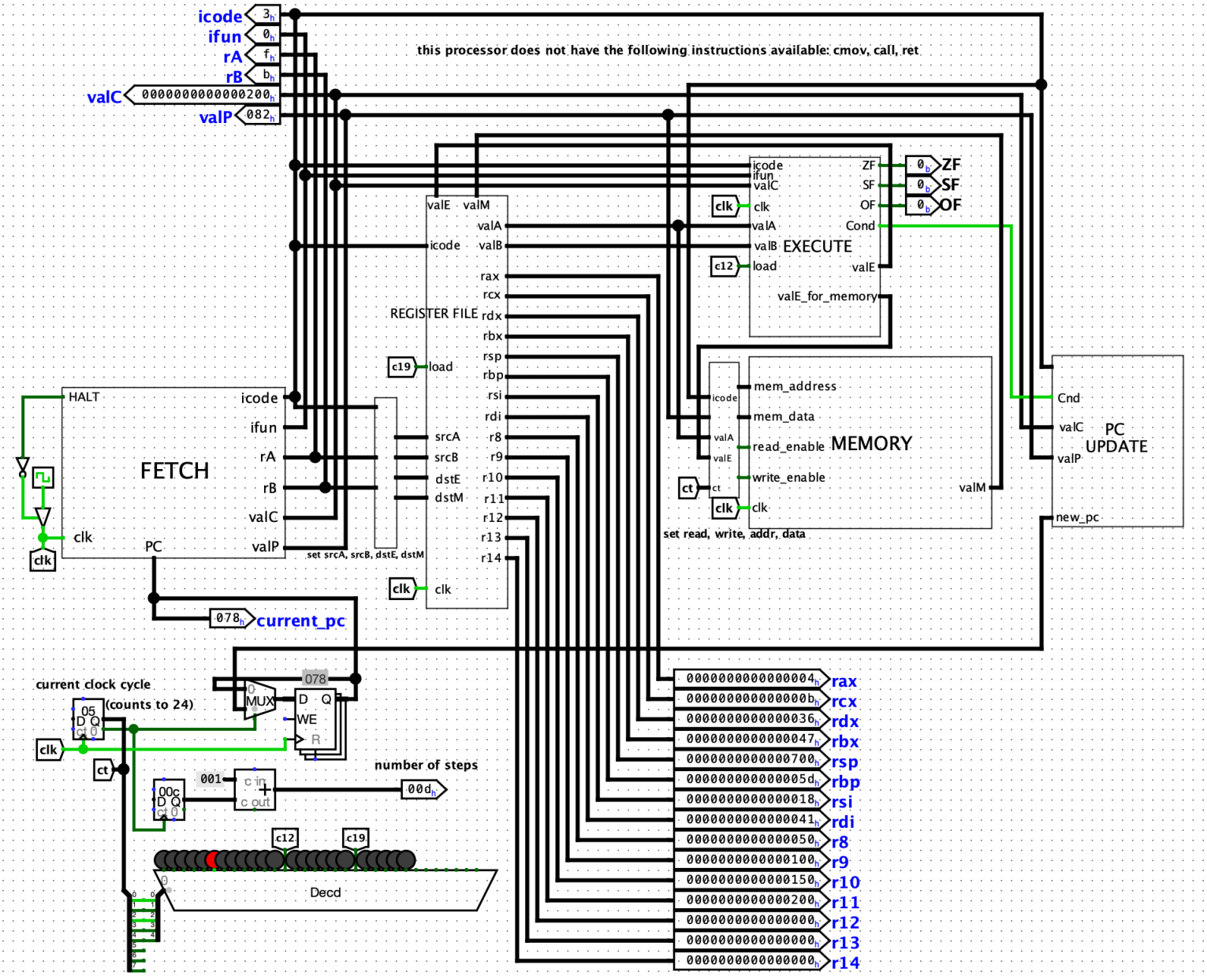
Final Project Report

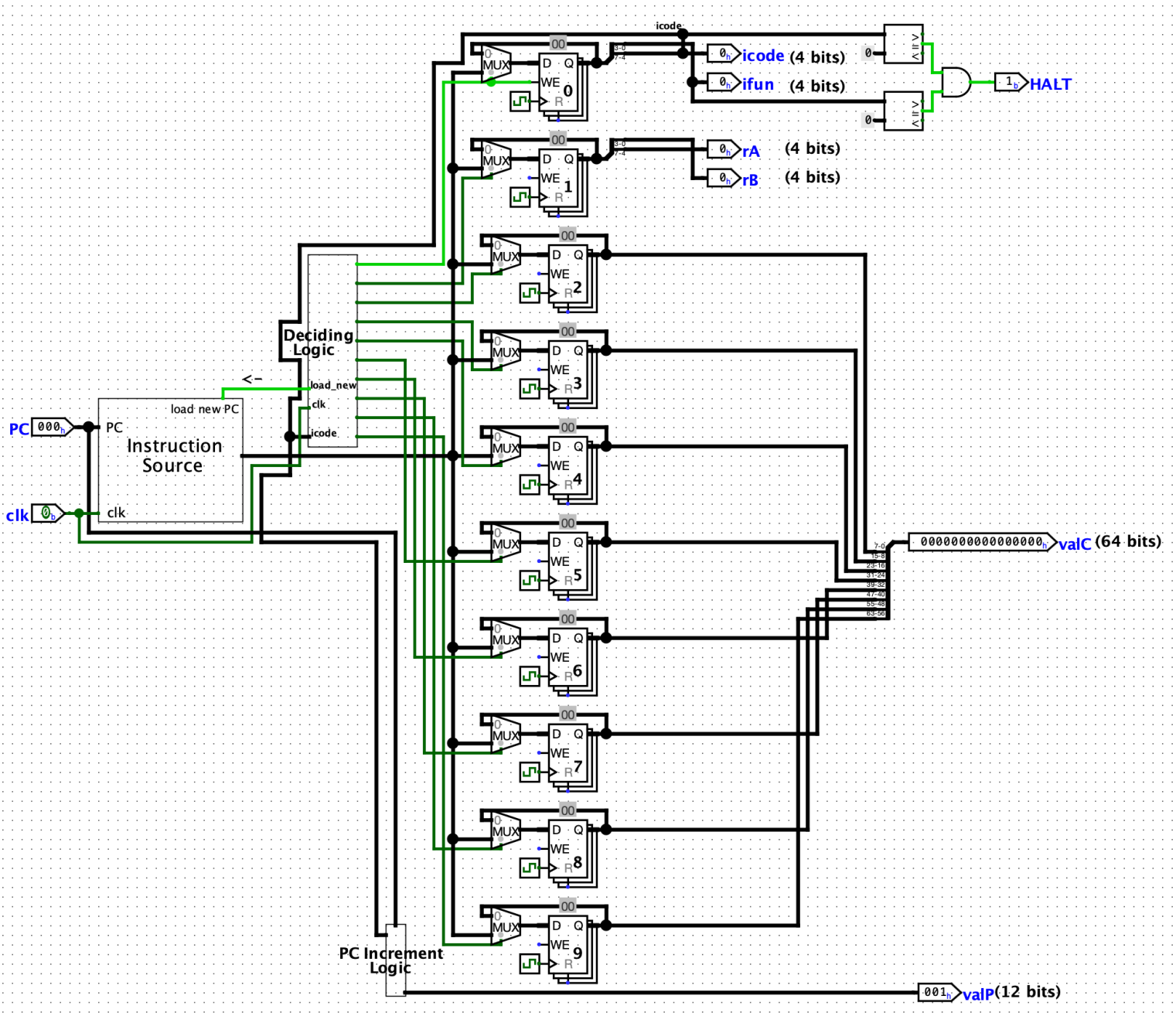
Nash Walters, Carson Coen, Vikram Kanthadai

External view of entire processor



Processor Stages

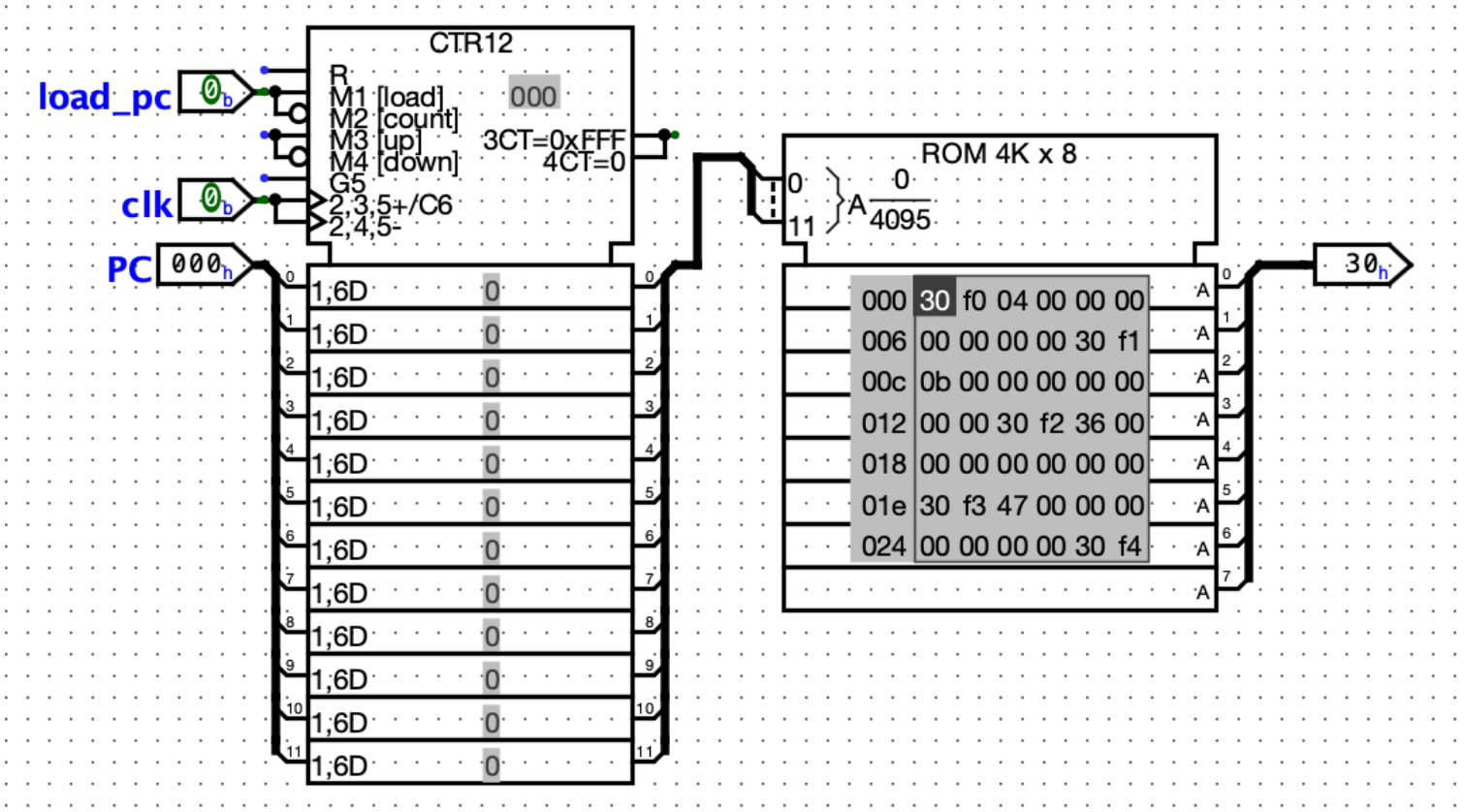
Fetch



(Fetch Stage)

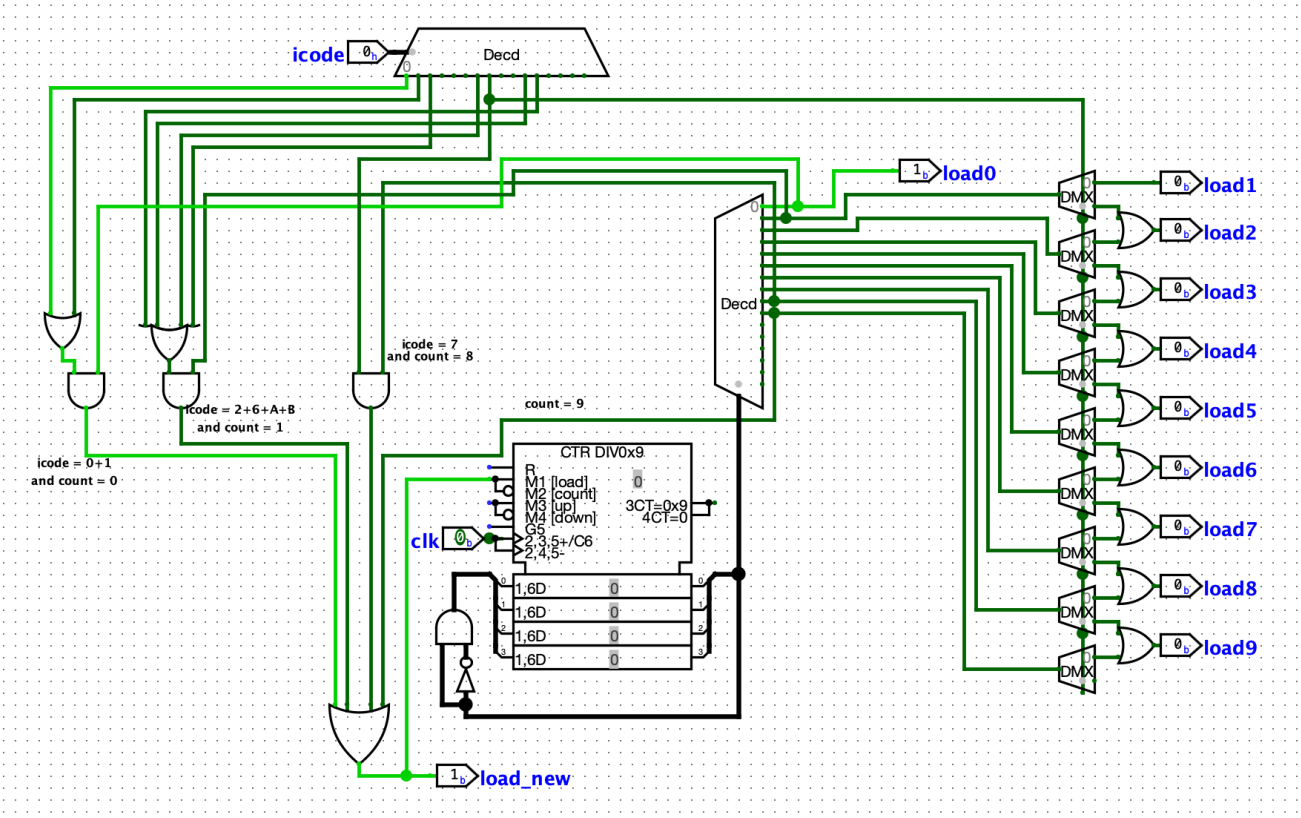
Our Fetch stage begins by obtaining data starting at the first PC address (0x000) of the ROM, which is already loaded with the contents of a Y86 assembly program. This program data was uploaded as a memory file, produced through a Python script that converts object files into a format that Logisim can understand. This program data is found within a subcircuit called Instruction Source that contains the processor's source of instructions.

Each instruction's byte data is outputted from this subcircuit and stored in ten different registers. Each register holds one byte of data. The first register is responsible for obtaining the icode and ifun values and the second register is responsible for loading the register identifiers rA and rB. The registers are loaded one at a time at each rising clock signal, so it takes ten cycles to fetch a single instruction's data.



(Instruction Source)

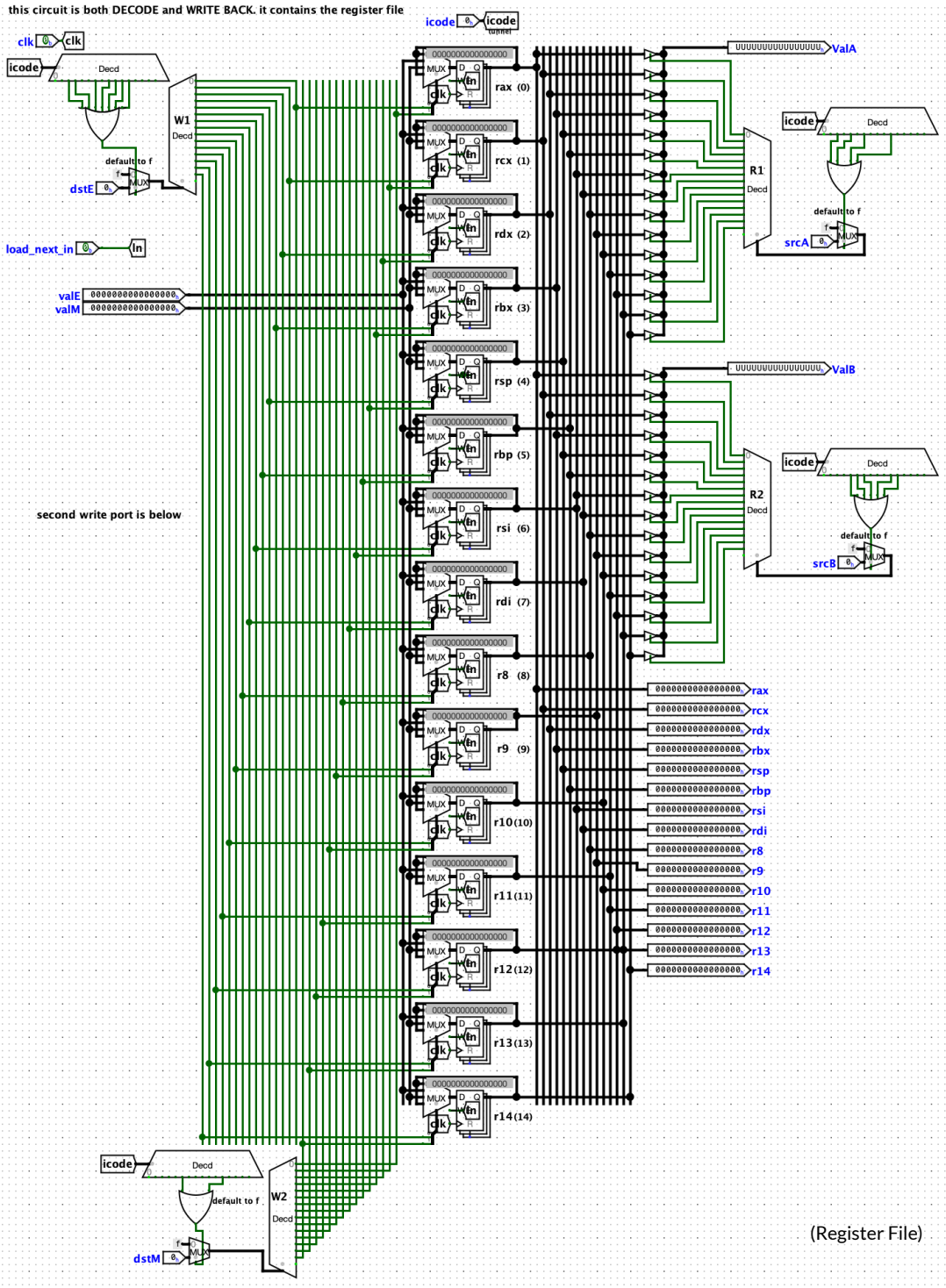
The remaining registers hold the value of valC (which is 64 bits long). The enable inputs for each of the registers come from a subcircuit called Deciding Logic. This subcircuit is featured below and decodes the value of icode to determine which byte is going to be loaded (and when). This is for synchronization purposes, allowing the byte data of each instruction to be obtained in an ordered and timely manner. A counter (which counts from 0 to 9) is used in tandem with the icode value to accomplish this synchronization.



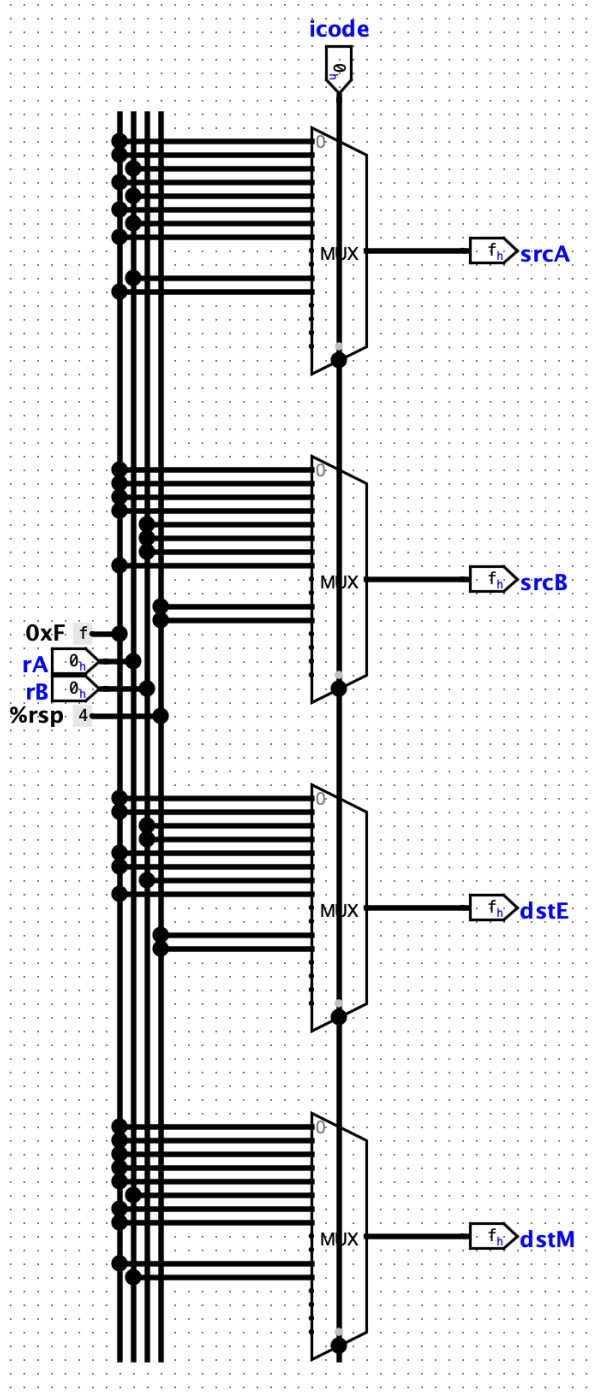
(Deciding Logic)

Finally, the PC Increment Logic subcircuit calculates the next PC value and stores the result in valP. The result is based entirely on the value of icode. Within this subcircuit, the icode signal serves as the selector input of a decoder. This is used to determine what the current instruction is. The formula valP = PC + 1 + r + 8 * i is then implemented using adders, OR gates, and muxes in order to calculate the value of valP. The next PC value will later be determined to be either valP or valC in the PC Update stage of the processor. The final point to note about our Fetch system is the halt functionality. If the current icode and ifun values are 0, the system's halt flag will signal, and no more changes to the processor state will occur. (This is accomplished through the use of a controlled buffer on the outgoing signal of the clock in the main processor circuit.) Overall, our Fetch stage will take ten clock cycles to guarantee that each necessary value has been obtained, including icode, ifun, rA, rB, valC, and valP.

Decode/Write Back (Register File)



The Register File subcircuit contains both the Decode and Write Back stages. Our Decode and Write Back stages both depend on the subcircuit called Set Sources and Destinations, which is responsible for determining the values of srcA, srcB, dstE, and dstM. These values are based solely on the icode signal and serve as inputs to the Register File subcircuit. The four possible values that srcA, srcB, dstE, and dstM can take on are rA, rB, 0x4 (which represents register %rsp), and, to signify no use, 0xF. This subcircuit utilizes 4 muxes to set the values properly, and the icode signal is used as the selector input of each of the muxes. The dstE value is used if writing is taking place from the ALU to one of the 15 registers, and the dstM value is used if writing is taking place from the memory to one of the 15 registers. This subcircuit can be seen featured on the right.



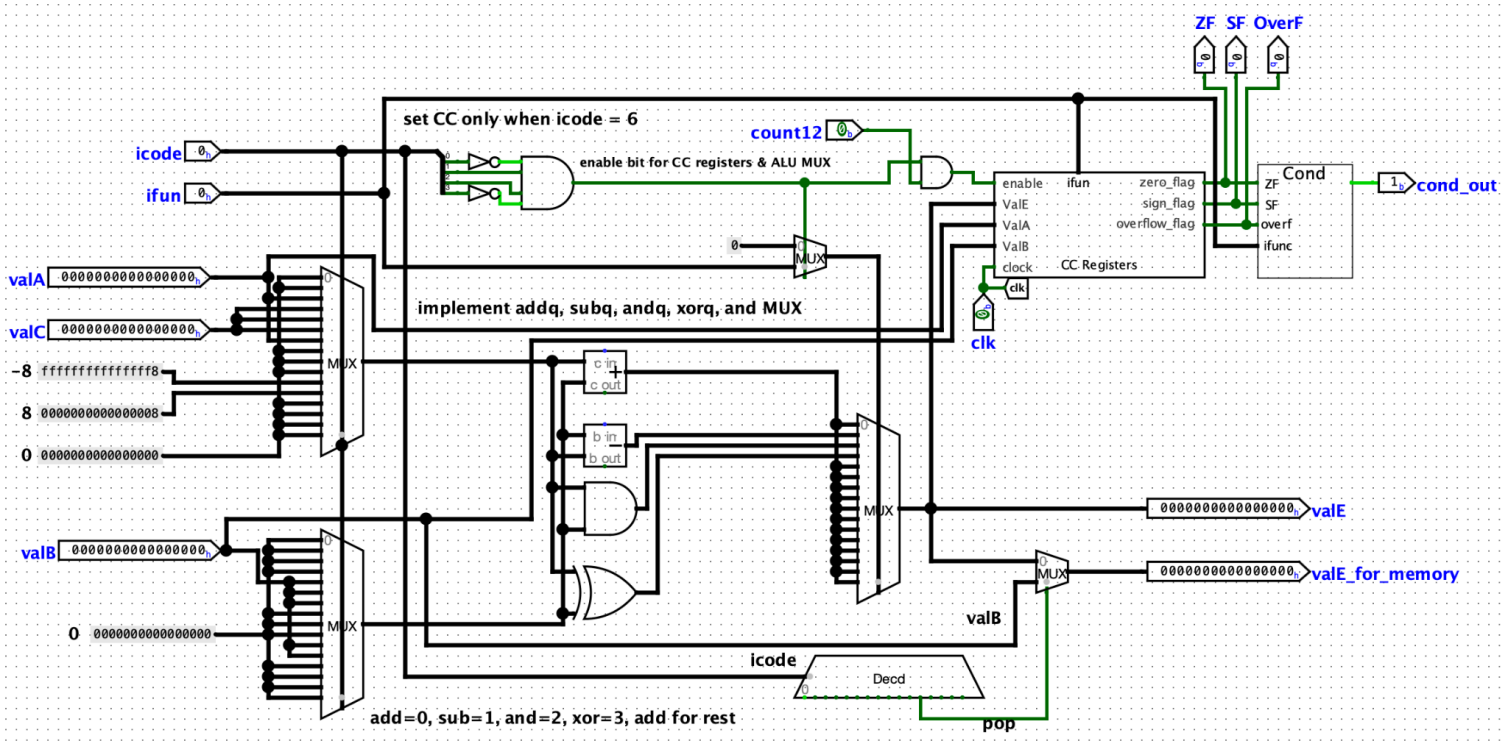
The Decode stage is when reading from the registers takes place, and the Write Back stage is when writing to the registers takes place. Both stages are found in the Register File subcircuit. Once srcA, srcB, dstE, and dstM are set properly, they are passed directly into the Register File subcircuit, which contains two write ports and two read ports. The two addresses for the write ports include dstE and dstM, and the values that can be sent into the registers designated by these addresses are valE and valM. The register file contains 15 registers that contain 64 bits each. Muxes are utilized to determine whether to load valE, load valM, or maintain the register's current value. Which option to choose out of these three is decided by the outputs of both write decoders, which are combined into 2-bit signals using a splitter. These 2-bit signals are used as the selection inputs into each of the muxes in the register file, selecting whether to load valE, load valM, or maintain the current value of the register.

The outputs of the read ports include valA and valB, which are obtained from reading the value at the addresses indicated by srcA and srcB. The values of valA and valB are guaranteed to be obtained by the end of the tenth clock cycle of each instruction, so the Decode stage is done by the end of the tenth cycle. In addition to the outputs valA and valB, the Register File subcircuit also includes outputs that show the current values stored in each of the 15 registers.

Importantly, our Register File stage includes an input called load_next_in whose job it is to tell the processor when to write back either from the ALU (valE) or the memory (valM) into the registers of the Register File. In our implementation, we utilize a tunnel to allow the load_next_in signal to be sent directly to the enable input of each of the 15 registers. The load_next_in signal is set to 1 only when the correct clock cycle count has been reached. This ensures that the register values stored in the Register File are always preserved when they should be preserved and that they are changed only when they should be changed. This ensures the accuracy of

the processor's performance, especially since the register file is such an important part of the programmer's visible state.

Execute

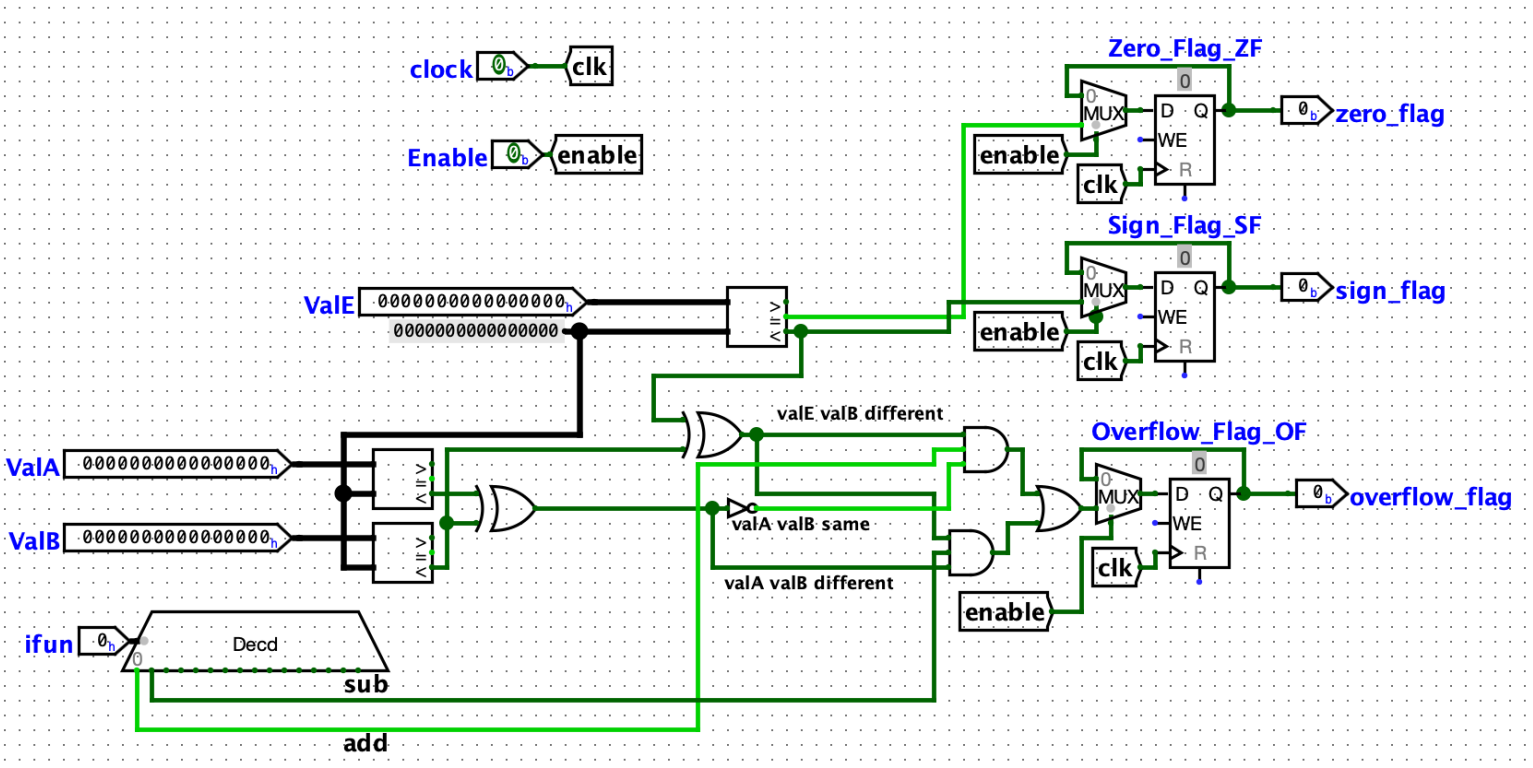


(Execute Subcircuit)

The Execute stage takes in the values (valA and valB) from the previous Decode stage as well as valC from the Fetch stage. The first ALU fan is formulated via a mux which takes in inputs valA, valC, 8, -8, and zero. The second ALU fan uses a similar approach, however this time it uses valB and zero. These multiplexers both have icode as their selector input.

Operations including addition, subtraction, logical AND, and logical XOR are then performed using the two previously mentioned mux outputs along with an adder, a subtractor, an AND gate, and finally an XOR gate. All four of these outputs are then fed into another mux. The selector bits for this mux are based entirely on the value of ifun. The ifun value determines which operation to allow through the mux and stores the result in valE, which is the main output of the Execute stage. A separate valE output called valE_for_memory holds the address that is sent to the memory subcircuit.

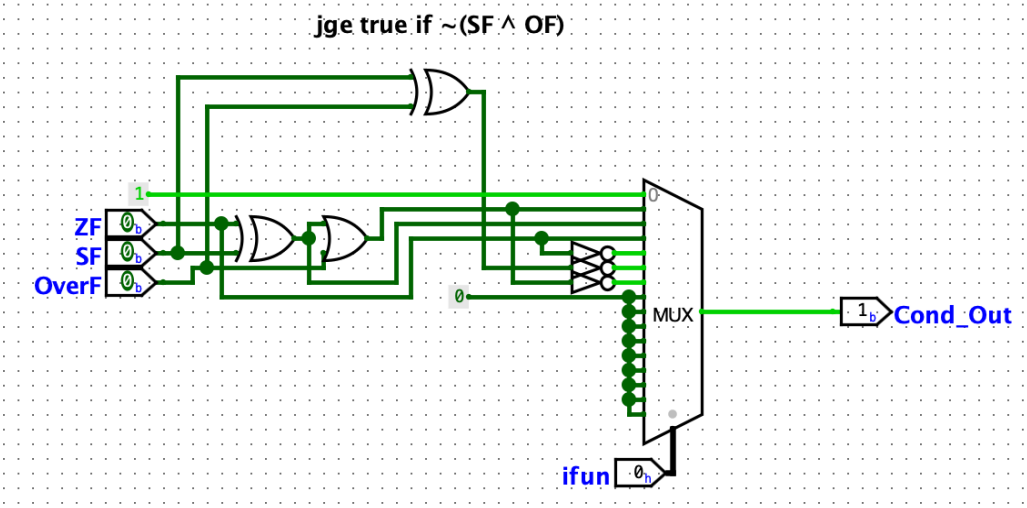
This was necessary in our implementation simply because, in the case of the pop instruction, valB (instead of valE) had to be used as the address sent to the memory subcircuit when popping a value from the stack. The final part of the Execute stage was the construction of the condition code module. So long as the current icode value is 6 (which indicates that an operation is taking place), the condition code module is enabled. This allows the condition code flags to only be set when an operation such as addition, subtraction, logical AND, or logical XOR is taking place.



(Condition Code Registers)

This condition code subcircuit takes all of the same inputs as the Execute subcircuit does (along with valE from the Execute stage) and plugs them into separate comparators to determine which flags to set as a result of the ALU operation. The first comparator takes in valE, along with a constant value of 0. If the two values are equal, then the zero_flag (ZF) value will be set. In addition, the first comparator checks if valE is less than 0, and if so, the sign_flag (SF) will be set.

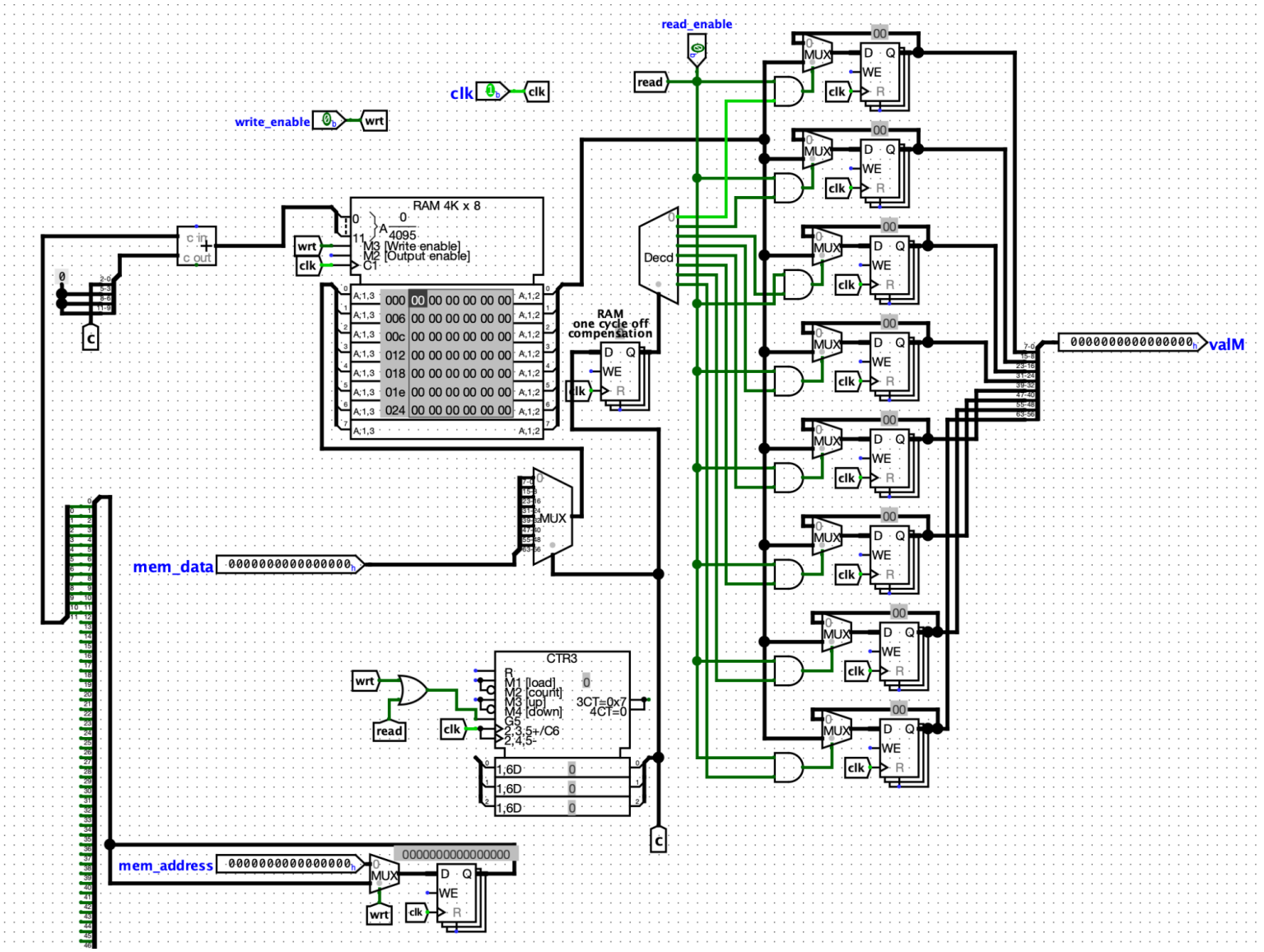
The overflow flag is determined using all three comparators. Our implementation sets the overflow flag in two scenarios. The first scenario that sets the overflow flag is when addition is taking place, valA and valB have the same sign, and valE and valB have different signs. The second scenario that sets the overflow flag is when subtraction is taking place, valA and valB have different signs, and valE and valB also have different signs. Either of these scenarios will result in overflow, and the overflow_flag (OF) output will be set to 1 to indicate that an overflow occurred as a result of the most recent ALU operation.



(Condition Code Subcircuit)

Once each of the three flags has been determined, these outputs are fed into another sub-circuit called Cond which, using two XOR gates, an OR gate, a mux, and ifun as a selector, determines whether the condition output will be 0 or 1. The logic here is based on the value of ifun in addition to the current flags because that value is how the processor knows what type of jump instruction is taking place. If the condition is true, then the condition output is set to 1 and the PC Update stage will allow the jump to occur. If the condition is false, then the condition output is set to 0 and the PC Update stage will prevent the jump from occurring.

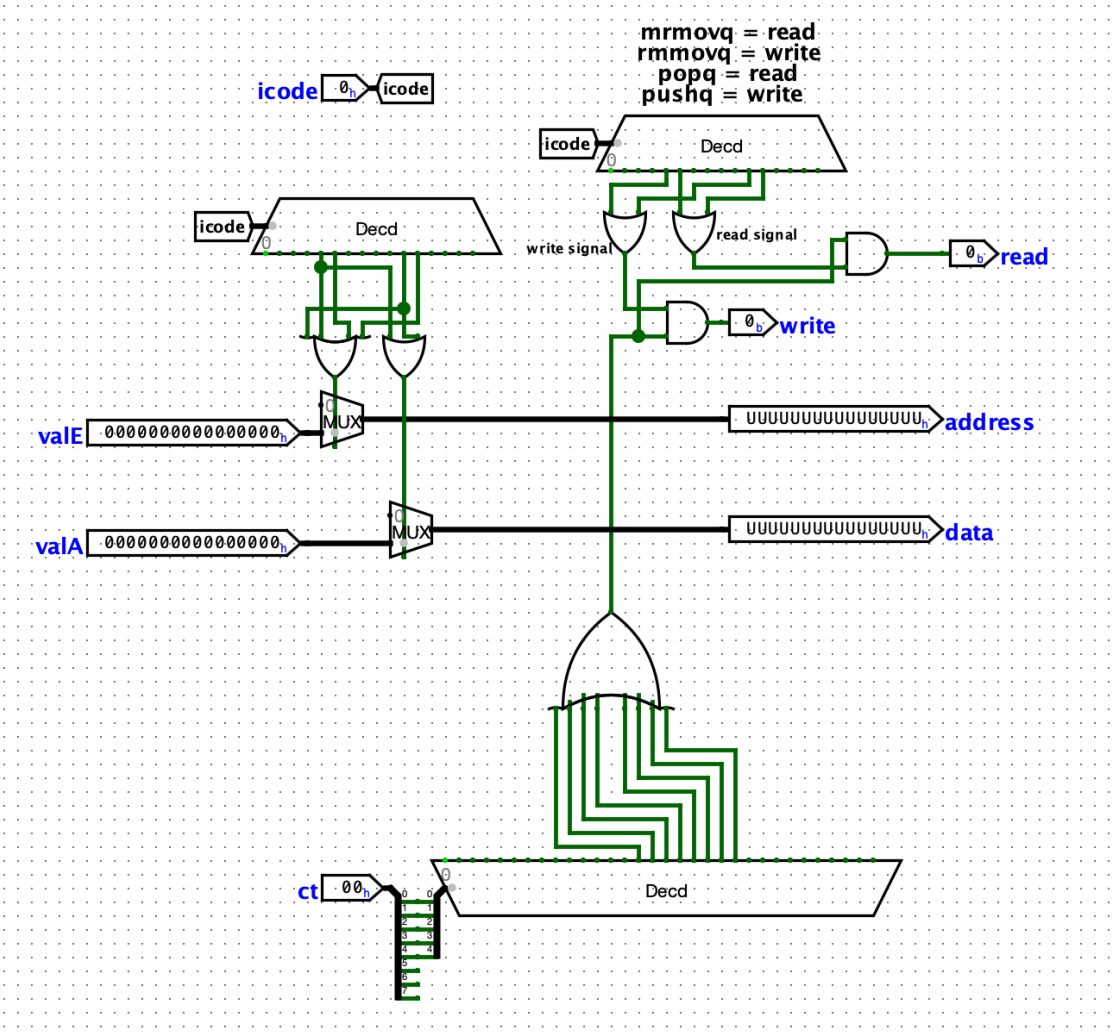
Memory



(Memory Stage)

The Memory stage begins in a similar way to our Decode stage, using a subcircuit to preprocess data and determine the values of the stage's inputs. The subcircuit for this stage is called Set Sources and Destinations, and, depending on the current icode and clock cycle count, it determines whether to read, write, or do nothing. The value of icode is used to determine whether to allow valE to pass to the memory subcircuit as the memory address and also to determine whether to allow valA to pass to the memory subcircuit as the memory data. This subcircuit, in addition to using a decoder for icode, uses a second decoder to determine if writing or reading is allowed during a given clock cycle. The current clock cycle count must be between 14 and 21 in order for either

reading or writing to take place. This means that reading or writing from/to the memory can only occur during a small 8 cycle window.



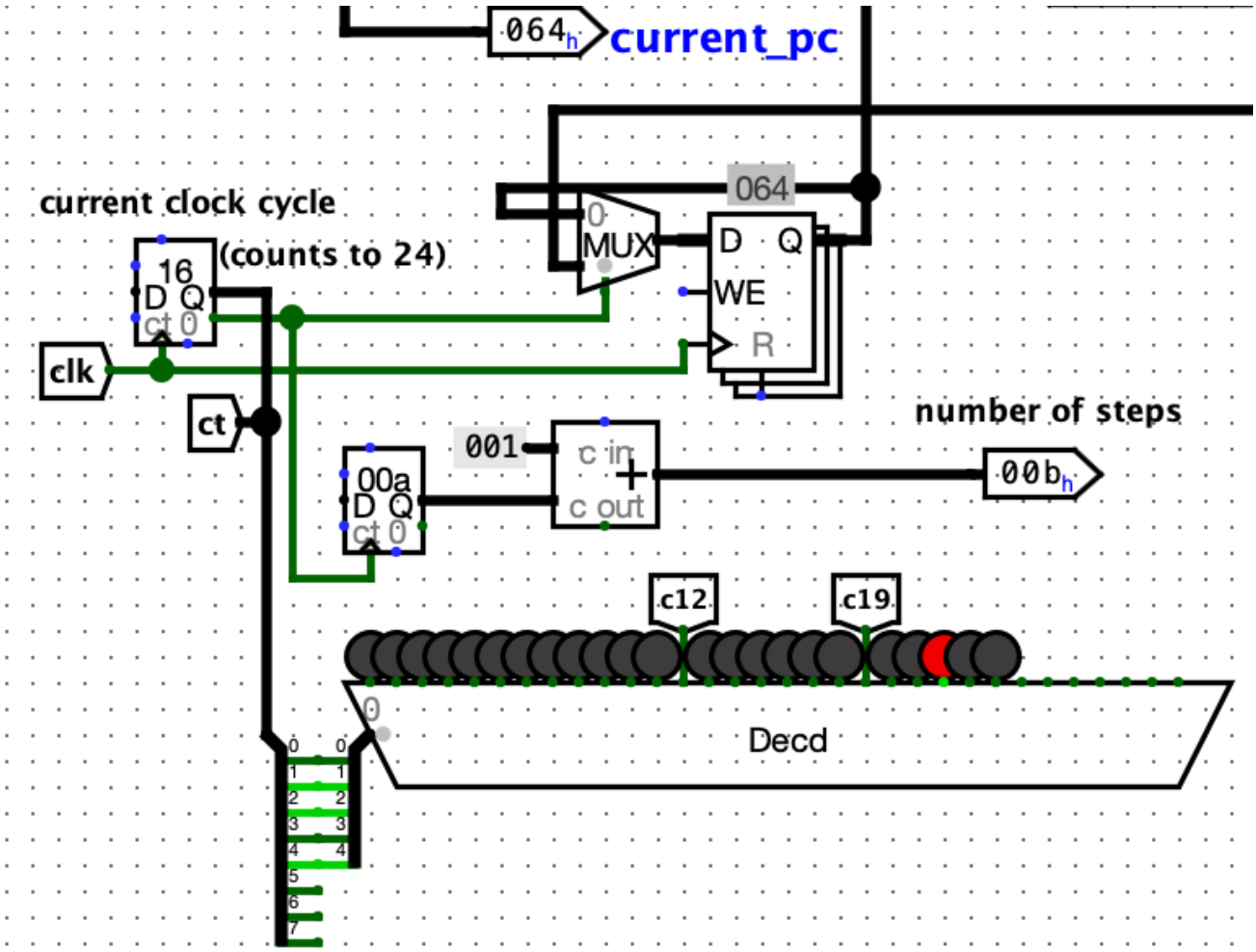
(Set Data and Address Subcircuit)

Once the values of memory data and address have been set, the operations of the main memory circuit begin. The memory data input is plugged into the RAM, and a counter as well as a mux are used to load a single byte at a time into the RAM when writing is enabled. The address that this data is written to is obtained from the subcircuit's memory address input. As long as writing is not enabled, this address will be used, and nothing in memory will be changed. Reading from the memory takes place in a similar manner to writing. The same memory address input is used to obtain the address in the RAM to read from, and a counter (the same one used for writing) is also used to cycle through the value being read, one byte at a time. Each byte is placed in one of eight registers, and when the cycling is done, the data from these eight registers is loaded into a final output called valM, which is written back to the Register File subcircuit (as discussed earlier in the Decode/Write Back stage section of this report).

PC Update

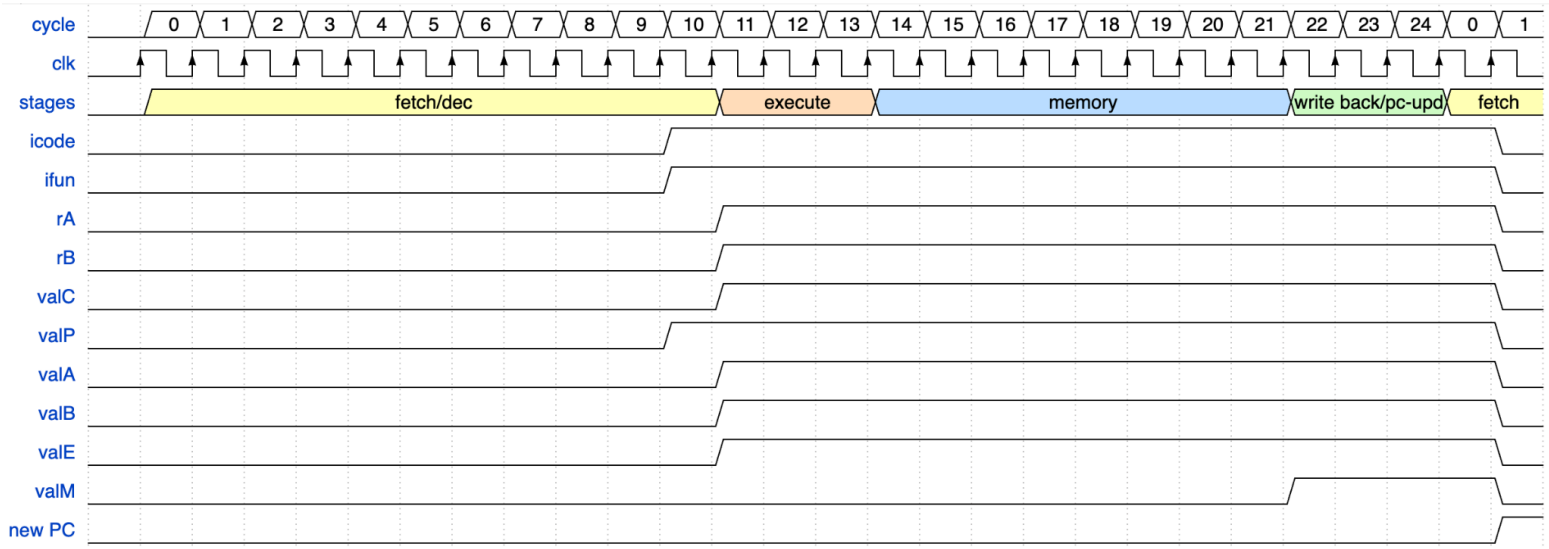
Our PC Update stage takes the first 12 bits of data from valC, along with valP, and determines what the next value of the program counter will be. A single decoder is used to determine whether the current instruction is a type of jump. If the current instruction is a type of jump and the condition (which is based off of the conditional codes set by the most recent ALU operation) is true, then the new PC value will be valC. Otherwise, the new PC value will be valP, which was calculated as a result of the byte size of the instruction in the Fetch stage.

Below is a photo of a part of the main processor circuit. The new PC value is loaded into a mux, which waits for the counter to reach its final cycle (24) before loading the new PC value. Until this happens, a register is used to maintain the current PC value without changing it. The new PC value is then loaded into the Fetch stage to start the next instruction and repeat the entire process over again.



(Location of Current PC Value)

Timing Diagram



(Timing Diagram)

The above timing diagram includes when the following values are guaranteed to be obtained: icode, ifun, rA, rB, valC, valP, valA, valB, valE, valM, and new PC. Some of these values are obtained earlier than the timing diagram states, although this is not a problem or a detriment to the performance of the processor, since no value is ever used until the correct clock cycle according to the timing diagram. This allows for synchronization between the different subcircuits and stages of the processor, and it also ensures that no values are used at an improper time. The stage names and the range of cycles that each stage uses to complete its task are showcased in the timing diagram as well. The timing diagram displays the timing of one single instruction, and every type of instruction that the processor is capable of performing follows this timing diagram.

All necessary values (icode, ifun, rA, rB, valC, valP, valA, valB, and valE) are obtained by the end of the Fetch stage, which finishes at cycle 10. The Execute stage then is fully complete by the end of cycle 13, and the Memory stage follows right after, taking exactly eight cycles (from cycle 14 to 21) to either write or read to the memory (or do neither), depending on the current instruction. The remaining cycles are used to write back to the register files and make sure the program counter is ready to be updated. On cycle 0 of the next instruction (since our timing diagram is zero-indexed and the cycles of a single instruction range from 0 to 24), the current PC value is changed to either valP or valC, which is decided in the logic of the PC Update stage.

Decode/Write Back Table

#	instructions	srcA	srcB	dstE	dstM		
0	halt	0xF	0xF	0xF	0xF		
1	nop	0xF	0xF	0xF	0xF		
2	rrmovq rA, rB	rA	0xF	rB	0xF		
3	irmovq V, rB	0xF	0xF	rB	0xF		
4	rmmovq rA, D(rB)	rA	rB	0xF	0xF		DECODE /
5	mrmovq D(rB), rA	0xF	rB	0xF	rA		WRITE BACK
6	OPq rA, rB	rA	rB	rB	0xF		
7	jXX Dest	0xF	0xF	0xF	0xF		
A	pushq rA	rA	%rsp	%rsp	0xF		
B	popq rA	0xF	%rsp	%rsp	rA		

Memory Table

#	instructions	mem read	mem write	mem addr	mem data		
0	halt	0	0	X	X		
1	nop	0	0	X	X		
2	rrmovq rA, rB	0	0	X	X		
3	irmovq V, rB	0	0	X	X		
4	rmmovq rA, D(rB)	0	1	valE	valA		MEMORY
5	mrmovq D(rB), rA	1	0	valE	X		
6	OPq rA, rB	0	0	X	X		
7	jXX Dest	0	0	X	X		
A	pushq rA	0	1	%rsp	valA		
B	popq rA	1	0	%rsp	X		

PC Update Table

#	instructions	PC					
0	halt	X					
1	nop	valP					
2	rrmovq rA, rB	valP					
3	irmovq V, rB	valP					
4	rmmovq rA, D(rB)	valP					PC UPDATE
5	mrmovq D(rB), rA	valP					
6	OPq rA, rB	valP					
7	jXX Dest	valC/valP					
A	pushq rA	valP					
B	popq rA	valP					