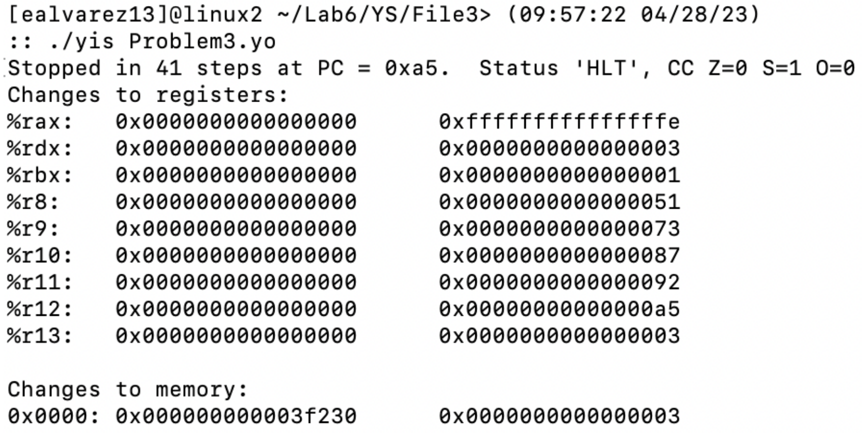
Samuel Cole, Grant Martinez, Eduardo Alvarez Hernandez
Group 14
Final Project Report

1. Instruction to Demo

1.1 Test Programs:

Problem 1 (Folder: File3)
In this file the following operations are performed and therefore tested:

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

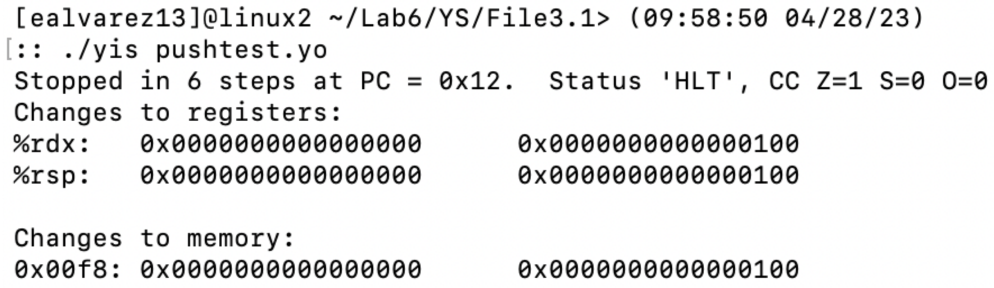


```
[ealvarez13]@linux2 ~/Lab6/YS/File3> (09:57:22 04/28/23)
:: ./yis Problem3.yo
Stopped in 41 steps at PC = 0xa5.  Status 'HLT', CC Z=0 S=1 O=0
Changes to registers:
%rax:   0x0000000000000000      0xfffffffffffffffe
%rdx:   0x0000000000000000      0x0000000000000003
%rbx:   0x0000000000000000      0x0000000000000001
%r8:    0x0000000000000000      0x0000000000000051
%r9:    0x0000000000000000      0x0000000000000073
%r10:   0x0000000000000000      0x0000000000000087
%r11:   0x0000000000000000      0x0000000000000092
%r12:   0x0000000000000000      0x00000000000000a5
%r13:   0x0000000000000000      0x0000000000000003

Changes to memory:
0x0000: 0x000000000003f230      0x0000000000000003
```

Problem 2 (Folder: File3.1)
In this file the following operations are performed and therefore tested:

- pushq
- popq
- rrmovq



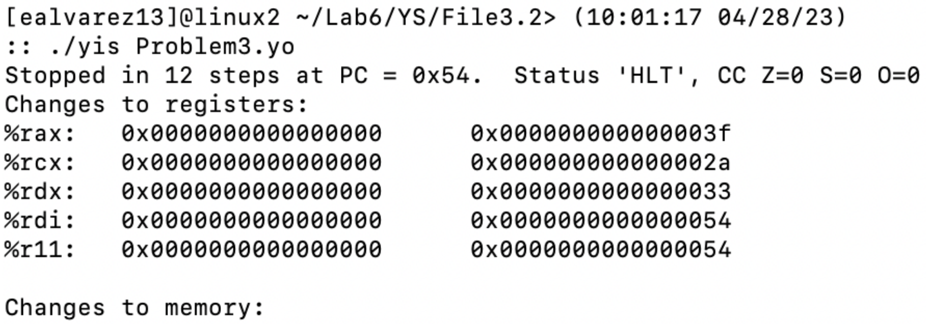
```
[ealvarez13]@linux2 ~/Lab6/YS/File3.1> (09:58:50 04/28/23)
:: ./yis pushtest.yo
Stopped in 6 steps at PC = 0x12.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rdx:   0x0000000000000000      0x0000000000000100
%rsp:   0x0000000000000000      0x0000000000000100

Changes to memory:
0x00f8: 0x0000000000000000      0x0000000000000100
```

Problem 3(Folder: File3.2)

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

- andq - jle
- xorq - jne



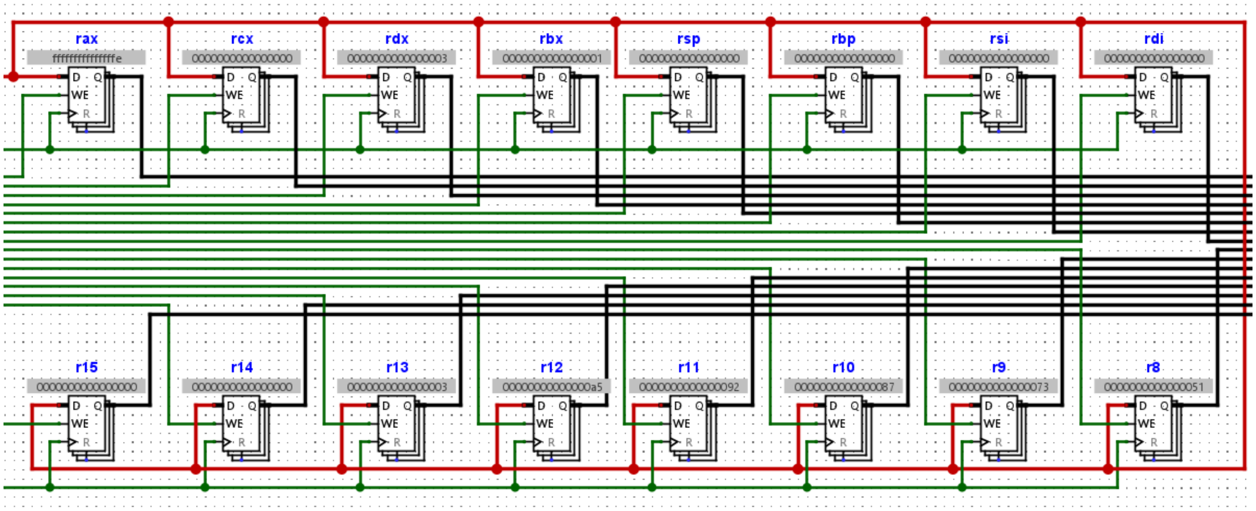
```
[ealvarez13]@linux2 ~/Lab6/YS/File3.2> (10:01:17 04/28/23)
:: ./yis Problem3.yo
Stopped in 12 steps at PC = 0x54.  Status 'HLT', CC Z=0 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x000000000000003f
%rcx:   0x0000000000000000      0x000000000000002a
%rdx:   0x0000000000000000      0x0000000000000033
%rdi:   0x0000000000000000      0x0000000000000054
%r11:   0x0000000000000000      0x0000000000000054

Changes to memory:
```

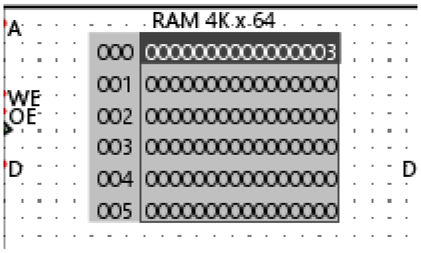
<u>1.2: Program run on processor:</u>

Problem 1 (Folder: File3)

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



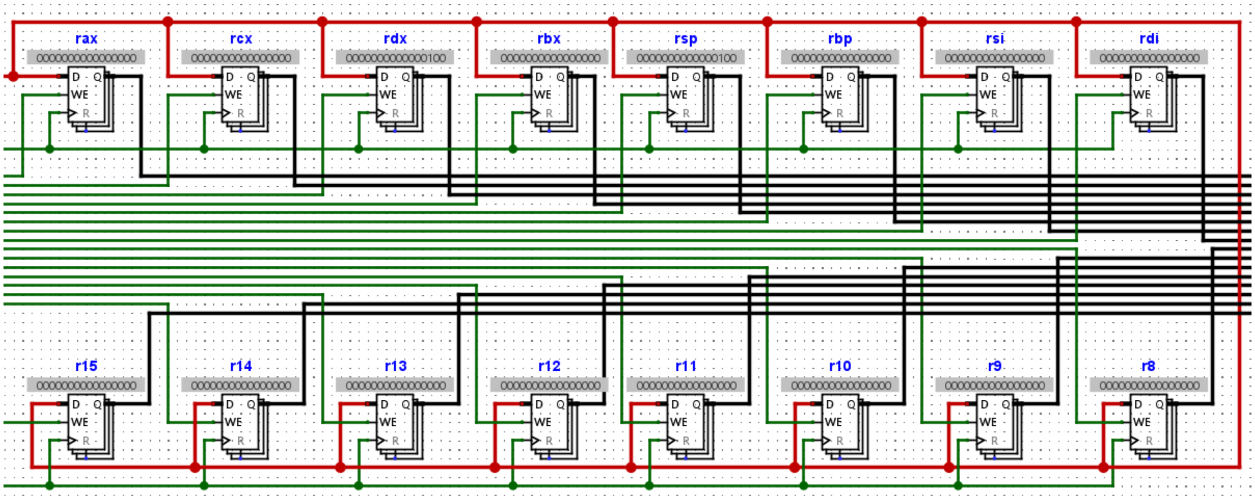
screenshot of ram inside of memory unit, value is as expected



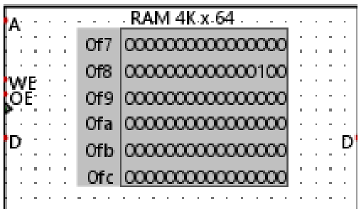
```
000 0000000000000003
001 0000000000000000
002 0000000000000000
003 0000000000000000
004 0000000000000000
005 0000000000000000
```

Problem 2 (Folder: File3.1)

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



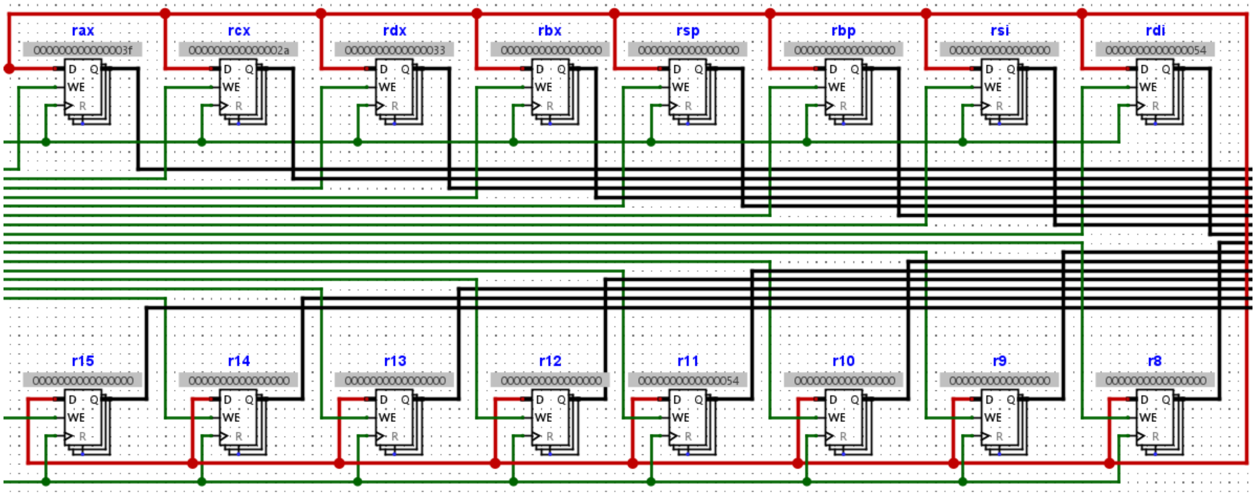
screenshot of ram inside of memory unit, value is as expected



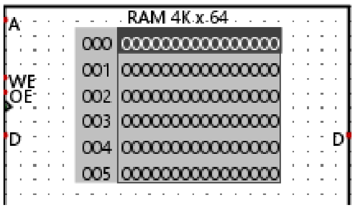
```
RAM 4K x 64
0f7 0000000000000000
0f8 0000000000000100
0f9 0000000000000000
0fa 0000000000000000
0fb 0000000000000000
0fc 0000000000000000
```

Problem 3(Folder: File3.2)

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



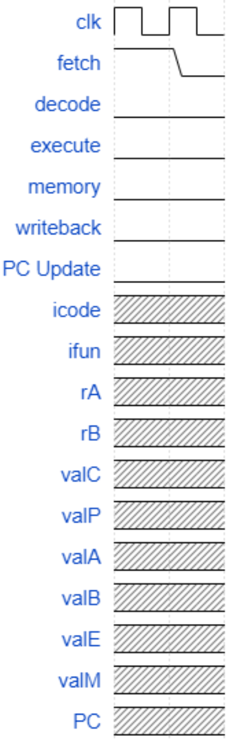
screenshot of ram inside of memory unit, value is as expected (no updates)



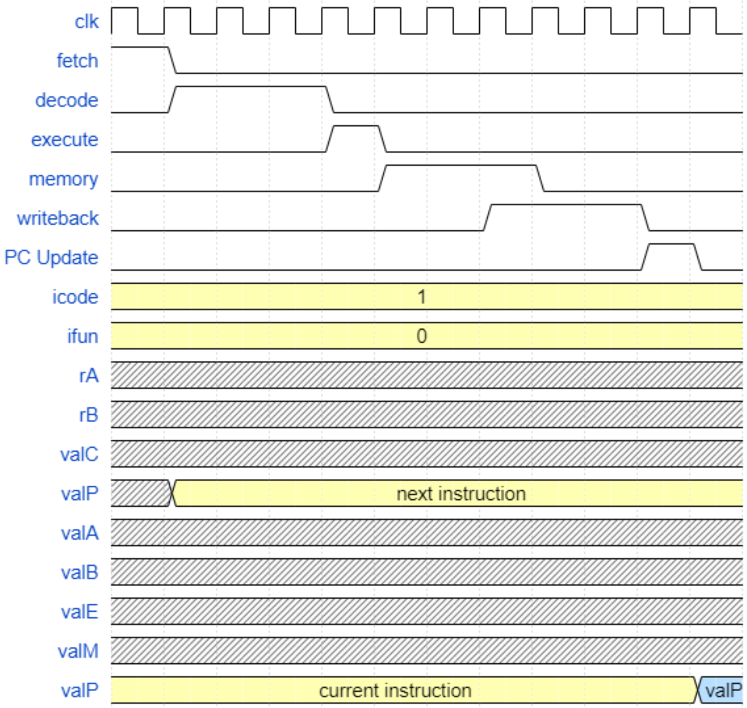
```
RAM 4K x 64
000 0000000000000000
001 0000000000000000
002 0000000000000000
003 0000000000000000
004 0000000000000000
005 0000000000000000
```

1.3 Timing of processor stages:

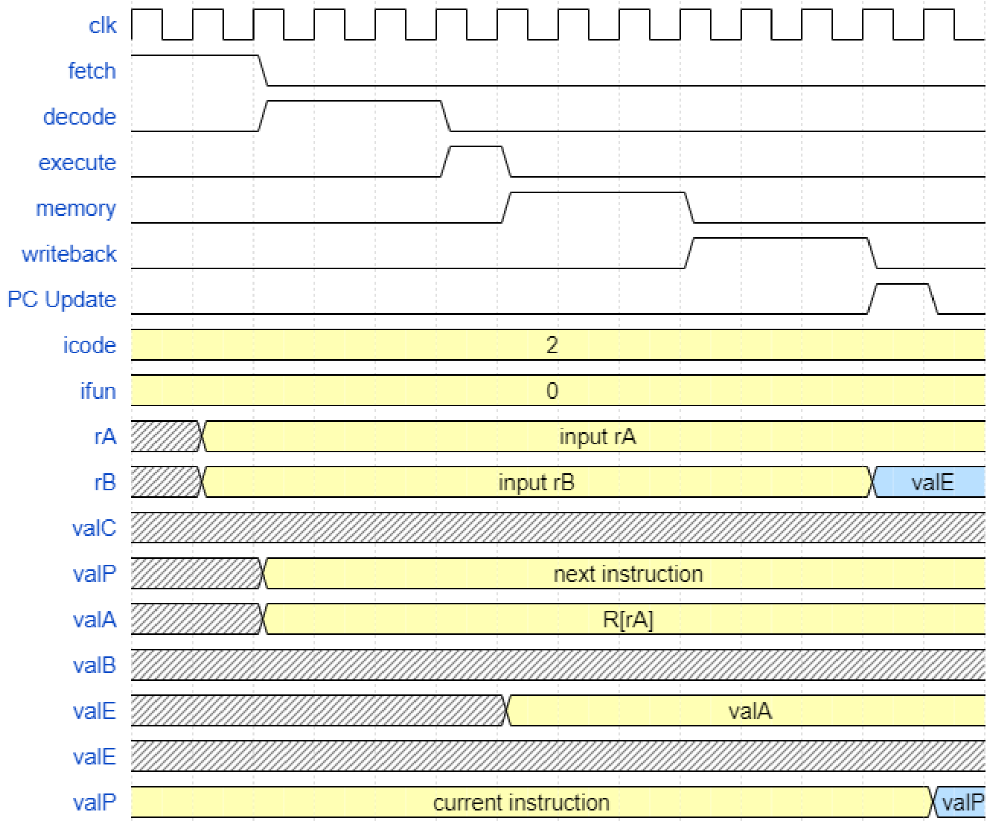
Halt time diagram:



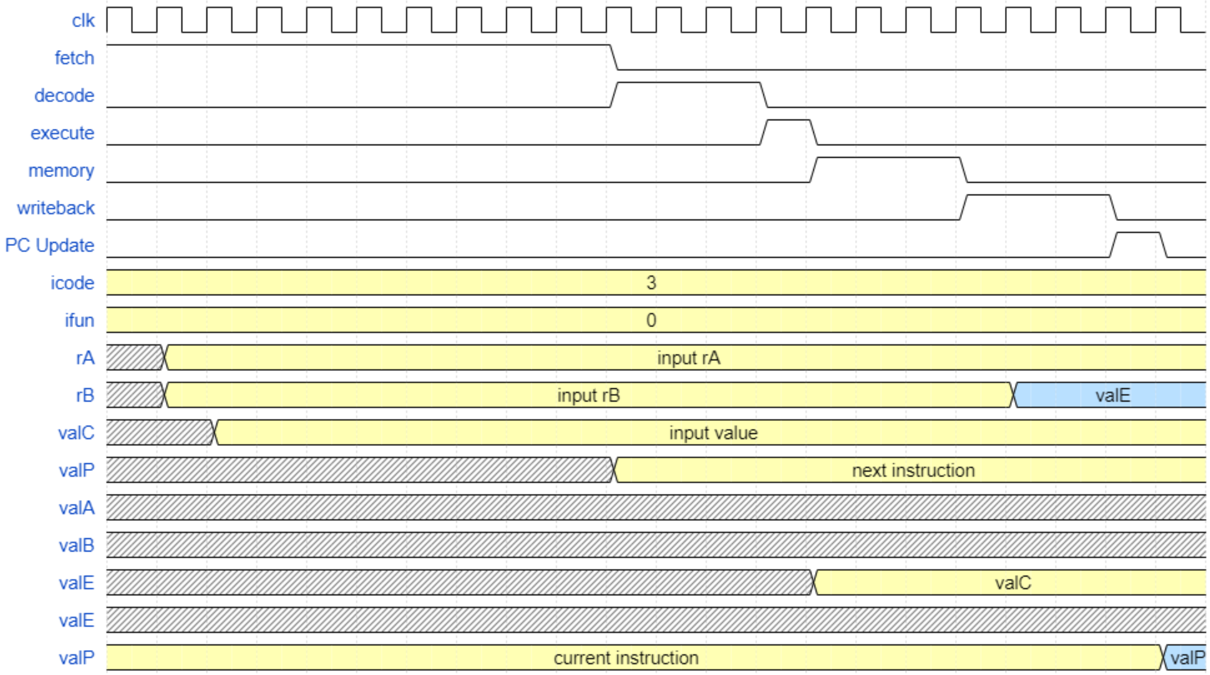
nop time diagram:



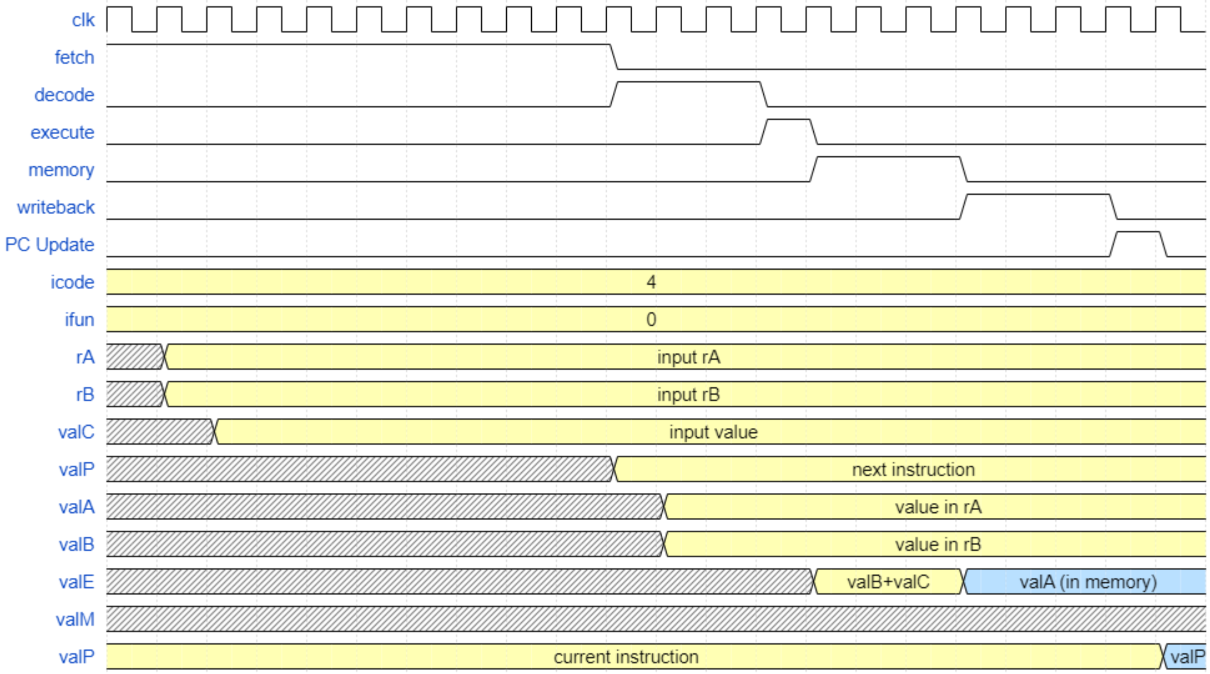
rrmovq time diagram:



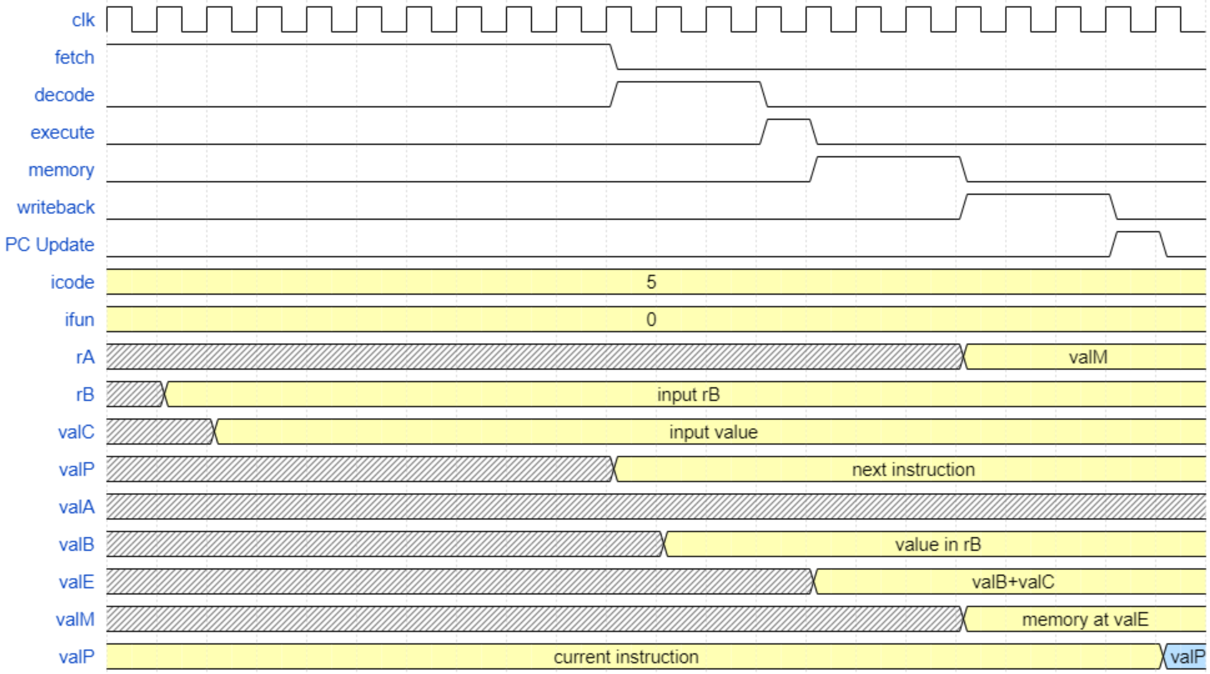
irmovq time diagram:



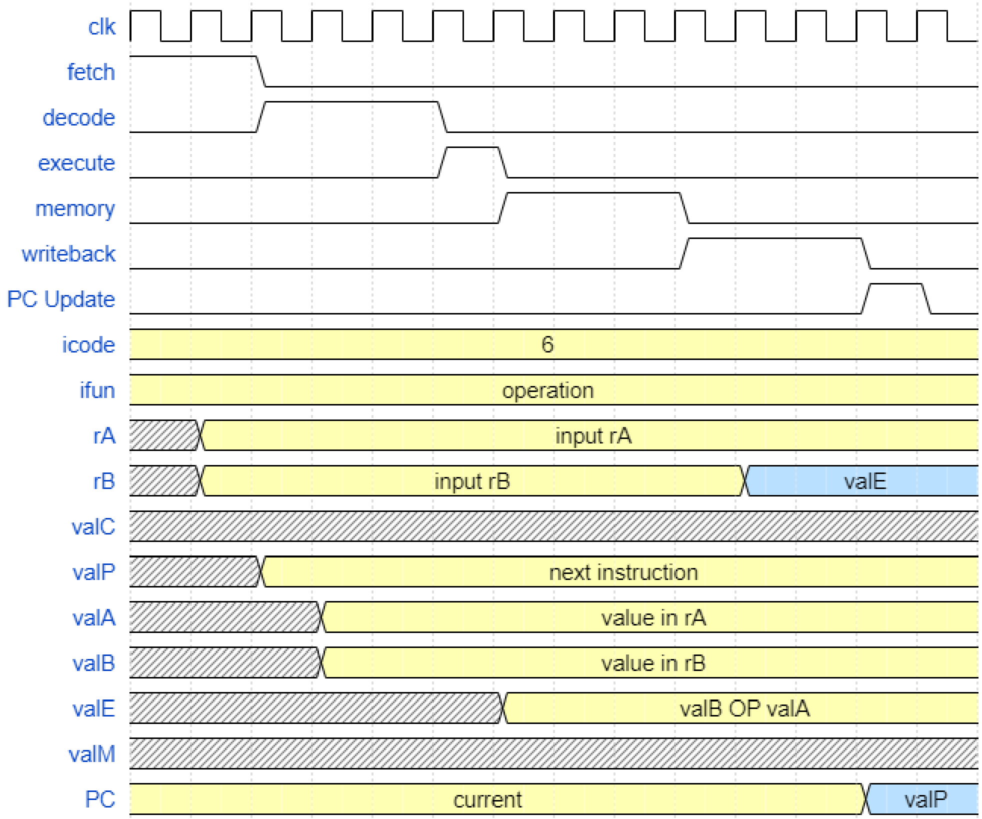
rmmovq time diagram:



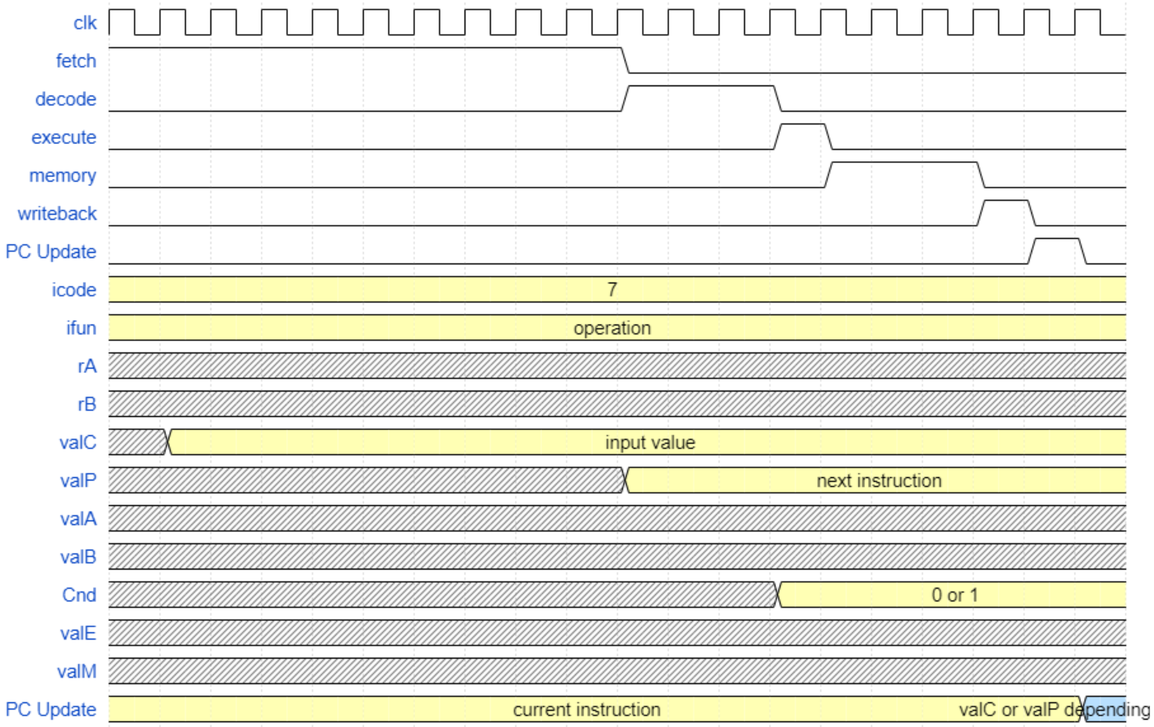
mrmovq time diagram:



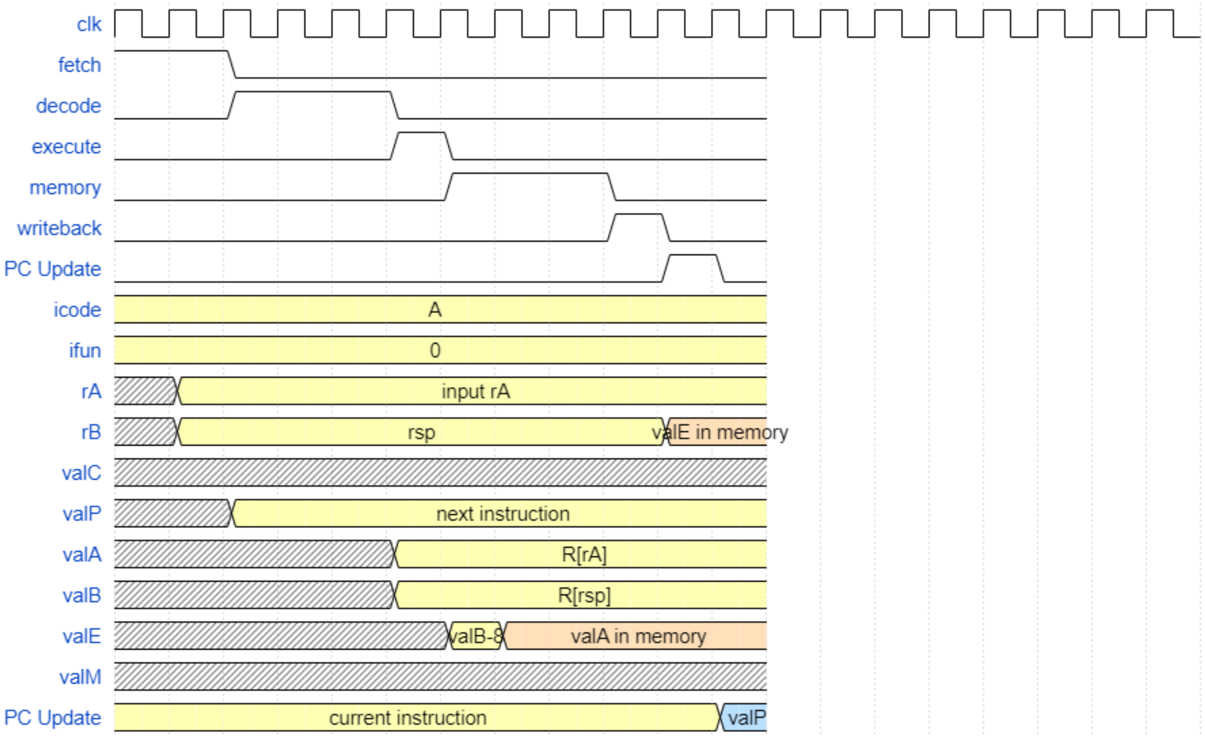
OPq time diagram:



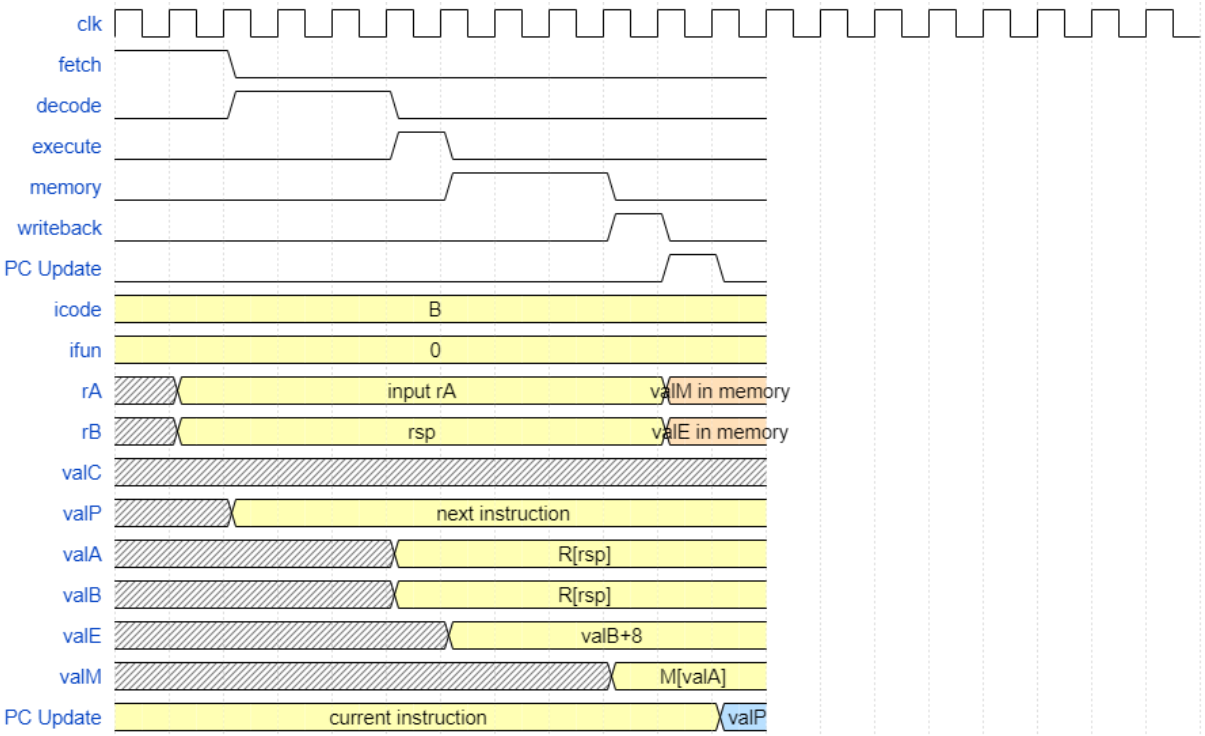
jmp time diagram:



pushq time diagram:



popq time diagram:

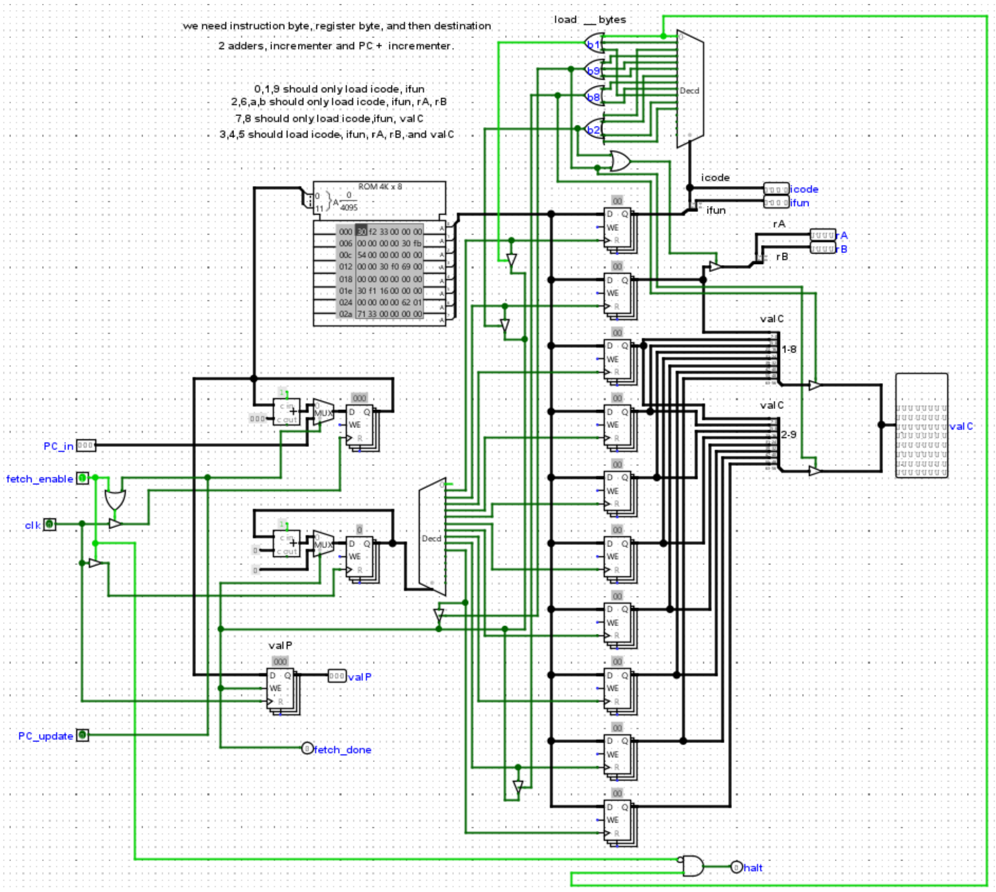


1.4 Design

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



we need instruction byte, register byte, and then destination

2 adders, incrementer and PC + incrementer.

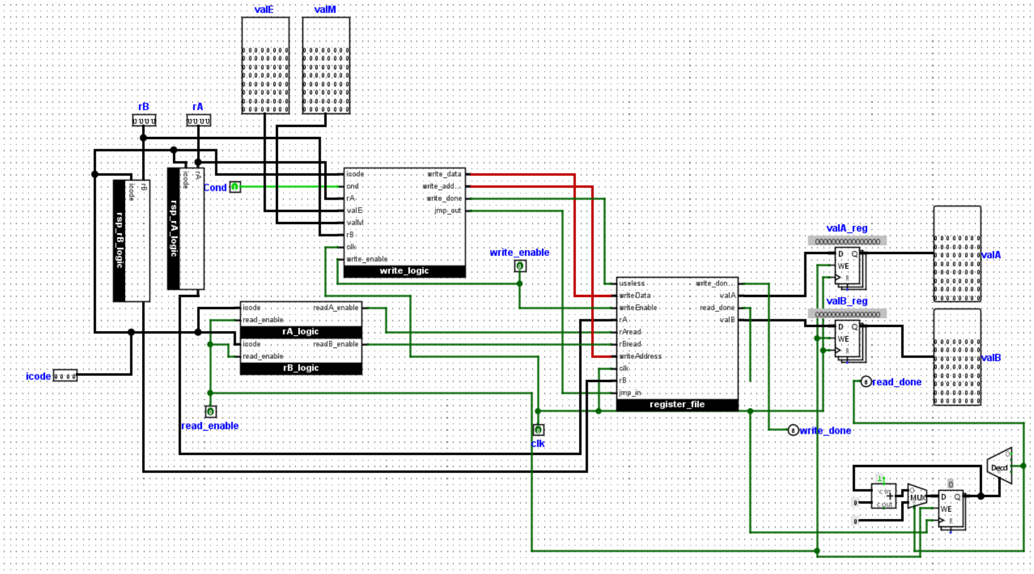
0,1,9 should only load icode, ifun
2,6,a,b should only load icode, ifun, rA, rB
7,8 should only load icode,ifun, valC
3,4,5 should load icode, ifun, rA, rB, and valC

```
000 30 f2 33 00 00 00
006 00 00 00 00 30 fb
00c 54 00 00 00 00 00
012 00 00 30 f0 69 00
018 00 00 00 00 00 00
01e 30 f1 16 00 00 00
024 00 00 00 00 62 01
02a 71 33 00 00 00 00
```

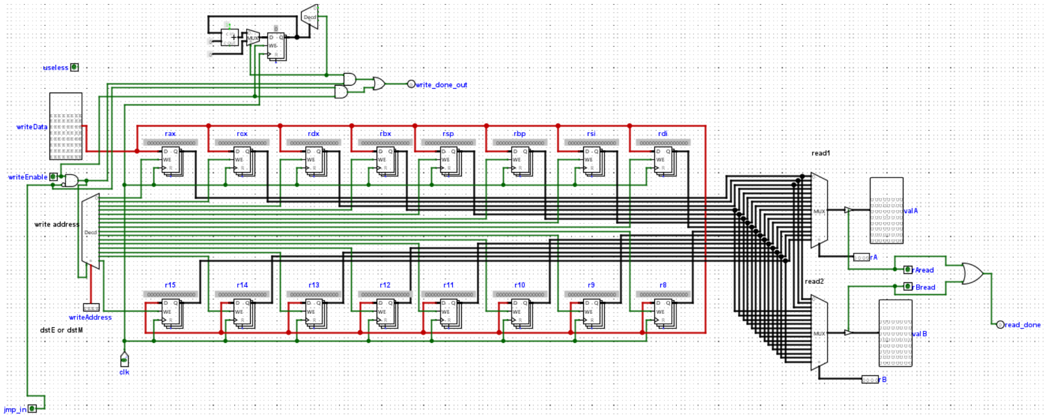
Decode:

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.

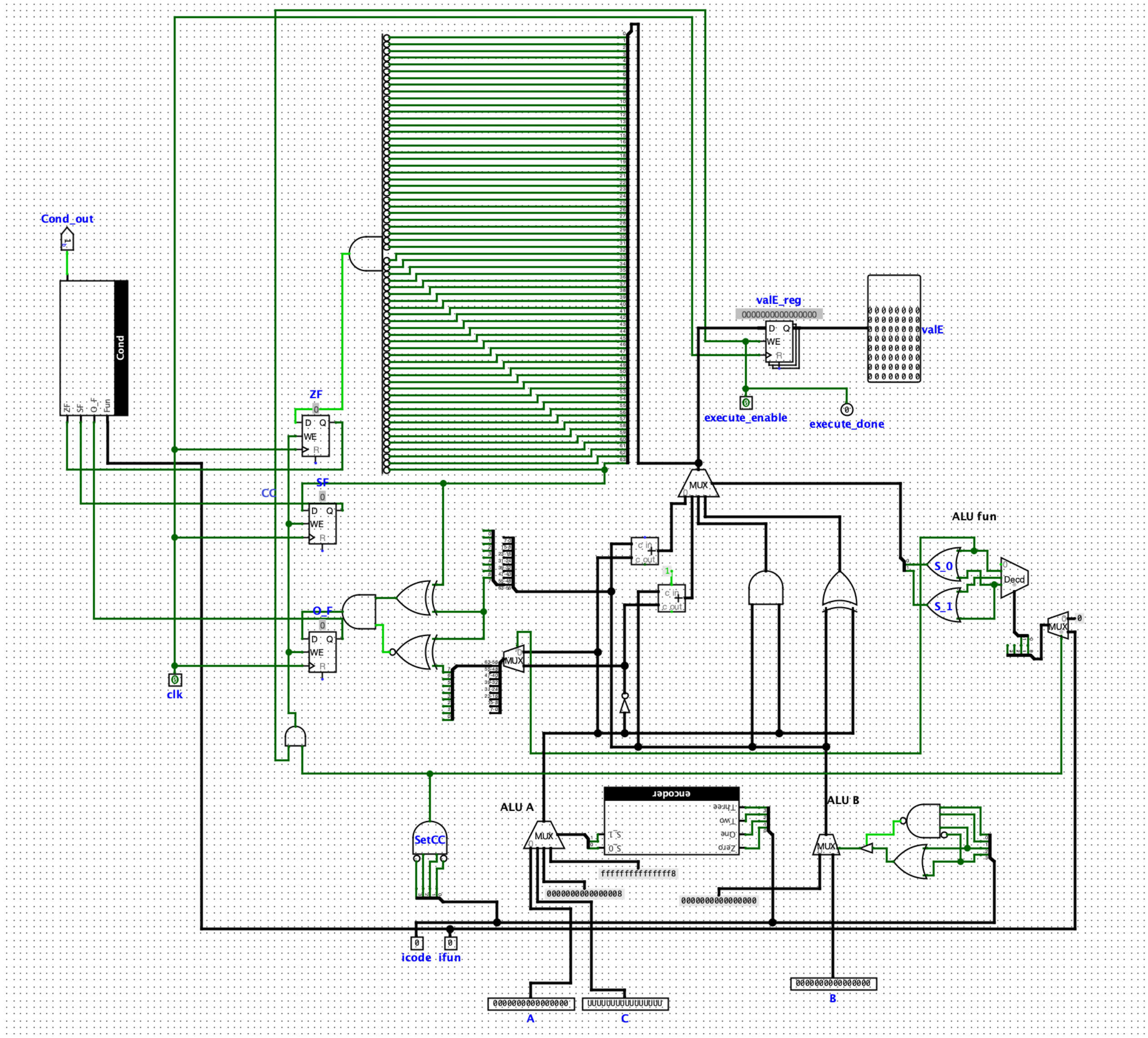
Inside the execute, ifun decides what kind of operation will be performed inside the ALU, for all operations other than OP(icode = 6) and Jmp (icode = 7) a sum will be performed between ValB and ValA. If the icode = 6 then ifun can be any number between 0-3, this will perform either a sum(0), subtraction(1),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

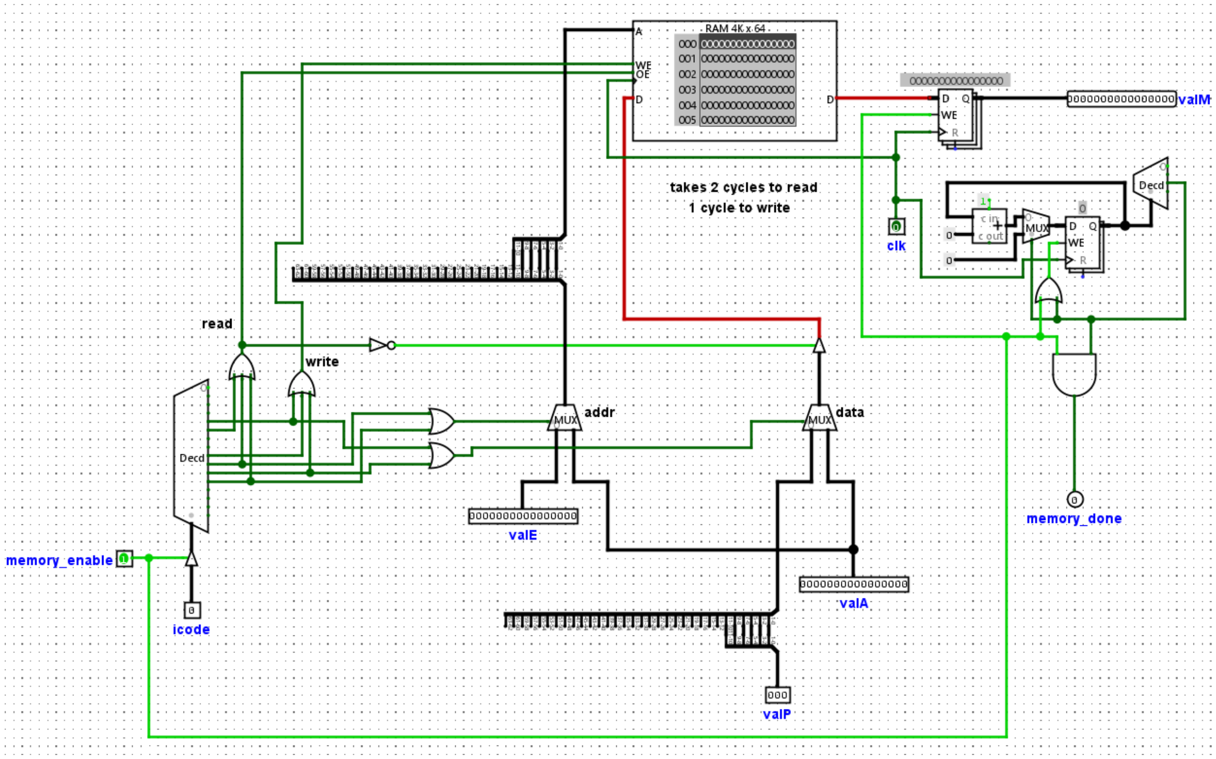
screenshot inside of Execute genie box



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



takes 2 cycles to read
1 cycle to write

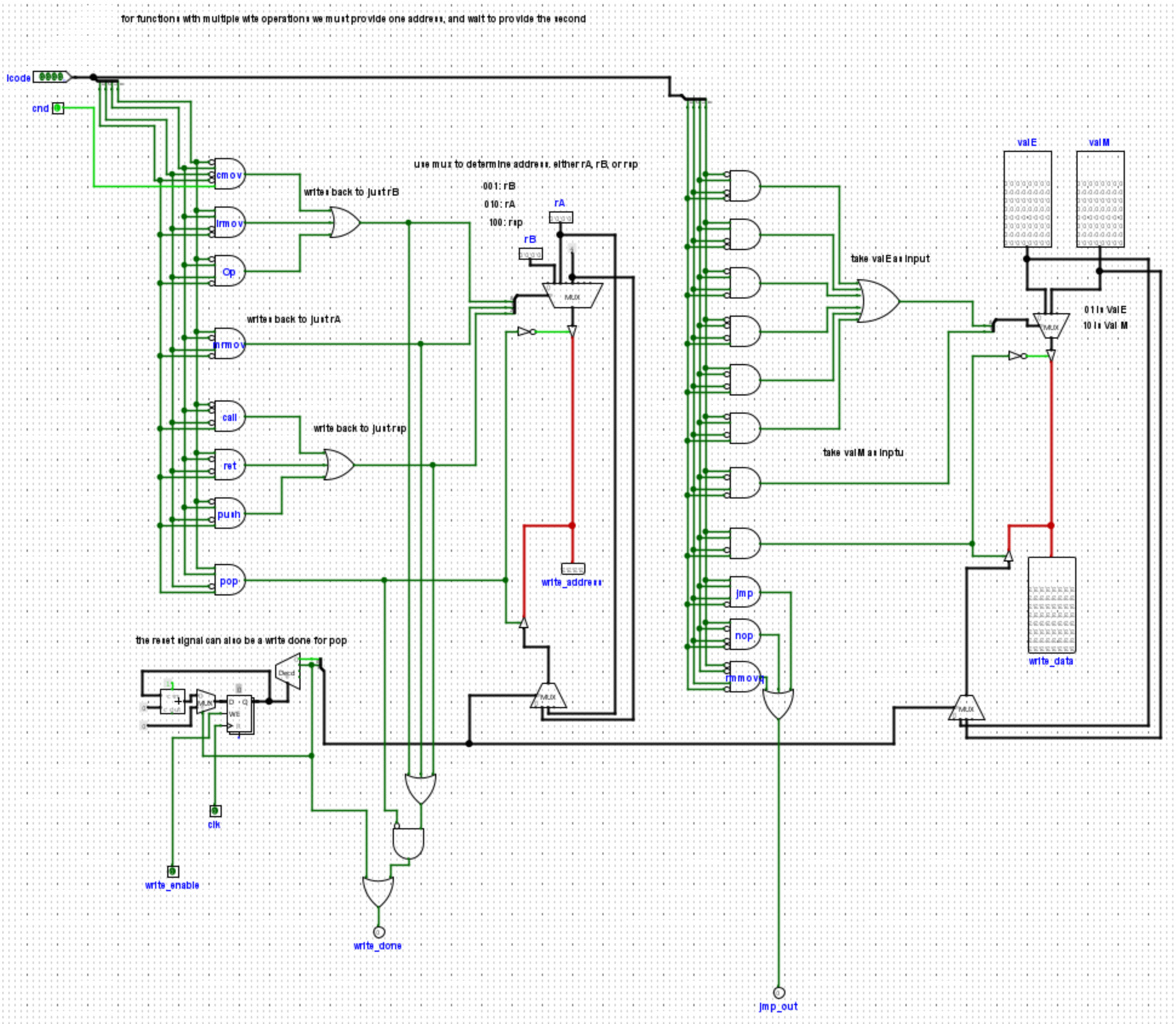
Write Back:

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



for functions with multiple wite operations we must provide one address, and wait to provide the second

use mux to determine address. either rA, rB, or rsp

001: rB
010: rA
100: rsp

write back to just rB

writes back to just rA

write back to just rsp

the reset signal can also be a write done for pop

take valE as input

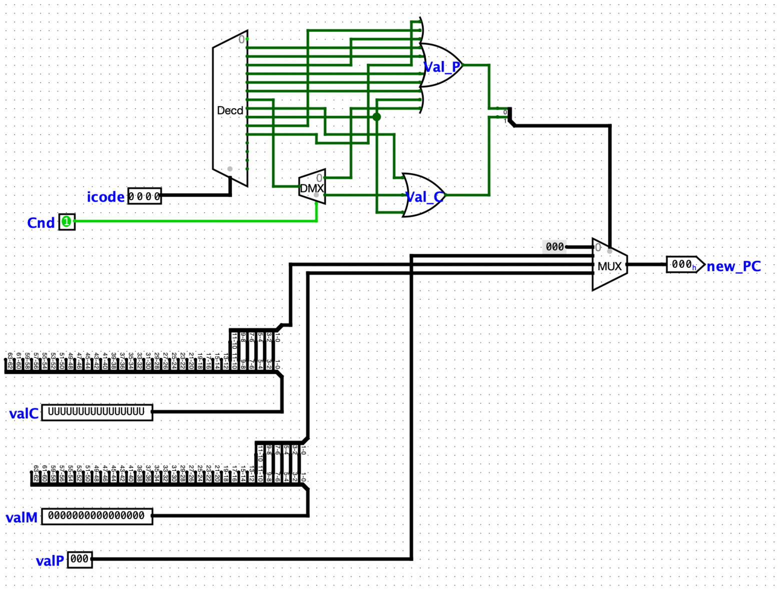
take valM as inptu

01 is ValE
10 is Val M

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.

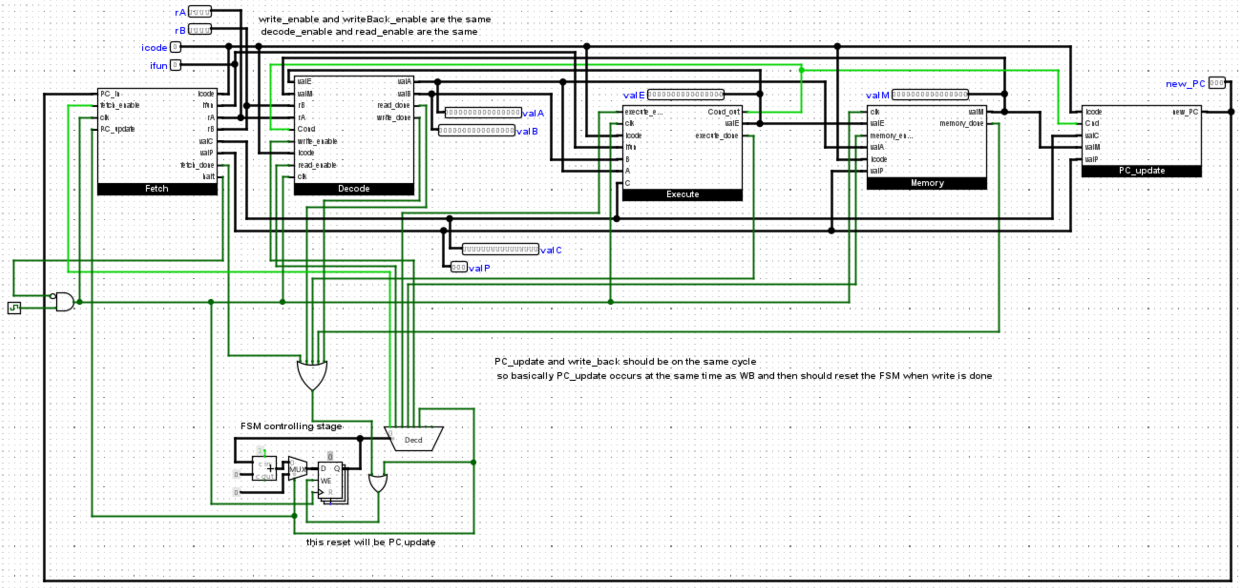
screenshot inside of PC_update genie box



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


rA
rB
icode
ifun
write_enable and writeBack_enable are the same
decode_enable and read_enable are the same
PC_in
fetch_enable
clk
PC_update
icode
ifun
rA
rB
valC
valP
fetch_done
halt
Fetch
valE
valM
rB
rA
Cond
write_enable
icode
read_enable
clk
valA
valB
read_done
write_done
Decode
valA
valB
valE
execute_e...
clk
icode
ifun
B
A
C
Cond_out
valE
execute_done
Execute
valM
clk
valE
memory_en...
valA
icode
valP
valM
memory_done
Memory
new_PC
icode
Cnd
valC
valM
valP
new_PC
PC_update
valC
valP
PC_update and write_back should be on the same cycle
so basically PC_update occurs at the same time as WB and then should reset the FSM when write is done
FSM controlling stage
MUX
D
Q
WE
R
Decd
this reset will be PC update