



The Fastest Verification

ZeBu™ zTIDE Manual

Document revision – b –
August 2005

Version 1.2_7



Table of Contents

ABOUT THIS MANUAL.....	5
OVERVIEW.....	5
INTENDED AUDIENCE.....	5
HISTORY.....	5
MANUAL CONTENT.....	6
RELATED DOCUMENTATION.....	6
TYPOGRAPHIC CONVENTIONS USED IN THIS MANUAL.....	7
1 INTRODUCTION.....	9
1.1 OVERVIEW.....	9
1.2 PURE SIMULATION ON zTIDE.....	9
1.3 MIXED-SIMULATION ON zTIDE AND ZeBU.....	10
2 zTIDE ELEMENTS.....	11
2.1 THE zTIDE WRAPPER.....	11
2.1.1 Description.....	11
2.1.2 Clock Ports.....	11
2.1.3 zTIDE Wrapper Example.....	12
2.2 TRANSACTOR MODULES.....	13
2.3 ZCEI MODULES.....	13
2.4 MIXED LANGUAGE.....	13
3 PROCEEDING WITH zTIDE FOR PURE SIMULATION.....	15
3.1 INTRODUCTION.....	15
3.2 COMPILING FOR PURE SIMULATION.....	15
3.2.1 Using ModelSim.....	15
3.2.2 Using VCS.....	15
3.2.3 Using NC-Verilog.....	16
3.3 SETTING THE DYNAMIC LIBRARY PATH.....	16
3.4 LAUNCHING SIMULATION.....	17
3.4.1 Using ModelSim.....	17
3.4.2 Using VCS.....	17
3.4.3 Using NC-Verilog.....	17



4	PROCEEDING WITH ZTIDE FOR MIXED SIMULATION	18
4.1	INTRODUCTION	18
4.2	COMPILATION FOR MIXED SIMULATION.....	19
4.2.1	<i>Compiling the DUT for ZeBu.....</i>	<i>19</i>
4.2.2	<i>Compilation for the HDL Simulator.....</i>	<i>19</i>
4.3	SETTING THE DYNAMIC LIBRARY PATH.....	20
4.4	LAUNCHING SEQUENCE	21
4.4.1	<i>Using ModelSim</i>	<i>21</i>
4.4.2	<i>Using VCS.....</i>	<i>21</i>
4.4.3	<i>Using NC-Verilog.....</i>	<i>21</i>
5	ENVIRONMENT ADVANCED SETTINGS.....	22
5.1	CONFIGURING ZTIDE LIBRARIES.....	22
5.1.1	<i>Configuration Files</i>	<i>22</i>
5.1.2	<i>Optimizing Transaction Speed</i>	<i>23</i>
5.1.3	<i>Inter Process Communication.....</i>	<i>24</i>
5.1.4	<i>Design Clock Modeling Precision.....</i>	<i>24</i>
5.1.5	<i>Clock Advancement.....</i>	<i>25</i>
5.2	CONFIGURING DUT USING THE DESIGNFEATURES FILE.....	29
5.2.1	<i>Defining the DUT Clocks</i>	<i>29</i>
5.2.2	<i>Defining the Driver Clock Frequency.....</i>	<i>30</i>
5.2.3	<i>Defining a Reset.....</i>	<i>31</i>
6	INSTALLATION.....	32
6.1	INSTALLING ZTIDE SOFTWARE RELEASE.....	32
6.1.1	<i>Software Installation.....</i>	<i>32</i>
6.1.2	<i>Required Environment Variables.....</i>	<i>32</i>
6.2	INSTALLING LICENSE FEATURE	33
7	REFERENCE RESOURCES.....	34
7.1	ZEBU-XL DOCUMENTS.....	34
7.2	ZEBU-ZV DOCUMENTS	35
8	EVE CONTACTS.....	36



Figures

Figure 1: Using zTIDE for Pure Simulation.....	9
Figure 2: Using zTIDE in association with ZeBu.....	10
Figure 3: zTIDE Wrapper Example.....	12
Figure 4: zTIDE Flow for Pure Simulation	15
Figure 5: zTIDE Flow for Mixed Simulation	18
Figure 6: Clock Advancement Mode.....	25
Figure 7: uclock Advancement.....	26
Figure 8: Synchronization Modes	28
Figure 9: Example of DUT reset	31



About This Manual

Overview

This manual describes how to develop and debug synthesizable transactors using zTIDE (ZeBu Transaction Interface Development Environment). zTIDE connects a transaction-level test bench (C/C++/SystemC/SystemVerilog) to an HDL simulator so that you can work on your transactor without requiring a dedicated ZeBu system.

This manual integrates examples using industry-standard simulators: ModelSim®, VCS™ and NC-Verilog®.

This manual deals with the ZeBu Interface. The SCE-MI interface is described in the *[ZeBu SCE-MI Manual](#)*.

ModelSim® is a registered trademark of Mentor Graphics, Inc in the United States of America.

VCS™ is a registered trademark of Synopsys Inc, in the United States of America.

NC-Verilog® is a registered trademark of Cadence Design Systems, Inc. in the United States of America.

Intended Audience

This manual is written for experienced EDA hardware and software engineers to help them use zTIDE to perform HDL co-simulation for transactor testing and debugging. Engineers will typically have experience with the C/C++/SystemC/SystemVerilog languages, Verilog, VHDL, and an industry-standard simulator such as ModelSim, VCS, or NC-Verilog. It is important to have read the *[ZeBu Transaction-Based Verification Manual](#)* to understand the best way to design transactors, and to follow Lab 6 and/or Lab 8 of the *[ZeBu-ZV Tutorial Manual](#)*.

History

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

Doc Revision	Product Version	Date	Evolution
b	1.2_7	Aug 05	Complete revision of the structure of the manual. Addition of features introduced since previous manual edition.
a	1.2_1	Apr 05	First edition.



Manual Content

Chapter 1, “Introduction”, presents the zTIDE environment and describes how to use it in pure simulation or in association with ZeBu system (mixed simulation).

Chapter 2, “zTIDE Elements”, describes the different elements required (zTIDE wrapper, transactors, zcei modules) to proceed with zTIDE for both pure and mixed simulation.

Chapter 3, “Proceeding with zTIDE for Pure Simulation”, describes the HDL compilation process and simulation launching for pure simulation with the 3 industry standards HDL simulators listed before.

Chapter 4, “Proceeding with zTIDE for Mixed Simulation”, describes the compilation process and simulation launching in mixed simulation with the 3 industry standards HDL simulators listed before.

Chapter 5, “Environment Advanced Settings”, describes the various parameters that can be set to optimize configuration for the zTIDE libraries and for the DUT.

Chapter 6, “Installation”, describes how to install the zTIDE environment before proceeding with compilation and simulation. When using zTIDE for the first time, you should read this section first.

Related Documentation

The following documents give relevant information for zTIDE understanding and are available for both ZeBu-ZV and ZeBu-XL:

- The *[ZeBu Transaction-Based verification Manual](#)* contains descriptions how to design and use transactors with ZeBu.
- The *[ZeBu Compilation Manual](#)* describes the complete steps of the compilation process.
- The *[ZeBu Reference Manual](#)* contains descriptions about the various ZeBu tools and their use, and detailed descriptions about the syntax and keywords for the DVE file, drivers and macros.
- The *[ZeBu HDL Co-simulation Manual](#)* describes the use of the HDL co-simulation driver for ZeBu.
- The *[ZeBu C API Reference Manual](#)* and *[ZeBu C++ API Reference Manual](#)* provide detailed information on C/C++ library, files, classes and methods necessary to write a C/C++ test bench to verify your design.

You can also refer to Labs 6 and 8 of the *[ZeBu-ZV Tutorial Manual](#)* for examples of how to perform transaction-based verification.

- Lab 6 explains how to perform transaction-based verification with a C++ testbench.
- Lab 8 explains how to perform transaction-based verification with a SystemC testbench.

The complete documentation packages for ZeBu-ZV and ZeBu-XL are listed at the back of this manual, in Chapter 7.



Typographic Conventions Used in This Manual

ZeBu tools are shown in bold, mono-space “Courier New” font: **zx1Build**, **zx1Par**, **zx1FW**, **zNetgen**, **zRun**...

Code examples, command lines and scripts are shown in mono-space “Courier New” font with a grey shaded background.

For example, the following line 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 or a file name .

`[rw-mode]` Options are presented inside square brackets. The available options will be listed and described in the text that follows the example.

Following example of a shell command includes the shell prompt (which depends on your configuration):

```
$> zx1Build <script_file_name>
```

The command itself may be shown in bold characters for easier reading.

Driver declarations and other declarations/instantiations are shown with the same convention as code examples. The following shows a driver declaration used in the Design Verification Environment (DVE) file:

```
BUS_DRIVER main_bus (  
  .reset      ( global_reset ),  
  .enable     ( 1'b1 ),  
  .load       ( load ),  
  .input1     ( {a[15:0], asic.pll.clockout} ),  
  .input2     ( { sign, c[15:15], b[13:9], 1'b0, b[7:0]} ),  
  .output1    ( data_out[19:7] ),  
  .output2    ( data_out[6:0] )  
);
```



Reports, logs and any output data (except Tcl scripts), generated by ZeBu-XL tools are shown in mono-space “Courier New” font with a border and no grey shaded background. The following is an example of a **zx1Par** log for the place and route step for one FPGA module:

```
#####  
# zx1Par STEP : Place the design  
#####  
# step PLACE : Placement result for module m0 placed on zx1_8c_8000_v1:  
# step PLACE : +-----+-----+-----+-----+  
# step PLACE : | 0 | Z8V1_XC2V8000_T0 | fpga2  
# step PLACE : | 1 | Z8V1_XC2V8000_B0 | fpga1  
# step PLACE : | 2 | Z8V1_XC2V8000_T1 | unused  
# step PLACE : | 3 | Z8V1_XC2V8000_B1 | fpga3  
# step PLACE : | 4 | Z8V1_XC2V8000_T2 | fpga4  
# step PLACE : | 5 | Z8V1_XC2V8000_B2 | fpga6  
# step PLACE : | 6 | Z8V1_XC2V8000_T3 | fpga0  
# step PLACE : | 7 | Z8V1_XC2V8000_B3 | fpga5  
# step PLACE : | 21 | hub0 | -  
# step PLACE : | 22 | hub1 | -  
# step PLACE : | 23 | hub2 | -  
# step PLACE : | 24 | hub3 | -  
# step PLACE : | 25 | hub4 | -  
# step PLACE : +-----+-----+-----+-----+
```

GUI elements (menu items, buttons, check box, edition field) 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 Overview

zTIDE, standing for **Z**eBu **T**ransaction **I**nterface **D**evelopment **E**nvironment, is a development and debugging environment for ZeBu synthesizable transactors based on HDL simulation.

With ZeBu, the DUT and the transactor hardware parts run on the ZeBu system. zTIDE speeds up design of synthesizable transactors since the DUT and the transactor hardware parts run on the HDL simulator (pure simulation). It is also possible to run the DUT on Zebu and keep the transactor hardware part in the HDL simulator (mixed simulation).

The transaction communication infrastructure is similar in ZeBu and zTIDE environment. Your transactors are fully compatible and they run in both environments without any modification.

1.2 Pure Simulation on zTIDE

Using zTIDE for pure-simulation consists in using the HDL simulator for both DUT and hardware part of the transactors.

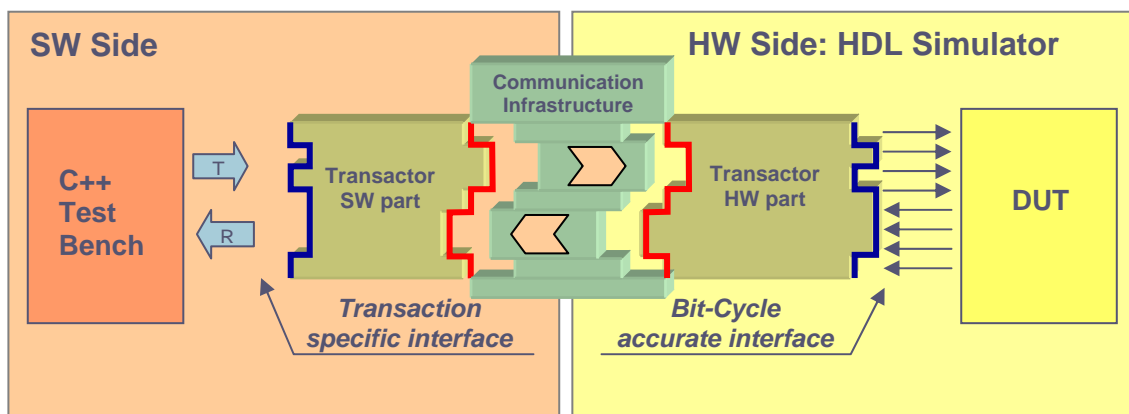


Figure 1: Using zTIDE for Pure Simulation

The communication infrastructure between the transactor software part and the simulator is managed by the following zTIDE libraries:

- libZebu on the software side
- libZebuSim on the simulator side

1.3 Mixed-Simulation on zTIDE and ZeBu

If your DUT is too big, it may strongly impact the simulation performance and thus the test sequence for transactor validation will be limited.

In order to improve the test sequence coverage, you can speed up the DUT by mapping it on ZeBu while running the transactor hardware part on the HDL simulator.

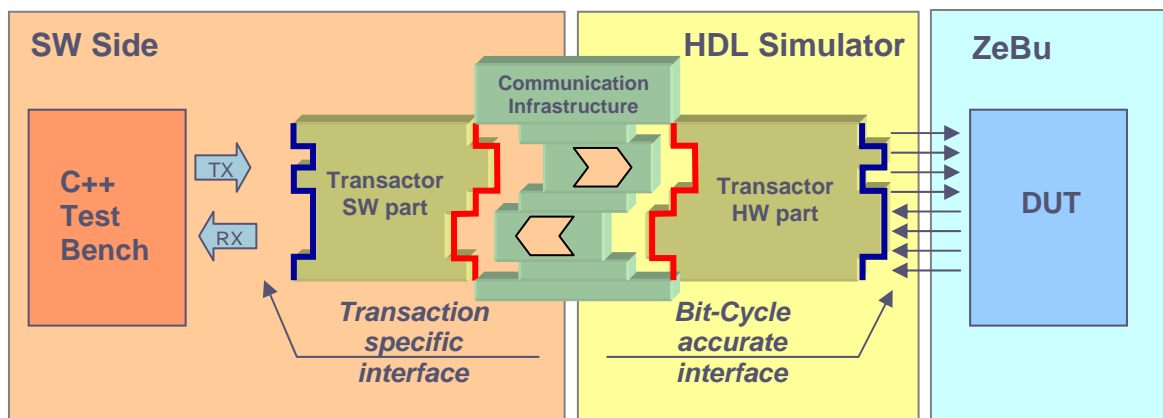


Figure 2: Using zTIDE in association with ZeBu

The communication infrastructure between the transactor software part and the simulator is the same as for zTIDE pure simulation. A communication infrastructure is set between the HDL simulator and ZeBu, in the same way as for HDL co-simulation with ZeBu, using the following Zebu libraries:

- `libZebu` on Zebu side
- `libZebuSim` on the simulator side

In this configuration you will first compile your DUT for ZeBu as if you were doing HDL co-simulation and then use the generated wrapper during the HDL compilation process for zTIDE.

Note:

It is recommended to proceed with ZeBu HDL Co-simulation of your DUT before starting integration with the transactors.



2 zTIDE Elements

2.1 The zTIDE Wrapper

2.1.1 Description

The zTIDE wrapper describes the overall environment of the simulation, instantiating all the components:

- DUT top module
- Transactors (including their connection to the DUT)
- Clock port modules `zceiClockPort` (one for each DUT clock controlled by transactor).

The zTIDE wrapper is the simulation top-level, written in Verilog language.

The zTIDE wrapper file contains the same type of information as the ZeBu Design Verification Environment (DVE) file but with a slightly different syntax.

2.1.2 Clock Ports

The `zceiClockPort` clock module is used to connect a clock signal (`cclock`) and a reset signal (`crestet` or `crestetn`) to the DUT instance.

The numbering of design clocks connected to the DUT and controlled by transactors is defined by the `clockNum` parameter in the `zceiClockPort` and `zceiClockControl` clock modules.

See the description files for the Verilog clock modules available in the zTIDE package:

```
$ZTIDE_ROOT/verilog/zceiClockPort.v
```

```
$ZTIDE_ROOT/verilog/zceiClockControl.v
```

2.1.3 zTIDE Wrapper Example

Figure 3 presents an example of a zTIDE wrapper including a DUT connected to 2 transactors:

- reader driver: reads a ROM inside the DUT
- ROM driver: software emulation of the ROM in the DUT

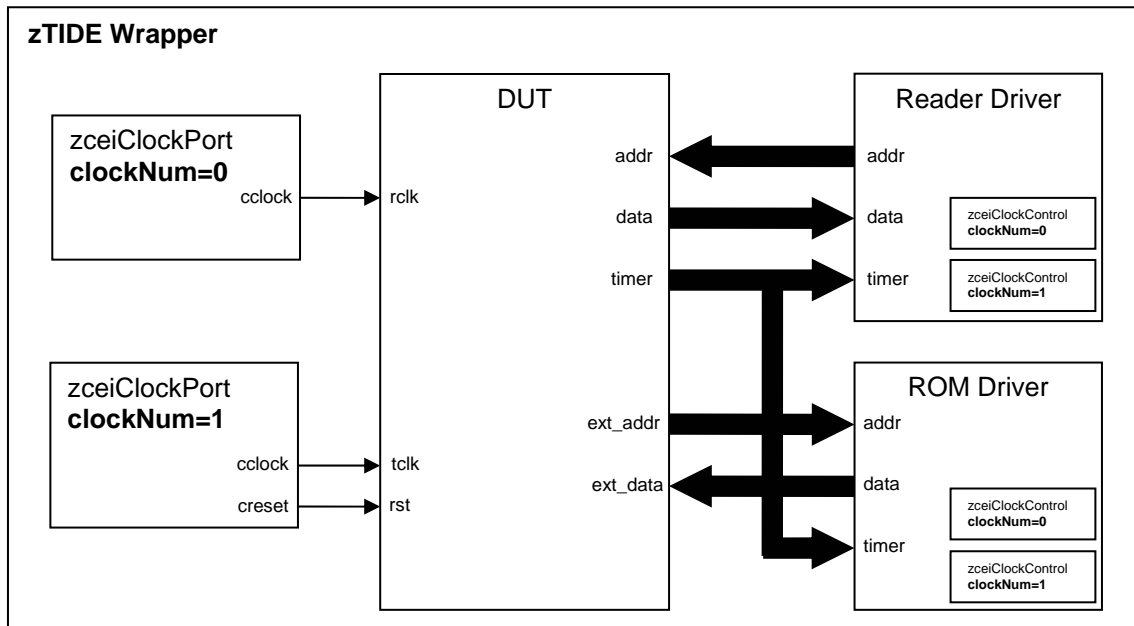


Figure 3: zTIDE Wrapper Example

DUT uses 2 clocks:

- `rclk`: ROM reading clock
- `tclk`: internal timer clock

These clocks are connected to the DUT with 2 `zceiClockPort` module instances in the zTIDE wrapper, and controlled by both transactors with 2 `zceiClockControl` module instances in the hardware part of transactors.

The reader driver enables `rclk` posedge to start memory reading while the ROM driver enables `rclk` negedge when memory data is ready. Both drivers enable `tclk` when processing a transaction.

This example is part of the zTIDE package:

```
$ZTIDE_ROOT/examples/ztide_pure/parallelRomReader/
```



2.2 Transactor Modules

The transactor modules are described according to Chapter “Designing a Transactor” of the *ZeBu Transaction Based Verification Manual*.

A basic transactor example is part of the zTIDE package:

```
$ZTIDE_ROOT/examples/ztide_pure/simpleCounterDriver/driver/simple_driver/v_src/simple_driver.v
```

2.3 zcei Modules

The ZeBu co-emulation interface modules that are instantiated by the transactor modules and the zTIDE wrapper must also be compiled for HDL simulation.

The corresponding Verilog source files are part of the zTIDE package:

```
$ZTIDE_ROOT/Verilog/zceiClockControl.v  
$ZTIDE_ROOT/Verilog/zceiClockPort.v  
$ZTIDE_ROOT/Verilog/zceiMessageInPort.v  
$ZTIDE_ROOT/Verilog/zceiMessageOutPort.v
```

These Verilog files are associated with the `libZebuSim.so` dynamic library which must be loaded by the HDL simulator. Commands to load this library in industry standard simulators are given in the following sections.

2.4 Mixed language

The zTIDE environment does not provide a dedicated API for VHDL. However, the Verilog PLI `libZebuSim.so` can be used in mixed language.

`zcei` modules can be instanced in VHDL and they are declared in your VHDL sources as in the following example:

```
component zceiClockPort  
  generic (  
    clockNum : integer;  
    cclockName : string  
  );  
  port (  
    cclock : out std_logic;  
    creset : out std_logic;  
    cresetn : out std_logic  
  );  
end component;
```

...



```
component zceiClockControl
  generic (
    clockNum : integer
  );
  port (
    uclock           : out std_logic;
    ureset           : out std_logic;
    readyForCclock  : in  std_logic;
    cclockEnabled   : out std_logic;
    readyForCclockNegEdge : in  std_logic;
    cclockNegEdgeEnabled : out std_logic
  );
end component;

component zceiMessageInPort
  generic (
    Pwidth : integer
  );
  port (
    transmitReady : out std_logic;
    receiveReady  : in  std_logic;
    message       : out std_logic_vector (Pwidth - 1 downto 0)
  );
end component;

component zceiMessageOutPort
  generic (
    Pwidth : integer
  );
  port (
    transmitReady : in  std_logic;
    receiveReady  : out std_logic;
    message       : in  std_logic_vector (Pwidth - 1 downto 0)
  );
end component;
```

A mixed language example is part of the zTIDE package:

```
$ZTIDE_ROOT/examples/ztide_pure/simplePipe_VHDL
```

Following is an example of compilation command line for ModelSim:

```
$> vlib work
$> vlog $ZTIDE_ROOT/verilog/zcei*.v
$> vcom <options> Your_zTIDE_Wrapper.v <list of transactors> dut.vhdl
```

3 Proceeding with zTIDE for Pure Simulation

3.1 Introduction

Figure 4 shows the list of Verilog files to compile for the HDL simulator in pure simulation.

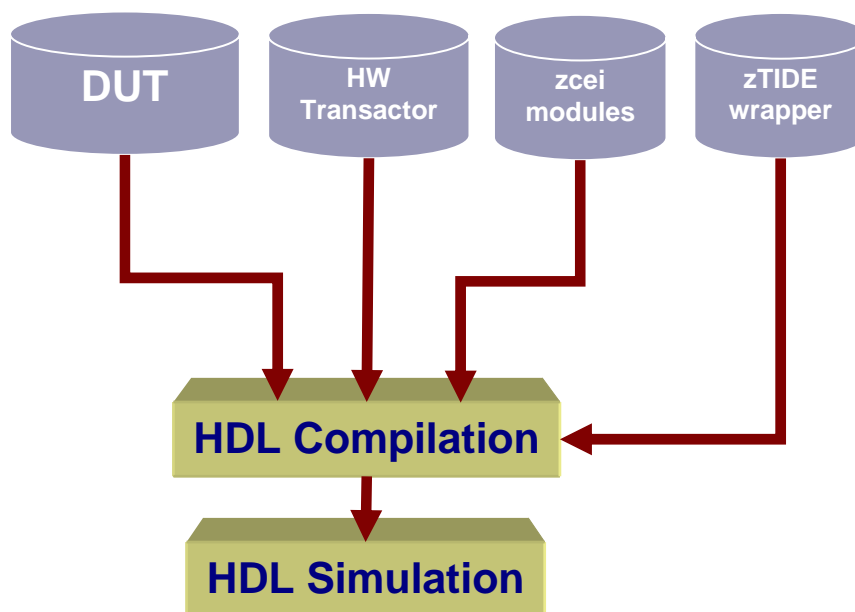


Figure 4: zTIDE Flow for Pure Simulation

3.2 Compiling for Pure Simulation

3.2.1 Using ModelSim

Use the following commands for compilation:

```
$> vlib work
$> vlog <options> <Your_zTIDE_Wrapper.v> <list of transactors>
$ZTIDE_ROOT/verilog/zcei*.v dut.v
```

3.2.2 Using VCS

Use the following command for compilation:

```
$> vcs <options> <Your_zTIDE_Wrapper.v> <list of transactors>
$ZTIDE_ROOT/verilog/zcei*.v dut.v -P $ZTIDE_ROOT/lib/pli.tab
$ZTIDE_ROOT/lib/libZebuSim.so
```



3.2.3 Using NC-Verilog

With NC-Verilog, a single command is used for compilation and simulation launching:

```
$> ncverilog <options> +access+rwc +nbasync  
+loadplil=libZebuSim.so:bootstrap <Your_zTIDE_Wrapper.v> <list of  
transactors> $ZTIDE_ROOT/verilog/zcei*.v dut.v
```

However, you can run only the compilation:

```
$> ncvlog <options> <Your_zTIDE_Wrapper.v> <list of transactors>  
$ZTIDE_ROOT/verilog/zcei*.v dut.v  
$> ncelab -loadplil $ZTIDE_ROOT/lib/libZebuSim.so:bootstrap -access  
+rwc <Your_zTIDE_Wrapper_Module_Name>
```

3.3 Setting the dynamic library path

The zTIDE environment libraries are in \$ZTIDE_ROOT/lib.

The environment variable ZTIDE_ROOT must point to the installation directory of the zTIDE environment.

To run a pure simulation in the zTIDE environment, modify the LD_LIBRARY_PATH variable as follows:

```
export LD_LIBRARY_PATH=$ZTIDE_ROOT/lib:$LD_LIBRARY_PATH
```

See Chapter 6 of this document for details.



3.4 Launching Simulation

The HDL simulator should be started first and then the C++ test bench, allowing the simulation side to reset the inter process communication environment before establishing the connection between both processes.

Following are examples of HDL simulator launching commands for industry standard products (corresponding to the compilation commands described in Section 3.2).

Some simulation parameters for zTIDE libraries and for the DUT can be set with the appropriate files, as described in Chapter 5. However, default configuration files are automatically generated by zTIDE tools for easier use of this environment.

3.4.1 Using ModelSim

Once you have compiled your design with transactors and wrapper, launch the HDL simulation:

```
$> vsim -c <your_top_module_name> -do "run -all; exit" -pli  
libZebuSim.so
```

3.4.2 Using VCS

Once you have compiled your design with transactors and wrapper, launch the HDL simulation:

```
$> ./simv
```

3.4.3 Using NC-Verilog

With NC-Verilog, a single command is used for compilation and simulation launching:

```
$> ncverilog <options> +access+rwc +nbasync  
+loadplil=libZebuSim.so:bootstrap <Your_zTIDE_Wrapper.v> <list of  
transactors> $ZTIDE_ROOT/verilog/zcei*.v dut.v
```

However, you can run only the simulation:

```
$> ncsim <Your_zTIDE_Wrapper_Module_Name>
```

4 Proceeding with zTIDE for Mixed Simulation

This chapter describes the process for zTIDE in mixed simulation environment.

4.1 Introduction

Figure 4 shows the list of Verilog files to compile with the HDL simulator in mixed simulation.

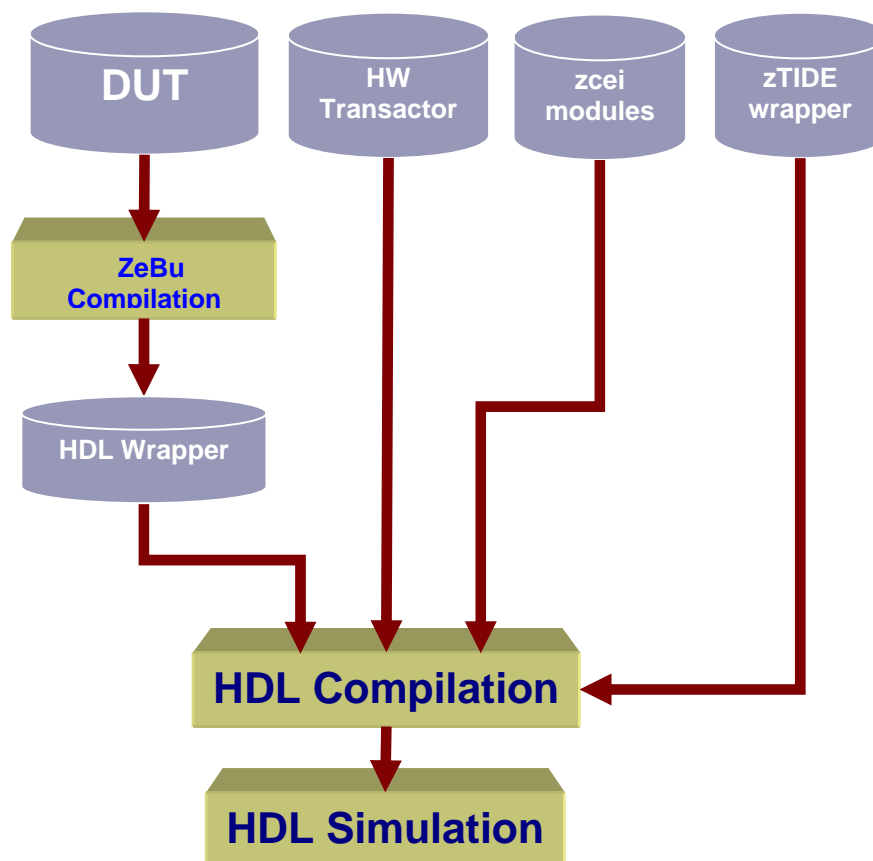


Figure 5: zTIDE Flow for Mixed Simulation

You first compile DUT for ZeBu as if proceeding with HDL co-simulation and then you compile the wrapper generated by ZeBu compilation flow with the HDL simulator.



4.2 Compilation for Mixed Simulation

4.2.1 Compiling the DUT for ZeBu

This process is similar to DUT compilation for HDL verification with ZeBu. It generates a Verilog wrapper file that replaces the DUT modules in the HDL simulator. Refer to the *ZeBu HDL Co-Simulation Manual* for details.

The file is generated in the `zebu.work` directory and must not be confused with the zTIDE wrapper that you create.

4.2.2 Compilation for the HDL Simulator

The HDL simulator links the zTIDE environment and the ZeBu environment. The HDL simulator must load the zTIDE environment dynamic library `libZebuSim.so` and the ZeBu environment library `libZebuPLI.so`. Using two different installation directories is quite confusing, thus care must be taken with access paths `$ZEBU_ROOT` and `$ZTIDE_ROOT` when entering the command lines.

A mistake in the command lines may not cause in compilation errors, but could result in erroneous runtime.

4.2.2.1 Using ModelSim

Compilation commands are:

```
$> vlib work
$> vlog <options> <Your_zTIDE_Wrapper.v> <list of transactors>
$ZTIDE_ROOT/verilog/*.v $ZEBU_WORK_DIR/Top_dut.v
```

4.2.2.2 Using VCS

Compilation commands are:

```
$> vcs <options> <Your_zTIDE_Wrapper.v> <list of transactors>
$ZTIDE_ROOT/verilog/*.v
$ZEBU_WORK_DIR/Top_dut.v -P
$(ZTIDE_ROOT)/lib/pli.tab.mixedWithZebuPLI
$(ZTIDE_ROOT)/lib/libZebuSim.so $(ZEBU_ROOT)/lib/libZebuPLI.so
```

Note that the `pli.tab.mixedWithZebuPLI` file, included in the zTIDE package, is built from the `pli.tab` files of the `$ZEBU_ROOT/lib/libZebuPLI.so` and `$ZTIDE_ROOT/lib/libZebuSim.so` libraries.



4.2.2.3 Using NC-Verilog

With NC-Verilog, a single command is used for compilation and simulation launching:

```
$> ncverilog <options> + access+rcw +nbasync  
+loadpli1=$ZTIDE_ROOT/lib/libZebuSim.so:bootstrap  
+loadpli1=$ZEBU_ROOT/lib/libZebuPLI.so:bootstrap  
Your_zTIDE_Wrapper.v <list of transactors>  
$ZTIDE_ROOT/verilog/*.v $ZEBU_WORK_DIR/Top_dut.v
```

However, you can run only the compilation:

```
$> ncvlog <options> <Your_zTIDE_Wrapper.v> <list of transactors>  
$ZTIDE_ROOT/verilog/*.v $ZEBU_WORK_DIR/Top_dut.v  
$> ncelab -loadpli1 $ZTIDE_ROOT/lib/libZebuSim.so:bootstrap  
+loadpli1=$ZEBU_ROOT/lib/libZebuPLI.so:bootstrap -access +rcw  
<Your_zTIDE_Wrapper_Module_Name>
```

4.3 Setting the dynamic library path

The standard ZeBu environment libraries and zTIDE environment libraries have similar names and interfaces, but they are located in different directories:

- The ZeBu environment libraries are in \$ZEBU_ROOT/lib.
- The zTIDE environment libraries are in \$ZTIDE_ROOT/lib.

The environment variable ZTIDE_ROOT must point to the zTIDE installation directory.

To run a mixed-simulation with zTIDE, modify the LD_LIBRARY_PATH variable as follows (possible syntax differences according to the shell you are using):

- From where you launch the HDL simulator:

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

- From where you launch the test bench:

```
export LD_LIBRARY_PATH=$ZTIDE_ROOT/lib:$LD_LIBRARY_PATH
```

See Chapter 6 of this document and the appropriate *ZeBu Installation Manual* for details.



4.4 Launching Sequence

The HDL simulator should be started first and then the C++ test bench, allowing the simulation side to reset the inter process communication environment before establishing the connection between both processes.

Following are examples of HDL simulator launching commands for industry standard products (corresponding to the compilation commands described in Section 3.2).

Some simulation parameters for zTIDE libraries and for the DUT can be set with the appropriate files, as described in Chapter 5. However, default configuration files are automatically generated by zTIDE tools for easier use of this environment.

4.4.1 Using ModelSim

Once you have compiled your design with transactors and wrapper, launch the HDL simulation:

```
$> vsim -c <Your_top_module_name> +zebu.work=$ZEBU_WORK_DIR -do  
"run -all; exit" -pli "$ZTIDE_ROOT/lib/libZebuSim.so  
$ZEBU_ROOT/lib/libZebuPLI.so" +zebu.work=$ZEBU_WORK_DIR
```

4.4.2 Using VCS

Once you have compiled your design with transactors and wrapper, launch the HDL simulation:

```
$> ./simv +zebu.work=$ZEBU_WORK_DIR
```

4.4.3 Using NC-Verilog

With NC-Verilog, a single command is used for compilation and simulation launching:

```
$> ncverilog <options> + access+rwc +nbasync  
+loadplil=$ZTIDE_ROOT/lib/libZebuSim.so:bootstrap  
+loadplil=$ZEBU_ROOT/lib/libZebuPLI.so:bootstrap  
Your_zTIDE_Wrapper.v <list of transactors>  
$ZTIDE_ROOT/verilog/*.v $ZEBU_WORK_DIR/Top_dut.v
```

However, you can run only the simulation:

```
$> ncsim <Your_zTIDE_Wrapper_Module_Name>
```



5 Environment Advanced Settings

5.1 Configuring zTIDE libraries

5.1.1 Configuration Files

zTIDE libraries are configured independently:

- On the software side, the `libZebu.so` dynamic library is configured by the `libZebu.cfg` file in the C++ testbench run directory.
- On the HDL simulator side, the `libZebuSim.so` dynamic library is configured by the `libZebuSim.cfg` file in the simulation run directory.

These configuration files are optional. If no file is found in a run directory, the appropriate default files are created.

Configuration files contain description of the following parameters:

libZebu (C++)	libZebuSim (HDL)	Keyword	Type/ Value Range	Default	Description
Simulation Event					
	X	ON_CLOSE	STOP, EXIT	EXIT	Action on close
	X	ON_WARNING	STOP, EXIT	no action	Action on warning
	X	ON_ERROR	STOP, EXIT	no action	Action on error
Message Displaying					
X	X	DISPLAY_INFO	ON, OFF	ON	Enable information display
X	X	DISPLAY_WARNING	ON, OFF	ON	Enable warning display
X	X	DISPLAY_ERROR	ON, OFF	ON	Enable error display
X	X	DISPLAY_DEBUG	ON, OFF	ON	Enable debug display
Inter-Process Communication					
X		IP_ADD	String **	"127.0.0.1"	Address of the host on which the HDL simulator runs
X		IPC_MODE	String **	"ANY"	ANY: to allow usage of a local pipe. TCP: for TCP only.
X	X	PORT	32-bit number	9000	Socket port number to manage socket port sharing
X	X	TIMEOUT	32-bit number	0	Timeout used in inter process communication
X		RUN_LENGTH *	32-bit number	10	Set HW transaction speed to obtain an environment similar to the HW transactor in ZeBu
X		RUN_VAR	32-bit number	0	Not used



libzebu (C++)	libzebuSim (HDL)	Keyword	Type/ Value Range	Default	Description
Simulation Configuration					
	X	FREQUENCY_PRECISION_NUMERATOR	32-bit number	1	Design Clock modeling precision
	X	FREQUENCY_PRECISION_DENOMINATOR	32-bit number	1	Design Clock modeling precision
	X	UCLOCK_ADVANCEMENT_MODE	String **	"DESIGN_CLOCK_BASED"	uclock Advancement Mode
	X	ALL_MESSAGE_IMPORTS_ARE_SYNCHRONOUS	String **	"no"	Message Port synchronization
	X	ALL_MESSAGE_EXPORTS_ARE_SYNCHRONOUS	String **	"no"	Message Port synchronization
	X	MESSAGE_PORT_SYNCHRONISATION_MODE	String **	"UCLOCK_BASED"	Message Port synchronization
	X	MESSAGE_PORT_SYNCHRONISATION_TIMEOUT	32-bit number	10	Message port synchronization
	X	MESSAGE_IMPORT_ACTIVE_VALUE	String **	"X"	Set the value to "0" or "X" to apply on the message vector of imports if no more messages have arrived from the SW side.

* In the pre-release version of zTIDE (prior to V 1.0), this keyword was spelled RUN_LENGTH (the G and H were transposed).

** String values must be written between double quotes ("string").

5.1.2 Optimizing Transaction Speed

You can optimize your zTIDE environment to be as close as possible of the ZeBu environment in terms of clock speed. The RUN_LENGTH parameter sets the number of driver clock (uclock) cycles performed between two software transactions (port connection, read or write operations on the ports).

This adjustment allows you to:

- Simulate the progress of the driver clock as if the transactors and DUT were emulated by ZeBu.
- Simulate the flow of messages between the software and the hardware in a similar manner as with ZeBu.
- Be sure that the progression of the test bench and the progression of the HDL simulation are linked, deterministic and that the message flow is reproducible.



To adjust RUN_LENGTH and optimize the execution time you should refer to following indicators displayed when closing the HDL simulator:

- Number of software requests: number of requests for message port connection, message reception, message sending, polling for message reception and message sending ... during the simulation.
- Cycle stamp: number of uclock cycles during which at least one design clock edge has been generated.
- Elapsed time for the simulation: execution time without initialization phase in the HDL simulator.

The ratio between the final cycle stamp and the execution time can be considered as the optimized co-simulation frequency.

5.1.3 Inter Process Communication

zTIDE is a multi process environment requiring a communication channel between the C/C++ test bench and the HDL simulator. This communication is based on a Linux socket using TCP protocol.

The Inter process communication parameters are defined in the `libZebu.cfg` file.

Default values for these parameters are optimized for local installation on a single PC: `IP_ADD="127.0.0.1"` is a local host address and `IPC_MODE="ANY"` uses a faster communication mode than TCP protocol.

If `IPC_MODE="ANY"` and the IP address is not a local one, the TCP protocol will be used.

If you want to use TCP protocol, whatever the local or remote configuration you use, set `IPC_MODE="TCP"` and the appropriate IP address in `IP_ADD`.

5.1.4 Design Clock Modeling Precision

`FREQUENCY_PRECISION_NUMERATOR` and `FREQUENCY_PRECISION_DENOMINATOR` parameters specify ratio of the maximum tolerance for the precision of the virtual frequency of controlled design clocks linked in the same group.

$$\frac{\text{RoundedFrequency}}{\text{SpecifiedFrequency}} \leq \frac{\text{FREQUENCY_PRECISION_NUMERATOR}}{\text{FREQUENCY_PRECISION_DENOMINATOR}}$$

The default ratio allows rounding the frequency of each clock according to 1/128.

This tolerance allows simplification of group clock modeling and simulation acceleration when clock group specification contains many different frequencies with a large spectrum. The greater the ratio, the faster the simulation is.

If you want to forbid rounding of frequencies you must set the `FREQUENCY_PRECISION_NUMERATOR` parameter to 0 in the `libZebuSim.cfg` configuration file.

Example:

If a design contains a group of 3 clocks for which the specified frequencies are 111.111 kHz, 11.111 kHz and 1 kHz, the default clock precision ratio will have the following impact on the actual rounded frequency:

Specified frequency	Rounded Frequency
111.111 kHz	110.721 kHz
11.111 kHz	11.110 kHz
1 kHz	1 kHz

5.1.5 Clock Advancement

The time advancement in a zTIDE simulation is based on the uncontrolled clock (`uclock`) supplied by the Clock Control module. Moreover, the controlled clocks (`cclocks`) supplied by the Clock Port module to the DUT might need to be disabled while transactors are proceeding with following operations:

- Converting received message(s) from software side to stimulate the DUT
- Formatting DUT data into message(s) to send to software side
- Waiting for message reception from the software side
- Waiting to send a message to the software side

Consequently many `uclock` cycles are often generated while controlled design clocks are stopped, as shown on Figure 6.

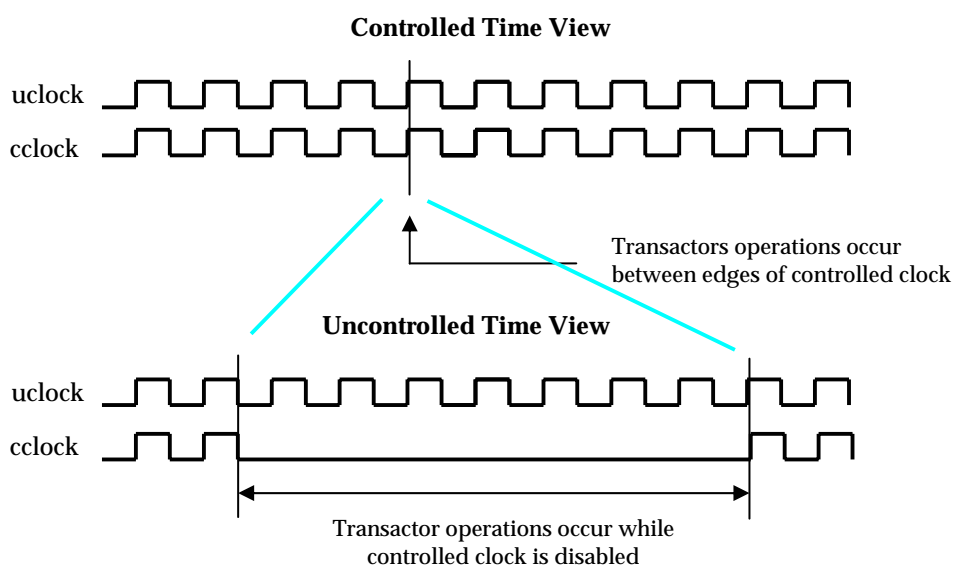


Figure 6: Clock Advancement Mode

With zTIDE you can obtain waveform like the Uncontrolled Time View that is interesting to debug transactors. Yet, if you want have a higher level view you can also use some zTIDE features to obtain waveform close to a Controlled Clock View. These features are:

- `uclock` advancement mode, can be either:
 - a regular frequency (as the Uncontrolled Time View in Figure 6)
 - a controlled design clock-based frequency as waveform (as the Controlled Time View in Figure 6)
- Message Port synchronization: possibility to block the `uclock` advancement while the hardware part of a transactor is waiting to send or receive a message.

5.1.5.1 `UCLOCK_ADVANCEMENT_MODE` parameter

You can select one the two following modes:

- `DESIGN_CLOCK_BASED`: `uclock` advances with the **driver clock** frequency if at least one controlled design clock is enabled. Else, it advances with the fastest possible frequency in order that `uclock` cycles are not possible to observe while all controlled design clocks are disabled.
- `REGULAR_FREQUENCY`: `uclock` advances with a constant frequency independently on the state of controlled design clocks.

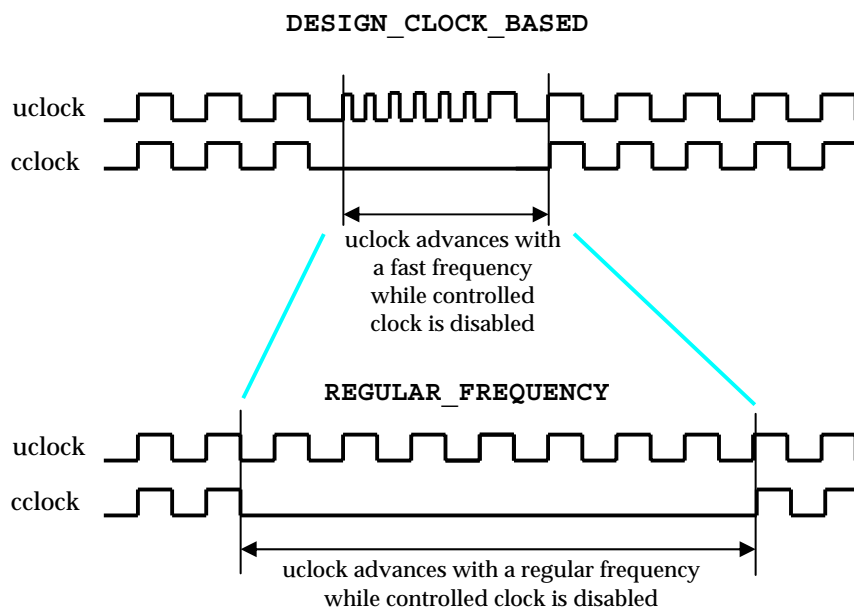


Figure 7: uclock Advancement

Figure 7 shows zTIDE simulations waveforms where clocks are impacted by various parameters such as synchronization between hardware and software parts of transactors, order of software requests and `RUN_LENGTH` parameter value.



5.1.5.2 Message Port Synchronization

The message port synchronization consists in blocking the advancement of `uclock` while the hardware part of a transactor is waiting to receive or send a message. It stops `uclock` cycles while the hardware part of a transactor is waiting.

Verilog Synchronization Parameter

The Verilog synchronous parameter of `zceiMessageInPort` and `zceiMessageOutPort` modules enables or disables the synchronization of a message port. If `synchronous` is assigned to "yes" (default value) the synchronization is enabled; in contrast if it is assigned to "no" the synchronization is disabled.

Synchronization Activation parameters

The following zTIDE configuration parameters allow you to enable or disable the synchronization of all message ports:

- `ALL_MESSAGE_INPORT_ARE_SYNCHRONOUS`: if this parameter is assigned to "yes" the synchronization of all message inports is enabled; in contrast if this parameter is assigned to "no" the synchronization of all message inports is disabled, and the `synchronous` parameter of instances of the module `zceiMessageInPort` has no effect.
- `ALL_MESSAGE_OUTPORT_ARE_SYNCHRONOUS`: if this parameter is assigned to "yes" the synchronization of all message outports is enabled; in contrast if this parameter is assigned to "no" the synchronization of all message outports is disabled, and the parameter `synchronous` of instances of the module `zceiMessageOutPort` has no effect.

By default, synchronization is disabled for all message inports and outports.

Setting a Timeout

A synchronization based on a never-ending blocking might be problematic. A part of the hardware transactor might be expecting reception or transmission of a message while the software side of the transactor is not ready to do so because that depends on some future hardware interaction that will occur if `uclock` advances. Such issue is resolved by a timeout that interrupts the synchronization and resumes the `uclock` advancement.

The parameter `MESSAGE_PORT_SYNCHRONIZATION_TIMEOUT` allows you to set the timeout in number of software requests for connection, reading, and writing operations on the transactor ports. You can also specify a negative value in order to disable the timeout.

By default the timeout is set to 10 software requests.

Note: a timeout signal is available among internal signals of the Verilog Message Port modules in order to make every expired timeout visible in waveform of zTIDE

simulation. A pulse is generated on this signal each time the synchronization timeout expired.

Synchronization Mode

The MESSAGE_PORT_SYNCHRONIZATION_MODE parameter allows you to select the mode of synchronization of message ports. The available modes are the followings:

- “NO_DELAY”: when uclock is blocked, the message to send or receive is transferred immediately from/to hardware at the end of the synchronization.
- “UCLOCK_BASED”: when uclock is blocked, the message to send or receive is transferred from/to hardware on the next uclock posedge after the end of the synchronization.

The following table illustrates the different modes of synchronization, according to MESSAGE_PORT_SYNCHRONIZATION_MODE and synchronization activation parameters.

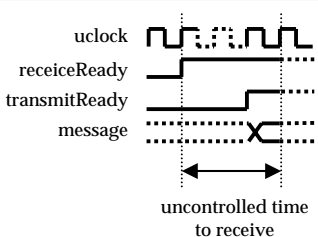
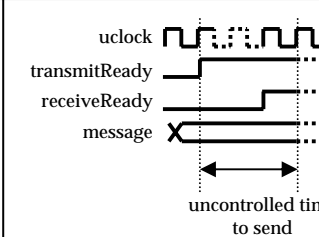
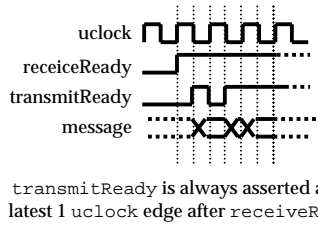
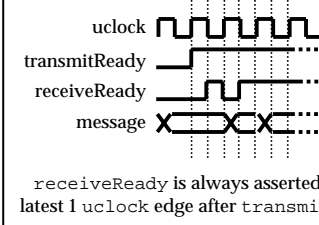
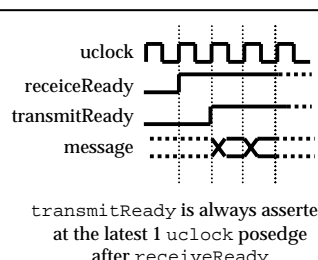
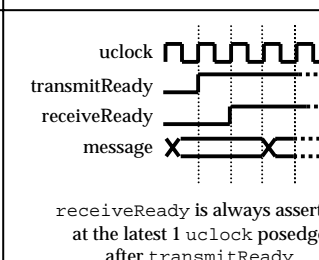
	MessageInPort	MessageOutPort
NO SYNCHRO ACTIVATED	 <p>uclock receiveReady transmitReady message</p> <p>uncontrolled time to receive</p>	 <p>uclock transmitReady receiveReady message</p> <p>uncontrolled time to send</p>
NO DELAY SYNCHRO	 <p>uclock receiveReady transmitReady message</p> <p>transmitReady is always asserted at the latest 1 uclock edge after receiveReady</p>	 <p>uclock transmitReady receiveReady message</p> <p>receiveReady is always asserted at the latest 1 uclock edge after transmitReady</p>
UCLOCK_BASED SYNCHRO	 <p>uclock receiveReady transmitReady message</p> <p>transmitReady is always asserted at the latest 1 uclock posedge after receiveReady</p>	 <p>uclock transmitReady receiveReady message</p> <p>receiveReady is always asserted at the latest 1 uclock posedge after transmitReady</p>

Figure 8: Synchronization Modes

By default the synchronization mode is “UCLOCK_BASED” if the synchronization has been enabled by means of the synchronization activation parameters presented earlier in this section.



5.2 Configuring DUT using the designFeatures file

The `designFeatures` file contains the definition of simulation elements:

- DUT clocks, either directly, or by referring to clock files.
- Driver Clock frequency.
- Reset signal.

The zTIDE environment has default values for these parameters, which are overwritten by the values defined in the `designFeatures` file. This file is declared in the software test bench, either explicitly or implicitly:

- Explicitly: usually through the `designFile` parameter of the `Board::open` method (C++ testbench) or the `Board_open` function (C testbench) of the `libZebu` API. See section “Base class Reference” of the [ZeBu C++ API Reference Manual](#) or [ZeBu C API Reference Manual](#).
- Implicitly: The tool will search for this file in the current path, then in the `zebu.work` directory. If not found in these two directories, no `designFeatures` file is loaded and all default values are used.

5.2.1 Defining the DUT Clocks

zTIDE clock definition is fully compatible with ZeBu clock definition described in the [Zebu Reference Manual](#), with the exception of in-situ clocks which are not supported in zTIDE.

The format for clock definition is the same in specific clock files or in the `designFeatures` file. The clock name used in the definition is the one declared using `zceiClockPort` module in the zTIDE wrapper and you can not define clocks that would not be declared in the zTIDE wrapper.

When using one or several clock files, the `designFeatures` file will contain the definition of these files as in the following example:

```
$B0.userClk_0.file = "clockDefinition.eve";  
$B0.userClk_1.file = "clockDefinition.eve";  
$B0.userClk_2.file = "./clock_dir/otherDefinition.eve";
```

Where the reference directory is the current one.

zTIDE supports controlled clocks and free-running clocks, but in-situ clocks are not supported.



5.2.1.1 Defining a controlled clock

The following parameters are used when defining a controlled clock:

- Mode: optional parameter since controlled clock is the default mode.
- Waveform: clock waveform (relative to other clocks of the same group)
- Group Name (optional): link definition between different clocks; if there is only one clock or if no clock relationship required, this parameter is optional.
- Virtual Frequency: ratio between clocks in the same group.

Example:

```
$B0.clk.mode="controlled";  
$B0.clk.GroupName = "grp_1";  
$B0.clk.virtualFrequency=5000;  
$B0.clk.waveform="_-";  
$B0.clk_half.mode="controlled";  
$B0.clk_half.GroupName = "grp_1";  
$b0.clk_half.virtualFrequency=2500;  
$B0.clk_half.waveform="__-";
```

clk_half is in the same group as clk and virtual frequency parameter is set to have a half frequency, and clocks have different waveforms.

5.2.1.2 Defining a free-running clock

The following parameters are mandatory when defining a free-running clock:

- Mode: “free-running”
- Frequency: absolute frequency of the clock in kHz.

Example:

```
$B0.userClk.mode = "free-running";  
$B0.userClk.Frequency = 1000;           #1 MHz clock
```

5.2.2 **Defining the Driver Clock Frequency**

The driver main clock frequency (in kHz) is defined in the designFeatures file as in the following example:

```
$driverClk.Frequency = 3000;
```

5.2.3 Defining a Reset

A reset is defined using `tgClk`. It is attached to a DUT clock, which must be declared in the zTIDE wrapper.

The following parameters are defined:

- `resetInactive`: number of the driverClock or uclock cycles between the start of the simulation and the reset posedge
- `resetActive`: number of selected design clock cycles between the reset posedge and the reset negedge. If this value is zero, the reset is not asserted.
- `selectResetClkName`: select the design clock used to generate the reset negedge. The clock name used may be the `cclockName` Verilog parameter of the `zceiClockPort`.

Example:

The following example is available in the zTIDE package:

```
$ZTIDE_ROOT/examples/ztide_with_zebu/simpleCounterDriver/bench/simu/run/designFeatures
```

```
$tgClk.resetInactive = 5;
$tgClk.resetActive = 5;
$tgClk.selectResetClkName = "cclock";
```

Figure 9 shows clocks and resets waveform corresponding to this example:

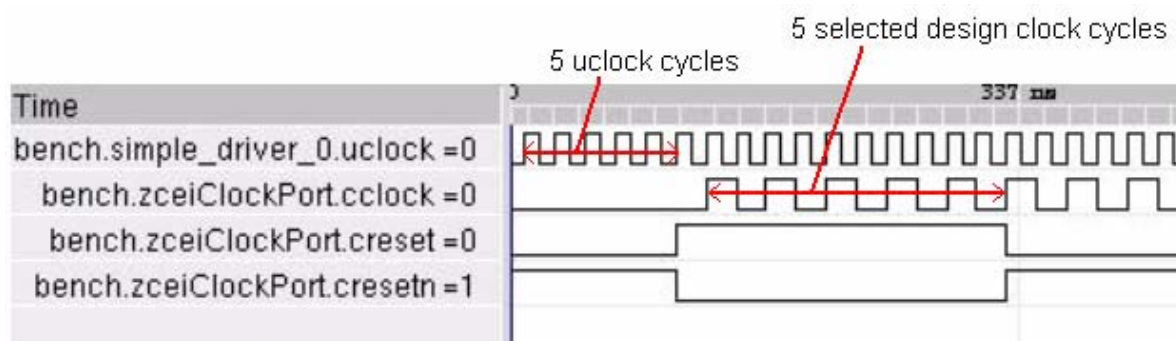


Figure 9: Example of DUT reset



6 Installation

6.1 Installing zTIDE Software Release

The zTIDE software is delivered on a CD-ROM. You must be logged as “root” to install the software. Insert the zTIDE software CD-ROM in the CD device and mount it.

Note: some syntax differences can exist between this document and your environment, according to the shell you are using.

6.1.1 Software Installation

```
$ mount /mnt/cdrom
$ mkdir ~/tmp
$ cd ~/tmp
$ /mnt/cdrom/ztide/configure <your installation path>
```

The script will create a "config.in.mk" file describing the paths and the version numbers of the tools, in your temporary directory. To install them, run:

```
$ make
```

This creates a "ztide/version" directory (e.g. ztide/ZTIDE_V1.2_7).

The temporary directory may be deleted once the installation is complete. Use the `umount` command to eject the CD-ROM from the CD device:

```
$ umount /mnt/cdrom
$ eject
```

6.1.2 Required Environment Variables

The zTIDE software requires the following variables:

```
export ZTIDE_ROOT=<your installation path>/ztide/ZTIDE_version
export LD_LIBRARY_PATH=$ZTIDE_ROOT/lib:$LD_LIBRARY_PATH
```

You can either set the above variables by yourself or source the `ztide_env.bash` and `ztide_env.bash` (or other ones corresponding to your shell type) located in the zTIDE installation directory.



6.2 Installing License Feature

zTIDE software requires a specific license feature, named z_Tide, and the FLEXlm license server to work.

The same FLEXlm server is used for ZeBu and zTIDE, therefore if ZeBu is already installed, the FLEXlm server is also and only the license feature has to be installed.

If ZeBu is not already installed, refer to the “Installing FLEXlm” Section of the *ZeBu Installation Manual* and install FLEXlm before proceeding for the z_Tide license feature.



7 Reference Resources

7.1 ZeBu-XL Documents

To learn how to use the ZeBu-XL system, please refer to the corresponding manuals provided with the ZeBu-XL modules. The following Manuals are available:

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

The *ZeBu-XL Release Note* describes the new features of the current version of ZeBu-XL, compatibility with the previous version and details bug fixes.

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

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

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

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

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

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

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

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 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.

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.



7.2 ZeBu-ZV Documents

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

The *ZeBu-ZV Release Note* describes the new features of the current version of ZeBu-ZV, compatibility with the previous version and details bug fixes.

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

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

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

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

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

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

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

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

The *ZeBu-ZV 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.

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

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



8 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 820 San Jose, CA 95113 USA Tel: 1-888-7EveUSA (+1-888-738-3872)
Japan Headquarters	Nihon EVE KK Fuji Building 703 2-7-4, Shinyokohama 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