Lab 3: CPU Lab

Ryan Eggert, Shane Skikne, Marie-Caroline Finke, William Saulnier

November 20, 2015

1 Description and Block diagram

We implemented our own version a single-cycle CPU, that can handle 13 different MIPS instructions. The 13 MIPS instructions are LW, SW, J, JAL, JR, BNE, XORI, ADD, ADDI, SUB, SLT, SYSCALL, and NOOP. The CPU was made up of a program counter, an instruction memory, a control unit, registers, an ALU, a data memory, and a few other basic components to connect them. The design can be seen in Figure 1. Since this was a single cycle CPU, we didn't need to make many major design decisions.

In our control system, we took in the instruction and output the correct coding to drive each of our functional blocks as well as the various muxes. Using the structures of the various instructions as well, we were able to properly divide up the system into it's consummate parts without too much effort. The one instruction which required the most additional implementation was Jump and Link, being the only jump instruction that also requires a register write at the same time, it necessitated the insertion of three additional muxes to support its inclusion.

The processor is clocked at 50MHz. No gate delays are modeled in this design, so this speed was chosen fairly arbitrarily. Only two modules in the design are clocked, the PC, which is clocked on the positive edge and the Data Memory, which is clocked on the negative edge. This makes it so the CPU's cycle is effectively broken into 2 pieces. In the first piece, after the clock rises, the PC is used to retrieve the instruction and decode it. The decoded instruction is then passed onto the Control Unit, to set the control signals, and the Register, to retrieve the necessary information. From there, the CPU is able to start actually performing the instruction based on what the control signals were set to and the data retrieved. When the clock goes down, then the data memory is clocked. At this time, the CPU can either store data in the data memory or load data from Data Memory. For the instructions that don't touch the data memory, the instruction would be complete before clock goes down, so the CPU is waiting for the clock to rise again to handle the next instruction.

2 Test Plan

Our test plan had 4 different components. First, we have the master test which tests each individual component of our CPU. These tests are completely independent and they simply ensure each piece functions correctly. Next, we have our overall CPU test, which simply ensures the CPU compiles. This test is quite simple, but allowed us to immediately see if we broke anything. Third, we also performed visual tests on the waveform to ensure that the signals were actually moving through the CPU as we expected. Finally, we have all the assembly tests, which ensure that our CPU is actually functioning properly.

Every major component module of the CPU had a dedicated testbench to verify its basic functionality. A few tests were taken directly from previous homework or lab assignments, some were modified from previous homework or lab assignments, and some were written from scratch for this lab. `/components/all.t.v` runs all of the components' test benches simultaneously, allowing quick validation the individual pieces of the CPU. This was important, as some components (e.g. the ALU) were completely rewritten for this lab.

Simply attempting to compile the CPU's code proved very useful in helping us debug most of our naming issues and ensure we were using most of our wires correctly. As soon as we had resolved all the errors and our CPU compiled, we could consider the next part of the test plan.

Waveforms proved to be tremendously useful in diagnosing the causes of CPU behavior issues when used in conjunction with assembly tests. Notice in `/cpu.do` the great number of waveforms used for monitoring the processor's behavior. For example, waveforms allowed us to see that we were decoding PC addresses incorrectly, resulting in our processor generally executing every fourth instruction. This also allowed us to diagnose the cause of our processor behaving unexpectedly when branching. Ruling out issues with the ALU and register file, we found a misnamed wire and a missing inverting gate.

Finally, the assembly tests allowed us to verify that our processor could process machine code and demonstrate complex, programmed behaviors. Our assembly tests were designed to use all of the required instructions.

We used two simple assembly programs for initial processor debugging. `reggert/simpleadds` loads three numbers into registers using `addi` ,then performs three addition operations using these values, and finally send a SYSCALL 10 to end the program. `reggert/simplejump` performs the same loading and three addition operations as `reggert/simpleadds`, but adds a jump instruction which, if executed properly, will skip a fourth addition operation and perform an SLT before sending a SYSCALL 10.

Our main assembly test, `mc_finke_and_the_boys/asmtest`, tests every required instruction. Based on two integer inputs, the program repeatedly branches, switching between two looping conditions until they conclude with a final write to register. The program can be represented by the following pseudocode.

```
while counter > 0
    if counter > total
        total = total - counter
    else
        total = total + counter
    counter--
```

At the end of the program, the value in total is written to a register, which can then be verified to be correct by the user. A SYSCALL 10 is finally raised, ending execution.

3 Performance and Area Analysis

We found that the assembly programs written ranged anywhere between 9 and 106 clock cycles depending on how many computations were being run. Table 1 shows the results for our tests.

group	time	clock cycles
mcfinkeandtheboys	1970 ns	99 CC
JasonJordanThuc	690 ns	35 CC
KaiIsaacDavid	490 ns	25 CC
meggriffbrenna	330 ns	17 CC
nurpatricksarah	indefinite	indefinite
reggert: simpleadds	170 ns	9 CC
reggert: simplejump	210 ns	11 CC
yoloisso2013	indefinite	indefinite
YuzhongSelinaHieu	310 ns	16 CC

Table 1: Timing and performance results from running a variety of assembly programs.

We found that some of the assembly test benches weren't working at all in ModelSim as they were missing system calls that actually ended the program once it was done running. This was true for the KaiIsaacDavid

assembly, the yoloisso2013 assembly and the nurpatricksarah assembly. For the KaiIsaacDavid assembly we were able to add in a system call to stop the program once it was done, however the other two run indefinitely. We see at least in the case of yoloisso2013's test that this test fails because it relies on instructions that our processor does not handle (out of spec instructions, e.g. BEQ, BLT, BGT), and as such cannot be executed by our CPU.

There are no significant gated delays in our system's modelsim tests, because of the positive and negative edge clocking we had no case where the processor had to be put into a locked state.

Component Type	**Inputs**	**Width [bits]**	**Quantity**
Adder	3	33	2
Adder	2	32	2
XOR	2	32	1
Register	N.A.	32	34
RAM	N.A.	128K	1
MUX	2	32	6
MUX	9	32	1
MUX	7	6	1
MUX	5	6	1
MUX	2	5	2
MUX	2	4	1
MUX	9	4	1
MUX	9	3	1
MUX	9	2	5
MUX	9	1	8
MUX	7	1	5
MUX	3	1	4
MUX	4	1	1

Table 2: Synthesis of the CPU design in Vivado yielded these component requirements.

We successfully synthesized our CPU implementation in Vivado. This showed that our design requires 77 discrete components. Vivado synthesized our design into RAM, a register, a two-input XOR gate, two adders, and an assortment of multiplexers [MUXes] No component required more than 9 inputs. Table 2 shows a summary of the synthesis results. The complete generated synthesis report can be viewed in the `/vivado_sythesis/MIPS_CPU/reports directory`.

4 Work Plan Reflection

Overall, our workplan allocated a pretty accurate chunk of time to complete this lab, though some individual pieces were significantly off. For example, with the exception of control, which we wrote for this lab, each individual component took much less time than we allocated. On the other hand, connecting all the wires and getting our CPU up and running, took much time than we expected. Once the CPU compiled successfully, it still took hours of testing and running programs to get it functioning properly, which we didn't accurately plan for. We also came across issues working with new team members, where we had to get used to each other's working styles.

While there are no more labs remaining, there is a project, which might use Verilog. Our major takeaway in planning is that it would be wise to allocate equal time to putting the entire system together and testing it as building all the pieces. Even though putting the pieces together should theoretically be simple once they're all implemented, it proved to much more complicated than it seems.

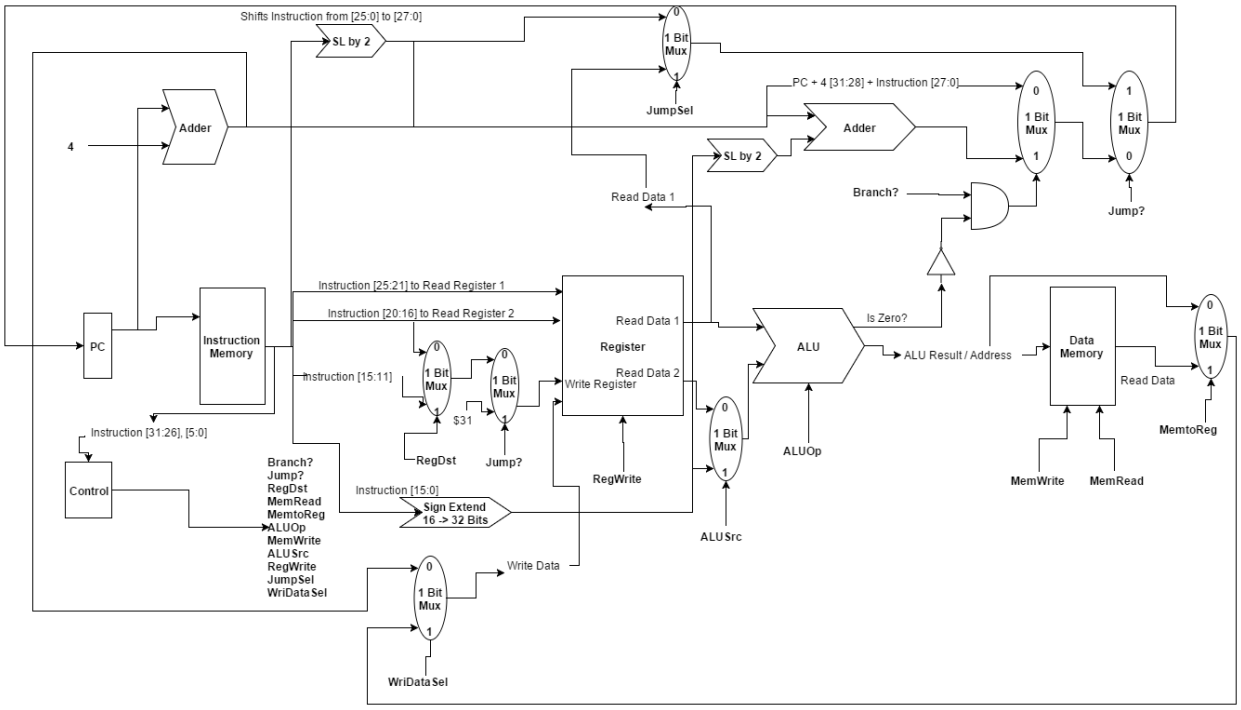

Figure 1: *The Block Diagram of our Single-cycle CPU.*