Lab 3: Single-Cycle CPU

Prava Dhulipalla, Annie Ku, and Judy Xu

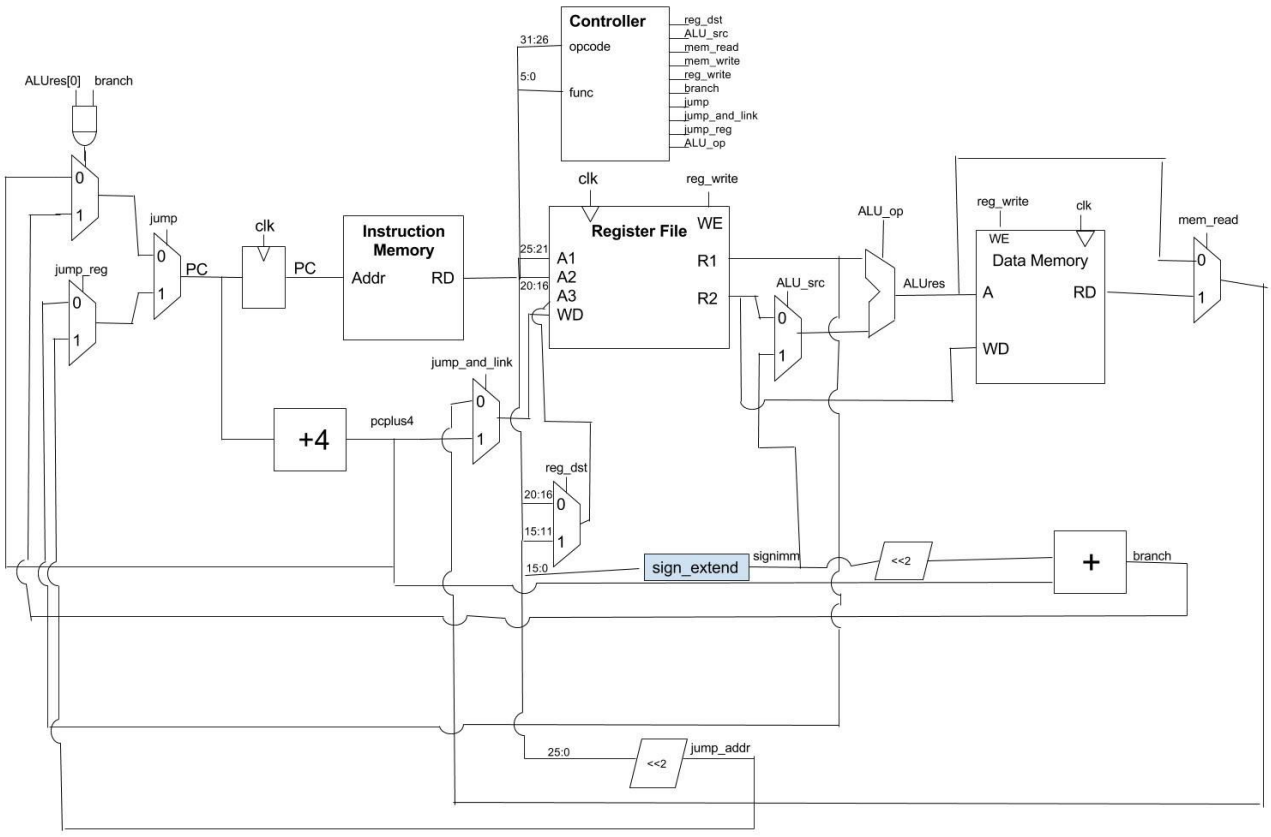
Overview

For our third lab, we designed and implemented a single-cycle CPU with a limited set of instructions. These instructions were:
`LW, SW, BNE, J, JR, JAL, XORI, ADDI, ADD, SUB, SLT`

This lab report serves as an overview of our implementation, iteration, testing, and documentation throughout the process of implementing a single-cycle CPU.

Processor Architecture

Block Diagram



Control Table

As you can see from the block diagram above, part of our CPU implementation includes the control logic, which takes in the op code and the function code and outputs a variety of signals based on that. However, the diagram doesn't explain what signals get asserted for what combination of op codes and function codes.

To determine this, we implemented a LUT that assigns values to control signals based on the type of instruction being passed to the CPU. The following is a table of op codes and function codes for each of the instructions supported by our CPU:

Instruction	Op Code	Function Code
LW	h23	--
SW	h2B	--
J	h2	--
JAL	h3	--
BNE	h5	--
XORI	hE	--
ADDI	h8	--
ADD	h0	h20
SUB	h0	h22
SLT	h0	h2A
JR	h0	h08

Based on these instructions, we could then set the control signals. The control signals we used were:
`reg_dst`: determines destination of write address, which is different for R-type and I-type instructions
`alu_src`: control signal for the mux that determines what gets send at the second operand into the ALU
`mem_to_reg`: write enable for the register file from data from memory
`mem_read`: 'read enable' signal for data memory
`mem_write`: write enable signal for data memory
`reg_write`: write enable signal for the register file

`branch:` is the instruction a branch instruction
`jump:` is the instruction a jump instruction
`jump and link:` is the instruction a jump and link instruction
`jump register:` is the instruction a jump register instruction
`ALU_op:` setting the appropriate ALU op code, if needed for the instruction

These control signals were set accordingly with the instruction determined by the op and function codes, as per the following table:

	Ctrl Signals										
Instr	**reg_dst**	**ALU_src**	**mem_to_reg**	**mem_read**	**mem_write**	**reg_write**	**branch**	**jump**	**jump_and_link**	**jump_register**	**ALU_op**
`LW`	0	1	1	1	0	1	0	0	0	0	opADD
`SW`	0	1	0	0	1	0	0	0	0	0	opADD
`J`	0	0	0	0	0	0	0	1	0	0	--
`JAL`	0	0	0	0	0	1	0	1	1	0	--
`BNE`	0	0	0	0	0	0	1	0	0	0	opCNE
`XORI`	0	1	0	0	0	1	0	0	0	0	opXOR
`ADDI`	0	1	0	0	0	1	0	0	0	0	opADD
`ADD`	1	0	0	0	0	1	0	0	0	0	opADD
`SUB`	1	0	0	0	0	1	0	0	0	0	opSUB
`SLT`	1	0	0	0	0	1	0	0	0	0	opSLT
`JR`	1	0	0	0	0	0	0	1	0	1	--

Modified ALU

Our control table uses op codes for the ALU that we created for our single-cycle CPU. We ended up modifying our previous ALU to reflect our reduced subset of instructions. We also added a more useful command for our implementation of the `BNE` instruction - Check Not Equal, or CNE. This would return a 0 if two values are equal and a 1 otherwise.

The following table represents the op codes for our reduced subset of ALU commands:

ALU Operation	Corresponding Op Code
ADD	d0
SUB	d1
XOR	d2
SLT	d3
CNE	d4

Description

Overall, our single-cycle CPU utilizes several important functional blocks: muxes, memories, registers, ALUs, adding blocks, shifts, and sign-extends.

We start off with the program counter. This program counter selects which line of the instruction to call. This instruction memory is 16KiB, organized into a 4096 element array of 32-bit words (each 32-bit work is an instruction). This instruction is then decoded with various components being saved into the RegFile (depending on the instruction), and from there the instruction is parsed. Aside from that, most of the other components are simply to help execute the various instructions that our CPU needs to execute.

Note that we use adding blocks instead of ALUs in two cases during which we always wanted to add. We did this at a suggestion that it was better to use a behavioral summing block if we know that all we want to do at a certain point is add, rather than reinstantiate an ALU - a far more complex component that would take up a lot of space on a real circuit board. Therefore, we replaced the ALUs originally used to increment PC by 4 and the ALU used to calculate the branch address with simple summing blocks.

In addition, we chose to use the Harvard architecture for implementing memory. This means we separated the instruction memory and the data memory, In reality, there is only one memory; however this approach leads to a simpler implementation. However, one pitfall with this design (other than the fact that it is not wholly representative of an actual CPU that one would find in a laptop) is that we have to be more careful, when testing, to make sure the address we put into both correspond with each other.

Our schematic shows a lot of 2-by-1 multiplexers, some in series. Although we could have simplified certain areas by simply creating larger multiplexers, we didn't. One was partially because we wanted to reuse components from earlier labs and homework assignments, not just for simplicity, but to truly get into the spirit of the lab being the culmination of everything we have

learned/done so far. Secondly, we felt that sending the control signals was a lot more straightforward with a bunch of smaller muxes.

To showcase the general performance of our CPU, we have highlighted two very similar instructions.

ADD

```
IF:   Instruction Register = Memory[PC]
      PC = PC + 4
ID:   A = RegFile[Rs]
      B = RegFile[Rt]
EX:   Result = A + B
WB:   RegFile[Rd] = Result
```

The ADD instruction, as its name probably reveals, adds two operands together. In the instruction fetch phase, the instruction registers grabs the instruction from the appropriate address in Memory (at PC). The Program Counter is then updated to the next instruction - which is located at PC + 4. In the instruction decode phase, the instruction is ‘decoded’ into actually useful commands. Operand A is located at the Rs address of the Register File, and Operand B is located at the Rt address of the Register File. During the Execute phase, Operand A and B are added together with an Arithmetic Logic Unit. This result is then written back to the Register File at address Rd, during the Write Back to Memory stage. This instruction is an R-type instruction, which means it has three address (Rs, Rt, and Rd) within it. To select this instruction, then, it not as simple as an op code - the function code must be taken into account as well.

ADDI

```
IF:   Instruction Register = Memory[PC]
      PC = PC + 4
ID:   A = RegFile[Rs]
      imm = instruction[15:0]
EX:   Result = A + SignExtend(imm)
WB:   RegFile[Rt] = Result
```

The ADDI instruction is very similar as the ADD instruction - it adds together an operand and an immediate. However, notice that in the instruction decode phase, while Operand A is still read from the Register File at address Rs, the second operand is the immediate, which is located in the instruction itself. This instruction is an I-Type instruction, has it holds an immediate as well as addresses for Rs and Rt. In the execute phase, there is also a difference. Though the use of the addition operation in the Arithmetic Logic Unit is still utilized, the operands are A and the immediate.

A More Specific Look into an Instruction

Now that we've taken a general look into the instruction, let's go through specifically how the instruction relates to the RTL, and how that related to the schematic and the Verilog implementation of the instruction.

```
LW
IF:   Instruction Register = Memory[PC]
      PC = PC + 4
ID:   A = RegFile[Rs]
EX:   Result = A + SignExtend(imm)
MEM:  DataReg = mem[Result]
WB:   RegFile[Rt] = DataReg
```

Instruction Fetch

The purpose of the LW instruction, known as the "Load Word" instruction, is to get a value from the Data Memory located at a specified address. To start off, then, we need to fetch the actual instruction from the Instruction Memory. This is common across all instruction types. In the schematic, the instruction is read from the instruction at address PC. PC is then incremented by 4 to reach the next instruction. We do this by using an adder block - in Verilog we implemented with with a behavioral adder at the suggestion of the professor. Our memory in Verilog follows the same indexing strategy, which each instruction taking 4 bytes. PC is a D-flip-flop which is triggered at the positive edge of the clock (the only input into the system). This is then loaded into our instruction memory at the address input - and out comes an instruction. Our instruction memory reads from .txt files with the instructions generated from MIPs Assembly programs we have written. Each instruction line is then saved as instruction.

Instruction Decode

The instruction has to be split into its various parts in order to do useful operations. LW uses an I-Type instruction, because relies on the use of two addresses and an immediate. In Verilog, during this phase, the instruction is split up into it's various parts - the addresses, the immediates, etc. Then is then put into the controller to determine what instruction it actually is. LW would then be triggered, and it would read a value from the Register File at address Rs.

Execute

During the execute phase, the address is computed. The address is the data value received in the previous phase added to an offset, or a sign-extended immediate. In Verilog, we use the Arithmetic Logic Unit and use its add operation. We also make sure that the second operand is the immediate by sending the proper control signal to the mux that controls whether the B

operand is the operand read from the Register File or the sign-extended immediate. This was actually a mistake in our CPU - originally, we did not realize that LW (and SW) needed the ADD operation on the ALU - so when testing it, without offsets it had worked, but when we had arbitrarily introduced offsets, it didn't. This is a minor error that was easily fixed, but it was still important to the functionality of the CPU.

Read from Memory

The instruction is "Load Word," and in order to do so, we use the computed address we received and index Data Memory with it. The value is read. In Verilog we implement this by sending the appropriate address to the Data Memory in order to read the proper value from Data Memory.

Write Back to Memory

The final instruction of LW is storing the value that we had loaded back into the Register File at the address Rt. This is implemented, in the schematic and through Verilog, by sending the right control signal to mux that controls what data the Data Memory reads.

Testing Individual Components

Our test plan started with individual testing of each component. We focused specifically on the ALU and made sure it behaved accordingly with the provided operands. We could not comprehensively test this component, so we often select specific modules within this one to make sure it worked correctly.

This is the output from our test file, alu.t.v:

```
operand A:      20
operand B:      5
command: 1
result:         15
```

This result makes sense because command = 1 represents the "subtract" operation.

```
operand A:      5
operand B:      25
command: 3
result:         1
```

This result makes sense because command = 3 represents the "Set Less Than" operation.

```
operand A:      25
operand B:      24
command: 4
result:         1
```

This result makes sense because command = 4 represents the “Check Not Equal” operation

Testing Individual Instructions

In addition to testing individual components and piecing them together, we tested individual instructions with a set of “simple tests”. We have a mini test for each instruction in our set.

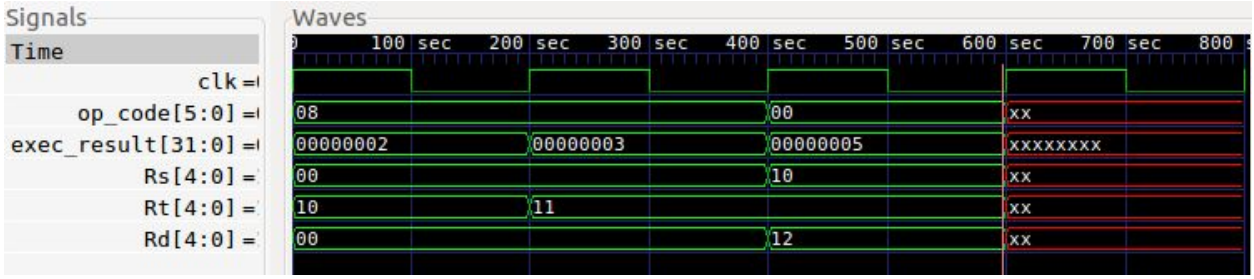
A sample instruction is `ADD`, which takes in two registers and sums the contents of those registers. For our particular ADD test, we saved the values `2` and `3`. We then add the variables with those results together and saved the sum into a third variable, s2.

ADD test:

```
addi $s0, $zero, 2
addi $s1, $zero, 3
add $s2, $s0, $s1
```

We were able to demonstrate that this test works by analyzing the the parsed outputs from our testbench. As we can see with this picture, we were able to get the right instruction outputs. If you look closely, the Rt and Rd variables are parsed correctly too. The op codes are all zero because it represents a control value that tells the ALU to “add the values”.

```
Time |          pc | instruction|        Read 1 |    Rs   | Rt      | Rd      | exec result| wb result
 200 |           0 |   537919490 |            0  | 00000 | 10000  | 00000  |          2 |          2

data stored in Rt for addi:          2
 400 |           4 |   537985027 |            0  | 00000 | 10001  | 00000  |          3 |          3

data stored in Rt for addi:          3
 600 |           8 |    34705440 |            2  | 10000 | 10001  | 10010  |          5 |          5

data stored in reg file to read:          2,          3

alu_src: 0, op_code: 000000, func: 100000

two operands into ALU:          2,          3
```



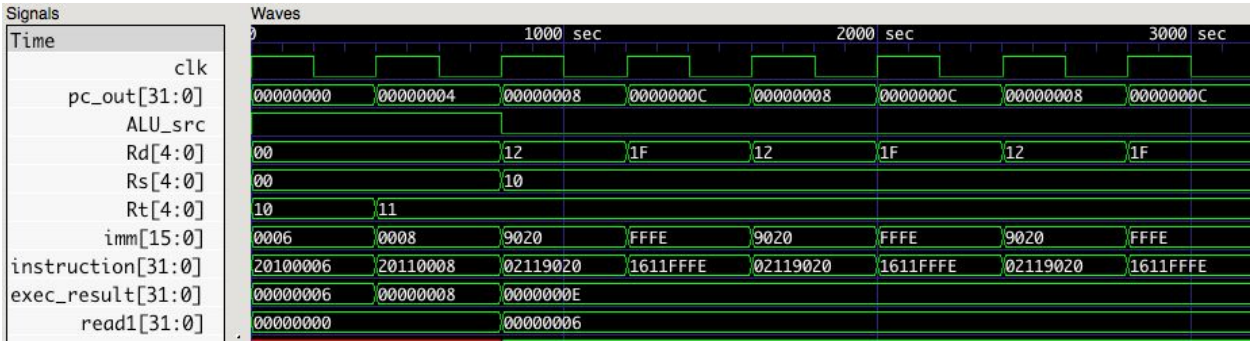
GTKwave result showing addi and add instructions working

This GTKWave output basically repeats the information shown in the table above. The op code line comes directly from the first six bits of the instruction. Since the first two instructions are type `ADDI`, we get the ADDI op code (8) for two cycles. Then we get the `ADD` op code (0) for one cycle.

We also tested `BNE`, which compares the contents of two registers and branches to another address if the contents of those registers are different. For our particular `BNE` test, we compared the contents of $s0 (contains 6) and $s1 (contains 8). We then use `BNE` to see whether the contents of those registers are equal. If they are not equal, the program jumps to the label address. If they are not equal, the program does not do anything.

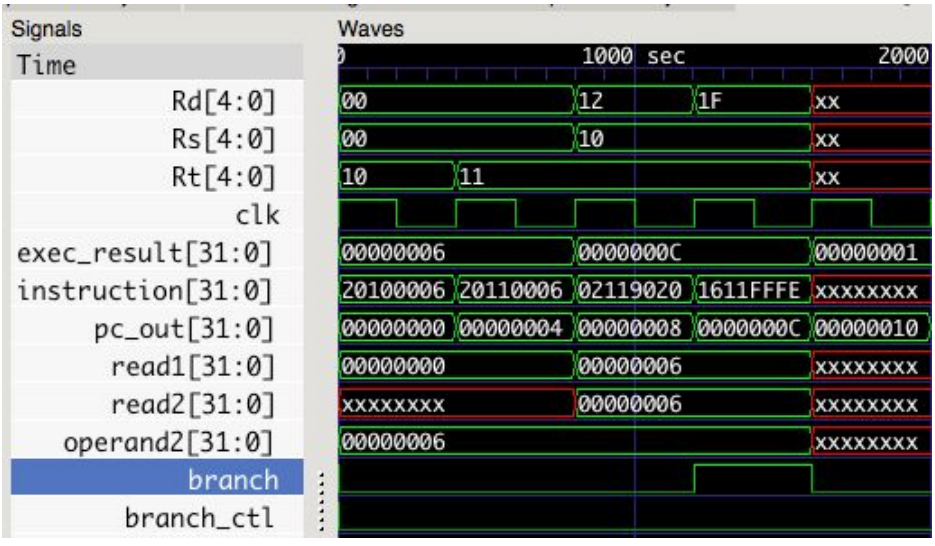
BNE test:

```
addi $s0, $zero, 6
addi $s1, $zero, 8
label: add $s2, $s0, $s1
bne $s0, $s1, label
```



As you can see from this gtkwave file that the program loops between the last two instructions, always adding $s0 and $s1 into $s2.

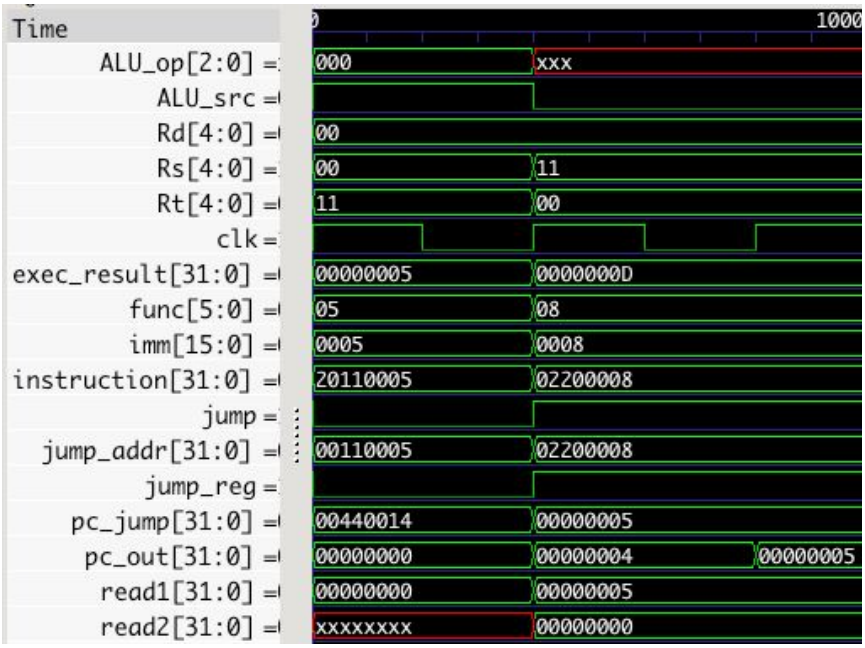
When $s0 and $s1 are equal (to 6 as shown below), the program doesn't branch and just executes once.



JR

addi $s1, $zero, 5

jr $s1

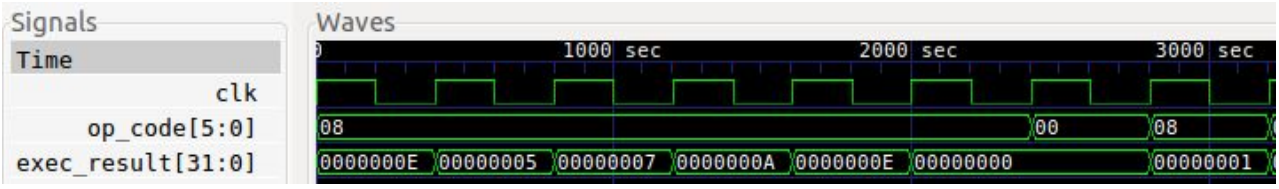


This screenshot shows PC jumping from 0, 4 to 5, the value stored in $s1. RTL: PC=R[rs]

Instructions for running each test:
The tests are in the simpleasmtest/ directory with the (instruction name).v format. Before running each test, make sure you are in the Lab3/ directory and modify instructionmemory.v with the right directory and filename in the $readmemh line.

Testing Multiple Instructions At Once

Finally, we made a few, more general, Assembly test programs that utilize multiple instructions at once. We mostly worked with one of our assembly tests: findmin. This script looks at four values 14, 5, 7, 10 and saves the minimum value (5) in assembly.



The GTKwave output was long and hard to see, so we are showing the first half in this report. This findmin.asm script starts off with initializing the arguments and compares the first argument with the second argument using a slt instruction. When you see the op_code line change to 00, you see that exec_result changes to 1 because 14, the first argument, is less than 5, the second argument. The op_code then returns back to 8 so that the current minimum can be updated to 5. This pattern continues throughout the rest of the argument set.

PERFORMANCE/AREA Analysis
In our CPU we modeled a delay of #10 per not gate, and #30 per and gate. The average delay of a signal (alu_result/exec_result) is around #40 to #50. A clock cycle of #300 is enough to accommodate all instructions.
Our single cycle CPU design is more area efficient than a multi-cycle or pipelined design because it doesn't need intermediate register to save the result of each stage into. We used adders for the branch address and pc+4 rather than ALU which also saves area.

Work Plan Reflection

Instead of going through each element of the work plan and comparing actual hours to expected hours, we're going to reflect more broadly on how everything went. This is because there were too many unknowns introduced throughout the course of the lab that it seems trivial to reflect on a work plan that simply doesn't accurately reflect what occurred.

Primarily, our main pivot was from a pipelined CPU single-cycle CPU to a single-cycle CPU. This was after the advice from two NINJAs who stated that implementing a single-cycle CPU is a fairly arduous task already - let alone pipelined with its various hazards. Thus, we moved away from pipelined and made our entire focus on the single-cycle CPU. Although we had allotted time to a single-cycle CPU and its implementation in Verilog, we had allotted far less time than it actually took - primarily because we underestimated its complexity.

Our second pivot was moving from a two person team to a three person team. Although the addition of an extra person was welcome, it added extra difficulties with meeting times and dividing up work effectively. This is especially true when trying to meet before break, when our schedules synced up fair irregularly and thus team communication was extra imperative at that time to inform others what was done.

In addition, all three of us happened to be sick at somewhat overlapping but inconvenient times. This was especially a nuisance because it meant a lot of individual work happened, and then we had limited time to come together and integrate our work. Adding extra difficulties was the fact that one team member actually had to start the lab a week and a half later - causing the need for an extension. Though it probably could have been roughly completed beforehand, the extension granted all three members to effectively contribute to the lab and also learn from its implementation - so even though it wasn't the most fun thing to do over break (and finding times to work during break where all three of us was free was also a very difficult process). As a result, though good work occurred over break (and we did heed the caveat for our extension - shift work hours, don't work more hours) - it was less effective than actually meeting each other. However, luckily with our extension, we were able to finish in a timely fashion.

Overall, though, our scoping for the overall implementation of the CPU - in terms of overall hours, was correct. However, that is combining the single-cycle and pipelined CPU. With this in mind, we underscoped for the single-cycle CPU, but appropriately scoped for the entire CPU.