

Verification Continuum™

HAPS® ProtoCompiler

Instrumentor User Guide

April 2022

SYNOPSYS®

solvnetplus.synopsys.com

Synopsys Confidential Information

Copyright Notice and Proprietary Information

© 2022 Synopsys, Inc. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

Free and Open-Source Licensing Notices

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

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Third-Party Links

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

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

April 2022

Synopsys Statement on Inclusivity and Diversity

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

Contents

Chapter 1: Overview of the Instrumentor

The Instrumentor Design Flow	10
Instrumentor Basics	14
Launching the Instrumentor	16

Chapter 2: Using the Instrumentor

Adding Instrumentation	20
Setting Watchpoints	20
Instrumenting Buses	24
Instrumenting Fields Within a Record or Structure	26
Assigning Signals to Groups	27
Instrumenting the Verdi Signal Database	28
Instrumenting Signals Directly in the idc File	29
Adding Breakpoints	31
VHDL Instrumentation Limitations	33
Verilog Instrumentation Limitations	35
SystemVerilog Instrumentation Limitations	38
Finding Design Objects in the Instrumentor	42
Saving Instrumentation	44
Instrumenting a UC Design	46
Adding UC Instrumentation	46
Setting Instrumentation Limits for UC Designs	48
Specifying Post-Compile Instrumentation for UC Designs	48
Using SystemVerilog Assertions	53
Adding SVAs for Instrumentation	53
Detecting Assertion Failures with a HAPS System Reset	58
Debugging Designs with SVAs	59
Using \$dumpvars	61
Adding \$dumpvars for Instrumentation	62

Using Wildcard Characters in \$dumpvars	66
Specifying \$dumpvars for RTL Hierarchies	67
Exporting Instrumentation for Debug	69

Chapter 3: Setting up the IDC and IICE

Working with an IDC File	72
Creating an IDC File	72
Evaluating IDC Files	73
Using a Sample IDC for Essential Signal Analysis	75
Working with IICE Units	76
Adding IICE Units	77
Setting Common IICE Parameters	79
Setting Individual IICE Parameters	81
Setting Buffer Parameters for an IICE	82
Defining the IICE Sample Clock	83
Working with Multiple IICE Units	87
Using Multiple IICEs for Multiple Instrumentation Modes	87
Examples of \$dumpvars Labels for Multi-IICE Grouping	88
Setting Triggers in the Instrumentor	92
Setting IICE Trigger Options	93
Setting up Simple Triggers	94
Setting up Complex Counter Triggers	95
Setting up State Machine Triggers	97
Setting up an IICE for Cross-Triggering	100
Setting up RTL Cross Triggers	102
Setting up for Real-Time Debug (RTD)	104

Chapter 4: HAPS Deep Trace Debug

Setting up DTD for HAPS-100 Designs	108
Specifying Built-in Memory for DTD with HAPS-100 Designs	108
Specifying DTD Buffers for HAPS-100 Multi-FPGA Designs	110
Link Training Messages	113
Limitations to HAPS-100 DTD	114
Setting up DTD for HAPS-80 Designs	116
Setting up DTD for HAPS-70 Designs	119
Setting up DDR3 Memory for DTD with HAPS-70 Designs	119
Setting up DTD for Multi-FPGA HAPS-70 Designs	121

Chapter 5: The Instrumentor GUI

Instrumentor Windows and Views 124

Instrumentor Preferences 131

Edit IICE Settings 134

 IICE SamplerTab 134

 IICE ClockTab 136

 IICE Controller Tab 137

 IICE Options Tab 138

 RTD Tab 140

 DTD Tab 140

 MGB Tab 141

CHAPTER 1

Overview of the Instrumentor

The HAPS® ProtoCompiler instrumentor is a tool identifies signals for monitoring during debug, and this document describes how to use this tool. See the following topics for introductory information about the instrumentor.

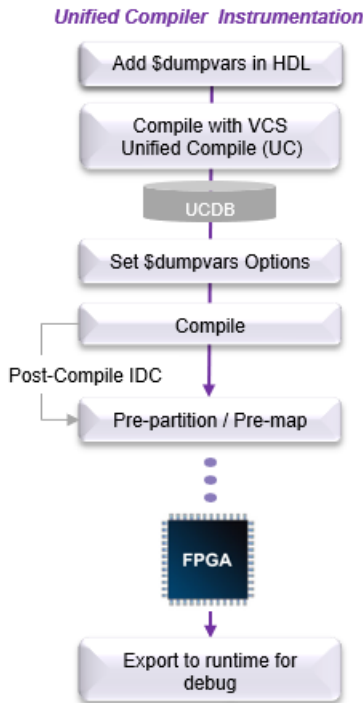
- [The Instrumentor Design Flow](#), on page 10
- [Instrumentor Basics](#), on page 14
- [Launching the Instrumentor](#), on page 16

The figure shows how the instrumentor fits into both the standard compiler flow and the unified compile (UC) flow. In the standard compile flow, you can add instrumentation directly in the HDL (pre-compile) or to a compiled design. With UC, HDL instrumentation is through \$dumpvars and is handled through VCS® compile, so there is no pre-compile instrumentation through the instrumentor. However, you can specify IDC instrumentation in a post-compile IDC file. See [Unified Compiler Instrumentation Flow, on page 11](#) and [Standard Compiler Instrumentation Flow, on page 12](#) for more details on each.

After compile, the design follows the normal flow through synthesis and place and route. When the bit files are generated and the design is programmed onto the hardware system, the instrumentation information is passed to the debugger. The design is run on the target hardware and the debugger verifies the design by monitoring the instrumented signals.

Unified Compiler Instrumentation Flow

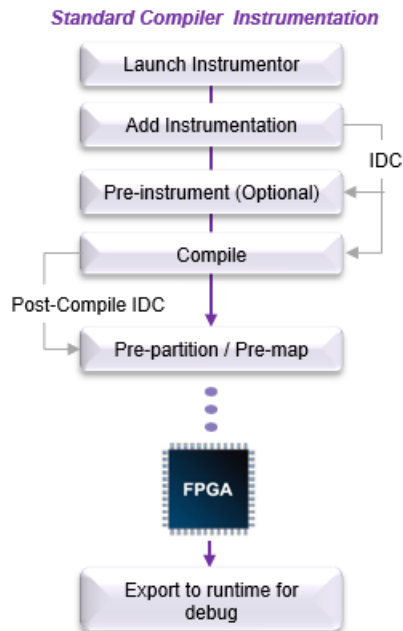
The figure shows more details of instrumentation with the UC flow, which is the recommended flow for newer HAPS technologies and going forward. The UC flow is the only flow available for HAPS-100 designs.



With the UC flow you use \$dumpvars to enter instrumentation in the HDL, and this is processed by the VCS unified compiler. The instrumentor is only used for post-compile instrumentation on an SRS (compiled) database, and does not process \$dumpvars instrumentation.

Standard Compiler Instrumentation Flow

The figure shows more details of instrumentation with the standard compiler flow. You can use the instrumentor to insert pre-compile instrumentation directly in the HDL source files, as well as for post-compile instrumentation on an SRS (compiled) database.



Instrumentor Basics

The following topics describe some basic concepts in the instrumentor:

- [The Instrumentation Process](#), on page 14
- [Signal Instrumentation: Watchpoints and Breakpoints](#), on page 14
- [The IDC File](#), on page 14
- [The Intelligent In-Circuit Emulator \(IICE\)](#), on page 15

The Instrumentation Process

Instrumentation consists of these steps:

1. Specify IICE parameters and HAPS settings.
2. Select signals to sample.
3. Select breakpoints to instrument.
4. Export the files for debug. Optionally, include the original HDL source.

Signal Instrumentation: Watchpoints and Breakpoints

You can mark signals as watchpoints and breakpoints. Inserting these probe points adds a small amount of logic to the HDL.

- A watchpoint is a signal or node that is connected to the instrumentation logic, and which can be used for sampling or as a trigger point. You can set watchpoints from the GUI or through a command. For more information, see [Setting Watchpoints, on page 20](#).
- A breakpoint is an HDL control flow statement (IF, THEN, CASE) that you identify to trigger sampling. For more information, see [Adding Breakpoints, on page 31](#).

The IDC File

Signal information is stored as constraints in an *instrumentation design constraints* (.idc) file. This is the basic instrumentation file. See [Working with an IDC File, on page 72](#) for more information.

The Intelligent In-Circuit Emulator (IICE)

The IICE™ (Intelligent In-Circuit Emulator) is a communication block that is added to the design for instrumentation and debug. Parameters for the IICE are specified in the IDC file. A design can have more than one IICE. See [*Working with IICE Units, on page 76*](#) for more information.

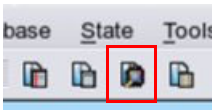
Launching the Instrumentor

The instrumentor can be run from the command line or through a graphic user interface (GUI). The instrumentor is integrated into the prototyping tool, and can be launched from within that tool.

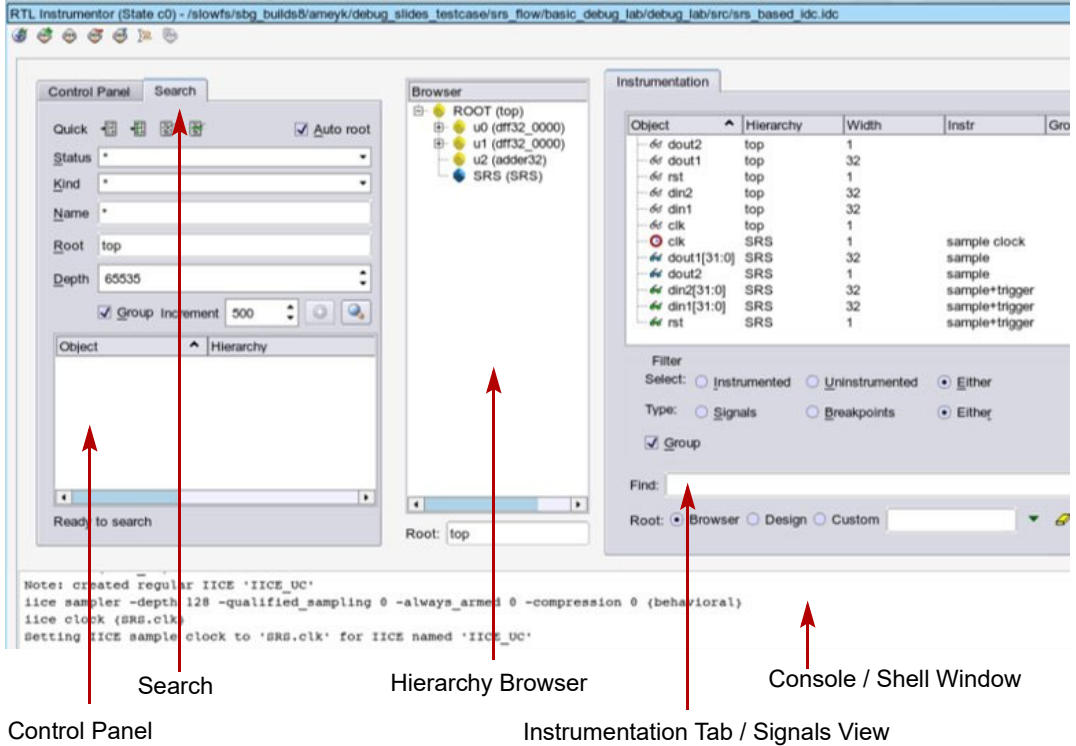
1. To launch the instrumentor from the command line, specify this command from the appropriate database state:

```
edit idc -mode gui fileName.idc
```

2. To start the instrumentor from the GUI, click the icon in the prototyping tool:



You can launch the instrumentor at different design stages. For UC designs, you can only run the instrumentor on a compiled design, but with the standard compiler flow you can run it before and after compilation.



CHAPTER 2

Using the Instrumentor

The instrumentor defines the instrumentation for the HDL design, creates the instrumented HDL design, creates the associated IICE, and creates the design database.

The remainder of this chapter describes the instrumentor environment that lets you carry out these tasks:

- [Adding Instrumentation](#), on page 20
- [Language Limitations for Instrumentation](#), on page 33
- [Finding Design Objects in the Instrumentor](#), on page 42
- [Saving Instrumentation](#), on page 44
- [Instrumenting a UC Design](#), on page 46
- [Using SystemVerilog Assertions](#), on page 53
- [Using \\$dumpvars](#), on page 61

Adding Instrumentation

The following sections describe basic instrumentor commands and procedures.

- [Setting Watchpoints](#), on page 20
- [Instrumenting Buses](#), on page 24
- [Instrumenting Fields Within a Record or Structure](#), on page 26
- [Assigning Signals to Groups](#), on page 27
- [Instrumenting the Verdi Signal Database](#), on page 28
- [Instrumenting Signals Directly in the idc File](#), on page 29
- [Adding Breakpoints](#), on page 31


For information specific to the UC flow, see these topics:

- [Instrumenting a UC Design](#), on page 46
- [Using SystemVerilog Assertions](#), on page 53
- [Using \\$dumpvars](#), on page 61
- [Specifying Post-Compile Instrumentation for UC Designs](#), on page 48

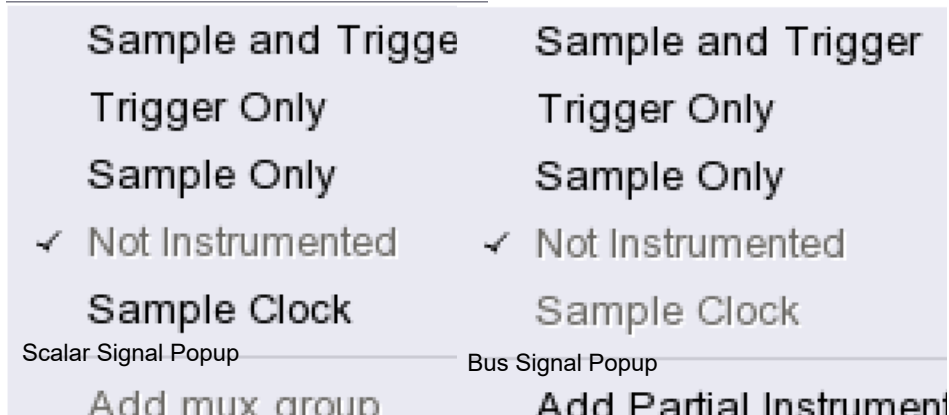
Setting Watchpoints

A watchpoint marks a signal to be sampled. With the standard compiler flow, you can instrument watchpoints in the RTL. The following procedure focuses on the procedure to set watchpoints from the RTL tab in the GUI, but you can also set them directly in the idc file with the signals add -iice command.

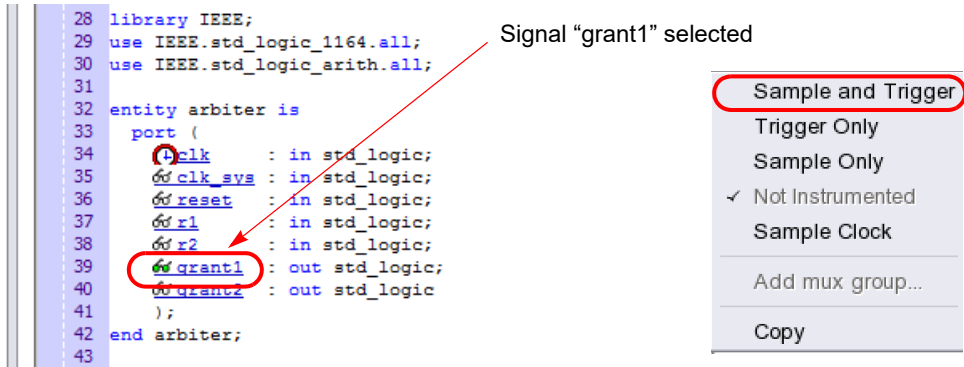
1. In the RTL tab, select a signal to be sampled.

Click the glasses icon () next to a signal name and select a sampling or triggering choice from the popup menu. Once instrumented, the color of the icon changes, according to what was selected.

Green	Sample and trigger
Blue	Sample only
Red	Trigger only



- Always instrument signals that are not needed for triggering as Sample only, to reduce the overhead for trigger logic,. The watchpoint icon is blue for sample-only signals.
- Signals specified as Sample and trigger or Sample only can be included in multiplexed groups as described in [Assigning Signals to Groups, on page 27](#).
- Qualified clock signals can be specified as the Sample Clock (see [Sample Clock, on page 18](#)).
- Use Find to recursively search for signals and then instrument selected signals directly from the Find dialog box (see [Capturing Commands from the Console, on page 45](#)).
- Do not instrument the following input and output buffer signals: inputs of IBUF and IBUFGs and outputs of OBUFs and OBUFTs. These inputs and outputs drive or are driven by user logic, and this could cause errors during mapping.



The script window at the bottom displays the corresponding Tcl command that implements the selection (signals add -iice {IICE} -sample -trigger {/beh/arb_inst/grant1}) and the results of executing the command.

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

When you save the design, the instrumentation is saved in the IDC file.

2. You can specify partial instrumentation in certain cases.

- You can set watchpoints on individual bus segments (see [Instrumenting Buses](#), on page 24).
- You can instrument individual signals that are part of a folded hierarchy. A *folded hierarchy* is a design that has multiple instantiations of an entity or module, and therefore multiple instances of signals in a folded entity or module.

In such a case, the popup menu lists all the signal instances, displayed as an absolute signal path name originating at the top-level entity or module, and you can select the one you want. See [Example: Instrumenting a Folded Signal](#), on page 23

You can also do this by specifying the signals add command and the absolute path to the signal: `signals add /rtl/cnt_inst1/val`

For related information on folded hierarchies in the debugger, see [Activating/Deactivating Folded Instrumentation](#), on page 41 and

[Displaying Data from Folded Signals, on page 49](#) in the *HAPS Prototyping Debugger User Guide*.

3. Use these methods to insert or edit post-compile instrumentation:
 - Open the schematic view for an SRS (compiled) database. Right-click a selected signal in the schematic, and select Instrumentor and then the mode from the popup menu.
 - After compile, you can also open the IDC file with the `edit idc` command and add or modify the instrumentation. You can also use this command to create a new IDC file.
4. To disable a signal, select it on the RTL tab and then select Not instrumented from the popup menu.

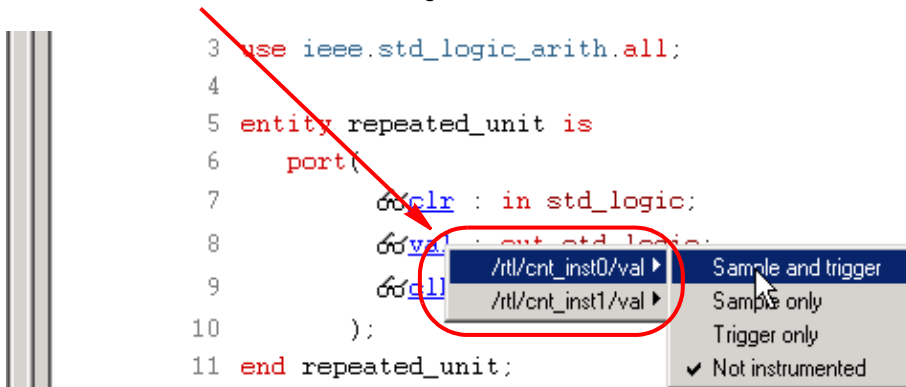
Example: Instrumenting a Folded Signal

The example below consists of a top-level entity called `two-level` which has two instances of the `repeated_unit` entity. So, there are two `repeated_unit` instances of the signal `val` source code:

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

Select either or both of these instances 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 icon changes to reflect the status.

Instrumenting Buses

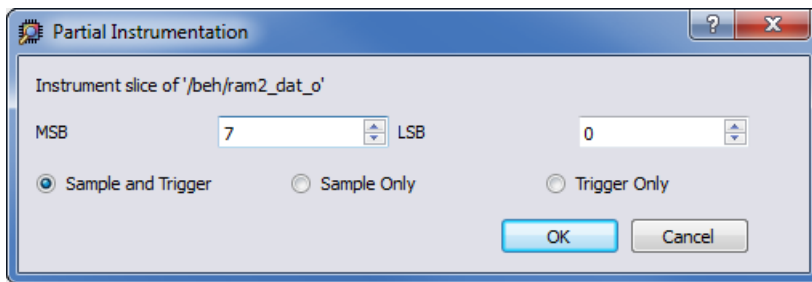
Entire buses, individual bits, or groups of bits of a bus can be individually instrumented.

- [Instrumenting a Partial Bus](#), on page 24
- [Instrumenting Non-Contiguous Bits or Bit Ranges](#), on page 25

Instrumenting a Partial Bus

This procedure describes how to instrument a sequence (range) of bits of a bus:

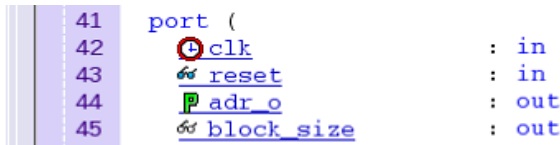
1. Right-click a bus that is not fully instrumented and select Add partial instrumentation from the popup menu to display the following dialog box.



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 25.
 - To instrument a single bit of a bus, enter the bit value in the MSB field and leave the LSB field blank.
 - When specifying the MSB and LSB values, you must follow the index order of the bus. For example, when defining a partial bus range for bus [63:0], the MSB value must be greater than the LSB value. Conversely, for bus [0:63], the MSB value must be less than the LSB value.
3. Select the type of instrumentation for the specified bit range from the radio buttons and click OK.

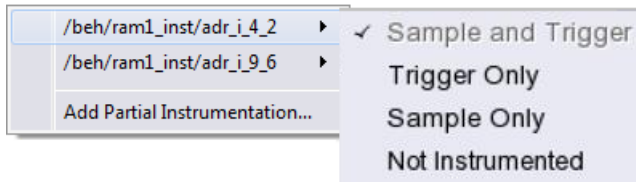
When you click OK, a letter “P” is displayed to the left of the bus name in place of the watchpoint icon. See the figure below. The color of this letter indicates the kind of instrumentation:

- Green – all bits are instrumented for sample and trigger
- Blue – all bits are instrumented for sample only
- Red – all bits are instrumented for trigger only
- Yellow – not all bits are instrumented the same way



4. Follow these steps to change the instrumentation of a partial bus.

- Right-click on the bus. Select the bit range or bit to be changed and select another kind of instrumentation from the adjacent menu.

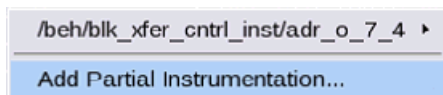


- To remove previous instrumentation from a bit or bit range, select Not instrumented from the menu.

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. Right-click the bus, and 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.
 - Bits cannot overlap groups (a bit cannot be instrumented more than once).

Instrumenting Fields Within a Record or Structure

You can instrument individual fields within a record or a structure. The following procedure describes this partial instrumentation technique:

1. Selecting a signal for instrumentation, either on the RTL tab or through the Instrumentor Search dialog box.
2. In the dialog box that opens, specify the field name and type of instrumentation.

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

Once instrumented, the signal is displayed with a P icon to indicate that portions of the record are instrumented. The color of the icon indicates the kind of instrumentation:

- Green – all fields of the record are instrumented for sample and trigger
- Blue – all fields of the record are instrumented for sample only
- Red – all fields of the record are instrumented for trigger only
- Yellow – 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

```

Assigning Signals to Groups

To reduce the amount of memory required for debugging, you can assign signals to logical groups. The debugger can then only load those groups that are required at that time into memory. The following procedure describes how to define multiplexed groups for signals.

1. Right-click a signal or bus that was instrumented as either Sample and trigger or Sample only and select Add mux group from the popup menu.

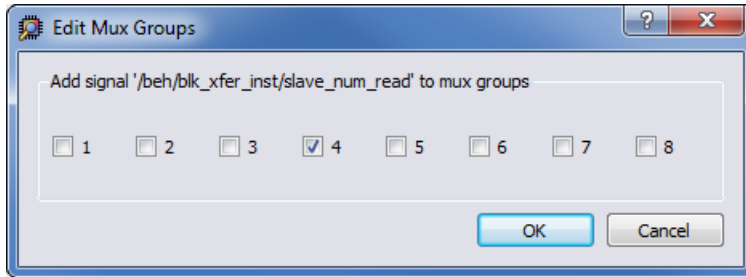
These signals cannot be included in multiplexed groups:

- Partial buses
- Signals instrumented as Sample Clock or Trigger only



2. Add signals individually to groups.
 - In the dialog box, select a group by checking the group number and then click OK to assign to the signal or bus to that group. You can have a maximum of eight groups, with 2000 signals per group.

- You can assign a signal to multiple groups by checking additional group numbers before clicking OK. Note however that only one group can be active in the debugger at any one time.



The equivalent command is `signals group`, which you can use to assign groups from the console window (see [signals](#), on page 111 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 debugger, see [Selecting Multiplexed Instrumentation Sets](#), on page 40 in the *HAPS ProtoCompiler Debugger User Guide*.

Instrumenting the Verdi Signal Database

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

1. Load the design into the instrumentor.
2. Parse the essential signal list from the ESDB using this command:

```
verdi getsignals ESDBpath
```

ESDBpath is the location where `es.esdb++` is installed. For example:

```
verdi getsignals path/es
```

3. Instrument the essential signal list using this command:

```
verdi instrument
```

The signals are automatically instrumented as sample and trigger. It may not be possible to instrument all ESDB signals because of changes to the original signal names because of synthesis optimization or because of naming differences between the instrumentor and Verdi tools.

4. Instrument the sample clock (a sample clock is required by the instrumentor).
5. Configure the IICE and instrument the design.
6. Continue with the rest of the flow.

Run through the flow and program the design into the FPGA. When you run the debugger, it samples the data and generates the fast signal database (FSDB) which you can display in the Verdi nWave viewer. For more information on the fast signal database and using the Verdi nWave viewer, see [Generating the Fast Signal Database, on page 58](#) in the *HAPS ProtoCompiler Debugger User Guide*.

Instrumenting Signals Directly in the idc File

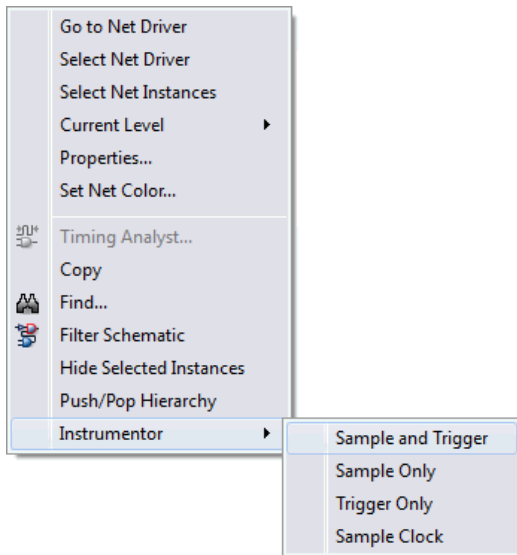
In addition to the methods described in the previous sections, signals can be instrumented directly within the HDL Analyst (`srs` file) outside of the instrumentor. This is a post-compile operation, and can be used in both the UC and standard compile flows.

By working in the `idc` file, you can easily edit previous instrumentation. You can also use this technique to allow signals within a parameterized module, which was previously unavailable for instrumentation, to be successfully instrumented. Follow these steps:

1. From a compiled state, open the schematic view by clicking the View the schematic icon or by typing the view schematic command.

The compiled state can be from either the UC flow or the standard compile flow.

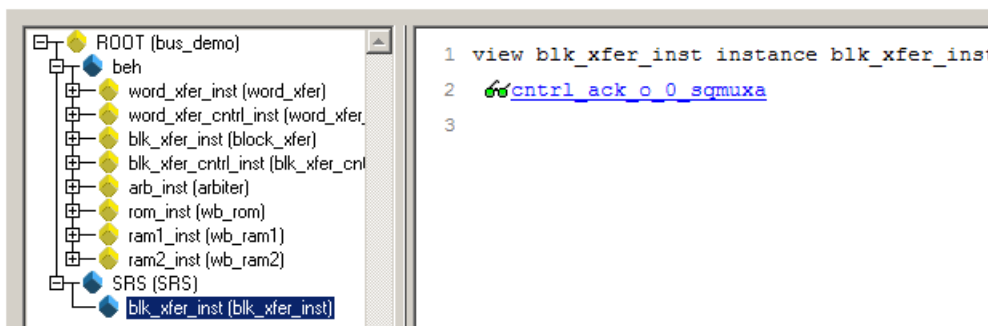
2. Select the signal and specify the kind of instrumentation.
 - In the schematic view, highlight the net of the signal to be instrumented.
 - Right-click the net, select Instrumentor from the popup menu, and select the type of instrumentation to be applied.



The instrumentation is automatically applied to the open instrumentor view.

3. Save the updated instrumentation and rerun compilation.

When you open the debugger, an SRS entry is included in the hierarchy browser; selecting this entry displays the additional signal or signals added to the instrumentation. 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 instrumentation must use the VHDL style format.

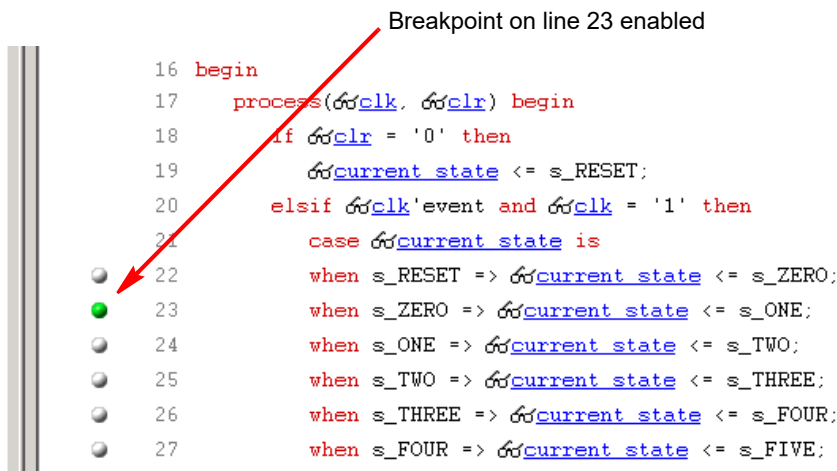
Adding Breakpoints

Add breakpoints to trigger data sampling in the debugger.

1. To add a breakpoint in the instrumentor, simply the circular icon to the left of the line number (●).

Only the breakpoints that are instrumented in the instrumentor can be enabled as triggers in the debugger. The color of the icon changes to green when enabled.

The instrumentor creates trigger logic for the breakpoint, which becomes active when the code region where 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.



Breakpoints can also be pre-triggered from the command line to arm the IICE in the debugger to run immediately after configuration (see [breakpoints, on page 18](#) in the *Debugging Environment Reference*).

2. Do the following to instrument breakpoints for individual instances from a folded hierarchy.

A *folded hierarchy* is a design that has multiple instantiations of an entity or module, and therefore multiple instances of breakpoints in a folded entity or module.

- Click the signal where you want to add a breakpoint.

- Select the instance you want from the popup list of instances, which shows the instances as absolute signal path name originating at the top-level entity or module. You can select any or all of them. See [Example: Instrumenting a Breakpoint in a Folded Hierarchy, on page 32](#).
- You can also do this by specifying the breakpoints add command and the absolute path to the signal:

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

See [breakpoints, on page 18](#) in the Debugging Environment Reference for more information.

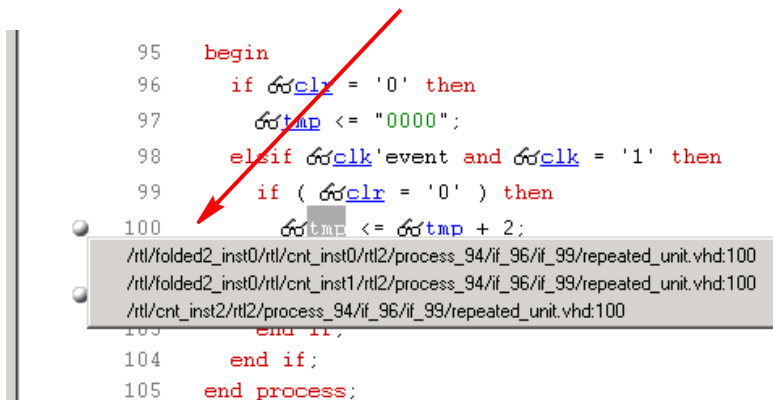
3. To disable a breakpoint or an instance of a breakpoint, click it.

Breakpoint settings are toggle settings, so clicking again changes the status.

Example: Instrumenting a Breakpoint in a Folded Hierarchy

The example below consists of a top-level entity called top which has two instances of the repeated_unit entity. When the icon at line 100 is selected, multiple of a breakpoint are available for sampling, as shown in the following figure:

Folded breakpoint on line 100 selected



Clicking one or all of them sets the breakpoint, and the color of the icon changes to reflect the status:

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

Language Limitations for Instrumentation

There are some limitations on which parts of a design can be instrumented in designs compiled with the standard compile flow. The instrumentation limitations are described here:

- [VHDL Instrumentation Limitations](#), on page 33
- [Verilog Instrumentation Limitations](#), on page 35
- [SystemVerilog Instrumentation Limitations](#), on page 38

VHDL Instrumentation Limitations

The synthesizable subsets of VHDL IEEE 1076-1993 and IEEE 1076-1987 are supported.

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 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 instrumentation and synthesis.

Transient Variables

Transient variables defined locally in VHDL processes cannot be instrumented.

Scalar Signal Syntax

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

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

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

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

Breakpoints and Register Inferencing

Breakpoints inside processes that infer registers can only be instrumented if they follow the coding styles outlined below. If you use any other coding style, no breakpoints can be instrumented inside the corresponding process. When the design is compiled, the instrumentor issues a warning that the code cannot be instrumented.

The reset polarity and clock-edge specifications in the examples. There are no restrictions for reset or clock polarity.

For registers 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 registers with synchronous reset or without reset, use one of these styles:

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

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

A coding style that uses wait statements must have only one wait statement and it must be at the top of the process.

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 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. Only wires and registers declared in the module scope 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, the following limitations apply:

- If the keyword `module` or `endmodule`, or if the closing `'` of the module port list is located inside an include file that is referenced multiple times, 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 an include file that is referenced multiple times, no breakpoints inside the corresponding `always` block can be instrumented.

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

Consider the following three files:

adder.v File

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

adder8.v File

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

adder16.v File

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

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

Macro Definitions

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

Always Blocks

Breakpoints inside a synchronous flip-flop inferring an `always` block can only be instrumented if the `always` block follows the coding styles outlined below. If you use another coding style, the instrumentor issues a warning that the code is not instrumentable.

For registers with asynchronous reset:

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

For registers 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 examples only. There are no restrictions with regard to these parts of the syntax.

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 for instrumentation.

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 you can only instrument an entire structure (sample mode only). The following code segment illustrates these limitations:

```
module lddt_P_Struc_top (
    input sigsig_clk, sigsig_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 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 (you cannot set a watch point on the signal). If this signal is instrumented for sample and trigger, the instrumentor allows only sampling and ignores the triggering instruction.

Union Construct

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

The example below shows an acceptable sample code segment for a packed union; with the trigger expression for union `d1`:

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

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

```

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

```

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

```

Similarly, to set **st2**:

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

This is how the trigger expression is defined:

```

16b'   00110110  11001000
      |   st1   |   st2   |

```

Arrays

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

Interface

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

```

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

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

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

```

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

Port Connections for Interfaces and Variables

You cannot instrument named port connections on instantiations to implicitly instantiate ports.

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

You cannot instrument the concatenation syntax on an array watchpoint signal. Take this signal declaration:

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

This is the normal syntax to set a watchpoint on this signal:

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

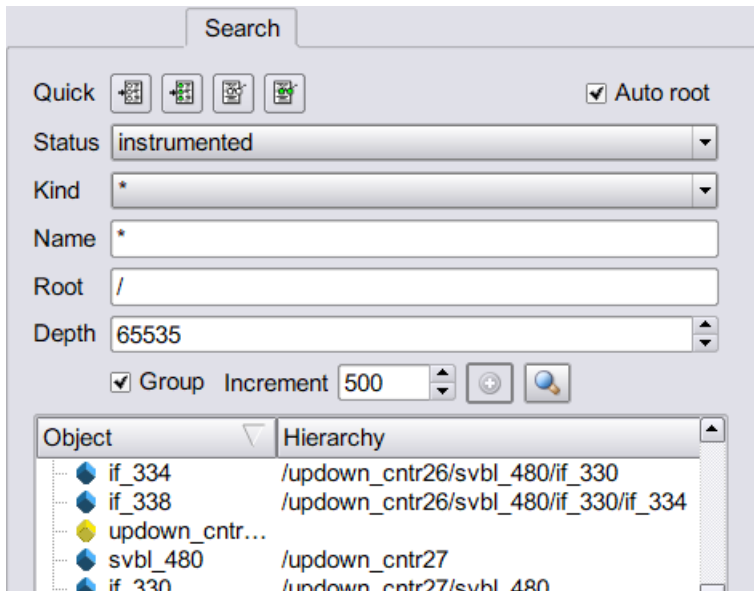
However, the 4-bit vector cannot be divided into smaller vectors and concatenated (as accepted in SystemVerilog). For example, the syntax below is not supported:

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

Finding Design Objects in the Instrumentor

You can use the Search panel to search for signals, breakpoints, and/or instances and also specify instrumentation.

1. Open the Search panel by clicking the Search tab in the GUI.



2. Use the panel settings to find the objects you want.

These are some ways to search:

- Use the icons at the top as shortcuts to list all instrumented signals or breakpoints, or signals available for instrumentation as watchpoints and breakpoints.
- Search for particular instances by typing a name. You can use wild cards.
- Restrict the scope of the search using the Root and Depth fields to specify the starting hierarchical level and the depth of the sample buffer to search.

The objects found are displayed in the lower section of the panel along with their hierarchical locations. Breakpoints and signal include the corresponding icon.

3. Set instrumentation from the search results.

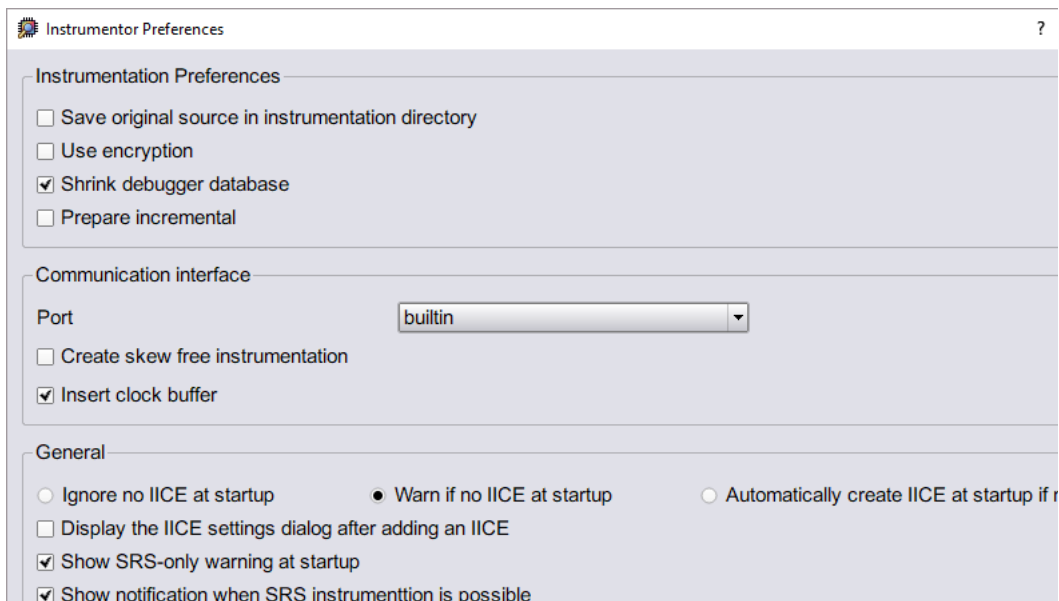
- To change instrumentation for a watchpoint, click the icon and select the instrumentation type from the popup menu. To specify instrumentation for multiple objects at the same time, use Ctrl-Shift to select multiple signals and then apply the change to all selected signals.
- To toggle the instrumentation status of a breakpoint, click the breakpoint icon. You also can use Ctrl-Shift keys to select multiple breakpoints and change the status of all of them at the same time.

Saving Instrumentation

When you save your design, the instrumentation information is written to an IDC file. The following procedure describes how to save your design, and optionally save the original HDL source files with it.

In the standard compile flow, including the original HDL source with the instrumented design simplifies design transfer when instrumentation and debugging are performed on separate machines. It is especially useful when a design is being debugged on a system that does not have access to the original sources.

1. To include the original HDL source, follow these steps:
 - Select Instrumentor->Instrumentor Preferences from the main menu to display the dialog box.
 - Select the Save original source in instrumentation directory check box in the Instrumentor Preferences dialog box.



2. To encrypt the source files when they are written out, also enable the Use encryption check box.

The encryption is based on a password that is requested when you write out the instrumented design. Encryption allows you to debug on a machine that you feel would not be sufficiently secure to store your sources. You are prompted to reenter the encryption password when you open the design in the debugger from this machine. The decrypted files are never written to the hard disk on the machine.

Use these criteria when selecting a 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
- Passwords are a user responsibility; Synopsys cannot recover a lost or forgotten password

3. Instrument the design as usual.

4. Save the instrumentation with File -> Save.

Saving the 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.

Capturing Commands from the Console

To capture all text written to the console window, use the log console command (see the *Reference Manual*). To capture all commands executed in the console window use the transcript command (see the *Reference Manual*). To clear the text from the console window, use the clear command.

Instrumenting a UC Design

There are two ways to designate probe signals and instrument the RTL design with the UC flow: using SystemVerilog assertions and using \$dumpvars. In addition to these RTL instrumentation techniques, you can add post-compile instrumentation.

The instrumented signals are managed through the Instrumentation-Debug Constraints file (IDC), which contains constraints for the instrumented signals and break points, as well as the Intelligent In-Circuit Emulator (IICE) which is the debug IP.

See the following for more information:

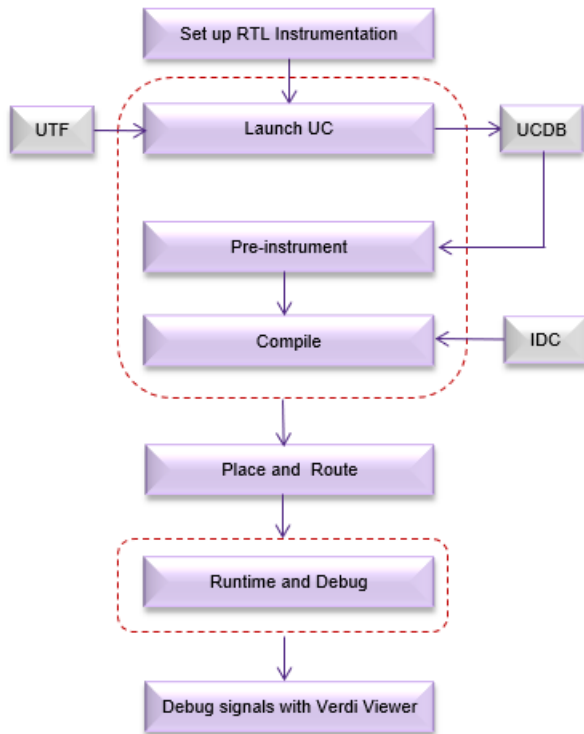
Instrumentation	<ul style="list-style-type: none">• Adding UC Instrumentation, on page 46• Setting Instrumentation Limits for UC Designs, on page 48• Using SystemVerilog Assertions, on page 53• Using \$dumpvars, on page 61• Specifying Post-Compile Instrumentation for UC Designs, on page 48
IDC and IICE	<ul style="list-style-type: none">• Working with an IDC File, on page 72

At runtime, you can run the design on the hardware and debug the design. You can use the Synopsys Verdi software to check the waveforms from the instrumented signals against a golden version. You can also debug the design on hardware, but the details vary, depending on the hardware you use.

Adding UC Instrumentation

The figure summarizes the steps in the UC instrumentation flow. The dashed lines indicate the steps that are performed in the prototyping tool. Unlike the standard compiler flow, in the UC flow you cannot use the instrumentor to

directly add instrumentation to the RTL. Instead, it is handled by the VCS tools. You can however, add post-compile or SRS instrumentation to the design after compile.



The steps shown apply to both instrumentation modes available in the UC flow, SVA and \$dumpvars. For specifics, refer to these topics:

- [Adding SVAs for Instrumentation](#), on page 53
- [Adding \\$dumpvars for Instrumentation](#), on page 62
- [Specifying Post-Compile Instrumentation for UC Designs](#), on page 48

Setting Instrumentation Limits for UC Designs

You can set limits to the number of bits instrumented in a UC design, either globally or on a per-module level. Do this before running the prototyping flow. Global instrumentation limits are set in the UTF control file, and local limits are set in the RTL for the module. The two ways to specify instrumentation limits provide flexibility in a large design.

1. To set a global instrumentation limit, add this command to the UTF file:

```
debug -dumpvars_maxbits <number>
```

The UTF file provides the controls to run the VCS compiler as a part of unified compile.

The instrumentation limit set in the UTF is a global command that sets a maximum limit on the number of signals that can be instrumented. For example, if it is set to 12, and more than 12 signals are instrumented, you get an error in the uc.log file:

```
Error-[FS_MAXBITS_EXCEEDED] FSDB maxbits limit exceeded
Total bit count (24) for nodes dumped by $dumpvars task call
exceeds maxbits limit (12).
```

2. To set a local limit, add it to the module definition in the RTL.

```
(* maxbits=< > *)
```

The local limit overrides the global limit. The following example shows the instrumentation limit for the module set to 24:

```
initial
begin
  (* maxbits=24 *) $dumpvars (1, top.temp1,top.temp2,top.temp3);
end
```

If the listed signals are 8 bits each and the global limit is set to 12, the compiler would normally error out because 24 exceeds the maximum limit of 12. However, because the local limit on \$dumpvars is set to 24, there is no compilation error.

Specifying Post-Compile Instrumentation for UC Designs

With the UC flow, the prototyping software can only define RTL instrumentation through \$dumpvars and SVAs. However, you can edit the instrumentation file (IDC) after the compile stage. Follow these guidelines.

1. Compile the UC design with the `launch uc` and `run compile` commands to generate a compiled database.

You can now edit the existing IDC file or create and add a new IDC file to the design.

2. To edit a file, open the IDC file generated from the RTL instrumentation.

- The `instrument.idc` file generated after `run compile` does not contain instrumented signals; it only contains basic IICE settings.
- Run the `export idc` command to export the IDC files with instrumentation. The `identify_compile_out.idc` file contains all the \$dumpvars and SVA signal information, but there are also individual files available for each: `dumpvars_identify_compile_out.idc` and `sva_identify_compile_out.idc`.
- Edit the combined IDC file:

```
edit idc identify_compile_out.idc
```

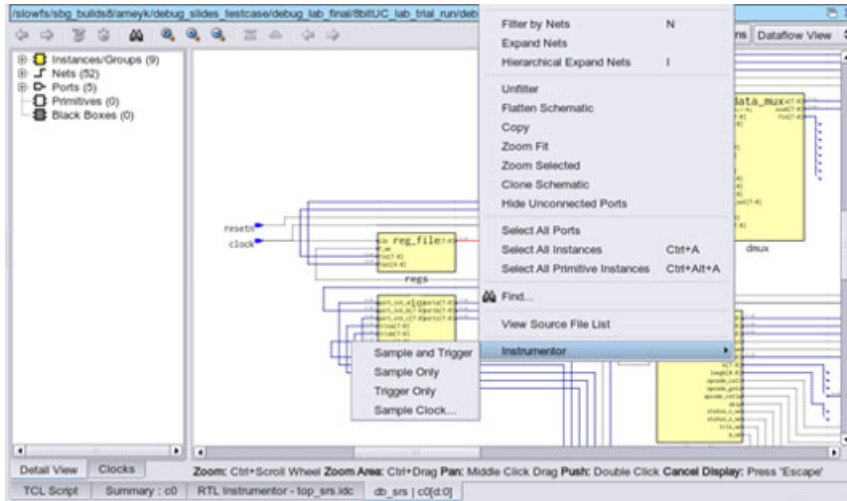
The GUI opens, where you can edit the signals.

3. To create a new IDC file with SRS (post-compile) instrumentation, follow these steps:

- After `run compile`, create a new IDC file with this command:

```
edit idc -mode gui SRS_based_file.idc
```

- In the GUI that opens, select the signals and set sample and trigger instrumentation for them.



4. Follow these guidelines when editing the IDC information in the file
 - Make sure the hierarchical signal path starts with the top-level module.
 - Use a period (.) not a slash (/), as the path hierarchy separator.
 - Specify signals in the generate scope with square brackets [] instead of brackets (), because there is no need to specify the range for a signal in IDC.

Example:

```
iice clock -iice {IIICE_UC} -edge positive {top.clock_c}
signals add -iice {IIICE_UC} -silent -trigger -sample
{top.i1.i2.s1}
```

5. Specify the post-compile instrumentation file at the next stage of the design, with the run pre_partition or run pre_map commands.

Example: Post-Compile Instrumentation with UC Flow

This is a typical sequence of commands to specify post-compile instrumentation in a multi-FPGA design flow:

```
# Compile the design
launch uc -utf run.utf -ucdb ucdb -v 2.0
run compile -ucdb ucdb
# Add post-compile instrumentation
edit idc -mode gui srs_based.idc
# Run with post-compile IDC
run pre_partition -idc ./srs_based.idc -tss ./board.tss -fdc
./top.fdc
# Run flow, through individual FPGA synthesis and place and route
.
.
.
# Export instrumentation for runtime and debug
export runtime -path ./multi_fpga_debug
```

Example: Instrumentation Log

Generating IICE 'IICE_0' for the following settings:

Design settings:

```
Device family      HAPS100_4F
Communication port umrbus
Skew-resistant logic off
Additional pipe stages 3
Export Trigger     no
Import Trigger     0
Language          verilog
```

Sample Buffer:

```
Type      haps100_DTD_builtin
Depth     16384
Max depth 67108863
Width     89 bits
```

Trigger controller:

```
Type      Simple Triggering
Sync-tree Delay 5 cycles
Primary partition FB1.uA
Sync-tree FPGAs FB1.uA FB1.uB FB1.uC FB1.uD FB2.uA FB2.uB FB2.uC FB2.uD
```

Clock groups for IICE IICE_0:
none

Signal distribution for IICE IICE_0:

FPGA	Width	Type	Instrumentation Name
FB1.uA	10	primary	
	1	S&T	top.reset_n
	8	S&T	top.counter1
	1	S&T	top.trigger1
FB1.uB	9	secondary	
	8	S&T	top.counter2
	1	S&T	top.trigger2
FB1.uC	9	secondary	

Definitions of the Trigger Controller terminologies used in the log:

- Type—Type of trigger control.
- Sync-tree Delay—Delay on the trigger entering the IICE block from other FPGAs in cycles.
- Primary Partition—Indicates where the main trigger pulse is generated or where the triggers from other FPGAs are combined.
- Sync-tree FPGAs—List of FPGAs used to create an efficient FPGA synchronization tree.

Using SystemVerilog Assertions

You can use SystemVerilog assertions (SVAs) to create an unlimited number of observation points anywhere in the design. The details of using SVAs are discussed here:

- [Adding SVAs for Instrumentation](#), on page 53
- [Detecting Assertion Failures with a HAPS System Reset](#), on page 58
- [Debugging Designs with SVAs](#), on page 59

Adding SVAs for Instrumentation

To use SVAs to define instrumentation signals, specify the assertions in the RTL. The SVAs are processed as part of RTL compilation and the compiler creates the required assertion logic. This information is translated into hardware debug signals, and the signal data is read back at runtime. Refer to [Adding UC Instrumentation](#), on page 46 for an overview of the flow.

The following procedure describes the steps to define probe signals using SVAs. For information about defining probe signals is to use \$dumpvars, refer to [Using \\$dumpvars](#), on page 61.

1. Do the following in the RTL source files:
 - Define the signals you want to probe by adding the SVA task to the RTL for the module.
 - Include SVAs in the RTL. You can use immediate (assert) or concurrent (assert property) SVA statements. Review the limitations ([Limitations of Using SVA Mode](#), on page 56). For an example, see [Examples: SVAs in the RTL](#), on page 57.
 - If the inputs for SVAs in a module are only top ports or do not have top I/O ports, add the `/*synthesis syn_noprune=1*/` synthesis directive to prevent the module containing the SVAs from being optimized away at the run compile stage.
2. Set additional settings in the UTF file besides the required commands.

The UTF file contains a pointer to the essential VCS and Simon setup commands.

 - Enable assertions by adding this option in the UTF file:

```
assertion_synthesis -enable ALL
```

Refer to the VCS documentation for a complete description of the syntax.

- If required, specify instrumentation limits in the UTF file. For more about instrumentation limits, see [Setting Instrumentation Limits for UC Designs, on page 48](#).
- Optionally, use `wire_resolution` commands to specify how multi-driver XMR conflicts are resolved. The default is WAND.

```
wire_resolution -default_xmr_conflict {WAND | WOR | XMR}
```

Refer to the VCS documentation for comprehensive information about the UTF commands.

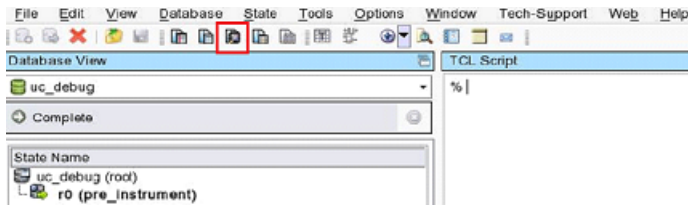
3. Start the software, and compile the design with unified compile:
 - Set the `VERDI_HOME` environment variable and specify the path to the Verdi® software.
 - Set path to VCS installation: `VCS_HOME`.
 - Set this option: `option set debug_sva 1`
 - Use the `launch uc` command to run VCS unified compile on the design.

The tool compiles the design with the VCS compiler and automatically adds the assertion signals as instrumentation signals. This is a typical command sequence:

```
database load uc_debug -autocreate -technology HAPS-100_4F
option set debug_sva 1
launch uc -utf myUTF1.utf -ucdb myUcdbOutput
```

4. Set up the debug IP.
 - Run `pre_instrument` on the UC database to generate an instrumentation database:

```
run pre_instrument -ucdb myUcdbOutput
```
 - Use the `edit idc` command to edit the IDC file and add the appropriate IICE settings and clock instrumentation. You can also use the GUI:



- Define required IICE settings like depth, sampling clock, sampling clock edge in the file.
- Instrument the clock.
- Define other functionality as needed with the `iice new` and trigger group commands in the IDC file.
- Save the IDC file. This is an example of a user IDC:

```

hierarchy delimiter .
device jtagport builtin
device umr_pipe 1
iice new {IICE_UC} -type regular
iice controller -iice {IICE_UC} none
iice sampler -iice {IICE_UC} -depth 1000
iice sampler -iice {IICE_UC} -pipe 3
iice clock -iice {IICE_UC} -edge positive {top.inst1.clock}

```

5. Compile the design with the IDC file.

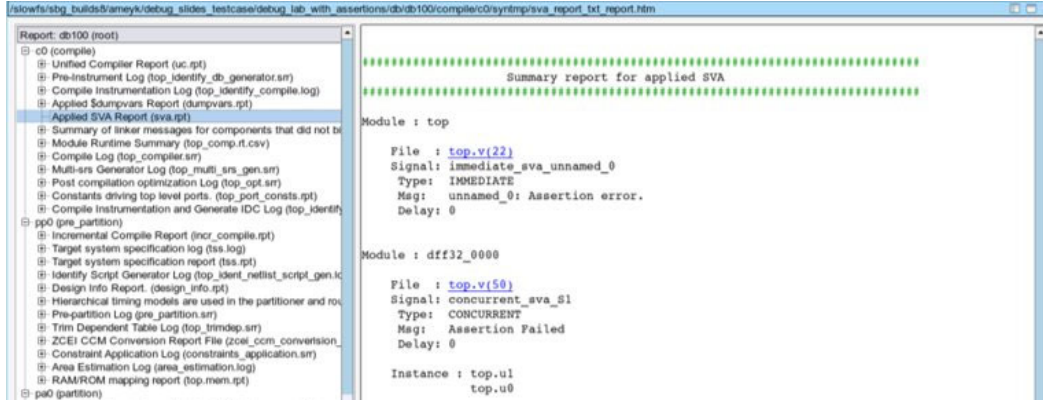
Specify the IDC file when you compile the design:

```
run compile -idc myIDC.idc
```

The signals are instrumented during the compile stage and no further instrumentation is required. The compile stage uses the IDC file settings to instrument the SVA probe signals specified in the RTL.

6. Check that the instrumented signals have been correctly implemented.

- In the GUI, check the Applied SVA report under the compile stage of the design. It shows the assertions that were applied from the RTL along with their type (concurrent or immediate). Click the link to cross-probe from the report to the assertion in the source code.



- Check the compile log report, `<design_name>identify_idc_generator.log`. After the system generate and pre-map stages, you can also check the signals in the instrumentation log of the log report.
- Use the `export idc` command to check the debug signals that were created. You can also run the command to make incremental changes to the IDC file for incremental debug. described in [export idc](#), on page 55 in the *Command Reference*.

```
export idc -path <path>
```

This command generates a `dumpvars_report.txt` report in the exported IDC directory, which lists all the signals and their status (applied/not applied). Compare these signals with the signals in the HDL code under the unified compiled database.

- At runtime, you can reset the HAPS system to check for assertion failures .
7. To specify a post-compile instrumentation file with information generated after the compile stage, refer to [Specifying Post-Compile Instrumentation for UC Designs](#), on page 48.
 8. Run through the rest of the flow as usual, and use the Verdi interface to view and analyze the probe signals.

Limitations of Using SVA Mode

The following features are currently not supported:

- `$assertkill` and `$assertoff`

- Cover property
- SVA property with multi-clock events
- Trigger groups for DTD2/3/4 IICEs

Examples: SVAs in the RTL

You can directly add assertions to the RTL as shown below, with *p* being any property: S1:assert property (*p*)

You can also add immediate or concurrent assertions. The following example shows how an immediate assertion is implemented in the design, with the schematic showing the assertion net:

```
module test (input a, input b, input c, output out);
  assign out = a;
  always_comb
  begin
    assert ((a & b) != c);
  end
endmodule
```



Example: Command Sequence for the SVA Flow

The following snippet shows a typical sequence of commands to implement SVAs in a multi-FPGA design. The synthesis scripts for individual FPGAs include the run commands for compile, pre-map (with IDC), map, and place and route.

```
# Set SVA option
option set debug_sva 1
# Enable option in UTF file
assertion_synthesis -enable all
# Start unified compile flow
launch uc -utf run.utf -ucdb ucdb -v 2.0
# Run the flow up to System Generate
...
```

```
# Synthesize and place and route individual FPGAs in parallel
launch protocompiler -parallel 4 -tclcmd {set ENABLE_PAR 1}
-script ./db/dumpvar_sg_exp/fb1_uA/fb1_uA_srs.tcl -script
./db/dumpvar_sg_exp/fb1_uB/fb1_uB_srs.tcl
# Export files for runtime
export runtime -path ./multi_fpga_debug
```

Detecting Assertion Failures with a HAPS System Reset

Before debugging, check for assertion failures by doing a HAPS system reset. Earlier in the cycle, you can use reports and log files to check that the assertions were implemented correctly (see [Adding SVAs for Instrumentation, on page 53](#)).

The following procedure describes how to use the system reset method:

1. Program the HAPS system using Confpro.
2. Start ProtoCompiler runtime and Confpro simultaneously.
3. Enable the reset in Confpro.

For example, if the RTL uses active low reset, then the reset value should be 0.

4. Configure the IICE in runtime and run it.

- This is an example of an IICE configuration command:

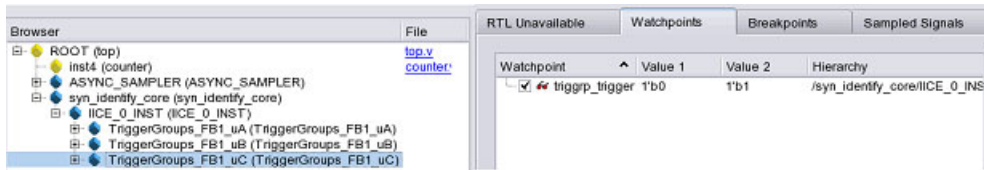
```
watch enable -language verilog \
syn_identify_core.IICE_UC_INST.TriggerGroups_regular.triggrp_trigger
1'b0 1'b1.
```

- Click Run. The runtime software waits for the trigger.

5. In Confpro, release the reset.

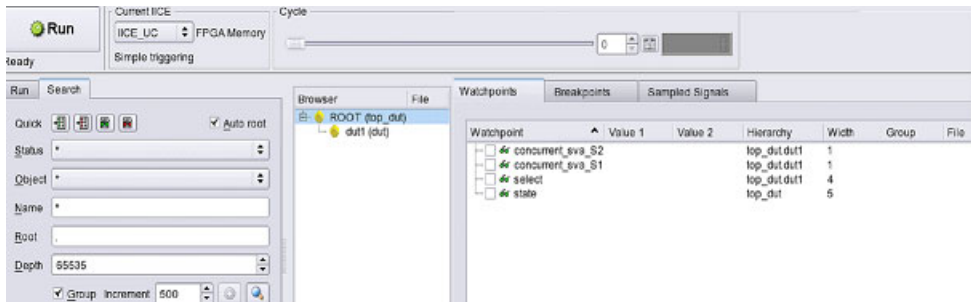
For example, modify the reset value to 1. The runtime software detects the trigger if SVA assertion fails, and starts to download the samples. In the waveform, the trigger position appears one cycle later after the assertion failure. In case of assertion failure, the user must debug the failing condition and fix the code accordingly.

For every trigger group added in the IDC file, you can see a corresponding trigger signal in runtime.



Debugging Designs with SVAs

The tool lists all the synthesized assertions with SVA instance names that were passed through the IDC file, along with other debug signals.



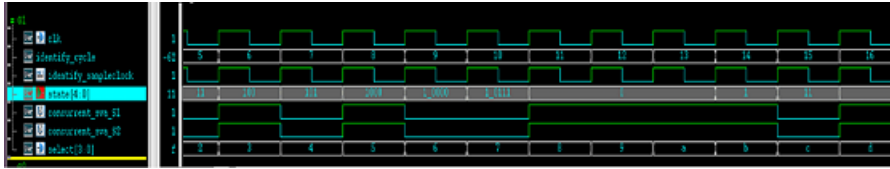
1. Enable the debug signal of the SVA instance using regular runtime commands to monitor assertion failures. For example:

```
watch enable -language verilog {top_dut.dut1.concurrent_sva_S2} {1'b0} {1'b1}
```

2. Check for signal value changes in the waveform.

When the assertion fails, the corresponding debug signal value changes from 0 to 1 in the waveform. For details, refer to the *HAPS ProtoCompiler User Guide*.

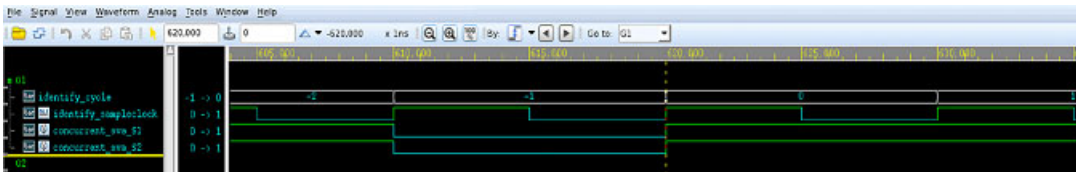
If you evaluate the SVA property with respect to the negedge clock when the iICE sampling is at the posedge clock, or vice versa, the SVA samples appear one cycle earlier in the waveform. For example: `assert property (@(negedge clk) req##1 grant).`



3. To filter SVAs in the waveform, use these commands:

- `signals set_property -name show_in_waveform -value 0 [hierarchy find -all -maxdepth 100 -name "*" -stat "*" -type "signal" .]`
- `signals set_property -name show_in_waveform -value 1 [hierarchy find -all -maxdepth 100 -name "*" -stat "*" -type "assert" .]`
- `write fsdb`

The following figure illustrates the filtered SVA signals.

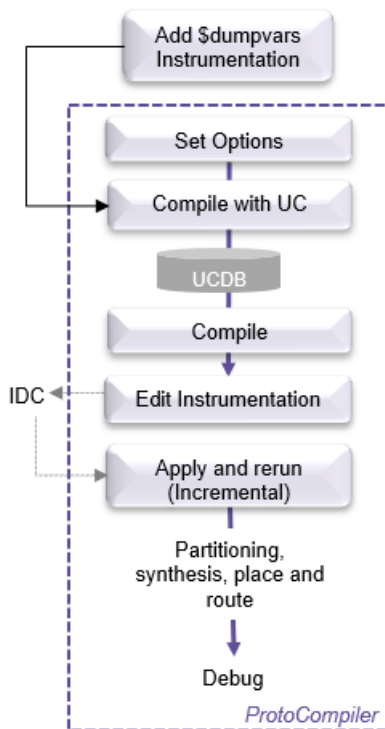


Using \$dumpvars

SystemVerilog includes a native \$dumpvars task construct, which provides a way to write out signal values. This provides an alternative way to instrument the design, instead of using SystemVerilog assertions (SVA). The SVA method is described in [Using SystemVerilog Assertions, on page 53](#).

- [Adding \\$dumpvars for Instrumentation](#), on page 62
- [Specifying \\$dumpvars for RTL Hierarchies](#), on page 67

Adding \$dumpvars for Instrumentation



The SystemVerilog \$dumpvars task provides a way to add RTL instrumentation to your design. Follow these steps to use the \$dumpvars construct to instrument signals that you can later analyze at the debug stage:

1. Define the signals you want to probe by adding the SystemVerilog \$dumpvars task to the RTL for the module.
 - Use the \$dumpvars task to list the variables to dump into a file specified by \$dumpfile. You can invoke the \$dumpvars task multiple times throughout the model (for example, in different blocks), but the tool executes all the \$dumpvars tasks at the same time.
 - Specify the signals as described in [Specifying \\$dumpvars for RTL Hierarchies, on page 67](#), providing the path to the signals, instance name, or top module with the \$dumpvars command in the RTL. All \$dumpvars signals are instrumented as sample and trigger by default.

- Use wildcard characters (*,?) to match hierarchy or instance signal names, specified within \$dumpvars. See [Using Wildcard Characters in \\$dumpvars](#) , on page 66 for details.

```

initial
//Add appropriate label for grouping the following signals
begin: IICE_A
    $dumpvars (1,eight_bit_uc.uc_alu);
    $dumpvars (1,eight_bit_uc.uc_alu.U1);
end
initial
//Add appropriate label for grouping the following signals
begin: IICE_B
    $dumpvars(1, eight_bit_uc.portc);
    $dumpvars(1, eight_bit_uc.resetn);
    $dumpvars(1, eight_bit_uc.decode.skip);
end

```

2. Start the synthesis software, and compile the design with unified compile:
 - Set the VERDI_HOME environment variable and specify the path to the Verdi® software.
 - Set this option: option set debug_dumpvars 1
 - Optionally, specify instrumentation limits in the UTF file as described in [Setting Instrumentation Limits for UC Designs](#), on page 48.
 - Use the launch uc command to run VCS unified compile on the design. The tool compiles the design with the VCS compiler.

For example:

```

database load uc_debug -autocreate -technology HAPS-100_4F
option set debug_dumpvars 1
launch uc -utf myUTF1.utf -ucdb myUcdb

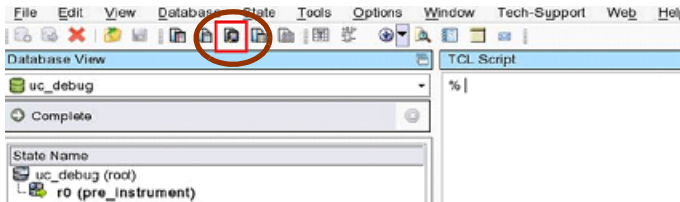
```

3. Set up the debug IP.

- Run pre-instrument on the UC database to generate an instrumentation database:

```
run pre_instrument -ucdb myUcdbOutput
```

- Use the edit idc command to edit the IDC file and add the appropriate IICE settings and clock instrumentation. You can also use the GUI:



- Set up the debug IP (IICE) and define clocks. Define required IICE settings like depth, sampling clock, sampling clock edge in the file. Define the memory buffer as FPGA Memory and specify a DTD sample clock.
- Define other functionality as needed with the `iice new` and `trigger group` commands in the IDC file. This is an example of a user IDC:

```

hierarchy delimiter .
device jtagport builtin
device umr_pipe 1
iice new {IICE_UC} -type regular
iice controller -iice {IICE_UC} none
iice sampler -iice {IICE_UC} -depth 1000
iice sampler -iice {IICE_UC} -pipe 3
iice clock -iice {IICE_UC} -edge positive {top.inst1.clock}

```

4. Specify the IDC file when you compile the design:

```
run compile -idc myNewIdc -ucdb myUcddb
```

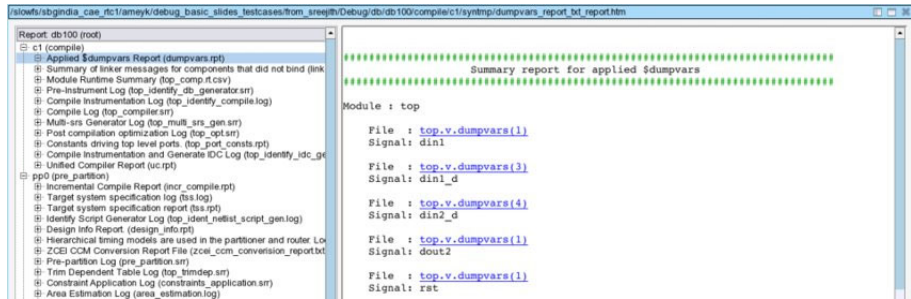
The signals are instrumented during the compile stage and no further instrumentation is required. The compile stage uses the IDC file settings to instrument the RTL \$dumpvars signals as sample and trigger by default.

5. Check that the instrumented signals have been correctly implemented.

- Use the `export idc` command to export the file and check the debug signals that were created. You can also run the command to make incremental changes to the IDC file for incremental debug, described in [export idc, on page 55](#), to view the signals.

```
export idc -path <exported_compile.idc>
```

This command generates a `dumpvars_report.txt` report in the exported IDC directory, which lists all the signals and their status (applied/not applied). You can compare these signals with the signals in the HDL code under the unified compiled database. The report is also available from the GUI (`dumpvars.rpt`) under the compile database state.



- Check the log files. Check the \$dumpvars instrumentation in the compile log report, <design_name>identify_idc_generator.log. You can also check the compile log report to make sure that the signals were correctly instrumented.
 - After the system generate and pre-map stages, you can check the signals in the instrumentation log section of the log file.
6. To specify instrumentation based on the compiled design, refer to [Specifying Post-Compile Instrumentation for UC Designs, on page 48](#).
 7. Run through the rest of the flow as usual, and use the Verdi interface to view and analyze the probe signals.signals.

Example: Commands for \$dumpvars Instrumentation Flow

The following snippet shows a typical sequence of commands to implement \$dumpvars instrumentation in a multi-FPGA design. The synthesis scripts for individual FPGAs include the run commands for compile, pre-map (with IDC), map, and place and route.

```
# Set option
option set debug_dumpvars 1
# Start unified compile flow
launch uc -utf run.utf -ucdb ucdb -v 2.0
# Run the usual flow up to System Generate
...
# Synthesize and place and route the FPGAs in parallel
launch protocompiler -parallel 4 -tclcmd {set ENABLE_PAR 1}
-scrip ./db/dumpvar_sg_exp/mb1_uA/mb1_uA_srs.tcl -scrip
./db/dumpvar_sg_exp/mb1_uB/mb1_uB_srs.tcl
# Export files for runtime
export runtime -path ./multi_fpga_debug
```

Using Wildcard Characters in \$dumpvars

Use wildcard characters (*,?) to match hierarchy or instance signal names, specified within \$dumpvars.

- Use asterisk (*) character to match multiple characters
- Use question mark (?) to match a single character

Hierarchy or instance signal names are matched either fully or partially with wildcard characters. By using wildcard characters appropriately, you can instrument a group of signals using a single \$dumpvars signal than specifying all the signals separately. Synopsys recommends that you use wildcard characters with instance names within a particular scope instead of hierarchy signals.

The following example shows how the wildcard characters are used within \$dumpvars:

```
initial
//Usage of wildcard character ? to match,
//test.u1.inst_u1.out1
//test.u1.inst_u1.out2
begin: label_1
$dumpvars(1, "test.u1.inst_u1.out?");
end

initial
//Usage of wildcard character * to match,
//test.u1.inst_u1.out1
//test.u2.inst_u1.out1
begin: label_2
$dumpvars(1, "test.*.inst_u1.out1");
end

initial
//Usage of wildcard characters ? and * to match,
//test.u1.inst_u1.out1
//test.u1.inst_u1.out2
//test.u2.inst_u1.out1
//test.u2.inst_u1.out2
begin: label_3
$dumpvars(1, "test.*.inst_u1.out?");
end
```

Specifying \$dumpvars for RTL Hierarchies

To instrument the signals, you must provide the path to the signals, instance name, or top module by specifying the \$dumpvars command in the RTL. All \$dumpvars signals are instrumented as sample and trigger by default.

Do not use \$dumpvars to specify VHDL hierarchies. \$dumpvars is a System-Verilog construct and cannot be used for VHDL hierarchies. Use \$dumpoort instead for VHDL hierarchies.

1. Specify the correct paths.

Incorrect paths will result in the following error during VCS compile:

```
Error-[XMRE] Cross-module reference resolution error
```

2. To specify complete hierarchies for \$dumpvars, the signal hierarchies must always start with the top module name.

Do not use \$dumpvars (0, top) to dump the signals of all levels of the hierarchies. This syntax only dumps the signals at the top-level module and ignores all other instances and hierarchies; it is equivalent to specifying \$dumpvars (1, top). Dumping signals for all levels of the hierarchies is not effective because the resulting instrumentation for a large design could over-burden the limited hardware debug resources.

3. To dump a specific hierarchy, specify the complete hierarchical path using \$dumpvars.

For example:

```
initial
begin
  $dumpvars (0, top.sub1_inst.in1,
    top.sub2_inst.out1, top.sub3_inst.in3);
end
```

Even though the following command is specified at the top level, it only affects the signals of the specified instance, not the top-level module.

```
$dumpvars (1, top.sub1_inst);
```

4. To dump signals for the current module level or submodules, use syntax like the following:

```
$dumpvars (1, top.sub1_inst);
```

In the following example, the first argument is a 1, so the tool dumps all variables within the module top; it does not dump variables in any of the modules instantiated by module top.

```
$dumpvars (1, top);
```

Similarly, the next example is specified within the sub1_inst module, and dumps all variables within the module sub1_inst.

```
$dumpvars (1, sub1_inst);
```

5. To dump signals for multiple modules as well as specific signals, use syntax like the following:

```
$dumpvars (0, top.mod1, top.mod2.net1);
```

6. Do the following to set instrumentation limits:
 - To set a global instrumentation limit, use the UTF file.
 - To set an instrumentation limit on an individual module, add it to the RTL for that module with the *maxbits attribute.

For details about global and module-level instrumentation limits, see [Setting Instrumentation Limits for UC Designs, on page 48](#).

7. To specify instrumentation other than the default of sample and trigger, modify the signals in the IDC file and run incremental debug, as described in [Working with an IDC File, on page 72](#).

You can add and delete instrumented signals or change the sample depth or trigger type.

Exporting Instrumentation for Debug

After the FPGAs have been synthesized, placed, and routed in the UC flow, generate the instrumentation information for the debugger using this procedure:

1. Export the instrumentation information using this command:

```
export idc -path./path_to_export_dir
```

2. Export the file that contains all the RTL information from \$dumpvars and SVAs, identify_compile_out.idc.

This is an example of the exported IDC file, showing both SVA and \$dumpvars signals:

```
hierarchy delimiter .
iice new {IICE1_UC} -type regular -mode {none} -uc_groups {}
iice controller -iice {IICE1_UC} none
iice sampler -iice {IICE1_UC} -depth 128
iice clock -iice {IICE1_UC} -edge positive {top.clk}
# These are $dumpvars signals:
signals add -iice {IICE1_UC} -silent -sample -trigger {top.din2}\
{top.rst}\
{top.din2}\
{top.u2.sum}\
# These are SVA signals:
{top.immediate_sva_unnamed_0}\
{top.u0.concurrent_sva_S1}\
{top.u1.concurrent_sva_S1}\
```


CHAPTER 3

Setting up the IDC and IICE

The Intelligent In-Circuit Emulator (IICE™) is an important part of the instrumentor. See these topics for information about setting up IICE units.

- [Working with an IDC File](#), on page 72
- [Working with IICE Units](#), on page 76
- [Setting Triggers in the Instrumentor](#), on page 92
- [Setting up for Real-Time Debug \(RTD\)](#), on page 104

Working with an IDC File

The Instrumentor Debugger Constraints (IDC) file defines the configuration for the hardware debug IP, along with clock specifications. These parameters are required. The IDC file can also be used to modify some parameters for incremental debug.

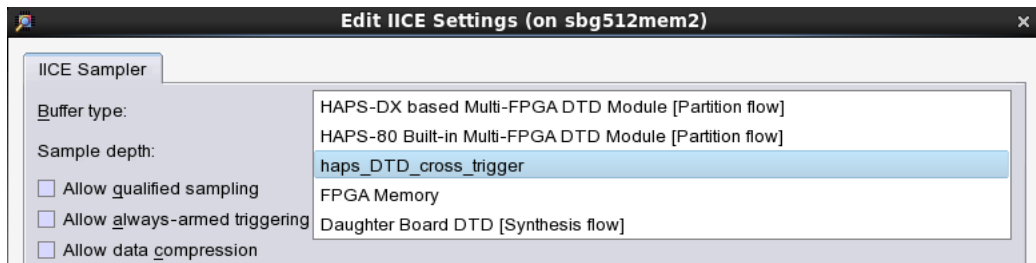
- [Creating an IDC File](#), on page 72
- [Evaluating IDC Files](#), on page 73
- [Using a Sample IDC for Essential Signal Analysis](#), on page 75

Creating an IDC File

You can create an IDC file from the GUI or the command line.

1. To create an IDC file through the GUI, click the Edit IICE icon () in the instrumentor GUI.

The dialog box that opens has multiple tabs, where you can specify various options. The most important settings are on the IICE Sampler and IICE Controller tabs.



2. To create an IDC file from the command line, use the `edit iice` command.

The equivalent commands for the IICE Controller and IICE Sampler tabs are `iice controller` and `iice sampler`.

3. You can have multiple IICE files in a design.
4. Specify the IICE file with the `run` command for that stage of design.

For example, specify it with the `run compile`, `run pre_partition`, or `run pre_map` commands.

Evaluating IDC Files

You can evaluate the instrumentation in an IDC file at various instrumentable states of the database, such as `compile`, `pre-partition`, `system generate`, `pre-map`, and `map`. Evaluating the IDC provides information about the feasibility of the instrumentation (or “instrumentability”) at that or any of the previous instrumentable states. Use this information to determine further instrumentation or incremental instrumentation.

1. Run the `report_debug_visibility` command from an appropriate database state.

```
report debug_visibility [-idc idc | -idclist idcListFile] -export_path path
```

The command does a DRC check and generates a comprehensive report of the instrumented signals for the current state and all previous visibility states. It shows the required signals and the stage where they are required, and reports whether the signal is available at the required stage. For example, if the command is run from `system generate (sg0)`, you get reports from the `compile (c0)` and `pre-partition (pp0)` states as well.

The command also generates IDC files. At the `compile`, `pre-partition`, and `pre-map` stages, this IDC is based on the compiled database, with RTL signal paths.

- For single-FPGA designs, type this command from a compiled or pre-map database state.
- For multi-FPGA designs, run this command from the system-level `compile`, `pre-partition`, and `system generate` stages, as well as the FPGA-level pre-map stage.

```
report debug_visibility -idc <modified.idc> -export_path path
```

This is an example of the report:

```

***** Debug Visibility Report *****
-----
Required signal: count_top.count_top_33.assert_signal
Not available: Signal is not part of RTL database. Possible cause wrong RTL path.
-----
Required signal: SRS.count_top_14.count[18]
Available at: pre partition
Available as: count_top_14.count[31:0]
-----
Required signal: SRS.count_top_18.count[28:18]
Available at: pre partition
Available as: count_top_18.count[31:0]
-----
Required signal: count_top.clk
Available at: pre partition
Available as: clk
-----
Required signal: count_top.count_top_1.assert_signal
Available at: pre partition
Available as: count_top_1.assert_signal
-----

```

This is an example of a generated IDC file. Refer to [report debug_visibility](#), on page 104 in the *Command Reference Manual* for more examples.

```

hierarchy delimiter .
device jtagport unibus
device prepare incremental 1
size new {IICE UC} -type regular
iice controller -iice {IICE UC} none
iice sampler -iice {IICE UC} -depth 1024
iice clock -iice {IICE UC} -edge positive {SRS.clk}
signals add -iice {IICE UC} -silent -sample -trigger {SRS.count_top_18.count[31:0]} \
{SRS.count_top_5.assert_signal} \
{SRS.count_top_6.assert_signal} \
{SRS.count_top_7.assert_signal} \
{SRS.count_top_8.assert_signal} \
{SRS.count_top_9.assert_signal} \
{SRS.count_top_11.assert_signal} \
{SRS.count_top_10.assert_signal} \

```

- When run from the system generate stage, the command generates a separate IDC file for each FPGA with the signals that can be instrumented for that FPGA. The following figure shows IDC files for FPGAs A and B from this stage.

```

hierarchy delimiter .
device jtagport unibus
device prepare incremental 1
iice new {IICE UC} -type regular
iice controller -iice {IICE UC} none
iice sampler -iice {IICE UC} -depth 1024
iice clock -iice {IICE UC} -edge positive {SRS.clk}
signals add -iice {IICE UC} -silent -sample -trigger {SRS.count_top_14.
count[31:0]} \
{SRS.count_top_18.count[31:0]} \
{SRS.count_top_13.assert_signal} \
{SRS.count_top_12.assert_signal} \
{SRS.count_top_14.assert_signal} \
{SRS.count_top_15.assert_signal} \
{SRS.count_top_17.assert_signal} \
{SRS.count_top_18.assert_signal} \

```

Generated IDC for FPGA A

```

hierarchy delimiter .
device jtagport unibus
device prepare incremental 1
iice new {IICE UC} -type regular
iice controller -iice {IICE UC} none
iice sampler -iice {IICE UC} -depth 1024
iice clock -iice {IICE UC} -edge positive {SRS.clk}
signals add -iice {IICE UC} -silent -sample -trigger
{SRS.count_top_16.assert_signal} \
{SRS.count_top_1.assert_signal} \
{SRS.count_top_3.assert_signal} \
{SRS.count_top_5.assert_signal}

```

Generated IDC for FPGA B

- If an IICE is not generated, check that the IICE clock signal is generated from that state. If it is not generated at the state where the evaluation is run, the command does not generate an IDC file.
2. Use the generated IDC file to make incremental changes and apply the modified IDC to the database state.

You must enable the incremental debug option to do this. For partitioned designs, edit the IDC files for an individual FPGA and then apply the modified IDC to the pre-map stage of that FPGA.

Using a Sample IDC for Essential Signal Analysis

Essential signals are those signals that make the Verdi data expansion engine achieve maximum visibility for the database. It is important to identify the essential signals required for instrumentation to achieve maximum debug visibility. You can generate a sample IDC file to determine the minimum set of essential signals needed to achieve maximum debug visibility. You must add these signals to the RTL through the \$dumpvars construct.

1. Set up the design.
 - Set the VCS_HOME and VERDI_HOME environment variables, making sure to use the same versions of the Verdi and VCS-MX tools.
 - Generate the UC database, with the verdi_mode option set to 1.

2. Specify the report debug_essential_signals command.

The command generates a report and sample IDC for the essential signals (see [report debug_essential_signals, on page 102](#)). The following example puts the essential signal data for design dut, and Unified Compile database from the ucdb directory in a directory named esa_data:

```
report debug_essential_signals -export_path esa_data -top_module  
dut -ucdb ucdb
```

3. Add the signals in the sample IDC to the \$dumpvars definition in the RTL.
4. Continue with the rest of the flow as usual.

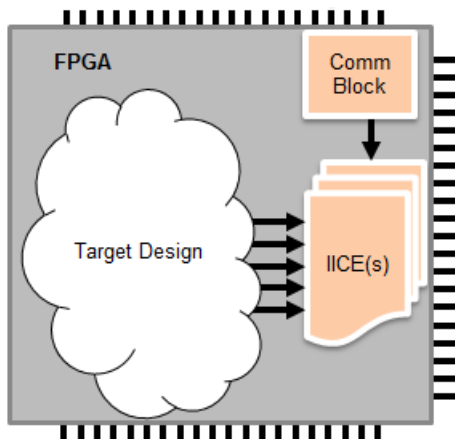
Working with IICE Units

The Intelligent In-Circuit Emulator or IICE is required for proper communication with the debugger. These topics describe IICE units, and how to set up and configure them.

- [About IICE Units](#), on page 76
- [Adding IICE Units](#), on page 77
- [Setting Common IICE Parameters](#), on page 79
- [Setting Individual IICE Parameters](#), on page 81
- [Setting Buffer Parameters for an IICE](#), on page 82
- [Defining the IICE Sample Clock](#), on page 83
- [Setting IICE Trigger Options](#), on page 93

About IICE Units

The Intelligent In-Circuit Emulator or IICE consists of logic added to the design for proper communication with the debugger. It acts as an on-chip logic analyzer, and contains its own sample buffer and trigger logic. Each IICE can be configured and debugged independently. One IICE can trigger another.



You can define one or more IICE units per design and configure each individually for communication with the debugger. Multiple IICE units allow signals to be triggered and sampled 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.

Parameters that are common to all IICE units are set in the Instrumentor Preferences dialog box. Parameters that are unique to an individual IICE are set in the IICE Configuration dialog box. These are the two main tabs used to set deep trace debug (DTD) and triggering information for each IICE.

- The IICE Sampler tab
Set sample depth and buffer settings here.
- IICE Controller tab
Set trigger information, including complex counter trigger width and state machine triggering, here.

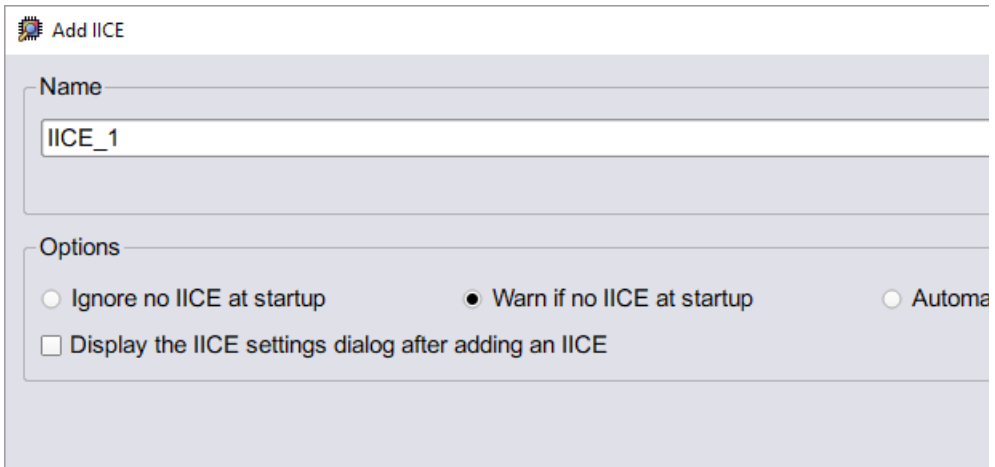
The IICE configurations set in the instrumentor impact the operations available in the debugger.

Adding IICE Units

You can add an IICE from the instrumentor GUI or from the command line. The IICE information is stored in the IDC file.

1. To add an IICE from the instrumentor GUI, click the Add IICE icon () or select the Instrumentor -> IICE -> Add IICE command.

The Add IICE dialog box opens to allow you to define the type and name of the new IICE unit.



Add IICE

Name

IICE_1

Options

☐ Ignore no IICE at startup
 ☒ Warn if no IICE at startup
 ☐ Automate


☐ Display the IICE settings dialog after adding an IICE

- You can add multiple IICE units to a design.
 - The recommendation is to create a different IICE for each distinct clock domain.
2. To add an IICE unit with a command, add the `iice new` command to the IDC file.

`iice new [iiceID] -type type`

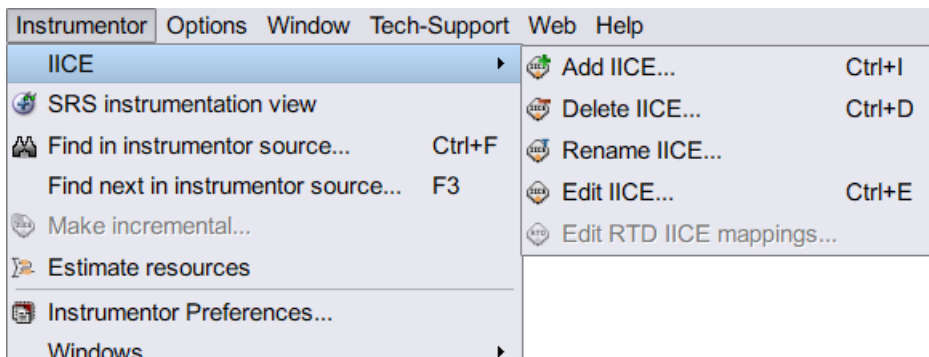
3. Specify parameters for the IICE:
 - Type a name for the IICE. The default IICE name is formed by adding an integer suffix to the word IICE (for example, IICE 0, IICE 1, etc.).
 - Select Regular (the default) to add a normal IICE unit or select RTD to add a real-time debug IICE unit in the *HAPS Prototyping User Guide*.
 - Set parameters that are shared by all IICE units as described in [Setting Common IICE Parameters, on page 79](#).
 - Set parameters that are unique to this IICE. Make sure to define the sample clock for the IICE.

The IICE information is stored in the IDC file.

4. To modify information for an IICE, use one of these methods:
 - From the GUI, select the IICE you want to work on and click the Edit IICE icon () to reset or change parameters.
 - Edit the IDC file directly, using the `edit idc file.idc` command.

5. Follow these steps to delete an IICE unit:

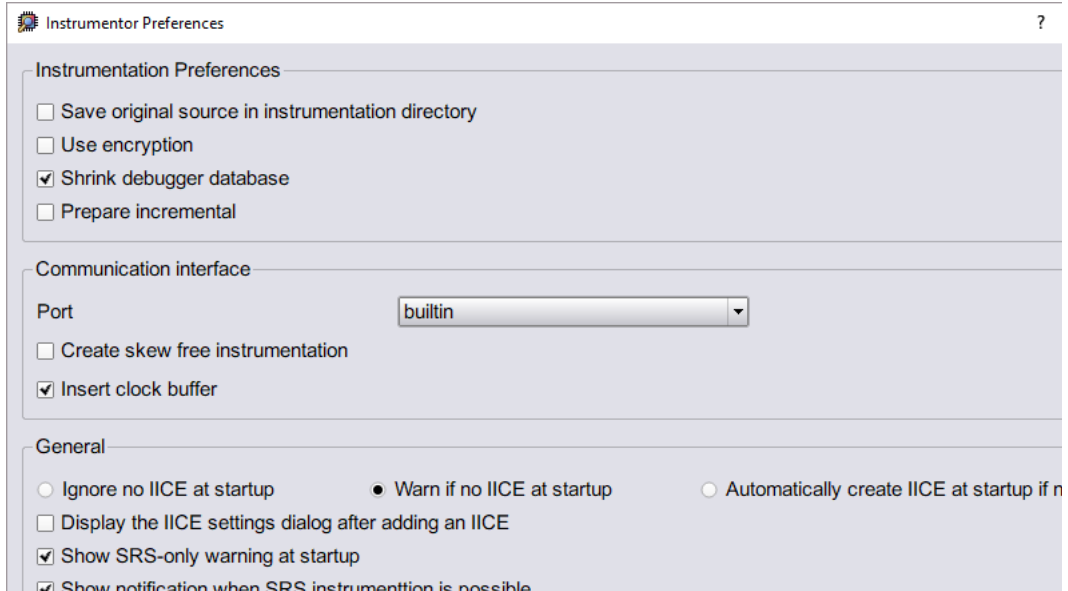
- If there is more than one IICE unit, first select the IICE to delete in the Control panel tab.
- Delete the IICE by selecting Instrumentor -> IICE -> Delete IICE from the top menu bar.



Setting Common IICE Parameters

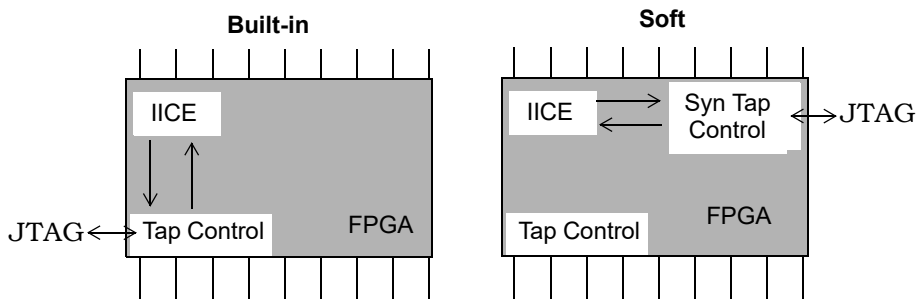
Follow these instructions to set IICE parameters that are shared by all the IICE units in a multi-IICE design. For information about setting parameters for individual IICE units, see [Setting Individual IICE Parameters, on page 81](#)

1. Select Instrumentor->Instrumentor Preferences from the GUI menu and click the Communication Interface entry in the Control Panel).



See [Instrumentor Preferences, on page 131](#) for descriptions of each option.

2. Set the Port option to reflect how the IICE is connected to the target device.
 - Set it to builtin when the IICE is connected to the JTAG tap controller on the target device.



- Use the soft option when the IICE is connected to the Synopsys tap controller on the target device. This option uses more resources because it is implemented it requires four user I/O pins to connect to the communication cable.

- If the IICE is connected to the target device through the UMRBus, set the option to umrbus.

3. Set other parameters.

These are some commonly used settings:

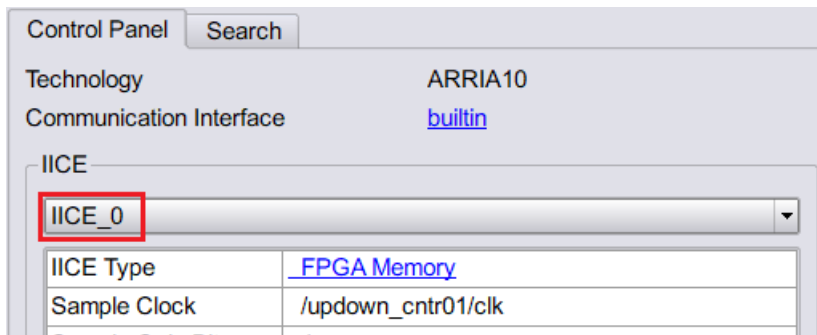
- With the standard compiler flow, you can choose to include the original HDL source files with the exported design files when the design is exported for debug (Save original source in instrumentation directory). There are options to encrypt the source files and shrink the database size.
- If no global clock resources are available for the JTAG clock, enable Create skew free instrumentation. When enabled, it incorporates skew-free controller-agent registers into the JTAG chain so that the instrumentation logic can operate without requiring an additional global clock buffer resource for the JTAG clock.
- To include more JTAG clock resources, enable Insert clock buffer to insert two global clock buffers in the debug IP.


Setting Individual IICE Parameters

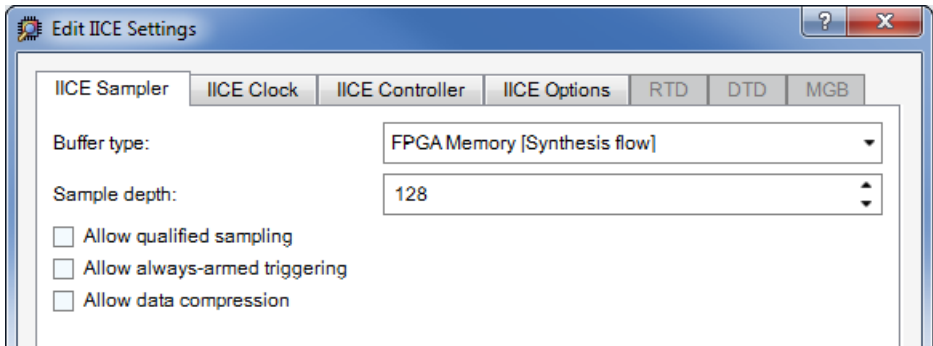
Follow these instructions to set individual IICE parameters. For information about setting parameters that are common to all IICE units, see [Setting Common IICE Parameters, on page 79](#).

1. If there are multiple IICE units, first select the IICE.

Go to the Control Panel and select the IICE to work on.



2. Open the Edit IICE Settings dialog box for individual parameters using one of these methods.
 - In the Control Panel, click the entry for IICE Type field.
 - Click the Edit IICE icon () to reset or change parameters.



- Edit the IDC file directly with the edit iice command.

Set individual IICE parameters from this interface, with the main tabs being IICE Sampler, IICE Clock, and IICE Controller. The equivalent commands in the IDC file are `iice sampler`, `iice clock`, and `iice controller`.

3. On the IICE Sampler tab, specify parameters for the memory buffer to be used for debugging.

See [Setting Buffer Parameters for an IICE, on page 82](#).

4. On the IICE Clock tab, specify the sample clock to be used for the IICE.

See [Defining the IICE Sample Clock, on page 83](#).

5. Set trigger options for the IICE on the IICE Controller tab.


See [Setting IICE Trigger Options, on page 93](#).

Setting Buffer Parameters for an IICE

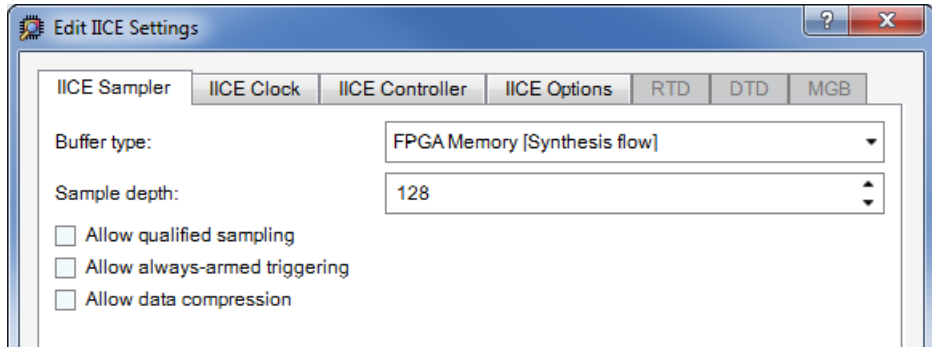
The IICE Sampler tab is used to define parameters for the memory to be used for sample storage during debug. You must define buffer parameters for each IICE. For more details about each option, see [IICE SamplerTab, on page 134](#).

1. If there are multiple IICE units, first select the IICE.

Go to the Control Panel and select the IICE to work on.

2. Click the Edit IICE icon (), open the Edit IICE Settings dialog box, and go to the IICE Sampler tab.

The command equivalent is `edit iice`.




3. In the dialog box, set the buffer type.
4. Specify sampling parameters.
 - Define the amount of data to be captured for each sampled signal in Sample Depth. The upper limit is determined by the capacity of the FPGAs being used, but the value must be a minimum of 8. The appropriate value is determined by the kind of debug you are planning to run. Deep trace debug (DTD) schemes could require more buffer memory. Also check the RAM usage to determine signal depth. If the design only uses some FPGA RAM resources, you can use RAM to capture signal data.
 - To sample a single step of data for selective viewing, enable Allow Qualified Sampling. Use this option when you want to follow the operation over a longer period of time.
 - Set other parameters as needed.

Defining the IICE Sample Clock

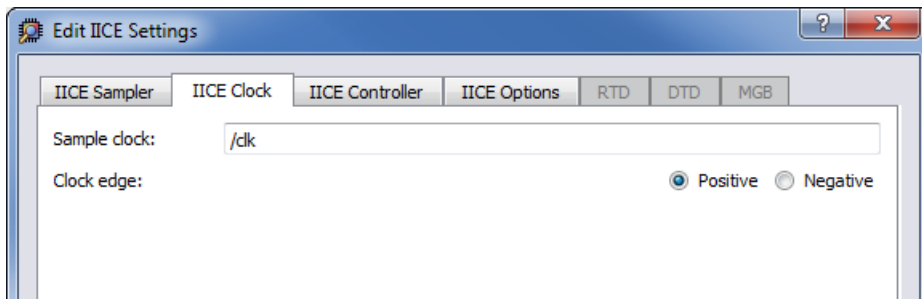
You must define a sample clock for each IICE. It determines when signal data is captured by the IICE. As with other parameters, you can do this through a command or from the GUI. For more information about the options mentioned below, refer to [IICE ClockTab, on page 136](#).

1. If there are multiple IICE units, first select the IICE.

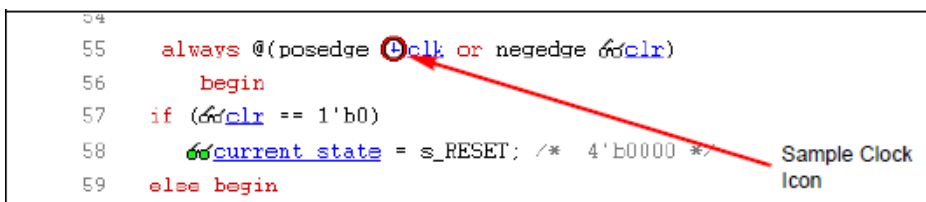
Go to the Control Panel and select the IICE to work on.

2. Click the Edit IICE icon (), open the Edit IICE Settings dialog box, and go to the IICE Sampler tab.

The equivalent command is `edit iice -gui`.



3. Specify the sample clock.
 - Refer to the guidelines ([Guidelines for Selecting a Sample Clock, on page 85](#)) to select a valid sample clock. Usually, the sample clock is either the clock that the sampled signals are synchronous with or a multiple of that clock.
 - To set your selection from the IICE Clock tab, enter the name of the signal in the Sample Clock field.
 - To select the sample clock from the RTL window, right-click the watchpoint icon for a signal (single bit) and select Sample Clock from the popup menu. The icon changes to a clock face as shown in the following figure. You cannot set the clock edge from this window.



4. Specify a positive or negative clock edge for sampling.

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 clock occurs, so they must be synchronous with the clock or a multiple of the clock.

The default is the positive edge.

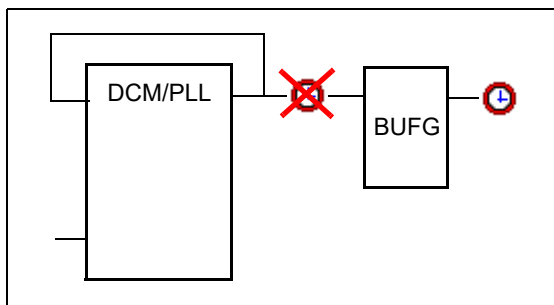
Guidelines for Selecting a Sample Clock

Follow these guidelines when selecting a sample clock for an IICE:

- The sample clock must be a single-bit scalar signal.
 - For the UC flow with \$dumpvars instrumentation, the sample clock must be a top-level clock.



- For post-compile SRS instrumentation, the sample clock must be a signal from the compiled database.
- Make sure that a clock buffer is assigned to the selected signal. The source of the sample clock must be the output from a BUFG/IBUFG device.



- Signals are sampled with respect to the sample clock so the clock must be synchronous or a multiple.

- The sample clock must be capable of being sampled by the instrumentor.
- If the IICE is to be cross-triggered from another IICE, the clocks for the affected IICE units must work together.

Working with Multiple IICE Units

See the following topics for information about working in a multi-IICE design.

- [Using Multiple IICEs for Multiple Instrumentation Modes](#), on page 87
- [Examples of \\$dumpvars Labels for Multi-IICE Grouping](#), on page 88

Using Multiple IICEs for Multiple Instrumentation Modes

If you use multiple modes of instrumentation in the same design, such as \$dumpvars and SystemVerilog assertions (SVA), it is advisable to route the different probe signals to separate IICE blocks. To do this, specify different IICE blocks with appropriate debug IP settings and clock instrumentation. You can specify the blocks in a single IDC file, or in individual IDC files.

1. In the IDC file, define separate IICE blocks for each instrumentation mode using the `iice new` command.

See [iice new](#), on page 76 for an overview of the syntax.

The following example shows how the `iice new` command is used to define two separate IICE blocks; one uses \$dumpvars for RTL instrumentation and the other uses assertions:

```
device jtagport builtin
device umr_pipe 1
iice new {IICE_DV} -type regular -mode {dumpvars}

device jtagport builtin
device umr_pipe 1
iice new {IICE_SVA} -type regular -mode {sva}
```

2. For \$dumpvars, use labels to group signals into individual IICEs.
 - Declare multiple IICEs in the IDC file with the `iice new` command.
 - In each RTL \$dumpvars definition, add a label that corresponds to an IICE name. Use the same label for the \$dumpvars that you want to group together, and at compile time they are grouped together in the same IICE. See [Examples of \\$dumpvars Labels for Multi-IICE Grouping](#), on page 88 for examples.

The following snippet adds \$dumpvars labels called `iice_bram` and `iice_dtd`:

```

initial
begin: iice_bram
$dumpvars(1,top.inst1.sig1,top.inst2.sig2,top.inst3.sig3);
end

initial
begin: iice_dtd
$dumpvars(1,top.inst4.sig1,top.inst5.sig2,top.inst6.sig3);
end

```

The IDC file contains matching IICE names. This ensures that the signals from the corresponding \$dumpvars are routed to the IICE with the matching name.

```

device jtagport builtin
iice new {iice_bram} -type regular
device jtagport builtin
iice new {iice_dtd} -type regular

```

Examples of \$dumpvars Labels for Multi-IICE Grouping

This section contains examples of multiple IICE units using \$dumpvars label-based grouping. The examples illustrate the various methods to declare multiple IICEs in the IDC using \$dumpvars labels in the RTL, and describes the results in different scenarios.

Multiple IICEs and \$dumpvars with the Same IICE and Label Names

This example has two IICEs and two \$dumpvars definitions with the same label names. In this scenario, the tool routes instrumentation from each \$dumpvars to the corresponding IICE name defined in the IDC file.

```

initial
begin: iice_bram
$dumpvars(1,top.inst1.sig1,top.inst2.sig2,top.inst3.sig3);
end

initial
begin: iice_dtd
$dumpvars(1,top.inst4.sig1,top.inst5.sig2,top.inst6.sig3);
end

device jtagport builtin
iice new {iice_bram} -type regular

```



```
device jtagport builtin
iice new {iice_dtd} -type regular
```

Multiple IICE Units and \$dumpvars with Different IICE and Label Names

This example has two IICEs and two \$dumpvars definitions with different label names. In this scenario, instrumentation from both \$dumpvars must be routed to each IICE, that is each IICE must have instrumentation from both the \$dumpvars.

```
initial
begin: iice_bram
$dumpvars(1,top.inst1.sig1,top.inst2.sig2,top.inst3.sig3);
end

initial
begin: iice_dtd
$dumpvars(1,top.inst4.sig1,top.inst5.sig2,top.inst6.sig3);
end

device jtagport builtin
iice new {iice_0} -type regular -mode {dumpvars}

device jtagport builtin
iice new {iice_1} -type regular -mode {dumpvars}
```

Multiple IICE Units and \$dumpvars with Escaped Characters in Names

This example has two IICEs and two \$dumpvars definitions, where the label and IICE names contain escaped characters. In this case, the instrumentor displays an error, as IICE names cannot have special characters.

```
initial
begin: \i$$e_uc
$dumpvars(1,top.inst1.sig1,top.inst2.sig2,top.inst3.sig3);
end

initial
begin: \i*_ce_sm
$dumpvars(1,top.inst4.sig1,top.inst5.sig2,top.inst6.sig3);
end

device jtagport builtin
iice new {i$$e_uc} -type regular

device jtagport builtin
iice new {i*_ce_sm} -type regular
```

Multiple IICE Units with Same Name, Single \$dumpvars

This example design contains two IICEs and one \$dumpvars definition. The IICEs have the same label and names but different configurations. As the IICE names are the same, the second IICE replaces the first IICE in the flow.

```
initial
begin: iice_bram
$dumpvars(1,top.inst1.sig1,top.inst2.sig2,top.inst3.sig3);
end

device jtagport builtin
  iice new {iice_bram} -type regular
  iice controller -iice {iice_bram} none
  iice sampler -iice {iice_bram} -depth 128
  iice sampler -iice {iice_bram} -pipe 3

device jtagport builtin
  iice new {iice_bram} -type regular
  iice controller -iice {iice_bram} none
  iice sampler -iice {iice_bram} -depth 1024
  iice sampler -iice {iice_bram} -pipe 6
```

Single IICE, Multiple \$dumpvars with Name Mismatch

This example design contains one IICE and two \$dumpvars definitions where the first label name matches the IICE name while the second \$dumpvars label is different from IICE name and the first \$dumpvars label name. In this case, instrumentation from the first \$dumpvars label must be routed to the IICE while the other \$dumpvars label is ignored.

```
begin: iice_bram
$dumpvars(1,top.inst1.sig1,top.inst2.sig2,top.inst3.sig3);
end

initial
begin: iice_dtd
$dumpvars(1,top.inst4.sig1,top.inst5.sig2,top.inst6.sig3);
end

device jtagport builtin
  iice new {iice_bram} -type regular
```

Multiple IICE Units and \$dumpvars: Labels with Different Cases

In this example, the design contains two IICEs and two \$dumpvars definitions with label names that match the IICE names but have different cases. In this case, each IICE will have instrumentation from both the \$dumpvars because the cases are different. That is, IICE_BRAM and IICE_DTD must be instrumented with signals coming from both the \$dumpvars labels.

```
initial
begin: iice_bram
$dumpvars(1,top.inst1.sig1,top.inst2.sig2,top.inst3.sig3);
end

initial
begin: iice_dtd
$dumpvars(1,top.inst4.sig1,top.inst5.sig2,top.inst6.sig3);
end

device jtagport builtin
iice new {IICE_BRAM} -type regular

device jtagport builtin
iice new {IICE_DTD} -type regular
```

Single IICE and \$dumpvars: Matching IICE and \$dumpvars Instance

In this example, the design contains one IICE and one \$dumpvars definition with matching label names and IICE names. Additionally, the instance name of the \$dumpvars definition is the same as the IICE name and label name.

This scenario is not supported, so no \$dumpvars will be created. It is recommended that you avoid using label names that match the \$dumpvars instance name, as shown in the example below.

```
initial
begin: top
$dumpvars(1,top);
end

device jtagport builtin
iice new {top} -type regular
```

Setting Triggers in the Instrumentor

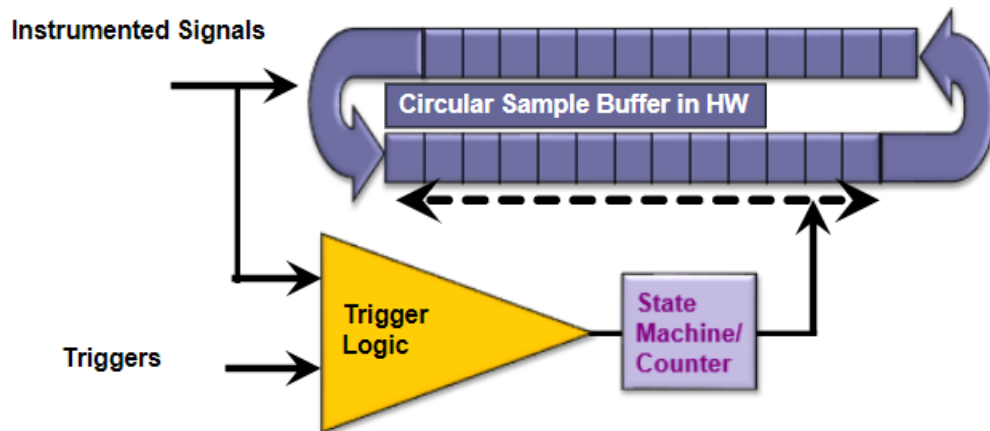
Triggering modes can be broadly classified as simple and complex. Simple mode is the default. Complex triggering modes use advanced triggering techniques that use counters or state machines. For example, state machine-based triggering sets triggers based on a complex sequence of events: “trigger if pattern A occurs exactly five cycles after pattern B, but only if pattern C does not intervene.”

The general methodology is to initially set up the trigger through the instrumentor, and then program and activate the trigger at the debug stage.

- [Setting IICE Trigger Options](#), on page 93
- [Setting up Simple Triggers](#), on page 94
- [Setting up Complex Counter Triggers](#), on page 95
- [Setting up State Machine Triggers](#), on page 97
- [Setting up an IICE for Cross-Triggering](#), on page 100
- [Setting up RTL Cross Triggers](#), on page 102

Trigger Logic and Sample Buffer

The trigger halts sampling, but not the hardware.




Setting IICE Trigger Options

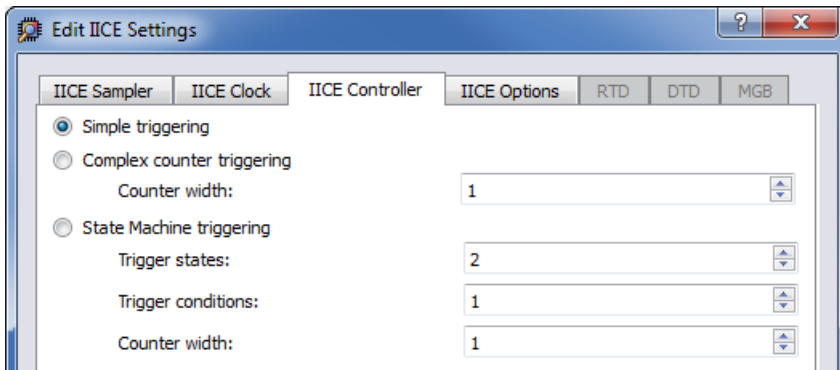
You must set triggering options for each IICE individually. All triggering options affect the area cost of the IICE.

1. If there are multiple IICE units, first select the IICE.

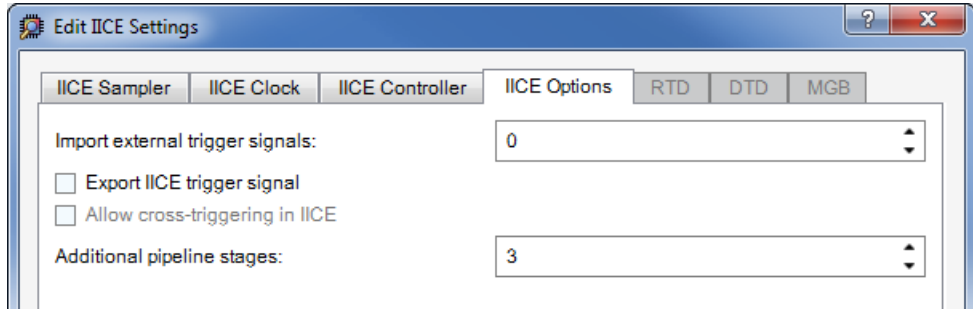
Go to the Control Panel and select the IICE to work on.

2. Click the Edit IICE icon (), open the Edit IICE Settings dialog box, and go to the IICE Controller tab.

The equivalent command is `edit iice -gui`. In the IDC file, the commands are `iice controller` commands.



3. Set the triggering mode: simple triggering, complex counter triggering, or state machine triggering.
 - Simple triggering allows you to combine breakpoints and watchpoints to create a trigger condition for capturing the sample data. See [Setting up Simple Triggers, on page 94](#).
 - Complex-counter triggering augments the simple triggering by instrumenting a variable-width counter that can be used to create a more complex trigger function. See [Setting up Complex Counter Triggers, on page 95](#).
 - State-machine triggering allows you to pre-instrument a variable-sized state machine that can be used to specify an ultimately flexible trigger condition. See [Setting up State Machine Triggers, on page 97](#).
4. For cross-triggering, set trigger details on the IICE Options tab.



Cross-triggering allows a source external to the IICE to be the trigger. You can specify the external source to be another IICE or external logic on the board rather than the result of instrumentation.

Setting up Simple Triggers

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

1. In the instrumentor, designate signals as watchpoints and breakpoints.

With simple triggering, the tool compares signals to values (including don't cares) and triggers when the signals match the values. This scheme can be enhanced by combining it with breakpoints to denote branches in control logic. If a breakpoint is enabled, that particular branch must be active at the same time that the signals match their respective values.

Follow these trigger logic rules when setting up watchpoints and breakpoints:

- All signals must match their respective comparison values in order to trigger.
- All breakpoints must be OR-connections, so that any one enabled breakpoint is enough to be the trigger.
- Combine signals and breakpoints with AND; all signals must match their values AND at least one enabled breakpoint must occur.

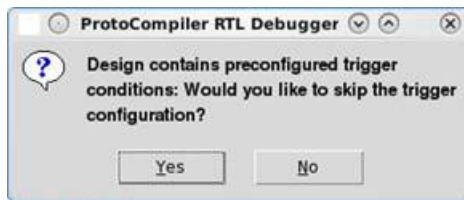
You can now enable the trigger when you debug the design.

2. To preconfigure a trigger for debugging initial cycles, add the preconfigure argument when you designate signals with the breakpoints, watchpoints, or signals commands in the instrumentor.

For example:

```
breakpoints preconfigure /case_88/if_90/alu.v:72 -condition {1  
3} -state 0
```

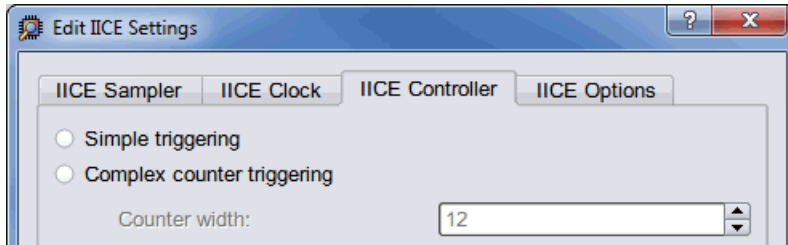
Preconfiguring sets specific trigger conditions in advance for the debugger, and arms the IICE to run immediately after configuration. This lets you trap error conditions that might occur during the first several cycles. When you start the debugger with preconfigured signals, a prompt asks if you want to run them. To download the sample data with the pre-configured trigger, select Yes. To ignore the pre-configured trigger and enable normal sampling, select No.



Setting up Complex Counter Triggers

A complex counter connects the output of the breakpoint and watchpoint event logic to the sampling block and allows the user to implement complex triggers. Complex-counter triggers augment simple triggering by instrumenting a variable-width counter that can be used to create a more complex trigger function.

1. Define a complex counter in the instrumentor.
 - In the instrumentor, use the iice controller command or go to the IICE Controller tab of the Edit IICE Settings dialog box to create the counter.



- Enable Complex counter triggering.
- Set the counter width, which determines the width for the counter. The default is 16, which implements a 16-bit counter with a range from 0 to 65535. If you set the counter width to 0, no counter is inserted in the design.

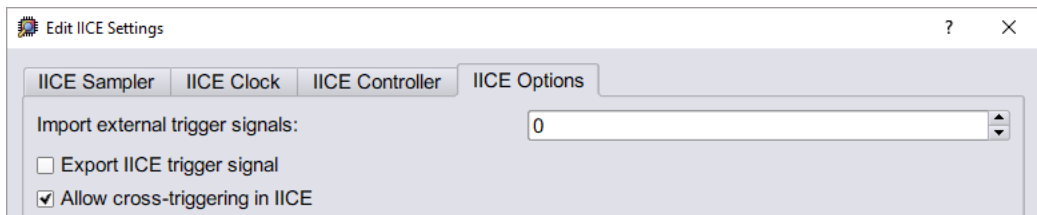
The equivalent commands are shown below:

```
iice controller counter 1
iice controller counterwidth <value>
```

2. Set other options for the counter trigger.

- To enable cross triggering, use the following instrumentor command or enable the Allow cross-triggering option on the IICE Options tab, as shown in the figure below.

```
iice controller -crosstrigger 1
```



- On the IICE Sampler tab, set the buffer type. Configure the sample depth for the buffer with the following instrumentor command:
- Set the sampling mode on the same tab. For example, you can enable always-armed triggering.

```
iice sampler -depth 2048
```

3. Add the counter to the design.

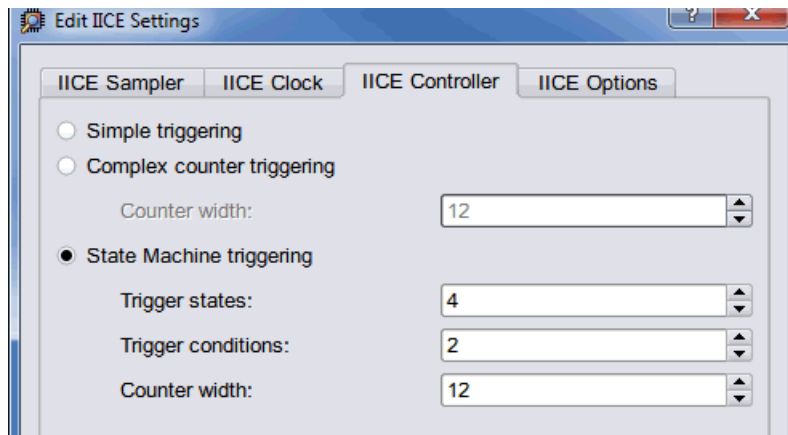
You can now activate the trigger when you debug the design.

Setting up State Machine Triggers

State machine triggers are based on a variable-sized state machine that specifies a flexible trigger condition. With this method, you use sequences of trigger conditions to create counter-based state machines and create very effective triggers. 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 from the FPGA by the debugger executable.

Set up the state machine trigger during instrumentation and then program the state machine dynamically during debug, to create a complex, design-specific trigger. Trigger conditions can be set at the instrumentation stage as described here, but can also be set dynamically in the debugger.

1. Open the dialog box.
 - Select Instrumentor->IICE->Edit IICE from the menu bar or click the Edit IICE icon.
 - In the instrumentor Edit IICE Settings dialog box, go to the IICE Controller tab.



2. Enable State Machine triggering.

This allows you to pre-instrument a variable-sized state machine that can be used to specify an ultimately flexible trigger condition. The command equivalent is `iice controller statemachine`.

3. Specify the number of trigger states, trigger conditions, and the counter width in the corresponding fields.

- Use Trigger states to define the number of states available in the state machine. To use a state machine for cross-triggering it must have at least three states. The range is 2 to 16; powers of 2 are preferable as other numbers limit functionality and do not provide any cost savings.

This is the most cost-critical setting, because each trigger condition results in an additional address bit on the RAM and doubles the size of the RAM table with each bit.

- In Trigger condition, specify the number of independent trigger conditions in the state machine. The range is from 1 to 16. All trigger conditions are identical in terms of signals and breakpoints connected to them, but they can be programmed separately in the debugger.
- If the state machine includes a counter, use Counter width specify a value greater than 0 for the desired width of the counter. A counter in the state machine augments the functionality of the state machine, just as with a simple trigger. Counter width is a direct contributor to the width of the RAM table, and this might be significant for FPGA RAM primitives that allow a trade-off of width for depth.

See [State Machine Implementation for Triggering, on page 99](#) for details about the implementation.

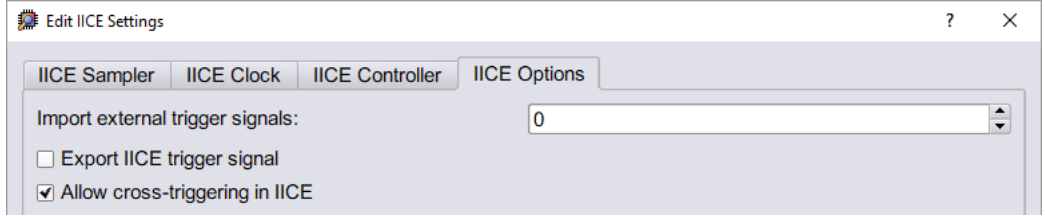
This example shows a sequence of commands that sets up a state machine for triggering, with two trigger conditions and two trigger states.

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

4. Set other options for the state machine trigger.

- To enable cross triggering, use the following instrumentor command or enable the Allow cross-triggering option on the IICE Options tab, as shown in the figure below.

```
iice controller -crosstrigger 1
```



- On the IICE Sampler tab, set the buffer type to `internal_memory`. Configure the sample depth for the buffer with the following instrumentor command:

```
iice sampler -depth 2048
```

- Set the sampling mode on the same tab. For example, you can enable qualified sampling or always-armed triggering.

5. Check the implementation to ensure that resources are being used effectively.

For example, it may be advantageous in a particular situation to trade away states for additional trigger conditions or for increased counter width. You can make these changes at the debug stage as well.

6. Add the state machine to the design.

You can now activate triggers when you run debug.

State Machine Implementation for Triggering

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

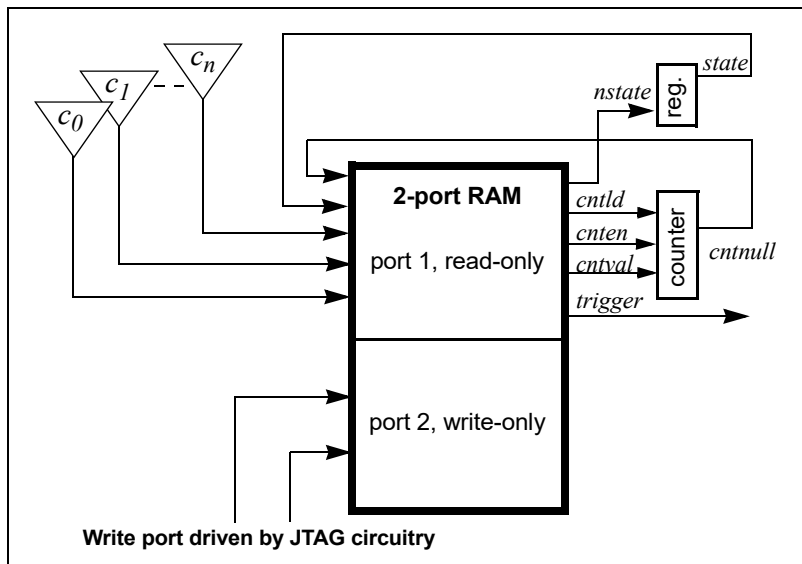
If the state machine includes a counter, the counter output is compared to constant 0, and the single-bit output of that comparison is also connected to the address inputs of the same RAM table. The other address inputs are provided by the state register.

The 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).

The outputs of the RAM table are the next-state value *nstate*, and the trigger signal *trigger* (which causes the sample buffer to take a snapshot if high).

If the state machine includes a counter, there are additional outputs:

- The counter-enable signal *cnten* (if '1', the 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*)



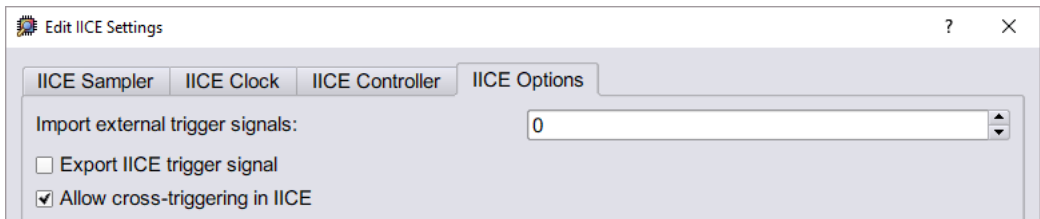
Setting up an IICE for Cross-Triggering

Cross-triggering allows the trigger from one IICE unit to qualify a trigger on another IICE unit, even when the two IICE units are in different clock domains. The IICE to be cross-triggered is called the *destination* IICE and the IICE initiating the trigger is called the *source* IICE. Multi-IICE cross-triggering provides a way for subsystems in an SOC to interact and synchronize by exchanging debug triggers. You can cross-trigger between IICE units on different FPGAs.

You can use cross-triggering with simple triggers and complex counters. State machines support cross triggering by allowing the IICE unit IDs to be included in state machine equations.

The following procedure describes how to set up an IICE for cross-triggering. This method is useful to cross-trigger IICEs that are at same level, such as system-level IICEs or IICE units in the same partitioned FPGA. For IICEs at different levels, use the method described in [Setting up RTL Cross Triggers, on page 102](#).

1. Define the IICE in the instrumentor.
2. For a destination IICE, enable cross-triggering in the instrumentor.
 - From the GUI, enable cross-triggering by going to the IICE Options tab of the Edit IICE Settings dialog box, and enabling the Allow cross-triggering in IICE check box:



This is the equivalent command:

```
iice controller -iice {IICE_0, IICE_1, ..., IICE_n} -crosstrigger
```

3. Instrument the signals in each IICE and complete the Bit file generation flow
4. Program the hardware and open the ProtoCompiler Runtime tool.
5. Activate cross-triggering.
 - In Runtime, open Setup Debugger > Instrumentation > Cross trigger mode.
 - From the drop-down list, select the IICE and activate or deactivate cross triggering as required.
6. Set required watchpoints and breakpoints.
7. Run the Debugger tool and observe the waveforms.

Setting up RTL Cross Triggers

This method sets up cross-triggering by instantiating IICE blocks in the RTL and then instrumenting the trigger_out signal of an IICE in other IICEs. The RTL cross-trigger method is well-suited to triggering IICEs that are at different levels. For example, use it cross-trigger between a system IICE and a partitioned FPGA IICE, or for cross-triggering IICEs in multiple partitioned FPGAs. For another method of cross-triggering, see [Setting up an IICE for Cross-Triggering](#), on page 100.

1. Instantiate IICE blocks in the RTL.

Do this by adding the `<installDir>/lib/di/haps_controlled_clocks.v` IICE definition file to the source list which is passed to the run compile command.

```
* syn_noprune, syn_keep *) wire trigger_o_IICE_1, trigger_o_IICE_2, trigger_o_IICE_3 ;
* syn_noprune *) iice_trigger #("IICE_1") trigger_inst_IICE_1 (trigger_o_IICE_1);
* syn_noprune *) iice_trigger #("IICE_2") trigger_inst_IICE_2 (trigger_o_IICE_2);
* syn_noprune *) iice_trigger #("IICE_3") trigger_inst_IICE_3 (trigger_o_IICE_3);
```

2. In the instrumentor, instrument the trigger outputs of the IICE blocks in each other.

The following IDC file example illustrates this:

```
iice new {IICE_1} -type regular
iice controller -iice {IICE_1} none
iice sampler -iice {IICE_1} -depth 2048
iice clock -iice {IICE_1} -edge positive {/SRS/osc_ext}
signals add -iice {IICE_1} -sample -trigger {/SRS/reset_n}\
{/SRS/u_cnt_demo_inst1/capture_flag}\
{/SRS/trigger_o_IICE_2}\
{/SRS/trigger_o_IICE_3}\
{/SRS/trigger_o_IICE_4}
```

3. Partition the design such that the IICE blocks and the trigger outputs goes into the respective FPGAs where the instrumented signals are partitioned.
4. Generate bit files for each FPGA.
5. Export the design.
 - Use export runtime command in the ProtoCompiler tool.
 - If the IICEs are at different FPGA levels, there is separate debug project (debug.prj) created for each FPGA.

6. Program the hardware.
7. If an IICE is to be cross-triggered, open the debug project in the Runtime instance of that IICE and set watchpoints for the trigger-out signal of other IICE that is instrumented in the exiting IICE.
8. Run the debugger. The IICE will move to the waiting for trigger state as the trigger from other IICEs has not been triggered yet.
9. Open another Runtime session and open the debug project for the IICE that is to be used to crosstrigger the previous running IICE which was in waiting for trigger state.
10. Set the watchpoints for user signals and run the debugger tool.

After the trigger is detected, a new debugger session is opened. The previous Debugger session also gets triggered and the waveforms are captured in it.

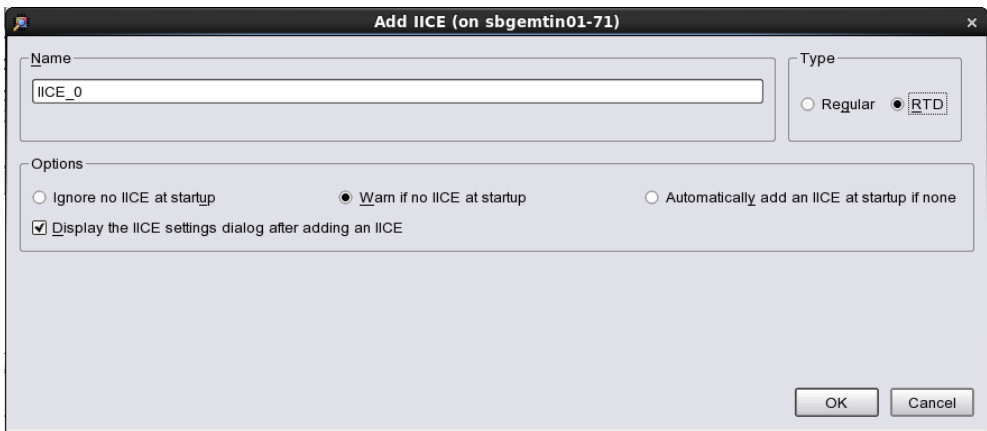
Limitations

- For IDC cross-triggered IICE of BRAM type at the same level (system or SLP):
 - There is latency of five clock cycles in Runtime waveform if the IICE type is Simple or Complex counter triggering.
 - There is latency of four clock cycles in Runtime waveform if the IICE type is State machine triggering.
- For IDC cross-triggered IICE of DTD type, the latency is unknown because runtime isn't aware of DTD IICE link delay. But you can capture the correct sample and waveforms after cross-triggering.
- (*HAPS-100 Only*) For RTL cross triggered ADTD IICE from BRAM, there is latency of 11 clock cycles in Runtime waveform for ADTD IICE.

Setting up for Real-Time Debug (RTD)

Real-time debugging is a feature lets you access instrumented signals directly through a Mictor board interface connector installed on the HAPS board.

1. Set up the hardware with a Mictor daughter card and a HapsTrak3-to-HapsTrak II adapter.
2. Add an IICE for RTD.
 - To specify the IICE from the user interface, click the Add IICE icon. Select the RTD radio button and click OK.

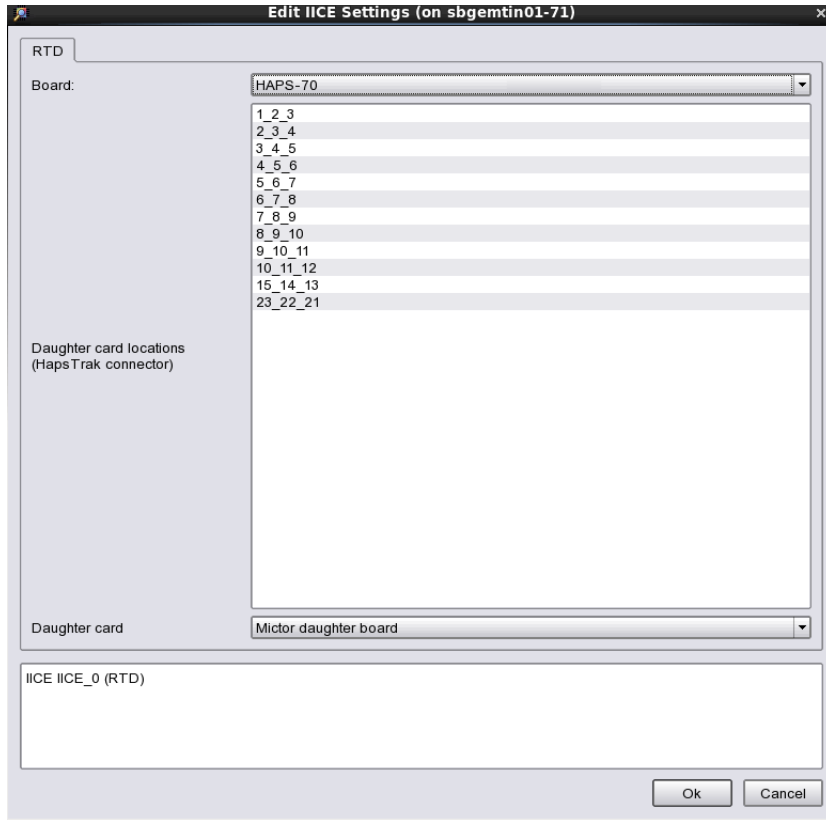


- To define the IICE from the TCL Script shell, enter the `iice new` command:

`iice new [iice/D] -type rtd`

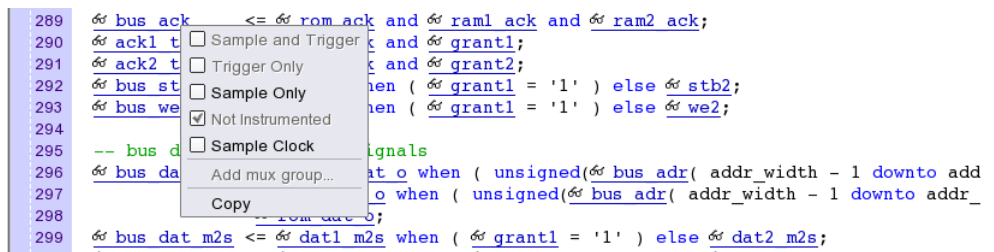
This creates a new, real-time IICE for the design with un-instrumented signals.

3. Select the Mictor daughter card locations from the Edit IICE->RTD tab.



4. Instrument signal watchpoints and breakpoints as usual.

You can only set the watchpoints to Sample only.



You can run RTD when you debug the design.

CHAPTER 4

HAPS Deep Trace Debug

HAPS Deep Trace Debug (DTD) provides a mechanism to use memory beyond FPGA memory as storage for debug samples. Using external memory makes it possible to have a deeper storage buffer with more capacity for the debug signal data.

The following topics describe how to use DTD with various HAPS systems.

- [Setting up DTD for HAPS-100 Designs](#), on page 108
- [Setting up DTD for HAPS-80 Designs](#), on page 116
- [Setting up DTD for HAPS-70 Designs](#), on page 119

Setting up DTD for HAPS-100 Designs

HAPS-100 systems include built-in DDR4 memory on daughter cards; the daughter cards are installed by default on each FPGA in a HAPS-100 system. You can use the DDR memory to capture debug samples. This expands the storage capacity, and makes larger visibility windows possible by using external memory instead of onboard resources like BRAM.

The following topics describe the DTD schemes available for HAPS-100 systems:

- [Specifying Built-in Memory for DTD with HAPS-100 Designs](#), on page 108
- [Specifying DTD Buffers for HAPS-100 Multi-FPGA Designs](#), on page 110
- [Limitations to HAPS-100 DTD](#), on page 114

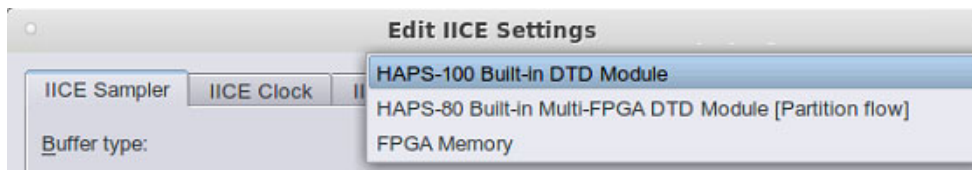
Specifying Built-in Memory for DTD with HAPS-100 Designs

The following procedure describes how to use the built-in DDR4 memory on HAPS-100 systems as DTD buffer memory. Use this methodology for single-FPGA designs. For multi-FPGA designs, refer to [Specifying DTD Buffers for HAPS-100 Multi-FPGA Designs](#), on page 110.

1. Specify the DTD buffer when instrumenting the design.

For multi-FPGA designs, select the DTD buffer type in the Instrumentor at the system-level partitioning stage; for single-FPGA designs select the buffer type at the FPGA level after partitioning.

- Open the Edit IICE Settings dialog box and set the buffer type to HAPS-100 Built-in DTD Module.



- Alternatively, type the equivalent command:

```
iice sampler -iice <iice_name> haps100_DTD_builtin
```

This setting specifies the dedicated debug memory which has a higher capacity and bandwidth. Unlike the built-in memory on HAPS-80 systems, no FPGA connectors are used. All MIG and trigger information data are also stored in the dedicated memory to minimize impact on the user FPGA.

2. Set parameters for the DTD buffer.

- The maximum number of instrumented bits that can be sampled with DTD per FPGA is 32768. The sample frequency is based on the number of instrumented bits. Use the following guidelines to optimize performance:

Instrumented Bits	Working Maximum Sampling Frequency
(Beta) Up to 32768	2.5 MHz
(Beta) Up to 16384	5 MHz
Up to 8192	10 MHz
Up to 4096	20 MHz
Up to 2048	40 MHz
Up to 1024	80 MHz
Up to 512	160 MHz

To instrument signals beyond 8192 bits, make sure:

- You have specified the `% iice sampler -max_width {16384|32768}` command in the IDC file.
- You do not have more than 8k signals of Sample and Trigger. The Trigger only option is not supported and gets converted to Sample and Trigger.
- You have the remaining signals, other than the 8k signals specified above, as Sample only.

See [iice](#) command in HAPS Prototyping Debugging Environment Reference manual.

Based on the selected buffer type and configuration settings, the tool automatically calculates the sample depth and source clock. If you do not specify otherwise, the tool uses a depth of 256 and simple triggering for the controller mode as the defaults for the DTD. At

runtime, the configured sample depth can be dynamically edited from the minimum depth to the maximum configured depth.

You can have 8192 bits per FPGA of Sample and Trigger type and all other with Sample Only type. The Trigger Only signals are not supported in haps100_DTD_builtin buffer type. They are converted to Sample and Trigger type.

- To capture samples for multi-FPGA designs at the system-level compile or pre-partition phases, see [Specifying DTD Buffers for HAPS-100 Multi-FPGA Designs, on page 110](#).
3. After instrumentation, synthesize, place, and route the design as usual, and program the design onto the HAPS-100 system.

You can now debug the instrumented samples.

Example: HAPS-100 Single-FPGA DTD Using Built-in Memory

The memory used to store samples is the dedicated debug memory. The capacity is larger, so you can set a larger sample depth.

```
device jtagport umrbus3
iice new {IICE_0} -type regular
iice sampler -iice {IICE_0} haps100_DTD_builtin
iice sampler -iice {IICE_0} -depth 512
iice sampler -iice {IICE_0} -pipe 3

signals add -iice {IICE_0} -silent -trigger
{/SRS/counter_top_0/counter_out_2[47:0]}

iice clock -iice {IICE_0} -edge positive {/SRS/clk_in}
```

Specifying DTD Buffers for HAPS-100 Multi-FPGA Designs

The following procedure describes how to use the built-in DDR4 memory on HAPS-100 systems as DTD buffer memory. Use this methodology for multi-FPGA designs. For single-FPGA designs, refer to [Specifying Built-in Memory for DTD with HAPS-100 Designs, on page 108](#).

The HAPS-100 multi-FPGA flow for DTD buffers is similar to the flow with BRAM buffers, but differs in the maximum number of instrumented bits used per FPGA. In addition to the user-instrumented bits per FPGA, there are 18 bits of user bandwidth to transfer through links to assist the FPGA. Refer to the table below to determine the maximum sampling frequency, based on the

number of bits that can be instrumented per FPGA. The maximum number of instrumented bits that can be sampled per FPGA in a HAPS-100 system with DTD, is $32768 - 18 = 32750$ bits.

Instrumented bits (per FPGA)	Maximum sampling frequency
(Beta) Up to $32768 - 18 = 32750$ bits	2.5 MHz
(Beta) Up to $16384 - 18 = 16366$ bits	5 MHz
Up to $8192 - 18 = 8174$ bits	10 MHz
Up to $4096 - 18 = 4078$ bits	20 MHz
Up to $2048 - 18 = 2030$ bits	40 MHz
Up to $1024 - 18 = 1006$ bits	80 MHz

1. At the compile or pre-partition stage, create an instrumentation with the DTD buffer type. For the example IDC file, see [Example: HAPS-100 Single-FPGA DTD Using Built-in Memory](#), on page 110.
2. Run the design through the usual partition flow.
3. Launch individual FPGA runs with place and route, and generate FPGA bit files.

When you open the project in runtime, you can use the instrumentation and storage for debug.

Viewing DTD Samples at Runtime

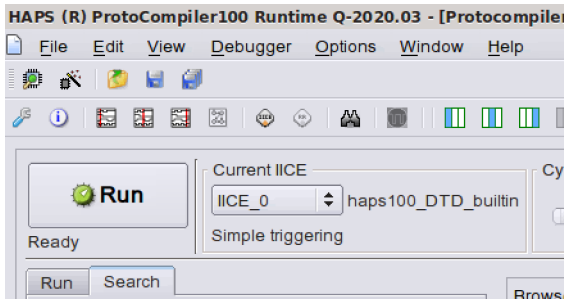
Once you have completed the steps described in [Specifying DTD Buffers for HAPS-100 Multi-FPGA Designs](#), on page 110, follow the procedure below to use the buffer to debug the samples.

1. Start runtime, and open the project.
 - Launch runtime:


```
% protocompiler100_runtime &
```
 - In the window that opens select the appropriate project (*debug.prj*) according to the design flow (single-FPGA or multi-FPGA).
2. Select the buffer and run link training.

- Select the DTD IICE that was enabled during instrumentation. Do this from the main runtime window as shown in the following figure, or use this equivalent command:

```
iice current <iice_name>
```



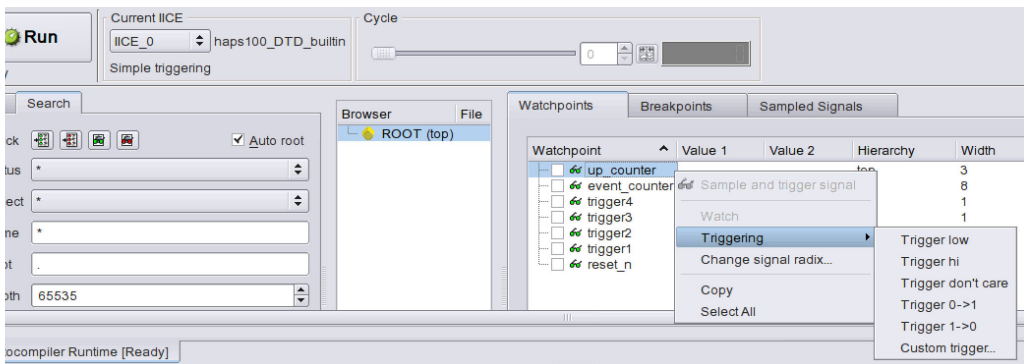
- Run link training for DTD IICE using this command:

```
iice link -train
```

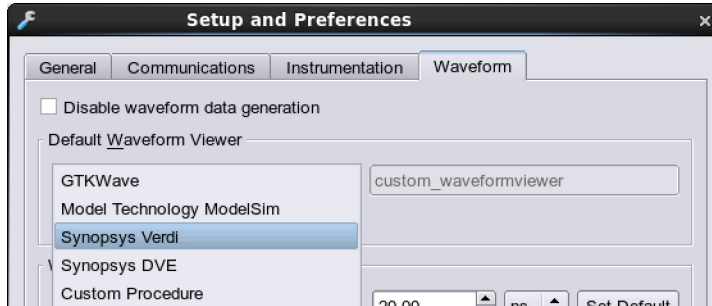
3. Enable watchpoints as a trigger condition.

Do this from the Watchpoints tab in the GUI as shown below, or with this command:

```
watch enable -language verilog {<signal_names>}
{trigger_condition_values}
```



4. Go to Setup Debugger -> Waveforms and specify the viewer and format to use.



5. Start debug by clicking Run in the GUI or entering this command in the Tcl shell for the debugger:

```
run -iice <iice_name> -wait -timeout <n>
```

In this command, *n* represents the time within which the trigger event occurs.

6. View and analyze signals in the waveform viewer that opens.

The waveforms are VCD or FSDB, according to the waveform type you chose. You can also explicitly write out VCD or FSDB, using the `-range` argument with the appropriate Tcl command:

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

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

Link Training Messages

This section provides messages that may appear after you run the link training command `iice link -iice <IICE_NAME> -train` on HAPS-100.

1. Link training success messages:

```
#####
Auto-detecting the device chain...
ADTD link training start
DTD core reset for FB1.uA
Poll for channel up
Waiting on link training to finish...
```

```

Links are successfully trained!
Connect to hardware...
Setting up hardware driver ...
ok.
#####

```

2. Link training failure messages:

```

Waiting on link training to finish...
\ Channel UP L1 Pass SCLK DTD reset L2 Pass L3 Pass Link training
done
-----
FB1.uA yes          yes      yes    yes no          no          no
=====
Fpga Name(s)  Error messages and handling
-----
FB1.uA        L2 fails.
A failure means test patterns are missing with the user sampling
frequency.
Causes:

                Higher than recommended sampling frequency.
                Timing violations in DTD IP in user design.
=====
Error: Link training check on iice 'IICE_0' hubs { FB1.uA } failed.
Possible cause link training was not executed since last board
configuration, or it has failed.
Please rerun link training with command 'iice link -train' and
check log in case of failure.

```

Limitations to HAPS-100 DTD

Hardware Limitations:

- Multiple DTDs at the same level partition (System or SLP) is not allowed.

Software Limitations:

- Does not support the following features:
 - Data compression for DTD
 - Always armed triggering
 - Qualified sampling
 - Mux Groups 1 and 2

- IDC cross-trigger for DTD buffer type. Use the RTL cross-triggering method.
- There is known latency in DTD when it is cross triggered by BRAM, depending on the number of trigger conditions.

Setting up DTD for HAPS-80 Designs

With HAPS-80 systems, sample data is stored outside the FPGA and does not consume any BRAM resources. You can use external DDR memory or use the built-in DDR resources:

- [Setting up HAPS-80 DTD with External DDR Memory](#), on page 116
- [Setting up HAPS-80 DTD with Built-in Memory](#), on page 117

Setting up HAPS-80 DTD with External DDR Memory

For single-FPGA designs, as for IP or multi-design mode (MDM), use external DDR memory for storage, as described below:

1. Set the IICE buffer type to `haps_dtd`.
 - For single-FPGA designs, set it to `haps_dtd`. Use this setting when working with IP or for multi-design mode (MDM). This mode uses external DDR memory.
 - To use the built-in memory, set the buffer type to `haps_dtd_builtin`.
 - For built-in memory, also specify the DTD clock.
2. Specify the depth of the memory buffer.
3. Specify the DDR memory to use for storing samples.
 - Specify the system.
 - Specify the location.
4. Instrument the signals and configure triggers.

Once you complete synthesis and place and route, you are set up to use the signals and storage memory when you debug the design.

Example: HAPS-80 DTD with External Memory

```
device jtagport umrbus
iice new {IICE_0} -type regular
iice controller -iice {IICE_0} none
iice sampler -iice {IICE_0} haps_DTD
iice sampler -iice {IICE_0} -ram {board HAPS-80}
iice sampler -iice {IICE_0} -ram {locations {1_2_3}}
iice sampler -iice {IICE_0} -ram {type DDR3_8G}
iice sampler -iice {IICE_0} -depth 128
iice sampler -iice {IICE_0} -pipe 3

signals add -iice {IICE_0} -silent -trigger
{/SRS/counter_top_0/counter_out_2[47:0]}

iice clock -iice {IICE_0} -edge positive {/SRS/clk_in}
```

Setting up HAPS-80 DTD with Built-in Memory

The following procedure shows how to use the built-in DDR memory on HAPS-80 systems for sample storage during debug.

1. Set these buffer parameters:
 - Set the buffer type to `haps_dtd_builtin`. This setting uses the built-in DDR memory mounted on A1, A2, and A3.
 - Also specify the DTD clock and reset.
2. Specify the depth of the memory buffer.
3. Specify the DDR memory to use for storing samples.
 - Specify the system.
 - Specify the location.
4. Specify the MGB connectors to use for debug.
5. Instrument the signals and configure triggers.

Once you complete synthesis and place and route, you can start runtime. After you run link training, you can use the instrumentation and storage setup to debug the design.

Example: HAPS-80 Built-in DTD

```
device jtagport umrbus
iice new {IICE_0} -type regular
iice controller -iice {IICE_0} none
iice sampler -iice {IICE_0} haps_DTD_builtin
iice sampler -iice {IICE_0} -ram {debugboard HAPS-80_S52}
iice sampler -iice {IICE_0} -dtd_clock {FB1.GCLK3}
iice sampler -iice {IICE_0} -dtd_reset_type {AUTO}
iice sampler -iice {IICE_0} -dtd_reset_bins {FB1.uA FB1.uB}
iice mgb -iice {IICE_0} -add_hub {FB1.uA}
iice sampler -iice {IICE_0} -depth 131072
iice sampler -iice {IICE_0} -pipe 3

signals add -iice {IICE_0} -silent -trigger -sample
{/SRS/counter_top_0/counter_out_2[47:0]}\
{/SRS/data_mux_top_0/in_data_reg[47:0]}\
{/SRS/nreset}

iice clock -iice {IICE_0} -edge positive {/SRS/clock}
```

Setting up DTD for HAPS-70 Designs

The following topics describe ways to use DTD with HAPS-70 designs:


- [Setting up DDR3 Memory for DTD with HAPS-70 Designs](#), on page 119
- [Setting up DTD for Multi-FPGA HAPS-70 Designs](#), on page 121

This is the expanded memory used for HAPS-70 and HAPS ProtoCompiler DX7 designs:

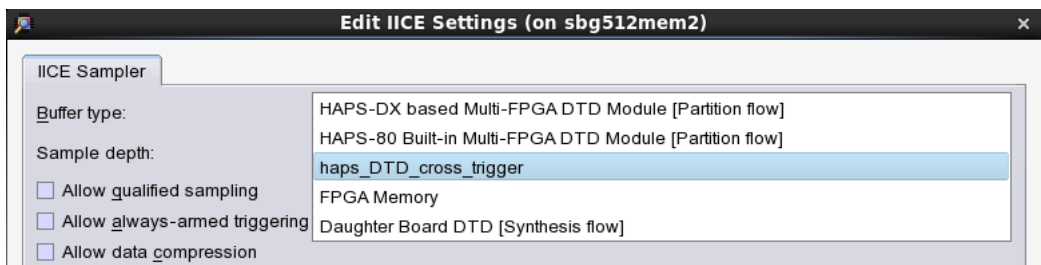
- DDR3_SODIMM_HT3 (4GB single-rank daughter board)
- DDR3_SODIMM2R_HT3 (8GB dual-rank daughter board)
- Multi-FPGA DTD Module

Setting up DDR3 Memory for DTD with HAPS-70 Designs

The following steps provide an overview of configuring deep trace debug using either the on-board DDR3 memory card (HAPS ProtoCompiler DX) or a DDR3 daughter board installed on a HAPS-70 series system. Use the procedure described here with a single-FPGA design.

1. Click the Edit IICE icon () in the instrumentor GUI to open the IICE Sampler tab.

The equivalent command is `edit iice`.



2. Set buffer parameters.

- Set the Buffer type to Daughter Board DTD. ‘
- Make sure that the Sample depth field is set correctly. Once set in the instrumentor, the depth cannot be changed in the debugger.

3. Go to the DTD tab and set additional parameters.
 - Set the Memory module type from the drop-down menu. The ONBOARD DX/DDR3 SODIMM 2R HT3 (8GB) setting is used by both the HAPS ProtoCompiler DX and the HAPS ProtoCompiler products; the DDR3 SODIMM HT3 (4GB) applies to only HAPS ProtoCompiler systems.
 - Select the HapsTrak connector locations where the daughter board is physically installed. Daughter boards are installed in three adjacent connectors. To optimize performance, make sure that the selected connector location set aligns with an SLR layer (for example, use 1_2_3 or 4_5_6 instead of 2_3_4 or 3_4_5).
4. Save the IICE settings and save the instrumented design.

You can close the instrumentor.

5. Continue with FPGA synthesis and place and route as usual.


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

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

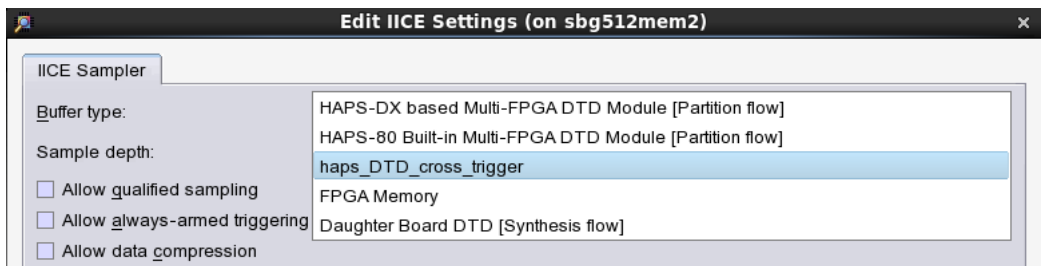
Instrumented Bits	Max Sample Frequency	Max Sample Depth (4GB single rank)	Max Sample Depth (8GB dual rank)
1 to 250	140 MHz	134,217,727	268,435,455
251 to 506	70 MHz	67,108,863	134,217,727
507 to 1018	35 MHz	33,554,431	67,108,863
1019 to 2042	17.5 MHz	16,777,215	33,554,431

Setting up DTD for Multi-FPGA HAPS-70 Designs

The following steps provide an overview of configuring deep trace debug using the Multi-FPGA DTD module with a HAPS-70 series system:

1. Click the Edit IICE icon () in the instrumentor GUI to open the IICE Sampler tab.

The equivalent command is `edit iice`.



2. Set buffer parameters.
 - Set Buffer type to Multi-FPGA DTD Module.
 - Make sure that the Sample depth field is set correctly. Once set in the instrumentor, the depth cannot be changed in the debugger.
3. Go to the MGB tab and set additional parameters.
 - Specify the MGB connectors.
 - Specify the DTD reset to use for training. Set it to Default via HT3 to distribute the reset via HapsTrak[®] connectors. Set it to Auto via GCLK to use a global routing resource to distribute the reset signal, with the FPGA source identified by the DTD2 Reference Clock parameter. If you use a GCLK for the DTD2 reset, you must define the source of the GCLK clock net as `fpga` in the TSS file, and not as `pll`.
 - Specify a DTD clock.
4. Save the IICE settings and save the instrumented design.

You can close the instrumentor.
5. Continue with the rest of the design flow as usual.

When you debug the design later, the tool automatically calculates the sample depth and source clock based on the configuration settings you supplied. The configured sample depth can be varied dynamically in the debugger from the minimum depth to the maximum configured depth. See [Viewing Captured Deep Trace Debug Samples, on page 32](#) of the *Debugger User Guide*.

The guidelines in the table below determine the maximum sample frequency based on the number of signals per MGB (multi-gigabit) link. The maximum number of signals per link is 256.

Signals Per MGB Link	MGB Connection Mux Ratio	Maximum Frequency	Max Sample Depth
1 to 32	1	70 MHz	268,435,455
33 to 64	2	35 MHz	134,217,727
65 to 128	4	17.5 MHz	67,108,863
129 to 256	8	8.74 MHz	33,554,431

CHAPTER 5

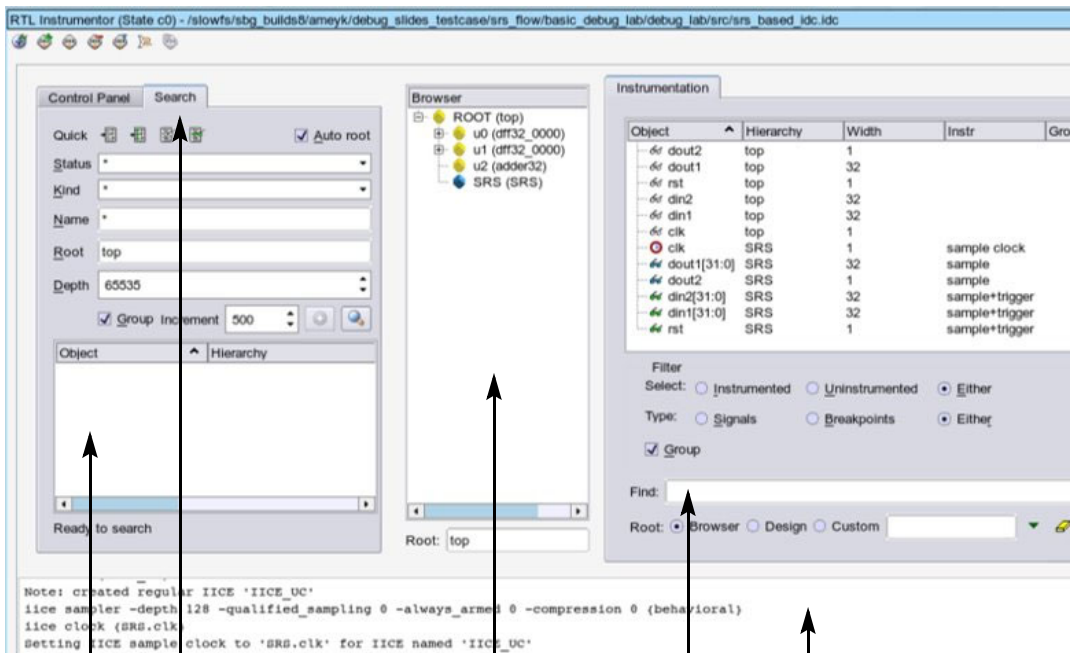
The Instrumentor GUI

The following topics describe the instrumentor GUI:

- [Instrumentor Windows and Views](#)
- [Instrumentor Preferences](#)
- [Edit ICE Settings](#)

Instrumentor Windows and Views

The Graphical User Interface (GUI) has these areas:



Search

Hierarchy Browser

Console / Shell Window

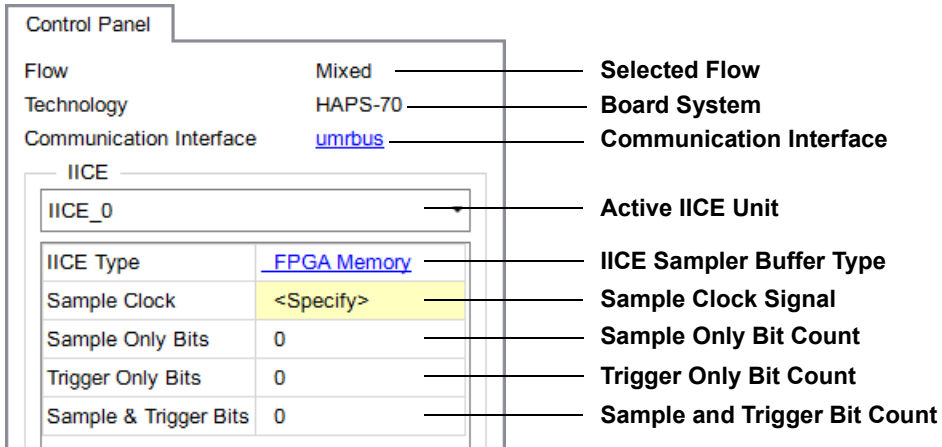
Control Panel

Instrumentation Tab / Signals View

- [Control Panel](#), on page 124
- [Search Panel](#), on page 125
- [Hierarchy Browser](#), on page 127
- [RTL Tab](#), on page 128
- [Instrumentation Tab](#), on page 129

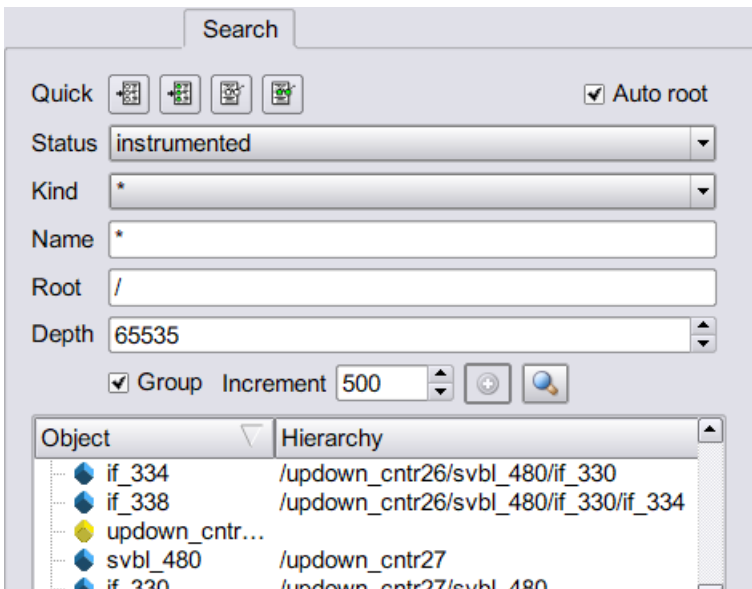
Control Panel

The Control Panel tab describes the current status of the instrumented design. Note that some entries are dependent on the IICE sampler buffer type selected.







Search Panel

The Search panel is a general utility to search for signals, breakpoints, and/or instances. The panel includes an area for specifying the objects to find and an area for displaying the results of the search. .



Search Panel Icons

- List Instrumented Signals 
Lists all signals currently instrumented in the entire design
- List Signals Available for Instrumentation 
Lists only signals available for instrumentation
- List Instrumented Breakpoints 
Lists all the breakpoints that have been instrumented
- List Breakpoints Available for Instrumentation 
Lists all breakpoints that are available for instrumentation

Search Panel Options

The upper section of the panel lists the search criteria, and the lower section shows the results.

- Status: specifies the status of the object to be found from the drop-down menu. The values can be instrumented, sample trigger, sample_only, trigger_only, not-instrumented, or “*” (any). The default is “*” (any).
- Name: specifies a name, or partial name to search for in the design. Wild cards are allowed in the name. The default is “*” (any).
- Root: specifies the location in the design hierarchy to begin the recursive search. Root (/) is the default setting.
- Object: specifies the type of object to search for from the drop-down menu: breakpoint, signal, instance, or “*” (any). The default is “*” (any).
- Depth: specifies the depth of the sample buffer to be searched.
- Results increment: adds *results increment* items to the displayed list. Click the View more search results button to the right of the search button to add more results to the list.
- Sync Root: specifies ... when checked.

Search Panel Results

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

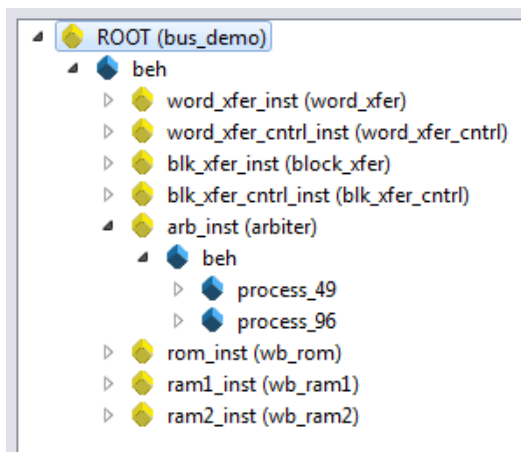
- You can change the instrumentation status of a signal by clicking the watchpoint icon and selecting the instrumentation type from the popup menu. Apply changes to multiple signals by selecting them with the Ctrl+Shift keys and apply a change to all the selected signals.
- To toggle the instrumentation 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.

Hierarchy Browser

The hierarchy browser shows a graphical representation of the design hierarchy. At the top of the browser is the ROOT node, which 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, and the next level shows the entities instantiated at the top level. For Verilog design, the level below ROOT shows the modules instantiated at the top level.

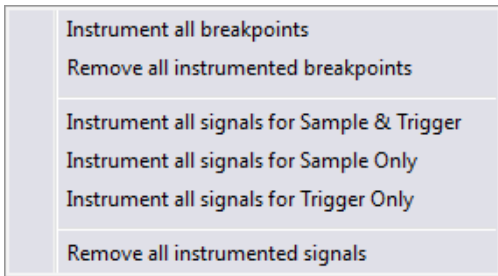
Double-clicking an entry opens the entity/module instance and displays the hierarchy below that instance. Lower hierarchical levels show instantiations, case statements, if statements, functional operators, and other statements.

Single clicking any element in the hierarchy browser causes the associated HDL code to be visible in the RTL tab display.



Certain modules cannot be instrumented, like the contents of black-box modules, which are represented by a black icon in the browser. Modules cannot be instrumented are described in [Chapter 5, *Support for Instrumenting HDL*](#).

The popup menu opens when you right-click on an element in the browser. It contains commands to set or clear breakpoints or watchpoints at any level of the hierarchy.



Selecting an operation from the popup menu applies it to all breakpoints or signal watchpoints at the selected level of hierarchy. You cannot instrument signals when a sample clock is included in the defined group.

RTL Tab





The RTL tab displays the HDL source code and is the default view when you launch the instrumentor from the ProtoCompiler shell in the native compiler flow. The code 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.


```

113  -- the state-machine
114  process ( curr_state, ack_i, delay_cnt )
115  begin
116      reg_o <= '0';
117      set_delay_cnt <= '0';
118      dec_delay_cnt <= '0';
119      case (curr_state) is
120      when st_idle =>
121          next_state <= st_run;
122      when st_run =>
123          reg_o <= '1';
124          if ( ack_i = '1' ) then
125              next_state <= st_delay1;
126          else

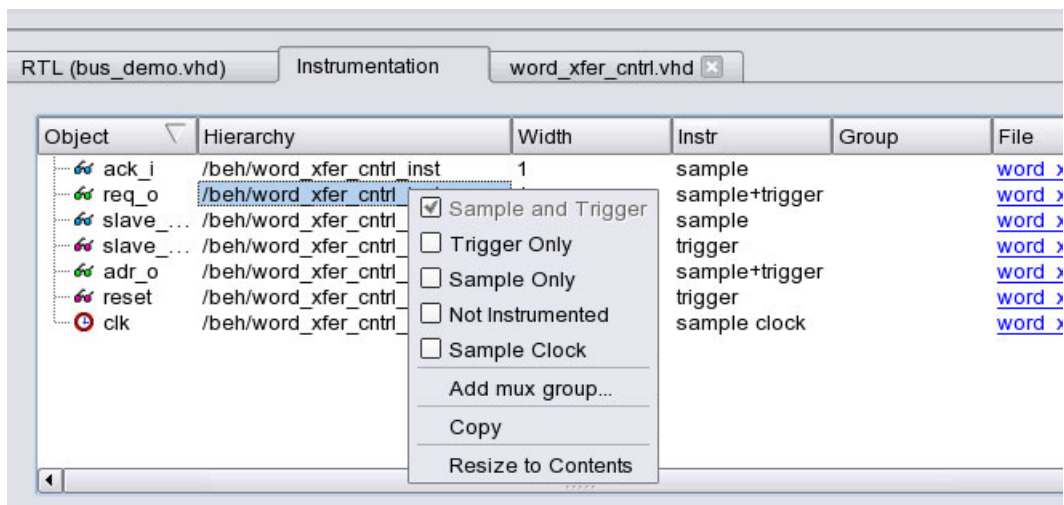
```

The panel uses symbols to represent watchpoints and breakpoints at different stages of the design:

	Clear	Watchpoint signal; not enabled
	Red	Watchpoint signal, enabled for triggering only
	Green	Watchpoint signal, enabled for both sampling and triggering
	Blue	Watchpoint signal, enabled for sampling only

Instrumentation Tab

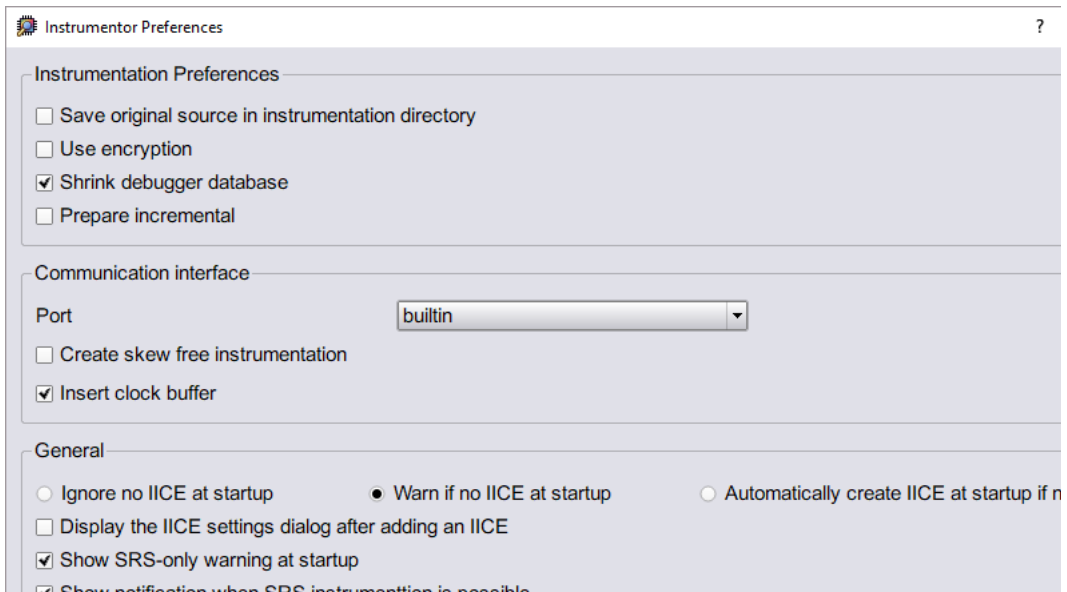
The instrumentation tab is also called the Signals view. It lists the active watchpoint and breakpoint entries that have been set within the active module or entity. The same symbols are used as to indicate watchpoints and breakpoints (see [RTL Tab, on page 128](#)). The entries can be modified by selecting the entry and assigning a new value from the popup menu.



Instrumentor Preferences

Sets parameters common to all IICE units in the design. This includes parameters like the communication port setting and the use of the optional skew-free hardware.

Select Instrumentor->Instrumentor Preferences from the top menu bar or click the Communication Interface entry in the Control Panel to open the dialog box shown below.



Instrumentation Preferences Section

The Instrumentation Preferences section includes the following check boxes:

- Save original source in instrumentation directory – Includes the original HDL source with the exported design files when checked.
- Use Encryption – Encrypts the original source.
- Shrink debugger database – when checked, the design database includes only instrumented hierarchies; when left unchecked, the full design database is loaded into the debugger.

- Prepare incremental – The Prepare incremental check box enables the ECO-based incremental flow feature. For more information about this feature, see *Running Incremental Debug for ECOs in the Prototyping User Guide (ProtoCompiler only)*.

Communication Interface Section

The Communication interface section includes the following parameters and check boxes:

- Port – specifies the type of connection to be used to communicate with the on-chip IICE. The connection types available from the drop-down menu 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 through the UMRBus.

See [Chapter 5, *Connecting to the Target System*](#) in the *Debugger User Guide* for a description of the communication interface.

- Create skew free instrumentation – incorporates skew-free master/slave hardware when checked 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-free 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 instrumentor to NOT explicitly define the JTAG clock as requiring global clock resources.

- Insert clock buffer – when using the JTAG interface, enabling the check box automatically inserts two global clock buffers to the debug IP.


General Section

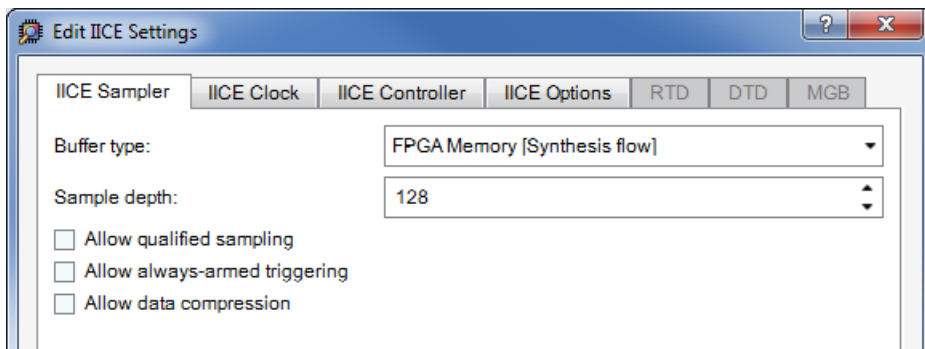
The General section includes the following parameters and check boxes:

- Startup Controls – a set of start-up radio buttons:
 - Ignore no IICE at startup – take no action when opening the instrumentor and there is no IICE defined
 - Warn if no IICE at startup – issue a warning when opening the instrumentor and there is no IICE defined
 - Automatically create IICE at startup if none – create an IICE when opening the instrumentor and there is no IICE defined
- Display the IICE settings dialog after adding an IICE – after creating an IICE (either in the Add IICE dialog box or automatically at startup), display the IICE Settings dialog box.
- Show SRS-only warning at startup – when opening the instrumentor, issue a warning if only SRS instrumentation is possible.
- Show notification when SRS instrumentation is possible –
- Automatically launch Analyst in SRS-only mode – When opening the instrumentor, automatically launch the HDL Analyst when SRS instrumentation is possible.

Edit IICE Settings

Sets individual parameters for an IICE.

You can open the Edit IICE Settings dialog box using the icon () or by clicking the IICE name in the Control Panel tab in the GUI. The Edit IICE Settings dialog box has various tabs where you can set parameters for the current IICE unit.



See the following for information on the fields in each tab:

- [IICE SamplerTab](#), on page 134
- [IICE ClockTab](#), on page 136
- [IICE Controller Tab](#), on page 137
- [IICE Options Tab](#), on page 138
- [RTD Tab](#), on page 140
- [DTD Tab](#), on page 140
- [MGB Tab](#), on page 141

IICE SamplerTab

The IICE Sampler tab is the default tab and defines buffer parameters and sampling options.

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 FPGA Memory (Synthesis flow); the Daughter Board DTD (Synthesis flow) setting configures the IICE to use the DDR3 memory of a daughter board, the Multi-FPGA DTD Module (Partition flow) setting configures the IICE to use the DDR3 memory of the multi-FPGA DTD module, and the haps_DTD_cross_trigger setting configures the IICE to use the DDR3 memory connected to each FPGA (partition flow). For more information, see [Chapter 4, 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 FPGA RAM available. 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. The sample depth increases significantly with the deep-trace debug feature.

Allow Qualified Sampling

When enabled, Allow qualified sampling builds an IICE block that samples a single time step of data for selective viewing (qualified sampling). 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

When enabled, Allow always-armed triggering saves the sample buffer for the most recent trigger until the next trigger or until it is interrupted. With this option, the tool takes a snapshot each time the trigger condition becomes true.

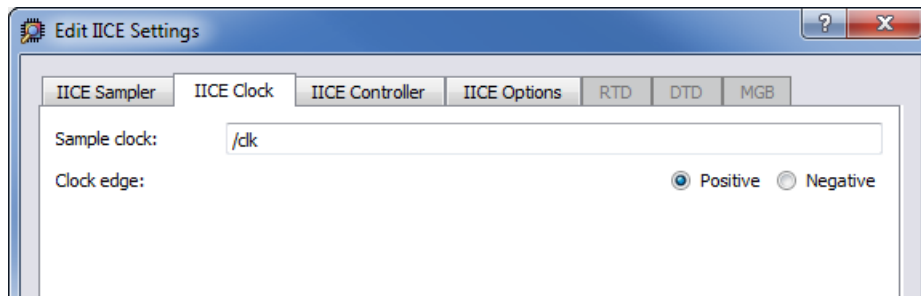
With always-armed triggering, you always acquire the data associated with the last trigger condition prior to the interrupt. This mode is helpful when analyzing a design that uses a repeated pattern as a trigger (for example, bus cycles) and then randomly freezes. You can retrieve the data corresponding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

Allow Data Compression

When enabled, Allow data compression adds logic to the IICE to compress stable sample data in the debugger. When unchecked (the default), compression logic is excluded from the IICE, and data compression in the debugger is unavailable. Note that there is a logic data overhead associated with data compression. Do not check this box when sample data compression will not be used during debug.

IICE ClockTab

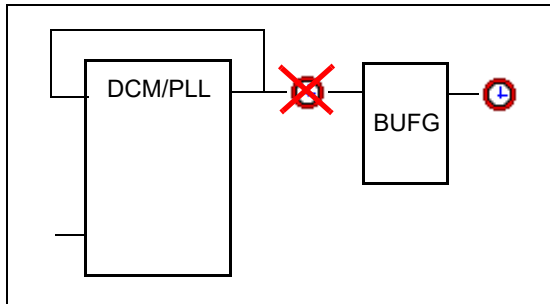
The IICE Clock tab defines the sample clock for the selected IICE unit.



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 source of the sample clock must be the output from a BUFG/IBUFG device.



You can also select the sample clock from the RTL 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

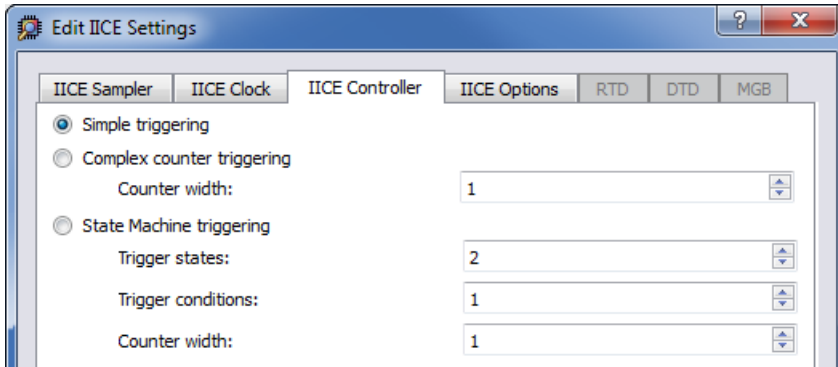
Note: The sample clock edge can only be set from the IICE Clock tab.

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 IICE.



Simple Triggering

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

Complex-Counter Triggering

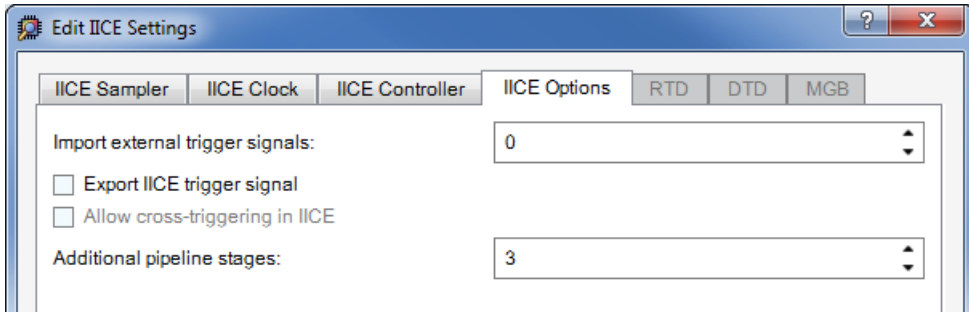
Complex-counter triggering augments the simple triggering by instrumenting a variable-width counter that can be used to create a more complex trigger function. Use the width setting to control the desired width of the counter.

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 conditions 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 90](#) in the *HAPS Prototyping Debugger User Guide*.

IICE Options Tab

The IICE Options tab configures external triggering to allow a 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 instrumentation.



Import External Trigger Signals

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.

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.

Additional Pipeline Stages

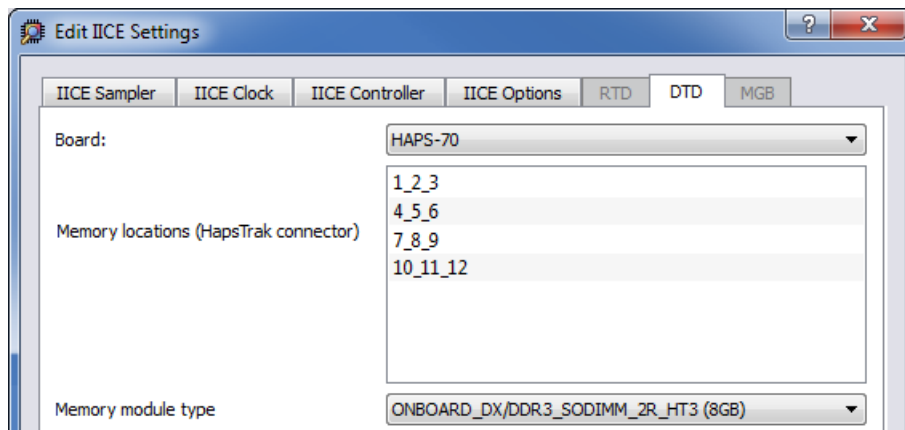
The Additional pipeline stages selection allows additional pipeline registers to be introduced for high-speed sampling of the instrumented signals that cannot meet timing. Based on a user-specified value, an equivalent number of registers are dynamically inserted for every bit to be instrumented to help relax timing on large FPGAs. The number of registers to be added is selected from the adjacent drop-down menu.

RTD Tab

The RTD tab is active when the selected IICE unit is a real-time debug (RTD) IICE. Real-time debugging is a feature that provides access to instrumented signals directly through a Mictor board interface connector installed on the HAPS board. It allows you to specify the Mictor board locations.

DTD Tab

This tab becomes available when the buffer type is set to Daughter Board DTD on the IICE Sampler tab. This tab sets parameters for deep-trace debug (DTD) with the HAPS ProtoCompiler and HAPS ProtoCompiler DX products.



The individual parameters on the tab are defined in the following table. The parameters can also be set directly from the TCL Script window using the `iice sampler` command.

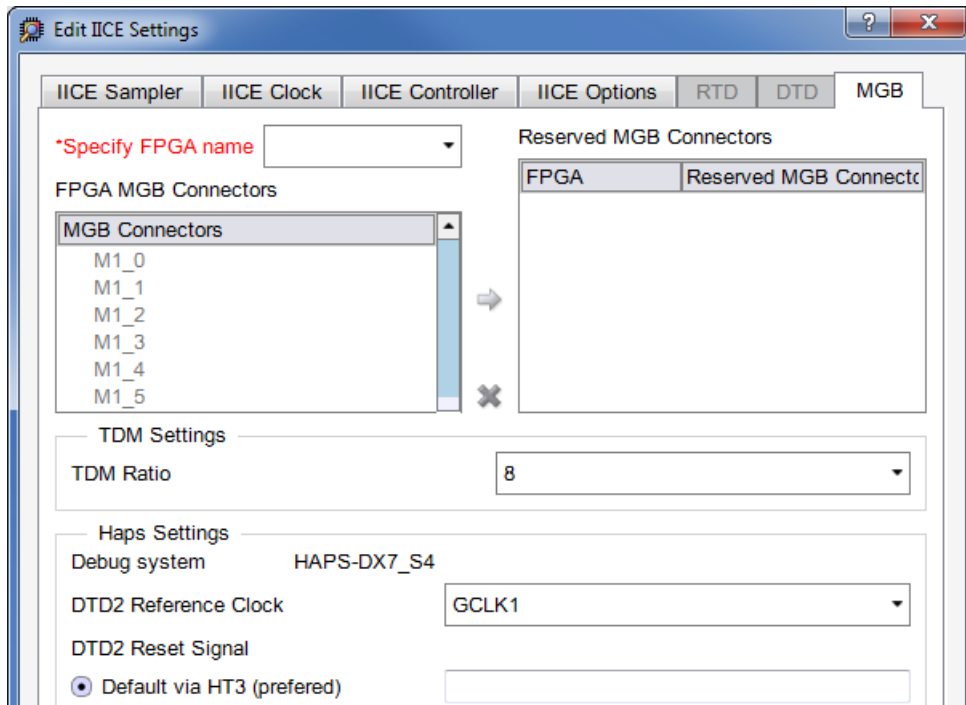
Parameter	Description
Board	Specifies the HAPS system type; the HAPS ProtoCompiler and HAPS ProtoCompiler DX products both use the HAPS-70 setting.
Memory locations	Specifies the HapsTrak connector locations where the daughter board is physically installed on a HAPS-70 system; the memory location setting is ignored when the target is HAPS ProtoCompiler DX.
Memory module type	Selects the memory module type from the drop-down menu. <ul style="list-style-type: none"> ONBOARD DX/DDR3 SODIMM 2R HT3: for 8-GByte sample buffer memory DDR3 SODIMM 2R: 4-GByte daughter board

The Daughter Board DTD selection enables the DTD tab which supports the HAPS® ProtoCompiler DX and the DDR3 daughter boards. The Multi-FPGA DTD Module selection enables the MGB tab which supports the Multi-FPGA DTD module. The haps_DTD_cross_trigger selection supports the storage of sampled signals onto DDR buffer connected to each FPGA.

Note: Sample depth can only be set from the instrumentor and cannot be changed from within the debugger.

MGB Tab

The MGB tab is enabled when the buffer type on the IICE Sampler tab is set to Multi-FPGA DTD Module. It is only available with the multi-FPGA DTD partition flow.



The individual parameters on the tab are defined in the following table. The parameters can also be set directly from the TCL Script window using the `iice sampler` command.

Parameter	Description
*Specify FPGA name	Identifies the target FPGA for the configuration.
FPGA MGB Connectors	Lists the connectors available at the specified FPGA.
Reserved MGB Connectors	Lists the connectors to be reserved. Non-reserved connectors are available for use by the debugger.
Arrow (add selected)	Moves the highlighted connectors in the User FPGA MGB Connectors area to the Reserved MGB Connectors area.
X (remove selected)	Removes the selected connector from the Reserved MGB Connectors area.

Parameter	Description
TDM Ratio	Selects the TDM ratio from the adjacent drop-down menu. Selecting AUTO enables automatic TDM ratio selection.
DTD2 Reference Clock	Selects the GCLK for the multi-FPGA DTD module from the drop-down list of GCLKs.
DTD2 Reset Signal	Specifies the source of the reset signal for the multi-FPGA DTD module. Default via HT3 (the default) auto inserts the reset signal and AUTO via GCLK uses the reset from the specified FPGA device (see Setting up DTD for Multi-FPGA HAPS-70 Designs , on page 121 for more information).

Index

Symbols

\$dumpvars
command sequence for
instrumentation flow [65](#)

A

always-armed triggering [135](#)
auto reset insertion [121](#)

B

black boxes [128](#)
breakpoint icon
color coding [32](#)
breakpoints
instance selection [32](#)
listing available [126](#)
listing instrumented [126](#)
selecting [31](#)
buffers
instrumenting restrictions [21](#)
buses
instrumenting partial [24](#)

C

clock face icon [84](#)
clocks
edge selection [137](#)
sample [83](#), [136](#)
complex counter [95](#)
size [95](#)
complex triggering [93](#), [95](#), [138](#)
Configure IICE dialog box
IICE Controller tab [93](#), [137](#), [140](#)
IICE Sampler tab [82](#), [134](#)
console window operations [45](#)
cross triggering [100](#)

D

DDR3 memory
configuring [119](#)
designs
writing instrumented [14](#)
DTD
supported memory [107](#)

E

encrypting source files [44](#)
essential signal database [28](#)

F

files
encrypting source [44](#)
idc [29](#)
folded hierarchy [22](#)

H

hardware
skew-free [81](#), [132](#)
HDL source
including [44](#)
hierarchy
folded [22](#)
hierarchy browser
popup menu [128](#)
hierarchy browser window [127](#)

I

IDC
exporting for UC RTL instrumentation
[49](#)
syntax rules, post-compile UC [50](#)
idc file
editing [29](#)

IICE

- adding [77](#)
- configuration [76](#)
- definition [15](#)
- deleting [79](#)

IICE Controller tab [93](#), [137](#), [140](#)

IICE parameters

- buffer type [135](#)
- common [79](#)
- JTAG port [132](#)

IICE Sampler tab [82](#), [134](#)

IICE settings

- sample clock [83](#), [136](#)
- sample depth [135](#)

IICE units

- cross triggering [100](#)

instrumentation

- partial records [26](#)
- post-compile [29](#)

instrumenting partial buses [24](#)

instrumentor

- running [10](#)

J**JTAG port**

- IICE parameter [132](#)

L**limitations**

- Verilog instrumentation [35](#), [38](#)
- VHDL instrumentation [33](#)

M

MGB tab [141](#)

mixed language considerations [33](#)

Multi-FPGA DTD module

- configuring [121](#)

multiplexed groups

- assigning [27](#)

O**original source**

- including [44](#)

P**P symbol**

- buses [25](#)

parameterized modules

- instrumenting [29](#)

parameters

- IICE [76](#)
- IICE common [79](#)

partial buses

- instrumenting [24](#)

passwords

- encryption/decryption [44](#)

post-compile instrumentation, UC [48](#)

Q

qualified sampling [135](#)

R

RAM resources [135](#)

records

- partially instrumented [26](#)

resets

- inserting IHAPS-70) [121](#)

restrictions

- instrumenting buffers [21](#)

RTL

- clock icon for sample clock [84](#)

S

sample clock [83](#), [136](#)

- icon [84](#)

sampling

- in folded hierarchy [22](#)
- qualified [135](#)

sampling signals [20](#), [31](#), [126](#), [129](#)

settings

- sample clock [83](#), [136](#)
- sample depth [135](#)

signals

- disabling sampling [23](#)
- exporting trigger [139](#)
- instance selection [23](#), [32](#)
- listing available [126](#)

- listing instrumented [126](#)
- sampling selection [20](#), [31](#), [126](#), [129](#)
- simple triggering [93](#), [94](#), [138](#)
- skew-free hardware [81](#), [132](#)
- source files
 - encrypting [44](#)
- SRS instrumentation, UC. *See*
 - post-compile instrumentation, UC
- state machines
 - triggering [99](#)
- state-machine triggering [93](#), [97](#), [138](#)
- SVAs
 - command sequence [57](#)
 - syntax in RTL [57](#)

T

- trigger signal
 - exporting [139](#)
- triggering
 - always-armed [135](#)
 - complex [93](#), [95](#), [138](#)
 - modes [92](#)
 - simple [93](#), [94](#), [138](#)
 - state machine [93](#), [97](#), [99](#), [138](#)
- triggers
 - complex [95](#)

U

- UC
 - exporting RTL instrumentation files [49](#)
 - post-compile instrumentation [49](#)
 - SRS instrumentation [49](#)
 - syntax rules, post-compile IDC [50](#)
- UC flow
 - post-compile instrumentation [29](#)

V

- Verdi platform [28](#)
- Verilog
 - instrumentation limitations [35](#), [38](#)
- VHDL
 - instrumentation limitations [33](#)

W

- watch icon
 - color coding [23](#), [32](#)
- watchpoint
 - definition [14](#)
- watchpoints
 - definition [14](#)
- windows
 - hierarchy browser [127](#)

