Lab Report 2
Using the Software Development Kit

Christopher Padilla

ECEN 449-504

Purpose/Introduction

- This lab is intended to guide students through the process of creating a MicroBlaze processor system using the Vivado Block Design Builder. It is similar to the first lab in that we are making the LEDs perform some functionality based on the inputs from the buttons and switches. One of the distinguishing features of this lab, however, is that we are developing using a pure software implementation in conjunction with the C programming language.

Procedure

- This lab was divided into two parts. The first part of the manual demonstrated how set up the MicroBlaze processor and instructed how to write and import C code so the hardware can function property. The instructions for the first part as to how to set up a GPIO block and set up the peripherals and load it onto the hardware was self-explanatory in the lab manual. It really helped see the differences between designing and exporting something in C using a pure hardware method vs. designing it in Verilog.
- The second part instructed us to have the FPGA perform different functions, such as incrementing and decrementing using two buttons, displaying the current value, and displaying the status of the switches on the board. Setting up the MicroBlaze processor was similar to the first part, except I needed to add an 8-input GPIO block for buttons and switches on the board in addition to the 4-output one already present for the LEDs in part one.

Results

- The main part of this lab, especially for the second part, was developing good, working C code. Mapping the inputs and outputs correctly was also a crucial part in ensuring I'd end up with the correct result. There was not much information online about the Xilinx Standard Development Kit, but I found their documentation and ended up using it to help myself understand the code provided in the first part of the lab.
- The second part of the lab was quite difficult to implement for me, since this was my first time using the SDK. I also had to quickly familiarize myself with C and all of the obscure operations (such as bitwise AND) that were necessary to complete the second part. I developed this code in chunks, and I tested it as such, rather than trying to implement the whole thing only to find out it is all wrong. I have provided a screen capture of the console showing inputs (buttons and switches) and outputs (LEDs) changing in values. After correction of all the little errors, I was able to deliver a fully-working design and feel more confident using MicroBlaze moving forward.

Conclusion

-
- All my results were as expected in the very end after heavy debugging, and I was amazed when everything finally worked. This included making sure inputs were recorded upon changes immediately, as well as keeping track of the different count variables and only displaying them when need be. After spending 12 hours in the lab, I do feel quite confident that I would be able to implement a similar MicroBlaze design using C. There were a few small errors in my code that made it hard to debug the program. Overall, most of the difficulty I had was just understanding the Xilinx SDK C

code. Now that I feel more confident with the syntax and using bitwise arithmetic I am glad I had this learning experience.

Include C Code

- My C code is stapled to this report.

Questions

- The same delay I created in the first part of the lab I also used in the second part of the lab as a counter. The clock rate we used with the FPGA in lab one was 125 MHz; therefore, we had to divide the clock rate by 125 million in order to bring it down to approximately one pulse per second (1 Hz). For this lab the clock rate was smaller to begin with, took more than 1 cycle each, since it actually takes processing time, which is why we only felt necessary to divide it by 10 million (which gave about 1 Hz). Assuming a clock rate of 100 MHz for this lab, each while look takes 6 cycles (since we divided by 10 million).
- The count variable can change at any time unexpectedly. We declare it as volatile to prevent the C compiler from trying to perform optimizations; this means the compiler can't change the order of which data commands are written. The only reason you would ever use this keyword is when dealing with hardware implementations (which we are).
- while(1) is another way of saying while (true), meaning that the "condition" in question will ALWAYS be true, therefore creating an infinite loop, until the user chooses to terminate the program manually. This while loop makes it so we can check for changes in inputs almost instantaneously (as far as humans are concerned). This was really the only way to develop this program where we can continuously check for changes without missing out on much.
- By far the Verilog implementation was easier for me. The main advantage of using Verilog for this type of lab instead of C because Verilog and other HDLs more easily support parallel programming for these kinds of tasks. With C you can only perform one task at a time, therefore making it more difficult to catch changes. However, an advantage to a software build using C and other standard programming languages, anyone (such as Xilinx) can make their own SDK and make some many things in C more possible using pure software implementation. Lab two felt more like a program, rather than a bunch of wires and registers such as in the first lab.