CompArch: Lab 3: CPU

Griffin Tschurwald, Meg McCauley, Brenna Manning

November 23, 2015

1 Processor Diagram and Description

We decided to implement a single cycle CPU similar to the design we went over in class, with some small changes. These changes were mainly to increase the ease of our understanding, but also served as some speed improvements similar to how parts of a multi cycle CPU would be implemented. A diagram of our processor design can be seen in Figure 1 on page 2. One change is that we added a dedicated register for holding jump destinations. We decided to do this because we thought it made the JR and JAL steps easier to understand. There also is a speed increase during the JR step because of this design decision. Instead of getting the address out of memory, and putting it through several components that all take time, we just get it from the RA register and send it back to the program counter. The downside of this method is increased cost due to needing another register, and increased cost in the instruction decoder due to needing another control signal.

We made another design decision similar to this for the LW and LS operations. These require an immediate to be passed to the Addr field of the data memory. Instead of passing that immediate through the ALU source multiplexer and the ALU itself, we now have a control signal that will just let it pass directly through the multiplexer to the address port. The downside of this is increased cost because of the new control signal and extra multiplexer. Instructions are held in instruction memory. These instructions are manually loaded with a .dat file, generated from MIPS assembly code. The instruction goes to the instruction decoder, which generates all of the control signals for that operation.

Our team went through each instruction by hand and logically determined which control signals would be high for each operation. This information was then put into a table, which can be seen here in Table 1 on page 3. We created our instruction decoder using this information. This table shows what each of the control signals will be set to for each of the operations our single cycle CPU performs. The signals are used to determine which of two outputs a multiplexer will output or as read or write enables. They are also used as read enables, the second input to an and gate, or to control ALU operation. They are indicated on Figure 1 on page 2 in blue writing.

Different segments of the instruction are sent to different blocks of the CPU to be used or not used based on what operation the instruction is for. We split up the different segments the same way they are for I-Type, J-Type, and R-Type instructions: Sending Rt to Rt, Rs to Rs, etc. For example, [15:0] of the instruction is always sent to the Sign Extend Immediate block. However, this piece of information is only used when the instruction is an I-Type, where [15:0] is the address, and thusly the control signals generated by the instruction decoder when the instruction is an I-Type, are set to choose to use that signal from the Sign Extend Immediate block.

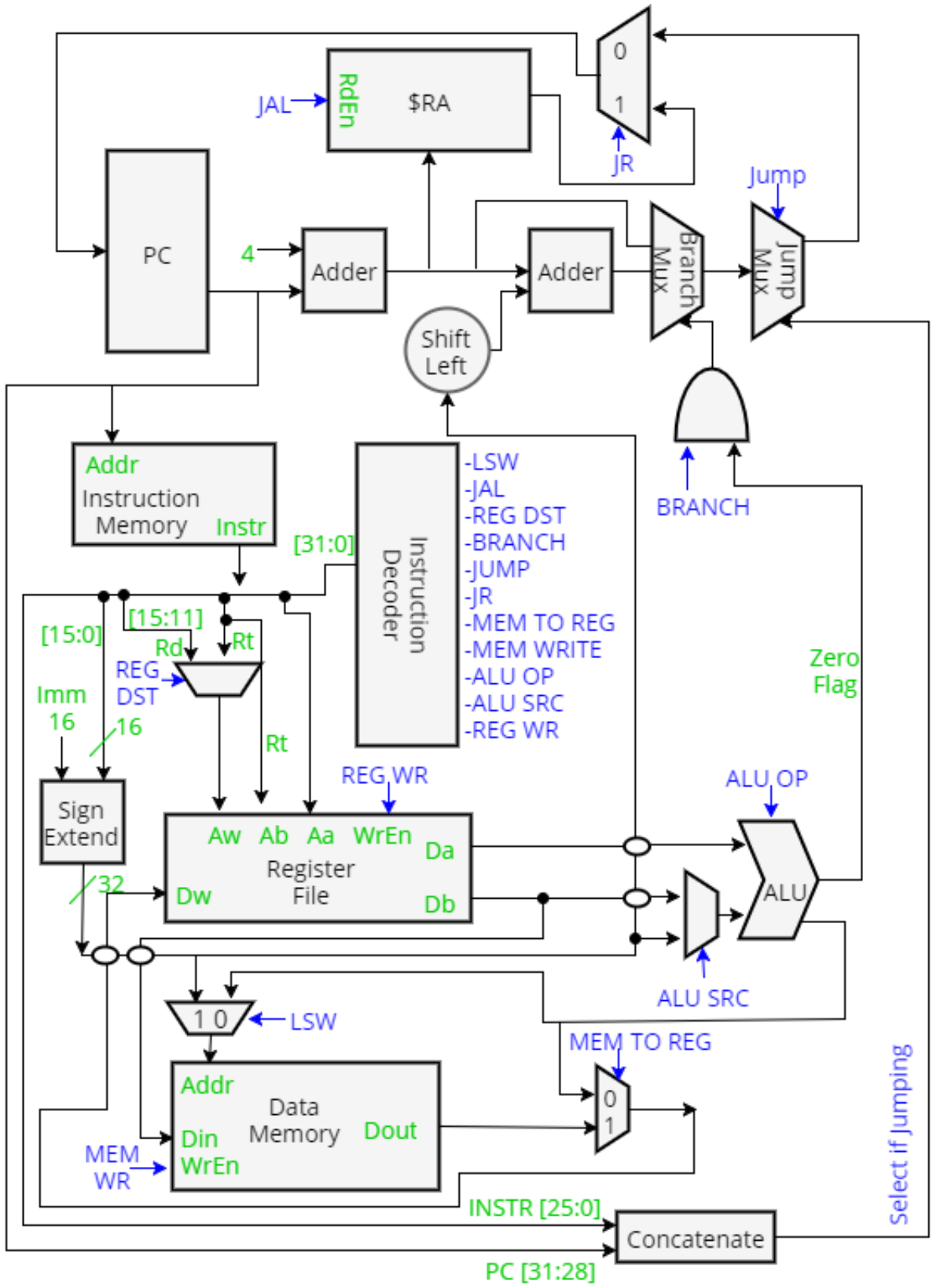

Figure 1: *Our Single Cycle CPU Design. Named inputs are written in green. All control signals come from outputs of the instruction decoder and are written in blue.*

Load Word Control										
RegDst	AluSrc	MemtoRg	MemWr	RegWr	Brnch	Jump	AluCtrl	LSW	JAL	JR
0	0	1	0	1	0	0	add	1	0	0
Store Word Control										
RegDst	AluSrc	MemtoRg	MemWr	RegWr	Brnch	Jump	AluCtrl	LSW	JAL	JR
0	0	0	1	0	0	0	add	0	0	0
Jump Control										
RegDst	AluSrc	MemtoRg	MemWr	RegWr	Brnch	Jump	AluCtrl	LSW	JAL	JR
x	x	x	x	x	0	1	x	0	0	0
Jump Register Control										
RegDst	AluSrc	MemtoRg	MemWr	RegWr	Brnch	Jump	AluCtrl	LSW	JAL	JR
x	x	x	x	x	x	x	x	x	0	1
Jump and Link Control										
RegDst	AluSrc	MemtoRg	MemWr	RegWr	Brnch	Jump	AluCtrl	LSW	JAL	JR
x	x	x	x	x	0	1	x	0	1	0
Branch if Not Equal to Control										
RegDst	AluSrc	MemtoRg	MemWr	RegWr	Brnch	Jump	AluCtrl	LSW	JAL	JR
0	0	0	0	0	1	0	sub	0	0	0
XORI Control										
RegDst	AluSrc	MemtoRg	MemWr	RegWr	Brnch	Jump	AluCtrl	LSW	JAL	JR
0	0	0	x	1	0	0	xor	0	0	0
ADD/SUB/SLT Control										
RegDst	AluSrc	MemtoRg	MemWr	RegWr	Brnch	Jump	AluCtrl	LSW	JAL	JR
1	1	0	x	1	0	0	add/sub/slt	0	0	0

Table 1: *A table displaying which control signals are high and low based on instruction type.*

2 Testing Plan for Modules

We wrote the code for each module individually and, as such, tested each module individually as well. This was done by writing the main Verilog code for the module in a `.v` file, creating a `.t.v` test bench file with a test bench harness for each module, and then creating a `.do` file to run each module. In this way, we were able to create as many test cases as each module needed and be sure that each module functioned properly before adding them together in the complete system.

To test the completed single cycle CPU that we had built, we needed a single script to test all of our modules. Since had already written individual test benches with test harnesses, we determined that to best use the architecture we had created, we needed a way to start each test harness and tell if it was successful. To do this, we gave each harness one input that would tell it to start and two output wires to indicate when it was done and whether the tests had all succesfully passed. Each of these individual test benches is called from `allModules.t.v`. The file `testBenches.do` will run all of these tests. The output from `testBenches.do` has three parts, one each for the single input and the two outputs to the test bench harness, as previously mentioned. Through these outputs, we can guarantee that each of our modules has started testing, finished testing, and returns the result (pass or fail) of the test. For ease of use and readability, there is also a final print statement that indicates whether all modules passed or if there were any modules

that failed. If any modules have failed, it will be indiciated in the individal module print statments from the above outputs.

3 Assembly Testing

We wrote our own assembly testing program, which can be seen in our GitHub repository with an in depth `README` detailing our work.

However, in order to test our processor, we downloaded an assembly program from another team. The reason for this is that our program utilized commands outside of the subset we were required to have in our instruction decoder, such as `addi`, `li`, and `la`. Testing our CPU with another team's assembly code allowed us not to have to implement the additional commands our assembly code used in our CPU.

We were able to instantiate a single cycle CPU module that used all of our component modules and were able to load the .dat file into our instruction memory. However, we could not get our module fully working. We did not see the right signals being passed around our module, even after several hours of testing, debugging, and fiddling with the Verilog code. As of Monday, November 23, we are going to turn in all of our materials at their current state, but hope to continue working on our lab and get our CPU fully functioning within the next few days.

4 Cost/Performance Analysis

In our design we decided to implement the $RA register as a separate register in the instruction fetch unit. We did this mainly so that it would be easier for us to understand how the jump address is stored during a JAL operation. We have realized that this implementation, though easier to visualize and for us to implement, would not be the best design in terms of space or cost efficiency. In order to implement the $RA register in this way, we had to add another control signal, another single 32 bit register, and another multiplexer. However, we think this would increase the speed of operation for JR and JAL operations. Instead of the JR step needing to get the jump address from the register, send it through a mux and an ALU, then through an and gate and 2 muxes, we now just set a control signal high, and the value for jump simply passes through one mux and then to the PC. Although this increases speed for the JR step, it will somewhat lower speed for all other operations, as they will have to pass through an extra mux.

In addition, we added a multiplexer that takes the sign extended immediate and the ALU output as inputs, and outputs one of them to the address field of the data memory based on a new control signal. This could potentially increase speed, as instead of going through the ALU and ALUSRC multiplexer (refer to Figure 1 on page 2) the immediate can go straight through the multiplexer into the data memory address field. This method does add some cost as there will be extra gates in this new multiplexer. Like the previous change, we think this addition will increase speed. Instead of the immediate going through one mux and the ALU before going into the Addr field of the data memory, it will just go though one mux. However, this extra mux will slow down other operations that need to pass through it.

5 Work Plan Reflection

Our original work plan dictated that the lab would take us a total of 27 hours. We expected this to break down to approximately 3 hours to write the assembly test, 2 hours writing the Verilog module and test bench for the data memory, 2 hours for the register file, 1 hour for the ALU, 4.5 hours for the instruction fetch unit, 4 hours for the instruction decoder, and 2 hours for the multiplexers and the rest of the smaller blocks. We expected to spend 4 hours on final testing and debugging, and to account for previous tasks taking longer than expected, and we planned to spend a couple hours each on the write-up. Going into this lab, this work plan seemed very reasonable and even overestimated, perhaps we would even get everything done in less time than we thought we might need.

This lab took us significantly longer than anticipated. Writing the assembly test took around 3 hours as planned. Writing each of the modules and test benches varied quite a bit. We used some Verliog modules from past labs, but because past labs were done almost entirely in structural Verilog, we wound up rewriting many of the modules as they were relatively simple in behavioral Verilog. Our predictions for writing these individual modules were pretty close, although we didn't account for having to write one test script to run all our test benches. Some took far longer than others due to strange errors or little things we got stuck on when creating them. There were some modules we thought would be relatively quick to write, that ended up taking hours to complete since it took so long to debug them when tests failed or would not even compile.

We also didn't account for planning all our control signals and diagramming out the entire system. We ended up structuring our CPU differently than we had thought we would. At first, we thought we would primarily be using the single cycle CPU design shown to us in class. Our implementation ended up being different from this in order to clearly accommodate all of the instructions we needed to include. Our final design did not even have a block for the instruction fetch unit. Instead, we broke up the pieces of the instruction fetch unit and included them as independent blocks/modules witin our higher level design. The one thing we vastly underestimated was the time for wiring everything together and testing the assembly program, as well as debugging at the end. We thought testing the assembly program on our CPU would be relatively simple and straight forward, but after working on this for hours, even after incoporating the exported .dat file from our assembly test into our memory, we were not able to run the assembly code on our CPU, though we spent a significant amount of time on this.

6 Our Team's Final Deliverable

We have completed a CPU design that can complete each of the instructions included in this lab. This design is represented in our block diagram. We have written and tested completeVverilog modules for every element of this design. We have created an individual test bench for each module as well as a test harness that tests all of them at once. We have written a Verilog file that instantiates each of the modules written and connects them all together with the appropriate inputs and outputs moving between modules. We have a test bench for this CPU and are able to load any .dat file into the instruction memory file.

7 Next Steps

The one component of this lab that we were not able to complete and make functional, was running the data exported from our assembly test on our Verilog code. We struggled with how best to implement this and how to get the data incorporated with the rest of what we had created for our CPU. As of Sunday, November 23, we hope to work on this for the next few days and get it fully functioning, as we are confident that we are just a few small changes away from getting it fully working.