ECEN 449: Microprocessor System Design
Department of Electrical and Computer Engineering
Texas A&M University

Prof. Peng Li
TA: Andrew Targhetta
(Lab exercise created by A Targhetta and P Gratz)

Laboratory Exercise #3

Creating a Custom Hardware IP and Interfacing it with Software

Objective

The purpose of lab this week is to familiarize you with the process of creating and importing a custom IP module for a MicroBlaze based system. We will be using the 'Create and Import Peripheral Wizard' in XPS in conjunction with ISE to develop a custom peripheral for performing integer multiplication. We will then integrate the integer multiplication peripheral into a microprocessor system and develop software to interact with the peripheral using XPS. This lab serves as a simple hardware/software co-design example.

System Overview

The microprocessor system you will build in this lab is depicted in Figure 1. In place of the GPIO modules, utilized in the last lab, is a multiplication block, which represents the integer multiplication peripheral you will create. The UART in Figure 1 will be used to connect the XUP board to the workstation computer which will display the output of software executing on the MicroBlaze processor. The software you will develop in this lab will provide proof of operation for your multiplication peripheral.

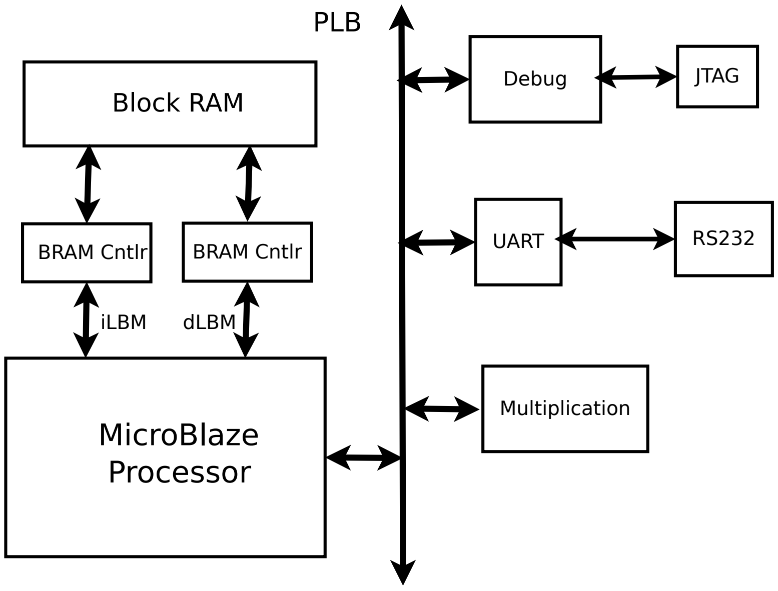


Figure 1: MicroBlaze System Diagram

Procedure

1. To begin, create the base system shown in Figure 1 using a similar configuration as last lab. We do not have the multiplication peripheral yet but will add that after we create it.

   (a) Create a folder for Lab 3 and open XPS as outlined in Lab 2.

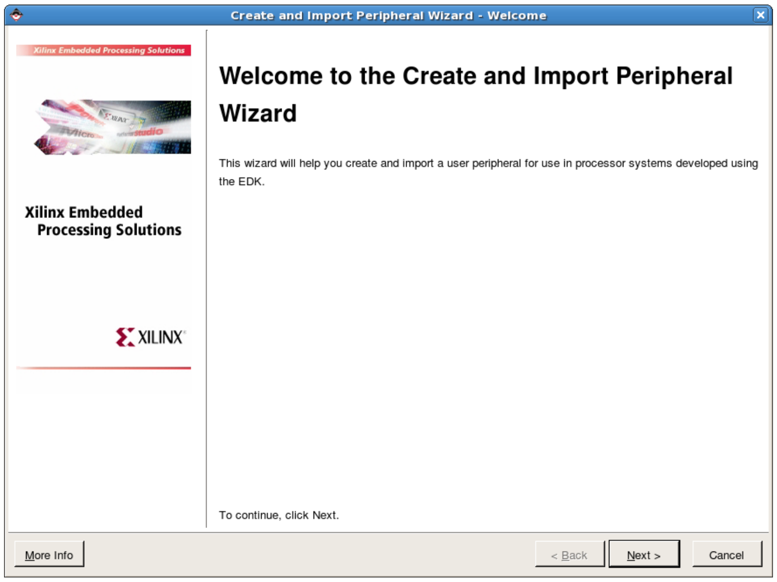
   (b) Use the Base System Builder (BSB) to create a system similar to that which you built in Lab 2. Ensure a UART is the only peripheral your system contains (i.e. no Ethernet MAC, PCIExpress, etc.).

   (c) After the BSB completes, navigate to the ‘System Assembly View’ in XPS. Then click on the ‘Addresses’ tab. Change the size field on both the dlmb_cntrl and the ilmb_cntrl to 128K. Hit ‘Generate Addresses’ to regenerate the address ranges of the other components on the bus.

2. At this point, we should have everything in Figure 1 except for the multiplication peripheral. The next

few steps will guide you through the process of creating a custom IP peripheral. For today's lab, our custom peripheral will have three software accessible 32-bit registers and a hardware multiplication block. Two of the software registers will be read and write accessible by the microprocessor and will hold the multiplicand and multiplier. The third register will be read accessible by the microprocessor and write accessible by the multiplication logic within the peripheral. The third register will hold the product of the multiplication.

(a) From the XPS main menu, select **Hardware→Create or Import Peripheral**. The 'Create and Import Peripheral Wizard' should come up (Figure 2). Hit 'Next' to continue.


Create and Import Peripheral Wizard - Welcome
Xilinx Embedded Processing Solutions
XILINX
More Info
< Back
Next >
Cancel

Welcome to the Create and Import Peripheral Wizard

This wizard will help you create and import a user peripheral for use in processor systems developed using the EDK.

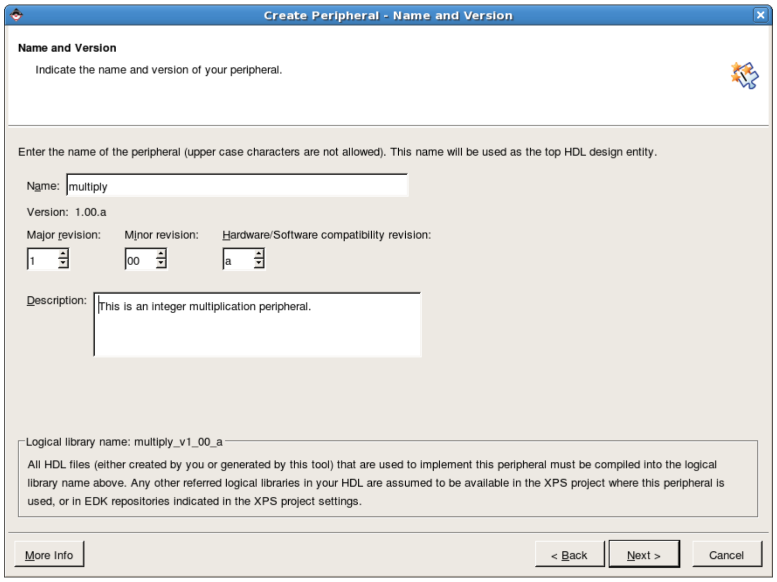
To continue, click Next.

Figure 2: Create and Import Peripheral Wizard - Welcome

(b) The next window allows us to either create templates for a new peripheral or import an existing peripheral. Ensure 'Create templates for a new peripheral' is selected and leave 'Load an existing .cip settings file' unchecked. Press 'Next' to continue.

(c) The window that follows gives us the option to store our peripheral design files in a location other than our current XPS project directory. Leave everything as default and hit 'Next' to continue.

(d) A window will appear prompting you to assign a name and version number to your peripheral (Figure 3). Name the peripheral 'multiply' and leave the version number as default (i.e. 1.00.a). You can enter a short description of your peripheral if you would like in the 'Description' window.



Name and Version

Indicate the name and version of your peripheral.

Enter the name of the peripheral (upper case characters are not allowed). This name will be used as the top HDL design entity.

Name: multiply

Version: 1.00.a

Major revision: 1

Minor revision: 00

Hardware/Software compatibility revision: a

Description: This is an integer multiplication peripheral.

Logical library name: multiply_v1_00_a

All HDL files (either created by you or generated by this tool) that are used to implement this peripheral must be compiled into the logical library name above. Any other referred logical libraries in your HDL are assumed to be available in the XPS project where this peripheral is used, or in EDK repositories indicated in the XPS project settings.

Figure 3: Peripheral Name and Version

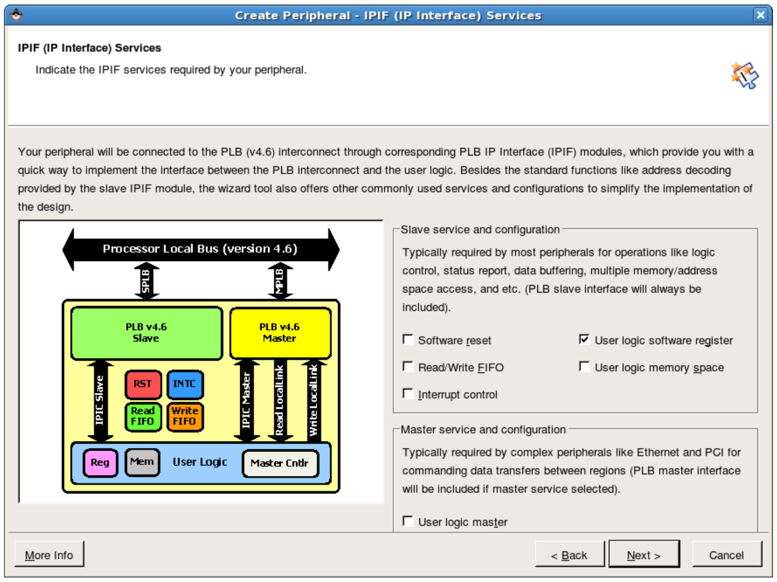
(e) The next window allows us to select which bus we would like our peripheral to connect to. We want to connect to the PLB as the On-chip Peripheral Bus(OPB) has been phased out by Xilinx. Ensure ‘Processor Local Bus (PBL v4.6)’ is selected and ‘Enable OPB and PLB v3.4 bus interfaces’ is unchecked. Press ‘Next’ to move forward.

(f) Figure 4 shows the next prompt. On the left, Xilinx provides us with a block diagram of our peripheral and the various functionality it can have. The right hand side of the window allows us to choose the services or functionality for our peripheral. Our peripheral is very simple and will only have software accessible registers. Thus, ensure only ‘User logic software registers’ is selected and press ‘Next’ to proceed.

(g) For the next window, leave everything as default. The ‘Burst and cache-line support’ option is for high performance data transfer, which is unnecessary for our current application.

(h) In the window that follows, set the ‘Number of software accessible registers’ field to 3. This will cause XPS to autogenerate the portion of our logic required to interface with the bus. Included in that logic is the instantiation the three registers along with the sequential logic for reading and writing to those registers from the PLB. Hit ‘Next’ to move to the ‘IP Interconnect (IPIC)’ portion of the wizard.

(i) Examine the IPIC window. It provides us with a listing of the signals from the PLB, which will be connected to our peripheral. It also allows us to trim out any extraneous signals from



IPIF (IP Interface) Services

Indicate the IPIF services required by your peripheral.

Your peripheral will be connected to the PLB (v4.6) interconnect through corresponding PLB IP Interface (IPIF) modules, which provide you with a quick way to implement the interface between the PLB interconnect and the user logic. Besides the standard functions like address decoding provided by the slave IPIF module, the wizard tool also offers other commonly used services and configurations to simplify the implementation of the design.

Slave service and configuration

Typically required by most peripherals for operations like logic control, status report, data buffering, multiple memory/address space access, and etc. (PLB slave interface will always be included).

- [ ] Software reset
- [x] User logic software register
- [ ] Read/Write FIFO
- [ ] User logic memory space
- [ ] Interrupt control

Master service and configuration

Typically required by complex peripherals like Ethernet and PCI for commanding data transfers between regions (PLB master interface will be included if master service selected).

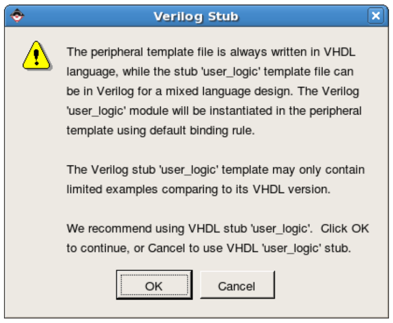
- [ ] User logic master

Figure 4: Peripheral configuration

our logic. Until you have a better idea of what is going on, leave everything as default in this window.

(j) The subsequent window provides us with the option of generating a Bus Functional Model (BFM). For more complicated peripherals, BFMs are quite handy, allowing one to simulate transactions on the PLB without simulating a microprocessor. For our simple design, a BFM is not necessary, and we do not have the required components installed anyway. Leave the ‘Generate BFM simulation platform for ModelSim-SE or ModelSim-PE’ unchecked and press ‘Next’ to continue.

(k) In the ‘Peripheral Implementation Support’ window that follows, select all three check boxes. The first option causes XPS to autogenerate portions of peripheral logic in Verilog, while the second option causes XPS to create project directories and files for designing the remaining user logic in ISE. The last option instructs XPS to autogenerate template driver files. Although we will not be using the autognerated driver template files, they are always nice to look at as a guideline for creating our own softare (hint: find these files and look at them for the last part of lab).
Note: When you select the first of the aforementioned options, a window will pop up (Figure 5) warning us about using Verilog for our user logic. Be sure to read this warning think about its implications. In this case, hit ’OK’ to accept the warning and continue.



The peripheral template file is always written in VHDL language, while the stub 'user_logic' template file can be in Verilog for a mixed language design. The Verilog 'user_logic' module will be instantiated in the peripheral template using default binding rule.

The Verilog stub 'user_logic' template may only contain limited examples comparing to its VHDL version.

We recommend using VHDL stub 'user_logic'. Click OK to continue, or Cancel to use VHDL 'user_logic' stub.

Figure 5: Verilog Stub Warning

(l) The final window provides you with a summary of what files will be generated. Read this window CAREFULLY and then click 'Finish'. XPS will then generate the peripheral templates specified by the previous steps.

(m) While still inside user_logic.v, comment out any code that assigns a value to slv_reg2 (i.e.' slv_reg2 <='). This will deactivate the PLB write capabilities to the last software register. Remember that we cannot have two drivers for a particular register unless we add multiplexing logic.

(n) Under the peripheral hardware templates directory provided in the last screen, you will find the autogenerated files. Navigate to this directory and look through the various subdirectories. Located and open the user_logic.v file. Under the "//USER logic implementation added here" line, insert the following code:

```
reg [0 : C_SLV_DWIDTH-1]   tmp_reg;
always @(posedge Bus2IP_Clk )
  begin
    if(Bus2IP_Reset == 1 )
      begin
        slv_reg2  <=0;
        tmp_reg   <=0;
      end
    else
      begin
        tmp_reg <= slv_reg0*slv_reg1;
        slv_reg2 <= tmp_reg;
      end
  end
```

Examine the above code and comment it appropriately.

3. At this point in the lab, we have used XPS to autogenerate template hardware peripheral files for our multiplication peripheral based on our specifications. We have also added some user logic Verilog to our template for the multiplication functionality of our peripheral. We are now ready to import our peripheral into XPS and add it to our MicroBlaze system. The following steps will guide you through this process

   (a) Select **Hardware→Create or Import Peripheral** from the main menu of the XPS window.

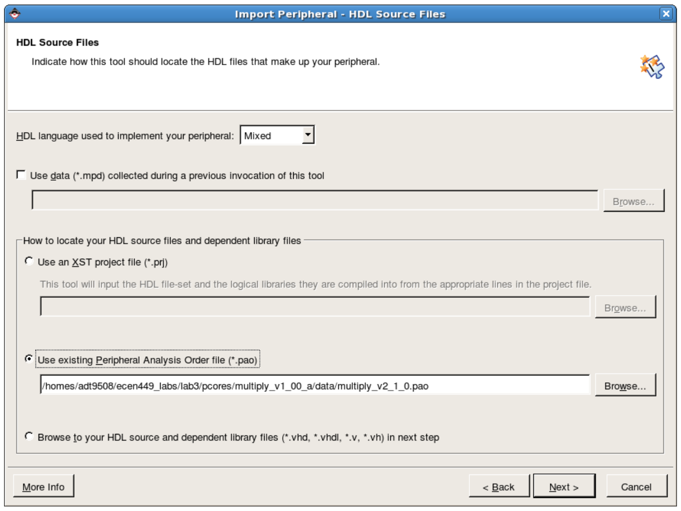
   (b) Hit ‘Next’ in the ‘Welcome’ window of the peripheral wizard to advance to the next screen. In the next screen, select ‘Import existing peripheral’ and click ‘Next’.

   (c) Assuming you followed directions, you should be able to leave the peripheral directory in the next window as default. Hit ‘Next’ to continue.

   (d) In the ‘Name and Version’ screen, type the name of your peripheral exactly as you did previously (i.e. ‘multiply’) and check ‘Use Version: 1.00.a’. Hit ‘Next’ to proceed.

   (e) Select ‘HDL source files’ in the subsequent window and click ‘Next’. There are many ways to describe logic. We used Verilog so we must tell XPS to look for HDL files.

   (f) For the ‘HDL Source Files’ window, set the ‘HDL language used to implement your peripheral’ to ‘Mixed’. This is needed because the default language in Xilinx is VHDL so the bus logic is generated in VHDL and our user logic is in Verilog. In addition, select ‘Use existing Peripheral Analysis Order file (*.pao)’ and click ‘Browse’ to navigate to the *.pao file under the ‘data’ directory within the ‘pcores/multiply_v1_00_a’ directory (Figure 6). Click ‘Next’ to continue.


Import Peripheral - HDL Source Files

HDL Source Files

Indicate how this tool should locate the HDL files that make up your peripheral.

HDL language used to implement your peripheral: Mixed

☐ Use data (*.mpd) collected during a previous invocation of this tool

How to locate your HDL source files and dependent library files

○ Use an XST project file (*.prj)

This tool will input the HDL file-set and the logical libraries they are compiled into from the appropriate lines in the project file.

● Use existing Peripheral Analysis Order file (*.pao)

/homes/adt9508/ecen449_labs/lab3/pcores/multiply_v1_00_a/data/multiply_v2_1_0.pao

○ Browse to your HDL source and dependent library files (*.vhd, *.vhdl, *.v, *.vh) in next step

Figure 6: HDL Source Files

(g) The next window provides a list of the files that will be imported. Click ‘Next’.

(h) At this time the ‘Bus Interfaces’ window should appear. Under the ‘Processor Local Bus (version 4.6) interface’, select ‘PLBV46 Slave’. Ensure all other boxes are unchecked. This tells XPS that our peripheral is to act as a slave device on the bus. For our system, only the MicroBlaze is bus master. Hit ‘Next’ to continue.

(i) The next few windows should all remain as default. Please examine the information each window provides. Hit ‘Next’ until you see the ‘Finish’ window. Hit ‘Finish’ to import the peripheral.

4. We now have successfully created a custom peripheral for our MicroBlaze system. It is now time to add it to our existing system. The next few steps will outline how to do this. Please note that the steps to add our custom IP block are similar to those used to add peripherals supplied by Xilinx.

   (a) In the XPS window, click on the ‘IP Catalog’ tab in the upper left corner. Expand the ‘Project Local pcores’ category in the IP catalog. Your ‘multiply’ peripheral should show up under ‘USER’. Right click on it and select ‘Add IP’. See Figure 7)

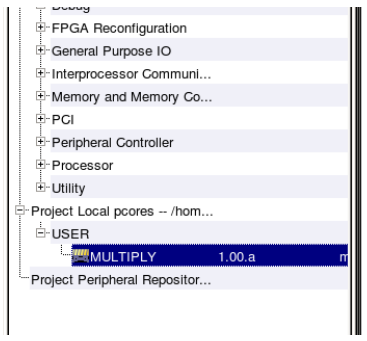


Figure 7: Add IP

   (b) As you did in the previous lab, connect the ‘multiply’ peripheral to the PLB in the ‘System Assembly’ window. Additionally, specify a 64K address size and regenerate system addresses. Please note that we do not have to modify the UCF or make ports external for our peripheral because it has not external signaling.

   (c) Generate the netlist and software libraries for your system as outlined in the last lab.

5. Currently, our hardware should be ready to go. Add a source file in the ‘Applications’ tab by following the steps from Lab 2. The source file should write values to the registers ‘slv_reg0’ and ‘slv_reg1’ and read the multiplication result from ‘slv_reg2’. The value stored in each of these three registers should be printed to the terminal using printf. Demonstrate this to the TA upon completion by using kermit. The following Hints will help:

- You will need to include the xparameters.h and multiply.h, which are located in the 'microblaze_0/include' subdirectory of your lab 3 directory.
- Look for functions to read and write to a register in multiply.h
- The *RegOffset* value for 'slv_reg0' is 0, and the three 32-bit registers are consecutively located in memory.
- Use a for-loop to write different values (varying from 0 to 16) in 'slv_reg0' and 'slv_reg1' and read the corresponding multiplication result from 'slv_reg2'.

Deliverables

1. [4 points.] Demo the working multiplier to the TA.

   Submit a lab report with the following items:

2. [5 points.] Correct format including an Introduction, Procedure, Results, and Conclusion.
3. [4 points.] Commented C file.
4. [2 points.] Commented Verilog code.
5. [2 points.] The output of the terminal (kermit).
6. [3 points.] Answers to the following questions:
   (a) What is the purpose of the tmp_reg from the Verilog code provided in lab, and what happens if this register is removed from the code?
   (b) What values of 'slv_reg0' and 'slv_reg1' would produce incorrect results from the multiplication block? What is the name commonly assigned to this type of computation error, and how would you correct this? Provide a Verilog example and explain what you would change during the creation of the corrected peripheral?