The Fastest Verification

ZeBu™ Application Note

Importing Memories for Easy Runtime Access

Document revision – a –

March 2009

Version 3.1_x / 4.2_x / 4.3_x / 5.1_x

AN021

Copyright Notice
Proprietary Information

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

Figures

About This Manual

Overview

This Application Note describes the memory import process which allows easy runtime access to design memories based on Xilinx FPGA memory resources (BRAM-based, LUT-based or register-based memories), independently of the physical implementation of the memory.

Note that memory import operation is not necessary if you created a memory model with **`zMem`**.

History

This table gives information about the content of each revision of this document, with indication of specific applicable version:

Doc Revision	Product Version	Date	Evolution
a	3.1_2 4.2_x 4.3_x 5.1_x	Mar 09	First Edition.

Related Documentation

The following document provides additional information related to the present application note:

- The *ZeBu Compilation Manual* contains detailed information on memory modeling, in particular with **`zMem`** for BRAM-based and zrm-based memory models.

Typographic Conventions Used in This Manual

ZeBu tools are shown in bold, mono-space "Courier New" font, e.g. **`zBuild`**.

Input file contents, such as code examples, scripts or driver declarations are shown in mono-space "Courier New" font with left and bottom borders.
For example, the following line of a script describes a memory command that accepts a user-declared variable and a user-selected option:

```
memory set_rw_mode <port-name> [rw-mode]
```

Where:

`memory set_rw_mode`	is a command.
`<port-name>`	User-declared variables are inside angled brackets ("<" and ">"), such as a port name in the present example.
`[rw-mode]`	Options are presented inside square brackets. The available options will be listed and described in the text following the syntax rules.

Shell command lines are shown in mono-space "Courier New" font with left and bottom borders, including the shell prompt (which depends on your configuration); the command itself may be shown in bold characters for easier reading. :

```
$ zBuild <script_file_name>
```

Parameters and options are displayed in the same way as for input files contents: parameters inside angles brackets and options inside square brackets.

The shell prompt in this manual is $ for user environment and # for supervisor environment.

Reports, logs and any output data (except Tcl scripts), generated by ZeBu tools are shown in mono-space "Courier New" font with complete border. The following example is a log header:

```
##############################################
# Copyright (c) 2002-2009                      #
# Emulation and Verification Engineering SA  #
#--------------------------------------------#
# <Tool Name>                                #
# revision :                                 #
# date :                                     #
#--------------------------------------------#
##############################################
```

GUI elements (menu items, buttons, check boxes, edition fields) are shown in bold characters. Following is an example of GUI description:

> In the **Generate** menu, select the appropriate wrapper type to generate for the probes and then select **Generate**.

1 Introduction

1.1 Description

Using **zDbPostProc**, you can import memories from Xilinx physical memory resources (BlockRAMs, LUTRAMs or registers) to create a logical memory model in the runtime database. Note that memory import operation is not necessary if you created a BRAM-based memory with **zMem**.

The resulting memory models are accessible at runtime as logical resources via **zRun** or the co-simulation APIs.

As for probes selection, **zDbPostProc** requires the complete run database `ZeBuDB.zdb`, available once the ZeBu compilation has been completed.

The memory import operation uses an input file called the Memory Import File which contains the description of the physical memory resources and of the logical memory mapping.

1.2 Supported Physical Resources for Memory Import

To proceed with memory import, you can use any type of physical Xilinx memory resources: BRAMs, LUTRAMs or registers instantiated in your design. However, a single memory model can only be created with memory import from homogeneous physical memory resources: only BRAM primitives, or only LUTRAM primitives or only registers.

The following Xilinx primitives can be used as physical banks for memory import:

- Supported BRAM primitives: `ramb16_s36*`, `ramb16_s18*`, `ramb16_s9*`, `ramb16_s4*`, `ramb16_s2*`, `ramb16_s1*` where `*` indicates an optional second port on the BRAM.
  Note that BRAM memory import supports only the first port width in the primitive name: an optional second port is ignored (for example, the `ramb16_s1_s2` which can be seen as 1-bit or 2-bit wide will only be considered as 1-bit wide for import).
- Supported LUTRAM primitives: `ram16x1`, `ram16x2`, `ram16x4`, `ram16x8`, `ram32x1`, `ram32x2`, `ram32x4`, `ram32x8`, `ram64x1`, `ram64x2`, `ram128x1`. Note that for ZeBu-UF, LUTRAM-based memory models do not support write-access.

Any register or register vector of the design can be used for memory import.

1.3 Important Rules

Some rules must be respected in order to successfully import memories:

- A design memory must be composed of a unique type of physical resources (BRAMs, LUTRAMs or registers).
- A design memory must be fully implemented by one physical memory.

Note that several design memories can share the same physical bank.

1.4 Functional Usage Example

This section presents a typical usage of the memory import process to create a processor register file based on Xilinx registers instantiated in the design.

A processor register file is a memory which has a lot of access ports (for example 8 ports). Such a memory does not correspond to any Xilinx primitive. It can be implemented with registers. In this example, the register file is 32-bit wide and 128-bit deep. In the ZeBu database, it corresponds to 128 32-bit register vectors, from `reg_file[0]` to `reg_file[127]`.

During debug, the value of the register file is a useful data. To dump a file containing the value of the register file, or to see it in real time in **`zRun`**, it has to be imported as a memory.

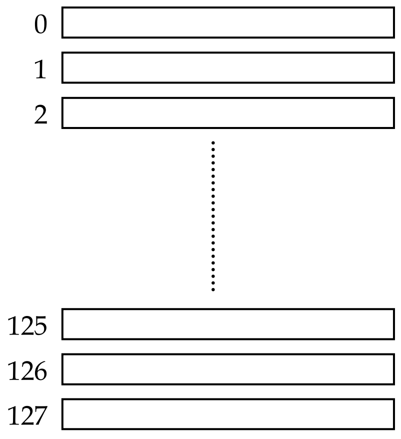

Figure 1: Register File before Import

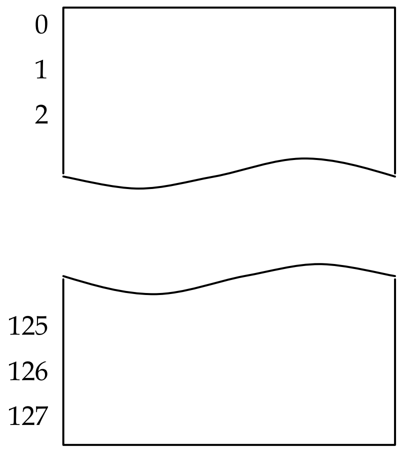

Figure 2: Register File after Import

After import, the register file is seen as a 32-bit wide, 128-bit deep memory and is accessible for file load/dump.

2 Memory Import Process

Memories are imported into the database with **`zDbPostProc`** using a memory import file, which describes the physical memory resources and the mapping which defines the resulting memory model.

2.1 Memory Import File

2.1.1 Separator Declaration

The memory import file uses hierarchical names. Those names contain a hierarchy separator character between instance names. The default separator character is '.' If it does not correspond to your design conventions, it can be modified in the memory import file:

```
Separator /;
```

2.1.2 Declaring the Physical Banks

2.1.2.1 Before Starting

It is recommended to check the existing memory resources in the database before writing the memory import file. For that purpose, you should go to the `zebu.work` directory and use the `show -memories` command of **`zDbPostProc`**:

```
$> cd zebu.work
$> echo "show -memories" | zDbPostProc -p .
```

The result shows the list of the memories present in the database, including any declared memory.

2.1.2.2 Definitions

A **bank** is a physical entity of the design, which has some properties:

- **Name:** user-defined name used to refer to that bank in the memory import file. It does not correspond to any physical name in the database.
- **Path:** full hierarchical name of the physical memory instance in the design. This information is mandatory in the memory import file.
- **Type:** physical implementation of the bank; can be `BRAM`, `LRAM` or `register`.
- **Model:** Xilinx primitive (for BRAM and LUTRAM banks) instantiated in the design. This property is not defined for a register-based bank.
- **Width:** physical width of the bank. For BRAM and LUTRAM banks, this property is optional since it is implicit in the model property. The width is mandatory for a register-based bank.
- **Depth:** physical depth of the bank. The model of the bank can implicitly declare it as in the above example. For example, a BRAM bank using the primitive is implicitly declared as a 512-word deep memory.

Type and model are used to check the coherency of the memory during the import.

2.1.2.3 Example

The instance `top.node2_0.node1_0` of a `RAMB16_S36` Xilinx primitive is a bank with the following properties:

Type	BRAM
Model	RAMB16_S36
Width (implicit)	36
Depth (implicit)	512
Path to the bank	`top.node2_0.node1_0.ramb16_s36_0`

If this memory instance is declared as `bank_0` in a memory import file, its description is:

```
bank bank_0 {               // bank_0 is the name of the bank
  type  bram;               // type of the bank
  model RAMB16_S36;         // => means 36x512
  path  top.node2_0.node1_0.ramb16_s36_0;
};
```

2.1.3 Declaring the Resulting Logical Memory

2.1.3.1 Definitions

An **imported memory** is the logical memory resulting from memory import operation. It has the following properties: a **name**, a **width**, a **depth**, a **path** and one or more **bank instances**:

- **Name:** name of the imported memory into the design, as it will be seen at runtime.
- **Depth:** defines the depth of the imported memory.
- **Width:** defines the width of the imported memory.
- **Path:** location of the imported memory in the design. It should correspond with its name. For example, if the name of the declared memory is `memory0`, the path of the memory has to end with `memory0`, as in the following example: `top.mod0.memory0`.
- **Bank instance:** section describing a part of a physical bank used to get the imported memory. A bank instance contains different parts:
    - **Name** of the bank: defines which previously declared bank is used
    - **Physical bit** and **physical address**
    - **Logical bit** and **logical address**
    - **Depth** and a **width**

  The **physical bit/address**, the **logical bit/address,** the **depth** and the **width** are used to specify the mapping of the bank in the resulting logical memory.

2.1.3.2 Determining the Bank Instance Parameters

The physical reference has its origin at address `0` and bit `0` of the physical bank. The logical reference has its origin at address `0` and bit 0 of the imported memory.

- The part of the bank used in the logical memory is called the **sub-bank**.
- The coordinates of the sub-bank in the physical reference give the `physical_bit` and the `physical_addr` parameters of the bank instance.
- The coordinates of the sub-bank in the logical reference give the `logical_bit` and the `logical_addr` parameters of the bank instance.
- The **depth** of the bank instance is the depth of the sub-bank.
- The **width** of the bank instance is the width of the sub-bank.

The parameters of a bank instance are detailed in the following figure:

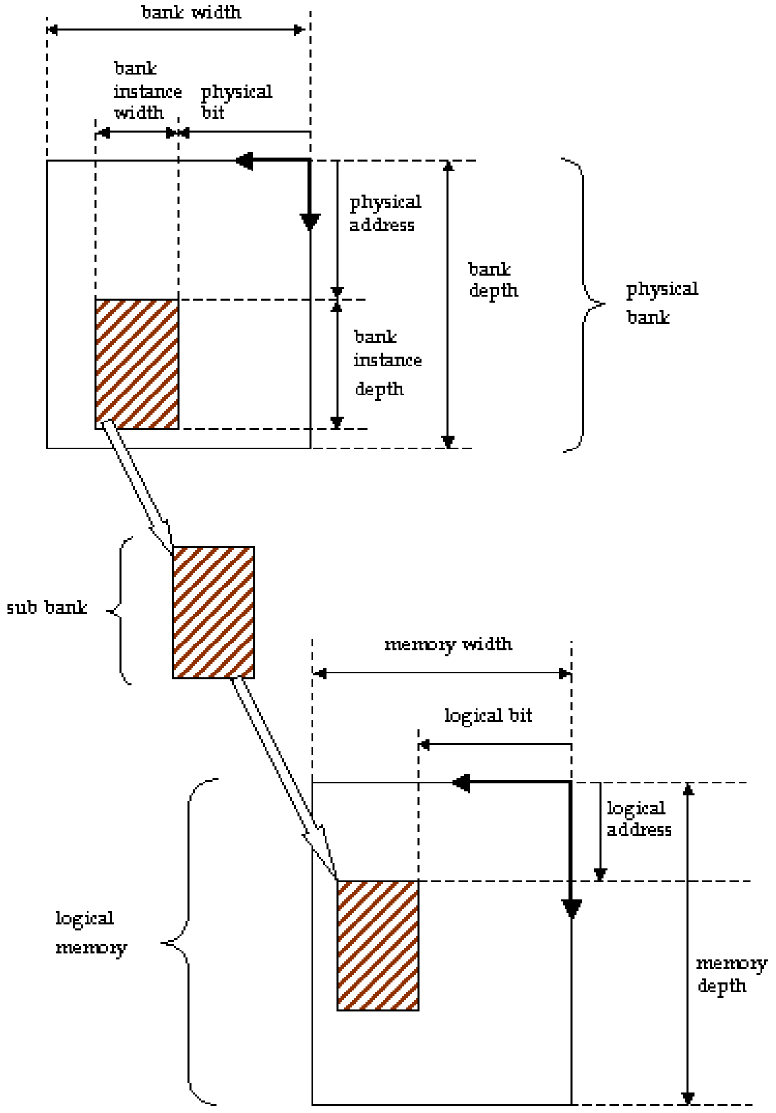

Figure 3: Bank Instance Parameters

The following examples illustrate bank mapping:

- In the first example, the memory is composed of 4 sub-banks and the sub-banks are equal to the banks.
- In the second example, the memory is composed of one sub-bank, and the memory is equal to the sub-bank.

2.1.3.3 Basic Example

In this example, an imported 32x512 memory uses 4 entire 16x256 BRAM banks. This is a special case where the sub-bank is equal to the bank. Figure 4 shows the mapping for the imported memory, with details for 1 of the 4 banks.

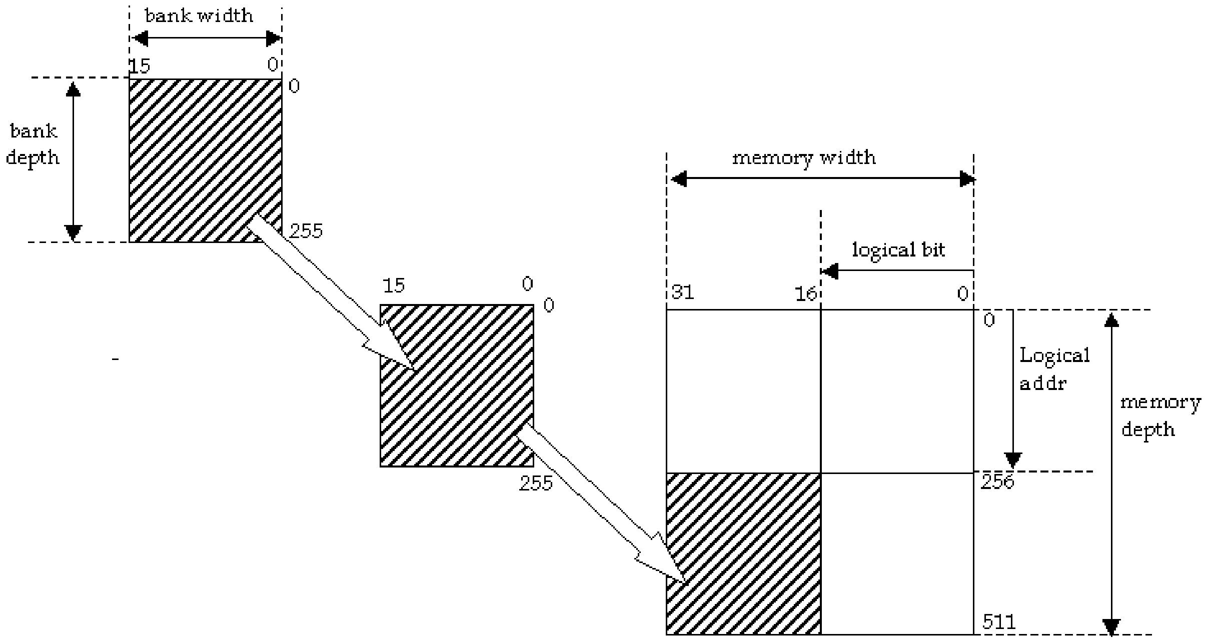

Figure 4: Bank Mapping Example (1)

The parameters of the striped bank instance called `Bank_2` are shown below.

```
bank Bank_2
  logical_bit   = 16,  // memory relative bit
  logical_add   = 256, // memory relative address
  physical_bit  = 0,   // bank relative bit
  physical_add  = 0,   // bank relative addr
  width = 16,
  depth = 256;
```

The parameters of the entire memory instance are shown below.

```
memory mem {
  width 32;
  depth 512;
  path top.mem;

 /* …declaration of Bank_0, Bank_1, Bank_2 and Bank_3…*/

  bank Bank_0
    logical_bit   = 0,  // memory relative bit
    logical_add   = 0, // memory relative address
    physical_bit  = 0,   // bank relative bit
    physical_add  = 0,   // bank relative addr
    width = 16,
    depth = 256;

  bank Bank_1
    logical_bit   = 16,  // memory relative bit
    logical_add   = 0, // memory relative address
    physical_bit  = 0,   // bank relative bit
    physical_add  = 0,   // bank relative addr
    width = 16,
    depth = 256;

  bank Bank_2
    logical_bit   = 16,  // memory relative bit
    logical_add   = 256, // memory relative address
    physical_bit  = 0,   // bank relative bit
    physical_add  = 0,   // bank relative addr
    width = 16,
    depth = 256;

  bank Bank_3
    logical_bit   = 0,  // memory relative bit
    logical_add   = 256, // memory relative address
    physical_bit  = 0,   // bank relative bit
    physical_add  = 0,   // bank relative addr
    width = 16,
    depth = 256;
}
```

2.1.3.4 Advanced Example

In this case, an imported 1x512 memory uses a 36x512 BRAM bank.

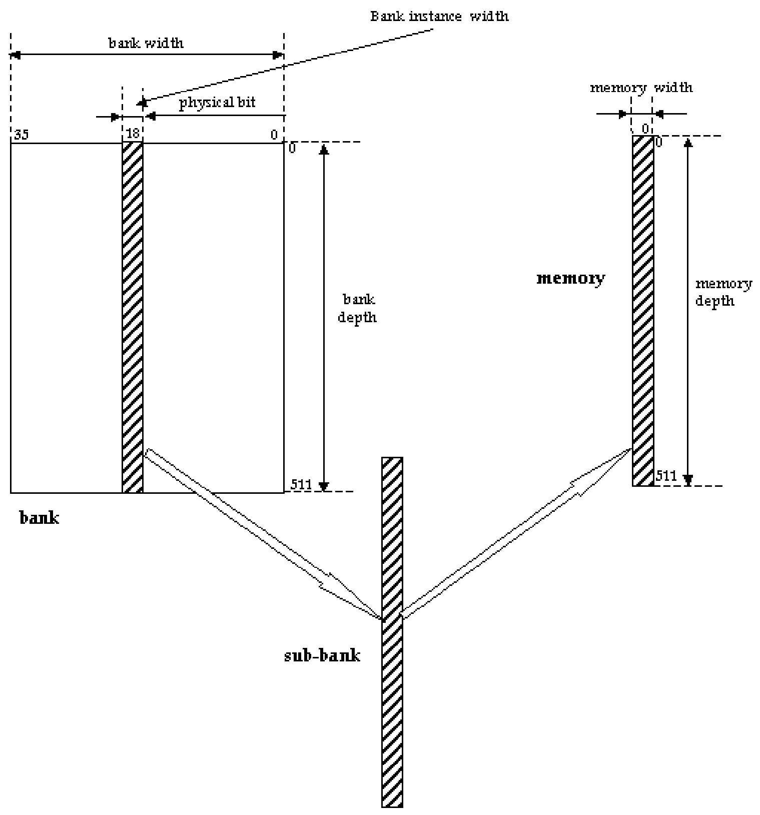

Figure 5: Bank Mapping Example (2)

The parameters of the entire bank instance are shown below.

```
memory mem {
  width 1
  depth 512;
  path top.mem;

  bank Bank0
    logical_bit   = 0,  // memory relative bit
    logical_add   = 0,  // memory relative address
    physical_bit  = 18, // bank relative bit
    physical_add  = 0,  // bank relative addr
    width = 1,
    depth = 256;
}
```

2.1.4 General Syntax for the Memory Import File

Following is the description of the general syntax for the memory import file which is a proprietary ZeBu file format:

```
Memory_description_file := separator_desc bank_list memory_list;

Separator_desc :=
     | SEPARATOR CHAR
bank_list := bank_desc bank_list
     | bank_desc ;
bank_desc := BANK identifier '{'
     TYPE TYPE_ID
     MODEL MODEL_ID
     PATH PATH_ID
     '}' ;
memory_list := memory_desc memory_list
     |memory_desc ;
memory_desc := MEMORY identifier '{'
     DEPTH integer
     WIDTH integer
     PATH PATH_ID
     bank_ref_list
     '}' ;

bank_ref_list := BANK identifier ':' mxcoord ',' mycoord ','
          bxcoord ',' bycoord ','
          bwcoord ',' bdcoord ;

mxcoord := 'logical_bit =' integer;
mycoord := 'logical_add =' integer;
bxcoord := 'physical_bit =' integer;
bycoord := 'physical_add =' integer;
bwcoord := 'width =' integer;
bdcoord := 'depth =' integer;

BANK := 'bank' ;
TYPE := 'type' ;
TYPE_ID := 'bram' | 'lram' | 'reg' ;
PATH := 'path'
DEPTH := 'depth' ;
WIDTH := 'width' ;
MEMORY := 'memory'
```

Note: Comments are C or C++ like comments (// and /*…*/).

2.2 zDbPostProc Usage for Memory Import

The **zDbPostProc** command for memory import is:

```
import_mem <filename>
```

Where `<filename>` is the name of the memory import file that has to be used.

Usually, a Tcl script is used for memory import operation with **zDbPostProc**. The memories are imported into the database from the memory import file by **zDbPostProc**.

If you intend to use several memory import files, you can repeat the `import_mem` command for each memory import file in the **zDbPostProc** script.

It is recommended to check the resulting memories in the database after the memory import operation. For that purpose, you can add the `show -memories` command after the `import_mem` command in the **zDbPostProc** script.

Do not forget to use the `save_db` command to save the database after the import, and to generate the co-simulation wrapper you need.

Example of zDbPostProc Tcl script for Memory Import:

```
import_mem src/import.txt
show -memories
save_db
wrapper -cpp
exit
```

2.3 Accessing the Memory from a C++ Testbench

In the driver files generated by **zDbPostProc** for C++ co-simulation `TOP_top.hh` and `TOP_top.cc`, the imported memory is present like any other memory, in addition to the original physical memory instances used as banks.

Of course, the content of the physical banks is also changed when the imported memory is written from the testbench, and *vice versa*.

Example:

To set/get the content of the memory imported (`top.memory.mem`) with a buffer, the testbench is the following:

```
// init of the board and of the driver
[...]

// allocation of the memory buffer
uint *buffer = TOP_top.memory.mem->getbuffer();
memset(buffer, 0, 512*sizeof(uint));

// filling the memory with the pattern 0
TOP_top.memory.mem->set(0);
```

```
Driver->run(2000);
// update of the cache
TOP_top.memory.mem->storeTo(NULL);
// get of the values
TOP_top.memory.mem->storeTo(buffer);

[...]
```

2.4 Accessing the Memory from zRun

In the **zRun** graphical interface, you can use the memory browser to access the imported memory and the physical memory banks. Of course, the content of the physical banks is also changed when the imported memory is written from the testbench, and *vice versa*.

To control the emulation from **zRun** with a C++ testbench, you should use the following command:

```
$> zRun –testBench "<my_testbench>"
```

Where `<my_testbench>` is the name of the executable file for your C++ testbench.

3 Detailed Examples

3.1 Importing a Single Bank Memory

3.1.1 Goal of the Example

This example accesses with a C++ testbench the 16 higher bits of a `ramb16_s36`. It demonstrates how to declare the memory import file, how to import the memory into the database, and how to access it from in the C++ testbench.

- The imported memory is expected in a module called `memory`, which is at the top level of the design, with the following path: `top.memory.mem`.
- The path to the bank instance is: `top.mod0.mod1.mod2.mod3.bram_0` (with '.' as hierarchy separator).

The following figure illustrates the organization of the memory:

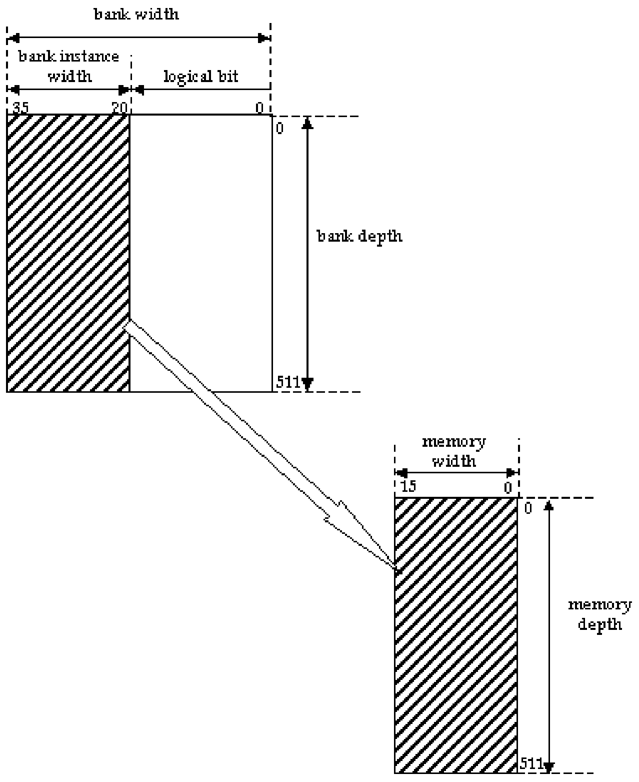

Figure 6: Example for Single Bank Memory Import

3.1.2 Available Memory Resources in the Original Database

It is recommended to check the existing memory resources in the database before writing the memory import file.

For that purpose, you should go to the `zebu.work` directory and use the `show –memories` command of **`zDbPostProc`**:

```
$> cd zebu.work
$> echo "show –memories" | zDbPostProc –p .
```

The result shows the list of the memories present in the database, including any declared memory:

```
         ##############################################
         #           Copyright (c) 2002-2009          #
         # Emulation and Verification Engineering  SA #
         #--------------------------------------------#
         # zDbPostProc                                #
         # revision :                                 #
         # date :                                     #
         #--------------------------------------------#
         ##############################################

# zDbPostProc -p .

Checking for license ... OK
# Loading ./ZebuDB.zdb
# Loaded ...

top.mod0.mod1.mod2.mod3.bram_0
```

3.1.3 Describing the Memory Import File

As shown on Figure 6, the different parameters of the bank instance to set in the memory import file are:

- Memory depth: 512
- Bank instance depth: 512
- Bank instance width: 16

In the description of the imported memory, this bank will be used with the following parameters:

- Logical bit: 0
- Logical address: 0
- Physical bit: 20
- Physical address: 0
- Depth: 512
- Width: 16

The full memory import file (`my_memory.txt`) is:

```
bank b0 {
  type  bram;             // type of the bank
  model RAMB16_S36;       // 36x512
  path  top.mod0.mod1.mod2.mod3.bram_0;
};

memory mem {
  width 16
  depth 512;
  path top.memory.mem;    // full path to the importedd memory

  bank b0
    logical_bit   = 0,  // memory relative bit
    logical_add   = 0,  // memory relative address
    physical_bit  = 20, // bank relative bit
    physical_add  = 0,  // bank relative addr
    width = 16,
    depth = 512;
}
```

3.1.4 Importing the Memory File

To import the memory file, it is necessary to create a Tcl script for **`zDbPostProc`** (`script_import.tcl` in the present example):

```
puts "******** LIST OF THE MEMORIES BEFORE IMPORT ********"
show –memories
puts "******** LOAD THE MEMORY IMPORT FILE ********"
import_mem my_memory.txt
puts "******** LIST OF THE MEMORIES AFTER IMPORT *********"
show –memories
puts "******** SAVING DB **********"
save_db
puts "********GENERATING THE FILES FOR C++ CO-SIMULATION********"
wrapper -cpp
exit
```

The command to import the memories is:

```
$> zDbPostProc script_import.tcl –p ../zebu.work
```

The result of the command gives the following result:

```
  ##############################################
  #           Copyright (c) 2002-2009          #
  # Emulation and Verification Engineering  SA #
  #--------------------------------------------#
  # zDbPostProc                                #
  # revision :                                 #
  # date :                                     #
  #--------------------------------------------#
  ##############################################

# zDbPostProc script_import.tcl –p  ../zebu.work

Checking for license ... OK
# Loading zebu.work/ZebuDB.zdb
# Loaded ...
******** LIST OF THE MEMORIES BEFORE IMPORT ********
top.mod0.mod1.mod2.mod3.bram_0
******** LOAD THE MEMORY IMPORT FILE ********
importing memory 'top.memory.mem'
******** LIST OF THE MEMORIES AFTER IMPORT ********
top.mod0.mod1.mod2.mod3.bram_0
top.memory.mem
******** SAVING DB **********
# Saving ../zebu.work/ZebuDB.zdb
# Saved ...
# Dynamic probe frequency : 40000000 Hz (with dynamic probe clock at 40 MHz)
# Saving ZebuPrb.tcl
# Saved ...
********GENERATING THE FILES FOR C++ CO-SIMULATION********
# step C++ GENERATION : generating 'TOP_top.cc' & 'TOP_top.hh' for internal state capture
# step C++ GENERATION : generating ../zebu.work/top_ccosim.cc', ../zebu.work/top_ccosim.hh'
```

3.1.5 Accessing the Memory in a C++ Testbench

In the generated files, `TOP_top.hh` and `TOP_top.cc`, the imported memory is present like any other memory with the path `top.memory.mem`.

Note that the memory `top.mod0.mod1.mod2.mod3.bram_0` is also accessible from the testbench. If this memory is written from the testbench, the content of the imported memory `top.memory.mem` will be changed.

The code in the testbench to access the imported memory is the following:

```
/* init of the board and of the driver */
[...]

// allocation of the memory buffer
uint *buffer = TOP_top.memory.mem->getbuffer();
memset(buffer, 0, 512*sizeof(uint));

// filling the memory with the pattern 0
TOP_top.memory.mem->set(0);

Driver->run(2000);
// update of the cache
TOP_top.memory.mem->storeTo(NULL);
// get of the values
TOP_top.memory.mem->storeTo(buffer);
[...]
```

If you use **`zRun`** to control your C++ co-simulation (as described in Section 2.4), the memory browser contains the imported memory `top.memory.mem` and the original memory `top.mod0.mod1.mod2.mod3.bram_0`.

3.2 Importing a 3-bank Memory

3.2.1 Goal of the Example

This example shows how to design a memory from 3 sub-parts of 3 `ramb16_s36`.

The imported memory has a size of 54x256, with the following path:

```
top.memory.mem
```

The instances of the 3 banks have the following paths:

```
top.node2_0.node1_0.ramb16_s36_0
top.node2_0.node1_0.ramb16_s36_1
top.node2_0.node1_0.ramb16_s36_2
```

The following figure illustrates the organization of the memory:

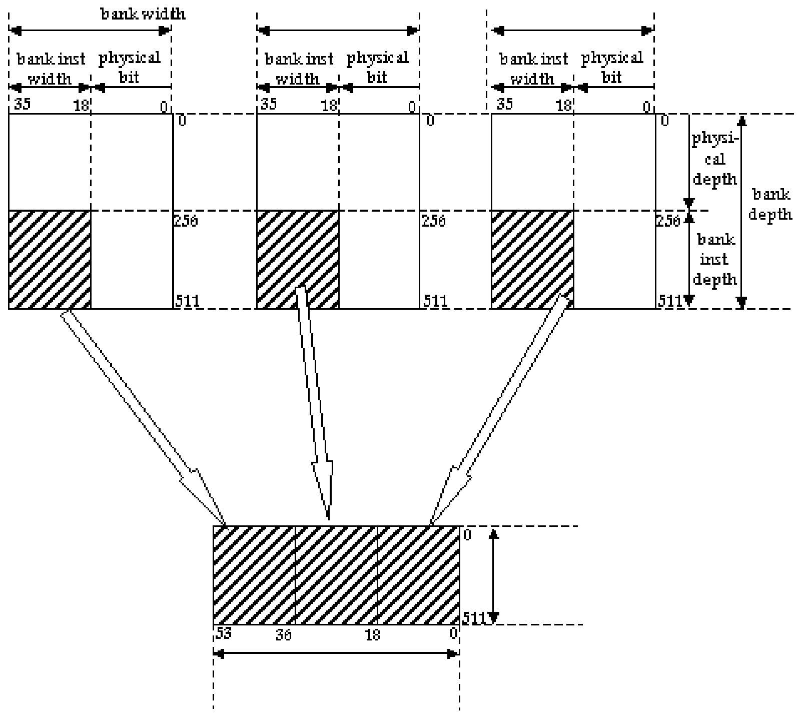

Figure 7: Example for 3-bank Memory Import

3.2.2 Available Memory Resources in the Original Database

It is recommended to check the existing memory resources in the database before writing the memory import file.

For that purpose, you should go to the `zebu.work` directory and use the `show –memories` command of **`zDbPostProc`**:

```
$> cd zebu.work
$> echo "show –memories" | zDbPostProc –p .
```

The result shows the list of the memories present in the database, including any declared memory:

```
  ##############################################
  #          Copyright (c) 2002-2009           #
  # Emulation and Verification Engineering  SA #
  #--------------------------------------------#
  # zDbPostProc                                #
  # revision : 433                             #
  # date : Wed 11 3 2009 - 18:15:37            #
  #--------------------------------------------#
  ##############################################

# zDbPostProc -p .
# start time is Fri Mar 13 11:39:05 2009


Checking for license ... OK
#   step open DB : Loading ./ZebuDB.zdb
#   step open DB : Loaded ...
zDbPostProc [~/bench/V43X/mem3d.path_pb.solved/3d/zcui.work/zebu.work]
top.mem3d
```

3.2.3 Describing the Memory Import File

According to Figure 7, the content of the memory import file (`my_memory.txt`) is the following:

```
bank b0 {
  type  bram;
  model RAMB16_S36;
  path  top.node2_0.node1_0.ramb16_s36_0;
};
bank b1 {
  type  bram;
  model RAMB16_S36;
  path  top.node2_0.node1_0.ramb16_s36_1;
};
bank b2 {
  type  bram;
  model RAMB16_S36;
  path  top.node2_0.node1_1.ramb16_s36_0;
};

memory mem {
  width 54;
  depth 256;
  path top.memory.mem;

  bank b0
    logical_bit   = 36,
    logical_add   = 0,
    physical_bit  = 18,
    physical_add  = 256,
    width = 18,
    depth = 256;
  bank b1
    logical_bit   = 18,
    logical_add   = 0,
    physical_bit  = 18,
    physical_add  = 256,
    width = 18,
    depth = 256;

  bank b2
    logical_bit   = 0,
    logical_add   = 0,
    physical_bit  = 18,
    physical_add  = 256,
    width = 18,
    depth = 256;
};
```

3.2.4 Importing the Memory File

To import the memory file, it is necessary to create a Tcl script for **zDbPostProc** (`script_import.tcl` in the present example):

```
puts "******** LIST OF THE MEMORIES BEFORE IMPORT ********"
show –memories
puts "******** LOAD THE MEMORY IMPORT FILE ********"
import_mem my_memory.txt
puts "******** LIST OF THE MEMORIES AFTER IMPORT *********"
show –memories
puts "******** SAVING DB **********"
save_db
puts "********GENERATING THE FILES FOR C++ CO-SIMULATION********"
wrapper -cpp
exit
```

The command to import the memories is:

```
$> zDbPostProc script_import.tcl –p ../zebu.work
```

4 EVE Contacts

For product support, contact: support@eve-team.com.

For general information, visit our company web-site: http://www.eve-team.com

Europe Headquarters	EVE SA 2-bis, Voie La Cardon Parc Gutenberg, Bâtiment B 91120 Palaiseau FRANCE Tel: +33-1-64 53 27 30
US Headquarters	EVE USA, Inc. 2290 N. First Street, Suite 304 San Jose, CA 95054 USA Tel: 1-888-7EveUSA (+1-888-738-3872)
Japan Headquarters	Nihon EVE KK KAKiYA Building 4F 2-7-17, Shin-Yokohama Kohoku-ku, Yokohama-shi, Kanagawa 222-0033 JAPAN Tel: +81-45-470-7811
Korea Headquarters	EVE Korea, Inc. 804 Kofomo Tower, 16-3, Sunae-Dong, Bundang-Gu, Sungnam City, Kyunggi-Do, 463-825, KOREA Tel: +82-31-719-8115
India Headquarters	EVE Design Automation Pvt. Ltd. #B-15, Raheja Arcade, 80 Ft. Road, 5th Block, Koramangala Bangalore - 560 095 Karnataka INDIA Tel: +91-80-41460680/30202343
Taiwan Headquarters	14F1, No 371, Sec. 1 Guangfu Rd., East District Hsinchu City 300 TAIWAN (R.O.C.) Tel: +886-(3)-564-7900