Zack Fravel

010646947

Lab 1 : M 4:10 - 5:55 PM

Lab 6 - Single-Cycle CPU

4/4/16

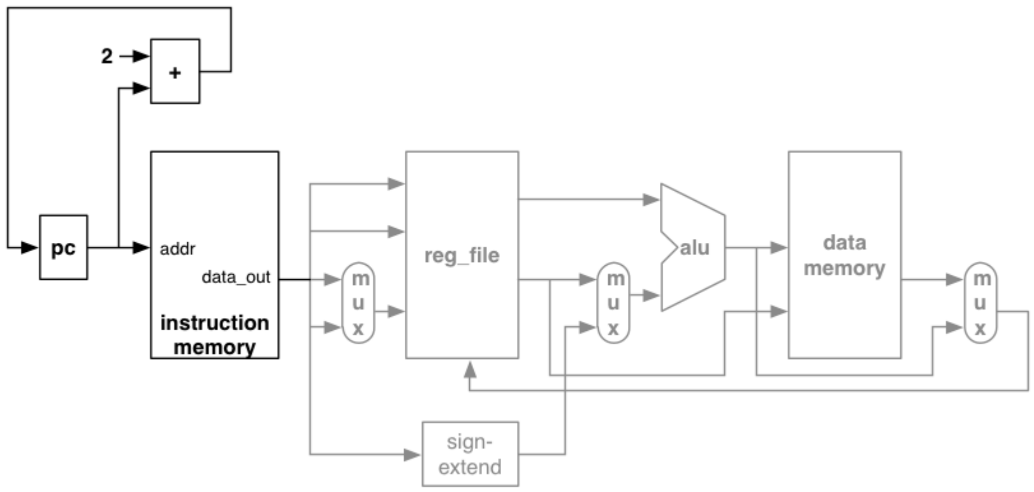
Introduction

The objective for this lab is to complete the task we set out from the beginning and put all the pieces together for our final Single-Cycle CPU. Our CPU by the end of the lab has what it needs to perform any of the instructions listed on the lab ISA for the instruction set we're using. (apdx. Figure 7)

The lab is broken up into four parts to complete what we already have set up. Part one is to add an instruction memory module as well as a program counter to allow the datapath to accept a file full of instructions and run through each instruction over a set period of time. Part two involves adding functionality to the ALU to perform set-on-less-than (SLT) operations. Part three has us add a few new modules to support the last of our two instructions, branch-on-not-equal (BNE), allowing the program counter (PC) to go to another instruction based on a binary operation. Finally, part four has us add the final pieces to the CPU to support the jump function, which allows the PC to go to advance to a specified instruction location.

Approach

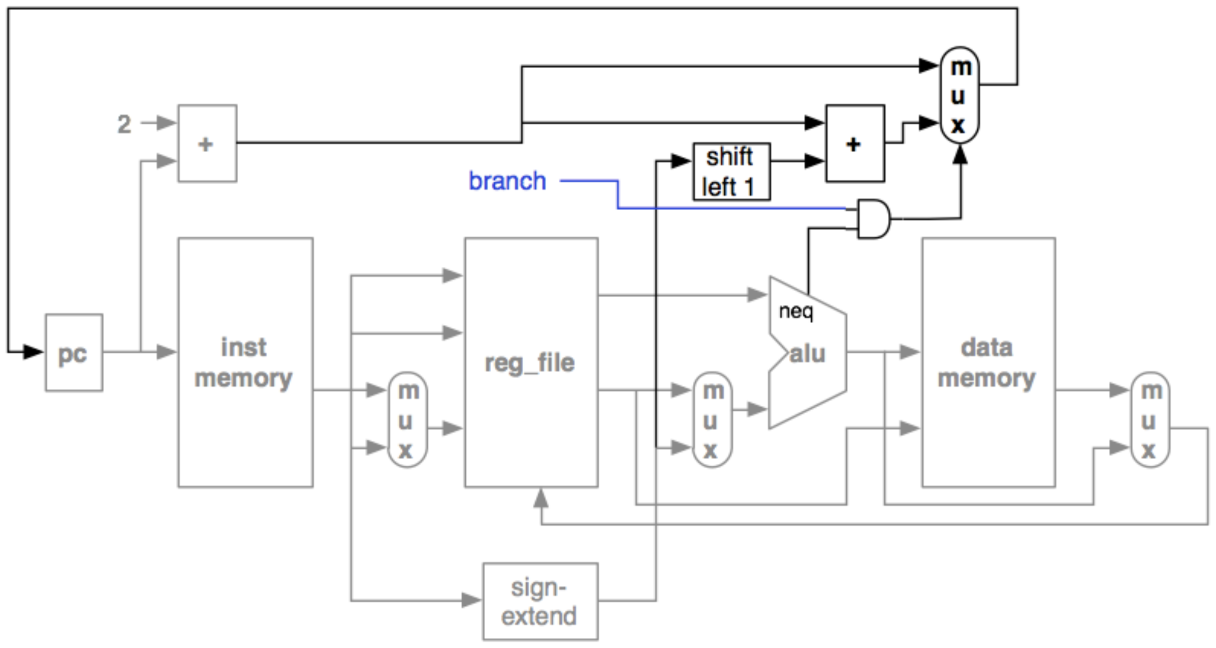
Part one of the lab is fairly simple as far as implementation goes. We're using the exact same memory module we were given for our data memory, so everything is fairly familiar. Below is an illustration of the datapath for part one; a few things are omitted for simplicities sake (i.e. control entity).



Along with the instruction memory module, we want to add a program counter. This comes in the form of a 16-bit register, not to be confused with the 16-bit register-file we use for our 16 registers in the datapath. This register, connected to another entity which adds +2 to its output and feeds it back into its input. What this does is feeds the address +2 after each cycle to the instruction memory, creating a incrementing list of executable instructions. Another question we have when connecting these are the signals for the write_en, read_en, data_in, and mem_dump on the instruction memory. Since the instruction memory is always reading, we want read_en := '1', write_en := '0', and mem_dump := '0.' For data_in, we just put x"0000" as it doesn't really have much of an impact on the functionality. VHDL code for all the updated entities are attached.

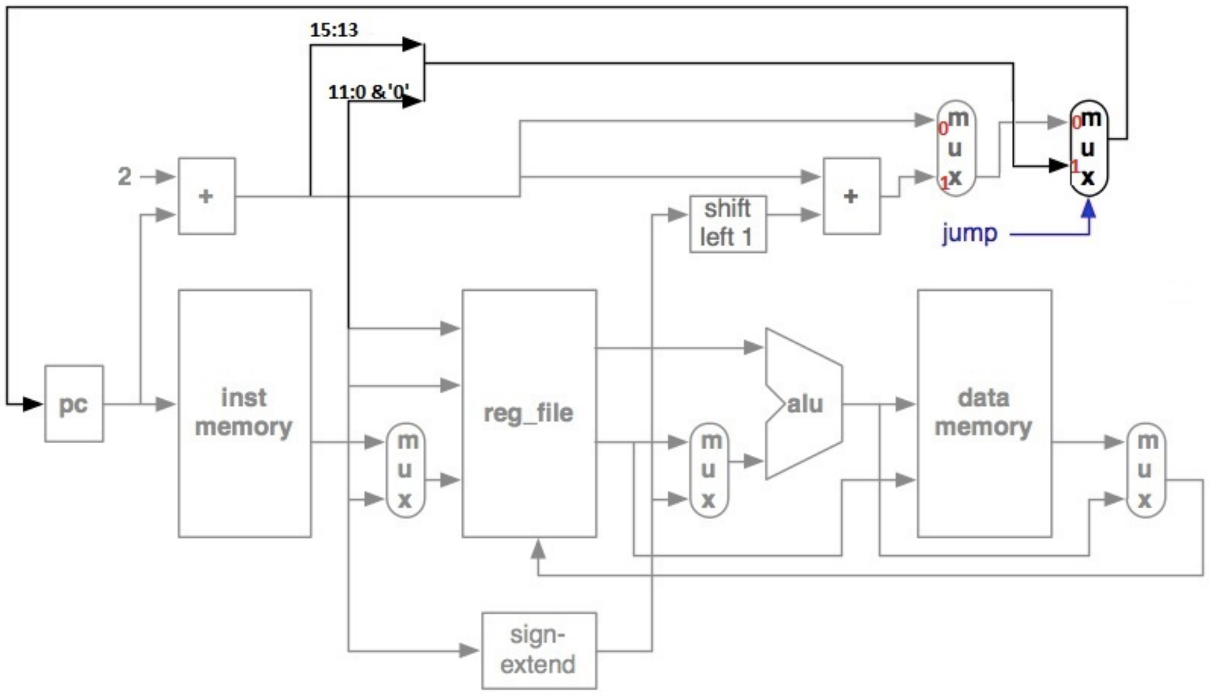
Part two of the lab had us add some functionality to the datapath in the form of the SLT (set-on-less-than) instruction. SLT, opcode := x"7", allows us to compare two registers and set a destination register (Rd) to either x"0000" or x"0001" based on whether Rs < Rt. The way this function is actually implemented in the circuit is through an additional signal added to the ALU implementation, and changing the alu_sel signal from a 2-bit signal to a 3-bit signal since we now have more than four ALU operations. Now that the alu_sel signal is 3-bits, we're able to differentiate more instructions. We already know how to do less than, greater than, and equal to logic within the ALU, so we just use that and assign the new signal, slr_r, to the result of the less-than logic and set that signal equal to the result whenever alu_sel := "100." With these changes to the ALU and making sure to change the aluop control signal in the system entity to 3-bits, the circuit was now able to take in SLT instructions. VHDL for everything discussed is attached.

Part three of the lab had us implement the second to last function on our ISA, branch-on-not-equal (BNE). This step was a little more involved than part two in that we are actually adding more entities to the datapath again. BNE is described in the ISA as follows; with opcode := x"9" if Rs != Rt then the PC := PC + 2 + address, with the address being derived from the 4-bit offset in the instruction. This allows the program to jump to another instruction, rather than the next one, based on a binary operation. The way the datapath is set up and how the logic is formed, this offset actually specifies the number of instructions you want to branch and the datapath converts that into an actual address. Below is the datapath to support BNE instructions.



As can be seen above, to support BNE there is a little more involved than you'd first think. The first step in calculating the branch comes from the immediate value in the last 4-bits, sign extended to 16-bits and shifted left by 1. The reason it shifts is every instruction is 2-byte addresses, so you want to multiply the number of instructions you want to skip by two to get the actual address you're looking for. Once you have this value, it is added to the current PC + 2 and sent through a mux. This mux only puts this value through if the new branch signal specified in the control entity := '1' and the inverted eq logic (neq) on the ALU := '1.' The logic for equal-to already exists in the ALU, all that was needed to do was to created another output on the ALU and set it to that logic, and then invert that in the system entity. The reason these signals are ANDed together is because to specify a branch, you need both branch := ‘1’ and Rs != Rt or neq := ‘1.’ Otherwise, the mux sends through the original PC + 2 signal and the datapath works as usual. All the VHDL for the described logic is attached.

The final step in completing our single-cycle CPU is to add jump instruction functionality. This step also involves adding a couple new pieces to our datapath to support the final instruction. The jump instruction has opcode := “B” and the rest of its 12-bits are used to specify the address for the jump. First of all, a new control signal, “jump” is added for specifying a jump instruction. This jump signal controls a mux that specifies whether or not the PC gets modified by the output of the previous mux, or by the new jump mux. The address is calculating by bit-slicing already existing signals; 15:13 comes from the PC + 2 signal, so we are in the correct address range, and 11:0 from the J instruction are added together along with an appended ‘0’ since all addresses end in ‘0.’ Below is the circuit described above.



As stated before, the jump signal controlling the mux := ‘1’ whenever opcode := “B.” This completes our implementation of the single-cycle CPU; VHDL for everything described is attached.

Experimentation

There were a lot of different errors and problems that came up in the implementation of these new functions. Part one was relatively straight forward in adding the program counter and instruction memory. The instruction memory is the exact same entity as our data memory and we had already designed the 16-bit register for the PC. The experimentation came in with how the input signals needed to be set on the instruction memory as well as how to implement the adder and correctly load in the set of instructions. I began with creating a new adder entity but thought this was overly complicating the signals in my system entity so I just ended up creating a constant 16-bit signal in the system entity := x”0002” and add that to the output of the PC for its input. I was having issues loading in the instruction file with the extension “.txt” but was able to get it to work after changing it to “.mem.” Since part two didn’t require adding any extra entities, this was the easiest step as all it required was adding simple functionality to the ALU. The test bench runs through the same set of instructions as our Lab 4, along with a new set of SLT instructions.

Part three and four required a little more experimentation to get working properly. Implementing the shift_left entity wasn’t too hard to make sure it worked, just needed to load it in the situation and observe its output. The addition for the branch address is also done i the system entity, similar to how the PC addition is handled. The AND logic, as well as the invert logic for the eq signal for the branch mux selector is all also done in the system entity. The mux being used is the same 2-to-1 16-bit mux that is used in two other places in the datapath. Part four was very easy since all it used was another 2-to-1 mux which is controlled by a new stand-alone signal in the control entity and some simple bit-slicing to determine the input of the mux. The tricky part about calculating the address is getting used to how it is formatted in the instruction and targeting the address you want properly.

Results

Part one results use the same exact instruction set as was used in the previous lab. We start with 2 ADDi instructions, 2 SW's, another ADDi, 2 LW's, and to test part two of the lab we add 2 SLT operations.

(apdx. Figure 4)

I have attached the instruction_in file and written the translations of each instruction alongside. This test bench shows that all the previous operations, along with the new SLT operation on the simulation diagram shown in the figure.

(apdx. Figure 1)

With this simulation, at the end of the SLT instructions we expect R[3] := x"0000" and R[9] := x"0001" along with all the previous memory and register results.

Part three had us add the BNE instruction. To properly test this, I kept all the previous instructions from the last test and put a blank instruction to set apart the new ones. After the blank instruction, I added a BNE that checks whether 0 = 1, if not it branches 2 instructions. The next two instructions are blank with the third next instruction being an add. So, we should see in the waveform diagram all the previous instructions, a blank instruction, a branch, and an add.

(apdx. Figure 2 & 5)

Finally, part four of the lab we added the jump instruction. Testing this was a little trickier, but with some experimentation I got it to work as expected. I kept all the previous instructions, as before, and instead of the two blank instruction after the branch I put two instructions that will each add 1 to R[2]. And after the branch add, I have a jump instruction go to those two add instructions.

(apdx. Figure 3 & 6)

This in essence creates a loop after the branch that, after each jump, + 2 is added to R[2] but R[5] always stays the same. This loop goes until the test bench finishes running its specified time. This completes the testing of our single-cycle CPU.

Conclusions

As shown in the results and the VHDL attached to the report, the design works as intended and I expect any other combination of our ISA instruction set would work properly as well. This lab did a really good job at bringing all the pieces together and showing how simple it can be to add on to your design if it is properly modular and has good readability. There are no known problems with my current CPU.