CSCE 312 Lab 6 - Term Project

Rita Hernandez Guerrero, Kevin Lei

May 2, 2024

1 Y86 Instruction Set Architecture

The circuit for the Y86 Instruction Set Architecture was implemented through Logisim. The circuit has five main subcircuits: the Fetch stage, the Decode and Write Back stage, the Execute stage, the Memory stage, and the PC Update stage.

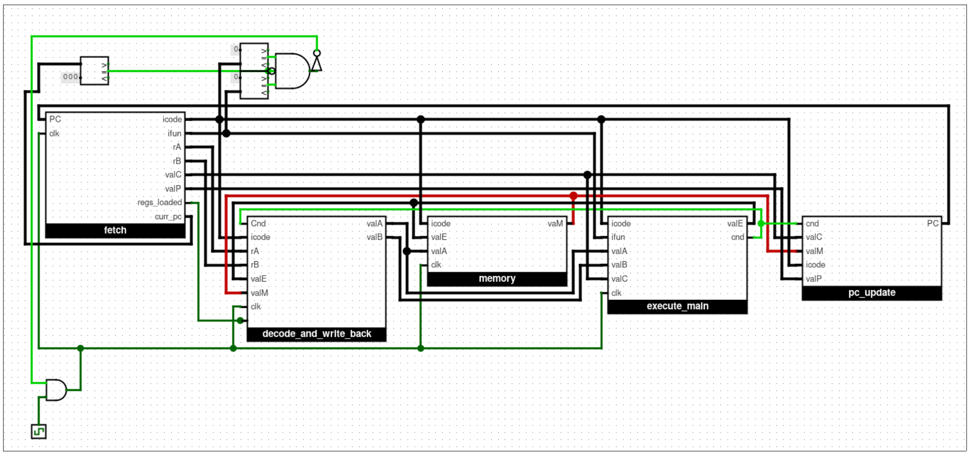
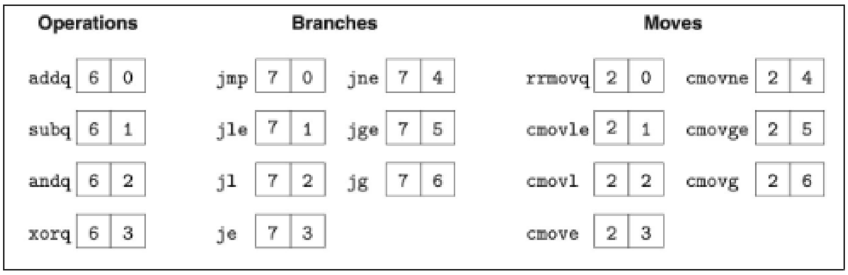


Figure 1: Main circuit for the Y86 ISA

Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0 0									
`nop`	1 0									
`rrmovq` **rA, rB**	2 0	rA rB								
`irmovq` **V, rB**	3 0	F rB	V							
`rmmovq` **rA, D(rB)**	4 0	rA rB	D							
`mrmovq` **D(rB), rA**	5 0	rA rB	D							
`OPq` **rA, rB**	6 fn	rA rB								
`jXX` **Dest**	7 fn	Dest								
`cmovXX` **rA, rB**	2 fn	rA rB								
`call` **Dest**	8 0	Dest								
`ret`	9 0									
`pushq` **rA**	A 0	rA F								
`popq` **rA**	B 0	rA F								

Figure 2: Instruction format



Operations			Branches						Moves					
addq	6	0	jmp	7	0	jne	7	4	rrmovq	2	0	cmovne	2	4
subq	6	1	jle	7	1	jge	7	5	cmovle	2	1	cmovge	2	5
andq	6	2	jl	7	2	jg	7	6	cmovl	2	2	cmovg	2	6
xorq	6	3	je	7	3				cmove	2	3			

Figure 3: Function specific instructions

1.1 Fetch Stage

In the Fetch stage, the instruction memory is implemented. The Fetch stage reads 10 bytes from memory at a time and the address of the first byte is stored in the PC (program counter). Then the stage produces meaningful data and control signals from the instruction for later stages. The Split module divides instruction byte into icode and ifun. The Align module gets fields for rA (register A), rB (register B), and valC. valC is the value constant which represents the immediate data that may be used for certain instructions (loading into a register or calculating in an arithmetic operation).

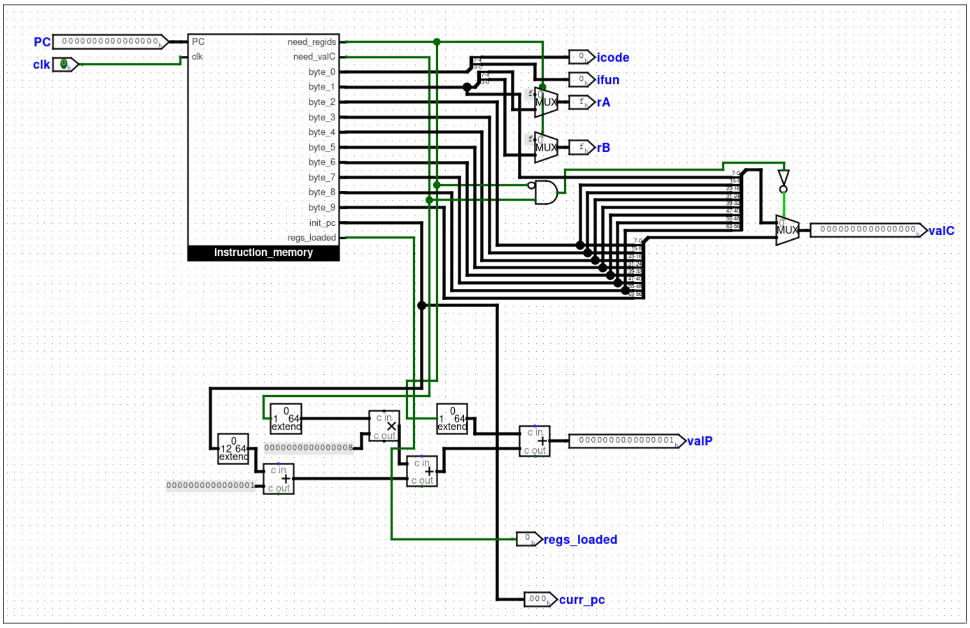


Figure 4: Fetch stage

Within the instruction memory, a 4kx8 RAM is used to read the instruction memory. The total size is 4KB because of the 12-bit address width and the 8-bit of data bit width that are both read. The value from the RAM is then broken up into 10 different register files, each register file representing a byte of the instruction memory line. A counter is set up to make sure that each line of the instruction memory is read.

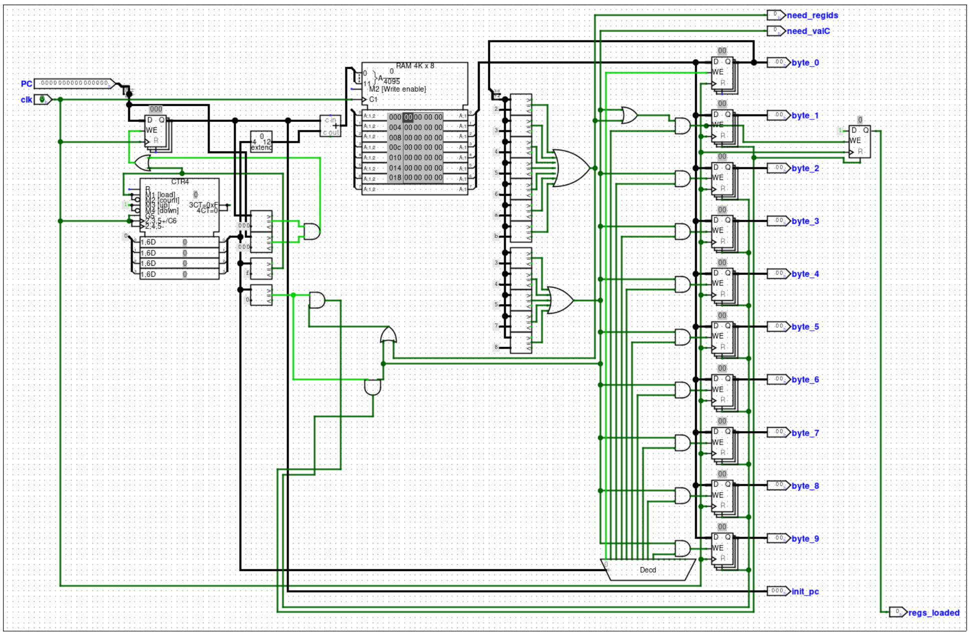


Figure 5: Instruction memory

1.2 Decode and Write Back Stage

Data is read from register files in the decode stage. The register file has two ports for reads. One being srcA and the other being represented as srcB. dstE, dstM are the outputs and are the write port addresses. Register ids are selected by the instruction code (icode). If the fetched instruction does not have to read/write register, a default value of 0xF is selected. The module also takes into account the Cnd value which indicates whether or not a conditional move is needed. The Cnd value is computed from the Execute stage.

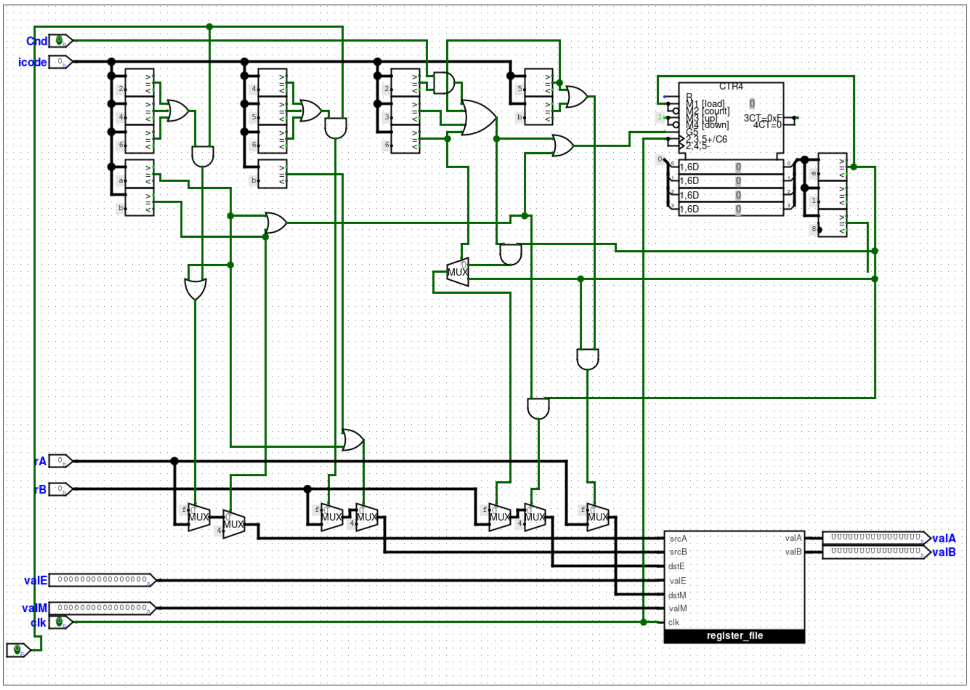


Figure 6: Decode and Write Back stage

Within the Decode and Write Back stage, a register file component is implemented to read and write data from the 15 registers.

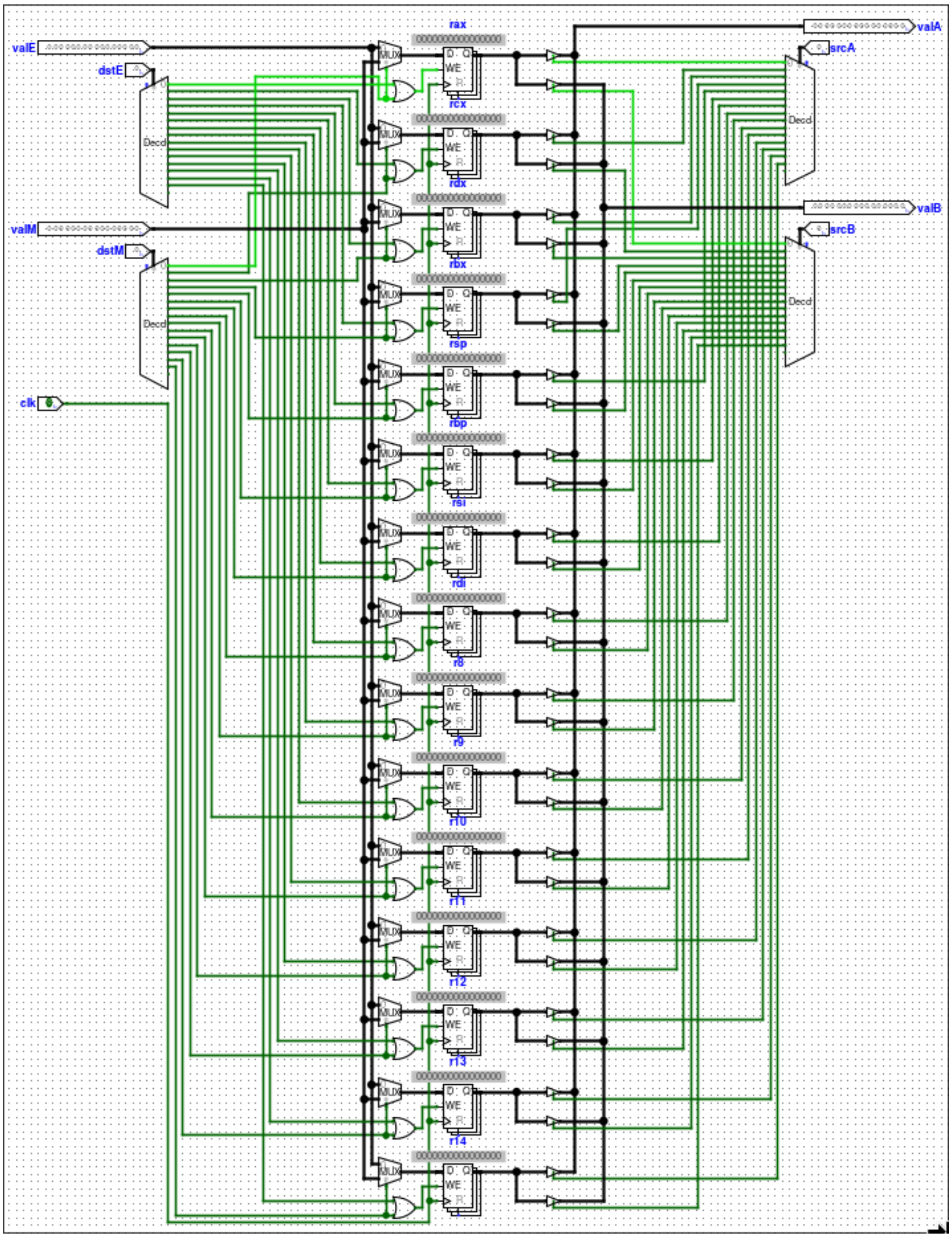


Figure 7: Register file

1.3 Execute Stage

The Execution stage mainly surrounds the Arithmetic Logic Unit (ALU) performing the four operations (addq, subq, andq, and xorq). A condition code is also calculated in a form of (Zero Flag/Sign Flag/Overflow Flag). The Condition Code (CC) module lets the CC store the last condition code. The SetCC module generates a signal to store CC. It is set if the icode is 6 (OPq). The signal of Cond is used by jXX in NewPC, cmovXX in Write Back stage. ALU_fun determines the operation that must be operated by the ALU. ALU_A and ALU_B compute the inputs for the ALU.

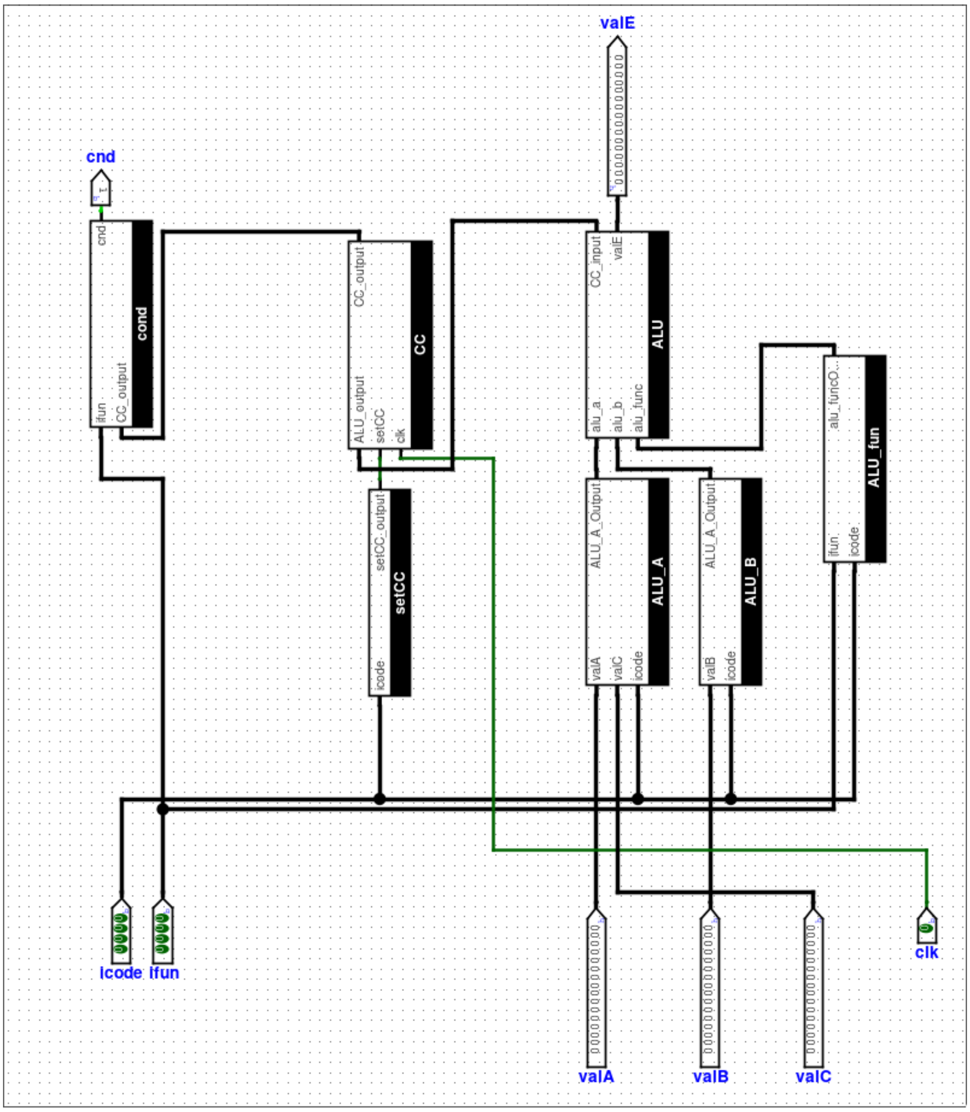


Figure 8: Execute stage

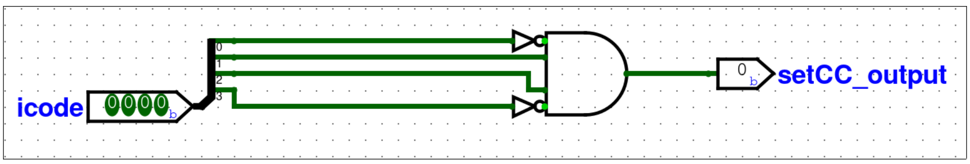


Figure 9: Set Condition Code module

As mentioned above, ALU_A and ALU_B both determine the inputs of the ALU depending on the instruction code given. For example, the instruction code (icode) determines if ALU_A's output will be either valA, valC, 8 or -8 through a 16x1 multiplexer.

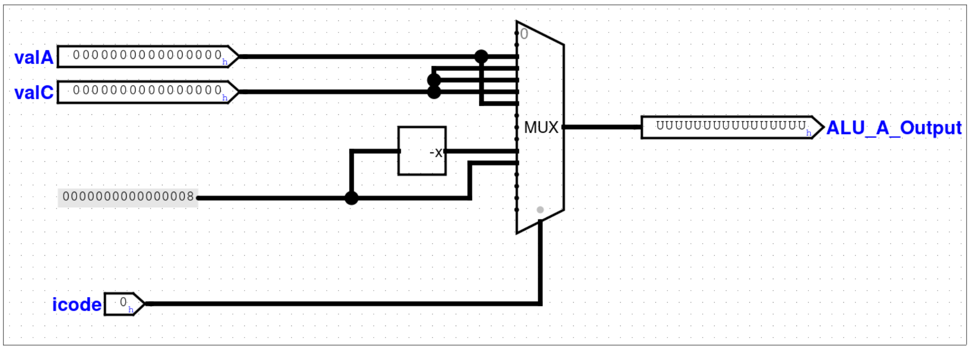


Figure 10: ALU_A module

Likewise, the output for ALU_B was derived from a 16x1 multiplexer that chose from either 0 or valB, depending on the instruction code.

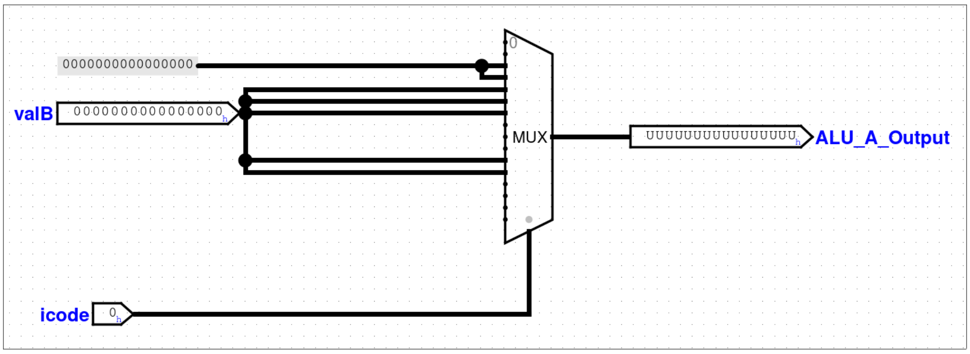


Figure 11: ALU_B module

To find out which function the ALU has to perform, a 16x1 multiplexer uses the instruction code to choose between 0 (addition) or other functions determined by ifun (the instruction function).

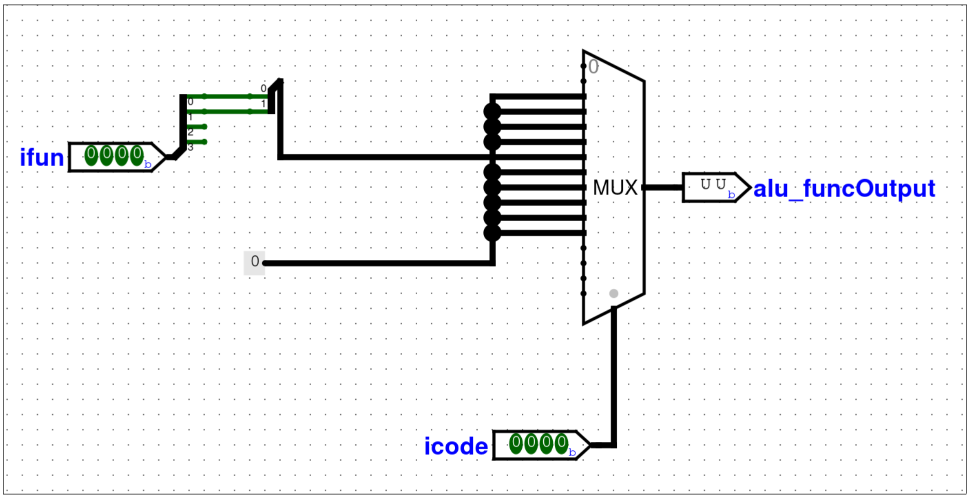


Figure 12: ALU_fun module

The condition module sets the condition flag off (cnd) depending on the condition code found by the ALU and the instruction function being read. For example, if the instruction function were jl (ifun = 2), the expected condition code would be an input of 011 (ZF/SF/OF). AND gates are set in place to make sure both inputs are met. If both inputs meet the requirement, then cnd will output 1.

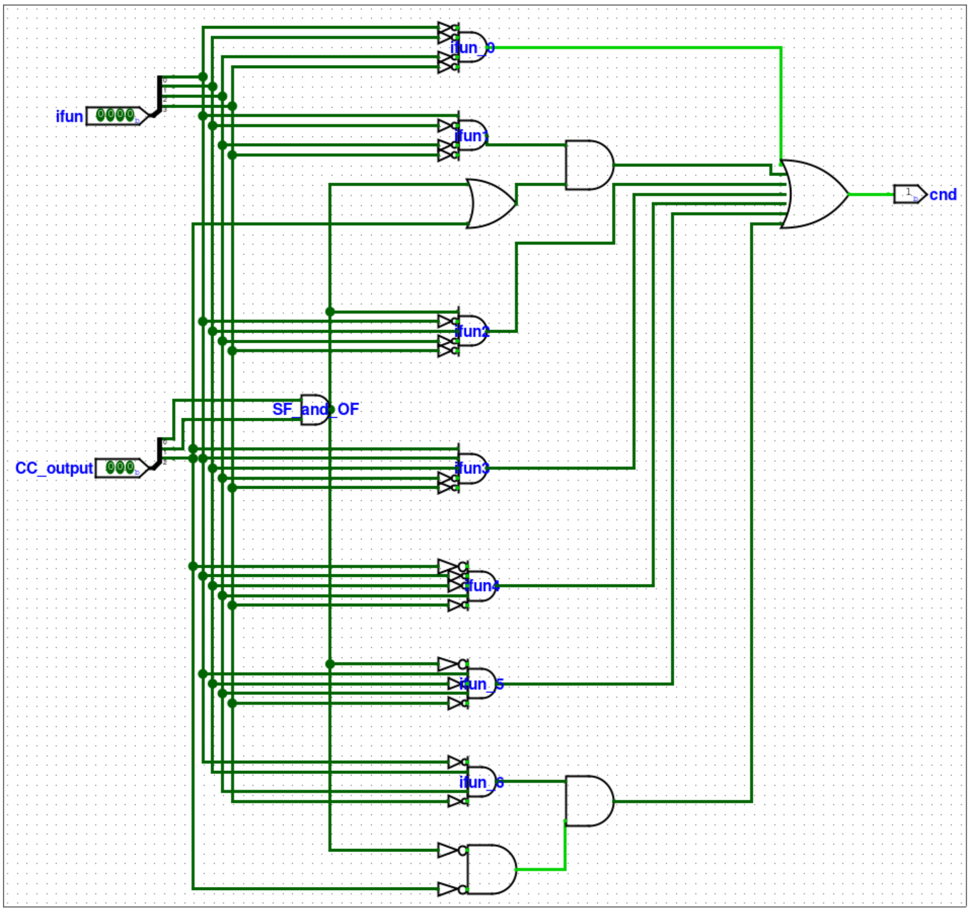


Figure 13: Condition module

In the actual ALU module, both inputs go through a certain function determined by a 4x1 multiplexer that selects based on the ALU_func module's output. The four possible functions are addition, subtraction, the bitwise AND operation, or the bitwise XOR operation. The condition code is determined by comparing the new value with 0. Overflow is also checked through the built-in adder operation.

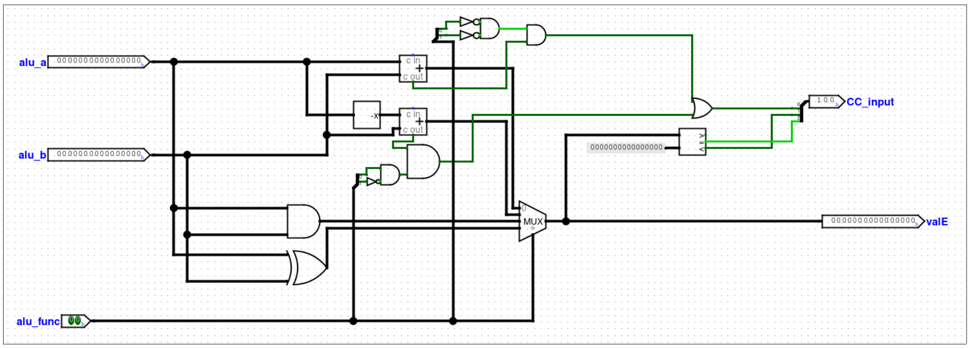


Figure 14: ALU module

1.4 Memory Stage

The main function for the Memory stage is to read or write memory word. The control logic surrounding this stage is the stat, mem.read, mem.write, mem.addr, and the mem.data modules. The stat module concludes what the status of the instruction is. Mem.read determines if the word should be read. Likewise, Mem.write determines if the word should be written. Mem.addr selects the address and Mem.data selects the data.

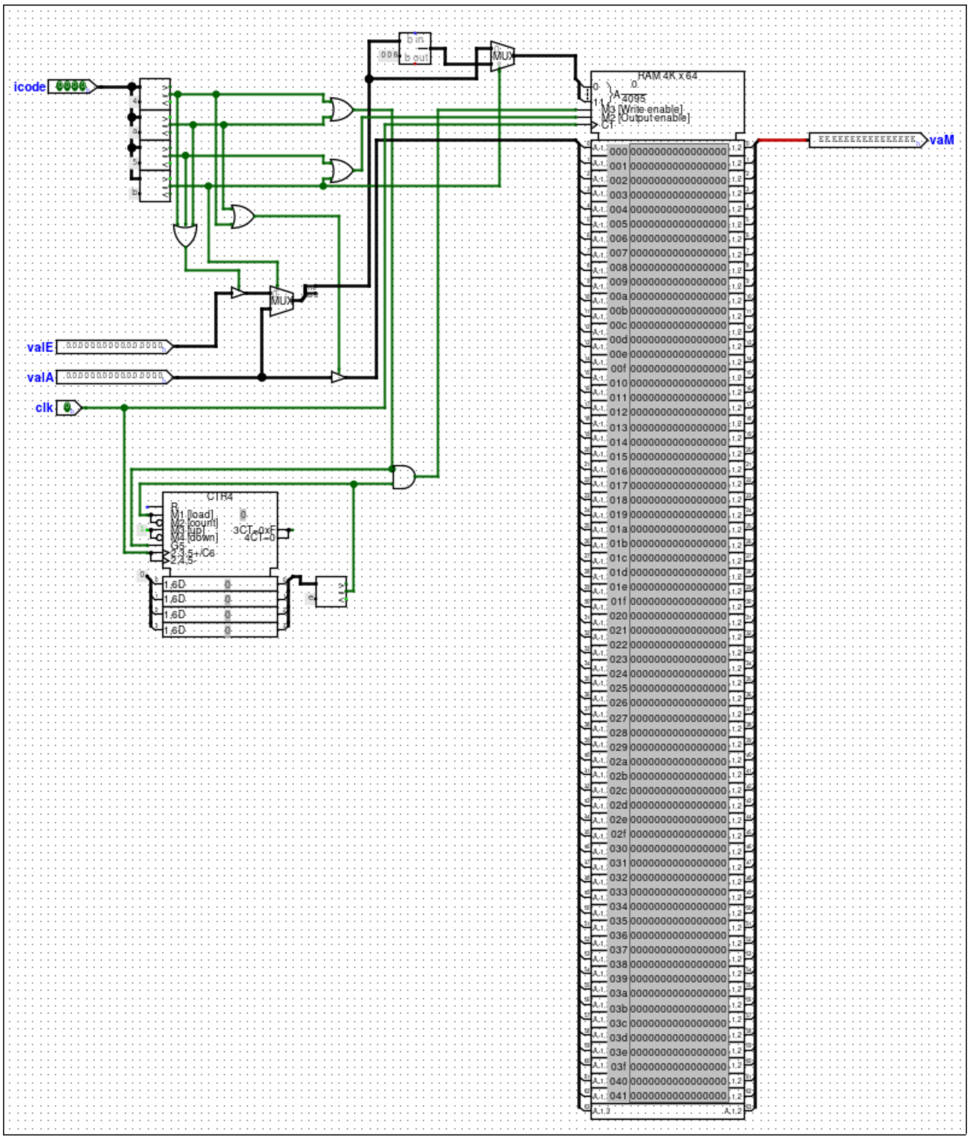


Figure 15: Memory stage

1.5 PC Update Stage

The PC Update stage sets a new value for the program counter (PC). This new value is computed through a 16x1 multiplexer that is based on the instruction code. The multiplexer determines the current PC value and the result of certain operations or conditions within the processor. A special case is for instruction code 7 (any jump function) which uses a 2x1 multiplexer to determine the new PC between valC or valP, depending if the condition flag is on.

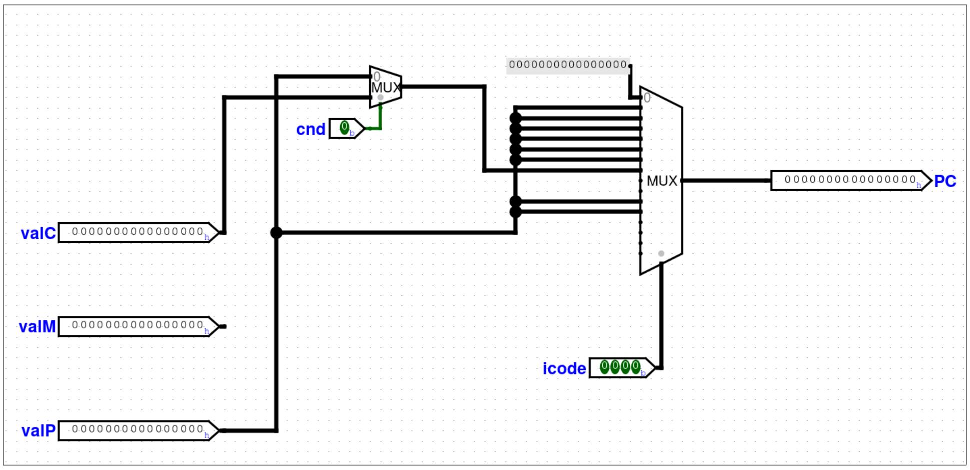


Figure 16: PC Update stage

2 Timing Analysis

The following is a timing diagram for the Y86 ISA implementation.

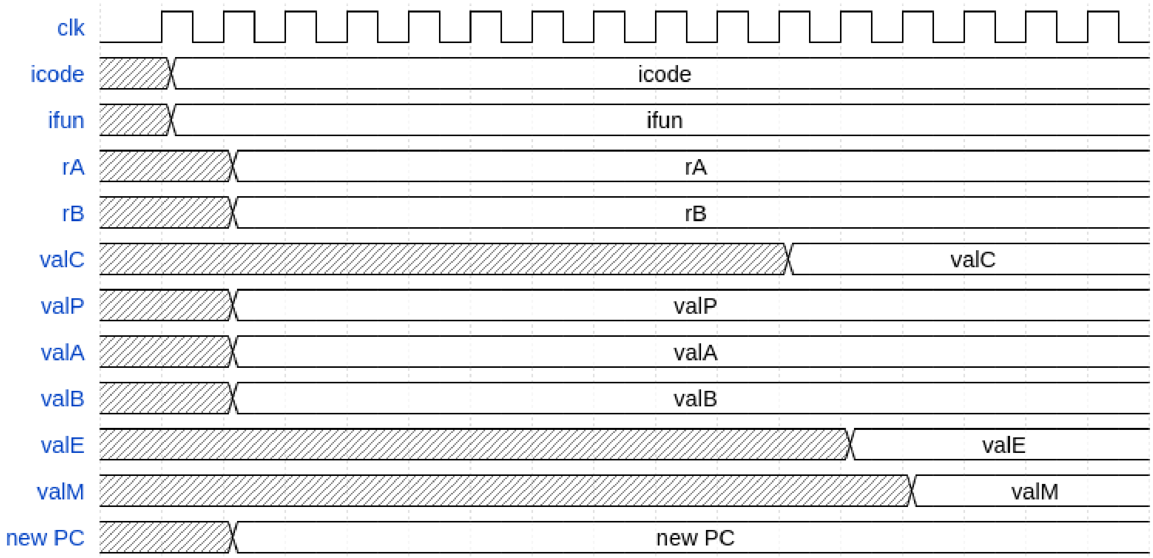


Figure 17: Timing diagram for the Y86 ISA implementation

3 Testing

The following programs were used to test the correctness of the Y86 ISA implementation.

3.1 Program 1

```
    .pos 0
    irmovq $100, %rsp
    irmovq $0xabc, %rax
    pushq %rax
    irmovq $0xdef, %rax
    pushq %rax
    irmovq $0xabcdeff, %rax
    popq %rcx
    popq %rdx
```

```
    rmmovq %rcx, 0x10
    mrmovq 0x10, %rdi
    jmp newpos
    nop
    nop
    rrmovq %rdi, %r11
newpos:
    rrmovq %rdi, %r12
    halt
```

This program verifies the following instructions:

- irmovq
- pushq
- popq
- rmmovq
- mrmovq
- jmp
- nop
- rrmovq
- halt

The expected output is:

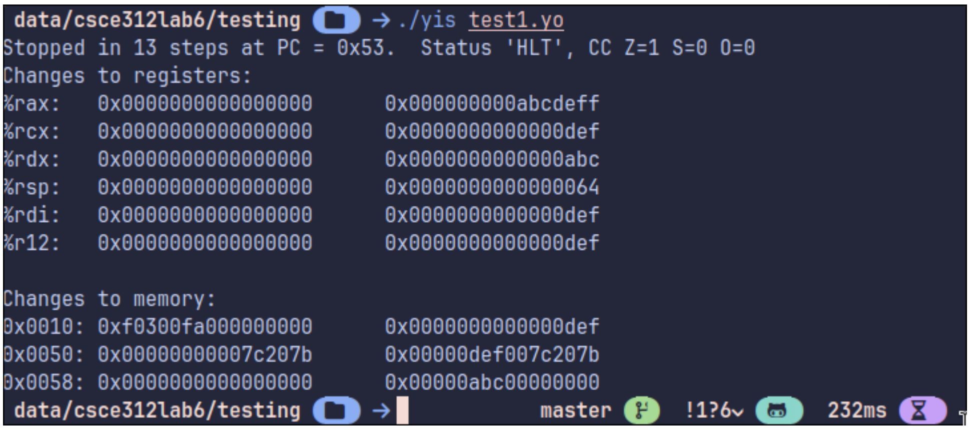


Figure 18: Expected output for Program 1

The final values of the registers and memory are as follows:

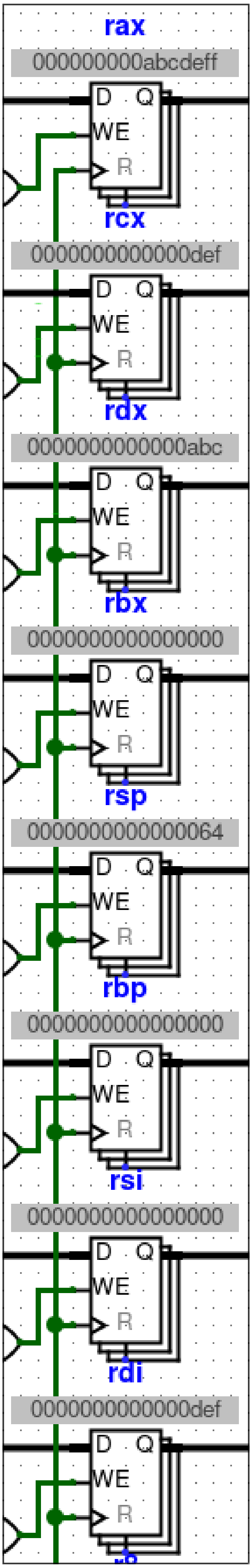


Figure 19: Final register values for Program 1

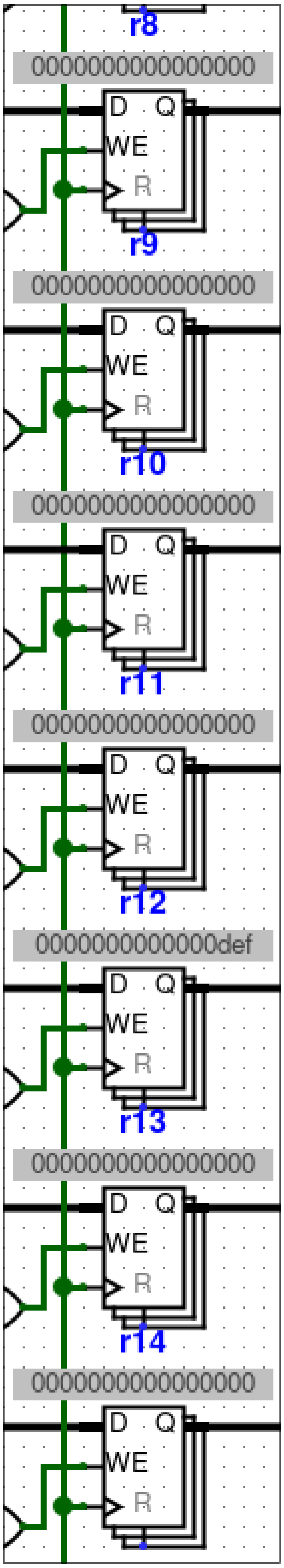


Figure 20: Final register values for Program 1

010	0000000000000def
014	0000000000000000
018	0000000000000000
01c	0000000000000000
020	0000000000000000
024	0000000000000000
028	0000000000000000
02c	0000000000000000
030	0000000000000000
034	0000000000000000
038	0000000000000000
03c	0000000000000000
040	0000000000000000
044	0000000000000000
048	0000000000000000
04c	0000000000000def
050	0000000000000000
054	0000000000000abc
058	0000000000000000

Figure 21: Final memory values for Program 1

3.2 Program 2

```
    irmovq $0, %rax           # Load immediate 0 into %rax
    irmovq $10, %rbx          # Load immediate 10 into %rbx
    irmovq $1, %rcx           # Load immediate 1 into %rcx (loop counter)
    jmp loop

label:
    irmovq $20, %rdi
    jmp end

loop:
    addq %rbx, %rax           # Increment %rax by %rbx
    rrmovq %rax, %rdx         # Move %rax to %rdx

    irmovq $1, %rsi           # Load immediate 1 into %rsi
    subq %rsi, %rcx           # Decrement %rcx by 1
    irmovq $0, %rsi           # Reset %rsi to 0 for comparison

    # Use register value directly to control the loop: simple decrement
       and stop condition
    rrmovq %rcx, %rsi         # Copy %rcx to %rsi
    je label                  # Continue loop if %rcx is not equal to %rsi
       (i.e., not zero)

end:
    halt                      # Stop the program
```

This program verifies the following instructions:

- irmovq
- addq
- rrmovq
- subq
- jne
- halt

The expected output is:

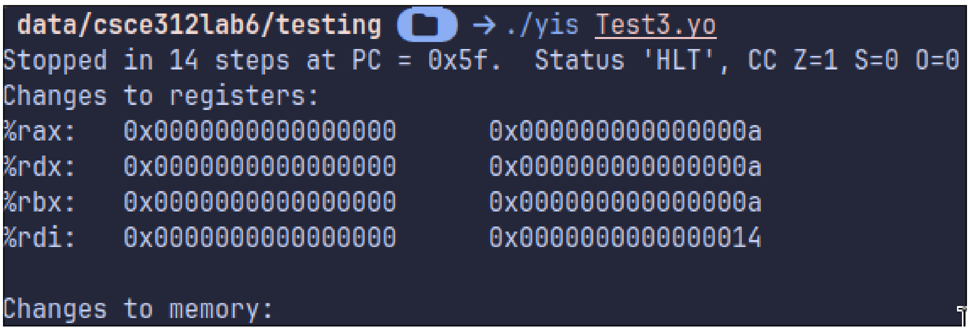


Figure 22: Expected output for Program 2

The final values of the registers are as follows:

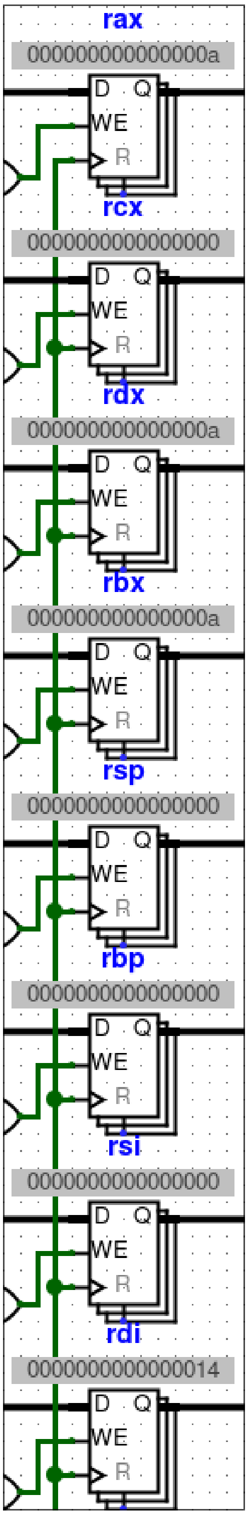


Figure 23: Final register values for Program 2

3.3 Program 3

```
# Initialize stack pointer
irmovq $0x200, %rsp

# Initialize registers
irmovq $10, %rax       # Load immediate 10 into %rax
rmmovq %rax, 8(%rsp)  # Store %rax at memory address 8(%rsp)
irmovq $5, %rdx        # Load immediate 5 into %rdx

# Use memory and stack operations
mrmovq 8(%rsp), %rbx  # Load %rbx from memory address 8(%rsp)
pushq %rax             # Push %rax onto the stack
pushq %rdx             # Push %rdx onto the stack as well

# Perform arithmetic on the stack's top values
popq %rcx              # Pop the top of the stack into %rcx (was %rdx)
popq %rsi              # Pop the next top of the stack into %rsi (was
   %rax)
addq %rsi, %rcx        # Add %rsi to %rcx, store the result in %rcx
rrmovq %rcx, %rbx      # Move result back into %rbx for further use

# More stack manipulations
pushq %rbx             # Push the result back onto the stack
pushq %rcx             # Push %rcx onto the stack again

# Double the value at the top of the stack
popq %rdi              # Pop the top of the stack into %rdi
addq %rdi, %rdi        # Double the value in %rdi
pushq %rdi             # Push the doubled value back onto the stack

# Clean up and repeat operations
nop                    # No operation
halt                   # Stop the program
```

This program verifies the following instructions:

- irmovq
- rmmovq
- mrmovq
- pushq
- popq
- addq
- rrmovq
- nop
- halt

The expected output is:

```
data/csce312lab6/testing  → ./yis Test4.yo
Stopped in 18 steps at PC = 0x49.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x000000000000000a
%rcx:   0x0000000000000000      0x000000000000000f
%rdx:   0x0000000000000000      0x0000000000000005
%rbx:   0x0000000000000000      0x000000000000000f
%rsp:   0x0000000000000000      0x00000000000001f0
%rsi:   0x0000000000000000      0x000000000000000a
%rdi:   0x0000000000000000      0x000000000000001e

Changes to memory:
0x01f0: 0x0000000000000000      0x000000000000001e
0x01f8: 0x0000000000000000      0x000000000000000f
0x0208: 0x0000000000000000      0x000000000000000a
```

Figure 24: Expected output for Program 3

The final values of the registers and memory are as follows:

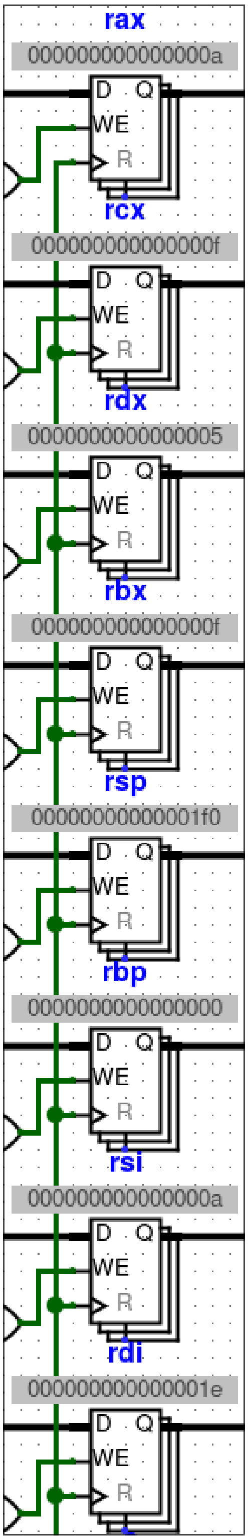


Figure 25: Final register values for Program 3

1f0	000000000000001e
1f8	000000000000000f
200	0000000000000000
208	000000000000000a

Figure 26: Final memory values for Program 3