ZeBu®
LPDDR4 SDRAM Memory Models

Version 2014.09

Document Revision – b –
August 2014

VSIP024

Copyright Notice and Proprietary Information

Destination Control Statement

Disclaimer

Trademarks

Third-Party Links

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Table of Contents

Figures

Tables

About This Manual

Overview

This manual describes how to use the ZeBu ZLPDDR4 SDRAM synthesizable memory model libraries with optimized memory capacity and performance.

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	2014.09	Aug 14	**New features:** • Compatibility with Cadence NCSim tool for simulation (Sections 1.6.3 and 4.3) • `probe` output for the memory model interface (Chapter 3) • Alias files for Verdi provided (Section 6.2) • Waveform examples (Section 7.1) **Updated features:** • zrm address translation information updated for 6Gb and 12Gb memory models (Section 3.6 and Chapter 5) • Warning about the `mem_core_sp` instance in load and dump operations (Section 5.1) • `probe` signals updated (Section 6.1) • Xilinx dedicated environment variable modified (Section 7.2.3.1)
a	2014.06	May 14	First edition.

Related Documentation

For details about the ZeBu supported features and limitations, you should refer to the *ZeBu Release Notes* in the ZeBu documentation package which corresponds to the software version you are using.

For information about synthesis and compilation for ZeBu, you should refer to the *ZeBu Compilation Manual* and the *ZeBu zFAST Synthesizer Manual*.

For additional guidelines for an optimal integration process of ZeBu DRAM memory models, you should refer to the dedicated Application Note, VSAN001: *Guidelines for the use of ZDDRx Models*.

1 Introduction

1.1 ZLPDDR4 Memory Models

The ZeBu ZLPDDR4 synthesizable memory models can be used to model any Synchronous Low Power Double-Data Rate 4 (LPDDR4) DRAM.

The ZLPDDR4 library is provided as a set of IP models, with various densities and architectures listed in Section 1.4, compliant with LPDDR4 SDRAM memory devices.

These models are based on ZeBu zrm-based memory models. The type and size of zrm-based memory models depend on the ZLPDDR4 size and architecture.

The ZLPDDR4 memory models provide the usual ZeBu hardware debugging features such as runtime memory upload/download and memory cell READ/WRITE.

1.2 LPDDR4 Compliance

The ZLPDDR4 memory models are compliant with the LPDDR4 specifications documents issued by the JEDEC (JC42.6) and available to JEDEC members (www.jedec.org).

1.3 ZLPDDR4 Features

The ZeBu ZLPDDR4 SDRAM memory models provide the following features:

- Provides cycle-accurate SDRAM device model implemented in ZeBu.
- Provides memory blocks and locations initialization and dump at runtime.
- Includes HDL simulation model for DUT integration testing.
- Provides bidirectional RTL blackbox components located in the `component` directory.
- Provides bidirectional Verilog or VHDL RTL wrapper files to include the ZLPDDR4 memory model within the user DUT, located in the `wrapper_rtl` directory

1.4 ZLPDDR4 Library

ZeBu ZLPDDR4 memory models are available for 4Gb, 6Gb, 8Gb, 12Gb and 16Gb memory capacity components on ZeBu Server-1 and ZeBu-Server-2.

For each capacity, the ZLPDDR4 memory component comes in 2-channel and x16 architectures.

Table 1: ZLPDDR4 Models

Total Memory Density	Memory Density (per channel)	Memory Model
4 Gb	**2 Gb**	`zlpddr4_4Gb_2CHANNEL_x16`
6 Gb	**3 Gb**	`zlpddr4_6Gb_2CHANNEL_x16`
8 Gb	**4 Gb**	`zlpddr4_8Gb_2CHANNEL_x16`
12 Gb	**6 Gb**	`zlpddr4_12Gb_2CHANNEL_x16`
16 Gb	**8 Gb**	`zlpddr4_16Gb_2CHANNEL_x16`

1.5 ZLPDDR4 Capacity with ZeBu

The ZeBu Server-1 and ZeBu-Server2 systems can both handle up to 8 ZLPDDR4 memory model instances.

1.6 Requirements

1.6.1 `FLEXlm` License Features

You should receive the license features you need by email or you can get it from the SolvNet database.

If you encounter any problem, please contact your local Synopsys support team.

1.6.2 Compilation Setup

The following setup is mandatory to compile the IP within the DUT:

Environment	ZeBu Server-1	ZeBu Server-2	ZeBu Server-3
64-bit Linux OS	6_3_1 or later	6_3_2 or later	8_0_2 or later

1.6.3 Simulation Setup

The following setup is mandatory to use the ZLPDDR4 simulation model:

Environment	Synopsys VCS	MTI ModelSim	Cadence NCSim
32-bit and 64-bit Linux OS	Compatible with: `vcs-mx-2006.9` or later. Tested version: `vcs-mx-2012.9`	Compatible with: `modelsim-6.4` or later. Tested version: `modelsim-6.6b`	Compatible with: `ncsim-13.20.s005` or later. Tested version: `ncsim-13.20.s005`

Note The ZLPDDR4 simulation model can be used on any platform (32 or 64 bits) and does not require ZeBu software.

1.7 Performance

1.7.1 Logic Resources

The ZLPDDR4 synthesizable ZeBu models use the following FPGA resources:

Table 2: Logic Resources Example

Memory Model	Resources
`zlpddr4_4Gb_2CHANNEL_x16`	1856 registers / 2679 LUTs / 1 single-port zrm
`zlpddr4_16Gb_2CHANNEL_x16`	1874 registers / 2567 LUTs / 1 single-port zrm

1.7.2 Operating Frequency on ZeBu

The following performance table has been tested with a primary clock connected directly to the ZLPDDR4 `CK_t_A/CK_t_B` clock input.

Table 3: Operating Frequency

ZeBu	`driverClk`	ZLPDDR4 clock (`CK_t_A/CK_t_B`)	`zTime` report (for `driverClk`)
ZeBu Server-1 (DDR3 Mem Server)	12.5 MHz	6.25 MHz	5.8 MHz
ZeBu Server-3 (DDR3 Mem Server)	4.76 MHz	2.38 MHz	5.0 MHz

Note These figures may drop by 10 to 20% if the memory server is shared by several large design memories.

1.8 ZLPDDR4 DIMM Memory Models

ZeBu ZLPDDR4 memory models can be used to build UDIMM (Unbuffered DIMM) and RDIMM (Registered DIMM) memory models.

ZLPDDR4_UDIMM and ZLPDDR4_RDIMM memory models are customized and supplied by Synopsys, upon request, according to detailed user requirements.

1.9 Limitations

The following LPDDR4 operations/features are not supported and ignored by the current ZLPDDR4 models. A probe bit is raised when these features/operations are received by the ZLPDDR4 (please refer to Chapter 6 for further details):

- MPC command :
    - `WRITE FIFO`
    - `READ FIFO`
    - `READ DQ CALIBRATION`
- Write leveling (`MR2[7]`)
- Command bus training (`MR13[0]`)
- Post-package repair (`MR4[4]`)
- The Refresh and Self Refresh operations are ignored by the ZLPDDR4 model.

2 Installation

2.1 Installing the ZLPDDR4 Package

2.1.1 Installation Procedure

To install the ZLPDDR4 package, proceed as follows:

1. Make sure the `ZEBU_IP_ROOT` environment variable in your shell points to your IP installation directory. Set it accordingly otherwise.
2. Launch the installation script as follows:

```
./ZLPDDR4.<VERSION>.sh install
```

Note If the `ZEBU_IP_ROOT` environment variable is not set, you may launch the installation script as follows:

```
./ZLPDDR4.<VERSION>.sh install <ZEBU_IP_ROOT>
```

The installation script executes the following operations:

- It creates the `$ZEBU_IP_ROOT/HW_IP/ZLPDDR4.<VERSION>` directory.
- It extracts all files of the package into this directory.
- It creates a `$ZEBU_IP_ROOT/HW_IP/ZLPDDR4` symbolic link to target the `$ZEBU_IP_ROOT/HW_IP/ZLPDDR4.<VERSION>` directory.
- It unzips the `.gz` simulation library files of the package to the `$ZEBU_IP_ROOT/HW_IP/lib` directory, and symbolic links are created to target the most recent libraries.

2.1.2 Package Description

Once correctly installed, the ZLPDDR4 memory models come with the following elements:

- Encrypted ZeBu models' EDIF netlists, for implementation (`edif` directory)
- RTL wrappers to include the component in the DUT as a Verilog RTL source file for synthesis (`wrapper_rtl` directory)
- ZLPDDR4 component blackbox, for design synthesis (`component` directory)
- Verilog gate-level netlist, for model HDL simulation (`simu/gate` directory)
- Verilog RTL-level netlist, for model HDL simulation (`simu/rtl` directory)
- Simulation and emulation examples (`example` directory)
- User scripts (`script` directory)
- `.so` libraries, for model HDL simulation (`libIpSimu` directory)
- This manual (`doc` directory)

2.1.3 File Tree

Here is the file tree for the ZLPDDR4 memory model after package installation:

```
$ZEBU_IP_ROOT
  `--HW_IP
     `--ZLPDDR4.<version>
        |-- <size>
        |    `-- <arch>
        |         |-- component
        |         |    |-- <model_name>_bidir.v
        |         |    `-- <model_name>_bidir.vhd
        |         |-- edif
        |         |    `-- <model_name>.edf
        |         |-- simu
        |         |    `-- <simu_type>
        |         |         `-- <simulator>
        |         |              `-- <model_name>.vp
        |         `-- wrapper_rtl
        |              |-- <model_name>_bidir.v
        |              `-- <model_name>_bidir.vhd
        |-- doc
        |-- example
        |    `-- waveform
        |-- script
        |    `-- nWave
        |         |-- cmd.alias
        |         `-- probe.alias
        |-- install
        |-- lib
        |    |-- libIpSimu_32.<version>.so.gz
        |    `-- libIpSimu_64.<version>.so.gz
        `-- uninstall
```

where:

- `<size>=4Gb/6Gb/8Gb/12Gb/16Gb`
- `<arch>=x16` architecture
- `<model_name>=` any of the memory models listed in Section 1.4.
- `<simu_type>` = “`gate`” or “`rtl`”
- `<simulator>` = “`mti`”, “`vcs`” or “`ncsim`”

2.2 Uninstalling the ZLPDDR4 Package

To uninstall the ZLPDDR4 package, launch the automatic uninstallation script as follows:

```
source ${ZEBU_IP_ROOT}/HW_IP/ZLPDDR4.<VERSION>/uninstall
```

The uninstallation script executes the following operations:

- It removes the `$ZEBU_IP_ROOT/HW_IP/ZLPDDR4.<VERSION>` directory.
- It removes the `$ZEBU_IP_ROOT/HW_IP/ZLPDDR4` symbolic link for this package version.

3 ZLPDDR4 Memory Models Description

The ZLPDDR4 synthesizable memory models are internally configured as multi-bank DRAMs and instantiate a ZeBu memory primitive (zrm) to model the DRAM memory array.

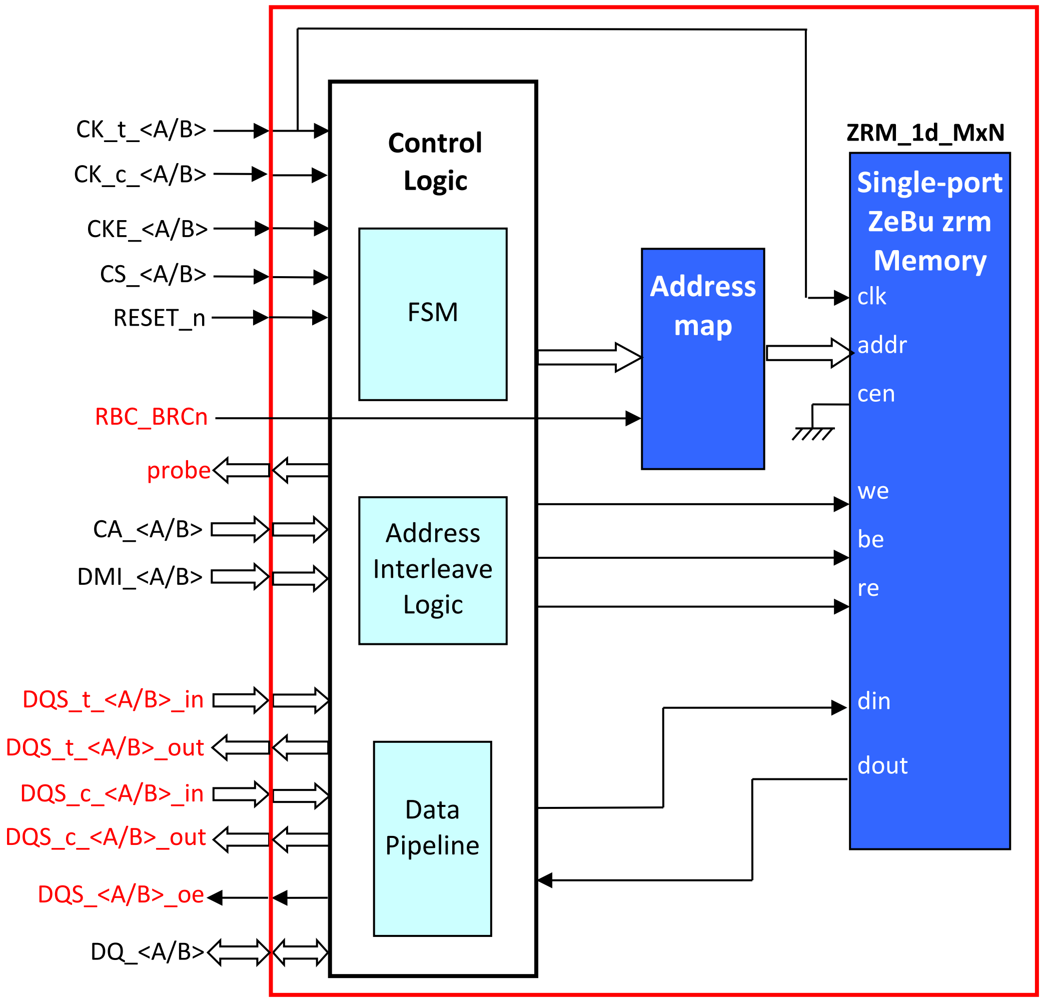
The zrm-based memory model used depends on the LPDDR4 size and architecture (see Section 1.4).

3.1 Overview

The ZLPDDR4 synthesizable memory models can be used to model any Synchronous Low Power Double-Data Rate 4 (LPDDR4) DRAM, using zrm memory resources. Such zrm-based memory models provide the usual ZeBu hardware debugging features such as runtime memory upload/download and memory cell READ/WRITE.

3.2 Functional Block Diagram

The ZLPDDR4 synthesizable ZeBu memory models are architectured as follows:



Remarks:

- *A/B is the channel number A or B*
- *Signals in red are specific ZLPDDR4 signals that do not exist in the LPDDR4 standard interface (see Section 3.4.1 for further details.).*

Figure 1: ZLPDDR4 Model Architecture

3.3 Interfaces of ZLPDDR4 Memory Model

Figure 1 shows the interface of the ZLPDDR4 memory. It is provided with the ZLPDDR4 Verilog or VHDL wrapper that allows using the model with a JEDEC-compliant bi-directional interface. Please refer to Section 4.1 for further details on how to use the wrapper.

The tables below describe the unidirectional and bidirectional ZLPDDR4 interfaces. The gray rows indicate specific ZLPDDR4 signals that do not exist in the LPDDR4 standard interface.

Table 4: ZLPDDR4 Unidirectional Interface

Name	Type	Description

`CK_t_A`
`CK_t_B`
`CK_c_A`
`CK_c_B`
	Input	Clock: `CK_t_<A/B` and `CK_c_<A/B>` are differential clock inputs. All address command and control input signals are sampled on positive edges of `CK_t_<A/B>`. Each channel (A and B) has its own clock pair
		`CK_c_<A/B>` (negative) are not used in the ZLPDDR4 model.

`CKE_A` `CKE_B`	Input	Clock Enable: `CKE` HIGH activates and `CKE` LOW de-activates internal clock signals and therefore device input buffers and output drivers.
`CS_A` `CS_B`	Input	Chip Select: is considered part of the command code.
`RESET_n`	Input	Reset: when asserted LOW, this reset the both channels.
`CA_A` `CA_B`	Input	Command/Address Inputs
`DMI_A` `DMI_B`	I/O	Data Mask(DM) or Data Bus Inversion (DBI) according to the mode register configuration
`DQ_A` `DQ_B`	I/O	Data Input/Output: Bi-directional data bus.

`DQS_t_A_in`
`DQS_t_B_in`
`DQS_c_A_in`
`DQS_c_B_in`
	Input	Data Strobe Input (Differential, `DQS_t` and `DQS_c`).
		`DQS_c_A_in` and `DQS_c_B_in` are not used in the ZLPDDR4 model.

`DQS_t_A_out`
`DQS_t_B_out`
`DQS_c_A_out`
`DQS_c_B_out`
	Output	Data Strobe Output (Differential: `DQS_t` and `DQS_c`). It is output with READ data from the memory for each channel.
		`DQS_t_out` is edge-aligned to READ data.
		`DQS_c_out` (negative) is edge-aligned to READ data.

`DQS_A_oe` `DQS_B_oe`	Output	DQS output enable signal. Refer to Section 3.4.1 for further details.
`probe`	Output	Debug probe. Please refer to Chapter 6 for further details.
`RBC_BRCn`	Input	Addressing mode: it is defined as a parameter (`USER_RBC_BRCn`) for the component instance. The parameter value could be 0 for BRC mode and 1 for RBC mode. Default value is 0.

Table 5: ZLPDDR4 Bi-directional Interface

Name	Type	Description
CK_t_A CK_t_B CK_c_A CK_c_B	Input	Clock: CK_t_<A/B> and CK_c_<A/B> are differential clock inputs. All address command and control input signals are sampled on positive edges of CK_t_<A/B>. Each channel (A and B) has its own clock pair . **CK_c_<A/B> (negative) are not used in the ZLPDDR4 model.**
CKE_A CKE_B	Input	Clock Enable: CKE HIGH activates and CKE LOW de-activates internal clock signals and therefore device input buffers and output drivers.
CS_A CS_B	Input	Chip Select: is considered part of the command code.
RESET_n	Input	Reset: when asserted LOW, this reset the both channels.
CA_A CA_B	Input	Command/Address Inputs.
DQ_A DQ_B	I/O	Data Input/Output: Bi-directional data bus.
DQS_t_A DQS_t_B DQS_c_A DQS_c_B	I/O	Data Strobe (Bi-directional, Differential): The data strobe is bi-directional (used for READ and WRITE data) and differential (DQS_t_<A/B> and DQS_c_<A/B>). It is output with READ data and input with WRITE data. DQS_t is edge-aligned to READ data and centered with WRITE data.
DMI_A DMI_B	I/O	Data Mask(DM) and/or Data Bus Inversion (DBI) according to the mode register configuration for READ/WRITE operations.
ODT_CA_A ODT_CA_B	Input	CA ODT control: Used in conjunction with the mode register to turn on/off the on-die-termination for CA pins **ODT_CA_A and ODT_CA_B are not used in the ZLPDDR4 model**
ZQ	Input	Calibration reference: Used to calibrate the output drive strength and termination resistance. **ZQ is not used in the ZLPDDR4 model**

Note All differential signals are modeled as single-ended signals. Therefore on all differential signals available at a component's pinout, only xxx_t signals are really connected inside the memory model.

3.4 Differences with LPDDR4 SDRAM Models

3.4.1 LPDDR4 Device Interface Modifications

3.4.1.1 DQS_t_<A/B>_in/DQS_c_<A/B>_in and DQS_t_<A/B>_out/DQS_c_<A/B>_out Ports

The DQS_t and DQS_c bi-directional differential ports have been replaced by 4 unidirectional ports:

- DQS_t_<A/B>_in and DQS_c_<A/B>_in: inputs to the ZLPDDR4 model
- DQS_t_<A/B>_out and DQS_c_<A/B>_out: outputs from the ZLPDDR4 model

Since the DQS port is used to latch data, this modification allows avoiding gated clocks. For proper use, the original DQS bidirectional signal should be split into four unidirectional ports inside the LPDDR4 controller mapped in your design.

3.4.1.2 DQS_<A/B>_oe Output Enable Port

An additional output enable port, DQS_<A/B>_oe, is available to manage the direction of DQ and DQS bi-directional signals: DQS_<A/B>_oe=1 when DQ and DQS buses are driven by the memory

3.4.1.3 ODT_CA_<A/B> and ZQ Input Ports

The ODT_CA_<A/B> and ZQ input signal are JEDEC-compliant signal that are not used in the ZLPDDR4 interface.

3.4.2 LPDDR4 Operations

The ZLPDDR4 model is functionally equivalent to the LPDDR4 memory device, but timing requirements are not applicable. The ZLPDDR4 model is accurate up to a half cycle but cannot take into account setup and hold time for example.

All commands and operating modes (except Write Leveling, Bus training and MPC command) are accepted by the ZLPDDR4 model, including the programming of mode registers defining Read/Write latencies and burst lengths.

Refresh and self-refresh commands are ignored.

Refer to the reference LPDDR4 device datasheets for descriptions of correct operations of the LPDDR4 SDRAM.

3.4.3 LPDDR4 Timing Modeling in ZLPDDR4

3.4.3.1 Read Operations

The real Read latency seen on the component interface is different from the Read latency value (RLmrs) set in the mode registers. The difference is caused by the DQS output access time from CK_t (t_{DQSCK}), which is device-dependent:

$$\textbf{Data Read Latency} = \mathtt{RLmrs} + t_{DQSCK}$$

In order to model the behavior with DDR cycle accuracy, ZLPDDR4 models contain a specific mode register named `zebuReg[19:16]` (possible values: `0` to `15`) to control the t_{DQSCK} timing delay. A t_{DQSCK} unit is equivalent to a half-cycle of `CK_t_<A/B>` clock, in other words `0` means a `0 ns` delay and `15` means a delay equivalent to `7.5*CK_t_<A/B>` periods.

Programming information for `zebuReg` register is available in Section 3.5.

Example:

With a `CK_t_<A/B>` running at 800 MHz (1.25 ns) for the memory device, the t_{DQSCK} must be programmed as `zebuReg[19:16]=4h'3` to model a t_{DQSCK} equal to 1.875 ns.

Corresponding Tcl script for modification of the register from **zRun**:

```
ZEBU_Signal_write top.zlpDDR4.rank_0_ins_zebuMR.zebuReg 0x3XXXX %h
ZEBU_Signal_write top.zlpDDR4.rank_1_ins_zebuMR.zebuReg 0x3XXXX %h
ZEBU_Monitor_flush
```

3.4.3.2 Write operations

The LPDDR4 uses an unmatched `DQS DQ` path for lower power, so the first `DQ` data seen on the component interface is different from the Write latency value (`WLmrs`) set in the mode registers. The difference is caused by the `DQS` to `DQ` delay time (t_{DQS2DQ}), which is device-dependent:

Data Write Latency = `WLmrs` + t_{DQS2DQ}

In order to model the behavior with DDR cycle accuracy, ZLPDDR4 models contain a specific mode register named `zebuReg[23:20]` (possible values: `0` to `4`) to control the t_{DQS2DQ} timing delay. This indicates first valid edge (rising or falling) of `CK_t_<A/B>` for sampling first data `DQ`.

A t_{DQS2DQ} unit is equivalent to a half-cycle of `CK_t_<A/B>` clock, in other words `0` means a `0 ns` delay and `4` means a delay equivalent to `2*CK_t` periods.

Programming information for `zebuReg` register is available in Section 3.5.

Example:

With a `CK_t` running at 800 MHz (1.25 ns) for the memory device, the t_{DQS2DQ} must be programmed as `zebuReg[23:20]=3b'001` to model a t_{DQS2DQ} equal to 625 ps.

Corresponding Tcl script for modification of the register from **zRun**:

```
ZEBU_Signal_write top.lpDDR4.rank_0_ins_zebuMR.zebuReg 0x1XXXXX %h
ZEBU_Signal_write top.lpDDR4.rank_1_ins_zebuMR.zebuReg 0x1XXXXX %h
ZEBU_Monitor_flush
```

3.5 Configurable Mode Register

The ZLPDDR4 memory models have a common register (`zebuMR_common`) and a specific register (`zebuMR`) for each channel A and B for runtime control of the t_{DQSCK} and t_{DQS2DQ} values.

3.5.1 **`zebuMR_common`** Register

Each instance of the ZLPDDR4 model has a `zebuMR_common` register common to both channels of this instance. It is divided into several Mode Registers (`MR`), each one corresponding to specific information.

The following table describes the common `zebuMR_common` register content:

bits	23 16	15 8	7 0
	MR7[7:0]	MR6[7:0]	MR5[7:0]
default value	0x3	0x0	0xff
information	Revision ID2	Revision ID1	Manufacturer ID

Figure 2: `zebuMR_common` Mapping with Default Values

The path to the `zebuMR_common` register is:

```
<path_to_zlpddr4_inst>.ins_zebuMR_common.zebuReg[23:0]
```

3.5.2 Specific **`zeBuMR`** Register for Each Channel

Each instance of the ZLPDDR4 model has two `zebuMR` registers: one dedicated to channel A and one to channel B.

Each `zebuMR` is divided into several Mode Registers (`MR`), each one corresponding to specific information.

The following table describes the `zebuMR` register content for each channel:

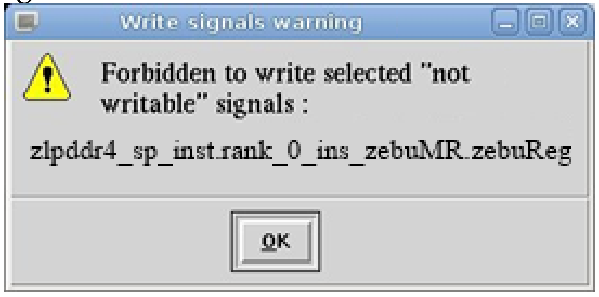
bits	23 20	19 16	15	14 13	12	11 5	4	3 0
	t_{DQS2DQ}	t_{DQSCK}		MR3[7:6]	MR3[1]	MR2[6:0]	MR1[7]	MR1[3:0]
default value	0x4	0xd	0	0x0	0x0	0x0	0x0	0x0
infor-mation	see Section 3.4.3		reserved	read/write dbi enable	Write post-amble length	Read and Write latencies	Read post-amble length	Burst length and Read/Write preamble

Figure 3: specific `zebuMR` mapping with default values

The paths to the `zebuMR` register for each channel are:

```
<path_to_zlpddr4_inst>.rank_0_ins_zebuMR.zebuReg[23:0]
<path_to_zlpddr4_inst>.rank_1_ins_zebuMR.zebuReg[23:0]
```

Note If the compilation settings for register write were not the one described in Section 4.2.2, the register write operation is not possible at runtime. The following message is displayed at runtime when attempting to write to a configuration register:



Forbidden to write selected "not writable" signals :

zlpddr4_sp_inst.rank_0_ins_zebuMR.zebuReg

Figure 4: Error Message in `zRun` for Forbidden Register-Write

3.6 zrm Address Translation

The memory space of the LPDDR4 device is three-dimensional and is organized in banks, rows and columns. In the ZLPDDR4 memory model, the memory space is flat (bank*row*column depth array of words).

The zrm address is decoded from `Bank`, `Row` and `Column` addressing of LPDDR4.

The `RBC_BRCn` input is available at the ZLPDDR4 interface to select the zrm memory array addressing mode at compilation time. Two modes are available:

- `BRC` for `{Bank,Row,Column}` addressing
- `RBC` for `{Row,Bank,Column}` addressing

In BRC mode, the starting address for zrm will be transformed into `{Bank, Row, Column}` where `Bank` is the most significant address bit.

In RBC mode, the starting address for zrm will be transformed into `{Row, Bank, Column}` where `Row` is the most significant address bit.

To change zrm addressing:

- `RBC_BRCn = 0` for `BRC` mode (default)
- `RBC_BRCn = 1` for `RBC` mode

3.6.1 For Binary Density Memory Models (4Gb, 8Gb, 16Gb Memory Models)

For example, if you want to read your memory in a design using the 4Gbx16 (2Gb/channel) ZLPDDR4 model with:

- `Bank=1(3bits)`
- `Row=1(14bits)`
- `Column=4(10bits)`

then the starting address for zrm (in hexadecimal) will be as follows:

- <u>BRC Mode</u>: (`RBC_BRCn = 0`)
  `zrm_addr[26:0] = {001,00000000000001,0000000100} = 27'h1000404`
- <u>RBC Mode</u>: (`RBC_BRCn = 1`)
  `zrm_addr[26:0] = {00000000000001,001,0000000100} = 27'h0002404`

3.6.2 For Non-Binary Density Memory Models (6Gb and 12Gb Memory Models)

The 6Gb and 12Gb ZeBu ZLPDDR4 memory models are non-binary density devices. As a consequence, only three quarters of the row address space is valid. When the MSB of `row` address bit is HIGH, the MSB-1 address bit must be LOW.

This has no impact in RBC mode. However in BRC mode, the zrm address will be converted to have a linear addressing as follows:

$$\begin{aligned} zrm\ address = {} & (bank\_addr \times MAX\_ROW\_SIZE \times MAX\_COL\_SIZE) \\ & + (row\_addr \times MAX\_COL\_SIZE) + column\_addr \end{aligned}$$

Example

If you want to read your memory in a design using the 6Gbx16 (3Gb/channel) ZLPDDR4 model with:

- `Bank=1(3bits)`
- `Row=1(15bits)`
- `Column=4(10bits)`
- `MAX_ROW_SIZE` = 24576 (2^{15}*3/4) for this model
- `MAX_COL_SIZE` = 1024 (2^{10}) for this model

then the starting address for zrm (in hexadecimal) will be as follows:

- BRC Mode: (`RBC_BRCn = 0`)
  `zrm_addr[27:0] = (1*24576*1024) + (1*1024) +4 = 28'h1800404`
- RBC Mode: (`RBC_BRCn = 1`)
  `zrm_addr[27:0] = {000000000000001,001,0000000100} = 28'h0002404`

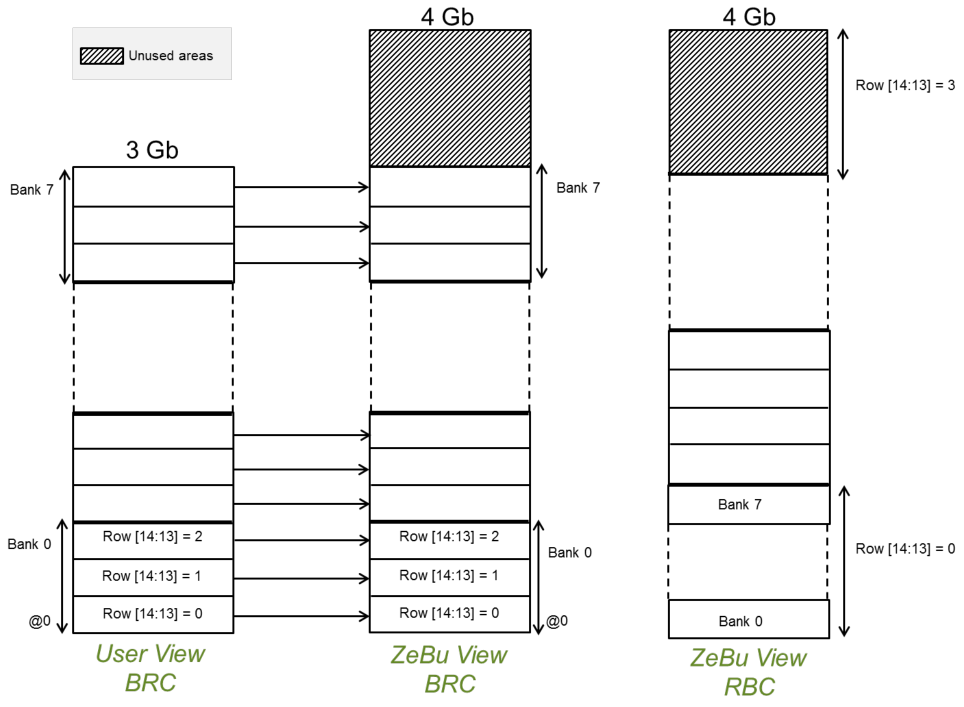


Figure 5: 6Gb ZLPDDR4 Models with 15-bit Row Address

4 Integration with the DUT

This section describes how to integrate the ZLPDDR4 memory model with the DUT.

You should first have properly set the RTL wrapper in your compilation project, as described in Section 4.1 below.

In this manual:

- `<hw_ip_path>` stands for `$ZEBU_IP_ROOT/HW_IP`
- `<ip_path>` stands for `<hw_ip_path>/ZLPDRR4.<version>`
- `<model_path>` stands for `<ip_path>/<size>/<arch>`
- `<xilinx_verilog>` stands for the Xilinx Verilog source. This path depends on the ZeBu release :
    - o prior to V6_3_2B: `$ZEBU_ROOT/ise_x_y`
    - o V6_3_2B: `$ZEBU_ROOT/ise13.4_patched/ISE_DS/ISE`
    - o V8_0_1 : `$ZEBU_ROOT/ZeBu_Vivado_2013.2_patched/data`

4.1 Setting the RTL Wrapper in the Compilation Project

4.1.1 Why Using an RTL Wrapper?

The ZLPDDR4 memory has a unidirectional interface with additional ports which are not part of the JEDEC standard. Then, in order to substitute a standard DRAM memory with a JEDEC-compliant bidirectional interface, the ZLPDDR4 memory model should be integrated with the DUT using a dedicated RTL wrapper.

The RTL wrapper for bidirectional `DQS` signal has two parameters:

- `USER_RBC_BRCn`: addressing mode. The parameter value should be 0 for BRC mode and 1 for RBC mode. Default value is 0.
- `USER_DQS_DELAY`: additional delay on `DQS` signal.

4.1.2 Wrappers and Associated Model Files Locations

Source code files for RTL wrappers of the ZLPDDR4 interface are available at `<model_path>/wrapper_rtl`.

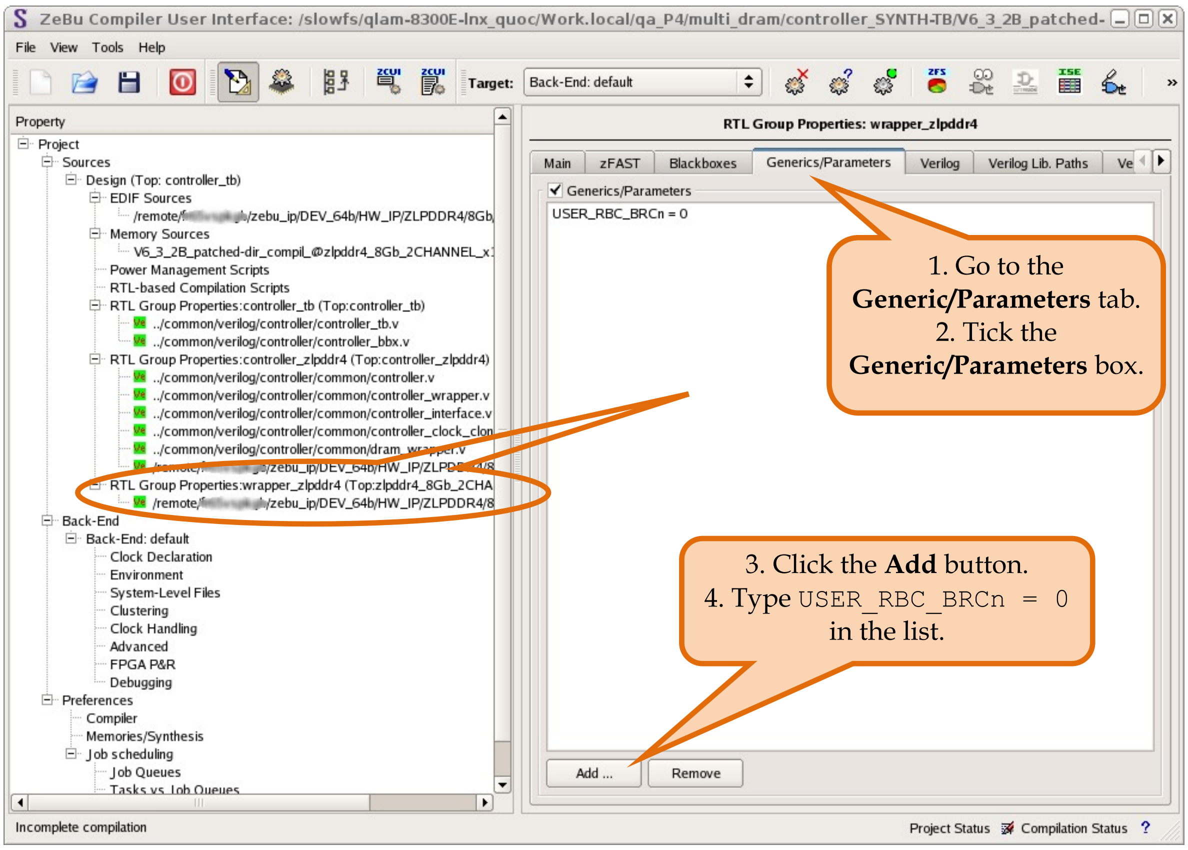
Associated blackbox description Verilog and VHDL files are available at `<model_path>/component`: `<model_name>_bidir.<v/vhd>`.

4.1.3 Integrating the RTL Wrapper with the DUT for ZeBu Compiler

This section describes how to add the RTL wrapper to an existing **`zCui`** compilation project. The EDIF netlist is available at `<model_path>/edif/<model_name>.edf`.

1. In the DUT, change the ZLPDDR4 model name to match the new bidirectional component name and modify the source pathname as follows:
   `<model_path>/component/<model_name>_bidir.<v/vhd>`.
2. Create a new dedicated RTL group (for example, `GRP_ZLPDDR4)` in the DUT group that contains the following RTL wrapper file:
   `<model_path>/wrapper_rtl/<model_name>_bidir.v`.

The assignment of the parameters for the RTL wrapper is made through the **Generic/Parameters** panel of **`zCui`** for the RTL group instantiating the DUT.


ZeBu Compiler User Interface: /slowfs/qlam-8300E-lnx_quoc/Work.local/qa_P4/multi_dram/controller_SYNTH-TB/V6_3_2B_patched-
File View Tools Help
Target: Back-End: default
RTL Group Properties: wrapper_zlpddr4
Main zFAST Blackboxes Generics/Parameters Verilog Verilog Lib. Paths
Generics/Parameters
RTL Group Properties:wrapper_zlpddr4 (Top:zlpddr4_8Gb_2CHA
Add ... Remove
1. Go to the Generic/Parameters tab.
2. Tick the Generic/Parameters box.
3. Click the Add button.
4. Type USER_RBC_BRCn = 0 in the list.
Incomplete compilation
Project Status Compilation Status

USER_RBC_BRCn = 0

Figure 6: RTL Wrapper Parameter Assignment in **`zCui`** Interface

4.2 Synthesizing and Compiling for ZeBu

4.2.1 Synthesis

During your design synthesis you must use a blackbox for the ZLPDDR4 component. The blackbox Verilog and VHDL files are available at `<model_path>/component` and integrated as described earlier in Sections 4.1.2 and 4.1.3.

4.2.2 Compilation

To compile the design for use with ZLPDDR4 memory models, you must add the appropriate encrypted ZLPDDR4 EDIF netlist in your system-level compiler script.

You can find the encrypted netlist at the following location:
`<model_path>/edif/ <model_name>.edf`

Notes

The two following compilation settings must also be checked to be sure that the Configurable Mode Register (see Section 3.5) can be written at runtime:

- The Enable BRAM Read&Write/Write Register/Save&Restore item in the Debugging tab of **`zCui`** is activated.
- The ZeBu ZLPDDR4 memory model logic is mapped on an FPGA where there is no RLDRAM instantiated (RLDRAM memories are present on 4C and 8C FPGA modules only). For a design instantiating RLDRAM memories in a 4C or 8C module, it is highly recommended to map the ZLPDDR4 model in the ZeBu memory server FPGA (F4 FPGA for the 4C module, F8 FPGA for the 8C module). For that purpose, manual mapping commands should be added for the design compilation in **`zCui`**.

Warning No message is displayed during compilation but the register write operation is not possible at runtime.

4.3 Simulation

The ZLPDDR4 package is supplied with:

- a `libIpSimu_<32/64>.so` library in the `<hw_ip_path>/lib` directory for simulation purposes
- a gate-level Verilog simulation model in an encrypted format with encryption depending on the target HDL simulator. It is available in the `<model_path>/simu/gate` directory
- an RTL-level Verilog simulation model in an encrypted format with encryption depending on the target HDL simulator. It is available in the `<model_path>/simu/rtl` directory.

4.3.1 Simulation with Gate-Level Model

To simulate your design with the ZLPDDR4 component:

1. Compile the provided ZLPDDR4 gate-level simulation model found in `<model_path>/simu/gate`.
2. Instantiate a `glbl` Xilinx component in your top-level module.
3. Include the following Xilinx HDL simulation model library in your simulation environment: Unisims for ISE 9.1 and later

4.3.1.1 Gate-Level Simulation with Synopsys VCS

If your design includes SystemVerilog source files, it is recommended to add the following lines mentioning ZLPDDR4 at the end of the script:

```
vcs +v2k <my_options> <file_list> -sverilog
 <model_path>/simu/gate/vcs/<model_name>.vp
 <hw_ip_path>/lib/libIpSimu /libIpSimu_<32/64>.so
 <xilinx_verilog>/verilog/src/glbl.v
 -y <xilinx_verilog>/verilog/src/unisims +libext+.v
```

If you need to use several different ZLPDDR4 models, you should compile each one separately as VCS does not support multiple compilations of the same sub-modules in the same command line.

Example:

```
vlogan +v2k -sverilog
 <model_path>/simu/gate/vcs/<model_name>.vp
 <xilinx_verilog>/verilog/src/glbl.v
 -y <xilinx_verilog>/verilog/src/unisims +libext+.v

vlogan +v2k -sverilog
 <model_path>/simu/gate/vcs/<model_name>.vp
 <xilinx_verilog>/verilog/src/glbl.v
 -y <xilinx_verilog>/verilog/src/unisims +libext+.v

vcs <my_options> <top_level_name> <hw_ip_path>/lib/libIpSimu_<32/64>.so
```

Otherwise, you would get the following error message:

```
Error-[MPD] Module previously declared
```

4.3.1.2 Gate-Level Simulation with MTI ModelSim

```
vlog -sv
 <model_path>/simu/gate/mti/<model_name>.vp
 <xilinx_verilog>/verilog/src/glbl.v
 -y <xilinx_verilog>/verilog/src/unisims +libext+.v
vsim -sv_lib
 <hw_ip_path>/lib/libIpSimu_<32/64> <my_options>
```

4.3.1.3 Gate-Level Simulation with Cadence NCSim

```
ncvlog
 <xilinx_verilog>/verilog/src/unisims/*.v
ncvlog -sv
 <model_path>/simu/gate/ncsim/<model_name>.vp
 <xilinx_verilog>/verilog/src/glbl.v
ncelab
 <my_options>
 <my_top_level_name>
ncsim
 -sv_lib <hw_ip_path>/lib/libIpSimu_<32/64>
 <my_options>
 <my_top_level_name>
```

4.3.2 Simulation with RTL-Level Model

To simulate your design with the ZLPDDR4 component:

1. Compile the provided ZLPDDR4 RTL-level simulation model found in `<model_path>/simu/rtl`.
2. Instantiate a `glbl` Xilinx component in your top-level module.
3. Include the following Xilinx HDL simulation model library in your simulation environment: Unisims for ISE 9.1 and later

4.3.2.1 RTL-Level Simulation with Synopsys VCS

If your design includes SystemVerilog source files, it is recommended to add the following lines mentioning ZLPDDR4 at the end of the script:

```
vcs +v2k <my_options> <file_list> -sverilog
 <model_path>/simu/rtl/vcs/<model_name>.vp
 <hw_ip_path>/lib/libIpSimu /libIpSimu_<32/64>.so
 <xilinx_verilog>/verilog/src/glbl.v
 -y <xilinx_verilog>/verilog/src/unisims +libext+.v
```

If you need to use several different ZLPDDR4 models, you should compile each one separately as VCS does not support multiple compilations of the same sub-modules in the same command line.

Example:

```
vlogan +v2k -sverilog
 <model_path>/simu/rtl/vcs/<model_name>.vp
 <xilinx_verilog>/verilog/src/glbl.v
 -y <xilinx_verilog>/verilog/src/unisims +libext+.v

vlogan +v2k -sverilog
 <model_path>/simu/rtl/vcs/<model_name>.vp
 <xilinx_verilog>/verilog/src/glbl.v
 -y <xilinx_verilog>/verilog/src/unisims +libext+.v

vcs <my_options> <my_top_level_name> <hw_ip_path>/lib/libIpSimu_<32/64>.so
```

Otherwise, you would get the following error message:

```
Error-[MPD] Module previously declared
```

4.3.2.2 RTL-Level Simulation with MTI ModelSim

```
vlog -sv
 <model_path>/simu/rtl/mti/<model_name>.vp
 <xilinx_verilog>/verilog/src/glbl.v
 -y <xilinx_verilog> /verilog/src/unisims +libext+.v
vsim
 -sv_lib <hw_ip_path>/lib/libIpSimu_<32/64> <my_options>
```

4.3.2.3 RTL-Level Simulation with Cadence NCSim

```
ncvlog
 <xilinx_verilog>/verilog/src/unisims/*.v
ncvlog -sv
 <model_path>/simu/rtl/ncsim/<model_name>.vp
 <xilinx_verilog>/verilog/src/glbl.v
ncelab
 <my_options>
 <my_top_level_name>
```

```
ncsim
 -sv_lib <hw_ip_path>/lib/libIpSimu_<32/64>
 <my_options>
 <my_top_level_name>
```

4.3.3 Result of ZeBu IP License Checking

For simulation with VCS, ModelSim or NCSim, you should get the following (according to the model) when the license check is successful:

```
---------- At time 5.0 ps Testing license ----------
      ##############################################
      #         Copyright (c) 2005-2014            #
      # Emulation and Verification Engineering  SA #
      #--------------------------------------------#
      # ZeBu libIpSimu                             #
      # revision : 2.2 64 bit                      #
      # date : Fri 28 3 2014 - 12:50:08            #
      ##############################################
Testing ZLPDDR4 8192Mb ZeBu IP license

Checking out ZLPDDR4 8192Mb license

---------- At time 5.0 ps check license OK ----------
```

Otherwise, please contact your local representative.

5 Accessing ZLPDDR4 Models (zrm Load & Dump)

5.1 Running on ZeBu

To access the content of the memory at runtime, you can use the standard way to read from or write to ZeBu memories with its appropriate hierarchical path:

```
<path_to_zLPDDR4_mem_A>=<path_to_zlpddr4_inst>.rank_0_mem_core_logic
<path_to_zLPDDR4_mem_B>=<path_to_zlpddr4_inst>.rank_1_mem_core_logic
```

Example:

You want to initialize a memory in a design using a 4Gb(x16) ZLPDDR4 model with the `memory.init` content file. If the path to the ZLPDDR4 instance is `Top_dut.my_zlpddr4_ins`, then the full path to the zrm memory would be:

```
<path_to_zlpddr4_mem_A>=Top_dut.my_zlpddr4_ins.rank_0_mem_core_logic
<path_to_zlpddr4_mem_B>=Top_dut.my_zlpddr4_ins.rank_1_mem_core_logic
```

Warning Another memory named `mem_core_sp` exists: it is the physical view of the memory array. **Do not use it.**

Therefore you have to add the following lines in specific files:

- In a `designFeatures` file (default mode for synthesizable testbenches):

  ```
  $memoryInitDB = "init_mem"
  ```

  where `init_mem` is a file consisting of a collection of lines, with each line listing a memory and the corresponding content file name. In this example, the content of the `init_mem` file is:

  ```
  <path_to_zlpddr4_mem_A> memory.init
  <path_to_zlpddr4_mem_B> memory.init
  ```

- In a C++ co-simulation testbench:

  ```
  my_memory_A = my_board->getMemory("<path_to_zlpddr4_mem_A>");
  my_memory_A->loadFrom("memory.init");
  my_memory_B = my_board->getMemory("<path_to_zlpddr4_mem_B>");
  my_memory_B->loadFrom("memory.init");
  ```

- In a Verilog HDL co-simulation testbench:

  ```
  $ZEBU_readmem("memory.init", "<path_to_zlpddr4_mem_A>");
  $ZEBU_readmem("memory.init", "<path_to_zlpddr4_mem_B>");
  ```

5.2 Initializing Memories for Simulation

In a Verilog simulation testbench, there are two methods for memory initialization. These methods can only be applied to ZLPDDR4 models simulated at gate or RTL level.

The following displayed message at the beginning of the simulation will be helpful to retrieve the exact memory path:

```
-----The logical memory array path is tb.ins_zlpddr4.zlpddr4.rank_0_mem_core_sp.mem_logical
(width = 16, depth = 268435456, size = 4Gb, expansion = 4)
-----The physical memory array path is tb.ins_zlpddr4.zlpddr4.rank_0_mem_core_sp.mem
(width = 64, depth = 67108864, size = 4Gb)
----- The logical memory array path is tb.ins_zlpddr4.zlpddr4.rank_1_mem_core_sp.mem_logical
(width = 16, depth = 268435456, size = 4Gb, expansion = 4)
----- The physical memory array path is tb.ins_zlpddr4.zlpddr4.rank_1_mem_core_sp.mem
(width = 64, depth = 67108864, size = 4Gb)
```

5.2.1 Using Verilog System Tasks

These methods can be only used to access the physical view of the ZLPDDR4 memory model:

```
$readmemh("data0.hex",
   "<path_to_zlpddr4_inst>.rank_0_mem_core_sp.mem"[,start@,
stop@]);

$writememh("dumpdata0.hex"
   "<path_to_zlpddr4_inst>.rank_1_mem_core_sp.mem"[,start@,
stop@]);
```

Note With this method, you can load and dump the physical views of memory arrays using multiple files (by calling `$readmemh` and `$writememh` system functions with different file names).

5.2.2 Using Specific Verilog Tasks

These methods are defined in each memory array and are used to access the logical view of the ZLPDDR4 memory model:

```
<path_zlpddr4_inst>.rank_0_mem_core_sp.zip_readmemh(“file_name”,sta
rt@,stop@)
<path_zlpddr4_inst>.rank_1_mem_core_sp.zip_readmemh(“file_name”,sta
rt@,stop@)
```

OR

```
<path_zlpddr4_inst>.rank_0_mem_core_sp.zip_writememh(“file_name”,st
art@,stop@)
<path_zlpddr4_inst>.rank_1_mem_core_sp.zip_writememh(“file_name”,st
art@,stop@)
```

Note Due to the VCS® and MTI ModelSim® simulators' limitation on the maximum size of the array, the logical view is divided into parts of the 1G word which size depends on the `DQ` signal. Besides, if the start and stop addresses access different banks, an error occurs and an error message is displayed.

6 Debug Information

For debugging purposes, a list of signals is available at runtime to trace the internal behavior of ZLPDDR4 models (see Section 6.1).

Besides, an alias file for Verdi is also provided in the memory model package. It aims at facilitating decoding these signals (see Section 6.2).

6.1 `probe` Signals

A trace vector called `probe[97:0]` is available on interface and can be accessed with dynamic probes at ZeBu runtime. It contains the necessary information to analyze the behavior of your memory models on 2 channels.

Table 6: `probe` Signals

Signal		Description
Channel A	Channel B	
ZLPDDR4 Protocol checker		
`probe[52:45]`		IP version
`probe[44]`	`probe[97]`	*Reserved*
`probe[43]`	`probe[96]`	Illegal command according to JEDEC and ignored by model
`probe[42]`	`probe[95]`	Illegal command according to JEDEC but accepted by model
`probe[41]`	`probe[94]`	Row addressing error detected
`probe[40]`	`probe[93]`	Column addressing error detected
`probe[39]`	`probe[92]`	Command/feature currently unsupported
ZLPDDR4 Debug Information		
`probe[38:33]`	`probe[91:86]`	Real read latency (`RLmrs + tDQSCK`)
`probe[32:27]`	`probe[85:80]`	Real write latency (`WLmrs + tDQS2DQ`)
`probe[26]`	`probe[79]`	Set to 1 when burst length 16 used for current operation
`probe[25:22]`	`probe[78:75]`	Read/Write latency set (`MR2[6:0]`)
`probe[21]`	`probe[74]`	DBI read enable (`MR3[6]`)
`probe[20:18]`	`probe[73:71]`	Read/Write latency set (`MR2[6:0]`)
`probe[17:16]`	`probe[70:69]`	Burst length (`MR1[1:0]`)
`probe[15]`	`probe[68]`	Frequency Set Point Operation mode (`MR13[7]`)
`probe[14]`	`probe[67]`	Frequency Set Point Write enable (`MR13[6]`)
`probe[13]`	`probe[66]`	Data mask disable (`MR13[5]`)
`probe[12]`	`probe[65]`	DBI write enable (`MR3[7]`)
`probe[11]`	`probe[64]`	Read pre-amble type (`MR1[3]`)
`probe[10]`	`probe[63]`	Read post-amble length (`MR1[7]`)
`probe[9]`	`probe[62]`	Command Read valid
`probe[8]`	`probe[61]`	Command Write valid
`probe[7]`	`probe[60]`	Command Active valid
`probe[6]`	`probe[59]`	Command MRR valid

Signal		Description
Channel A	Channel B	
`probe[5]`	`probe[58]`	Command MRW valid
`probe[4]`	`probe[57]`	Command Precharge valid
`probe[3]`	`probe[56]`	Command MPC valid
`probe[2:0]`	`probe[55:53]`	Current state of ZLPDDR4

In conjunction with this probe vector, the whole interface of the zrm memories is fully visible at runtime to trace the real operations performed on the memory array. The full pathname of the memory to trace should be similar to:

```
<path_to_zlpddr4_mem> = top_dut.my_zlpddr4_ins.rank_0_mem_core_sp
<path_to_zlpddr4_mem> = top_dut.my_zlpddr4_ins.rank_1_mem_core_sp
```

Note The `probe[]` signals and the zrm memory waveforms provide enough information for efficient support of ZLPDDR4 behavior and integration issues.

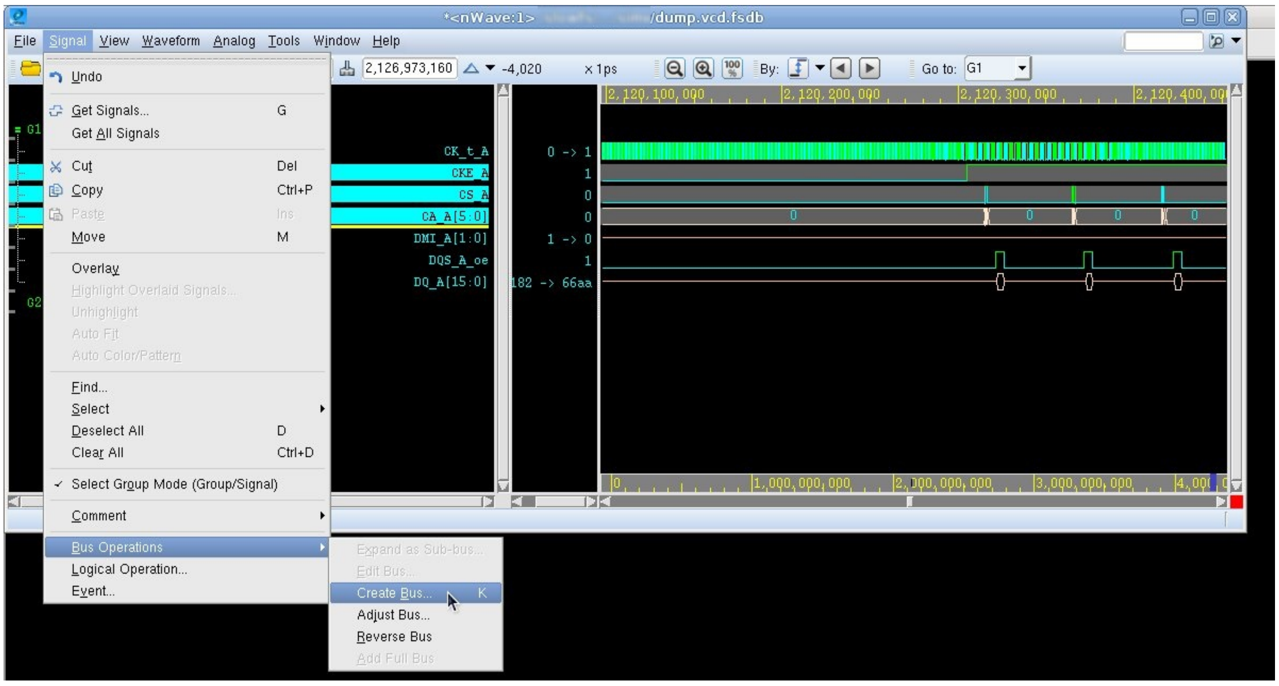
6.2 Alias Files for Verdi™

The ZLPDDR4 package provides two alias files in the `script/nWave` directory to facilitate debug in Verdi:

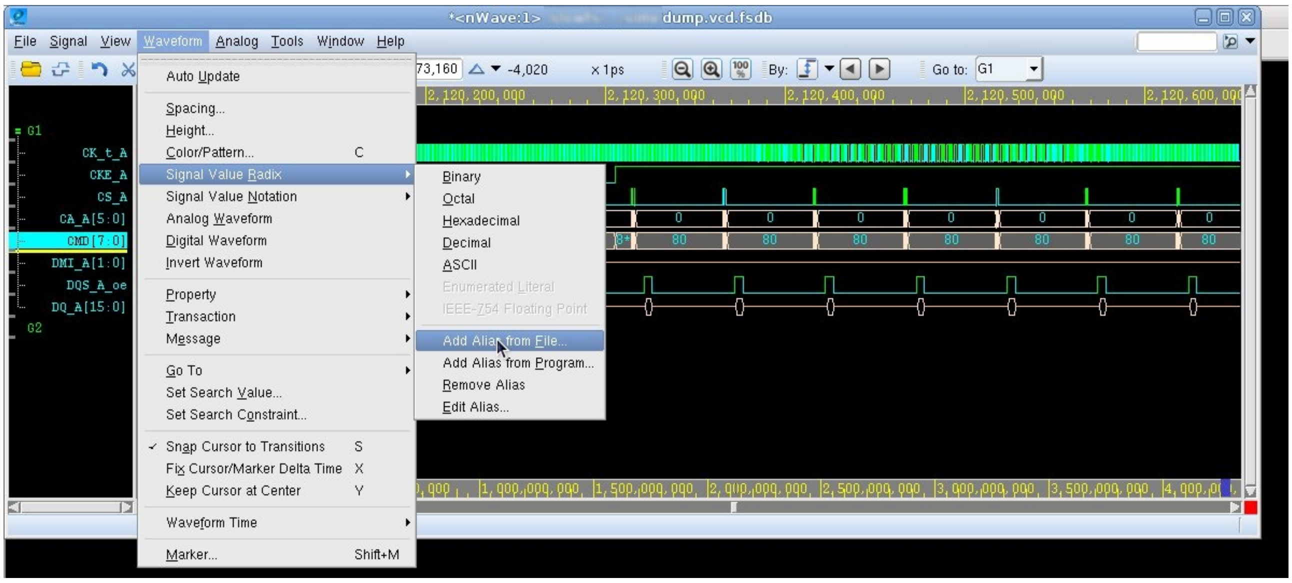
- `cmd.alias` automatically interpret the memory interface signals
- `probe.alias` automatically interpret the `probe[]` vector values

6.2.1 Using `cmd.alias`

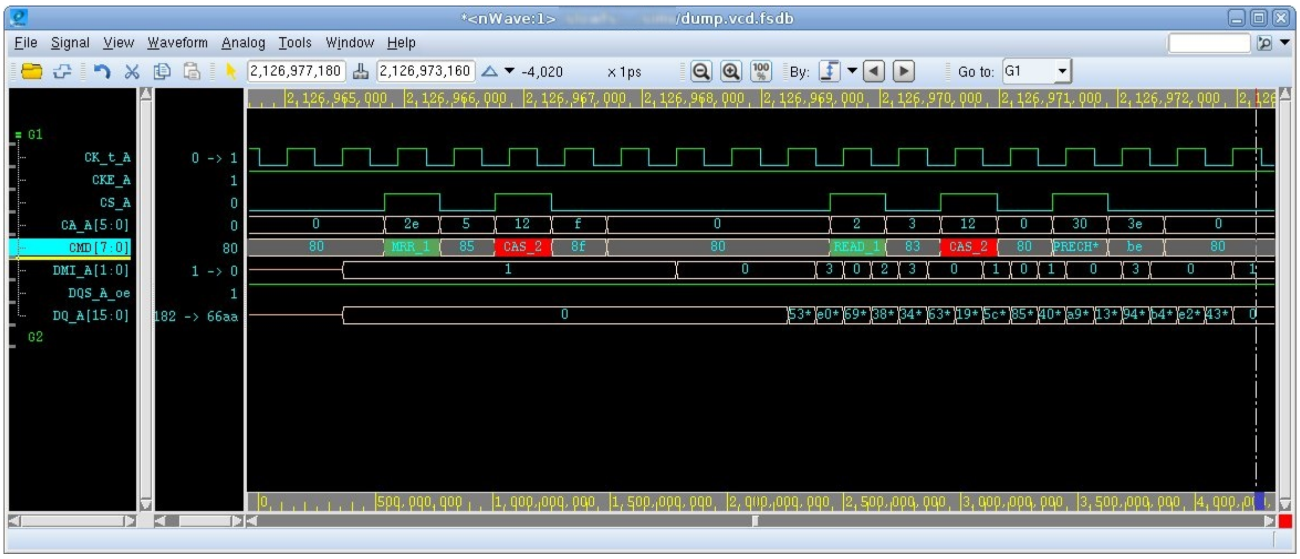
1. In **`nWave`**, select **Signal > Bus Operations > Create Bus** and create a new 8-bit width bus:
   - from `CKE_<A/B>`, `CS_<A/B>` and `CA_<A/B>[5:0]` in MSB to LSB
   - name it `CMD` for example



2. Select the created bus vector (named `CMD` in the previous step).
3. Assign the `cmd.alias` file to it by selecting **Waveforms** > **Signal Value Radix** > **Add Alias from File**:



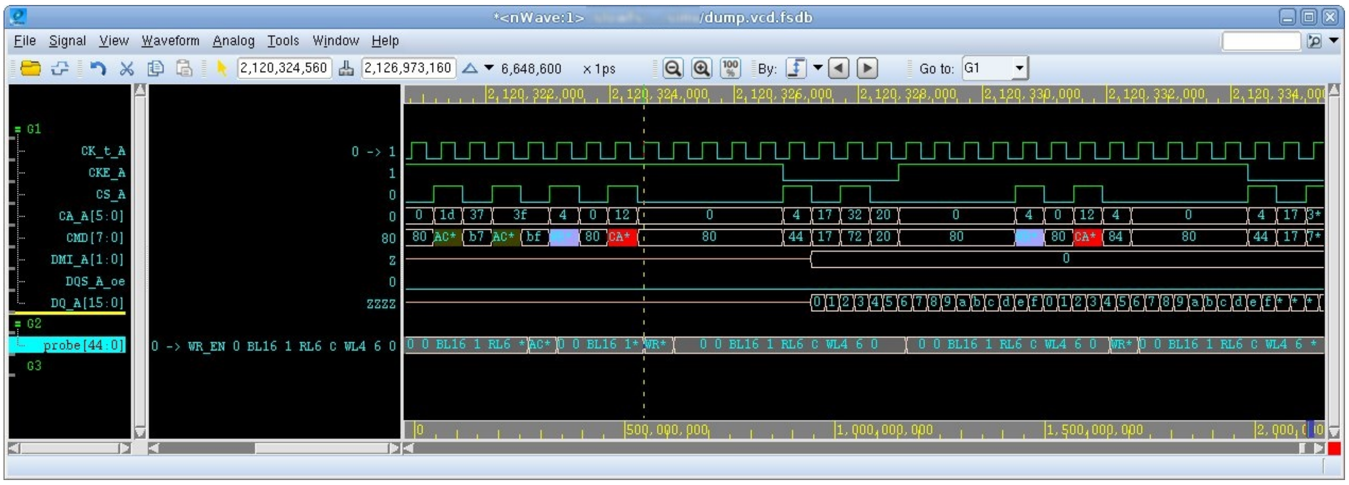
You should get a display result similar to the one below:



6.2.2 Using `probe.alias`

1. In **nWave**, select **Signal** > **Bus Operations** > **Create Bus** and create a new 45-bit width bus from `probe[97:0]`:
   - o `probe[44:0]` for channel A
   - o `probe[97:53]` for channel B
2. Select the created bus vector.
3. Assign the `probe.alias` file to it by selecting **Waveforms** > **Signal Value Radix** > **Add Alias from File.**

You should get a display result similar to the one below:



7 Examples

The ZLPDDR4 package provides waveform example and a tutorial.

7.1 Waveform Example

The memory model package provides an example of waveform file in `.fsdb` format. It can be open with **`nWave`**.

This waveform file comes from the simulation executed as an example in the tutorial of this chapter (see Section 7.2 hereafter).

The waveform file provided in the `example/waveform` directory.

7.2 Tutorial

This tutorial shows how to use the ZLPDDR4 Memory Models in the two following situations:

- HDL Simulation
- Emulation on Zebu with **`zRun`** and user-defined Tcl script

7.2.1 Tutorial's Files

The provided files for this tutorial are available in the `example` directory as shown below:

```
$ZEBU_IP_ROOT
`-- HW_IP
    `-- ZLPDDR4.<version>
        `-- example
            |-- simu
            |   |-- src
            |   |   |-- bench
            |   |   |   `-- demo_tb.v
            |   |   |-- compil
            |   |   |   `-- pattern.tcl
            |   |   |-- dut
            |   |   |   `-- demo.v
            |   |   |-- run
            |   |   |   `-- pattern.txt
            |   |   `-- mk
            |   |       `-- Makefile-vcs-gate.mk
            |   `-- Makefile
```

```
`-- zebu
    |-- src
    |   |-- compil
    |   |   |-- clock_zRun.dve
    |   |   `-- demo.zpf
    |   |-- dut
    |   |   `-- demo.edf.gz
    |   |-- run
    |   |   |-- designFeatures
    |   |   |-- pattern.mem
    |   |   `-- zRun.tcl
    |   `-- mk
    |       `-- Makefile-zRun.mk
    `-- Makefile
```

7.2.1.1 Files Used for HDL Simulation

The following files of the `example/simu` directory shown above are used in case of HDL Simulation:

- testbench files: `simu/src/dut/demo.v` and `simu/src/bench/demo_tb.v`
- compilation files: `simu/src/compil/pattern.tcl`
- run files: `simu/src/run/pattern.txt` (memory settings)
- automatic flow: `Makefile` and `simu/src/mk/Makefile-vcs-gate.mk`

7.2.1.2 Files Used for Emulation on ZeBu with `zRun` and User-Defined Tcl Script

The following files of the `example/zebu` directory shown above are used in case of emulation on ZeBu with **`zRun`** and user-defined Tcl script:

- testbench files: `zebu/src/dut/demo.edf.gz`
- compilation files:
    - `zebu/src/compil/clock_zRun.dve`
    - `zebu/src/compil/demo.zpf`
    - `simu/src/compil/pattern.tcl`
- run files:
    - `zebu/src/run/designFeatures`
    - `simu/src/run/pattern.txt` (memory settings)
    - `zebu/src/run/pattern.mem` (memory settings)
    - `zebu/src/run/zRun.tcl`
- automatic flow: `Makefile` and `zebu/src/mk/Makefile-zRun.mk`

7.2.2 Description

7.2.2.1 Overview

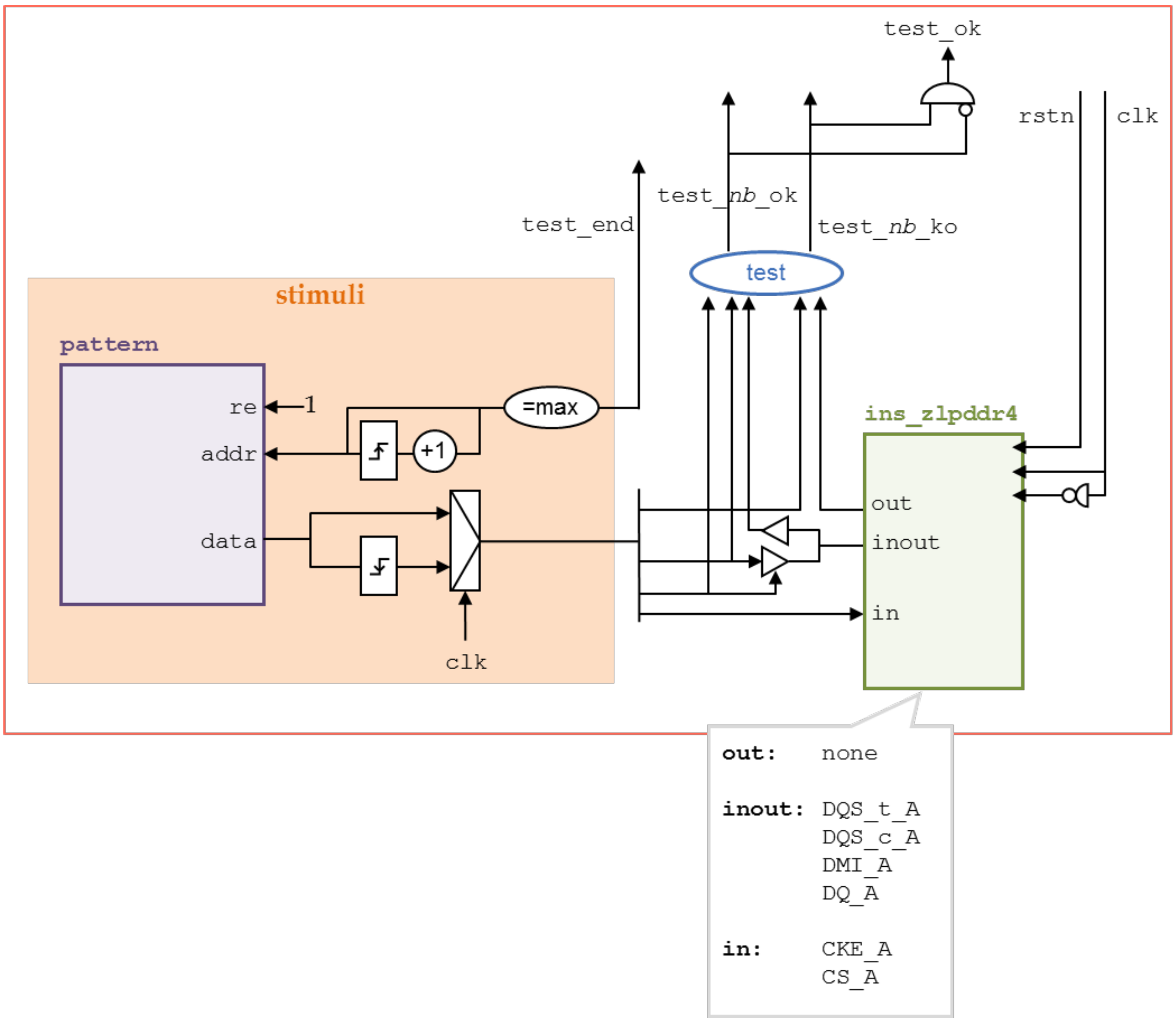


Figure 7 : Tutorial DUT overview

In the figure above, the DUT instances:

- a ZLPDDR4 Memory Models (`ins_zlpddr4`)
- a ZeBu Memory (`pattern`) which contains commands and data for the ZLPDDR4 Memory.

To validate the correctness of the ZLPDDR4 instance, read data are compared to expected data stored in the `pattern` memory.

The DUT have two outputs:

- `test_end` is set when the end of the tutorial is reached
- `test_ok` is set when no error is detected.

7.2.2.2 Content

The pattern performs the following operations:

1. It configures the ZLPDDR4 model with minimal latency and no Data Bus Inversion (Mode Register 1, 2 and 3)
2. It executes the Active/Write/Read/Mask Write/Read/Precharge sequence with a Burst length of 16 words
3. It executes this sequence with a Burst length of 32 words
4. It configures the ZLPDDR4 with maximal latency and Data Bus Inversion (Mode Register 1, 2 and 3)
5. It executes the sequence again with a Burst length of 16 words
6. It executes the sequence once more with a Burst length at 32 words

In addition, the `zebuMR` register is also set:

- in the `simu/src/bench/demo_tb.v` file for HDL simulation
- in the `simu/src/run/zRun.tcl` file for emulation with **`zRun`** and user-defined Tcl script

7.2.3 Running the Tutorial Examples

7.2.3.1 Setting the Environment

Make sure the following environment variables are set correctly before running the tutorial examples:

- `ZEBU_ROOT` must be set to a valid ZeBu installation.
- `ZEBU_IP_ROOT` must be set to the package installation directory.
- `FILE_CONF` must be set to your system architecture file (for example, on a ZeBu Server-1 system: `../config/zse_configuration`)
- `REMOTECMD` can be specified if you want to use remote synthesis and remote ZeBu jobs.
- `ZEBU_XIL` or `ZEBU_XIL_VIVADO` must be set to a valid ISE installation (the script default value for the ISE installation directory is `$ZEBU_ROOT/zebu_env.bash`)

7.2.3.2 Running HDL Simulation

The HDL simulation is performed by Synopsys VCS simulator and for the gate level model.

To compile and run the example:

1. Go to the `example/simu` directory.
2. Launch the compilation flow with the `Makefile`:
```
make compil
```
3. Launch the emulation flow with the `Makefile`:
```
make run
```

4. Optionally, clean the compilation and run directory (the working directory automatically created at compilation and run is removed):
   ```
   make clean
   ```

7.2.3.3 Running Emulation with `zRun` and User-Defined Tcl Script

To compile and run the example:

1. Select the synthesis tool of your choice by either:
   - changing the selected synthesis tool in **`zCui`** graphical interface
   - changing the `SYNTH_TOOLS` environment variable
2. Go to the `example/zebu` directory
3. Launch the compilation flow with the `Makefile`:

`make compil`	without **`zCui`** Graphical User Interface
`make compil_gui`	with **`zCui`** Graphical User Interface

4. Launch the emulation flow with the `Makefile`:

`make run`	without **`zRun`** Graphical User Interface
`make run_gui`	with **`zRun`** Graphical User Interface

5. Optionally, clean the compilation and run directory (the working directory automatically created at compilation and run is removed):
   ```
   make clean
   ```