ZeBu LPDDR4 Memory Model User Manual

Version V-2024.03-SP1, February 2025

Feedback

Contents

Feedback

1 About This Manual

Overview

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

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, see *ZeBu Compilation Manual* and the ZeBu *zFAST Synthesizer Manual*.

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

Synopsys Statement on Inclusivity and Diversity

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

2

Introduction

This section explains the following topics:

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.

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*).

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 blackbox components located in the *component* directory.
- Provides bidirectional Verilog or VHDL wrapper files to include the ZLPDDR4 memory model within the user DUT, located in the *wrapper_rtl* directory.
- Provides analyzer feature, that logs memory model activity, decoded command and data payload.

ZLPDDR4 Library

ZeBu ZLPDDR4 memory models are available for 4Gb, 6Gb, 8Gb, 12Gb and 16Gb memory capacity components on any compatible ZeBu systems.

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*

ZLPDDR4 Capacity with ZeBu

Up to 8 ZLPDDR4 memory model instances are supported on any compatible ZeBu systems. This value is the same for ZLPDDR4_DIMM memory models.

Performance

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

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	**driver Clk**	**ZLPDDR4 clock (*CK_t_A*/*CK_t_B*)**	***zTime* report (for *driverClk*)**
ZeBu Server-1 (DDR3 Mem Server)	25 MHz	12.5 MHz	5.8 MHz
ZeBu Server-3 (DDR3 Mem Server)	10 MHz	5 MHz	5.0 MHz

Note:

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

Limitations

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

- Post-package repair (*MR4[4]*)
- The Refresh and Self Refresh operations are ignored by the ZLPDDR4 model.

Feedback

3

Installation

This section explains the following topics:

- Installing the ZLPDDR4 Package
- Uninstalling the ZLPDDR4 Package

Installing the ZLPDDR4 Package

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. Make sure you have WRITE permissions on the IP directory and on the current directory.
3. 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 process is complete and successful when the following message is displayed:

```
<IP>.<VERSION> successfully installed.
```

The memory models are installed under *$ZEBU_IP_ROOT/HW_IP* sub-directory. This sub-directory is automatically created when necessary.

If an error occurred during the installation, a message is displayed to point out the error. Here is an error message example:

```
ERROR: /auto/path/directory is not a valid directory.
```

Package Structure and Content

After the ZLPDDR4 memory model package has been correctly installed, it provides the following elements:

- under *$ZEBU_IP_ROOT/HW_IP/<IP>.<version>*

:

lib directory	Compressed *.so* libraries (*.gz*) of the memory models for simulation.
script directory	User scripts for Verdi®.
example directory	HDL simulation and emulation example files described in the Chapter 7.
doc directory	This manual.

- for each memory model capacity (<*size*>) and architecture (<*archi*>) under *$ZEBU_IP_ROOT/HW_IP/<IP>.<version>/<size>/<archi>*:

component directory	Memory model component black box for the design synthesis.
simu directory	*/gate*: Verilog gate-level netlists for model simulation.*/rtl*: Verilog level simulation model.
wrapper_rtl directory	Wrappers to include the component in the DUT as Verilog file.

During installation, symbolic links are created for an easy access from all ZeBu tools or simulation tools in:

- *$ZEBU_IP_ROOT/HW_IP/<IP>*: accesses the latest package of the memory models.
- *$ZEBU_IP_ROOT/HW_IP/lib*: accesses the latest *libIpSimu_<32/64>.so* files.

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.

4

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).

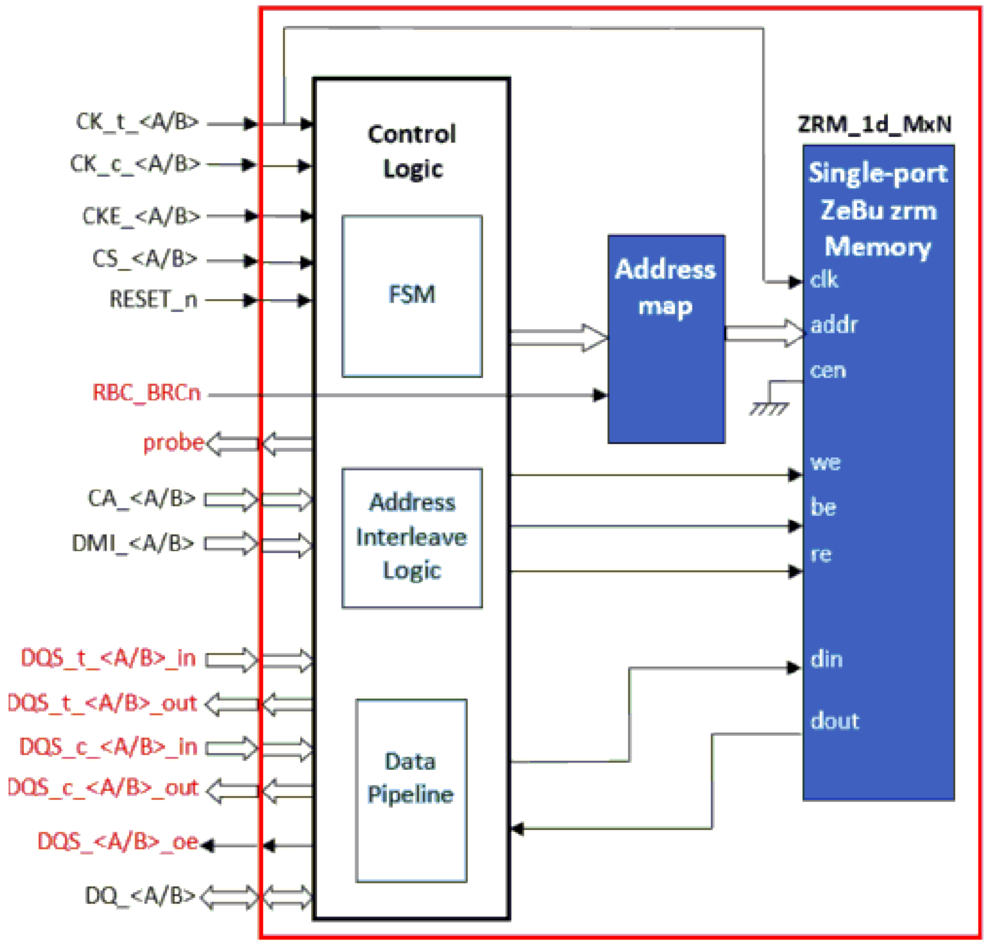
Overview

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.

Functional Block Diagram

The ZLPDDR4 synthesizable ZeBu memory models are architectured as follows:

Figure 1 ZLPDDR4 Model Architecture



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.).*

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

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

Table 4 ZLPDDR4 Unidirectional Interface (Continued)

Name	Type	Description
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
for further details.	probe	Output
Diagnostic port.	Please refer to Chapter	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

Table 5 ZLPDDR4 Bi-directional Interface (Continued)

Name	Type	Description
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.

Table 5 ZLPDDR4 Bi-directional Interface (Continued)

Name	**Type**	**Description**
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.

Differences with LPDDR4 SDRAM Models

LPDDR4 Device Interface Modifications

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.

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.

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.

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.

LPDDR4 Timing Modeling in ZLPDDR4

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* (*tDQSCK*), which is device-dependent:

Data Read Latency = RLmrs + tDQSCK

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 *tDQSCK* timing delay. A *tDQSCK* 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

Example:

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

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

```
force top.zlpDDR4.rank[0].ins_zebuMR.zebuReg 0x3XXXX -radix hexa -deposit
force top.zlpDDR4.rank[1].ins_zebuMR.zebuReg 0x3XXXX -radix hexa -deposit
```

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 (*tDQS2DQ*), which is device-dependent:

Data Write Latency = *WLmrs* + *tDQS2DQ*

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 *tDQS2DQ* timing delay. This indicates first valid edge (rising or falling) of *CK_t_<A/B>* for sampling first data *DQ*.

A *tDQS2DQ* 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 *tDQS2DQ* must be programmed as *zebuReg[23:20]=3b'001* to model a *tDQS2DQ* equal to 625 ps.

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

```
force top.zlpDDR4.rank[0].ins_zebuMR.zebuReg 0x3XXXX -radix hexa -deposit
force top.zlpDDR4.rank[1].ins_zebuMR.zebuReg 0x3XXXX -radix hexa -deposit
```

Configurable Mode Register

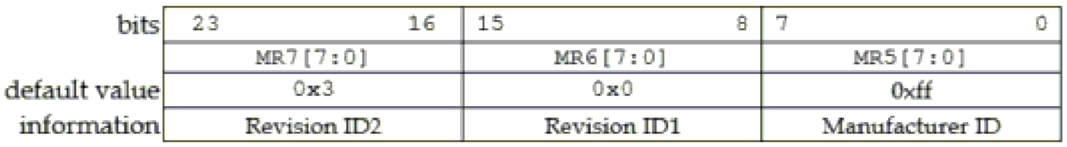
The ZLPDDR4 memory models have a common register (*zebuMR_common*) and a specific register (*zebuMR*) for each channel A and B for runtime control of the *tDQSCK* and *tDQS2DQ* values.

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:

Figure 2 zebuMR_common Mapping with Default Values



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

The path to the *zebuMR_common* register is:

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

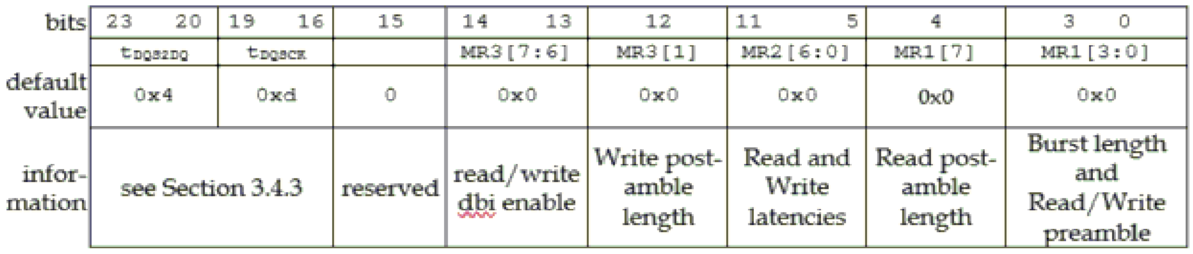
Specific *zeBuMR* Register for Each Channel

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:

Figure 3 Specific zebuMR mapping with default values



bits	23 … 20	19 … 16	15	14 … 13	12	11 … 5	4	3 … 0
	tDQS2DQ	tDQSCK		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

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]
```

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.

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

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:

- BRC Mode: (RBC_BRCn = 0)

  ```
  zrm_addr[26:0] = {001,00000000000001,0000000100} = 27'h1000404
  ```

- RBC Mode: (RBC_BRCn = 1)

  ```
  zrm_addr[26:0] = {00000000000001,001,0000000100} = 27'h0002404
  ```

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)

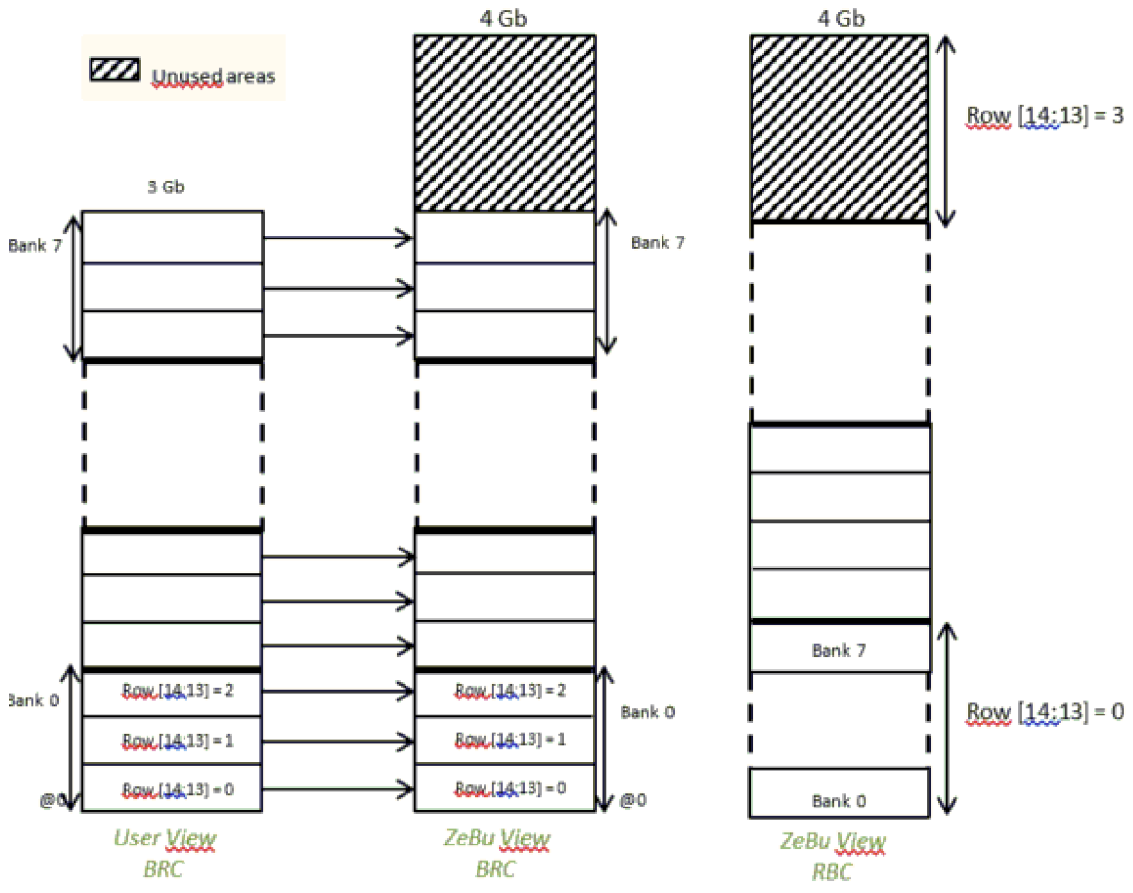
  ```
  zrm_addr[27:0] = (1*24576*1024) + (1*1024) +4 = 28'h1800404
  ```

- RBC Mode: (RBC_BRCn = 1)

Feedback

```
zrm_addr[27:0] = {000000000000001,001,0000000100} = 28'h0002404
```

Figure 4 6Gb ZLPDDR4 Models with 15-bit Row Address



5

Integration with the DUT

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

You should first have properly set the 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>*

Setting the Wrapper in the Compilation Project

Using a 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 wrapper.

The wrapper for bidirectional *DQS* signal has three 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.
- *USER_analyzer_en* : Setting this parameter at 0 will disable the analyzer feature. Memory activity during the runtime will not be reported. Default value is 1.

Wrappers and Associated Model Files Locations

Source code files for 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>*.

Integrating the Wrapper with the DUT for ZeBu Compiler

This section describes how to add the wrapper to an existing *zCui* compilation project.

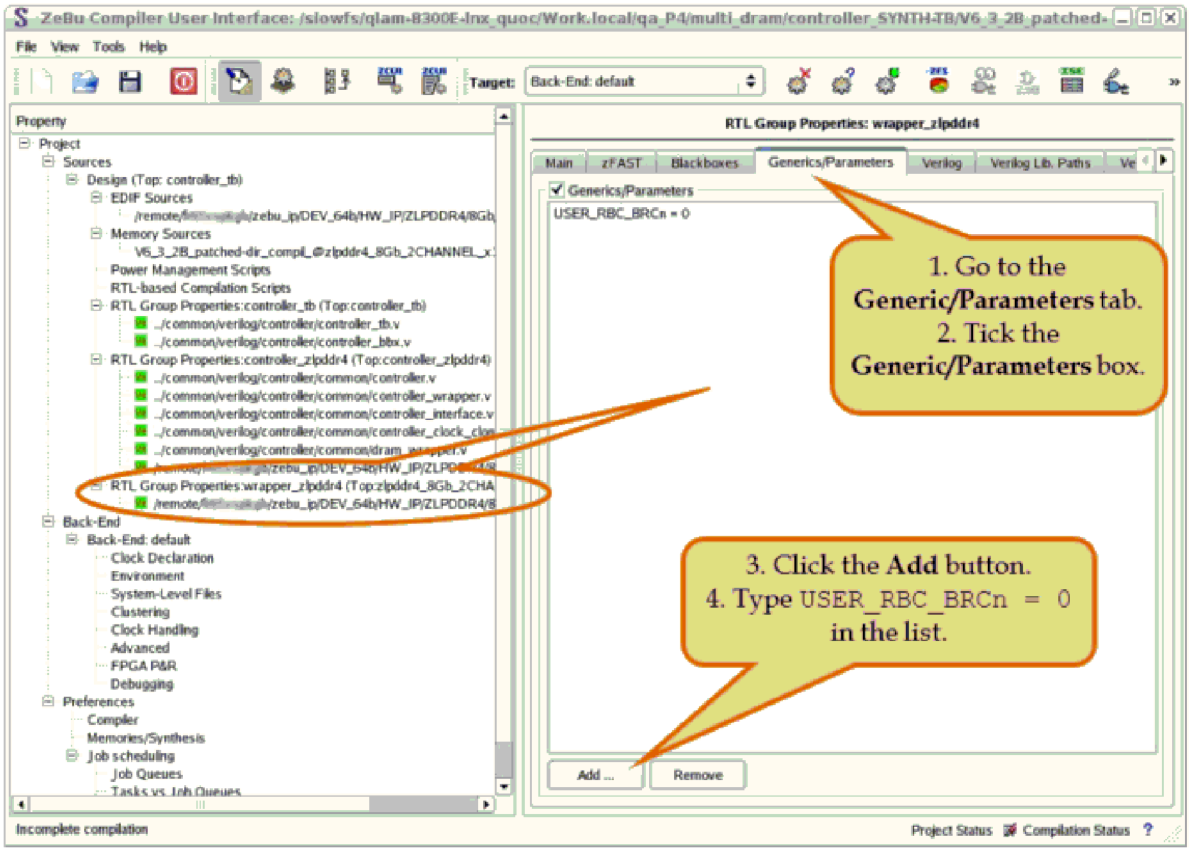
1. In the DUT, change the ZLPDDR4 model name to match the new bidirectional component name and modify the pathname as follows: *<model_path>/component/ <model_name>_bidir.<v/vhd>*.
2. Create a new dedicated group (for example, *GRP_ZLPDDR4)* in the DUT group that contains the following wrapper file: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

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

1. Go to the *Generic/Parameters* tab.

2. Tick the Generic/Parameters box.

Figure 5 Wrapper Parameter Assignment in zCui Interface


ZeBu Compiler User Interface
File View Tools Help
Target: Back-End: default
Property
RTL Group Properties: wrapper_zlpddr4
Main zFAST Blackboxes Generics/Parameters Verilog Verilog Lib. Paths
Add ... Remove
Incomplete compilation
Project Status Compilation Status

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.

USER_RBC_BRCn = 0

Synthesizing and Compiling for ZeBu

Synthesis

During 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 Section 4.1.2 and 4.1.3.

Enable the DPI synthesis for the Analyzer feature.

Compilation

To compile the design with the ZeBu ZLPDDR4 memory models, specify the relevant path of the RTL file in your zCui project.

Note:

For ZS5/EP1 following additional define need to be added in vcs command line to ensure the ZRM latency encoding matching with ZeBu firmware:

```
+define+ZRM_NEW_LATENCY
```

Simulation

The ZLPDDR4 package is supplied with:

- A *libIpSimu_<32/64>.so* library in the *<hw_ip_path>/lib*directory for simulation purposes.
- a liblpddr4_analyzer_simu_<32/64>.so library in the `<hw_ip_path>/lib` directory of analyzer.
- 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.

  Note:

  Do not use +define+ZEBU_SYNTH in simulation mode, the model is not functional.

Simulation with Synopsys VCS

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

```
vcs <my_options> <file_list> -sverilog
<model_path>/simu/rtl/vcs/<model_name>.vp
<hw_ip_path>/lib/libIpSimu /libIpSimu_<32/64>.so
<hw_ip_path>/lib/liblpddr4_analyzer_simu_<32/64>.so
```

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 -sverilog
<model_path>/simu/rtl/vcs/<model_name>.vp
```

Otherwise, you would get the following error message:

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

Result of ZeBu IP License Checking

For simulation with VCS, 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, contact your local representative.

Feedback

6 Accessing ZLPDDR4 Models (zrm Load & Dump)

This section explains the following topics:

- Running on ZeBu
- Initializing Memories for Simulation

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_sp.mem_lo
gic
<path_to_zLPDDR4_mem_B>=<path_to_zlpddr4_inst>.rank[1].mem_core_sp.mem_lo
gic
```

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>=<path_to_zlpddr4_inst>.rank[0].mem_core_sp.mem_lo
gic
<path_to_zLPDDR4_mem_B>=<path_to_zlpddr4_inst>.rank[1]mem_core_sp.mem_lo
gic
```

Note:

Also, when dumping the memory content, if the size of the dump files is bigger than 512Mbytes, these dump files are split into smaller files during runtime. However, you can disable splitting of dump files into smaller size files by setting the value of the ZEBU_DONT_SPLIT_MEMDUMP variable to false.

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");
  ```

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.

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_logic (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_logic (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)
```

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).

Note:

Due to the VCS® simulator' 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.

Feedback

7

Analyzer

The ZLPDDR4 Analyzer feature enables high level debugging and is embedded in the memory model. This feature reports DRAM transaction, with data payload.

This section explains the Analyzer feature under following topics:

- Overview
- Viewing Output
- Setup

Overview

The Analyzer feature provides relevant information on each command that the memory model receives, including Read and Write data payload.

As it may lead to generate huge log file, the Analyzer feature is disabled by default. Enabling the analyzer feature will increase the size of the memory model and may have a small impact on the performance.

To enable it, set the *USER_analyzer_en* to 1 when instantiating the ZLPDDR4 memory wrapper (in *$ZEBU_IP_ROOT/HW_IP/wrapper_rtl*).

The communication between the ZLPDDR4 memory model and the software library that generates the analyzer log file relies on ZDPI feature.

In order to minimize the size of the analyzer log file and to store only relevant information, a dedicated zebu register is provided to enable or disable the writing of data in the analyzer log file.

For example, user can enable the analyzer log file writing during a specific timing window.

It can be done dynamically at runtime by controlling the following zebuReg:

The path to the zebuMR_common register is:

<path_to_zlpddr4_inst>.ins_zebuMR_common.zebuReg[24]

- Setting this register to 1: disable write in the log file
- Setting this register to 0: enable write in the log file (default value)

Package Content

The existing Analyzer package contains an additional library in $ZEBU_IP_ROOT/HW_IP/lib:

- For Zebu emulation: libzlpddr4_analyzer_64.so
- For simulation: libzlpddr4_analyzer_simu_64.so (or _32.so, depending on your architecture).

Viewing Output

You can configure the display for the Analyzer messages to display these in a file or a terminal. There is one analyzer file per rank.

You can perform this configuration using the *DRAM_ANALYZER_OUTPUT* environment variable.

The following are the permissible values for the *DRAM_ANALYZER_OUTPUT* environment variable and the tool behavior:

- *Not defined*: Analyzer messages are displayed in a log file.
- *TERM*: Analyzer messages are displayed in the terminal.
- *FILE*: Analyzer messages are displayed in a log file.

This section explains the following topics:

- Setting the Path of the Log File
- Filename
- Log File Content
- Read And Write Commands
- Unsupported Commands
- Data filtering

Setting the Path of the Log File

You can set the path to the Analyzer log file using the *DRAM_ANALYZER_FILE_PATH* environment variable.

- If not defined, default path is the current directory where the run is launched
- If defined to a specific path, the log file will be created in this specific path.

Filename

The filename for a specific instance of ZLPDDR4 is the path of the instance in the design followed by *analyzer.txt*.

Note:

The filename cannot be changed.

Example:

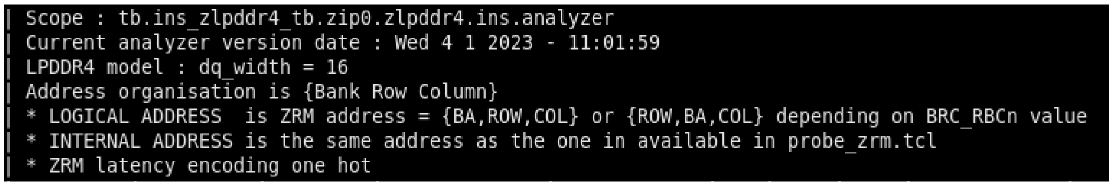
```
tb.ins_zlpddr4_tb.zip0.zlpddr4.analyzer.txt
```

Log File Content

The Analyzer log file consists of the following sections:

- Header
- Table Column

Header



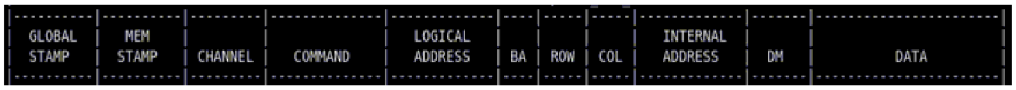
```
| Scope : tb.ins_zlpddr4_tb.zip0.zlpddr4.ins.analyzer
| Current analyzer version date : Wed 4 1 2023 - 11:01:59
| LPDDR4 model : dq_width = 16
| Address organisation is {Bank Row Column}
| * LOGICAL ADDRESS  is ZRM address = {BA,ROW,COL} or {ROW,BA,COL} depending on BRC_RBCn value
| * INTERNAL ADDRESS is the same address as the one in available in probe_zrm.tcl
| * ZRM latency encoding one hot
```

The header of the Analyzer log file consists of:

- Scope, which signifies the LPDDR4 instance name on which the analyzer is reporting information.
- Compile date where analyzer library has been compiled.
- General information about the memory model, such as, size of the Bank Group, and size of the Bank Address. The following information is reported for the DIMM model:
  - dq_width of the memory model
- Address organization
  - Logical address is the memory address organization
  - Internal address is the address translation used internally in the zebu memory model
- ZRM_Latency encoding is a debug message for internal use

Table Column

The following are the columns in the Analyzer log file:



```
|----------|---------|--------|-------------|------------|----|-----|-----|------------|-------|-----------------------|
| GLOBAL   |  MEM    |        |             |  LOGICAL   |    |     |     |  INTERNAL  |       |                       |
| STAMP    |  STAMP  |CHANNEL |   COMMAND   |  ADDRESS   | BA | ROW | COL |  ADDRESS   |  DM   |         DATA          |
|----------|---------|--------|-------------|------------|----|-----|-----|------------|-------|-----------------------|
```

This section explains the columns in the Analyzer log file:

- Global Stamp
- Mem Stamp
- Channel
- Command
- Logical Address
- BA
- ROW
- COL
- INTERNAL ADDRESS
- DM
- Data

Global Stamp

Time stamp reported by *svGetTimeFromScope()*.

Mem Stamp

Reports the zLPDDR4 CK_t_<A/B> clock cycle counter.

Channel

Related channel of the zLPDDR4 (Channel A or B)

Command

Name of the reported command. Name displayed on the log file is the same as on the JEDEC specification.

Example

```
|  0|  42497  |  ChA    |WR    |   0|    0|     0|     0|      -  |  -  |
```

Logical Address

The reported Read and Write address is built based on the Bank Address (BA), Row (R) and Column (C). Depending the *RBC_BRCn* parameter value, reported address is:

- *RBC: {R,BA,C}*
- *BRC: {BA,R,C}*

Each width may vary depending on the memory model (Refer to SDRAM addressing in the LPDDR4 SDRAM specification).

BA

Reports the Bank Address decoded from the related command.

ROW

Reports the Row Address decoded from the related command.

COL

Reports the Column Address decoded from the related command.

INTERNAL ADDRESS

Reports the address where the data is located in the ZRM memory array.

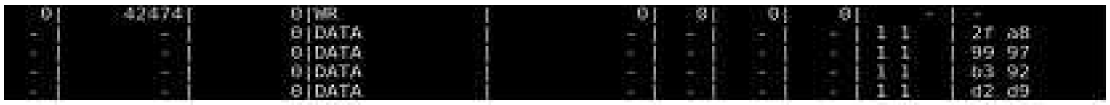
DM

Reports byte enable. Relevant for mask write command. 0 corresponding data byte is masked and not written in the memory. Else corresponding data byte is written in the memory. For non-masked command, byte enable is displayed at 1.

Data

Reports the MRW/MRR and Read/Write/Mask-Write data from/to memory.

Example

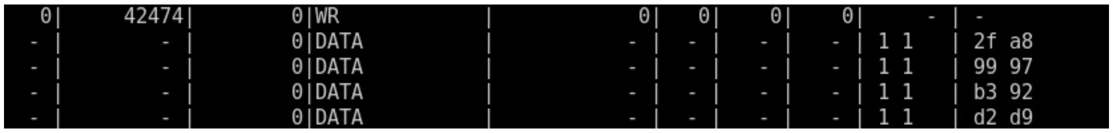


```
  0|    42474|          0|WR        |       0|   0|   0|   0|    - | -
  - |       - |         0|DATA      |       - |  - |  - |  - |1 1   | 2f a8
  - |       - |         0|DATA      |       - |  - |  - |  - |1 1   | 99 97
  - |       - |         0|DATA      |       - |  - |  - |  - |1 1   | b3 92
  - |       - |         0|DATA      |       - |  - |  - |  - |1 1   | d2 d9
```

First line is reporting the first data of the burst, second line is reporting the second data of the burst, and so on.

Read And Write Commands

Since Read/Write command (RD/WR) and Data Payload (DATA) arrive separately on the bus, it is difficult to find the related command for a specific data payload. To ease the understanding of the Analyzer log file, Read and Write Command and Data Payload are reported in a single block in the log file. These have a single time stamp, where the command is issued.

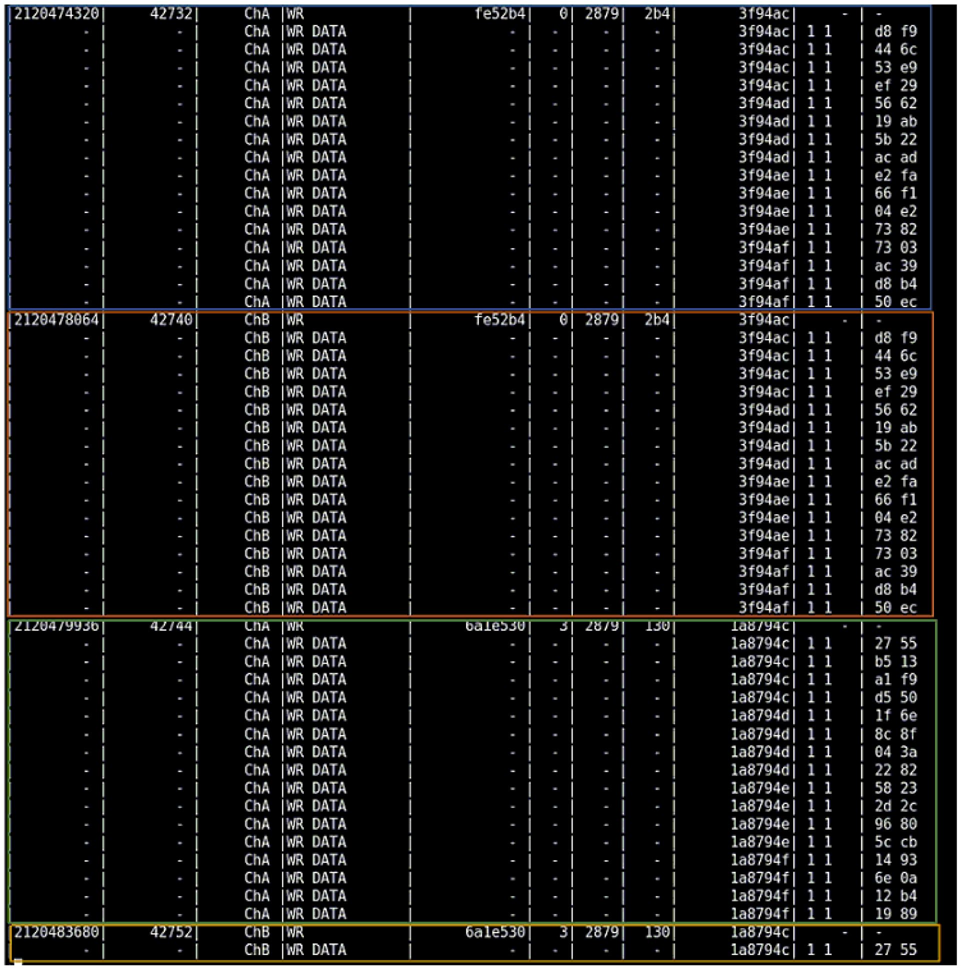


```
  0|     42474|          0|WR                     |           0|    0|     0|     0|       - | -
  - |         - |         0|DATA                   |          - |  - |    - |    - | 1 1    | 2f a8
  - |         - |         0|DATA                   |          - |  - |    - |    - | 1 1    | 99 97
  - |         - |         0|DATA                   |          - |  - |    - |    - | 1 1    | b3 92
  - |         - |         0|DATA                   |          - |  - |    - |    - | 1 1    | d2 d9
```

In the above example, a write command arrives at Mem Stamp 42474, and data arrives at the zLPDDR4 after the write latency. Data command and Data payload are displayed in the log file when the last Data is received, in a single block.

The following figure illustrates the consecutive data command with their attached data payload:

Figure 6 Data command and Related Payload



```
|2120474320|     42732|     ChA |WR           |      fe52b4|  0| 2879|  2b4|      3f94ac|        - | -
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ac| 1 1      | d8 f9
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ac| 1 1      | 44 6c
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ac| 1 1      | 53 e9
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ac| 1 1      | ef 29
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ad| 1 1      | 56 62
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ad| 1 1      | 19 ab
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ad| 1 1      | 5b 22
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ad| 1 1      | ac ad
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ae| 1 1      | e2 fa
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ae| 1 1      | 66 f1
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ae| 1 1      | 04 e2
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94ae| 1 1      | 73 82
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94af| 1 1      | 73 03
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94af| 1 1      | ac 39
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94af| 1 1      | d8 b4
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |      3f94af| 1 1      | 50 ec
|2120478064|     42740|     ChB |WR           |      fe52b4|  0| 2879|  2b4|      3f94ac|        - | -
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ac| 1 1      | d8 f9
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ac| 1 1      | 44 6c
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ac| 1 1      | 53 e9
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ac| 1 1      | ef 29
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ad| 1 1      | 56 62
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ad| 1 1      | 19 ab
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ad| 1 1      | 5b 22
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ad| 1 1      | ac ad
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ae| 1 1      | e2 fa
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ae| 1 1      | 66 f1
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ae| 1 1      | 04 e2
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94ae| 1 1      | 73 82
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94af| 1 1      | 73 03
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94af| 1 1      | ac 39
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94af| 1 1      | d8 b4
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |      3f94af| 1 1      | 50 ec
|2120479936|     42744|     ChA |WR           |     6a1e530|  3| 2879|  130|     1a8794c|        - | -
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794c| 1 1      | 27 55
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794c| 1 1      | b5 13
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794c| 1 1      | a1 f9
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794c| 1 1      | d5 50
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794d| 1 1      | 1f 6e
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794d| 1 1      | 8c 8f
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794d| 1 1      | 04 3a
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794d| 1 1      | 22 82
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794e| 1 1      | 58 23
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794e| 1 1      | 2d 2c
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794e| 1 1      | 96 80
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794e| 1 1      | 5c cb
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794f| 1 1      | 14 93
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794f| 1 1      | 6e 0a
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794f| 1 1      | 12 b4
|        - |        - |     ChA |WR DATA      |          - | - |    - |   - |     1a8794f| 1 1      | 19 89
|2120483680|     42752|     ChB |WR           |     6a1e530|  3| 2879|  130|     1a8794c|        - | -
|        - |        - |     ChB |WR DATA      |          - | - |    - |   - |     1a8794c| 1 1      | 27 55
```

Unsupported Commands

Unsupported commands not displayed in the log file.

The following MPC command are reported as unsupported in the analyzer log file:

- START DQS
- STOP DQS
- ZQCAL START
- ZQCAL LATCH

Data filtering

You can filter some reported information out from the log file, using the following three filters, each available for Channel A (_CHA) and one set for Channel B (_CHB):

- Filtering memory register access
- Filtering non-data related command
- Filtering data based on an address field

These filtering options rely on specific environment variables.

Register access

Use this filter to mask all the Memory Register Read or Write access. To enable this filter, set the *SNPS_VS_ZIP_LPDDR4_FILTER_MR_CH<A/B>* environment variable to 1.

Non-Data Commands

Use this filter to masking all non-data related commands, such as, Precharge and Activate.

To enable this filter, set the *SNPS_VS_ZIP_LPDDR4_FILTER_NONDATA_CH<A/B>* environment variable to 1.

Data filtering

This filter is based on an address range. It masks every Read / Write command that is addressing data outside the address range. However, you can define the address range using the following environment variables with and hexadecimal value (Prefix 0x):

- *SNPS_VS_ZIP_LPDDR4_FILTER_ADD_LOW_<A/B> = 0x<addr_low>*
- *SNPS_VS_ZIP_LPDDR4_FILTER_ADD_HIGH_<A/B> = 0x<addr_high>*

Once address range is set, you can activate the data filter by setting the *SNPS_VS_ZIP_LPDDR4_FILTER_DATA_CH<A/B>* environment variable to 1.

Note:

If address low is superior to address high all data related commands are masked.

Use the following syntax to set these environment variables in a zRci .tcl project:

```
set ::env(SNPS_VS_ZIP_LPDDR4_FILTER_ADD_LOW)  "0x09FF0000"
```

Setup

Before design compilation

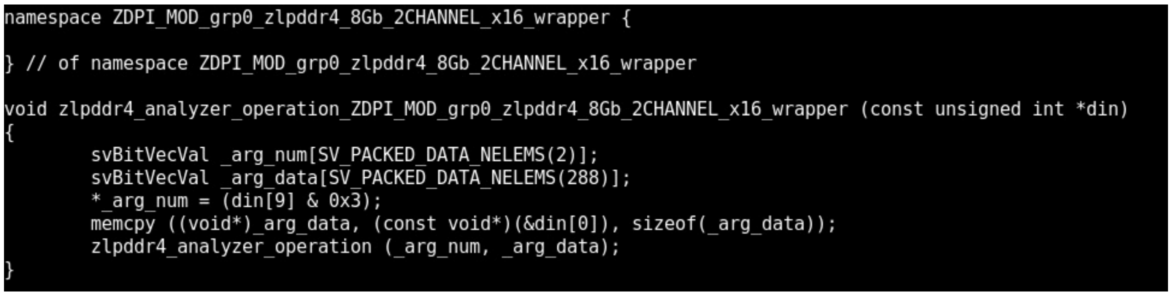
Before compiling your design, ensure the following:

- Enable the Analyzer feature by setting *USER_analyzer_en* parameter to 1 in the wrapper file.
- Analyzer is using System Verilog DPI feature to generate analyzer log file, ensure to activate Zebu DPI support.
- Ensure to update your LD_LIBRARY_PATH to add the directory where the ZLPDDR4 analyzer library (*libzlpddr4_analyzer_64.so*) is located.

After design compilation

After design compilation, check the availability of the analyzer feature by searching the following text in the grp0_ccall.cc (or *_ccall.cc) file. This file is available in the zebu.work directory.

The following is an example for 8Gb_2CHANNEL_x16 model:



```
namespace ZDPI_MOD_grp0_zlpddr4_8Gb_2CHANNEL_x16_wrapper {

} // of namespace ZDPI_MOD_grp0_zlpddr4_8Gb_2CHANNEL_x16_wrapper

void zlpddr4_analyzer_operation_ZDPI_MOD_grp0_zlpddr4_8Gb_2CHANNEL_x16_wrapper (const unsigned int *din)
{
        svBitVecVal _arg_num[SV_PACKED_DATA_NELEMS(2)];
        svBitVecVal _arg_data[SV_PACKED_DATA_NELEMS(288)];
        *_arg_num = (din[9] & 0x3);
        memcpy ((void*)_arg_data, (const void*)(&din[0]), sizeof(_arg_data));
        zlpddr4_analyzer_operation (_arg_num, _arg_data);
}
```

Runtime

To emulate the LPDDR4 memory model on ZeBu with the analyzer feature, enable the DPI at the runtime.

Otherwise, DPI feature is disabled and therefore, disables the generation of the Analyzer messages.

Using zRci

Load the Analyzer shared library in order to emulate the ZDDR4 memory model on ZeBu, using the Analyzer feature.

Specify the following using zRci in the zRci TCL script:

```
ccall -load  $::env(ZEBU_IP_ROOT)/HW_IP/lib/libzlpddr4_analyzer_64.so
ccall -enable
```

Set the value of the LD_LIBRARY_PATH environment variable to the path the library.

Using a C++ Testbench

Specify the following in between board open and board init, in your C++ testbench:

```
// Board init
Board * <board> = Board::open("zcui.work/zebu.work");
//Enabling the DPI call
CCall::LoadDynamicLibrary(<board>,"libzddr4_DIMM_analyzer_64.so");
CCall::SelectSamplingClocks (<board>,"posedge <CLK>");
CCall:Start (<board>,NULL,NULL,-1);
<board> -> init();
```

Where *<CLK>* is the signal pathname connected to the *CK_t* port of the *zddr4* instance.

You can disable the Analyzer report by disabling DPI.

8 Debug Information

For debugging purposes, a list of signals is available at runtime to determine the internal behavior of ZLPDDR4 memory models.

A set of alias files for Verdi are also provided in the memory model package. These aim at facilitating decoding these signals. The alias feature is an independent Verdi feature that displays data in a more meaningful representation. In this case, alias files provide both the memory commands and the diagnostic port .

Diagnostic Port

A diagnostic port is provided on the interface of the transactor. The diagnostic port can be accessed at runtime by existing debug mechanisms such as dynamic-probes. The name of this 102-bit diagnostic port is *probe[101:0]*; the designation of the 102 bits is described in the Table below. This diagnostic port encodes, into a single bit-vector, all the information needed to analyze the behavior of your memory models on two channels.

Table 6 Diagnostic Port Signals

Signal		**Description**
Channel A	**Channel B**	
ZLPDDR4 Protocol checker		
probe[54:47]		*IP version*
probe[46]	*probe[101]*	Reserved
probe[45]	*probe[100]*	Reserved
probe[44]	*probe[99]*	Illegal command according to JEDEC and ignored by model
probe[43]	*probe[98]*	Illegal command according to JEDEC but accepted by model
probe[42]	*probe[97]*	Row addressing error detected
probe[41]	*probe[96]*	Column addressing error detected

Table 6 Diagnostic Port Signals (Continued)

Signal		**Description**
Channel A	**Channel B**	
probe[40]	*probe[95]*	*Command/feature currently unsupported*
ZLPDDR4 Debug Information		
probe[39:34]	*probe[94:89]*	Real read latency (RLmrs + tDQSCK)
probe[33:28]	*probe[88:83]*	Real write latency (WLmrs + tDQS2DQ)
probe[27]	*probe[82]*	Set to 1 when burst length 16 used for current operation
probe[26:23]	*probe[81:78]*	Read/Write latency set (MR2[6:0])
probe[22]	*probe[77]*	DBI read enable (MR3[6])
probe[21:19]	*probe[76:74]*	Read/Write latency set (MR2[6:0])
probe[18:17]	*probe[73:72]*	Burst length (MR1[1:0])
probe[16]	*probe[71]*	Frequency Set Point Operation mode (MR13[7])
probe[15]	*probe[70]*	Frequency Set Point Write enable (MR13[6])
probe[14]	*probe[69]*	Data mask disable (MR13[5])
probe[13]	*probe[68]*	DBI write enable (MR3[7])
probe[12]	*probe[67]*	Read pre-amble type (MR1[3])
probe[11]	*probe[66]*	Read post-amble length (MR1[7])
probe[10]	*probe[65]*	*Command Read valid*
probe[9]	*probe[64]*	*Command Write valid*
probe[8]	*probe[63]*	*Command Active valid*
probe[7]	*probe[62]*	*Command MRR valid*
probe[6]	*probe[61]*	*Command MRW valid*
probe[5]	*probe[60]*	*Command Precharge valid*
probe[4]	*probe[59]*	*Command MPC valid*
probe[3:0]	*probe[57:55]*	*Current state of ZLPDDR4*

In conjunction with this diagnostic port, the whole interface of the zrm memories is available at runtime, and can be used to determine the actual operations performed on the memory. The full pathname of the memory should be similar to:

```
<path_to_zlpddr4_mem> =
 top_dut.my_zlpddr4_ins.rank[0].mem_core_sp.mem_logic
<path_to_zlpddr4_mem> =
 top_dut.my_zlpddr4_ins.rank[1].mem_core_sp.mem_logic
```

The diagnostic port bit-vector and the zrm memory waveforms provide enough information to analyze ZLPDDR4 behavior and integration issues.

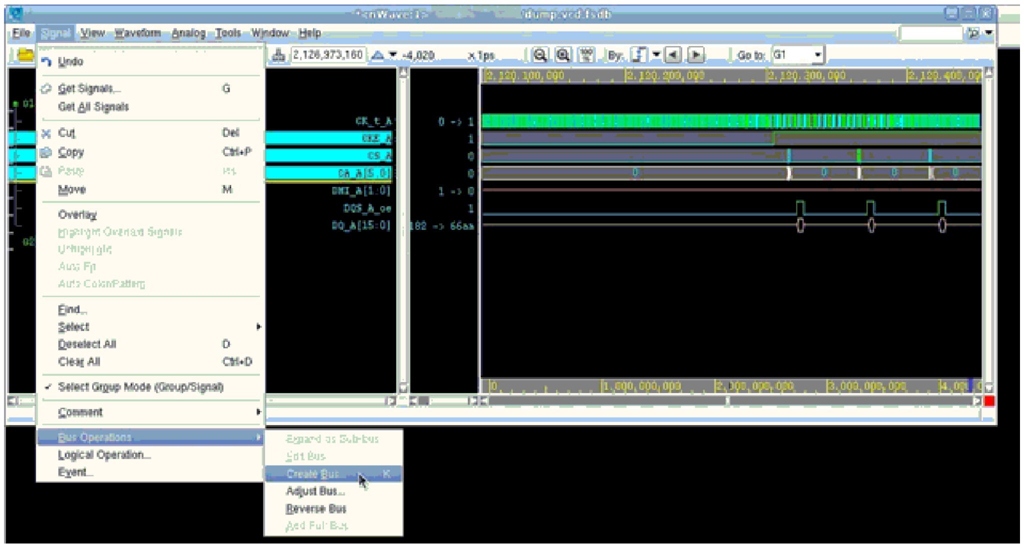
Alias Files for Verdi™

The ZLPDDR4 package provides two alias files in the *script/nWave* directory in Verdi to facilitate data viewing in Verdi:

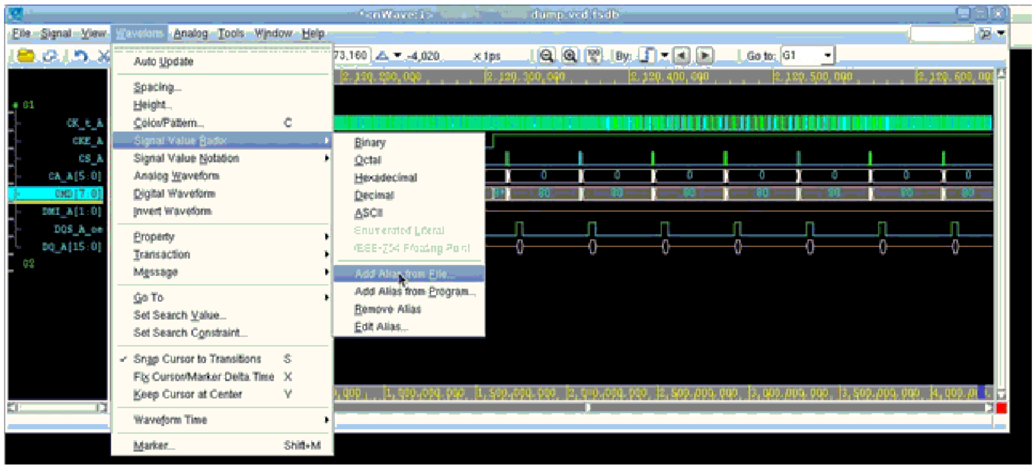
- cmd.alias automatically interpret the memory interface signals
- probe.alias automatically interpretsthe diagnostic port vector values

Using *Memory Interface Alias File*

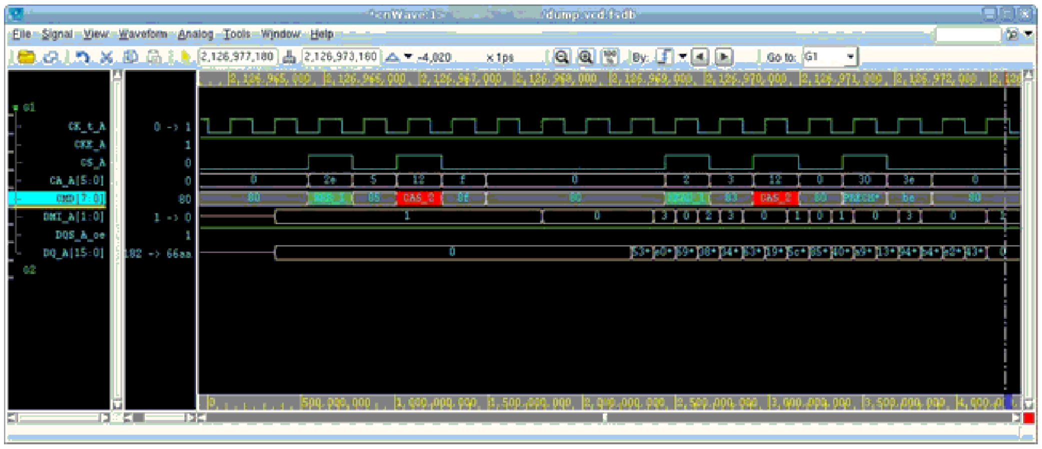
1. In *nWave*, select *Signal* > *Bus Operations* > *Create Bus* and create a new 5-bit width bus:
   - from CKE_<A/B>, CS_<A/B> and CA_<A/B>[5:0] in MSB to LSB name it *CMD* for example



1. Select the created bus vector (named *CMD* in the previous step).
2. 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:

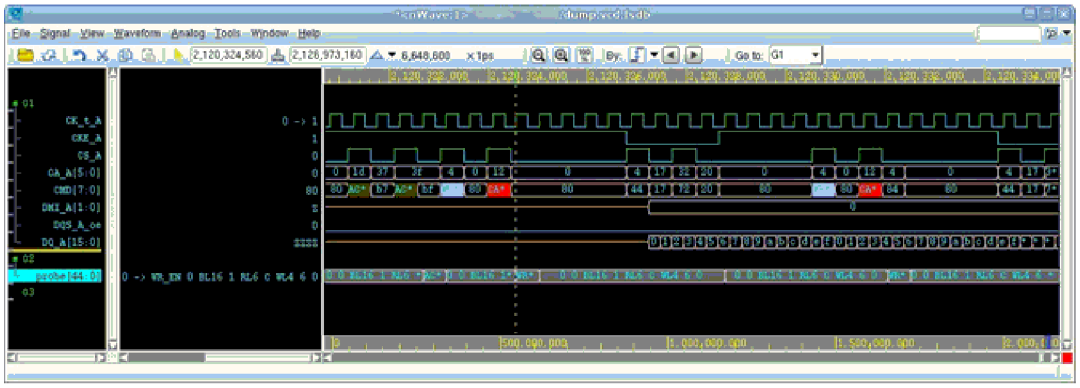


Using *Diagnostic Port Alias File*

1. In nWave, select Signal > Bus Operations > Create Bus and create a new 47-bit width bus from probe[101:0]:
   - 46:0 for channel A
   - 101:55 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:



9

Examples

The ZLPDDR4 package provides waveform example and a tutorial.

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.

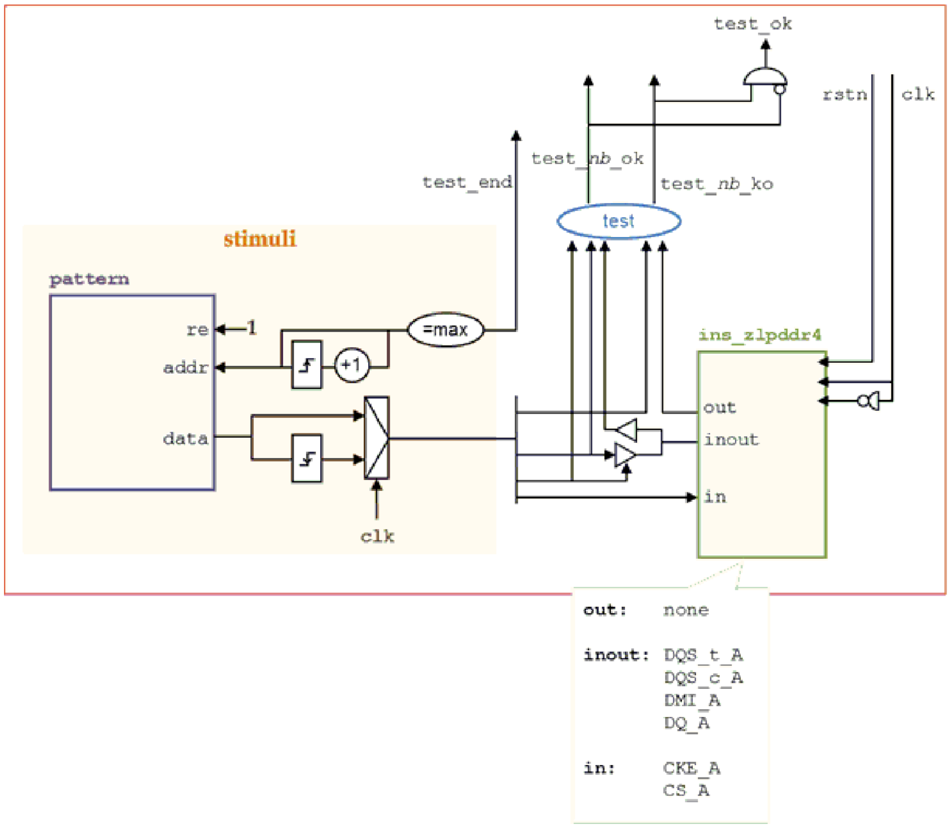
Tutorial

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

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

Description

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.

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/zRci.tcl* file for emulation with *zRci* and user-defined Tcl script

Running the Tutorial Examples

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 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)

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
   ```

1. Launch the emulation flow with the *Makefile*:

   ```
   make run
   ```

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

   ```
   make clean
   ```

Running Emulation with *zRci* 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 *zRci* Graphical User Interface
make run_gui	with *zRci* Graphical User Interface

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

   ```
   make clean
   ```