

Identify[®] **User Guide**

June 2013

<http://solvnet.synopsys.com>

SYNOPTSYS[®]

Copyright Notice and Proprietary Information

© 2013 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
June 2013

Contents

Chapter 1: Getting Started

Manual Conventions	12
Text Conventions	12
Syntax Conventions	12
Tool Conventions	13
File System Conventions	13
Design Hierarchy Conventions	14

Chapter 2: System Overview

The Debugging System	18
The Design Flow	19
System Components	21
IICE	21
Identify Instrumentor	22
Identify Debugger	22

Chapter 3: Project Handling

Projects in the Identify Instrumentor	24
Integrated Identify Instrumentor Projects	24
Assisted Identify Instrumentor Projects	26
Viewing a Compiled File	26
Instrumenting and Saving a Project	27
Projects with Distributed Instrumentation	28
Identify Debugger Projects	29
Opening an Identify Debugger Project	29
Configuring an Identify Debugger Project	29
Saving a Project	31

Chapter 4: IICE Configuration

Multiple IICE Units	34
Adding an IICE Unit	34
Deleting an IICE Unit	34
Common IICE Parameters	35
Device Family	35
Communication Port	36
Board Type	36
Use Skew-Resistant Hardware	36
Prepare Incremental	37
Individual IICE Parameters	38
IICE Sampler Tab	38
IICE Controller Tab	43

Chapter 5: HAPS Deep Trace Debug

External Memory Instrumentation and Configuration Steps	47
SRAM Clocks	50
Sample Depth Calculation	51
Sample Clock Calculation	51
Hardware Configuration Verification	52

Chapter 6: Support for Instrumenting HDL

VHDL Instrumentation Limitations	56
Verilog Instrumentation Limitations	58
SystemVerilog Instrumentation Limitations	61

Chapter 7: Identify Instrumentor

Identify Instrumentor Windows	65
Instrumentation Window	66
Project Window	69
Console Window	70
Commands and Procedures	71
Opening Projects	71
Executing Script Files	72
Selecting Signals for Data Sampling	72
Instrumenting Buses	75
Partial Instrumentation	77
Multiplexed Groups	79

Sampling Signals in a Folded Hierarchy	80
Instrumenting Signals Directly in the idc File	82
Selecting Breakpoints	84
Selecting Breakpoints Residing in Folded Hierarchy	84
Configuring the IICE	86
Real-time Debugging	86
Writing the Instrumented Design	90
Synthesizing Instrumented Designs	92
Listing Signals	92
Searching for Design Objects	94
Console Text	96

Chapter 8: Identify Debugger

Invoking the Identify Debugger	98
Synthesis Tool Launch	98
Operating System Invocation	98
Identify Debugger Windows	99
Instrumentation Window	100
Console Window	102
Project Window	102
Commands and Procedures	104
Opening and Saving Projects	104
Executing a Script File	105
Activating/Deactivating an Instrumentation	105
Selecting Multiplexed Instrumentation Sets	109
Activating/Deactivating Folded Instrumentation	110
Run Command	113
Sampled Data Compression	114
Sample Buffer Trigger Position	115
Stop Command	116
Sampled Data Display Controls	117
Displaying Data from Folded Signals	119
Displaying Data for Partial Buses	120
Displaying Data for Partial Instrumentation	120
Saving and Loading Activations	121
Cross Triggering	123
Listing Watchpoints and Signals	125
Show Watchpoint/Breakpoint Icons	126
Debugging on a Different Machine	127
Simultaneous Debugging	128

Identify-Analyst Integration	129
Waveform Display	134
Logic Analyzer Interface Parameters	137
Logic Analyzer Scan Tab	137
Logic Analyzer Properties Tab	138
Logic Analyzer Submit Tab	139
IICE Assignments Report Tab	140
Console Text	141

Chapter 9: Incremental Flow

Requirements	144
Setting up the Original Design	144
Creating the Incremental Instrumentation	145
Redefining the Instrumented Signals	146
Generating the Bitfile	146
Debugging the Incremental Version	147

Chapter 10: IICE Hardware Description

JTAG Communication Block	149
Breakpoint and Watchpoint Blocks	150
Breakpoints	150
Watchpoints	151
Multiply Activated Breakpoints and Watchpoints	151
Sampling Block	152
Complex Counter	153
Creating a Complex Counter	153
Debugging with the Complex Counter	153
Disabling the Counter	155
State Machine Triggering	156
Simple or Advanced Triggering	156
Advanced Triggering Mode	157
State-Machine Editor	167
State-Machine Examples	171

Chapter 11: Connecting to the Target System

Basic Communication Connection	180
Identify Debugger Communications Settings	180
Identify Debugger Configuration	186

JTAG Communication	191
JTAG Hardware in Instrumented Designs	193
Using the Built-in JTAG Port	193
Using the Synopsys Debug Port	195
Boards Without Direct Built-in JTAG Connections	197
Setting the JTAG Chain	199
JTAG Communication Debugging	201
Basic Communication Test	201
On-chip Identification Register	202
JTAG Chain Tests	202
UMRBus Communications Interface	203
HAPS Board Bring-up Utility	204
Setting Initial Values	205
ConfPro GUI	205
Utility and Board-Test Commands	207

CHAPTER 1

Getting Started

The Identify[®] tool set includes the Identify instrumentor and the Identify debugger. These two tools allow you to debug your HDL design:

- in the target system
- at the target speed
- at the VHDL/Verilog RTL source level

The Identify tool set helps you debug any design that is implemented by an electronically programmable logic device such as an FPGA or PLD. For the first time you will be able to debug live hardware with the internal design visibility you need while using intuitive debugging techniques.

This guide provides comprehensive information about how the Identify tool set works within your HDL design flow.

This chapter contains:

- [Manual Conventions](#)
- [Tool Conventions](#)

Manual Conventions

There are several conventions that this manual uses in order to communicate command information.

Text Conventions

There are several text conventions that this manual uses to organize command, path, and directory information. These conventions or text styles are:

This convention...	Organizes this information...
Bold	command titles
Monospacing type	command, path name, and directory examples
Sans-serif type	commands, literals, and keywords
<i>Italics</i>	variable arguments

Syntax Conventions

There are several conventions that this manual uses to convey command syntax. These conventions are:

This convention...	Organizes this information...
bold	Commands and literal arguments entered as shown.
<i>italics</i>	User-defined arguments or example command information.
[]	Optional information or arguments for command use. Do not include these brackets with the command within the command line.

This convention...	Organizes this information...
...	Items that can be repeated any number of times.
	Choices you can make between two items or commands. The items are located on either side of this symbol.
#	Comments concerning the code or information within the command line.

Tool Conventions

There are tool concepts you must familiarize yourself with when using the Identify tool set. These concepts help you to decipher structural and HDL-related information.

File System Conventions

The term file system refers to any command that uses file, directory, or path name information in its argument. A file system reference must follow these conventions:

Path Separator “/”

All file system commands that contain a directory name use only forward slashes, regardless of the underlying operating system:

```
/usr/syn/data.dat
```

```
c:/synopsys/data.dat
```

Wildcards

A wildcard is a command element you can use to represent many different files. You can use these wildcards as arguments to the file system commands. Conventions for wildcards are as follows:

Syntax	Description
*	Matches any sequence of characters
?	Matches any single character

Square brackets are used in pattern matching as follows:

Syntax	Description
[abcd]	Matches any character in the specified set.
[a-d]	Matches any character in a specified range.

To use square brackets in wildcard specifications, you must delimit the entire name with curly braces {}. For example,

```
{ [a-d] 1 }
```

matches any character in the specified range (a-d) preceding the character 1.

Design Hierarchy Conventions

Design hierarchy refers to the structure of your design. Design hierarchy conventions define a way to refer to objects within the design hierarchy.

The Identify tool set supports both VHDL and Verilog. These languages vary in their hierarchy conventions. The VHDL and Verilog languages contain design units and hierarchies of these design units. In VHDL these design units are entity/architecture pairs, in Verilog they are modules. VHDL and Verilog design units are organized hierarchically. Each of the following HDL design units creates a new level in the hierarchy.

VHDL

- The top-level entity
- Architectures
- Component instantiation statements
- Process statements
- Control flow statements: if-then-else, and case
- Subprogram statements
- Block statements

Verilog

- The top-level module
- Module instantiation statements
- Always statements
- Control flow statements: if-then-else, and case
- Functions and tasks

Design Hierarchy References

A reference to an element in the design hierarchy consists of a path made up of references to design units (similar to a file reference described earlier). Regardless of the underlying HDL (VHDL or Verilog), the path separator character is always “/”:

```
/inst/reset_n
```

Absolute path names begin with a path separator character. The top-level design unit is represented by the initial “/”. Thus, a port on the top-level design unit is represented as:

```
/port_name
```

The architecture of the top-level VHDL design unit is represented as:

```
/arch
```

Relative path names do not start with the path separator, and are relative to the current location in the design hierarchy. Initially, the current location is the top-level design unit, but commands exist that allow you to change this location.

Note: Design unit references can be case sensitive depending on the HDL language. VHDL names are not case sensitive. In contrast, Verilog names are case sensitive.

Wildcards

A wildcard is a command element you use to represent many different design hierarchy references. You can use wildcard design references as arguments to the design hierarchy commands. Conventions for wildcards are as follows:

Syntax	Description
*	Matches any sequence of characters
?	Matches any single character

Square brackets are used in hierarchy pattern matching as follows:

Syntax	Description
[abcd]	Matches any character in the specified set.
[a-d]	Matches any character in a specified range.

To use square brackets in pattern matching, you must delimit the entire name with curly braces {}. For example,

```
{ [a-d] 1 }
```

matches any character in the specified range (a-d) preceding the character 1.

CHAPTER 2

System Overview

The Identify tool set is part of an HDL design flow process. The tool set is a dual-component system that allows you to probe your HDL design in the target environment. The system fits easily into your existing design flow, with only a few modifications.

The dual-component system consists of the Identify instrumentor software that allows you to select your design instrumentation at the HDL level and then create an on-chip hardware probe, and the Identify debugger software that interacts with the on-chip hardware probe and allows you to perform live debugging of your design. A combined system of instrumentor and debugger allows you to debug your design faster, easier, and more efficiently.

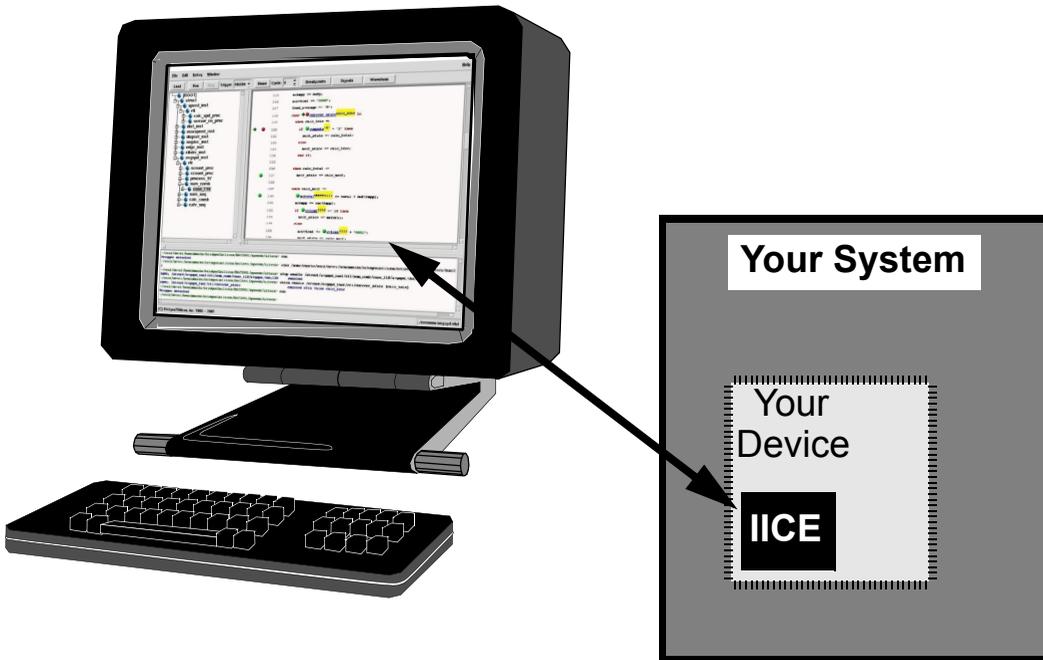
This chapter provides a high-level description of the Identify tool set, detailing how the instrumentor and debugger work and how they fit into your design flow. The chapter describes:

- [The Debugging System](#)
- [The Design Flow](#)
- [System Components](#)

The Debugging System

The Identify tool set is based on the principle of in-system debugging. Using these tools allows you to debug your device in the target system, at target speed while debugging at the HDL level.

Starting with an HDL design, the Identify instrumentor enables you to create a debuggable HDL version of your design. Then, the Identify debugger communicates with this debuggable design, captures the operation of the design, and relates the captured data back to the original HDL design.

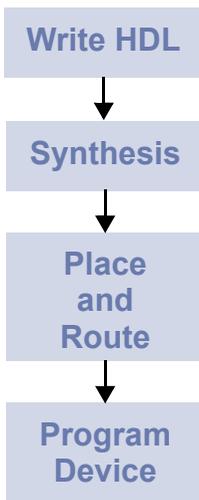


The Identify tool set captures the device operation by inserting an IICE (Intellectual In-Circuit-Emulator) device into your design. The IICE is connected to signals of interest in your design and communicates with a host computer through a JTAG cable.

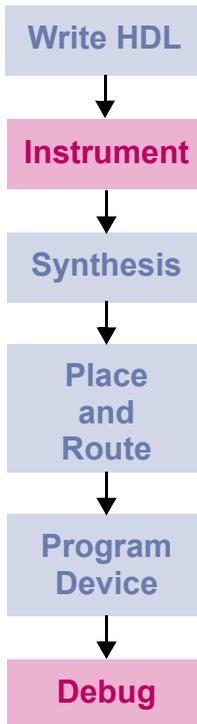
The Design Flow

Design flows for HDL design and debugging vary according to the type of hardware and device you use. Displayed below is the typical HDL design flow for a programmable hardware device and the design flow with the Identify tool set:

Typical Design Flow



Identify Design Flow



In the typical design flow, the first step is to create the HDL design files which are then synthesized to the target device. Following synthesis, the design is placed and routed, targeting a particular device. Then, the design is implemented on the actual hardware by programming the device.

The flow through the Identify tool set makes a very minor change to the typical flow. After the HDL is successfully created, the Identify instrumentor is used to identify the specific signals to be monitored. Saving the project generates an *instrumentation design constraints* (*idc*) file and adds constraint files to the design RTL for the instrumented signals and break points. The design is synthesized and then run through the rest of the typical process. After the device is programmed with the debuggable design, the Identify debugger is used to debug the design while it is running in the target system.

Because the Identify tool set provides logic debugging capabilities in the target system which is running at the target speed, the need for extensive system-level simulation can be reduced. Instead, the effort in the simulation step can concentrate on the much simpler and faster module-level simulation and some system-level simulation. The Identify debugger is used to help debug system-level functionality.

System Components

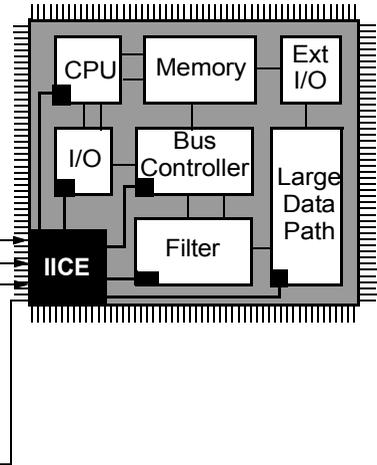
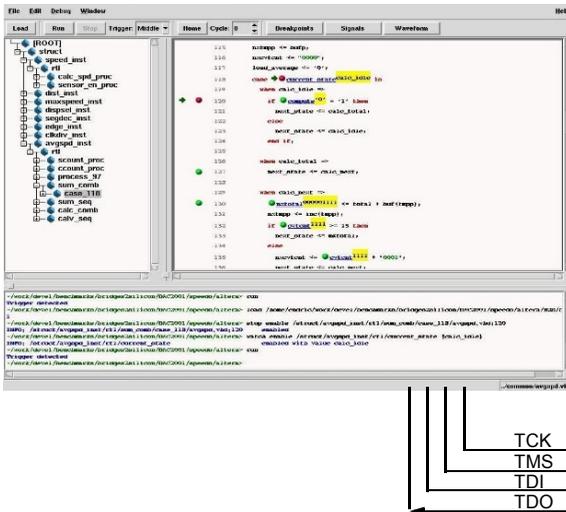
The Identify system is made up of the following:

- [IICE](#)
- [Identify Instrumentor](#)
- [Identify Debugger](#)

IICE

The Intelligent In Circuit Emulator (IICE) is a custom block that is inserted into your design and connected to signals in the design by the Identify instrumentor in accordance with your interface specifications. The IICE samples internal signals and feeds the sampled signal information back to the Identify debugger where the data is transformed for interpretation at the HDL level.

Identify Debugger



The IICE block is actually made up of two different types of blocks. The first is the controller block and the second is the probe block. The probe block samples internal signal data and communicates with the controller block. The controller block communicates with the JTAG port.

The probe block contains the sample buffer where signal value data is stored. The probe block also contains the trigger logic that determines when the signal data is stored in the sample buffer.

The controller block receives the sample data from the probe block and sends it through the JTAG port to the Identify debugger.

Identify Instrumentor

The Identify instrumentor reads and analyzes the pre-synthesis HDL design and provides detailed information about the signals that can be observed and allows you to specify how to control signal observation.

The Identify instrumentor uses the HDL design files and the user-selection information to create the custom IICE block. The Identify instrumentor connects the IICE to the appropriate signals in the design and writes the new design to a user-specified location.

Identify Debugger

The Identify debugger lets you interact with the debuggable hardware at the HDL level. In the Identify debugger, you can activate breakpoints, set watchpoints, and view captured data related to the original source code or as waveforms. In the Identify debugger, you set trigger conditions that determine when to capture signal data. A trigger condition is a set of on-chip signal values or events. When a trigger condition occurs, data is transferred from the hardware device to the Identify debugger through the JTAG communication cable or the UMRBus on a HAPS board.

CHAPTER 3

Project Handling

A project contains all of the information required to instrument and debug a design. This information includes references to the HDL design files, the user-selected instrumentation, the settings used to create the IICE, the watchpoint and breakpoint activations from the debugger, and other system settings. Additionally, you can save the original design, in either an encrypted or non-encrypted format, in a project subdirectory. The project file is used to reproduce the exact state of the project in the Identify tool set.

This chapter describes the procedures for managing projects and includes:

- [Projects in the Identify Instrumentor](#)
- [Identify Debugger Projects](#)

Projects in the Identify Instrumentor

This section describes how projects are defined between the Identify instrumentor and the synthesis tool.

- **Integrated** – an Identify implementation is defined in the Synopsys FPGA synthesis tool and the Identify instrumentor is then launched directly from the Identify implementation in that project.
- **Assisted** – an Identify implementation is defined in the synthesis tool project, the project is saved, and the synthesis tool exited. The Identify instrumentor is then launched independently to open the saved project file.

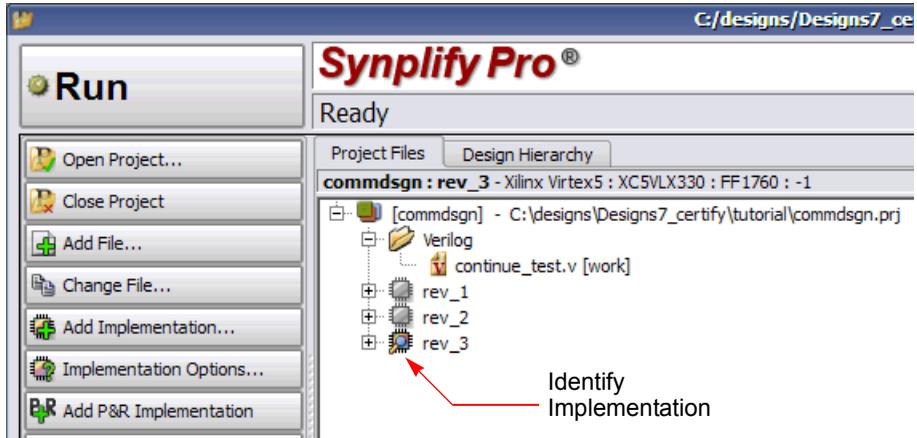
Integrated Identify Instrumentor Projects



For Synplify, Synplify Pro, and Synplify Premier users and users of Certify releases D-2010.06 and later, a Synopsys FPGA project (`prj`) file is passed to the Identify instrumentor to define the project. This project file specifies the design files and their order, the device technology, and the design's top-level module or entity.

To make a Synplify, Synplify Pro, Synplify Premier, or compatible Certify project available to the Identify instrumentor, open the project in the corresponding Synopsys FPGA synthesis tool's Project view and:

1. Right click on the project (Synplify) or implementation (Synplify Pro, Synplify Premier, or Certify) and select **New Identify Implementation** from the popup menu.
2. Set/verify any technology, device mapping, or other pertinent options (see the implementation option descriptions in the Synopsys FPGA synthesis tool user guide) and click **OK**. A new, Identify implementation is added to the project view.



Note: Because multiple implementations are not supported in the Synplify synthesis tool, the existing implementation is replaced with the Identify implementation.

3. Either right click on the Identify implementation and select Launch Identify Instrumentor from the popup menu or click the Launch Identify Instrumentor toolbar icon to launch the Identify instrumentor.

Note: If you are prompted to locate the executable, enter the path to the Identify software in the Locate Identify Installation field (use the browse button to the right of the field). If you are prompted to select a license type, select the appropriate type from the list; check the Save as default license type box to avoid the license prompt in future sessions.

Launching the Identify instrumentor from the Synopsys FPGA synthesis or Certify tool:

- Brings up the Identify instrumentor graphical interface.
- Extracts the list of design files, their work directories, and the device family from the prj file.
- Automatically compiles the design files using the synthesis tool compiler and displays the design hierarchy and the HDL file content with all the potential instrumentation marked and available for selection.

Assisted Identify Instrumentor Projects



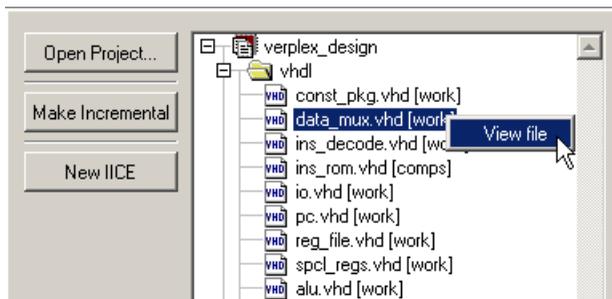
A Synplify, Synplify Pro, Synplify Premier, or Certify project with an existing Identify implementation can be opened directly in the Identify instrumentor. To open a synthesis project directly from the Identify instrumentor, select File->Open project from the main menu, click the Open existing project icon, or click the Open Project button in the project window. Each of these commands brings up the Open Project file dialog box to allow you to navigate to the directory containing the synthesis project (.prj) files.

On opening a Synplify, Synplify Pro, Synplify Premier, or Certify project file, the data for that project is loaded into the Identify instrumentor. Loading a project retrieves all of the information stored in the project (user settings and file histories are automatically saved). The Identify instrumentor automatically compiles the files. When compilation is complete, the instrumentation window displays the design hierarchy and the HDL file content with all the potential instrumentation marked and available for selection.

A list of the four, most recent projects is maintained by the Identify instrumentor. Any of these projects can be opened directly from the File->Recent Projects menu.

Viewing a Compiled File

Once a design has been successfully compiled, the contents of individual HDL files can be opened in the currently active IICE. To open an HDL file, either double click on the filename or icon in the project view or right click on the filename or icon and select View file from the popup menu. The selected file is displayed in the currently active IICE display.



Technology Cell Definitions

The Identify instrumentor normally instantiates module definitions for technology-specific cells. When a technology cell is not previously defined, the Identify instrumentor creates a dummy (empty) black box for the undefined module to allow the compilation to complete.

To disable the generation of black boxes for uninstantiated module definitions, enter the following command in the Identify instrumentor console window:

```
device technologydefinitions 0
```

Note: The command accepts all Boolean arguments (0, off, false, 1, on, or true). Entering the command without an argument returns the current setting.

Instrumenting and Saving a Project

After setting up the IICE as described in [Chapter 4, IICE Configuration](#) and after selecting the instrumentation (selecting the signals for sampling, and setting breakpoints) as described in [Chapter 7, Identify Instrumentor](#), the project is instrumented and saved.

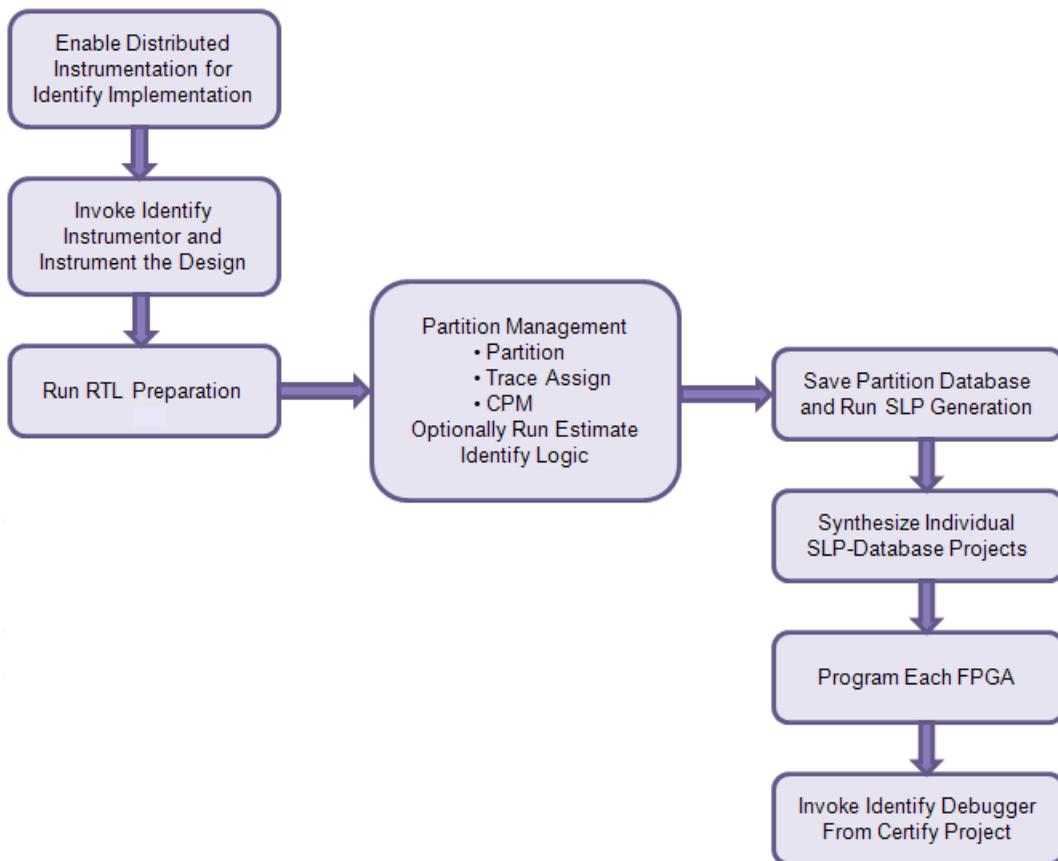


To save your instrumented project, select File->Save project instrumentation from the menu or click on the Save project's activated instrumentation icon.

Saving a project generates an *instrumentation design constraints* (.idc) file and adds compiler pragmas in the form of constraint files to the design RTL for the instrumented signals and breakpoints. This information is then used by the synthesis tool to incorporate the instrumentation logic (IICE and COMM blocks) into the synthesized netlist.

Projects with Distributed Instrumentation

Distributed instrumentation allows an instrumented design to be partitioned into multiple FPGAs and subsequently debugged through a single Identify Debugger session. Distributed instrumentation generates an *instrumentation design constraints* (idc) file that describes the instrumented signals and break points. Before running RTL preparation in Certify, the design is instrumented normally. During SLP generation in Certify, the Identify logic is added to the partitioned netlist. Following SLP generation, the SLP srs projects are synthesized and the individual FPGAs are programmed. The Identify Debugger is then invoked from the Certify project. The Debugger communicates with each of the individual FPGAs to configure the IICE and to download the sample data. For more information on distributed instrumentation, see the *Certify User Guide*.



Identify Debugger Projects

This section includes descriptions of the following tasks associated with the Identify debugger:

- [Opening an Identify Debugger Project](#)
- [Configuring an Identify Debugger Project](#)
- [Saving a Project](#)

Opening an Identify Debugger Project



To open a project in the Identify debugger, the project must have been created in the Identify instrumentor (only the Identify instrumentor can create a project) and synthesized in the synthesis tool.

To open an existing Identify debugger project from the Synopsys FPGA synthesis or Certify tool, highlight the Identify implementation and do one of the following:

- select Launch Identify Debugger from the popup menu
- click the Launch Identify Debugger icon in the menu bar
- select Run->Launch Identify Debugger (synthesis tool) or Tools->Launch Identify Debugger (Certify) from the main menu

You can also open an existing project from the Identify debugger interface by selecting File->Open Project from the menu or by clicking on the Open existing project icon. The name of the project file will have a prj extension. A list of the four, most recent projects is maintained by the Identify debugger. Any of these projects can be opened directly from the File->Recent Projects menu. On opening, the data for the project is loaded into the Identify debugger. Loading a project retrieves all of the information stored in a project and makes that project the current working project.

Configuring an Identify Debugger Project

To configure an Identify debugger project, click the project tab to reopen the project window (reopening the project window shows the instrumentation and communication settings).

Reviewing the Instrumentation Settings

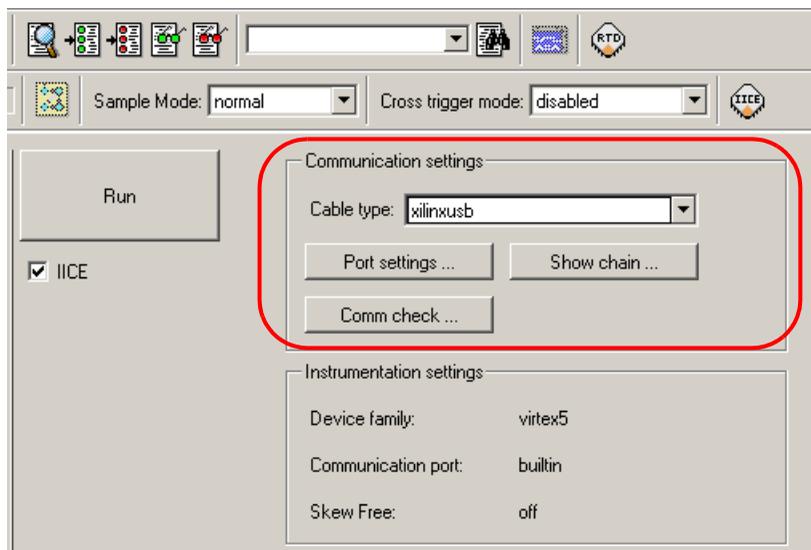
The Instrumentation settings are displayed in the Instrumentation settings section of the project window. Because these configuration settings are inherited from the Identify instrumentor and used to construct the IICE, you cannot change these settings in the Identify debugger.

Changing the Communication Settings

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

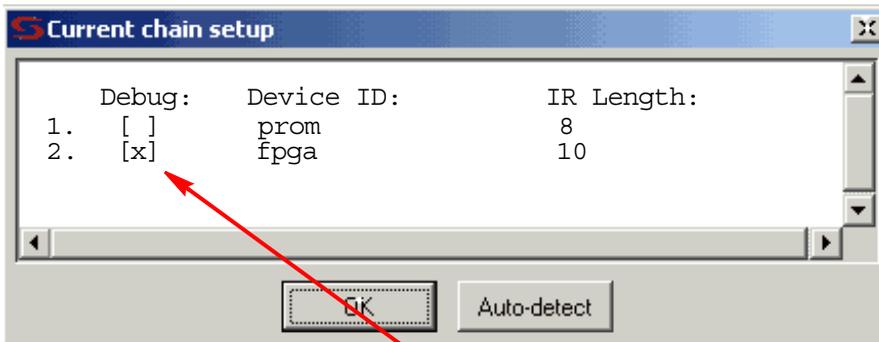
There is a large list of possible cable type settings including: byteblaster, xilinx-auto, xilinxusb, xilinxparallel, Microsemi_BuiltinJTAG, JTAGTech3710, Altera_BuiltinJTAG, and demo. A umrbus setting is also available to setup UMRBus communications between the host and a HAPS board (see [UMRBus Communications Interface, on page 203](#)). 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 a Project

Saving a project in the Identify debugger saves the following additional project information to the project file:

- IICE settings
- Instrumentations and activations



To save a project in the Identify debugger, click the Save current activations icon or select File->Save activations from the menu.

CHAPTER 4

IICE Configuration

An important part of the configuration of a project is setting the parameters for the Intelligent In-Circuit Emulator (IICE) in the Identify instrumentor. The IICE parameters determine the implementation of one or more IICE units and configure the units so that proper communication can be established with the Identify debugger. The IICE parameters common to all IICE units are set in the Identify instrumentor project window and apply to all IICE units defined for the implementation; the IICE parameters unique to each IICE definition in a multi-IICE configuration are interactively set on one of two IICE Configuration dialog box tabs:

- **IICE Sampler Tab** – includes sample depth selection and clock specification and also defines the external memory configuration for the HAPS Deep Trace Debug feature.
- **IICE Controller Tab** – includes complex counter trigger width specification, selection of state machine triggering, and import/export of the IICE trigger signal.

This chapter describes how to configure one or more IICE units.

Multiple IICE Units

Multiple IICE units allow triggering and sampling of signals from different clock domains within a design. Each IICE unit is independent and can have unique IICE parameter settings including sample depth, sampling/triggering options, and sample clock and clock edge. During the debugging phase, individual or multiple IICE units can be armed.

Adding an IICE Unit

A New IICE button is included in the Identify instrumentor project window to allow an additional IICE to be defined for the current instrumentation. When you click the New IICE button, the project window is reduced to a tab and a new instrumentation window is opened with the HDL source code redisplayed without any signals instrumented.

Note: When you create a new IICE from the GUI, the name of the IICE unit is formed by adding an `_n` suffix to IICE (for example, IICE_0, IICE_1, etc.). If you create a new IICE from the command line using the Identify instrumentor `ice new` command, you can optionally include a name for the IICE.

Right-clicking on the IICE tab and selecting Configure IICE from the popup menu brings up the Configure IICE dialog box which allows you to define the parameters unique to the selected IICE (see [Individual IICE Parameters, on page 38](#)).

Note: Communication port selection, which is common to all IICE units for an instrumentation, is defined in the Identify instrumentor project window (see [Common IICE Parameters, on page 35](#)).

Deleting an IICE Unit

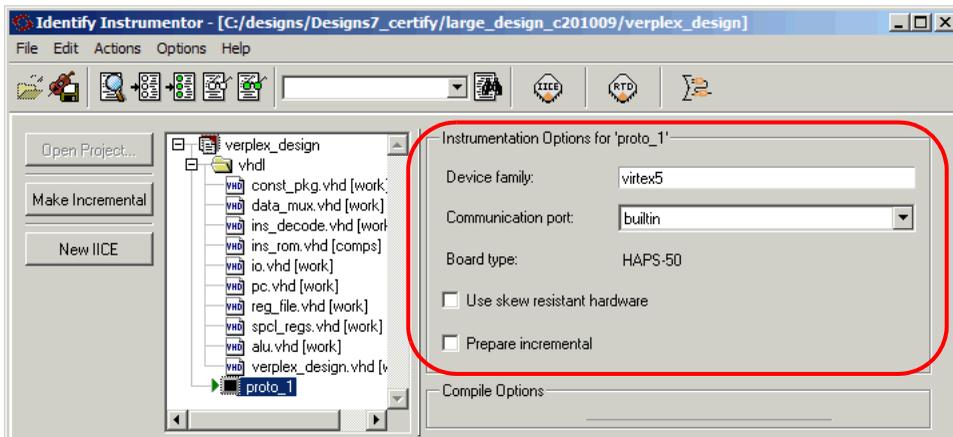
To delete an IICE unit from your Identify instrumentor project, right-click on the tab of the IICE to be deleted and select Delete IICE from the popup menu.

Common IICE Parameters

The IICE parameters common to all IICE units defined for an instrumentation include:

- the IICE device family as defined by the synthesis tool
- the communication port
- if optional skew-resistant hardware is to be used
- if incremental updating is enabled (Xilinx technologies only)

These parameters are set/displayed in the project window for the currently-active instrumentation. All IICE units in a multi-IICE configuration share these same parameter values.



Device Family

The device family selected in the synthesis tool is reported in the Device family field (the field is read-only) and cannot be changed.

Note: If the device family specified in the synthesis tool is not supported by the Identify tool set, an error message is issued and you are prompted to exit the Identify instrumentor.

Communication Port

The Communication port parameter specifies the type of connection to be used to communicate with the on-chip IICE. The connection types are:

- **builtin** – indicates that the IICE is connected to the JTAG Tap controller available on the target device.
- **soft** – indicates that the Synopsys Tap controller is to be used. The Synopsys FPGA Tap controller is more costly in terms of resources because it is implemented in user logic and requires four user I/O pins to connect to the communication cable.
- **umrbus** – indicates that the IICE is connected to the target device on a HAPS-60 or HAPS-70 series system through the UMRBus.

See [Chapter 11, *Connecting to the Target System*](#), for a description of the communication interface.

Board Type

The Board Type read-only field is only present when one of the supported Synopsys HAPS device technologies is selected in the synthesis tool and indicates the targeted HAPS board type.

Use Skew-Resistant Hardware

The Use skew-resistant hardware check box, when checked, incorporates skew-resistant master/slave hardware to allow the instrumentation logic to operate without requiring an additional global clock buffer resource for the JTAG clock.

When no global clock resources are available for the JTAG clock, this option causes the IICE to be built using skew-resistant hardware consisting of master-slave flip-flops on the JTAG chain which prevents clock skew from affecting the logic. Enabling this option also causes the Identify instrumentor to NOT explicitly define the JTAG clock as requiring global clock resources.

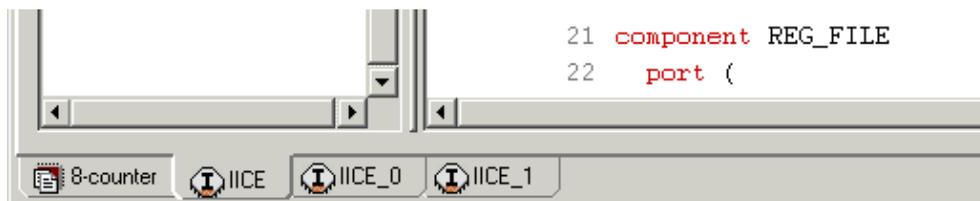
Prepare Incremental

The Prepare incremental check box is only enabled when one of the supported Xilinx Virtex technologies is selected in the synthesis tool. Checking this box enables the multi-pass, incremental flow feature available for specific Xilinx technologies (see [Chapter 9, *Incremental Flow*](#)).

Individual IICE Parameters



The individual parameters for each IICE are defined on two tabs of the Configure IICE dialog box. To display this dialog box, first select the active IICE by clicking the appropriate IICE tab in the Identify instrumentor project window and then select Actions->Configure IICE from the menu or click on the Edit IICE settings icon. You can also right-click directly on the IICE tab and select Configure IICE from the popup menu.



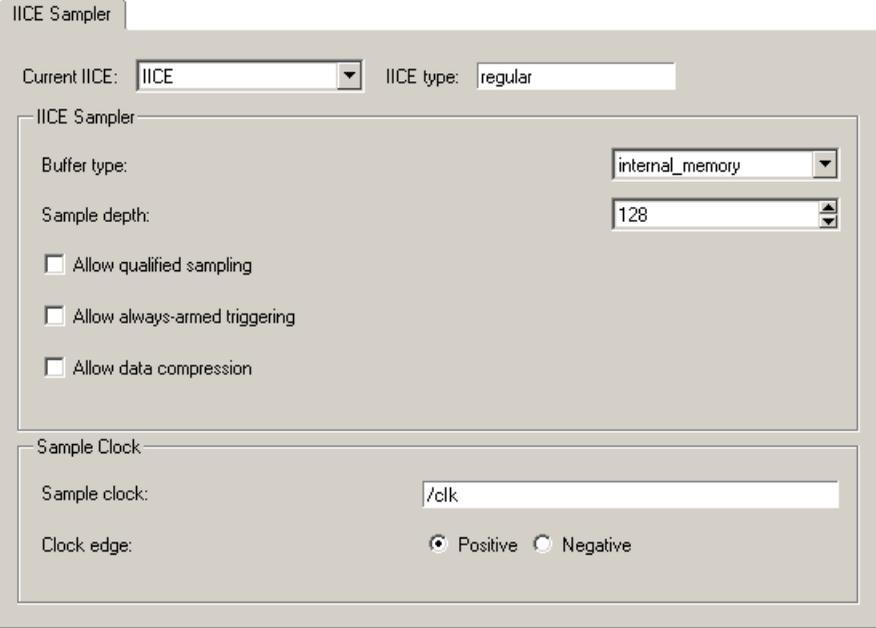
IICE tabs

IICE Sampler Tab

The IICE Sampler tab defines:

- IICE unit for multi-IICE configurations
- IICE type (regular or real-time debugging)
- Buffer type
- Sample depth
- Sampling/triggering options
- Data compression use
- Sample clock and clock edge

Note: The IICE Sampler tab is redefined when the Buffer type is set to hapssram.



The screenshot shows the IICE Sampler configuration window. It includes a 'Current IICE' dropdown menu set to 'IICE' and an 'IICE type' text field set to 'regular'. The 'IICE Sampler' section contains a 'Buffer type' dropdown menu set to 'internal_memory', a 'Sample depth' spin box set to '128', and three unchecked checkboxes: 'Allow qualified sampling', 'Allow always-armed triggering', and 'Allow data compression'. The 'Sample Clock' section contains a 'Sample clock' text field set to '/clk' and 'Clock edge' radio buttons with 'Positive' selected.

Current IICE

The Current IICE field identifies the target IICE when there are multiple IICE units defined within an implementation. The IICE is selected from the drop-down menu.

IICE type

The IICE type parameter is a read-only field that specifies the type of IICE unit currently selected – regular (the default) or rtd (real-time debugging). The IICE type is set from the project view in the user interface when a new IICE is defined or by an `iice sampler` Tcl command with a `-type` option. For information on the real-time debugging feature, see [Real-time Debugging, on page 86](#).

Buffer Type

The Buffer type parameter specifies the type of RAM to be used to capture the on-chip signal data. The default value is `internal_memory`; the `hapssram` setting configures the IICE to additionally use external HAPSRAM (for more information, see [Chapter 5, HAPS Deep Trace Debug](#)).

Sample Depth

The Sample depth parameter specifies the amount of data captured for each sampled signal. Sample depth is limited by the capacity of the FPGAs implementing the design, but must be at least 8 due to the pipelined architecture of the IICE.

Sample depth can be maximized by taking into account the amount of RAM available on the FPGA. As an example, if only a small amount of block RAM is used in the design, then a large amount of signal data can be captured into block RAM. If most of the block RAM is used for the design, then only a small amount is available to be used for signal data. In this case, it may be more advantageous to use logic RAM.

Allow Qualified Sampling

The Allow qualified sampling check box, when checked, causes the Identify instrumentor to build an IICE block that is capable of performing qualified sampling. When qualified sampling is enabled, one data value is sampled each time the trigger condition is true. With qualified sampling, you can follow the operation of the design over a longer period of time (for example, you can observe the addresses in a number of bus cycles by sampling only one value for each bus cycle instead of a full trace). Using qualified sampling includes a minimal area and clock-speed penalty.

Allow Always-Armed Triggering

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

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

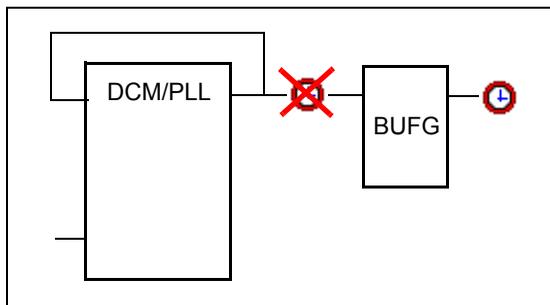
Allow data compression

The Allow data compression check box, when checked, adds compression logic to the IICE to support sample data compression in the Identify debugger (see [Sampled Data Compression, on page 114](#)). When unchecked (the default), compression logic is excluded from the IICE, and data compression in the Identify debugger is unavailable. Note that there is a logic data overhead associated with data compression and that the check box should be left unchecked when sample data compression is not to be used.

Sample Clock

The Sample clock parameter determines when signal data is captured by the IICE. The sample clock can be any signal in the design that is a single-bit scalar type. Enter the complete hierarchical path of the signal as the parameter value.

Care must be taken when selecting a sample clock because signals are sampled on an edge of the clock. For the sample values to be valid, the signals being sampled must be stable when the specified edge of the sample clock occurs. Usually, the sample clock is either the same clock that the sampled signals are synchronous with or a multiple of that clock. The sample clock must use a scalar, global clock resource of the chip and should be the highest clock frequency available in the design. The source of the clock must be the output from a BUFG/IBUFG device.



You can also select the sample clock from the instrumentation window by right-clicking on the watchpoint icon in the source code display and selecting Sample Clock from the popup menu. The icon for the selected (single-bit) signal changes to a clock face as shown in the following figure.

```
54
55     always @(posedge  Clk or negedge  Clr)
56         begin
57             if ( Clr == 1'b0)
58                  current_state = s_RESET; /* 4'b0000 */
59             else begin
```

Sample Clock Icon

Note: You must set the other individual IICE parameters from the Configure IICE dialog box including the sample clock edge.

Clock Edge

The Clock edge radio buttons determine if samples are taken on the rising (positive) or falling (negative) edge of the sample clock. The default is the positive edge.

IICE Controller Tab

The IICE Controller tab selects the IICE controller's triggering mode. All of these instrumentation choices have a corresponding effect on the area cost of the Identify IICE.

The screenshot shows the IICE Controller configuration window. At the top, there is a tab labeled "IICE Controller". Below the tab, there are two fields: "Current IICE:" with a dropdown menu showing "IICE_1" and "IICE type:" with a text input field containing "regular".

The main configuration area is divided into two sections:

- IICE Controller:** This section contains three radio button options for triggering modes:
 - Simple triggering
 - Complex counter triggering
 - State Machine triggeringBelow these options are four spinners:
 - Width: 16
 - Trigger states: 4
 - Trigger conditions: 4
 - Counter width: 16
- IICE Options:** This section contains:
 - Import external trigger signals: 0 (dropdown menu)
 - Export IICE trigger signal
 - Allow cross-triggering in IICE

Current IICE

The Current IICE field is used to identify the target IICE when there are multiple IICE units defined within an implementation. The IICE is selected from the drop-down menu.

IICE type

The IICE type parameter is a read-only field that specifies the type of IICE unit currently selected – regular (the default) or rtd (real-time debugging). The IICE type is set from the project view in the user interface when a new IICE is defined or by an `iice sampler Tcl` command with a `-type` option. For information on the real-time debugging feature, see [Real-time Debugging, on page 86](#).

Simple Triggering

Simple triggering allows you to combine breakpoints and watchpoints to create a trigger condition for capturing the sample data.

Complex-Counter Triggering

Complex-counter triggering augments the simple triggering by instrumenting a variable-width counter that can be used to create a more complex trigger function. Use the width setting to control the desired width of the counter.

State-Machine Triggering

State-machine triggering allows you to pre-instrument a variable-sized state machine that can be used to specify an ultimately flexible trigger condition. Use `Trigger states` to customize how many states are available in the state machine. Use `Trigger condition` to control how many independent trigger conditions can be defined in the state machine. For more information on state-machine triggering, see [State Machine Triggering, on page 156](#).

Import External Trigger Signals

External triggering allows the trigger from an external source to be imported and configured as a trigger condition for the active IICE. The external source can be a second IICE located on a different device or external logic on the board rather than the result of an Identify instrumentation. The imported trigger signal includes the same triggering capabilities as the internal trigger sources used with state machines. The adjacent field selects the number of external trigger sources with 0, the default, disabling recognition of any external trigger. Selecting one or more external triggers automatically enables state-machine triggering.

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

Export IICE Trigger Signal

The Export IICE trigger signal check box, when checked, causes the master trigger signal of the IICE hardware to be exported to the top-level of the instrumented design.

Allow cross-triggering in IICE

The Allow cross-triggering in IICE check box, when checked, allows this IICE unit to accept a cross-trigger from another IICE unit. For more information on cross-triggering, see [Cross Triggering, on page 123](#).

CHAPTER 5

HAPS Deep Trace Debug

The HAPS Deep Trace Debug feature uses external SRAM as sample memory which allows Identify to use both the FPGA internal block RAM as well as the external HAPSRAM. With the added external memory, a much deeper, signal-trace buffer is available.

Identify enables the use of HAPS SRAM daughter-boards as an external buffering memory for designs being targeted for the HAPS platform in a single-FPGA debugging mode.

This chapter provides detailed descriptions of the feature and its use. A step by step tutorial using an example design is also available. The HAPS Deep Trace Debug feature is only available with Synopsys HAPS-50, HAPS-60, and HAPS-70 series prototyping boards using a HAPS SRAM_1x1_HTII daughter board (HAPS-50 and HAPS-60) or a HAPS SRAM_HT3 (HAPS-70) daughter board.

External Memory Instrumentation and Configuration Steps

With the HAPS Deep Trace Debug mode, the Synplify/Identify flow remains unchanged. The only difference is in the configuration of a HAPS SRAM memory as the external sample buffer using IICE parameters.

The HAPS Deep Trace Debug feature includes the capability to control the configured sample depth. This depth can be dynamically varied using the Sample depth option available on the IICE Sampler tab. The depth can be varied between the minimum depth to the maximum configured depth, but cannot exceed the maximum configured depth.

The following figure shows the HAPS Settings dialog box for the HAPS Deep Trace Debug mode when the buffer type on the IICE Sampler tab is set to hapssram.

Note: You can also select the hapssram buffer type using the iice sampler hapssram Tcl command.

The individual HAPS Deep Trace Debug parameters are described in the ensuing table.

HAPS Settings

Current IICE: IICE IICE type: regular

HAPS Settings

Board: HAPS-61

SRAM locations (HapsTrak connector): 1, 2, 3

SRAM stack: 1

SRAM module type: HapsTrakII GS8320Z36GT-200 2

SRAM clock type: Internal External

SRAM clock frequency (MHz): 100.000000

Parameter	Description
Board	The HAPS Deep Trace Debug feature is available only with specific HAPS boards. The instrumentor allows selection of hapssram buffer type only when the device used on the HAPS board is set in the implementation options. For example, if a Synopsys HAPS-50 device is specified in Synplify Pro, the Identify Instrumentor enables HAPS-50 series board selection from the drop-down menu. Similarly, if a Synopsys HAPS-60 or HAPS-70 device is specified for the project, HAPS-60 or HAPS-70 series board selection is enabled from the drop-down menu.
SRAM locations	The buffering of the instrumented samples is performed using an external SRAM daughter card connected to any or all the HapsTrak II or HapsTrak 3 connectors of a single FPGA. The selection of the connectors where the daughter cards are physically connected is done by selecting one or more HapsTrak locations (locations 1 through 6 for HAPS-50/HAPS-60 or a matrix location for HAPS-70) of the daughter cards for the FPGA under debug.
SRM stack	The depth of SRAM on each daughter card is 4M locations of 72-bit words for HTII SRAM cards and 8M locations of 90-bit words for HT3 SRAM cards. To increase the external SRAM memory depth beyond 4M x 72 or 8M x 90, the daughter cards can be stacked. For the HTII type SRAM, 1, 2, or 4 daughter cards can only be stacked for the selected SRAM locations and for HT3 type SRAM cards, 1, 2, 3, or 4 cards can be stacked. The stack number specified applies to all connector locations specified by SRAM locations.
SRAM module type	The HapsTrak SRAM daughter card type is selected using this drop-down option.
SRAM clock type	The clock to the SRAM daughter card can be either fed from the clock used within the design (Internal) or from an external clock source present on the HAPS board (see SRAM Clocks, on page 50).
SRAM clock frequency	Specifies the frequency of the clock source to the SRAM. The supported SRAM operating frequency ranges for various HAPS board and SRAM card stacks using the FPGA internal PLL output as the SRAM clock are listed in SRAM Clocks, on page 50 .

SRAM Clocks

When the clock source is internal to the design:

- select the Internal radio button
- specify the clock signal to be used as the source clock in the adjacent text box.

Any clock signal within the design at any hierarchy level can be instrumented as the SRAM clock.

When the clock source is external:

- select the External radio button
- Specify a suitable pin-lock constraint in the synthesis constraint file for the `deepbuf_sclk_iceName_p` and `deepbuf_sclk_iceName_n` ports (these ports are created automatically in the instrumented design)
- provide the external clock source to this FPGA port

Because of performance considerations, users are recommended to use FPGA internal PLL output as the source of the SRAM clock.

Specify the frequency of the clock source to the SRAM. The supported SRAM operating frequency ranges for various HAPS boards and SRAM card stacks using the internal PLL output as the SRAM clock are given in the following table:

SRAM	H-SRAM-1x1-HTII (2.5V)		HT3-SRAM (1.8V)
	HAPS-50 Series	HAPS-60 Series	HAPS-70 Series
1 SRAM stack	88 to 140 MHz	96 to 155 MHz	150 MHz
2 SRAM card stack	88 to 100 MHz	92 to 116 MHz	100 MHz
3 SRAM card stack	Not Supported	Not Supported	Not Supported
4 SRAM card stack	75 MHz	80to 110 MHz	Not Supported

Sample Depth Calculation

For a given, user-defined external memory configuration setting, the maximum allowed depth can be calculated based on the formula described below.

- Number of HapsTrak slots used: N_{slot}
- Number of SRAM cards stacked: $NSRAM$
- Number of 72-bit or 90-bit words per SRAM card: N_{word} (4194304 for HapsTrak II; 8388608 for HapsTrak 3)
- Number of signals to be sampled (instrumented): N_{signal}

$$K_{sample} \leq \left\lceil \frac{N_{word} N_{SRAM}}{\frac{N_{signal} + 6}{72 N_{slot}}} \right\rceil$$

HapsTrak II

$$K_{sample} \leq \left\lceil \frac{N_{word} N_{SRAM}}{\frac{N_{signal} + 6}{90 N_{slot}}} \right\rceil$$

HapsTrak 3

For example, if $N_{slot} = 1$, $NSRAM = 1$, $N_{word} = 4M$ (4194304) and $N_{signal} = 1900$, the maximum sampling depth for K samples for a HapsTrak II SRAM card is 155344.

Sample Clock Calculation

For a given set of user-defined external memory configuration settings, the sample clock frequency can be estimated using the formula described below.

$$f_{sampling} \leq \left\lceil \frac{f_{SRAM}}{\frac{N_{signal} + 6}{72 N_{slot}}} \right\rceil + 2$$

HapsTrak II

$$f_{sampling} \leq \left\lceil \frac{f_{SRAM}}{\frac{N_{signal} + 6}{90 N_{slot}}} \right\rceil + 2$$

HapsTrak 3

In the above expressions:

- Number of HapsTrak slots used: N_{slot}
- Number of signals to be sampled (instrumented): N_{signal}

- SRAM bus frequency: f_{SRAM}

For example, if $f_{\text{SRAM}} = 100\text{MHz}$, $N_{\text{slot}} = 1$, and $N_{\text{signal}} = 1900$, the maximum sampling frequency for a HapsTrak II SRAM card is 3.44MHz.

Note: For every parameter/option that is set, as described above, an equivalent Tcl command is also available.

Hardware Configuration Verification

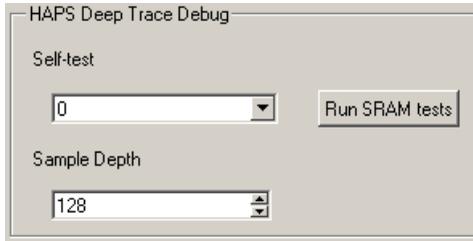
A self-test is available for verifying the deep trace debug hardware configuration. The self-test writes data patterns to the external memory and reads back the data pattern written to detect configuration errors, connectivity problems, and SRAM frequency mismatches.

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 Identify 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
```


CHAPTER 6

Support for Instrumenting HDL

The Identify tool set fully supports the synthesizable subset of both Verilog and VHDL design languages. Designs that contain a mixture of VHDL and Verilog can be debugged – the Identify software reads in your design files in either language.

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

The instrumentation limitations are usually related to language features. These limitations are described in this chapter.

- [VHDL Instrumentation Limitations, on page 56](#)
- [Verilog Instrumentation Limitations, on page 58](#)
- [SystemVerilog Instrumentation Limitations, on page 61](#)

VHDL Instrumentation Limitations

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

Design Hierarchy

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

Subprograms

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

Loops

Breakpoints within loops cannot be instrumented.

Generics

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

Transient Variables

Transient variables defined locally in VHDL processes cannot be instrumented.

Breakpoints and Flip-flop Inferencing

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

For flip-flops with asynchronous reset:

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

For flip-flops with synchronous reset or without reset:

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

Or:

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

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

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

Verilog Instrumentation Limitations

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

Subprograms

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

Loops

Breakpoints within loops cannot be instrumented.

Parameters

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

Locally Declared Registers

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

Verilog Include Files

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

- If the keyword `module` or `endmodule`, or if the closing `'` of the module port list is located inside a multiply-included file, no constructs inside the corresponding module or its submodules can be instrumented.
- If significant portions of the body of an `always` block are located inside a multiply-included file, no breakpoints inside the corresponding `always` block can be instrumented.

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

As an example, consider the following three files:

adder.v File

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

adder8.v File

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

adder16.v File

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

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

Macro Definitions

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

Always Blocks

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

For flip-flops with asynchronous reset:

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

For flip-flops with synchronous reset or without reset:

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

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

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

SystemVerilog Instrumentation Limitations

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

Typedefs

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

Struct Construct

A structure data type represents collections of data types. These data types can be either standard data types (such as int, logic, or bit) or, they can be user-defined types (using SystemVerilog typedef). Signals of type structure can only be sampled and cannot be used for triggering; individual elements of a structure cannot be instrumented, and it is only possible to instrument (sample only) an entire structure. The following code segment illustrates these limitations:

```
module lddt_P_Struc_top (
  input sig sig_clk, sig sig_rst,
  .
  .
  .
  output struct packed {
    logic_nibble up_nibble;
    logic_nibble lo_nibble;
  } sig_oport_P_Struc_data
);
```

Cannot be instrumented
(no sampling and
no triggering)

Instrumentable only for
sampling; no triggering

In the above code segment, port signal `sig_oport_P_Struc_data` is a packed structure consisting of two elements (`up_nibble` and `lo_nibble`) which are of a user-defined datatype. As elements of a structure, these elements cannot be instrumented. The signal `sig_oport_P_Struc_data` can be instrumented for sampling, but cannot be used for triggering (setting a watch point on the signal is not allowed). If this signal is instrumented for sample and trigger, the Identify instrumentor allows only sampling and ignores triggering.

Union Construct

A union is a collection of different data types similar to a structure with the exception that members of the union share the same memory location. Unions are not supported for instrumentation and it is not possible to select a union datatype signal for either sampling or triggering. The following code segment illustrates this limitation:

```
module lddt_P_Union_top (
  input sig sig_clk, sig sig_rst,
  input union packed {
    type_P_Struc_datapkt P_Struc_data;
    logic [7:0] [7:0] P_Array_data;
  } sig_iport_P_Union_data,
  output union packed {
    type_P_Struc_datapkt P_Struc_data;
    logic [7:0] [7:0] P_Array_data;
  } sig_oport_P_Union_data
);
```

Cannot be instrumented
(no sampling and
no triggering)

In the above code, port signals `sig_iport_P_Union_data` and `sig_oport_P_Union_data` are of type union. Code that includes union constructs compiles successfully without error in the synthesis tool, but is disabled for instrumentation in the Identify instrumentor.

Arrays

Arrays having more than one dimension cannot be instrumented for triggering. The following code segment illustrates these limitation:

```
module test (clock,waddress,waddress1,
  raddress,we,data,data1,q,q1);
input clock;
input [8:0] waddress,waddress1,raddress;
input we;
input [255:0] [31:0] data,data1;
output reg [31:0] q,q1;
```

Instrumented for sampling only

In the above code segment, `waddress`, `waddress1`, and `raddress` are one dimensional packed arrays. These signals can be instrumented for both sampling and triggering. Signals `data` and `data1` are 2-dimensional arrays. These signals can only be instrumented for sampling, and triggering is not allowed.

Partial instrumentation of multi-dimensional and multi-dimensional arrays of struct is not permitted, and instrumentation of multi-dimensional arrays of packed unions is not supported.

Interface

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

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

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

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

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

Port Connections for Interfaces and Variables

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

Packages

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

Concatenation Syntax

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

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

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

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

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

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

CHAPTER 7

Identify Instrumentor

The Identify instrumentor performs the following functions:

- defines the instrumentation for the user's HDL design
- creates the instrumented HDL design
- creates the associated IICE
- creates the design database

The remainder of this chapter describes:

- [Identify Instrumentor Windows](#)
- [Commands and Procedures](#)

Identify Instrumentor Windows

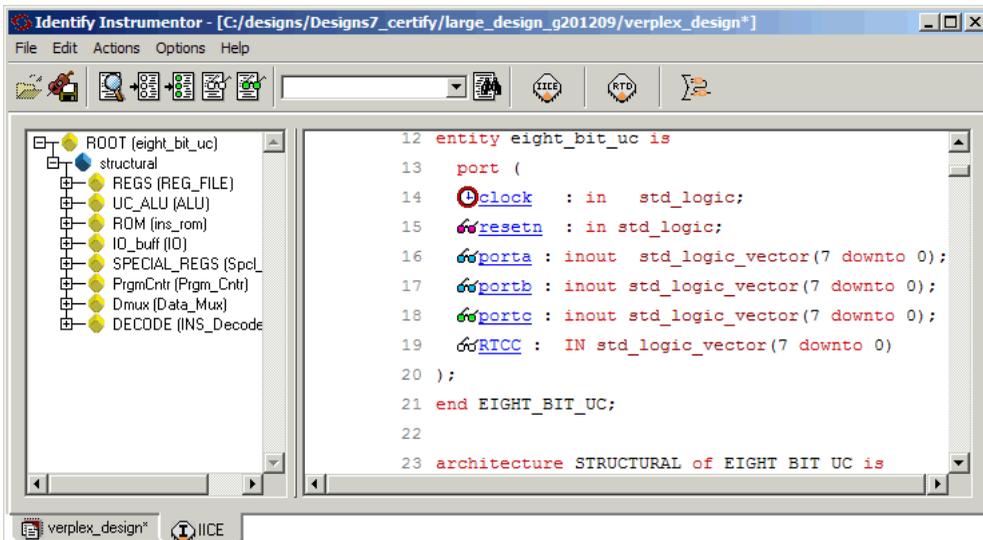
The Graphical User Interface (GUI) for the Identify instrumentor is divided into the following three major areas:

- [Instrumentation Window](#)
- [Project Window](#)
- [Console Window](#)

In this section, each of these areas and their uses is described. The following discussions assume that a project (with an HDL design) has been loaded into the Identify instrumentor.

Instrumentation Window

The instrumentation window includes a hierarchy browser on the left and the source-code display on the right. The window is displayed when you open a synthesis project (.prj) file in the Identify instrumentor by either launching the Identify instrumentor from a synthesis project or explicitly loading a synthesis project into the Identify instrumentor.



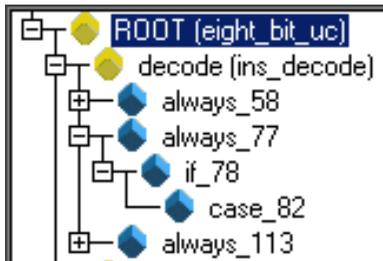
Hierarchy Browser

Source-Code Display

Hierarchy Browser

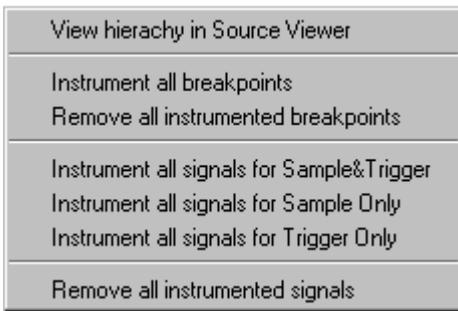
The hierarchy browser on the left shows a graphical representation of the design's hierarchy. At the top of the browser is the ROOT node. The ROOT node represents the top-level entity or module of your design. For VHDL designs, the first level below the ROOT is the architecture of the top-level entity. 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 visible in the adjacent source code display.

A popup menu is available in the hierarchy browser to set or clear breakpoints or watchpoints at any level of the hierarchy. Positioning the cursor over an element and clicking the right mouse button displays the following menu.



The selected operation is applied to all breakpoints or signal watchpoints at the selected level of hierarchy.

Note: You cannot instrument signals when a sample clock is included in the defined group.

The popup menu functions can be duplicated in the console window using the Identify instrumentor hierarchy command in combination with either the Identify instrumentor breakpoints or signals command. A typical Identify instrumentor command sequence to instrument the signal set within a design would be:

```
signals add [hierarchy find -type signal /]
```

In the above sequence, the slash (/) indicates that all of the signals in the root hierarchy (entire design) are to be instrumented. If you specify a lower level of hierarchy following the slash, the command only applies to that hierarchical level. For more information on the Identify instrumentor breakpoints, signals, and hierarchy commands, see the *Reference Manual*.

Black-box modules are represented by a black icon, and their contents can not be instrumented. Also, certain modules cannot be instrumented (see [Chapter 6, Support for Instrumenting HDL](#), for a specific description). Modules that cannot be instrumented are displayed in a disabled state in a grey font.

Source Code Display

The HDL source code in the source code display is annotated with signals that can be probed and breakpoints that can be selected. Signals that can be selected for probing by the IICE are underlined, colored in blue, and have small watchpoint icons next to them. Source lines that can be selected as breakpoints have small circular icons in the left margin adjacent to the line number.

```

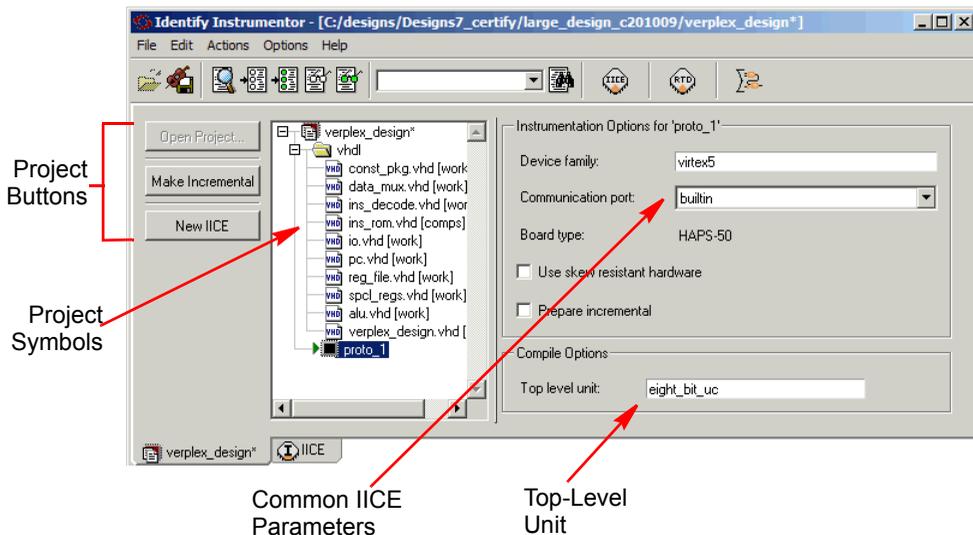
56  begin
57      if clk = '0' then
58          current_state <= s_RESET;
59      elsif clk'event and clk = '1' then
60          case current_state is
61              when s_RESET    => current_state <= s_ONE;
62              when s_ONE      => current_state <= s_TWO;

```

Project Window

The project window is displayed when you first start up the Identify instrumentor (to invoke the Identify instrumentor, see [Projects in the Identify Instrumentor, on page 24](#)).

The project window includes a set of project management buttons and a symbolic view of the project on the left and an area for setting the common IICE parameters on the right (the area below the common IICE parameters shows the top-level unit when the design was compiled).



The project window can be displayed at any time by clicking the project name tab that remains visible when an instrumentation window is displayed.

Console Window

The console window appears below the project or instrumentation window and displays a history of Identify instrumentor commands that have been executed, including those executed by menu selections and button clicks. The Identify instrumentor console window allows you to enter commands as well as to view the results of those commands. Command history recording is available through a transcript command (see the *Reference Manual*).

```
INFO: D:/Designs/8-bit-verilog/data_mux.v(1): compiling module data_mux
INFO: D:/Designs/8-bit-verilog/alu.v(1): compiling module alu
WARNING: D:/Designs/8-bit-verilog/alu.v(83): full_case directive is effective
: might cause synthesis - simulation differences
INFO: D:/Designs/8-bit-verilog/ins_rom.v(1): compiling module ins_rom
INFO: D:/Designs/8-bit-verilog/spcl_regs.v(1): compiling module spcl_regs
INFO: D:/Designs/8-bit-verilog/io.v(1): compiling module io
INFO: Current design is 'eight_bit_uc'
D:/tools/ident200_021R/bin$
```

When you save your instrumentation (click Save project's active implementation), the information at the bottom of the display reports the total number of instrumented signals to implement in each IICE. To display the resource estimates for the FPGA device, click the IICEs resource estimation icon.



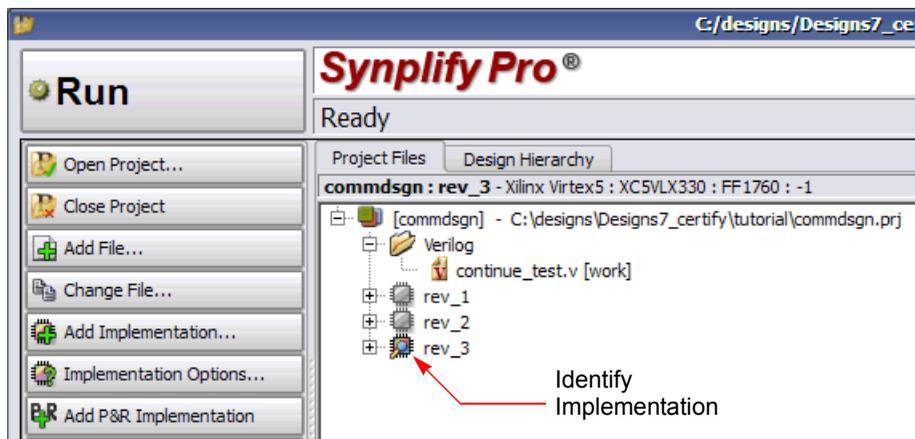
Commands and Procedures

The following sections describe basic Identify instrumentor commands and procedures.

Opening Projects

To launch the Identify instrumentor from the Synopsys synthesis tool (Synplify, Synplify Pro, Synplify Premier, or Certify) graphical interface:

1. Right click on the implementation and select New Identify Implementation from the popup menu.
2. Set/verify any technology, device mapping, or other pertinent options (see the implementation option descriptions in the *Synopsys FPGA Synthesis* or *Certify User Guide*) and click OK. A new, Identify implementation is added to the synthesis tool project view.



3. Either right click on the Identify implementation and select Launch Identify Instrumentor from the popup menu or click the Launch Identify Instrumentor toolbar icon to launch the Identify instrumentor.

Launching the Identify instrumentor from a Synopsys synthesis tool:

- Brings up the Identify instrumentor graphical interface.

- Extracts the list of design files, their work directories, and the device family from the `prj` file.
- Automatically compiles the design files using the synthesis tool compiler and displays the design hierarchy and the HDL file content with all the potential instrumentation marked and available for selection.

When importing a synthesis tool project into the Identify instrumentor, the working directory is automatically set from the corresponding project file. See [Chapter 3, Project Handling](#), for details on setting up and managing projects.

Executing Script Files

A script file is a file that contains Identify instrumentor Tcl commands. A script file is a convenient way to capture a command sequence you would like to repeat. To execute a script file, select the File->Execute Script menu selection in the Identify instrumentor user interface and navigate to the location of your script file or use the source command (see Chapter 4, *Alphabetical Command Reference*, in the *Reference Manual*).

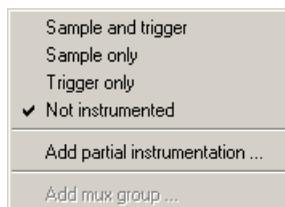
Selecting Signals for Data Sampling



To select a signal to be sampled, simply click on the watchpoint icon next to the signal name in the Identify instrumentor instrumentation window; a popup menu appears that allows the signal to be selected for sampling, triggering, or both.



Scalar Signal Popup



Bus Signal Popup

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

Qualified clock signals can be specified as the Sample Clock (see [Sample Clock, on page 41](#)) and bus segments can be individually specified (see [Instrumenting Buses, on page 75](#)). In addition, signals specified as Sample and trigger or Sample only can be included in mux groups (see [Multiplexed Groups, on page 79](#)).

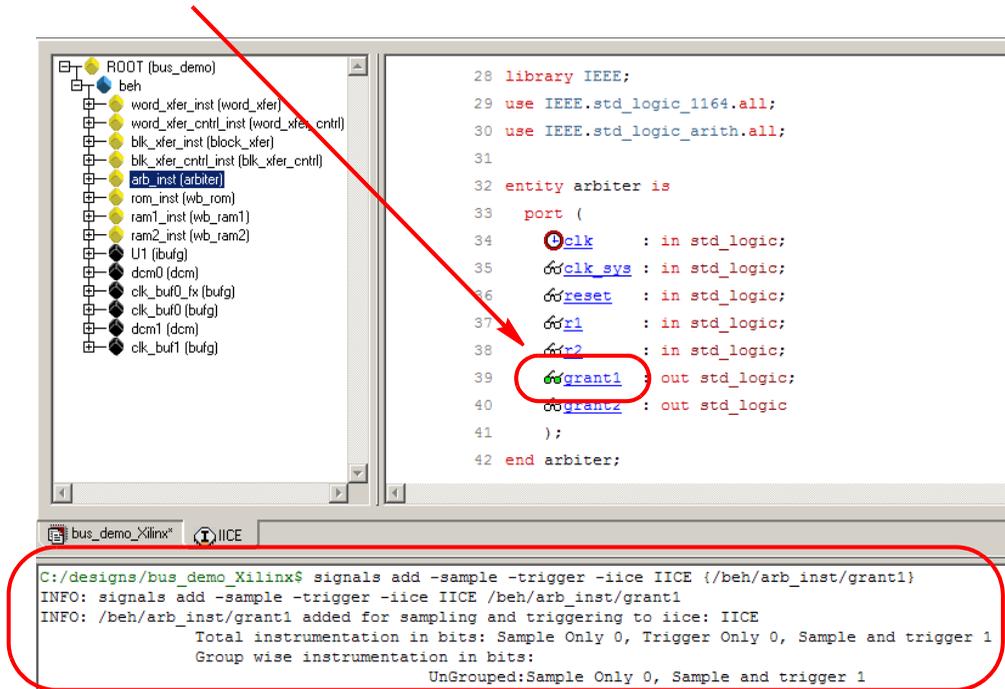
```

27  port (
28      ⚙️clk      : in std_logic;
29      ⚙️reset    : in std_logic;
30      ⚙️req1     : in std_logic;
31      ⚙️req2     : in std_logic;
32      ⚙️grant1   : out std_logic;
33      ⚙️grant2   : out std_logic;
34  );
35  end arbiter;
36
37  architecture beh of arbiter is
38      type states is ( st_idle1, st_idle2, st_grant1, st_grant2
39      signal ⚙️curr_state, ⚙️next_state: states;

```

If the icon is clear (unfilled), the signal is disabled for sampling (not instrumented), and if the icon is red, the signal is enabled for triggering only. If the icon is blue, the signal is enabled for sampling only, and if the icon is green, the signal is enabled for both sampling and triggering. In the example below, notice that when signal “grant1” is enabled, the console window displays the text command that implements the selection and the results of executing the command.

Signal "grant1" selected



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

Note: You can use Find to recursively search for signals and then instrument selected signals directly from the Find dialog box (see [Searching for Design Objects, on page 94](#)).

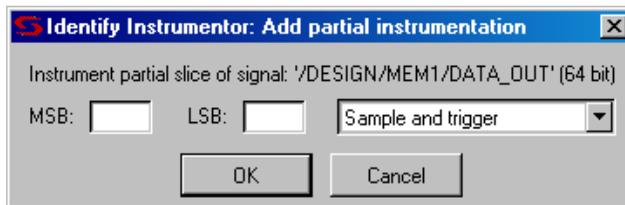
Instrumenting Buses

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

Instrumenting a Partial Bus

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

1. Place the cursor over a bus that has not been fully instrumented and select Add partial instrumentation to display the following dialog box.



2. In the dialog box, enter the most- and least-significant bits in the MSB and LSB fields. Note that the bit range specified is contiguous; to instrument non-contiguous bit ranges, see the section, [Instrumenting Non-Contiguous Bits or Bit Ranges](#), on page 76.

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

3. Select the type of instrumentation for the specified bit range from the drop-down menu and click OK.

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

```

277 input MEM RD, MEM WR:
278 input [63:0] DATA IN:
279 output [63:0] P DATA OUT:

```

Instrumenting Single Bits of a Bus

To instrument a single bit of a bus, enter the bit value in the MSB field of the Add partial bus dialog box, leave the LSB field blank, and select the instrumentation type from the drop-down menu as previously described.

Instrumenting Non-Contiguous Bits or Bit Ranges

To instrument non-contiguous bits or bit ranges:

1. Instrument the first bit range or bit as described in one of the two previous sections.
2. Re-position the cursor over the bus, click the right mouse button, and again select Add partial instrumentation to redisplay the Add partial bus dialog box. Note that the previously instrumented bit or bit range is now displayed.



3. Specify the bit or bit range to be instrumented as previously described, select the type of instrumentation from the drop-down menu, and click OK. If the type of instrumentation is different from the existing instrumentation, the letter "P" will be yellow to indicate a mixture of instrumentation types.

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

Changing the Instrumentation Type

To change the instrumentation type of a partial bus:

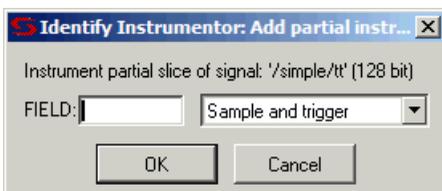
1. Position the cursor over the bus and click the right mouse button.
2. Highlight the bit range or bit to be changed and select the new instrumentation type from the adjacent menu.



Note: The above procedure is also used to remove the instrumentation from a bit or bit range by selecting Not instrumented from the menu.

Partial Instrumentation

Partial instrumentation allows fields within a record or a structure to be individually instrumented. Selecting a compatible signal for instrumentation, either in the RTL window or through the Find dialog box, enables the partial instrumentation feature and displays a dialog box where the field name and its type of instrumentation can be entered.



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

- Green indicates that all fields of the record are instrumented for sample and trigger
- Blue indicates that all fields of the record are instrumented for sample only

- Pink indicates that all fields of the record are instrumented for trigger only
- Yellow indicates that not all fields of the record are instrumented the same way

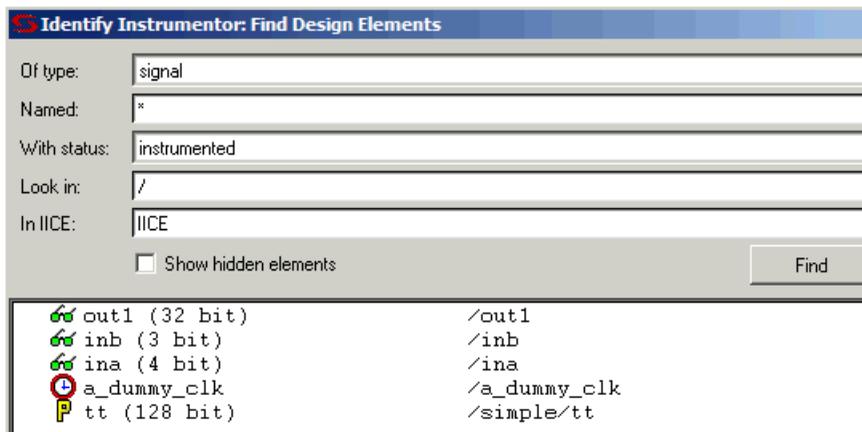
The figure below shows the partial instrumentation icon on signal tt. The yellow color indicates that the individual fields (tt.r2 and tt.c2) are assigned different types of instrumentation.

```

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

```

The Find dialog also uses the partial instrumentation icon to show the state of instrumentation on fields of partially instrumented records.



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

Multiplexed Groups

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

Only signals or buses that are instrumented as either Sample and trigger or Sample only can be added to a multiplexed group. To create multiplexed groups, right click on each individual instrumented signal or bus and select Add mux group from the popup menu.



In the Add mux group dialog box displayed, select a corresponding group by checking the group number and then click OK to assign to the signal or bus to that group. A signal can be included in more than one group by checking additional group numbers before clicking OK.



When assigning instrumented signals to groups:

- A maximum of eight groups can be defined; signals can be included in more than one group, but only one group can be active in the Identify debugger at any one time.
- Signals instrumented as Sample Clock or Trigger only cannot be included in multiplexed groups.
- Partial buses cannot be assigned to multiplexed groups.

- The signals group command can be used to assign groups from the console window (see [signals](#), on page 79 of the *Reference Manual*). Command options allow more than one instrumented signal to be assigned in a single operation and allow the resultant group assignments to be displayed.

For information on using multiplexed groups in the Identify debugger see [Selecting Multiplexed Instrumentation Sets](#), on page 109.

Sampling Signals in a Folded Hierarchy

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



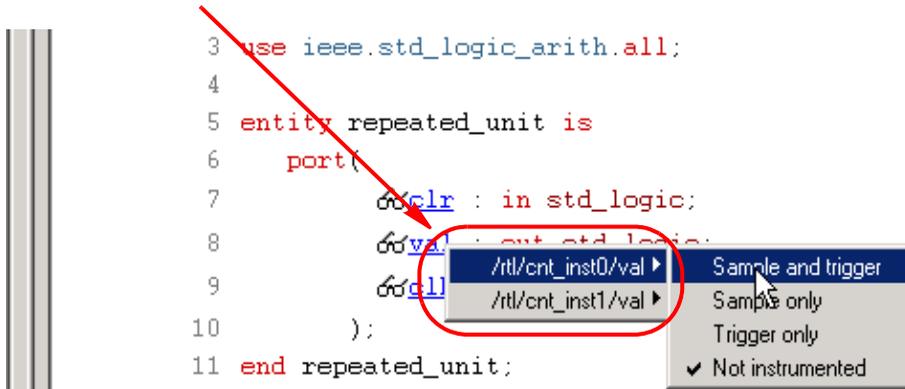
The choices are displayed in terms of an absolute signal path name originating at the top-level entity or module. The list of choices for a particular signal is accessed by clicking the watchpoint icon or corresponding signal.

The example below consists of a top-level entity called `two-level` and two instances of the `repeated_unit` entity. The source code of `repeated_unit` is displayed, and the list of instances of the `val` signal is displayed by clicking on the watchpoint icon or the signal name. Two instances of the signal `val` are available for sampling:

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

Either, or both, of these instances can be selected for sampling by selecting the signal instance and then sliding the cursor over to select the type of sampling to be instrumented for that signal instance.

The list of instrumentable instances of signal **val**



The color of the watchpoint icon is determined as follows:

- If no instances of the signal are selected, the watchpoint icon is clear.
- If some, but not all, instances of the signal are defined for sampling, the watchpoint icon is yellow.
- If all instances are defined for sampling, the color of the watchpoint icon is determined by the type of sampling specified (all instances sample only: blue, all instances trigger only: red, all instances sample and trigger: green, all instances in any combination: yellow).

Alternately, any of the instances of a folded signal can be selected or deselected at the Identify instrumentor console window prompt by using the absolute path name of the instance. For example,

```
signals add /rtl/cnt_inst1/val
```

See the *Reference Manual* for more information.

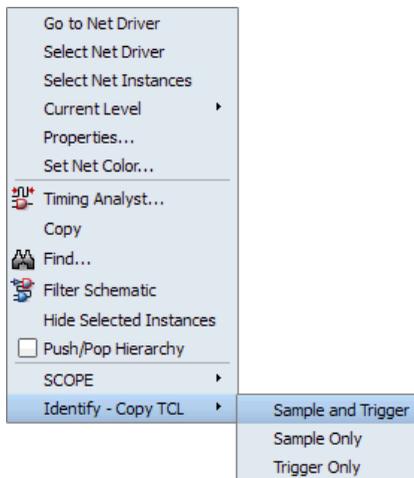
To disable an instance of a signal that is currently defined for sampling, click on the watchpoint icon or signal, select the instance from the list displayed, and select Not instrumented.

For related information on folded hierarchies in the Identify debugger, see [Activating/Deactivating Folded Instrumentation, on page 110](#) and [Displaying Data from Folded Signals, on page 119](#).

Instrumenting Signals Directly in the idc File

In addition to the methods described in the previous sections, signals can be instrumented directly within the `srs` file in the synthesis tool outside of the Identify instrumentor. This methodology facilitates updates to a previous instrumentation and also allows signals within a parameterized module, which were previously unavailable for instrumentation, to be successfully instrumented. This technique is referred to as “post-compile instrumentation.” To instrument a signal directly within the synthesis tool using this technique:

1. Compile the existing instrumented design in the synthesis tool.
2. Open the RTL view (`srs` file) in the synthesis tool.
3. Highlight the net of the signal to be instrumented.
4. With the net highlighted, click the right mouse button, select Identify - Copy TCL from the popup menu, and select the type of instrumentation to be applied.



5. Create a new or open an existing `identify.idc` file in the `designName/rev_n` directory and paste the signal string into the file. The following figure shows the `cntrl_ack_o_0_sqmuxa` signal (from the `block_xfer` block) on line 10 being pasted into the file as a sample-only signal.

```

1 iice new {IICE} -type regular
2 iice controller -iice {IICE} none
3 iice sampler -iice {IICE} -depth 128
4
5 signals add -iice {IICE} -silent -trigger {/SRS/blk_xfer_inst/cntrl_ack_o_0_sqmuxa}
6 signals add -iice {IICE} -silent -sample {/beh/arb_inst/reset}
7 signals add -iice {IICE} -silent -trigger -sample {/beh/arb_inst/beh/curr_state}\
8 {/beh/arb_inst/beh/next_state}
9 iice clock iice {IICE} edge positive {/clk}
10 signals add -sample {/SRS/blk_xfer_inst/cntrl_ack_o_0_sqmuxa}

```

Note: If you are creating a new identify.idc file, you must add the IICE definition on lines 1, 2, and 3 to the beginning of the file as shown in the above figure.

6. Edit the entry and add an `-iice` option to the line as shown in the example below (the Copy TCL command does not automatically include the IICE unit in the entry):

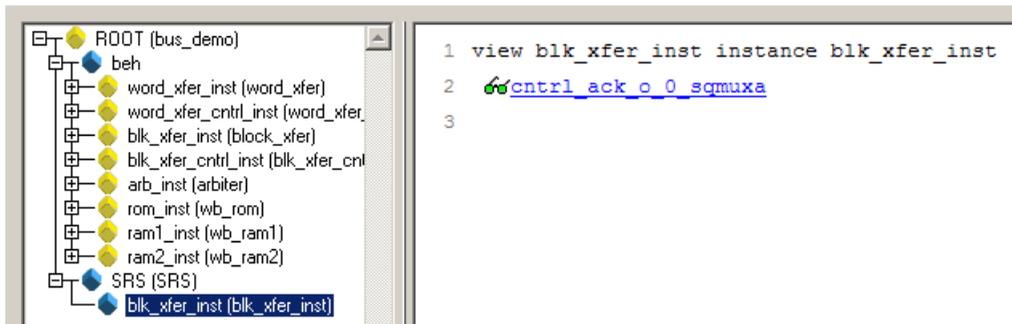
```

signals add -iice {IICE} -sample
    {/SRS/blk_xfer_inst/cntrl_ack_o_0_sqmuxa}

```

7. Save the edited identify.idc file and rerun synthesis.

When you open the Identify debugger, an SRS entry is included in the hierarchy browser; selecting this entry displays the additional signal or signals added to the identify.idc file. Selecting a signal in the instrumentation window brings up the Watchpoint Setup dialog box to allow a trigger expression to be assigned to the defined signal.

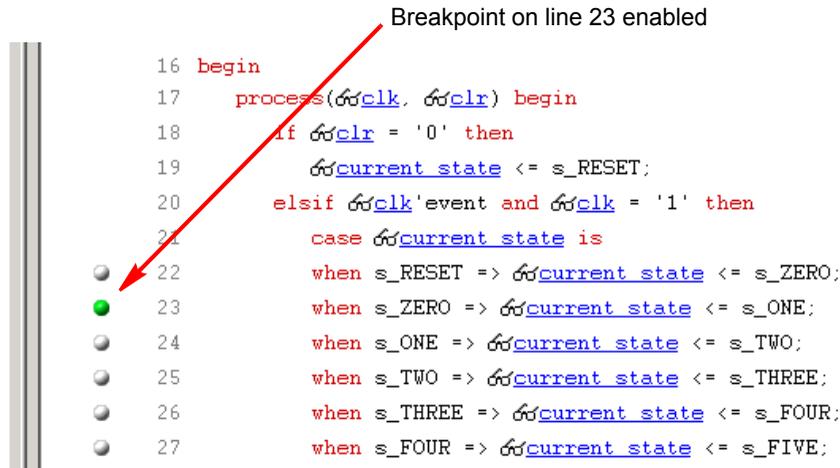


Note that trigger expressions on signals added to the identify.idc file must use the VHDL style format.

Selecting Breakpoints

 Breakpoints are used to trigger data sampling. Only the breakpoints that are instrumented in the Identify instrumentor can be enabled as triggers in the Identify debugger. To instrument a breakpoint in the Identify instrumentor, simply click on the circular icon to the left of the line number. The color of the icon changes to green when enabled.

Breakpoint on line 23 enabled



```

16 begin
17   process(clk, clr) begin
18     if clr = '0' then
19       current_state <= s_RESET;
20     elsif clk'event and clk = '1' then
21       case current_state is
22         when s_RESET => current_state <= s_ZERO;
23         when s_ZERO => current_state <= s_ONE;
24         when s_ONE => current_state <= s_TWO;
25         when s_TWO => current_state <= s_THREE;
26         when s_THREE => current_state <= s_FOUR;
27         when s_FOUR => current_state <= s_FIVE;

```

Once a breakpoint is instrumented, the Identify instrumentor creates trigger logic that becomes active when the code region in which the breakpoint resides is active.

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

Selecting Breakpoints Residing in Folded Hierarchy

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

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

The example below consists of a top-level entity called `top` and two instances of the `repeated_unit` entity. The source code of `repeated_unit` is displayed; the list of instances of the breakpoint on line 100 is displayed by clicking on the breakpoint icon next to the line number. As shown in the following figure, three instances of the breakpoint are available for sampling.

Any or all of these breakpoints can be selected by clicking on the corresponding line entry in the list displayed.

Folded breakpoint on line 100 selected

```

95     begin
96         if clk = '0' then
97             tmp <= "0000";
98         elsif clk'event and clk = '1' then
99             if ( clr = '0' ) then
100                tmp <= tmp + 2;
                /rtl/folded2_inst0/rtl/cnt_inst0/rtl2/process_94/if_96/if_99/repeated_unit.vhd:100
                /rtl/folded2_inst0/rtl/cnt_inst1/rtl2/process_94/if_96/if_99/repeated_unit.vhd:100
                /rtl/cnt_inst2/rtl2/process_94/if_96/if_99/repeated_unit.vhd:100
103            end if;
104        end if;
105    end process;

```

The color of the breakpoint icon is determined as follows:

- If no instances of the breakpoint are selected, the icon is clear in color.
- If some, but not all, instances of the breakpoint are selected, the icon is yellow.
- If all instances are selected, the icon is green.

Alternately, any of the instances of a folded breakpoint can be selected or deselected at the Identify instrumentor console window prompt by using the absolute path name of the instance. For example,

```

breakpoints add
    /rtl/inst0/rtl/process_18/if_20/if_23/repeated_unit.vhd:24

```

See the *Reference Manual* for more information.

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

Configuring the IICE



If the IICE configuration parameters for the active IICE need to be changed, use the Actions->Configure IICE menu selection or the Edit IICE settings icon to change them. [Chapter 4, IICE Configuration](#), discusses how to set these parameters for both single- and multi-IICE configurations and for the HAPS deep trace debug feature.

Real-time Debugging

Real-time debugging is a feature that allows users of HAPS-50 and HAPS-60 series boards to provide scope or logic analyzer access to instrumented signals directly through a Mictor board interface connector installed on the HAPS board.

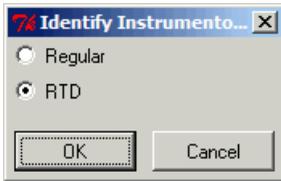
The use of this feature requires:

- A HAPS-50 or HAPS-60 series board
- One or more Mictor boards installed in HapsTrakII connectors
- Synopsys HAPS device family specified in the synthesis tool

Enabling the Real-time Debugging Feature

The real-time debugging feature can be used only when the above requirements are met. To use real-time debugging, a special IICE is defined in either the user interface or by command entry in the console window.

To specify the IICE from the user interface, open the Project view and click the New IICE button to display the following dialog box. Select the RTD radio button and click OK.



To define the IICE from the console window, enter the `iice new` command:

```
iice new [iiceID] -type rtd
```

In the command syntax, *iiceID* is the name of the new IICE and, if omitted, defaults to an incremental number (for example, IICE_0).

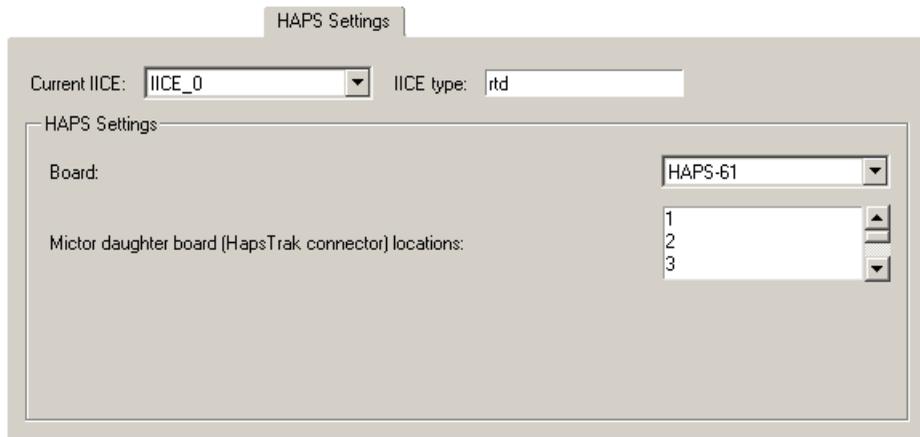
Either of the above methods creates a new, real-time IICE for the design with all of the signals “not instrumented.” This new IICE is identified by the “R” symbol in the IICE tab.



Before you can instrument any of the signals, you must configure the HAPS Settings tab as outlined in the next section.

HAPS Settings Tab

The HAPS Settings tab for the real-time debugging feature includes a drop-down menu for selecting the HAPS board type and a set of Mictor board location check boxes. The available board selections are determined by the Synopsys HAPS device selection on the Device tab of the Implementation Options dialog box in the synthesis tool. When a HAPS-60 based device is targeted, the available selections are HAPS-61, HAPS-62, and HAPS-64 as shown in the following figure. Similarly, when a HAPS-50 based device is targeted, the available radio buttons are HAPS-51, HAPS-52, and HAPS-54. The Mictor daughter board location check boxes identify the corresponding HapsTrakII connector location.

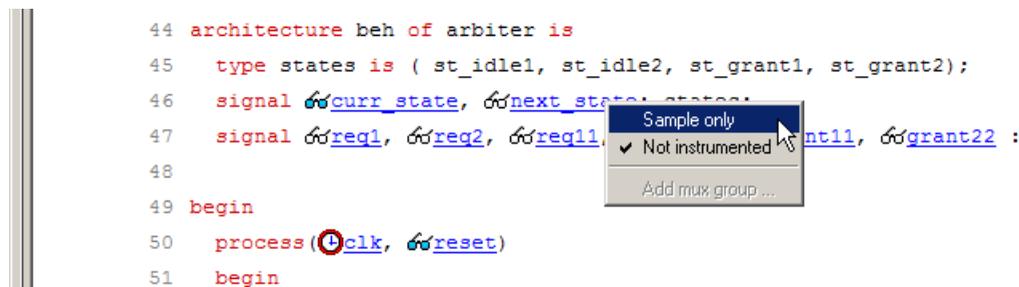


On the IICE Sampler tab:

1. Specify the HAPS board you are using from the Board drop-down menu.
2. Specify one or more Mictor board HapsTrakII connector locations by clicking on the connector number.
3. When finished with the above entries, click the OK button at the bottom of the tab.

Instrumenting the Real-Time Debug Signals

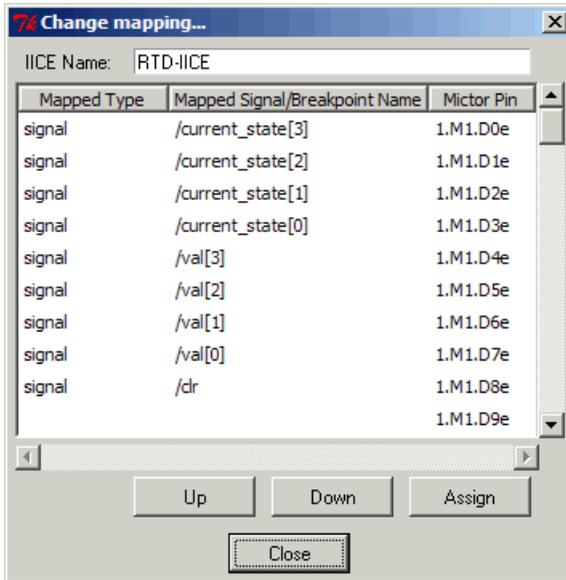
Instrumenting signals for real-time debugging is similar to normal instrumentation in that signal watchpoints and breakpoints are identified and activated in the instrumentation window. The exception is that the only watchpoint selection available from the popup menu for real-time debugging signals is Sample only.



Viewing the Signal Assignments

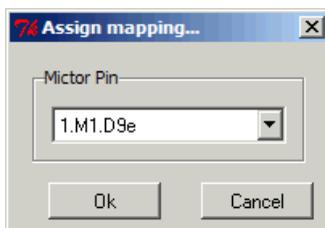


Watchpointed signals are automatically assigned to the specified Mictor daughter board pin locations. These assignments are listed in the Change mapping dialog box. To display the dialog box, click the 'R' icon (Change 'RTD' type IICE signals/breakpoints mapping) in the top menu bar.



Individual assignments can be changed by highlighting the assignment to be changed and then either:

- moving the assignment up or down using the Up or Down buttons.
- clicking the Assign button to display the Assign mapping dialog box and then selecting the new pin location from the drop-down menu and clicking OK.



Logic Analyzer Interface

The logic analyzer interface at the Mictor connector is configured in the Identify debugger (see [Logic Analyzer Interface Parameters](#), on page 137).

Writing the Instrumented Design

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

1. Define a synthesis project with all source files defined
2. Specify IICE parameters/HAPS settings
3. Select signals to sample
4. Select breakpoints to instrument
5. Optionally include the original HDL source

Note: To include the original HDL source with the project, select Options->Instrumentation preference from the menu to display the Instrumentation Preferences dialog box and select the Save original source in instrumentation directory check box. If the original source is to be encrypted, additionally select the Use encryption check box. Selecting Save original source in instrumentation directory saves the original HDL source to the `orig_sources` subdirectory in the instrumentation directory when you instrument your design.



Finally, click on the Save project's active instrumentation icon to capture your instrumentation. Saving a project's instrumentation generates an *instrumentation design constraints* (`.idc`) file and adds compiler pragmas in the form of constraint files to the design RTL for the instrumented signals and break points. This information is then used by the synthesis tool to incorporate the instrumentation logic (IICE and COMM blocks) into the synthesized netlist. If you are including an encrypted HDL source (Use encryption box checked), you are first prompted to supply a password for the encryption.

Including Original HDL Source

Including the original HDL source with the instrumented project simplifies design transfer when instrumentation and debugging are performed on separate machines and is especially useful when a design is being debugged on a system that does not have access to the original sources.

As explained in [Debugging on a Different Machine, on page 127](#), you can simply copy the entire implementation directory to the debug system; the Identify project will be able to locate the original sources for display. To include the original HDL source, select the Save original source in instrumentation directory check box from the Instrumentor Preferences dialog box (Options->Instrumentation preferences) before you save and instrument your project. The clear text or encrypted source is included in the `orig_sources` subdirectory.

When the Use encryption check box is additionally selected, the original sources are encrypted when they are written into the `orig_sources` subdirectory. The encryption is based on a password that is requested when you write out the instrumented project. Encryption allows you to debug on a machine that you feel would not be sufficiently secure to store your sources. After you transfer the instrumentation directory to the unsecure machine, you are prompted to reenter the encryption password when you open the project in the debugger.

For maximum security when selecting an encryption password:

- use spaces to create phrases of four or more words (multiple words defeat dictionary-type matching)
- include numbers, punctuation marks, and spaces
- make passwords greater than 16 characters in length

Note: Passwords are the user's responsibility; Synopsys cannot recover a lost or forgotten password.

The decrypted files are never written to the unsecure machine's hard disk. Users are discouraged from transferring the `instr_sources` directory to the unsecure machine, as this directory is not required for debugging.

Synthesizing Instrumented Designs

When you save your instrumentation, the following files and subdirectories are updated or created in the synthesis project implementation directory (*projectName/rev_n*):

- an `identify.idc` file describing the instrumented watchpoints and breakpoints; this file can be manually edited to add additional watchpoints (see [Instrumenting Signals Directly in the `idc` File, on page 82](#)).
- `instr_sources` subdirectory containing:
 - the IICE core file (`syn_dics.v`)
 - a Synopsys FPGA constraints file (`syn_dics.sdc`) – this file is used directly by the synthesis tool
 - a Synopsys FPGA compiler design constraints file (`syn_dics.cdc`)
- an optional `orig_sources` subdirectory containing the original HDL files in either encrypted or clear-text format
- an `identify.db` encrypted data file

Note: When instrumenting a VHDL file that is not compiled into the work library, make sure that the `syn_dics.vhd` file is included in the synthesis project ahead of the user design files. Additionally, this file must be compiled into the work library.

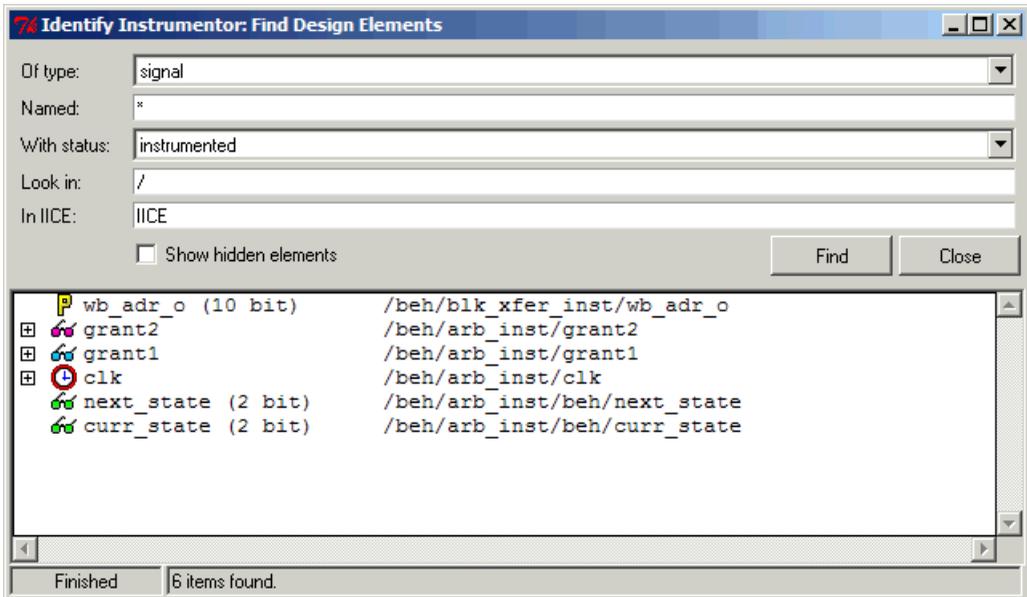
Listing Signals

The Identify instrumentor includes a set of menu commands and, in most cases, icons for listing watchpoint and breakpoint conditions.

List Instrumented Signals



To view all of the signals currently instrumented in the entire design, use the Actions->Show Instrumented Signals menu selection or click the Show Instrumented Signals icon. The result of listing the signals is displayed in the Find dialog box.



List All Signals

To view all of the signals in the design, use the Actions->Show All Signals menu selection.

List Signals Available for Instrumentation



To see only the signals in the design available for instrumentation, use the Actions->Show Possible Signals menu selection or the Show Possible Signals icon.

List Instrumented Breakpoints



To list all of the breakpoints that have been instrumented, use the Actions->Show Instrumented Breakpoints menu selection or the Show Instrumented Breakpoints icon.

List All Breakpoints

To list all breakpoints, use the Actions->Show All Breakpoints menu selection.

List Breakpoints Available for Instrumentation



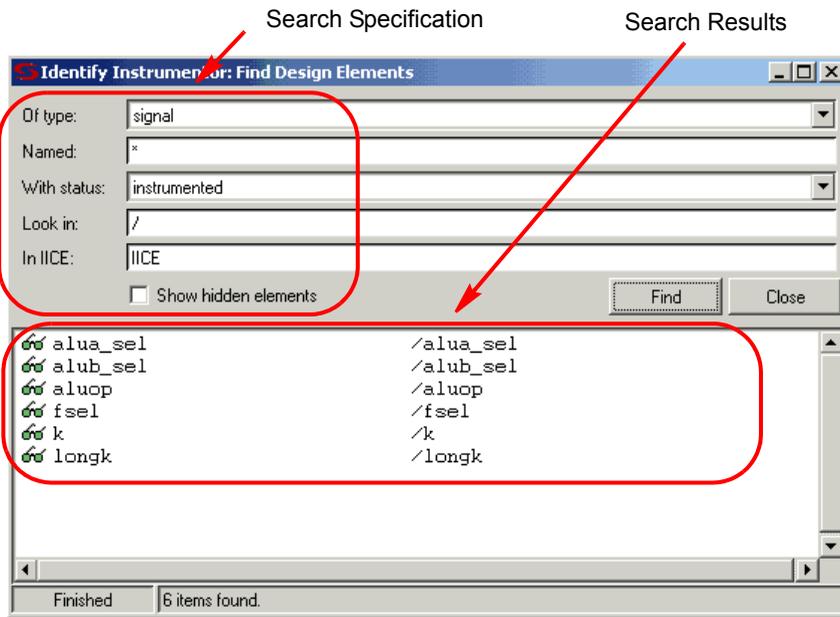
To list all of the breakpoints that are available for instrumentation, use the Actions->Show Possible Breakpoints menu selection or the Show Possible Breakpoints icon.

Searching for Design Objects



The Find dialog box is a general utility to recursively search for signals, breakpoints, and/or instances. To invoke the Find dialog box, use the Edit->Find menu selection or the Display find dialog icon.

The Find dialog box has an area for specifying the objects to find and an area for displaying the results of the search.



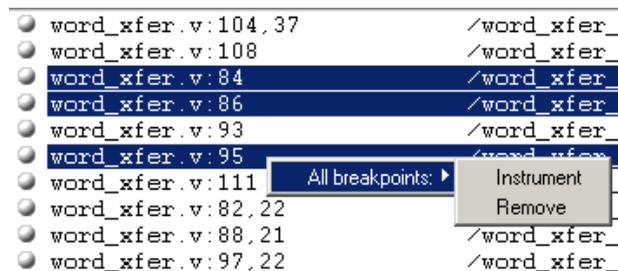
The search specification includes these options:

- **Of type:** – specify which type of object to search for: breakpoint, signal, instance, or “*” (any). The default is “*” (any).
- **Named:** – specify a name, or partial name to search for in the design. Wild cards are allowed in the name. The default is “*” (any).
- **With status:** – specify the status of the object to be found. The values can be instrumented, sample trigger, sample_only, trigger_only, not-instrumented, or “*” (any). The default is “*” (any).
- **Look in:** – specify the location in the design hierarchy to begin the recursive search. The root (“/”) is the default setting.
- **In IICE:** – a read-only field that indicates which IICE is to be searched when multiple IICEs are defined. To search another IICE, close the Find dialog box, click the desired IICE tab, and reopen the Find dialog box.

The search specification also includes a Show hidden elements check box. When this check box is enabled, the search also includes objects that would not normally be searched such as breakpoints in dead code.

The search results window shows each object found along with its hierarchical location. In addition, for breakpoints and signals, the results window includes the corresponding icon (watchpoint or breakpoint) that indicates the instrumentation status of the qualified signal or breakpoint.

To change the instrumentation status of a signal, click directly on the watchpoint icon and select the instrumentation type from the popup menu. You can use the Ctrl and Shift keys to select multiple signals and then apply the change to all of the selected signals. To toggle the instrumentation status of a breakpoint, click the breakpoint icon. You also can use the Ctrl and Shift keys to select multiple breakpoints and then apply the change to all selected breakpoints from the popup menu.



Console Text

To capture all 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 from the console, use the clear command or click the right mouse button from within the console window and select Clear from the menu.

CHAPTER 8

Identify Debugger

The Identify debugger enables HDL designs to be debugged 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 Identify debugger through a communications port where it can be displayed in a variety of formats.

This chapter describes:

- [Invoking the Identify Debugger, on page 98](#)
- [Identify Debugger Windows, on page 99](#)
- [Commands and Procedures, on page 104](#)
- [Debugging on a Different Machine, on page 127](#)
- [Simultaneous Debugging, on page 128](#)
- [Identify-Analyst Integration, on page 129](#)
- [Waveform Display, on page 134](#)
- [Logic Analyzer Interface Parameters, on page 137](#)
- [Console Text, on page 141](#)

Invoking the Identify Debugger

The Identify debugger can be launched directly from a synthesis project or the Identify debugger can be explicitly opened directly from a Windows or Linux system.

Synthesis Tool Launch

If you are using a Synopsys FPGA synthesis tool or the Certify tool, invoke the Identify 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 Identify debugger IICE instrumentation window opens with the corresponding project displayed (see [Instrumentation Window, on page 100](#)).

Operating System Invocation

The Identify 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 Identify debugger from a Linux system:

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

The initial Identify 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 Identify instrumentation (IICE) window opens with the corresponding project displayed (see [Project Window, on page 102](#)).

Identify Debugger Windows

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

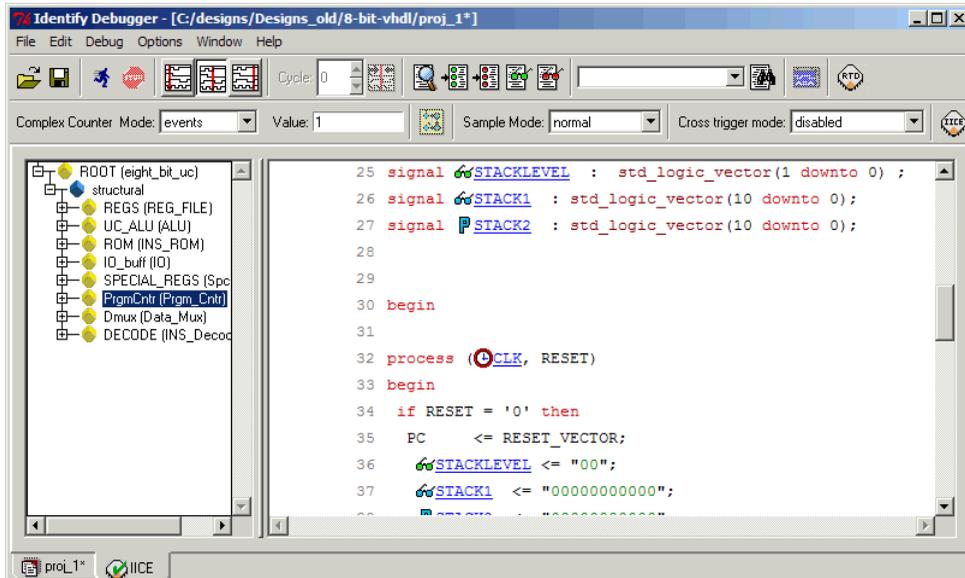
- IICE instrumentation window
- Console window
- Project window

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

- a project (with an HDL design) has been loaded into the Identify instrumentor and instrumented
- the design has been synthesized in your 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 Identify debugger

Instrumentation Window

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



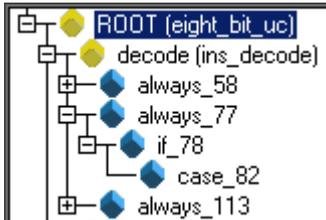
Hierarchy Browser

Source-Code Display

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 Identify instrumentor are not displayed in the Identify 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 105](#).

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 113](#).

Console Window

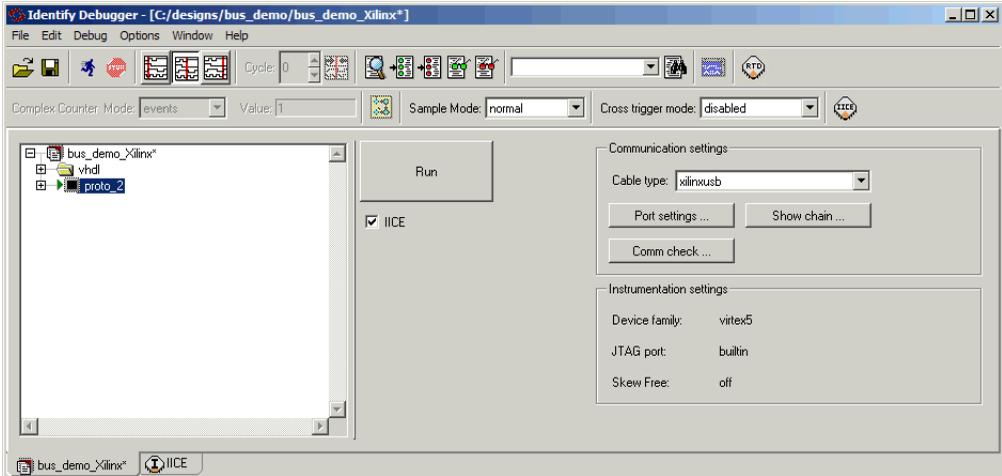
The Identify debugger console window displays commands that have been executed, including those executed by menu selections and button clicks. The Identify debugger console window also allows you to type Identify debugger commands and to view the results of command execution.

```
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$
```

Project Window

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

Opening and Saving Projects

The Identify 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.

See [Chapter 3, Project Handling](#), for details on setting up and managing projects.

Executing a Script File

A script file contains Identify 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 Chapter 4, *Alphabetical Command Reference*, 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 10, 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 Identify 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.

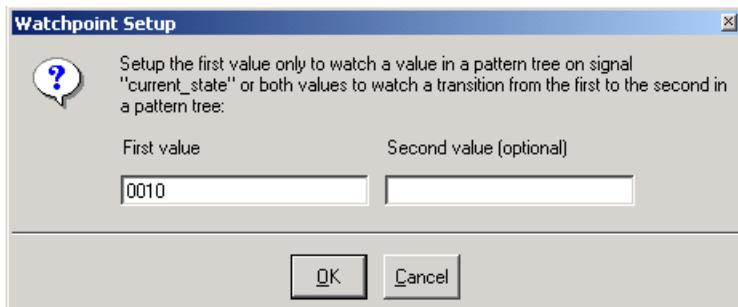


A watchpoint is set on a partial bus signal by clicking-and-holding on the signal or the “P” icon next to the signal, selecting the partial bus group from the list displayed, and then selecting the Set Trigger Expressions menu item to bring up the Watchpoint Setup dialog box.

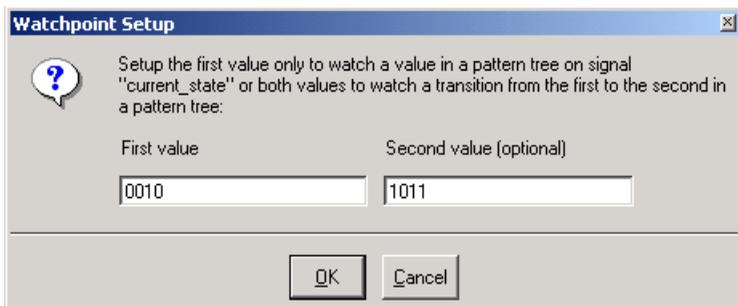
There are two forms of watchpoints: value and transition.

- 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.



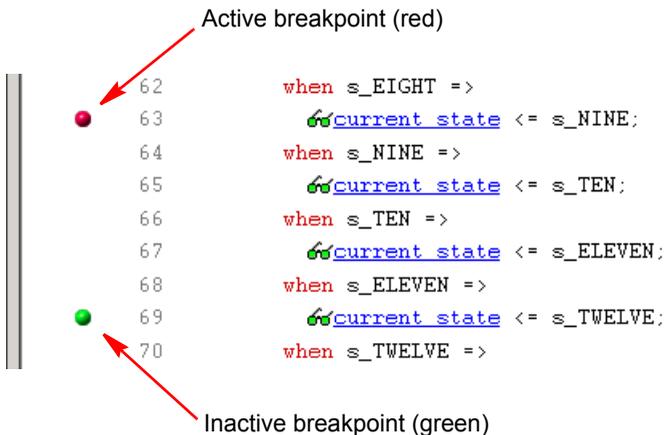
The Watch menu selection will have a check mark to indicate that the watchpoint is activated. Click on the Watch menu selection to toggle the check mark and deactivate the watchpoint.

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 Identify 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.

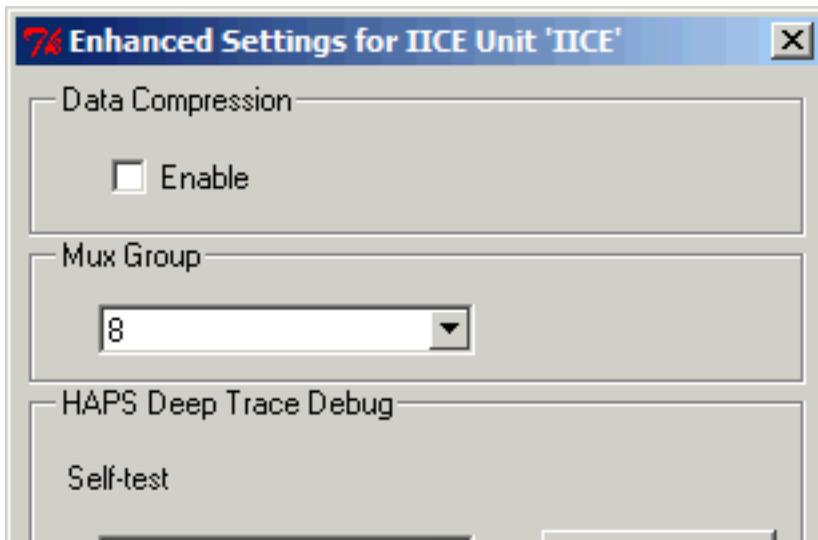
Selecting Multiplexed Instrumentation Sets

Multiplexed groups of instrumented signals defined in the Identify instrumentor can be individually selected for activation in the Identify debugger (for information on defining a multiplexed group in the Identify instrumentor, see [Multiplexed Groups, on page 79](#)).

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 Identify 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 79](#) 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 Identify 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.

```

-
5  entity repeated_unit is
6    port(
7      clr : in std_logic;
8       val : out std_logic;
9      clk : in std_logic;
10   );
11 end repeated_unit;
12
13 architecture rtl of repeated_unit is
14 begin
15
16    val <= clk and clr;

```



The color of the watchpoint icon is determined as follows:

- If no instances of the signal are activated, the watchpoint icon is green in color.
- If some, but not all, instances of the signal are activated, the watchpoint icon is yellow in color.
- If all instances are activated, the watchpoint icon is red in color.

For related information on folded hierarchies, see [Sampling Signals in a Folded Hierarchy, on page 80](#) and [Displaying Data from Folded Signals, on page 119](#).

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:

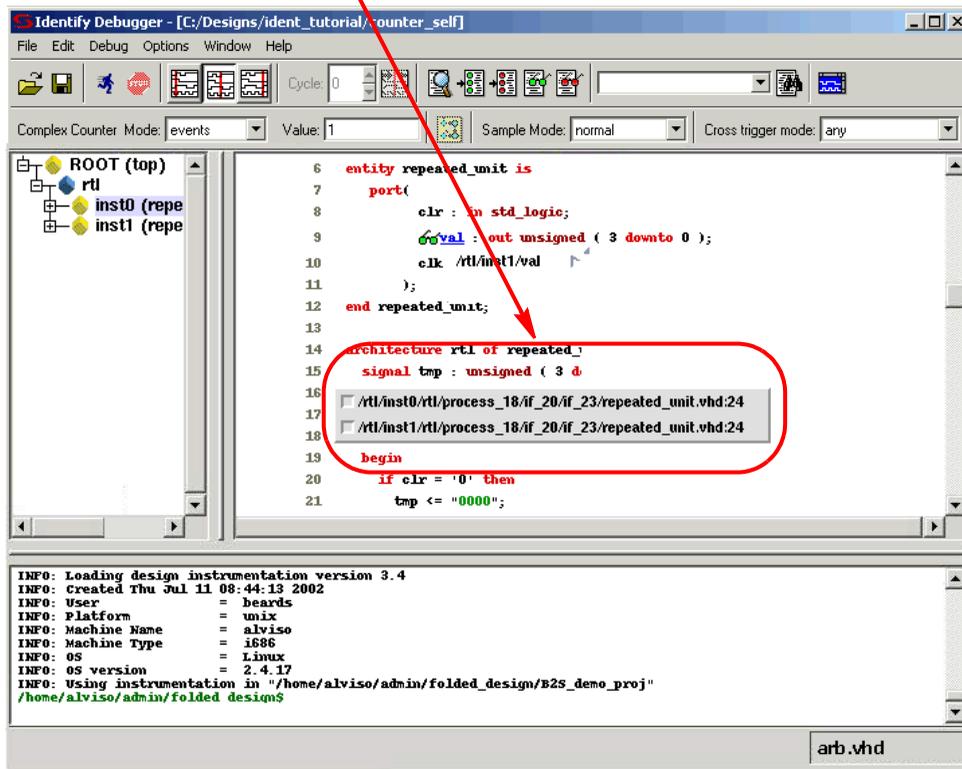
```

/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:

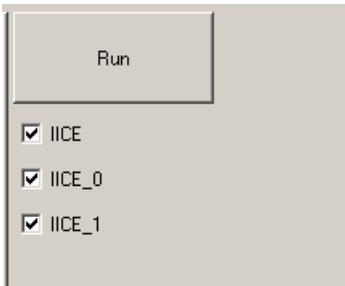
- If no instances of the breakpoint are activated, the breakpoint icon is green.
- If some, but not all, instances of the breakpoint are activated, the breakpoint icon is yellow.
- If all instances are activated, the breakpoint icon is red.

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 114](#). 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 Identify debugger waveform command) or written out to a file (see the Identify 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.

Activated and triggered breakpoint

Sampled data (in yellow)

```

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

```

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 Identify 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 Identify 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.

Stop Command



The Stop command sends control back to the Identify 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 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 Identify 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.

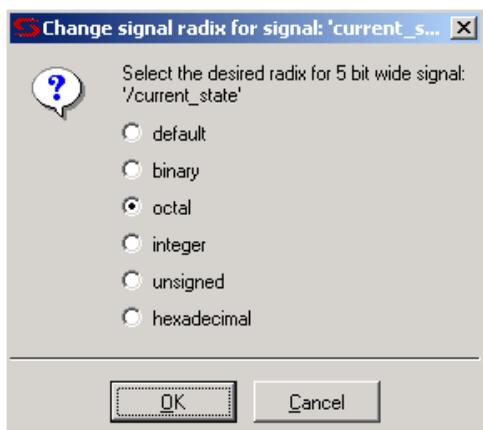


To reset the cycle number to the default position (the zero time position), use the Debug->Cycle->home menu selection or click on the Goto trigger event in sample history icon.

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 watchpoint 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 Identify 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.

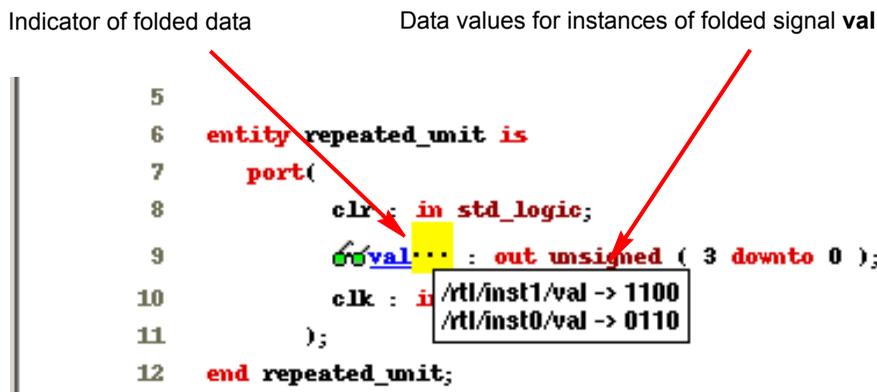
Displaying Data from Folded Signals

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 Identify 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 Identify debugger displays an ellipsis (...) to indicate that there are multiple values present. To display all of the instrumented values, click-and-hold on the ellipsis indicator.

The example below consists of a top-level entity called top and two instances of the repeated_unit entity. In the example, the source code of repeated_unit is displayed, and both of the lists of instances of the signal val have been instrumented. The two instances are /rtl/inst0/val and /rtl/inst1/val, and their data values are displayed in the pop-up window as shown in the following figure:



For related information on folded hierarchies, see [Sampling Signals in a Folded Hierarchy](#), on page 80 and [Activating/Deactivating Folded Instrumentation](#), on page 110.

Displaying Data for Partial Buses

When debugging designs with partially instrumented buses, the Identify 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.

The screenshot shows a debugger window with the following text:

```

342
343 input 64 wr cmd;
344 input [63:0] → P data_in 64'h3fad7910d1????36
345 output [63:0] → data_out 64'b111111001110011
346

```

Two red arrows point to the text above the code: "Composite bus value" points to the entire line 344, and "Data values of partial buses" points to the tooltip. The tooltip contains the following text:

```

data_in_63_32 -> 32'h3fad7910
/data_in_7_0 -> 8'h36
/data_in_31_24 -> 8'hd1

```

Note that 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 Identify 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_Struct_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   clk_ip,
12   output type_Unsigned_P_Struc_data  sig_oport_P_Struc_data
13 );
14
15 logic           tb_rst 1'b1;
16 shortint      unsigned  rst_cnt 65535;
17
18 type_P_Struc_instr  sig_iport_P_Struc_instr CMP {{4'b0000} {4'b0010}}
19
20 always @ (posedge  clk_ip) //rst generation

```

The Find dialog in the Identify 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).

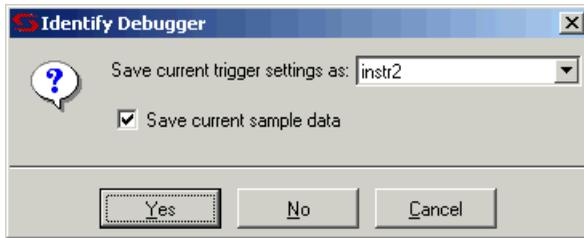
Saving and Loading Activations

The Identify 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 Identify 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.

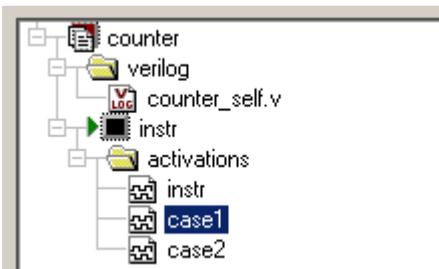


4. Enter (or select) an activation name in the Save current trigger settings as: field. Selecting an existing activation from the drop-down menu overwrites the selected activation.
5. To include the sample data set with the activation, enable the Save current sample data check box.
6. Click Yes to save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

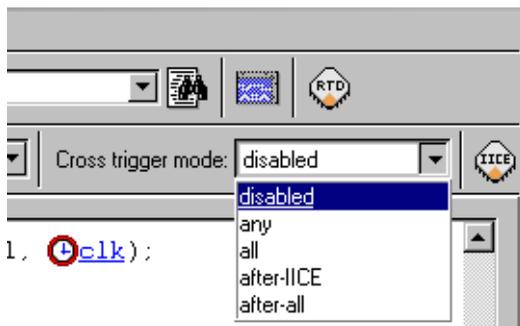
Note: To save a specific activation, always use Save current activations to explicitly name the file and prevent the data from overwriting the `last_run.adb` file.

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 Identify 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 Identify debugger as shown below.



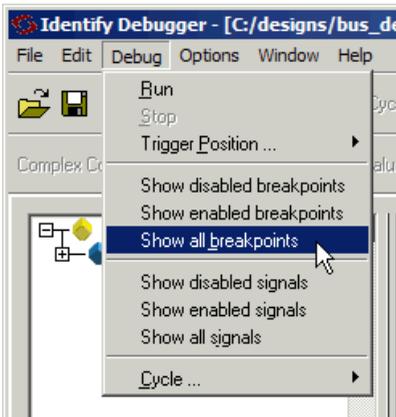
The drop-down menu selections are as follows:

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>iiceName</i>	Event trigger from local IICE unit occurs only after the event at selected external IICE unit <i>iiceName</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

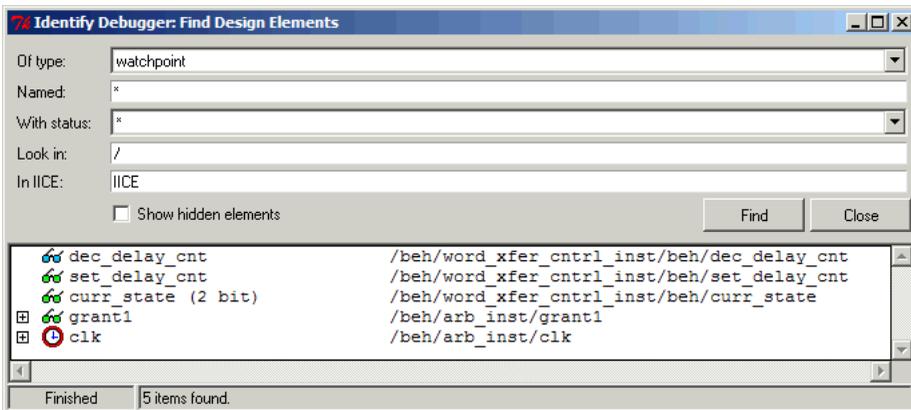
Note: If the drop-down menu does not display, make sure that Allow cross-triggering in IICE is enabled on the IICE Controller tab of the Identify instrumentor and that you have defined more than one IICE unit.

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.



Show Watchpoint/Breakpoint Icons

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.

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 both the project file (*projectName.prj*) and the Identify implementation directory (e.g., *rev_1*) to the lab machine.

Because the Identify tool set allows you to debug your design in the HDL, the Identify debugger must have access to the original source files. Depending on the type of your network, the Identify 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 Identify 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:

- Identify 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 Identify 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 Identify 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.

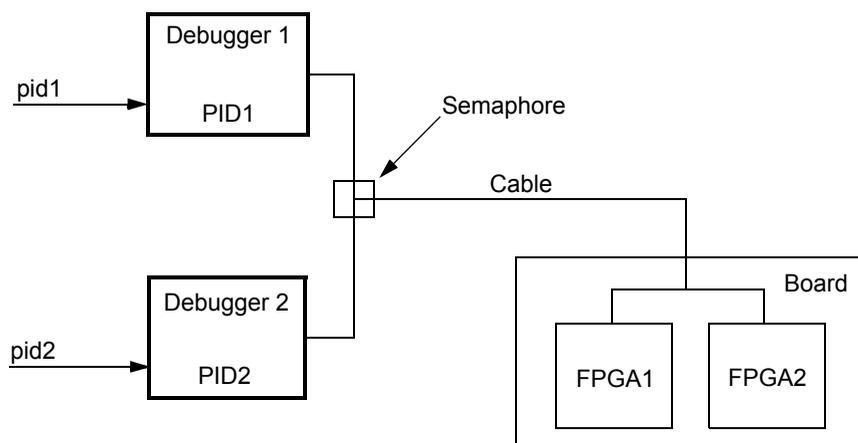
```
searchpath {d:/temp;c:/Documents and Settings/me/my_design/}
```

The Identify debugger will only display 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 Identify instrumentor can password-protect the original sources on the development machine for use with the Identify debugger (for information on file encryption, see [Including Original HDL Source, on page 91](#)).

Simultaneous Debugging

When multiple Identify 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.



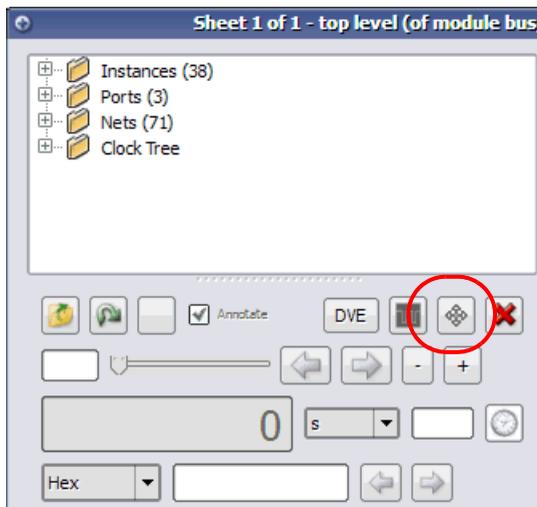
Identify-Analyst Integration

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

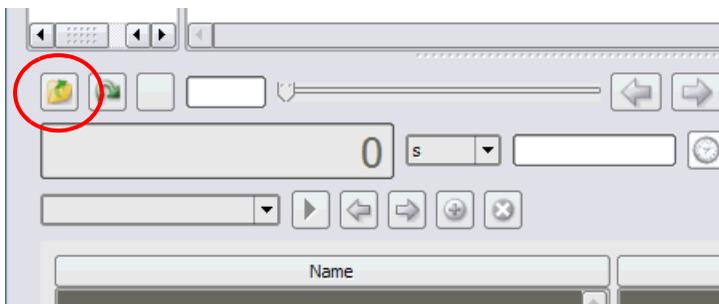
The following steps outline an abbreviated procedure for using an Identify-generated VCD file with the HDL Analyst. For a complete description of this feature, see VCD-Analyst Integration in the *Synopsys FPGA Synthesis User Guide*.

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

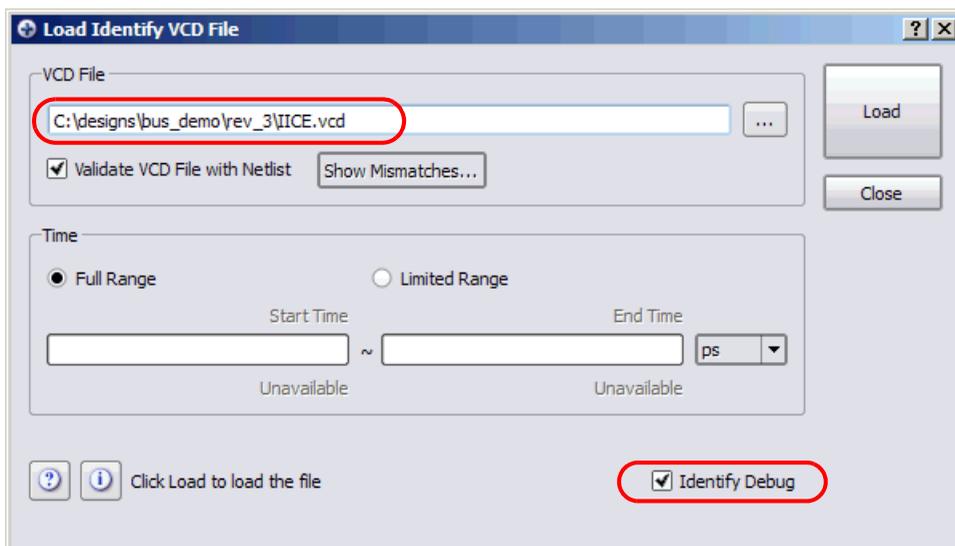
1. Click the VCD Panel icon () or select VCD->VCD Panel from the HDL-Analyst menu to display the VCD control panel.
2. If necessary, click the Move this panel to an alternate location button to relocate the VCD control panel under the RTL view.



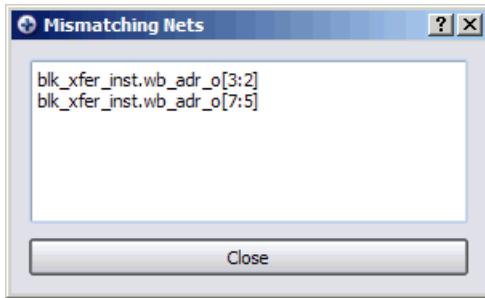
3. Click the Open a VCD File icon () or select VCD->Load VCD File from the HDL-Analyst menu to open the Load Identify VCD File dialog box.



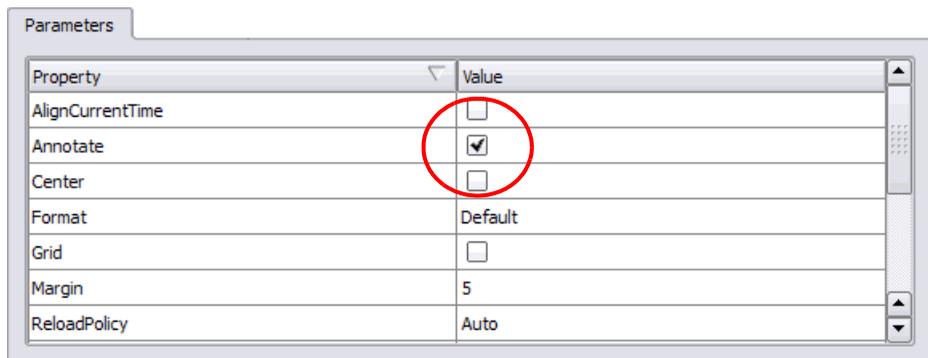
- In the dialog box, enter the path to the vcd file generated by the Identify debugger (use the browse ... button) and make sure that the Identify Debug box is checked. The Validate VCD File with Netlist check box, when enabled, checks for mismatches between the design netlist and the VCD file loaded.



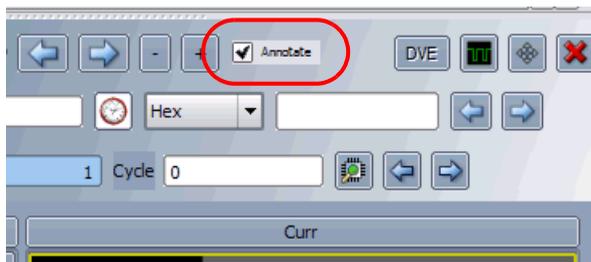
- Click the Load button to load the VCD file and display the instrumented signals from the VCD file in the waveform viewer.
- If Validate VCD File with Netlist is checked, click the Show Mismatches button to display any mismatched nets. Mismatches are reported in the Mismatching Nets dialog box.



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.



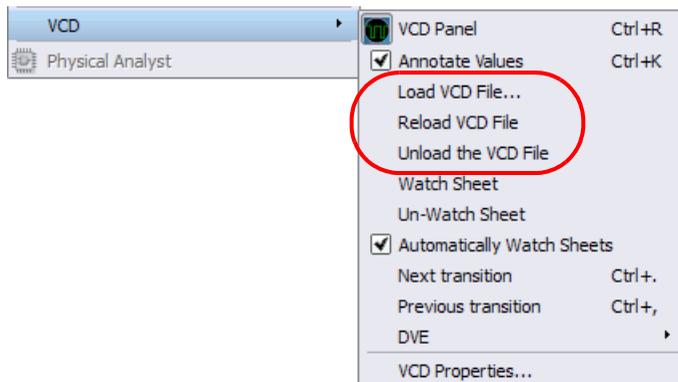
9. To annotate values on the waveform viewer to their respective HDL Analyst sheet, check the Annotate box on the control panel.



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 Identify VCD Files

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



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

When the Identify 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 an Identify VCD file is reloaded, the tool preserves information as much as possible such as the current time and watched signals.

- To unload an Identify VCD file, select Unload the VCD File. This option frees up memory used by the Identify 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:

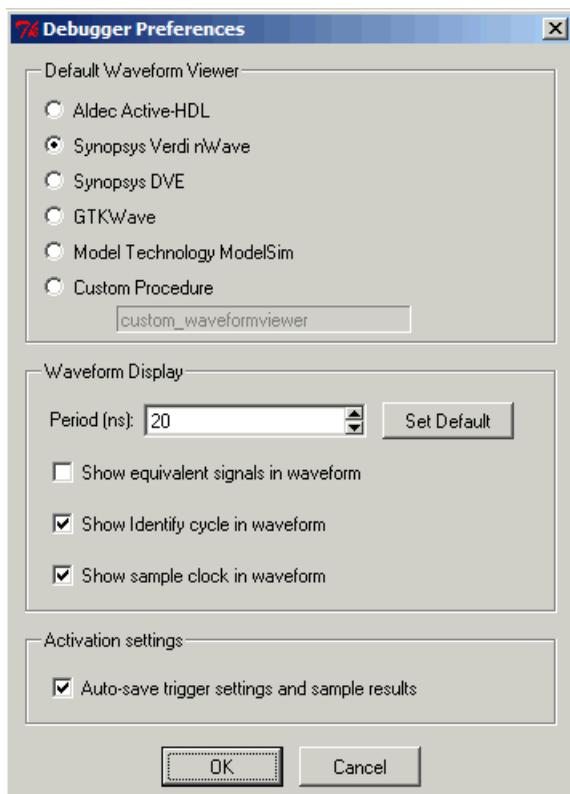
- Observing nets on a particular HDL Analyst sheet
- Changing the format of signals displayed in the viewer

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 Identify debugger. For details, see the application note, “Interfacing Your Waveform Viewer with the Identify 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.

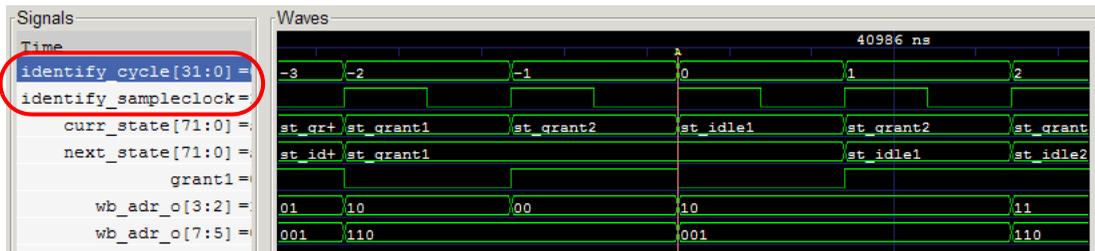


The Synopsys DVE and Verdi waveform viewers are only available on Linux platforms. To use the included GTKWave viewer, click the GTKWave radio button in the Default Waveform Viewer section.

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



After running the Identify 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).



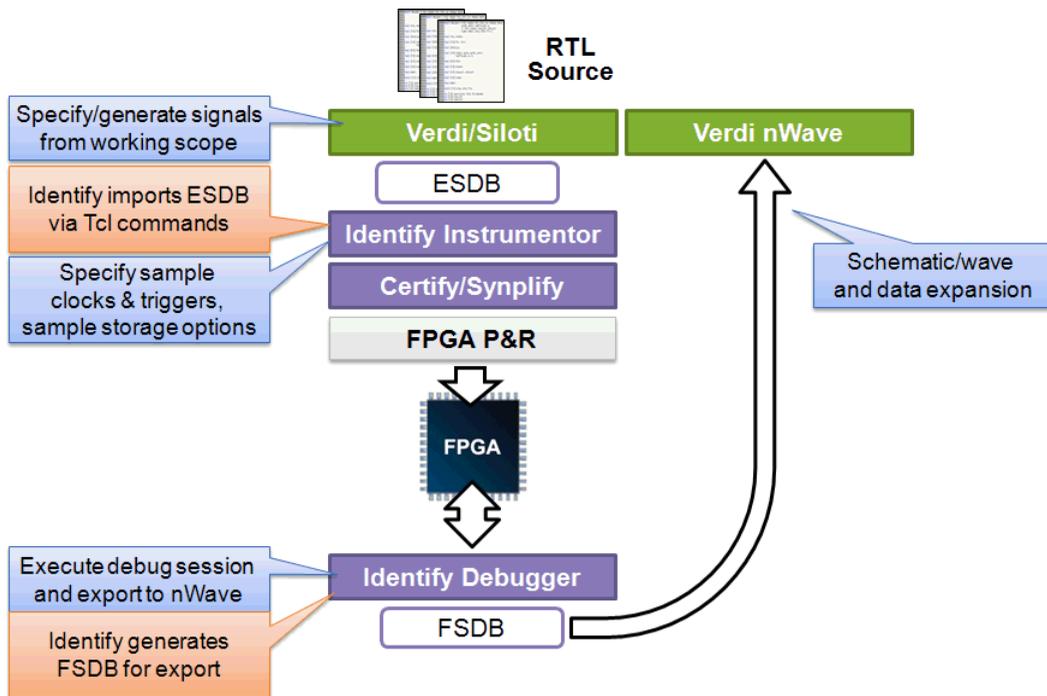
If you select a waveform viewer from the Waveform preference dialog box that is not installed, an error message is displayed when you attempt to invoke the viewer. To install the waveform viewer:

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 Identify debugger, select Window->Waveform or click on the Open Waveform Display icon.

Verdi nWave Viewer

The Verdi nWave viewer is used to display the fast signal database (FSDB) generated by the Identify debugger as shown in the flow diagram below.



In the diagram, Verdi/Siloti generates the essential signal database (ESDB) which is imported into the Identify instrumentor where the desired signals are instrumented. The instrumented design is then synthesized, placed and routed, and programmed into the FPGA. The Identify debugger samples the data and generates the FSDB database which is then displayed in the Verdi nWave viewer.

Logic Analyzer Interface Parameters



The logic analyzer interface parameters for the real-time debug feature in the Identify 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.

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

The screenshot shows a dialog box titled "Logic Analyzer Scan". It has a tab labeled "Logic Analyzer Scan". The dialog contains the following fields and controls:

- "Type of Logic Analyzer:" with a drop-down menu showing "tla".
- "TLA Script Program:" with a text box containing "c:\Program Files\TLA 700\System\Tlascript".
- "User Name:" with a text box containing "user".
- "Host Name/ IP Address:" with a text box containing "10.9.148.51".
- An unchecked checkbox labeled "Assign Pods automatically to mictor conenctors".
- A "Scan Logic Analyzer" button at the bottom.

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 Identify debugger host.

Assign Pods automatically to Mictor connectors

When checked, automatically assigns pods to the Mictor connectors.

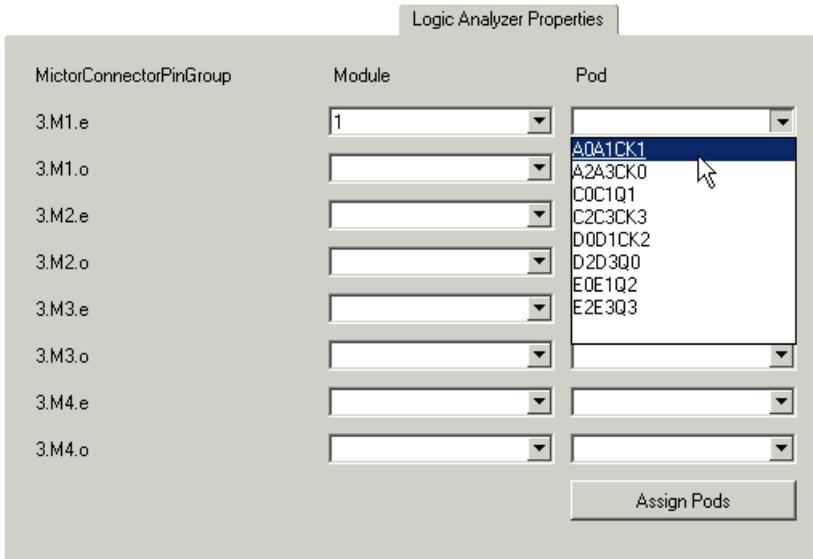
Scan Logic Analyzer

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.



Logic Analyzer Submit Tab

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



IICE Assignments Report Tab

When using the real-time debugging feature in the Identify instrumentor (see [Real-time Debugging, on page 86](#)), the signal/breakpoint interface assignments to the Mictor connector are reported in the Identify 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 55](#) in the *Reference Manual*).

The screenshot shows a window titled "IICE Assignments Report" with a scrollable content area. The content is divided into two sections: "Pod Assignments" and "Signal/breakpoint Assignments".

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]

Console Text

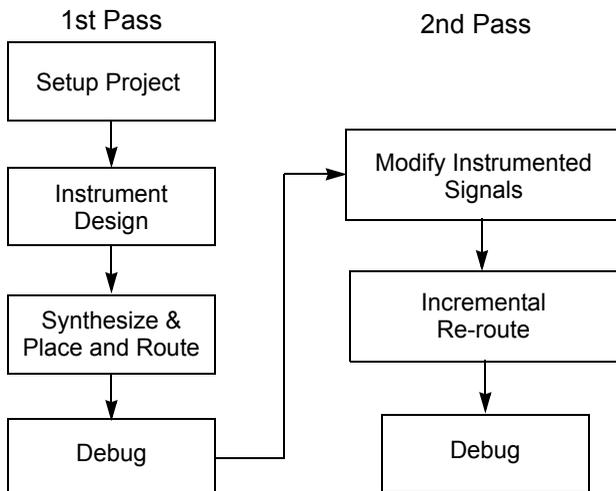
To capture all the text that is written to the console, use the `log console` command (see the *Reference Manual*). Alternately, you can click the right mouse button inside the console window and select `Save Console Output` from the menu. To capture all commands executed in the console window, use the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

CHAPTER 9

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

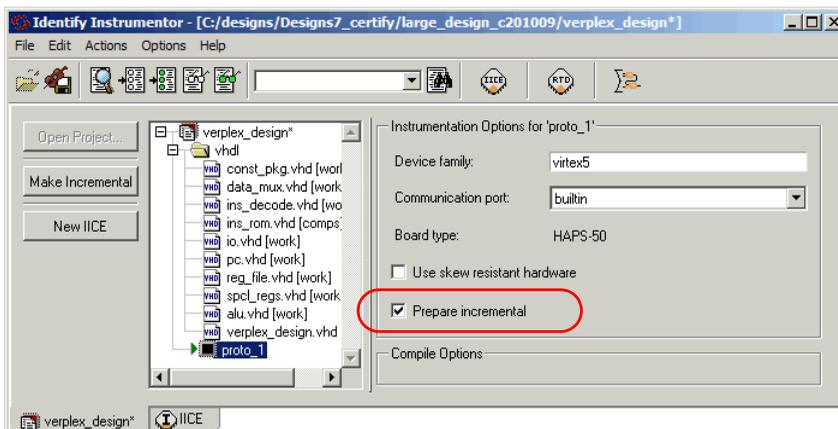
The incremental flow supported by the Identify tool set is available only when using a Virtex-5 or Virtex-6 Xilinx-based technology.

Note: Incremental flow is supported by the most recent versions of the Xilinx ISE software; see the release notes for the specific ISE release version used with the Identify tool set.

Setting up the Original Design

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

1. Open the design in the Identify instrumentor by clicking on the Identify implementation and selecting Launch Identify Instrumentor from the popup menu.
2. Select the project tab to display the project view.
3. In the project view, make sure that the Prepare incremental check box is checked.



4. Instrument and save your original design and close the Identify instrumentor.

5. 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 ncd file

Creating the Incremental Instrumentation

To make incremental changes to your original design:

1. From the synthesis tool, open the Identify 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 ncd file.

Note: Make sure that you enter the path to the correct ncd 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 file in the instrumentation directory. Use this ncd file to generate your new bit file.

Generating the Bitfile

Generating the bitfile from the new ncd file can be done manually or through the normal Xilinx tool flow.

Xilinx Tool Flow

To use the normal Xilinx flow to create the bitfile:

1. Copy the new ncd file into the Xilinx directory of the original instrumentation.
2. Rerun **ONLY** the bitgen portion of the flow.

Note: Be careful not to rerun the entire flow which would overwrite the new ncd file and revert to the initial instrumentation.

Manual Generation

The Xilinx bitgen command can be used to manually generate the bitfile from the new ncd file. The syntax for this command is:

```
bitgen designName.ncd designName.bit
```

Note: There are many options to the bitgen command that are usually contained in a board-specific ut file. You can call bitgen with the *board-Name.ut* file as an option to pick up any site-specific command options.

Debugging the Incremental Version

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

CHAPTER 10

IICE Hardware Description

The Identify instrumentor adds instrumentation logic to your HDL design that allows you to understand and debug the operation of your design. There are some aspects of the instrumentation logic that are important to understand in order to use the Identify 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 Identify implementation of the TAP controller (the soft option). See [Chapter 11, *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 151](#)
- [Multiply Activated Breakpoints and Watchpoints, on page 151](#)

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.

Multiply Activated Breakpoints and Watchpoints

How breakpoints and watchpoints operate individually has been described earlier in this 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 sampling block. When the sampling 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 sampling block are uploaded to the Identify debugger for display and formatting.

Whenever possible, the sampling 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 Identify instrumentor IICE Controller tab of the IICE Configuration dialog box or using the Identify 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

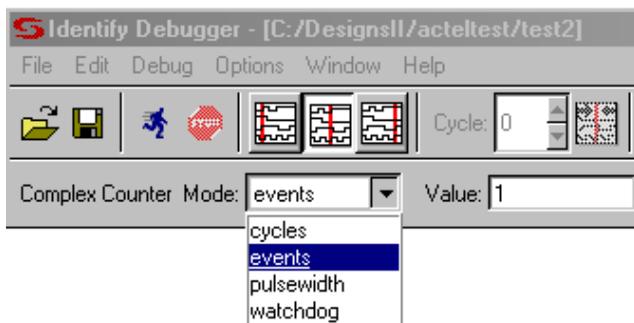
During debugging in the Identify debugger, the complex counter is used to produce complex triggering behavior.

During the debugging of the design, the complex counter is set to zero on invocation of the Identify 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 Identify debugger. It explains the different choices available during instrumentation and the functionality these choices provide in the Identify 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 Identify 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 Identify 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 Identify 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 Identify instrumentor console window.

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

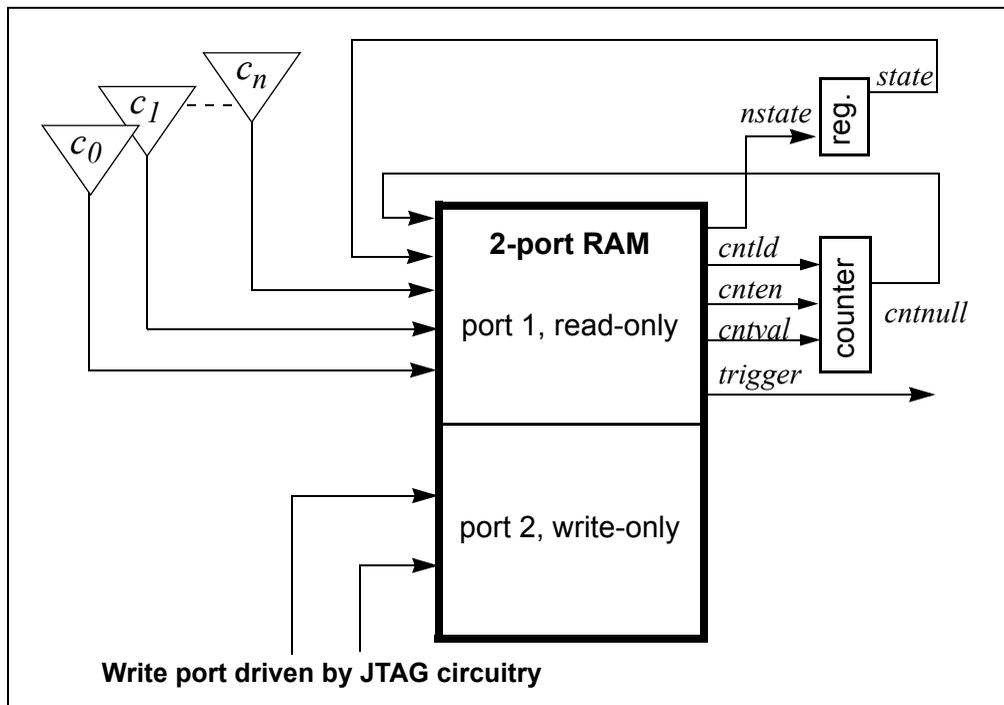
The other address inputs are provided by the state register.

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 10-1: 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 Identify instrumentor, this information should be monitored to make sure that no resources are wasted.

Using State Machine Triggering in the Identify Debugger

Perform the following steps in the Identify 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 Identify debugger watch and stop commands.
- setup the trigger state machine behavior using the Identify 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 Identify 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 Identify 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 Identify 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 Identify 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 Identify debugger user interface to facilitate state-machine development and understanding (see [State-Machine Editor, on page 167](#)). 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 Identify debugger `statemachine` command are:

- **statemachine addtrans** **-from** *state* [**-to** *state*] [**-cond** "*equation|triggerInID*"] [**-cntval** *integer*] [**-cnten**] [**-trigger**]
- **statemachine clear** (**-all**|*state* [*state* ...])
- **statemachine info** [**-raw**] (**-all**|*state* [*state* ...])

Subcommand `statemachine addtrans`

The Identify 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|triggerInID*" – specifies the condition or external trigger input under which the transition is to be taken. The default is “true” (i.e., the transition is taken regardless of input data; see below for more details).
- **-cntval** *integer* – specifies that if this transition is taken, the counter is loaded with the given value. Only valid when a counter is instrumented.
- **-cnten** – when this flag is given, the counter is decremented by 1 during this transition. Only valid when a counter is instrumented.
- **-trigger** – when this flag is given, a trigger occurs during this transition.

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

Conditions

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

- **c0, . . . 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 Identify 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 Identify 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 Identify 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 *identify/InstallDir/share/contrib/syn_trigger_utils.tcl* which is loaded into the Identify 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 Identify 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 Identify 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 Identify instrumentor (cross triggering can be selectively disabled in the Identify debugger). To enable a destination IICE unit to accept a trigger from a source IICE unit, enter the following command in the Identify 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 watch-points. Concurrently run these units either by selecting the individual IICE units and clicking the RUN button in the Identify debugger project view or by entering one of the following commands in the Identify 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 Identify debugger cross-trigger commands are available:

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

```
iice controller -crosstriggermode DISABLED
```

- The following Identify 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 Identify 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 Identify 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 Identify debugger command uses a single source IICE unit (*iiceID*), and the second Identify 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 Identify debugger `statemachine` command sequences are available in the Identify debugger console window:

- The following Identify 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 Identify 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 Identify 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 Identify 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"
```

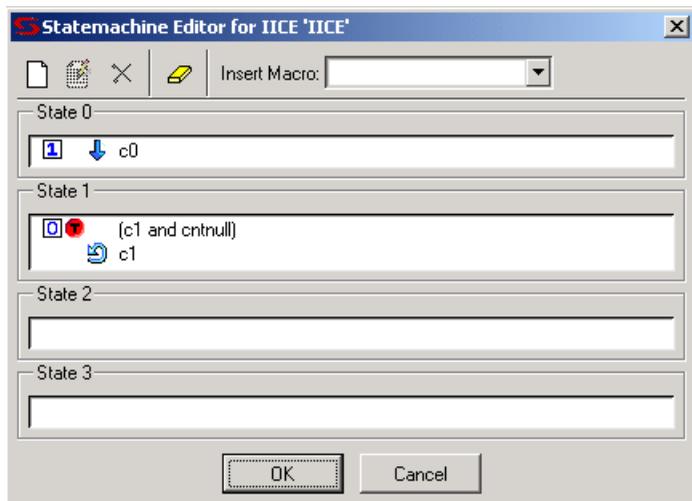
The incorporation of a counter in the state-machine configuration is similar to the use of a counter in non-cross trigger mode for a state machine.

State-Machine Editor

The Identify 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 Identify instrumentor.



To bring up the state-machine editor in the Identify debugger, click the Configure StateMachine Trigger icon in the Identify debugger toolbar. Note that the icon will be grayed out if state-machine triggering was not enabled in the Identify 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.



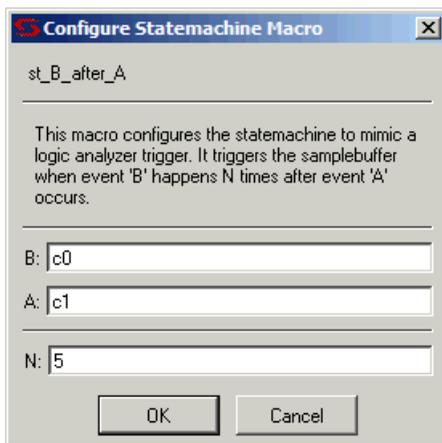
Each state is defined in an individual entry field. Within each entry, you can add multiple definitions for transitioning from that state. Each transition includes either one or two actions and a condition. The actions and conditions are defined in the following tables.

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 162)

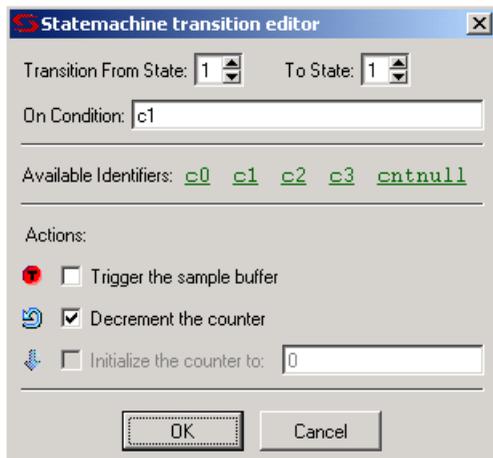
To use the dialog box:

- 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 Identify 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$
```

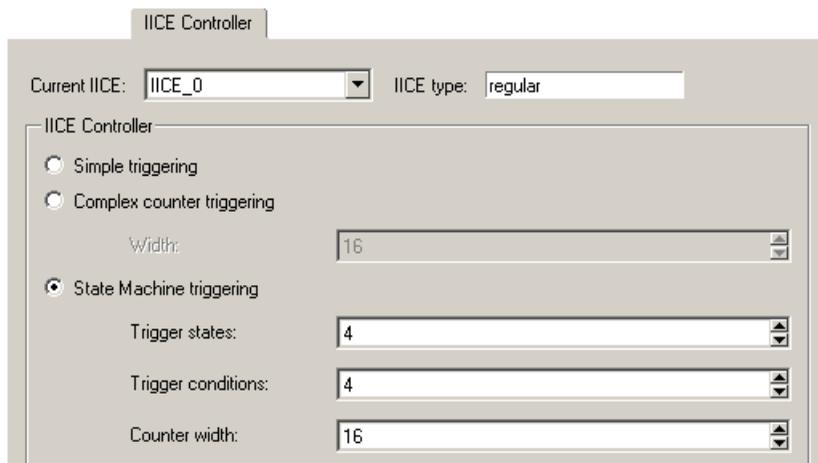
State-Machine Examples

The Identify 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 Identify 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 Identify 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. In the Identify instrumentor graphical user interface, select Actions->Configure IICE from the top menu bar or click the IICE icon.
2. From the Identify 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.



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

```

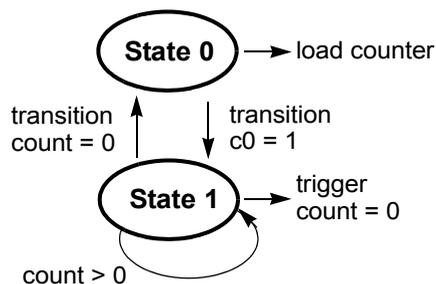
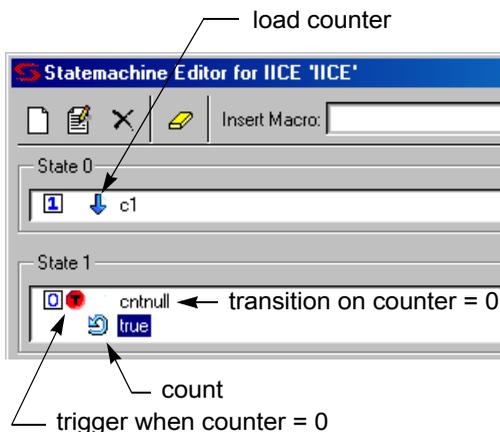
statemachine addtrans -from 0 -to 1 -cond c0 -cntval 7 -trigger
statemachine addtrans -from 1 -to 0 -cond "cntnull"
statemachine addtrans -from 1 -to 1 -cnten -trigger

```

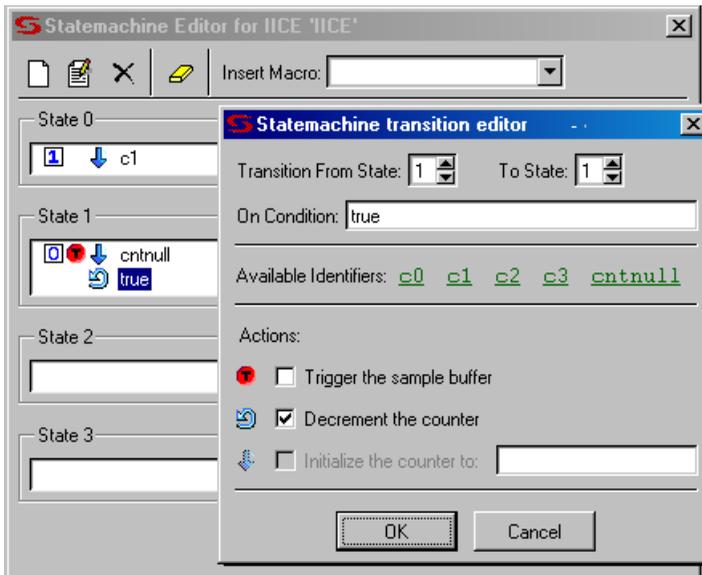
Note that in the last Identify debugger `statemachine` command, the `-to 1` can be omitted (unnecessary because there is no change in state) and that because the `-from` states are the same in the second and third commands, execution falls through to the third command when the second condition is not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

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

The state-machine editor in the Identify 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 Identify 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 Identify instrumentor and Identify debugger console windows. Some of the example commands are as follows:

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

```
statemachine clear -iice all
```

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

```
iice controller complex
```

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

```
iice controller -counterwidth 8
```

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

```
iice controller -crosstrigger 1
```

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

```
iice sampler -depth 2048
```

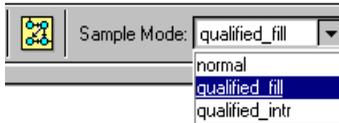
Note that the only option for buffer type is `internal_memory`.

Qualified Sampling

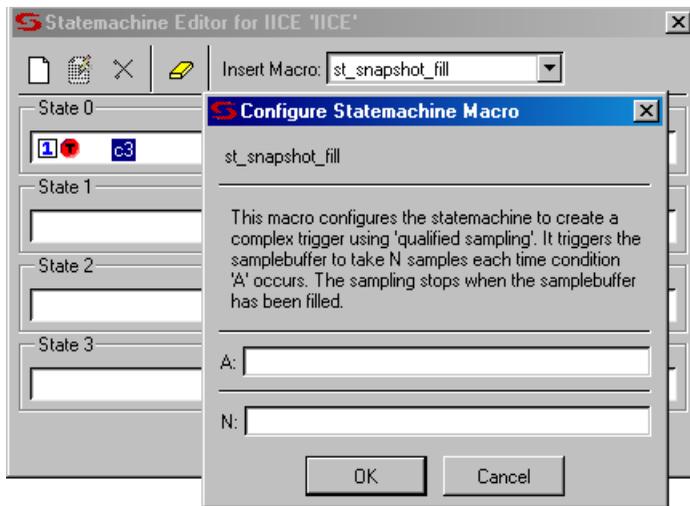
During qualified sampling, data is sampled on every clock. The following example uses qualified sampling to examine the data for a given number of clock cycles. To create a complex trigger event to perform qualified sampling:

1. From the Configure IICE dialog box in the Identify instrumentor GUI, 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.
2. Select the IICE Sampler tab and enable the Allow qualified sampling check box.

- From the Identify debugger GUI, select `qualified_fill` from the Sample Mode drop-down menu.



- From the Identify debugger GUI, click on the adjacent Configure Statemachine Trigger icon and define the state-machine trigger event.
- From the Identify 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 Identify 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 Identify 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 Identify debugger Tcl commands. The following Identify debugger example command sequence samples the data every N cycles beginning with the first trigger event.

```
iice sampler -samplemode qualified_fill
statemachine clear -iice IICE -all
statemachine addtrans -iice IICE -from 0 -to 1
    -cond "true" -cntval 0

statemachine addtrans -iice IICE -from 1 -to 2
    -cond "c0" -cntval 15 -trigger
statemachine addtrans -iice IICE -from 2 -to 2
    -cond "! cntnull" -cnten
statemachine addtrans -iice IICE -from 2 -to 2
    -cond "cntnull" -cntval 15 -trigger
```

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 Identify debugger `remote_trigger` command (see the command description in the *Reference Manual*).

As an example, the Identify 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 11

Connecting to the Target System

This chapter describes methods to connect the Identify 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 Identify 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](#)
- [HAPS Board Bring-up Utility](#)

Basic Communication Connection

The components that make up the debugging system are:

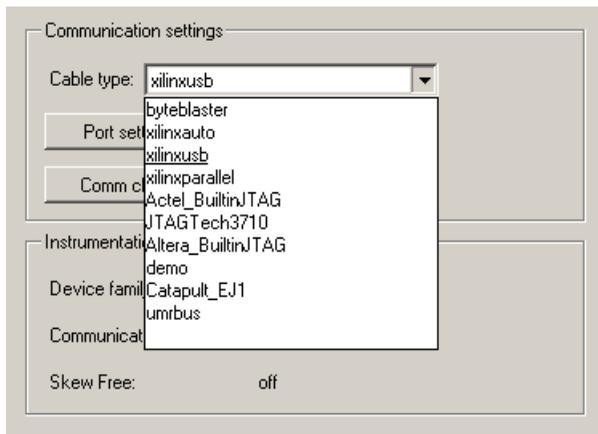
- The host machine running the Identify tool set with a loaded project.
- The communication cable connecting the host machine to the programmable device.
- The programmable device or devices loaded with the instrumented version of the design to be debugged.

Identify Debugger Communications Settings

Identify 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 Identify debugger project window (see following figure).



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

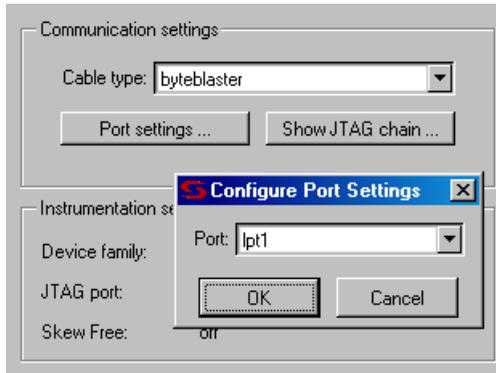
Cable Type Setting	Compatible Hardware Cables
xilinxparallel	Xilinx Parallel III and Xilinx Parallel IV
xilinxusb	Xilinx USB (Windows only)
byteblaster (soft JTAG port)	Altera ByteBlaster and ByteBlaster MV
Microsemi_BuiltinJTAG	Microsemi FlashPro, FlashProLite, or FlashPro3
Altera_BuiltinJTAG (builtin JTAG port)	Altera MasterBlaster (parallel, serial, or USB) or Altera USBBlaster

If you are using the command interface, set the `com` command's `cabletype` option to `xilinxparallel`, `xilinxusb`, `xilinxauto`, `byteblaster`, `Microsemi_BuiltinJTAG`, `JTAGTech3710`, 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.

A `umrbus` setting is also available for establishing communications between HAPS hardware and the Identify debugger (see [UMRBus Communications Interface, on page 203](#)).

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).

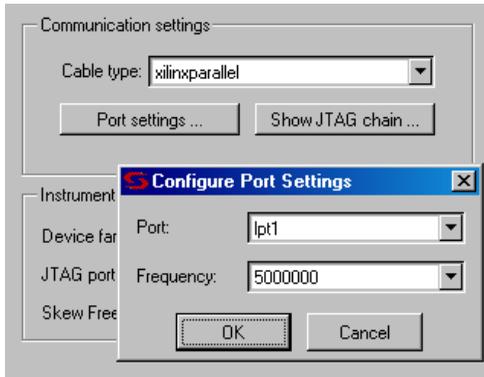


If you are using the command interface, set the `com` command's `cableoptions` `byteblaster_port` option to 1 (lpt1), 2 (lpt2), 3 (lpt3), or 4 (lpt4). Different computers have their lpt ports defined for different address ranges so the port you use depends on how your computer is configured.

The Identify debugger uses the “standard” I/O port definitions: lpt1: 0x378-0x37B, lpt2: 0x278-0x27B, lpt3: 0x3BC-0x3BF, and lpt4: 0x288-0x28B if it cannot determine the proper definitions from the operating system. If the hardware address for your parallel port does not match the addresses for lpt1 through lpt4, you can use the `setsys` `set` command variable `lpt_address` to set the hardware port address (for example, `setsys set lpt_address 0x0378` defines port lpt1).

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).

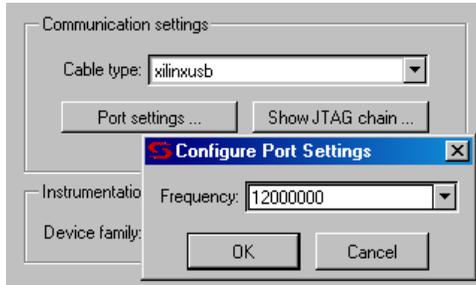


If you are using the command interface, set the `com` command's port cable options `xilinxparallel_port` option to 1 (lpt1), 2 (lpt2), 3 (lpt3), or 4 (lpt4) and set the `xilinxparallel_speed` option to 5000000 (5MHz), 2500000 (2.5 MHz), or 200000 (200kHz). Note that different computers have their lpt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

The Identify debugger uses the “standard” I/O port definitions: lpt1: 0x378-0x37B, lpt2: 0x278-0x27B, lpt3: 0x3BC-0x3BF, and lpt4: 0x288-0x28B if it cannot determine the proper definitions from the operating system. If the hardware address for your parallel port does not match the addresses for lpt1 through lpt4, you can use the `setsys` set command variable `lpt_address` to set the hardware port address (for example, `setsys set lpt_address 0x0378` defines port lpt1).

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).

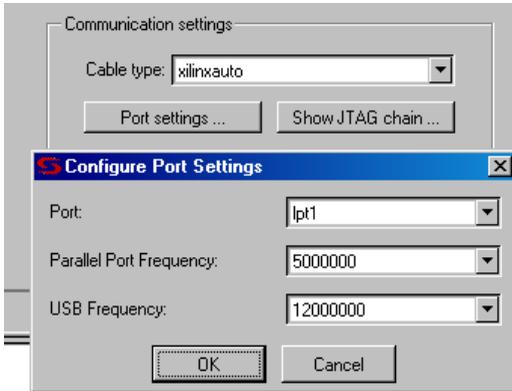


Note: The Xilinx USB cable is only supported on the Windows platform. Using the older Xilinx USB black cable with a HAPS-70 system limits the communication frequency. The newer Xilinx USB red cable does not have this limitation.

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

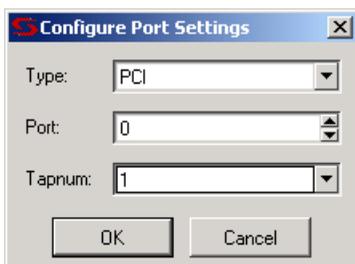
Selecting the Xilinxauto cable type allows the Identify debugger to dynamically select the appropriate Xilinx cable (parallel or USB) for the hardware configuration. From the project window, click the Port Settings button to display the Configure Port Settings dialog box and select the appropriate parallel 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 183](#) and [Xilinx USB Cable Setting, on page 184](#).

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).

Demo Cable Settings

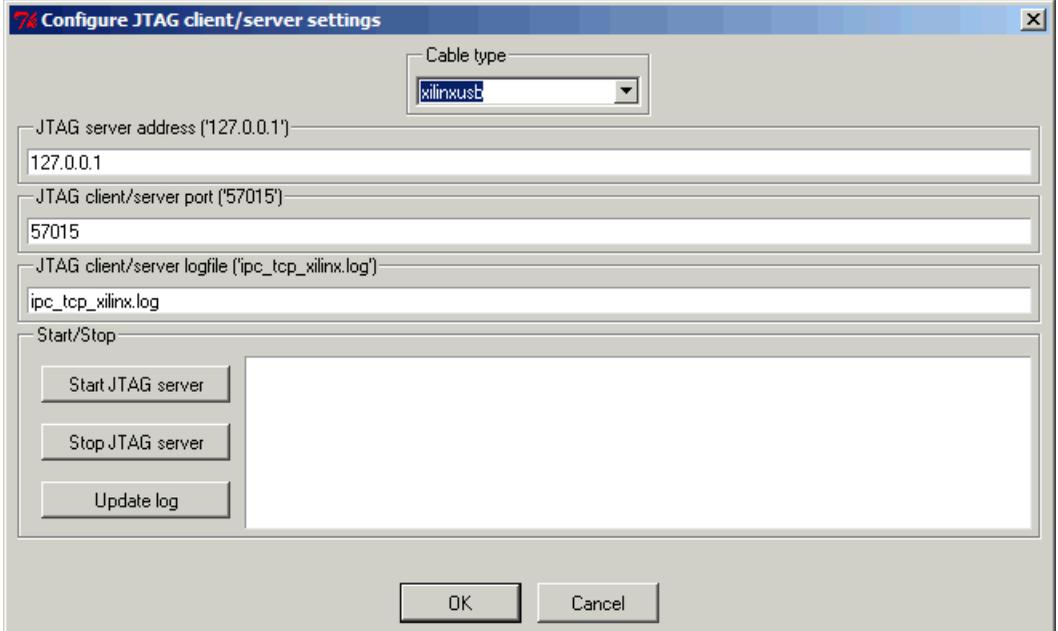
The Port Settings button is disabled when the demo cable is selected.

Identify Debugger Configuration

All parts of the debugging system must be configured correctly to make a successful connection between the Identify debugger and the instrumented device or devices through the cable. In addition to selecting the cable type and port parameters described in [Identify Debugger Communications Settings, on page 180](#), the following additional requirements must be met to ensure proper communications.

JTAG Client-Server Configuration

The client-server configuration is set from a dialog box available by selecting Options->Configure JTAG server in the Identify 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 Identify 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 188](#)).



In the dialog box:

Cable type – the type of interface cable (see [Cable Type](#), on page 180).

JTAG server address – the address of the server. The address localhost is used when the Identify 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 Identify debugger is run from a different machine.

JTAG client/server port – the port number of the server. For Xilinx cable types, the default port number is 57015; for Microsemi cable types, the default port number is 58015. For Altera devices, client-server port configuration is part of the Altera driver and the setting is ignored. Change the server port setting when there is a conflict with another tool on the machine.

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

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

Client-Server Configuration for Remote Debugging

The Identify debugger uses a client-server architecture to communicate with FPGAs over the JTAG interface. Client-server architecture lets you work remotely with the Identify debugger using Ethernet as the backbone for client-server communication. The Identify 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 Identify debugger and connect to the server is termed the client. You can use the Configure JTAG 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, Configure JTAG server from the Options drop-down menu.
 - Specify the server address, port number, and log file name in the Configure JTAG 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 JTAG client-server parameters does not start the server.
 - To start the server, select the Start JTAG 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 or acteljtag 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, Configure JTAG server from the Options drop-down menu.
 - Specify the server address, port number, and log file name in the Configure JTAG 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 JTAG server.

The following is the syntax for the equivalent TCL command to configure the JTAG server:

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

To view the existing JTAG 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 Identify 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 Identify debugger uses the “standard” I/O port definitions: `lpt1: 0x378-0x37B`, `lpt2: 0x278-0x27B`, `lpt3: 0x3BC-0x3BF`, and `lpt4: 0x288-0x28B` if it cannot determine the proper definitions from the operating system. If the hardware address for your parallel port does not match the addresses for `lpt1` through `lpt4`, you can use the `setsys set` command variable `lpt_address` to set the hardware port address (for example, `setsys set lpt_address 0x0378` defines port `lpt1`).

- 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.

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 Identify debugger first attempts to detect the devices in the scan chain. If auto-detection is unsuccessful, describe the device chain to the Identify debugger using the `chain` command.

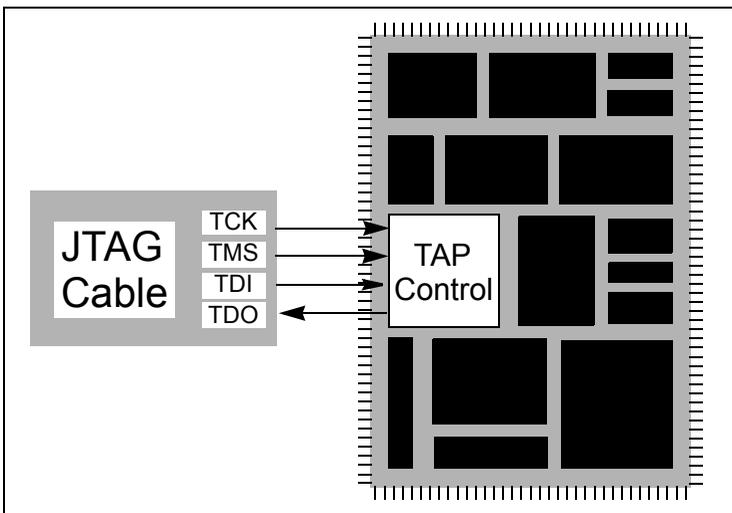
Chip Programming

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

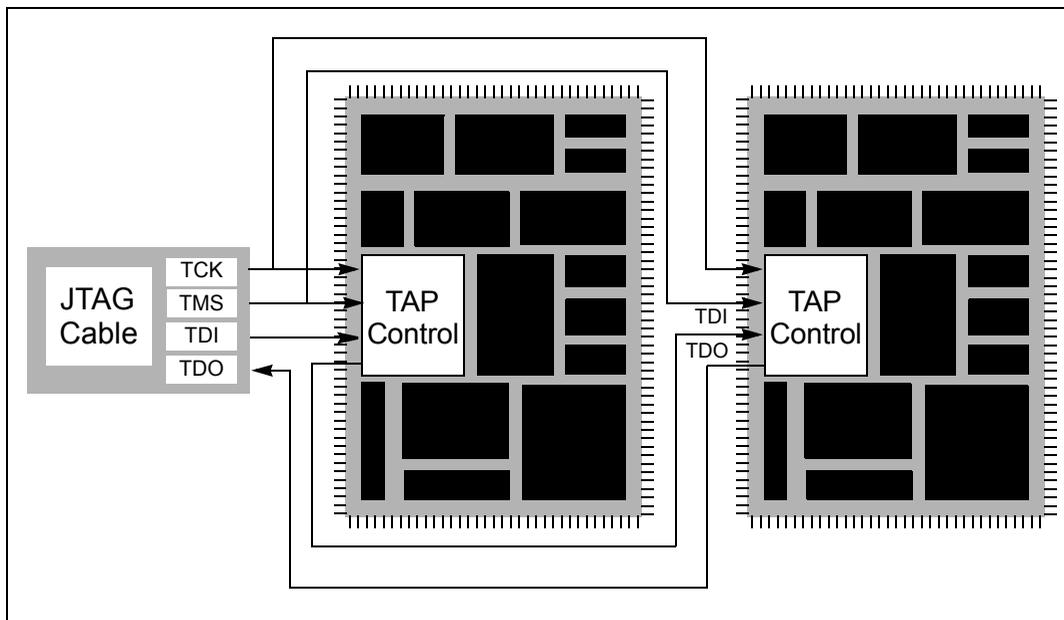
JTAG Communication

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 Identify tool set 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 Identify tool set 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

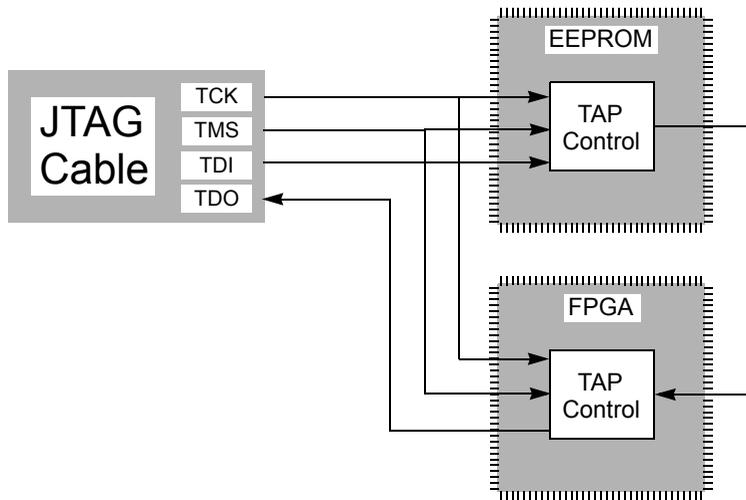
Some programmable device families employ a built-in TAP controller as a means for device configuration. In most cases, the IICE also can be configured to use this built-in TAP controller. Using this TAP controller saves the user logic necessary to implement the controller and also saves four I/O pins.

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

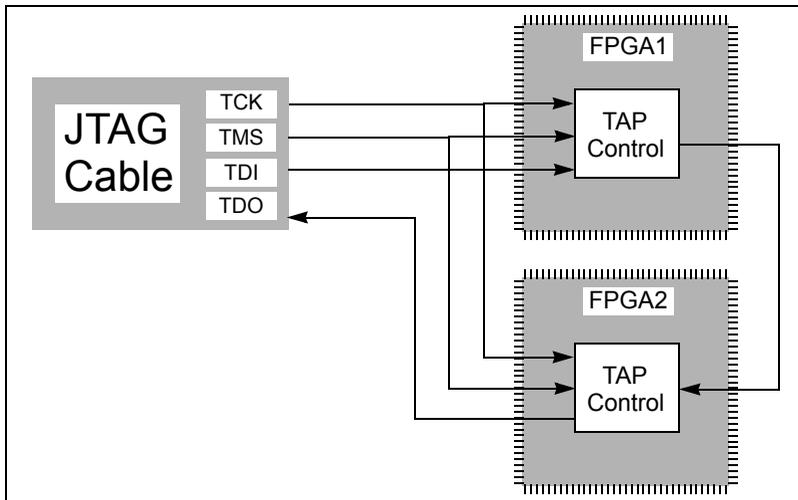
HAPS boards and other boards that connect the built-in JTAG port directly to four header pins on the board allow the JTAG cable to simply be connected directly to the header pins. This configuration works for both directly connected devices and serially chained devices.

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 Identify debugger and works just like any other serially connected chain.

Similarly, when using the Identify tool set with the Certify tool and 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 Identify 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 Identify 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 Identify 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.

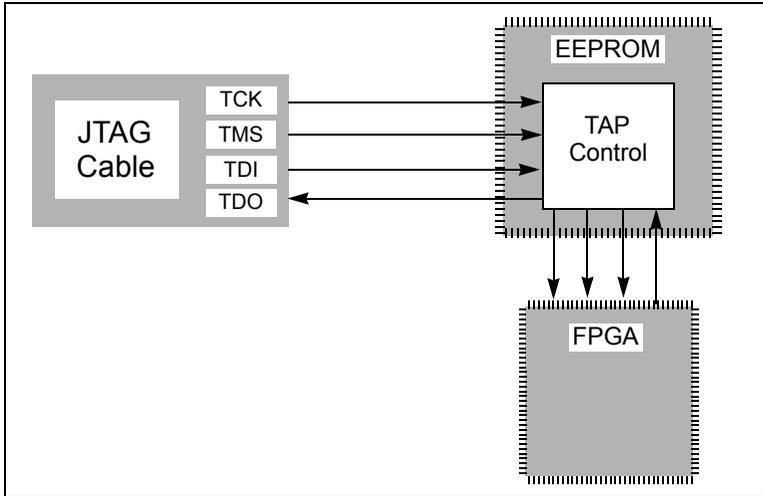
JTAG Registers

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.

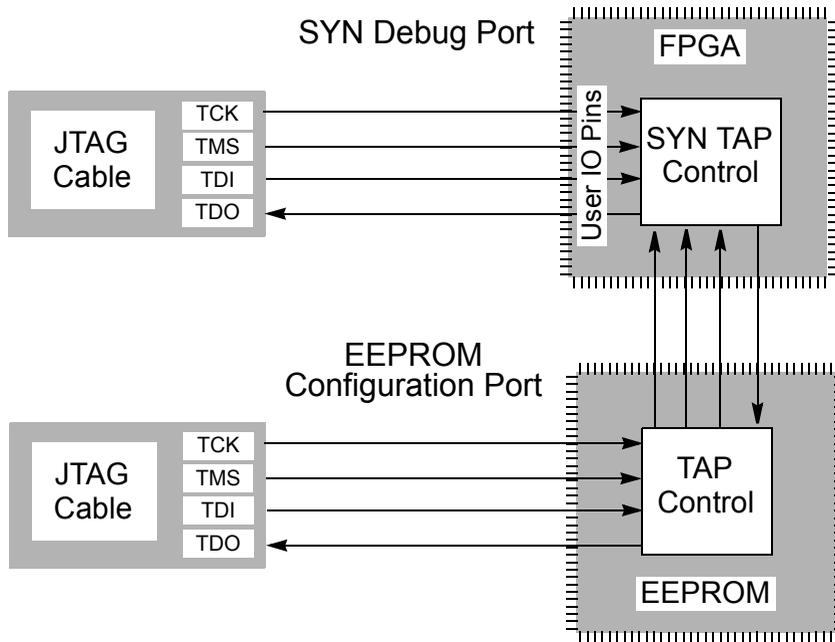
The Identify instrumentor requires two boundary-scan registers. For Virtex-4, Virtex-5, Virtex-6, and Virtex-7 applications, you can specify which two registers are dedicated to Identify to avoid any contention among the available registers for user applications (two BSCAN_VIRTEX* cells cannot share the same address). By default, registers USER3 and USER4 are reserved for Identify. To change the default register settings, use the `xilinxjtagaddr1` and `xilinxjtagaddr2` options to the device command (see [device](#), on [page 39](#) of the *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 Identify 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 Identify 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 Identify 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 Identify 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.

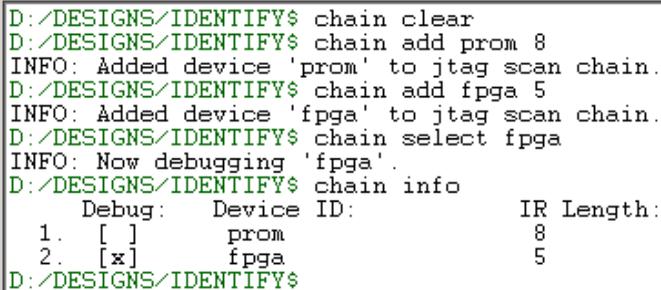
Note: The names PROM and FPGA have no meaning to the Identify debugger – they simply are used for convenience. The two devices could be named device1 and device2, and the debugger would function exactly the same.

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 Identify 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 Identify debugger performs a number of diagnostic communication tests. The first time the Identify 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 Identify 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 Identify 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.

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

- **ERROR: Cannot find valid instrumented design.**
The Identify 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 Identify debugger verified that the chip is instrumented but the instrumentation does not match the project that was loaded into the Identify debugger.

JTAG Chain Tests

The Identify 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 Identify 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 Identify 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 Identify 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 currently is available only for HAPS-60 series and HAPS-70 series systems and requires a *HAPS UMRBus Interface Kit* to replace the Xilinx USB cable. The UMRBus supports only the `internal_memory` buffer type and not `happssram`. To enable the use of the UMRBus:

- In the Identify instrumentor, select `umrbus` from the Communication port drop-down menu in the project view or set the device `jtagport` option to `umrbus` in the console window.
- In the Identify debugger, select `umrbus` from the Cable type drop-down menu in the project view or set the `com_cabletype` option to `umrbus` in the console window.

HAPS Board Bring-up Utility

Identify users with a HAPS-60 or HAPS-70 series system can use the HAPS board bring-up utility to help define and verify their board configuration.

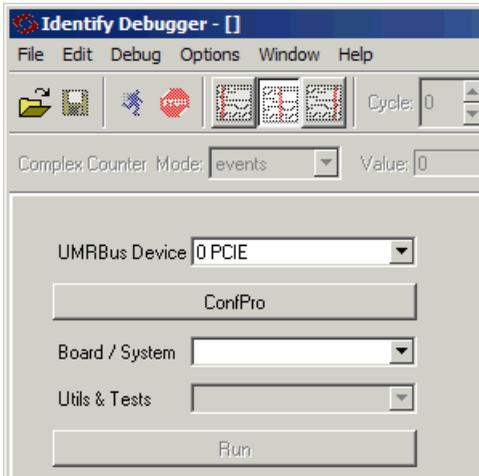
Before you can use the HAPS board bring-up utility, the following software must be installed:

- Current version of the Identify tool set
- ConfPro GUI

The board bring-up utility is launched directly from the command prompt using the `-board_bringup` option to the `identify_debugger` command:

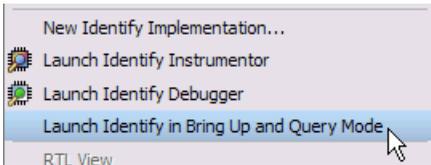
```
identify_debugger -board_bringup
```

The above command opens a special Identify debugger window with the board bring-up utility GUI displayed in the upper left corner of the window as shown below.



The board bring-up utility also can be launched directly from the Certify GUI which requires both a project and an Identify implementation to be defined for that project. 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).

To launch the HAPS Board Bring-up Utility from the Certify GUI, highlight the Identify implementation in the Certify GUI and, with the right mouse button, select Launch Identify in Board Bring Up and Query Mode to display the board bring-up utility GUI in the Identify debugger.



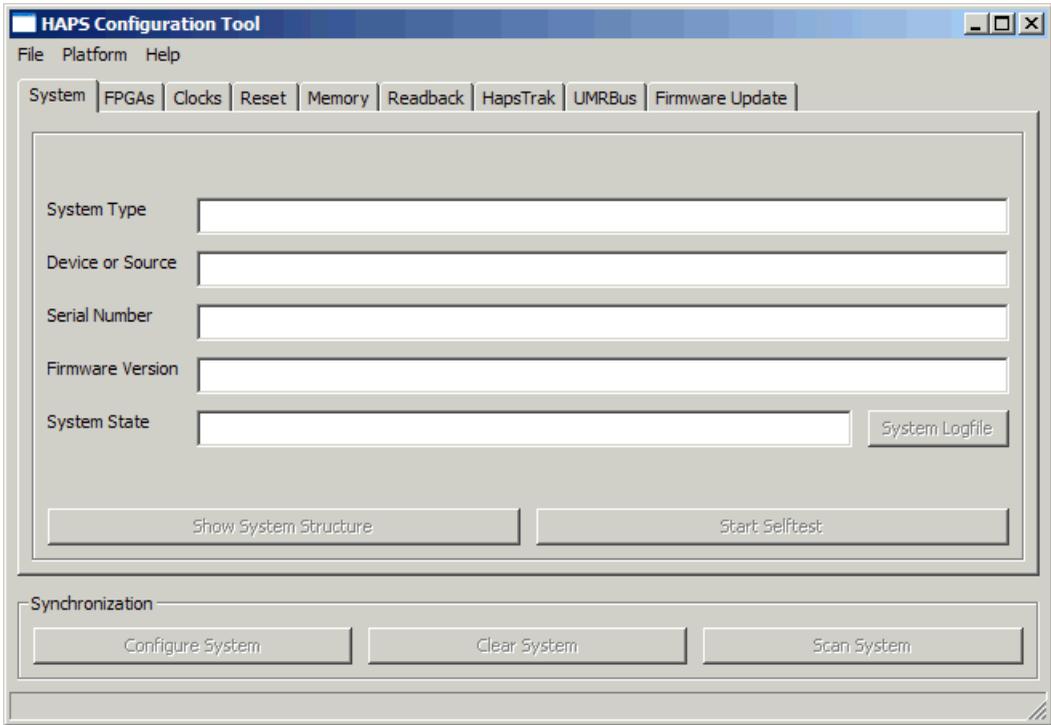
Setting Initial Values

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

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	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 205 .
Board/System	Selects the type of board/system connected to the host. The allowed selections are haps6x (HAPS-60 system) and haps7x (HAPS-70 system) which 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 Utility and Board-Test Commands, on page 207
Run	Runs the selected query/test to be executed.

ConfPro GUI

The ConfPro GUI is launched from the board bring-up utility in the Identify 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 Identify debugger menu.



Utility and Board-Test Commands

Sets of utility and board test commands are available 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 individual commands have Tcl equivalents that can be run from the command prompt (see [haps](#), on [page 45](#) of the reference manual). The following sections describe the individual utility and test commands.

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.

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.

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.

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.

restart

Restarts the board.

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.

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).

clock_check

Reports the clock frequency of each GCLK output to allow all of the GCLK frequencies to be verified. As an option, the location of a Tcl script can be specified to execute any of the individual haps commands.

con_speed

Verifies the connectivity between HapsTrak connectors as well as the speed at which HSTDM can run. The test speed is selected from the Speed drop-down menu. The Mode drop-down menu sets the run mode. The default is fast mode. When Mode is set to sweep, the test sweeps every channel of the connection which can require up to four hours to complete.

umr_check

Verifies the basic functionality of the UMRBus for the FPGA specified in the FPGA ID field. The GCLK1 frequency is specified in the Frequency field and unless specified otherwise, is in kHz (the default is 140000 kHz).

self_test

Replaces the traditional, STB2 test card self test. The self_test is only available with haps6x board/system selection.

Index

A

- activations
 - auto-saving [123](#)
 - loading [122](#)
 - saving [121](#)
- always-armed triggering [40](#)
- asynchronous clocks [165](#)

B

- bitfile [146](#)
- bitgen command (Xilinx) [147](#)
- black boxes [68](#)
- blocks
 - controller [21](#)
 - JTAG communication [149](#)
 - probe [21](#)
 - sampling [152](#)
- board query [204](#)
- board-test commands [207](#)
- board-utility commands [207](#)
- boundary-scan registers [197](#)
- breakpoint icon
 - color coding [85](#)
- breakpoints
 - activating [108](#)
 - combined with watchpoints [152](#)
 - finding [94](#)
 - folded [111](#)
 - in folded hierarchy [84](#)
 - instance selection [85](#)
 - listing all [94](#)
 - listing available [94](#)
 - listing instrumented [94](#)
 - multiple [151](#)
 - selecting [84](#)
- bring-up utility [204](#)
- buckets
 - sample and trigger [146](#)

- buffer type
 - hapssram [48](#)
- buses
 - instrumenting partial [75](#)
- Byteblaster cable settings [182](#)

C

- cable compatibility [181](#)
- cable type [180](#)
- cable type settings
 - Byteblaster [182](#)
 - JTAGTech3710 [185](#)
 - Microsemi [186](#)
 - Xilinx parallel [183](#)
 - Xilinx USB [184](#)
 - Xilinxauto [184](#)
- cables
 - connection [190](#)
- client-server configuration [186](#)
- clocks
 - asynchronous [165](#)
 - edge selection [42](#)
 - sample [41](#)
 - SRAM [50](#)
- commands
 - bitgen (Xilinx) [147](#)
- communication cable
 - settings [30](#)
- communications settings [180](#)
- complex counter [153](#)
 - cycles mode [155](#)
 - disabling [155](#)
 - events mode [155](#)
 - modes [153](#)
 - pulsewidth mode [155](#)
 - size [153](#)
 - watchdog mode [155](#)
- complex triggering [44](#)
- condition operators [162](#)

- Configure IICE dialog box [38](#), [137](#)
 - IICE Controller tab [43](#)
 - IICE Sampler tab [38](#)
- ConfPro [205](#)
- ConfPro installation [206](#)
- console window [70](#), [102](#)
 - operations [141](#)
- console window operations [96](#)
- controller block [21](#)
- convenience functions [164](#)
- conventions
 - design hierarchy [14](#)
 - file system [13](#)
 - syntax [12](#)
 - text [12](#)
 - tool [13](#)
- cross triggering [123](#), [128](#), [165](#)
 - commands [165](#)
 - enabling [165](#)
 - state machine commands [166](#)
- cycles mode
 - complex counter [155](#)

D

- data compression [114](#)
 - masking [115](#)
- Debugger tool
 - invoking [98](#)
- debugger tool
 - opening projects [29](#)
- debugging
 - on separate machines [127](#)
- design hierarchy conventions [14](#)
- designs
 - writing instrumented [90](#)
- devices
 - supported families [35](#)
- dialog boxes
 - Configure IICE [38](#), [137](#)
- directories
 - instrumentation [92](#)

E

- encrypting source files [91](#)
- environment variables
 - PAR_BELDLYRPT [144](#)

- events mode
 - complex counter [155](#)
- external memory
 - self test [52](#)
- external sample memory [47](#)

F

- file system conventions [13](#)
- files
 - bitfile [146](#)
 - encrypting source [91](#)
 - idc [82](#)
 - IICE core [92](#)
 - last_run.adb [123](#)
 - ncd [145](#)
 - project [24](#), [26](#), [27](#)
 - script [72](#), [105](#)
 - syn_trigger_utils.tcl [164](#)
- folded breakpoints [111](#)
- folded hierarchy [80](#)
- folded signals [119](#)
- folded watchpoints [110](#)

H

- HAPS
 - board bring-up [204](#)
- HAPS deep trace debug [47](#)
- hapssram buffer type [48](#)
- hardware
 - skew-resistant [36](#)
- HDL source
 - including in project [91](#)
- hierarchy
 - folded [80](#)
- hierarchy browser
 - popup menu [67](#)
- hierarchy browser window [66](#)
- hierarchy separator [15](#)

I

- idc file
 - editing [82](#)
- identification register [202](#)
- IICE
 - configuration [33](#)

- cross triggering [165](#)
- JTAG connection [193](#)
- operation [21](#)
- technology settings [35](#)
- IICE Controller tab [43](#)
- IICE core file
 - compiling [92](#)
- IICE parameters
 - buffer type [39](#)
 - common [35](#)
 - individual [38](#), [137](#)
 - JTAG port [36](#)
- IICE Sampler tab [38](#)
- IICE selection
 - multi-IICE [39](#), [43](#)
- IICE settings
 - sample clock [41](#)
 - sample depth [40](#)
- IICE units
 - cross triggering [123](#)
- incremental flow [143](#)
 - restrictions [144](#)
- instances
 - finding [94](#)
- instrumentation
 - partial records [77](#)
- instrumentation directory [92](#)
- instrumenting partial buses [75](#)

J

- JTAG
 - chain tests [202](#)
 - communication [191](#)
 - communication block [149](#)
 - communication test [201](#)
 - connections [197](#)
 - debugging [201](#)
 - direct connection [195](#)
 - serial connection [196](#)
- JTAG chain
 - settings [31](#)
- JTAG port
 - IICE parameter [36](#)
- JTAG registers [197](#)
- JTAGTech3710 cable settings [185](#)

L

- last_run.adb file [123](#)

- limitations
 - Verilog instrumentation [58](#), [61](#)
 - VHDL instrumentation [56](#)

M

- macros
 - st_snapshot_fill [175](#)
 - st_snapshot_intr [176](#)
- Microsemil
 - cable type settings [186](#)
- mixed language considerations [55](#)
- modes
 - cross triggering [124](#)
- multi-IICE
 - tabs [38](#), [137](#)
- multiple signal values [119](#), [120](#)
- multiplexed groups
 - assigning [79](#)
 - selecting [109](#)

N

- ncd file [145](#)

O

- objects
 - finding [94](#)
- operators
 - condition [162](#)
- original source
 - including [91](#)
- original source files
 - searchpath [127](#)
- original sources [127](#)

P

- PAR_BELDLYRPT variable [144](#)
- parameterized modules
 - instrumenting [82](#)
- parameters
 - IICE [33](#)
 - IICE common [35](#)
- partial buses
 - instrumenting [75](#)
- passwords
 - encryption/decryption [91](#)
- path names [15](#)

- path separator [13](#)
 - .prj files [24](#), [26](#)
 - probe block [21](#)
 - contents [22](#)
 - project files [27](#)
 - projects
 - importing Synplify [24](#), [26](#)
 - instrumenting [27](#)
 - opening [26](#)
 - opening in debugger [29](#)
 - saving [31](#)
 - pulsewidth mode
 - complex counter [155](#)
- ## Q
- qualified sampling [40](#), [174](#)
- ## R
- radix
 - sampled data [117](#)
 - RAM resources [39](#), [153](#)
 - records
 - partially instrumented [77](#)
 - registers
 - boundary scan [197](#)
 - remote triggering [176](#)
 - resource estimation [70](#)
 - run command [113](#)
- ## S
- sample and trigger buckets [146](#)
 - sample buffer [117](#)
 - trigger position [115](#)
 - sample clock [41](#)
 - sample memory [47](#)
 - sample modes [175](#)
 - sampled data
 - changing radix [117](#)
 - compressing [114](#)
 - display controls [117](#)
 - masking [115](#)
 - sampling
 - in folded hierarchy [80](#)
 - qualified [40](#)
 - sampling block [152](#)
 - sampling signals [72](#), [80](#), [84](#), [86](#), [89](#), [90](#), [93](#), [94](#),
[101](#)
 - saving a project [31](#)
 - script file [72](#)
 - script files [105](#)
 - searches
 - objects [94](#)
 - separator
 - hierarchy [15](#)
 - settings
 - cable [30](#)
 - IICE technology [35](#)
 - JTAG chain [31](#)
 - sample clock [41](#)
 - sample depth [40](#)
 - signal values
 - displaying multiple [119](#), [120](#)
 - signals
 - disabling sampling [74](#)
 - exporting trigger [45](#)
 - finding [94](#)
 - folded [119](#)
 - instance selection [81](#)
 - listing all [93](#)
 - listing available [93](#), [101](#)
 - listing instrumented [93](#), [94](#), [101](#)
 - multiply instrumented [119](#), [120](#)
 - partially instrumented [120](#)
 - replacing [143](#)
 - sampling selection [72](#), [80](#), [84](#), [86](#), [89](#), [90](#), [93](#),
[94](#), [101](#)
 - status [160](#)
 - simple triggering [44](#)
 - skew-resistant hardware [36](#)
 - source files
 - copying [127](#)
 - encrypting [91](#)
 - SRAM clocks [50](#)
 - st_snapshot_fill macro [175](#)
 - st_snapshot_intr macro [176](#)
 - state machines
 - transitions [161](#)
 - triggering [157](#), [160](#)
 - statemachine command [160](#)
 - state-machine editor [167](#)
 - state-machine triggering [44](#)
 - status reporting [160](#)
 - stop command [116](#), [160](#)
 - syn_trigger_utils.tcl file [164](#)
 - syntax conventions [12](#)

synthesizing designs 92

T

TAP controller 193

technology settings

IICE 35

text conventions 12

tool conventions 13

tool descriptions 22

tools

invoking Debugger 98

transition watchpoint 106

trigger conditions 156

trigger signal

exporting 45

triggering

advance mode 157

always-armed 40

between IICEs 165

complex 44

modes 156

remote 176

simple 44

state machine 44, 157, 160

triggers

complex 153

U

UMRBus 203

V

value watchpoint 106

variables

PAR_BELDLYRPT 144

Verdi waveform viewer 136

Verilog

hierarchy 15

instrumentation limitations 58, 61

VHDL

hierarchy 15

instrumentation limitations 56

W

watch command 160

watch icon

color coding 81

watchdog mode

complex counter 155

watchpoints 151

activating 105, 108

combined with breakpoints 152

deactivating 107

folded 110

hexadecimal values 107

listing 125

multiple 152

transition 106

value 106

waveform display 134

waveform viewers 134

Verdi 136

wildcards

in hierarchies 16

in path names 14

windows

console 70, 102

hierarchy browser 66

X

Xilinx parallel cable settings 183

Xilinx USB cable settings 184

Xilinxauto cable settings 184

