



The Fastest Verification

zTIDE Tutorial Manual

Document revision – a –
August 2005

Version 1.2_7



Table of Contents

ABOUT THIS MANUAL.....	4
OVERVIEW.....	4
INTENDED AUDIENCE.....	4
HISTORY.....	4
MANUAL CONTENTS.....	4
RELATED DOCUMENTATION.....	5
TYPOGRAPHIC CONVENTIONS USED IN THIS MANUAL.....	5
0 INTRODUCTION.....	7
0.1 zTIDE OVERVIEW.....	7
0.2 TUTORIAL REQUIREMENTS.....	7
0.2.1 Tools.....	7
0.2.2 System Environment.....	7
1 LAB 1: ZTIDE FOR A SIMPLE DESIGN.....	10
1.1 INTRODUCTION.....	10
1.2 DESIGN DESCRIPTION.....	11
1.3 TRANSACTOR.....	12
1.3.1 Description.....	12
1.3.2 Hardware Side.....	13
1.3.3 Software Side.....	20
1.4 THE ZTIDE WRAPPER.....	22
1.5 HDL COMPILATION.....	23
1.5.1 Compilation with ModelSim.....	24
1.5.2 Compilation with VCS.....	24
1.5.3 Compilation with NC-Verilog.....	25
1.6 C++ TESTBENCH ENVIRONMENT.....	25
1.6.1 Testbench Description.....	25
1.6.2 Compiling C++ Testbench.....	27
1.6.3 Clock Configuration.....	28
1.6.4 Running Testbench.....	28
1.7 SUMMARY.....	28
2 REFERENCE RESOURCES.....	29
2.1 ZEBU-XL DOCUMENTS.....	29
2.2 ZEBU-ZV DOCUMENTS.....	30
2.3 OTHER EVE DOCUMENTS.....	30
3 EVE CONTACTS.....	32



Figures

Figure 1: DUT.....	11
Figure 2: Testbench connected to Transactor and DUT.....	11
Figure 3: Transactional Architecture	12
Figure 4: Communication Infrastructure	13
Figure 5: Transactor Custom Part Principle	15
Figure 6: zTIDE wrapper.....	22

Tables

Table 1: Environment Variables	8
--------------------------------------	---



About this Manual

Overview

This Tutorial Manual illustrates, using a basic lab, how to use the ZeBu Transaction Interface Development Environment (zTIDE) to develop your transactors with a pure HDL simulation environment, using an industry-standard simulator, such as ModelSim®, VCS™ and NC-Verilog®.

The zTIDE lab presented in this manual is based on ZeBu Interface. The SCE-MI interface is described in the *ZeBu SCE-MI Manual* that includes a dedicated lab.

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 understand how to use zTIDE for transactor testing and debugging.

Engineers will typically have experience with the Verilog and VHDL languages, C/C++ languages, the Verilog API and an industry-standard simulator.

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
a	1.2_7	Aug 05	First Edition.

Manual Contents

Chapter numbering starts at zero so that the lab numbers correspond to the actual chapter numbers.

Chapter 0, “Introduction” highlights the prerequisites and the tools needed to run the labs, and describes how to set up your system environment to run the labs.

Chapter 1, “Lab 1: zTIDE for a Simple Design”, demonstrates the uses of zTIDE to simulate a simple design driven by a transaction-based testbench with a pure HDL simulation environment.



Related Documentation

The *zTIDE Manual* describes the ZeBu Transaction Interface Environment (zTIDE) describes how to develop and debug synthesizable transactors using zTIDE by connecting a transaction-level test bench (C/C++/SystemC/SystemVerilog) to an HDL simulator.

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* describes how to design and use transactors with ZeBu.
- The *ZeBu Compilation Manual* describes the complete steps of the compilation process.
- The *ZeBu Reference Manual* describes the various ZeBu tools and their use, and detailed descriptions about the syntax and keywords for the DVE file, drivers and macros.
- The *ZeBu-XL HDL Co-simulation Manual* describes the use of the HDL co-simulation driver for ZeBu.

The complete documentation package for ZeBu-XL and ZeBu-ZV are listed at the back of this manual, in Chapter 2.

Typographic Conventions Used in This Manual

ZeBu tools are shown in bold, mono-space “Courier New” font: **zxlBuild**, **zxlPar**, **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] )  
);
```



0 Introduction

0.1 zTIDE Overview

ZeBu supports transaction-level communication between a transaction-level testbench and an emulated design through synthesizable transactors. The synthesizable transactors may require a significant amount of time to develop and debug. zTIDE has been developed to reduce this time by allowing the connection of the same C transaction-level testbench with an HDL simulator, which simulates the synthesizable part of the hardware transactors and the Design Under Test (DUT).

A full description of the zTIDE environment is given in the *zTIDE Manual*.

0.2 Tutorial Requirements

0.2.1 Tools

0.2.1.1 zTIDE release

It is necessary to have the appropriate zTIDE version installed on your PC. For more details about installing zTIDE, refer to the *zTIDE Manual*.

0.2.1.2 HDL simulator (ModelSim or VCS or NC)

You must install an HDL simulator (currently either ModelSim or VCS or NC) and the corresponding `bin` directory on your PC and update your `PATH` variable.

If you obtain a response to one of the commands the commands “`which vsim`”, or “`which vcs`”, or “`which ncsim`” give responses, the software is installed on your PC.

0.2.2 System Environment

0.2.2.1 Directory Organization

The zTIDE software release includes a tutorial directory that contains all the files required to run the lab described in this manual. The `labs` directory contains the lab sources. Each lab corresponds to a directory with the following structure:

```
labn
|
|--+ src
|   |
|   |-- *.v
|   |-- *.cpp
```

The `labn/src` directory includes all the source files for the DUT and the testbench. Make sure that the labs are set in a directory you can access. If not, change the permissions or copy it to your home directory.



0.2.2.2 Setting up the Environment Variables

Before starting any lab, set up some environment variables as shown in the following table:

Table 1: Environment Variables

Environment Variable	Assignment	Example
\$ZTIDE_ROOT	zTIDE software Install Directory	/usr/local/ztide

It is also necessary to add `$ZTIDE_ROOT/lib` to your `LD_LIBRARY_PATH` variable.

Following scripts show how to proceed in `bash` or in `cshell`. It is also possible to set up your environment in `sh` or `tcsh`.

Setting the Environment Variables using bash

The following template script, “`ztide_env.bash`” is provided in the root directory. This script sets up all the variables and updates the `PATH` and `LD_LIBRARY_PATH` variables.

```
#Setting the ztide environment variables for bash.
#ZTIDE home directory
export ZTIDE_ROOT=/usr/local/ztide

#***** No Change Required Below *****
export LD_LIBRARY_PATH=$ZTIDE_ROOT/lib:$LD_LIBRARY_PATH
echo " ZTIDE_ROOT      : " $ZTIDE_ROOT
echo " LD_LIBRARY_PATH  : " $LD_LIBRARY_PATH
```

You may have to edit this file to fit your local environment

Run the command “`source ztide_env.bash`” in the root directory to set your environment with this script.

Setting the Environment Variables using cshell

The following template script, “`ztide_env.csh`” is provided in the root directory. This script sets up all the variables and updates the `PATH` and `LD_LIBRARY_PATH` variables.

```
#Setting the zTIDE environment variables for cshell
#ZTIDE home directory
setenv ZTIDE_ROOT /usr/local/ztide

#***** No Change Required Below *****
setenv LD_LIBRARY_PATH $ZTIDE_ROOT/lib:$LD_LIBRARY_PATH
echo " ZTIDE_ROOT      : " $ZTIDE_ROOT
echo " LD_LIBRARY_PATH  : " $LD_LIBRARY_PATH
```

You may have to edit this file to fit your local environment



Run the command “`source ztide_env.csh`” in the root directory to set your environment.



1 Lab 1: zTIDE for a Simple Design

1.1 Introduction

This lab demonstrates how to use zTIDE to simulate a simple design driven by a transaction-based testbench written in C++ in a pure simulation environment.

You will do the following:

1. Write and check the hardware part of the transactor.

Write the zTIDE wrapper

2. Compile HDL sources and start the simulation of the DUT and the hardware part of the transactor
3. Write the C++ testbench
4. Launch the testbench that will establish a connection to the simulation and then drive the DUT through the transactor

All the source files needed for this lab are stored in the `$ZTIDE_ROOT/labs/lab1/src` directory as follows:

```
lab1
|
|--+ src
|   |-- dut.v
|   |-- testbench.v
|   |-- Testbench.cc
|   |-- ztidewrapper.v
|
|--+ transactor
|   |-- fakezcei.v
|   |-- transactor.v
|   |-- serializer.v
|   |-- deserializer.v
|   |-- Transactor.hh
|   |-- Transactor.cc
```

To go to the `lab1` directory, execute “`cd lab1`” from the `labs` directory.

1.2 Design Description

The design is very simple in order to focus on the writing of the transaction-based testbench. It is a simple flip-flop.

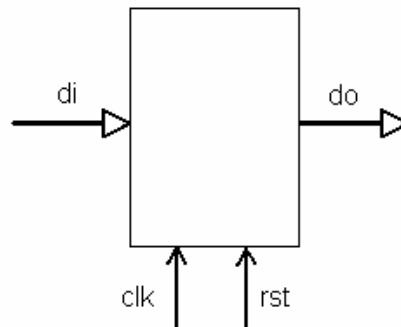


Figure 1: DUT

The corresponding Verilog source file, `dut.v`, is available in the `src` directory:

```
module dut (clk, rst, di, do);
  input  clk;
  input  rst;
  input  di;
  output do;
  reg do;

  always@(posedge clk or posedge rst)
    if (rst) begin
      do <= 0;
    end
    else begin
      do <= di;
    end
endmodule
```

The testbench will send and receive transactions of 32-bit packets instead of sending and receiving bits 1 by 1, and proceed with comparison:

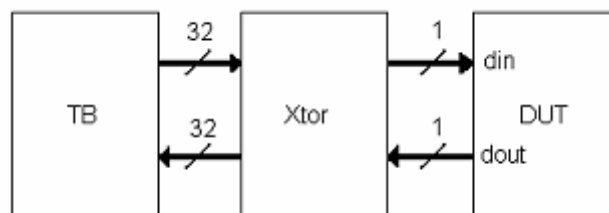


Figure 2: Testbench connected to Transactor and DUT

1.3 Transactor

1.3.1 Description

The transactor has to translate each 32-bit packet into a serial flow of 32 bits.

The transactor consists of a software part and a hardware part:

- The hardware part of the transactor is written at synthesizable level (Verilog or VHDL in this case).
- The software part is written for a C or C++ API, thus you can write it directly in C or C++ or integrate the C or C++ API with other languages such as System C or System Verilog.

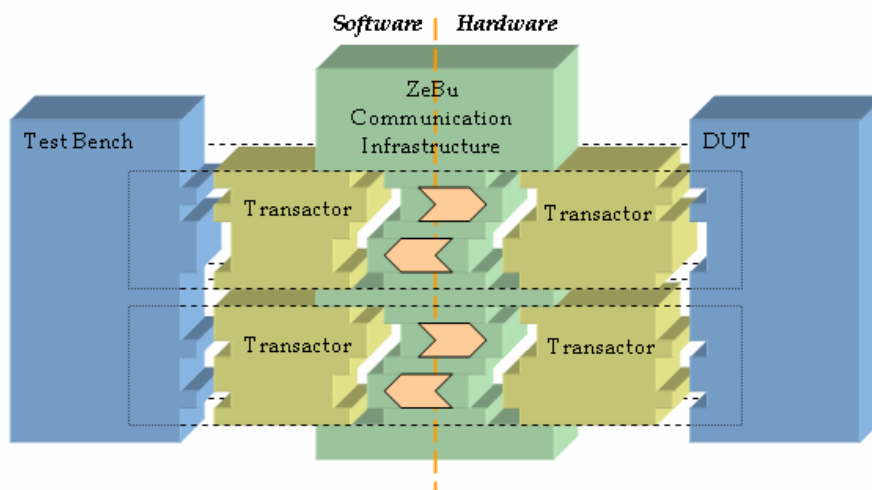


Figure 3: Transactional Architecture

For general information about transactor architecture, you should refer to the *ZeBu Transaction Based Verification Manual*.

1.3.2 Hardware Side

1.3.2.1 ZeBu Communication Infrastructure

The hardware part of the transactor instantiates a set of macros which control the clocks and communications between the software and hardware sides.

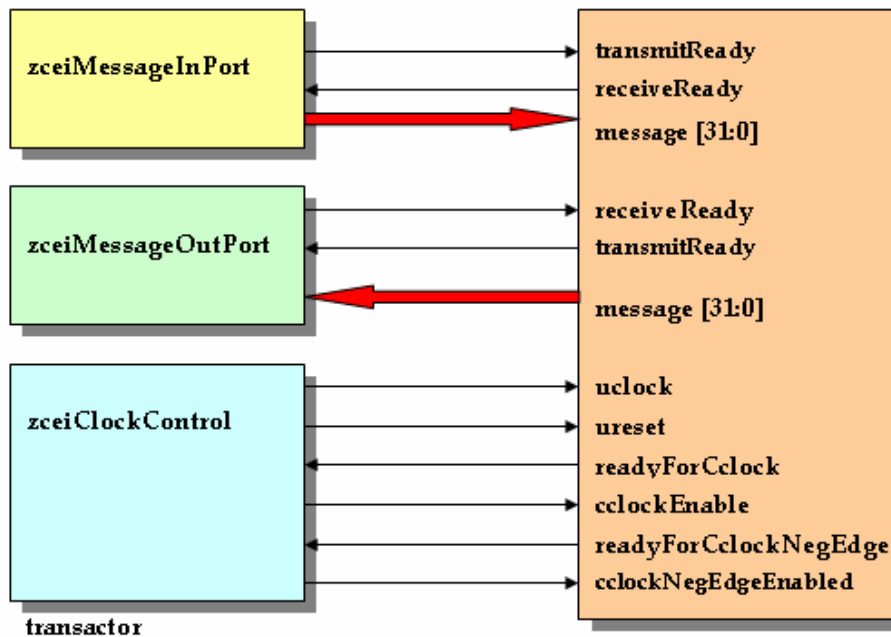


Figure 4: Communication Infrastructure

zceiClockControl:

Producing a posedge on the DUT clock

The posedge on the DUT clock occurs on a posedge front of `unlock` when the ZeBu `cclockEnabled` signal is asserted, which is only possible when all of the `readyForCclock` signals on all transactors are asserted.

Producing a negedge on the DUT clock

The negedge on the DUT clock occurs on a posedge front of `unlock` when the ZeBu `cclockNegEdgeEnabled` signal is asserted, which is only possible when all of the `readyForCclockNegEdge` signals on all transactors are asserted.



The information is exchanged between the hardware and software parts of the transactor through a set of ports. Two macros are used for input (from testbench to DUT) and output (from DUT to testbench):

zceiMessageInPort (from testbench to DUT) :

```
module zceiMessageInPort( transmitReady, receiveReady, message );
  parameter      Pwidth = 32;
  output         transmitReady;
  input          receiveReady;
  output [Pwidth - 1:0] message;
endmodule
```

The message is available on the `uclock` (transactor clock) posedge when both `transmitReady` and `receiveReady` are asserted, and remains available as long as `receiveReady` is asserted.

You can instantiate as many ports as needed.

Message size is set by default to 32 bits in the Verilog macro supplied for the zTIDE simulation. You can change this size by means of the parameter `Pwidth`, as described in Section 1.3.3.

zceiMessageOutPort (from DUT to testbench):

```
module zceiMessageOutPort( transmitReady, receiveReady, message );
  parameter      Pwidth = 32;
  input          transmitReady;
  output         receiveReady;
  input  [Pwidth - 1:0] message;
endmodule
```

Makes the message available and assert `transmitReady` signal. The message must remain available and `transmitReady` remain asserted until `receiveReady` is asserted on the posedge of `uclock`.

You can instantiate as many ports as needed.

Message size is set by default to 32 bits in the Verilog macro supplied for the zTIDE simulation. You can change this size using the `Pwidth` parameter, as described in Section 1.3.3.

See the *[ZeBu Transaction-Based Verification Manual](#)* for more details on protocol.

1.3.2.2 Transactor Custom part

Basically, the custom part of the transactor hardware side does the following:

- Receives 32-bit packets from software (through a `zceiMessageInPort`).
- Serializes and enables DUT clock (`serializer.v`).
- De-serializes DUT output.

- Sends 32-bit packets to software (through a zceiMessageOutPort).

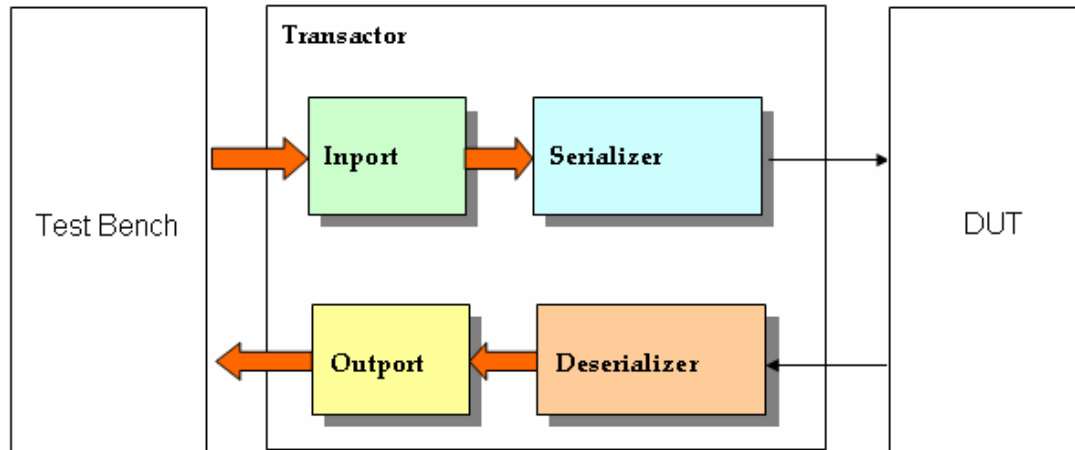


Figure 5: Transactor Custom Part Principle

The transactor must carefully take into account the `cclockEnabled` signal from `zceiClockControl`, because it indicates when the DUT clock is running.

You can find the transactor source files in `src/transactor` directory: `transactor.v`, `serializer.v` and `deserializer.v`.

1.3.2.3 Simulating the hardware part of the transactor

If you design your own transactor, it is recommended to simulate the hardware part before connecting it to the software part. In that case, you have to write your own models for hardware macros and a testbench which instantiates both DUT and transactor. Clocks are not part of the macros: you have to connect them via global assignment in the testbench.

Simulation models for hardware macros provided below are not complete and are not supposed to cover every case, but allow development of the transactor. The interest is only to permit a fast verification of the transactor in the early stages of the design. You can write models as accurate as you want.

Simulation models are in `src/transactor` directory:

- `fakezcei.v` for the 3 macros
- `transactor.v` for the transactor hardware part



Macros implementation in fakezcei.v:

```
module zceiMessageInPort(  
    transmitReady,  
    receiveReady,  
    message);  
  
    output transmitReady;  
    input receiveReady;  
    output [31:0] message;  
  
    wire clk; // driven by global assignement in testbench  
    wire rst; // driven by global assignement in testbench  
  
    reg transmitReady;  
    reg [31:0] message;  
  
    always@(negedge transmitReady)  
        #5555 transmitReady <= 1;  
  
    always @(posedge clk or posedge rst)  
        if(rst) begin  
            message <= 32'ha0000005;  
            transmitReady <= 0;  
        end  
        else if(receiveReady & transmitReady) begin  
            message <= message + 1;  
            transmitReady <= 0;  
        end  
    end  
  
    initial begin  
        transmitReady = 0;  
        #5555 transmitReady = 1;  
    end  
end  
  
endmodule  
  
module zceiMessageOutPort(  
    transmitReady,  
    receiveReady,  
    message);  
  
    input transmitReady;  
    output receiveReady;  
    input [31:0] message;  
  
    wire clk; // driven by global assignement in testbench  
    wire rst; // driven by global assignement in testbench  
  
    wire receiveReady;
```



```
assign receiveReady = 1;

reg [31:0] messageOutPort;

always @(posedge clk or posedge rst)
  if(rst) begin
    messageOutPort <= 0;
  end
  else begin
    if(transmitReady & receiveReady)
      messageOutPort <= message;
    end

endmodule

module zceiClockControl(
  uclock,
  ureset,
  readyForCclock,
  cclockEnabled,
  readyForCclockNegEdge,
  cclockNegEdgeEnabled);

  output uclock;
  output ureset;
  input  readyForCclock;
  output cclockEnabled;
  input  readyForCclockNegEdge;
  output cclockNegEdgeEnabled;

  wire cclockEnabled;
  wire cclockNegEdgeEnabled;

  reg rstn_sys;
  reg clk_sys;

  reg ureset;
  reg uclock_reg;
  reg cclock_reg;

  wire uclock;
  wire cclock;

  always #5 clk_sys <= ~clk_sys;

  initial begin
    clk_sys = 0;
    rstn_sys = 1;
    #40 rstn_sys = 0;
  end
end
```



```
assign #4 cclockEnabled = ~cclock_reg & readyForCclock;  
assign #4 cclockNegEdgeEnabled = cclock_reg & readyForCclockNegEdge;  
  
always @(posedge clk_sys or negedge rstn_sys) begin  
    if (rstn_sys) begin  
        ureset <= 1;  
        uclock_reg <= 0;  
        cclock_reg <= 0;  
    end  
    else begin  
        ureset <= 0;  
        uclock_reg <= ~uclock_reg;  
        if (uclock_reg == 0)  
            cclock_reg <= (~cclock_reg & readyForCclock) | (cclock_reg &  
~readyForCclockNegEdge);  
        end  
    end  
end  
  
assign #3 cclock = cclock_reg;  
assign #3 uclock = uclock_reg;  
  
endmodule
```

Below is the sample Verilog testbench that you can find in `testbench.v` in the `src` directory:

```
module top;  
  
    wire clk;  
    wire rst;  
    wire di;  
    wire do;  
  
    assign clk = xtor.clkCtrl.cclock;  
    assign xtor.inport.clk = xtor.clkCtrl.uclock;  
    assign xtor.inport.rst = xtor.clkCtrl.ureset;  
    assign xtor.outport.clk = xtor.clkCtrl.uclock;  
    assign xtor.outport.rst = xtor.clkCtrl.ureset;  
  
    dut dut(.clk(clk), .rst(rst), .di(di), .do(do));  
  
    transactor xtor(.dut_rst(rst), .dut_di(di), .dut_do(do));  
  
endmodule
```

You have to create a simulation directory in which you will simulate the design. From the `lab1` directory, do the following:

```
mkdir rundir  
cd rundir
```



1.3.2.4 Using ModelSim for Transactor Simulation

```
>$ vlib work
>$ vlog ../src/testbench.v ../src/transactor/deserializer.v
    ../src/transactor/fakezcei.v ../src/transactor/serializer.v
    ../src/transactor/transactor.v
>$ vsim top
```

1.3.2.5 Using VCS for Transactor Simulation

```
>$ vcs ../src/testbench.v ../src/transactor/deserializer.v
    ../src/transactor/fakezcei.v ../src/transactor/serializer.v
    ../src/transactor/transactor.v -RI -line -M
```

1.3.2.6 Using NC-Verilog for Transactor Simulation

```
>$ ncverilog +gui ../src/testbench.v
    ../src/transactor/deserializer.v ../src/transactor/fakezcei.v
    ../src/transactor/serializer.v ../src/transactor/transactor.v
+access+rw +nbasync
```



1.3.3 Software Side

The software part of the transactor is implemented as a C++ class. You can find source files for the transactor in the `src/transactor` directory: `Transactor.hh` and `Transactor.cc`.

The following methods are implemented:

- A constructor to initialize the transactor.
- A destructor to release resources once simulation is finished.
- A `send()` method to send data to DUT.
- A `receive()` method to get data from DUT.
- A `read()` method to read the last data received from DUT.

```
class Transactor
{
public:
    Transactor(Board *board, const char *name);
    ~Transactor();

    void send(unsigned int data);
    unsigned int receive();
    unsigned int read();

private:
    TxPort _txp;
    RxPort _rxp;

    int _count;
};
```

1.3.3.1.1 Constructor

The constructor instantiates data structures that enables communication between hardware and software. The information is exchanged between hardware and software through a set of ports. Two classes are used for input and output:

- TxPort to send data from software to hardware
- RxPort to receive data into software from hardware.

To create a port connected to the `zceiMessageInPort` macro in the hardware, instantiate an object of class TxPort (transmit port) with, as argument, the name given to the `zceiMessageInPort` instance in the transactor (`inport`):

```
_txp("inport"); // SW => HW
```

To create a port connected to the `zceiMessageOutPort` macro in the hardware, instantiate an object of class RxPort (receive port) with, as argument, the name given to the `zceiMessageOutPort` instance in the transactor (`outport`):

```
_rxp("outport"); // HW => SW
```



When ports are created, connect them to the hardware transactor. For this, use the connect method:

```
_txp.connect(<board>, <name>);  
_rxp.connect(<board>, <name>);
```

Where <board> is a handler to the ZeBu board and <name> is the transactor instance name as declared in the zTIDE wrapper (e.g. “xtor”).

1.3.3.1.2 Destructor

Destructor releases resources and disconnects ports.

```
_txp.disconnect();  
_rxp.disconnect();
```

1.3.3.1.3 Send method

Sending data consists of three steps:

1. Write the data into the buffer

```
_txp.write(0, data);
```

The first argument is the word identification number to write. The second argument is the value to write (unsigned integer value).

2. Verify that it is possible to send the data:

```
while(!_txp.isPossibleToSend()) {  
or
```

```
if(_txp.isPossibleToSend()) {
```

The method `isPossibleToSend()` returns true when you can send data, and returns false otherwise.

3. Send the data

```
_txp.sendMessage();
```

The `sendMessage()` method sends the message to the hardware.

1.3.3.1.4 Receive method

Receiving data also requires three steps:

1. Verifying that it is possible to receive data

The method `isPossibleToReceive()` returns true when you can receive data, and returns false otherwise.

```
while(!_rxp.isPossibleToReceive()) {
```

2. Receiving message

The `receiveMessage()` method receives the message from the hardware.

```
_rxp.receiveMessage();
```

3. Reading data

The `read()` method takes as argument the word number you want to read and returns the value of the word (unsigned integer).

```
return _rxp.read(0);
```

1.4 The zTIDE wrapper

The zTIDE wrapper describes how the transactor, the DUT and the design clock and reset are connected together.

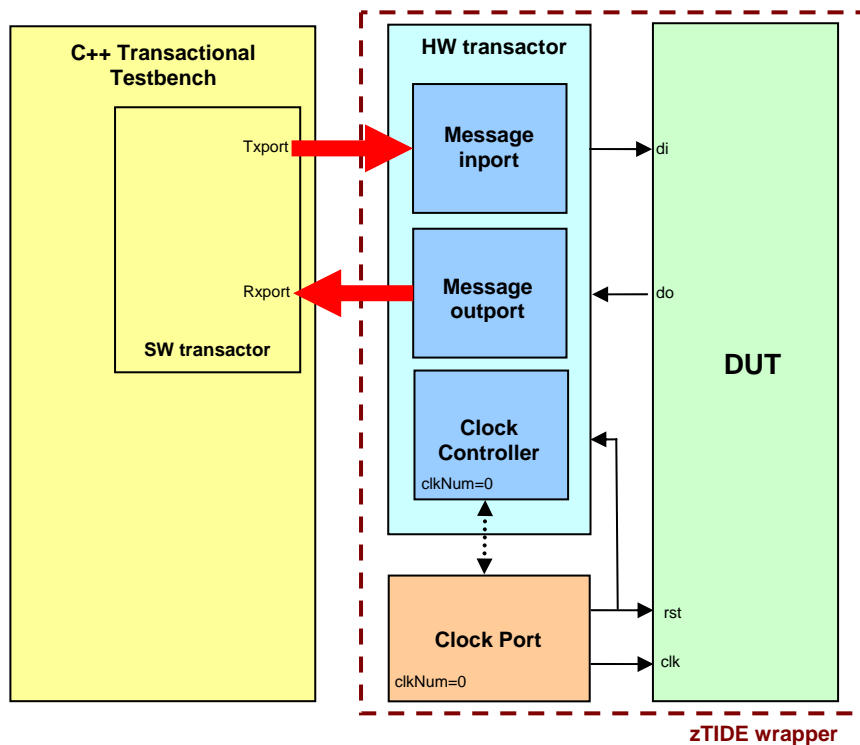


Figure 6: zTIDE wrapper

The zTIDE wrapper source file, `ztidewrapper.v`, is available in the `src` directory:



```
`timescale 1ns/100ps

module wrapper;

    wire clk;
    wire rst;
    wire di;
    wire do;

    dut dut(.clk(clk), .rst(rst), .di(di), .do(do));

    transactor xtor(.dut_rst(rst), .dut_in(di), .dut_out(do));
    defparam xtor.clkCtrl.clockNum = 0;
    defparam xtor.inport.Pwidth = 32;
    defparam xtor.outport.Pwidth = 32;

    zceiClockPort zceiClockPort(.cclock(clk), .creset(rst));
    defparam zceiClockPort.clockNum = 0;

endmodule
```

In this example, “xtor” is the name of the driver you connect to the DUT. You will use this name as a definition name in the software part.

The design clock and reset are not standard inputs: they are connected to a reserved clock port, “zceiClockPort” in present example.

The system supports several independent clocks. The `clockNum` parameter of the `zceiClockPort` macro, instantiated in the zTIDE wrapper, and the `zceiClockControl` macro, instantiated in the transactor, allow DUT clock identification number, when controlled by the transactor.

1.5 HDL compilation

You need to compile the Verilog source files of the DUT for both the hardware part of the transactor and the ZCEI macros installed in `$ZTIDE_ROOT/verilog` for the zTIDE simulation.

Create the compilation directory from the `lab1` directory and move to this one:

```
mkdir rundir
cd rundir
```

Compile the Verilog files and start the HDL simulation according to the simulator you are using as described in the following sections.

If the simulation starts and waits for the testbench connection, the following messages should display:

```
Initialize connection of hardware side to software side
Wait for software side connection
```



1.5.1 Compilation with ModelSim

To compile with ModelSim, launch the following command:

```
$> vlib work

$> vlog ../src/dut.v \
        ../src/ztidewrapper.v \
        ../src/transactor/deserializer.v \
        ../src/transactor/serializer.v \
        ../src/transactor/transactor.v \
        $(ZTIDE_ROOT)/verilog/zceiClockControl.v \
        $(ZTIDE_ROOT)/verilog/zceiClockPort.v \
        $(ZTIDE_ROOT)/verilog/zceiMessageInPort.v \
        $(ZTIDE_ROOT)/verilog/zceiMessageOutPort.v
```

To start the HDL simulator from the compilation directory:

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

1.5.2 Compilation with VCS

To compile with VCS, launch the following command:

```
$> vcs ../src/dut.v \
        ../src/ztidewrapper.v \
        ../src/transactor/deserializer.v \
        ../src/transactor/serializer.v \
        ../src/transactor/transactor.v \
        $(ZTIDE_ROOT)/verilog/zceiClockControl.v \
        $(ZTIDE_ROOT)/verilog/zceiClockPort.v \
        $(ZTIDE_ROOT)/verilog/zceiMessageInPort.v \
        $(ZTIDE_ROOT)/verilog/zceiMessageOutPort.v \
        -P $ZTIDE_ROOT/lib/pli.tab \
        $ZTIDE_ROOT/lib/libZebuSim.so
```

To start the HDL simulator from the compilation directory:

```
$> ./simv
```



1.5.3 Compilation with NC-Verilog

To compile and start the HDL simulator with NC Verilog:

```
$> ncverilog ../src/dut.v \  
    ../src/ztidewrapper.v \  
    ../src/transactor/deserializer.v \  
    ../src/transactor/serializer.v \  
    ../src/transactor/transactor.v \  
    $(ZTIDE_ROOT)/verilog/zceiClockControl.v \  
    $(ZTIDE_ROOT)/verilog/zceiClockPort.v \  
    $(ZTIDE_ROOT)/verilog/zceiMessageInPort.v \  
    $(ZTIDE_ROOT)/verilog/zceiMessageOutPort.v \  
    +access+rw +nbasync +loadplil=libZebuSim.so:bootstrap
```

1.6 C++ Testbench Environment

1.6.1 Testbench Description

The software testbench sends and receives 32-bit words to/from hardware and compares them for checking. The C++ transactional testbench, `Testbench.cc`, is in the `src` directory.

The following methods are implemented:

- Initialization of the simulation.
- Instantiate transactor.
- Run.
- Close.
- Display a message.

1.6.1.1 Initialization

At the beginning of the C++ testbench, the simulation must be initialized and properly connected to the software environment.

1. Opening the session:

```
board = Board::open();
```

If (`board == NULL`): there is a problem and you must stop.

2. Instantiating the transactor:

```
Transactor xtor(board, "xtor")
```

“xtor” is the instance name of the transactor as given in the zTIDE wrapper.

3. Initializing ZeBu:

```
board->init();
```



In this example, clock is automatically created.

You must check the return value of the `init()` method. If it is different from zero you should close ZeBu (using the `close()` method) and stop your testbench.

1.6.1.2 Run

The testbench shows two ways to use the transactor:

Reactive mode

In this mode, you alternatively send and receive data with transactor methods `send()` and `receive()`.

```
for(i = 0; i < 100; i++) {
    data = random();
    xtor.send(data);
    if (xtor.receive() != data) {
        error++;
        cerr << "ERROR : "
            << (dec) << i
            << " " << (hex) << data
            << " " << (hex) << xtor.read() << endl;
    }
}
```

Streaming mode

In this mode, you send as many messages as possible, then receive as many messages as possible with the transactor methods `send()` and `receive()`.

```
for(j = 0; j < 100; j++) {
    for(i = 0; i < 16; i++) {
        table[i] = random();
        xtor.send(table[i]);
    }
    for(i = 0; i < 16; i++) {
        unsigned int received;
        received = xtor.receive();
        if (table[i] != received) {
            error++;
            cerr << "ERROR : "
                << (dec) << i
                << " " << (hex) << table[i]
                << " " << (hex) << received << endl;
        }
    }
}
```

The two modes presented before are based on a pooling principle. They are usually used for sequential testbench, yet it is not very useful for testbench decomposed into concurrent parts.



When using transactors based on parallel architecture, you may develop event-based transactors using callbacks to send and receive messages. For more information see *ZeBu Transaction Based Verification Manual* (especially Section 3.2), as well as the `parrallelRomReader` example in `zTIDE_ROOT/example/ztide_pure/`.

1.6.1.3 Close

Close the session, the HDL simulation and the C/C++ API.

```
board->close();
```

1.6.1.4 Display a message

Indicate if an error occurred during the run: `NO ERROR DETECTED` appears if the run succeeded.

```
if (error) {
    status = 1;
    cerr << error << " ERROR(S) DETECTED" << endl;
}
else {
    cerr << " NO ERROR DETECTED" << endl;
}
```

1.6.2 **Compiling C++ Testbench**

Create the run directory from the `lab1` directory and move to this one.

```
mkdir rundir
cd rundir
```

To compile the object file, type the following command from the run directory:

```
g++ -c ../src/transactor/Transactor.cc \
-I$ZTIDE_ROOT/include \
-I../src/transactor

g++ -c ../src/Testbench.cc \
-I$ZTIDE_ROOT/include \
-I../src/transactor
```

The results are the object files: `Transactor.o` and `Testbench.o`.

To link the object files, type the following command:

```
g++ -o testbench Testbench.o Transactor.o -L$ZTIDE_ROOT/lib -lzebu
```



1.6.3 Clock Configuration

The waveform of the design clock can be customized using a configuration file, “clk.eve”, in the run directory. You have to write this file, and it has to be specified exactly as follows. For more details refer to Configuration of clocks and reset of DUT in *zTIDE Manual*.

In the clk.eve file:

```
$B0.clk.Waveform = “_-”;  
$B0.clk.VirtualFrequency = 1;
```

In the run directory write a designFeatures file too, which contains the path of the “clk.eve” clock file. For more details refer to Configuration of clocks and reset of DUT in *zTIDE Manual*).

In the designFeatures file:

```
$B0.clk.File = “clk.eve”;
```

1.6.4 Running Testbench

In the run directory launch the testbench executable:

```
./testbench
```

If the testbench succeeded in establishing a connection to the HDL simulation the message “Connection of client socket has been established” displays, otherwise an error will display.

Then, if the run succeeded, the message “NO ERROR DETECTED” displays, otherwise an error will display.

1.7 Summary

This lab showed how to use the zTIDE environment to develop your transactors. You designed a transactor composed of a hardware part and a software part. Then you created a C++ testbench that instantiated your transactor.



2 Reference Resources

2.1 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++ testbench 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.



2.2 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 Testbench Manual* describes the use of a Synthesizable Testbench 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++ testbench 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.

2.3 Other EVE Documents

The *zTIDE Manual* describes the ZeBu Transaction Interface Environment (zTIDE) describes how to develop and debug synthesizable transactors using the zTIDE utility by connecting a transaction-level C++ test bench to an HDL simulator.



The *SCE-MI for ZeBu Manual* describes the use of the Standard Co-Emulation API for Modeling Interface (SCE-MI) for the ZeBu platform, in compliance with SCE-MI 1.1 standard. It describes the infrastructure linkage process within ZeBu compilation flow and the usage of the runtime SCE-MI API.



3 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