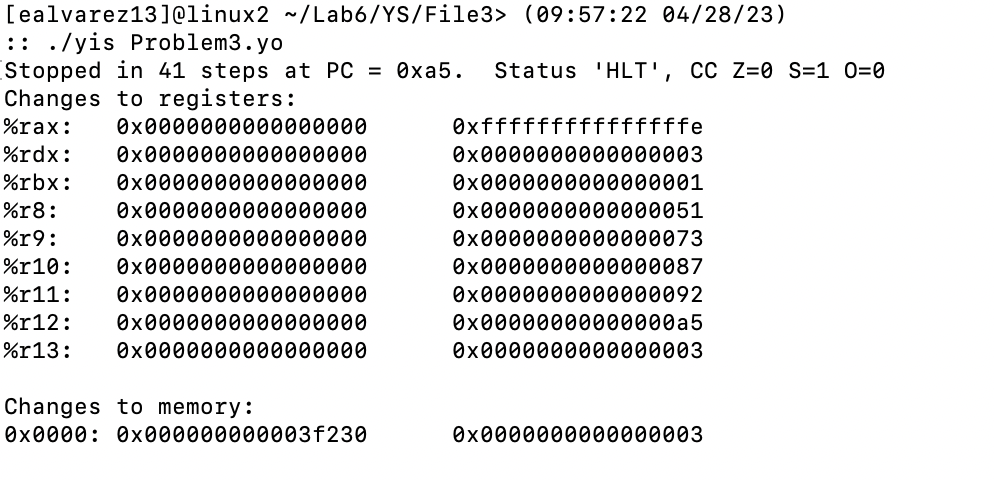
**1. Instruction to Demo**

1.1 Test Programs:

Problem 1 (Folder: File3)

In this file the following operations are performed and therefore tested:

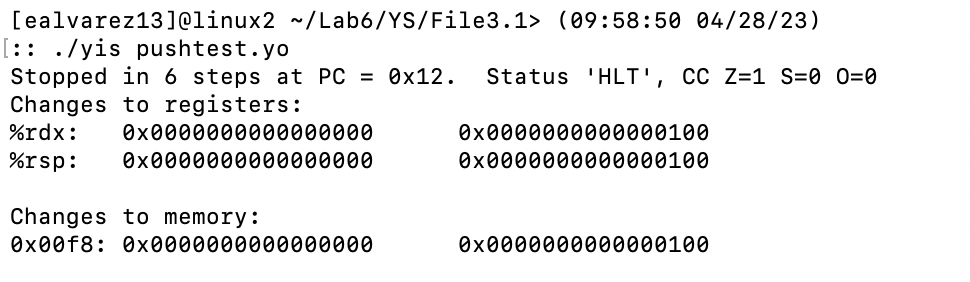
* irmovq - jge
* rmmovq - addq
* mrmovq - nop
* subq - jg
* jl - je
* jmp - halt



Problem 2 (Folder: File3.1)

In this file the following operations are performed and therefore tested:

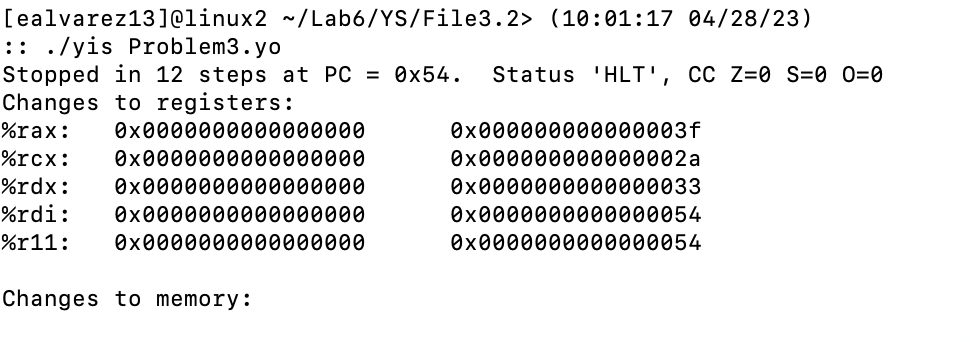
* pushq
* popq
* rrmovq



Problem 3(Folder: File3.2)

In this file the following operations are performed and therefore tested:

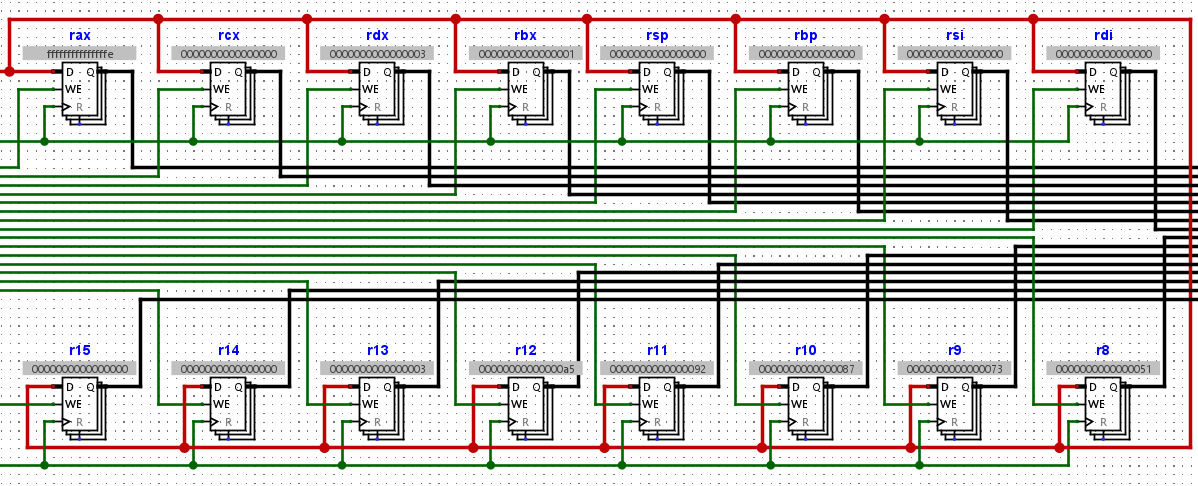
* andq - jle
* xorq - jne



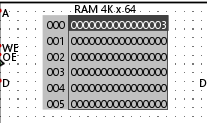
1.2: Program run on processor:

Problem 1 (Folder: File3)

screenshot of inside register file, notice all values are as expected

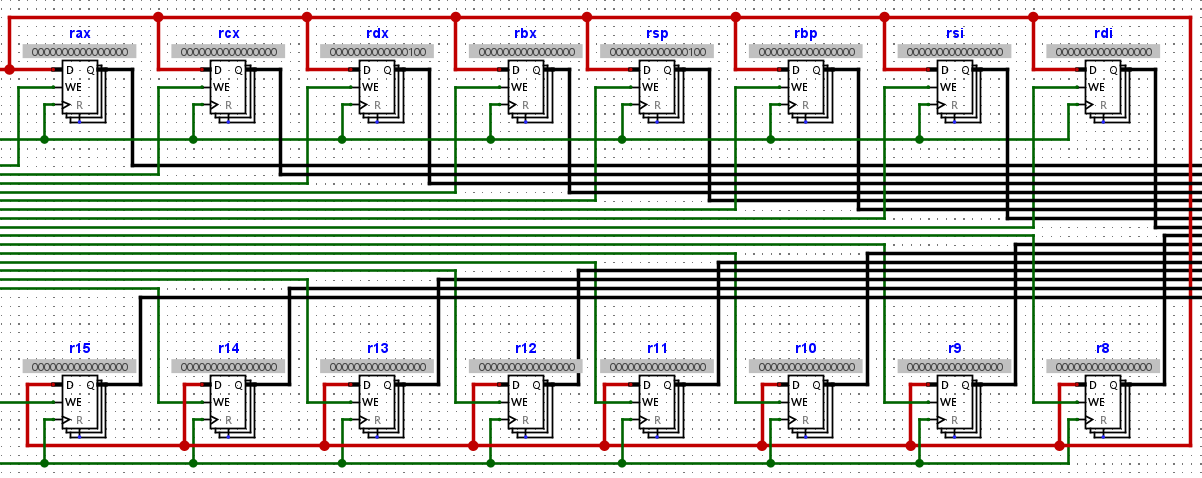


screenshot of ram inside of memory unit, value is as expected

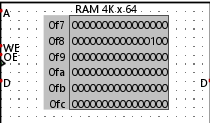


Problem 2 (Folder: File3.1)

screenshot of inside register file, notice all values are as expected

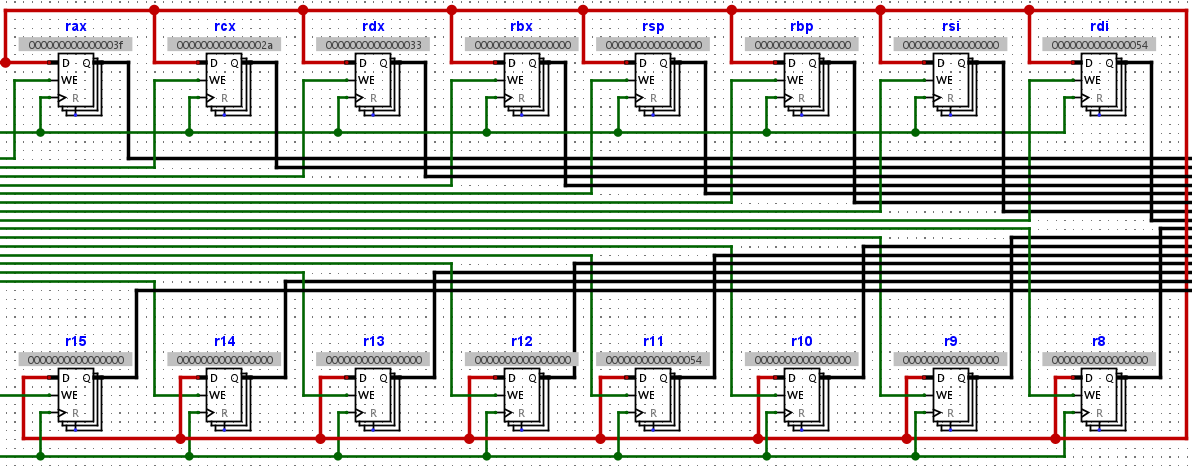


screenshot of ram inside of memory unit, value is as expected

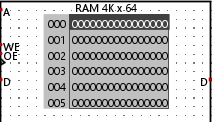


Problem 3(Folder: File3.2)

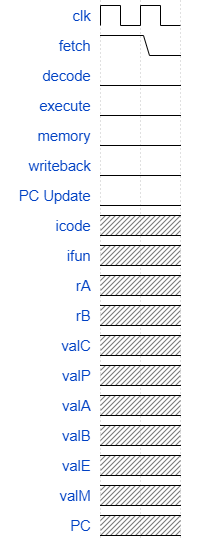
screenshot of inside register file, notice all values are as expected



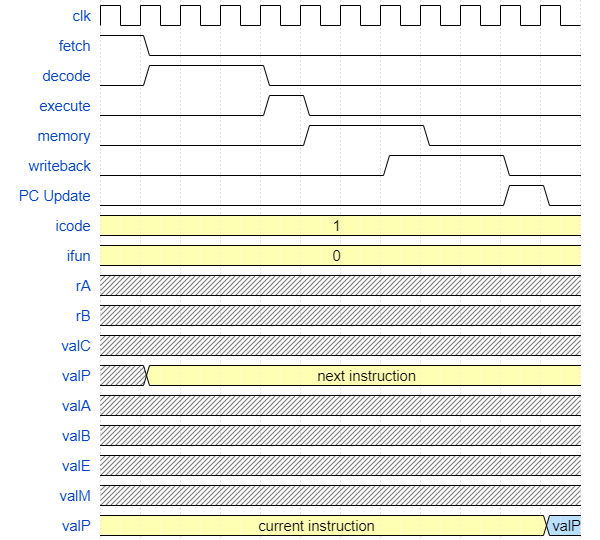
screenshot of ram inside of memory unit, value is as expected (no updates)



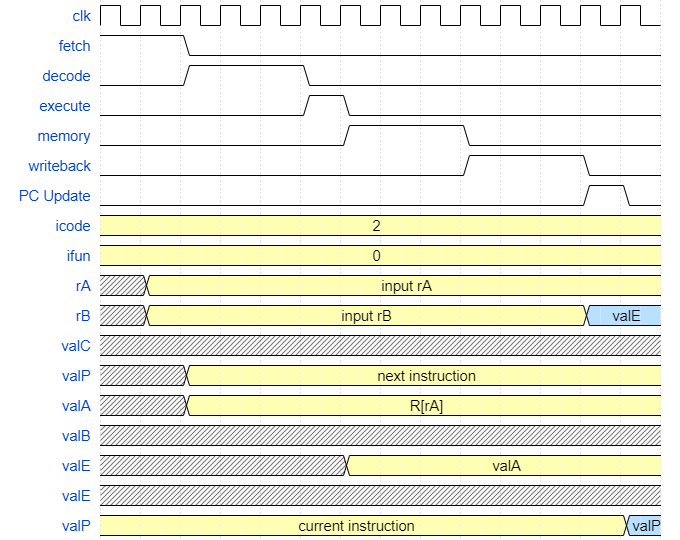
1.3 Timing of processor stages:

Halt time diagram:  


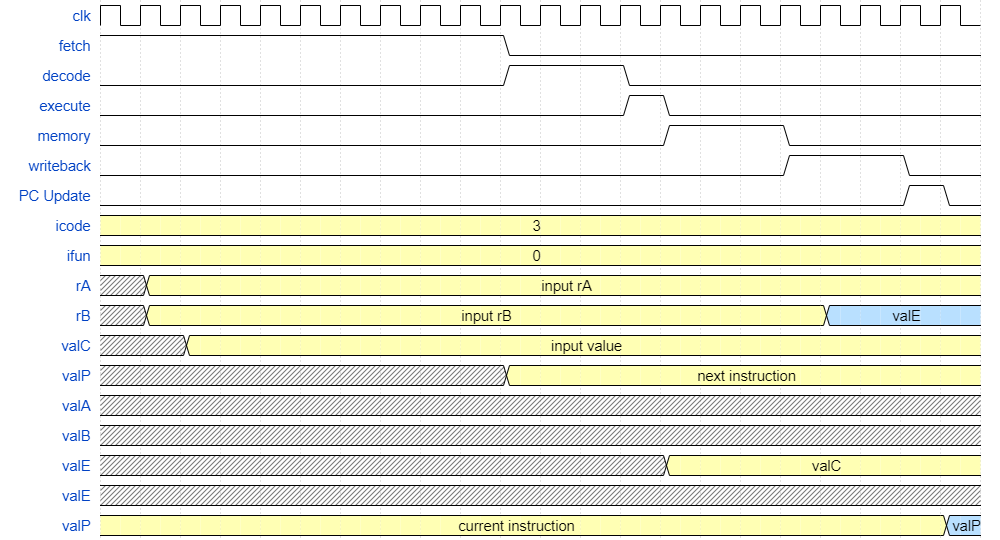
nop time diagram:



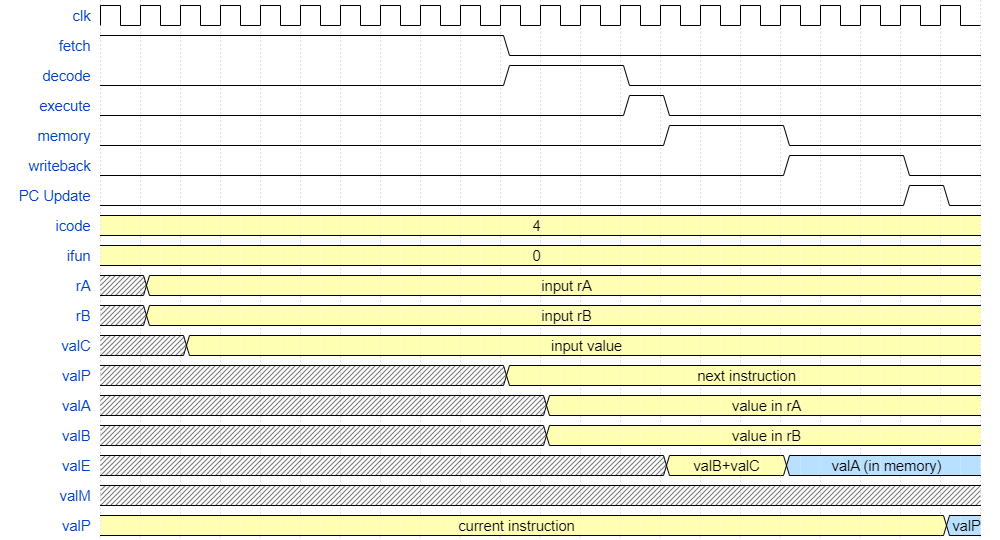
rrmovq time diagram:



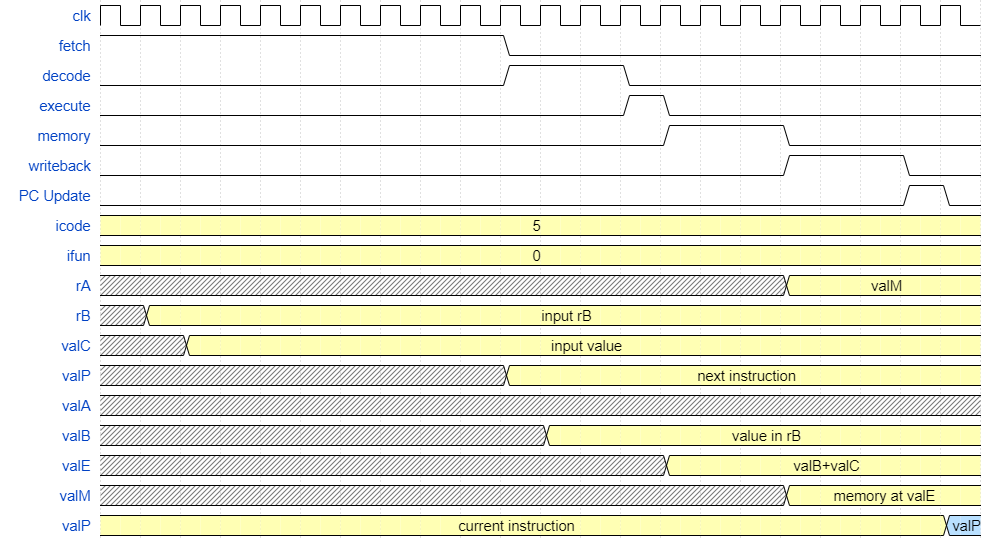
irmovq time diagram:



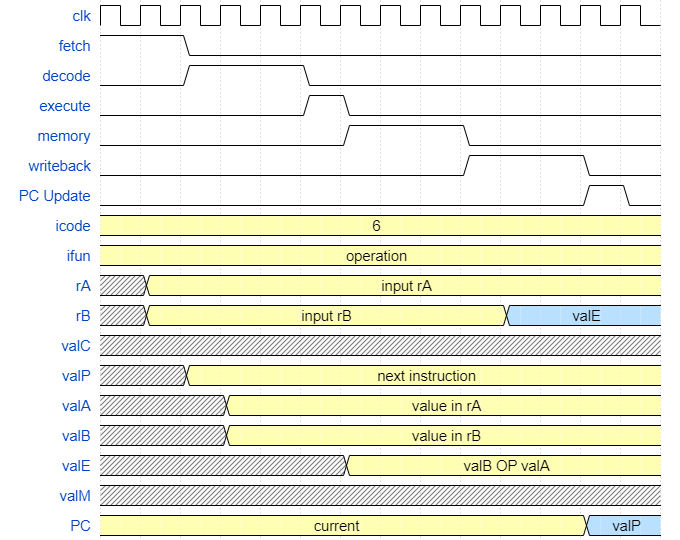
rmmovq time diagram:



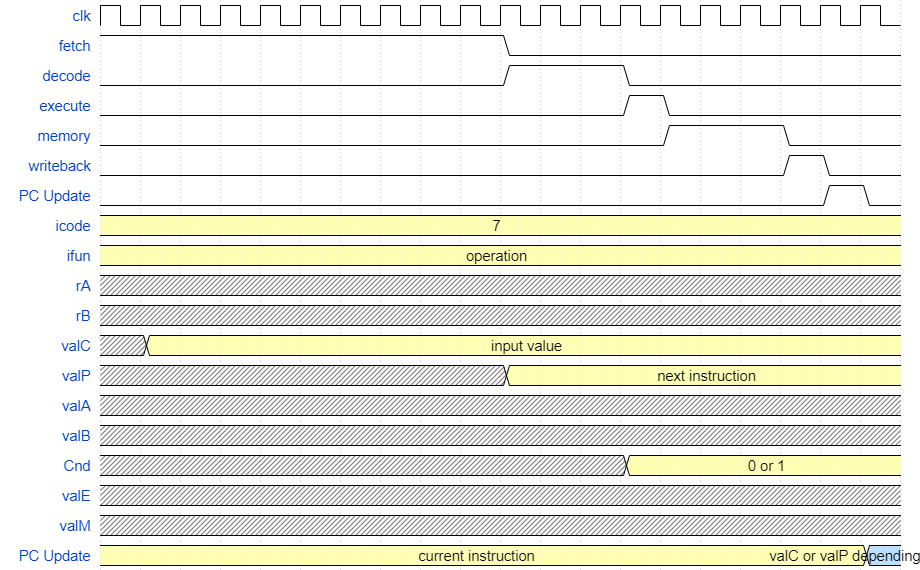
mrmovq time diagram:



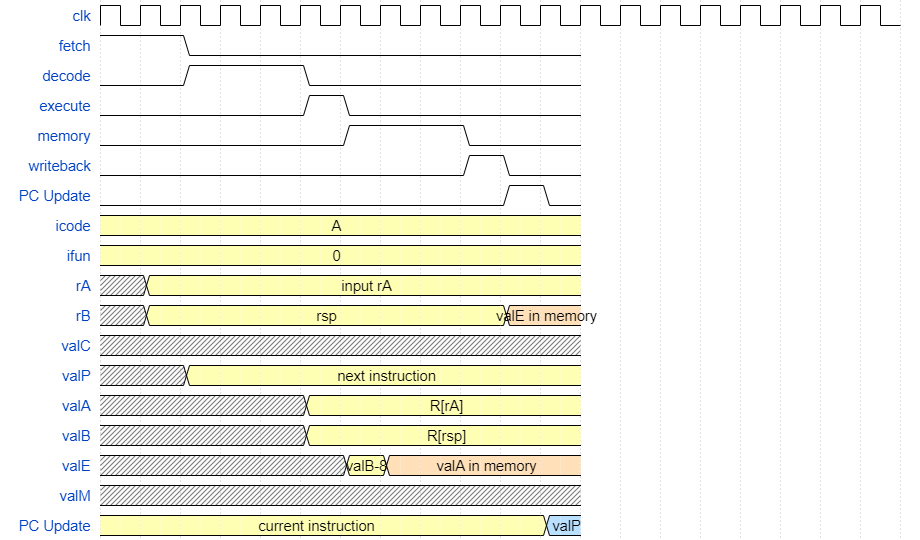
OPq time diagram:



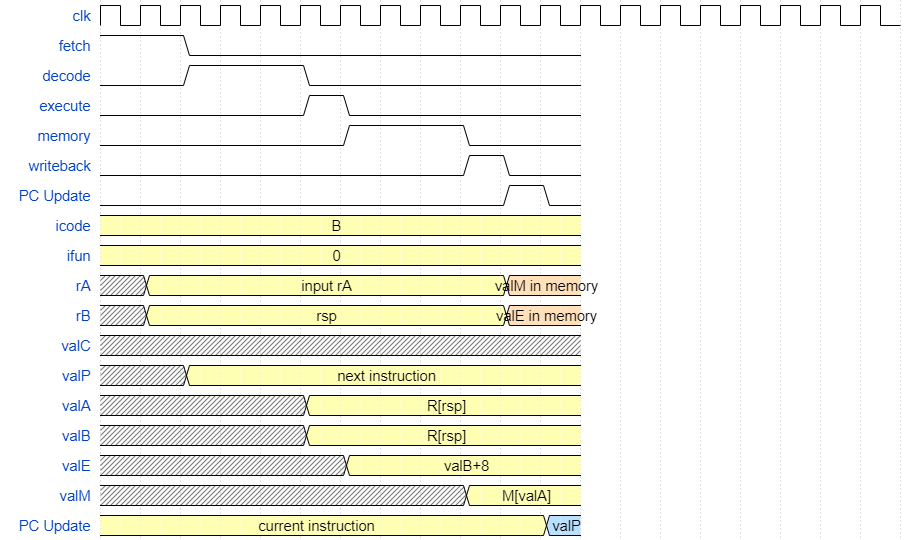
jmp time diagram:



pushq time diagram:



popq time diagram:

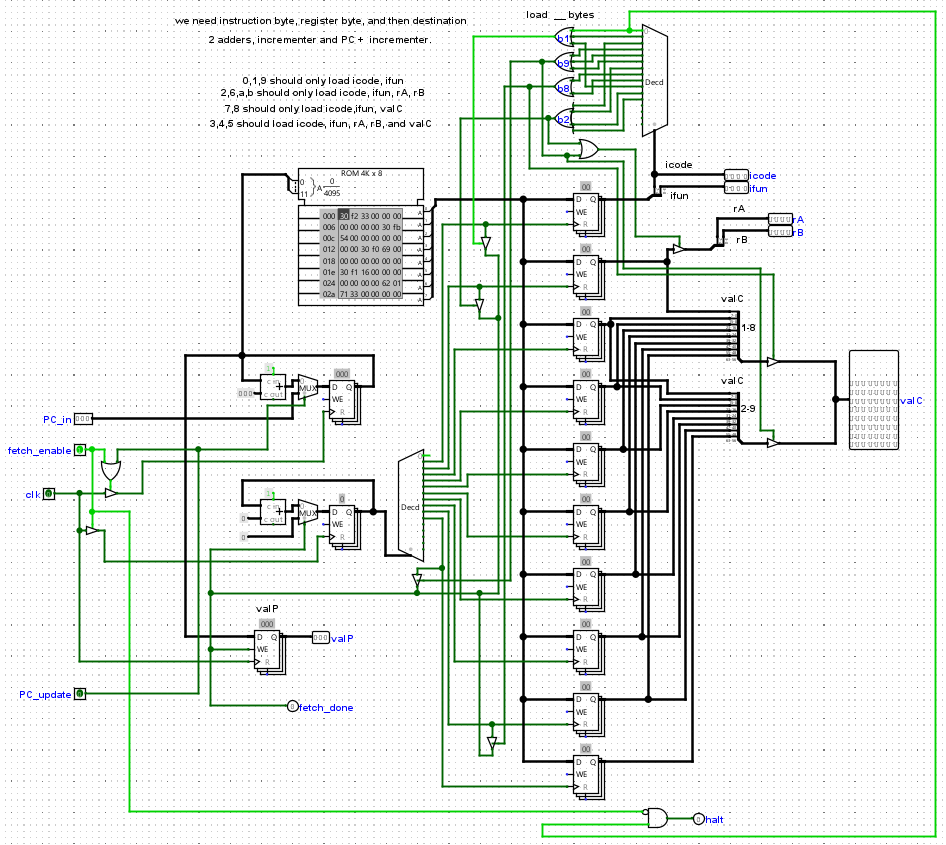


1.4 Design

**Fetch:**

Fetch takes in PC\_in in order to update the PC after each cycle, fetch\_enable to manage the FSM, clock signal, and PC\_update to help manage the FSM. The circuit has 2 incrementers, one for the PC, and another one to keep track of what Byte should be loaded. The circuit will then increment and load one byte each clock cycle as long as that is what the icode allows. icode is decoded in the top right corner and then depending on the amount of bytes that this specific instruction should read will reset the incrementer, moving onto the next section of the FSM, and saving ValP as well. Fetch outputs icode, ifun, rA, rB, valC, valP, fetch\_done, and halt.

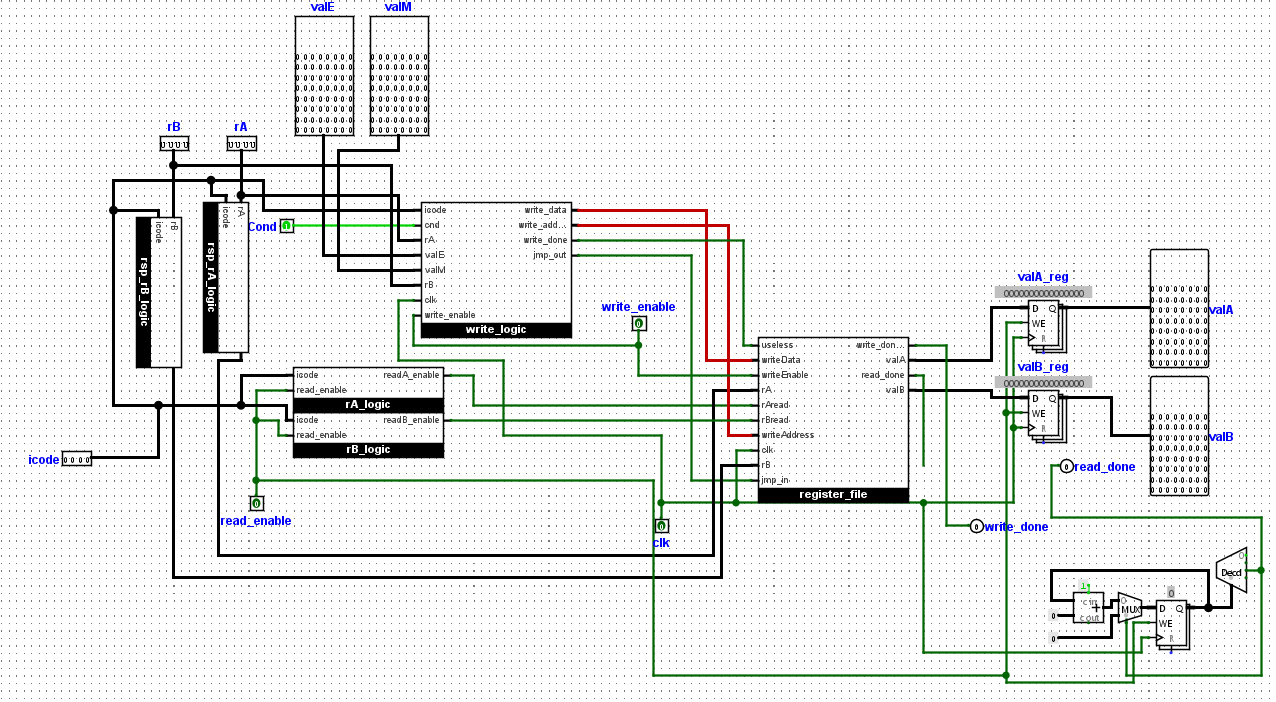
screenshot inside of Fetch genie box



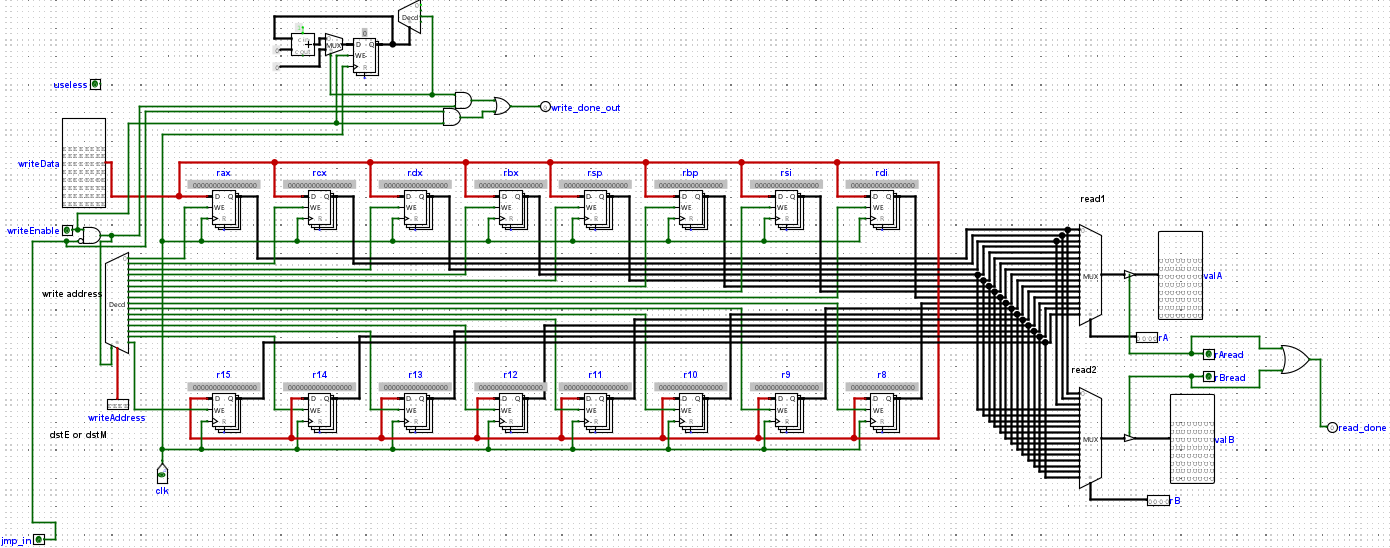
**Decode:**

Decode takes in the clock, rA, rB, icode, and a read enable flag from the finite state machine. If the read\_enable flag is on it decodes icode to determine whether to read from rA, rB, or both. These values are then passed into the 16x64bit register file. The values corresponding to the register addresses rA and rB are then read into valA and valB respectively. Again, the decoding is done prior to this to determine whether to read to valA, valB or both. The decode step was hardcoded to take 3 cycles at all times to maintain synchronization of the finite state machine.

screenshot inside of Decode genie box



screenshot inside of register file genie box



**Execute:**

Execute uses the icode in order to decode the executing function, based on this and combinational logic, a mux decides for ALU A whether to pass in (ValA/ValC/8/-8) and for ALU B whether to pass in (0/ValB).

Whenever icode = 6, the setCC will activate the register for the flags (ZF, SF, OF) since there’s an operation being performed.

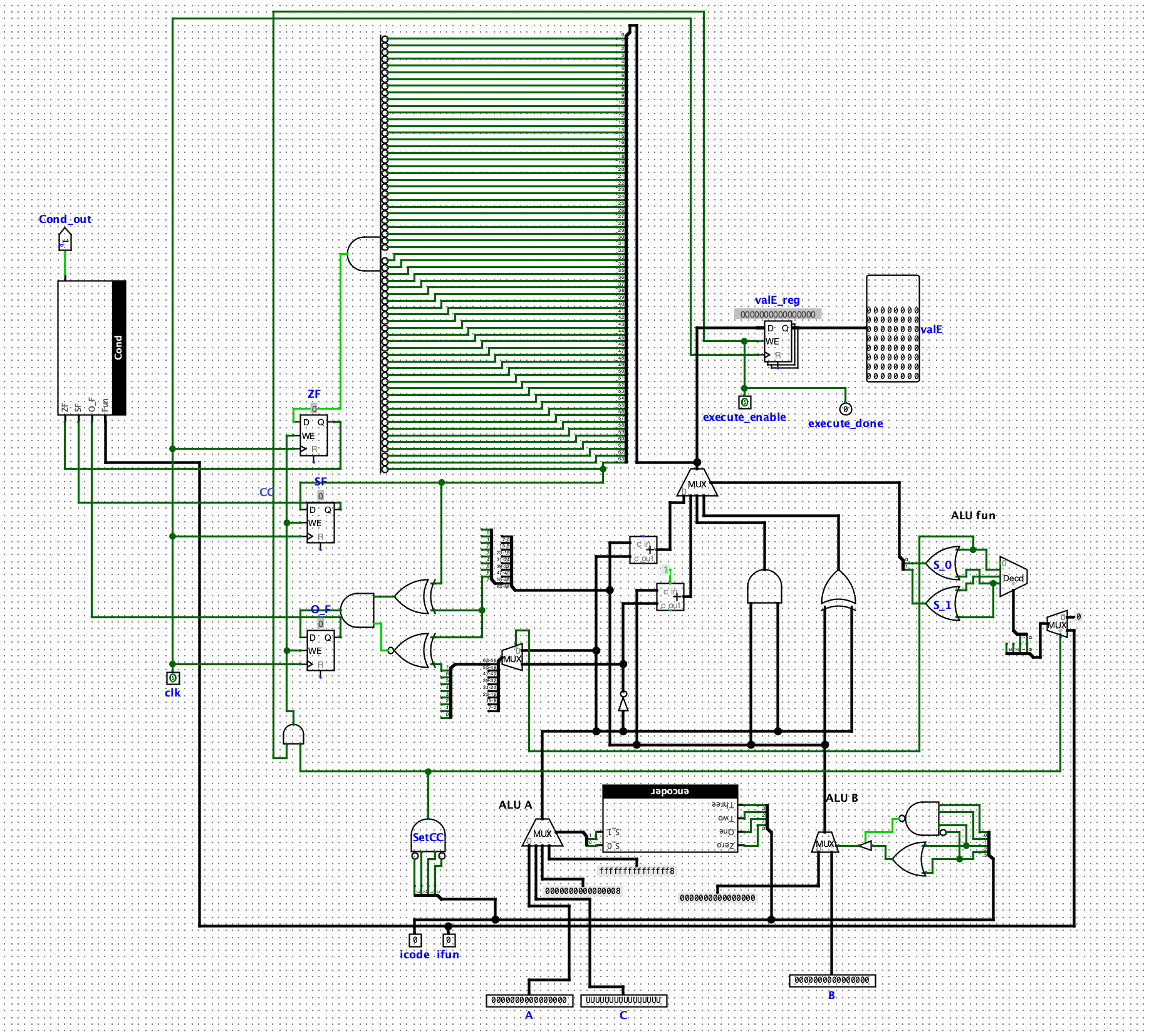
Inside the execute, ifun decides what kind of operation will be performed inside the ALU, for all operations other than OP(icode = 6) and Jmp (icode = 7) a sum will be performed between ValB and ValA. If the icode = 6 then ifun can be any number between 0-3, this will perform either a sum(0), subtraction(1), AND(2) or XOR(3) between valB and ValA.

After performing the requested operation, if the result = 0, the ZF will be set, the SF will activate if the result MSB is 1 and the OF flag will be set if overflow happens in the ALU.

The condition will be set depending on the ifun function that it is requested and the flags that are set. Eg. ifun = 2 and ZF | SF.

Lastly, ValE from the result will be stored into its register and output.

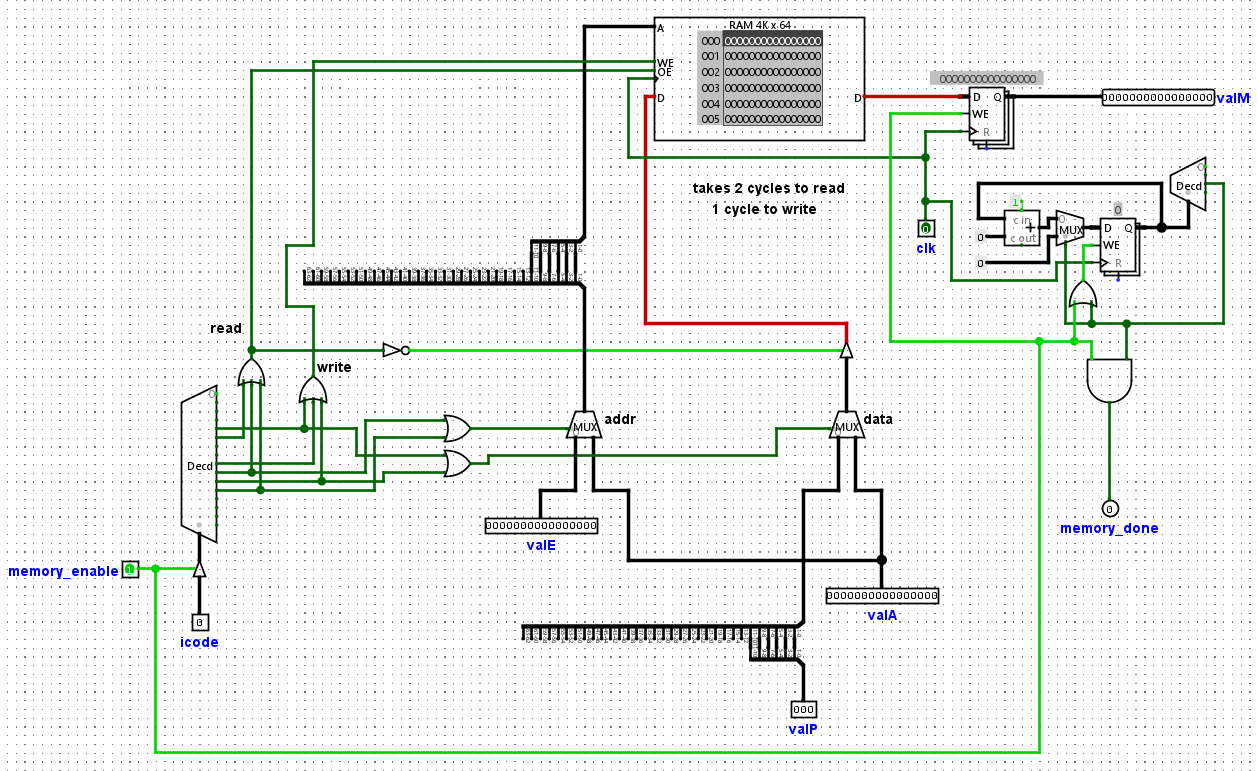
screenshot inside of Execute genie box



**Memory:**

Memory takes in clock, valE, memory\_enable to manage the FSM, valA, icode, and valP. If memory enable is on (it is that stage of the FSM) then depending on the icode, either read or write will be enabled for the RAM. Also depending on the icode the addr and data values taken in will be decided. Then the actual RAM operations will happen and the value will be saved in the register, and output through valM. Similar to the other stages the counter is used to guarantee synchronization of the FSM, outputting memory\_done after 3 clock cycles so the machine can move to the next state.

screenshot inside of Memory genie box



**Write Back:**

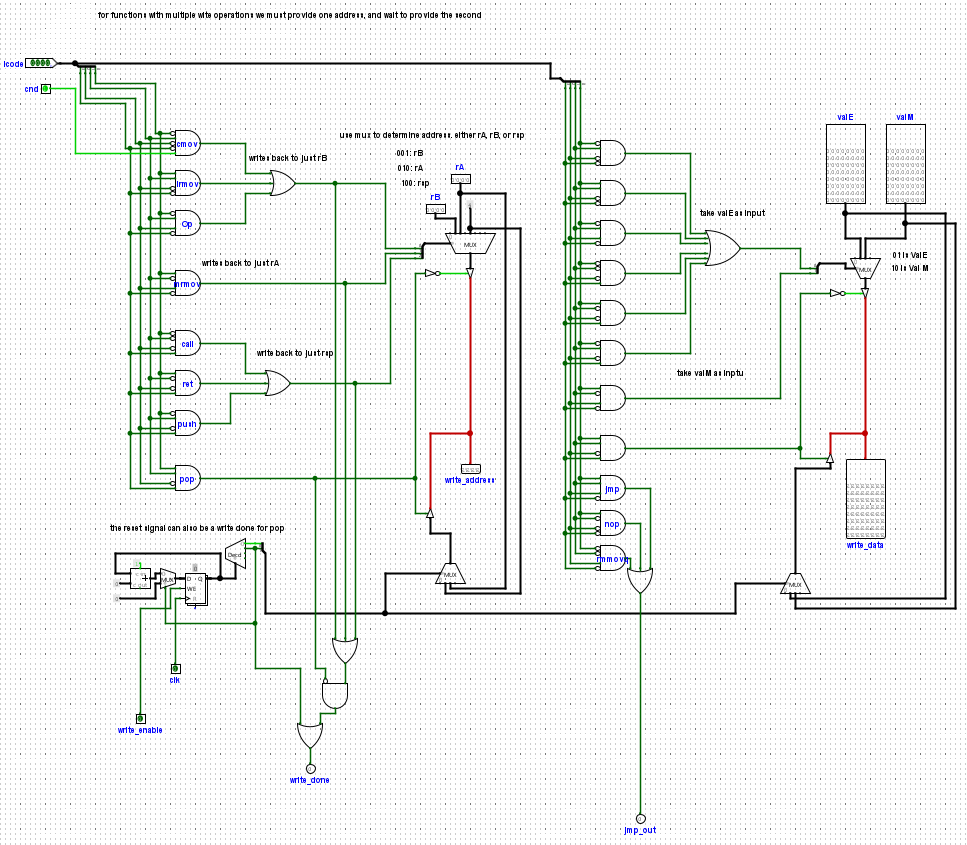
Writeback is implemented within the decode genie box. It takes in valE, valM, rA,rB, the clock, and the write\_enable flag. Inside the write\_logic box, icode is decoded to determine whether to write back to rB, rA, rsp, or the special case of popq. This is how the write address is found. The write data is determined by the same process. The write address and write data are then passed into the register file and the proper register is updated with the new value.

There are 3 special cases, nop, jmp, and pop. Both nop and jmp were handled in the same way. In the register file we did not allow anything to be written in and instead fast tracked the write\_done bit. This prevents bad data from being loaded into the register file. For popq there is a double write operation. First pop writes to rA and then it writes to rsp. This is done by delaying the second write operation in the write\_logic box. If the icode is pop then the address for rA is passed in along with valM for the write data. Once that write operation is completed the second write operation for rsp and valE is done.

Write back takes a total of 3 cycles in order to account for the worst case which is pop.

shares logic with Decode

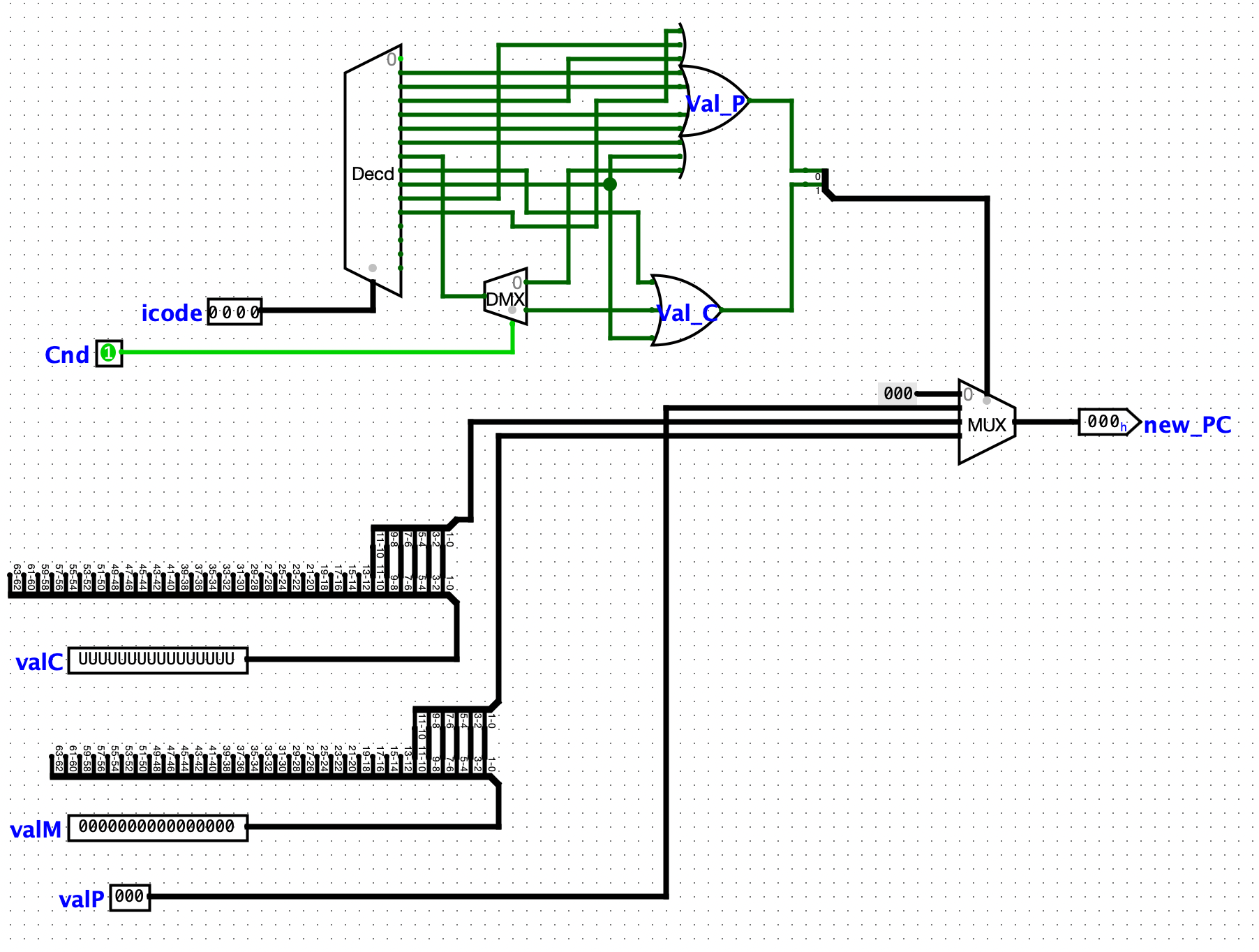
screenshot inside of write\_logic genie box



**PC update:**

PC Update takes in ValC, ValM, ValP, icode and Cnd in order to decide which will be the next value of the PC, it only considers the first 12 bits of ValC as well as for ValM since the addresses are 12 bits. A mux that uses combinational logic between the icode and Cnd decides whether to output 0, ValP, ValC, ValM.

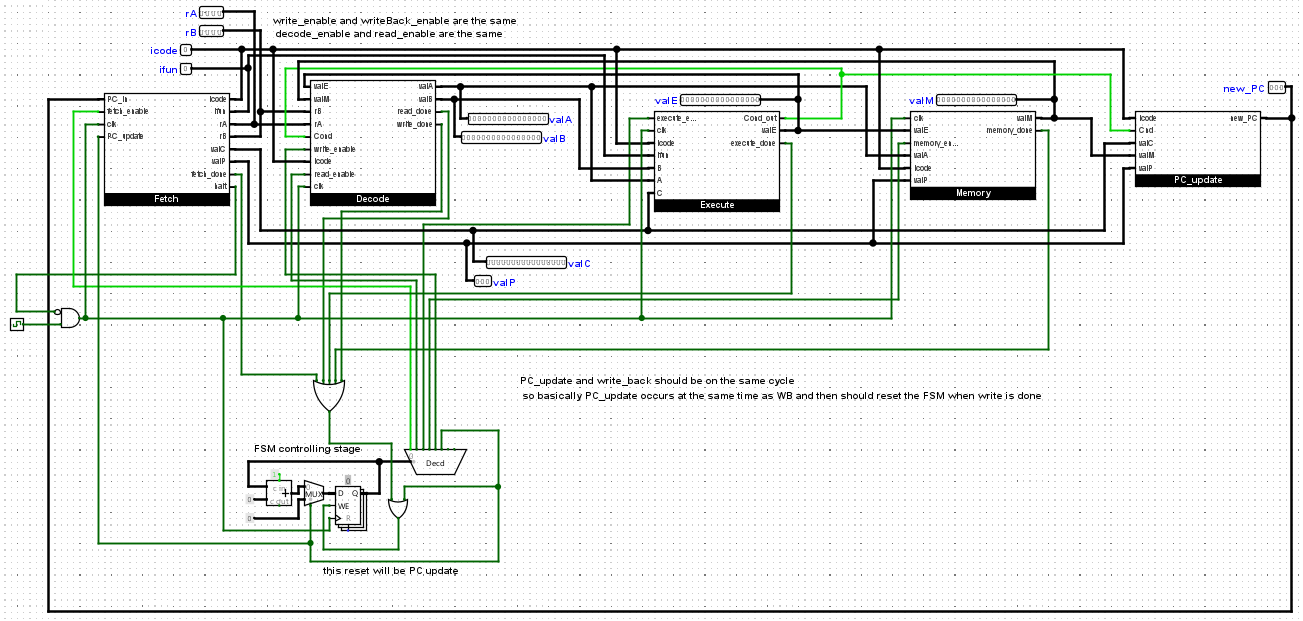
screenshot inside of PC\_update genie box

****

**Synchronization / Finite State Machine:**

Synchronizing all of the parts in our processor was done by creating a Finite State Machine. Each state of the finite state machine corresponds to one stage of the machine (fetch, decode, execute, memory, write back, and PC update). Each stage has an enable which allows it to operate before sending out a “done” signal, so that the FSM may move to the next stage. PC update being the last stage sends a PC\_Update signal into fetch and then resets the FSM to fetch once again, looping the FSM. The Halt instruction is capable of stopping the entire FSM by preventing any more clock signals from being passed into the entire machine.

screenshot of main circuit

****