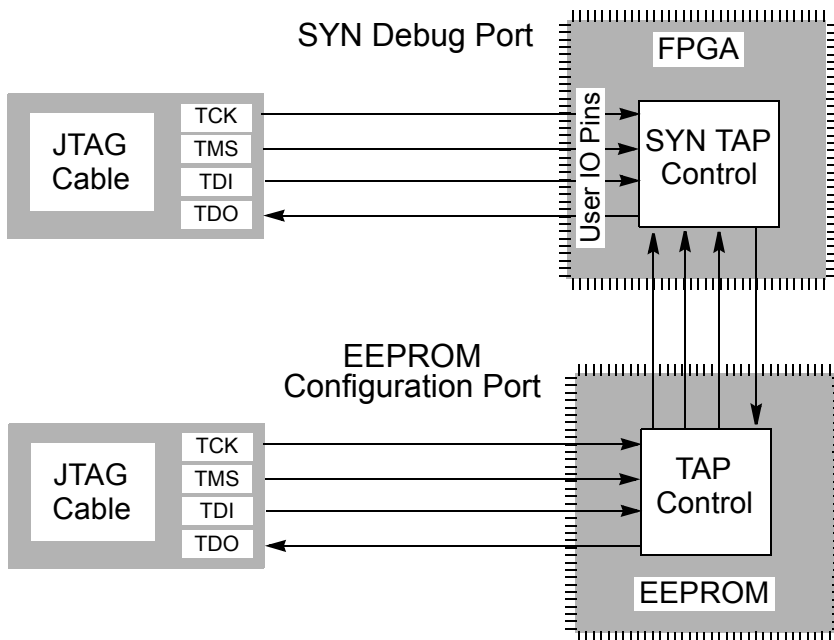
The instrumentor requires two boundary-scan registers. For Virtex-4, Virtex-5, Virtex-6, and Virtex-7 applications, you can specify which two registers are dedicated to avoid any contention among the available registers for user applications (two BSCAN_VIRTEX* cells cannot share the same address). By default, registers USER3 and USER4 are reserved for the instrumentor. To change the default register settings, use the `xilinxjtagaddr1` and `xilinxjtagaddr2` options to the device command (see [device](#), on [page 33](#) of the *Debug Environment Reference Manual*).

Boards Without Direct Built-in JTAG Connections

Some boards are designed so that the built-in JTAG port cannot be reached from pins on the board. For example, a board may connect an EEPROM directly to the built-in JTAG port on the programmable device. The EEPROM is directly programmable from the JTAG connection (see following figure).



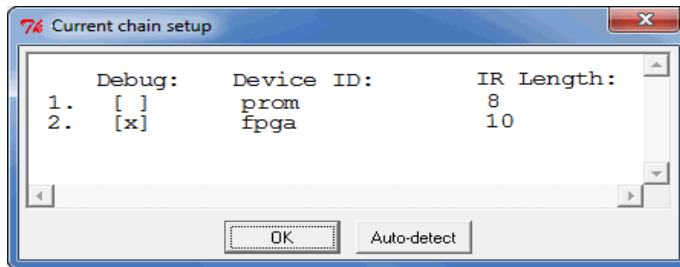
In this case, the only connection that allows the debugger to communicate with the programmable device is a **soft JTAG Port**. This configuration requires a second JTAG cable to directly connect to the four I/O pins on the programmable device as shown in the figure below.



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 JTAG chain button in the Communications settings section of the project window.



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.

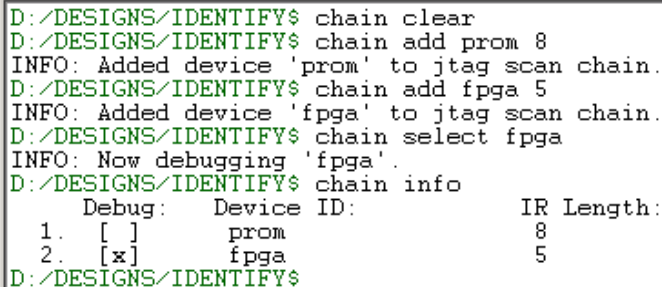
Note: The names PROM and FPGA have no meaning to the debugger – they simply are used for convenience. The two devices could be named device1 and device2, and the debugger would function exactly the same.

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.



```
D:/DESIGNS/IDENTIFY$ chain clear
D:/DESIGNS/IDENTIFY$ chain add prom 8
INFO: Added device 'prom' to jtag scan chain.
D:/DESIGNS/IDENTIFY$ chain add fpga 5
INFO: Added device 'fpga' to jtag scan chain.
D:/DESIGNS/IDENTIFY$ chain select fpga
INFO: Now debugging 'fpga'.
D:/DESIGNS/IDENTIFY$ chain info
      Debug:  Device ID:      IR Length:
1.  [ ]      prom           8
2.  [x]      fpga           5
D:/DESIGNS/IDENTIFY$
```

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 zeroes. 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 project 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.

UMRBus Communications Interface

The UMRBus is available as a communication interface between the HAPS hardware and the host machine running the debugger. With the UMRBus, all communications are performed over the UMRBus communication system, and the JTAG port is no longer used. During instrumentation, the top level of the user design is automatically extended with the additional top-level ports for the UMRBus.

The UMRBus requires a *HAPS UMRBus Interface Kit* to replace the Xilinx USB cable. The UMRBus communications over the USB port supports both the `internal_memory` and `haps_DTD` buffer types; the UMRBus is intended only for use by the debugger. To enable the use of the UMRBus in the debugger:

- In the instrumentor, select `umrbus` from the Port drop-down menu in the Communication interface section of on the Instrumentor Preferences dialog box or set the device `jtagport` option to `umrbus` in the console window.
- In the debugger, set the `com cabletype` option to `umrbus` in the console window.

Index

A

- activations
 - auto-saving [36](#)
 - loading [36](#)
 - saving [35](#)
- asynchronous clocks [97](#)

B

- blocks
 - JTAG communication [81](#)
 - sampling [84](#)
- board bring
 - HAPS [63](#)
- board configuration tests [66](#)
- board query [60](#)
- boundary-scan registers [130](#)
- breakpoints
 - activating [22](#)
 - combined with watchpoints [84](#)
 - folded [25](#)
 - multiple [83](#)
- buckets
 - sample and trigger [78](#)
- Byteblaster cable settings [114](#)

C

- cable compatibility [113](#)
- cable type [112](#)
- cable type settings
 - Byteblaster [114](#)
 - JTAGTech3710 [117](#)
 - Microsemi [117](#)
 - Xilinx parallel [115](#)
 - Xilinx USB [115](#)
 - Xilinxauto [116](#)
- cables
 - connection [122](#)

- client-server configuration [118](#)
- clocks
 - asynchronous [97](#)
- communication cable
 - settings [10](#)
- communications settings [112](#)
- complex counter [85](#)
 - cycles mode [87](#)
 - disabling [87](#)
 - events mode [86](#)
 - modes [85](#)
 - pulsewidth mode [87](#)
 - size [85](#)
 - watchdog mode [87](#)
- con_speed test [67](#)
- condition operators [93](#)
- Configure IICE dialog box [54](#)
- ConfPro [64](#)
- ConfPro installation [65](#)
- connGen.tcl script [68](#)
- console window [16](#)
 - operations [17](#)
- convenience functions [96](#)
- cross triggering [36, 45, 97](#)
 - commands [97](#)
 - enabling [97](#)
 - state machine commands [98](#)
- cycles mode
 - complex counter [87](#)

D

- data compression [28](#)
 - masking [29](#)
- DDR3 performance [40](#)
- debug sample data
 - viewing [41](#)
- Debugger tool

- invoking [12](#)
- debugger tool
 - opening projects [12](#)
- debugging
 - on separate machines [44](#)
- deep trace debug configurations [40](#)
- dialog boxes
 - Configure IICE [54](#)

E

- environment variables
 - PAR_BELDLRPT [77](#)
- events mode
 - complex counter [86](#)

F

- fast signal database [53](#)
- files
 - last_run.adb [36](#)
 - ncd [77](#)
 - script [19](#)
 - syn_trigger_utils.tcl [96](#)
- folded breakpoints [25](#)
- folded signals [32](#)
- folded watchpoints [24](#)

H

- HAPS
 - board bring-up [63](#)

I

- identification register [135](#)
- IICE
 - cross triggering [97](#)
 - JTAG connection [126](#)
- IICE parameters
 - individual [54](#)
- IICE units
 - cross triggering [36](#)
- incremental flow [75](#)
 - restrictions [76](#)

J

- JTAG
 - chain tests [135](#)
 - communication [124](#)
 - communication block [81](#)
 - communication test [134](#)
 - connections [130](#)
 - debugging [134](#)
 - direct connection [128](#)
 - serial connection [129](#)
- JTAG chain
 - settings [11](#)
- JTAG registers [130](#)
- JTAGTech3710 cable settings [117](#)

L

- last_run.adb file [36](#)

M

- macros
 - st_snapshot_fill [107](#)
 - st_snapshot_intr [107](#)
- Microsemil
 - cable type settings [117](#)
- modes
 - cross triggering [37](#)
- multi-IICE
 - tabs [54](#)
- multiple signal values [33](#)
- multiplexed groups
 - selecting [23](#)

N

- ncd file [77](#)

O

- operators
 - condition [93](#)
- original source files
 - searchpath [44](#)
- original sources [44](#)

P

PAR_BELDLYRPT variable 77

projects

opening in debugger 12

saving 12

pulsewidth mode

complex counter 87

Q

qualified sampling 106

R

radix

sampled data 31

RAM resources 85

registers

boundary scan 130

remote triggering 108

run command 26

S

sample and trigger buckets 78

sample buffer 31

trigger position 29

sample data

viewing 41

sample modes 106

sampled data

changing radix 31

compressing 28

display controls 31

masking 29

sampling block 84

sampling signals 15

saving a project 12

script files 19

settings

cable 10

JTAG chain 11

signal values

displaying multiple 33

signals

folded 32

listing available 15

listing instrumented 15

multiply instrumented 33

partially instrumented 34

replacing 75

sampling selection 15

status 92

source files

copying 44

st_snapshot_fill macro 107

st_snapshot_intr macro 107

state machines

transitions 93

triggering 89,91

statemachine command 92

state-machine editor 99

status reporting 92

stop command 28,92

syn_trigger_utils.tcl file 96

T

TAP controller 126

Tcl scripts

connGen.tcl 68

tools

invoking Debugger 12

transition watchpoint 20

trigger conditions 88

triggering

advance mode 89

between IICEs 97

modes 88

remote 108

state machine 89,91

triggers

complex 85

U

UMRBus 136

UMRBus test 67

V

- value watchpoint [20](#)
- variables
 - PAR_BELDLRPT [77](#)
- Verdi nWave viewer [53](#)

W

- watch command [92](#)
- watchdog mode
 - complex counter [87](#)
- watchpoints [83](#)
 - activating [19, 22](#)
 - combined with breakpoints [84](#)
 - deactivating [21](#)
 - folded [24](#)
 - hexadecimal values [21](#)
 - listing [38](#)
 - multiple [84](#)
 - transition [20](#)
 - value [20](#)
- waveform display [51](#)
- waveform viewers [51](#)
 - Verdi [53](#)
- windows
 - console [16](#)

X

- Xilinx parallel cable settings [115](#)
- Xilinx USB cable settings [115](#)
- Xilinxauto cable settings [116](#)