Tutorial 1: Sharing DRAM between Modules Using MPMC

University of California, Berkeley

Department of Electrical Engineering and Computer Sciences

EECS150 Components and Design Techniques for Digital Systems

John Wawrzynek, Shaoyi Cheng, Vincent Lee

Table of Contents

0 Introduction

This tutorial would guide you through the process of creating a system where coprocessors and the MicroBlaze core can all access the DRAM on the FPGA board. There are a few issues with the EDK in making connections and generating constraints, which we would point out and show the solutions in this tutorial.

There are no deliverables required from you for this tutorial. It is created to help those of you whose projects would contain multiple modules accessing DRAM.

1 Prerequisite

You are expected to have done Lab5B, in which a simple MicroBlaze system is built, and a simple coprocessor is added. This tutorial would not be providing details in how to build the basic system, but rather how to use the multiport memory controller.

Before you start, get the source code distribution from the class website.

```
wget http://inst.eecs.berkeley.edu/~cs150/sp13/tutorials/tut1.tgz
tar –xvf tut1.tgz
```

2 Hardware System

Base System Creation

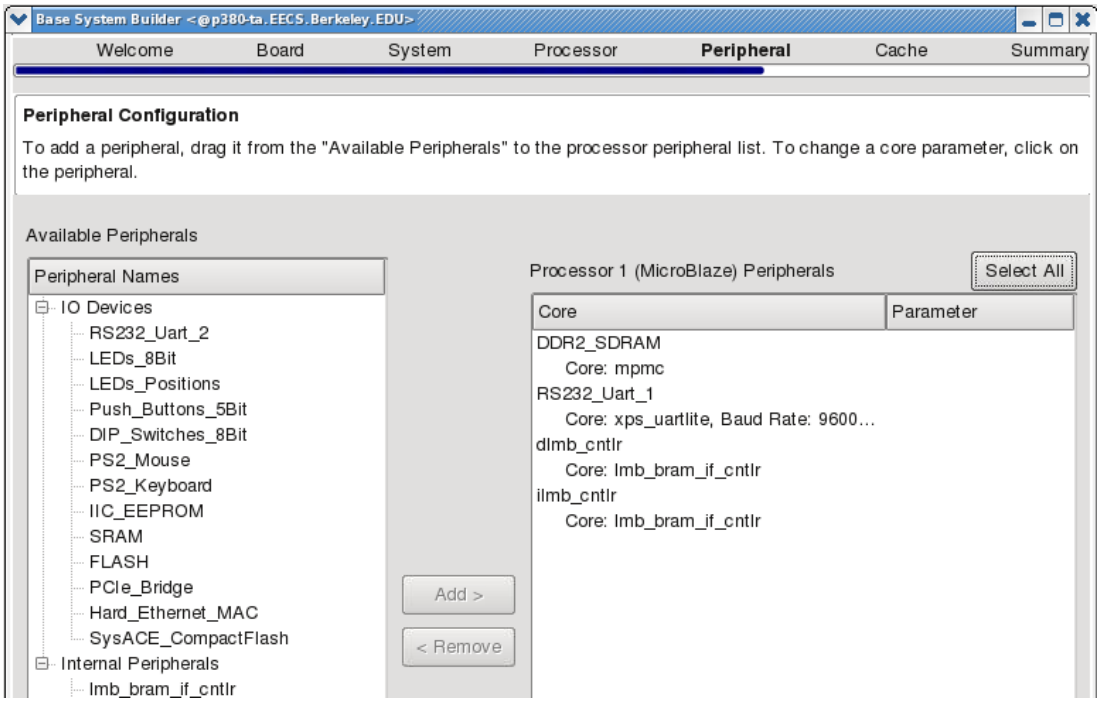
The system we are going to construct includes the following hardware components:

- MicroBlaze
- Local Memory Bus (LMB)
  - LMB_BRAM_IF_CNTLR
  - BRAM_BLOCK
- Processor Local Bus (PLB)
  - XPS_UARTLITE
  - MDM
  - **MPMC (Multiport Memory Controller)**
- Fast Simplex Link (FSL)
  - User defined coprocessors

Start Xilinx Platform Studio by typing the following command:

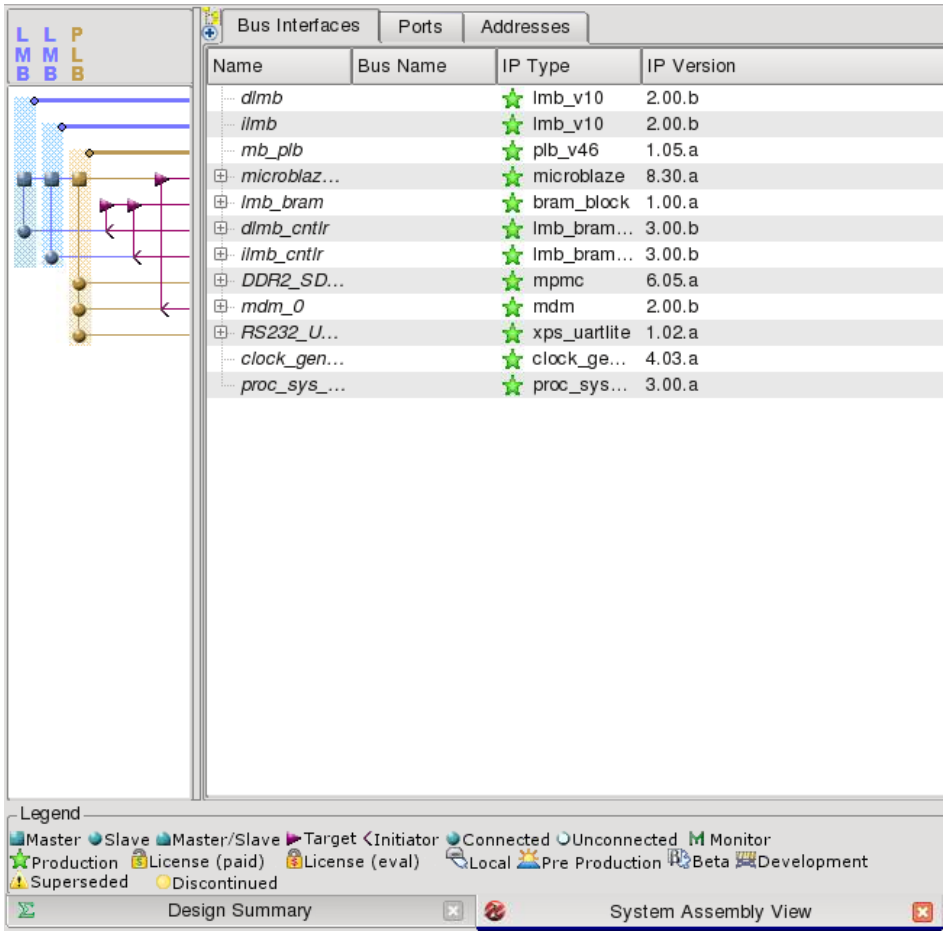
```
% xps
```

Create a new PLB system with a single MicroBlaze processor running at 62.5MHz, with 64KB of local memory; remember to use the board definition file from Lab5B. When you reach the peripheral configuration page, remove all peripherals except RS232_Uart_1, dlmb_cntlr , ilmb_cntlr and DDR2_SDRAM.



Similar to Lab5B, the RS232_Uart_1 will be used to communicate with the host workstation, while the lmb_cntlr and ilmb_cntlr are used by the processor to access the local memory (Block RAM). The DRAM controller added has multiple ports. We can thus have a coprocessor writing into one port, while the MicroBlaze and other coprocessors reading from other ports.

Continue the system builder to finish the base system creation, when you should see the summary page below.

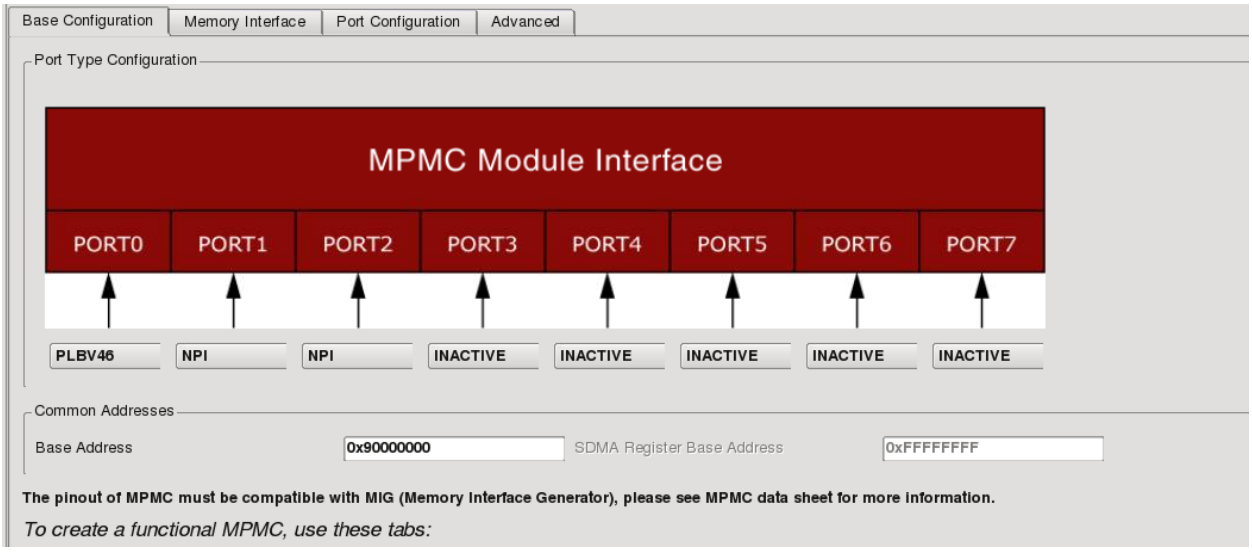


Now, add a coprocessor, named `writecop`, connected to the MicroBlaze using FSL link. For your coprocessor, you should only have the input FSL interface (uncheck `Output FSL Interface`). Configure the interface to have two 32-bit input words. In the `Peripheral Implementation Support` page, select all three options to generate the project files, the driver files and the example design in Verilog. Accept default for all the remaining pages before you generate the coprocessor. This coprocessor, as the name suggests, would be used to write to the DRAM.

To demonstrate how to use multiple coprocessors in your design and how to read from DRAM, here we would add another coprocessor `readcop`, again connected to the MicroBlaze using FSL. It should have both input and output FSL interfaces, each with one 32-bit word.

Since we would like to have the MicroBlaze and the coprocessors accessing the DRAM from different ports, it is necessary to activate more than one port in the DDR2 controller. To do so, right click

DDR2_SDRAM in the system assembly view, and select `configure IP`. In the configuration dialog, activate two other ports using the drop down menu. Select `NPI` for the new ports, this is the protocol the coprocessors will be using for communicating with the MPMC. It stands for Native Port Interface, which is a low overhead, high bandwidth communication mechanism between the DRAM and custom IP. Press `OK` to go back to the system assembly view.



Select `Hardware → Configure Coprocessor` to add the newly created coprocessors to the system (readcop first, then writecop). They are now connected to the MicroBlaze through FSL interface, but the connections to the NPI interfaces in MPMC have yet to be established.

Connecting Coprocessor with MPMC using NPI

Before we connect the newly created coprocessors to the NPI ports of the MPMC, let's take a look at what files are actually generated by the Xilinx tools.

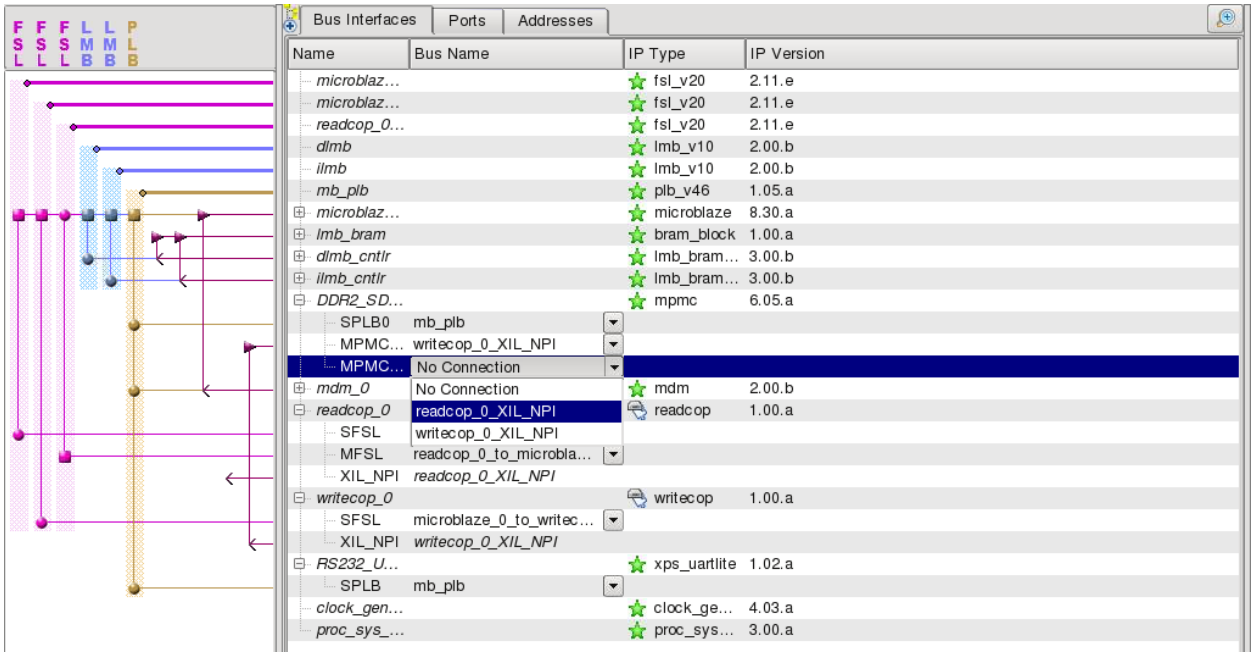
Under your XPS project directory, a `system.mhs` (Microprocessor Hardware Specification) file is created when XPS generates the system. In this file, the ports of the entire system, the peripherals the system contains, the ports and parameters of each peripheral, as well as the connections between different modules are all defined in human readable forms. The `system.ucf` in the `data` directory under your project contains the generated constraints for the design, which includes pin assignments and timing constraints. We would make manual changes to these files at times in this tutorial to resolve some issues the Xilinx tools have (incompatible constraints, incapability to connect some nets from the GUI etc.)

For the coprocessors, you can find the generated files in the folders `pcores/writecop_v1_00_a` and `pcores/readcop_v1_00_a`. In each of these folders, there are three directories `data, devel` and `hdl`. The `hdl` directory contains the example coprocessor implemented in Verilog. The `devel` directory contains the ISE project file the user can use for developing the coprocessor. In `data`, you can find an `.mpd` (Microprocessor Peripheral Definition) file which describes the interface of the peripheral. In order to establish the NPI connection, port definitions would need to be added to the Verilog file and the `.mpd` file of the coprocessor. The EDK GUI unfortunately does not have good support for creating NPI interfaces in custom IPs.

We have provided in our source code distribution (under `src/hardware`)the new `writecop.v`,

`writecop_v2_1_0.mpd, readcop.v` and `readcop_v2_1_0.mpd` file, copy them over to overwrite the existing version. In each of the `.mpd` files, an NPI section has been added. It should be obvious from the comments in the file. Similarly, the provided `.v` files also include the added NPI ports, again marked with comments. The `writecop` module reads in two inputs from the FSL bus, using one as address and the other as data, writes the value into the DRAM. The `readcop` module reads one input from the FSL bus, using it as the address, fetches the data from the DRAM and sends it back to the MicroBlaze using FSL.

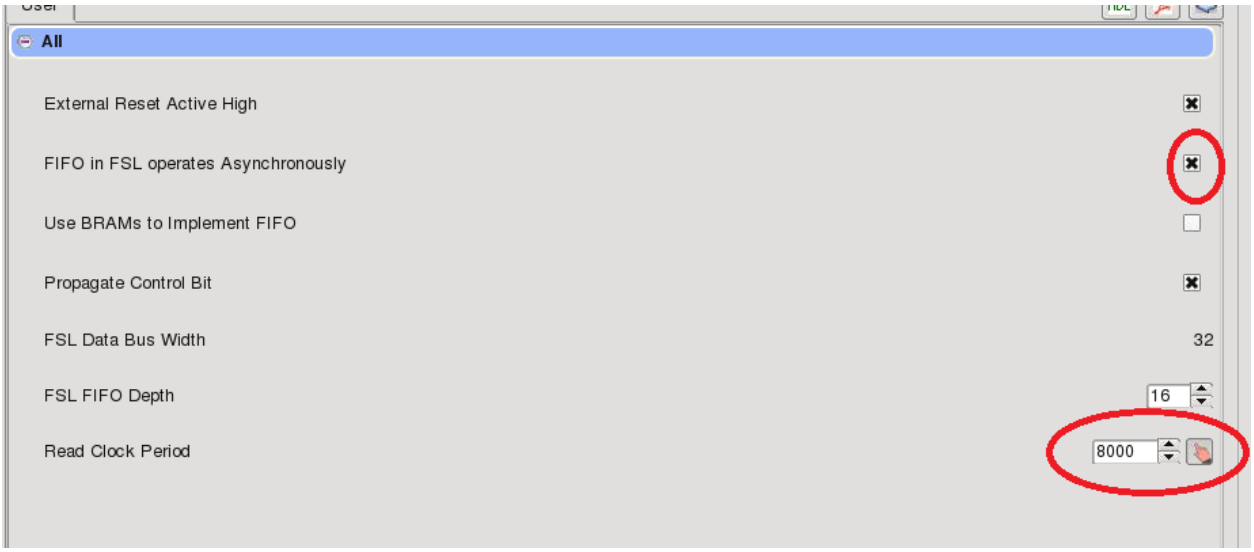
Once the `.mpd` and `.v` files are updated, go back to the XPS GUI, `Select Project→Rescan User Repositories`, you can now make NPI connections between the coprocessors and the MPMC, as shown in the screen below.



Clock Domain Issues

NPI provides a FIFO interface for both reading from and writing to DRAM. Read over the provided `.v` files to make sure you understand how to use NPI to write to and read from DRAM. One thing you might have noticed in the code is the use of `FSL_S_Clk` for reading/writing of NPI. This clock is different than the clock used by the MicroBlaze core. In fact, the coprocessor is operating in the clock domain of the MPMC. Therefore, we will need to set the operating mode of the FSL bus to be asynchronous. From the `System Assembly View` tab, find `microblaze_0_to_readcop_0`, right click to open the configure IP dialog. Check `FIFO in FSL operates Asynchronously,` also set the `Read Clock Period` to be 8000 (as shown in the screen below), and press `OK`. Do the same for the bus `microblaze_0_to_writecop_0.` These two buses are read by the coprocessor which runs at 125MHz, and thus the period of the read clock is 8000 ps.

For the data communication from the `readcop` coprocessor to the MicroBlaze, we have `readcop_0_to_microblaze_0`. Enable asynchronous FIFO for this bus as well, but set the `Read Clock Period` to 16000 instead. The FIFO would be read by the MicroBlaze core which runs at 62.5MHz.



After making the FSL FIFO asynchronous, we should connect the `FSL_S_Clk` and `FSL_M_Clk` of the FSL buses to the right clock. For the buses going from the MicroBlaze to the coprocessors, the slave clock `FSL_S_Clk` should be the same as the MPMC clock, and the master clock (`FSL_M_Clk`) should be the 62.5MHz clock used by the processor. On the other hand, for the FSL bus going from the coprocessor to the MicroBlaze, the master clock is the MPMC clock while the slave clock is the processor clock.

The XPS GUI has problems with connecting the clock nets, so we will directly modify the underlying specification file to make the connections.

Open the `system.mhs` file under your XPS project directory. You should be able to find multiple blocks of text starting with "`BEGIN fsl_v20`", each with a different `INSTANCE` parameter. These blocks configure the FSL buses used in our design. For the instances `microblaze_0_to_readcop_0` and `microblaze_0_to_writecop_0`, add the following lines just before the "`END`" in the block.

```
PORT FSL_M_Clk = clk_62_5000MHzPLL0
PORT FSL_S_Clk = clk_125_0000MHzPLL0
```

For the instance `readcop_0_to_microblaze_0`, add the following lines to the block:

```
PORT FSL_M_Clk = clk_125_0000MHzPLL0
PORT FSL_S_Clk = clk_62_5000MHzPLL0
```

The clock `clk_125_0000MHzPLL0` is used by the MPMC while `clk_62_5000MHzPLL0` is used by the processor. With the difference in clock domains handled by the asynchronous FIFO, we can have the coprocessor running much faster than the MicroBlaze.

Besides the FIFOs, we also need to connect the clocks of the coprocessors. In addition, each of the two coprocessors has a port `system_dcm_locked`, which tracks the stabilization of the clock generator in the system. We should connect all these nets in the `system.mhs` file as well.

For `writecop`, add the following lines:

```
PORT system_dcm_locked = Dcm_all_locked
PORT FSL_S_Clk = clk_125_0000MHzPLL0
```

For `readcop`, add the following lines:

```
PORT system_dcm_locked = Dcm_all_locked
PORT FSL_S_Clk = clk_125_0000MHzPLL0
PORT FSL_M_Clk = clk_125_0000MHzPLL0
```

Save the `system.mhs` file.

Now with the coprocessors properly connected, we are almost ready to implement the hardware design to bitstream. However, when EDK generates the MPMC module, it generates a set of problematic constraints which need to be removed. Open `system.ucf` in the `data` directory under your XPS project, scroll to the end of the file, you can see a group of `RLOC_ORIGIN` constraints. Remove these constraints and save the file.

Go back to the XPS GUI, do a DRC, if there are warnings, check with the TAs to make sure they are non-critical. Then export the design and launch SDK. (This process will take a while; find something else to do while you are waiting.)

In this design, the bandwidth of the NPI port is underutilized as the coprocessors only reads/writes two words at a time. In your design, you may want to transfer more words at a time, which can be accommodated by the interface. For further information, refer to page 33-36 in http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf.

3 Using the Coprocessors in Software

After the Xilinx SDK is launched, create an empty C application named `read_write_DRAM`. We have provided in our source code distribution a simple program to demonstrate the functionalities of the coprocessors. In addition, the linker script `lscript.ld` generated by default puts the program in the DRAM, which may get modified by the application. We have also provided a modified version of the linker script which puts the program in the BRAM. Copy all the `.c`, `.h` files and the `lscript.ld` file from the source code distribution (under `src/software`) to the `src` directory under your C application, refresh the project and wait for the executable to be generated.

Open up a terminal in your workstation and run:[1]

```
% screen $SERIALTTY 9600
```

[1] Note that if someone else has locked the serial cable on your workstation, you will not be able to execute this command. Since we don't have sudo access, the only way to resolve this problem is to either find the offending user and have him kill his session, or reboot the computer.

Configure the FPGA board with the generated elf file. You should be able to write to/read from memory directly from MicroBlaze through PLB bus, or indirectly through the NPI interfaces in the coprocessors. Read over the provided source code to learn how multiple coprocessors can be used in your design, and convince yourself the DRAM can be accessed from multiple ports.