Design and Implementation of an Memory Management Unit (MMU)

School of Engineering - Department of Informatics Engineering

T.E.I. Crete

by
George Timbakianakis R.N.:2805

1 10 2015

Abstract

As the modern world processes larger amounts of data and demand increases for massive amounts of memory, the need of efficient memory management also becomes critical. The aim of this study is to develop and implement a memory management unit (MMU) utilising the AXI interface protocol in order to create an ip module capable of obeying the basic rules of memory management through address translation. The MMU module will be implemented on a ZedBoard™ development platform using the Xilinx Zynq®-7000 All Programmable SoC. At the end of this study we ended up with an versatile MMU module that conforms with the AXI4 interface protocol that can translate given addresses from the Zynq®-7000 core and can perform memory writes to the translated addresses. The IP is fully designed, from the top up, with versatility in mind, enabling it to be used in a variety of situations as well as an educational tool for future student engineers.

Thesis Supervisor: Kornaros George
Title: Professor, Dept. of Informatics Engineering

Acknowledgements

There are many people who deserve recognition and thanks for contributing to the completion of this thesis project.
My parents, Demetres and Athena Timbakianakis, who have always supported my aspirations and endeavours in life.
My fellow student, Spyros Chiotakis for providing much-needed advice during my research.
Professor George Kornaros, for giving me the opportunity to participate in the ISCA lab facilitating the entire process from beginning to end. Finally, a huge thank you to Professor Fragopoulou Paraskevi that gave me the means and inspiration I needed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Memmory Managment Units

A memory management unit (MMU), sometimes called paged memory management unit (PMMU), is a computer hardware unit having all memory references passed through itself, primarily performing the translation of virtual memory addresses to physical addresses. It is usually implemented as part of the central processing unit (CPU), but it also can be in the form of a separate integrated circuit.
An MMU effectively performs virtual memory management, handling at the same time memory protection, cache control, bus arbitration and, in simpler computer architectures (especially 8-bit systems), bank switching.
Modern MMUs typically divide the virtual address space (the range of addresses used by the processor) into pages, each having a size which is a power of 2, usually a few kilobytes, but they may be much larger. The bottom bits of the address (the offset within a page) are left unchanged. The upper address bits are the virtual page numbers.

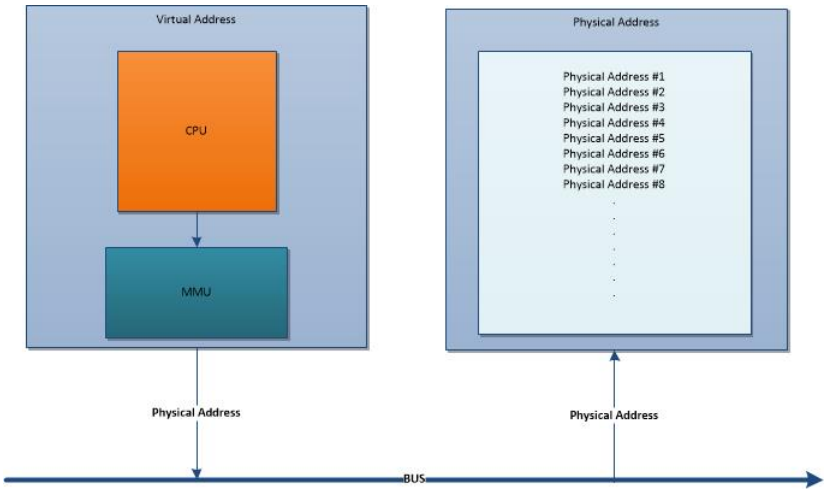

Figure 1.1: Compute System with MMU

1.2 Overview of FPGAs

Field Programmable Gate Arrays (FPGAs) are used for rapid prototyping of digital circuits. The design and test of digital systems are very time-efficient and cost-efficient with FPGAs. They are composed of programmable digital blocks and programmable interconnect. Figure 1.1 shows a typical architecture of an FPGA. Programmable Logic Blocks (PLBs) represent the programmable digital block. A cluster of PLBs is called a Configurable Logic Block (CLB). Switch Blocks (SB) represent the programmable interconnect. PLBs may be connected directly to a switch block (like the one in Figure 1-1) or to interconnect. PLBs can implement small digital circuits and programmable interconnect can connect the small digital blocks to constitute a complex digital system. If there are enough PLBs and interconnect resources, most digital circuits can be implemented on FPGAs.

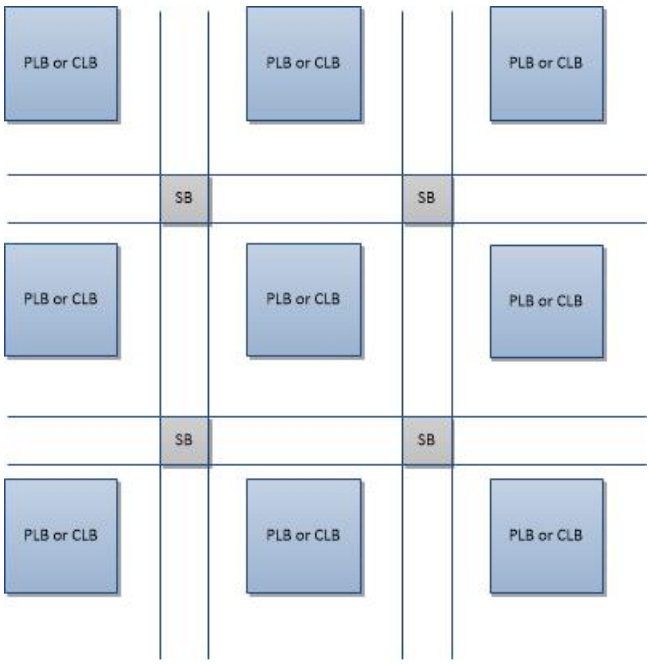

Figure 1.2: Generic FPGA Architecture

Since the switches in switch blocks consume more area and have a larger capacitance than wires in Application Specific Integrated Circuits (ASIC), FPGAs are typically 10 times larger and 3 times slower than ASICs. The higher capacitance also results higher power consumption even at equal speed. To make FPGAs suitable for more applications, their speed and power consumption should be improved.

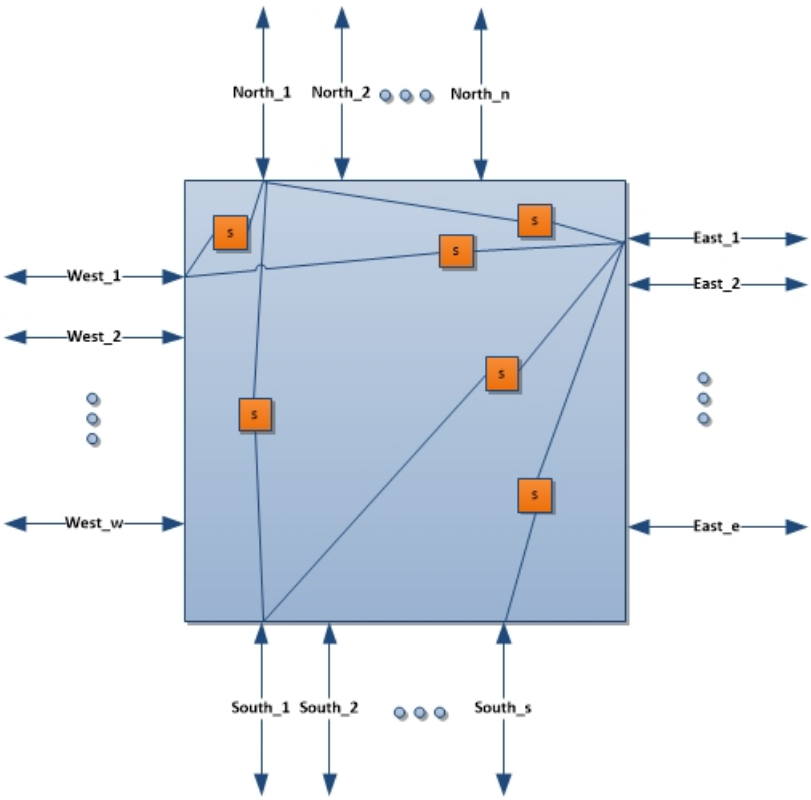

Figure 1.3: Switch block pins with different switch flexibility.

In switch blocks, the number of possible connections between one pin to other pins is called the flexibility of the pin. For example, in Figure 1-2, the switch flexibility of South_ W is 1, the flexibility of West_ 1 and South_ 1 is 2, the flexibility of North_ 1 is 3, and the flexibility of East_ 1 is 4. The pins that are not connected to anywhere have a flexibility of 0. Switch flexibility of 3 or 4 is suggested to be very optimum if all pins have the same flexibility.

1.3 Purpose

The purpose of this Thesis is to implement a MMU for use in Zynq-7000 SoC designs. The main reason for implementing this type of MMU is to be able to perform memory address translations without intervention from the processor.
There are some specific features that should be supported by the MMU. It must be connected to a AXI4 Master port of the Zynq PS and it must be able to write to multiple AXI4 Slave ports. It must also be able to be configured through its Slave port by the Zynq PS.
When implemented its functions must be tested and verified.

Chapter 2

Tools

2.1 Hardware

ZedBoard™

ZedBoard™ is a complete development kit for designers interested in exploring designs using the Xilinx Zynq®-7000 All Programmable SoC. The board contains all the necessary interfaces and supporting functions to enable a wide range of applications. The expandability features of the board make it ideal for rapid prototyping and proof-of-concept development.

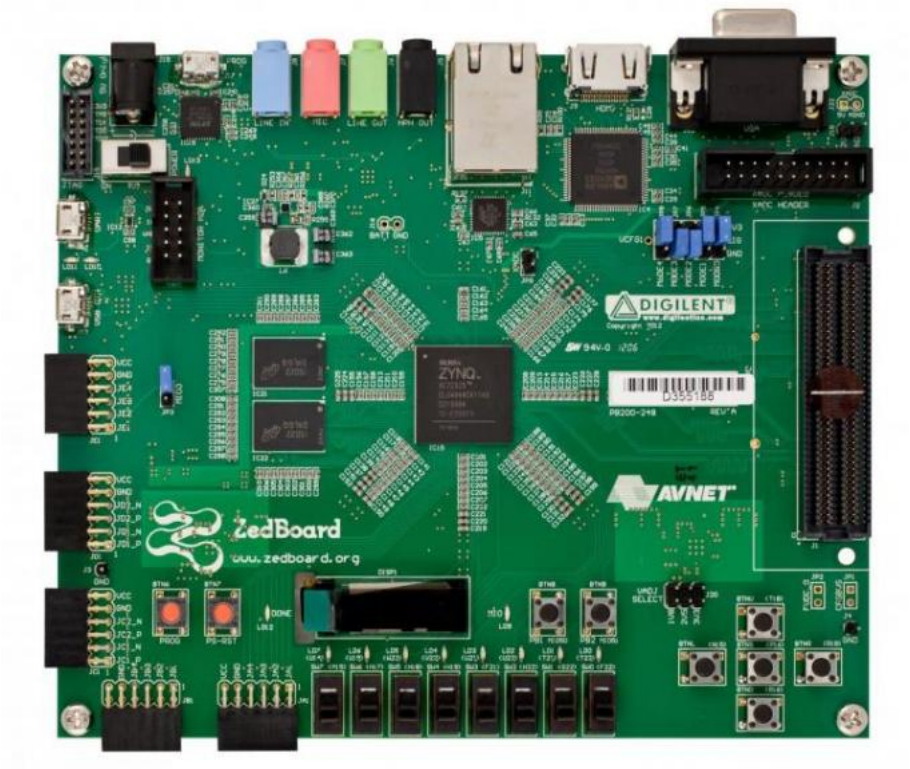

Figure 2.1: ZedBoard™

2.2 Software

Vivado®

The Vivado® Design Suite delivers a SoC-strength, IP-centric and system-centric, next generation development environment that has been built from the ground up to address the productivity bottlenecks in system-level integration and implementation. The Vivado Design suite is a Generation Ahead in overall productivity, ease-of-use, and system level integration capabilities.

Figure 2.2: Vivado Logo

SDK

The Software Development Kit (SDK) is the Xilinx Integrated Design Environment for creating embedded applications on any of Xilinx' award winning microprocessors for Zynq®-7000 All Programmable SoCs, and the industry-leading MicroBlaze™. The SDK is the first application IDE to deliver true homogenous and heterogeneous multi-processor design and debug.

AMBA AXI Interface

AXI is part of ARM AMBA, a family of micro controller buses first introduced in 1996. The first version of AXI was first included in AMBA 3.0, released in 2003. AMBA 4.0, released in 2010, includes the second major version of AXI, AXI4. There are three types of AXI4 interfaces:

- AXI4: For high-performance memory-mapped requirements.
- AXI4-Lite: For simple, low-throughput memory-mapped communication.
- AXI4-Stream: For high-speed streaming data.

In this MMU implementation, AXI4-Lite will be the interface of choice due to the fact that high throughput is not needed as well as the burst function is unnecessary.

2.3 Used Languages

In this Thesis programming languages were used to describe the hardware components as well as providing testing algorithms.

2.3.1 VHDL

VHDL (VHSIC Hardware Description Language) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits. VHDL can also be used as a general purpose parallel programming language.
in this project VHDL is used to describe the hardware components of the design tat will, later, be synthesized and implemented.

2.3.2 TCL

Tcl (originally from Tool Command Language, but conventionally spelled "Tcl" rather than "TCL"; pronounced as "tickle" or "tee-see-ell") is a scripting language created by John Ousterhout. Originally "born out of frustration", according to the author, with programmers devising their own languages intended to be embedded into applications, Tcl gained acceptance on its own.
It is commonly used for rapid prototyping, scripted applications, GUIs and testing. Tcl is used on embedded systems platforms, both in its full form and in several other small-footprint versions.
In this project TCL is used for scripting purposes.

Chapter 3

Implementation

This chapter describes how the MMU IP was implemented. It contains information about the structure of the IP and the implemented code. It also describes how the AXI4 Lite interfaces are implemented and how the finished module was incorporated in a final design. Explanations of the two FSM:s that provides the functionality of the MMU IP are included. A description of a test transaction is provided.

3.1 RAM

At the heart of the MMU is a rather simple design. As a lookup table, there is a 16 bit Random Access Memory design and all the routing and manipulating of the address is done with simple logic.

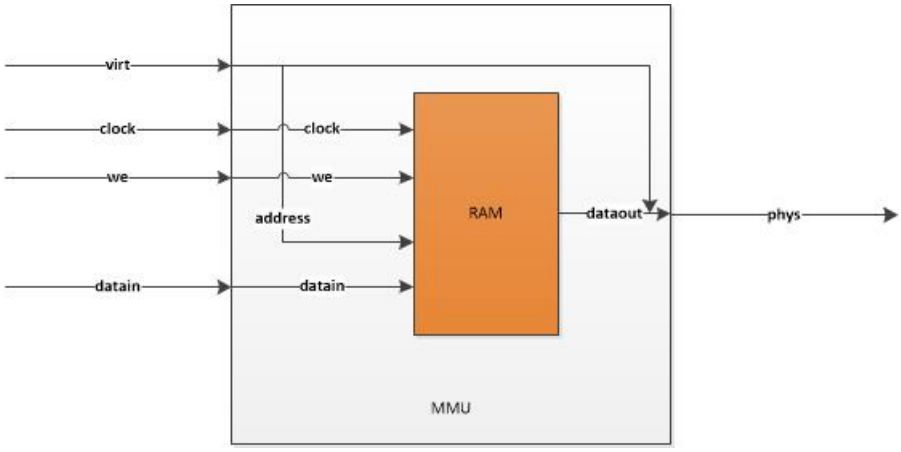

Figure 3.1: Abstract MMU Design

The RAM in this design is 16 bit(65536 cell) and is used to store the physical part of the translated addresses. To access the cell containing the physical part, the virtual part must be entered as the ram address.

Listing 3.1: RAM VHDL Code

```
5  entity ram is
6   port (
7      clock   : in  std_logic;
8      we      : in  std_logic;
9      address : in std_logic_vector(15 downto 0);
10     datain  : in  std_logic_vector(15 downto 0);
11     dataout : out std_logic_vector(15 downto 0)
12    );
13 end entity ram;
14
15 architecture RTL of ram is
16
17   type ram_type is array (0 to (2**address'length)-1) of
    std_logic_vector(datain'range);
18   signal ram : ram_type;
19   signal read_address : std_logic_vector(address'range);
20
21 begin
22  RamProc: process(clock) is
23
24   begin
25    if rising_edge(clock) then
26      if we = '1' then
27        ram(to_integer(unsigned(address))) <= datain;
28      end if;
29      read_address <= address;
30    end if;
31  end process RamProc;
32
33  dataout <= ram(to_integer(unsigned(read_address)));
34
35 end architecture RTL;
```

In order to save data at a specified address you have to enable the we signal and then enter the address you want to store and the data for storing.

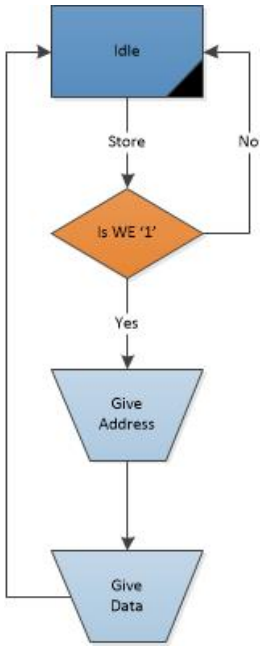

Figure 3.2: Ram Store Procedure

In order to read from a address you only have to specify the address signal and the dataout signal will become equal to the stored data.

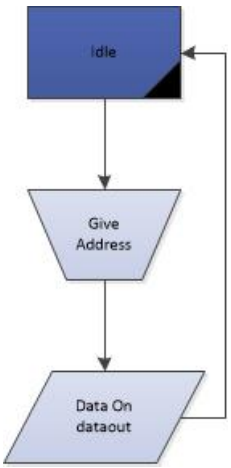

Figure 3.3: Ram Read Procedure

The code for the RAM is written in a versatile way that makes changes easy. For example we can change the width by changing the address length on the line 9.

3.2 MMU Top Level

In order to manage the routing and handling of the address a top level is needed. In the top level, user logic must be added to "break" the virt(31 downto 0) signal

into two (15 downto 0) signals. The first 16 bits are ported to the phys(15 downto 0) signal, at line 36 and the last 16 bits are ported to the address(15 downto 0) port of the ram component.

Listing 3.2: RAMs Port Map in Top Level

```
29 begin
30 ram1: ram port map (
31       clock  => clock,
32       we     => we,
33       address => virt(31 downto 16),
34       datain => datain,
35       dataout => phys(31 downto 16)
36       );
37       phys(15 downto 0) <= virt(15 downto 0);
38
```

This enables to pair the translated part, sourced from the RAM, with the non-translated part of the Virtual address.

3.3 AXI Interface

In order to include the MMU in a working design we must "encapsulate" is with an AXI4 interface in order to be able to connect it with other IP blocks that implement the same protocol.
The MMU IP will have 2 AXI4 interfaces. One AXI4 Lite Slave port will be utilised as the IPs input from the PS and one AXI4 Lite Master port used from the IP to write on a target(BRAM, DDR e.t.c.)

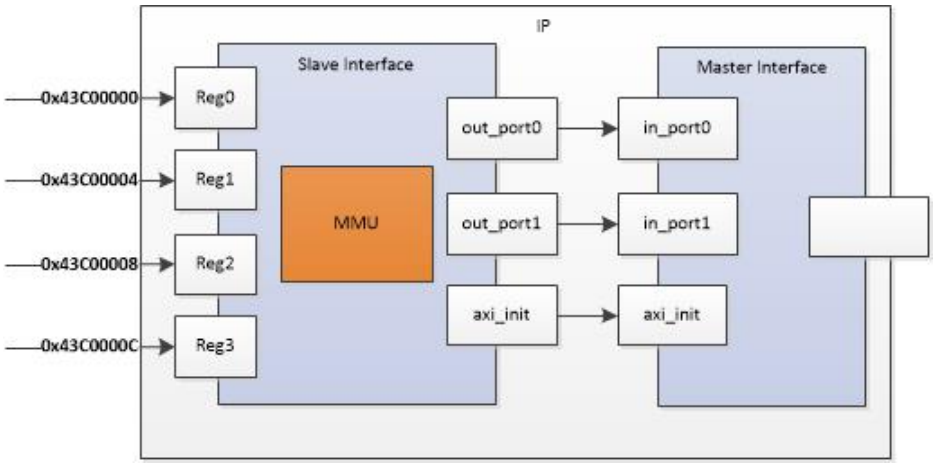

Figure 3.4: AXI4 Interface Design

3.3.1 Slave AXI Interface

The Slave interface will have 4 Input Registers where input data will be entered. Each register will serve a specific purpose.

Register	Address	Use	Size
0	0x43C00000	Targeted/Translated Address	16 bit/32 bit
1	0x43C00004	Write Enable/Association Address	1 bit/16 bit
2	0x43C00008	AXI4 Lite Transaction Initialisation	1 bit
3	0x43C0000C	Data	32 bit

Table 3.1: MMU Slave Registers

·**Register 0**
This is a 32 bit register that will have dual purpose. In translation mode it will act as the Virtual Address input. In store mode its first 16 bits will be used as the RAM target address.

·**Register 1**
This 32 bit register utilises only 17 bits. Bit 16 is used to control the mode of the MMU. If it is 'true' we are in Store mode. If it is 'false' we are in Translation Mode. The last 16 bits are used only when we are in Store Mode, ie when the 16th bit of Register 1 is '1'. The data that will be stored in the RAM address specified by Register 0, will be input through this Register.

·**Register 2**
Register 2 is a control register that initialises the AXI transaction when enabled. When we enable it (i.e. 0x43C00000 = '1') the MMU writes the data we input on the specified target using the translated address.

·**Register 3**
This is the data input Register. It is a 32 bit Register used as an input for the data we want to store.

Below you can see the port mapping of the MMU module, inside the Slave port and how we assign each Register to the MMUs input.

Listing 3.3: Port Map of MMU in Slave Port

```
263   mmu_0 : mmu
264   port map(
265        virt  => slv_reg0,
266        phys  => s_out_port0,
267        clock => S_AXI_ACLK,
268        we    => slv_reg1(16),
269        datain=> slv_reg1(15 downto 0)
270         );
271
272    s_txn_init <=slv_reg2(0);
273    s_out_port1 <=slv_reg3;
```

The Slave interface will also have 2 OUT ports(s_out_port0 and s_out_port1) used for communication between the Slave and Master port, within the IP. These ports will be used to transfer the translated address(line 266) and data(line 273) to the Master port in order for the Master port to perform the AXI write. To initialize the AXI Write on the Master there is a s_txn_init out port that is controlled, as marked before, from Register 2(line 272).

3.3.2 Master AXI Interface

The master interface will be implemented to enable the IP to complete AXI Writes to other AXI Slave ports in order to store data to specified targets.
The address, which at this point is already translated by the MMU in the Slave port, and data are input from the m_in_port0 and m_in_port1(line 14 and 15) to the Master interface and ported through 2 signals(line 440 and 441) to the M_AXI_AWADDR and M_AXI_WDATA OUT ports(line 114 and 115).
This essentially tells the Master port where to store the data that it got.

Listing 3.4: Master Port Routing Routine

```
14  m_in_port0 : in std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
15  m_in_port1 : in std_logic_vector(C_M_AXI_ADDR_WIDTH-1 downto 0);
.
.
114 M_AXI_AWADDR <= mmu_transfer0;
115 M_AXI_WDATA  <= mmu_transfer1;
.
.
440 mmu_transfer0 <= m_in_port0;
441 mmu_transfer1 <= m_in_port1;
```

3.3.3 Complete IP

After we finish writing and editing the code for the AXI4 Interfaces, with the use of Vivados Create and Package IP tool, we package everything into one IP that can be used in the Block Design environment as a IP Block.

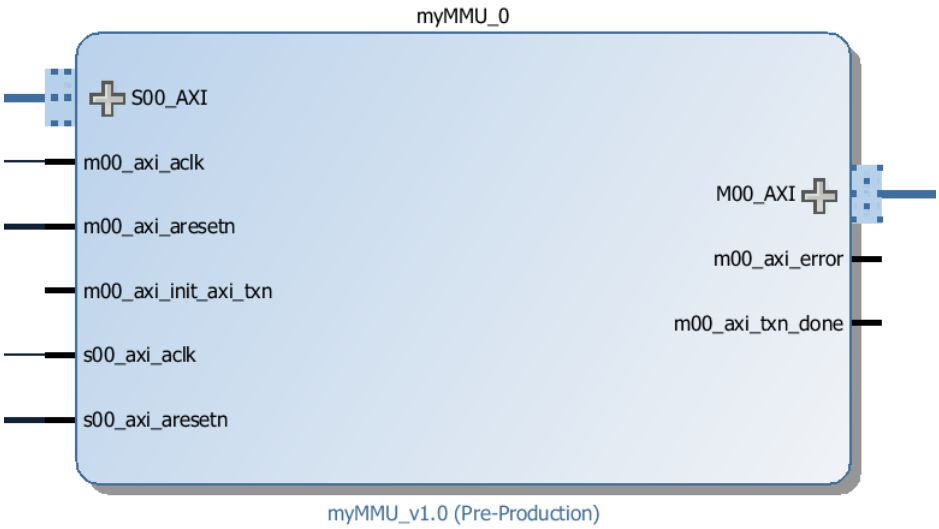

Figure 3.5: Packaged MMU IP Block

The tool automatically produces an IP with the appropriate ports that can be later added to a block diagram in Vivados IP Integrator.
The IP, besides the AXI Ports, has various other ports. The most important ports are the Clock ports(axi_aclk), that provide clock timing to the ports and the reset ports(sxi_aresetn) that provide reset signaling for the AXI Ports. All the other ports are rendered useless for our purpose.

3.3.4 IP Modes

By completing this IP we get an MMU IP with 2 AXI4 Ports. This MMU has 2 modes. The Translation Mode where it translation of an address will occur, according to the translation table that is configured in the Store mode.
The modes are user swappable and can be controlled through Register 1.

Store Mode

Store mode enables us to append a Physical Part to a Virtual Part of an address in order for it to be translated at a later time.
The whole process starts by giving an almost arbitrary 16 bit address, that will serve as the Virtual Part, to Register 0. Then another 16 bit address must be given at Register 1. This will be the 16 bits that the Virtual Address will be translated to. At the same time we have to enable Bit 16 of Register 1 so that the MMU will be put in Store Mode and store the translation in its RAM.

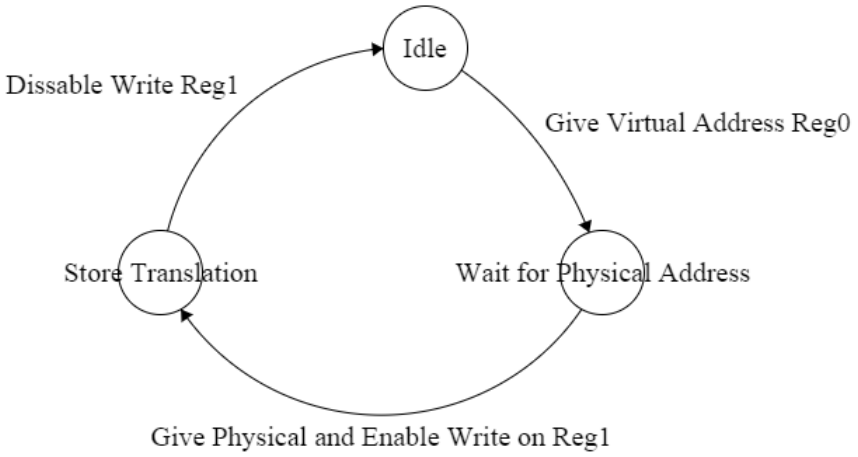

Figure 3.6: Store Mode FSM

In the event that we need to perform another Store, Bit 16 of Register 1 must be set to '0' and the process must start from the beginning. If this routine is not followed, overwrite issues may occur.

Translation Mode

Translation Mode is the "Normal" mode of the MMU in which it accepts a Virtual Address and Data, storing the Data on the Translated Address.
It all starts by giving the Virtual Address on Register 0. After that, the Data for storing must be given on Register 3. Then simply enable the AXI Transaction signal on Register 2 and the MMU will translate the address and store the Data on the translated address.

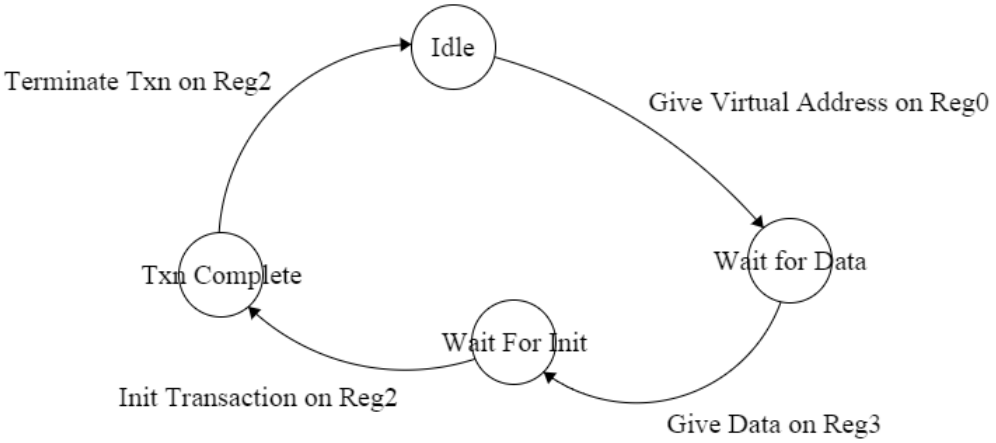

Figure 3.7: Translation Mode FSM

In order to perform another translation, the AXI Transaction signal on Register 2 must be set to ”0” again, to avoid overwriting existing data.

3.4 Design

The main focus of the Design is enable us to test multiple scenarios. For example we want to have multiple targets, where we could write, so that we can test the MMUs ability to translate and write on multiple targets as well as in different bases within the same target.
In order to implement the design we use Vivados IP intergrator and we create a new Block Design. Within this Block design we will add our IP Blocks and connect them appropriately

3.4.1 PS

The Processing System 7 core is the software interface around the Zynq-7000 platform processing system. The Zynq-7000 family consists of an SoC style integrated PS and a PL unit, providing an extensible and flexible SoC solution on a single die. The Processing System 7 core acts as a logic connection between the PS and the PL while assisting you to integrate customized and embedded IP cores with the processing system using the Vivado® IP integrator.

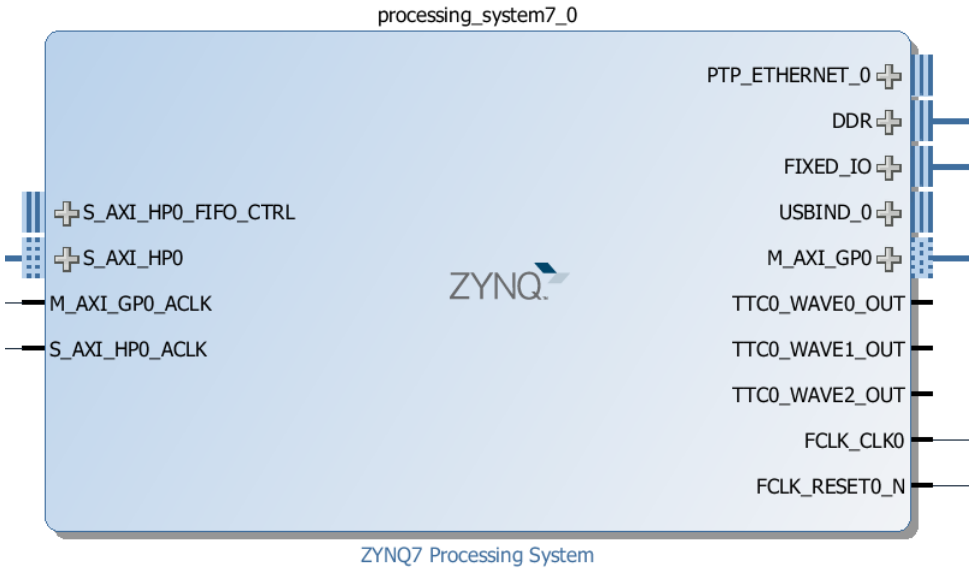

Figure 3.8: Zynq Processing System

The Processing System 7 wrapper instantiates the processing system section of the Zynq®-7000 All Programmable SoC for the programmable logic and external board logic. The wrapper includes unaltered connectivity and, for some signals, some logic functions. For a description of the architecture of the processing system, see the Zynq-7000 All Programmable SoC Technical Reference Manual (UG585)

3.4.2 Targets

As mentioned before, we need to have multiple targets in order to test various scenarios. For this Design we chose to use 2 Memory Blocks(one 128k and one 8k) and the DDR Memory of the Zedboard.
The Memory Blocks will be Dual Port ones to make us able to access them both from the PS and the MMU.

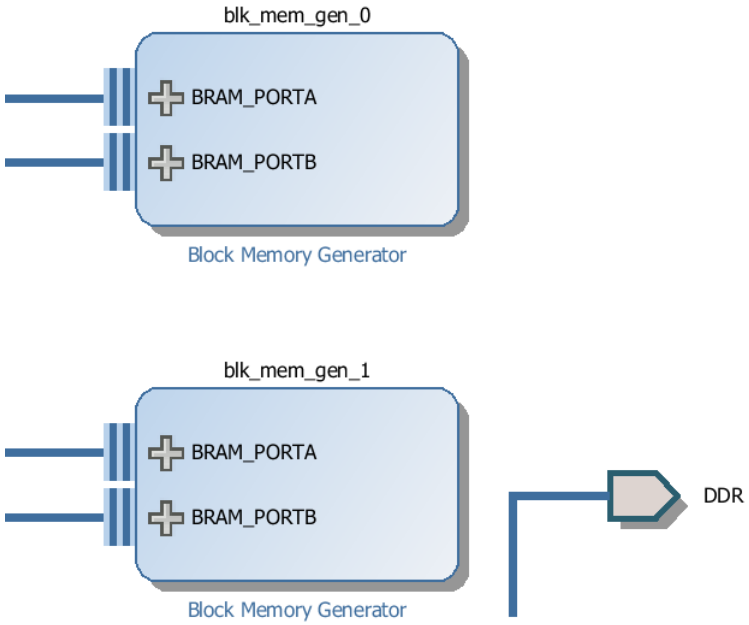

Figure 3.9: Targets

The MMU IP will access the Block Memory through its AXI4 Master Port. It will also access the DDR but this time through the Zynq PS HP0 AXI4 Port.
The Zynq® PS will also access the Block Memory through its AXI4 GP0 Port in order to confirm the correct function of the MMU IP.
In our design we will use an AXI BRAM Controller to enable us to access the Block Memory through an AXI4 Port. The DDR does not need such a controlled due to the fact that the PS has an integrated DDR Controller

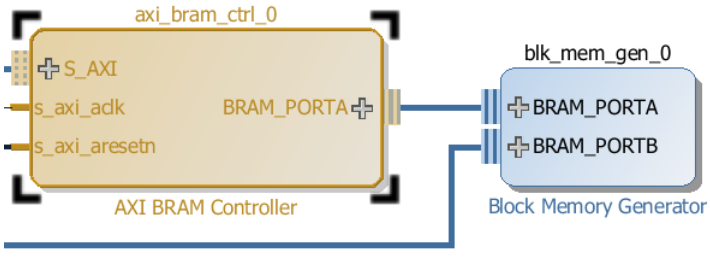

Figure 3.10: AXI BRAM Controller

3.4.3 AXI Interconnects

The AXI Interconnect core can be added to a Vivado® IP integrator block design in the Vivado Design Suite. The Interconnect IP core represents a hierarchical design block that become configured and connected during our system design session.
It allows any mixture of AXI master and slave devices to be connected to it, which can vary from one another in terms of data width, clock domain and AXI subprotocol (AXI4, AXI3, or AXI4 Lite). When the interface characteristics of any connected master or slave device differ from those of the crossbar switch inside the interconnect, the appropriate infrastructure cores are automatically inferred and connected within the interconnect to perform the necessary conversions.

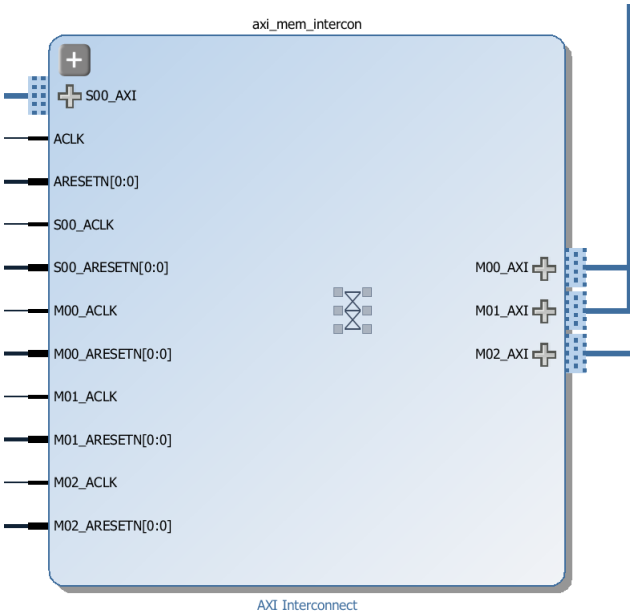

Figure 3.11: Interconnect

In this design we use the AXI Interconnects in order for the MMU and the PS to access all three Targets through one Master port.

3.4.4 Complete Design

Merging it all together we end up with the following design.

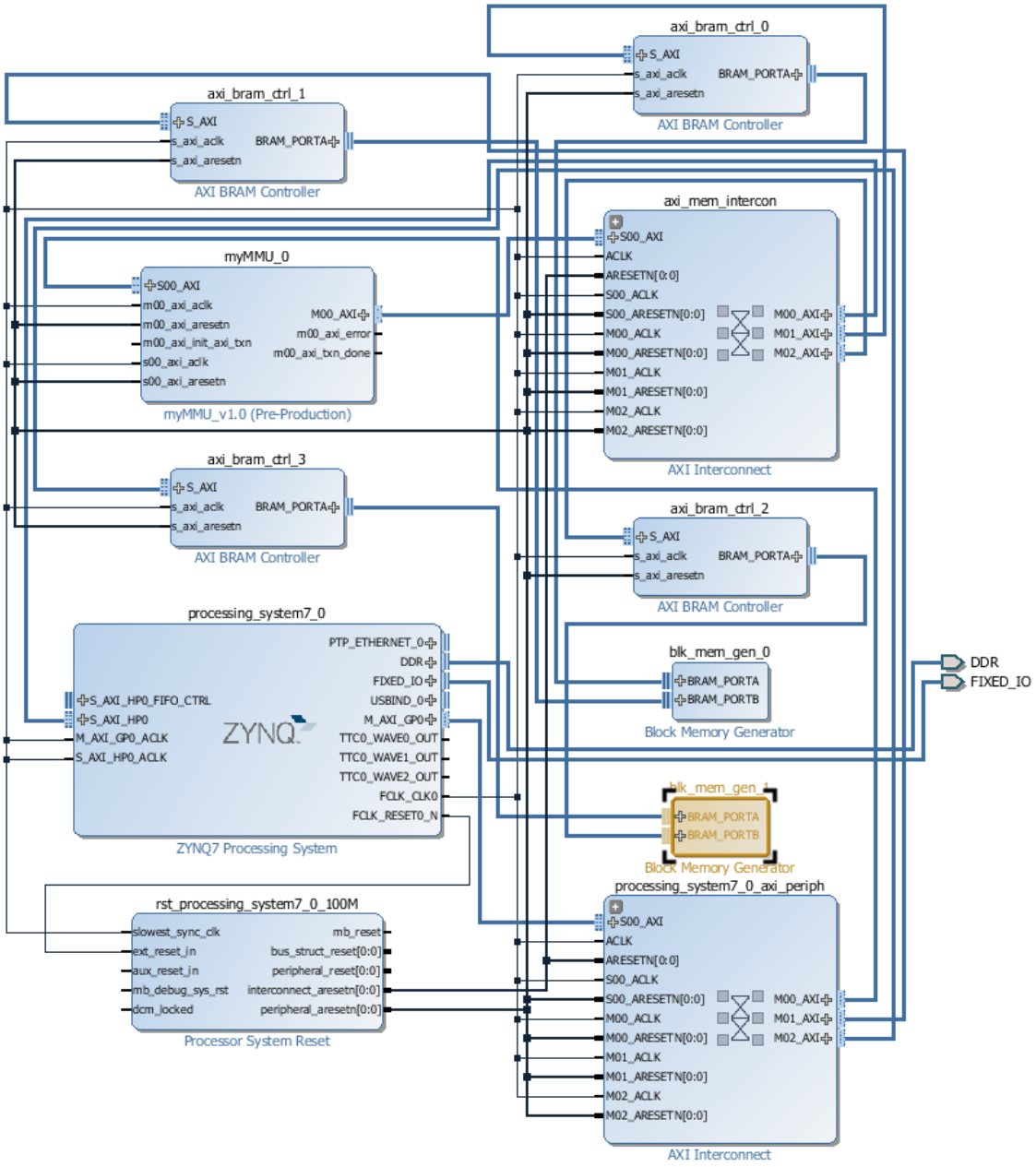

Figure 3.12: Complete Design

Vivado automatically configured the connection with the DDR.
After everything is set we let Vivado generate the Bitstream that will later be downloaded on the Zedboard so that we can test the MMU.

Chapter 4

Testing

After each stage of implementation, testing is required to ensure the proper function of each component of the IP.

4.1 Ram Testing

After we write the HDL code for the RAM we have to ensure that it actually works. The constraints we have is that it needs to be 16 bit wide with 16 bits of data per cell.

4.1.1 TestBench

Listing 4.1: RAM Test Bench

```
44 stimulus: process
45  begin
46   we      <='1';
47   wait for 10 ns;
48   address <="0000000000000000";
49   datain <="0000000000000000";
50   wait for 10 ns;
51   address <="0000000000000001";
52   datain <="0000000000000001";
53   wait for 10 ns;
54   address <="1111111111111110";
55   datain <="1111111111111110";
56   wait for 10 ns;
57   address <="1111111011111110";
58   datain <="1111111011111110";
59   wait for 10 ns;
60   address <="1111111111111100";
61   datain <="1111111111111100";
62   wait for 10 ns;
63   --disable writing on ram and read from previously filled cells
64   we     <='0';
```

```
65    wait for 10 ns;
66    address<="0000000000000000";
67    wait for 10 ns;
68    address<="0000000000000001";
69    wait for 10 ns;
70    address<="1111111111111110";
71    wait for 10 ns;
72    address<="1111111111111100";
73    wait for 10 ns;
74    --read from not filled cell for testing
75    address<="1101110111101100";
76    wait for 10 ns;
77    stop_the_clock <= true;
78    wait;
79  end process;
```

At line 31-35 we declare the component for testing and its port map. At 41-43 we initialise our signals and at 46-62 we fill random cells with random data.
The 64th line is used to disable RAM writing and after that we read from the previously written addresses to ensure correct function. At line 75 we read from a non-filled cell for testing reasons.

4.1.2 Results

After performing a Behavioural Simulation the waveforms, produced, showed that our writing attempts were successful.
It takes 1 clock cycle to complete one Write on the RAM.

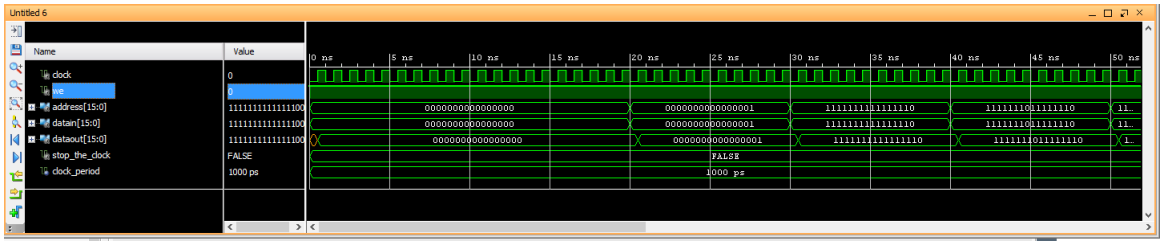

Figure 4.1: RAM Write Simulation

After 60 ns we disable writing and try to read from the cells we previously write on. As the waveforms suggest we have successfully read from the addresses and got valid data back.
Finally we tried to read from a address we did not write on and, as expected, got back invalid data(unsigned).

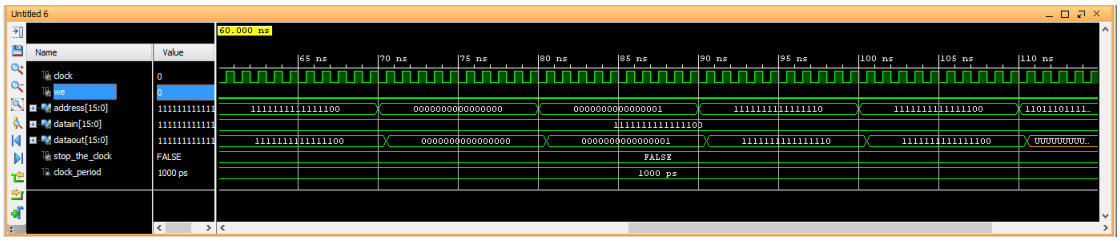

Figure 4.2: RAM Read Simulation

To ensure that we could not write on the ram while the WE signal is enabled, we simulated such situation and as expected we could not write when WE=0.

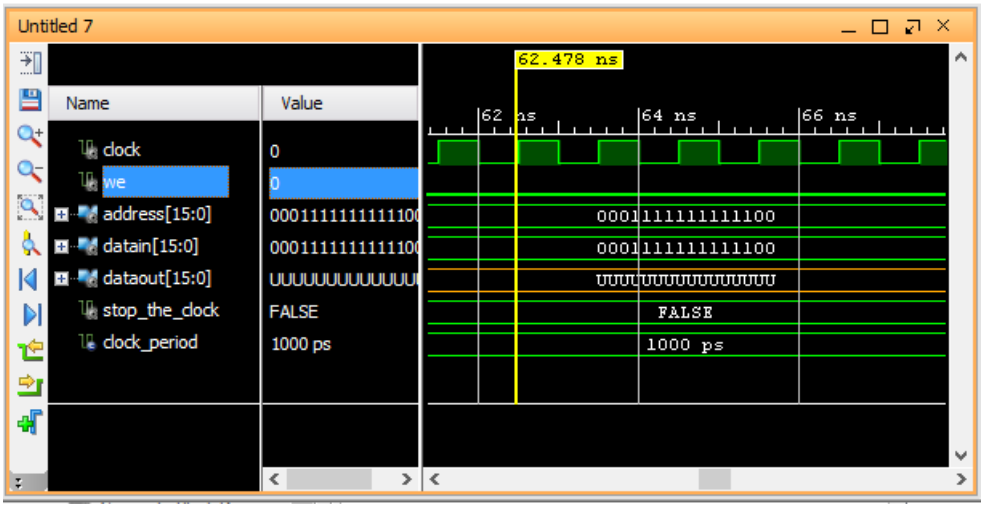

Figure 4.3: RAM Write Fail

4.2 Top Level Testing

After we have successfully tested the RAM module we move on writing a VHDL test bench for the top level of the MMU, which takes care of the bit routing.
The objective is to create a module that can accept a 32 bit address and either store and append the first 16 bits with 16 other bits, or translate it based on the stored translation.

4.2.1 TestBench

Our testbench will be very similar with our previous one in terms of testing.

Listing 4.2: Top Level Simulation Code

```
37  stimulus: process
38   begin
39
40    -- Put initialisation code here
41    virt   <="00000000000000000000000000000000";
42    datain <="0000000000000000";
43    we     <='0';
44    -- Put test bench stimulus code here
45    --enable writing on mmu ram and fill some cells with random data
46    we     <='1';
47    wait for 10 ns;
48    virt   <="00000000000000000000000000000000";
49    datain <="1111111111111111";
50    wait for 10 ns;
51    virt   <="00000000000000010000000000000000";
52    datain <="1111111111111110";
53    wait for 10 ns;
54    virt   <="11111111111111100000000000000000";
55    datain <="0000000000000001";
56    wait for 10 ns;
57    virt   <="11111110111111100000000000000000";
58    datain <="0000000100000001";
59    wait for 10 ns;
60    virt   <="11111111111111000000000000000000";
61    datain <="0000000000000011";
62    wait for 10 ns;
63    --disable writing on ram and read from previously filled cells
64    we     <='0';
65    virt <="00011111111111000001111111111100";
66    wait for 10 ns;
67    virt <="00000000000000000000000000000000";
68    wait for 10 ns;
69    virt <="00000000000000010000000000000001";
70    wait for 10 ns;
71    virt <="11111111111111101111111111111110";
72    wait for 10 ns;
```

```
73    virt <="11111111111111001111111111111100";
74    wait for 10 ns;
75    --read from not filled cell for testing
76    virt <="11011101111011001101110111101100";
77    wait for 10 ns;
78    stop_the_clock <= true;
79    wait;
80  end process;
```

At line 46 "we" is set to '1' so that we can access and write on the RAM. After that we store the virtual part of the address and append it to the physical part. For the virtual part, the first 16 bits of "virt" are used and for the physical part the "datain" is used.
Next we try to translate some addresses by giving a address on "virt". The first 16 bits are the ones that will be translated and the last 16 will remain unchanged, therefore can be random for the purpose of this test.

4.2.2 Results

As we can see from the waveforms our writes were completed successfully.

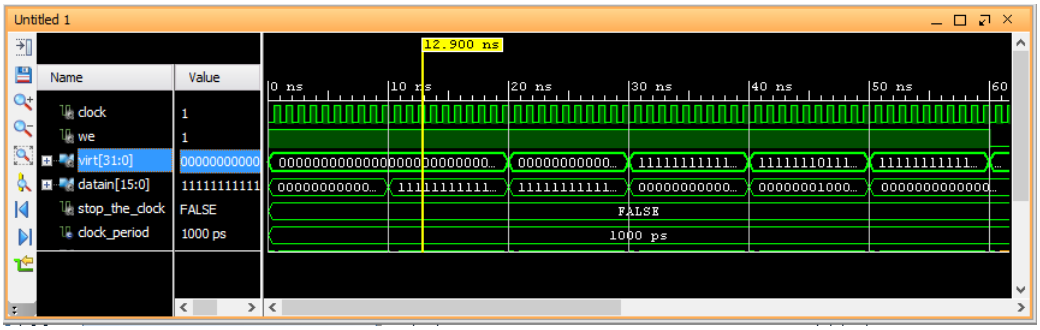

Figure 4.4: Top Level Storing

Next, we need to ensure the correct address translation, by entering a 32 bit address using a stored translation and expecting the module to change its first 16 bits with the ones we stored.

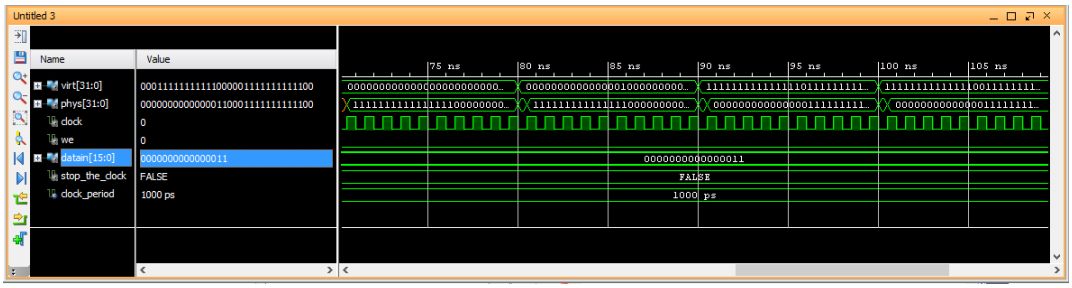

Figure 4.5: Top Level Translating

The waveform indicates that our addresses are translated accordingly. No problems were identified and the clock cycles it took to perform the actions needed are satisfactory.

4.3 IP Testing

After ensuring that the MMU works as defined and used Vivados Create and package IP tool to create our Master and Slave AXI4 Lite ports(as described in 3.3) we use Xilinxs Integrated Logic Analyser(ILA) to debug and monitor the AXI4 transactions.
The testing took place on the fully developed design described in 3.4.4.
During these tests, data was stored in the DDR-memory using the XMD environment.

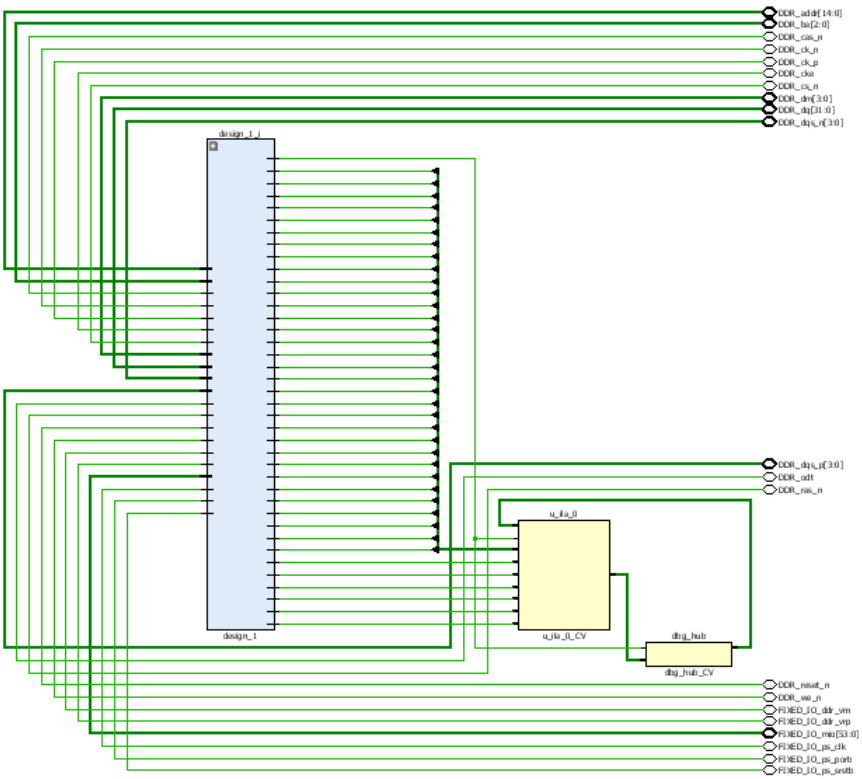

Figure 4.6: Synthesized Design with ILA core

First we have to choose which signals need to be monitored. We will choose them based on the importance hey have on the transaction. The target is to "see" the AXI transaction and validate that it is taking place as specified by the AMBA AXI4 protocol.
We also have to ensure that the Storing and Translation are taking place. We can do this by monitoring the AWADDR and WDATA signals.

In total, to have a complete picture of the transactions held place through our MMU IP we have to monitor the following signals.

Name	**Description**
Write Address Channel	
AWVALID	Write address valid. This signal indicates that the master signaling valid write address and control information.
AWREADY	Write address ready. This signal indicates that the slave is ready to accept an address and associated control signals.
AWADDR	Write address (issued by master, acceped by Slave)
Write Data Channel	
WDATA	Master Interface Write Data Channel ports. Write data (issued by master)
WVALID	Write valid. This signal indicates that valid write data and strobes are available.
WREADY	Write ready. This signal indicates that the slave can accept the write data.
Write Response Channel	
BREADY	Response ready. This signal indicates that the master can accept a write response.
BVALID	Write response valid. This signal indicates that the channel is signaling a valid write response
Inner IP Communications	
m_in_port0	This port on the Master interface is used to pass the Translated Address from the MMU, through the Slave Port to the Master Port.
m_in_port1	This port on the Master interface is used to pass the Data for storing from Register 3 of the Slave Port to the Master Port, to be later forwarded to the target.

Table 4.1: MMU AXI Signals

We choose these signals to ensure that the MMU is producing correct translations, to verify inter-IP communication and to prove the validity of the AXI4 Lite transaction.

4.3.1 Testing Methodology

In order to test the IP we will use the Vivado Hardware Manager and its Debug interface capture the values of the aforementioned signals. After we choose the signals for debugging we will set the data depth of the ILA to 1024 samples, which will be enough to monitor a couple of transactions in order to test the design.
After that we have to download the implemented bitstream on the FPGA and initialize it. When this is done we have to set a trigger. This trigger serves as a starting point for the ILA core to monitor. The AWVALID is perfect for this role as it is enabled at the start of an AXI4 Lite transaction.

Testing Scripts

For our ease, a couple of .tcl files are composed to initialise the MMU(see appendix). The test.tcl file stores 4 translations on the RAM of the MMU, described below.

Target	Virtual Base Address	Physical Base Address
DDR	0x11110000	0x00000000
8k BRAM	0x22220000	0x43000000
128K BRAM	0x33330000	0xC0000000
128K BRAM	0x44440000	0xC0010000

Table 4.2: Testing Translations

After the translations are stored, four discrete data words are stored on each target. That is when the ILA core starts monitoring the signals. After the test.tcl has run successfully we can study the waveforms produced.

Target	Address	Data
DDR	0x1111000C	0xAAAAAAAA
8k BRAM	0x2222001C	0xBBBBBBBB
128K BRAM	0x33330008	0xCCCCCCCC
128K BRAM	0x44440004	0xDDDDDDDD

Table 4.3: Testing Stored Data Addresses

As we can see the AXI4 Lite Transaction handshake has been completed and the inter-IP communication is as expected. On the write address channel we can see the translated address. On the write data channel the data is shown as well as the validity of the write. On the write response channel we see the BVALID signal signalling a successful response. All went as planed and our IP is translating the address as we programmed it to do as well as obeying the AXI4 Lite communication protocol.

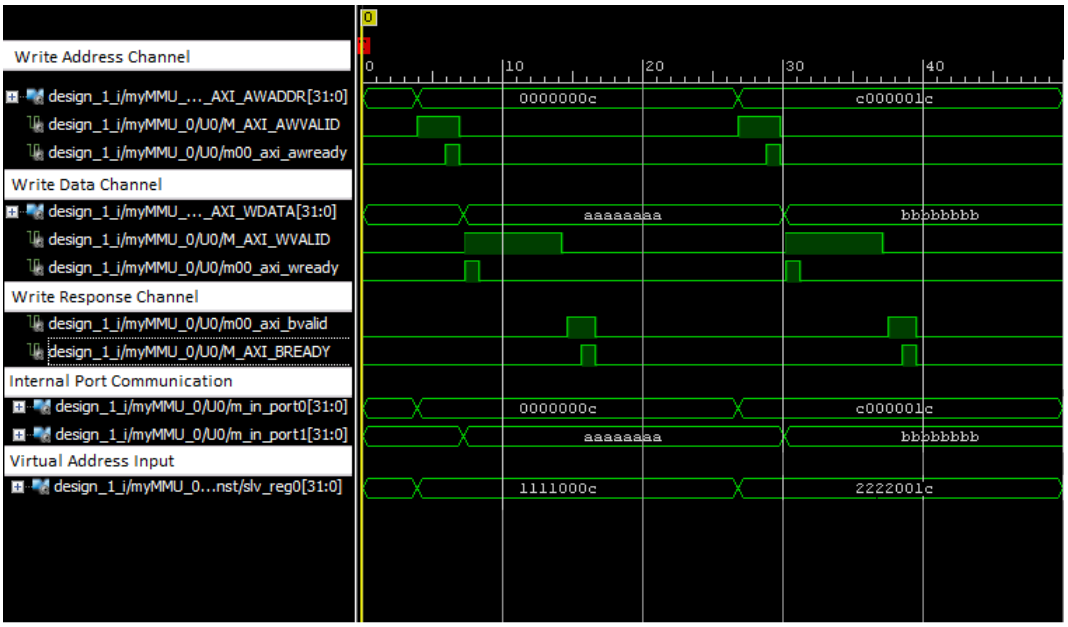

Figure 4.7: ILA Monitoring of Writes A

This figure provides an insight on what happens when we complete an write through the MMU. In figure 4.7 we see the two first writes. As soon as the address is translated an transfered to the Master port, the AWVALID signal becomes high. After the awready signals becomes high too, indicating that the slave is ready to accept an address and associated control signals, the Data are sent. THe wredy signal indicates that the slave can accept the write data. WVALID signals high when the AXI write is successful.

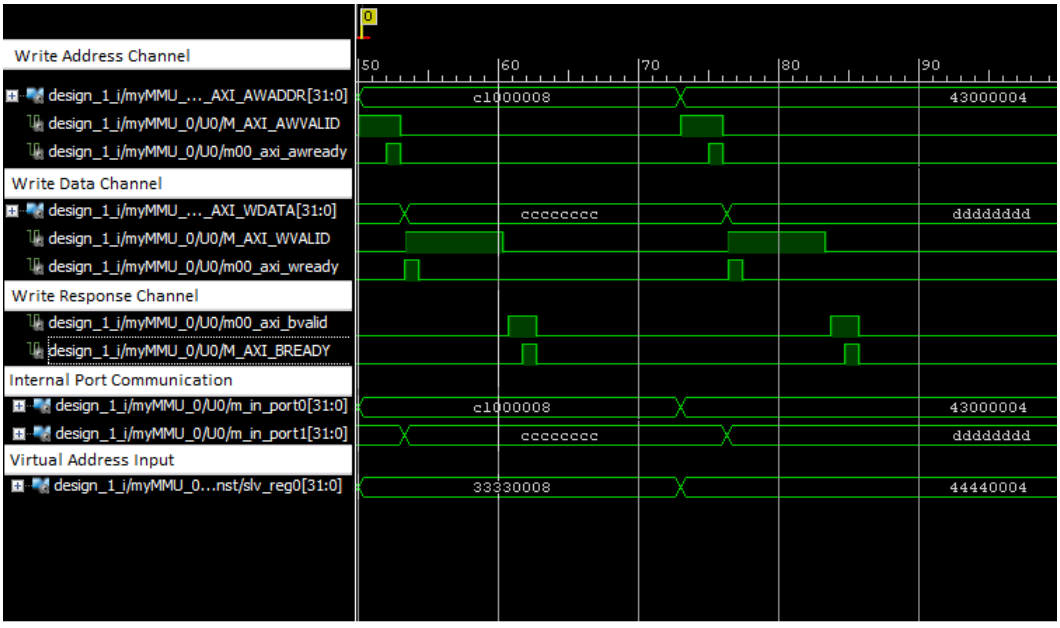

Figure 4.8: ILA Monitoring of Writes B

The m_in_port 0 and 1 ports indicates that the inner-IP communication is sound. The slv_reg0 port shows the Virtual Address making us sure that the correct Address

is being translated. Timing issues are not present and the translation consumes minimum clock cycles. Since our clock frequency is 100Mhz the clock period is 10 ns and a full AXI4 Lite Write takes 120 ns, 12 clock cycles are consumed per write.

Chapter 5

Conclusions

This chapter describes the conclusion of this THesis, future works that can be made on the MMU IP and thoughts about how to implement new features.
The main conclusion that can be made from this thesis is that a successfully working MMU IP for use in Zynq 7000 SoCs integrated with AXI4 could be implemented. Implemented functions works as expected and the used hardware presented in 2.1 is acceptable. The implementation should not need much adjustments in the future, but more features can of course be added. The speed of the translation is satisfying although it can be improved. The source code of the IP is open so it could easily be integrated as a lab example for academic courses on Computer Architecture to enable students to experiment on hardware design.

5.1 Future Work

A true dual port RAM could be implemented in the future to enable multiple translations at the same time. This will also, in theory, enable different applications to share data without the need of CPU intense computing.
In addition clocking improvements can be made so that the MMU IP is more efficient and less time consuming.

Appendix A

Abbreviations

AMBA .... Advanced Micro-controller Bus Architecture
AXI ..... Advanced eXtensible Interface
BRAM .... Block Random Access Memory
CLB ..... Configurable Logic Block
CPU ..... Central Processing Unit
DDR ..... Dual Data Rate
DPRAM .. Dual Port Random Access Memory
FPGA .... Field Programmable Gate Array
FSM ..... Finite State Machine
GP ..... General Purpose
HDL ..... Hardware Description Language
HP ..... High Performance
IDE ..... Integrated Development Environment
ILA ..... Integrated Logic Analyser
IP ..... Intellectual Property
MMU ..... Memory Management Unit
PA ..... Physical Address
PL ..... Programmable Logic
PLB ..... Programmable Logic Blocks
PS ..... Processing System
RAM ..... Random Access Memory
SB ..... Switch Blocks
SDK ..... Software Development Kit
SOC ..... System On Chip
TCL ..... Tool Command Language
TXN ..... Transaction

VA ..... Virtual Address

$VHDL$ .... VHSIC Hardware Description Language

WE ..... Write Enable

XMD ..... Xilinx Microprocessor Debug

Appendix B

.tcl Scripts

B.1 test.tcl

```
1  #store 1st Translation
2  mwr [expr 0x43C00000] [expr 0x11110000]
3  mwr [expr 0x43C00004] [expr 0x11110000]
4  mwr [expr 0x43C00004] [expr 0x00000000]
5  #store 2nd Translation
6  mwr [expr 0x43C00000] [expr 0x22220000]
7  mwr [expr 0x43C00004] [expr 0x111143C0]
8  mwr [expr 0x43C00004] [expr 0x00000000]
9  #store 3rd Translation
10 mwr [expr 0x43C00000] [expr 0x33330000]
11 mwr [expr 0x43C00004] [expr 0x1111C000]
12 mwr [expr 0x43C00004] [expr 0x00000000]
13 #store 4th Translation
14 mwr [expr 0x43C00000] [expr 0x44440000]
15 mwr [expr 0x43C00004] [expr 0x1111C001]
16 mwr [expr 0x43C00004] [expr 0x00000000]
17 #Store Random Data On 1st Target Using Virtual Base Address
18 mwr [expr 0x43C00000] [expr 0x1111000C]
19 mwr [expr 0x43C0000c] [expr 0xAAAAAAAA]
20 mwr [expr 0x43C00008] [expr 0x11111111]
21 mwr [expr 0x43C00008] [expr 0x00000000]
22 #Store Random Data On 2nd Target Using Virtual Base Address
23 mwr [expr 0x43C00000] [expr 0x2222001C]
24 mwr [expr 0x43C0000c] [expr 0xBBBBBBBB]
25 mwr [expr 0x43C00008] [expr 0x11111111]
26 mwr [expr 0x43C00008] [expr 0x00000000]
27 #Store Random Data On 3rd Target Using Virtual Base Address
28 mwr [expr 0x43C00000] [expr 0x33330008]
29 mwr [expr 0x43C0000c] [expr 0xCCCCCCCC]
30 mwr [expr 0x43C00008] [expr 0x11111111]
31 mwr [expr 0x43C00008] [expr 0x00000000]
32 #Store Random Data On 4th Target Using Virtual Base Address
33 mwr [expr 0x43C00000] [expr 0x44440004]
34 mwr [expr 0x43C0000c] [expr 0xDDDDDDDD]
35 mwr [expr 0x43C00008] [expr 0x11111111]
```

```
36 mwr [expr 0x43C00008] [expr 0x00000000]
```

B.2 call.tcl

```
1  puts "Give Address"
2  set addr [gets stdin]
3
4  puts "Give Data"
5  set data [gets stdin]
6
7  # Set address reg
8  mwr [expr 0x43C00000] $addr
9  # Set data reg
10 mwr [expr 0x43C0000c] $data
11
12 mwr [expr 0x43C00008] [expr 0x11111111]
13 mwr [expr 0x43C00008] [expr 0x00000000]
```

B.3 store.tcl

```
1 puts "Give Virtual Address"
2 set va [gets stdin]
3
4 puts "Give Translation Address"
5 set pa [gets stdin]
6
7 mwr [expr 0x43C00000] $va
8 mwr [expr 0x43C00004] $pa
9 mwr [expr 0x43C00004] [expr 0x00000000]
```

Bibliography

[1] Xilinx. *AXI Reference Guide*. UG761 (v13.2) July 6, 2011.

[2] Xilinx. *AXI Interconnect v2.1 LogiCORE IP Product Guide*. Vivado Design Suite PG059 April 1, 2015.

[3] Xilinx. *UltraFast Design Methodology Guide for the Vivado Design Suite*.UG949 (v2015.1) June 1, 2015.

[4] Emelie Nilsson. *DMA Controller for LEON3® SoC:s Using AMBA*.LiTH-ISY-EX–13/4663–SE Linköping 2013.

[5] Peter J. Ashenden. *The Designer's Guide to VHDL*.Second Edition (Systems on Silicon).

[6] Frederik Zandveld, Matthias Wendt, Marcel D. Janssens. *Sparc RISC based computer system including a single chip processor with memory management and DMA units coupled to a DRAM interface*.US5659797 A Grant US 07/896,062 Aug 19, 1997.

[7] Frank Uyeda. *Lecture 7: Memory Management*.CSE 120: Principles of Operating Systems. UC San Diego. Retrieved 2013-12-04.

[8] Baris Simsek. *Memory Management for System Programmers*. 2005 http://www.enderunix.org/simsek/.

[9] Richard E. Haskell Darrin M. Hanna. *Introduction to Digital Design Using Digilent FPGA Boards* . Oakland University, Rochester, Michigan.

[10] Pong P. Chu. *FPGA Prototyping By VHDL Examples: Xilinx Spartan-3 Version*. 1st Edition.

[11] Motorola Semiconductors. *MC68451 MEMORY MANAGEMENT UNIT*. APRIL, 1983.

[12] Peter J. Denning. *BEFORE MEMORY WAS VIRTUAL*. George Mason University DRAFT 6/10/96.

[13] Jessen, E. *Origin of the Virtual Memory Concept.*. IEEE Annals History of Computing, pp. pp 71-72, 2004.

[14] Michel Wilson. *Memory controller for a 6502 CPU in VHDL*. , 1047981 michel@crondor.net May 2006.

[15] Amir Shenouda. *Simple Memory Management Unit with VHDL*. SOFTWARE ENGINEER NOTES FRIDAY, JUNE 8, 2012.

[16] Liu, K.-J., You, H.-L., Yan, X.-L., Ge, H.-T. *Design of application specific embedded memory management unit*. (2007) Zhejiang Daxue Xuebao (Gongxue Ban)/Journal of Zhejiang University (Engineering Science), 41 (7), pp. 1078-1082+1087.

[17] Li, Shuguo, Liu, Shibin, Gao, Deyuan, Fan, Xiaoya. *Research of memory management unit*. (2000) Xibei Gongye Daxue Xuebao/Journal of Northwestern Polytechnical University, 18 (3), pp. 357-359.

[18] Milenkovic, Milan. *Microprocessor memory management units*. (1990) IEEE Micro, 10 (2), pp. 70-85.

[19] Bron, C., Dijkstra, E.J., Swierstra, S.D. *A memory-management unit for the optimal exploitation of a small address space*. (1982) Information Processing Letters, 15 (1), pp. 20-22.

[20] Bakshi, C., Bela, L. *A virtual memory system for real-time applications*. (1992) Proceedings - Real-Time Systems Symposium, art. no. 242661, pp. 219-222.

[21] WEIZER N. *Virtual memory. A combined hardware- software memory system*. (1971) 1 p.

[22] Bruce L. Jacob, Trevor N. Mudge. *A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations*. Dept. of Electrical Engineering and Computer Science University of Maryland, College Park.

[23] A. W. Appel and K. Li. *Virtual memory primitives for user programs*. In Proc. ASPLOS-4, April 1991, pp. 96–107

[24] B. L. Jacob and T. N. Mudge. *Virtual memory in contemporary microprocessors*. IEEE Micro, vol. 18, no. 4, July/August 1998.

[25] B. L. Jacob and T. N. Mudge. *Virtual memory: Issues of implementation*. IEEE Computer, vol. 31, no. 6, pp. 33–43, June 1998.

[26] Douglas J. Smith *HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog*. Doone Publications ©1998 ISBN:0965193438

[27] Xilinx *Integrated Logic Analyzer*. Vivado Design Suite PG172 October 1, 2014

[28] Andrew S. Tanenbaum, Albert S. Woodhull *OPERATING SYSTEMS DESIGN AND IMPLEMENTATION*. Operating Systems: Design and Implementation, (c) 2006 Prentice-Hall, Inc. All rights reserved. 0-13-142938-8

[29] Russell G. Tessier *Fast Place and Route Approaches for FPGAs*. Massachusetts Institute of Technology 1999

[30] Payam Lajevardi *Design of a 3-Dimension FPGA*. Massachusetts Institute of Technology 1999, July 2005.

[31] Yoongu Kim *Computer Architecture Lecture 17: Virtual Memory II*. Carnegie Mellon University Spring 2013, 2/27

[32] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi *Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management*. Proceedings of the 42nd International Symposium on Computer Architecture (ISCA), Portland, OR, June 2015. Spring 2013, 2/27