Jonathan Whitaker
U0752100

Final Project Report

I chose to build the 4-bit processor for my final project. I had already taken CS 3810 (Computer Organization), so I felt like I had a very thorough understanding of what needed to be done to get the processor working. Having that class helped me work through the project relatively smoothly. Going into the project, I already understood how the basic controller needed to operate. The challenge was to ensure that the FSM controller was operating how it was supposed to.

Import Design Decisions & Features

In my FSM controller, one of my design decisions was to distinguish between two modes of operation, *user mode* and *run mode*. My design utilizes a one-hot encoding for each of these different modes. By using a one-hot encoding, the user has the ability to select one button to indicate manual instruction entry, or the user could select the other button and the processor will execute a set of instructions loaded from ROM. I encoded the user mode operation as a binary `01` and the run mode operation as a binary `10`. This way, when the button B8 is pressed (see Figure 1) the processor will operate on an instruction manually entered by the user. When the button C4 is pressed, the processor will carry out a set of instructions loaded into ROM. Button B8 has the highest precedence, so if the user were to select both buttons at the exact same time, the processor would operate in user mode. I made the FSM controller behave in this way so as to reduce the possibility of accidentally overwriting the contents of a register.

Figure 1: Operation select buttons.

User mode

When the user selects the user mode operation, the processor decodes and executes the instruction input from the 8 dip switches (see Figure 2) on the Nexys 3 board. The processor was designed to execute 4 different types of instructions: load, store, move, and arithmetic/logic instructions. As a result of this, each instruction is uniquely recognized with its two bit *opcode* (operation code). An instruction's opcode is determined via the top two bits of the instruction encoding, so the user can select the desired operation with the upper 2 dip switches (button T5 and V8 below). Each operation has a unique encoding. For a load instruction, an immediate value is loaded into a destination register. The destination register (R_x) is selected via

Operation	Function performed	Instruction Encoding
Load $Rx, Data$	$Rx \leftarrow Data$	00,Rx,Data
Store $Rx, Data$	LEDs $\leftarrow [Rx]$	01,Rx,XXXX
Move Rx, Ry	$Rx \leftarrow [Ry]$	10,Rx,Ry,XX
Add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$	11,Rx,Ry,00
Sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$	11,Rx,Ry,01
And Rx, Ry	$Rx \leftarrow [Rx]$ & $[Ry]$	11,Rx,Ry,10
Not Rx, Ry	$Rx \leftarrow ![Rx]$	11,Rx,XX,11

Table 1: Instruction set architecture for the 4-bit processor.

buttons U8 and N8, and the value loaded into R_x is encoded with the last 4 switches on the board (T10, T9, V9, M8). For a store instruction, the contents of the register R_x are output to the 4 rightmost LEDs (circled in Figure 2). The lower 4 dip switches are ignored for a store instruction. Move instructions move the contents of a register R_y, encoded with buttons M8 and V9, into the destination register R_x. The lower 2 switches are ignored with a move instruction. Finally, arithmetic/logic instructions store the result of an operation between the two operands R_x and R_y back into the destination register R_x. The type of arithmetic/logic operation is selected via the last 2 switches on the board (T9, T10). This instruction set architecture is summarized in Table 1 above.

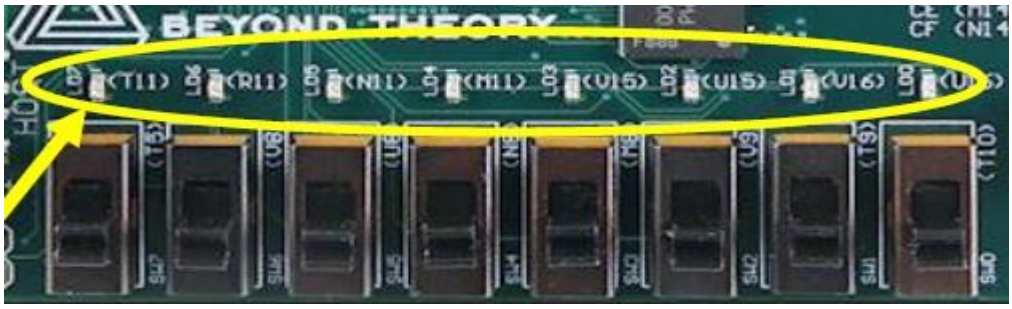

Figure 2: Switches used to encode an instruction.

Run mode

When the user selects the run mode operation, the processor reads in instructions from a ROM which has been populated with a set of instructions. Instructions are decoded in an identical fashion as was mentioned above in the user mode. When a given instruction has finished, the controller asserts a signal (*inst_done*), which increments an internal program counter in the ROM module. Each instruction is output from the ROM based on the value of the program counter. As the program counter increments, the set of instructions is traversed. After all of the instructions have been executed, the ROM module asserts a signal (*rom_done*) which indicates back to the controller that the processor has executed all of the instruction in the ROM.

This distinction between user mode and run mode allowed me to simplify my design significantly. I was able to design the control logic for the user mode and then duplicate the user mode control logic for the run mode control logic. My ASM chart describing my logic is shown on the next page (Figure 3). It is assumed that the user will assert the clear signal when the processor is first initialized. If the clear signal isn't asserted initially, then the CPU controller will automatically move into the idle state anyways. In the idle state, the controller waits to receive the user mode or run mode operation signal. As was described above, if the user mode signal is asserted, then the processor begins to operate on the instruction provided manually by the user. This process will traverse the right path in my ASM chart. If the user selects the run mode, then the left path is traversed. Notice, however, that both the left and right path have very similar behavior. For example, the path leading from the *run_exec* state on the left hand side is nearly identical to the path leading from the *usr_mod*e state on the right hand side. These similarities made the design much easier. I was able to complete the user mode branch and test it, and then duplicate the functionality in the run mode branch. Likewise, designing the control logic for the user mode helped me understand the signal paths and

the control flow better. Understanding the signal paths and control flow made it easier to extend the control logic with the additional ROM module for the run mode.

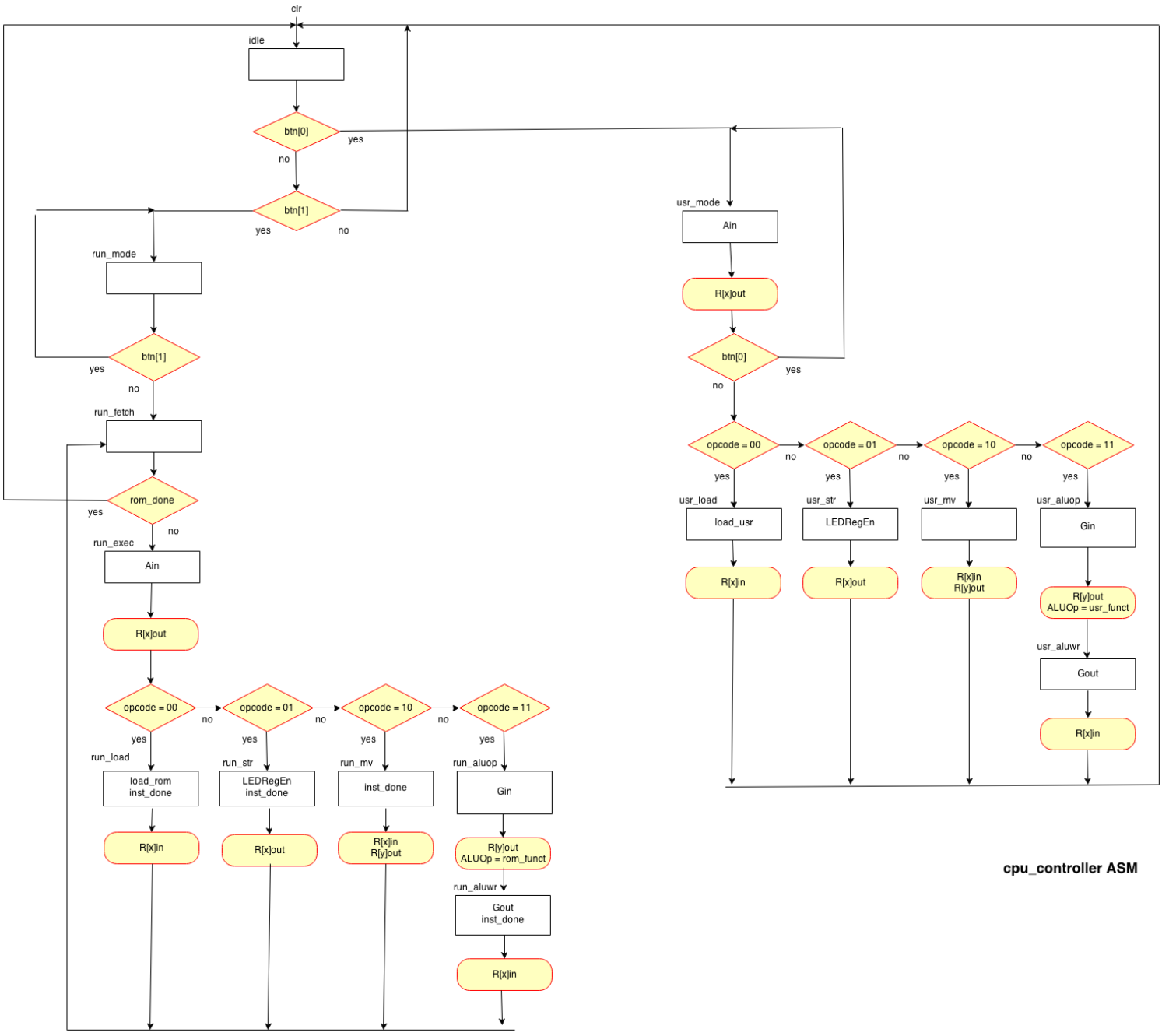

Figure 3: CPU Controller ASM chart.

Optimizations

The optimizations I performed were very negligible. However, one improvement I did perform was that I decoded the various instruction parameters to an appropriate register name. For example, the incoming instruction *usr_inst* is broken down into its individual parts *usr_opcode, usr_Rx, usr_Ry, usr_Data,* and *usr_funct.* I did the same for the incoming run instruction. This variable assignment made it much easier to derive the next state and output logic. Conceptually, it made more sense to leave my design un-optimized. In my opinion, unless you are designing a state of the art system, design clarity is more important than design performance. It was more valuable to me to have the clarity of the solution rather than the performance. Like I mentioned previously, my design was very simple and a lot of it was duplicate code. This made it easier to understand. Once I understood one branch of my ASM design, the other branch was extremely easy to derive.

Test Strategy

In order to test the 4-bit processor, I compartmentalized each module into its own project. This design strategy allowed me to test each individual module before instantiating it in a top level module. I created four subprojects for each of the modules which included: *regn*, *trin*, *cpu_controller*, and *ALU* modules respectively. To test the controller I stimulated each of the possible instruction types, and for each instruction type I manually verified the output signals were asserted correctly. Simulating the controller took the most time because of all of the output signals involved. I also created a self-checking test bench for the ALU. The regn and trin modules implemented generic register and tri-state buffer circuits. I didn't create individual tests for the regn and trin modules primarily because of their trivial implementation. The regn and trin modules were presented in the text. Likewise, these two modules behavior was easily revealed in the top level test. To put it all together, I created one top level test. A sample of this test is depicted below in Figure 4. As you can see from the first instruction (usr_inst = 00000100), the destination register R0 gets the value of 4 just like it should from the instruction format I described earlier. Likewise, destination register R1 gets the value of 3 on the next instruction (usr_inst = 00010011). Figure 4 just shows a clip of two of the instructions, but my top level test was composed of a test for each of the different instruction types for both run mode and user mode.

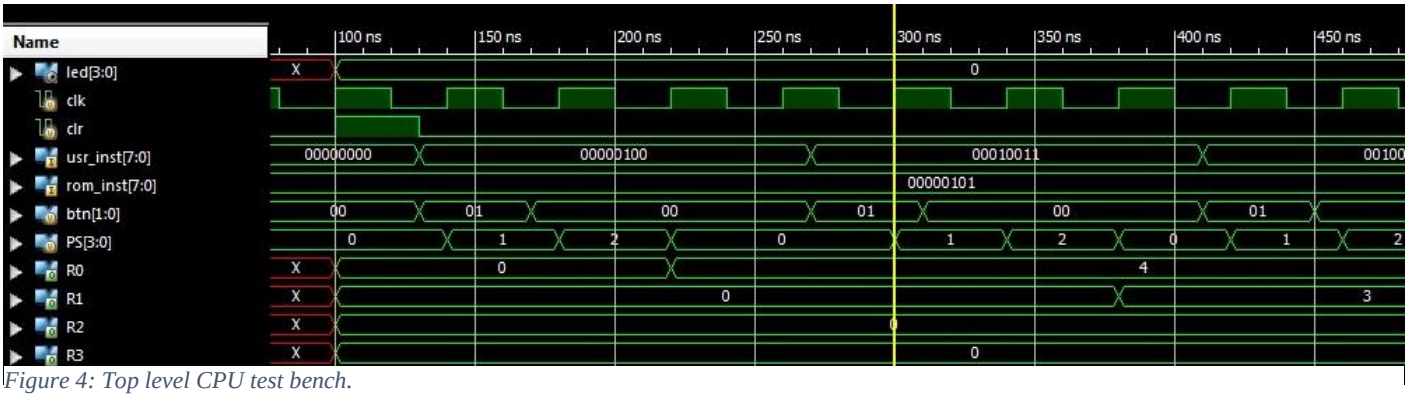

Figure 4: Top level CPU test bench.

Conclusion

Looking back at this project, I am very satisfied with my design approach and test strategy. Everything went very smoothly, and it all came together in a tidy top level module. This project was a great introduction to the hierarchical design strategy. I received great insight as to how things should be designed from the ground up with a high level picture in mind at all times. Compartmentalism is key in these larger projects. As 3710 approaches it is crucial that I learn how to exploit compartmentalism and abstraction, and this project gave me really good practice with both of these. Likewise, because of my design approach, I learned a great deal about design clarity versus the performance of the design. For me, the clarity of my solution outweighed the potential performance benefits I may have gained by reducing the logic. I wouldn't change my approach if I were to go back. Overall, I am very pleased with the experience that I gained with this project.