ZeBu DDR3 Memory Model User Manual

Version S-2023.03-SP2, July 2024

Contents

Feedback

1 About This Manual

Overview

This manual describes the ZeBu ZDDR3_SP SDRAM and ZDDR3_DIMM memory models library.

Related Documentation

For details about the SPD feature for ZDDR3_DIMM Memory Models, please refer to the ZeBu ZDDRx DIMM SPD Feature – User Manual.

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 Server 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.

Contents of This Book

The ZeBu Transactor User Manual has the following chapters:

Chapter	**Describes...**
Introduction	This chapter introduces ZeBu ZDDR3_SP SDRAM andZeBu ZDDR3_DIMM SDRAM memory models.
Installation	This chapter details about the installation anduninstallation procedures.
Memory Model Description	This chapters explains how ZDDR3_SP andZDDR3_DIMM synthesizable memory models areinternally configured as multi-bank DRAMs.

Chapter	Describes...
Integration with the DUT	This chapter describes how to integrate the ZDDR3_SPor ZDDR3_DIMM memory model with the DUT.
Accessing ZDDR3_SP/ZDDR3_DIMM Models (Load & Dump)	This chapter describes how to access ZDDR3_SP/ZDDR3_DIMM Models at run time.
Debug Information	This chapter details about a list of signals that areavailable at run time to determine the internal behaviorof ZDDR3_SP/ZDDR3_DIMM models.
Using the DRAM-Analyzer (ZDDR3_SP Models Only)	This chapter describes how to analyze the high-levelsystem or SoC architecture using the DRAM analyzer.
Examples (ZDDR3_SP Model Only)	This chapter provides a waveform example and atutorial.

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.

Feedback

2 Introduction

This chapter describes the following sub-topics:

- Overview
- Features
- Libraries
- Capacity with ZeBu
- Performance
- Limitations

Overview

Note:

This manual provides information for both the ZDDR3_SP and ZDDR3_DIMM memory model libraries. Thus, in this manual, “IP” refers to both ZDDR3_SP and ZDDR3_DIMM memory models.

The ZeBu ZDDR3_SP SDRAM and ZeBu ZDDR3_DIMM SDRAM memory models (also known as Fast DDR3 SDRAM) libraries come with optimized memory capacity and performance.

These libraries are provided as a set of memory models based on zrm-based memory models and compliant with DDR3 SDRAM and DDR3_DIMM SDRAM memory devices.

The ZDDR3_SP and ZDDR3_DIMM memory models are fully compliant with the JEDEC JESD79-3F (July 2012) specification available from the JEDEC website (*www.jedec.org*).

Features

The ZeBu ZDDR3_SP and ZDDR3_DIMM 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.
- Includes a DRAM-Analyzer (for ZDDR4 memory models only), which is a DDR3 SDRAM protocol analyzer and memory bus statistics collector, configurable at runtime, for easy system level debug and architecture analysis.
- Provides various blackbox components located in the *component* directory:
    - Verilog or VHDL
    - Bidirectional or Unidirectional
- Provides various Verilog wrapper files to include the ZeBu memory model within the user DUT:
    - Verilog or VHDL wrapper
    - Bidirectional or Unidirectional
    - System Verilog wrapper to use the embedded DRAM-Analyzer for ZDDR3_SP memory models.

Libraries

ZDDR3_SP Libraries

ZeBu ZDDR3_SP memory models are available for 512Mb and 1/2/4/8Gb memory capacity components on any compatible ZeBu systems. Each memory capacity component comes in x16/x8/x4 architectures.

Table 1 ZDDR3_SP Models

Memory Capacity	**Architectures**		
	x4	x8	x16
512 Mb	`zddr3_sp_512Mb_128x4`	`zddr3_sp_512Mb_64x8`	`zddr3_sp_512Mb_32x16`
1 Gb	`zddr3_sp_1Gb_256x4`	`zddr3_sp_1Gb_128x8`	`zddr3_sp_1Gb_64x16`
2 Gb	`zddr3_sp_2Gb_512x4`	`zddr3_sp_2Gb_256x8`	`zddr3_sp_2Gb_128x16`
4 Gb	`zddr3_sp_4Gb_1024x4`	`zddr3_sp_4Gb_512x8`	`zddr3_sp_4Gb_256x16`
8 Gb	`zddr3_sp_8Gb_2048x4`	`zddr3_sp_8Gb_1024x8`	`zddr3_sp_8Gb_512x16`

ZDDR3_DIMM Libraries

ZeBu ZDDR3_DIMM memory models can be used to build UDIMM (Unbuffered DIMM) and RDIMM (Registered DIMM) memory models.

The ZDDR3_DIMM memory models also exist with Memory Module Serial Presence-Detect (SPD) feature allowing standardization and storing of configuration, timing and manufacturing ZDDR3 DIMM information. Please refer to the ZeBu ZDDRx DIMM SPD Feature User Manual for further details.

The following ZDDR3_DIMM memory models are supplied in standard package. Each ZDDR3_DIMM memory model is entitled as follows:

zddr3_<DIMM_type>_<capacity>_<nb_of_ranks>_<nb_of_devices><depth><architecture>.edf

Table 2 ZDDR3_DIMM Memory Models

Memory Capacity	**Architectures**	
	x64	x72
512MB	`zddr3_UDIMM_512MB_1RANK_8d64x64.edf`	`zddr3_UDIMM_512MB_1RANK_8d64x72.edf`
	`zddr3_RDIMM_512MB_1RANK_8d64x64.edf`	`zddr3_RDIMM_512MB_1RANK_8d64x72.edf`
1 GB	`zddr3_UDIMM_1GB_1RANK_8d128x64.edf`	`zddr3_UDIMM_1GB_1RANK_8d128x72.edf`
	`zddr3_UDIMM_1GB_2RANK_8d64x64.edf`	`zddr3_UDIMM_1GB_2RANK_8d64x72.edf`
	`zddr3_RDIMM_1GB_1RANK_8d128x64.edf`	`zddr3_RDIMM_1GB_1RANK_8d128x72.edf`
	`zddr3_RDIMM_1GB_2RANK_8d64x64.edf`	`zddr3_RDIMM_1GB_2RANK_8d64x72.edf`
2 GB	`zddr3_UDIMM_2GB_1RANK_8d256x64.edf`	`zddr3_UDIMM_2GB_1RANK_8d256x72.edf`
	`zddr3_UDIMM_2GB_2RANK_8d128x64.edf`	`zddr3_UDIMM_2GB_2RANK_8d128x72.edf`
	`zddr3_UDIMM_2GB_4RANK_8d64x64.edf`	`zddr3_UDIMM_2GB_4RANK_8d64x72.edf`
	`zddr3_RDIMM_2GB_1RANK_8d256x64.edf`	`zddr3_RDIMM_2GB_1RANK_8d256x72.edf`
	`zddr3_RDIMM_2GB_2RANK_8d128x64.edf`	`zddr3_RDIMM_2GB_2RANK_8d128x72.edf`
	`zddr3_RDIMM_2GB_4RANK_8d64x64.edf`	`zddr3_RDIMM_2GB_4RANK_8d64x72.edf`

Table 2 ZDDR3_DIMM Memory Models (Continued)

Memory Capacity	Architectures	
4 GB	zddr3_UDIMM_4GB_1RANK_8d512x64.edf	zddr3_UDIMM_4GB_1RANK_8d512x72.edf
	zddr3_UDIMM_4GB_2RANK_8d256x64.edf	zddr3_UDIMM_4GB_2RANK_8d256x72.edf
	zddr3_UDIMM_4GB_4RANK_8d128x64.edf	zddr3_UDIMM_4GB_4RANK_8d128x72.edf
	zddr3_RDIMM_4GB_1RANK_8d512x64.edf	zddr3_RDIMM_4GB_1RANK_8d512x72.edf
	zddr3_RDIMM_4GB_2RANK_8d256x64.edf	zddr3_RDIMM_4GB_2RANK_8d256x72.edf
	zddr3_RDIMM_4GB_4RANK_8d128x64.edf	zddr3_RDIMM_4GB_4RANK_8d128x72.edf

Additional ZDDR3_UDIMM and ZDDR3_RDIMM memory models can be built and supplied upon request to your local representative, according to your detailed requirements.

Capacity with ZeBu

Up to 8 ZDDR3_SP memory model instances are supported on any compatible ZeBu systems. This value is the same for ZDDR3_DIMM memory models.

FLEXlm License Features

You need the hw_xtormm_ddr3 license feature.

Performance

ZDDR3_SP and ZDDR3_DIMM memory models use a single-port zrm memory with full-cycle access.

Logic Resources

The ZDDR3_SP and ZDDR3_DIMM memory models typically use the following FPGA resources:

Table 3 Logic Resources Example

Memory Model	Resources
`zddr3_sp_512Mb_32x16`	732 registers / 1,319 LUTs / 1 single-port zrm
`zddr3_sp_4Gb_256x16`	786 registers / 4,797 LUTs / 1 single-port zrm
`zddr3_UDIMM_512MB_1RANK_8d64x64`	<6500 registers / <6000 LUTs / 1 single-port zrm
`zddr3_UDIMM_4GB_1RANK_8d512x64`	<7000 registers / <6500 LUTs / 2 single-port zrm

Operating Frequency

ZDDR3_SP and ZDDR3_DIMM memory models performance depends on the zrm mapping (DDR2 or RLDRAM ZeBu resources). It also depends on the clock source provided to the clock input *Clk* signal: either the primary clock is directly connected to a clock signal sourced from the ZeBu clock generator, or the derived clock is connected to a gated or divided clock generated by the DUT from a ZeBu primary clock.

Table 4 Operating Frequency

Memory Model	ZeBu Server-Frequency	Memory Model Frequency
ZDDR3_SP/DIMM (Prim. clock)	16.66 MHz	8.33 MHz
ZDDR3_SP/DIMM (Div. clock)	25 MHz	6.25 MHz

Note:

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

Performance Tuning

Depending on the memory architecture of the targeted ZeBu system, some speed optimization can be obtained when the memory model is mapped in the FPGAs directly linked to ZeBu memory resources.

Optimization Using Memory Resources in the ZeBu Server Memory Server

For any ZDDR3_SP and ZDDR3_DIMM models, it is recommended to map the ZeBu synthesizable model in the ZeBu memory server FPGA logic. This will ensure optimal timing between the IP and the attached zrm device leading to predictable and maximum emulation performance between successive DUT compilations.

To map the IP into the memory server FPGAs, add an additional system-level compiler script as follows, depending on your design:

```
defmapping <path_to_ip_inst> U0.M0.F8
```

where *U0.M0.F8* is the ZeBu Server memory server FPGA.

Optimization using RLDRAM Resources in the ZeBu Server FPGA Module

For ZDDR3_SP (DIMM or not) memory models with capacities smaller than 1 Gb, it is possible to map the ZDDR3_SP synthesizable model in the ZeBu Server module containing RLDRAM memory resources. The DDR3 memory operating frequency can then be increased up to 7 MHz with RLDRAM-based models.

To use RLDRAM, add an additional system-level compiler script with following line:

```
defmapping -mem_type RLDRAM <path_to_ip>.mem_core_sp
```

Additional speed can be obtained by increasing the controller frequency at runtime from 200 to 300 MHz. To do so, add the following in your designFeatures file:

```
$rldramFrequency = 300000;
```

Limitations

For ZDDR3_SP Memory Models

- The Auto-refresh and Self-refresh DDR3 operations are ignored.
- It is not possible to use the DRAM-Analyzer with more than one instance of the ZDDR3_SP memory model. Please refer to Section Limitation for further details.

For ZLPDDR3_DIMM Memory Models

- The Auto-refresh and Self-refresh DDR3 operations are ignored.
- The DRAM-Analyzer feature is not available for ZDDR3_DIMM memory models.

3 Installation

This chapter describes the following sub-topics:

- Installing the ZDDR3 Package
- Uninstalling the ZDDR3 Package

Installing the ZDDR3 Package

Installation Procedure

The ZDDR3_SP and ZDDR3_DIMM memory models should be installed in the *ZEBU_IP_ROOT* directory.

To install the ZDDR3 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. Download the memory compressed shell archive (.*sh*) from SolvNet®.
4. Launch the installation script as follows:

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

5.

where *IP* is either *ZDDR3_SP* or *ZDDR3_DIMM*.

Note:

If the *ZEBU_IP_ROOT* environment variable is not set, you may launch the installation script as follows, specifying the path to *ZEBU_IP_ROOT*:

./<IP>.<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 *ZDDR3_SP/ZDDR3_DIMM* 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 (ZDDR3_SP only)	User scripts for Verdi® and the DRAM-Analyzer.
example directory (ZDDR3_SP only)	HDL simulation and emulation example files described in the Chapter Examples (ZDDR3_SP Model Only).
doc directory	This manual and the ZeBu ZDDRx DIMM SPD Feature User 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 blackbox for the design synthesis.**
simu directory	*/gate*: Verilog gate-level netlists for model simulation. */rtl*: Verilog source code-level netlists for model simulation.
wrapper_rtl directory	Verilog wrappers to include the component in the DUT as Verilog source code file for synthesis.

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 ZDDR3 Package

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

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

where IP is either ZDDR3_SP or ZDDR3_DIMM.

The uninstallation script executes the following operations:

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

Feedback

4

Memory Model Description

The ZDDR3_SP and ZDDR3_DIMM 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 IP size and architecture (see Section Libraries).

This chapter describes the following sub-topics:

Overview

The ZDDR3_SP and ZDDR3_DIMM synthesizable memory models can be used to model any Double-Data Rate 3 (DDR3) SDRAM memory, 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.

The ZDDR3_SP memory models have been validated and are functionally equivalent to the following DDR3 SDRAM memories:

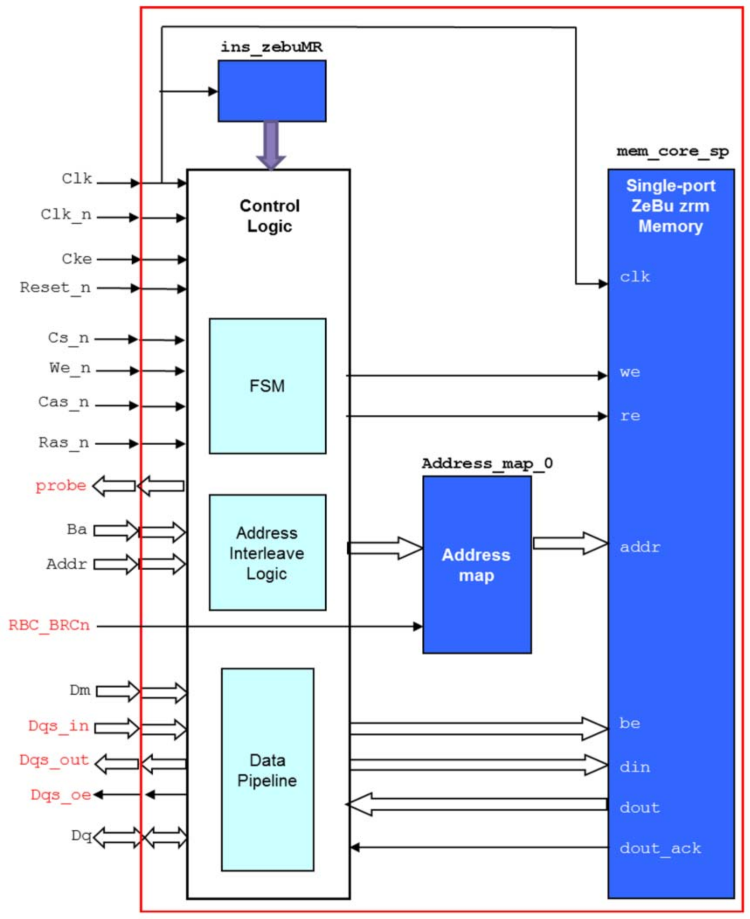
- Micron MT41xxxxxx DDR3 SDRAM family (*www.micron.com*)
- Samsung K4BxGxxxx DDR3 SDRAM family (*www.samsung.com*)

Functional Block Diagram

ZDDR3_SP Memory Model

The ZDDR3_SP synthesizable ZeBu memory models are architectured as follows:

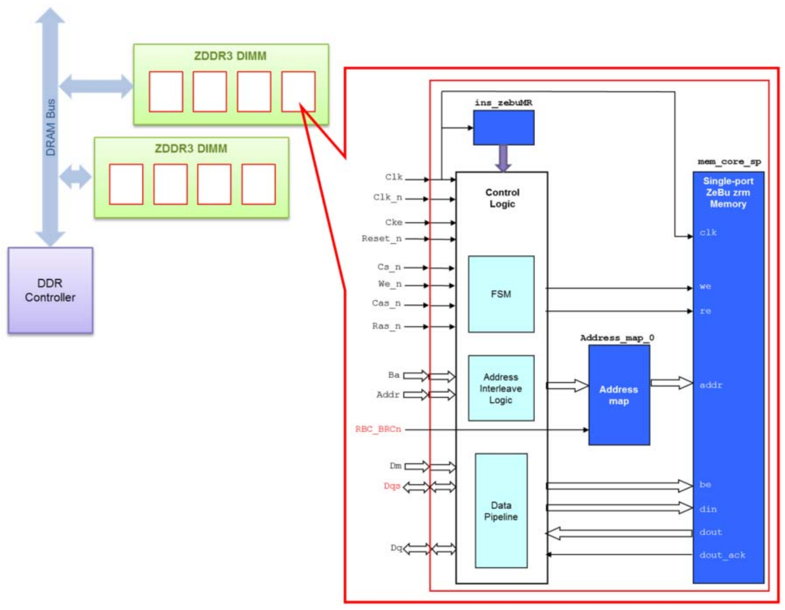
Figure 1 ZDDR3_SP Model Architecture



ZDDR3_DIMM Memory Models

The ZDDR3_DIMM synthesizable ZeBu memory models are architectured as follows:

Figure 2 ZDDR3_DIMM Memory Model Architecture



Interface Description

The ZDDR3_SP/ZDDR3_DIMM memory Verilog or VHDL model has a unidirectional interface. It is provided with a Verilog wrapper that allows using the model with a JEDEC-compliant bi-directional interface. Please refer to Section Setting the Verilog Wrapper in the Compilation Project for further details on how to use the wrapper.

The tables below describe the unidirectional and bidirectional ZDDR3_SP/ZDDR3_DIMM interface. The gray rows indicate specific ZDDR3_SP/ZDDR3_DIMM signals that do not exist in the DDR3 standard interface.

Table 5 ZDDR3_SP/ZDDR3_DIMM Unidirectional Interface

Name	Type	Description
`Clk` `Clk_n`	Input	Clock: *Clk* and *Clk_n* are differential clocks. All address and control input signals are sampled on the positive edges of *Clk*. *Dqs* signal is referenced on the *Clk* signal. *Clk_n* is not used in the ZDDR3_SP/ZDDR3_DIMM model.

Table 5 ZDDR3_SP/ZDDR3_DIMM Unidirectional Interface (Continued)

Name	Type	Description
`Cke`	Input	Clock Enable: *Cke* HIGH activates and *Cke* LOW de-activates internal clock signals and therefore device input buffers and output drivers.
`Cs_n`	Input	Chip Select: *Cs_n* is considered as part of the command code.
`Reset_n`	Input	Active low asynchronous reset.
`Ras_n, Cas_n, We_n`	Input	DDR3 SDRAM commands.
`Dm`	Input	Input Data Mask: *Dm* is the input mask signal for WRITE data.
`Dq`	I/O	Data Input/Output: Bi-directional data bus.
`Dqs_in`	Input	Data Strobe Input. It is used to sample WRITE data.
`Dqs_out`	Input	Data Strobe Output. It is output with READ data from memory.
`Dqs_oe`	Output	Output enable signal: output is active when HIGH.
`Ba`	Input	Bank address for ACTIVE, READ, WRITE and PRECHARGE command, number of mode register for LOAD MODE command.
`Addr`	Input	Address: row address in ACTIVE command column address in READ/WRITE command op-code in LOAD MODE command.
`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 6 ZDDR3_SP/ZDDR3_DIMM Bidirectional Interface

Name	Type	Description
`Clk` `Clk_n`	Input	Clock: *Clk* and *Clk_n* are differential clocks. All address and control input signals are sampled on the positive edges of *Clk*.
`Cke`	Input	Clock Enable: *Cke* HIGH activates and *Cke* LOW de-activates internal clock signals and therefore device input buffers and output drivers.

Table 6 ZDDR3_SP/ZDDR3_DIMM Bidirectional Interface (Continued)

Name	Type	Description
`Cs_n`	Input	Chip Select: *Cs_n* is considered part of the command code.
`Reset_n`	Input	Active low asynchronous reset.
`Ras_n,` `Cas_n,` `We_n`	Input	DDR2 SDRAM commands.
`Dm`	Input	Input Data Mask: *Dm* is the input mask signal for WRITE data. Input data is masked when *Dqm* is sampled HIGH. Dm[0] is the input data mask signal for the data on DQ0-3 (x4) or DQ0-7 (x8,x16).
`Dq`	I/O	Data Input/Output: Bi-directional data bus.
`Dqs`	I/O	Data Strobe Input. It is used to sample WRITE data (centered with WRITE data) and it is output with READ data from memory (edge-aligned to READ data).
`Ba`	Input	Bank address for ACTIVE, READ, WRITE and PRECHARGE command, number of mode register for LOAD MODE command.
`Addr`	Input	Address: row address in ACTIVE command column address in READ / WRITE command op-code in LOAD MODE command.
`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.

Differences with the DDR3 SDRAM Standard

DDR3 Device Interface Modifications

Dqs_in and Dqs_out Ports

The Dqs bi-directional differential port has been replaced by two unidirectional ports:

- *Dqs_in*: input to the ZDDR3_SP/ZDDR3_DIMM model
- *Dqs_out*: output from the ZDDR3_SP/ZDDR3_DIMM model

Since the *Dqs* port is used to latch data, this modification allows avoiding gated clocks. For proper use, the original *Dqs* bi-directional signal should be split into two unidirectional ports, *Dqs_in* and *Dqs_out*, inside the DDR3 controller mapped in the user design.

Dqs_oe Output Enable Port

An additional output enable port *Dqs_oe* is also available to manage the direction of Dq and Dqs signals: Dqs_oe=1 when *Dq* and *Dqs* buses are driven by the memory

Please refer to the Application Note VSAN001, ZDDRx ZIP Guidelines – Application Note, for more detailed information.

DDR3 Operations

The ZDDR3_SP/ZDDR3_DIMM model is functionally equivalent to the standard DDR3 memory device, but timing requirements are not applicable. The ZDDR3_SP/ ZDDR3_DIMM model is accurate up to a half cycle but will never consider setup and hold time for example.

All commands and operating modes are accepted by the ZDDR3_SP/ZDDR3_DIMM model, including the programming of CAS latencies, burst lengths, and additive latencies.

Auto-refresh and self-refresh commands are ignored. Refer to the reference DDR3 device datasheets for descriptions of correct operation of the DDR3 SDRAM.

Configurable Mode Register

The ZDDR3_SP/ZDDR3_DIMM memory models have a register dedicated to controlling runtime of the initial configuration: *zebuMR*.

The *zebuMR* register is divided into several Mode Registers (*MR*), each one corresponding to specific information as described in the JEDEC JESD79-3F (July 2012) specification. Please refer to it for further details on MR register's content.

The following table describes the zebuMR register content:

Table 7 zebuMR Mapping with Default Values

bits	**63:48**	**47:32**	**31 :16**	**15 :0**
	`MR3`	`MR2`	`MR1`	`MR0`
default value	`0`	`0`	`0`	`0`
information	Multi-purpose registers	Write latency	Additive latency	Operating modes & latency

The path to the *zebuMR* register is:

<path_to_ip>.ins_zebuMR.zebuMR[63:0]

zrm Address Translation

The memory space of the DDR3 device is three-dimensional and is organized in banks, rows and columns. In the ZDDR3_SP 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 DDR3.

The RBC_BRCn input is available at the ZDDR3_SP 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 Standard BRC mode
- *RBC_BRCn* = 1 Standard RBC mode

Example:

If you want to read your memory in a design using the 512Mb(x16) ZDDR3_SP model at Bank=1, Row=1, Column=4, then the starting address for zrm (in hexadecimal) will be as follows:

- BRC Mode: (*RBC_BRCn* = 0) *zrm_addr [24:0]* = *{01, 0000000000001, 0000000100}* = *0xh800404*
- RBC Mode: (*RBC_BRCn* = 1) *zrm_addr [24:0]* = *{0000000000001, 01, 0000000100}* = *0xh0014*

5 Integration with the DUT

This section describes how to integrate the ZDDR3_SP or ZDDR3_DIMM memory model with the DUT.

You should first have properly set the Verilog wrapper in your compilation project, as described in Section Setting the Verilog Wrapper in the Compilation Project below.

When using the DRAM-Analyzer feature, the integration procedure is slightly different from the one described hereafter, as detailed in Section Integration with the DUT.

In this chapter:

- IP stands for ZDDR3_SP or ZDDR3_DIMM
- *<hw_ip_path>* stands for *$ZEBU_IP_ROOT/HW_IP*
- *<ip_path>* stands for *<hw_ip_path>/<IP>.<version>*
- *<model_path>* stands for *<ip_path>/<size>/<arch>*
- *<xilinx_verilog>* stands for the Xilinx Verilog source. This path depends on the ZeBu platform:
    - ZeBu Server-1, Server-2 and Blade-2:

for 32-bit Linux OS: *$ZEBU_ROOT/ise/ISE_DS/ISE/bin/lin/xilinxd*

for 64-bit Linux OS: *$ZEBU_ROOT/ise/ISE_DS/ISE/bin/lin64/xilinxd*

- ZeBu Server-3:

for 32-bit Linux OS: *$ZEBU_ROOT/vivado/bin/unwrapped/lnx32.0/xilinxd*

for 64-bit Linux OS: *$ZEBU_ROOT/vivado/bin/unwrapped/lnx64.0/xilinxd*

This chapter describes the following sub-topics:

- Setting the Verilog Wrapper in the Compilation Project
- Synthesizing and Compiling for ZeBu
- Simulation

Setting the Verilog Wrapper in the Compilation Project

Why Using a Verilog Wrapper?

The ZDDR3_SP and ZDDR3_DIMM memory Verilog or VHDL models have 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 ZeBu memory model should be integrated with the DUT using a dedicated Verilog wrapper.

The Verilog wrapper for bi-directional 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.

The Verilog wrapper for unidirectional Dqs signal has only the USER_RBC_BCRn parameter as described above. Please refer to Section DDR3 Device Interface Modifications for further details on Dqs signal.

However, if the original component proposes an interface matching the IP pinout, it is also recommended to use the Verilog wrapper with a unidirectional interface for an easier integration.

Note:

To use the DRAM-Analyzer feature (ZDDR3_SP memory models only), it is mandatory to use the Verilog wrapper.

Wrapper and Associated Model File Location

Source code files for Verilog wrappers of the ZDDR3_SP and ZDDR3_DIMM interfaces are available at <model_path>/wrapper_rtl:

- <model_name>_unidir.<v/vhd> and <model_name>_bidir.<v/vhd> (when not using the DRAM-Analyzer feature)
- <model_name>_unidir_analyzer.sv and <model_name>_bidir_analyzer.sv (when using the DRAM-Analyzer feature)

Associated blackbox description Verilog and VHDL files are available at <model_path>/component:

- <model_name>_unidir.<v/vhd>
- <model_name>_bidir.<v/vhd>

- <model_name>.<v/vhd> (for compatibility with previous IP versions only)

  Note:

  In the ZDDR3_SP package, <model_name>.<v/vhd> file is available for compatibility with previous versions only. It is recommended to replace it by the <model_name>_unidir.<v/vhd> file (see Section below).

  If you use the DRAM-Analyzer feature, you must replace it by the <model_name>_unidir.<v/vhd> or <model_name>_bidir.<v/vhd> file. Please refer to Chapter for further details about it.

Integrating the Verilog Wrapper with the DUT for ZeBu Compiler

This section describes how to add the Verilog wrapper to an existing zCui compilation project. The procedure is the same for both the unidirectional and bidirectional wrappers.

Note:

In this section, we assume that you are not using the DRAM-Analyzer feature. Please refer to Section for further details on using the wrapper with the DRAM-Analyzer.

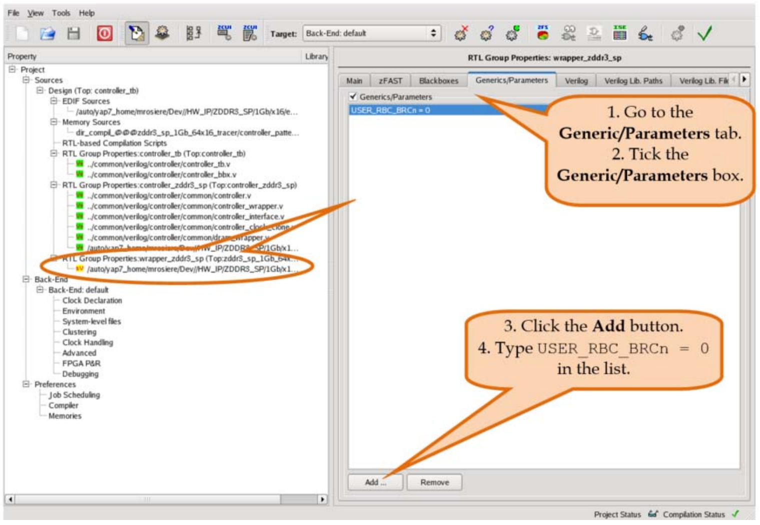
Case 1: IP is added in the DUT source

In this case, the DUT directly instantiates the ZDDR3_SP/ZDDR3_DIMM model.

1. In the DUT, change the ZDDR3_SP/ZDDR3_DIMM model name to match the new unidirectional component name and modify the source pathname as follows: <model_path>/component/<model_name>.v should become <model_path>/component/<model_name>_unidir.v
2. Create a new dedicated RTL-group (for example, GRP_ZDDR3_SP) that contains the following Verilog wrapper file : <model_path>/wrapper_rtl/<model_name>_unidir.v.

The assignment of the parameters for the Verilog wrapper is made through the Generic/ Parameters panel of zCui for the RTL-group instantiating the DUT:

Figure 3 Verilog Wrapper Parameter Assignment in zCui Interface


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.
RTL Group Properties: wrapper_zddr3_sp
Main
zFAST
Blackboxes
Generics/Parameters
Verilog
Verilog Lib. Paths
Add ...
Remove

Target: Back-End: default

✓ Generics/Parameters

USER_RBC_BRCn = 0

Case 2: Dedicated RTL Group to synthesize the memory model:

In this case, the ZDDR3_SP/ZDDR3_DIMM model instantiation is encapsulated in a specific RTL module, called ZIP_INSTANCE, which is the top of the dedicated RTL-group.

1. In the ZIP_INSTANCE module, replace the component file by the corresponding Verilog wrapper file : <model_path>/component/<model_name>.v should become <model_path>/wrapper_rtl/<model_name>_unidir.v
2. Change the ZDDR3_SP/ZDDR3_DIMM model name, adding the _unidir suffix.

The assignment of the parameters for the Verilog wrapper is made through the Generic/ Parameters panel of zCui for the RTL-group instantiating the ZIP_INSTANCE module.

Integrating the Verilog Wrapper with the DUT when Synthesizing for Simulation

This section describes how to add the Verilog wrapper to an existing simulation synthesis. The procedure is the same for both the unidirectional and bidirectional wrappers; the unidirectional wrapper is used as an example in the below procedure.

1. In the compilation procedure, replace the component file by the corresponding Verilog wrapper file: <model_path>/component/<model_name>.v should become <model_path>/wrapper_rtl/<model_name>_unidir.v
2. In the DUT, change the <model_name> model name into <model_name>_unidir.

The assignment of the parameters for the Verilog wrapper is added to the DUT.

Synthesizing and Compiling for ZeBu

Use the following commands to add the RTL bidirectional wrapper and the RTL model to an existing zCui compilation UTF project.

In VCS compilation command, add following option and files:

```
+define+ZEBU_SYNTH
$ZEBU_IP_ROOT/HW_IP/wrapper_rtl/<model_name>_bidir.v
$ZEBU_IP_ROOT/HW_IP/wrapper_rtl/<model_name>_bidir_analyzer.sv
$ZEBU_IP_ROOT/HW_IP/vlog/vcs/<model_name>.vp
```

Alternatively, you can use -y VCS option, as shown below:

```
+define+ZEBU_SYNTH +libext+.v+.vp+.sv
-y $ZEBU_IP_ROOT/HW_IP/wrapper_rtl
-y $ZEBU_IP_ROOT/HW_IP/vlog/vcs
```

Note:

The two following compilation settings must also be checked to be sure that internal configuration registers for DRAM-Analyzer (see Section) 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 ZDDR3_SP/ZDDR3_DIMM 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 ZDDR3_SP/ZDDR3_DIMM 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.
- No message is displayed during compilation but the register write operation is not possible at runtime.

Simulation

The ZDDR3_SP package is supplied with:

- A libIpSimu_<32/64>.so library in the libIpSimu directory for simulation purposes.
- A source code-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.

Simulation with Synopsys VCS

It is recommended to add the following lines mentioning ZDDR3_SP at the end of the script:

```
vcs <my_options> <file_list> -sverilog
+define+ZIP_NO_BLACKBOX_zddr3_sp
<model_path>/wrapper_rtl/<model_name>_bidir.v
<model_path>/wrapper_rtl/<model_name>_bidir_analyzer.sv
<model_path>/simu/rtl/vcs/<model_name>.vp
<hw_ip_path>/lib/libIpSimu_<32/64>.so
```

Note:

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

If you need to use several different ZDDR3_SP 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 +define+ZIP_NO_BLACKBOX_zddr3_sp
<model_path>/wrapper_rtl/<model_name>_bidir.v
<model_path>/wrapper_rtl/<model_name>_bidir_analyzer.sv
<model_path>/simu/rtl/vcs/<model_name>.vp
…
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
```

Result of ZeBu IP License Checking

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

```
# ---------- At time 0.0 ps Testing license ----------
#
#   ##############################################
#   #         Copyright (c) 2005-2014            #
#   # Emulation and Verification Engineering  SA #
#   #--------------------------------------------#
#   # ZeBu libIpSimu                             #
#   # revision : 2.2 64 bit                      #
#   # date : Tue 10 3 2009 - 12:50:08            #
#   #--------------------------------------------#

# ##############################################
#
#
# Testing ZDDR3 512Mb ZeBu IP license
#
# Checking out ZDDR3 512Mb license
#
# ---------- At time 0.0 ps check license OK ----------
```

Otherwise, please contact your local representative.

6

Accessing ZDDR3_SP/ZDDR3_DIMM Models (Load & Dump)

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. This path to the memory is like:

```
<path_to_ZDDR3_SP_mem> = <path_to_ZDDR3_SP_inst>.mem_core_sp.mem_logic
```

Example

You want to initialize memory in a design using an ZDDR3_SP model with the 'memory.init' content file. If the path to the ZDDR3_SP instance is 'top_dut.my_ZDDR3_SP_inst', then the full path to the ZRM memory would be:

```
<path_to_ZDDR3_SP_mem> = top_dut.my_ZDDR3_SP_inst.mem_core_sp.mem_logic
```

Therefore, add the following line 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_ZDDR3_SP_mem> memory.init
```

Note:

The 'designFeatures' file is a way to initialize the ZRM memories in an emulation-based environment like '$readmemh' Verilog system task used in simulation.

- In a C++ co-simulation testbench:

```
my_memory = my_board->getMemory("<path_to_ZDDR3_SP_mem>");
my_memory->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 ZDDR3_SP models simulated at RTL level.

Using Verilog System Tasks

Verilog system tasks, $readmemh and $writememh are used to initialize and dump memory data respectively.

- $readmemh system task reads address and corresponding data (in hexadecimal format) from a file and loads it into the memory.

  ```
  readmemh("load_data.hex",
   "<path_to_ZDDR3_SP_inst>.mem_core_sp.mem_logic[start@,stop@]");
  ```
-
- $writememh system task dumps address and corresponding data (in hexadecimal format) to a file from the memory.

  ```
  $writememh("dump_data.hex","<path_to_ZDDR3_SP_inst>.mem_core_sp.mem_lo
  gic[start@,stop@]");
  ```
-

Using Specific Verilog Tasks

Specific Verilog tasks are used to initialize the memory array to all 0 or 1:

```
<path_to_ZDDR3_SP_inst>.mem_core_sp.zip_init_mem_all(0|1)
```

7

Debug Information

For debugging purposes, a list of signals is available at runtime to determine the internal behavior of ZDDR3_SP/ZDDR3_DIMM models.

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

This chapter describes the following sub-topics:

- Diagnostic Port
- Alias Files for Verdi™

Diagnostic Port

A diagnostic port is provided on the interface of the transactor. The diagnostic port can be accessed at runtime by any of the existing debug mechanisms such as dynamic-probes. The name of this 35-bit diagnostic port is probe[34:0]; the designation of the 35 bits is described in the below table. This diagnostic port encodes into a single bit-vector all the information needed to analyze the behavior of your memory models.

Table 8 Bit Designation of the Diagnostic Port

Bit Position / Range	**Description**
34	Reserved
33:26	IP version
25	Write into reserved value for MR1[4:3]
24	Reads or writes in an invalid column.
23	Bank conflict between "activate" or "precharge" command and "read" or "write" command.
22	Reads in a deactivated bank.

Table 8 Bit Designation of the Diagnostic Port (Continued)

Bit Position / Range	**Description**
21	Writes in a deactivated bank.
20	Writes in mode register in an invalid state.
19	cmd_write signal: set to 1 when a WRITE operation is in progress.
18	cmd_cmd_read signal: set to 1 when a READ operation is in progress.
17	cmd_active signal, set to 1 when cmd_active is activated.
16	cmd_precharge signal: set to 1 when pre-charge is activated.
15	cmd_refresh signal, set to 1 when refresh is activated.
14	load_mode_register signal, set to 1 when MRS load is in progress.
13:12	Current state of ZDDR3_SP/ZDDR3_DIMM wrapper: Reserved.
11:10	Additive latency.
9:3	Mode register 0[6:0]
2:0	CAS Write latency.

In conjunction with this diagnostic port, the whole interface of the zrm memories is fully visible at runtime, which can be used to determine the actual operations done on the memory.

The full pathname of the memory should be similar to:

```
<path_to_ip_mem>=top_dut.my_ip_sp_instance.mem_core_sp
```

The diagnostic port bit-vector and the zrm memory waveforms provide enough information to analyze ZDDR3_SP/ZDDR3_DIMM behavior and integration issues.

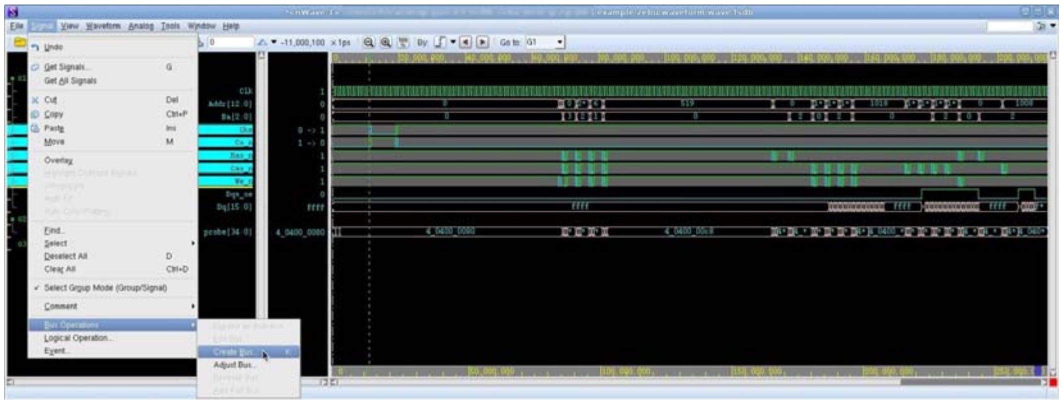
Alias Files for Verdi™

The ZDDR3_SP package provides two alias files in the script/nWave directory to facilitate debug in Verdi:

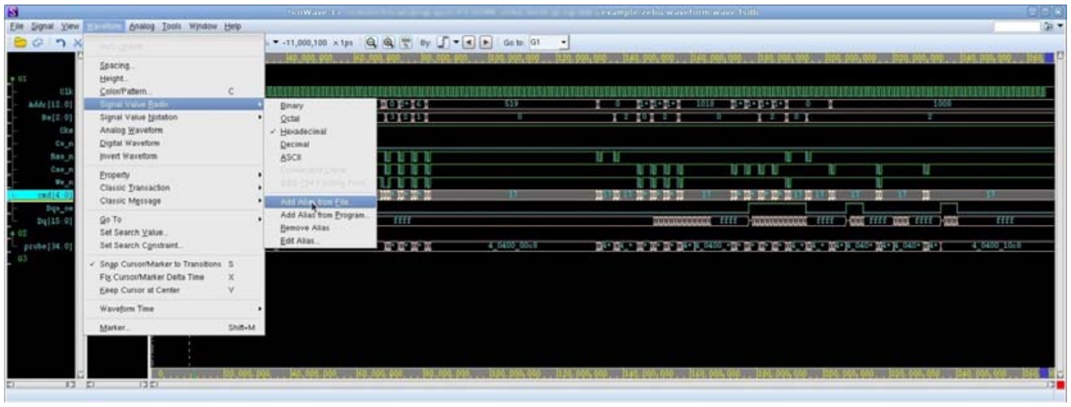
- cmd.alias automatically interprets the memory interface signals
- probe.alias automatically interprets the probe[]
- diagnostic port values

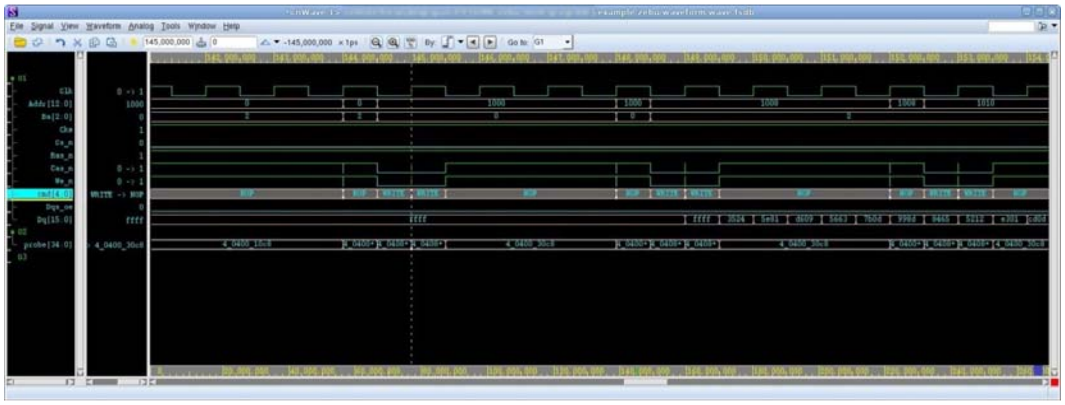
Using cmd.alias

1. In nWave, select Signal > Bus Operations > Create Bus and create a new 5-bit width bus:
   - from Cke, Cs_n, Ras_n, Cas_n and We_n 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:
4. You should get a display result similar to the one below:





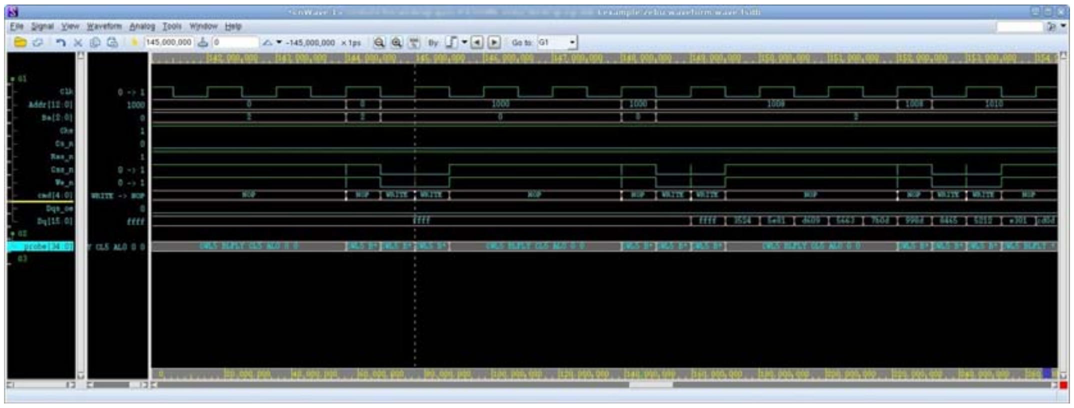
Note:

The command decode is only valid on the rising edge of the clock (in the figure above, only ACT command is valid, not READ).

Using probe.alias

1. Select the probe[34:0] bus vector.
2. 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:



8

Using the DRAM-Analyzer (ZDDR3_SP Models Only)

Note:

The DRAM-Analyzer feature is not available for the ZDDR3_DIMM memory models.

In this chapter:

- <hw_ip_path> stands for $ZEBU_IP_ROOT/HW_IP
- <ip_path> stands for <hw_ip_path>/ZDDR3_SP.<version>
- <model_path> stands for <ip_path>/<size>/<arch>
- <xilinx_verilog> stands for the Xilinx Verilog source. This path depends on the ZeBu platform:
  - ZeBu Server-1, Server-2 and Blade-2:

for 32-bit Linux OS: $ZEBU_ROOT/ise/ISE_DS/ISE/bin/lin/xilinxd

for 64-bit Linux OS: $ZEBU_ROOT/ise/ISE_DS/ISE/bin/lin64/xilinxd

- ZeBu Server-3:

for 32-bit Linux OS: $ZEBU_ROOT/vivado/bin/unwrapped/lnx32.0/xilinxd

for 64-bit Linux OS: $ZEBU_ROOT/vivado/bin/unwrapped/lnx64.0/xilinxd

This chapter describes the following sub-topics:

- Description
- Integration with the DUT
- DRAM-Analyzer Configuration
- DRAM-Analyzer Report Content
- DRAM-Analyzer Statistics

Feedback

- Monitoring DRAM-Analyzer Events
- Using the DRAM-Analyzer in HDL Simulation

Description

For high-level system debugging or SoC architecture analysis purposes, a DRAM-Analyzer is embedded in the ZeBu ZDDR3_SP memory model.

The DRAM-Analyzer has two main features:

- DRAM transactions tracing for Read, Write and Mode Registers operations with or without payload
- DRAM transactions profiling and DRAM activity reports

All DRAM-Analyzer monitoring results are dumped on the terminal or in a disk file.

You can configure it using specific control registers located inside the ZeBu ZDDR3_SP memory instantiation:

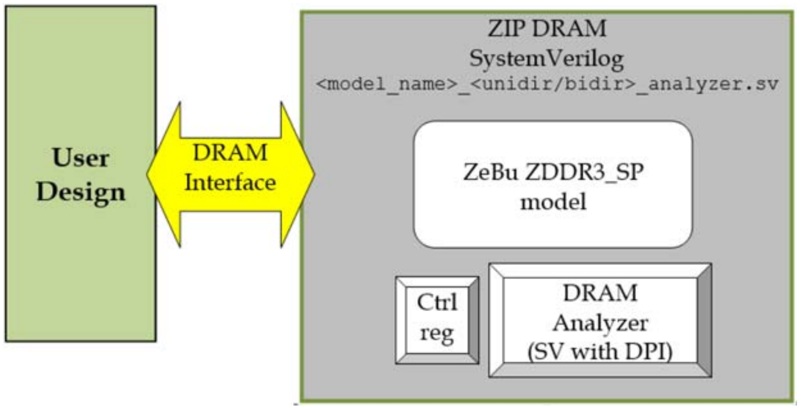
1. Activation/deactivation of the SDRAM analysis from the current cycle.
2. DRAM recording mode selection: Read/Write operations, Statistics, Transaction Payload display.
3. Results display definition: from the start cycle and for a defined duration. The time reference is the clk_ddr clock.

The DRAM-Analyzer feature is only available by the mean of a specific Verilog wrapper, whatever your interface (unidirectional or bidirectional). Please refer to Section Setting the Verilog Wrapper in the Compilation Project for further information on how to use the wrapper.

Overview

The ZeBu DRAM-Analyzer monitor is structured as follows:

Figure 4 ZeBu DRAM-Analyzer Interactions Overview



Requirements

The ZeBu embedded DRAM-Analyzer is written in SystemVerilog and requires the use of the ZeBu zFAST synthesizer when compiling.

Limitation

It is not possible to use the DRAM-Analyzer with more than one instance of the ZDDR3_SP memory model.

When your design instantiates several instances of ZDDR3_SP, whatever the model (ex. 2xZDDR3_SP 1Gb, or ZDDR3_SP 4Gb + ZDDR3_SP 8Gb, etc.), or several instances of different ZeBu Memory IPs (ex. ZDDR3_SP + ZSDRAM, etc.), you should compile your design in zCui with several RTL groups:

- one RTL-group with the ZDDR3_SP instance which uses the DRAM-Analyzer, and the implementation itself,
- one RTL-group with other ZeBu Memory IPs instances,

the DUT being in another group than the group with the ZDDR3_SP using the DRAM-Analyzer, as described in Section Synthesis and Compilation for ZeBu.

Integration with the DUT

Model File Definition

The DRAM-Analyzer is embedded with the ZeBu ZDDR3_SP memory model, but it needs specific blackbox and Verilog wrapper source files to be used for compilation.

SystemVerilog source code files for the Verilog wrapper are available at <model_path>/wrapper_rtl/:

- <model_name>_unidir_analyzer.sv
- <model_name>_bidir_analyzer.sv

Associated blackbox description Verilog and VHDL files are available at <model_path>/component/:

- <model_name>_unidir_analyzer.<v/vhd>
- <model_name>_bidir_
- analyzer.<v/vhd>

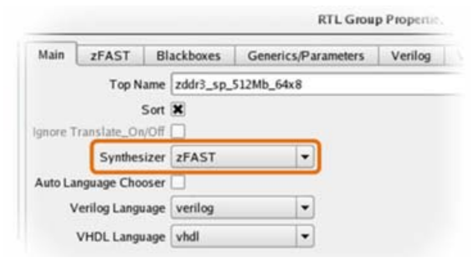
Synthesis and Compilation for ZeBu

You should set the following in zCui compilation interface to use the DRAM-Analyzer:

1. Use the <model_name>_analyzer.edf encrypted netlist file instead of <model_name>.edf.
2. In the RTL group of the DUT, add the Verilog/VHDL source file for the blackbox component: <model_name>_unidir.<v/vhd> or <model_name>_bidir.<v/vhd>.
3. Create a new dedicated RTL group (for example GRP_ZDDR3_SP), as described in the ZeBu Server Compilation Manual, and add the Verilog wrapper SystemVerilog file: <model_name>_unidir_analyzer.sv or <model_name>_bidir_analyzer.sv.

4. In the Main tab of the GRP_ZDDR3_SP RTL group, select zFAST synthesizer:

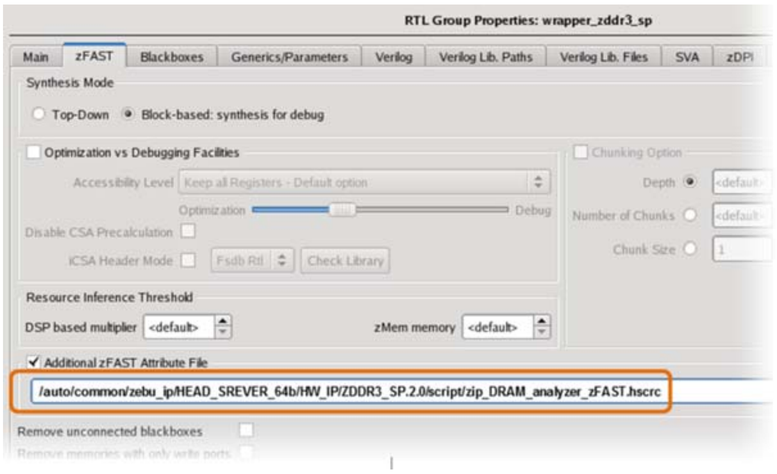
Figure 5 zFAST synthesizer choice for the DRAM-Analyzer in zCui



Top Name zddr3_sp_512Mb_64x8
Sort ✓
Ignore Translate_On/Off ☐
Synthesizer zFAST
Auto Language Chooser ☐
Verilog Language verilog
VHDL Language vhdl

5. In the zFAST tab of the GRP_ZDDR3_SP RTL group, declare the Additional zFAST Attribute File: zip_DRAM_analyzer_zFAST.hcsrc, located in the <ip_path>/script directory:

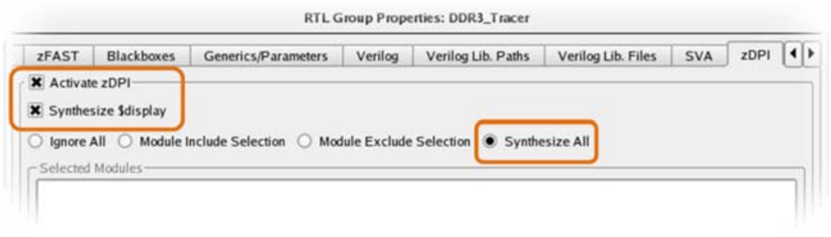
Figure 6 Declaring the Additional zFAST Attribute File for DRAM-Analyzer



Synthesis Mode
Top-Down [✓ Block-based: synthesis for debug]
☐ Optimization vs Debugging Facilities
Accessibility Level Keep all Registers - Default option
Optimization Debug
Disable CSA Precalculation ☐
ICSA Header Mode ☐ Fsdb Rtl
☐ Chunking Option
Depth ● <default>
Number of Chunks ○ <default>
Chunk Size ○ 1
Resource Inference Threshold
DSP based multiplier <default>
zMem memory <default>
✓ Additional zFAST Attribute File
/auto/common/zebu_ip/HEAD_SREVER_64b/HW_IP/ZDDR3_SP.2.0/script/zip_DRAM_analyzer_zFAST.hscrc
Remove unconnected blackboxes ☐
Remove memories with only write ports ☐

6. In the zDPI tab, activate the zDPI feature with the following options:

Figure 7 Activating zDPI feature in zCui



✓ Activate zDPI
✓ Synthesize $display
Ignore All / Module Include Selection / Module Exclude Selection [✓ Synthesize All]
Selected Modules

All other options for synthesis and compilation in zCui are not impacted by the DRAM-Analyzer feature.

Running on ZeBu

To emulate the ZDDR3_SP model with the DRAM-Analyzer, you should proceed as follows:

1. Update the environment variable LD_LIBRARY_PATH to include the following path:

   ```
   <hw_ip_path>/lib/libzddr3_sp_analyzer_64.so
   ```

2.

3. Load the DRAM_Analyzer dynamic library, select sampling clocks and start the DRAM-Analyzer as follows:

- Using zRci:

These actions are performed using the zddr3_sp_analyzer_zRCI.tcl template script provided in the package (script directory). This script contains the following commands:

```
ccall -load $ZEBU_IP_ROOT/HW_IP/lib/libzddr3_sp_analyzer_64.so
ccall -sampling_clock -expression "posedge <CLK>"
start_zebu
ccall -enable
```

where <CLK> is the signal pathname connected to the Clk port of the ZDDR3_SP instance.

To use the script:

1. Set the following variables:
   - zddr3_sp_path: canonical path to the ZDDR3_SP instance
   - zddr3_sp_clock: name of the clock connected to the Clk port
   - zddr3_sp_analyzer_enable

DRAM-Analyzer configuration, see Section

for more details

- zddr3_sp_analyzer_dbg_lvl
- zddr3_sp_analyzer_stat_period
- zddr3_sp_analyzer_trace_cycle_start
- zddr3_sp_analyzer_trace_duration

1. Include the zddr3_sp_analyzer_zRCI.tcl script into the user-defined Tcl script as follows:

   ```
   source
    $ZEBU_IP_ROOT/HW_IP/ZDDR2_SP.<version>/script/zddr3_sp_analyzer_zRCI.
   tcl
   ```

2.

- From a C++ testbench:

These actions are performed using the zddr3_sp_analyzer_C_COSIM.cpp template script provided in this package (script directory). This script contains the following code:

```
// Board init
Board * <board> = Board::open("zcui.work/zebu.work");
// enabling the DPI calls
CCall::LoadDynamicLibrary(<board>,"libzddr3_sp_analyzer_64.so");
CCall::SelectSamplingClocks(<board>,"posedge <CLK>");
CCall::Start (<board>, NULL, NULL, -1);
<board> ->init();
```

where <CLK> is the signal pathname connected at the Clk port of the ZDDR3_SP instance.

To use the script:

1. Set arguments of the zddr3_sp_analyzer_configuration function defined in the template script:
   - zebu: pointer to the ZeBu board
   - zddr3_sp_path: canonical path to the ZDDR3_SP instance
   - zddr3_sp_clock: name of the clock connected to the Clk port
   - zddr3_sp_analyzer_enable
   - zddr3_sp_analyzer_dbg_lvl
   - zddr3_sp_analyzer_stat_period
   - zddr3_sp_analyzer_trace_cycle_start
   - zddr3_sp_analyzer_trace_duration
2. Include the zddr3_sp_analyzer_C_COSIM.hh header file provided in the script directory to your C++ testbench. Please refer to AN029, zDPI Feature for ZeBu for more detailed information.

3. Set the dedicated environment variables to define the SDRAM monitor outputs:

```
export DRAM_ANALYZER_OUTPUT=<output>
export DRAM_ANALYZER_FILE_PATH=<file>
```

4.

where: <output> can be one of the following values:

- TERM: each DRAM-Analyzer writes in the standard output
- FILE (default value): each DRAM-Analyzer writes in a specific file. The filename depends on the scope.

<file> can be one of the following values:

- [pathname]: all files are written in the pathname directory
- /tmp is the default value for DRAM_ANALYZER_FILE_PATH.

DRAM-Analyzer Configuration

Configuration Registers

Specific registers are used to enable and configure the embedded DRAM-Analyzer during emulation runtime.

The pathname to DRAM-Analyzer configuration registers is:

```
<path_to_zddr3_sp>.analyzer_config.<register_name>
```

The following table summarizes all DRAM-Analyzer configuration registers:

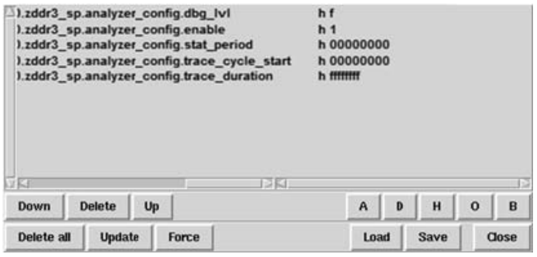
Table 9 DRAM-Analyzer Configuration Registers

Register name	Size (bits)	Default value	Description
`enable`	1	1	Enables the DRAM-Analyzer.
`dbg_lvl`	4	0xf	Enables and defines the configuration for the DRAM-Analyzer. Please refer to Section hereafter for further details.
`trace_cycle_start`	32	0	Starts the DRAM-Analyzer after the designated cycles of the SDRAM DDR clock.

Table 9 DRAM-Analyzer Configuration Registers (Continued)

Register name	Size (bits)	Default value	Description
`trace_duration`	32	-1	Stops the DRAM-Analyzer after the designated cycles of the SDRAM DDR clock. Default value -1 means unlimited length and the DRAM-Analyzer is not stopped.
`stat_period`	32	0	Defines the period in SDRAM clock cycles for dumping the statistics result of the DRAM-Analyzer. Please refer to Section hereafter for further details on statistics.

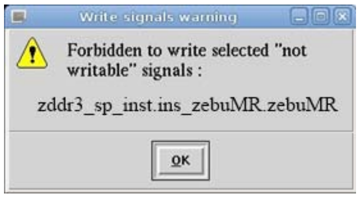
Figure 8 Configuration registers example as shown in zRci



```
).zddr3_sp.analyzer_config.dbg_lvl              h f
).zddr3_sp.analyzer_config.enable               h 1
).zddr3_sp.analyzer_config.stat_period          h 00000000
).zddr3_sp.analyzer_config.trace_cycle_start    h 00000000
).zddr3_sp.analyzer_config.trace_duration       h ffffffff
```

If the design instantiates RLDRAM memories and the compilation settings for register write were not the one described in Section Compilation, the register write operation is not possible at runtime. The following message is displayed at runtime when attempting to write to a configuration register:

Figure 9 Error Message in zRci for Forbidden Register-Write



Write signals warning

Forbidden to write selected "not writable" signals :

zddr3_sp_inst.ins_zebuMR.zebuMR

Setting the Content of the DRAM-Analyzer Report

The dbg_lvl register allows defining the content of the DRAM-Analyzer report. Each bit of dbg_lvl can enable/disable a recording type:

Table 10 dbg_lvl register description

Register bit	Description
dbg_lvl[0]	Dumps statistics
dbg_lvl[1]	Dumps read/write commands
dbg_lvl[2]	Dumps mode register write and bank activity (activate, precharge, auto-precharge)
dbg_lvl[3]	Dumps SDRAM read/write operation payload

Note:

Dumping of the payload involves dumping of the read/write command.

Registers Use Examples

Example 1: Enabling the DRAM-Analyzer with Statistics and Read/Write transactions monitoring:

```
<path_to_zddr3_sp>.zddr3_sp.analyzer_config.enable=1'b1
<path_to_zddr3_sp>.zddr3_sp.analyzer_config.dbg_lvl=4'b0011
```

Example 2: Disabling the DRAM-Analyzer:

```
<path_to_zddr3_sp>.zddr3_sp.analyzer_config.enable=1'b0
or
<path_to_zddr3_sp>.zddr3_sp.analyzer_config.dbg_lvl=4'b0000
```

DRAM-Analyzer Report Content

Read/Write Transaction to the DRAM Memory

Two lines are added in the report when a Read/Write DRAM command is received: the first line is generated when the transaction is initiated and the second line when the transaction is finished.

The DRAM transaction start description contains the following information:

```
{00001504} Read  address  (state active) (bank 1011/3) Ba 0, index 017,
 Address 0x0004b000
    (1)   |    (2)      |      (3)      |       (4)        |    (5)   |
 (6)
```

1. Current DDR clock cycle
2. Transaction type
3. DRAM state (active, write, read or idle)
4. Activated banks, selected bank
5. Transaction index, used to easily order the transactions
6. DRAM address in linear addressing

When the Read/Write operation is terminated (after BL/2 cycles), the end transaction description contains the following information:

```
{00001508} Read  finished   (state read  ) (bank 1011/3) index 017
    (1)   |    (2)        |      (3)      |     (4)     |   (5)
```

1. Current DDR clock cycle
2. Transaction type
3. DRAM state (active, write, read or idle)
4. Activated banks
5. Transaction index

Other DRAM Commands

The DRAM-Analyzer monitors the other types of commands like “Activate” command, “Precharge” command, Read/Write with auto-precharge, “Load” mode registers, etc.

The transaction start description contains the following information:

```
{00002202} Load Mode Reg (state idle) (bank 0000/0) Ba 0, Burst Length:
 BL 8 (fixed), Burst Type: interleaving, CAS:5
   (1)    |     (2)     |     (3)    |    (4)            |       (5)

{00002248} Bank  activity (state idle  ) (bank 0001/1)       Activate
 bank 0
   (1)    |      (2)      |    (3)     |    (4)            |       (5)
```

1. Current DDR clock cycle
2. Command type
3. DRAM state (active, write, read or idle)
4. Activated banks, selected bank (-1 if the command is a precharge all).
5. Extra specific command information

Payload

For the Read/Write transactions, the display of the payload content is optional in the information reported at the end of the transaction.

The DRAM transaction payload is displayed as follows:

- Address is Burst Length-aligned.
- Data in payload is written from LSB to MSB for the complete burst transfer. Data value X corresponds to a byte disabled write due to the data mask.
- Each data (depending on Dq size) is separated from each other by an underscore.

Example 1: Payload with 4-bit data size:

```
{00001627} Write payload index 030, Address 0x00800000, BL 8
{00001627}               data : e_f_2_8_5_4_a_a
{00001648} Read  payload index 032, Address 0x00800000, BL 8
{00001648}               data : e_f_2_8_5_4_a_a

{00002158} Write payload index 099, Address 0x00800100, BL 8
{00002158}               data : 5_b_c_e_X_X_X_X
```

Example 2: Payload with 16-bit data size:

```
{00001627} Write payload index 030, Address 0x00800000, BL 8
{00001627}               data : ef3f_f6d3_2f85_8878_595b_4b49_ae3f_af2a

{00001648} Read  payload index 032, Address 0x00800000, BL 8
{00001648}               data : ef3f_f6d3_2f85_8878_595b_4b49_ae3f_af2a

{00002158} Write payload index 099, Address 0x00800100, BL 8
{00002158}               data : 5bf8_b78d_c452_e784_XXXX_XXXX_XXXX_XXXX
```

DRAM-Analyzer Statistics

After each period, statistics are dumped by the DRAM-Analyzer. The Read/Write operations (number of cycles where the ZDDR3_SP model manages a Read/Write operation), the Read/Write command (depending on the Read/Write latency) and the throughput (Read/Write bytes per cycle) are printed, as shown in the example below:

```
{00100000} Since the beginning of statistics generation
{00100000} * Cycle Read  (mem) : 13428/100000  = 13 %
{00100000} * Cycle Write (mem) : 11624/100000  = 11 %
{00100000} * Cycle Idle  (mem) : 74948/100000  = 74 %
{00100000} * Cycle Read  (lat) : 39463/100000  = 39 %
{00100000} * Cycle Write (lat) : 27185/100000  = 27 %
{00100000} * Cycle Idle  (lat) : 33352/100000  = 33 %
{00100000} * READ  4605, Bytes=26856, Throughput = 268 Bytes/Kcycle
{00100000} * WRITE 4183, Bytes=23248, Throughput = 232 Bytes/Kcycle
```

Monitoring DRAM-Analyzer Events

To help in debugging the ZDDR3_SP behavior, the ZeBu DRAM-Analyzer detects some specific DRAM protocol events through dedicated triggers:

- Reads or writes in an invalid column.
- Bank conflict between “activate” or “precharge” command and “read” or “write” command.
- Reads in a deactivated bank.
- Writes in a deactivated bank.
- Writes in mode register in an invalid state.
- “Activate” command without effect (bank already activated)
- “Precharge” command without effect (bank already precharged)

Protocol Checker Messages

The DRAM-Analyzer works as a protocol checker and sends a message for each event listed above. Here are some examples of the messages sent by the DRAM-Analyzer:

Examples:

Read/Write in deactivated bank:

```
{00000339} Writes in a deactivated bank.
{00000350} Reads in a deactivated bank.
```

Read/Write an invalid column number in 1Gb ZDDR3_SP model:

```
{00000369} Write address (state active) (bank 00000010/1) Ba 1, index
 000, Address 0x00800000
{00000369} Reads or writes in an invalid column.
```

Load mode in invalid state:

```
{00000514} Bank  activity (state idle) (bank 00000010/1) Activate bank 1
{00000521} Load Mode Reg  (state active) (bank 00000010/1) Ba 0, Burst
 Length : BL 8 (fixed) , Burst Type : sequential, CAS : 5
{00000522} Writes in mode register in an invalid state.
```

Protocol Checker Signals

The protocol checker event detection is also available by reading the values at bit positions [20] to [24] of the diagnostic port (the diagnostic port is described in Chapter Debug Information):

Table 11 Protocol Checker Signals

Events	Type	Bit Position
Reads or writes in an invalid column.	Warning	`[24]`
Bank conflict between "activate" or "precharge" command and "read" or "write" command.	Error	`[23]`
Reads in a deactivated bank.	Error	`[22]`
Writes in a deactivated bank.	Error	`[21]`
Writes in mode register in an invalid state.	Error	`[20]`

The “Type” information corresponds to different runtime behaviors:

- Error: when the condition is verified, this signal is set to 1 and remains active after the event. It shows, at any time, that the behavior of the ZDDR3_SP model has been incorrect during emulation.
- Warning: the signal is set to 1 when the event occurs and is set back to 0 when the event is over. It shows that the behavior of the ZDDR3_SP model is incorrect only during the event.

Using the DRAM-Analyzer in HDL Simulation

In HDL simulation, the Verilog wrapper and the libzddr3_sp_analyzer_simu_<32/64>.so library must be used in addition to the libIpSimu_<32/64>.so library described in Section Simulation.

Using the DRAM-Analyzer with Synopsys VCS

```
vlogan +v2k -sverilog
 <model_path>/wrapper_rtl/<model_name>_unidir_analyzer.sv;

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

vcs <my_options>
 <hw_ip_path>/lib/libIpSimu_<32/64>.so
 <hw_ip_path>/lib/libzddr3_sp_analyzer_simu_<32/64>.so;
```

9

Examples (ZDDR3_SP Model Only)

The ZDDR3_SP package provides a waveform example and a tutorial (no example for ZDDR3 DIMM models).

This chapter describes the following sub-topics:

- Waveform Example
- Tutorial
- Description
- Running the Tutorial Examples

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 Tutorial hereafter).

The waveform file provided in the example/waveform directory.

Tutorial

This tutorial shows how to use the ZDDR3_SP Memory Models in the three following situations:

- HDL Simulation
- Emulation on ZeBu with C++ co-simulation
- Emulation of ZeBu with zRci and user-defined TCL scrip

Files Used for HDL Simulation

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

- testbench files: demo.v and demo_tb.v
- compilation files: pattern.tcl
- run files: pattern.txt (memory settings)
- automatic flow: Makefile

Files Used for Emulation on ZeBu with C++ Co-Simulation

The following files of the example/demo_analyzer directory shown above are used in case of emulation on ZeBu HDL with C++ co-simulation:

- testbench files: demo.v and testbench.cpp
- compilation files:
  - clock_C_COSIM.dve
  - demo.zpf
  - pattern.tcl
  - zTopBuild_option.tcl
- run files:
  - designFeatures
  - pattern.mem (memory settings)
  - pattern.txt (memory settings)
- automatic flow: Makefile

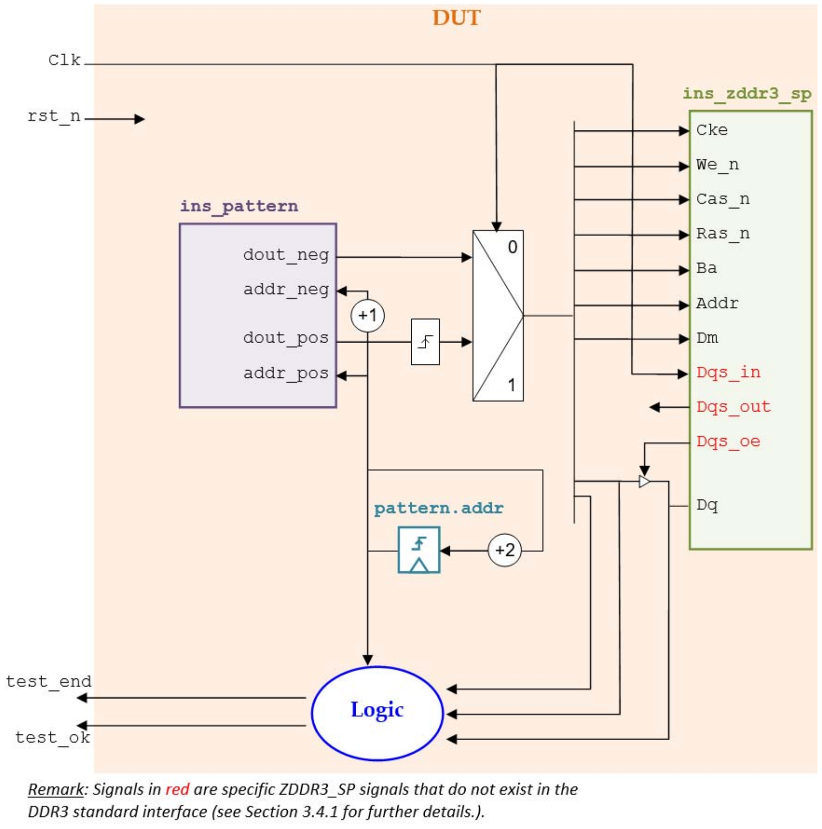
Files Used for Emulation on ZeBu with zRci and User-Defined Tcl Script

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

- testbench files: demo.v
- compilation files:
  - clock_zRun.dve
  - demo.zpf
  - pattern.tcl
  - zTopBuild_option.tcl
- run files:
  - designFeatures
  - pattern.txt (memory settings)
  - pattern.mem (memory settings)
  - zRci.tcl
- automatic flow: Makefile

Description

Figure 10 Tutorial DUT Overview



In the figure above, the DUT instances:

- a ZDDR3_SP Memory Models (ins_zddr3_sp)
- a ZeBu Memory (ins_pattern) which contains commands and data for the ZDDR3_SP Memory.

To validate the correctness of the ZDDR3_SP instance, read data are compared to expected data stored in the ins_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.

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 with the Synopsys VCS simulator.

To compile and run the example:

1. Go to the example/demo_analyzer directory.
2. Launch the Makefile (it contains the compilation and simulation flow):

`make simul_vcs`	HDL simulation with Synopsys VCS.
`make simul`	Executes simul_vcs

Running Emulation with C++ Co-Simulation

To compile and run the example:

1. Go to the example/demo_analyzer directory
2. Export the BENCH_TYPE variable to select the emulation with C++ co-simulation:

```
export BENCH_TYPE=C_COSIM
```

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
```

Running Emulation with zRci and User-Defined Tcl Script

To compile and run the example:

1. Go to the example/demo_analyzer directory
2. Export the BENCH_TYPE variable to select the emulation with C++ co-simulation:

```
export BENCH_TYPE=zRci
```

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.