Group 18: CSCE 312 CPU Project — Final Report

Lucian Chauvin
133003371
& *JoshuaLass*
531009387
& *BjornQuarfordt*
230003985

April 30, 2024

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`

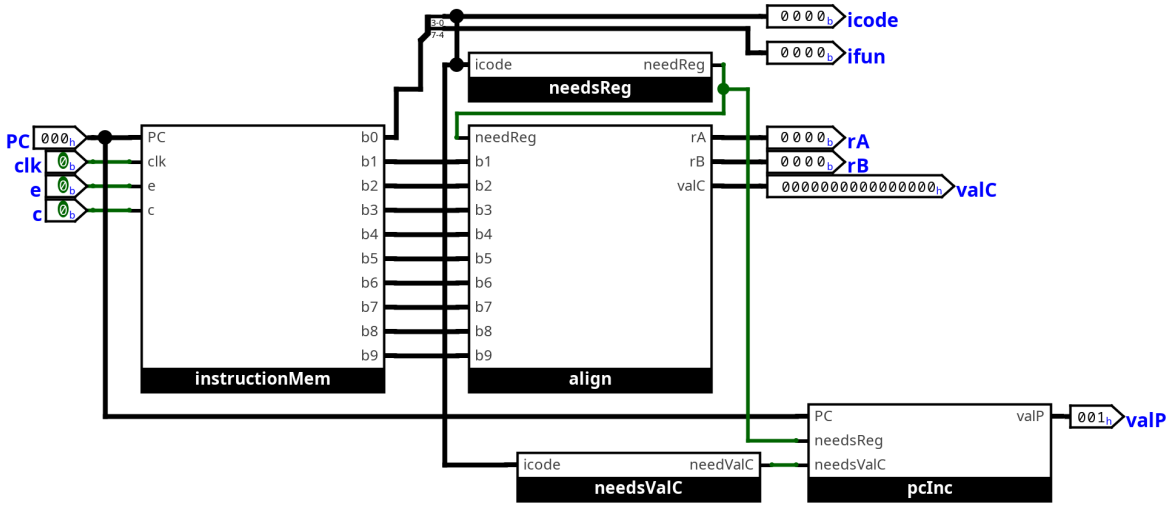
`pushq rA`	`icode:ifun← M_1[PC]`
	`rA:rB← M_1[PC+1]`
	`valP← M_8[PC+10]`
		`valA← R[rA]`
		`valB← R[\%rsp]`
			`valE← valB-8`	`M_8[valE]← valA`	`R[\%rsp]← valE`	`PC← valP`

`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 Fetch Implementation

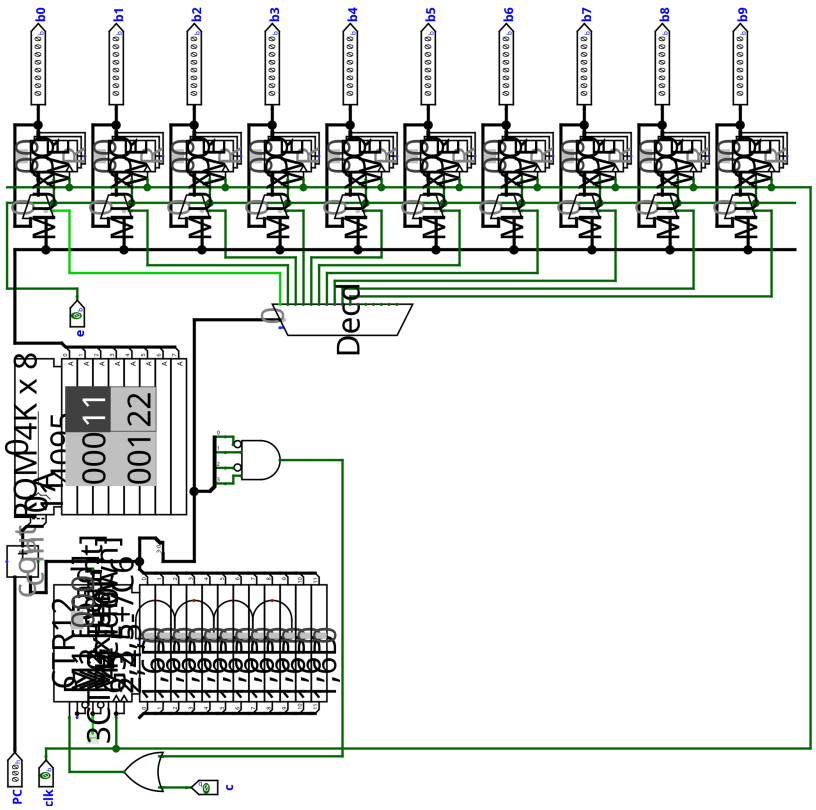
Our design is basically a one-to-one implementation of what is shown on the slides. Our instruction memory module stores the program in ROM and reads 10 bytes in from the current `PC`. We then pass these 10 bytes to our align module which — based on if we need registers or not — sets out `rA`, `rB`,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

2.1 Fetch



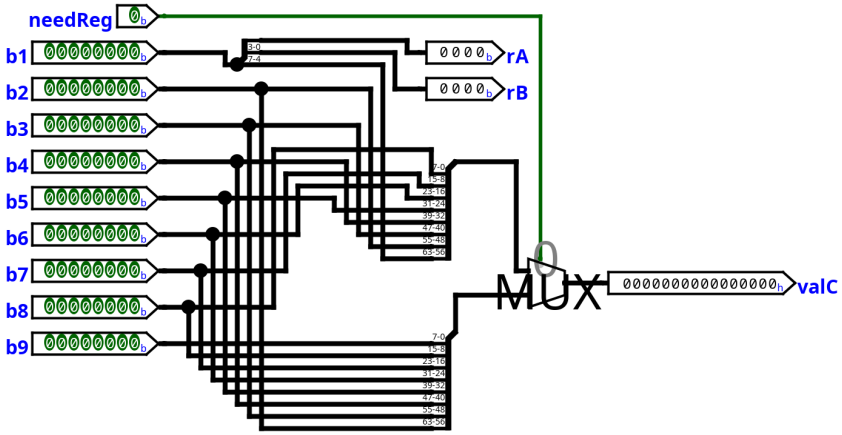
2.2 Instruction Memory

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).



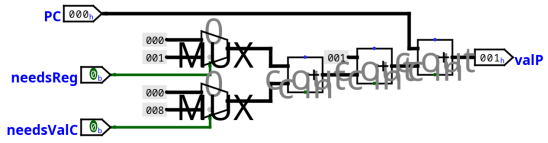
2.3 Align

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.



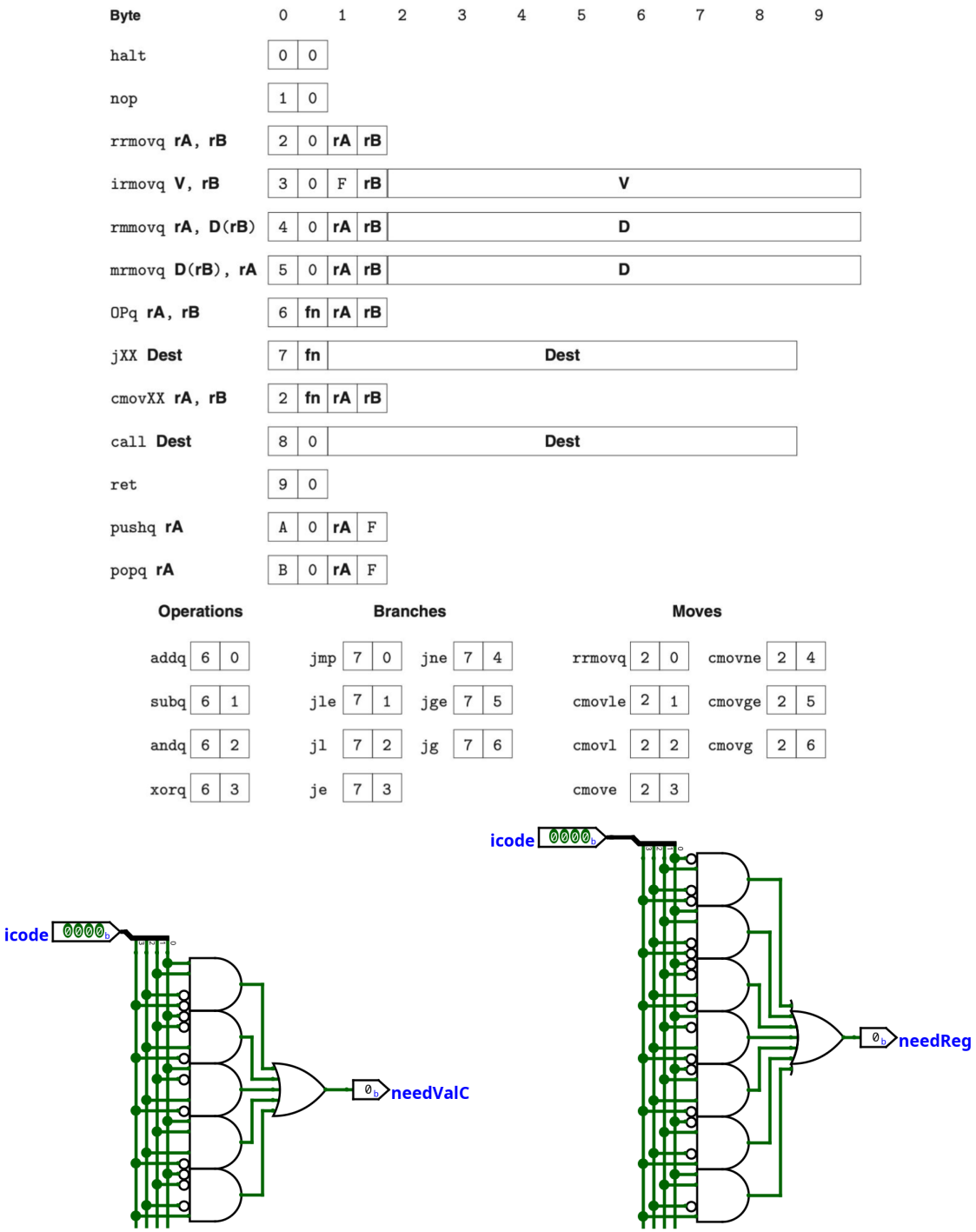
2.4 PC Increment

PC increment simply increments the `PC` based on if our instruction read 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`.



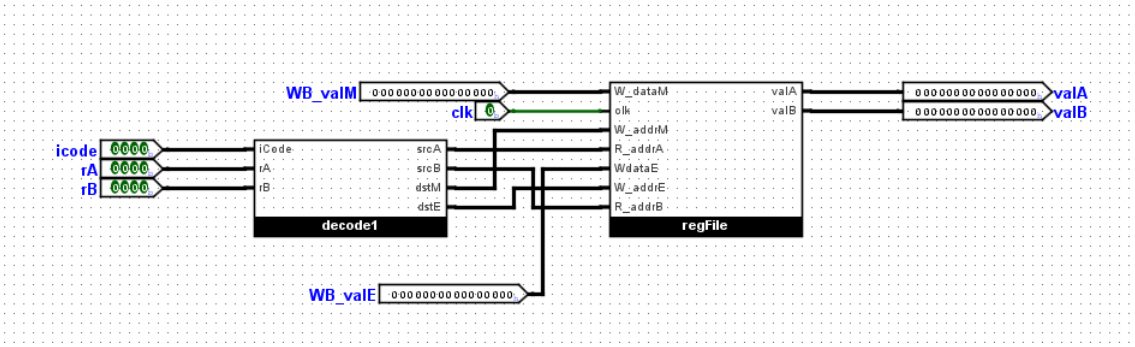
2.5 Needs ValC and Registers

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



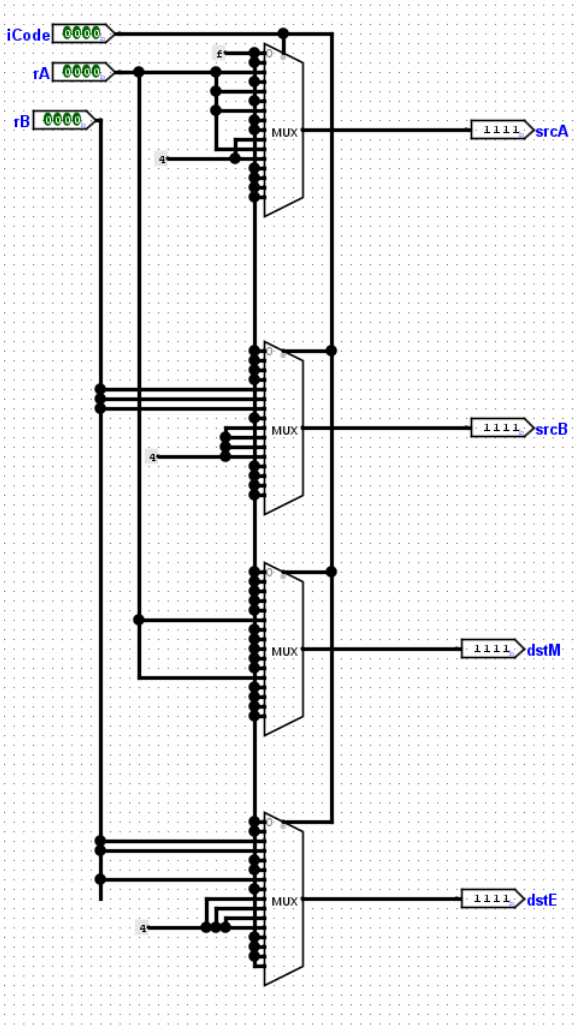
3 Decode and Write Back Implementation

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.



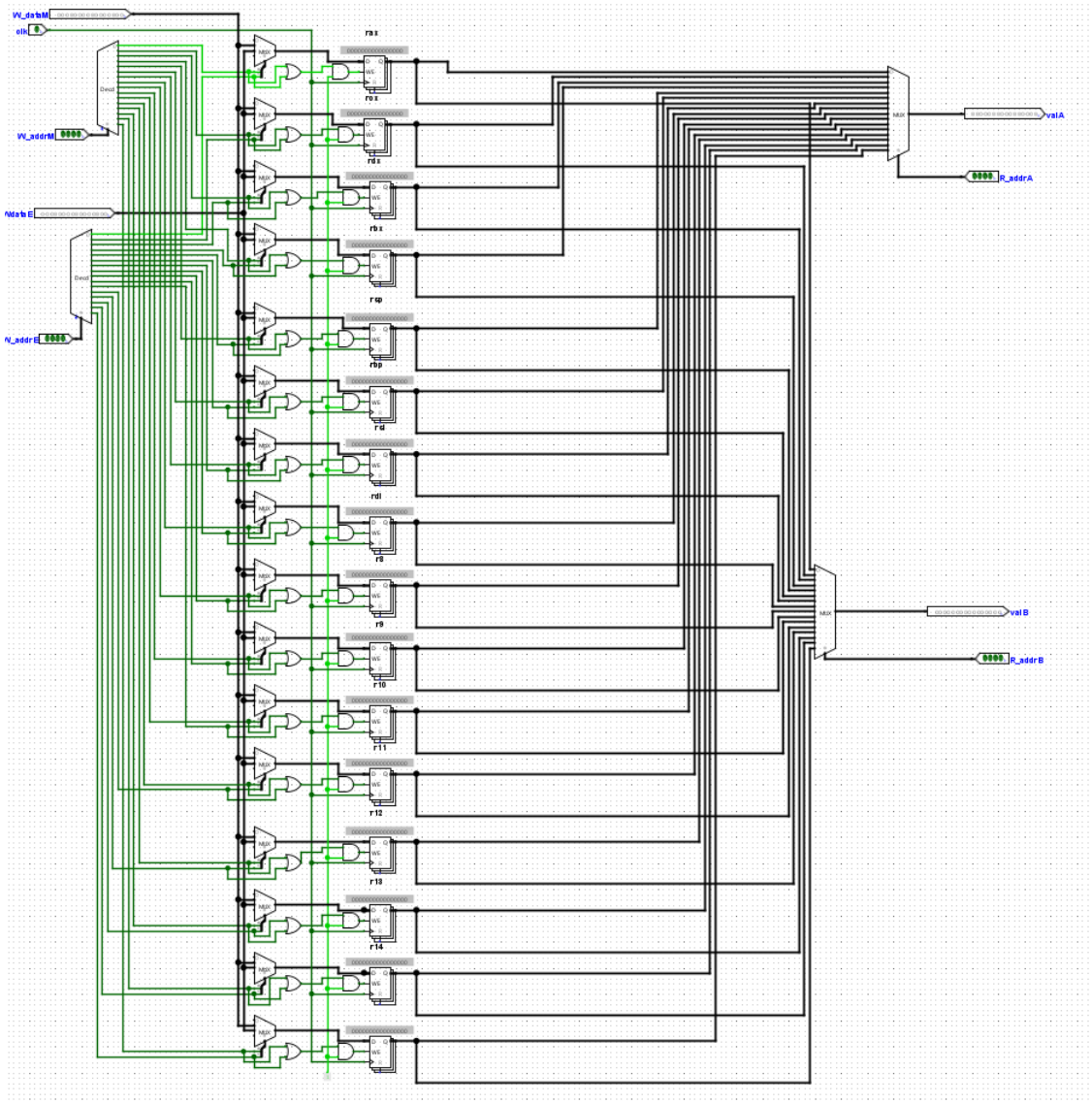
3.1 Decode1

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.



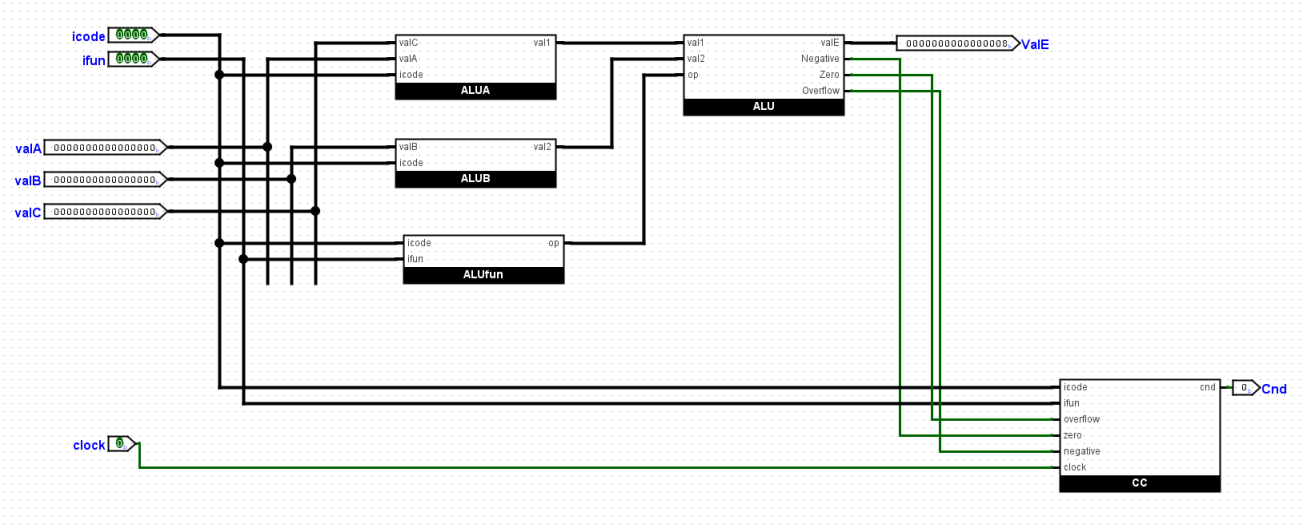
3.2 Register File

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.



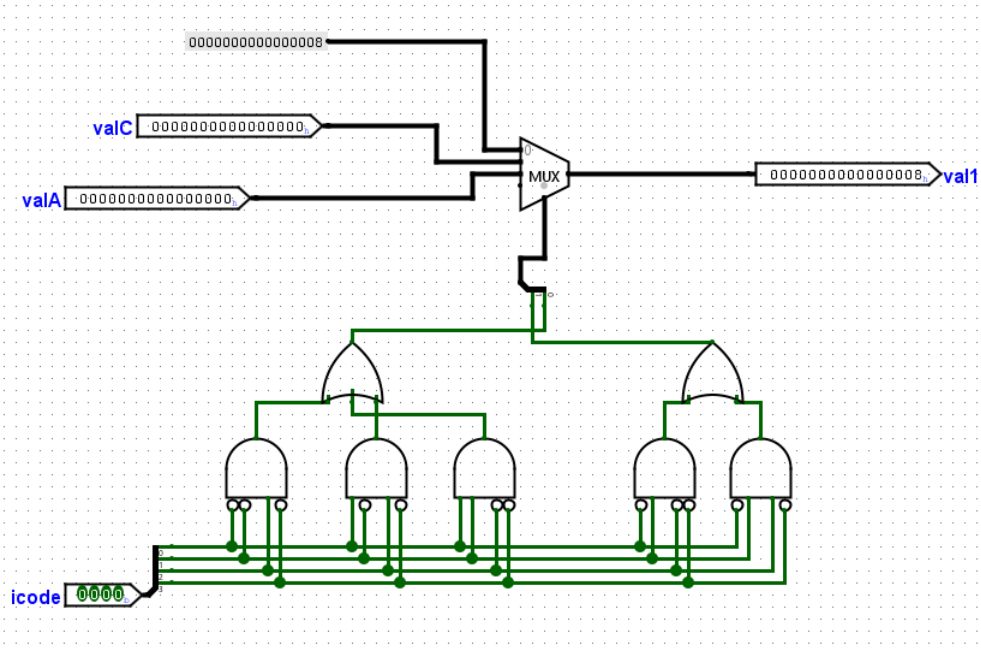
4 Execute Implementation

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.



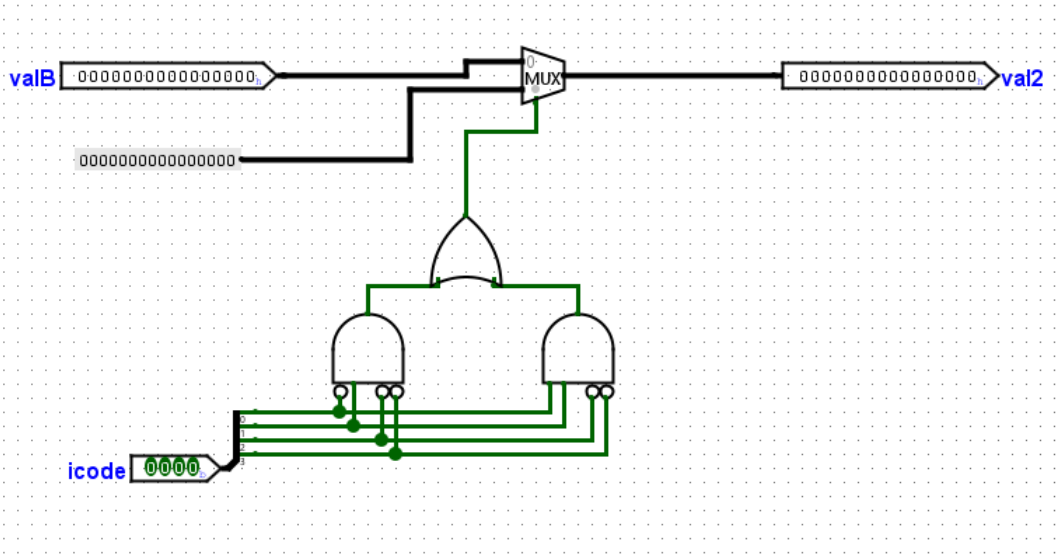
4.1 ALU A

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.



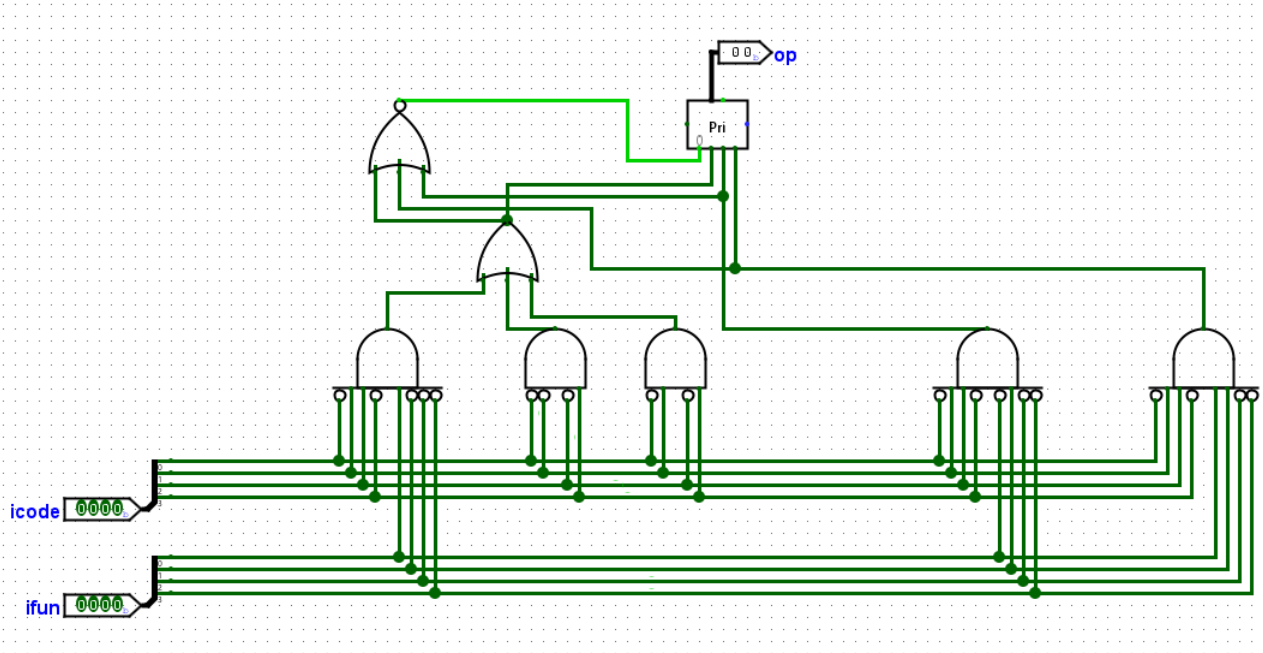
4.2 ALU B

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.



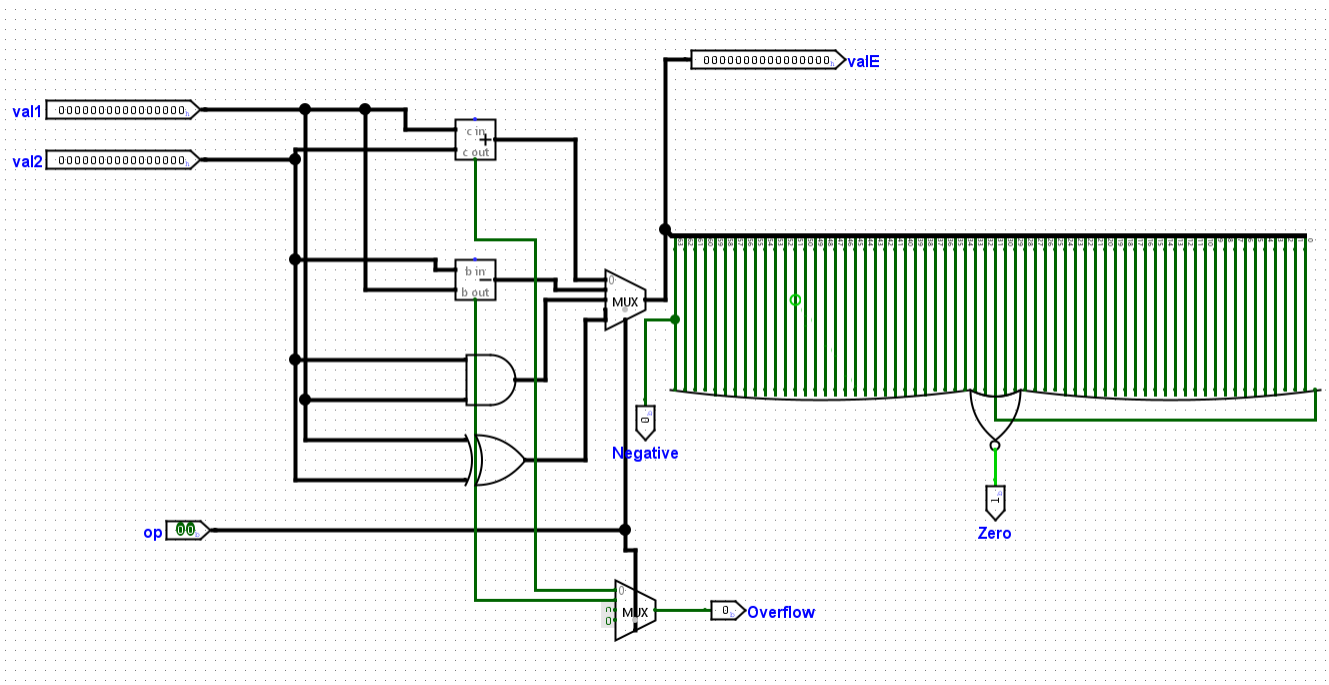
4.3 ALU Function

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.



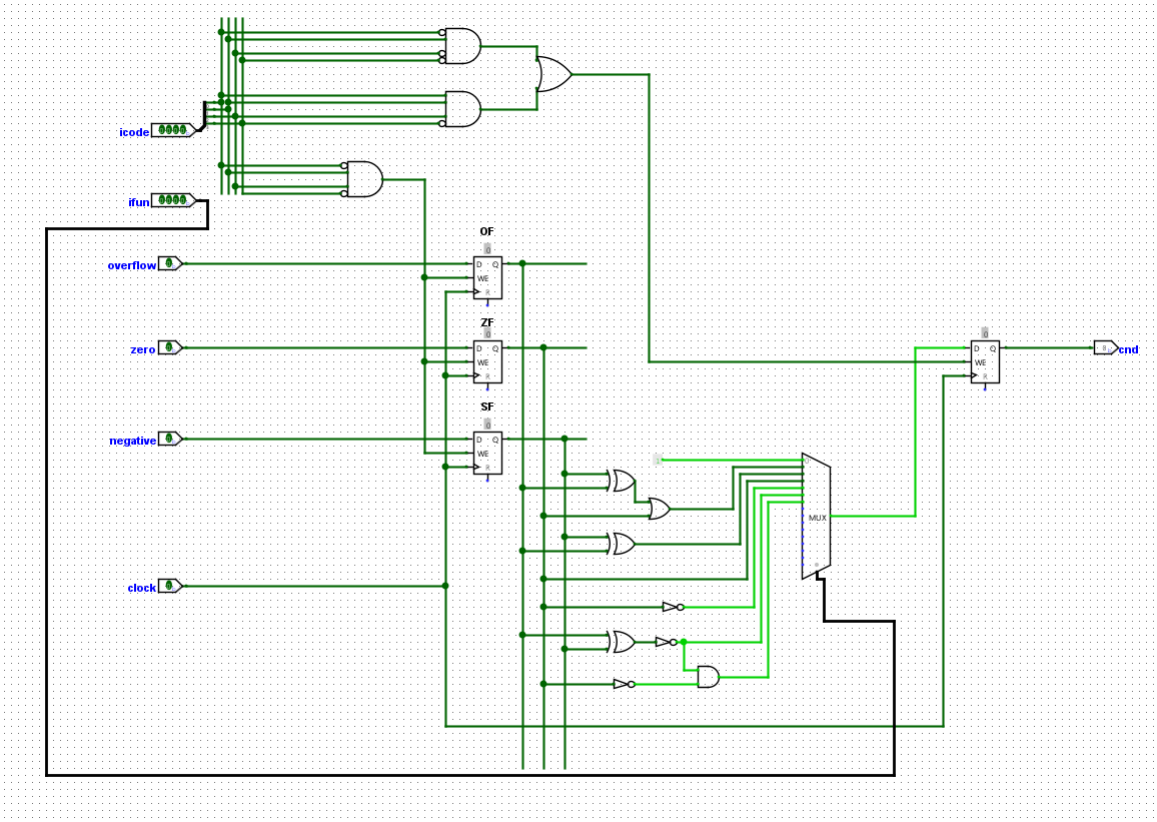
4.4 Arithmetic Logic Unit

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.



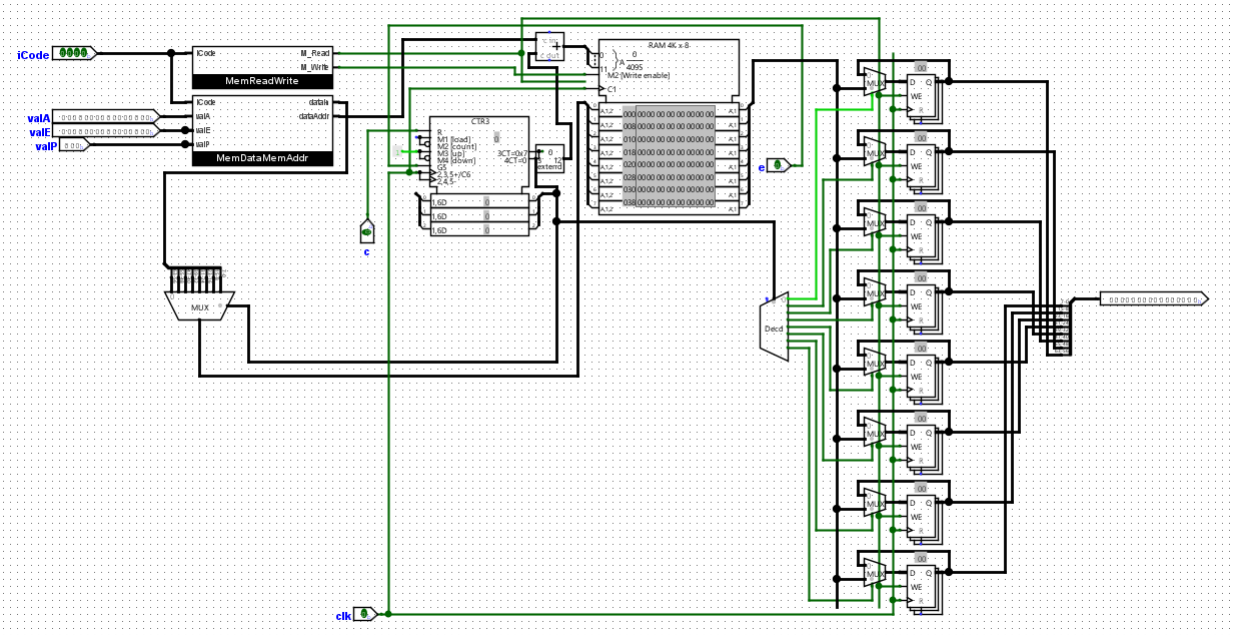
4.5 CC

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.



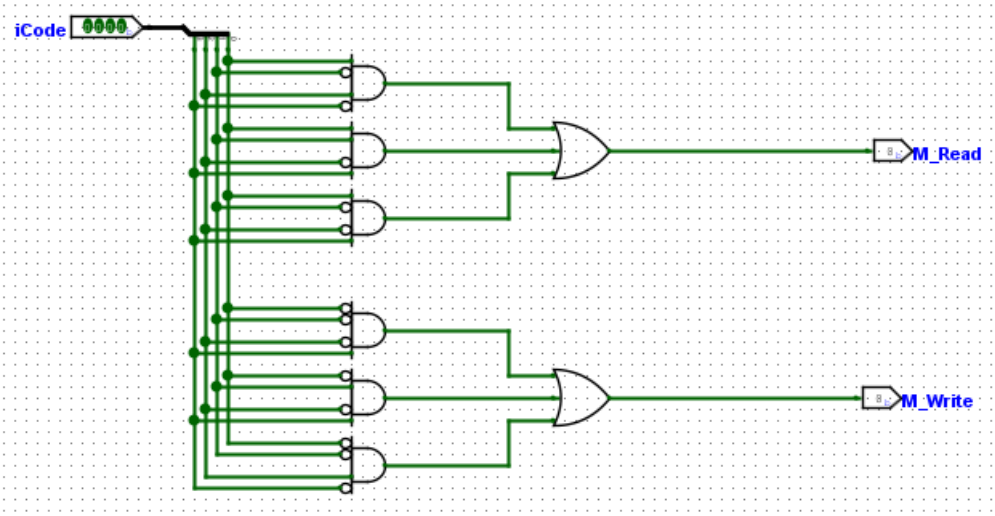
5 Memory Implementation

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



5.1 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.



5.2 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.