ZeBu®
LPDDR5 Transactor
–User Manual

V-2024.03-SP1, February 2025

This is an engineering document, which provides technical information to assist with the solution ramp-up.
The comprehensive User Guide for this title will be published in a future release.

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

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

Table of Contents

Figures

Tables

About This Manual

Overview

This manual describes how to use the ZeBu ZLPDDR5 SDRAM Transactor with your design being emulated in ZeBu.

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*.

1 Introduction

1.1 Overview

The ZLPDDR5 transactor memory models are compliant with the LPDDR5 specifications documents issued by Intel (Rev 0.99f, Jan 2020), based on JEDEC discussions and available to JEDEC members (www.jedec.org).

1.2 Features

The ZeBu ZLPDDR5 transactor memory models provide the following features:

- Cycle-accurate SDRAM device model implemented in ZeBu.
- Memory blocks and locations initialization and dump at runtime.
- Includes HDL simulation model for DUT integration testing.
- Bi-directional RTL black box components located in the component directory.
- Bi-directional Verilog RTL wrapper files to include the ZLPDDR5 transactor memory model within the user DUT, located in the wrapper_rtl directory

1.3 ZLPDDR5 Transactor Library

Table 1: ZLPDDR5 Transactor Library

Total Memory Density	Memory Model
2 Gb	xtor_lpddr5_2Gb_x16_svs.sv, xtor_lpddr5_2Gb_x8_svs.sv
3 Gb	xtor_lpddr5_3Gb_x16_svs.sv, xtor_lpddr5_3Gb_x8_svs.sv
4 Gb	xtor_lpddr5_4Gb_x16_svs.sv, xtor_lpddr5_4Gb_x8_svs.sv
6 Gb	xtor_lpddr5_6Gb_x16_svs.sv, xtor_lpddr5_6Gb_x8_svs.sv
8 Gb	xtor_lpddr5_8Gb_x16_svs.sv, xtor_lpddr5_8Gb_x8_svs.sv
16 Gb	xtor_lpddr5_16Gb_x16_svs.sv, xtor_lpddr5_16Gb_x8_svs.sv
24 Gb	xtor_lpddr5_24Gb_x16_svs.sv, xtor_lpddr5_24Gb_x8_svs.sv
32 Gb	xtor_lpddr5_32Gb_x16_svs.sv, xtor_lpddr5_32Gb_x8_svs.sv

Additional ZLPDDR5 transactor memory models can be built and supplied upon request to your local representative, according to your detailed requirements.

1.4 Limitations

The following LPDDR5 operations/features are not supported and are ignored by the current ZLPDDR5 models:

- PPR: Post Package Repair

  Link ECC
- tWCK2CK = -1/2 tWCK
- Differential signals used in single-ended / negative only (CK_c, WCK_c, RDQS_c single_ended)
- Byte mode upper byte

2 Installation

2.1 Installing the ZLPDDR5 Transactor Package

To install the ZeBu ZLPDDR5 Transactor, perform the following steps:

1. Make sure you have Write permission on the IP directory and on the current directory.
2. Download the transactor compressed shell archive (.sh).
3. Install the ZeBu ZLPDDR5 transactor as follows:

```
sh xtor_lpddr5_svs.<version>.sh install [ZEBU_IP_ROOT]
```

[ZEBU_IP_ROOT] is the path to your ZeBu IP root directory:

If no path is specified, the ZEBU_IP_ROOT environment variable is used automatically.

If the path is specified and a ZEBU_IP_ROOT environment variable is also set, the transactor is installed at the defined path and the environment variable is ignored.

The following message is displayed when the installation process is complete and successful:

```
xtor_lpddr5_svs.<version> has been successfully installed.
```

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

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

2.2 Package Structure and Content

Once the ZeBu ZLPDDR5 transactor is installed correctly, it provides the following elements under ZEBU_IP_ROOT/XTOR/xtor_lpddr5_svs.<version> directory:

• lib directory	.so libraries of the transactor
• components directory	Verilog blackbox file of the transactor
• example directory	Testbench, DUT and environment files with the Makefile necessary to run the transactor examples.
• include directory	.hh header files of the transactor
• doc directory	Contains user documentation
• misc directory	Bi-directional interface wrapper files

During installation, symbolic links are created in the following directories for an easy access from all ZeBu tools:

- $ZEBU_IP_ROOT/include
- $ZEBU_IP_ROOT/lib
- $ZEBU_IP_ROOT/vlog

3 Hardware Interface

The following figure describes the ZLPDDR5 transactor hardware interface:

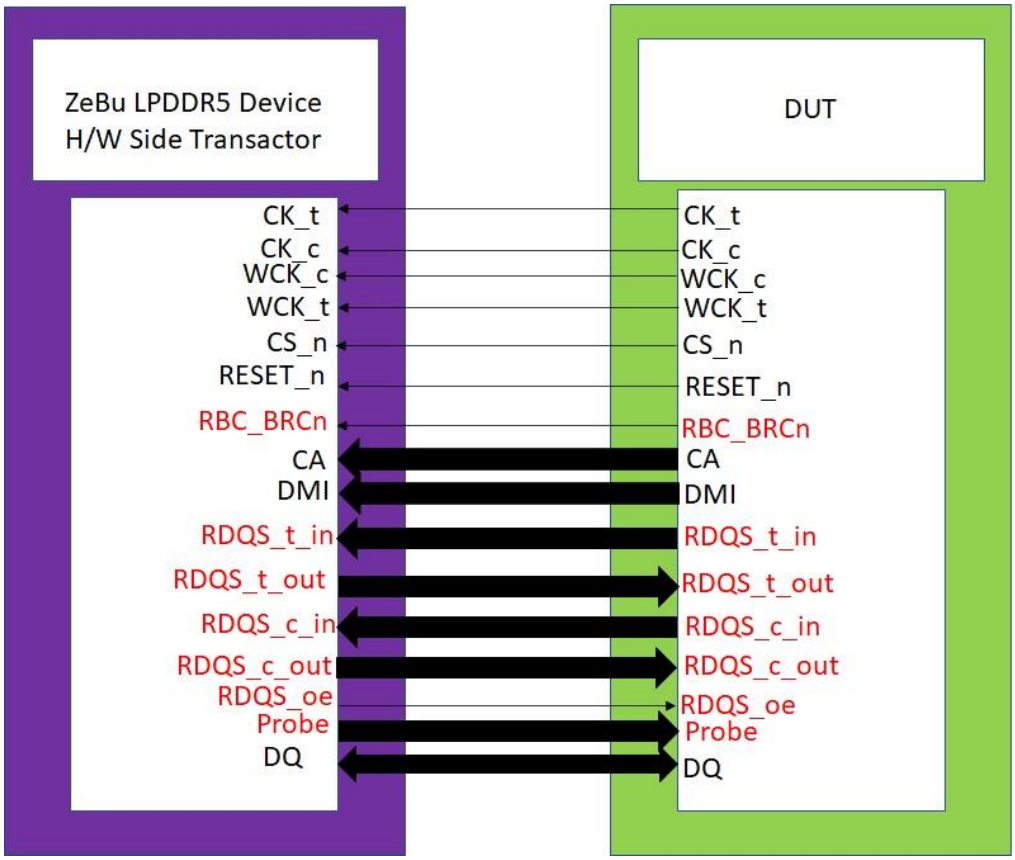


Figure 1: ZLPDDR5 Transactor Interface

Note:

- Signals in red are specific to ZLPDDR5 signals that do not exist in the LPDDR5 standard interface.

3.1 Interface of ZLPDDR5 transactor interface

Table 2: ZLPDDR5 Interface

Name	Type	Description
CK_t CK_c	Input	Clock: CK_t and CK_c are differential clock inputs. All address command and control input signals are sampled on positive edges of CK_t. CK_c (negative) is not used in the ZLPDDR5 model.
WCK_t WCK_c	Input	WCK_t and WCK_c are differential clocks used for WRITE data capture and READ data output.
C2	Input	Chip 2elect: is considered part of the command code.
RE2ET_n	Input	This resets the memory.
CA	Input	Command/Address Inputs.
DQ	I/O	Data Input/Output: Bi-directional data bus.
RDQ2_t RDQ2_c	I/O	Data 2trobe (Bi-directional, Differential): The data strobe is bi-directional (used for READ and WRITE data) and differential (RDQS_t and RDQS_c). It is output with READ data and input with WRITE data. RDQS_t is edge-aligned to READ data and centered with WRITE data.
DMI	I/O	Data Mask(DM) and/or Data Bus Inversion (DBI) according to the mode register configuration for READ/WRITE operations.
ZQ	Input	Calibration reference: Used to calibrate the output drive strength and termination resistance. ZQ is not used in the ZLPDDR5 model

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

3.2 Differences with DDR5 SDRAM Models

This section explains the following key differences in ZLPDDR5 transactor memory models with respect to the LPDDR5 SDRAM memory models:

- LPDDR5 Device Interface Modifications
- LPDDR5 Operations
- LPDDR5 Timing Modeling in ZLPDDR5

3.2.1 LPDDR5 Device Interface Modifications

3.2.1.1 3.2.1.1 RDQS_t_in/RDQS_c_in and RDQS_t_out/RDQS_c_out Ports

The RDQS_t and RDQS_c bi-directional differential ports have been replaced by 4 unidirectional ports:

- RDQS_t_in and RDQS_c_in: inputs to the ZLPDDR5 model
- RDQS_t_out and RDQS_c_out: outputs from the ZLPDDR5 model

Since the RDQS port is used to latch data, this modification allows avoiding gated clocks. For proper use, the original RDQS bidirectional signal should be split into four unidirectional ports inside the LPDDR5 controller mapped in your design.

3.2.2 LPDDR5 Operations

The ZLPDDR5 model is functionally equivalent to the LPDDR5 memory device, but timing requirements are not applicable. The ZLPDDR5 model is accurate up to a half cycle but cannot take into account setup and hold time for example.

All commands and operating modes are accepted by the ZLPDDR5 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 LPDDR5 device datasheets for descriptions of correct operations of the LPDDR5 SDRAM.

3.3 Memory Address Translation

The memory space of the LPDDR5 device is three-dimensional and is organized in banks, rows, and columns. In the ZLPDDR5 memory model, the memory space is flat (bank*row*column depth array of words).

The memory address is decoded from Bank, Row and Column addressing of DDR5.

The RBC_BRCn input is available at the ZLPDDR5 interface to select the memory array addressing mode at compilation time. Two modes are available: BRC for {Bank,Row,Column} addressing

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

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

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

Bank is defined as:

- BK[2:0] in 8 bank mode,
- BK[3:0] in 16 bank mode,
- {BG[1:0],BK[1:0]} in bank group mode.

The Row vector size depends on the memory configuration. Column is defined as

- {C[5:0],B4,B3,3'b000} in 8 bank mode,
- {C[5:0],B3,3'b000} in 16bank or bank group mode.

To change memory addressing:

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

3.3.1 For Binary Density Memory Models (2Gb, 4Gb, 8Gb, 16Gb, 32 Gb Memory Models)

For example, if you want to read your memory in a design using the 2Gbx16, 16 bank mode ZLPDDR5 model:

- Bank=0x1 (4 bits)
- Row = 0x2 (13 bits for 2Gb)
- Col = 0x40 (10 bits = {C5-C0, B3, 3'b000})

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

- BRC Mode:

```
addr[26:0] = {4'b0001,13'b0000000000010,10'b0001000000} =
27'h0800840
```

- RBC Mode:

```
addr[26:0] = {13'b0000000000010,4'b0001,10'b0001000000} =
27'h0008440
```

However, if you want to read your memory in a design using the 8Gbx16, 8 bank mode ZLPDDR5 model:

- Bank=0x7 (3 bits)
- Row = 0x20A (15 bits for 8Gb)
- Col = 0x60 (11 bits = {C5-C0, B4, B3, 3'b000})

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

- BRC mode:

```
addr[28:0] = {3'b1 11,15'b00 0001 0000 0101 0,11'b000 0110 0000} =
29'h13105060
```

- RBC mode:

```
 addr[28:0] = {15'b0 0000 1000 0010 10,3'b11 1,11'b000 0110
0000}= 29'h0082B860
```

3.3.2 For Non-Binary Density Memory Models (3 Gb, 6 Gb, 12Gb, 24 Gb Memory Models)

The 6Gb, 12Gb and 24Gb ZeBu ZLPDDR5 memory models are non-binary density devices. As a consequence, only three quarters of the row address space is valid.When the M2B of row address bit is HIGH, the M2B-1 address bit must be LOW. This has no impact in RBC mode. However in BRC mode, the memory address will be converted to have a linear addressing as follows:

$$zrm\ address = (bank\_addr \times MAX\_ROW\_SIZE \times MAX\_COL\_SIZE) + (row\_addr \times MAX\_COL\_SIZE) + column\_addr$$

Example

Assume that you want to read your memory in a design using the 6Gbx16 16 bank mode, ZLPDDR5 model with:

- Bank=1 (4 bits)
- Row=2 (15 bits)
- Column=4 (7 bits)
- `MAX_ROW_SIZE` = 2^15*3/4 = 24576 for this model `MAX_COL_SIZE` = 2^7 = 128 for this model

In this case, the starting address for zrm (in hexadecimal) should be as follows:

- BRC Mode:

```
addr[25:0] = (1*24576*128) + 2*128 +4 = 26'h300104
```

- RBC Mode:

```
addr[25:0] = {0000000000000010,0001,0000100} = 16'h0001084
```

The following figure illustrates a 6Gb ZLPDDR5 Models with 15-bit Row Address

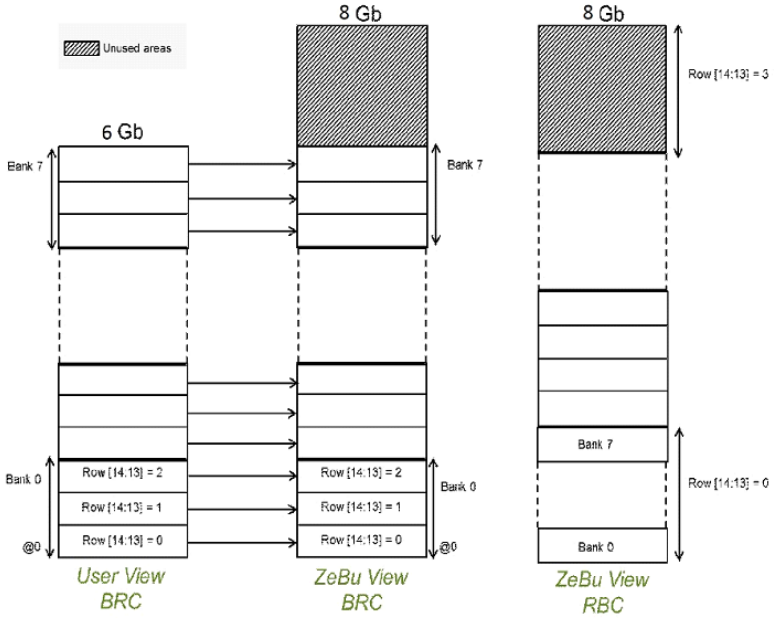


Figure 2: 6Gb ZLPDDR5 Models with 15-bit Row Address

4 Software Interface

4.1 Class Description

The ZLPDDR5 transactor is instantiated and accessed using the ZLPDDR5 Xtor C++ class, defined in the $ZEBU_IP_ROOT/include/xtor_lpddr5_svs.hh file.

The ZLPDDR5 transactor API is included in the ZEBU_IP::XTOR_DRAMSW_SVS namespace. The following header files are used to invoke a ZeBu ZLPDDR5 transactor:

- xtor_ddr5_svs.<version>.hh describes the ZLPDDR5Xtor class.
- xtor_ddr5_struct_svs.<version>.hh describes the different structures used by the ZLPDDR5Xtor class. This file is automatically included by the xtor_DDR5_svs.<version>.hh file

Example:
A typical testbench starts with the following lines:

```
#include "xtor_ddr5_svs.hh"
using namespace ZEBU_IP;
using namespace XTOR_DRAMSW_SVS;
```

4.2 Method List

The xtor_lpddr5_svs class provides the methods described in the following table:

Table 3: ZLPDDR5 Transactor Class Methods

Method	Description
`xtor_lpddr5_svs`	Constructor.
`~xtor_lpddr5_svs`	Destructor.
ZLPDDR5 Transactor Initialization	
`init`	Connects the ZLPDDR5 transactor to the ZeBu System.
`config`	Configures the ZLPDDR5 transactor.
ZLPDDR5 Transactor Control	
`start`	Starts the hardware part of transactor, enable DUT to access to memory.
`stop`	Stops the hardware part of transactor, disable DUT to access to memory.
ZLPDDR5 Transactor's Memory Word Access	
`readWord`	Get the value of a memory word at the address to

	be accessed.
writeWord	Set the value of a memory word at the address to be accessed.
storeTo(buffer)	Copy the content of the memory into the buffer.
storeTo(file)	Copy the content of the buffer into the memory.
loadFrom(buffer)	Copy the content of buffer to a memory block modelled by the ZLPDDR5 transactor.
loadFrom(file)	Copy the content of file to a memory block modelled by the ZLPDDR5 Transactor.
readWordBlock	Copy the content of the memory into the buffer.
writeWordBlock	Copy the content of the buffer into the memory.
writeWordBlockByteEn	Write the content of the buffer into the memory per Word with WordEnable.
set(full)/ set(part)	Set part of the memory with a pattern per word.
clear	Clear the content of the memory.

ZLPDDR5 Transactor's Memory Byte Access

getSizeBytes	Return the number of Bytes of the memory instance.
readByte	Read the value of a memory byte.
writeByte	Write the value of a memory byte.
readByteBlock	Read block of byte of the memory into the buffer.
writeByteBlock	Write the content of the buffer into the memory.
writeByteBlockwithBE	Write the content of the Bytebuffer into the memory with ByteEnable.
fillBytes	Set part of the memory with a pattern.
readLogicByte	Read the value of a memory byte.
writeLogicByte	Write the value of a memory byte.
readLogicByteBlock	Read block of byte of the memory into the buffer.
writeLogicByteBlock	Write the content of the buffer into the memory.
writeLogicByteBlockwithBE	Write the content of the Bytebuffer into the memory with ByteEnable.
fillLogicBytes	Set part of the memory with a pattern.

ZLPDDR5 Transactor's Settings and Statuses

getNumberOfFlushedLines	Return the number of lines flushed by hardware part.
getNumberOfAccesses	Return the number of accesses by hardware part.

`getNumberOfReadLines`	Return the number of lines read by the hardware part.
`resetStatistics`	Reset the statistics counter of the transactor.
`getDepth`	Return the number of words of the memory instance.
`getWidth`	Return the number of bits per memory word.
`getCycle`	Return the last memory cycle seen by the memory transactor.
`exitOnError`	Allow transactor to throw an exception when an error occurs.
`setTimeout`	set watchdogs timeout value.
`enableWatchdog`	enable or disable ZLPDDR5 watchdogs.

ZLPDDR5 Transactor's Log Settings

`setDebugLevel`	Sets the messages debug level.
`setLog`	Sets the messages log filename or file stream.
`printStaticstics`	Display the statistics of transactor in the VSLog.

ZLPDDR5 Transactor's Configurable Mode Register

`setZebuMR`	Set the value in the zebuMR register.
`getZebuMR`	Get the value of the zebuMR register.
`getZebuMRSize`	Get the size in bit of the zebuMR register.

4.3 The LPDDR5 Transactor Initialization

The following table gives an overview of the transactor initialization methods.

Table 4: List of Methods for Transactor Initialization

Method	Description
init	Connects the ZLPDDR5 transactor to the ZeBu System

4.3.1 init() Method

This method connects the ZLPDDR5 to the ZeBu system.

```
int init (Board *zebu_board,
          const char *driverName)
```

where:

- `board` is the pointer to the ZeBu board.
- `driverName` is the name of the transactor instance

4.4 ZLPDDR5 Transactor Control

Any ZLPDDR5 must implement functions to start and stop the DUT to access memory.

The following table gives an overview of the ZLPDDR5 transactor memory accessing start/stop methods.

Table 5: List of Methods for Transactor Control

Method	Description
start	Starts the hardware part of transactor, enable DUT to access to memory.
stop	Stops the hardware part of transactor, disable DUT to access to memory.

4.4.1 start() Method

Activates the ZLPDDR5 xtor transactor, enabling access to the modelled memory with memory management methods of the ZLPDDR5 transactor class.

```
bool start (void)
```

4.4.2 stop() Method

Stop the ZLPDDR5 transactor, disable any access to the modelled memory.

```
bool stop (void)
```

4.5 ZLPDDR5 Transactor's Memory Word Access

The following table gives an overview of the transactor interface methods for accessing the ZLPDDR5 modelled memory in word format.

Table 6: List of Methods for ZLPDDR5 Transactor's Memory Word Access

Method	Description
`readWord`	Get the value of a memory word at the address to be accessed.
`writeWord`	Set the value of a memory word at the address to be accessed.
`storeTo(buffer)`	Copy the content of the memory into the buffer.
`storeTo(file)`	Copy the content of the buffer into the memory.
`loadFrom(buffer)`	Copy the content of buffer to a memory block modelled by the ZLPDDR5 Transactor.
`loadFrom(file)`	Copy the content of file to a memory block modelled by the ZLPDDR5 Transactor.
`readWordBlock`	Copy the content of the memory into the buffer.
`writeWordBlock`	Copy the content of the buffer into the memory.
`writeWordBlockByteEn`	Write the content of the buffer into the memory per Word with WordEnable.
`set(full)/ set(part)`	Set part of the memory with a pattern per word.
`clear`	Clear the content of the memory.

4.5.1 readWord() Method

This method read the value of a memory word modelled by the ZLPDDR5 Transactor and copy it to a `wordBuffer` variable.

```
bool readWord (const uint64_t address, unsigned char* wordBuffer,
               int channel)
```

where:

- `address` is the address of memory location to be accessed.
- `wordBuffer` is the buffer to store the value of the memory word.
- `channel` is the Channel of memory which must be accessed. Default value is 0.
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.5.2 writeWord() Method

This method copies the value of wordBuffer to a memory word modelled by the ZLPDDR5 Transactor.

```
bool writeWord (const uint64_t address, unsigned char* wordBuffer,
                int channel)
```

where:

- `address` is the address of memory location to be accessed.
- `wordBuffer` is the buffer to store the value of the memory word.
- `channel` is the Channel of memory which must be accessed. Default value is 0.
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.5.3 storeTo(buffer) Method

This method read the value of a memory block modelled by the ZLPDDR5 Transactor and copy it to a buffer.

```
void storeTo(unsigned int *buffer, const uint64_t adr_beg,
             const unsigned int numberOfWords, int channel)
```

where:

- `buffer` is the buffer to store the content of the memory.
- `adr_beg` is the first address of memory location to be dumped.
- `numberOfWords` are the number of words to be dumbed from memory.
- `channel` is the Channel of memory which must be accessed. Default value is 0.

4.5.4 storeTo(file) Method

This method read the value of a memory block modelled by the ZLPDDR5 Transactor and copy it to a file.

```
bool storeTo(const char *fname, const uint64_t adr_beg,
             const uint64_t adr_end, unsigned int channel = 0,
             const bool binary = false, bool compression = false)
```

where:

- `fname` is the file to store the content of the memory.
- `adr_beg` is the first address of memory location to be dumped.
- `adr_end` is the last address of memory location to be dumped.
- `channel` is the Channel of memory which must be accessed. Default value is 0.

4.5.5 loadFrom(buffer) Method

This method copied the content of buffer to a memory block modelled by the ZLPDDR5
Transactor.

```
void loadFrom(const uint64_t adr_beg, const unsigned int
              numberOfWords, const unsigned int *buffer,
              int channel)
```

where:

- `adr_beg` is the first address of memory location where content of buffer to be load.
- `numberOfWords` are the number of words to be copy from buffer.
- `buffer` is the buffer from where content to be copy.
- `channel` is the Channel of memory which must be accessed. Default value is 0.

4.5.6 loadFrom(file) Method

This method copies the content of file to a memory block modelled by the ZLPDDR5 Transactor.

```
bool loadFrom(const uint64_t adr_beg, const uint64_t adr_end,
              const char *fname, FileType_t ftype,
              unsigned int channel=0, bool compression = false)
```

where:

- `adr_beg` is the Beginning address of memory to be loaded.
- `adr_end` is the Ending address of memory to be loaded.
- `fname` is the file name from where content needs to be copied.
- `FIleType_t` is the Format of load file text or binary (Ftext/FBin).
- `channel` is the Channel of memory which must be accessed. Default value is 0.

4.5.7 readWordBlock() Method

This method read the value of a memory block modelled by the ZLPDDR5 Transactor and copy it to a buffer and very similar to `storeTo()` method.

```
bool readWordBlock(unsigned int *buffer,const uint64_t adr_beg,
                   const unsigned int numberOfWords,
                   int channel)
```

where:

- `buffer` is the buffer to store the content of the memory.
- `adr_beg` is the first address of memory location to be dumped.
- `channel` is the Channel of memory which must be accessed. Default value is 0.
- `numberOfWords` are the number of words to be dumbed from memory.
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.5.8 writeWordBlock() Method

This method copies the content of buffer to a memory block modelled by the ZLPDDR5 transactor and very similar to `loadFrom()` method.

```
bool writeWordBlock(const uint64_t adr_beg,
                    const unsigned int numberOfWords,
                    const unsigned int *buffer, int channel)
```

where:

- `adr_beg` is the first address of memory location where content of buffer to be load.
- `numberOfWords` are the number of words to be copy from buffer.
- `buffer` is the buffer from where content to be copy.
- `channel` is the channel type of memory which must be accessed.
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.5.9 writeWordBlockByteEn() Method

This method writes the content of the buffer into the memory per word with WordEnable.

```
bool writeWordBlockByteEn(const uint64_t adr_beg,
                          const unsigned int numberOfWords,
                          const unsigned char *buffer,
                          const unsigned char *be,
                          int channel)
```

where:

- `adr_beg` is the first address of memory location where content of buffer to be load.
- `numberOfWords` are the number of words to be copy from buffer.
- `buffer` is the buffer from where content to be copy.
- `be` buffer to store the ByteEnable content to mask the write word value
  - `0xFF`: enable
  - `0x00`: disable
- `channel` is the Channel of memory which must be accessed. Default value is 0
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.5.10 set(full) Method

This method set the content with pattern of selected channel of memory modelled by the ZLPDDR5 Transactor.

```
bool set(const unsigned char *pattern, int channel)
```

where:

- `pattern` is the pattern to be set
- `channel` is the Channel of memory which must be accessed. Default value is 0.

4.5.11 set(part) Method

This method set the content with pattern of selected part of selected channel of memory modelled by the ZLPDDR5 Transactor.

```
bool set(const uint64_t adr_beg, const uint64_t adr_end,
         const unsigned char *pattern, int channel)
```

where:

- `adr_beg` is the first address of memory location where content of buffer to be set.
- `adr_end` is the last address of memory location where content of buffer to be set.
- `pattern` is the pattern to be set
- `channel` is the Channel of memory which must be accessed. Default value is 0.
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.5.12 clear() Method

This method clears the content of selected channel of memory modelled by the ZLPDDR5
Transactor.

```
void clear()
```

4.6 ZLPDDR5 Transactor's Byte Memory Access

The following table gives an overview of the ZLPDDR5 transactor's memory access methods at byte-level.

Table 7: List of Methods for ZLPDDR5 Xtor Byte Memory Access

Method	Description
`getSizeBytes`	Return the number of Bytes of the memory instance.
`readByte`	Read the value of a memory byte.
`writeByte`	Write the value of a memory byte.
`readByteBlock`	Read block of byte of the memory into the buffer.
`writeByteBlock`	Write the content of the buffer into the memory.
`writeByteBlockwithBE`	Write the content of the Bytebuffer into the memory with ByteEnable.
`fillBytes`	Set part of the memory with a pattern.

4.6.1 getSizeBytes() Method

This method returns the number of Bytes of the memory instance.

```
uint64_t getSizeBytes()
```

4.6.2 readByte() Method

This method read the value of a memory byte modelled by the ZLPDDR5 Transactor and
copies it to a byteBuffer variable.

```
bool readByte(const uint64_t byte_address, uint8_t &byteBuffer,
              int channel)
```

where:

- `address` is the address of memory location to be accessed.
- `byteBuffer` is the buffer to store the value of the memory byte.
- `channel` is the Channel of memory which must be accessed. Default value is 0.
- `bool` status:
  - `true`: ok

- `false`: out of range

4.6.3 writeByte() Method

This method copies the content of byteBuffer and write the value to a memory byte modelled by the ZLPDDR5 Transactor.

```
bool writeByte(const uint64_t byte_address, uint8_t &byteBuffer,
               int channel)
```

where:

- `address` is the address of memory location to be accessed.
- `byteBuffer` is the buffer to copy the value to the memory byte.
- `channel` is the Channel of memory which must be accessed. Default value is 0.
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.6.4 readByteBlock() Method

This method read the memory block of bytes modelled by the ZLPDDR5 Transactor and
copy it to a buffer.

```
bool readByteBlock(uint8_t *buffer,const uint64_t byte_adr_beg,
                   const unsigned int numberOfBytes,
                   int channel)
```

where:

- `buffer` is the buffer to store the content of the memory.
- `byte_adr_beg` is the first address of memory location to be dumped.
- `numberOfBytes` are the number of bytes to be dumbed from memory.
- `channel` is the Channel of memory which must be accessed. Default value is 0.
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.6.5 writeByteBlock() Method

This method copies a block of bytes from the buffer and write it to memory modelled by the ZLPDDR5 Transactor.

```
bool writeByteBlock(uint8_t *buffer, const uint64_t byte_adr_beg,
                    const unsigned int numberOfBytes,
                    int channel)
```

where:

- `buffer` is the buffer to copy the content of the memory.

- `byte_adr_beg` is the first address of memory location to be write.
- `numberOfBytes` are the number of bytes to be write in memory.
- `channel` is the Channel of memory which must be accessed. Default value is 0.
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.6.6 writeByteBlockwithBE() Method

This method writes the content of the buffer into the memory per byte with `ByteEnable`.

```
bool writeByteBlockwithBE(const uint64_t byte_adr_beg,
                          const unsigned int numberOfBytes,
                          uint8_t *Bytebuffer,
                          uint8_t *BEbuffer,
                          int channel)
```

where:

- `byte_adr_beg` is the first address of memory location where content of buffer to be load.
- `numberOfBytes` are the number of bytes to be copy from buffer.
- `Bytebuffer` is the buffer from where content to be copy.
- `BEbuffer` buffer to store the ByteEnable content to mask the byte value
  - `0xFF`: enable
  - `0x00`: disable
- `channel` is the channel type of memory which must be accessed.
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.6.7 fillBytes() Method

This method fills selected part of selected channel of memory modelled by the ZLPDDR5
Transactor with a designated pattern.

```
bool fillBytes(const uint64_t byte_adr_beg,
               const uint64_t byte_adr_end,
               const unsigned char pattern, int channel)
```

where:

- `byte_adr_beg` is the first address of memory location from where memory to be filled with pattern.
- `byte_adr_end` is the last address of memory location to be filled with pattern.
- `pattern` is the pattern to be set
- `channel` is the Channel of memory which must be accessed. Default value is 0.
- `bool` status:
  - `true`: ok
  - `false`: out of range

4.7 ZLPDDR5 Transactor's Settings and Statuses

The following table gives an overview of the ZLPDDR5 transactor settings and status methods.

Table 8: List of Methods for ZLPDDR5 Transactor Settings and Statuses

Method	Description
`getNumberOfFlushedLines`	Return the number of lines flushed by hardware part.
`getNumberOfAccesses`	Return the number of accesses by hardware part.
`getNumberOfReadLines`	Return the number of lines read by the hardware part.
`getDepth`	Return the number of words of the memory instance.
`getWidth`	Return the number of bits per memory word.
`getCycle`	Return the last memory cycle seen by the memory transactor.
`resetStaticstics`	Reset the statistics counter of transactor.
`printStaticstics`	Display the statistics of transactor in the VSLog.

.

4.7.1 getNumberOfFlushedLines() Method

Return the number of line flushed by the hardware part of transactor.

```
unsigned int getNumberOfFlushedLines(void)
```

4.7.2 getNumberOfAccesses() Method

Return the number of accesses by the hardware part of transactor.

```
unsigned int getNumberOfAccesses(void)
```

4.7.3 getNumberOfReadLines() Method

Return the number of line read by the hardware part of transactor.

```
unsigned int getNumberOfReadLines(void)
```

4.7.4 getDepth() Method

This method gets the information about the size of the memory and returns the number of words of the memory instance.

```
uint64_t getDepth(void)
```

4.7.5 getWidth() Method

This method gets the information about the size of the memory and returns the number of bits per memory word.

```
unsigned int getWidth(void)
```

4.7.6 getCycle() Method

This method gets the information about the size of the memory and returns the last memory cycle seen by the memory transactor.

```
uint64_t getCycle(void)
```

4.7.7 resetStaticstics() Method

This method reset the statistics counter of the transactor.

```
void resetStatistics(void)
```

4.7.8 printStaticstics() Method

This method displays the statistics of the transactor in the VSLog.

```
void printStatistics(bool duplicateonstdout)
```

4.8 ZLPDDR5 Transactor's Log Settings

The following table gives an overview of the methods for ZLPDDR5 transactor's log management.

Table 9: List of Methods for ZLPDDR5 Transactor's Log

Method	Description
`setDebugLevel`	Sets the messages debug level.
`setLog`	Sets the messages log filename or file stream.

4.8.1 setDebugLevel() Method

This method sets the debug information level for the printed debug messages. The higher the debug level, the more debug messages are displayed.

```
void setDebugLevel (uint lvl);
```

where `lvl` is the maximum level of displayed debug messages:

- `0`: No debug messages
- `1`: Main steps of the testbench processing (transactor settings, connection, etc.)
- `2`: Level 1 information with dump of sent/received data
- `3`: Level 2 information with low-level information (transactor messages content, service loop)

4.8.2 setLog() Method

This method activates and sets parameters for the transactor's log generation.
The log contains transactor's debug and information messages which is output into a log file that is defined with a file descriptor or by a filename.
The log file is closed upon the ZLPDDR5 transactor object destruction.

4.8.2.1 Log File Assigned through a File Descriptor

The log file where to output messages is assigned through a file descriptor:

```
bool setLog (FILE *stream, bool stdoutDup);
```

where:

- `stream` is the output stream (file descriptor).
- `stdoutDup` is the output mode:
  - `true`: messages are output both to the file and the standard output.
  - `false` (default): messages are only output to the file.

4.8.2.2 Log File Defined by a Filename

The log file where to output messages is defined by its filename, as shown below:

```
bool setLog (char *fname, bool stdoutDup);
```

where:

- `fname` is the name of the log file.
- `stdoutDup` is the output mode:
  - `true`: messages are output both to the file and the standard output.
  - `false` (default): messages are only output to the file.

The method returns:

- `true` upon success.
- `false` if the specified log file cannot be overwritten or if the method failed in creating the file.

4.9 Configurable Mode Register

The ZLPDDR5 memory models have a common register (zebubR_common) and a specific register (zebuMR) for each channel. This architecture is common to all ZeBu memory IPs. However, ZeBu LPDDR5 memory models have only one channel.

4.9.1 `zebuMR_common` Register

Each instance of the ZLPDDR5 model has a zebuMR_common register of this instance. It is divided into several Mode Registers (MR), each one corresponding to the specific information.

The following figure describes the common zebuMR_common register content:

Bits	31 26	25 24	23 16	15 8	7 0
MR		MR8[7:6]	MR7[7:0]	MR6[7:0]	MR5[7:0]
Default Value		0	0	0x0	0xff
Information	RFU	Type	Revision ID2	Revision ID1	Manufacturer ID

Figure 3: zebuMR_common Mapping with Default Values

4.9.2 zeBuMR Register

Each instance of the ZLPDDR5 model has one zebuMR register.

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

The following table describes the zebuMR register content:

bits	127 40	39 36	35 32	31 30	29 25	24 21	20 13	12 8	7 4	3 0
		tWCK2DQO	tWCK2DQI	tWCK2CK		MR10[7:4]	MR4[7:0]	MR3[7:3]	MR2[3:0]	MR1[7:4]
default value		0	0	0		0	0x0	0x0	0x0	0x0
information	RFU				RFU	RDQS_PST, RDQS_PRE	Refresh Rate	DBI, WLS, BK/BG ORG	Read Latency	Write Latency

Figure 4: zebuMR Mapping with Default Values

The following table gives an overview of the ZLPDDR5 transactor's configurable mode register:

Table 10: List of Methods for ZLPDDR5 Transactor's Configurable Mode Register

Method	Description
`setZebuMR`	Set the value in zebuMR register
`getZebuMR`	Get the value of zebuMR register
`setZebuMRCommon`	Set the value in zebuMR Common register
`getZebuMRCommon`	Get the value of zebuMR Common register
`getZebuMRSize`	Get the size in bit of zebuMR register
`getZebuMRCommonSize`	Get the size in bit of zebuMR Common register

4.9.3 setZebuMR() Method

This method sets the value of zebuMR register and return true if successful, else false.

```
bool setZebuMR (uint32_t value)
```

Where:

- `value` is the value to be set.

4.9.4 getZebuMR() Method

This method gets the value of zebuMR register.

```
uint32_t getZebuMR (uint32_t &value)
```

Where:

- `value` is the value to be read.

4.9.5 getZebuMRSize() Method

This method gets the size of zebuMR register in term of bit.

```
uint32_t getZebuMRSize ()
```

4.9.6 setZebuMRCommon() Method

This method sets the value of zebuMRCommon register and returns true if successful, else false.

```
bool setZebuMRCommon (uint32_t value)
```

Where:

- `value` is the value to be set.

4.9.7 getZebuMRCommon() Method

This method gets the value of zebuMRCommon register.

```
uint32_t getZebuMRCommon (uint32_t &value)
```

Where:

- `value` is the value to be read.

4.9.8 getZebuMRCommonSize() Method

This method gets the size of zebuMRCommonSize register in term of bit.

```
uint32_t getZebuMRCommonSize ()
```

5 Debug Information

For debugging purposes, a list of signals is available at runtime to trace the internal behavior of ZLPDDR5 models (see *probe Signals*).

Besides, an alias file for Verdi is also provided in the package. It aims at facilitating decoding these signals (see *Alias Files for Verdi™*).

5.1 Probe Signals

A trace vector called probe[97:0] is available on interface. It contains the necessary information to analyze the behavior of the models:

TABLE 5 Probe Signals

Signal	Description
probe[73:60]	reserved
probe[59:52]	freq (MR49)
probe[51:46]	rl_real
probe[45:40]	wl_real
probe[39:36]	tWCQ2DQO
probe[35:32]	tWCK2DQI
probe[31]	b1_32
probe[30:29]	fsp_op
probe[28:27]	fsp_wr
probe[26]	dm_disable
probe[25]	dbi_wr
probe[24]	dbi_rd
probe[23:22]	rd_pre

Signal	Description
probe[21:20]	rd_post
probe[19]	8 BK mode
probe[18]	16 BK mode
probe[17]	4BK/4BG mode
probe[16]	wls_set_B
probe[15]	cbt mode.
probe[14]	Clock ratio 1:2.
probe[13]	command READ (all types).
probe[12]	Command WRITE (all types).
probe[11]	Command ACT (ACT1 + ACT2 sequence).
probe[10]	Command MRR.
probe[9]	Command MRW.
probe[8]	Command PRE.
probe[7]	Command MPC.
probe[6]	Command RFF.
probe[5]	Command WFF.
probe[4]	Command RDC.
probe[3:0]	Current state of ZLPDDR5. ● IDLE = 0 ● MRR = 1 ● MRW = 2 ● ACT =3 (ACT means ACT2 following ACT1, no specific state for ACT1) ● READ = 4 ● WRITE = 5 ● TRAIN = 6 (CBT or WR leveling) ● RDC = 7 ● RFF = 8 ● WFF = 9

5.2 Alias Files for Verdi™

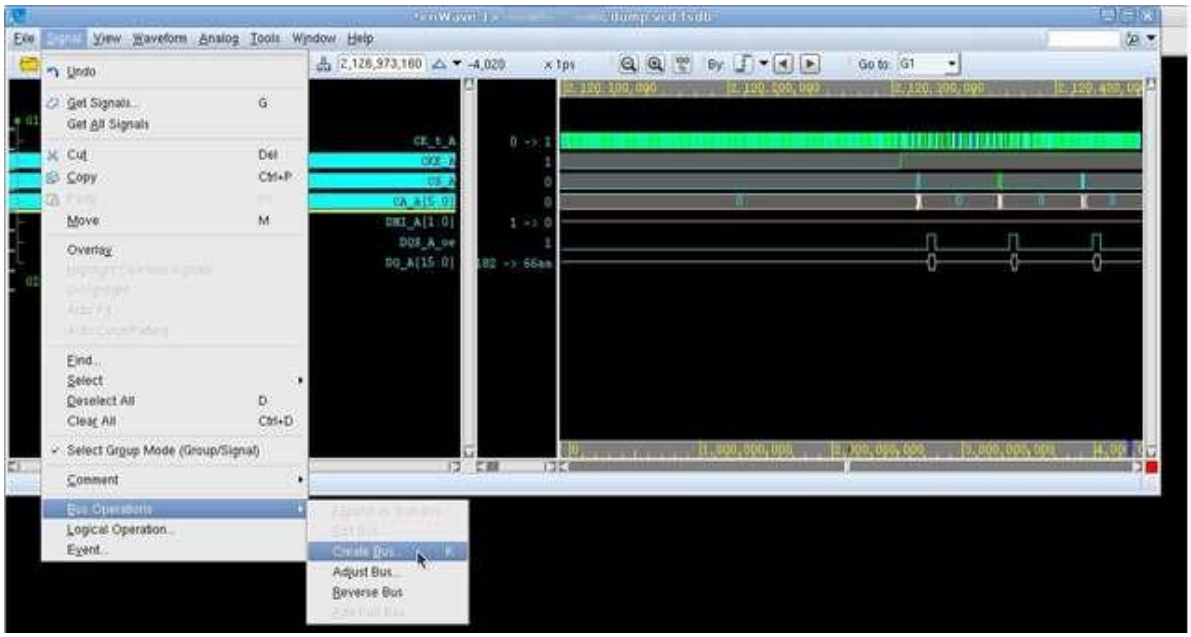
The ZLPDDR5 package provides the following two alias files in the misc/script/nWave directory to facilitate debug in Verdi:

cmd.alias probe.alias

5.2.1 cmd.alias

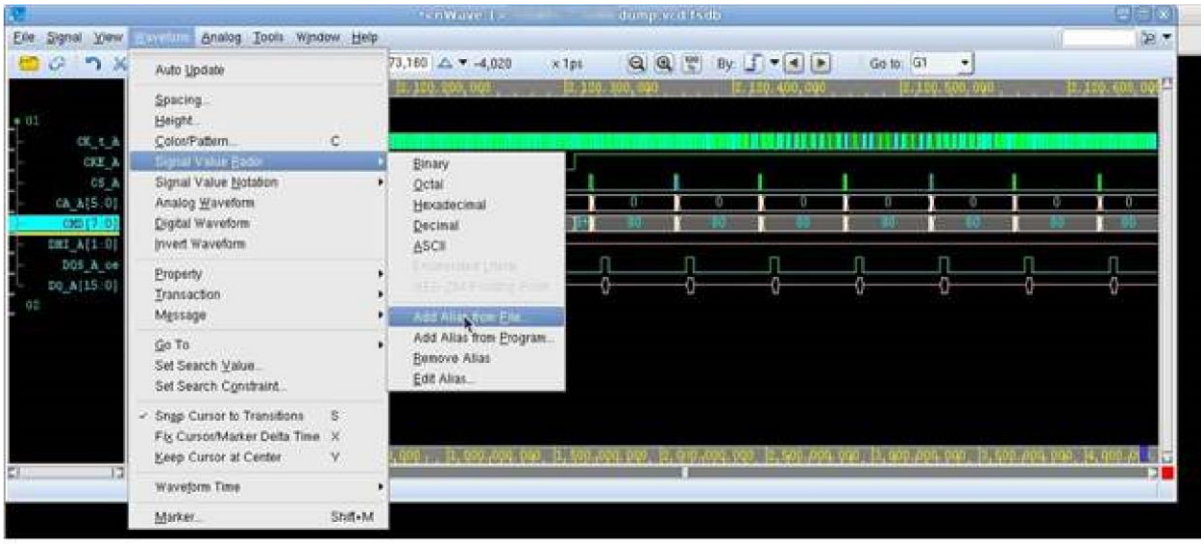
Allows you to automatically interpret the memory interface signals To use the cmd.alias file, perform the following steps:

1. In **nWave**, select **Signal > Bus Operations > Create Bus** and create a new 8-bit width bus: from CK_t, CS and CA[6:0] in MSB to LSB name it CMD for example

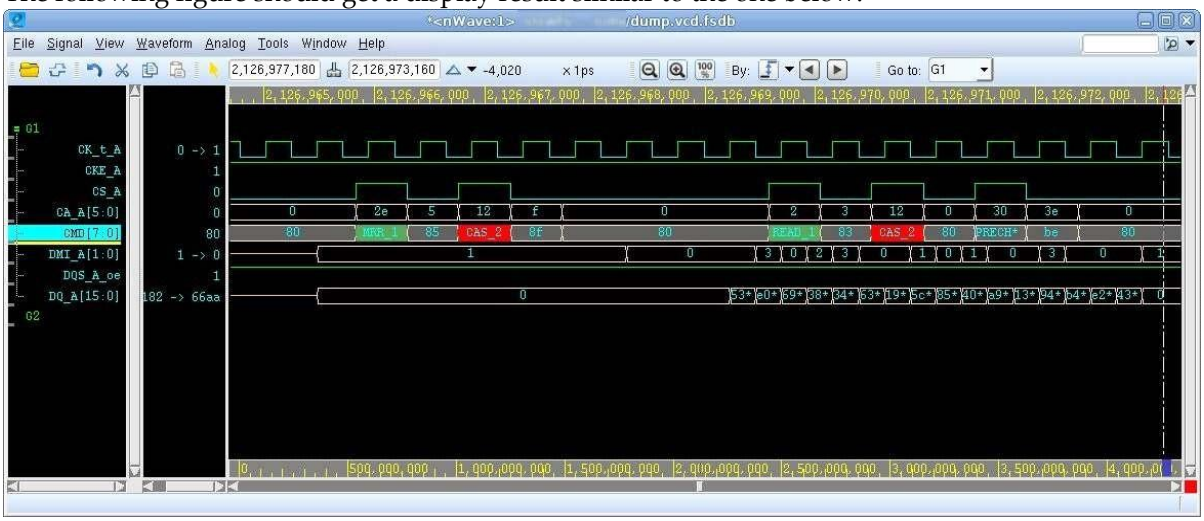


2. Select the created bus vector (named CMD in the previous step).

3. Assign the cmdealias file to it by selecting **Waveforms > Signal Value Radix > Add Alias from File**:



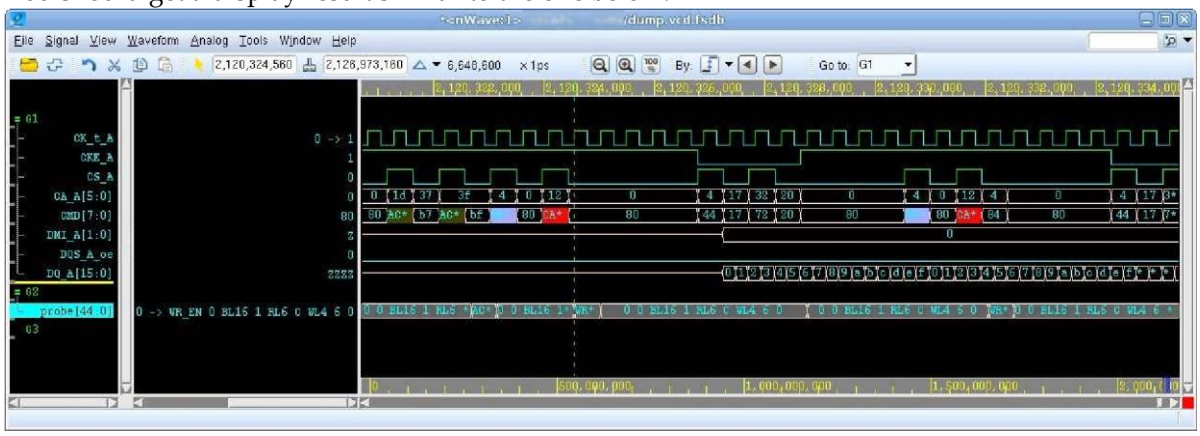
The following figure should get a display result similar to the one below:



5.2.2 Probe.alias

To use the probe.alias file, assign the probeealias file to it by selecting **Waveforms > Signal Value Radix > Add Alias from File.**

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



6 Tutorial

The ZLPDDR5 package provides an example, which describes how to use the ZLPDDR5 transactor Memory Models.

6.1 Prerequisite

Before running the tutorial/example, ensure that the following environment variables are correctly set:

ZEBU_ROOT: Specify a valid ZeBu installation.

ZEBU_IP_ROOT: Specify the path to the package installation directory.

6.2 Running the Tutorial

Perform the following steps to compile and run the example:

1. Go to the example/zebu directory.
2. Launch the compilation flow with the Makefile:

```
make compil
```

3. Launch the emulation flow with the Makefile:

```
make run
```

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

Examples

```
make clean
```