Computer Architecture Lab 3: Single Cycle CPU

Daniel Connolly, Josh Deng, and William Fairman

November 2, 2018

1 Overview

Our single-cycle, 32-bit CPU is a culmination of modules from previous labs and homework assignments. Running off of machine code compiled from assembly files, our CPU is capable of loading/storing variables to data memory, jumping (jump, jump register, jump and return), branching, and completing various ALU based tasks: ADDI, ADD, SUB, SLT, XORI.

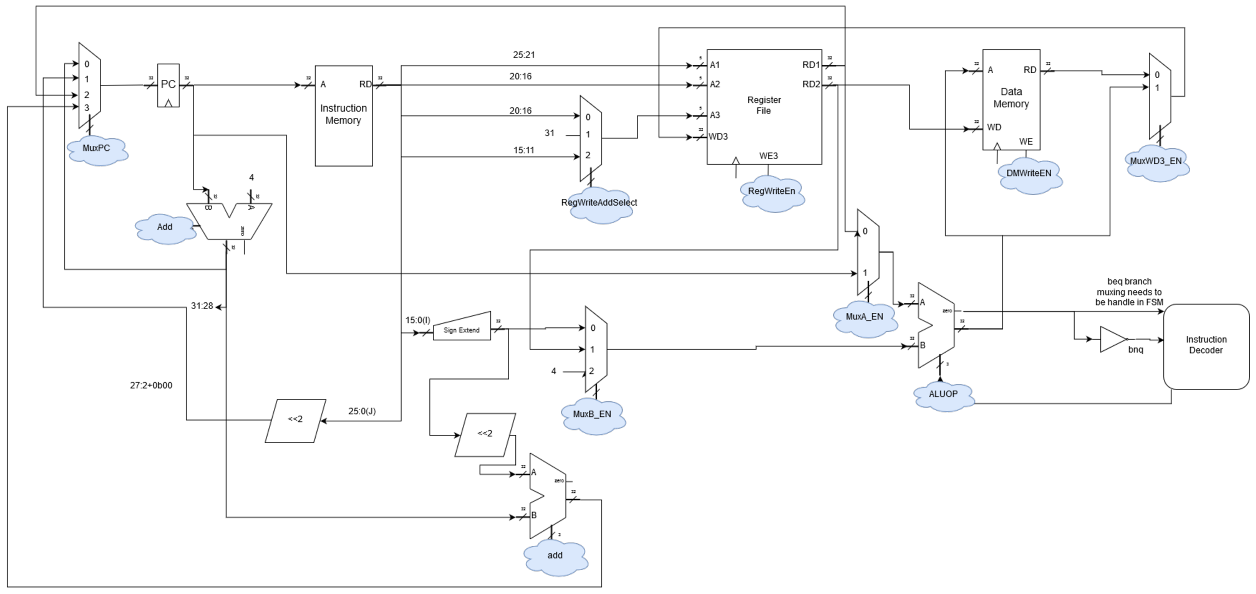
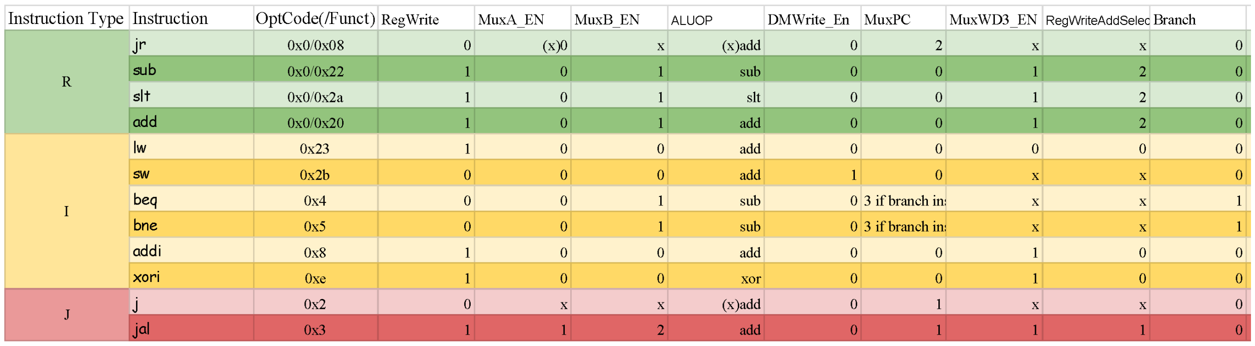


Figure 1: The block diagram of our Single-Cycle CPU.

In order to run our CPU, we must upload assembly code directly to the data memory, which the program counter then reads. The instructions our CPU needs to run, which is held in this assembly code, and the data memory, which can also be modified using a data file from the mars assembly simulator, are stored in the same memory array. However, in the diagram in Figure 1, we separated our instruction and data memories to simplify and improve the readability of our block diagram. To accommodate for all of its purposes, we included two read ports in the memory unit so instructions and stored variables can be read at the same time. In order to run a given instruction, our program counter sends its position to the instruction memory, at which point an instruction is selected and the instruction decoder defines all of the control logic for the CPU, as is shown in Figure 2. These control signals manipulate the CPU such that the instruction is properly run and the correct value is stored or loaded by our register file or memory unit and the program counter accurately adjusts itself on each clock cycle.



Instruction Type	Instruction	OptCode(/Funct)	RegWrite	MuxA_EN	MuxB_EN	ALUOP	DMWrite_En	MuxPC	MuxWD3_EN	RegWriteAddSelec	Branch
R	jr	0x0/0x08	0	(x)0	x	(x)add	0	2	x	x	0
	sub	0x0/0x22	1	0	1	sub	0	0	1	2	0
	slt	0x0/0x2a	1	0	1	slt	0	0	1	2	0
	add	0x0/0x20	1	0	1	add	0	0	1	2	0
I	lw	0x23	1	0	0	add	0	0	0	0	0
	sw	0x2b	0	0	0	add	1	0	x	x	0
	beq	0x4	0	0	1	sub	0	3 if branch in:	x	x	1
	bne	0x5	0	0	1	sub	0	3 if branch in:	x	x	1
	addi	0x8	1	0	0	add	0	0	1	0	0
	xori	0xe	1	0	0	xor	0	0	1	0	0
J	j	0x2	0	x	x	(x)add	0	1	x	x	0
	jal	0x3	1	1	2	add	0	1	1	1	0

Figure 2: Table of control signals based on various instructions.

2 Test Case Strategy

Because of the single-cycle nature of this CPU, debugging using waveforms proved far more complex than in previous assignments, which utilized components that took multiple cycles to run. Still, we were able to drill down into the components with these waveforms in order to identify issues with control signals and output values, as we discuss with regards to our multiplexer in Section 3. Additionally, we ran extensive tests with assembly code, during which we read the states of several registers during each program and when these registers reached their final stable states, which we then compared with the expected outcome based on results from the assembly simulator. We automated this process with a cpu.t.v file that ran multiple programs, checked the expected outcomes versus reality, and reported the results of those tests.

We strategically chose several programs in order to construct an effective array of tests. We varied our test cases from assembly code that tested only a few instructions in isolation to code that stressed the system as a whole at once in order to ensure the functionality of all of the individual commands that our CPU is designed to handle. Though code that relied on multiple instruction codes proved difficult in terms of pinpointing small issues, it helped to prove the validity of the system. Several of the tests we ran to observe our system under stress are listed below.

1. Test Cases 1, 2, and 3
    - We utilized our own assembly test cases in order to calibrate our CPU. These test cases involved utilizing all of the instructions our CPU is designed to handle.
2. Test Case Hanoi:
    - This program contained code that appeared rather complicated and was written by fellow students. We assumed this test would stress our system by running a large number of instructions with large variables. See Figure 4.
3. Test Case Fib:
    - The Fibonacci code is an assembly written following a pseudo c recursive Fibonacci function. This program utilized a stack pointer, which was another variable we needed to consider in constructing our CPU. Ultimately, we set the stack pointer using the ADDI command in our assembly code. See Figure3.

Combined with the overall CPU.t.v file, we had individual test benches for each of the modules. This helped with the initial implementation and validation of each module before we combined them all into the final system. While module test benches were not used in the final stages of debugging our computer, we are confident that they prevented us from spending even more hours debugging our system.

3 Results

During our initial implementation, we ran into several errors with hardware components and instruction decoding. Originally, our multiplexers returned only the four least significant bits of any input pin. We addressed this issue by first changing the output of the muxes to be of the register type such that we could assign them a value inside an always block with a combinational sensitivity list. The design previously mapped the four inputs to a two-dimensional wire with a width of four bits and chose one of the four two-dimensional wires to output based on the multiplexer's select pin inputs. Second, we used a "`define" statement to avoid recursive includes.

We also encountered issues with the stack pointer. We had initially planned to implement the initialization of our stack pointer with hardware logic; however, this would have required a major redesign of the shift register where our pointer lies. As a result, we chose to set the stack pointer at the beginning of each of our assembly test files.

After fixing our hardware setup, we were able to load and process machine code. However, this system could only run simple commands and failed to run recursive functions. What we initially thought was an error with our stack pointer was actually an extra right bit shift in our memory.v file. After removing the extraneous bit shift, we were able to run the hanoi assembly code and a Fibonacci to the tenth iteration. Figures 3 and 4 show glimpses of how the machine code drives various pins in the system.

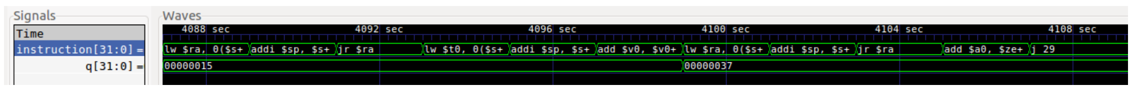


Figure 3: A snippet of the waveforms output by our CPU when we tested our Fibonacci function are shown above. These waveforms clearly display the moment that the q output value of register two in our register file transitioned to its final correct and stable state of 8'h37, which is fifty-five in the decimal base-ten system. Given that we had implemented our assembly code to look for the tenth number in the Fibonacci sequence, and fifty-five is the tenth number in that sequence, we know our CPU has successfully ran a variety of instructions, including recursive calls.

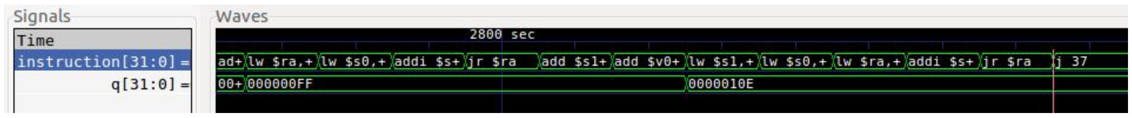


Figure 4: A snippet of the waveforms output by our CPU when we tested the Tower of Hanoi function are shown above. Prior to running this code, we had checked the final values that should be stored in our registers in the Mars assembly simulator. These waveforms demonstrate the q output value of register two as it transitions to its final stable state of 8'h10E, which is the value we had expected based on the Mars assembler.

4 Performance Analysis

The size of our CPU is quite large due to not only the register file array and the sheer size of the memory array used for the instructions and data/stack, but also due to the three arithmetic logic units used for our program counter, branch operation, and actual arithmetic/logic operation. As this is a single-cycle CPU, there are very few pieces of hardware that are being used across different stages of within a single instruction. The size of the computer can be reduced, for example, by directly feeding $JAL's\ PC + 4$ to the WriteData of the register file instead of introducing an extra input to the muxB or by utilizing adders rather than full ALUs for some arithmetic. Nonetheless, this leaves flexible space for us to expand our CPU to allow for pipelined CPU's $PC + 8$ way more easily without any significant hardware redesign.

At the same time, we chose to employ existing Verilog modules in an effort to reduce the number of distinct modules we have to write. That decision led us to, among other things, utilize a four input mux for a three-input mux with one of the inputs left floating. Thus, the area of our current design is significantly larger than what it actually could be if we spent time to optimize the system.

Unfortunately, we were unable to properly generate the Vivado file to get quantifiable performance characteristics.

5 Work Plan Reflection

We set up our work schedule to be heavily skewed toward the second week of this lab because of the PoE sprint reviews that happened during the first week of this lab. Despite the setback in time we had, we were able to read through Section 7.3 of Digital Design and Computer Architecture and work efficiently. We were able to move from just starting to draw our block diagrams and control tables to collecting all the necessary Verilog modules in less than a day. However, we were unable to produce a meaningful assembly test program by the time we started wiring our top level designs together. The assembly program ended up taking triple the amount of time than we expected as we rushed to finish our Assembly programs the day before they were due. Also, we spent approximately 25 hours as a whole working to complete the test benches for the top level cpu module and each submodule.

6 Module Reuse

For Lab 3, we reused the following modules, with alterations as deemed necessary, from the following sources:

- Arithmetic Logic Unit and its accompanying test bench from Daniel's and Josh's Lab 1 design
- Register, multiplexer, and decoder from Daniel's HW4
- RegisterFile from Josh's HW4