CSCE 312 Lab 6 - Final Project
Y86 CPU implementation

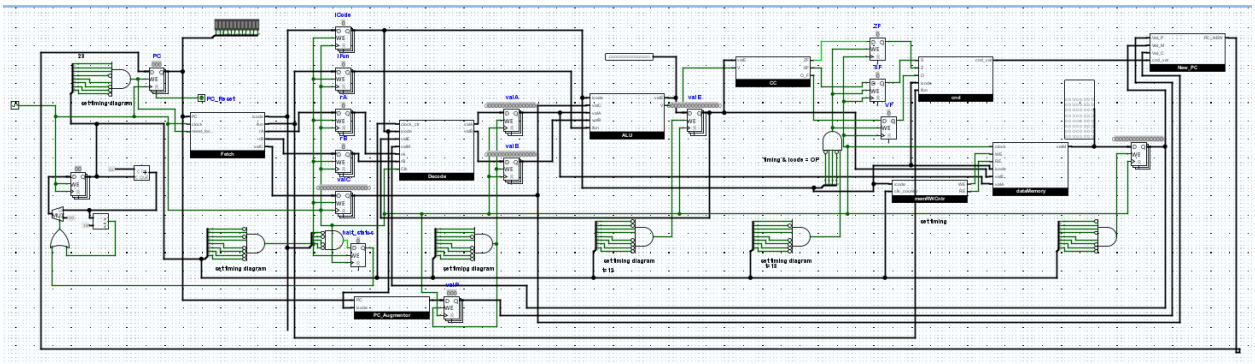
Krish Sareen
Alexandra Saxton
Caroline Jia

Table of Contents

Introduction

In this lab, we developed a fully functional single stage CPU, which can run machine code and produce accurate output in memory. It uses every digital logic and computer architecture concept we have learned so far in this class, and its creation / use will be broken down in this report.

Full CPU Implementation



Test Programs (Caroline)

In the computer organization course, we have learned several kinds of assembly languages and instruction set architectures. For this project, our application is a Y86 processor with 64 bit registers. This means that the assembly in the programs and the logic we implement to complete each instruction will keep this in mind, using well-defined machine code to instruction translations. Within the architecture, there are several kinds of commands. Of these, we used many to test our processor. In order to create this testing code, we wrote instructions in the y86 language, translating them later into files using yo2mem that can be used directly with the processor to run instructions.

Assembly Code

```
main:
        irmovq $1, %r9 #sets r9 to 1
        irmovq $1, %rsi #sets rsi to 1
        rmmovq %rsi, (%r11) #moves the value of one into the memory of r11
        mrmovq (%r11), %rdx #takes the memory value of r11 and stores it into rdx(1)
        irmovq $2, %rdi #sets rdi to 2
        subq    %rdi, %rdx #subtracts 2 from 1, sets SF to 1
        jle     part2
        irmovq $3, %rsi #if SF not set will set rsi to 3
        addq    %r9, %rdi #adds 3 to rdi, should be 5
        halt

part2:  irmovq $5, %r10 #sets r10 to 5
        addq    %r10, %rsi #adds 5 to rsi
        subq    %r9, %rdi #subtracts 1 from rdi
        halt
```

Binary/Object File

```
1
2   0x000:                      | main:
3   0x000: 30f90100000000000000 |         irmovq $1, %r9
4   0x00a: 30f60100000000000000 |         irmovq $1, %rsi
5   0x014: 406b000000000000000  |         rmmovq %rsi, (%r11)
6   0x01e: 502b000000000000000  |         mrmovq (%r11), %rdx
7   0x028: 30f70200000000000000 |         irmovq $2, %rdi
8   0x032: 6172                 |         subq   %rdi, %rdx
9   0x034: 714a00000000000000   |         jle    part2
10  0x03d: 30f60300000000000000 |         irmovq $3, %rsi
11  0x047: 6097                 |         addq   %r9, %rdi
12  0x049: 00                   |         halt
13                              |
14  0x04a: 30fa0500000000000000 | part2:     irmovq $5, %r10
15  0x054: 60a6                 |         addq   %r10, %rsi
16  0x056: 6197                 |         subq   %r9, %rdi
17  0x058: 00                   |         halt
```

Test Instructions used in each test

<table>
<tr><td>Test 1:</td><td rowspan="2">Extra Test (4):</td></tr>
<tr><td>irmovq, rrmovq
addq, subq
jne, je, jg, jle, jmp
pushq, popq
nop, halt</td></tr>
<tr><td>Test 2:</td><td rowspan="4">Instructions used multiple times
irmovq, rmmovq, mrmovq, xorq, andq, subq, halt</td></tr>
<tr><td>irmovq, rmmovq, mrmovq
subq, addq
jle
halt</td></tr>
<tr><td>Test 3:</td></tr>
<tr><td>irmovq
subq, addq, xorq
jl, jge
halt</td></tr>
</table>

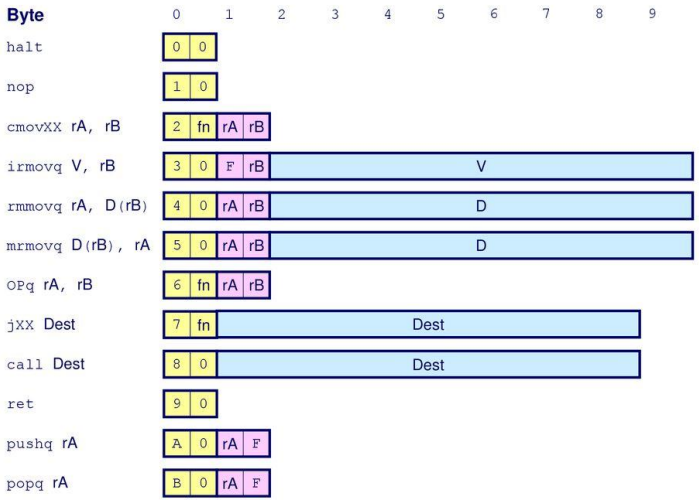
Processor Design

As a continuation of the projects we have already done in lab, we began to develop the CPU with y86 rules in logisim. To build it, large stages such as the fetch, decode, execute, memory, and PC update were implemented as blocks for easy combination in the main circuit file. This helped greatly with abstraction and dependencies. We were easily able to see the big picture of the circuit without seeing each individual gate and system level application. This helped the most in the decode stage, which contains the system register file. This was a massive task, as it had to be edited several times after being added to the circuit, as we added and changed functional structure.

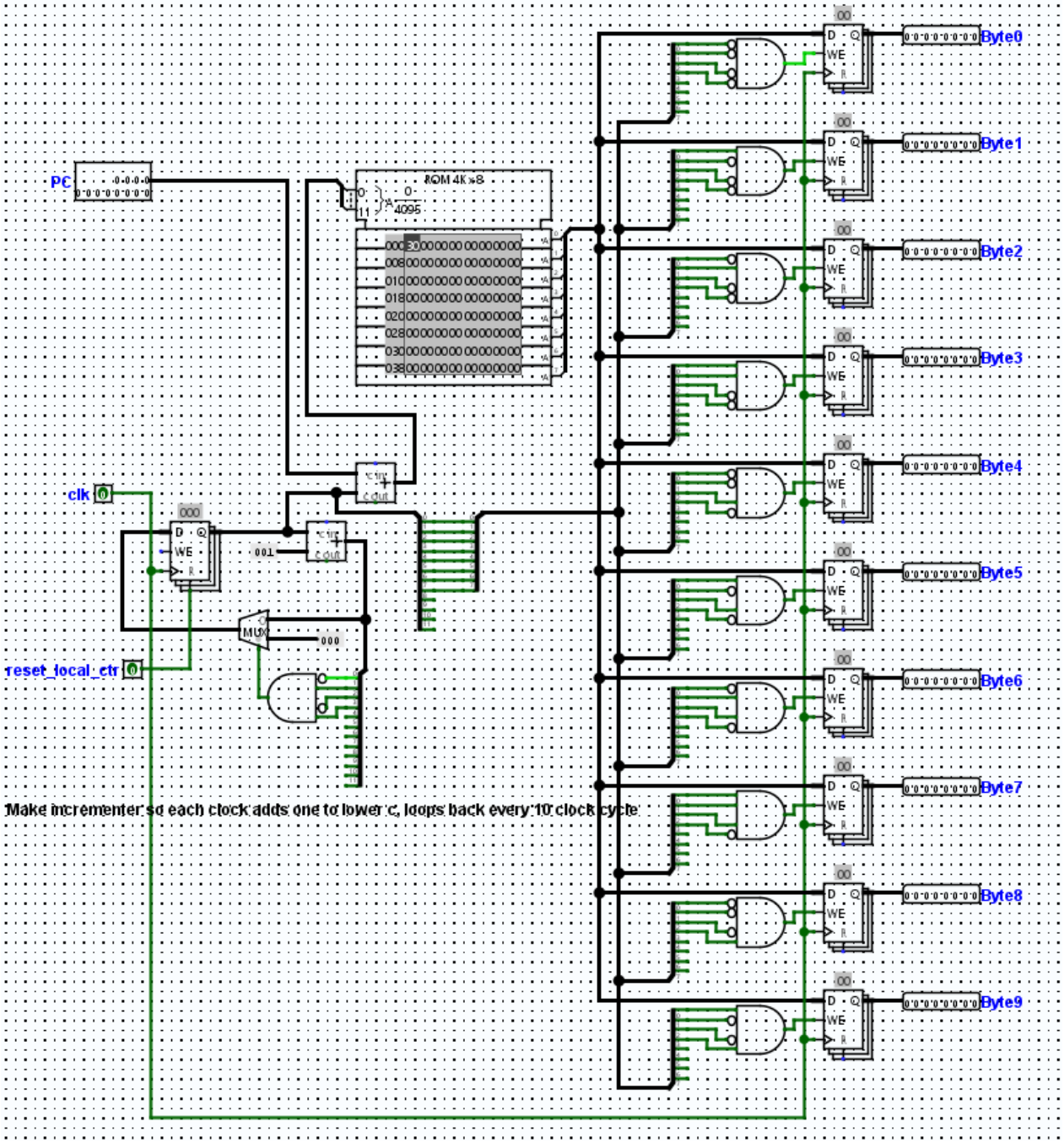
Fetch Stage (Alexandra)

In the fetch stage of any processor architecture, the instructions are read from memory and separated into several smaller pieces, which represent the coded instructions for the processor to compute. These include up to two register addresses, a constant value, and two sets of instruction code, which make up for up to 10 Bytes in this architecture. They are broken up into icode (4-bit), ifun (4-bit), rA (4-bit), rB (4-bit), and V (8 - Byte). The breakup provides the system with the ability to choose between 16 registers (4 bits x 2 registers), an instruction within y86 (8 bits), and any operation with a constant up to 8 Bytes in size. This provides the backbone for the fetch stage. One challenge with reading the ROM was that only 1 byte of memory is able to be read at a time, but the entire command takes up to 10 bytes. To solve this problem, we created a displacement register within the instruction memory circuit that counts the number of clock cycles since the start of the fetch phase. The address wired into the ROM is the sum of the PC counter and the displacement so that over the course of 10 clock cycles, the 10 bytes of ROM are read. After 10 bytes of memory are read, logic based on the specific instruction (iCode) is used to assign rA, rB, and valC with the remaining 9 bytes. Oftentimes not all 10 bytes are used, in which case the remaining bytes are simply ignored.

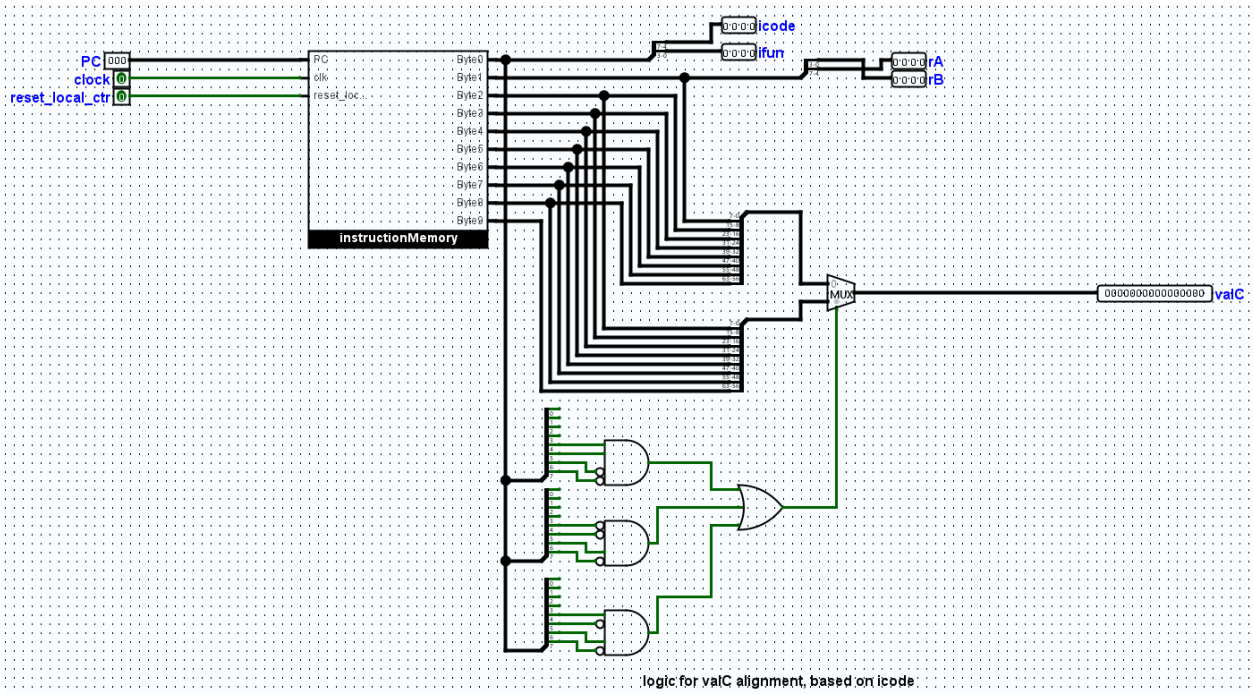
Y86 Instruction Architecture



Instruction Memory Access

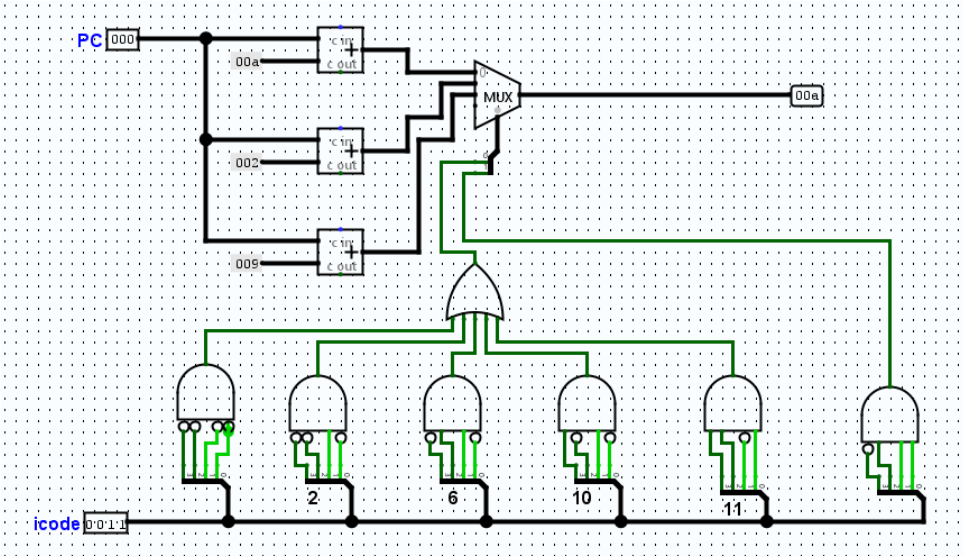


Fetch Logic



In addition to this, the location of the program counter, which reads the current instruction spot in memory, is augmented to proceed to the very next instruction (line of code), which is represented by valP. This also happens during the fetch stage. To implement this, we use three adders to add 2, 9, and 10 to the PC (since instructions can be either 2, 9, or 10 bytes long). Then, a MUX selects the correct augmented value to output using logic based on iCode.

PC Augmenter

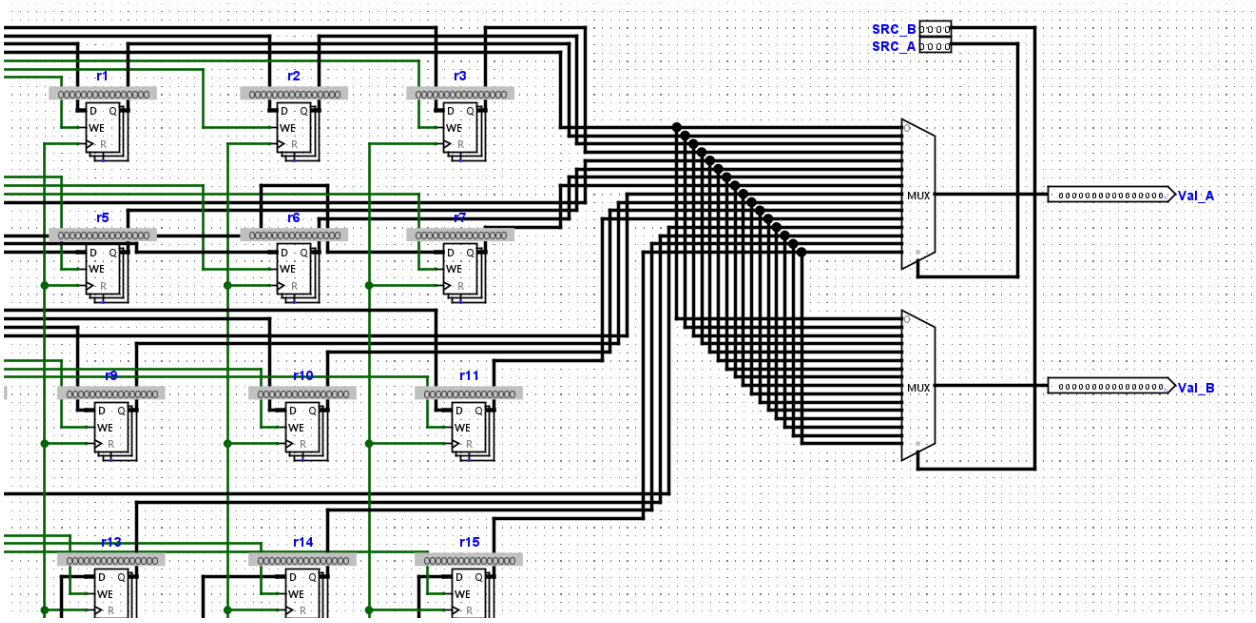


Decode Stage (Krish)

In the decode stage, the values that were previously used for register implementation (R/W) come into play. The file contains all 16 registers, including the stack pointer (RSP). Each of these registers holds 64 bits of data, hence the architecture size of 64 bits. They serve as the computational backbone of the CPU, and allow it to store data with which to make calculations.

Values rA and rB, which are 4 bits each, serve as select lines for the register file. At the decode stage of each instruction, both register addresses are used as selectors for 2 16x1 MUXs which allow reading of two registers simultaneously. At the end of this stage, there are two additional values, ValA and ValB, which are the values stored in both of the registers. These buses output to the ALU, memory, and other locations.

Register File Read Ports

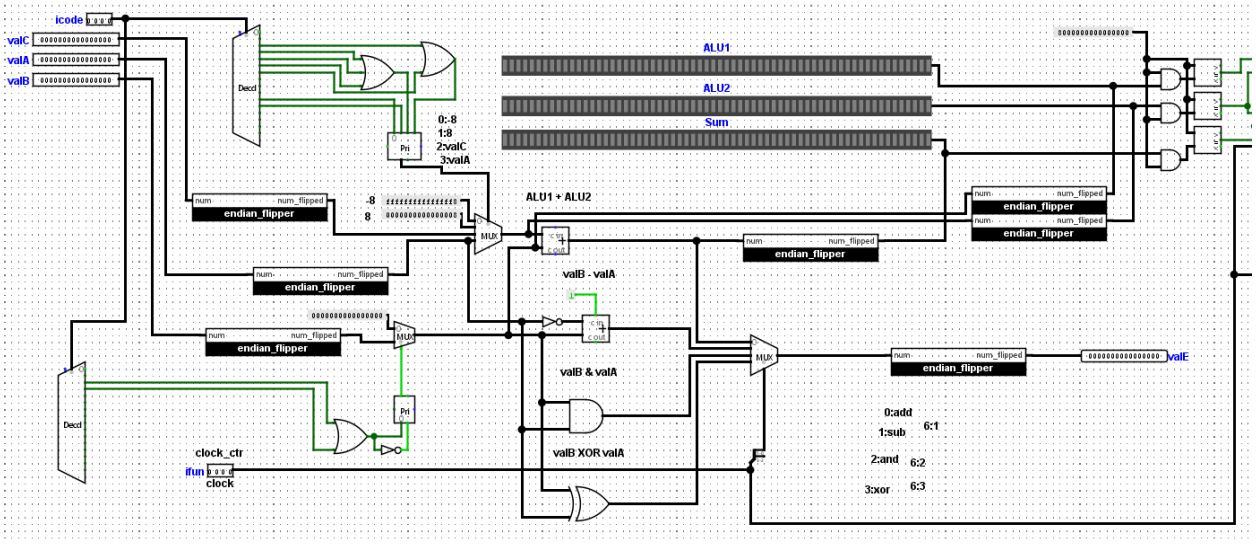


Execution Stage (Alexandra + Krish)

Now that all relevant information has been retrieved for computation, this stage deals with the ALU and its operations. In any addressing, math, or comparisons, this stage is vital to return the correct effective address or result (ValE). This is the value that is written back to write port E of the register file, and can be the result of many operations done by the ALU. Another significant portion of this stage is the inputs of the ALU. They are fed by multiplexors that contain select lines derived directly from icode and ifun, the instruction code. Using that information, these multiplexors select from ValA /ValB, hardcoded values of -8 and 8, and more. This allows for easy operations when incrementing and decrementing the stack, for example.

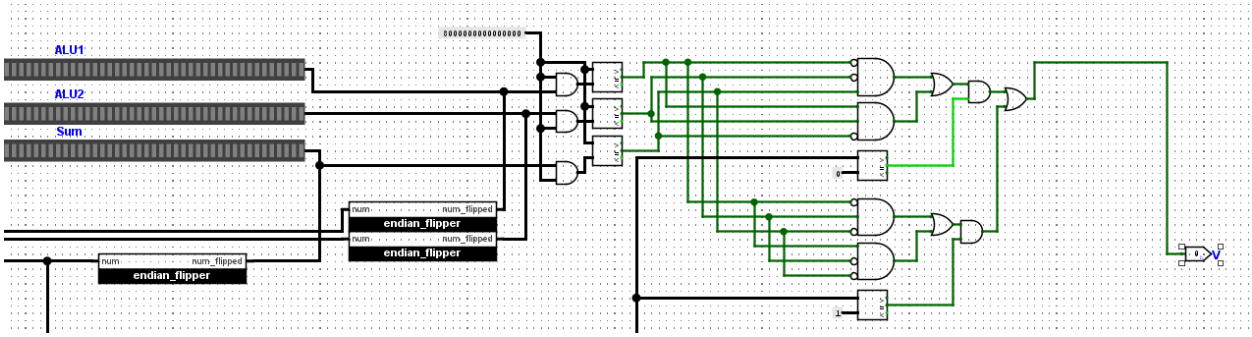
When an ALU operation requires adding a constant value to a register (memory offset, incrementing, storage), valC is input directly into the ALU.

ALU Implementation (Alexandra)

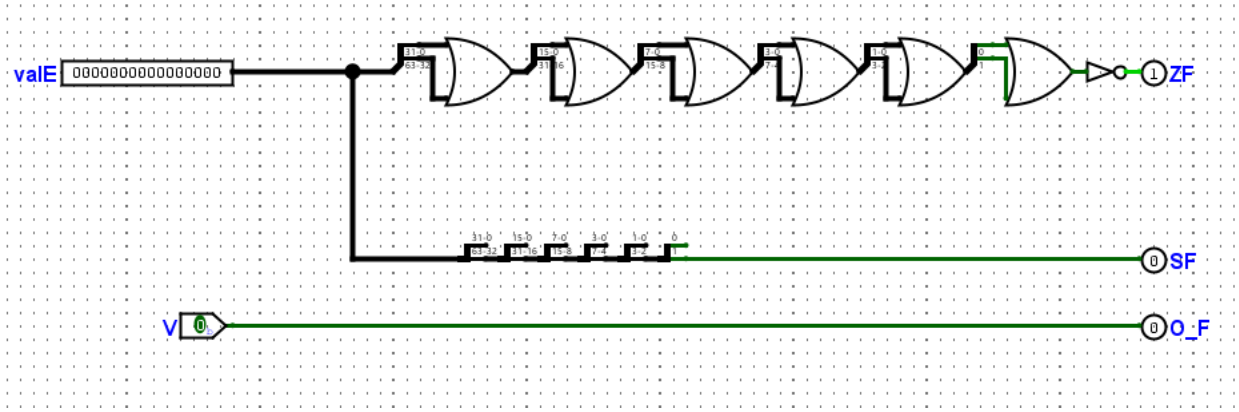


Equally important to ALU operations is the use of the condition code flags. These take the output of the ALU and determine if 3 things have happened: ZF determines if the operation resulted in a 0, and SF determines if an operation has resulted in a signed number (negative). Finally, OF determines if an operation has resulted in overflow. Using these flags, we can determine several conditions, which can be used later on to jump. The condition code flags are single bits, and are set / reset every operation, so they can be used in the next instruction to change the PC based on conditions. For example, the instruction jl will check if SF has been set, without overflow being present. This means that the previous operation resulted in a negative number, and this number is fully captured in the processor without overflow. This can be used when comparing two numbers. If 6 subtracts 17, for example, the CC would set SF, allowing for a successive instruction (jl) to jump the PC to another location given a specific thing has occurred. These conditions serve as the basic architecture for if/else and switching, which becomes the system level logic of the entire computer. At the end of this stage, there are four new values: Val E, SF, OF, and ZF. It should be noted that to change the memory storage method from Little Endian to Big Endian, we developed a basic device that flipped the orders of each byte in a 64 bit input to produce a flipped output. This is seen in several places within the ALU. The logic implementation for the overflow flag is shown to the right of the ALU logic, and will be elaborated on later.

Condition Codes (Krish + Alexandra)

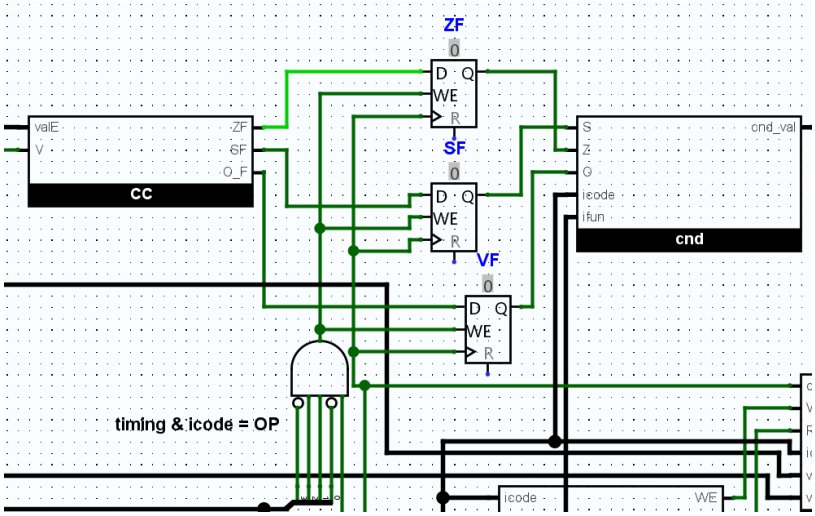


This is the logic that is used to determine overflow in mathematical operations in the processor. It takes ALU multiplexor inputs to check for signed numbers in inputs and output of the ALU (Val E), comparing them and the operation code (ifun).

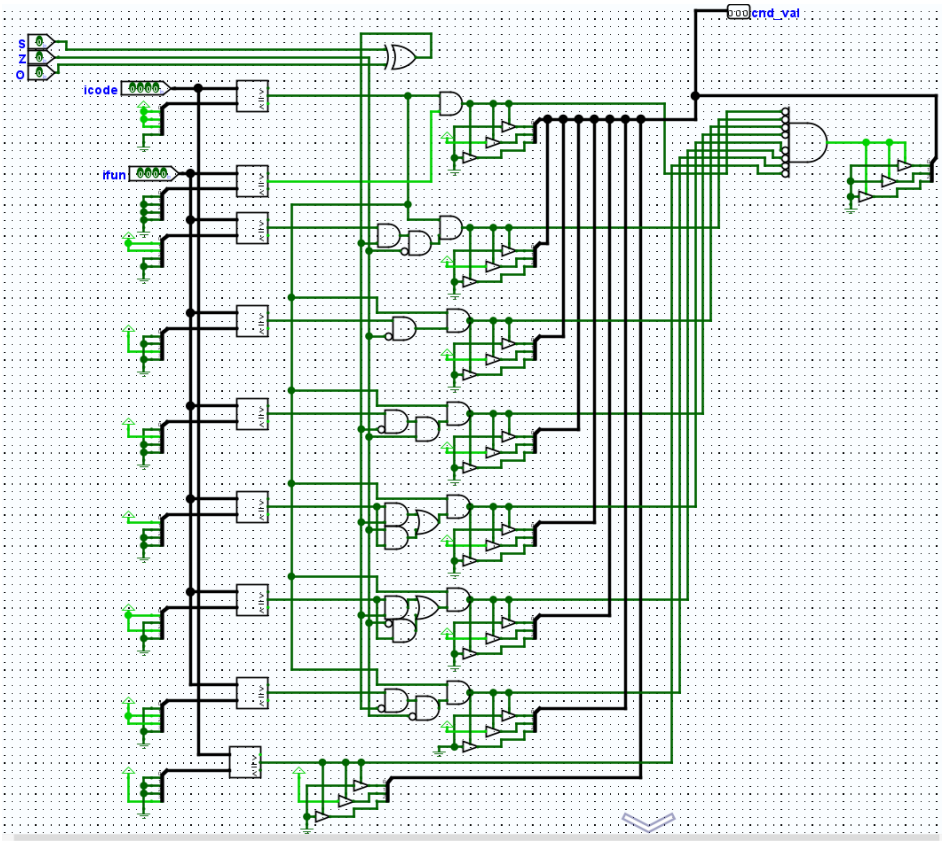


This is the implementation of two other flags, the signed flag and zero flag. Both are simply determined using the output of the ALU, with one checking the MSB bit of the output, and the other checking for a zero.

The CC logic circuits shown above reflect the cnd module, which determines if a particular condition in jxx or cmovxx is satisfied.



Cnd (Krish)

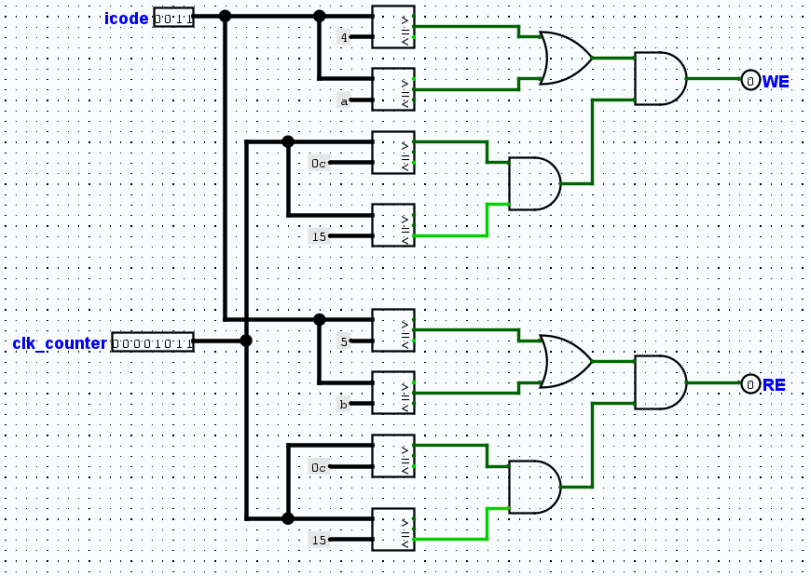


In this circuit, which outputs select bits for the PC update, the opcode byte (icode/ifun) is used to determine if the program should branch via conditional statements. It also decides when the program will jump to Val M or Val P, which are instruction labels from memory or the next possible instruction in memory. To determine the next instruction, CND uses the flags (OF, ZF, and SF) to determine if conditional statements are valid. For example, in the jne instruction, CND will check if icode is the right 4-bit value (7), and then if ifun is the correct value (4). At this stage, if ZF is 0, meaning the previous operation did not result in a 0, the logic in CND will enable a 3-bit select signal at output. This will select Val M at the new_PC output, resetting PC.

Memory Stage (Alexandra)

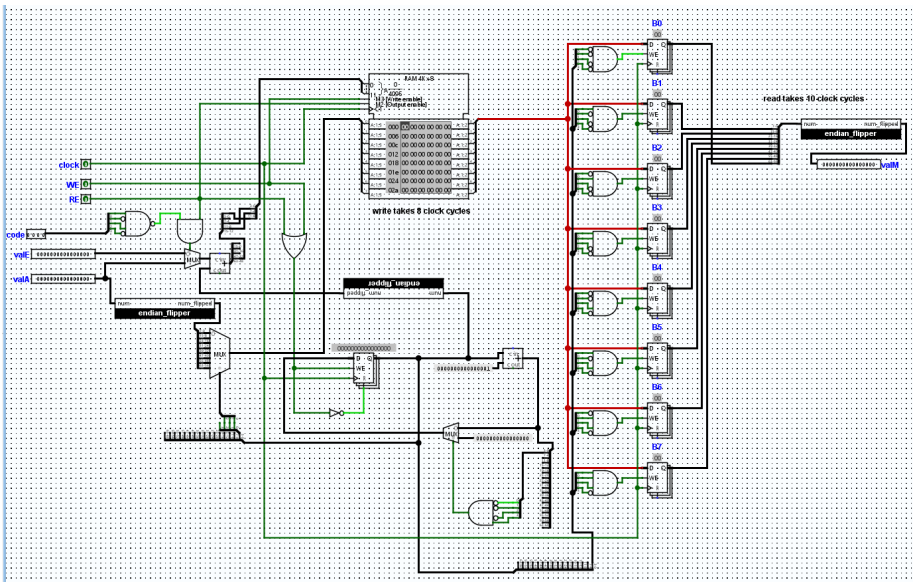
In system memory, a RAM module is located which can allow for memory storage. The stack, for which RSP remains a pointer, is located at the “top” of the stack (0xfff). For loading values into memory, or reading them, this stage is critical. The CPU will either read from the memory using the memory offsets and effective addresses determined during the execution stage (valE), or it will write to the memory using valA. For simplification, the controller for memory reading/writing is located in a separate circuit, and considers the iCode and the clock counter to determine if and when to enable reading and writing.

R/W controller



Since we are reading or writing 8 bytes of data and only are able to read 1 byte at a time, we must use a local clock counter to augment either valE or valA (whichever address is used, selected using icode). When reading data, the local clock counter is also used to enable writing to each of the 8 1-byte registers, which are used to store each byte of the memory being read. After all data is read, it is combined and outputted to valM. Likewise, when writing, valA is written byte by bite into the RAM using the local clock counter.

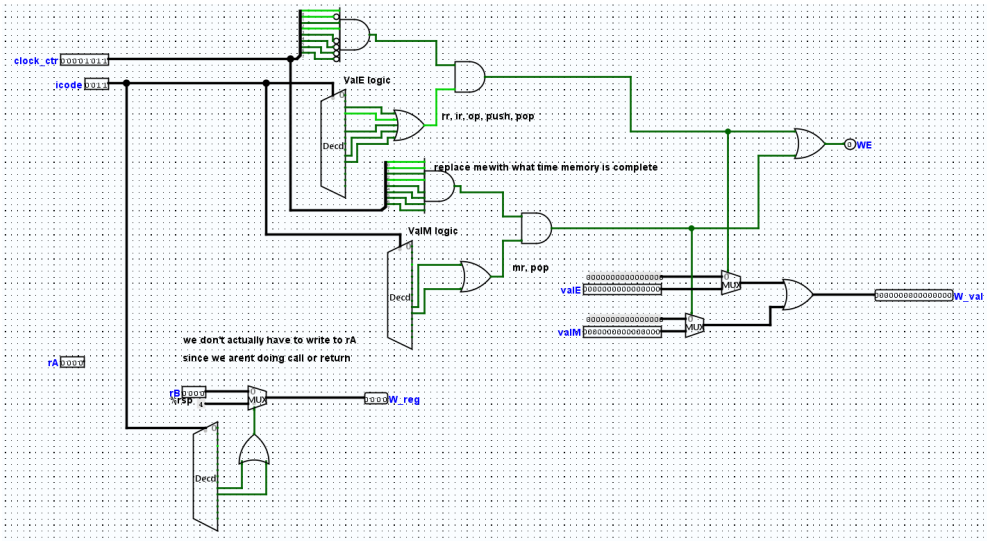
Memory Implementation



Write-Back Stage (Alexanrda)

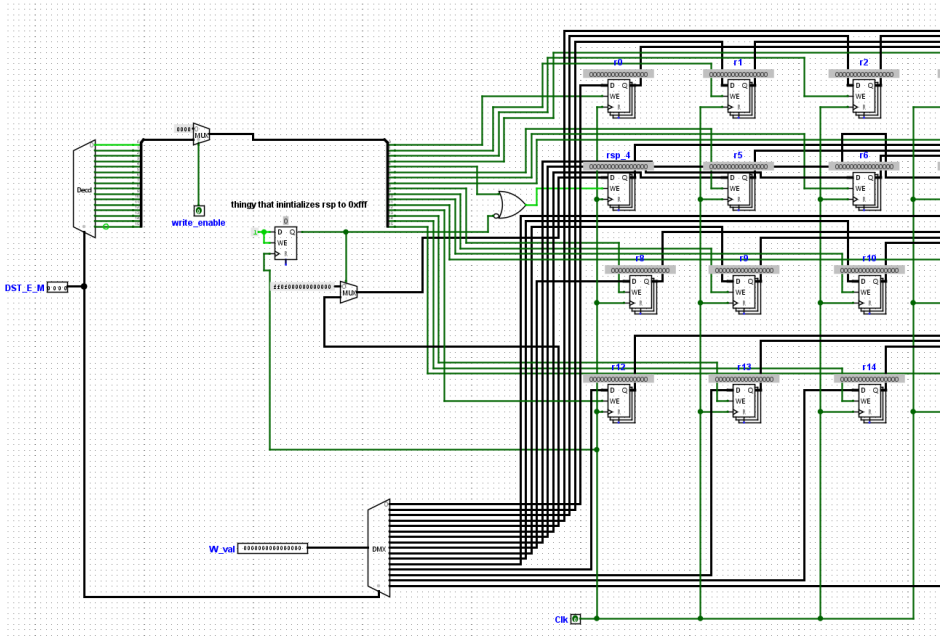
To assist with register writing, the logic has been moved into a circuit separate from the register file. This file outputs a write enabler which enables register writing depending on icode and the instruction's clock counter. The write controller also selects either valE or valM to be written, depending on if the instruction is writing a value from the execution stage or from the memory. The final purpose of the write controller is to determine the register address of the register being written to. In most cases the address is rB (rA is only used in call and return which are not implemented), however pushq and popq use %rsp instead. A multiplexer is used to select between rB and %rsp (address 0100), and uses icode to make this decision.

Write Controller



Below is the actual register file. The writing process considers the register address, the write enable, and the write value, which all are provided by the write controller. A demultiplexer is used to provide the correct register with the value being written.Additionally, a decoder is used to deliver the correct register the write enable.

Register file Write Ports

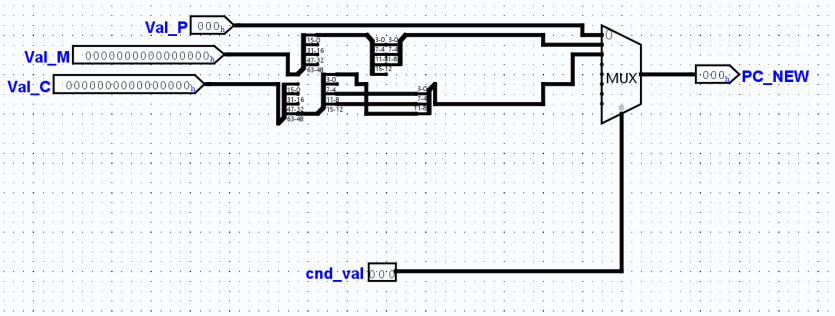


Program Counter (Krish)

This block allows for the processor to continue reading line after line of the instructions, as well as jump from line to line based on conditional statements. It can also read a location from memory to jump to, which is used during the call / ret instructions. This allows for the entire system to reset and begin working on the next instruction in the exact spot it is told to go to.

Inputs for this stage are valM (memory value only if we were to implement call and return), valC, valP valB, and the cnd_val from the execute stage. If the instruction is not a jump instruction, the PC updater simply outputs valP as the new PC. Otherwise, the PC update makes a decision using the instruction function (i.e. jle, je, etc) and cnd set during the execution stage. If a function is met, the new PC biomes valC.

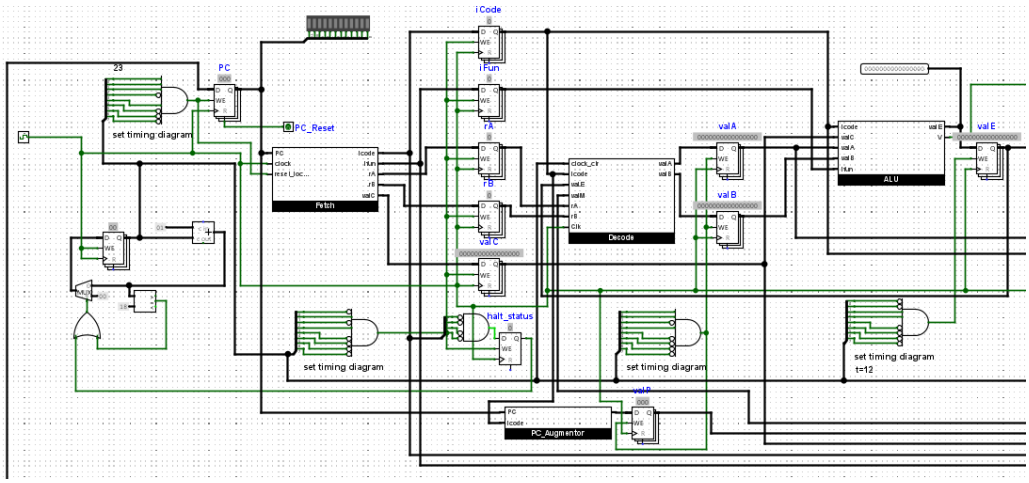
Program Counter

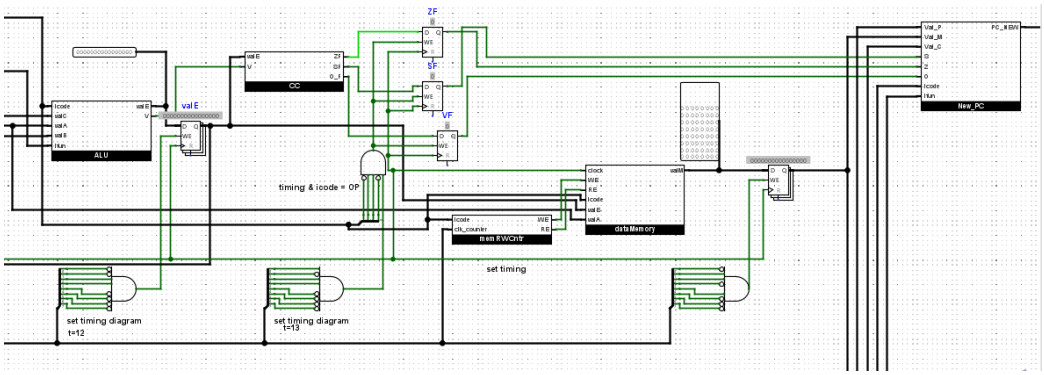


Timing and Sequencing

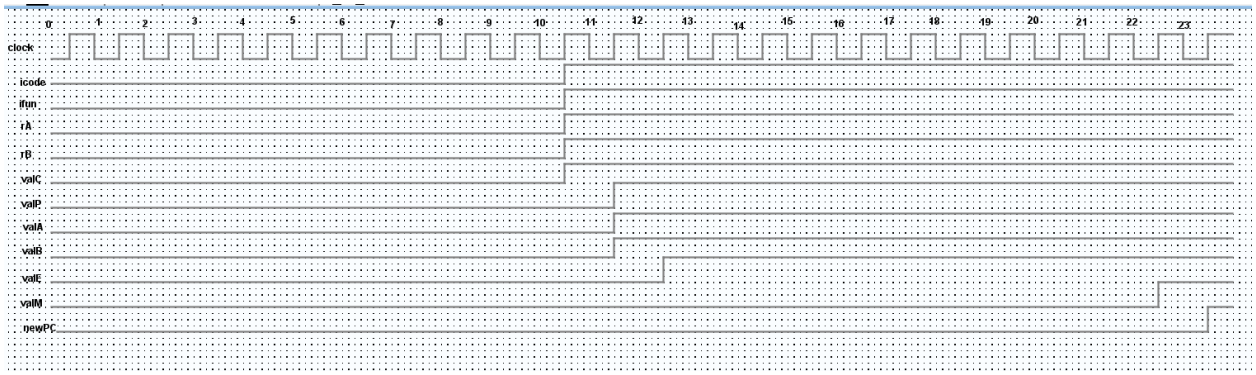
In this single stage processor architecture, each instruction must end before another begins. To time this process effectively, there is an additional counter that runs in tandem with the clock, only enabling the registers of each stage when their respective time comes. These are determined by the sequential organization of the processor. Each instruction will take a certain amount of clock cycles. The following diagram is our timing diagram. We recommend zooming in. When the clock's line is low, the clock is 0. When the clock line is high, the clock is 1. When a value's line is low, it has not been updated. When a value's line is high, it has been updated.

Sequencing Control





Timing Diagram



Input Selection Tables

Decode

instruction	srcA (rA/0x4/0xf)	srcB (rB/0x4/0xf)	dstE (rB/0x4/0xf)	dstM (rA/0xf)
halt	0xf	0xf	0xf	0xf
nop	0xf	0xf	0xf	0xf
rrmovq rA, rB	rA	0xf	rB	0xf
irmovq V, rB	0xf	0xf	rB	0xf
rmmovq rA, D(rB)	rA	rB	0xf	0xf
mrmovq D(rB), rA	0xf	rB	0xf	rA
OPq rA, rB	rA	rB	rB	0xf
jXX Dest	0xf	0xf	0xf	0xf
cmovXX rA, rB	N/A	N/A	N/A	N/A
call Dest	N/A	N/A	N/A	N/A
ret	N/A	N/A	N/A	N/A
pushq rA	rA	0x4	0x4	0xf
popq rA	0x4	0x4	0x4	rA

Execute

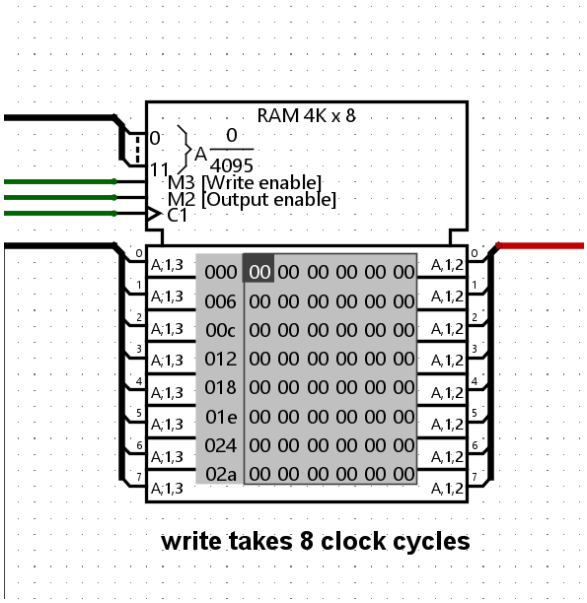
instruction	ALU_A (valA/valC/-8/8/X)	ALU_B (valB/0/X)	alufun (0/1/2/3)	set_cc (0/1)	valE (op1 OP op2)	CC (ZF/SF/OF)	Cnd (0/1)
halt	X	X	X	X	X	X	X
nop	X	X	X	X	X	X	X
rrmovq rA, rB	valA	0	0	0	0 + valA	0	0
irmovq V, rB	valC	0	0	0	valC + 0	0	0
rmmovq rA, D(rB)	valC	valB	0	0	valC + valB	0	0
mrmovq D(rB), rA	valC	valB	0	0	valC + valB	0	0
OPq rA, rB	valA	valB	depends on ifun	1	valA OP valB	0	0
jXX Dest	X	X	X	0	X	1	1
cmovXX rA, rB	valA	0	0	0	valA + 0	1	1
call Dest	-8	valB	0	0	-8 + valB	0	0
ret	8	valB	0	0	valB + 8	0	0
pushq rA	-8	valB	0	0	-8 + valB	0	0
popq rA	8	valB	0	0	8 + valB	0	0

Memory

instruction	Mem. read (1: read)	Mem. write (1: write)	Mem. addr (valE/valA)	Mem. Data (valA/valP)
halt	X	X	X	X
nop	X	X	X	X
rrmovq rA, rB	X	X	X	X
irmovq V, rB	X	X	X	X
rmmovq rA, D(rB)	X	1	Val E	Val A
mrmovq D(rB), rA	1	X	Val E	X
OPq rA, rB	X	X	X	X
jXX Dest	X	X	X	X
cmovXX rA, rB	X	X	X	X
call Dest	X	1	Val E	Val P
ret	1	X	Val A	X
pushq rA	X	1	Val E	Val A
popq rA	1	X	Val E	Val A

Design Verification

To generate the memory files necessary for Logisim, we wrote the y86 assembly code (as seen in the test programs section of the report). This was then compiled into object files using .yas. Using the given yo2mem python code with the object files, we produced memory files which could be loaded directly into ROM inside our instruction memory. Using the loaded instructions, we were able to demonstrate that our CPU functions properly.



In the ROM modules for each of the programs, we loaded saved files that contained compiled versions of our y86 assembly code. After running the code, the internal registers of the CPU yielded these tables:

Lab6_1	Lab6_2	Lab6_3
%rax 0x0000000000000005 5 %rcx 0x0000000000000000 0 %rdx 0x0000000000000005 5 %rbx 0x0000000000000000 0 %rsp 0x0000000000000fff 4095 %rbp 0x0000000000000000 0 %rsi 0x000000000000000c 12 %rdi 0x000000000000000b 11 %r8 0x0000000000000000 0 %r9 0x0000000000000001 1 %r10 0x0000000000000001 1	%rax 0x0000000000000000 0 %rcx 0x0000000000000000 0 %rdx 0xffffffffffffffff -1 %rbx 0x0000000000000000 0 %rsp 0x0000000000000100 256 %rbp 0x0000000000000100 256 %rsi 0x0000000000000006 6 %rdi 0x0000000000000001 1 %r8 0x0000000000000000 0 %r9 0x0000000000000001 1 %r10 0x0000000000000005 5	%rax 0x0000000000000000 0 %rcx 0x0000000000000000 0 %rdx 0x0000000000000000 0 %rbx 0x0000000000000000 0 %rsp 0x0000000000000100 256 %rbp 0x0000000000000100 256 %rsi 0x000000000000000d 13 %rdi 0x0000000000000000 0 %r8 0x0000000000000000 0 %r9 0x0000000000000000 0 %r10 0x0000000000000000 0

Lab6_4

REGISTERS

%rax	0x0000000000005554	21844
%rcx	0x0000000000005555	21845
%rdx	0x0000000000001234	4660
%rbx	0x000000000000ffff	65535
%rsp	0x0000000000000000	0
%rbp	0x0000000000005555	21845
%rsi	0x0000000000000000	0
%rdi	0x0000000000000000	0
%r8	0x0000000000000321	801
%r9	0x0000000000000000	0
%r10	0x0000000000000000	0

Conclusion

In this class, we have gone from a basic understanding of logic and digital systems to a complete design of a Y86-64 processing unit. This has taken us from logic gates to combinational logic circuits, to sequential logic and memory, and finally to larger units such as adders, registers, and more. We've taken this knowledge and applied it to so many kinds of circuits, each of which has been abstracted upon layer by layer to form one massive and complex system. We learned how to create logic on the gate level, and with this CPU, there are millions of gates within the entire system. With this project, we've learned how best to divide up tasks from stages of the processing architecture to smaller blocks such as the register file and ALU. This has allowed us to work efficiently to finish each smaller piece whilst simultaneously having a top-down visual approach in the main circuit, where each stage came together. As we tested instructions, we found the processor worked exactly as it should, with every part of the architecture fulfilling its design specifications, including some additional functionality that was not required by the assignment. It was designed with improvement in mind. Each portion of the system has space for improvement, whether that be the addition of logic for instructions, more presets, or additional functionality such as stack pointer reset.

As we completed each stage, we tested it by referencing our classwork and notes, finding that quiz 4 proved especially helpful in designing each portion of the CPU. It helped us to better understand the top-down approach to the processor, which was important, as most of the smaller pieces like the ALU and register files had been designed before. It was more about how everything would come together, which values would need to be fed into which parts of the stages, and what additional logic would go inside each unit to make the system work. This was especially true in the PC update stage and register file, which were updated several times with references and group work by the entire team so that they functioned correctly. Overall, this project was a great success in teaching computer architecture and helping us learn by creating a practical result of our content in the course. Going forward, this will be a great reference point and learning experience for future courses in the field.