Lab IV: CPU Datapath

ECE 3710
Stephan Stankovic, Jon Pilling, Jeremy Wu, Steen Sia
University of Utah

Abstract - This report discusses and shows the process of integrating CPU control logic with the CPU Datapath. This report also talks about the design and synthesis of the FSM control module to control signals all over the CPU. The objective was to wire the ALU, Regfile, and Memory all together to work synchronously in the CPU design. Additionally, we describe which peripheral(s) are needed moving forward to the final project. Then, we discuss any problems that were encountered during the integration of the CPU and how the group resolved these issues one step at a time. Ultimately, code design and block diagrams are shown to interpret how the CPU Datapath was integrated.

I. INTRODUCTION

The first approach to the CPU datapath was to visualize how the CPU was connected to everything else with a drawing. Using the professor's drawing below, we broke down which MUXs we needed to include and think about writing a FSM that handles all types of instructions, specifically loads, stores, and jump instructions. The ALU was already designed to handle all kinds of arithmetic operations.

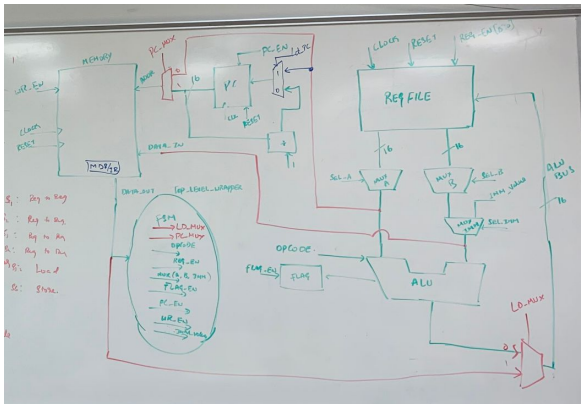

Figure 1. Block Diagram of CPU Datapath

Before going into the wiring and instantiation of the datapath, we will discuss how the FSM utilizes the output from memory, as its input and how it functions in every state.

II. DESIGN/RESULTS

In our FSM, we have chosen to implement the following states which acts as our decoder: r1, r2, store1, store2, load1, load2, jump1, jump2. We implemented a stop state for putting a value on the board and holding that state, so the value didn't change. Starting with our combinational logic with r1 and r2, these states deal with the R-type instructions that were already accounted for earlier when we designed the ALU. We simply extracted [15:12] and [7:4] bits from the data output as the opcode to be fed into the ALU and performed the necessary calculation.

Next, we go into our load and store instructions which we take Raddr then transfer that into Rdest,and take Rsrc then transfer into Rdaddr respectively. In these states, we initialize all the inputs that aren't used to 0, and activate specific enables that pass in values through the MUXes. For the store1 state, we set PC_MUX to 0, write enable to 1, and PC enable to 0. This means we only want to feed the address coming from mux A into memory and provide access to that data immediately. Next, our store1 goes to store2 which turns PC mux, PC enable to 1 and set wen to 0. This means we are incrementing the program counter and feeding that into memory, but don't provide the output to data out.

Then we have our load1 state, which enables LD mux and basically just feeds the data coming from memory into the regfile. The next state, load2, sets register enable to any of the 16 registers depending on the first four bits of the data coming out of memory, then we enable PC enable, PC mux, and LD mux. This means we are allowing PC to go into the memory while regfile is fed input from the LD mux.

Lastly we have jump1 state, which enables LD mux, PC enable, PC mux, then moving on to jump2, we disable PC_en and LD_mux. These two states accomplish the following: recognize that we are jumping to a new address, then on the next clock cycle, remain in that new address and treat it as the base address and normally increment as instructions come. The default state just initializes everything to 0. The stop state essentially ensures that once values have been written into memory and registers, we stop any other states from occuring, so that we are able to display the written value to the hex-to-7

segment display

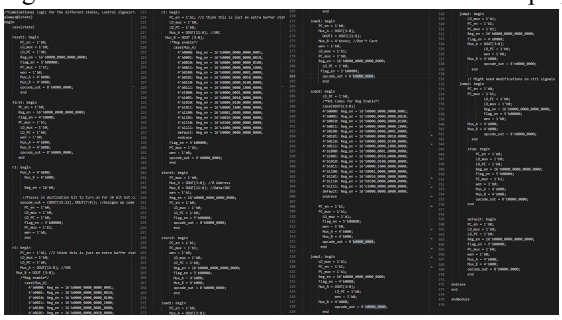

Figure 2. Combinational Logic of FSM

III. TESTING

Before going to the test phase, here is how our logic is for the states mentioned above:

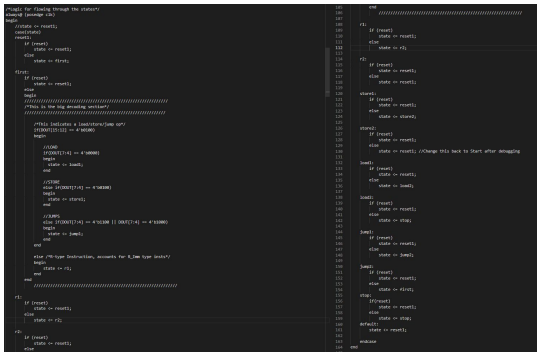

Figure 3. Sequence of FSM

In an always block, we take in the positive edge of the clock and case on the state. If reset is provided, go to reset state where everything is initialized to 0 except PC mux, which just allows input to memory. The first state establishes the decode of whatever instruction is passed in. It first checks the [7:4] bits for a load, store or jump instruction and goes to those states, if they are not any, then we know it is an R-type instruction and simply go to r1 and r2 states.

Having established the FSM and how it flows within the CPU, we created a top level-wrapper encompassing the ALU, regfile, memory and wired them all together just like fig. 1.

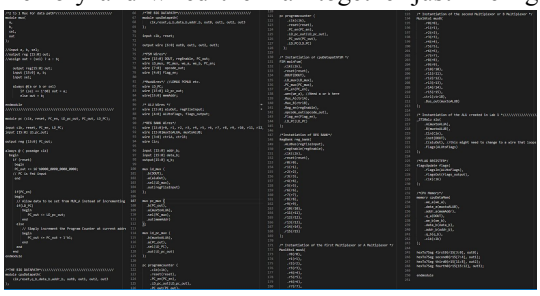

Figure 4. CPU Datapath Instantiation

As a testbench, we simply fed in the clock under the instantiation of our CPU datapath and observed the changing outputs through simulation in modelsim. The test we wrote utilized our assembler which conveniently took in any instruction and its src/dest values and spit out the machine code to be put into our memory text file. These are the sequence of instructions we provided: ADDI 25, R0; ADDI 1, R1; STORE RO, Addr(R1); LOAD R15, Addr(R1). The purpose of these series of instructions is to identify that the R type instructions are working as intended and that we are able to retrieve and store data into all of the valid registers. Moving on to ModelSim and observing the waveforms, we were able to validate that the immediate values 25 and 1 were stored in registers 0 and 1 respectively. Additionally, this verified that our states were working as intended and that the extra states added just for buffer, did not hinder our performance. Then, looking into register 15, we were able to load the value 25 into it from register 1. Then, we decided to output the value written in register 15 with the hex-to-7-segment on the board and successfully displayed 0x19 which is indeed 25 in decimal.

Throughout testing, we encountered several problems which involved memory, instantiation/ wiring issues, and logic of our FSM. Our first issue that came up was with our memory implementation that was a true dual port memory. For our purpose, we use one port for the CPU and leave the other for VGA later on. While trying to synthesize with our memory, it took far too long and did not generate the correct size of memory which should have been 1 GB. We later found out that it continually allocated memory over 16GB. We resolved this issue by using the true-dual port template given by quartus and was able to synthesize in reasonable amount of time with the correct memory size of 1GB.

We had an issue with mux enabling where we believed that the mux control signals sent by the FSM were 16 bit hot-coded values. We figured out that these signals were supposed to be 4 bit values, and this simplified our design.

Finally, the problems we had were wiring our mux controls for A and B. These controls were not being set from the FSM, which left us with don't care conditions assigned as outputs coming out mux A and mux B. This meant that we were not getting input where we expected to get them. We fixed this issue by placing the correct inputs under our instantiation of FSM.

IV. CONCLUSION

To summarize, our group was able to get the CPU datapath working as intended thankfully through our robust testing procedures and careful observation of the waveform to ensure that inputs and outputs were in sync. Each team member helped the building of the CPU datapath and the construction of the assembler was done with the help of Stephan.