Lab Report 3
Creating a Custom Hardware IP

Christopher Padilla

ECEN 449 - 504

Purpose/Introduction

- This lab is intended to guide students through the process of creating a custom hardware IP and interfacing it with software using the Standard Development Kit. We will use Vivado to build a custom peripheral for multiplying two integers and outputting a result; this will all be done using a hard processor (ARM Processor in Zynq Chip on Zybo Board) instead of the soft processor we used last lab.

Procedure

- The ARM Cortex A9 processor in the Zynq Chip is divided into Processing System and Programming Logic. We will use the UART port on the board to display what the FPGA is doing on the computer screen. I do this by creating a block design diagram but adding the ZYNQ7 PS and importing the XPS Settings instead of the MicroBlaze we used last lab.
- Commenting out the slv_reg2 assignments makes it so we deactivate the write capabilities of the third software register (out output). In other words, you shouldn't be assigning the same thing in two always@ blocks. I added the user logic that tells the program how we're going to multiply. First, we store the result in a temp register and then we store it in our output register (slv_reg2). I packaged everything and generated the bitstream with no errors, and all that was left was to develop the C code. The code in the header files multiply.h and xparameters.h was easy to decipher (i.e. there is a base register and a byte offset, depending on which register you want to write/read).

Results

- The FPGA does all the multiplication, but it will not display the result on the LEDS or anything. The board is doing the multiplication, but I developed the software so it knows what it should be multiplying. The peripheral is determined to be fully functional when we see the correct results on the screen. I first tested it using the two input register values 3 and 3. The output register value was indeed 9; therefore, our ARM processor is indeed doing the multiplication as we instructed.
- For the last part of testing, I developed the software to square all numbers from 0 to 16 and display their results. This was done using a for loop in C. The squares of the numbers were correct.

Conclusion

- When I got to the for-loop part of the lab, all my results were as expected. The only part I had to do alone was to develop the C code. I had to take many precautions during this lab; this included making sure the Verilog code was correct and all slv_reg2 assignments were commented out (except for the user logic part). This lab only took a couple hours to complete; there was a small issue for me to get the results to print out on the Linux Shell; this was because I didn't enable the UART print function before I started the terminal. Since the last lab was so Xilinx SDK intensive, doing this lab was much easier now that I was somewhat familiar with the kit.

Include C and Verilog Code

- My C and Verilog code are stapled to this report.

Questions

- While when testing it without tmp_reg I myself could not notice any results, I would imagine that since it only has one clock cycle to perform all of the operations, we must use tmp_reg instead of doing it all immediately on one line. Again, I did not notice a difference, but I have a feeling it may cause errors if we remove tmp_reg.
- If I were to try to multiply two values, and the result of the multiplication is **more than 32 bits wide**, this would cause an **overflow error**. All three registers only hold 32 bits, so it is bad news if our product is more than 32 bits wide. Changing the size of the registers as needed would correct this problem. During the creation of this peripheral, I would increase the size of the 4 registers we created (we only used 3, I know) to be 64 bits (or higher if the user needs). In the Verilog code we need slv_reg0, slv_reg1, and slv_reg2 to be configured to the same length in bits as our peripheral.
  If we change them to 64 bits, for example:

```
[63:0] slv_reg0;
[63:0] slv_reg1;
[63:0] slv_reg2;
```