Group 18: CSCE 312 CPU Project — Final Report

Lucian Chauvin 133003371 | **Joshua Lass** 531009387 | **Bjorn Quarfordt** 230003985

1 Transformation Tables

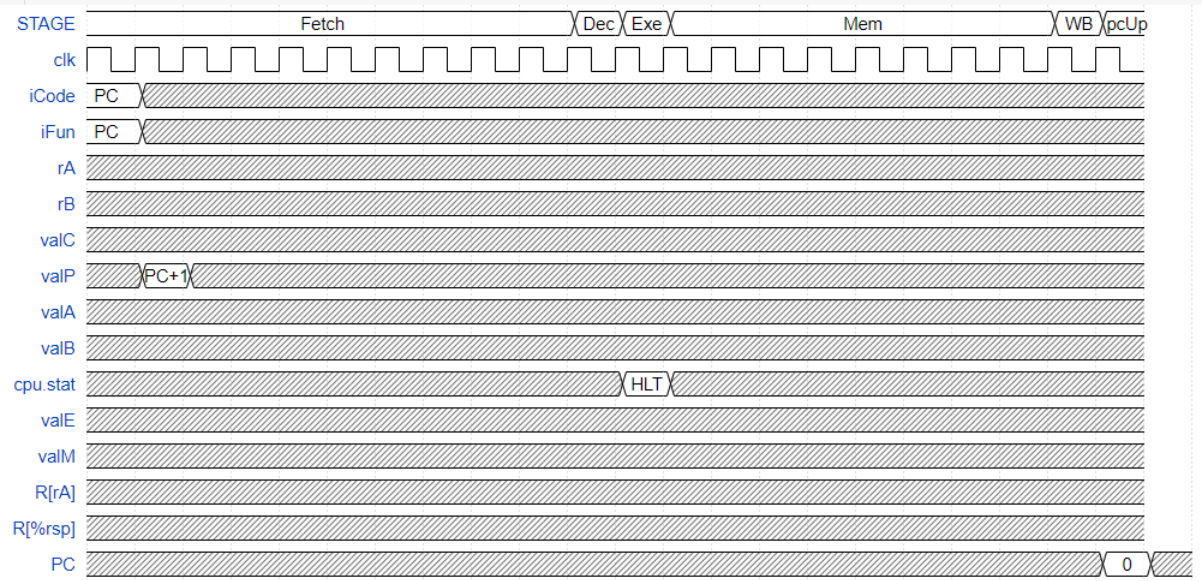
Instruction	Fetch	Decode	Execute	Memory	Write Back	PC Update
rrmovq rA, rB	icode:ifun← M_1[PC] rA:rB ← M_1[PC+1] valP← PC+2	valA← R[rA] valB← R[rB]			R[rB]← valA	PC← valP
irmovq V, rB	icode:ifun ← M_1[PC] F:rB ← M_1[PC+1] valC ← M_8[PC+2] PC ← PC+10				R[rB] ← valC	PC ← valP
rmmovq rA, D(rB)	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	valA← R[rA] valB← R[rB]	valE←valB+valC	M_8[valE]← valA		PC ← valP
mrmovq D(rB), rA	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	valB← R[rB]	valE←valB+valC	valM ← M_8[valE]	R[rA]← valM	PC ← valP
OPq rA, rB	icode:ifun← M_1[PC] rA:rB ← M_1[PC+1] valP← PC+2	valA← R[rA] valB← R[rB]	valE← valB OP valA		R[rB]← valE	PC← valP
jXX Dest	icode:ifun← M_1[PC] valC ← M_1[PC+1] valP← PC+9		cnd← cond(CC_1:ifun)			PC← cond ? valC:valP
cmovXX rA, rB	icode:ifun← M_1[PC] valC ← M_1[PC+1] valP← PC+2	valA← R[rA]	valE← valA		r[rB]← valE	PC← valP
call Dest	icode:ifun← M_1[PC] valC ← M_8[PC+1] valP← PC+9	valB← R[\%rsp]	valE← valB-8	M_8[valE]← valP	R[\%rsp]← valE	PC← valC
ret	icode:ifun← M_1[PC]	valA← R[\%rsp] valB← R[\%rsp]	valE← valB_8	valM← M_8[valA]	R[\%rsp]← valE	PC← valM
pushq rA	icode:ifun← M_1[PC] rA:rB← M_1[PC+1] valP← M_8[PC+10]	valA← R[rA] valB← R[\%rsp]	valE← valB-8	M_8[valE]← valA	R[\%rsp]← valE	PC← valP
pop rA	icode:ifun← M_1[PC] rA:rB← M_1[PC+1] valP← M_8[PC+2]	valA← R[\%rsp} valB← R[\%rsp]	valE← valB+8	valM← M_8[valA]	R[\%rA]← valM R[\%rsp]← valE	PC← valP

2 Timing Diagrams

Timing diagrams help represent the behavior of signals in the CPU for each clock cycle. Each clock cycle allows the CPU to update values and perform calculations needed for each value of iCode. Below are the following clock cycles for each value of iCode:

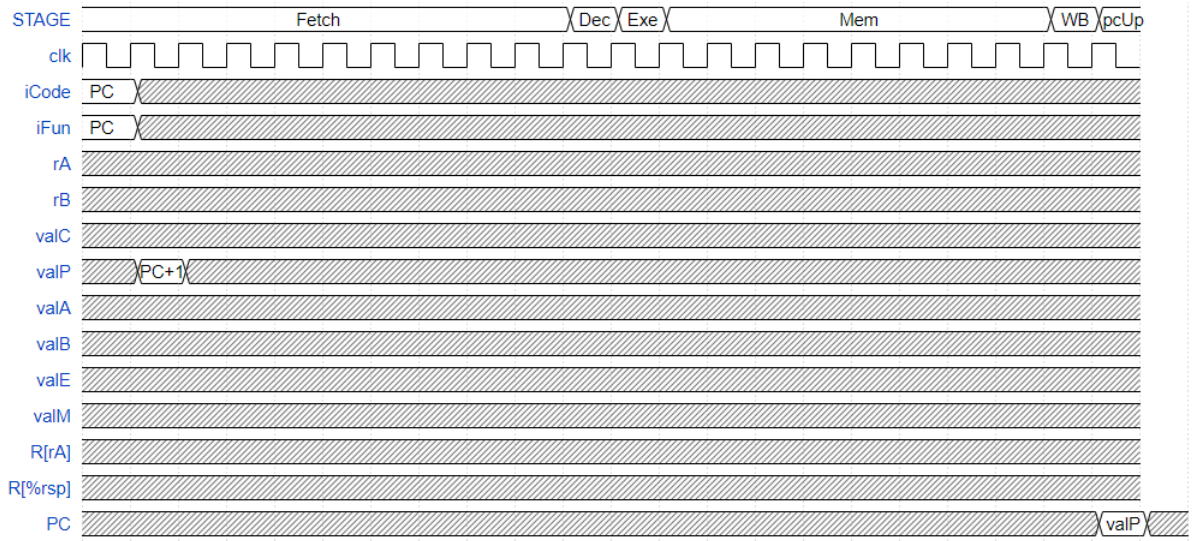
2.1 HALT

When iCode value is 0, a HALT is performed. This means that the entire CPU will stop performing processes completely.



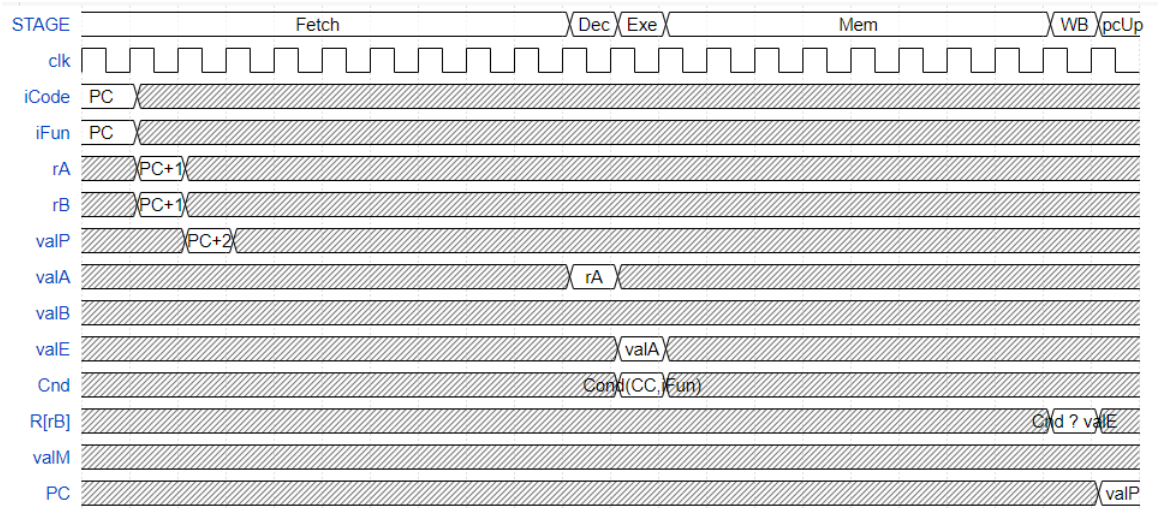
2.2 NOP

NOP means "No Operation," which acts like HALT, but means that this step is skipped and the CPU can receive future inputs. NOP can be used for pipelining to get rid of hazards that arise throughout the code.



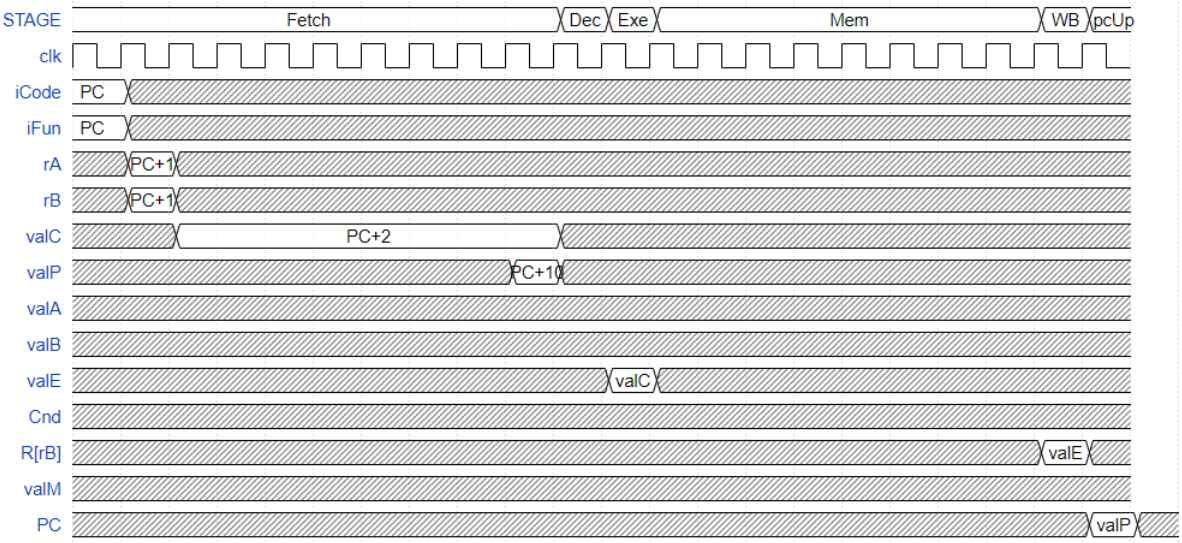
2.3 RRMOVQ

RRMOVQ occurs when iCode is 2. RRMOVQ copies data values from one register to another. This can be used to store one value, while manipulating the other. During OPq, the data in one register is overwritten by the output of the operation performed, and in some cases, the original data is needed for other lines of code.



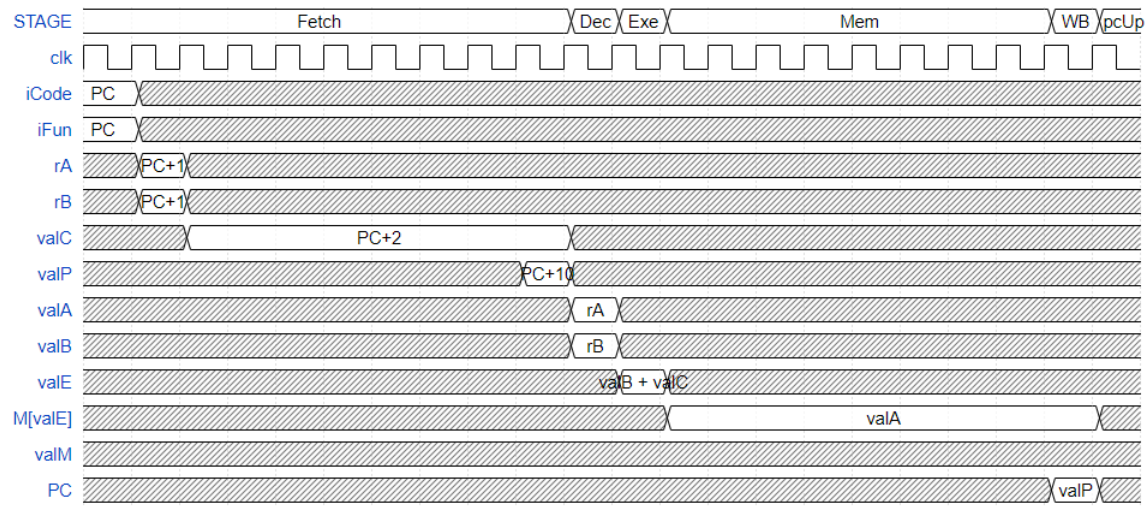
2.4 IRMOVQ

IRMOVQ is called when iCode is 3. IRMOVQ writes a specific value, say 4, to the specified register.



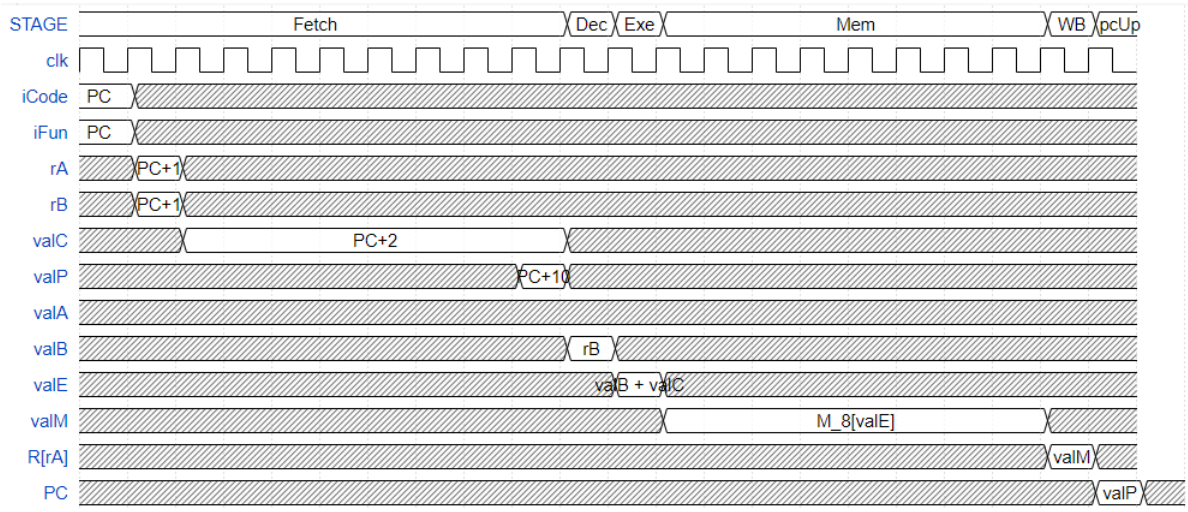
2.5 RMMOVQ

When iCode is 4, RMMOVQ is called to move a value from a specific register to a memory address. This is so the value in the memory can be accessed later on without the need to reserve a register for a value. This is important because we only have 16 registers to use, and with a clock speed of 2KHz, 2000 cycles occur every second which means that the space to hold values in registers is limited.



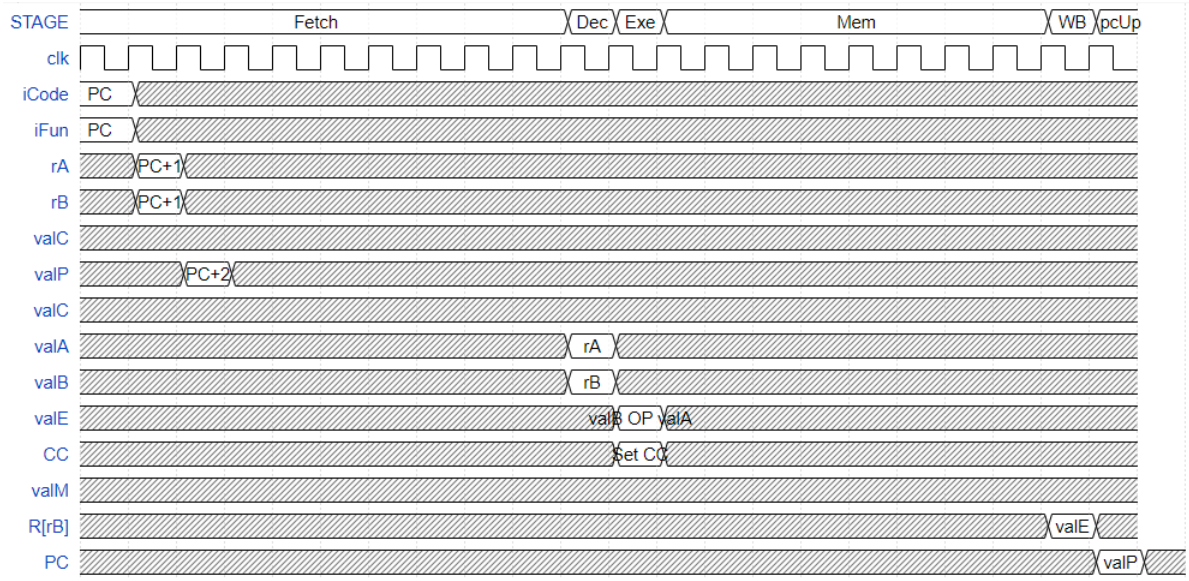
2.6 MRMOVQ

When iCode is 5, MRMOVQ is called to be performed. As stated above, register space is limited, so by storing values in the memory, we can utilize these values later on. By letting the CPU decide which memory address to fetch data from into a register, users can create more advanced programs that need more data than the 16 registers can hold.



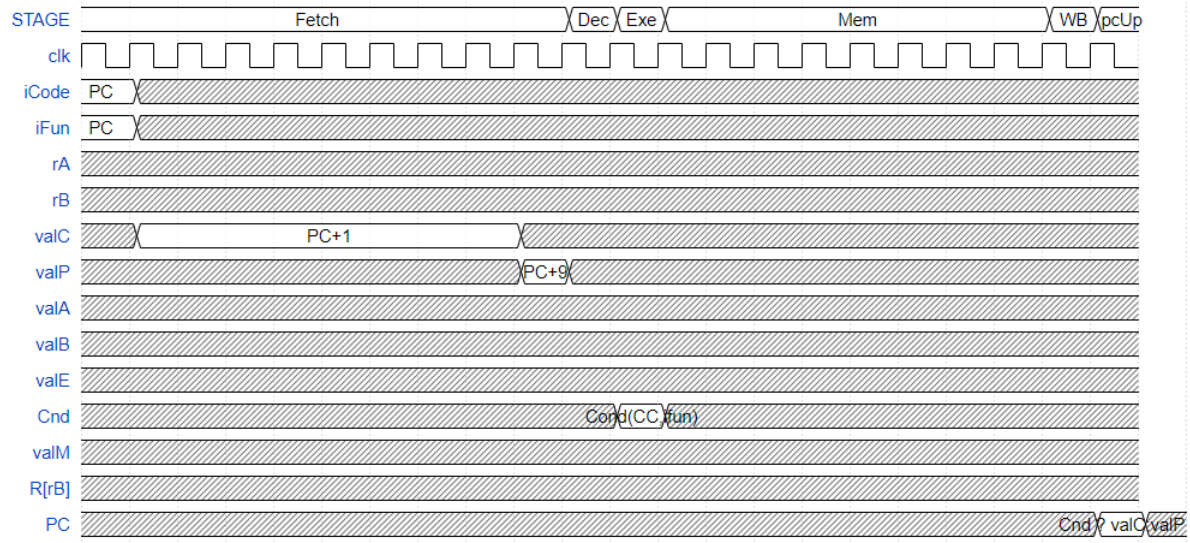
2.7 OPq

OPq occurs when the iCode value is 6. OPq means an operation is performed, whether it be addition, subtraction, multiplication, or XOR. This gives the computer the ability to perform arithmetic operations of values and store said data in registers



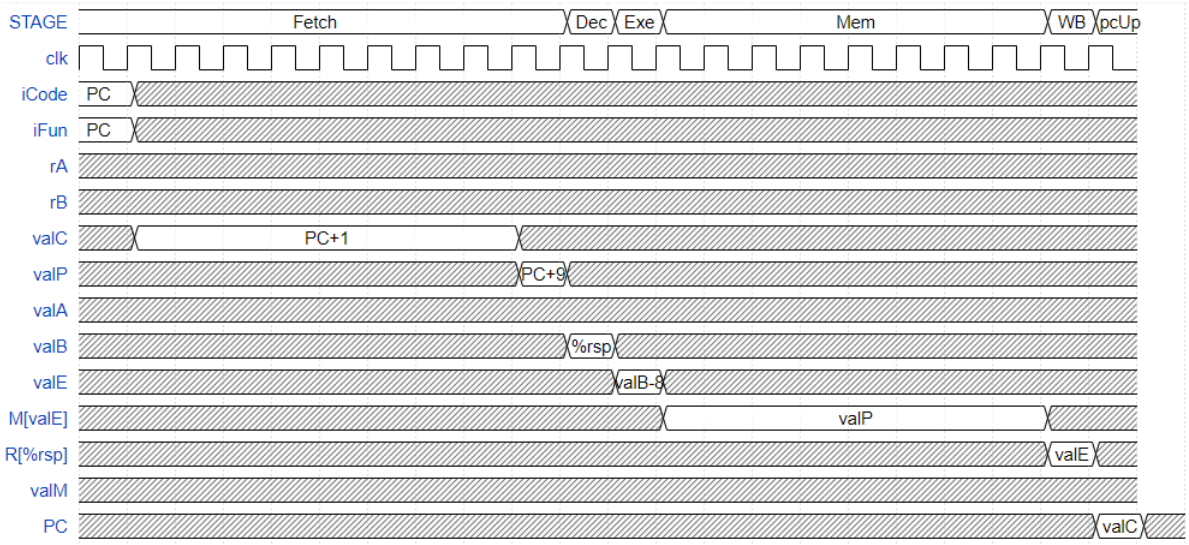
2.8 jXX

When iCode is 7, jXX is performed. The iFun value determines which jump to make: if iFun is 0, a jmp is called, if iFun is 1, jle is called, if iFun is 2, jl is called, etc.



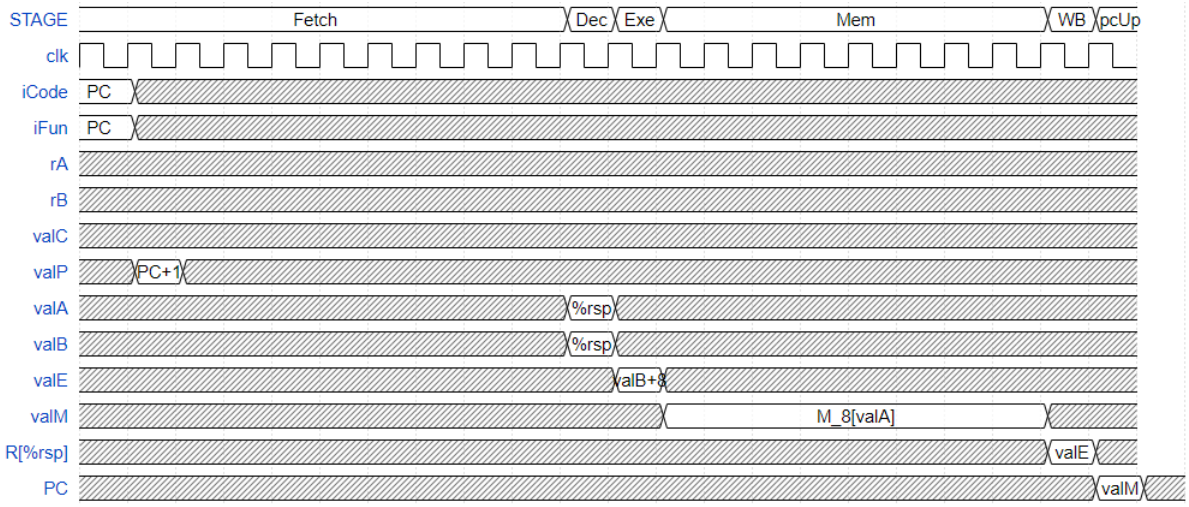
2.9 CALL

CALL happens when iCode is 8. It pushes the address of the next instruction onto the stack, %rsp, and jumps to the specified function.



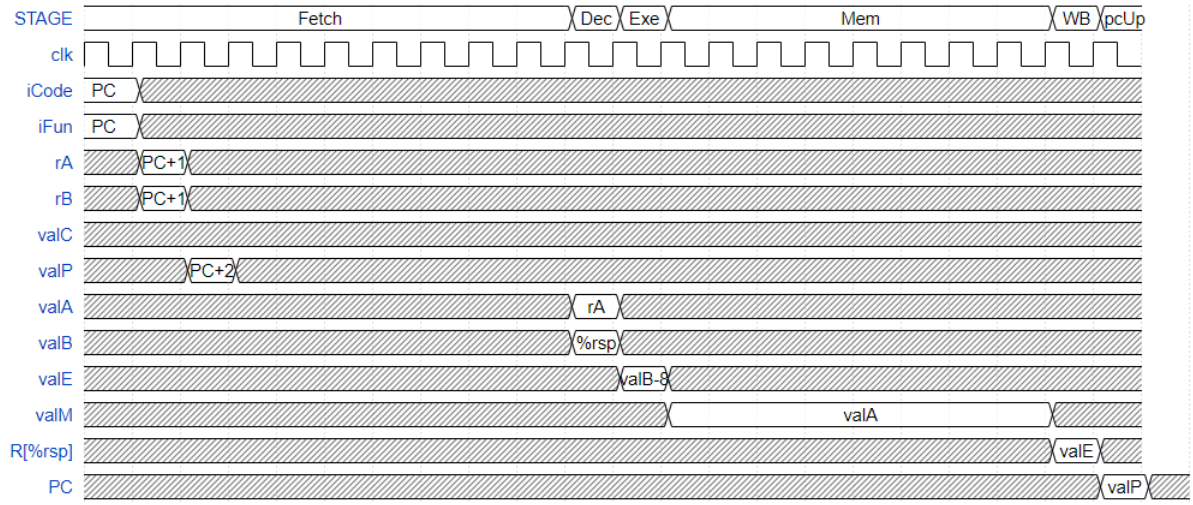
2.10 RET

RET means "return" and it occurs when the iCode value received is 9. RET tells the CPU to take in the value stored at %rsp, and jump back to the address received.



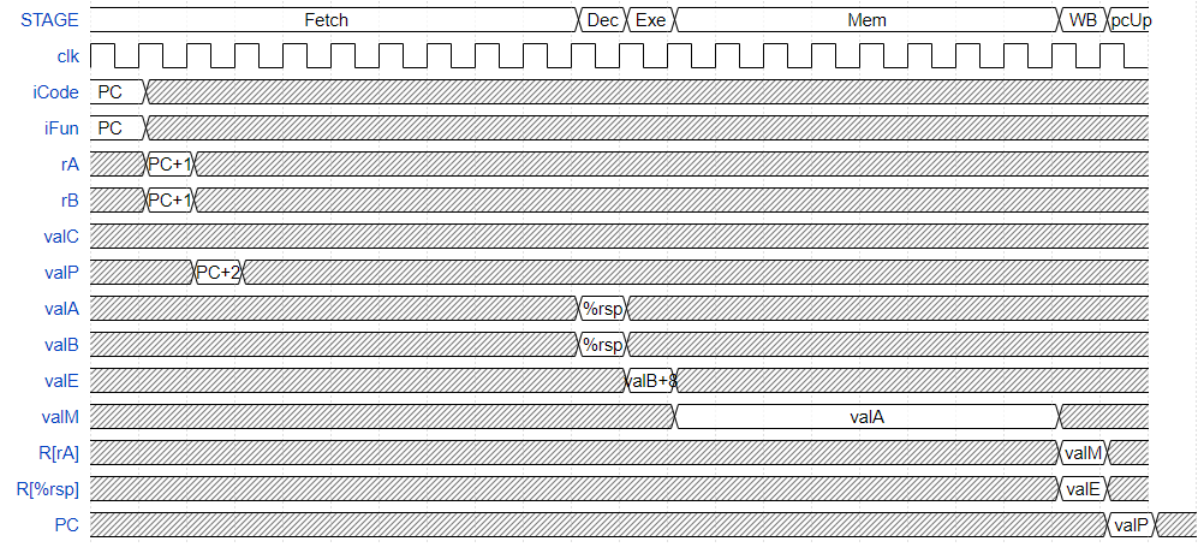
2.11 PUSHQ

PUSHQ is called when iCode is 'a', or 10. PUSHQ decrements %rsp by 8, and copies the value onto the stack at that specific address pointed by %rsp.



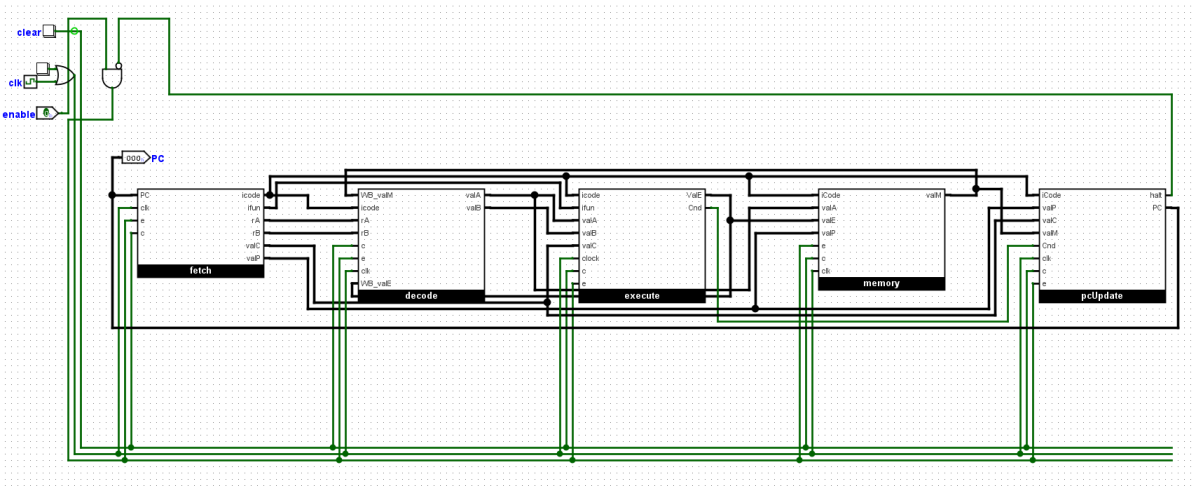
2.12 POPQ

POPQ occurs when iCode is 'b', or 11. POPQ copies the value in the address pointed by %rsp, and increments %rsp by 8 bytes.



3 Overall CPU

When combining each stage, we created our CPU and verified that all stages were performing properly. The completed circuit is down below:



3.1 iCode and Operations

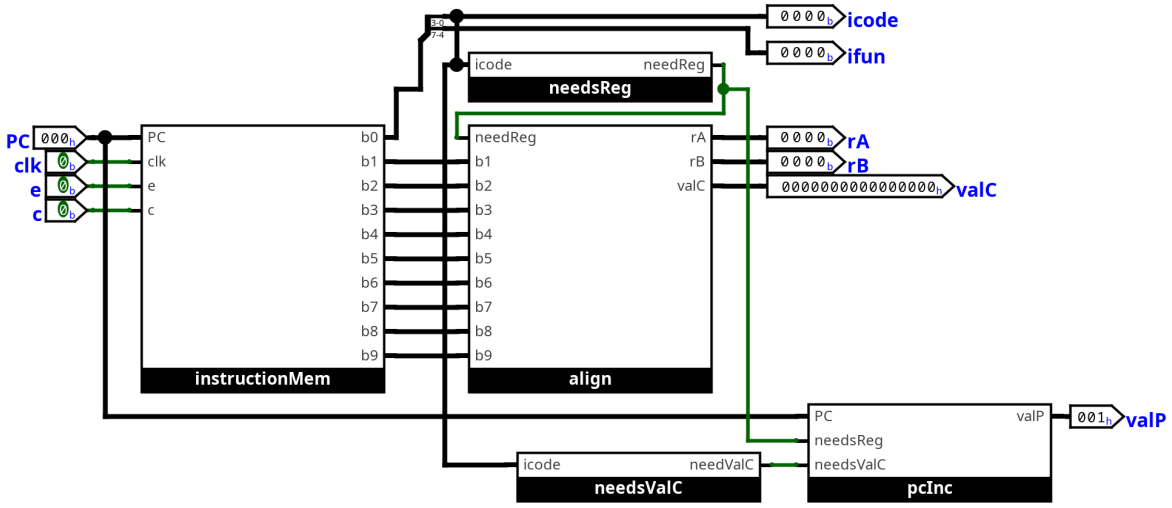
Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq **rA, rB**	2	0	**rA**	**rB**						
irmovq **V, rB**	3	0	F	**rB**	**V**					
rmmovq **rA, D(rB)**	4	0	**rA**	**rB**	**D**					
mrmovq **D(rB), rA**	5	0	**rA**	**rB**	**D**					
OPq **rA, rB**	6	**fn**	**rA**	**rB**						
jXX **Dest**	7	**fn**	**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						

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		

4 Fetch Implementation

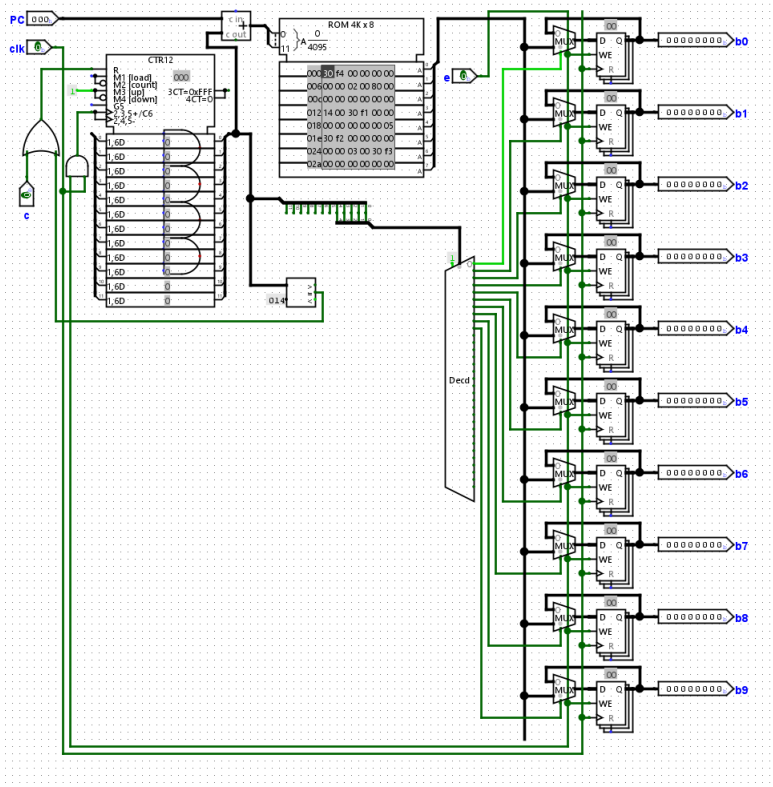
Our design is a one-to-one implementation of what is shown on the slides. Our instruction memory module stores the program in ROM and reads 10 bytes from the current `PC`. We then pass these 10 bytes to our align module which — based on whether we need registers or not — sets out `rA`, `rB`, and `valC` correctly. We determine whether we need registers based on the value of `icode`. Then based on whether we read in registers or if we read in a `valC`, we increment our `PC` using our PC increment module.

4.1 Fetch



4.2 Instruction Memory

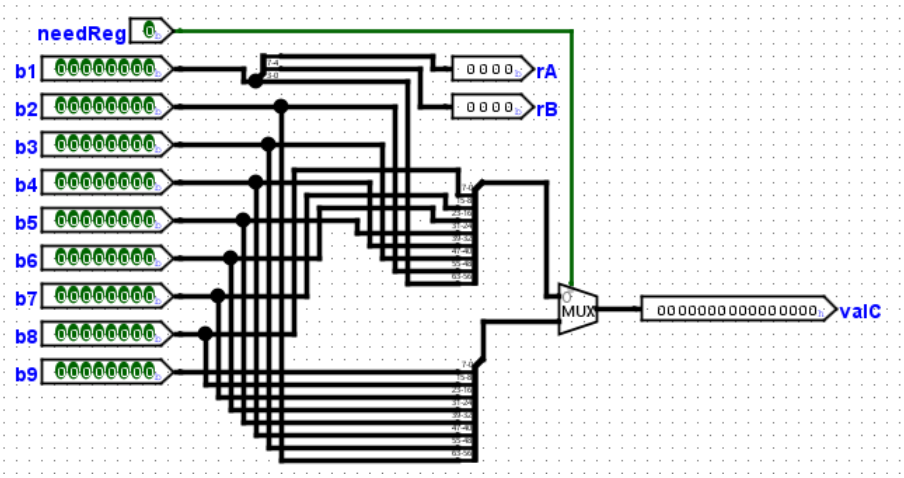
Our instruction memory module consists of a ROM module that stores the program along with a counter and 10 registers to store each byte of our program. Based on the counter we use a decoder to set the value of the corresponding register the count points to. We also utilize a simple 4-way `AND` gate to determine when to reset our counter. This module takes 10 cycles to read in all 10 bytes (one for each byte).



4.3 Align

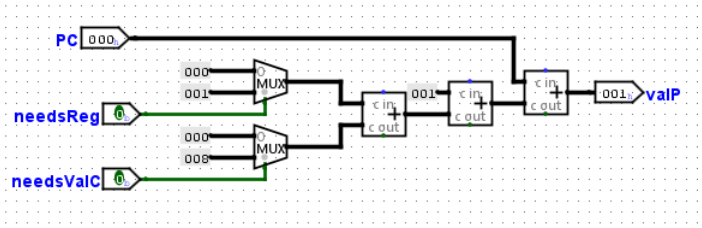
The align module determines the values of `rA`, `rB`, and `valC` based on whether our instruction needs registers or not. When the instruction doesn't need registers we simply construct `valC` based on the first byte being the most significant to the eighth being the least. We just let our registers still be the first byte in this case

as it does not matter what is in them. When we do have registers used in our instructions we start with our most significant byte in `valC` being the second byte to the least significant being the ninth.



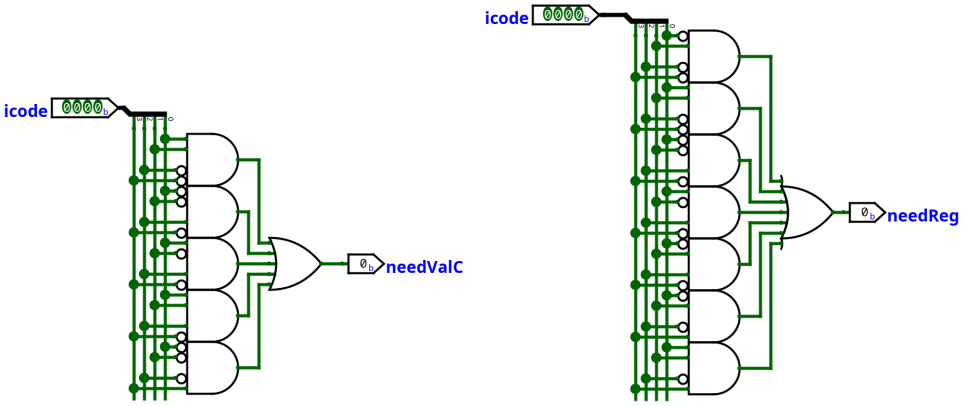
4.4 PC Increment

PC increment simply increments the `PC` based on whether our instruction reads in registers and/or a `valC`. If we read in registers we add 1 to our `PC` and if we read in a `valC` we add 8 to our `PC`. We then also just add 1 for our first byte containing `icode:ifun`.



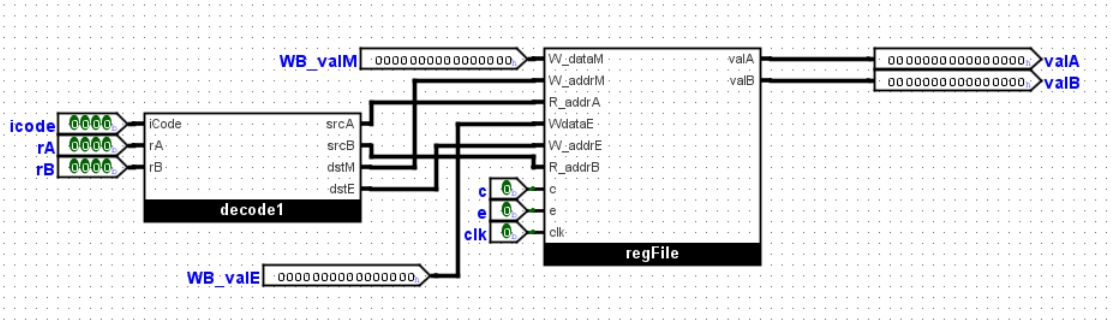
4.5 Needs ValC and Registers

We simply encode which instructions need a `valC` and register based on their `icode`.



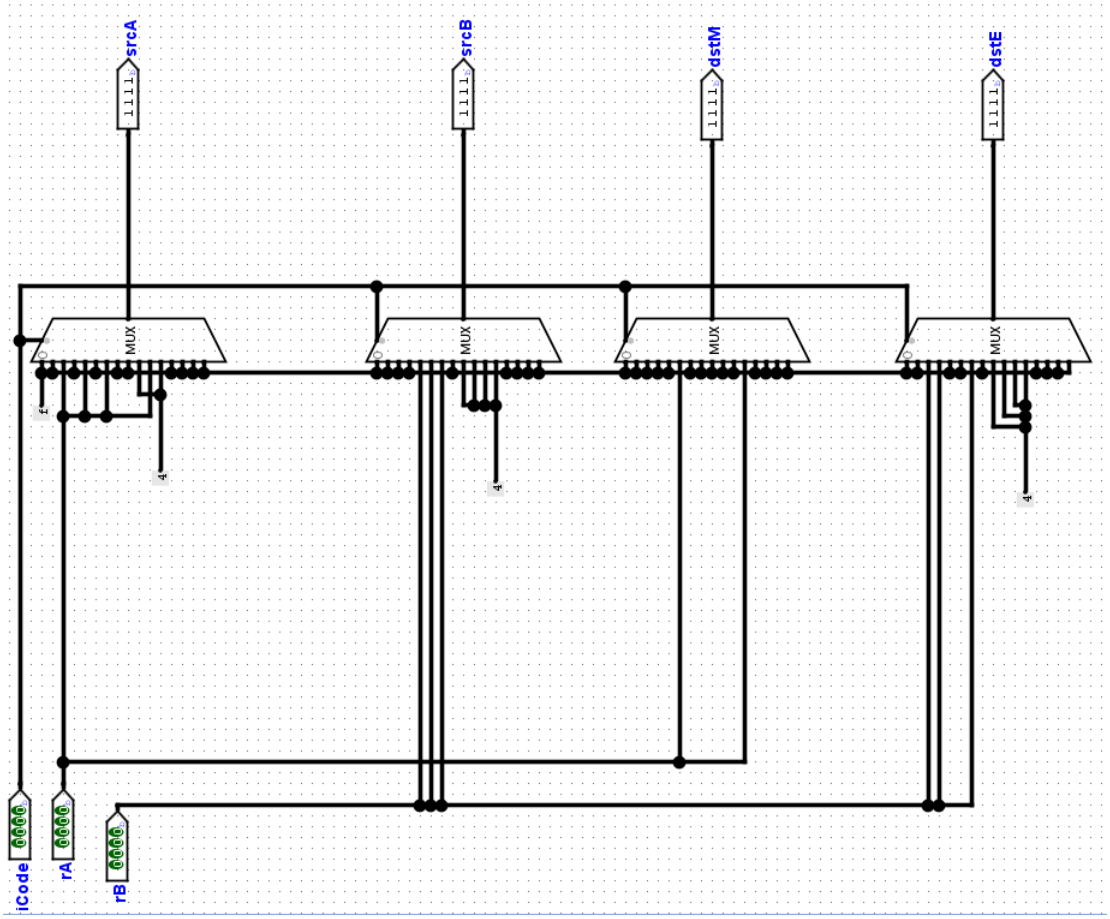
5 Decode and Write Back Implementation

Decode takes in 2 register addresses, rA and rB, and 2 write back values. During the decode stage, iCode to determines which registers to read from and outputs those values to valA and valB. During the write-back stage, the data is from valM and valE are written to the addresses outputted by dstM, and dstE, respectively.



5.1 Decode1

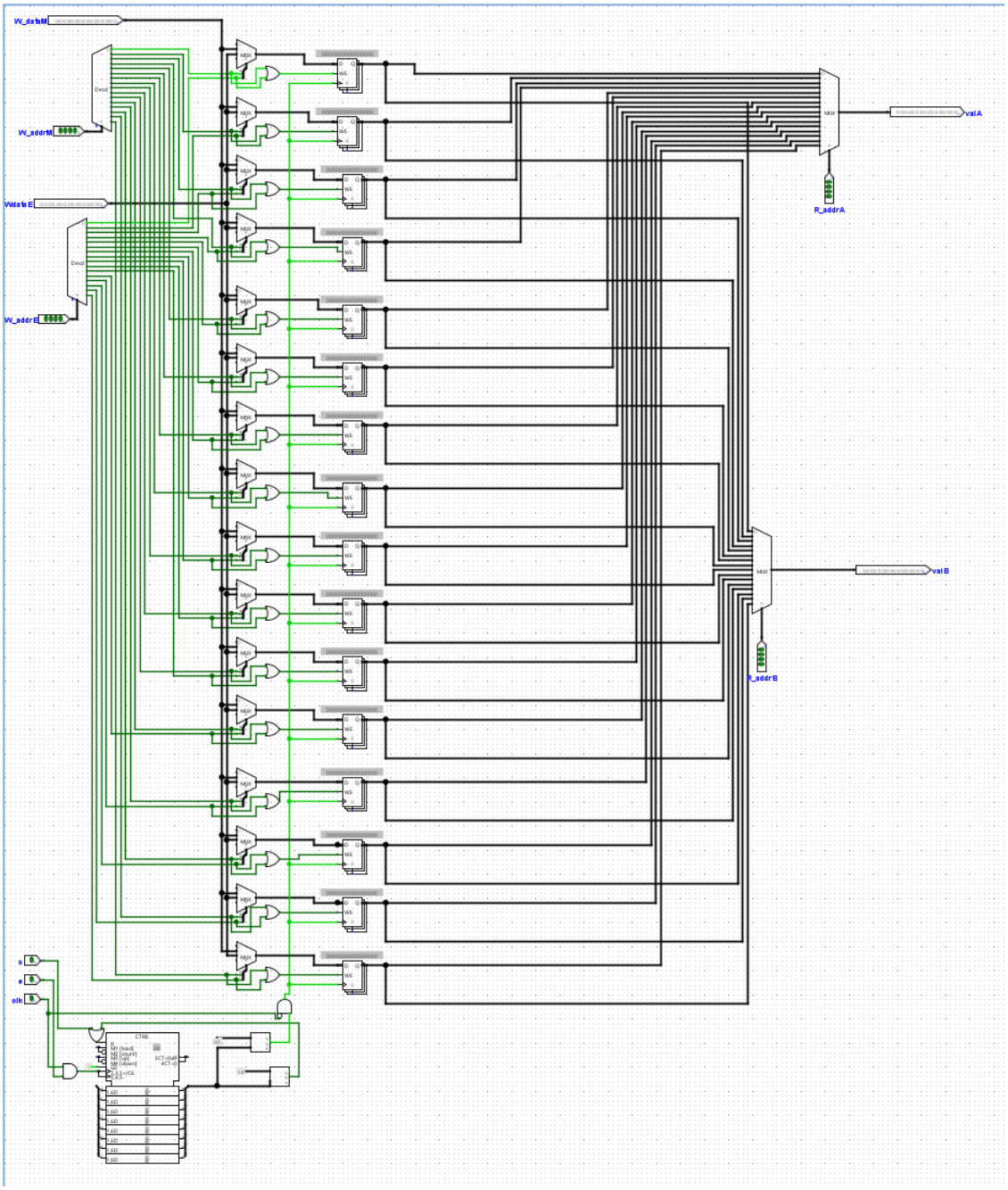
Decode1 is a circuit that was built to be used to determine the values of srcA, srcB, dstM, and dstE, using iCode, rA, and rB. SrcA and srcB are 4-bit address values that tell which register addresses to read from. A multiplexer was used for each output, using iCode as the selector bit to decide which value is outputted for each value. Using the transformation table in section 1, each multiplexer was routed for each instance of iCode. When a value should not be outputted for a specific register address, 0xf is used to say there is no register to write to. For example, if srcA was 1001, the 10th register in the register file will be read from and outputted at valA.



5.2 Register File

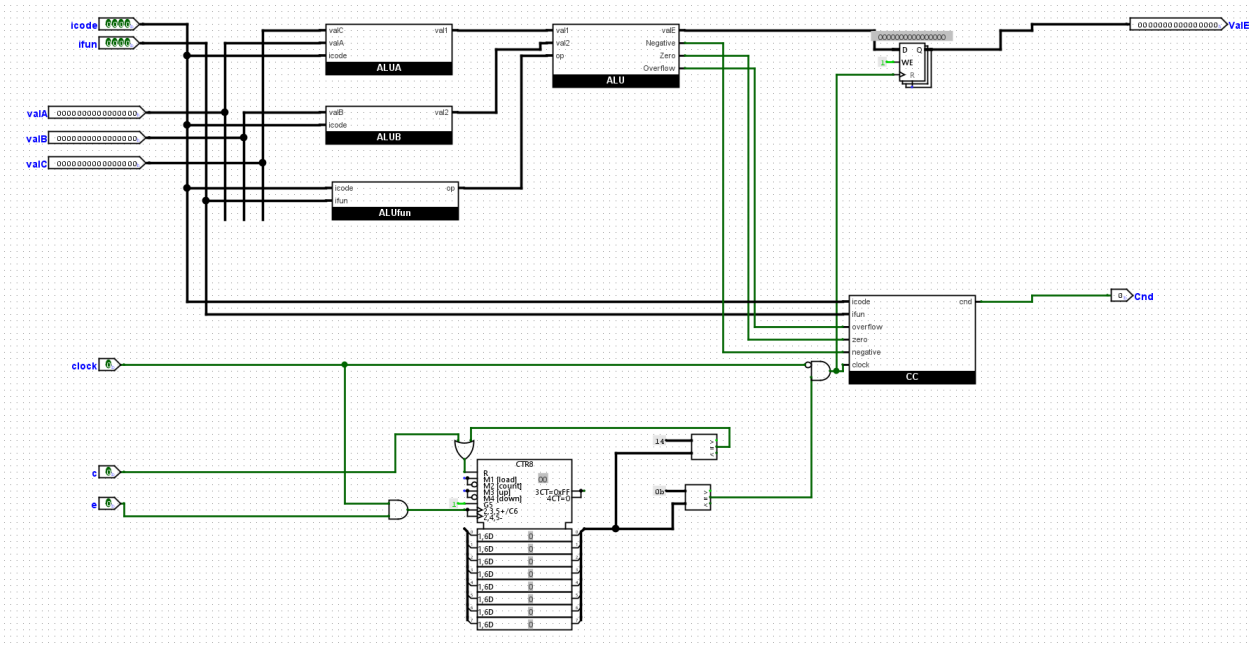
RegFile is the register file used to store the values obtained during the writeback process. There are 16 registers that each hold 64-bit values, that can be accessed using 4-bit addresses. SrcA and srcB are 4-bit addresses that read the corresponding register value and output them to valA and valB respectively. When

a valM or valE is supplied for dstM or dstE, those values are written to their register values to be used later on.



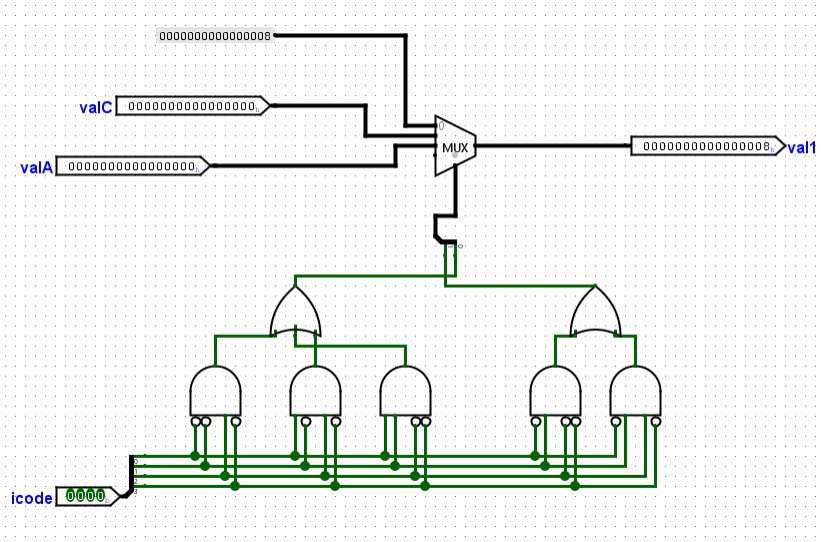
6 Execute Implementation

Execute uses 6 input values to perform arithmetic and logical operations on the valA, valB, or valC, given the iCode and iFun values. ICode and iFun determine which values go into which ALU circuit to then be inputted into the final ALU circuit to receive the output of valE and Cnd.



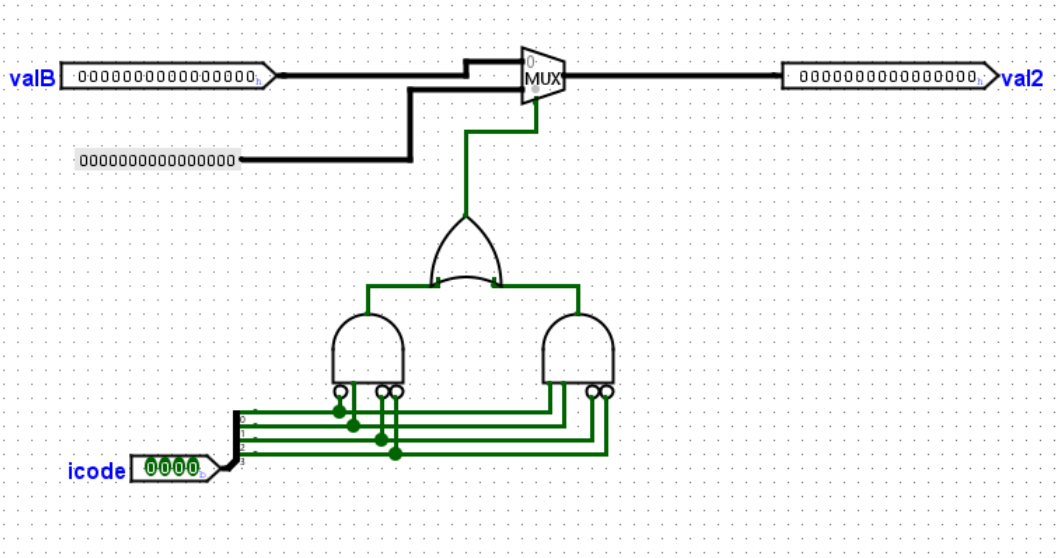
6.1 ALU A

ALUA takes input values of iCode, valA, and valC. When iCode is provided, it determines which values will be outputted to val1. For example, when iCode is 3, valC is outputted to val1.



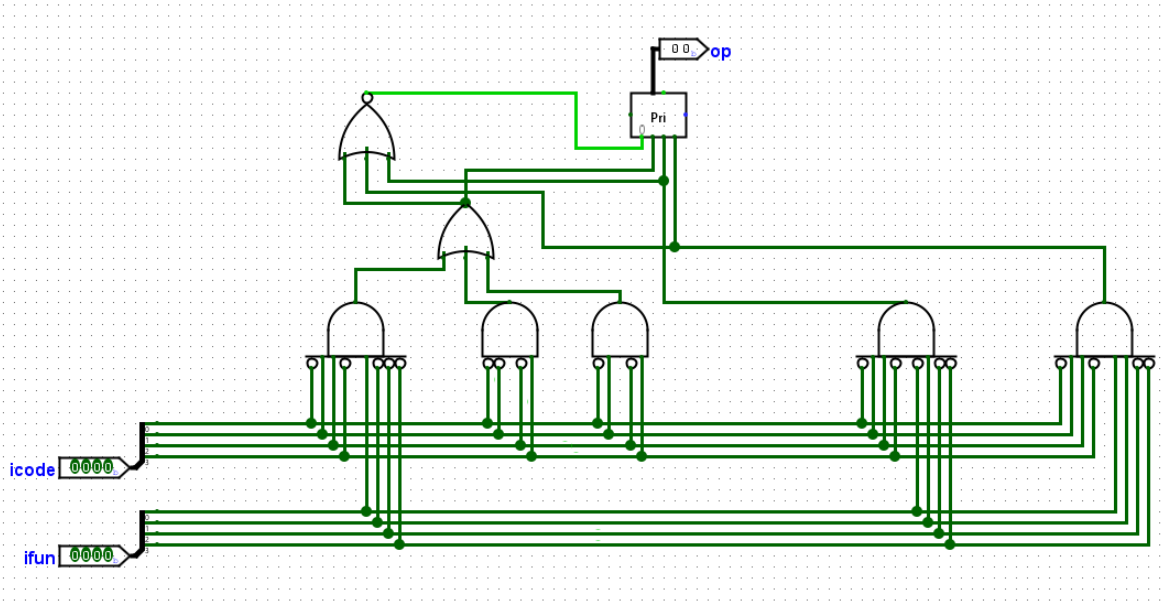
6.2 ALU B

ALUB performs the same way as ALUA, but iCode determines whether 0 or valB is outputted to val2. For instance, if iCode is 4, val2 receives valB's current value.



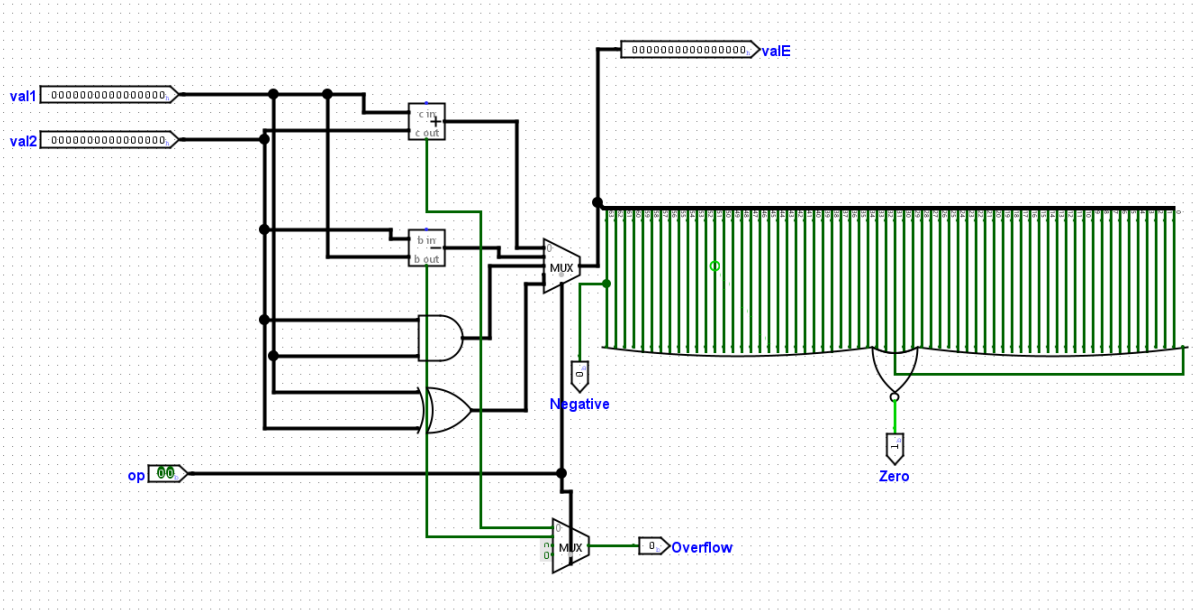
6.3 ALU Function

ALUfun takes in the values of iCode and iFun and provides a 2-bit output to tell the ALU which operation to perform. When iCode is 6 and iFun is 0, an addition operation is performed. When iCode is 6 and iFun is 1, a subtraction operation is performed, so on and so forth for each value of iCode and iFun.



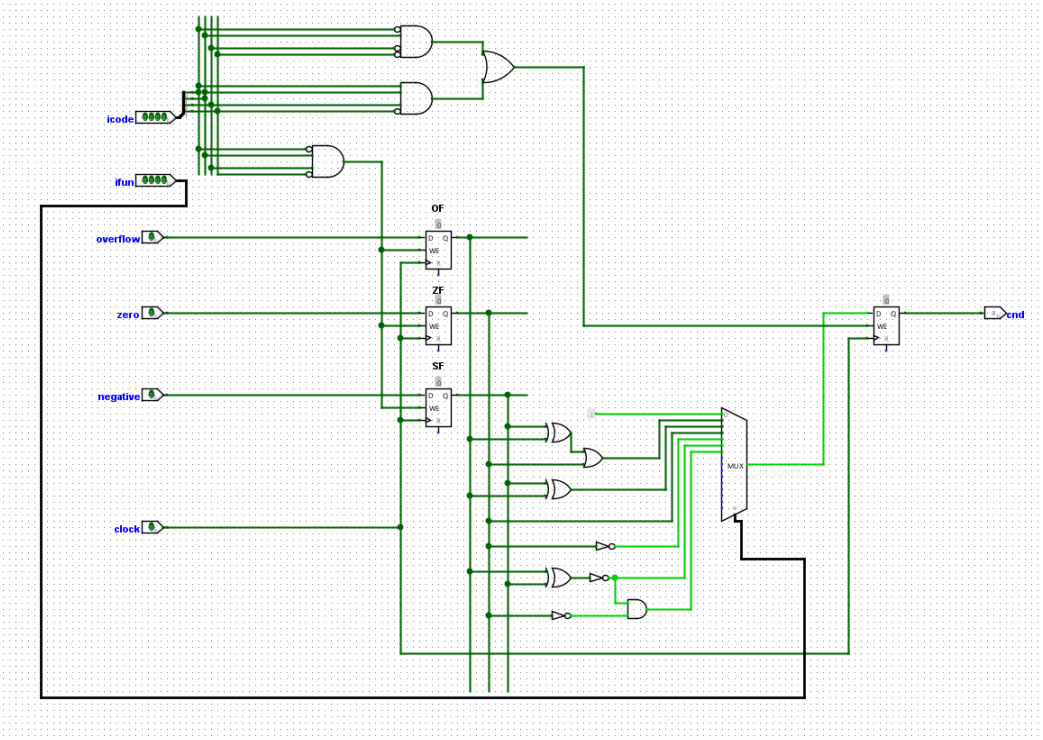
6.4 Arithmetic Logic Unit

The ALU circuit combines all values found using iCode and iFun and outputs the value after the operation is completed. ALU will perform all calculations no matter what value of iCode and iFun is taken in, but will only output the needed value for each required operation. This means that when "addq rA, rB" is called, it performs addition, subtraction, multiplication, and XOR, but will only output the addition value.



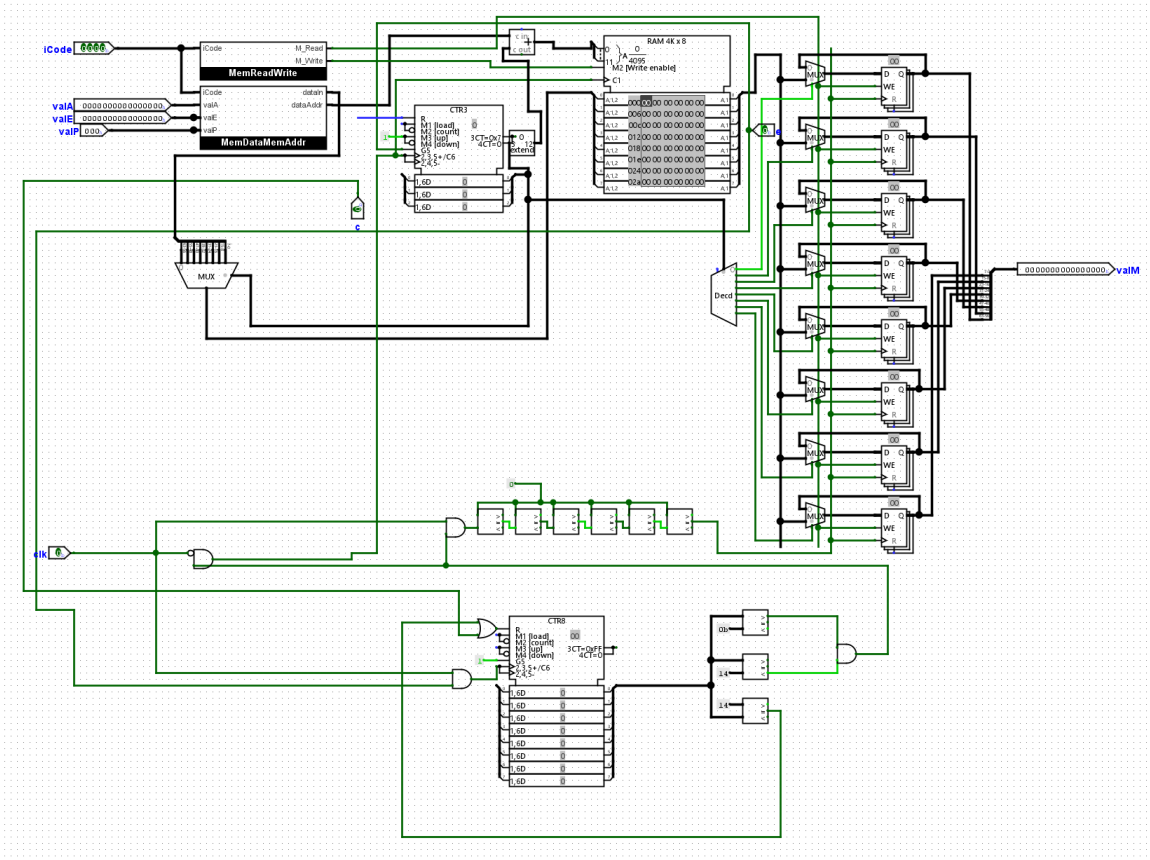
6.5 CC

Lastly, CC determines if a flag is computed, which can be a Zero Flag, Sign Flag, and Overflow Flag. A zero flag occurs when the most recent operation yields a 0, a sign flag occurs when the most recent operation yields a negative value, and an overflow flag occurs when the most recent operation causes a two's complement overflow. This happens when ALU A and ALU B both output a value less than zero and the output computed is more than zero. CC is used to determine which jumps happen, say for instance, a "jle" line is written, if the zero flag outputs a 0, then the jump will occur.



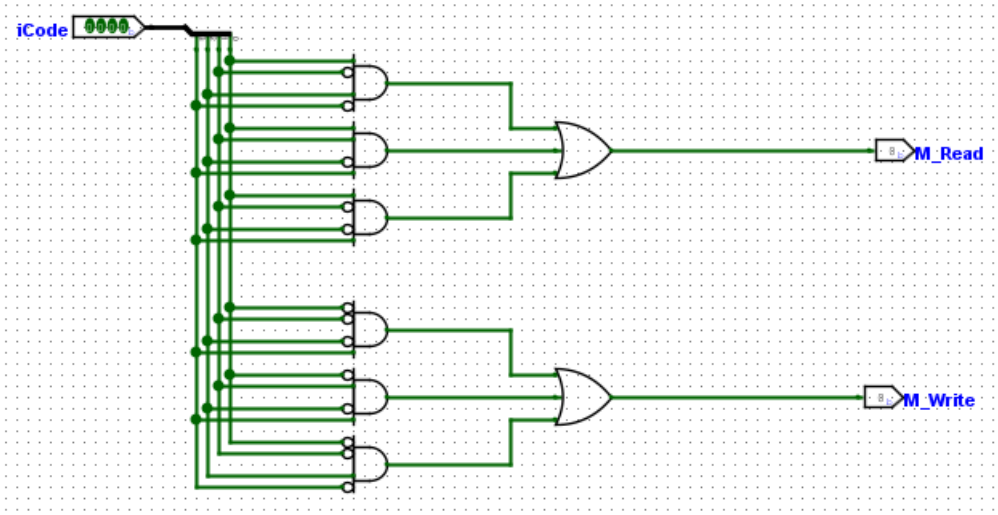
7 Memory Implementation

The memory accesses values inside the RAM and writes the data to registers to be used for processes. You also can move data from registers to memory to be used later on. The counter is used to enable the registers to read and write data during specific clock cycles



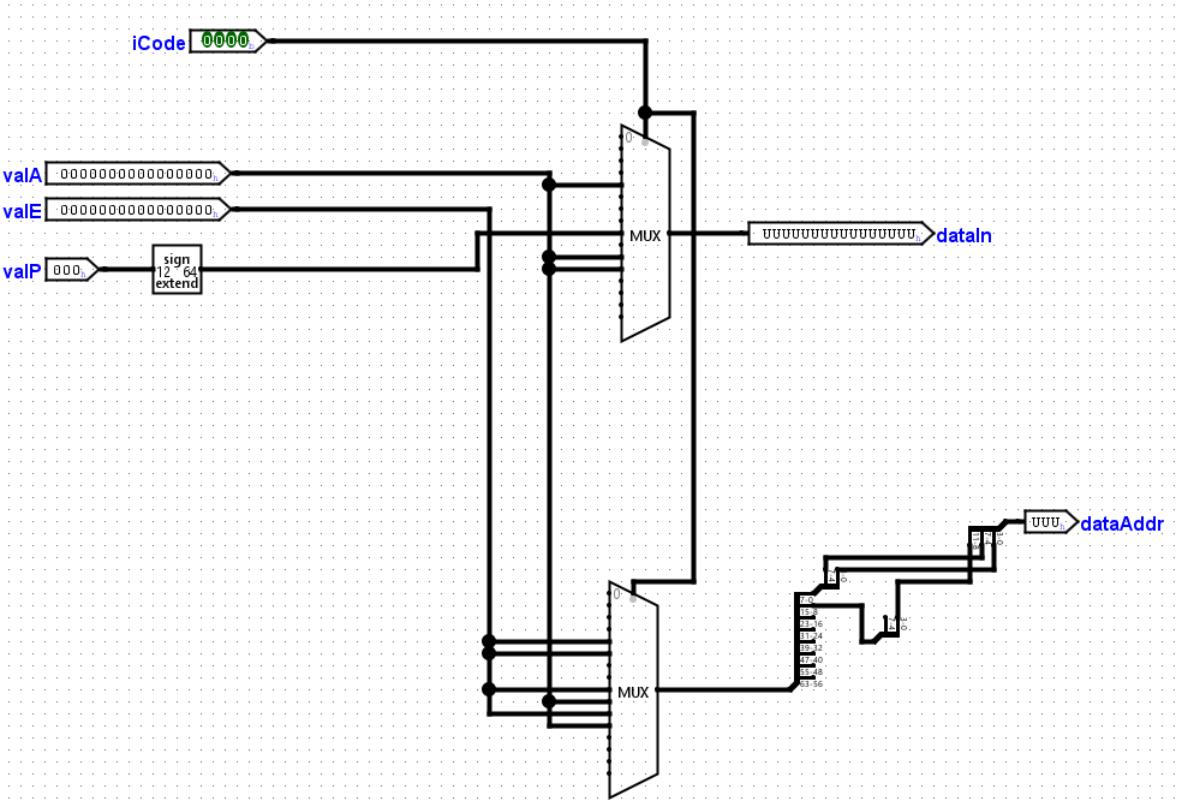
7.1 Memory Read/Write

The memory read and write circuit takes in the iCode value and determines if we are reading data from memory or writing data from memory. The circuit uses AND and OR gates to decide whether or not to read or write data.



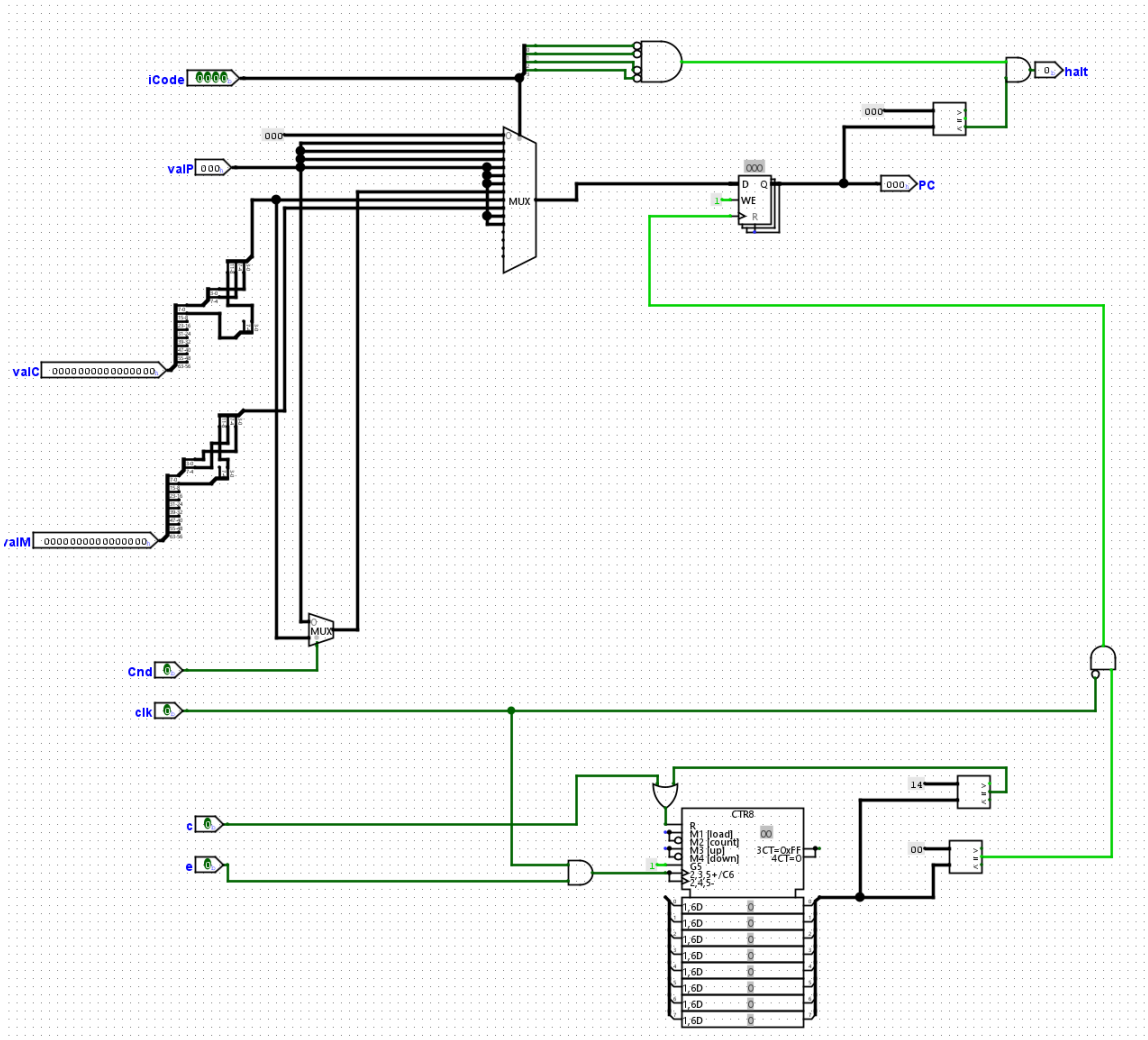
7.2 Memory Data and Memory Address

The memory data and memory address circuit uses iCode, valA, valE, and valP to find the data and the address where we will write the data. ICode is used as a selector bit for both multiplexers, and the data address is found by taking the first 12 bits of the output for the memory address.



8 PC Update

Although the wiring is one of the least complex circuits, this part is arguably one of the most important to the overall functionality to the CPU. Without a correct PC update circuit, the following steps will cause the CPU to perform incorrect calculations and processes for the rest of the code. PC update receives the iCode value, valP, valC, valM, and Cnd to figure out which value needs to be outputted to the PC value. ICode is used as a selector value in a multiplexer to determine which output is needed for PC. When iCode is 0, a HALT function was called; therefore, another 0 value is outputted to PC. When iCode is 1-6, or 10, 11, PC receives valP, if iCode is 8, then PC will receive valC, and if iCode is 9, then PC will receive valM. In certain cases, when iCode is 7, which correlates to jXX, the value of CND will determine whether valC or valP is outputted to PC. When CND is 0, valP is outputted and when CND is 1, valC is outputted.



9 Test Programs

9.1 Palindrome

We created a program that detects palindromes. The commented code is as follows:

```
#include <stdio.h>
int main(){
    int n;
    scanf("%d",&n);

    //initialize array
    int arr[n];
    //read in input
    for(int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    int ok = 1;
    int i = n - 1;
    int lo = 0;
```

```
    while(i >= 0) {
        if(arr[lo] != arr[i]) ok = 0;
        lo++;
        i--;
    }

    //optional printing
    printf("%d\n",ok);

    return 0;
}
```

Moreover, this translated into Y86 is:

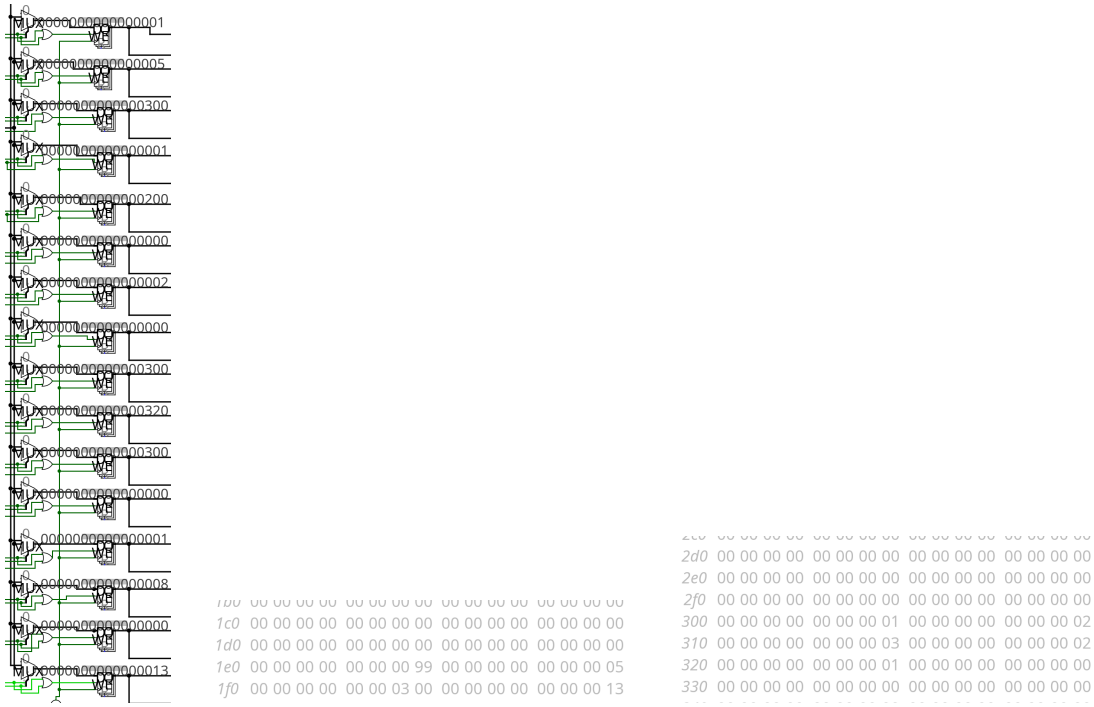
```
    .pos 0
    irmovq stack, %rsp
    call main
    halt

main:
    irmovq $5, %rcx #size of the array we are going to sort
    irmovq $0x300, %rdx #pointer for array
    #populate the array with values
    irmovq $1, %rbx
    rmmovq %rbx, 0(%rdx)
    irmovq $2, %rbx
    rmmovq %rbx, 8(%rdx)
    irmovq $3, %rbx
    rmmovq %rbx, 16(%rdx)
    irmovq $2, %rbx
    rmmovq %rbx, 24(%rdx)
    irmovq $1, %rbx
    rmmovq %rbx, 32(%rdx)
    pushq %rdx
    pushq %rcx

    call ispal
    popq %r8
    popq %r8
    ret

ispal:
    mrmovq 8(%rsp), %r8    #store size into register 8
    mrmovq 16(%rsp), %r9   #store pointer of array to register 9

    #set up constants
    irmovq $1, %rax
    rrmovq %rax, %rax
    irmovq $0, %r11
    irmovq $1, %r12
    rrmovq %r12, %r12
    irmovq $8, %r13

    #set up hi pointer
    mrmovq 16(%rsp), %r10   #hi pointer
increasehi:
```

```
    subq %r12, %r8
    rrmovq %r8, %r8
    je next
    addq %r13, %r10
    jmp increasehi
next:
    mrmovq 8(%rsp), %r8
loop:
    subq %r12, %r8
    je ispaldone
    mrmovq (%r9), %rsi
    rrmovq %rsi, %rsi
    mrmovq (%r10), %rdi
    subq %rsi, %rdi
    je ok
    irmovq $0, %rax
ok:
    addq %r13, %r9
    subq %r13, %r10
    jmp loop

ispaldone:
    ret

    .pos 0x200
stack:
```

Basically we do the same thing, but in our Y86 program we load our palindrome into memory instead of taking it from standard input. If we run our `yas` program (once compiling it to the object file we get):

```
Changes to registers:
%rax:   0x0000000000000000      0x0000000000000001
%rcx:   0x0000000000000000      0x0000000000000005
%rdx:   0x0000000000000000      0x0000000000000300
%rbx:   0x0000000000000000      0x0000000000000001
%rsp:   0x0000000000000000      0x0000000000000200
%rsi:   0x0000000000000000      0x0000000000000002
%r8:    0x0000000000000000      0x0000000000000300
%r9:    0x0000000000000000      0x0000000000000320
%r10:   0x0000000000000000      0x0000000000000300
%r12:   0x0000000000000000      0x0000000000000001
%r13:   0x0000000000000000      0x0000000000000008

Changes to memory:
0x01e0: 0x0000000000000000      0x0000000000000099
0x01e8: 0x0000000000000000      0x0000000000000005
0x01f0: 0x0000000000000000      0x0000000000000300
0x01f8: 0x0000000000000000      0x0000000000000013
0x0300: 0x0000000000000000      0x0000000000000001
0x0308: 0x0000000000000000      0x0000000000000002
0x0310: 0x0000000000000000      0x0000000000000003
0x0318: 0x0000000000000000      0x0000000000000002
0x0320: 0x0000000000000000      0x0000000000000001
```

When we run this program on our registers and mem look like this:



Yay everything works!

9.2 Sort

The second program we created was a sorting program. Here is the program in C:

```
#include <stdio.h>

int main()
{
    //read in length of input
    int n;
    scanf("%d",&n);

    //initialize array
    int arr[n];
    //read in input
    for(int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }


    //run some sort O(n^2) sort. names are hard.
    for(int i = 0; i < n; i++) {
        for(int j = 1; j < n; j++) {
            if(arr[j-1] > arr[j]) {
                int t = arr[j-1];
                arr[j-1] = arr[j];
                arr[j] = t;
            }
        }
    }
    return 0;
```

Moreover, this transalted into Y86 is:

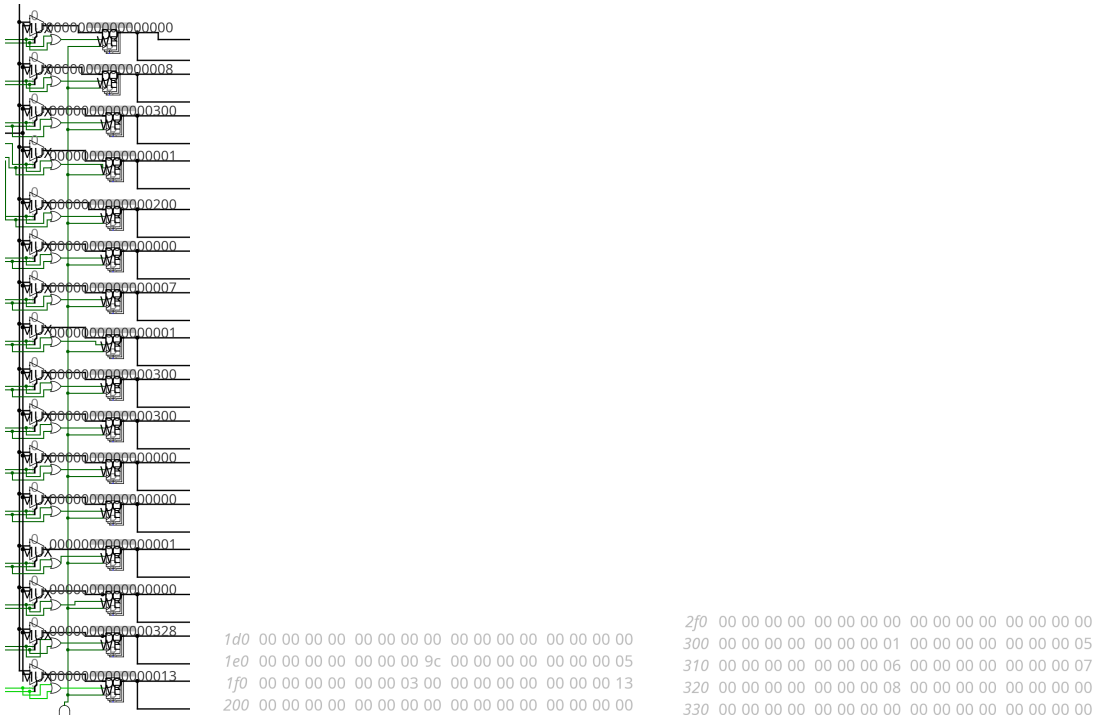
```
    .pos 0
    irmovq stack, %rsp
    call main
    halt


main:
    irmovq $5, %rcx #size of the array we are going to sort
    irmovq $0x300, %rdx #pointer for array
    #populate the array with values
    irmovq $7, %rbx
    rmmovq %rbx, 0(%rdx)
    irmovq $5, %rbx
    rmmovq %rbx, 8(%rdx)
    irmovq $6, %rbx
    rmmovq %rbx, 16(%rdx)
    irmovq $8, %rbx
    rmmovq %rbx, 24(%rdx)
    irmovq $1, %rbx
    rmmovq %rbx, 32(%rdx)
    pushq %rdx
    pushq %rcx

    nop
    nop
    nop

    call sort
    popq %r8
    popq %r8

    ret

sort:
    mrmovq 8(%rsp), %r8    #store size into register 8
    mrmovq 16(%rsp), %r9   #store pointer of array to register 9
    mrmovq 8(%rsp), %r10
    irmovq $0, %r11         #store constant 0
    irmovq $1, %r12         #store constant 1
    irmovq $8, %rcx

outerloop:
    subq %r11, %r10  #check if i == 0
    je sortdone     #if so, we are done
    subq %r12, %r10 #otherwise decrease i and go to inner loop

innerloop:
    mrmovq 8(%rsp), %r13
    mrmovq 16(%rsp), %r14

innerloopstart:
    addq %rcx, %r14 #j8 += 8
    subq %r12, %r13 #j--
```

```
    je outerloop    #if j - size == 0, go back to outer loop

    #compare elements at indices j - 1 and j
    subq %rcx, %r14
    mrmovq (%r14), %rsi
    mrmovq 8(%r14), %rdi
    addq %rcx, %r14
    subq %rsi, %rdi #CC = a[j] - a[j-1]
    jge innerloopstart #if >= , then we don't swap

    #we swap if we make it here
    addq %rsi, %rdi

    rmmovq %rsi, (%r14)
    subq %rcx, %r14
    rmmovq %rdi, (%r14)
    addq %rcx, %r14
    jmp innerloopstart

sortdone:
    ret

    .pos 0x200
stack:
```

To implement this in Y86 we implement it using jmp commands along with calls and return. It is almost one to one compared to our C code, just implmented in Y86. When we compile and run it (using `yis`) we get:

```
%rcx:   0x0000000000000000      0x0000000000000008
%rdx:   0x0000000000000000      0x0000000000000300
%rbx:   0x0000000000000000      0x0000000000000001
%rsp:   0x0000000000000000      0x0000000000000200
%rsi:   0x0000000000000000      0x0000000000000007
%rdi:   0x0000000000000000      0x0000000000000001
%r8:    0x0000000000000000      0x0000000000000300
%r9:    0x0000000000000000      0x0000000000000300
%r12:   0x0000000000000000      0x0000000000000001
%r14:   0x0000000000000000      0x0000000000000328

Changes to memory:
0x01e0: 0x0000000000000000      0x000000000000009c
0x01e8: 0x0000000000000000      0x0000000000000005
0x01f0: 0x0000000000000000      0x0000000000000300
0x01f8: 0x0000000000000000      0x0000000000000013
0x0300: 0x0000000000000000      0x0000000000000001
0x0308: 0x0000000000000000      0x0000000000000005
0x0310: 0x0000000000000000      0x0000000000000006
0x0318: 0x0000000000000000      0x0000000000000007
0x0320: 0x0000000000000000      0x0000000000000008
```

When we run it on our CPU we get:



Yay everything works!

9.3 Matrix Multiply

Finally, we implement matrix multiply. The C code can be formulated the following way:

```
int main()
{
    //read in dimensions of matrix
    //initialize initialize dimensions of matrices
    int m[2][2] = {{1,2},{3,4}};
    int n[2][2] = {{5,6},{7,8}};
    int nm[2][2];

    nm[0][0] = m[0][0]*n[0][0] + m[0][1]*n[1][0];
    nm[0][1] = m[0][0]*n[0][1] + m[0][1]*n[1][1];
    nm[1][0] = m[1][0]*n[0][0] + m[1][1]*n[1][0];
    nm[1][1] = m[1][0]*n[0][1] + m[1][1]*n[1][1];

    return 0;
}
```

We have equivalent Y86 code as:

```
    .pos 0

    irmovq stack, %rsp
    call main
    halt

main:
    irmovq $0x400, %rdx #pointer for array
    #populate the array with values
    irmovq $1, %rbx
    rmmovq %rbx, 0(%rdx)
```

```
    irmovq $2, %rbx
    rmmovq %rbx, 8(%rdx)
    irmovq $3, %rbx
    rmmovq %rbx, 16(%rdx)
    irmovq $4, %rbx
    rmmovq %rbx, 24(%rdx)
    irmovq $5, %rbx
    rmmovq %rbx, 32(%rdx)
    irmovq $6, %rbx
    rmmovq %rbx, 40(%rdx)
    irmovq $7, %rbx
    rmmovq %rbx, 48(%rdx)
    irmovq $8, %rbx
    rmmovq %rbx, 56(%rdx)

    pushq %rdx
    call matrixmultiply
    popq %r8
    ret

matrixmultiply:

    mrmovq 8(%rsp), %rbx   #pointer to first matrix

    irmovq $0, %rax

    rmmovq %rax, 64(%rbx)
    rmmovq %rax, 72(%rbx)
    rmmovq %rax, 80(%rbx)
    rmmovq %rax, 88(%rbx)

    mrmovq (%rbx), %rcx
    pushq %rcx
    mrmovq 32(%rbx), %rcx
    pushq %rcx
    call multiply
    popq %r13
    popq %r13
    mrmovq 64(%rbx), %rcx
    addq %rax, %rcx
    rmmovq %rcx, 64(%rbx)

    mrmovq 8(%rbx), %rcx
    pushq %rcx
    mrmovq 48(%rbx), %rcx
    pushq %rcx
    call multiply
    popq %r13
    popq %r13
    mrmovq 64(%rbx), %rcx
    addq %rax, %rcx
    rmmovq %rcx, 64(%rbx)

    mrmovq (%rbx), %rcx
```

```
pushq %rcx
mrmovq 40(%rbx), %rcx
pushq %rcx
call multiply
popq %r13
popq %r13
mrmovq 72(%rbx), %rcx
addq %rax, %rcx
rmmovq %rcx, 72(%rbx)

mrmovq 8(%rbx), %rcx
pushq %rcx
mrmovq 56(%rbx), %rcx
pushq %rcx
call multiply
popq %r13
popq %r13
mrmovq 72(%rbx), %rcx
addq %rax, %rcx
rmmovq %rcx, 72(%rbx)

mrmovq 16(%rbx), %rcx
pushq %rcx
mrmovq 32(%rbx), %rcx
pushq %rcx
call multiply
popq %r13
popq %r13
mrmovq 80(%rbx), %rcx
addq %rax, %rcx
rmmovq %rcx, 80(%rbx)

mrmovq 24(%rbx), %rcx
pushq %rcx
mrmovq 48(%rbx), %rcx
pushq %rcx
call multiply
popq %r13
popq %r13
mrmovq 80(%rbx), %rcx
addq %rax, %rcx
rmmovq %rcx, 80(%rbx)

mrmovq 16(%rbx), %rcx
pushq %rcx
mrmovq 40(%rbx), %rcx
pushq %rcx
call multiply
popq %r13
popq %r13
mrmovq 88(%rbx), %rcx
addq %rax, %rcx
rmmovq %rcx, 88(%rbx)
```

```
    mrmovq 24(%rbx), %rcx
    pushq %rcx
    mrmovq 56(%rbx), %rcx
    pushq %rcx
    call multiply
    popq %r13
    popq %r13
    mrmovq 88(%rbx), %rcx
    addq %rax, %rcx
    rmmovq %rcx, 88(%rbx)

    ret

multiply:
    mrmovq 8(%rsp), %r8
    mrmovq 16(%rsp), %r9
    irmovq $0, %rax         #set result to 0
    irmovq $0, %r10         #register used for constant 0
    irmovq $1, %r11         #register used for constant 1

multiplyloopa:
    subq %r10, %r8 #check if a != 0
    je multiplydone   #if a = 0, we are done
    subq %r11, %r8 #decrease a by 1.
    addq %r9, %rax
    jne multiplyloopa

multiplydone:
    ret

    .pos 0x5f8
stack:
```

When we run our Y86 object file we get these results:

```
%rax:   0x0000000000000000      0x0000000000000020
%rcx:   0x0000000000000000      0x0000000000000032
%rdx:   0x0000000000000000      0x0000000000000400
%rbx:   0x0000000000000000      0x0000000000000400
%rsp:   0x0000000000000000      0x00000000000005f8
%r8:    0x0000000000000000      0x0000000000000400
%r9:    0x0000000000000000      0x0000000000000004
%r11:   0x0000000000000000      0x0000000000000001
%r13:   0x0000000000000000      0x0000000000000004

Changes to memory:
0x0400: 0x0000000000000000      0x0000000000000001
0x0408: 0x0000000000000000      0x0000000000000002
0x0410: 0x0000000000000000      0x0000000000000003
0x0418: 0x0000000000000000      0x0000000000000004
0x0420: 0x0000000000000000      0x0000000000000005
0x0428: 0x0000000000000000      0x0000000000000006
0x0430: 0x0000000000000000      0x0000000000000007
0x0438: 0x0000000000000000      0x0000000000000008
0x0440: 0x0000000000000000      0x0000000000000013
0x0448: 0x0000000000000000      0x0000000000000016
```

```
0x0450: 0x0000000000000000      0x000000000000002b
0x0458: 0x0000000000000000      0x0000000000000032
0x05c8: 0x0000000000000000      0x00000000000002c6
0x05d0: 0x0000000000000000      0x0000000000000008
0x05d8: 0x0000000000000000      0x0000000000000004
0x05e0: 0x0000000000000000      0x00000000000000c9
0x05e8: 0x0000000000000000      0x0000000000000400
0x05f0: 0x0000000000000000      0x0000000000000013
```

When running this on our CPU we also get these results:

Once again, everything works out.

9.4 Code Tests Summary

In summary, all of our tests work as expected. We test all of our instructions, and they all follow what running `yis` on our object files outputs. Specifically for the honors requirement, we utilize a multiply function in our matrix multiply code (that uses call and return).

9.5 Compiler

We also implemented our own compiler. We modified the one given to us to work with big endian as this is how we built our CPU.