312 Final Group Project Report
Members: Joshua Hare, Oreoluwa Ogunleye-Olawuyi, Naveed Haq

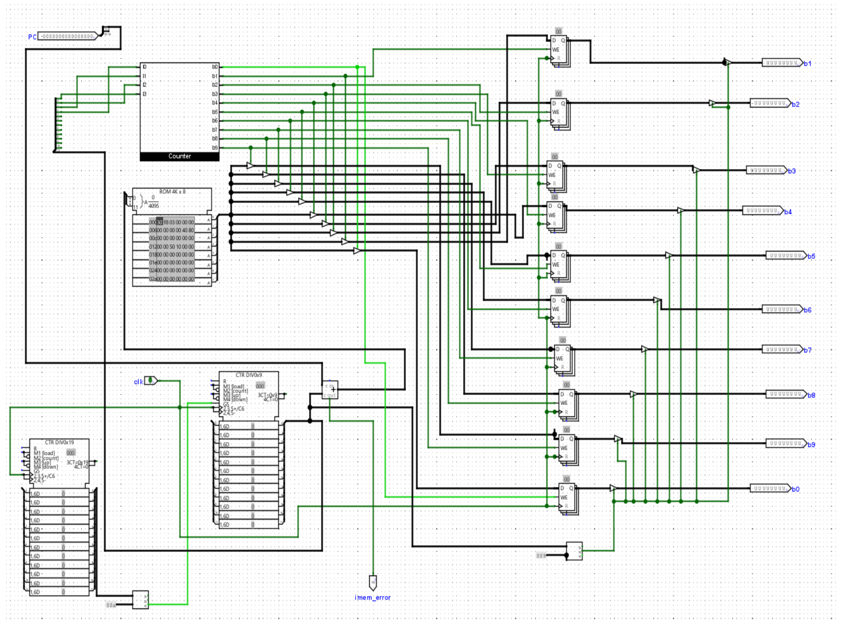
Introduction

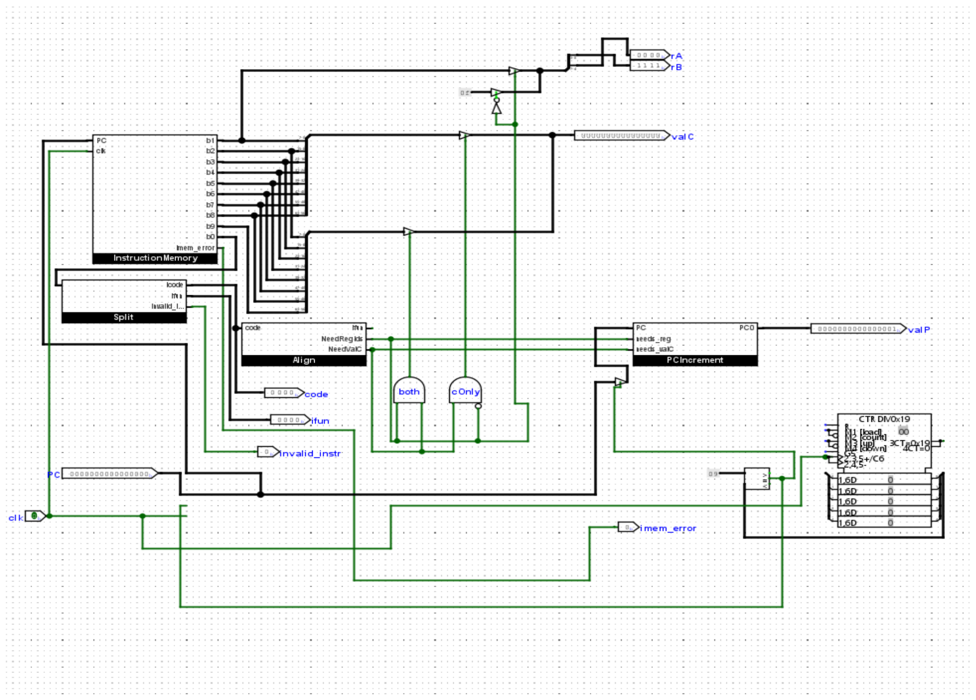
The implementation of a processor capable of executing programs written in Y86-64 ISA is implemented. The architecture is split into methods fetch, decode, write-back, execute, memory, and PC-Update. Tables used to implement these methods will be provided in a different document. The document will help understand how the planning that went into implementing each of the methods used in the processor architecture.
Another document has also been attached with the timing diagrams for the processor.
The processor was then tested. Ys and Yo files were written with possible instructions from the Y86-64 ISA, and have been attached. The .mem files have also been included in the folder.

Fetch(Joshua and Ore)

The fetch block of the processor gets a large set of instructions from memory (10 bytes), then breaks it up into the instructions that you can use to perform operations with the memory. The read values are icode, ifun, valC, rA, rB, and valP. This stage happens on cycles 0-9. It reads one byte every cycle from the ROM and once it has read all ten bytes, it assigns the data to the appropriate spot depending on the icode. It then enters the PC Increment block where valP is calculated. Also, signals for possible errors are outputted.

In between the fetch and decode stage, all of the values that were just read in are stored in registers so we can use them later. The first picture is our instruction memory that contains the ROM. The second picture is our entire fetch stage.

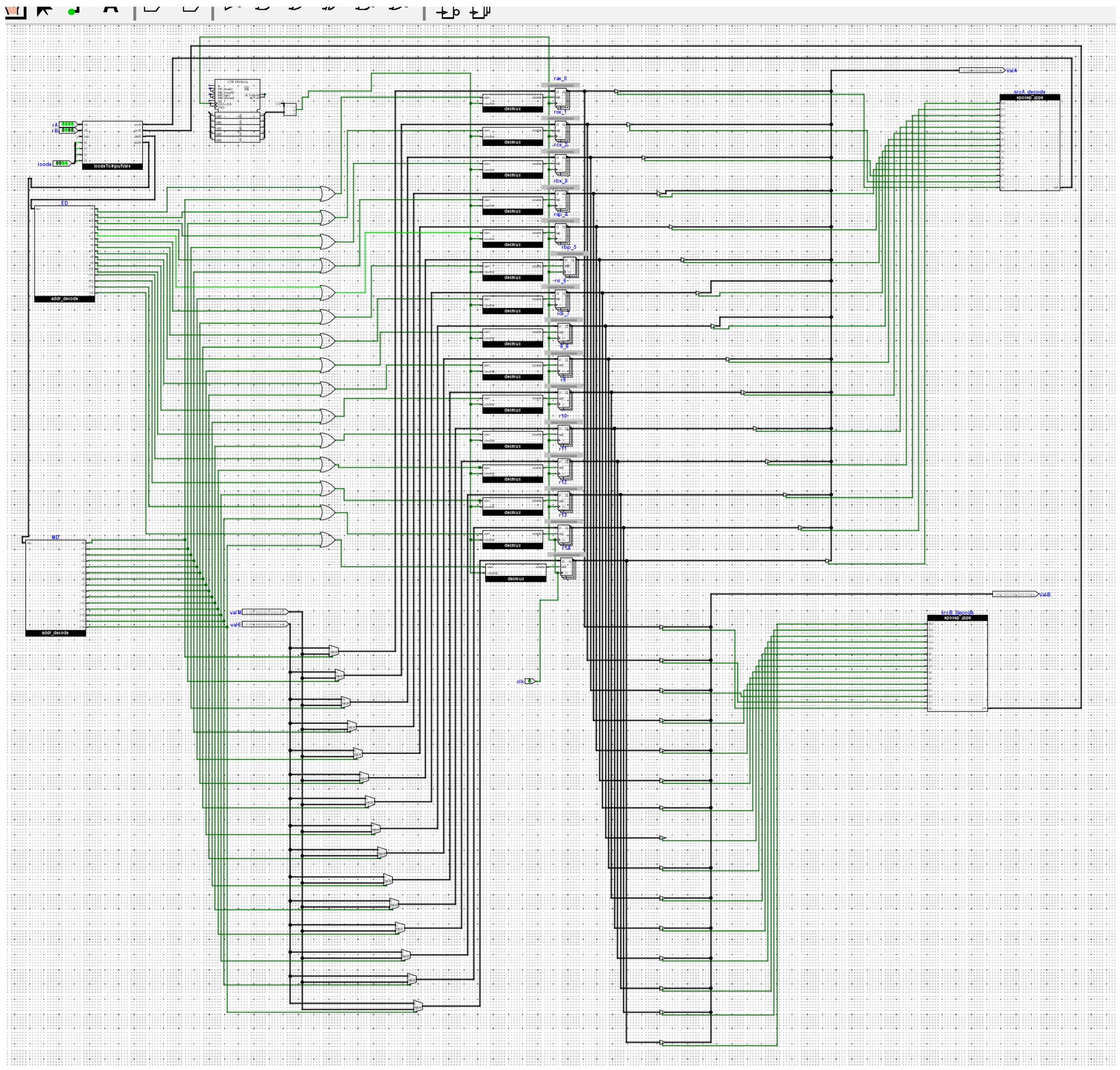

PC
Counter
ROM 4K x 8
CTR DIV0x19
clk
b1
b2
b3
b4
b5
b6
b7
b8
b9
b0
imem_error

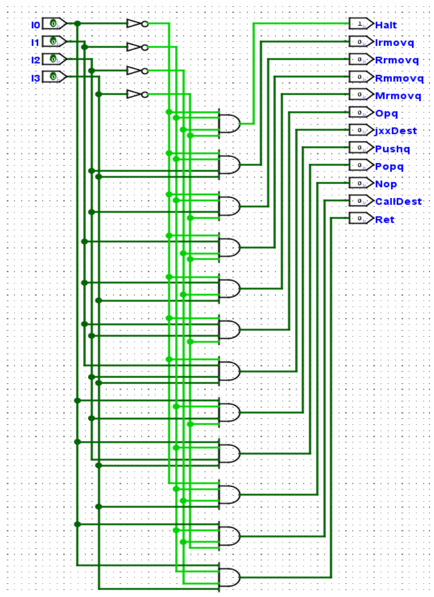

rA
rB
valC
InstructionMemory
Split
Align
PCIncrement
valP
both
cOnly
code
ifun
invalid_instr
PC
clk
imem_error
CTR DIV0x19

Decode and Writeback (Ore and Naveed)

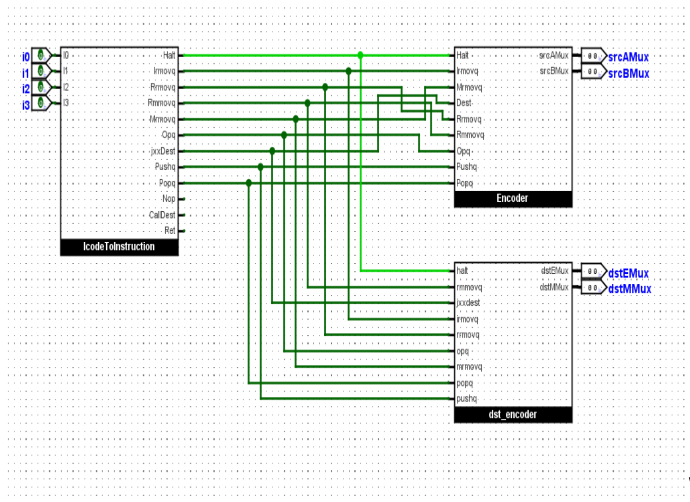
The decode stage, using a broad definition, allows the CPU to determine what operation or instruction needs to be performed. It reads data from the register file. The writeback stage is the final stage in executing instructions. It updates the processor with the results of the operation executed. It writes data to the register file.

For our implementation of the decode and writeback stage, we recognized that we would be getting icode as an input. The icode would be used to determine what operation would be performed. The write back happens after the memory stage. We used the 4-bit icode to generate select signals that would be connected to two muxes which determined the sources and destinations.

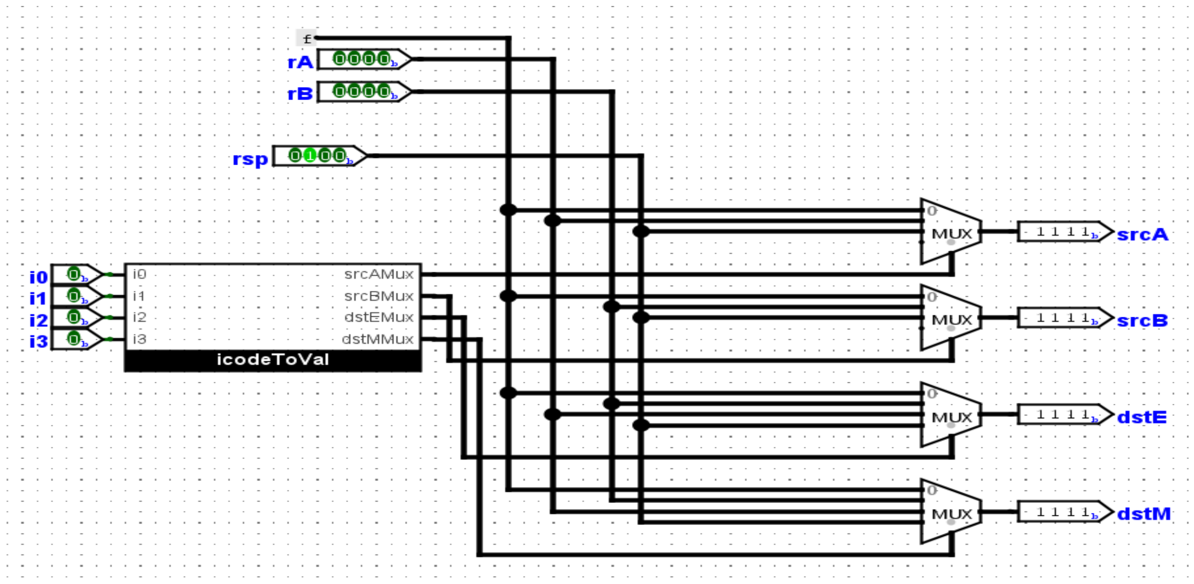




One mux would be used for srcA, and srcB. The other mux would be used for dstE and dstM. Values were generated for srcA, srcB, dstE, and dstM based on inputs rA, and rB.

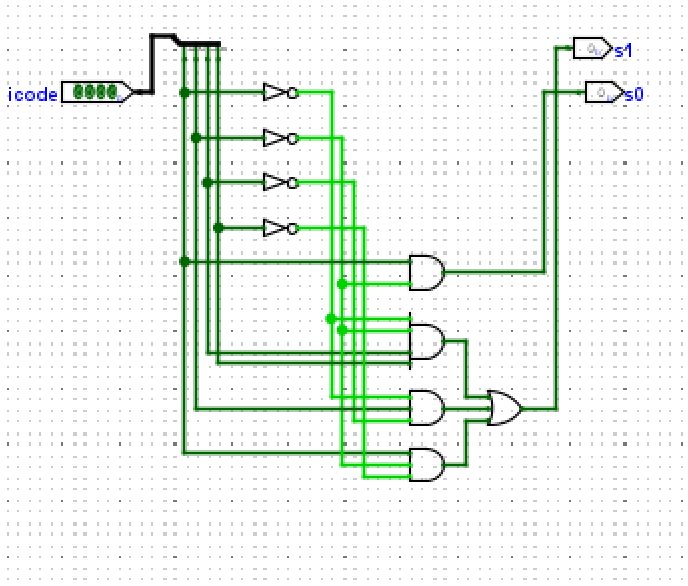


We implemented a register file that would allow values to be stored in a registers based on the index provided, and allow values to be written back to valE and valM.

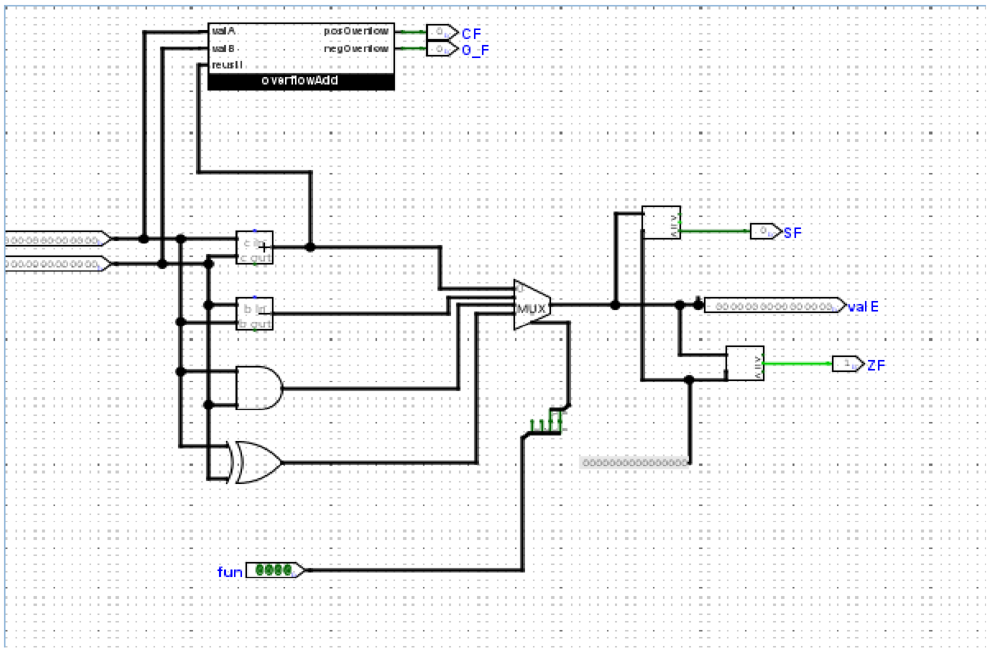


In general, srcA and srcB were the register file's read address inputs. valA and valB were the read data inputs. dstE and dstM were the write address inputs, and valE and valM were the write data inputs.
The table for which register would be written to based on the operation to be performed is provided above.

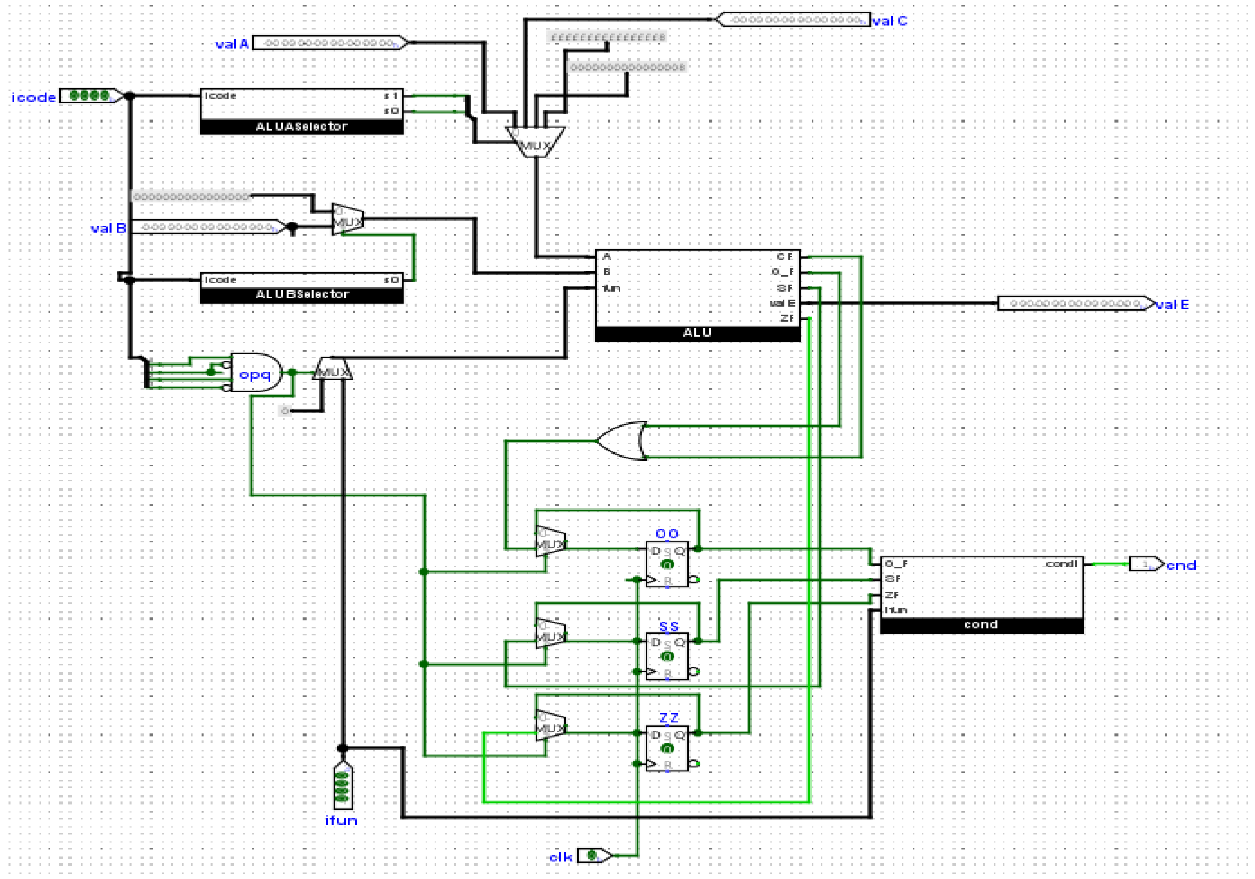
Execute(Oreoluwa and Naveed)
For the execute block, ALU selectors were implemented using icode. The implementation is provided below. Execute also stores the zero, negative, and overflow flags so that the jXX operation can be performed. The ALU selectors determine the operands and the ifun determines the operation. This all goes into the ALU which outputs valE



The implementation for the ALU is also provided below. The possibility of overflow was handled:

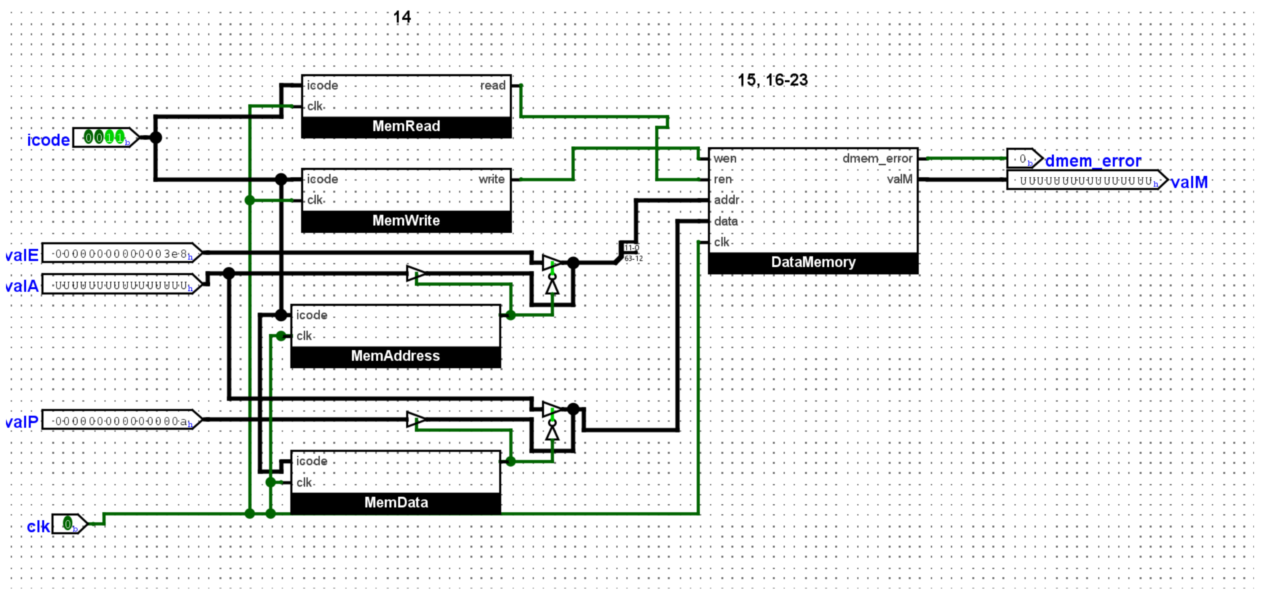


The full circuit for the execute:

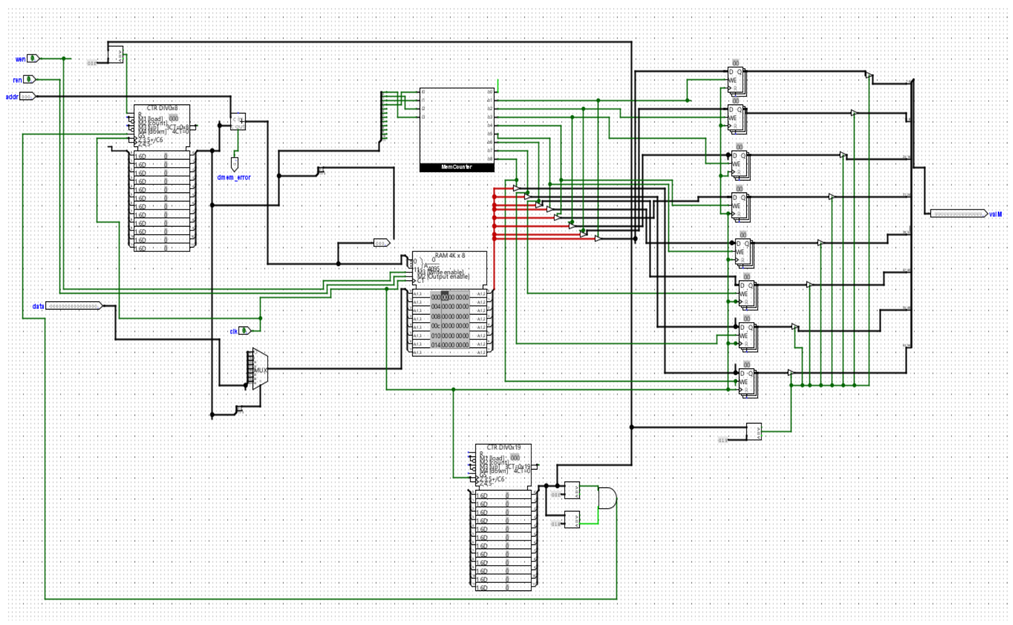


Memory(Joshua and Naveed)

Based on the implementation approach provided for memory, we implemented memRead and memWrite based on icode. These blocks output the read and write enables for our RAM. We then fed the output of the memRead and memWrite into a dataMemory circuit. We also chose what we are writing/reading with memAddress and memData. All of these values are then fed into the DataMemory unit. The dataMemory unit then reads or writes from the RAM. It takes one cycle per byte. The output of the dataMemory circuit was valM and we also accounted for memory errors.



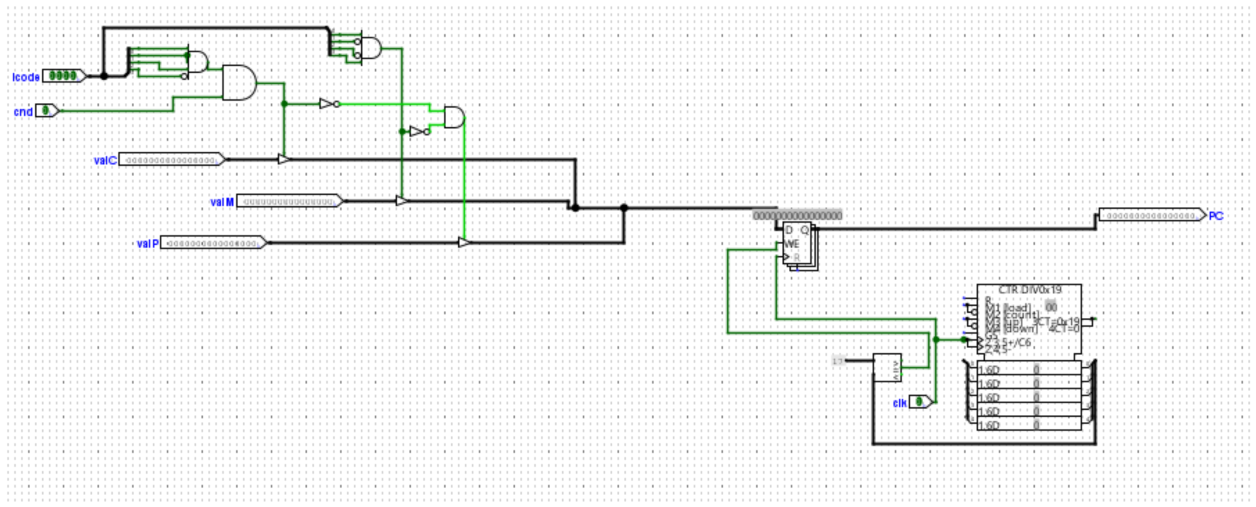
The implementation for dataMemory is provided. This is where the RAM is located:



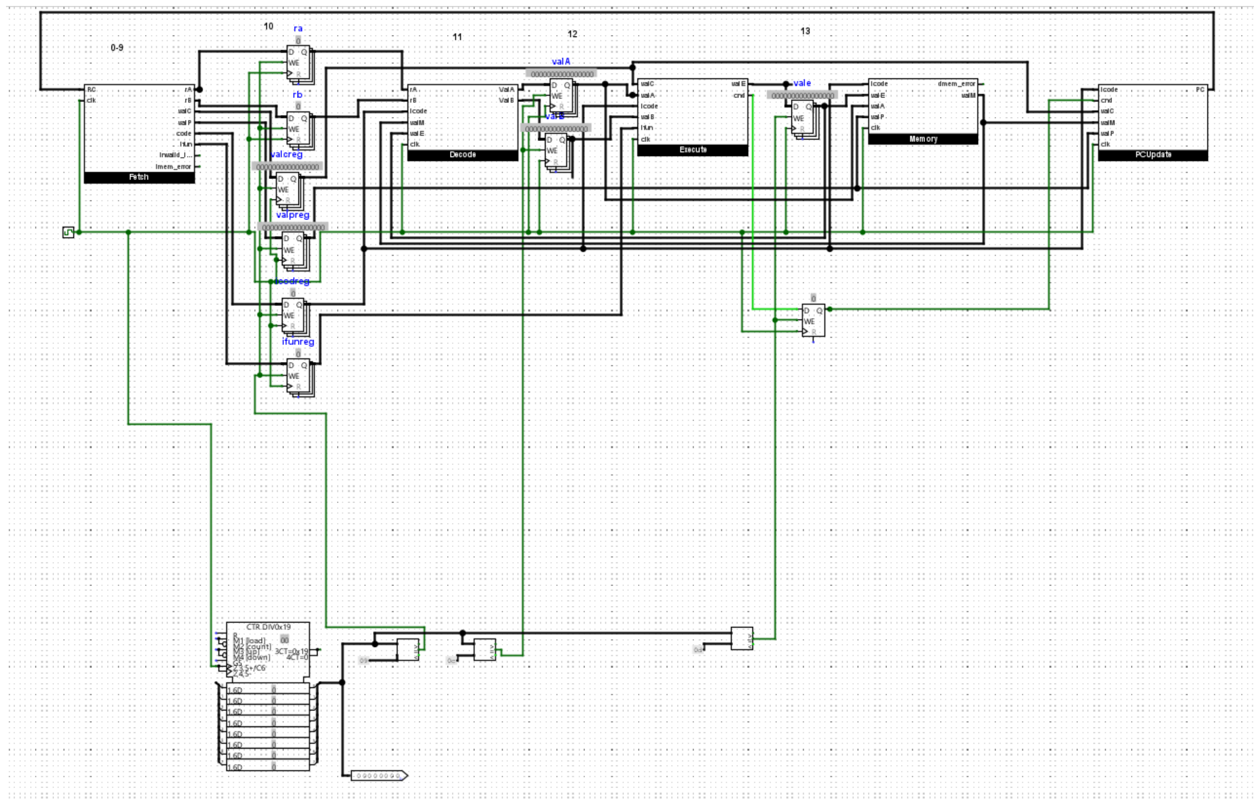
PC Update(Josh and Naveed)

The PC update method finds the next value of the PC so the next instruction can execute. It picks its input based on icode.

The implementation is provided below:



Then we combined all the methods to create the full circuit. The circuit is provided below: The full time to execute one instruction in 27 cycles.


0-9
10
11
12
13
ra
rb
valcreg
valpreg
icodreg
ifunreg
valA
valB
vale
Fetch
Decode
Execute
Memory
PCUpdate
CTR DIV0x19