# Group 18: CSCE 312 CPU Project — Final Report

**Lucian Chauvin**
133003371

**Joshua Lass**
531009387

**Bjorn Quarfordt**
230003985

## 1 Transformation Tables

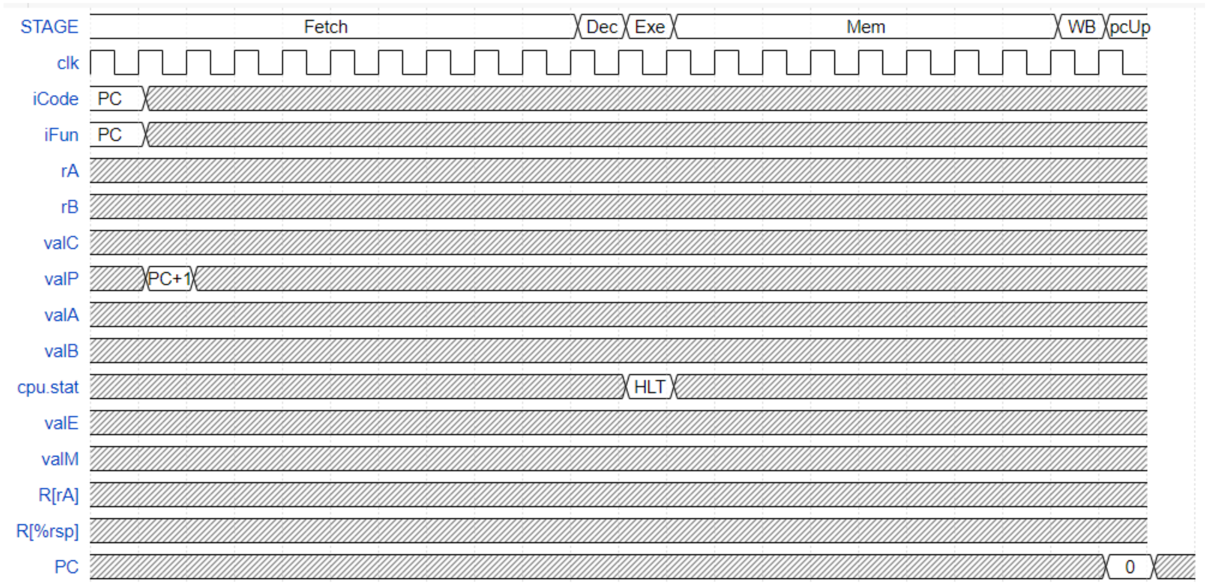
| Instruction | Fetch | Decode | Execute | Memory | Write Back | PC Update |
|---|---|---|---|---|---|---|
| `rrmovq rA, rB` | `icode:ifun← M_1[PC]`<br>`rA:rB ← M_1[PC+1]`<br>`valP← PC+2` | `valA← R[rA]`<br>`valB← R[rB]` | | | `R[rB]← valA` | `PC← valP` |
| `irmovq V, rB` | `icode:ifun ← M_1[PC]`<br>`F:rB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

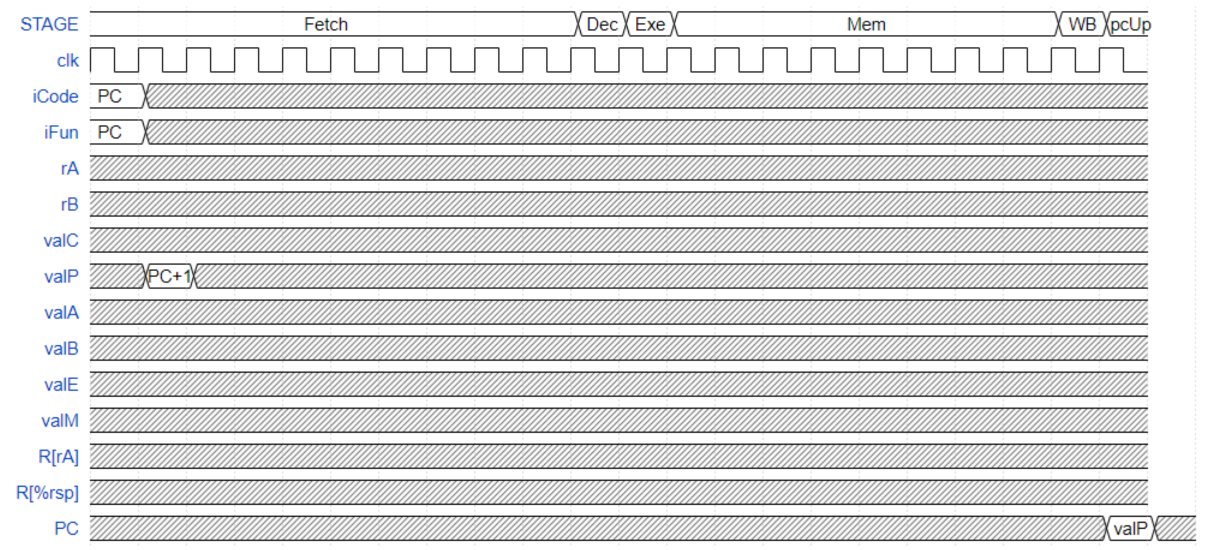
### 2.1 HALT

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



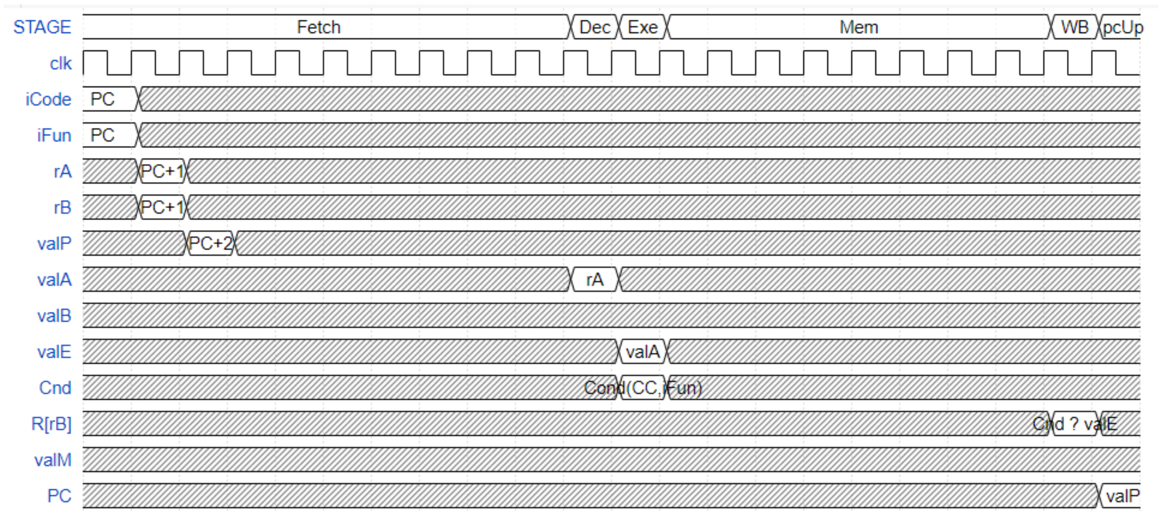
### 2.2 NOP

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.



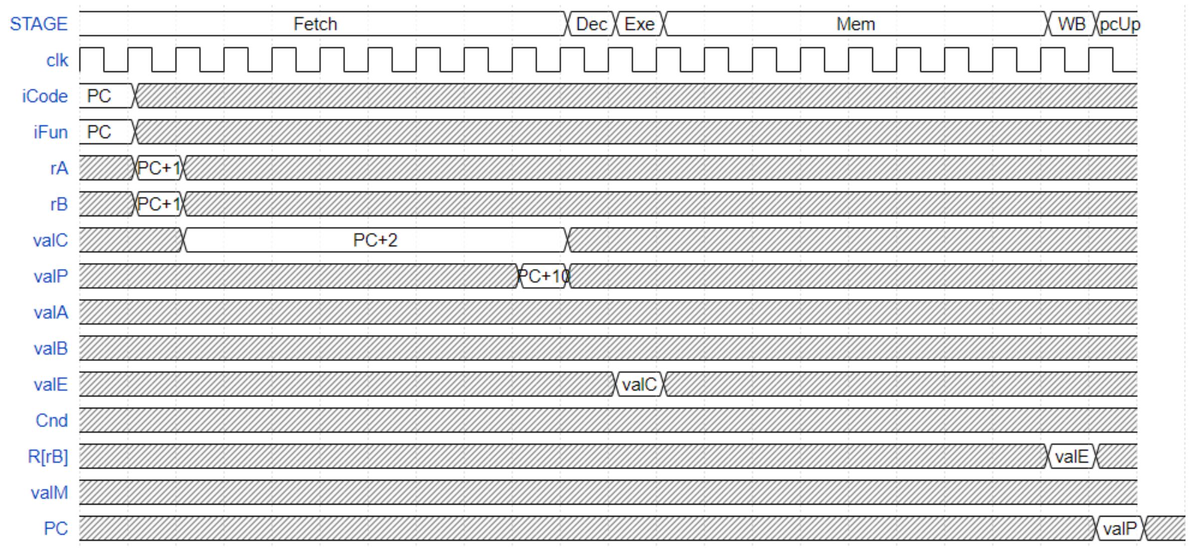
### 2.3 RRMOVQ

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.



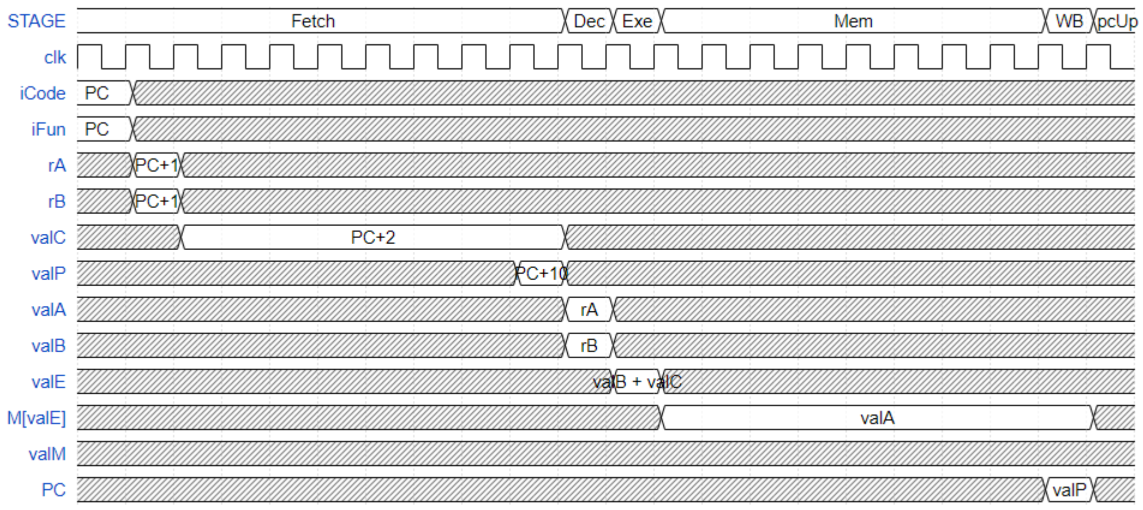
### 2.4 IRMOVQ

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



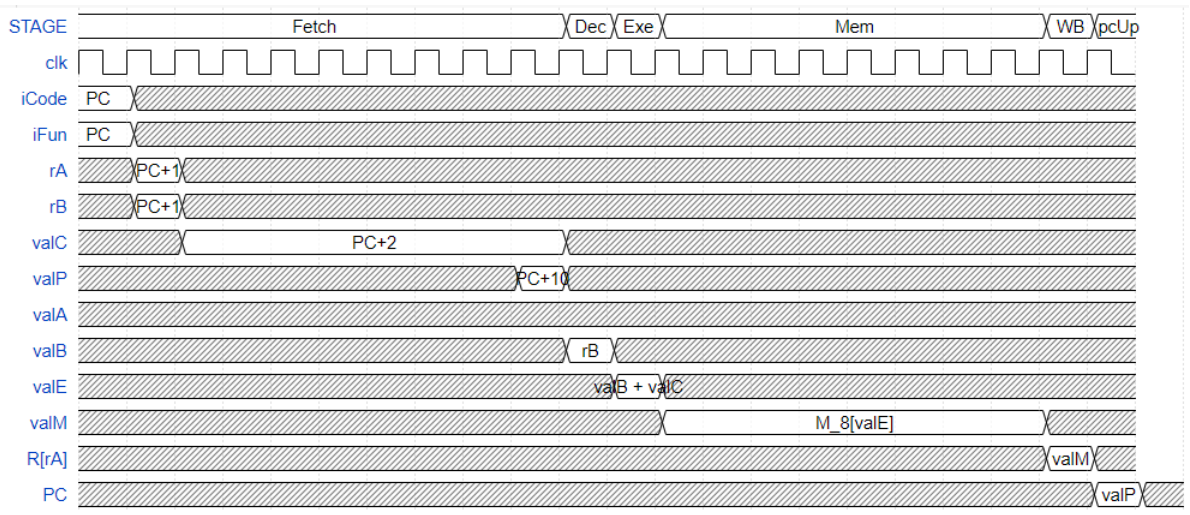
### 2.5 RMMOVQ

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.



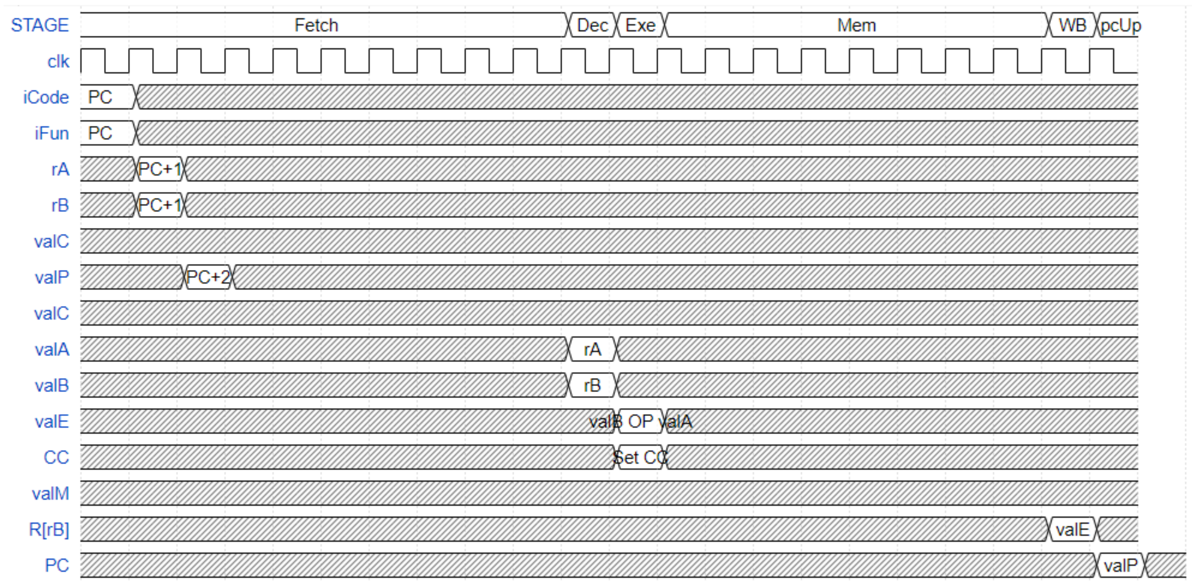
### 2.6 MRMOVQ

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.



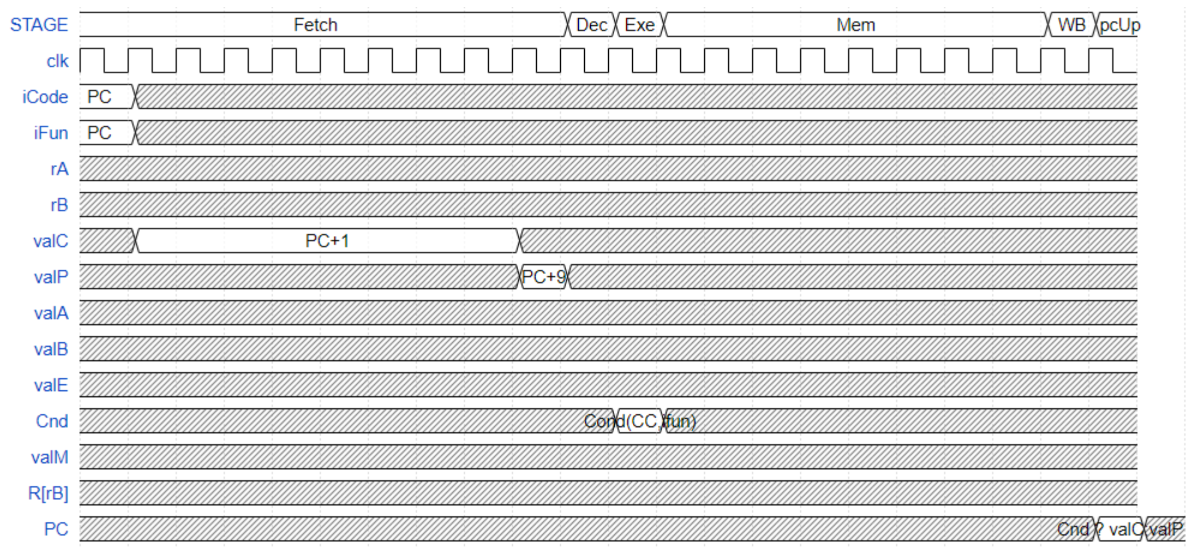
## 2.7 OPq

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



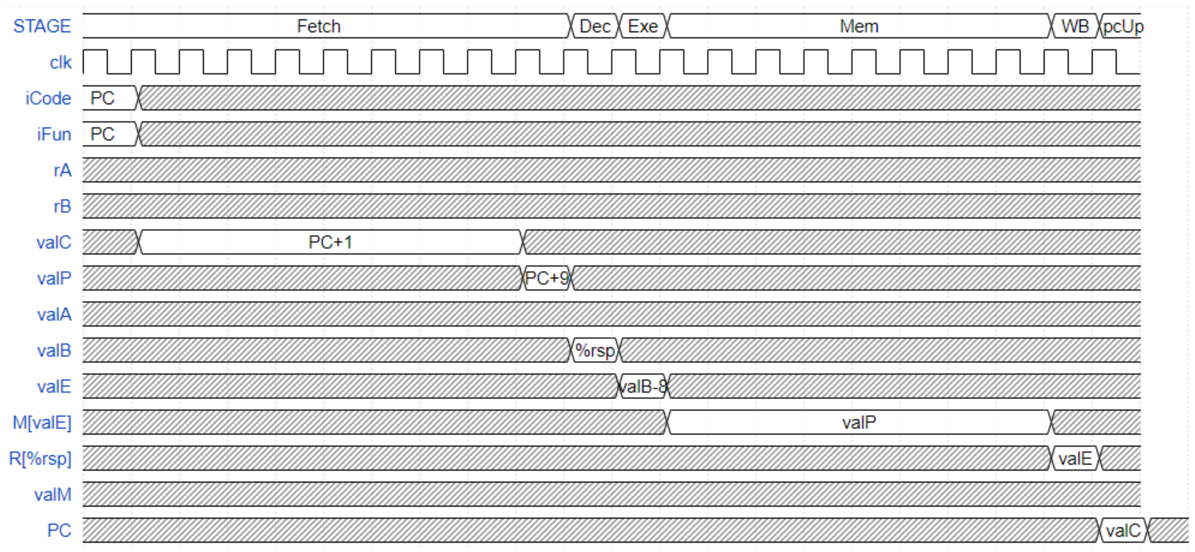
## 2.8 jXX

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.



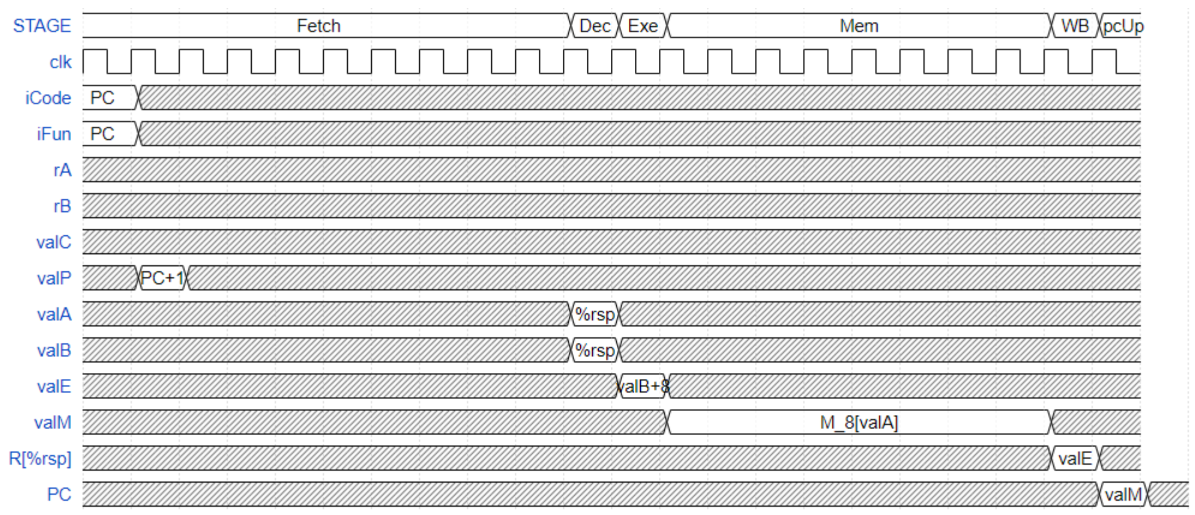
## 2.9 CALL

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



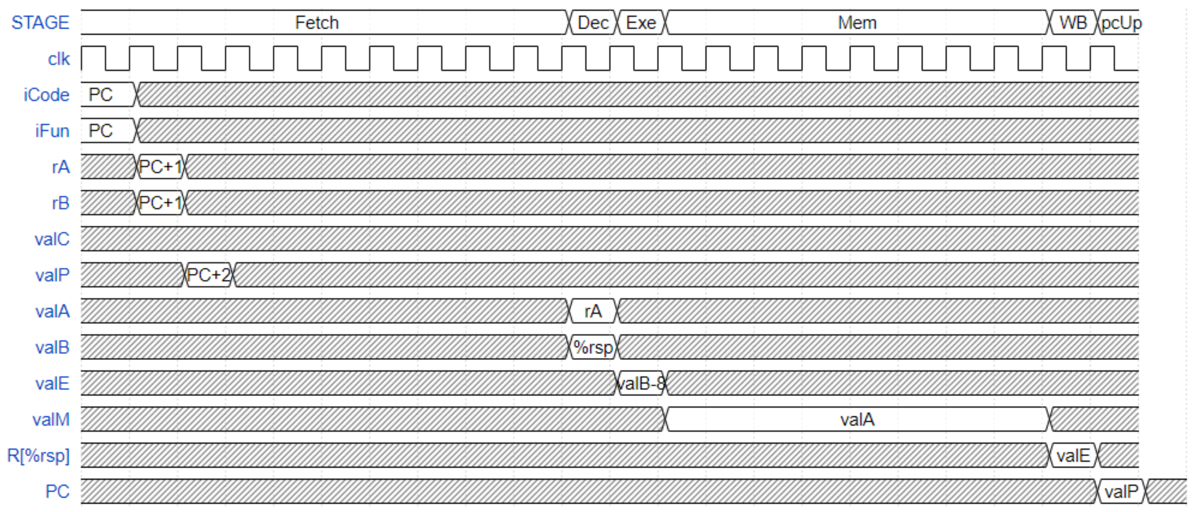
## 2.10 RET

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.



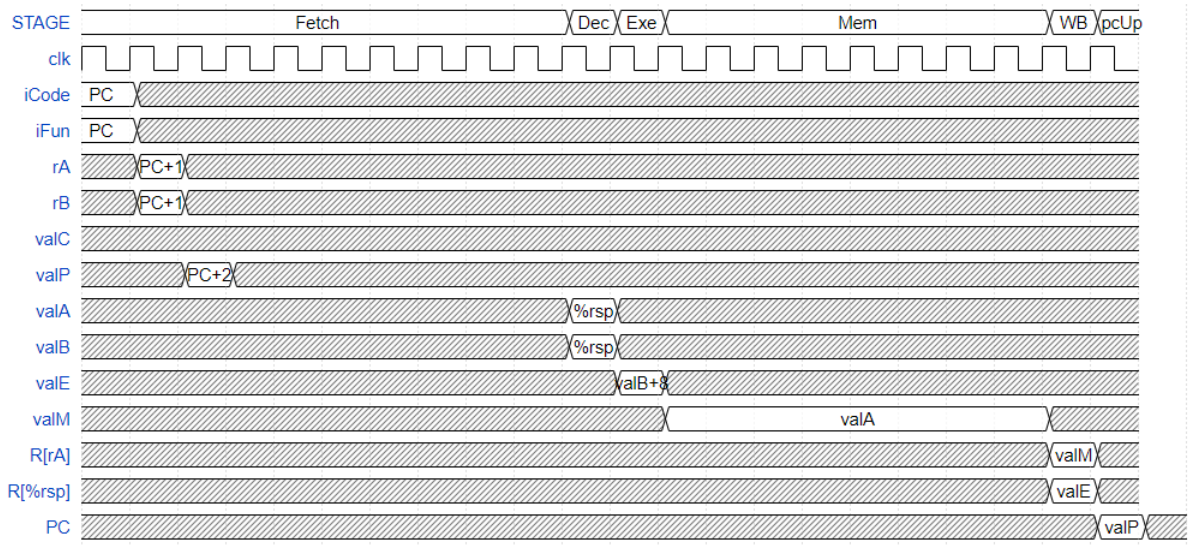
### 2.11 PUSHQ

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

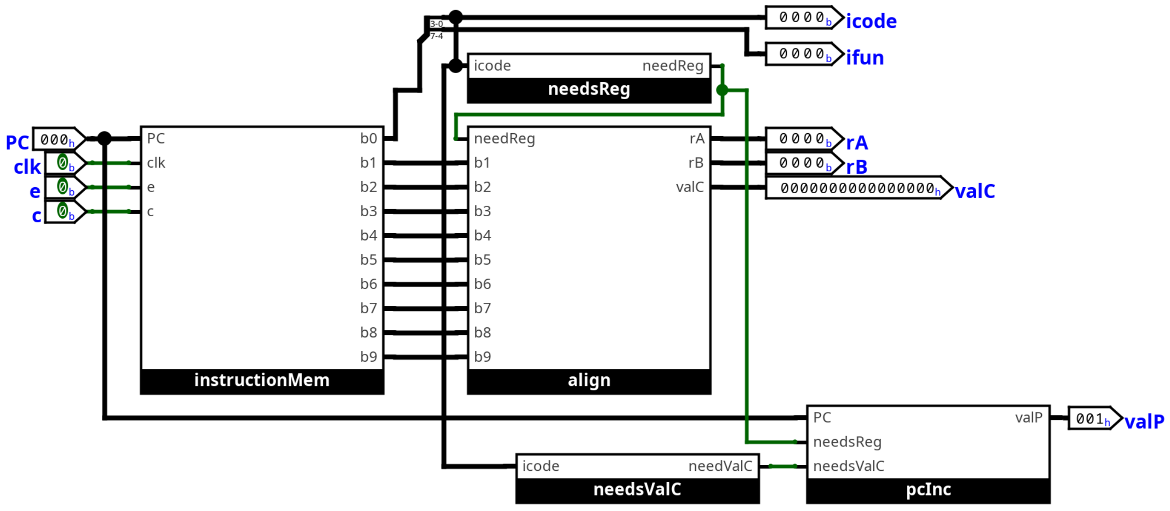
POPQ occurs when iCode is 'b', or 11. POPQ copies the value in the address pointed by %rsp, and increments %rsp by 8 bytes.



## 3 Fetch Implementation

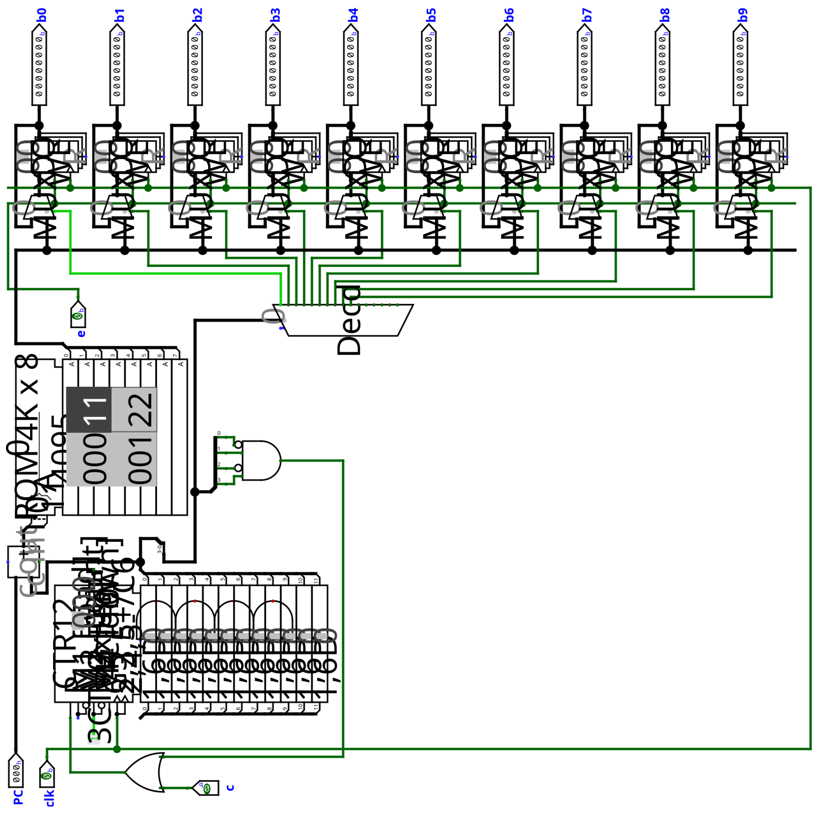
Our design is basically a one-to-one implementation of what is shown on the slides. Our instruction memory module stores the program in ROM and reads 10 bytes from the current `PC`. We then pass these 10 bytes to our align module which — based on if we need registers or not — sets out `rA`, `rB`, and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

### 3.1 Fetch



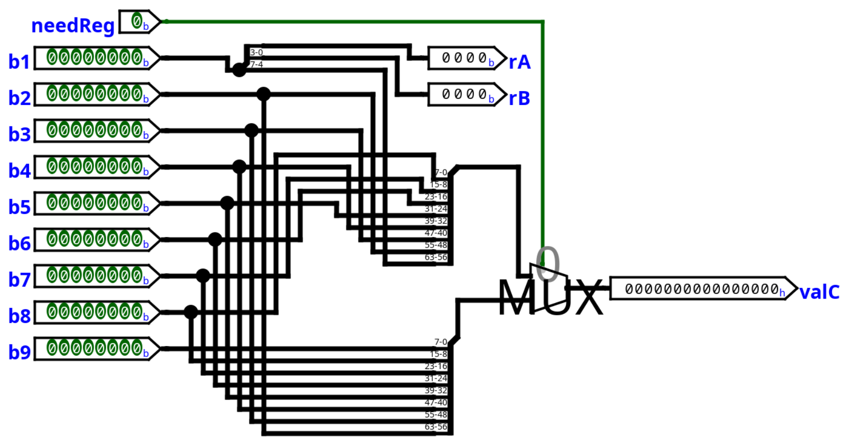
### 3.2 Instruction Memory

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).



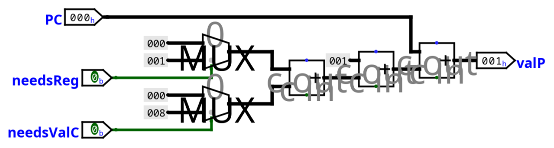
### 3.3 Align

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.



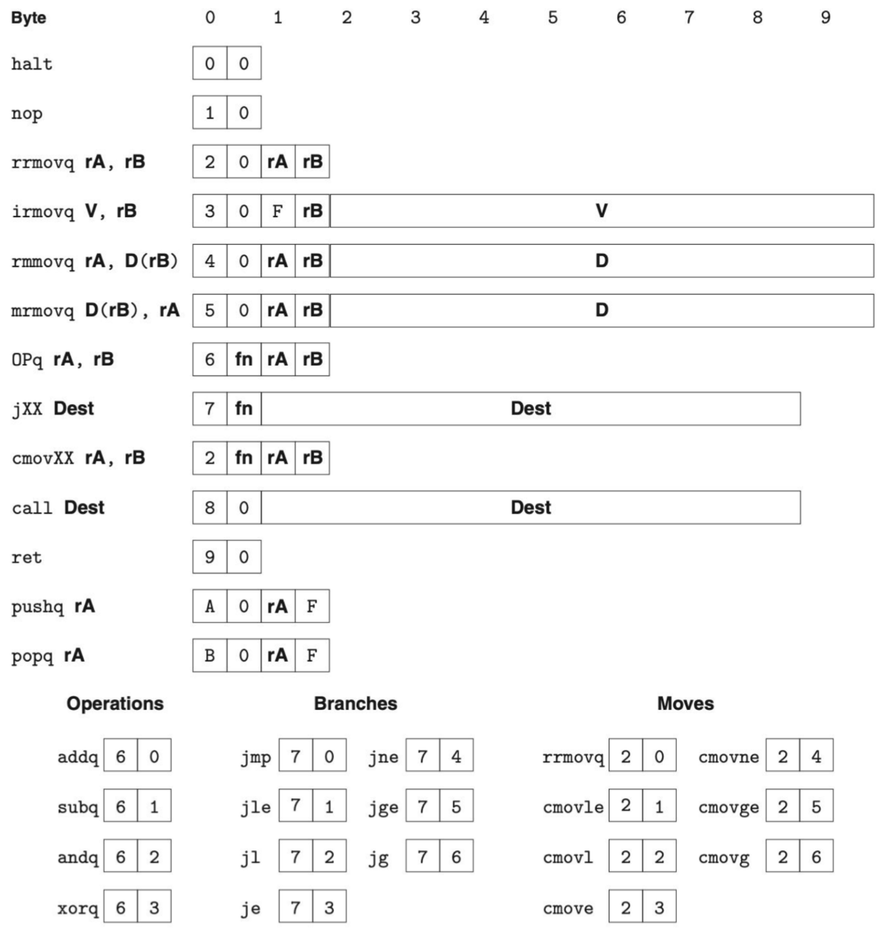
## 3.4 PC Increment

PC increment simply increments the `PC` based on if 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`.



## 3.5 Needs ValC and Registers

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



| 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**

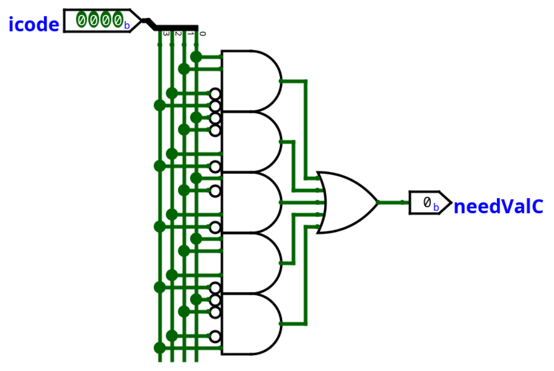
| | | |
|---|---|---|
| `addq` | 6 | 0 |
| `subq` | 6 | 1 |
| `andq` | 6 | 2 |
| `xorq` | 6 | 3 |

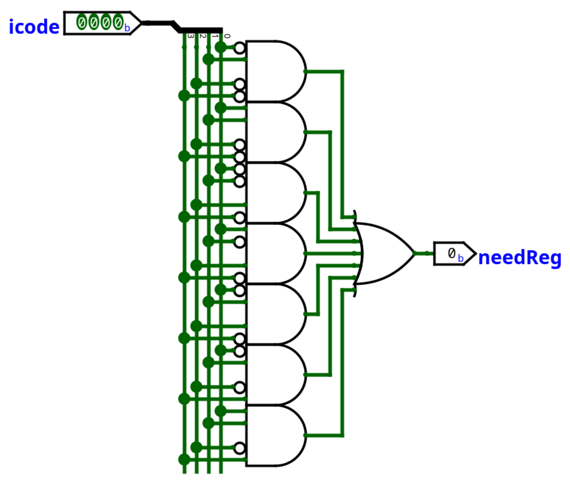
**Branches**

| | | | | | |
|---|---|---|---|---|---|
| `jmp` | 7 | 0 | `jne` | 7 | 4 |
| `jle` | 7 | 1 | `jge` | 7 | 5 |
| `jl` | 7 | 2 | `jg` | 7 | 6 |
| `je` | 7 | 3 | | | |

**Moves**

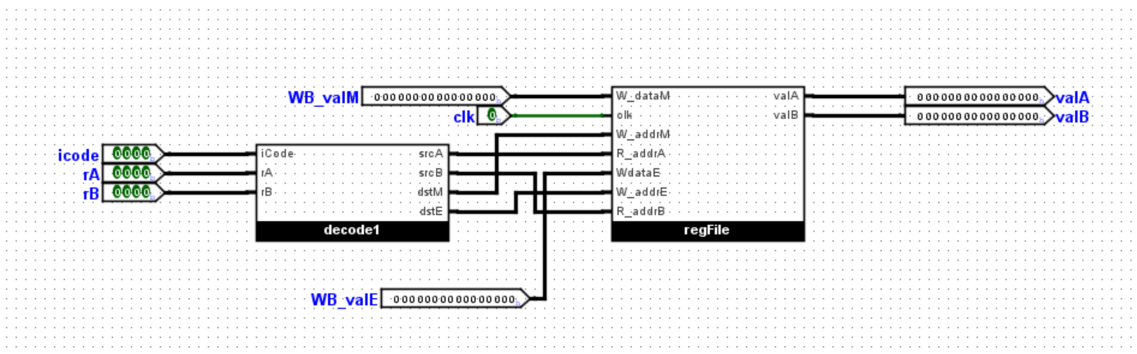
| | | | | | |
|---|---|---|---|---|---|
| `rrmovq` | 2 | 0 | `cmovne` | 2 | 4 |
| `cmovle` | 2 | 1 | `cmovge` | 2 | 5 |
| `cmovl` | 2 | 2 | `cmovg` | 2 | 6 |
| `cmove` | 2 | 3 | | | |





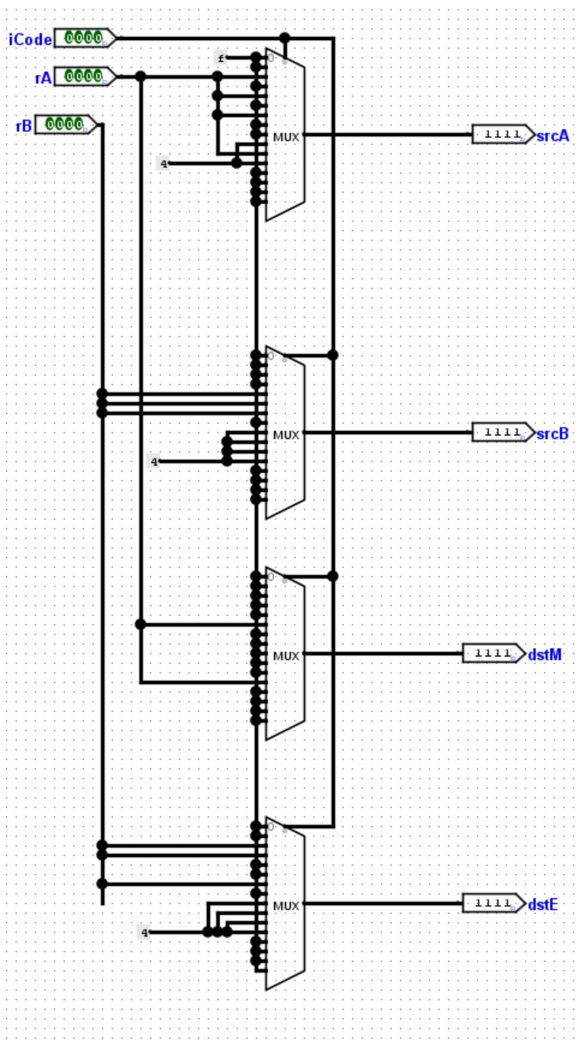
## 4 Decode and Write Back Implementation

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.



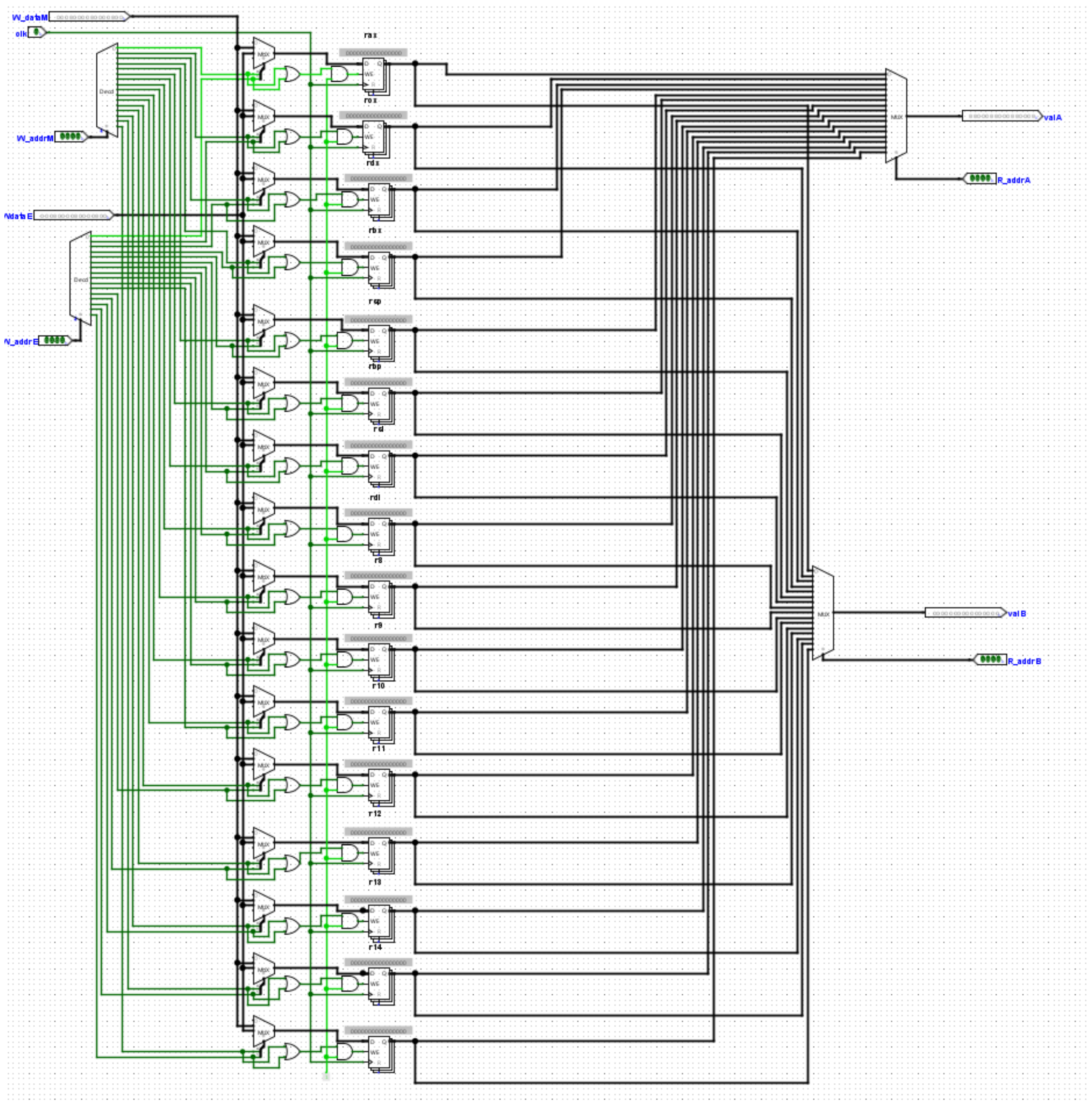
### 4.1 Decode1

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.



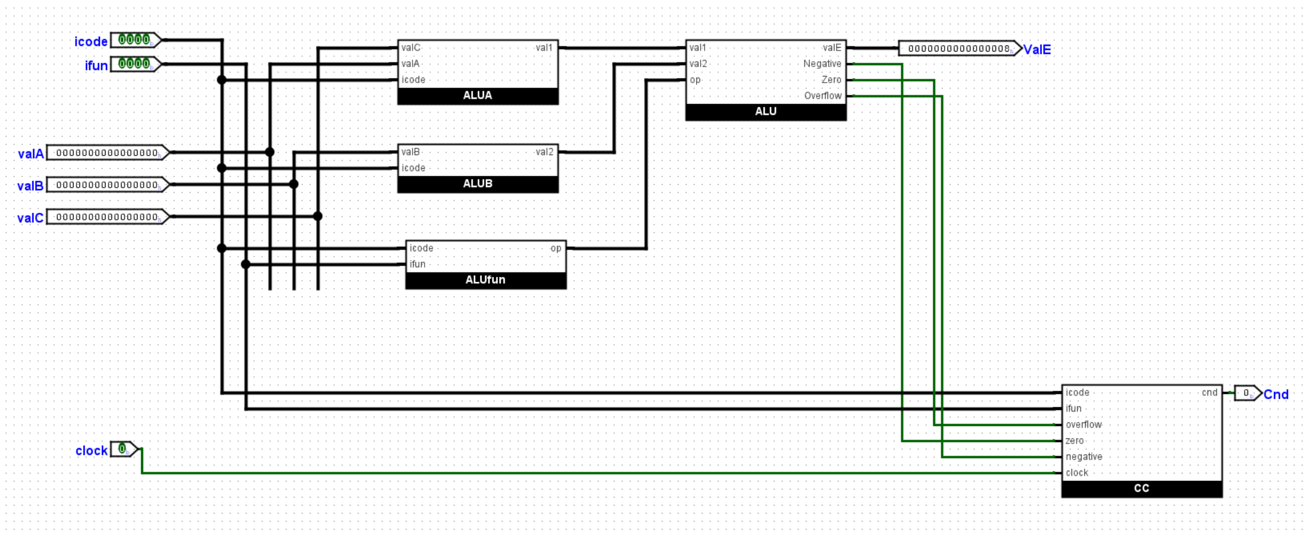
### 4.2 Register File

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.


W_dataM
clk
W_addrM
WdataE
W_addrE
rax
rcx
rdx
rbx
rsp
rbp
rsi
rdi
r8
r9
r10
r11
r12
r13
r14
valA
R_addrA
valB
R_addrB

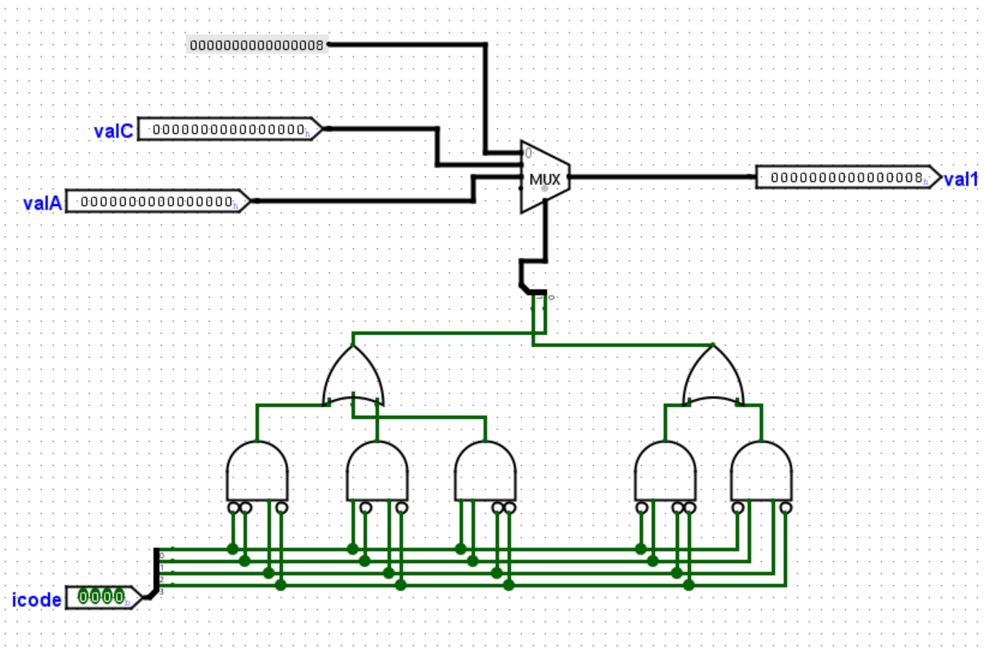
## 5 Execute Implementation

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.



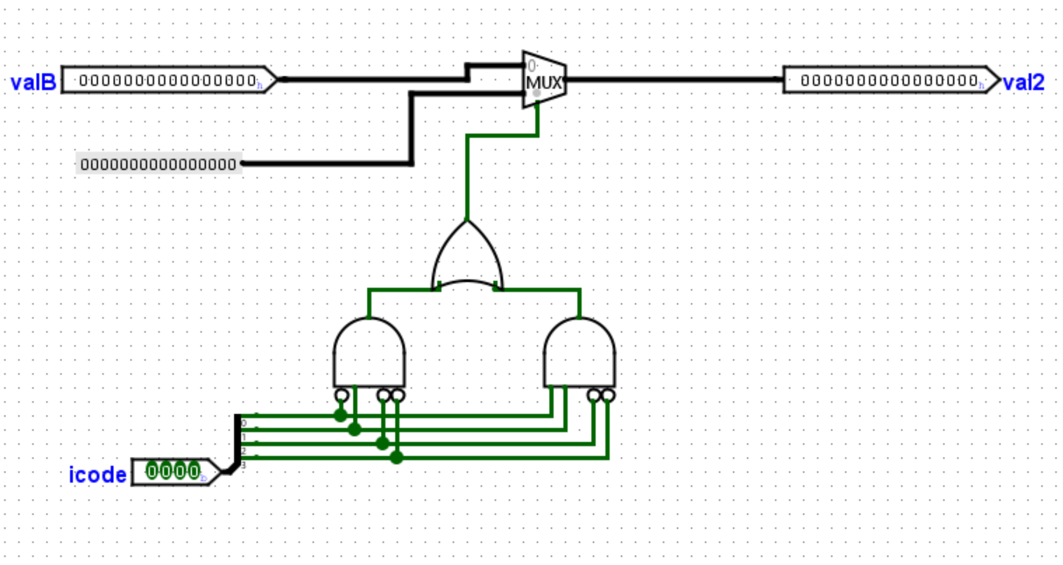
### 5.1 ALU A

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.



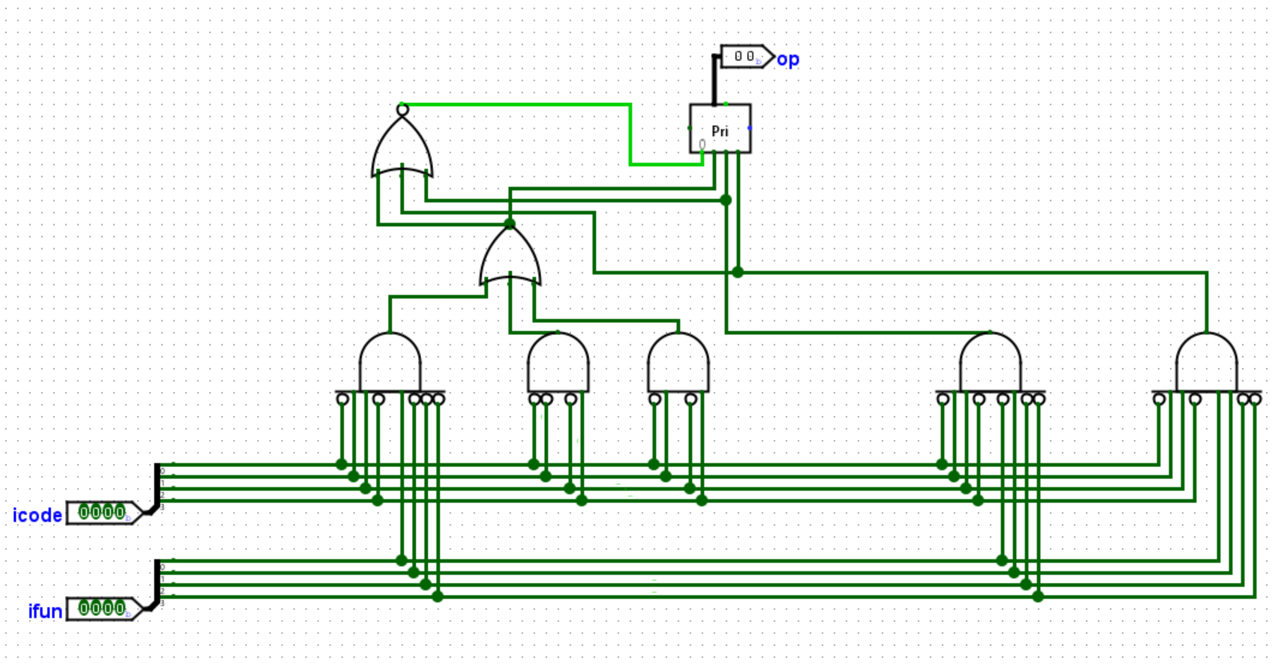
### 5.2 ALU B

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.



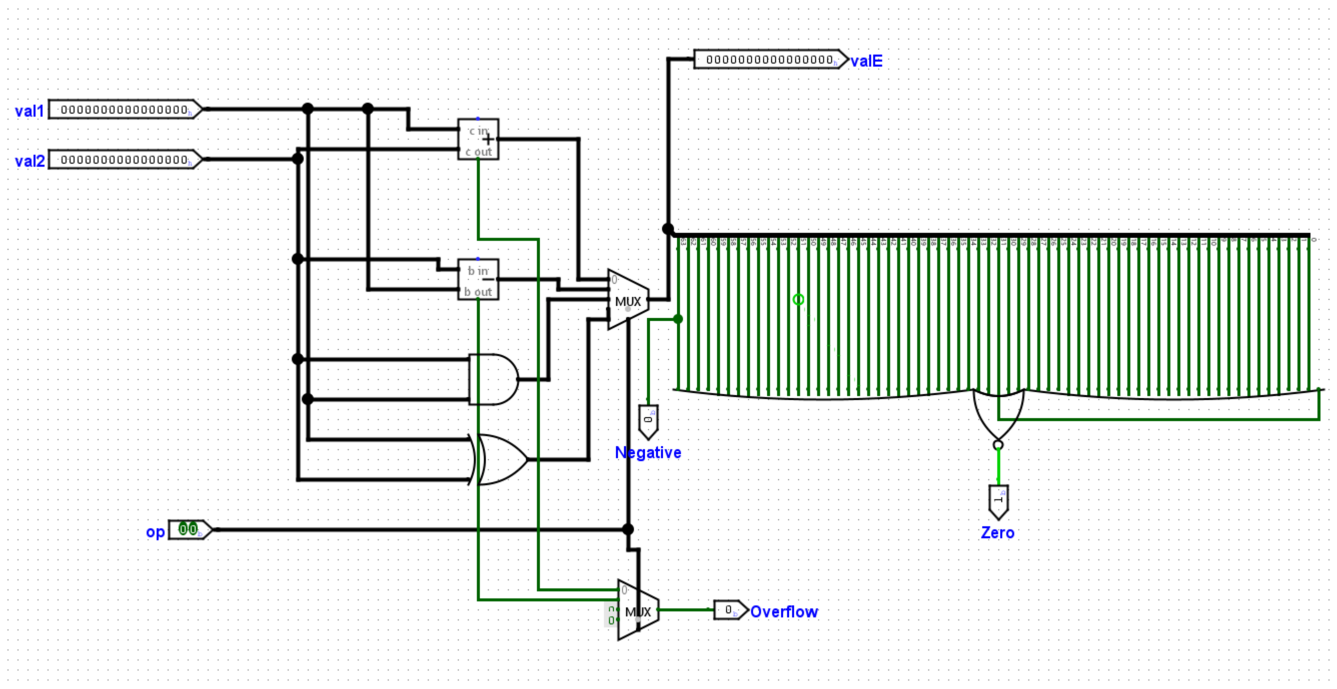
### 5.3 ALU Function

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.



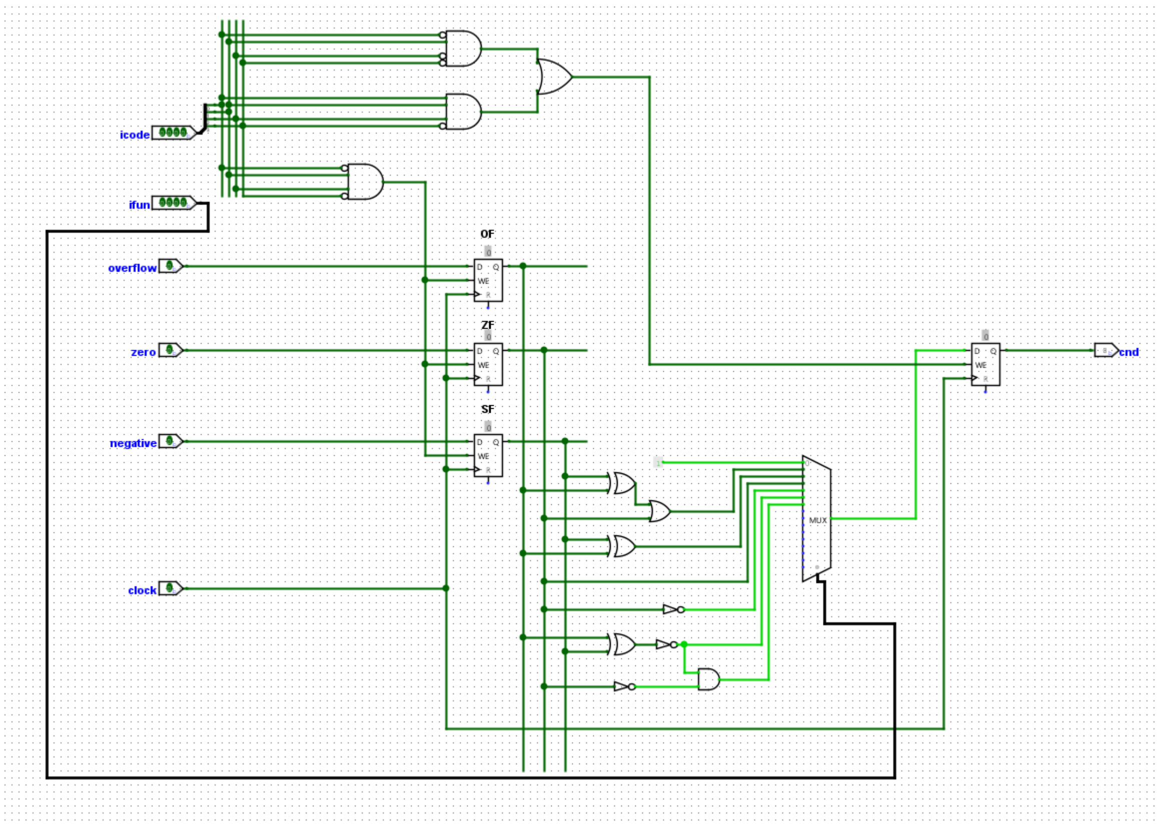
### 5.4 Arithmetic Logic Unit

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.


val1
0000000000000000
val2
0000000000000000
valE
0000000000000000
op
00
MUX
MUX
Negative
0
Zero
1
Overflow
0

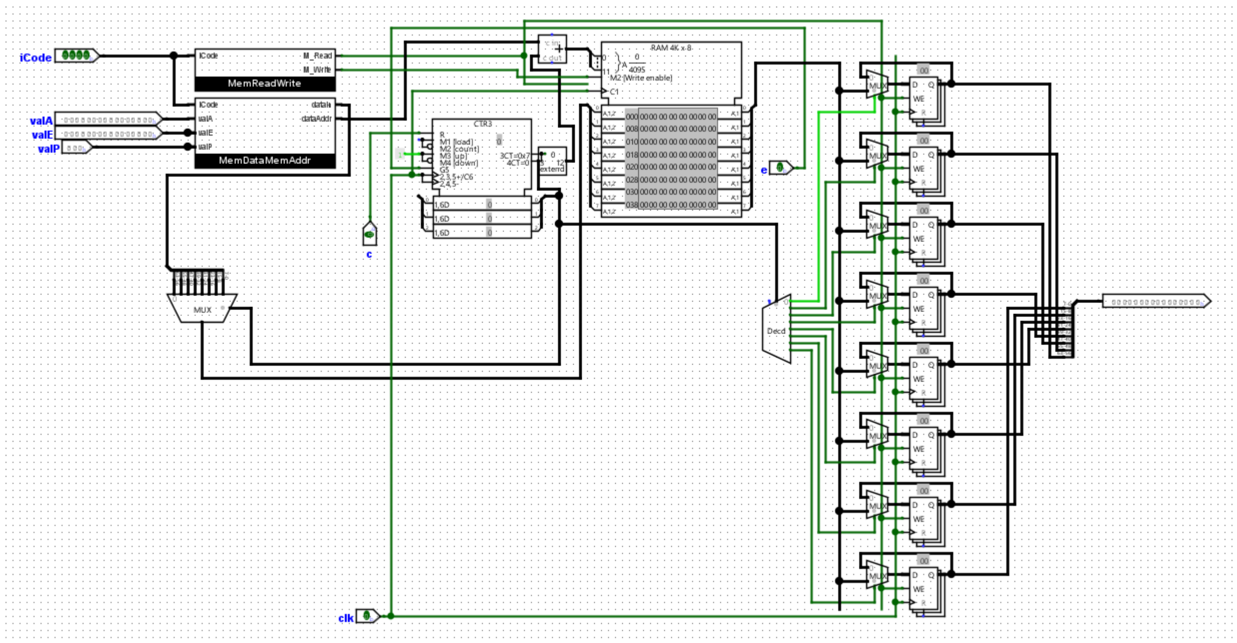
## 5.5 CC

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.



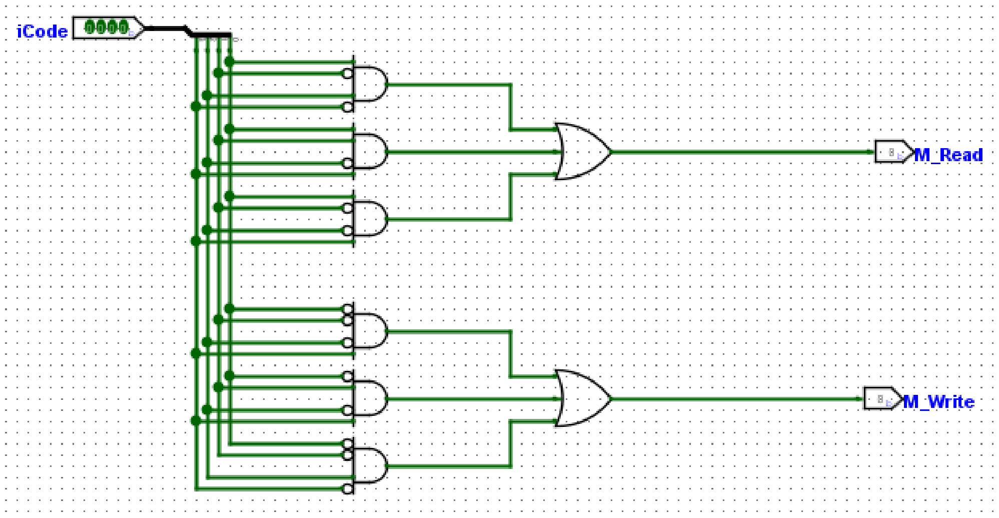
## 6 Memory Implementation

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



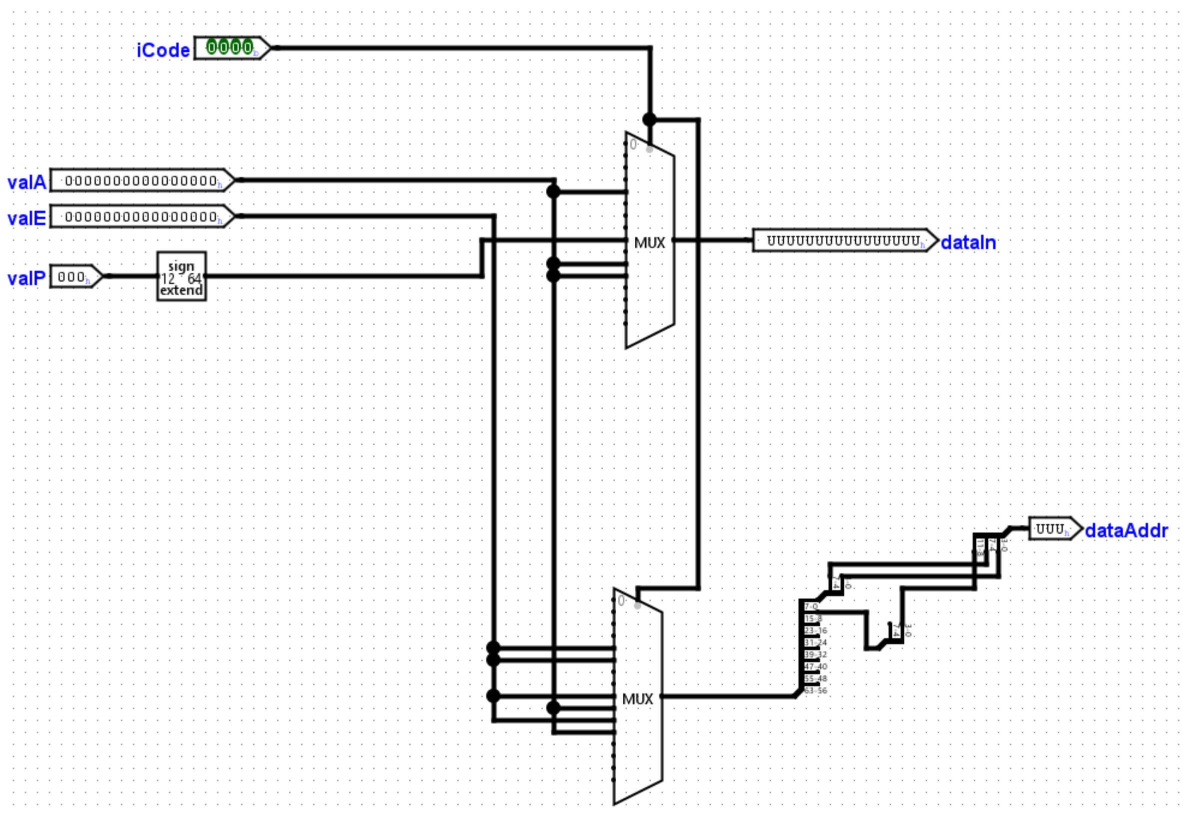
### 6.1 Memory Read/Write

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.



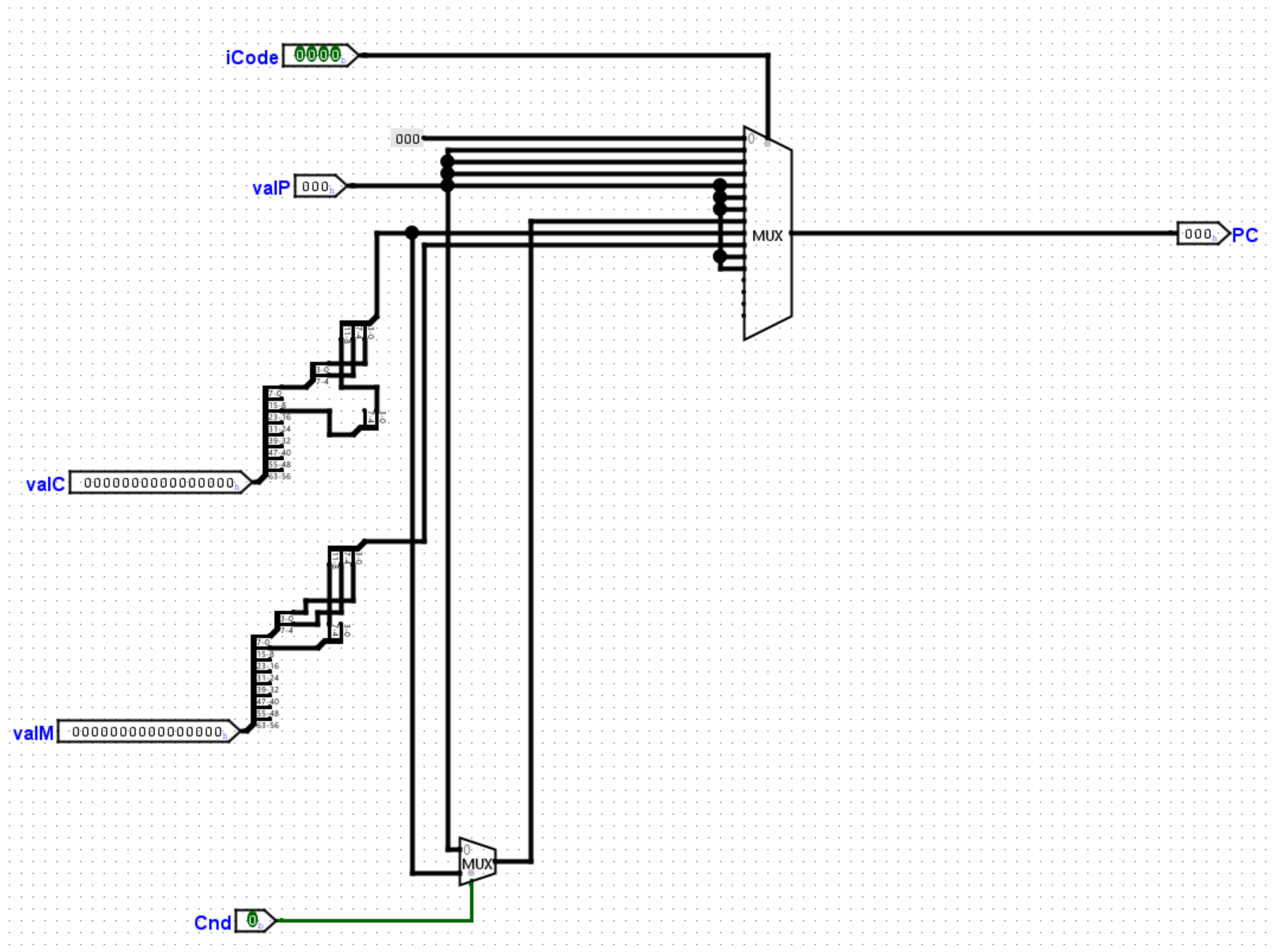
### 6.2 Memory Data and Memory Address

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.



## 7 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 figure out which output is needed for PC. When iCode is 0, this means a HALT function was called and 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.


iCode
0000
000
valP
000
MUX
000
PC
valC
0000000000000000
valM
0000000000000000
MUX
Cnd
0