CompArch Lab 3

Logan Sweet & Maggie Jakus

17 November 2017

1 Introduction

In this lab, our goal was to create a CPU that could execute a reduced MIPS instruction set. Despite many hours and a lot of help from the ninjas and Ben Hill, we were not able to complete the CPU by the due date. In this report we will discuss our CPU design, our modules and how we tested them, the results of the CPU as they are now, and how we will move forward to get our Verilog working in the next week or so.

2 CPU Implementation

Before we implemented our CPU, we created a spreadsheet of all the possible instructions and how they would be implemented in each phase of our CPU design. This table is copied in figure 1.

	IF	ID	EX	MEM	WB
LW	Instruction register = memory[PC] PC = PC + 4	A=RegFile[rs] B=RegFile[IR[16:20]]	Result=A+SignExtendImmediate	DataReg=Mem[Result]	RegFile[rt]= DataReg
SW	Instruction register = memory[PC] PC = PC + 4	A=RegFile[rs] B=RegFile[rt]	Result=A+SignExtendImmediate	Mem[result] = B	-
J	Instruction register = memory[PC] PC = PC + 4	PC=PC[31:28], IR[25:0], b00	-	-	-
JR	Instruction register = memory[PC] PC = PC + 4	A=RegFile[rs] PC=A	-	-	-
JAL	Instruction register = memory[PC] PC = PC + 4	RegFile[31]=PC+4 PC=PC[31:28] IR[25:0], b00	-	-	-
BNE	Instruction register = memory[PC] PC = PC + 4	A=RegFile[rs] B=RegFile[rt] Res=PC+SignExtendImmidiate	If(A=/=B) PC=Res	-	-
XORI	Instruction register = memory[PC] PC = PC + 4	A=RegFile[rs] B=SignExtendImmediate	Result = AxorB	-	RegFile[rt] = Result
ADD	Instruction register = memory[PC] PC = PC + 4	A=RegFile[rs] B=RegFile[rt]	Result = A+B	-	RegFile[rt] = Result
ADDI	Instruction register = memory[PC] PC = PC + 4	A=RegFile[rs] B=SignExtendImmediate	Result = A+B	-	RegFile[rd]=Result
SUB	Instruction register = memory[PC] PC = PC + 4	A=RegFile[rs] B=RegFile[rt]	Result = A-B	-	RegFile[rt]=Result
SLT	Instruction register = memory[PC] PC = PC + 4	A=RegFile[rs] B=RegFile[rt]	Result[31:1] = 0 Result[0] = 1 if (A<B) Result[0] = 0 otherwise	-	RegFile[rt]=Result

Figure 1: We made this table to use as a tool to guide our CPU creation. By separating each operation out into instruction fetch, instruction decode, execute, memory, and write back phases, we were better able to decide how to write each one.

To create our control signals, we used the idea of a finite state machine to define what things were enabled or not depending on the relevant instruction.

Our CPU is technically two cycles. Some of the instructions require writing back to the memory or regfile, which we implemented through two cycles. Most things happen in the first cycle, and are then disabled during the second cycle. Only the special cases that involve writing back have active controls during the second. A diagram of our CPU can be seen in figure 2

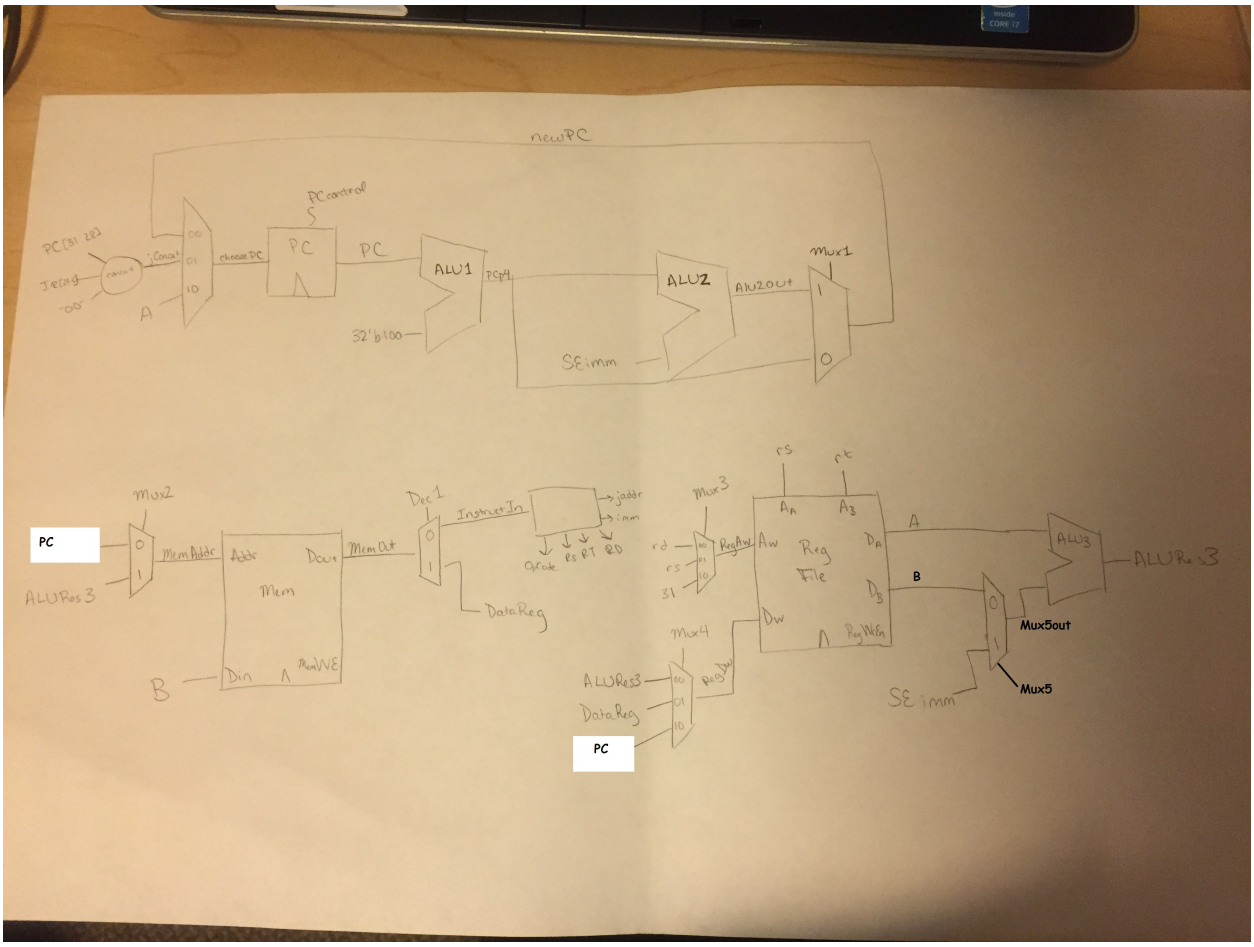

Figure 2: Our tastefully edited CPU diagram. The labels correspond to the variable names in our code, because we are very organized. This photo is also in our github.

3 Test Benches in Verilog

We used test benches in Verilog to check each module before we fully connected the CPU. We used these to check a few simple examples at the beginning, then attempted to test edge cases we thought could arrive as well. Many of these modules were recycled from our previous labs and homework assignments (we did not explicitly cite ourselves in these cases) so we were relatively confident that these modules worked as we expected anyway.

3.1 Testing Modules

We tested the mux, decoder, datamemory, regfile, and ALU all separately. We took this as an opportunity to work on our self-checking skills.

A challenge for us was implementing self-checking rather than our usual method of manual inspection. We realized that we would have to implement the same testing strategies for either manual inspection or automatic testing results for each module, so we implemented manual inspection methodology first. This

was helpful in debugging our CPU since we could see different outputs based on specific inputs that were easy to manually set. This allowed us to check the accuracy of a module even if a module or piece of the CPU that came before it wasn't working. This way, even if the full CPU was not feeling the module we wanted to test the value we wanted to test on, we were still able to see the output under those conditions. This also had the added benefit of not needing to chase signals through the GTKwave file of the full CPU test if we just needed a single module checked.

Even though we took advantage of the manual inspection method, we also wanted to try to do automatic validation. As of the writing of this text, we have only done self-checking on the decoder and one of the muxes. We checked all of the modules through the usual methods, and we will work on the self-checking in the future. Our current self-checking can be found in "testInstructions.sh".

3.2 Testing the FSM

Our FSM gives us controls to use in our single cycle CPU. We used Verilog and GTKWave to confirm that our FSM gave us the appropriate controls.

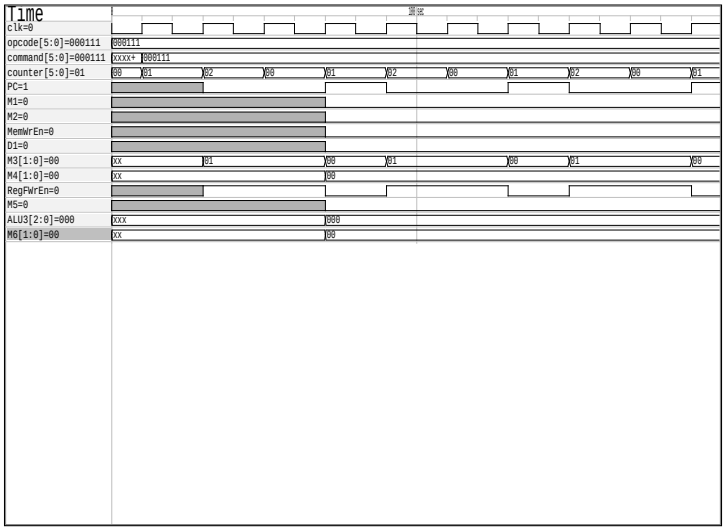

Figure 3: We input the opcode for "add" (000111). We can see this in command. We can see that most controls are zero. When counter equals 2, however, Mux 3 goes to 01 and RegFWrEn goes to 1. This follows what we assigned in the FSM, and gave us confidence that our FSM was working.

Sadly, this is an old photo - we later learned that the MIPS commands overlapped a lot more than we thought they did, so we needed to use both the opcode and the function code to specify which instruction we wanted to enact. We realized this around 8pm on Thursday evening and were able to make the necessary changes. We believe the behavior of the FSM was not affected, but this is something we will have to consider and double check as we move forward.

4 Test Benches in Assembly

4.1 Test Plan

We wrote three assembly tests using MARS, and submitted a pull request so that the whole class could use them.

4.2 Test Results

To preface this subsection, we didn't get a lot of test results.

We began by testing the simplest functionality: Add, Sub, and Addi. The instructions can be seen in figure 4.

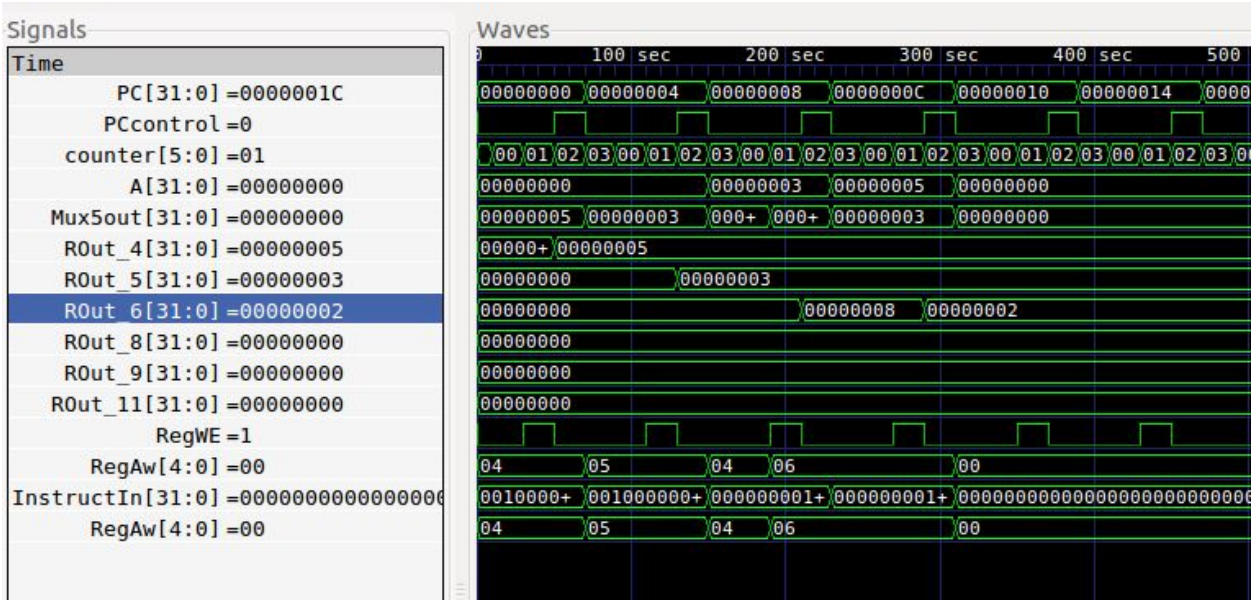

Figure 4: We add 5 to a0 and 3 to a1. We add a0 and a1 and save that in a2. Then, we overwrite a2 with a1-a2.

We can see that figure 4 works in the GTKWave results (figure 5).

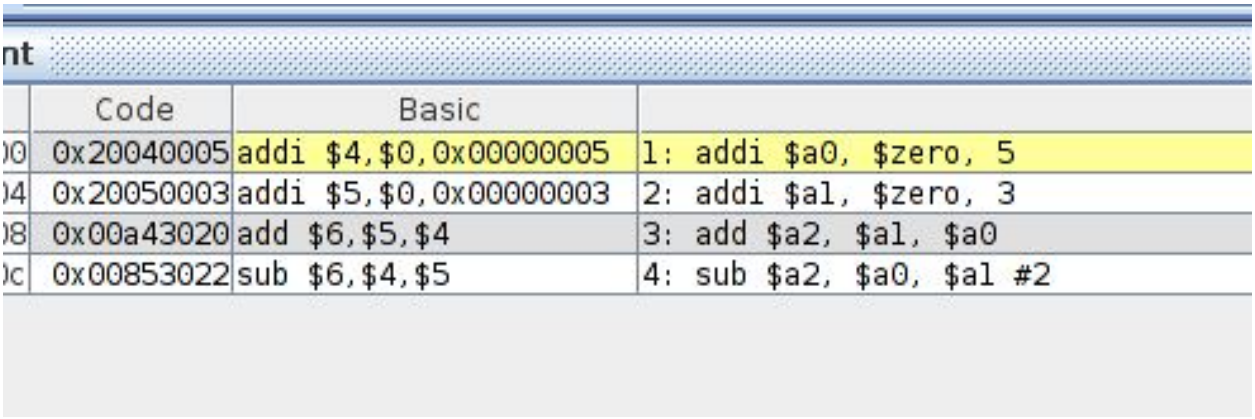

Figure 5: It works! ROut4 is a0, ROut5 is a1, and ROut6 is a2.

These look good and make sense. However, we had issues with adding more instructions, which we believe has to do with issues in our timing. We will be pursuing these in the future.

Next, we tried to instantiate XORI, but we ran into a lot of issues. We also tried to do jump, but, once again, we had problems. We believe these all have to do with timing.

5 Performance/ Area Analysis

We do not have the most efficient CPU in terms of performance. (even assuming that it worked). We wanted to be sure that we could easily follow our signals through the CPU, and created a sort of dual-cycle design that was roughly based on a single-cycle, but with a counter to drive the control signals to different modules in the CPU. This is less efficient than a pipelined design, but we hoped that it would be less complicated to implement.

6 Moving Forward

Ultimately, our CPU at the time of the due date does not have full functionality. Right now, our problem is in the transfer of control signals from our "finite state machine" to the rest of the CPU. We know this is more like a dual-state machine: right now it provides up to two different sets of control codes depending on a counter. Since these are not correctly moving from the control to the rest of the CPU, we are not getting the results we want to see. We'll begin fixing this by going over our controls again to confirm that we understand what should be on when. Timing was hard for us to understand - we will work on understanding when certain things happen (when is a value written? when is it read?), and this will help us build a more correct FSM and CPU.

We want to succeed at this lab, and both plan to work on it for around an hour each day over break. Before we dive into what we have written so far, we'll take a high-level view of the CPU as a whole and evaluate how we should define each of the pieces of the system, including the control signals, registers, and memory.

We also want to make our documentation clearer. We will remove old files, comment the remaining ones, and create more useful test benches. We worked on our self-checking testbench skills on this lab, but we didn't have the time to do all of the self-checking test benches that we wanted to. We will add those (for the registers, memory, and ALU), and we will hopefully have more comprehensive assembly tests.

We spent a lot of quality time with GTKwave as part of this lab, which was a very useful tool. To make it more useful and easier to read, from now on we will work on using more descriptive names for the module instantiations so that we can see and understand the signal paths on GTKwave more quickly.

7 Work Plan Reflection

We didn't do a great job tracking the amount of time we spent on each individual task, since different things tended to bleed over into one another during meetings. We ended up spending more time on writing the assembly test code than expected; we had some trouble understanding the full-stack concepts and how they were used in the Fibonacci example from the In-Class folder. After talking to Lisa and Tom we realized that we could heavily simplify our goals for the assembly test, and instead of one complicated test script we wrote three simpler ones.

Overall, this lab took much more time than we expected. We found ourselves continuously finding new different issues, and in the process of fixing them would break a different piece of our CPU.

Individual reflection: Maggie

This lab was lit. It was terrible and educational and an incredible amount of work. I feel like I learned a lot about a lot of things - I finally understand *some* of the nuances of Verilog, and I actually had to think about how timing works, which is hard but cool. I got to learn about the different instruction types, and I had the opportunity to be utterly confused by MIPS and what opcode and functcode go with what actual functionality. I learned about the importance of GTKWave. Like, I knew it from before, but this was the time when I really saw how it could be useful in debugging.

I also got to work on my self-checking test bench skills, which I'd meant to do all along and finally got to! I'm excited to keep working on this but with a fresh perspective - we talked about taking a day off and then looking at it from the beginning when we resume. I have pretty high hopes for us (like maybe 80%), and I think it'll be pretty great when we're done.

Individual reflection: Logan

This lab crushed my soul and destroyed my self worth as an electrical engineer. I felt like a bad partner because I was not able to devote as much time to the lab as my awesome partner Maggie, and because of this I think that my learning suffered a little. Despite this, I got more experience in evaluating what could be going wrong in the CPU, and chasing signals around in GTKwave. I think a lot of the negative feeling I have about this lab stem from the fact that we got started on this soon after getting the assignment, asked for help early and (maybe too) often, and worked hard to understand each piece of the system, but were still not able to be successful. I'm not sure if we need to do even more planning before we jump in from now on, or if we were just very unlucky in the locations and types of mistakes that we made along the way.