The Fastest Verification

ZeBu™ HDL Co-Simulation Manual

Document revision – b –

August 2006

Version 3.1_x

Copyright Notice
Proprietary Information

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

Right to Copy Documentation

The license agreement with EVE permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

"This document is duplicated with the permission of EVE, for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

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

Disclaimer

EVE 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.

Table of Contents

Figures

About This Manual

Overview

This manual describes how to use ZeBu for HDL co-simulation.

It describes how to perform HDL co-simulation (also referred as HDL Acceleration) with three major industry-standard simulators: ModelSim®, VCS™, VCSi™ and NC-Verilog®.

Intended Audience

This manual is written for experienced EDA hardware and software engineers to help them using ZeBu to perform HDL co-simulation for design testing and debugging using an embedded Verilog or VHDL testbench.

Engineers will typically have experience with the Verilog PLI, C language, VHDL, and an industry-standard simulator such as ModelSim, VCS, or NC-Verilog.

History

This table gives information about the content of each revision of this manual, with indication of specific applicable version:

Doc Revision	Product Version	Date	Evolution
b	3.1_0	Aug 06	New organization of Chapter 2, "HDL Co-Simulation Flow". Information for VHDL testbench with NC-Verilog (§ 3.4.3). New recommendations for `zebu.sensitivity` option (§ 4.3). Added information about delay specification (§ 4.4). Added information about selection of dynamic probes with **`zDbPostProc`** in batch mode (§ 5.2.3). Corrections in Chapter 6, "Accessing Memories".
a	3.0_0	Jun 06	First Edition for ZeBu-XL and ZeBu-UF, based on the ZeBu-XL manual (Revision a dated June 2005, for version 1.3_0). New diagram for Compilation Flow. Correction of the recommendations for clock management. Syntax correction of the command for memory list. Better description for non-regression testing.

Manual Contents

Chapter 2, "HDL Co-Simulation Flow", describes the general flow for HDL-co-simulation with ZeBu, including the specification of the test environment, the compilation of the DUT for ZeBu and the compilation of the HDL testbench.

Chapter 3, "Proceeding with Co-Simulation", describes how to perform co-simulation with ModelSim, VCS and NC-Verilog.

Chapter 4, "Advanced Clock Management", describes how clocks are modeled when using the HDL co-simulation.

Chapter 5, "Accessing Internal Signals", describes how to access to the DUT internal signals from the simulator.

Chapter 6, "Accessing Memories", describes the memory access from the simulator.

Chapter 7, "Non-Regression Testing" describes the pattern co-simulation mode for ZeBu-XL, which is especially useful for non-regression testing of designs.

Related Documentation

The following documents provide additional information related to the present manual:

- The *ZeBu Compilation Manual* describes the complete steps of the compilation process.
- The *ZeBu Reference Manual* contains descriptions about the various ZeBu-XL tools and their use, and detailed descriptions about the syntax and keywords for the DVE file, drivers and macros.
- The *ZeBu zRun Emulation Control Interface Manual* contains descriptions about how to control emulation using the **zRun** graphical interface or through Tcl scripts.

These manuals are available as dedicated manuals for ZeBu-XL or ZeBu-UF, or as common manuals for the complete ZeBu product range. The complete documentation package is listed at the back of this manual, in Chapter 8.

Typographic Conventions Used in This Manual

ZeBu tools are shown in bold, mono-space "Courier New" font, e.g. **zBuild**.

Input file contents, such as code examples, scripts or driver declarations are shown in mono-space "Courier New" font with left and bottom borders.
For example, the following line of a script describes a memory command that accepts a user-declared variable and a user-selected option:

```
memory set_rw_mode <port-name> [rw-mode]
```

Where:

`memory set_rw_mode`	is a command.
`<port-name>`	User-declared variables are inside angled brackets ("<" and ">"), such as a port name in the present example.
`[rw-mode]`	Options are presented inside square brackets. The available options will be listed and described in the text following the syntax rules.

Shell command lines are shown in mono-space "Courier New" font with left and bottom borders, including the shell prompt (which depends on your configuration); the command itself may be shown in bold characters for easier reading. :

```
$ zBuild <script_file_name>
```

Parameters and options are displayed in the same way as for input files contents: parameters inside angles brackets and options inside square brackets.

The shell prompt in this manual is $ for user environment and # for supervisor environment.

Reports, logs and any output data (except Tcl scripts), generated by ZeBu tools are shown in mono-space "Courier New" font with complete border. The following example is a log header:

```
##############################################
# Copyright (c) 2002-2006                    #
# Emulation and Verification Engineering SA  #
#--------------------------------------------#
# <Tool Name>                                #
# revision :                                 #
# date :                                     #
#--------------------------------------------#
##############################################
```

GUI elements (menu items, buttons, check boxes, edition fields) are shown in bold characters. Following is an example of GUI description:

> In the **Generate** menu, select the appropriate wrapper type to generate for the probes and then select **Generate**.

1 Introduction

1.1 ZeBu HDL Co-Simulation Features

ZeBu supports co-simulation with any HDL simulator based on the standard Verilog Programming Language Interface (PLI), which is a mechanism to interface the HDL testbench and ZeBu system. This operation mode is also known as "HDL Acceleration".

A dedicated HDL co-simulation driver is used in the ZeBu Reconfigurable Test Bench (RTB) to drive and monitor the Design-Under-Test (DUT) from a HDL testbench in ZeBu environment.

You can use a VHDL testbench without restriction, as long as your HDL simulator supports simultaneous use of VHDL and Verilog languages.

1.2 Co-Simulation Control

With HDL co-simulation, you control your verification process with an industry-standard HDL simulator. In addition, you can use **`zRun`**, ZeBu Emulation Control Interface, to access some advanced ZeBu features (e.g. multi-driver operation, memory/register operation for advanced debug).

1.3 Debugging Features

The following debugging features debugging are available when using HDL co-simulation with ZeBu:

- **Access DUT interface signals** (force/read).
- **Access internal signals** through static and dynamic probes.
- **Access memories**.
- **Generate wavefiles** in different formats: proprietary binary format in addition to `vcd` and `fsdb` formats.
- **Save and Restore:** Save the state of the design (FPGAs, memories and co-simulation buffers) at any moment of run-time; the snapshot can be restored in the emulator during the same emulation, or during a later emulation.
- **Perform non-regression tests** with the **`zPattern`** tool based on the proprietary binary format.

2 HDL Co-Simulation Flow

2.1 Description

Figure 1 shows the flow for HDL co-simulation using a Verilog testbench along with ZeBu, in comparison with Verilog-only simulation.

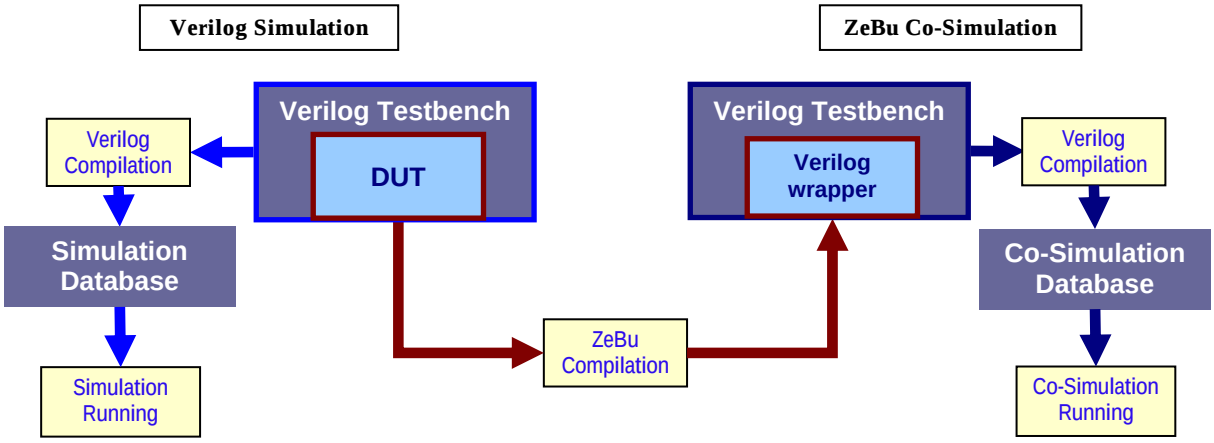


Figure 1: HDL Co-Simulation Flow for ZeBu

The Verilog testbench is compiled for the HDL simulator, along with a Verilog wrapper generated by **`zDbPostProc`** at the end of the ZeBu compilation flow (see Figure 2).

The Verilog wrapper sets up the link between your HDL simulator and ZeBu and makes the communication automatic when starting co-simulation. It has the same name as the top level module of the DUT, for example `top_module.v`, according to DVE file declaration.

Simulators differ in the way they use the Verilog PLI, in most cases because of different declaration of the ZeBu library. You should refer to your simulator User Manual for details about the Verilog PLI.

2.2 ZeBu Compilation

The DUT compilation step encompasses the generation of the firmware for the FPGAs from your synthesized DUT. The ZeBu tools generate the firmware netlist based on the DUT and the specific Design Verification Environment file (DVE). Each FPGA is then placed and routed using the Xilinx ISE P&R tools.

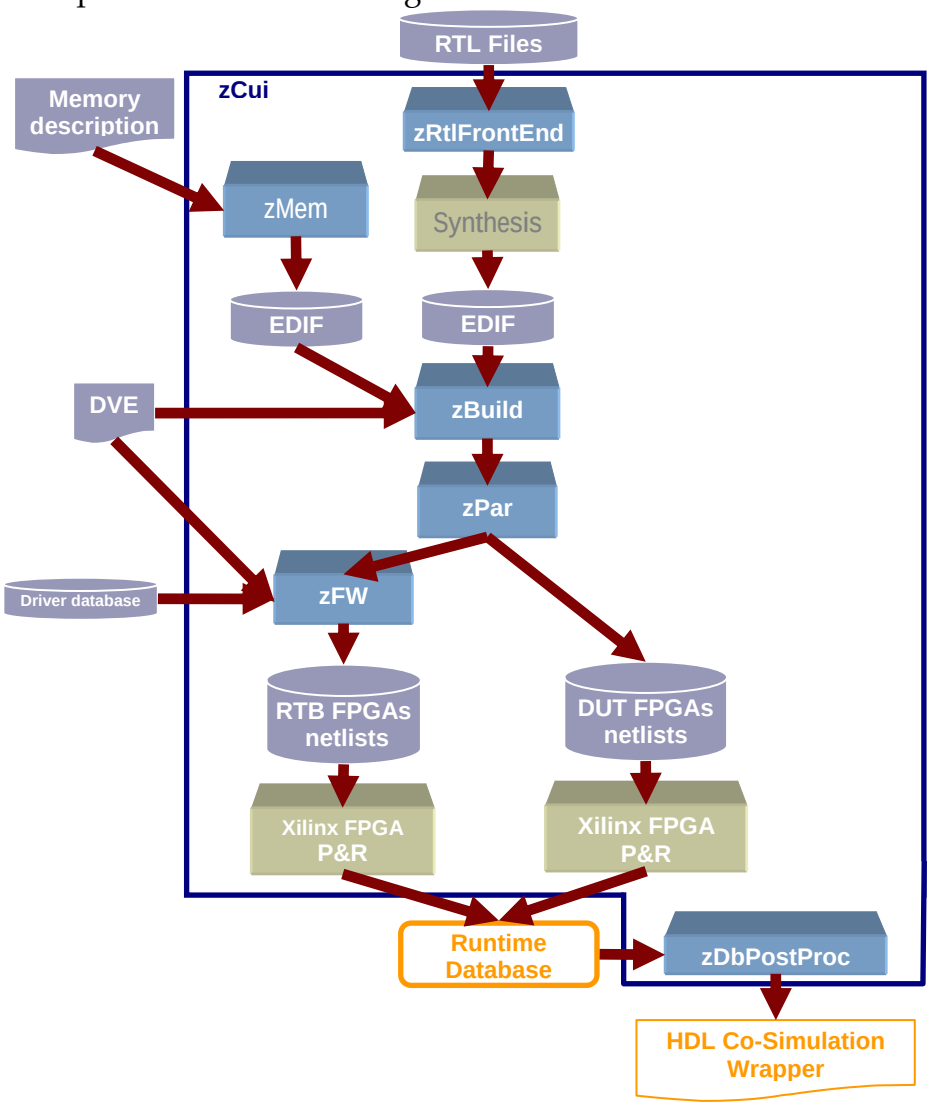


Figure 2: Compilation Flow

In HDL co-simulation with ZeBu, the configuration of the proprietary Reconfigurable Testbench (RTB) is controlled by one HDL driver declared in the Design Verification Environment (DVE) file.

When the DUT is compiled, the HDL driver produces the necessary files to configure ZeBu and to create an HDL co-simulation link to the HDL simulator.

ZeBu also allows the combination of different modes of operations, like transactors with HDL co-simulation, or Synthesizable Test Bench (STB) with HDL co-simulation.

For more details about the compilation process, refer to the *ZeBu Compilation Manual*.

2.3 Files for Testbench Compilation

Two types of files are used while compiling the testbench for HDL co-simulation: some general purpose files which are included in the ZeBu software release, and driver dependent files which are generated by the ZeBu compilation process.

2.3.1 ZeBu Release Files for Testbench Compilation

The ZeBu release includes some files for the HDL testbench compilation:

- `libZebuPLI.so`: This is a **mandatory** library file that contains the required functions to establish the communication between the simulator and ZeBu.
- `pli.tab`: This file is only used by VCS and VCSi. It contains the cross-references between Verilog symbols and C functions of the `libZebuPLI.so` library.

These files are located in the `$ZEBU_ROOT/lib` directory.

2.3.2 The Verilog Wrapper

The Verilog wrapper contains the necessary declarations to establish the communication link between the simulator and ZeBu. It is generated by the **`zDbPostProc`** at the end of the ZeBu compilation process, according to the design interface.

This file replaces the original DUT files in the HDL compilation process. It must be used without modification of its content.

The original DUT interface is reproduced in the Verilog wrapper so that it can be connected easily to the testbench.

An implicit instantiation of the DUT in the testbench may result in problems at compilation or in bad run-time results (due to the difference in the connector position between the original Verilog DUT and the Verilog wrapper used for the co-simulation).

Example:

```
module dut_top_module (
    d,
    clk,
    bus_in,
    bus_in2,
    d2,
    io,
    io2,
    q,
    q2,
    bus_out,
    bus_out2);
```

```
    input         d;
    wire          d;
    input         clk;
    wire          clk;
    input  [7:0]  bus_in;
    input  [7:0]  bus_in2;

    inout         io;
    trireg        io;
    inout         io2;
    trireg        io2;

    output        q;
    reg           q;
    output        q2;
    reg           q2;
    output [7:0]  bus_out;
    reg    [7:0]  bus_out;
    output [7:0]  bus_out2;
     reg    [7:0]  bus_out2);

/* Dynamic probes control
This part is inserted in case you use dynamic probes.
If not, it will remain inactive */

    reg zebu_readback;

    initial zebu_readback = 0;

    always @(zebu_readback)
        if (zebu_readback)
            $ZEBU_readback(1);
        else
            $ZEBU_readback(0);

/* ZeBu Link Initialization:
The ZEBU_task is the main task used to initialize
the link with the ZeBu system and to download the design into ZeBu */

    initial
        // DBG_MODULES
        $ZEBU_task;

  // Read Back registers and
  // DUT architecture may take place
  // here if selected

endmodule
```

2.4 Using a VHDL Testbench

If the testbench is written in VHDL, it is still possible to use the same Verilog wrapper generated by **`zDbPostProc`** in place of the original DUT file. Figure 3 shows the difference between pure VHDL simulation and co-simulation with ZeBu, and shows how the HDL simulator connects to the ZeBu Verilog Black Box used for the DUT.

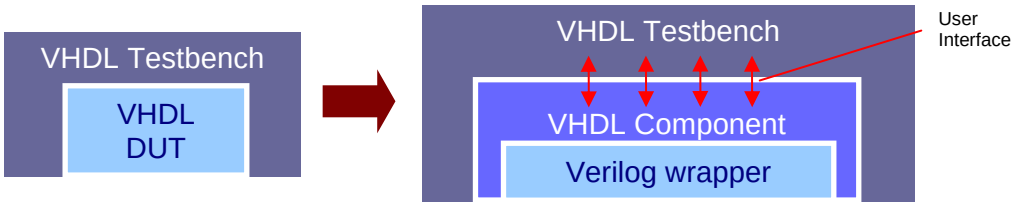


Figure 3: VHDL Testbench

When compiling the VHDL testbench, the Verilog wrapper has to be enclosed as a component in a VHDL wrapper. According to the HDL simulator you use, the VHDL component for co-simulation can be either generated by a specific tool (with ModelSim) or written manually in the VHDL testbench file (with NC-Verilog).

Note: When using a VHDL testbench with ZeBu, your HDL simulator must support the multi-language feature, requiring both VHDL and Verilog licenses.

2.5 DVE File for HDL Co-Simulation

2.5.1 Description

The DVE file describes your design verification environment, in particular the connections between the DUT and the ZeBu drivers and the clock connections. It is recommended to generate the DVE file using **`zNetgen`** (or the graphical interface **`zCui`**).

Further information to write the DVE file manually is given later in this manual and in the *ZeBu Compilation Manual*; the complete syntax for the DVE file is in the *ZeBu Reference Manual*.

The driver for driving and monitoring a design using an HDL testbench is `HDL_COSIM`. All the DUT inputs and outputs that have to be observed or forced in the HDL testbench must be connected to the `HDL_COSIM` driver interface.

2.5.2 Checking the content of the DVE file

It is recommended to use **`zNetgen`** to generate the DVE file for HDL co-simulation. However, you should check that the interface actually corresponds to your design and that the design clocks are declared correctly.

2.5.2.1 Port Types for HDL_COSIM Driver

The port types in the DVE file must be consistent with the actual interface of your design. For example, if an input port of the design is declared using an `inout_tri` port in the DVE, the ZeBu compilation adds a tri-state buffer interfering with the behavior of the HDL testbench.

The `HDL_COSIM` driver supports five port types:

`input_bin`	This binary port connects to the DUT inputs.
`input_tri`	This tri-state port connects to the DUT inputs, including high impedance (highZ) inputs. The DUT input is connected to a tri-state buffer that goes to high impedance as specified in the test bench.
`output_bin`	This binary port connects to the two-state outputs of the DUT
`output_tri`	This tri-state port connects to the tri-state outputs of the DUT, it returns a high impedance (highZ) value if the DUT output is at high impedance. There is a tri-state buffer on the output of the DUT.
`inout_tri`	This is a tri-state input and output port, tri-state conditions are handled in either direction.

2.5.2.2 DUT clocks

DUT clocks are specific signals, typically connected to the ZeBu clock generator and declared as zceiClockPort in the DVE file.

You can have up to eight clock ports, numbered from 0 to 7 and that should be specified as "`clock_0`" to "`clock_7`", **in ascending order**.

If your design requires more than 8 clocks, additional clocks can be declared as `input_bin` ports and it will be necessary to manage more carefully the clock mapping to avoid clock skews in the design FPGA implementation.

Note that it is mandatory to have at least one clock connected to the ZeBu clock generator in your design.

Detailed recommendations for clock modeling are available in a dedicated chapter of the *ZeBu Compilation Manual*.

2.5.3 Example of DVE File

Figure 4 represents the DUT interface. The DVE file content for the associated HDL co-simulation driver is given below the figure.

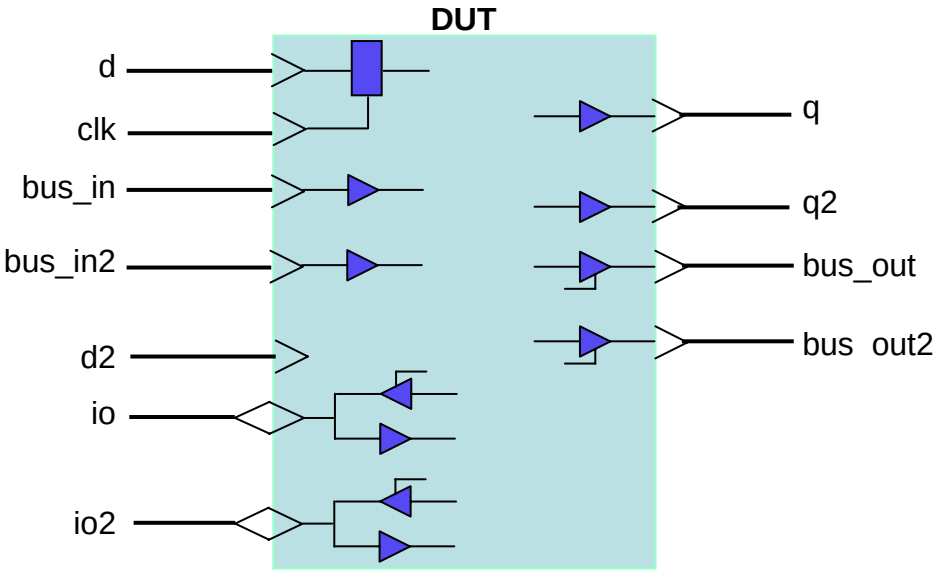


Figure 4: Typical HDL Co-simulation Driver

```
HDL_COSIM my_cosim (
   .input_bin ({ d, d2}),
   .input_tri ({ bus_in, bus_in2}),
   .output_bin ({ q, q2}),
   .output_tri ({ bus_out, bus_out2}),
   .inout_tri ({ io, io2})
);

defparam  my_cosim.clock_0 = clk;

zceiClockPort  clkport (.cclock(clk));
```

3 Proceeding with Co-Simulation

This chapter describes how to compile the HDL testbench and how to run the HDL co-simulation with ZeBu, using ModelSim, VCS/VCSi, and NC-Verilog simulators.

For detailed information about the features of a simulator, you should refer to its User Manual.

3.1 Run-Time Features for HDL Co-Simulation with ZeBu

3.1.1 ZeBu Run-time Options

When running the simulator, it is possible to pass run-time options to ZeBu. These options must be added to the simulator command line.

`+zebu.work=<path>`	Directory where the co-simulation kernel can find the ZeBu information. By default, the current path is used, but this option can point to another place. Path is either absolute or relative from the place where the simulator is launched.
`+zebu.verbose`	Adds debug messages when running simulator. It can help you locate any problems that might occur. Using this option may reduce the simulation speed: use it only when debugging.
`+zebu.dumpfile=<filename>`	Use this to dump waveforms file (binary, VCD or FSDB format) during the emulation. This file contains all the information sent and received by ZeBu. The proprietary binary format (`.bin` file) is used for non-regression testing with **`zPattern`**.
`+processname=<processname>`	Defines the name of the process associated to the driver in the DVE file.
`+zebu.design=<designFeatures path>`	Defines the path to the `designFeatures` file.
`+zebu.delayfile=<filename>`	Defines the path to the file listing the delays to be applied to output ports, as described in Section 4.4.
`+zebu.sensitivity=[option]`	Defines the operating mode of the simulation, determining the sensitivity of data exchange. Refer to Section 4.3 for more details.

3.1.2 Using the Save and Restore Feature

Save and Restore is a feature which allows you to capture the state of the design at run-time and to restore this state of the system during the same emulation, or during a subsequent emulation.

The save and restore feature is supported by ModelSim and NC-Verilog. When invoked from the HDL simulator, the save and restore feature transparently processes the state for the testbench and for the design in ZeBu.

You can also read the content of a saved state independently of the simulator and without ZeBu hardware using the C/C++ API of the `ZebuRestore` library, as described in the *ZeBu C++ Co-Simulation Manual*.

3.1.3 Generating Pattern Files for Non-Regression Testing

You can perform non-regression testing by generating a pattern file during a run and using this pattern file as a reference for comparison with a simulation performed at a later date on a modified design.

You must include the `+zebu.dumpfile` option in the command line when executing your initial simulation. The pattern file must be in the ZeBu proprietary binary format, with a `".bin"` extension.

The appropriate syntax to launch ZeBu co-simulation with generation of a pattern file depends on your HDL simulator. It is described for each of the three simulators in separate sections.

Refer to Chapter 7 for a description of how to perform non-regression testing with pattern co-simulation, using **`zPattern`**.

3.2 Co-Simulation with ModelSim

The compilation process and the co-simulation execution are controlled by two different commands: `vlog` and `vsim`.

When using ModelSim for the first time, create a library (using the `vlib` command) in your run directory.

For example, if your run directory is called `work`, the command is:

```
$> vlib work
```

3.2.1 Compilation for Co-Simulation

The compilation for co-simulation with ZeBu is similar to that for pure simulation. All modules that make up the testbench must be compiled together with the wrapper module generated by **`zDbPostProc`**, which replaces all DUT modules.

The following example compares the compilation command without ZeBu versus that for co-simulation with ZeBu.

Syntax for simulation:

```
$> vlog [options] <testbench_files.v> <dut_files.v>
```

Syntax for ZeBu co-simulation:

```
$> vlog [options] <testbench_files.v> zebu.work/<top_module>.v
```

If a script is available for the compilation of the testbench, you can reuse it with minor changes: replacing the DUT files with the wrapper generated by **zDbPostProc**.

3.2.2 Starting the Co-Simulation

The co-simulation command is slightly different from the pure simulation command.

ModelSim must load the ZeBu library dynamically: use the `-pli` option of ModelSim and set the dynamic library path environment variable correctly; i.e. pointing to the ZeBu release area. The connection of ZeBu to ModelSim is performed automatically at the start of the execution.

To verify that the dynamic library path environment variable (`LD_LIBRARY_PATH`) is set properly, just echo it:

```
$> echo $LD_LIBRARY_PATH
/usr/local/zebu/<release>/lib:<otherPath>
```

If it doesn't contain the ZeBu release area path, add it:

```
$> export LD_LIBRARY_PATH=$ZEBU_ROOT/lib:$LD_LIBRARY_PATH
```

The following compares the pure simulation command with the ZeBu co-simulation command, which includes the PLI library.

Syntax for simulation:

```
$> vsim [options] <testbench_topmodule>
```

Syntax for ZeBu co-simulation:

```
$> vsim [options] <testbench_topmodule> -pli libZebuPLI.so
```

3.2.3 Co-Simulation Transcript

The co-simulation transcript contains the ZeBu calls as displayed below:

```
vsim -c top -pli libZebuPLI.so -do "run -all; quit;" +zebu.work=../zebu.work +zebu.verbose
Reading /auto/import/modeltech/modelsim_6.0b/tcl/vsim/pref.tcl

# 6.0b
# vsim +zebu.work=../zebu.work +zebu.verbose -do {run -all; quit;} -c -pli libZebuPLI.so top
```

```
      ################################################
      #           Copyright (c) 2002-2006          #
      # Emulation and Verification Engineering  SA #
      #--------------------------------------------#
      # vsimk                                      #
      # revision :                                 #
      # date :                                     #
      #--------------------------------------------#
      ################################################

# /usr/import/modeltech/modelsim_6.0b/bin/../linux/vsim -c top -pli libZebuPLI.so -do "run -
all; quit;" +zebu.work=../zebu.work +zebu.verbose
# Loading /home/common/zebu/lib/libZebuPLI.so
# //  ModelSim SE VLOG 6.Ob Dec 1 2004 Linux 2.4.21-20.EL
# //
# //  Copyright Model Technology, a Mentor Graphics Corporation company, 2003
# //                         All Rights Reserved.
# //                  UNPUBLISHED, LICENSED SOFTWARE.
# //         CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
# //        PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS.
# //
# Loading work.top
# Loading ./mti_uselibs/zebu.work@counter8.counter8
# Loading ./mti_uselibs/counter8.counter8
# step : end of compile
# step : initialize
# info : zebu.work path is ../zebu.work

-- ZeBu : vsim : "default_process" is a full-capability process.
-- ZeBu : vsim : Looking for a connection ...

      ################################################
      #           Copyright (c) 2002-2006          #
      # Emulation and Verification Engineering  SA #
      #--------------------------------------------#
      # zServer                                    #
      # revision :                                 #
      # date :                                     #
      #--------------------------------------------#
      ################################################

# zServer -design designFeatures -zebu.work zebu.work

Checking for license ... WAITING
Checking for license ... OK
-- ZeBu : zServer : Evaluating the file designFeatures" to get the design's features.

-- ZeBu : zServer : Evaluating the file designFeatures" to get the controlled clock
"B0.clock"'s features.
-- ZeBu : zServer : Looking for a connection ...

-- ZeBu : zServer : A 1 board serial link chain is required (board : ZeBu-XL-2).
-- ZeBu : zServer : The ZeBu-XL-2 board "zebu_0130" is free.
-- ZeBu : zServer : Get the connection at Tue 26 4 2005 - 19:50:39.
-- ZeBu : zServer : Connection established.

-- ZeBu : zServer : The ZeBu-XL-2 board on PCI slot 02:0b.0 will use the interrupt line
number 11.

-- ZeBu : zServer : Initialization of the ZeBu-XL-2 board "zebu_0130" (ZeBu-XL family) -
(maui.eve z_0).

-- ZeBu : zServer : Initialize system frequency to 50 MHz.
-- ZeBu : zServer : System frequency Initialisation succeeded.

-- ZeBu : zServer : ZCPB board is detected.
-- ZeBu : zServer : Module 0 is detected on ZCPB board.
-- ZeBu : zServer : Module 1 is detected on ZCPB board.
-- ZeBu : zServer : Module 2 is detected on ZCPB board.
-- ZeBu : zServer : Module 3 is detected on ZCPB board.
```

```
-- ZeBu : zServer : Control Fpgas reset phase in progress ...
-- ZeBu : zServer : Control Fpgas reset phase Succeeded
-- ZeBu : zServer : Design Fpgas reset phase in progess ...
-- ZeBu : zServer : Design Fpgas reset phase Succeeded

-- ZeBu : zServer : XDR initialization.

-- ZeBu : zServer : Design path : /auto/mars/QA/XL/V1.3_0/05-04-22-
maui/counter8/HDL/zebu.work
-- ZeBu : zServer : Design loading in progress .....
-- ZeBu : zServer : Hubs loaded           : -4-
-- ZeBu : zServer : Module 0 Fpgas loaded : -1-

-- ZeBu : zServer : Initialization succeeded.

-- ZeBu : zServer : Waiting for client(s).

-- ZeBu : vsim : Connection established.

-- ZeBu : zServer : Getting the clock "B0.clock"'s features.
-- ZeBu : zServer : The maximum frequency of the clock "driverClk" is setup to 3 MHz.
-- ZeBu : zServer : Initialize "counter8_hdlcosim::txp".

-- ZeBu : zServer : Initialize "counter8_hdlcosim::rxp".

-- ZeBu : vsim : XDR activated.
-- ZeBu : zServer : ZeBu-XL-2 board 0, the system frequency is 50 MHz.

-- ZeBu : zServer : For your chosen frequencies (hub 41 MHz = Data Rate 82000 Mb/s), the
optimum frequency of the clock "driverClk" is 7.820 MHz.
-- ZeBu : zServer : WARNING : Due to cosim driver setup, the maximum frequency of the clock
"driverClk" is 3 MHz.
-- ZeBu : zServer : The frequency of the clock "driverClk" (3 MHz) will be adjusted to the
closer inferior available frequency (2.941 MHz).
-- ZeBu : vsim : INFO   : Zebu board 0 is initialized.
# info : Zebu board 0 opened
# info : HDL driver counter connected
# info : Zebu board 0 initialized
# step : Checking
# step : Checking module : counter8_hdlcosim
# step : Checking inputs
# step : Checking outputs
# step : Checking internal state capture
# step : Checking clock specifications about module : counter8_hdlcosim
# step : Check done
# run -all; quit;

-- ZeBu : vsim : INFO   : Elapsed time for initialization : 7
-- ZeBu : vsim : INFO   : Elapsed time for emulation      : 19
-- ZeBu : vsim : INFO   : Total elapsed time              : 26

-- ZeBu : vsim : Waiting for zServer (17679) to stop...

-- ZeBu : zServer : End of run :
-- ZeBu : zServer : driverClk cycle counter : 55,971,037.
-- ZeBu : zServer : B0.clock::clock cycle counter : 1,000,003.

-- ZeBu : zServer : XDR disabled.

-- ZeBu : zServer : Server closed correctly.
-- ZeBu : vsim : CLOSING ZEBU
# step : finish
# info : Zebu board 0 closed
```

3.2.4 Error Messages

If the PLI library has not been included, ModelSim issues error messages referring to undefined tasks, as displayed below:

```
# Loading work.testbench
# Loading work.dut_top_module
# ** Warning: (vsim-PLI-3003) zebu.work/topmodule.v(34): [TOFD] - System task or function
'$ZEBU_readback' is not defined.

#         Region: /testbench/dut_instance
# ** Warning: (vsim-PLI-3003) zebu.work/topmoule.v(36): [TOFD] - System task or function
'$ZEBU_readback' is not defined.
#         Region: /testbench/dut_instance
# ** Warning: (vsim-PLI-3003) zebu.work/topmodule.v(42): [TOFD] - System task or function
'$ZEBU_task' is not defined.
#         Region: /testbench/dut_instance
```

3.2.5 Using a VHDL Testbench

First operation is to create a working library and compile the Verilog wrapper generated by **`zDbPostProc`**:

```
$> vlib <work_library>
$> vlog [options] zebu.work/<top_module>.v
```

To create the VHDL component which integrates the wrapper compiled with `vlog`, use the `vgencomp` command of ModelSim:

```
$> vgencomp <top_module>
```

The resulting VHDL component is integrated in the VHDL testbench and the testbench is compiled:

```
$> vcomp [options] <testbench_files.vhdl>
```

To launch the co-simulation:

```
$> vsim [options]  <testbench topmodule> -pli libZebuPLI.so
```

Note: When using a VHDL testbench with ZeBu, your HDL simulator must support the multi-language feature, requiring both VHDL and Verilog licenses.

3.2.6 Integrating ZeBu Functions in ModelSim

You can develop a custom integration of ZeBu within ModelSim by using advanced ModelSim features. By using the Tcl/Tk ModelSim interface, you can add menus and buttons that directly command some ZeBu functions.

Example:

The following script, given as a "`.do`" file to ModelSim 6.1, adds the Internal Capture ON and OFF menu (Enable/Disable Internal Capture) in the ModelSim interface:

```
global ZEBU_WORK
global RECOMPILE

# Change the path to the zebu.work directory here :
set ZEBU_WORK "../zebu.work"
# Change the command for recompiling the verilog here :
set RECOMPILE "make build"

set zebuInternalCapture 0
set vsimVersion [vsimId]; # Gets vsim version
if { $vsimVersion >= "6.1" } {
   set wname [view wave –undock wave]; # Gets the path to the Wave window
   zebu_modelsim_init $wname
   view –dock wave

} else {
   set wname [view wave]; # Gets the path to the Wave window
   zebu_modelsim_init $wname
}

proc zebu_modelsim_init {wname} {
  add_menu $wname zebu 3
  add_menuitem $wname zebu "Enable Internal Capture" zebu_readback_on
  add_menuitem $wname zebu "Disable Internal Capture" zebu_readback_off
  add_menuitem $wname zebu "Internal Capture Browser" zebu_readback_add
}

proc zebu_readback_on {} {
  puts "ZeBu internal state capture is ON"
  environment /
  set zebuControlPath [find nets -recursive zebu_readback]
  force -freeze sim:$zebuControlPath 1 0
}

proc zebu_readback_off {} {
  puts "ZeBu internal state capture is OFF"
  environment /
  set zebuControlPath [find nets -recursive zebu_readback]
  force -freeze sim:$zebuControlPath 0 0
}
```

This script is part of the ZeBu release, in `$ZEBU_ROOT/etc/tcl/zebu_modelsim.tcl`

The script creates the menu in the ModelSim waveform window as displayed below:

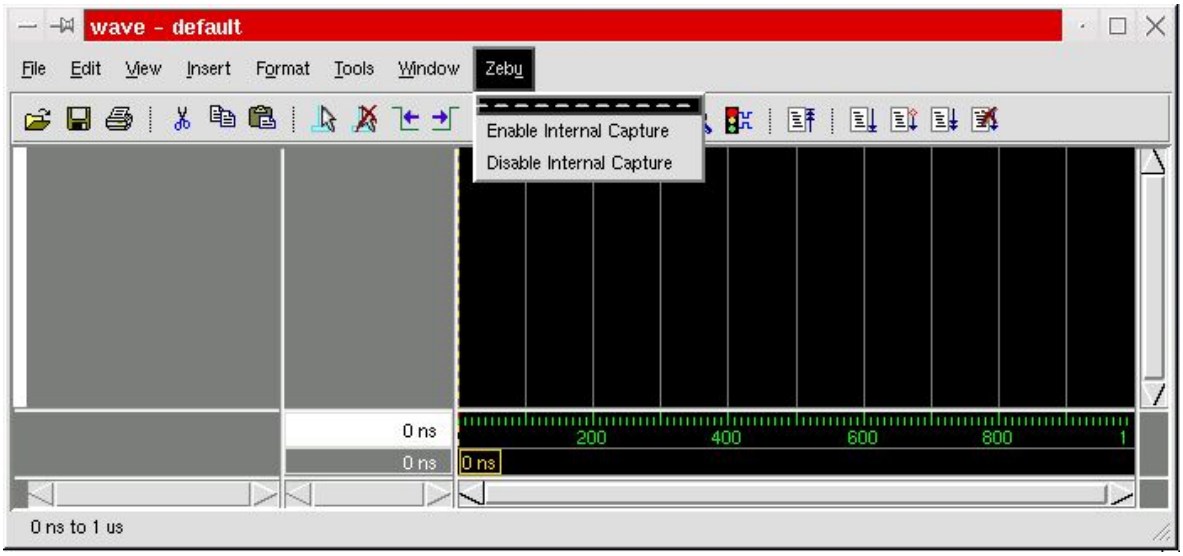


Figure 5: Customized ModelSim Waveform Window

3.2.7 Save and Restore with ModelSim

This section describes how to use the Save and Restore feature with ModelSim:

- For the "Save" operation, use the following syntax:

```
checkpoint <save_name>
```

- The "Restore" operation is possible in two different ways:
  - Warm Restore: restore during the same session.

```
restore <save_name>
```

  - Cold Restore: restore after the session.

```
vsim -restore <save_name>
```

Refer to the ModelSim documentation for more details.

3.2.8 Generating Pattern Files for Non-Regression Testing

For ModelSim, the appropriate syntax to launch ZeBu co-simulation with generation of a pattern file is the following one:

```
$> vsim [options>] TB_TopModule –pli libZebuPLI.so
+zebu.dumpfile=pattern.bin
```

3.3 Co-Simulation with VCS/VCSi

In VCS and VCSi, the compilation process and the simulation execution are controlled by the same command: `vcs` or `vcsi`.

3.3.1 Compilation for Co-Simulation

Compilation with ZeBu is very similar to the compilation without ZeBu. The only difference is that the ZeBu files replace the DUT files.

The following compares the compilation for simulation without ZeBu and compilation for co-simulation with ZeBu.

Syntax for simulation:

```
$> vcs [options] <testbench_files.v> <dut_files.v>
```

Syntax for ZeBu co-simulation:

```
$> vcs [options] <testbench_files.v> zebu.work/<top_module>.v
       -P $ZEBU_ROOT/lib/pli.tab $ZEBU_ROOT/lib/libZebuPLI.so
```

If a script written for simulation is to be used for co-simulation, it must include the above modifications.

Note that a space is required between the '`-P`' and the associated value. Note also that VCS uses the `libZebuPLI.so` library in a static way, thus the `LD_LIBRARY_PATH` does not need to include the ZeBu path.

Specific Recommendations for VCS Version 7.1 and 7.2

When using VCS 7.1 or 7.2 with a testbench that connects directly a port of the interface to '1' or '0', the simulator does not initialize this port correctly (the default value of the port is '0' in our PLI). The appropriate work-around consists in using the `+oldpli` option in the VCS compilation command. An example of instantiation follows:

```
top_module my_top_module(
    .d(1'b1),              // stucked at 1
    .clk(clk),
    .bus_in(bus_in),
    .bus_in2(bus_in2),
    .d2(1'b0),             // stucked at 0
    .io(io),
    .io2(io2),
    .q(q),
    .q2(q2),
    .bus_out(bus_out),
    .bus_out2(bus_out2));
```

In this case the syntax for ZeBu co-simulation must be the following:

```
$> vcs [options] <testbench_files.v> zebu.work/<top_module>.v
       -P $ZEBU_ROOT/lib/pli.tab $ZEBU_ROOT/lib/libZebuPLI.so +oldpli
```

3.3.2 Executing Co-Simulation

The co-simulation execution command is the same as for simulation:

```
$>./simv +zebu.work=../zebu.work +zebu.verbose +vcs+lic+wait
```

3.3.3 Co-Simulation Transcript

The log file will contain the ZeBu call for initialization, as displayed below (ZeBu-XL example):

```
      ################################################
      #           Copyright (c) 2002-2006          #
      # Emulation and Verification Engineering  SA #
      #--------------------------------------------#
      # simv                                       #
      # revision :                                 #
      # date :                                     #
      #--------------------------------------------#
      ################################################

# ./simv +zebu.work=../zebu.work +zebu.verbose +vcs+lic+wait

-- ZeBu : simv : "default_process" is a full-capability process.
-- ZeBu : simv : Looking for a connection ...

      ################################################
      #           Copyright (c) 2002-2006          #
      # Emulation and Verification Engineering  SA #
      #--------------------------------------------#
      # zServer                                    #
      # revision :                                 #
      # date :                                     #
      #--------------------------------------------#
      ################################################

# zServer -design designFeatures -zebu.work zebu.work

Checking for license ... WAITING
Checking for license ... OK
-- ZeBu : zServer : Evaluating the file designFeatures" to get the design's features.

-- ZeBu : zServer : Evaluating the file designFeatures" to get the controlled clock
"B0.clock"'s features.
-- ZeBu : zServer : Looking for a connection ...

-- ZeBu : zServer : A 1 board serial link chain is required.
-- ZeBu : zServer : The ZeBu-XL-2 board "zebu_0130" is free.
-- ZeBu : zServer : Get the connection.
-- ZeBu : zServer : Connection established.

-- ZeBu : zServer : The ZeBu-XL-2 board on PCI slot 02:0b.0 will use the interrupt line
number 11.

-- ZeBu : zServer : Initialization of the ZeBu-XL-2 board "zebu_0130" (ZeBu-XL family) -
(maui.eve z_0).

-- ZeBu : zServer : Initialize system frequency to 50 MHz.
-- ZeBu : zServer : System frequency Initialisation succeeded.

-- ZeBu : zServer : ZCPB board is detected.
-- ZeBu : zServer : Module 0 is detected on ZCPB board.
-- ZeBu : zServer : Module 1 is detected on ZCPB board.
-- ZeBu : zServer : Module 2 is detected on ZCPB board.
-- ZeBu : zServer : Module 3 is detected on ZCPB board.
                                                                                ...
```

```
-- ZeBu : zServer : Control Fpgas reset phase in progress ...
-- ZeBu : zServer : Control Fpgas reset phase Succeeded
-- ZeBu : zServer : Design Fpgas reset phase in progess ...
-- ZeBu : zServer : Design Fpgas reset phase Succeeded

-- ZeBu : zServer : XDR initialization.

-- ZeBu : zServer : Design path : zebu.work
-- ZeBu : zServer : Design loading in progress .....
-- ZeBu : zServer : Hubs loaded           : -4-
-- ZeBu : zServer : Module 0 Fpgas loaded : -1-

-- ZeBu : zServer : Initialization succeeded.

-- ZeBu : zServer : Waiting for client(s).

-- ZeBu : simv : Connection established.

-- ZeBu : zServer : Getting the clock "B0.clock"'s features.
-- ZeBu : zServer : The maximum frequency of the clock "driverClk" is setup to 3 MHz.
-- ZeBu : zServer : Initialize "counter8_hdlcosim::txp".

-- ZeBu : zServer : Initialize "counter8_hdlcosim::rxp".

-- ZeBu : simv : XDR activated.
-- ZeBu : zServer : ZeBu-XL-2 board 0, the system frequency is 50 MHz.

-- ZeBu : zServer : For your chosen frequencies (hub 41 MHz = Data Rate 82000 Mb/s), the
optimum frequency of the clock "driverClk" is 7.820 MHz.
-- ZeBu : zServer : WARNING : Due to cosim driver setup, the maximum frequency of the clock
"driverClk" is 3 MHz.
-- ZeBu : zServer : The frequency of the clock "driverClk" (3 MHz) will be adjusted to the
closer inferior available frequency (2.941 MHz).
-- ZeBu : simv : INFO   : Zebu board 0 is initialized.

step : end of compile
step : initialize
info : zebu.work path is ../zebu.work
info : Zebu board 0 opened
info : HDL driver counter connected
info : Zebu board 0 initialized
step : Checking
step : Checking module : counter8_hdlcosim
step : Checking inputs
step : Checking outputs
step : Checking internal state capture
step : Checking clock specifications about module : counter8_hdlcosim
step : Check done
$finish at simulation time             20000070
-- ZeBu : simv : INFO   : Elapsed time for initialization : 6
-- ZeBu : simv : INFO   : Elapsed time for emulation      : 48
-- ZeBu : simv : INFO   : Total elapsed time              : 54

-- ZeBu : simv : Waiting for zServer (17740) to stop...

-- ZeBu : zServer : End of run :
-- ZeBu : zServer : driverClk cycle counter : 139,912,509.
-- ZeBu : zServer : B0.clock::clock cycle counter : 1,000,003.

-- ZeBu : zServer : XDR disabled.

-- ZeBu : zServer : Server closed correctly.

-- ZeBu : simv : CLOSING ZEBU

step : finish
info : Zebu board 0 closed
```

3.3.4 Error Messages

The following log displays the error you get if the PLI and the library are missing:

```
Parsing design file 'testbench.v'
Parsing design file 'zebu.work/topmodule.v'
Error-[UST] Undefined system task
        Undefined System Task call to '$ZEBU_readback'
        "zebu.work/topmodule.v", 34: token is ';'
            $ZEBU_readback(1);
                           ^
Error-[UST] Undefined system task
        Undefined System Task call to '$ZEBU_task'
        "zebu.work/topmodule.v", 42: token is ';'
        $ZEBU_task;
                   ^
2 errors
```

3.3.5 Accessing Internal Signals

If you want to use ZeBu Dynamic Probe feature, you must authorize PLI access to simulator signals. This can be done in three ways:

1. Using the `+acc+2` option: when this option is given to VCS at compilation time, it authorizes the PLI functions to access any signal in the simulator. This is the simplest way to access signals. It works for all cases, but it may slow down the co-simulation.

```
$> vcs [options] <testbench_files.v> zebu.work/<top_module>.v
       $ZEBU_ROOT/lib/libZebuPLI.so +acc+2
```

2. Using the generated `pli.tab` file: when using an HDL driver in ZeBu, **`zDbPostProc`** generates a `pli.tab` file in the `zebu.work` directory. Using this file at compilation time will have the same effect as `+acc+2` but only on the modules that require access to their contents. When using this new `pli.tab` file, the `+acc+2` option is not necessary.

```
$> vcs [options>] <testbench_files.v> zebu.work/<top_module>.v
       -P zebu.work/pli.tab $ZEBU_ROOT/lib/libZebuPLI.so
```

3. Using the default `pli.tab` file: if a problem occurs using the generated `pli.tab` file, you can use a generic `pli.tab` file. It is used in the same way as described in the previous paragraph. This `pli.tab` file is in the `ZEBU_ROOT lib` directory. Using this file at compilation time will have the same effect as `+acc+2` but only on the modules that require access to their contents. When using this new `pli.tab` file, the `+acc+2` option is not necessary.

```
$> vcs [options] <testbench_files.v> zebu.work/<top_module>.v
      -P $ZEBU_ROOT/lib/pli.tab $ZEBU_ROOT/lib/libZebuPLI.so
```

3.3.6 Save and Restore

Save and Restore operation is not currently possible with VCS.

3.3.7 Generating Pattern Files for Non-Regression Testing

With VCS, the appropriate syntax to launch ZeBu co-simulation with generation of a pattern file is the following one:

```
$> simv +zebu.dumpfile=<pattern>.bin
```

3.4 Co-Simulation with NC-Verilog

3.4.1 All-in-one Co-Simulation with NC-Verilog

There are different ways of using NC-Verilog for co-simulation. In this example, **the compilation and run are done with the same command**.

The command to use NC-Verilog for an HDL co-simulation is slightly different to the command used in simulation:

- Add the `libZebuPLI.so` to the command in order to make the link between the simulator and ZeBu.
- Postfix the library with "`bootstrap.bootstrap`".

The connection to the board is done automatically when the run begins.

The following compares the pure simulation command with the co-simulation command:

Syntax for simulation:

```
$> ncverilog [options>] <testbench_files.v> <DUT_files.v>
```

Syntax for ZeBu co-simulation, including the PLI library:

```
$> ncverilog [options>] +access+rw +nbasync \
   +loadpli1=$ZEBU_ROOT/lib/libZebuPLI:bootstrap.bootstrap \
   <testbench_files.v> zebu.work/<top_module>.v
```

Don't forget the two following NC-Verilog options:

- `+access+rw`: allows ZeBu to have access to internal signals.
- `+nbasync`: enables correct read/write synchronization regarding Verilog PLI (refer to Cadence documents for more information).

3.4.2 Co-Simulation Using Incremental Compilation with ncelab and ncsim

The instructions for incremental compilation are similar to the all-in-one compilation. However you should **not use the `+access+rw` or `+nbasync` options** mentioned in the above note.

The steps for incremental compilation are the following ones:

```
$> ncvlog -messages [options] <testbench_files.v>
$> ncvlog -messages [options] zebu.work/<top_module>.v
$> ncelab -messages -access +rw -loadpli1
$ZEBU_ROOT/lib/libZebuPLI:bootstrap.bootstrap <testbench_topmodule>
```

To launch the simulator for co-simulation:

```
$> ncsim -messages +nbasync <testbench_topmodule>
```

3.4.3 Using a VHDL Testbench

First operation is to compile the Verilog wrapper generated by **`zDbPostProc`** and testbench (which includes the VHDL component description for the wrapper):

```
$> ncvlog [options] zebu.root/<top_module>.v
$> ncvhdl [options] <testbench_files.vhdl>
$> ncelab –loadpli1 $ZEBU_ROOT/lib/libZebuSim.so:boostrap –access
+rwc <library_name>.<testbench_topmodule>:<architecture>
```

To launch the simulator for co-simulation:

```
$> ncsim <library_name>.<testbench_topmodule>
```

Note: When using a VHDL testbench with ZeBu, your HDL simulator must support the multi-language feature, requiring both VHDL and Verilog licenses.

3.4.4 Save and Restore with NC-Verilog

In the NC-Verilog graphical interface, the simulation menu proposes a **save checkpoint** option and **restart from checkpoint** option.

When using a command line environment, you can perform save and restore using the following commands:

Save:

```
$ncsim> save top:ckpt1
```

Restore/Restart:

```
$ncsim> restart top:ckpt1
```

or

```
$ncsim> restart :ckpt1
```

The following option displays the available snapshots:

```
restart -show
```

3.4.5 Generating Pattern Files for Non-Regression Testing

For NC-Verilog, the appropriate syntax to launch ZeBu co-simulation with generation of a pattern file is the following one:

```
$> ncverilog [options] +access+rw +nbasync \
   +loadpli1=$ZEBU_ROOT/lib/libZebuPLI:bootstrap.bootstrap \
   <testbench_files.v> zebu.work/<top_module>.v \
   +zebu.dumpfile=<pattern>.bin
```

3.4.6 Co-Simulation transcript

```
ncverilog ../src/top.v +loadpli1=/home/common/zebu/lib/libZebuPLI:bootstrap.bootstrap
+zebu.work=../zebu.work +zebu.verbose +access+rw +nbasync
ncverilog: 05.00-s005: (c) Copyright 1995-2003 Cadence Design Systems, Inc.
file: ../src/top.v
      module worklib.top:v
              errors: 0, warnings: 0
file: ../zebu.work/counter8.v
      module zebu.counter8:v
              errors: 0, warnings: 0
file: counter8.v
      module rundir_maui.counter8:v
              errors: 0, warnings: 0
              Caching library 'rundir_maui' ....... Done
              Caching library 'worklib' ....... Done
              Caching library 'zebu' ....... Done
      Elaborating the design hierarchy:
      Building instance overlay tables: .................... Done
      Generating native compiled code:
              rundir_maui.counter8:v <0x36a9ff77>
                      streams:   4, words:  1268
              worklib.top:v <0x7070692a>
                      streams:   8, words:  5391
              zebu.counter8:v <0x68630d64>
                      streams:   1, words:   303
      Loading native compiled code:     ..................... Done
      Building instance specific data structures.
      Design hierarchy summary:
                              Instances  Unique
              Modules:                3       3
              Registers:              7       7
              Scalar wires:          10       -
              Vectored wires:         4       -
              Always blocks:          3       3
              Initial blocks:         2       2
              Pseudo assignments:     5       5
              Simulation timescale:  1ns
      Writing initial simulation snapshot: worklib.top:v
Loading snapshot worklib.top:v ..................... Done
```

```
      ################################################
      #           Copyright (c) 2002-2006          #
      # Emulation and Verification Engineering  SA #
      #--------------------------------------------#
      # ncsim                                      #
      # revision :                                 #
      # date :                                     #
      #--------------------------------------------#
      ################################################

# ncsim -MESSAGES -XLSTIME 1114537921 -XLVERSION TOOL:    ncverilog      05.00-s005 -NBASYNC
-LICQUEUE +zebu.work=../zebu.work +zebu.verbose +licq
+loadpli1=/home/common/zebu/lib/libZebuPLI:bootstrap.bootstrap +nbasync -XLMODE
./INCA_libs/snap.nc -xlname ncverilog -CDSLIB ./INCA_libs/snap.nc/cds.lib -HDLVAR
./INCA_libs/snap.nc/hdl.var -LCK_FD 5 -CHECK_VERSION TOOL:       ncverilog      05.00-s005 -
LOG_FD 4 -XLARGS +licq ../src/top.v
+loadpli1=/home/common/zebu/lib/libZebuPLI:bootstrap.bootstrap +zebu.work=../zebu.work
+zebu.verbose +access+rw +nbasync

step : end of compile
step : initialize
info : zebu.work path is ../zebu.work

-- ZeBu : ncsim : "default_process" is a full-capability process.
-- ZeBu : ncsim : Looking for a connection ...

      ################################################
      #           Copyright (c) 2002-2006          #
      # Emulation and Verification Engineering  SA #
      #--------------------------------------------#
      # zServer                                    #
      # revision :                                 #
      # date :                                     #
      #--------------------------------------------#
      ################################################

# zServer -design /auto/mars/QA/XL/V1.3_0/05-04-22-
maui/counter8/HDL/rundir_maui/designFeatures -zebu.work zebu.work

Checking for license ... WAITING
Checking for license ... OK
-- ZeBu : zServer : Evaluating the file designFeatures" to get the design's features.

-- ZeBu : zServer : Evaluating the file designFeatures" to get the controlled clock
"B0.clock"'s features.
-- ZeBu : zServer : Looking for a ...

-- ZeBu : zServer : A 1 board serial link chain is required (board : ZeBu-XL-2).
-- ZeBu : zServer : The ZeBu-XL-2 board "zebu_0130" is free.
-- ZeBu : zServer : Get the connection.
-- ZeBu : zServer : Connection established.

-- ZeBu : zServer : The ZeBu-XL-2 board on PCI slot 02:0b.0 will use the interrupt line
number 11.

-- ZeBu : zServer : Initialization of the ZeBu-XL-2 board "zebu_0130" (ZeBu-XL family) -
(maui.eve z_0).

-- ZeBu : zServer : Initialize system frequency to 50 MHz.
-- ZeBu : zServer : System frequency Initialisation succeeded.

-- ZeBu : zServer : ZCPB board is detected.
-- ZeBu : zServer : Module 0 is detected on ZCPB board.
-- ZeBu : zServer : Module 1 is detected on ZCPB board.
-- ZeBu : zServer : Module 2 is detected on ZCPB board.
-- ZeBu : zServer : Module 3 is detected on ZCPB board.
```

```
-- ZeBu : zServer : Control Fpgas reset phase in progress ...
-- ZeBu : zServer : Control Fpgas reset phase Succeeded
-- ZeBu : zServer : Design Fpgas reset phase in progess ...
-- ZeBu : zServer : Design Fpgas reset phase Succeeded

-- ZeBu : zServer : XDR initialization.

-- ZeBu : zServer : Design path : zebu.work
-- ZeBu : zServer : Design loading in progress .....
-- ZeBu : zServer : Hubs loaded           : -4-
-- ZeBu : zServer : Module 0 Fpgas loaded : -1-

-- ZeBu : zServer : Initialization succeeded.

-- ZeBu : zServer : Waiting for client(s).

-- ZeBu : ncsim : Connection established.

-- ZeBu : zServer : Getting the clock "B0.clock"'s features.
-- ZeBu : zServer : The maximum frequency of the clock "driverClk" is setup to 3 MHz.
info : Zebu board 0 opened
-- ZeBu : zServer : Initialize "counter8_hdlcosim::txp".

-- ZeBu : zServer : Initialize "counter8_hdlcosim::rxp".

info : HDL driver counter connected

-- ZeBu : ncsim : XDR activated.
-- ZeBu : zServer : ZeBu-XL-2 board 0, the system frequency is 50 MHz.

-- ZeBu : zServer : For your chosen frequencies (hub 41 MHz = Data Rate 82000 Mb/s), the
optimum frequency of the clock "driverClk" is 7.820 MHz.
-- ZeBu : zServer : WARNING : Due to cosim driver setup, the maximum frequency of the clock
"driverClk" is 3 MHz.
-- ZeBu : zServer : The frequency of the clock "driverClk" (3 MHz) will be adjusted to the
closer inferior available frequency (2.941 MHz).
-- ZeBu : ncsim : INFO   : Zebu board 0 is initialized.
info : Zebu board 0 initialized
step : Checking
step : Checking module : counter8_hdlcosim
step : Checking inputs
step : Checking outputs
step : Checking internal state capture
step : Checking clock specifications about module : counter8_hdlcosim
step : Check done
ncsim> source /usr/import/cadence/cadence5/tools/inca/files/ncsimrc
ncsim> run

step : finish
-- ZeBu : ncsim : INFO   : Elapsed time for initialization : 7
-- ZeBu : ncsim : INFO   : Elapsed time for emulation      : 24
-- ZeBu : ncsim : INFO   : Total elapsed time              : 31

-- ZeBu : ncsim : Waiting for zServer (17757) to stop...


-- ZeBu : zServer : End of run :
-- ZeBu : zServer : driverClk cycle counter : 71,823,030.
-- ZeBu : zServer : B0.clock::clock cycle counter : 1,000,003.

-- ZeBu : zServer : XDR disabled.

-- ZeBu : zServer : Server closed correctly.

-- ZeBu : ncsim : CLOSING ZEBU

info : Zebu board 0 closed
Simulation complete via $finish(1) at time 20000070 NS + 0
../src/top.v:24     $finish;
ncsim> exit
```

3.5 Starting the zRun Utility

In addition to your HDL simulator, you can use **zRun**, ZeBu Emulation Control Interface, to access some advanced ZeBu features (e.g. multi-driver operation, memory/register operation for advanced debug).

You can start **zRun** *before* executing the co-simulation (in this case **zRun** starts the testbench), or you can start **zRun** *during* co-simulation (the testbench is already running). The two sections that follow describe both situations.

Refer to the *ZeBu zRun Emulation Control Interface Manual* for more information about **zRun**.

3.5.1 Starting zRun Before Co-Simulation

Use a command line to start **zRun** and execute the testbench with appropriate options.

There are two ways to do this:

1. If your simulator allows/requires options setting, include them inside double quotation marks.
   ```
   $> zRun -testbench "simv +zebu.work=../zebu.work +vcs+lic+wait"
   ```

2. If you use a `makefile` to launch your testbench:
   ```
   $> zRun -testbench "make run"
   ```
   Where `run` is the name of your `makefile` target.

3.5.2 Starting zRun During Co-Simulation

You can start **zRun** when your simulator interface is open. In your shell, type:

```
$> zRun -attach [PID]
```

Where `<PID>` is the identification of the Linux process that corresponds to the server being used (this option can be omitted if only one Linux process accesses the ZeBu system).

The PID is displayed on the screen:

```
...
-- ZeBu : zServer : Looking for a connection (pid 26262 at <date>) ...

-- ZeBu : zServer : The ZeBu board "zebu_02f1" is free.
-- ZeBu : zServer : Get the connection at <date>.
-- ZeBu : zServer : Connection established.
...
```

4 Advanced Clock Management

In a simulation session, the primary clock waveforms are typically generated in the HDL testbench by using some non-synthesizable HDL code. The design may also include non-synthesizable blocks to generate the internal clocks such as PLL, DCMs…

In an HDL co-simulation session, the primary clock waveforms are still generated by the HDL testbench (thus the testbench remains unchanged), but the primary clocks are typically connected to the DUT mapped onto ZeBu via the ZeBu clock generator. Furthermore, the PLL or DCM are either replaced by an equivalent synthesizable block or moved to the HDL simulator so that they can be simulated while the rest of the design is running in ZeBu.

4.1 Clock Declaration in the DVE file

In HDL co-simulation, ZeBu can handle an unlimited number of clocks. The first 8 clocks (generally the main ones in term of speed and fan-out) are declared in the DVE file as `zceiClockPort` and are connected to the ZeBu clock generator, the others being declared as regular data inputs of the HDL driver.

You can specify the first 8 clocks to the DVE generator (in **`zNetgen`** or **`zCui`**), so that it generates automatically the `zceiClockPort` declarations.

When writing manually the DVE file, here are some recommendations about clock declarations:

- The DUT clocks must be driven by the HDL testbench.
- The first 8 main clocks are specified as `zceiClockPort` and declared through parameters, namely, "`clock_0`", "`clock_1`" … "`clock_7`". **Declare the clock parameters in sequence, in ascending order, starting from 0.**
- Each HDL driver must be connected to at least one clock declared as a `zceiClockPort`.
- If you have more than 8 clocks, the additional clocks are specified as regular data signals driven by the ZeBu system. These clocks generate clock skews in the ZeBu mapping, and it is necessary to offset those clock skews as described in the *ZeBu Compilation Manual*.

The DVE file only includes the clock declaration and the way the clocks are connected to the design. All other clock characteristics (frequency, phase, duty cycle...) are specified in the HDL testbench.

Example:

Below is a portion of a DVE file using three clocks (`main_clock, read_clock`, and `write_clock`) declared and driven by a set of three `zceiClockPort`.

```
HDL_COSIM topmodule_HDLCosim(
  .input_bin({
[…]
  }),
  .output_bin({
[…]
  }));

defparam  topmodule_HDLCosim.clock_0 = main_clock;
defparam  topmodule_HDLCosim.clock_1 = read_clock;
defparam  topmodule_HDLCosim.clock_2 = write_clock;

zceiClockPort clock_ClockPort0 (
    .cclock( main_clock )
);
zceiClockPort clock_ClockPort1 (
    .cclock( read_clock )
);
zceiClockPort clock_ClockPort2 (
    .cclock( write_clock )
);
```

4.2 Using non-synthesizable clock generators

The design may include some non-synthesizable blocks such as PLL or DCM to generate internal clocks. With HDL co-simulation, it is possible to move those blocks to the HDL simulator so that there is no need to re-model such blocks with equivalent synthesizable blocks.

During the RTL synthesis and the ZeBu compilation, the PLL and DCM are replaced in the design by black-boxes to keep all the logic surrounding those blocks in the ZeBu implementation.

Furthermore, it is necessary to manually modify the DVE file to connect the embedded black-boxes to the HDL simulator and to include the original models of the PLL and DCM in the Verilog wrapper or the HDL testbench.

4.2.1 Modifying the DVE file

The DVE file is modified to include the connections to the ports of the non-synthesizable models.

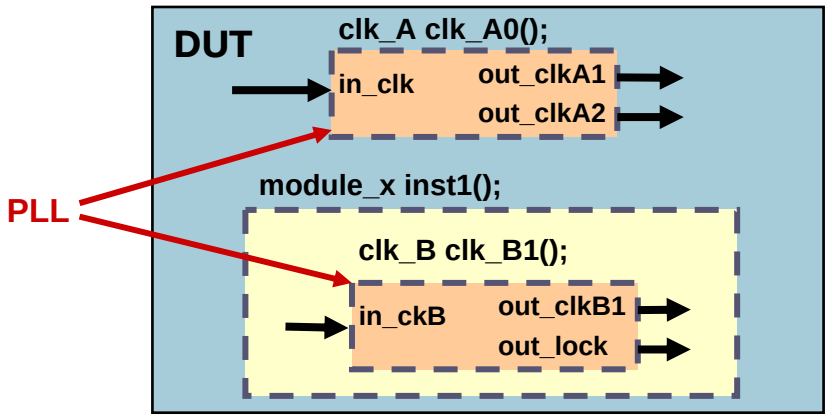


Figure 6: Connecting a non Synthesizable Clock Generator

In the following example, a design includes two PLLs, instantiated under the names `clock_A0` and `inst1.clk_B1`. The `input_bin` portion of the HDL driver is modified to include the output ports of the two PLLs and the `output_bin` portion the input ports of the two PLLs.

```
.input_bin ({
[…],
   dut.clk_A0.out_clkA1,
   dut.clk_A0.out_clkA2,
   dut.inst1.clk_B1.out_clkB1,
   dut.inst1.clk_B1.out_lock,
[…]
 }),

.output_bin ({
[…],
   dut.clk_A0.in_clk,
   dut.inst1.clk_B1.in_ckB,
[…],
 }),

[…]
defparam  topmodule_HDLCosim.clock_0 = dut.clk_A0.out_clkA1;
defparam  topmodule_HDLCosim.clock_1 = dut.clk_A0.out_clkA2;
defparam  topmodule_HDLCosim.clock_2 = dut.inst1.clk_B1.out_clkB1;

zceiClockPort out_clkA1Port (
    .cclock( dut.clk_A0.out_clkA1 )

zceiClockPort out_clkA2Port (
    .cclock( dut.clk_A0.out_clkA2 )

zceiClockPort out_clkB1Port (
    .cclock( dut.inst1.clk_B1.out_clkB1 )
```

The outputs of the two PLLs are routed via the internal clock generators of ZeBu.

4.2.2 Verilog wrapper or HDL testbench modification

The Verilog wrapper is modified to include the instantiation of the non-synthesizable models as follows.

```
// Clock generator 1
wire dut_clk_A0_in_clkA,
     dut_clk_A0_out_clkA1,
     dut_clk_A0_out_clkA2;

clk_A clk_A0(.in_clkA(dut_clk_A0_in_clkA),
.out_clkA1(dut_clk_A0_out_clkA1),
.out_clkA2(dut_clk_A0_out_clkA2));

// Clock generator 2
wire dut_inst1_clk_B1_in_clkB,
     dut_inst1_clk_B1_out_clkB1,
     dut_inst1_clk_B1_out_lock;

clk_B clk_B1(.in_clkB(dut_inst1_clk_B1_in_ckB),
.out_clkB1(dut_inst1_clk_B1_out_clkB1),
.out_clkB2(dut_inst1_clk_B1_out_lock));

// DUT with original ports
dut dut0( .in1(xxx),…, .outN(zzz),
// Additional ports
     .dut_clk_A0_in_clkA(dut_clk_A0_in_clkA),
     .dut_clk_A0_out_clkA1(dut_clk_A0_out_clkA1),
     .dut_clk_A0_out_clkA2(dut_clk_A0_out_clkA2),
     .dut_inst1_clk_B1_in_clkB(dut_inst1_clk_B1_in_clkB),
     .dut_inst1_clk_B1_out_clkB1(dut_inst1_clk_B1_out_clkB1),
     .dut_inst1_clk_B1_out_clkB2(dut_inst1_clk_B1_out_lock)
     );
```

Alternatively, the HDL testbench can be modified to include the instantiations of the two PLLs. In that case, there is no need to modify the Verilog wrapper each time it is generated (for example when generating dynamic probes). However, it might be necessary to include an additional layer in the HDL testbench for instantiating the PLLs, thus changing the hierarchy of the design and requiring the design hierarchical references to be modified in the testbench.

4.3 Optimizing the Speed with Synchronous Interface

If the DUT interface is fully synchronous to the design clocks, you can optimize the speed of co-simulation by limiting the data communication between the simulator and ZeBu on the design clock edges. This feature is not recommended early in the design-cycle; it is intended to optimize the performance of an operational design.

This feature may change the co-simulation behavior if the DUT interface includes some signals which are not synchronous with the specified design clock: edges of such signals are re-synchronized and exchanged between the simulator and ZeBu only on a clock edge. For example an asynchronous reset is re-synchronized on a clock edge.

With this feature, the simulation sends signal values to the DUT interface only on a clock edge. If a signal changes many times between two clock edges, the only two values sent to the DUT are those on the two edges of the clock. If more than one clock is used in the DUT, all the edges of all the clocks update the DUT interface.

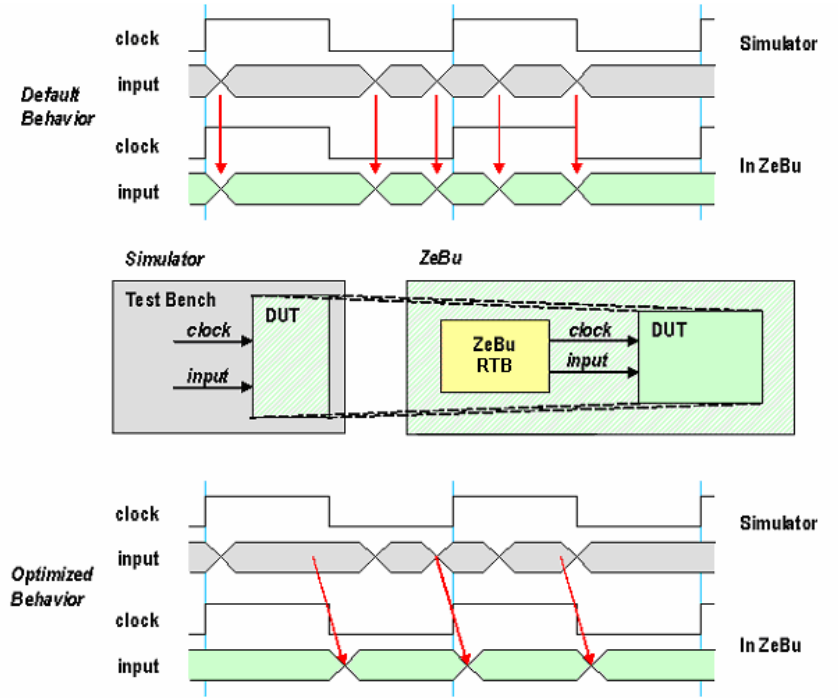


Figure 7: Optimized Clock Behavior

Figure 7 shows waveforms that correspond to the default behavior and contrasts that to the clock synchronous behavior:

- For the default behavior, a signal is updated on the DUT interface at each time it changes.
- When optimized, the input signal is updated on the DUT interface only when a clock edge occurs. If this input signal changes between two clock edges, the value is not updated on the DUT interface. As the input signal is supposed to be synchronous to the clock for this optimization, the DUT behavior remains the same.

The optimization declaration is done with the `+zebu.sensitivity` option at run-time in the simulator command line, as described in Section 3.1. The synchronization signals for communication between the DUT and the HDL testbench are defined as parameters for this option:

- For communication on any edge of the controlled clocks:
  ```
  +zebu.sensitivity=clock_edges
  ```
- For communication on edges of user defined interface signals:
  ```
  +zebu.sensitivity=<list_of_signals>
  ```
  Where the signal names are separated with a comma or a space.
- For communication on any modification of the ZeBu input signals (non-optimized behavior):
  ```
  +zebu.sensitivity=*
  ```

Important Note:

Using this option can introduce some erroneous co-simulation result: when ZeBu input data signals change asynchronously before the synchronization signals, the value is not propagated before the data are exchanged. When the synchronization signal is a clock and the input data is sampled by this clock, the design may not work as expected.

A possible workaround is that your DUT must be active only on one clock edge and your testbench must change input values only on the same clock edge or between this active edge and the opposite edge. This ensures that input signal changes are present on the DUT interface before next active clock edge.

4.4 Specification of Delay for Output Ports

To avoid races between signals driven by ZeBu, you can add some delay on the output or bi-directional ports of your DUT. The delay is expressed as a number of time units of your simulator.

This feature is typically intended to delay data output signals which are sampled in the HDL testbench by an output clock generated by ZeBu.

This option is added in the HDL simulator command line with the following syntax:

```
+zebu.delayfile=<filename>
```

Where `<filename>` is a manually written file in which each delayed signal is declared with its applicable delay.

The syntax of the file is very simple, since one line contains the name of the signal and the value of the delay:

```
<signal_name_1> <delay_1>
[...]
<signal_name_n> <delay_n>
```

Where `<signal_name>` can be either an output or a bidirectional signal; and `<delay_value>` is expressed according to the current timescale.

Note that for bidirectional signals, the delay is actually inserted only when the signal is driven by the ZeBu system.

When used with the `+zebu.verbose` option, the list of delayed signals is displayed:

```
signal '%s' will be delayed of '%d' in output.
```

5 Accessing Internal Signals

In HDL co-simulation, it is possible to access internal signals of the DUT within the simulator. Adding internal visibility reduces the co-simulation speed, but the visibility can be turned on or off interactively as required.

5.1 Signal Types

There are two signal types that you can access in the DUT: sequential and combinational signals, requiring each a specific access type.

5.1.1 Sequential Signals

You can access sequential signals, i.e. outputs of flip-flops or latches, during HDL co-simulation by selecting dynamic probes, using **`zDbPostProc`** utility.

5.1.2 Combinational Signals

You can access combinational signals of the DUT during HDL co-simulation by using **combinational dynamic probes** or **static probes** declared in the ZeBu compilation process. Refer to Chapter 13 of the *ZeBu Compilation Manual*.

5.2 Selecting Dynamic Probes

Dynamic probes are used to open at run-time a dynamic trace window into your design. They are assigned after compilation without requiring any modification of the DUT. Note that using dynamic probes substantially impacts the achievable run-time frequency, but you can interactively enable and disable probing as needed.

5.2.1 zDbPostProc Description

`zDbPostProc` allows to browse the ZeBu database resulting from the compilation process (`ZeBuDB.zdb`). **`zDbPostProc`** is especially intended for selection of dynamic probes and generation of the co-simulation wrapper, but it may be used for other purpose such as collecting information about the database, modifying the database, modifying the memory structure for run-time access to memories created without **`zMem`**, ...

Both graphical and batch modes are available for probes selection with **`zDbPostProc`**:

- **`zSelectProbes`** is a Graphical User Interface (GUI) to access **`zDbPostProc`**,
- **`zDbPostProc`** can be accessed in interactive mode or with Tcl script for batch mode.

`zDbPostProc` generates a co-simulation wrapper, as described in Figure 2, as well as two probe files for use in a later run-time:

- `ZebuPrb.lst`: lists the selected dynamic probes
- `ZeBuPrb.tcl`: Tcl script that can be sourced for use in batch mode.

5.2.2 Using the zDbPostProc Graphical Interface

5.2.2.1 Launching zSelectProbes

Use the **`zSelectProbes`** command to call the graphical interface of **`zDbPostProc`**.

`zDbPostProc` graphical interface displays signals and vectors at all levels of the design hierarchy, and allows selecting the internal probes for design debugging. There is no limit on the number of internal probes, as long as they monitor sequential signals, i.e. outputs of registers or latches.

`zSelectProbes` may take two options :

- `-p` the path of the `zebu.work` directory (default is `./zebu.work`)
- `-i` name of the database (default is `ZebuDB.zdb`)

Example:

```
$> zSelectProbes -p ../zebu.work
```

5.2.2.2 zDbPostProc Graphical Interface Description

Following is the Graphical User Interface for **`zDbPostProc`**:

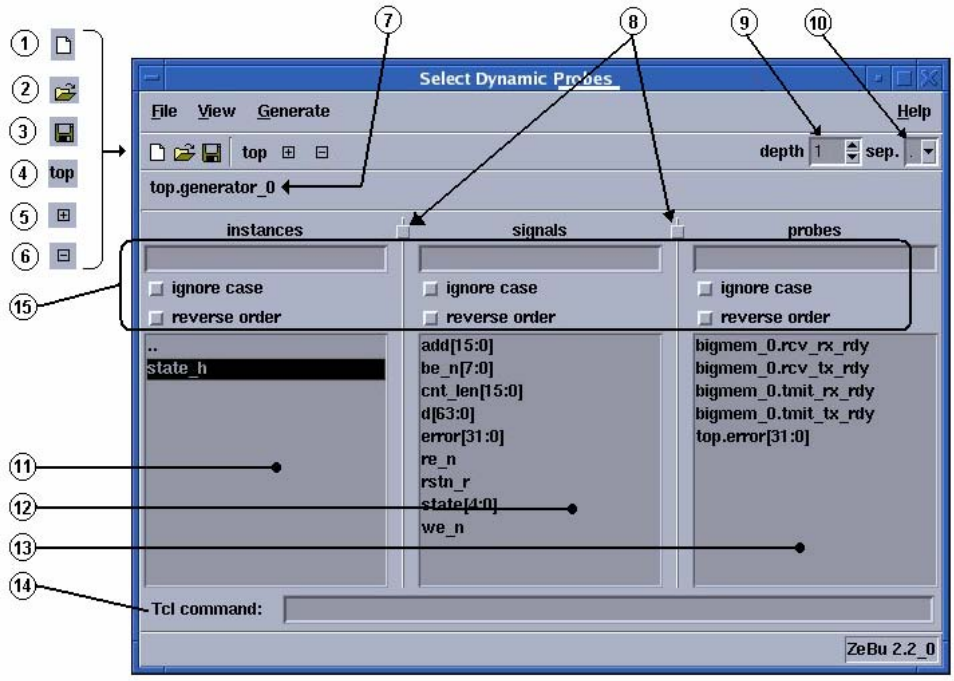


Figure 8: zDbPostProc Graphical User Interface

1	Clear the probes pane	9	Depth selector for current instantiation
2	Open or load a netlist	10	Select the hierarchy separator character
3	Save the selection	11	Design hierarchy pane
4	Navigate to the top of the design	12	Signals/wires pane
5	Add (select) a signal	13	Selected probes pane
6	Remove the selected probe(s)	14	Tcl command line text box
7	Current location in design hierarchy	15	Display filters
8	Pane resize handle		

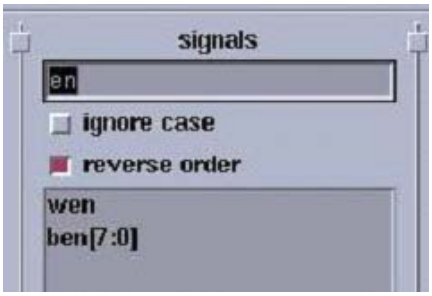
The left pane is the hierarchy browser: it displays the instances in your design EDIF netlist. Click on an instance to display the signals and wires in the center pane. Double-click on an instance to enter its hierarchy.

The center pane, the signals/wires pane, lists ports or wires for a selected instance. Double-click on a signal or wire to select the signals. For vectors, the entire vector is selected.

The "**depth"** selector sets the level of hierarchy that is displayed.

The right pane displays the selected probes. If you have not previously selected signals for your project, this pane is blank.

For each pane, the display filters have three functions:



Filter field: used to enter a string in order to display only signals containing the string as part of their name. For example, in the figure below, only signals containing the string "en" are displayed.
ignore case: displays the names in lower case.
reverse order: displays the pane list in reverse alphabetical order.

The following menus are available: **File**, **View** and **Generate**.

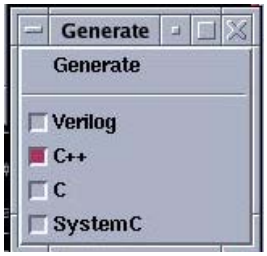
The **File** menu has the following options:

Clear all: clears all the probes from the probes pane.
Load: loads a saved probe list (typically a former `ZebuPrb.lst`).
Save: saves the current selected probes in the database and generate a probe file with the selected separator (see item 10 in the GUI figure).
Save As: allows to specify a name for your probe file. You must specify this new name manually when you want to use a probe file that has a name other than the default name.

The **View** menu has the following options:

Xilinx Primitives: Display Xilinx primitives.
EDIF Renamed: Display any instance aliases.

The **Generate** menu has the following options:



You can create a wrapper containing the selected probes for the selected co-simulation mode (Verilog, C++, C, or SystemC)

5.2.2.3 Adding Probes

1. Double-click on an instance name in the design hierarchy pane to display any instances declared inside the current instance. The list of available ports/signals is displayed in the center pane.
   a. To display Xilinx primitives, select the **Xilinx Primitives** option in the View menu (if specific option was set for database generation).
   b. To view any aliases that you have created, select the **EDIF Renamed** option.
2. Click on the signal or port that you want to add and click **Add (+)**. To select all of the wires or ports: click on the instance name in the left pane, then click **Add (+)**.
3. Click **Save** to save your selection of probes. This generates the probe files.
4. In the **Generate** menu, select the **Verilog** environment to generate the appropriate wrapper for HDL co-simulation, and then select **Generate**.

The following files are now available:

- `ZeBuPrb.lst` and `ZeBuPrb.tcl`: probe files for a later run-time.
- `<driver_name>.v`: Verilog wrapper, which name is defined in the DVE file.

You may now close the **`zDbPostProc`** Graphical Interface.

5.2.3 Selecting Probes in Batch Mode

5.2.3.1 Launching zDbPostProc

To select probes without using the graphical interface, you can use **`zDbPostProc`** in batch mode, with an input Tcl script or in interactive mode:

```
$> zDbPostProc [my_script.tcl] [-p <zebu.work>] [-i database name]
```

Where `[my_script.tcl]` is the optional Tcl script (if not present, interactive mode is used); `-p` is the optional path to the `zebu.work` directory (default is `./zebu.work`); `-i` is the optional name of the database (default is `ZebuDB.zdb`).

For further information, **`zDbPostProc`** includes on-line help using two options:

`-help` → lists all the commands
`-help <command>` → provides detailed information to use `<command>`

The following sections give the basic commands for the selection of dynamic probes and for the generation of the co-simulation wrapper.

5.2.3.2 Selecting an object

You can select signals or instances as dynamic probes:

```
select [-depth <my_depth>] [-all] [-xilinx] <object list>
```

Where `<my_depth>` is the optional number of hierarchical levels; `[-all]` option adds all the objects in the selection; `[-xilinx]` option includes the Xilinx primitives in the selection.

5.2.3.3 Unselecting an object

You can remove an object (signal or instance) from the list of dynamic probes:

```
remove <my_object>
```

Where `<my_object>` is the signal or instance that will no longer be available in the selection.

5.2.3.4 Clear of all objects

You can remove all the signals and instances from the list of available dynamic probes:

```
clear_probes
```

5.2.3.5 Saving database after modifications

Once you have modified your database, you need to save it and generate the corresponding file of signals and instances available as dynamic probes.

```
save_db [my_probe_file]
```

Where `[my_probe_file]` is the optional name for the probe file which is generated at the same time as the modified database (default is `ZebuPrb.lst`).

5.2.3.6 Loading probes from file

You can import the probes from an existing probe file (typically `ZebuPrb.lst`):

```
load_probes <my_probe_file>
```

5.2.3.7 Generating the HDL Co-simulation wrapper

Generating the HDL co-simulation wrapper is mandatory to access the dynamic probes at run-time from the testbench. If you are using a VHDL testbench, you can generate a specific Verilog wrapper that you will use to create the VHDL component, as described in Section 2.4.

```
wrapper -<my_wrapper> –[my_resolution]
```

Where `<my_wrapper>` can be either `verilog` or `vhdl`; `[my_resolution]` is the resolution (`pullup`, `pulldown` or `keeper`) that will be used for bidirectional signals in the Verilog wrapper (if not set explicitly, no resolution is applied).

5.2.3.8 Script Example for **`zDbPostProc`**

```
# selecting 5 level of hierarchy under scope top_design
select -depth 5 design_top

# saving selection in database and in probe file 'my_probefile.lst'
save_db my_probefile.lst

# generationg hdl wrapper using pullups
wrapper -hdl –pullup
```

5.3 Accessing Dynamic Probes

This section explains how to manage the access to the dynamic probes from the Verilog testbench.

Adding Dynamic probes does not change the architecture of the DUT in the simulation environment, except that the parts and instances containing probes become visible for the simulator.

Example:

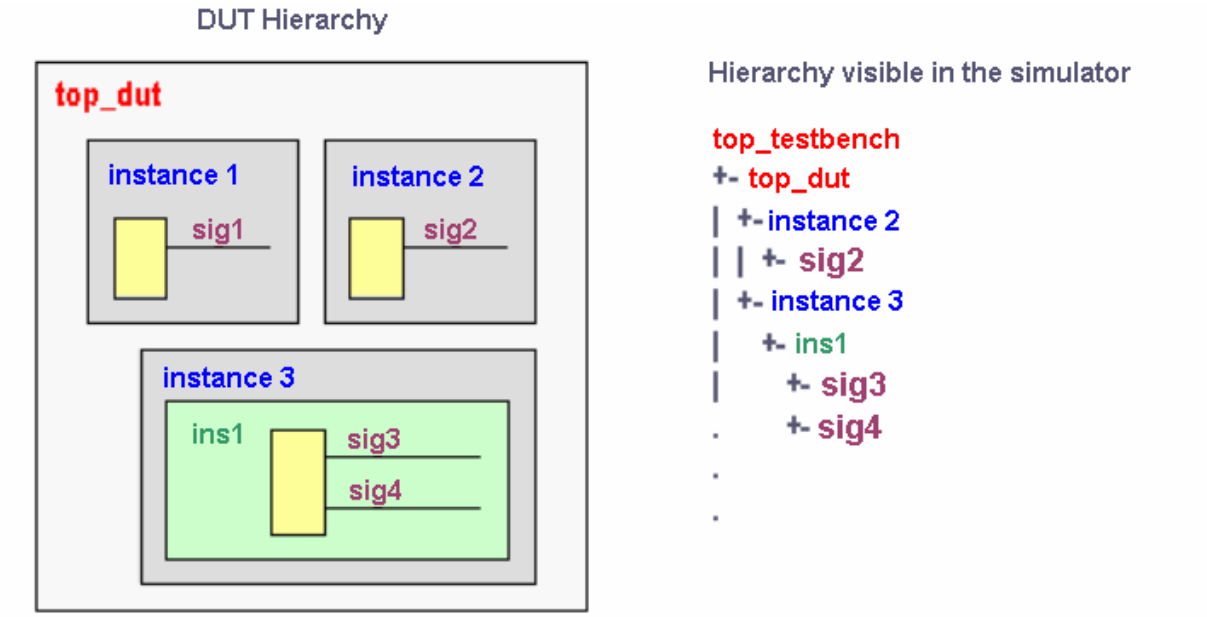


Figure 9: Dynamic Probes in the DUT and Simulator Hierarchies

Figure 9 shows an example of a hierarchical design with 3 internal probes (`sig2`, `sig3` and `sig4`), and an instance (`instance1`) without internal probing. The latter instance is not visible in the simulator environment.

5.3.1 Enabling and Disabling Access to Dynamic Probes

Access to dynamic probes can be enabled and disabled at runtime when required. This is done through a task that can be called directly in the testbench based on conditions that occur during the run.

The task, `ZEBU_readback`, takes one Boolean argument to enable or disable this feature:

- To enable access to dynamic probes:
```
$ZEBU_readback(1);
```
- To disable access to dynamic probes:
```
$ZEBU_readback(0);
```

Note that in ModelSim, you can easily switch on or off this feature by customizing the ModelSim GUI for ZeBu, as described in Section 3.2.6, “Integrating ZeBu Functions in ModelSim”.

Another way to enable and disable access to dynamic probes is to force to 1 the `zebu_readback` signal present in the top module of the generated wrapper.

Note that if the `ZEBU_readback` task is called in an initial statement (Verilog), you should add #0 before it. If not, the call will not be taken into account.

```
initial begin
   #0 ZEBU_readback(1);

   [...]
end
```

5.4 Accessing Static Probes from a Verilog Testbench

Static probes can be used to access DUT internal combinational signals. They are also recommended instead of dynamic probes when there is a need for more performance.

When internal DUT signals are declared in the DVE file, they are automatically managed as Static Probes by **`zBuild`**, without requiring dedicated probe file creation with **`zBrowser`**.

At run-time such signals are available in the HDL testbench as hierarchical references as in a pure simulation environment.

Note: it is possible to delete all hierarchical references for static probes in the Verilog wrapper, using the `delete_static_probes_hierarchy` command in **`zDbPostProc`**, as in the following script example:

```
opendb ZeBuDB.zdb
delete_static_probes_hierarchy
save_db
wrapper -verilog
```

6 Accessing Memories

The testbench can perform memory read and write access at run-time on any design memory.

6.1 zrm Memories

The zrm memories are declared in the DUT by instantiating memory models generated with **zMem**. These memories can be read without stopping the simulation/emulation execution.

No specific actions are required to access those memories from the testbench.

6.2 Embedded Memories

When the embedded Xilinx memories have been created with **zMem**, they can be accessed from the testbench, but require that the clocks are stopped.

If **zMem** was not used, the memory import operation with **zDbPostProc** is required.

6.3 Memory Access

Using the memory name, it is possible to load or dump the memory contents. In the testbench the memory name is the same as that in the hierarchy of the DUT.

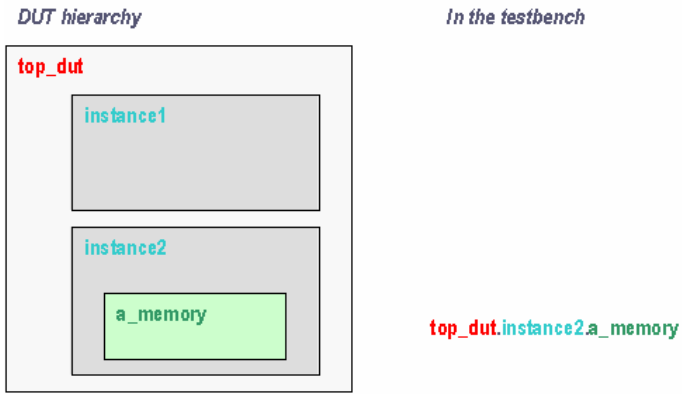


Figure 10: Memory Naming Convention

The read/write methods require the use of a memory content file as described below.

6.3.1 Memory Format File

The memory format file allows the description of all or part of the memory contents. The syntax is simple and is similar to a raw format, as described in the *ZeBu Reference Manual*.

The following example presents the main features of the memory file format:

- By default the addresses and values are given in hexadecimal format, but they can also be given in binary or decimal.
- A value can be given for a specific address or for a range of addresses.

- When no address is specified, the value corresponds to the address following the previously given address.
- If an address is given more than once, the last value for this address is the used (this can be used, for example, to initialize a memory with a value then to specify values for specific addresses).

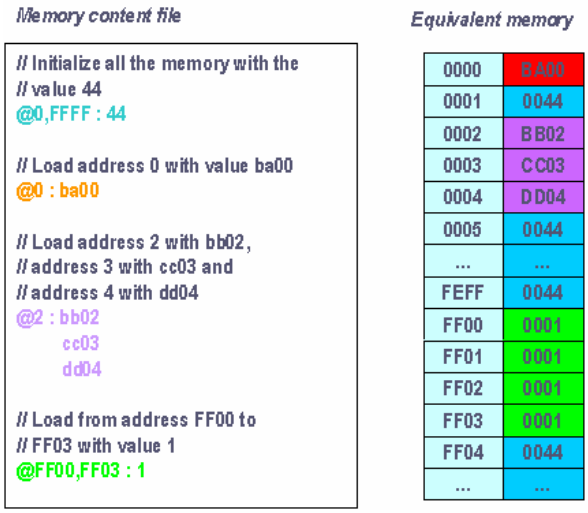


Figure 11: Memory File Example

This format is used for uploading and for downloading the memories.

6.3.2 Upload and Download

The upload and download of a memory contents are performed by calling the corresponding tasks. The name of the file can be given in a relative or absolute path. If no path is given, the file must be in the local directory.

Unlike the Verilog `$readmem`, the second argument of the ZeBu tasks must be a string and not a memory handler.

Uploading consists in reading a file and putting the contents in the stated memory:

```
$ZEBU_readmem("filename", "memory_name");
```

Downloading consists in writing a file with the content of a stated memory:

```
$ZEBU_writemem("filename", "memory_name");
```

or

```
$ZEBU_writemem("filename", "memory_name", first_address, last_address);
```

6.4 Initializing the Memories from ZeBu

Regardless of the HDL testbench, it is possible to load the memories when initializing ZeBu through an initialization file (for example `memory.init`).

This file consists of a collection of lines, each stating a memory name and the name if the corresponding initialization file:

```
<memory_name> <file_name>
<next_memory_name> <next_file_name>
```

The memory name is a complete name including path, as described in Section 6.3. The file name can include either a relative or an absolute path; a relative path is defined from the working directory.

For example, for the memory `a_memory` of `instance2` in the example of Figure 10, if the memory contents file is called `memcontent1`, the line is:

```
top_dut.instance2.a_memory memcontent1
```

The file that describes the memory initialization has to be specified in the `designFeatures` file. The syntax is:

```
$memoryInitDB = "memory.init";
```

6.5 Listing Your Design Memories

You can get the memory list of the design with **`zDbPostProc`**, using the `show` command for memories.

Example:

```
$> echo "show -memories" |zDbPostProc –p .
```

Refer to the *ZeBu Reference Manual* for details about the different options that you can use with **`zDbPostProc`**.

7Non-Regression Testing with zPattern

7.1 Description

This chapter describes the **pattern co-simulation mode** for ZeBu, which is especially useful for non-regression testing of designs.

This mode is not interactive. It is based on a method wherein a pattern file is first generated and secondly re-used for stimulating and monitoring the DUT.

The pattern file is generated during a first HDL co-simulation. The pattern file can then be re-used to stimulate a new version of the DUT without requiring the HDL simulator anymore, assuming that the interface of the DUT has not changed. The pattern execution tool detects differences on output signals between the recorded responses and the actual values. You must initialize the registers and memories of the reference and test designs in the same way.

This mode can also be used to determine the maximum achievable performance without the HDL-simulator but still keeping a similar event-driven communication between ZeBu and the host PC.

7.2 Principle

Using the pattern mode requires two main steps:

- Generate the pattern file database with your reference design, see Section 7.3.
- Use the pattern execution tool, **`zPattern`**, to apply the recorded set of patterns on the new compiled DUT, see Section 7.4.

`zPattern` performs two main functions:

- It reads the pattern database from the defined pattern file and applies the same set of values to the inputs of the DUT,
- It reads the output values from the DUT and compares them against the expected values recorded in the pattern database.

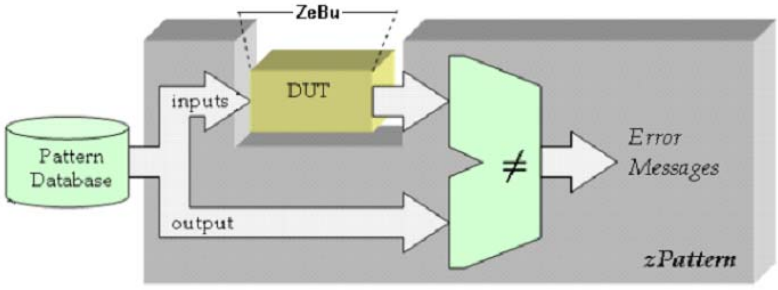


Figure 12: zPattern Principle

When the comparison yields a difference between the expected values and the actual values, **`zPattern`** reports an error in its log. Each message reports the sample number where the difference is detected and the affected signal name. An example of such an error list is shown in Section 7.4.4.

7.3 Generating the Pattern File

The pattern file is generated during the HDL co-simulation of the reference design. This generation is fully automated.

7.3.1 Compilation Requirements

Dumping a pattern file in HDL co-simulation requires no specific action when you compile the DUT for ZeBu.

The same simulator executable or simulator database can be used for generating the patterns.

7.3.2 Pattern Generation at Run-time

At run-time, an option to the simulator enables the generation of the pattern file. This option, `zebu.dumpfile`, takes as value the name of the output pattern file. The file format for use with **`zPattern`** must be the proprietary binary format, which requires a `.bin` extension for the pattern file name.

The options for pattern file generation using ModelSim, VCS or NC-Verilog are described in the relevant co-simulation sections of Chapter 3:

- For ModelSim, see Section 3.2.8,
- For VCS, see Section 3.3.7,
- For NC-Verilog, see Section 3.4.5

The generated file, e.g. `pattern.bin`, contains the activity on the design interface in a proprietary binary format. It must not be modified manually.

The file is located in the directory where the simulation has been executed. If required, you can provide an absolute or relative path to the `zebu.dumpfile` option.

7.4 Applying a Generated Pattern File on the DUT

zPattern returns the number of errors that were detected during the run and potential errors are listed in the log file, identifying the signal(s) and time(s) where errors occurred.

7.4.1 Launching zPattern

zPattern requires the name of the pattern file. The simplest call is:

```
$> zPattern –i <pattern.bin> [options]
```

Where <pattern.bin> is the name of the reference dump file (mandatory).

zPattern accepts the following options:

Syntax	Description	Default
-z <zebu_work_path>	ZeBu compilation directory path.	"./zebu.work"
-d <driver_name>	Name of the driver. This option is used when the database contains more than one driver.	
-n <number_of_pattern>	Specifies the number of patterns to be executed.	All
-m <memory_init_file>	Specifies a file for memory initialization. See Section 7.4.2 for more details.	
-e <nbmax>	Specifies the maximum number of error messages to be displayed.	
-f <startsample>	Specifies the cycle number from which errors are checked.	
-l <stopsample>	Specifies the last cycle to be checked; after this cycle, no more samples are checked for errors.	

7.4.2 Memory Initialization

When generating a pattern database via co-simulation, the internal memories of the design can be initialized at run-time. In order to get the same design behavior, it is essential to initialize the memory with the same contents.

Memory initialization with **zPattern** can be achieved either via the designFeatures file or in the **zPattern** command line, with the –m option:

- When the memories for the reference file have been initialized through the designFeatures file the same procedure can be repeated in pattern mode.
- When the memories are initialized by the testbench (using the ZEBU_readmem task in Verilog or the init method in C++), they can be initialized using the '-m' option in pattern mode.

This option reads a file that lists the memory to be initialized and the contents file to perform the task. The syntax is the same used in the memory feature of the designFeatures:

```
<memory name> <file name>
…
<memory name> <file name>
```

zPattern only supports memories loading at emulation launching.

7.4.3 Typical zPattern Log With no Errors

The following is a typical **zPattern** log when no errors occurred (the **zServer** log is not included):

```
        ##############################################
        #              Copyright (c) 2003-2006       #
        # Emulation and Verification Engineering  SA #
        #--------------------------------------------#
        # zPattern                                   #
        # revision :                                 #
        # date :                                     #
        #--------------------------------------------#
        ##############################################

zPattern -i patternTest1.bin

-- ZeBu : zPattern : default_process is a full-capability process.

-- ZeBu : zPattern : Opening the connection to the ZeBu board "zebu_0020" through
the device "/zebu/z_0" (pid 9901) ...

-- ZeBu : zPattern : INFO   : Zebu board 0 is initialized.
-- ZeBu : zPattern : INFO   : No error detected.

-- ZeBu : zPattern : Waiting for zServer to stop...

-- ZeBu : zPattern : zPattern end.
```

7.4.4 Typical zPattern Log with Errors

The following is a typical **zPattern** log when errors occurred during the run.

```
-- ZeBu : zPattern : WARNING: Error on signal "data_out", at pattern 5487.
-- ZeBu : zPattern : WARNING: Error on signal "data_out", at pattern 5489.
-- ZeBu : zPattern : WARNING: Error on signal "check_sum", at pattern 5489.
-- ZeBu : zPattern : WARNING: Error on signal "data_out", at pattern 19425.
-- ZeBu : zPattern : WARNING: Error on signal "mem_ctrl", at pattern 19425.
```

8 Reference Resources

8.1 ZeBu-UF Documents

For each version, the *ZeBu Release Note* describes the new features, the fixed bugs, the known limits, the evolutions of the documentation package and the compatibility information.

The following manuals constitute the ZeBu-UF documentation package (some are generic manuals for the ZeBu range):

[0] The *ZeBu-UF Product Overview* describes the different ZeBu-UF products and their features.

[1] The *ZeBu-UF Installation Manual* describes how to install the ZeBu-UF software and hardware.

[2] The *ZeBu-UF Compilation Manual* describes the ZeBu-UF compilation process.

[3] The *ZeBu HDL Co-simulation Manual* describes the use of the HDL co-simulation driver for the ZeBu systems.

[4] The *ZeBu C++ Co-simulation Manual* describes the use of the C++ co-simulation driver for the ZeBu systems.

[5] The *ZeBu SystemC Co-simulation Manual* describes the use of the SystemC co-simulation driver for the ZeBu systems.

[6] The *ZeBu Synthesizable Test Bench Manual* describes the use of a Synthesizable Test Bench to drive a target design mapped in ZeBu interface FPGAs.

[7] The *ZeBu Transaction-Based Verification Manual* describes the use of the transaction-based mode for the ZeBu systems.

[8] The *ZeBu zRun Emulation Interface Manual* describes the zRun emulation control interface and how to use the different functions.

[9] The *ZeBu-UF Tutorial* illustrates, via a set of examples, how to use the ZeBu-UF platform for verifying a design-under-test.

[10] The *ZeBu-UF Smart Z-ICE Manual* provides detailed information on how to configure and to connect the Smart Z-ICE interface to an external system.

[11] The *ZeBu-UF Direct ICE Manual* provides detailed information on how to configure and to connect an ICE module for connection emulated DUT I/O pins to a target system and hard cores.

[12] The *ZeBu-UF Reference Manual* provides detailed information on program commands, memory models, libraries, and files necessary to compile and verify a design-under-test.

[13] The *ZeBu C API Reference Manual* and *ZeBu C++ API Reference Manual* provide detailed information on C/C++ library, files, and interfaces necessary to write a C/C++ test bench to verify your design.

8.2 ZeBu-XL Documents

For each version, the *ZeBu Release Note* describes the new features, the fixed bugs, the known limits, the evolutions of the documentation package and the compatibility information.

The following manuals constitute the ZeBu-XL documentation package (some are generic manuals for the ZeBu range):

The *ZeBu-XL Product Overview* describes the different ZeBu-XL products and their features.

The *ZeBu-XL Installation Manual* describes how to install the ZeBu-XL software and hardware.

The *ZeBu-XL Compilation Manual* describes the ZeBu-XL compilation process.

The *ZeBu HDL Co-simulation Manual* describes the use of the HDL co-simulation driver for the ZeBu platforms.

The *ZeBu C++ Co-simulation Manual* describes the use of the C++ co-simulation driver for ZeBu platforms.

The *ZeBu SystemC Co-simulation Manual* describes the use of the SystemC co-simulation driver for ZeBu platforms.

The *ZeBu Synthesizable Test Bench Manual* describes the use of a Synthesizable Test Bench to drive a target design mapped in the ZeBu system FPGAs.

The *ZeBu Transaction-Based Verification Manual* describes the use of the transaction-based mode for ZeBu platforms.

The *ZeBu zRun Emulation Interface Manual* describes the zRun emulation control interface and how to use the different functions.

The *ZeBu-XL Reference Manual* provides detailed information on program commands, memory models, libraries, and files necessary to compile and verify a design-under-test.

The *ZeBu-XL Tutorial* illustrates, via a set of examples, how to use the ZeBu-XL platform for verifying a design-under-test.

The *ZeBu-XL Smart Z-ICE Manual* provides detailed information on how to configure and to connect the Smart Z-ICE interface to an external system.

The *ZeBu-XL Direct ICE Manual* provides detailed information on how to configure and to connect the ZeBu ICE module for connection emulated DUT I/O pins to a target system and hard cores.

The *ZeBu C/C++ API Reference Manual* provides detailed information on C++ library, files, classes and methods necessary to write a C++ test bench to verify your design.

9 EVE Contacts

For product support, contact: support@eve-team.com.

For general information, visit our company web-site: http://www.eve-team.com

Europe Headquarters	EVE SA 2bis, Voie La Cardon Parc Gutenberg, Batiment B 91120 PALAISEAU France Tel: +33-1-64 53 27 30
US Headquarters	EVE-USA, Inc 84 W Santa Clara, Suite 580 San Jose, CA 95113 USA Tel: 1-888-7EveUSA (+1-888-738-3872)
Japan Headquarters	Nihon EVE KK KAKiYA Building 5F 2-7-17, Shin-Yokohama Kohoku-ku, Yokohama-shi, Kanagawa 222-0033 JAPAN Tel: +81-45-470-7811
Korea Headquarters	EVE Korea, Inc. 804 Kofomo Tower, 16-3, Sunae-Dong, Bundang-Gu, Sungnam City, Kyunggi-Do, 463-825, KOREA Tel: +82-31-719-8115