Howdy 🤠

Logan Bibb - 504
Quinn Gleadell - 503
Si Holmes - 504
Lane Simmons - 504

LAB 6 Y86 Project

Problem 1

Explaining each component of the processor built with screenshots from Logisim.

Fetch

The fetch stage of the processor is shown below. Fetch works by taking in the PC value from the Next PC stage and referencing the program memory to determine what the next set of instructions should be. It computes iCode, iFun, rA, rB, valP, and valC. We have added two extra control signals to help the stages operate smoothly. The DONE signal indicates that all of the outputs from the fetch stage are valid and are ready for use by the decode stage; it is a timing mechanism. The CURRENT_PC value connects to the PC Update stage and will be explained in detail in the next section.

The Fetch stage consists of several main hardware blocks: Instruction Memory, Split, Align, Need RegIDs, Need ValC, and PC increment.

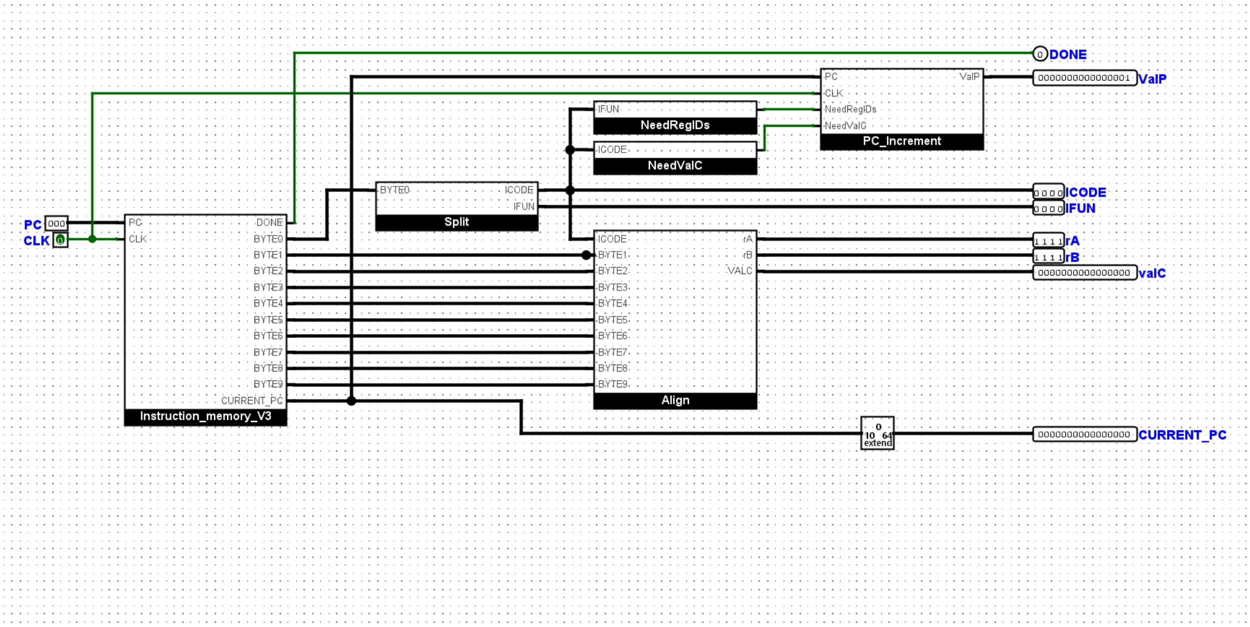


Figure: Fetch stage

Shown below is the Instruction Memory segment of Fetch. Instruction Memory outputs 10 bytes of data from the program memory regardless of what the iCode is. It uses a ROM unit to store the program instructions. The PC and a counter device are used to cycle through the 10 possible bytes of instruction and output them.

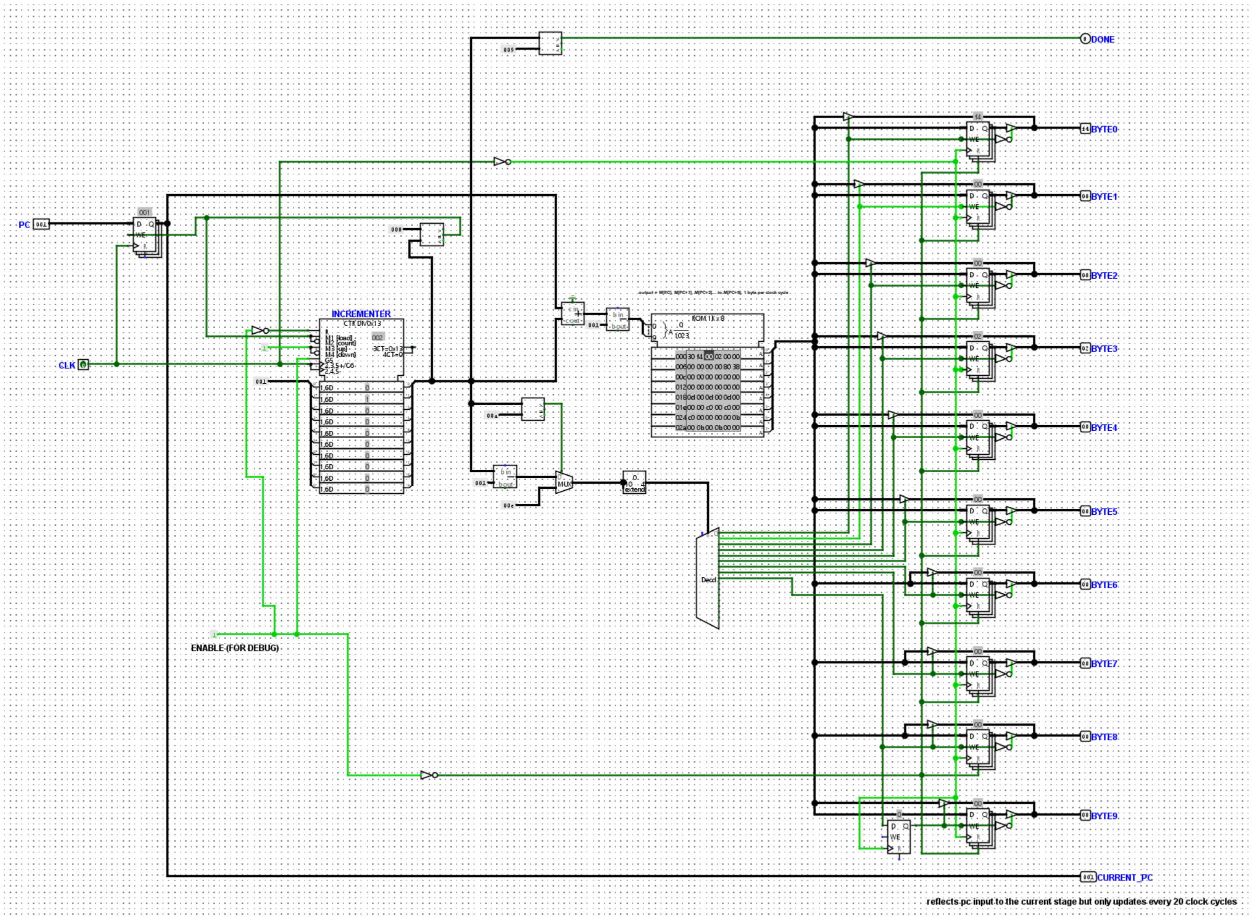


Figure: Instruction Memory part of Fetch

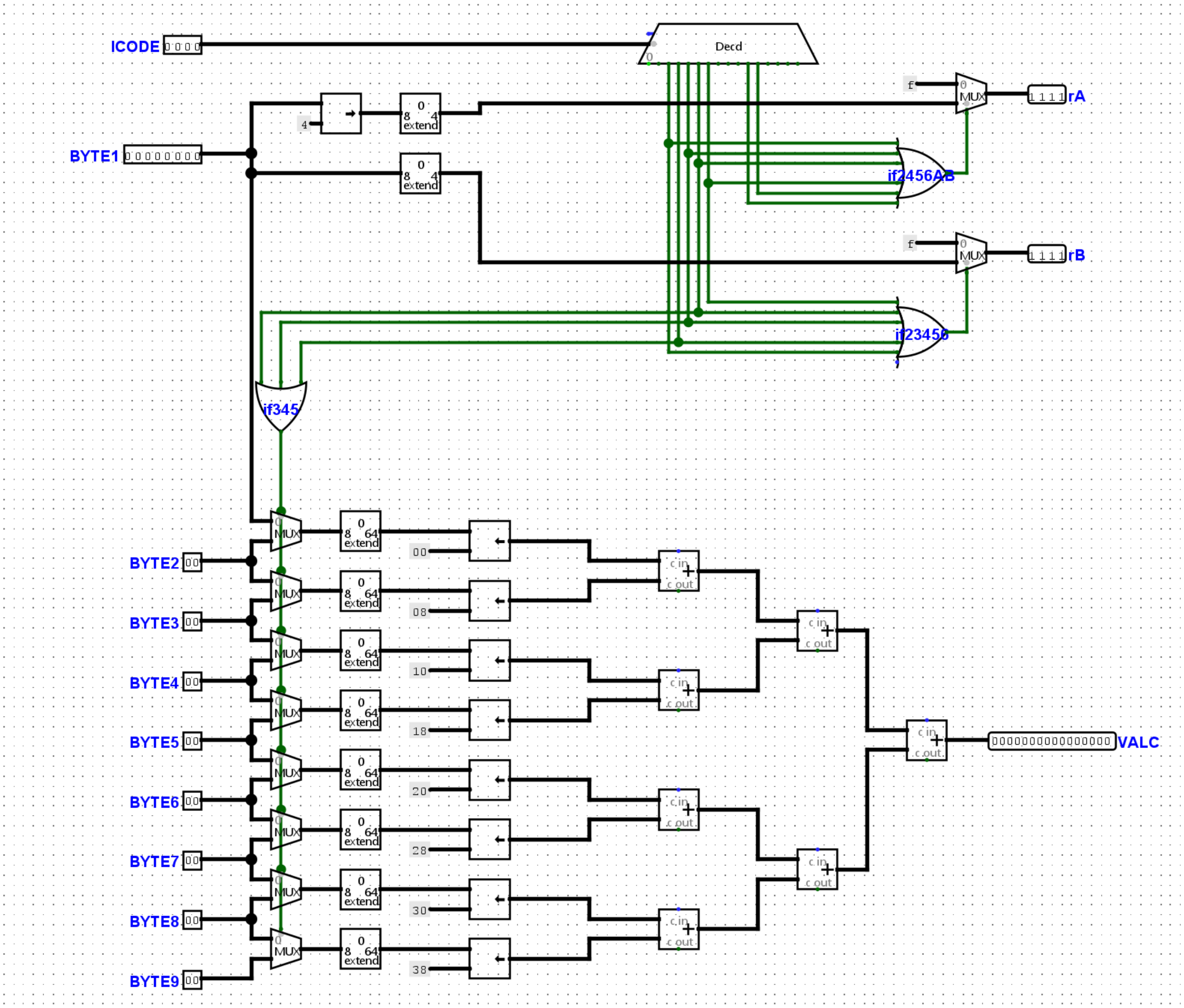
The PC is fed into a register with an enable signal that only goes high once every 20 clock cycles. This ensures that the PC output to the rest of the processor (called CURRENT_PC) is only updated every 20 clock cycles. Doing so prevents circuit oscillations and runaway PC values.

There are several adders and subtractors between the incrementer (or counter) and the ROM unit. These are solely used to align the value of PC and the incrementer with the desired address in the ROM unit, and to get the matching output from the decoder. To put it simply, at the beginning of the program, on the first clock tick high, the counter will go from zero to one. We want to access ROM[PC + 0] on this clock cycle, but the counter is currently one. To access the right data, we first add PC to the counter value and then subtract one, which gives ROM[PC + counter value - 1]. A similar computation is done for the decoder. A mux is also used for the decoder to avoid overflow and loopback causing undesired reads and outputs. Once the decoder select signal goes above nine (we only want to access outputs 0-9), we set the decoder to use an unused output e, regardless of the value of the counter.

Near the output, a set of circuits with control buffers and registers is used. This is done so that 10 bytes can be read and output in 10 clock cycles. When an output byte's decoder signal is high, we can pass the output of the ROM right through to that byte's output. Once the decoder value goes low, the value from ROM will have been stored in the register, so we can use the register output. This ensures that each byte's output is valid from the moment its byte is read from ROM until the PC resets after 20 clock cycles.

The Align block of the Fetch stage is shown below. It uses the iCode to determine which bytes (1-9 or 2-10) make up ValC. In the case that ValC is not required by the particular instruction, Align will fill ValC with arbitrary values. If iCode is 3, 4, or 5, the MUX select is one, and bytes 2-10 will be passed on to ValC. If not, the MUX select will be zero, and bytes 1-9 will be passed into ValC.

ValC is created using a series of multipliers and adders from bytes 1-9 or 2-10. Each byte is extended to 64 bits and then multiplied by a factor that essentially works as a bit shifter. Each byte is shifted into place according to the ”little endian” convention. The hardware also fills rA and rB, the register addresses for the register file. If either rA, rB, or both are not needed, they will be filled with 0xf.



Split (not shown) simply splits the first byte into iCode and iFun (4 bits each) using shifters and bit extenders.

Need ValC and Need RegIDs are shown below. They use iCode to determine whether ValC and the register IDs are needed in the instruction.

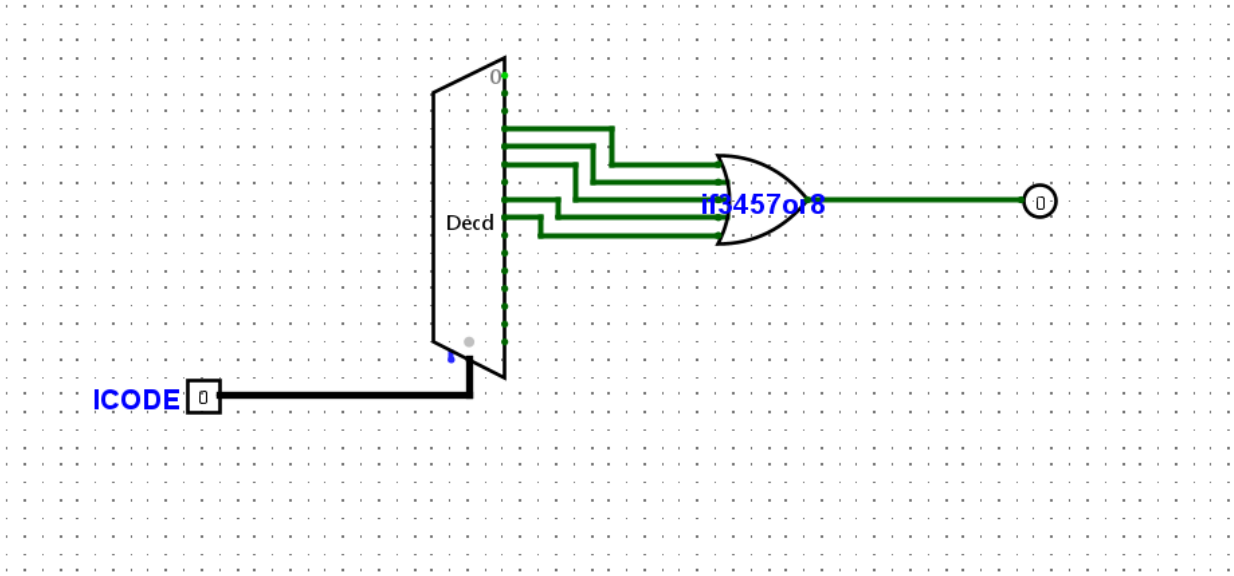


Figure: Need ValC

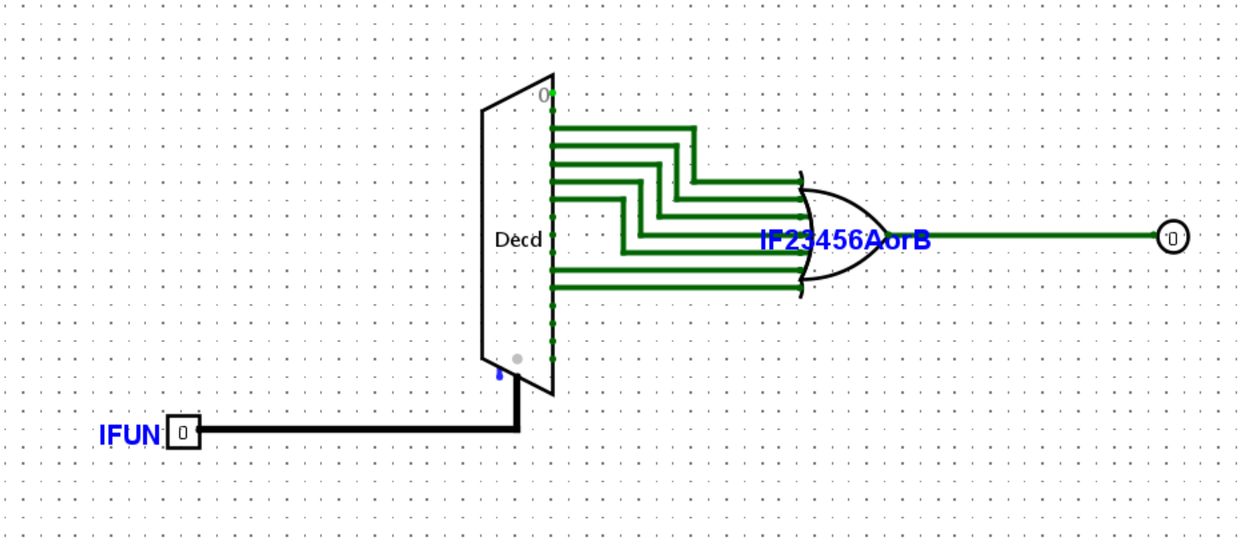


Figure: Need RegIDs

The PC Increment stage of Fetch is shown below. The equation for ValP is as follows:

$$ValP \ = \ PC \ + \ 1 \ + \ NeedRegIDs \ + \ 8 * NeedValC$$

This equation is realized directly in the hardware.

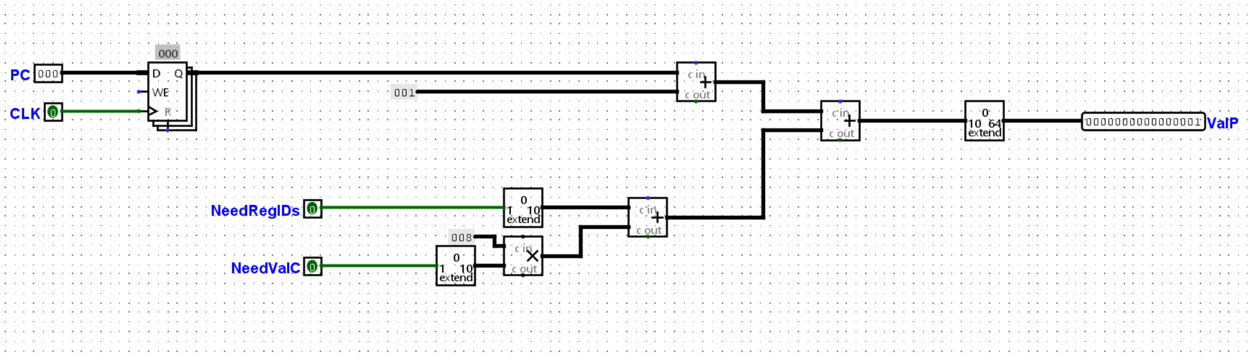
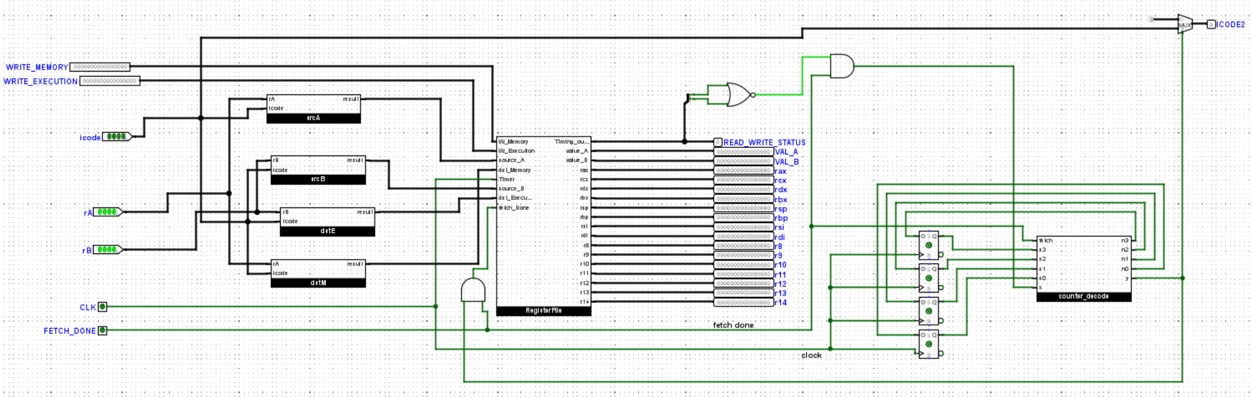
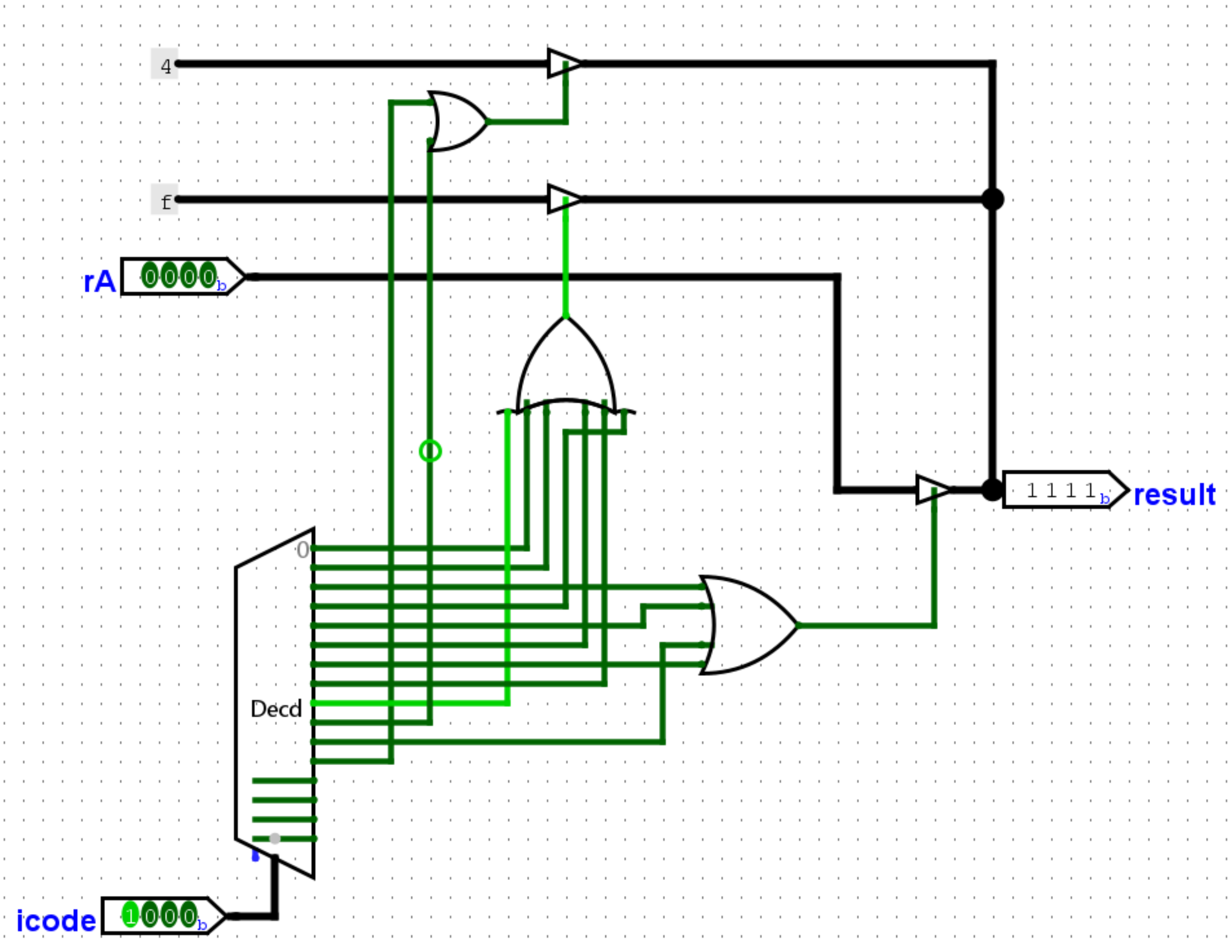


Figure: PC Increment

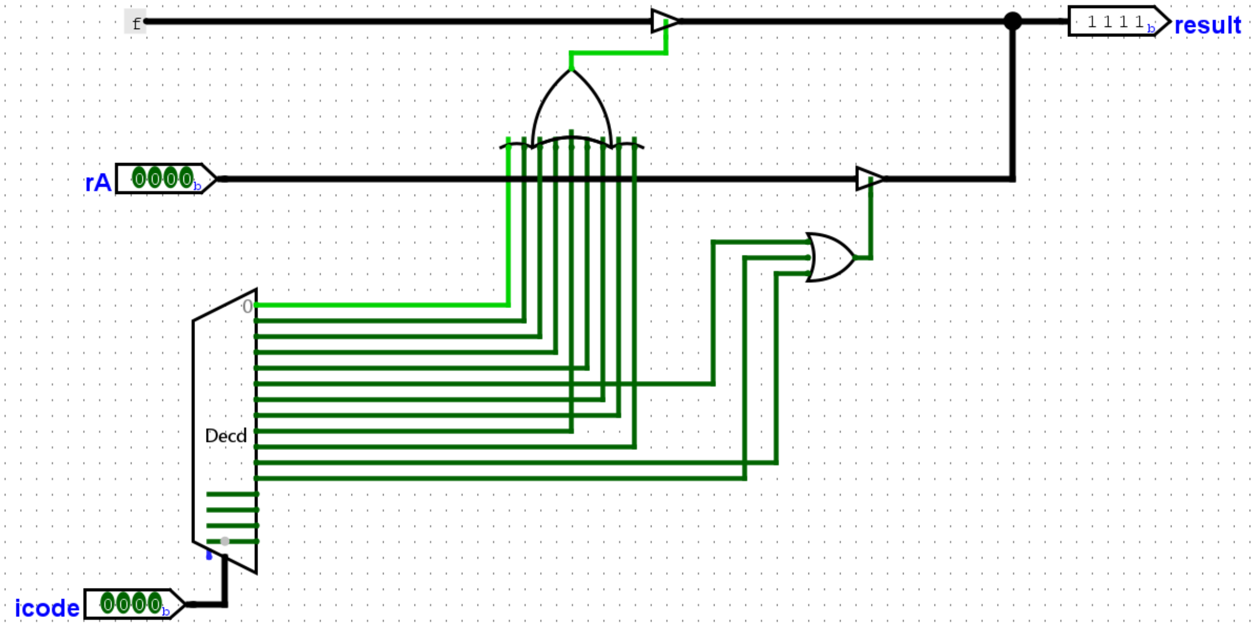
Decode



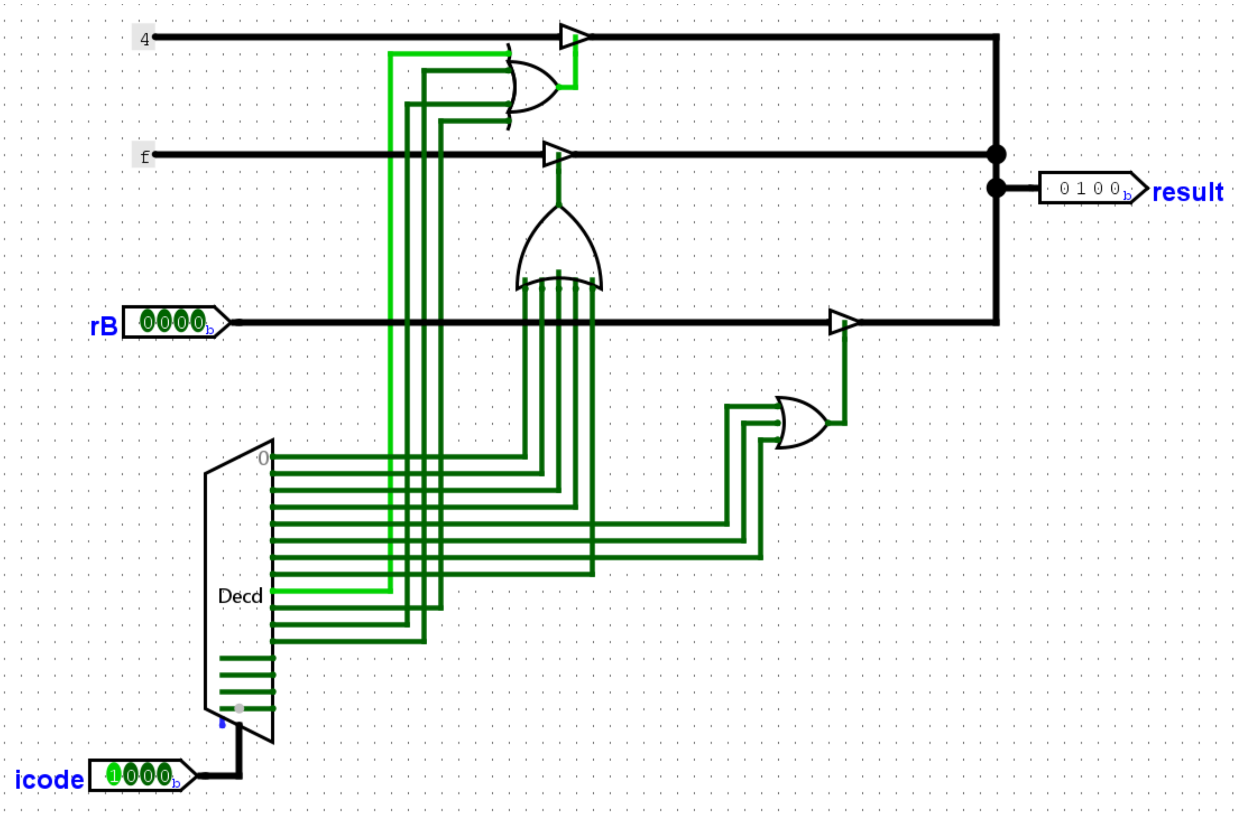
This is the Decode stage. It takes in 7 values, icode, rA, rB, Fetch_Done, valM, valE and CLK while outputting icode, valA, valB, the read_write_cycle (for debugging), and all the registers (for debugging). The first 4 values come from the fetch stage with valM coming from memory, valE coming from execution, and CLK coming from the universal clock. The fetch_done signal acts as an enable for the decode stage. Icode determines which values output from the four conditional blocks of code: srcA, dstM, srcB, and dstE. The outputs for srcA, dstM, srcB, and dstE all go into the RegisterFile which determines which registers to read and write from.



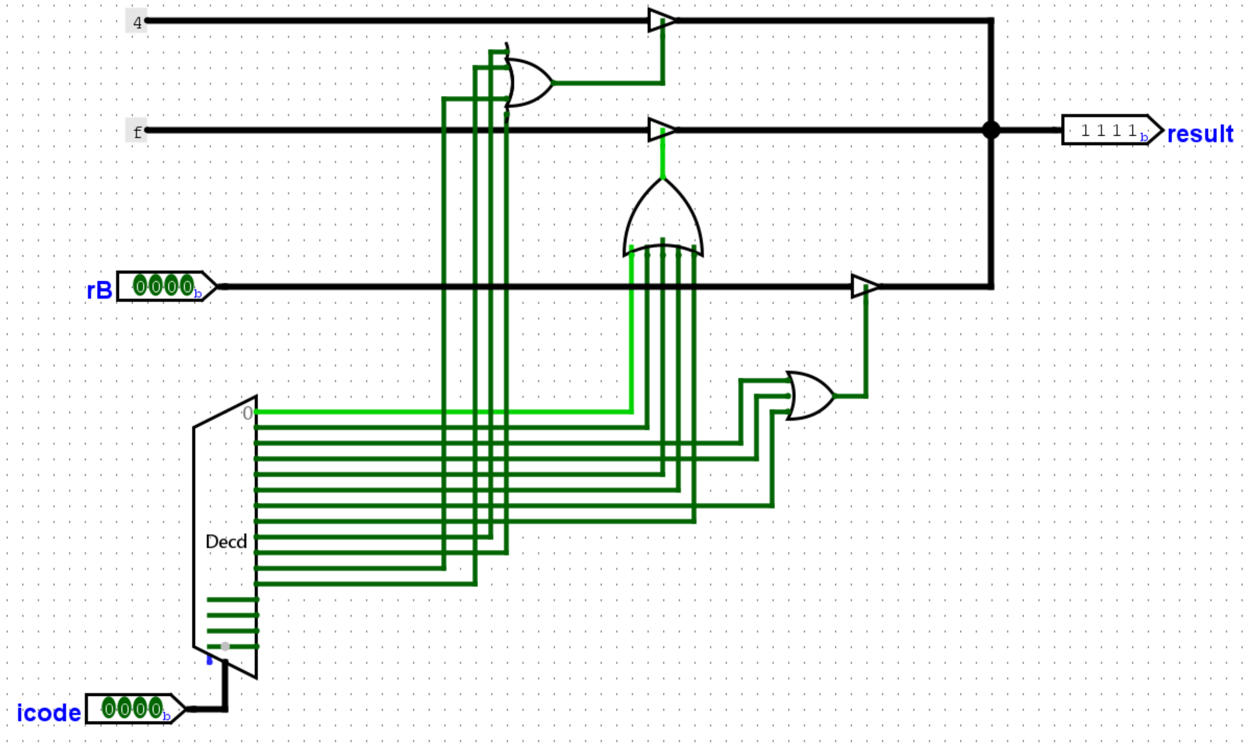
In srcA, the inputs are rA and icode with outputs either 0xf, 0x4, or rA. Icode goes through a decoder which selects the value of icode. This determines if the result or output of srcA is either 0x4, 0xf, or rA. This is very similar to dstM, srcB, and dstE.



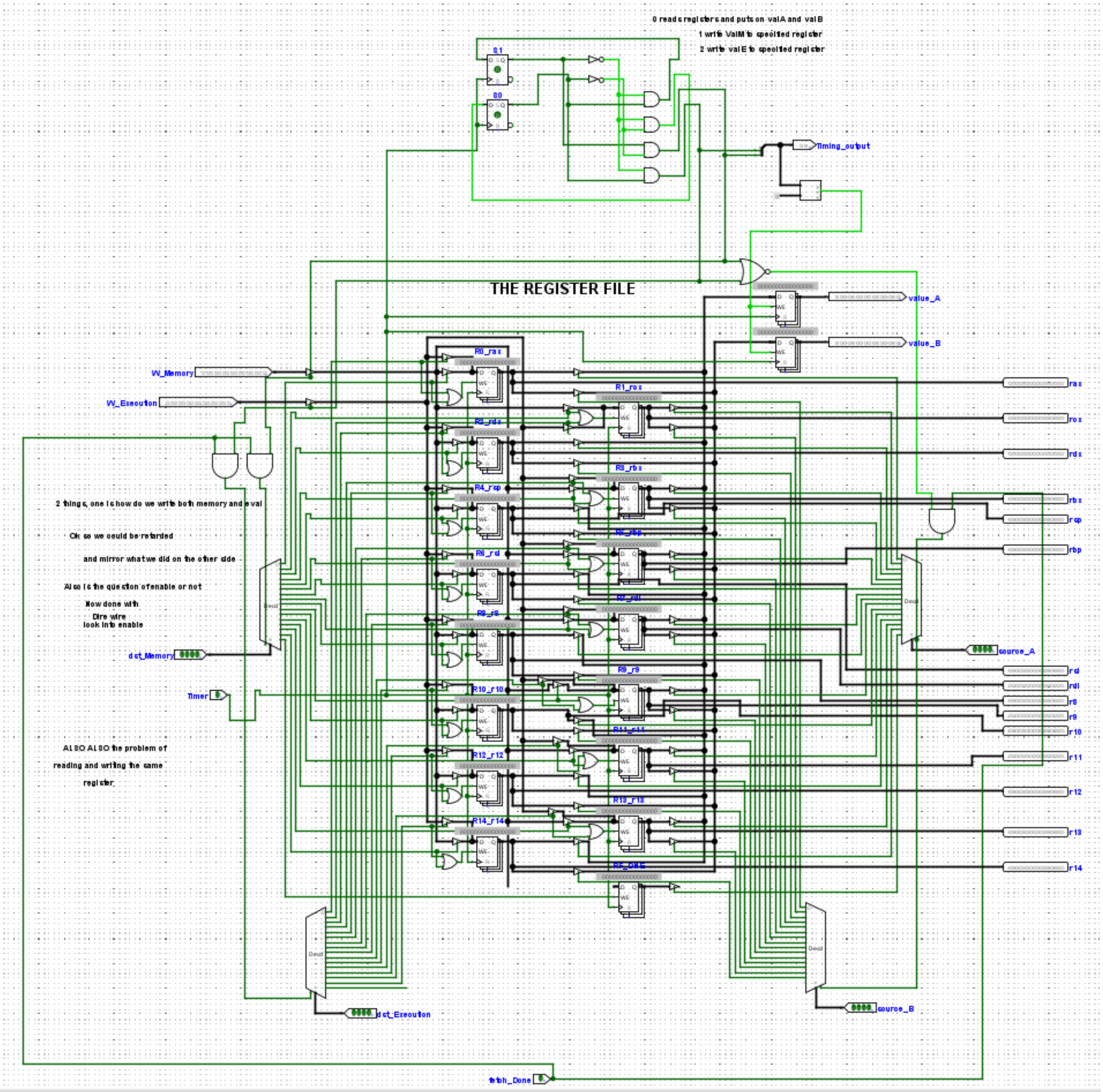
In dstM there are two inputs, rA and icode. Icode goes through a decoder which chooses to either send rA or 0xf into the output.



In srcB, the inputs are rB and icode. Depending on the value of icode, the output will be either 0x4, 0xf or rB.



In dstE, the inputs are rB and icode with output result. The icode goes into a decoder which chooses to make the output either 0x4, 0xf, or rB.



0 reads registers and puts on valA and valB

1 write ValM to specified register

2 write valE to specified register

2 things, one is how do we write both memory and e val

Ok so we could be retarded

and mirror what we did on the other side

Also is the question of enable or not

Now done with

Dire wire

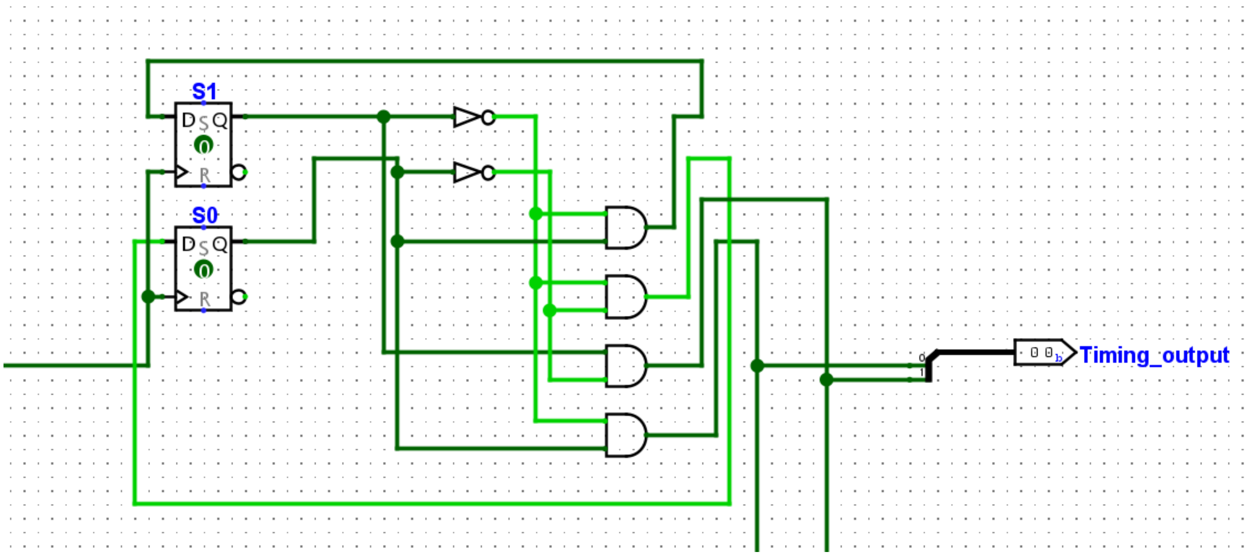
look into enable

ALSO ALSO the problem of

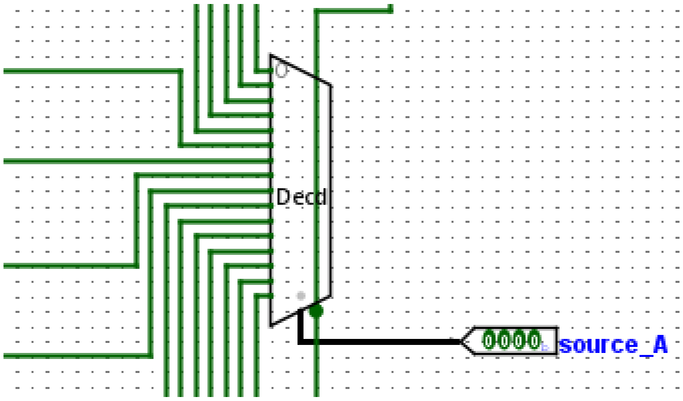
reading and writing the same

register.

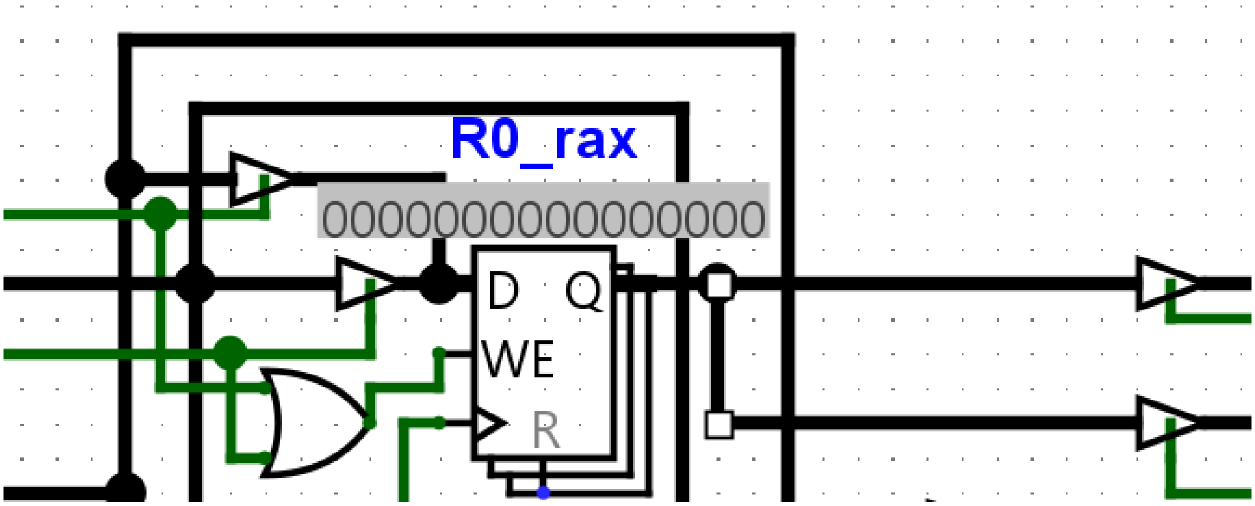
The register file takes in source_A, source_B, dst_Memory, dst_Execution, clock, valE, valM, and fetch_done signal outputting valA, valB, the current cycle, and the values of all the registers (for debugging). When the register file gets the inputs, it will start writing or reading depending on the internal register counter.



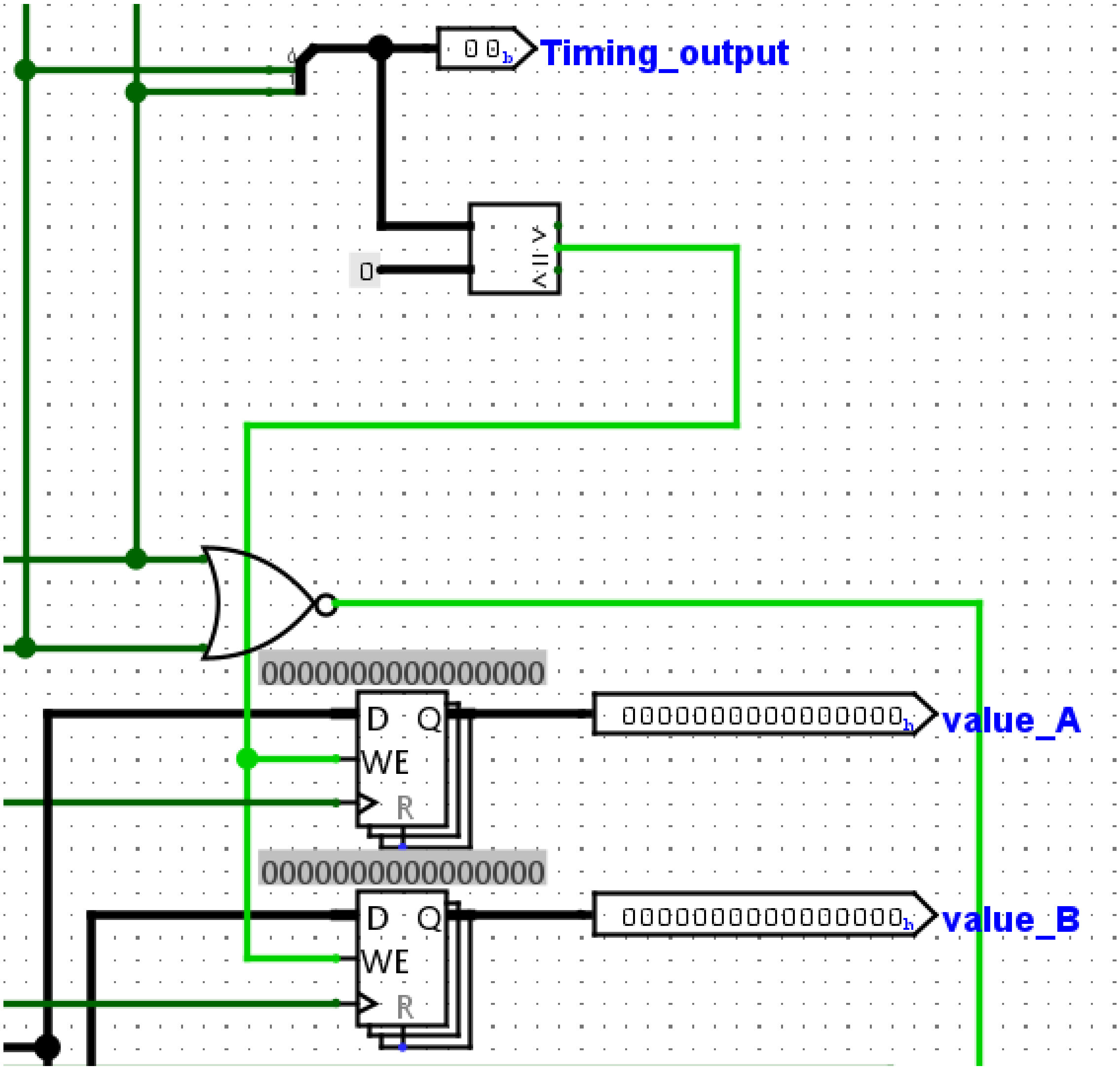
Inside the register file is a counter made of 2 d flip-flops which separates the simultaneous reads of valA and valB, the write of valE and the write of valM into 3 separate stages. When the counter is at 0, it will read two registers into valA and vaB. When the counter is at 1, it will write valE into the register file. When the counter is at 2, it will write valE into the register file. Then the cycle repeats. These writes and reads don't go through unless the fetch_done signal is high.



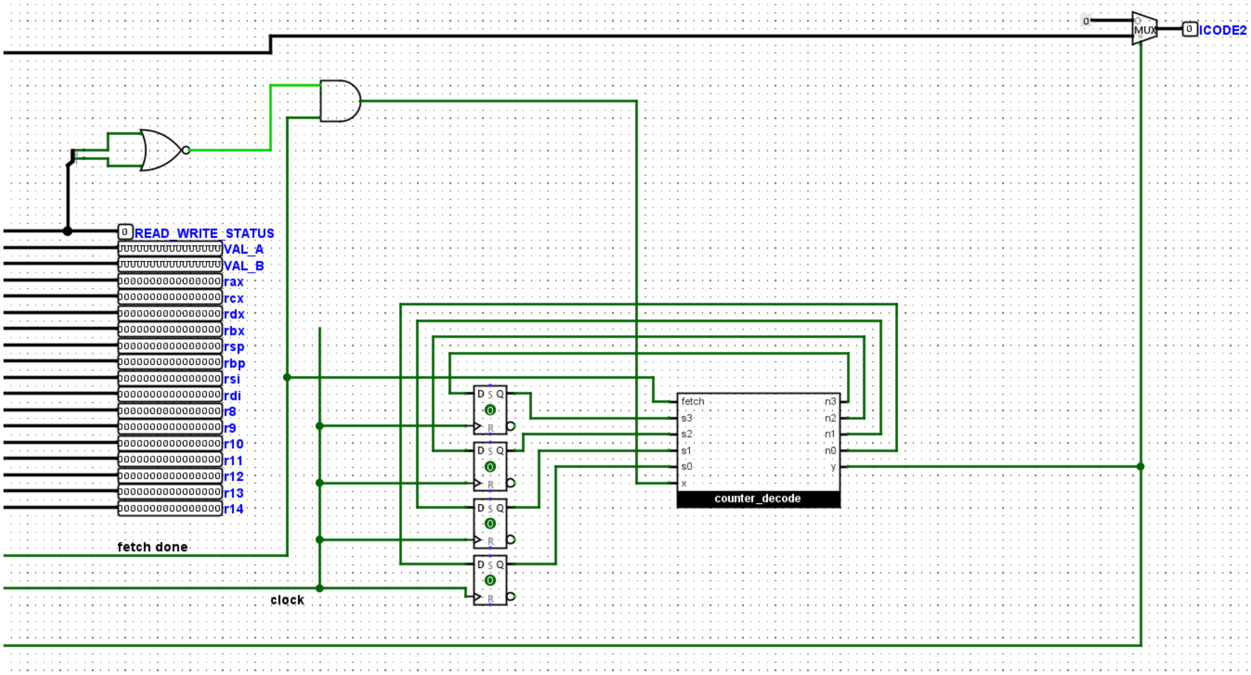
For each of srcA, srcB, dstE, and dstM input there is a decoder which outputs a high signal to the specified register from register 0 (%rax) to register 15 (which does not exist). The fetch_done and counter enable these decoders depending on their values.



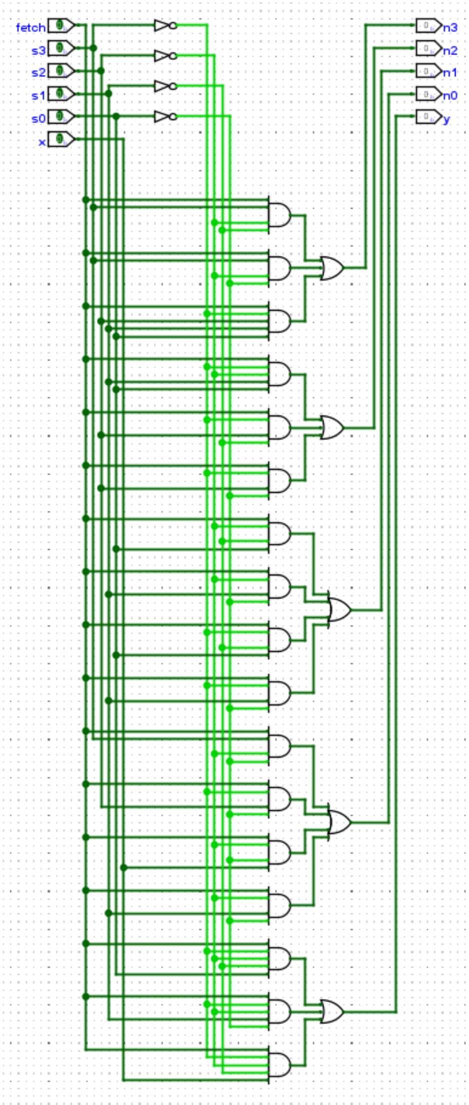
Each register has control buffers that are selected from the decoders so multiple writes do not happen and only one value is read per valA and valB.



The values for valA and valB are held in a separate register so that execute and memory stages can access them when the register file is not reading.



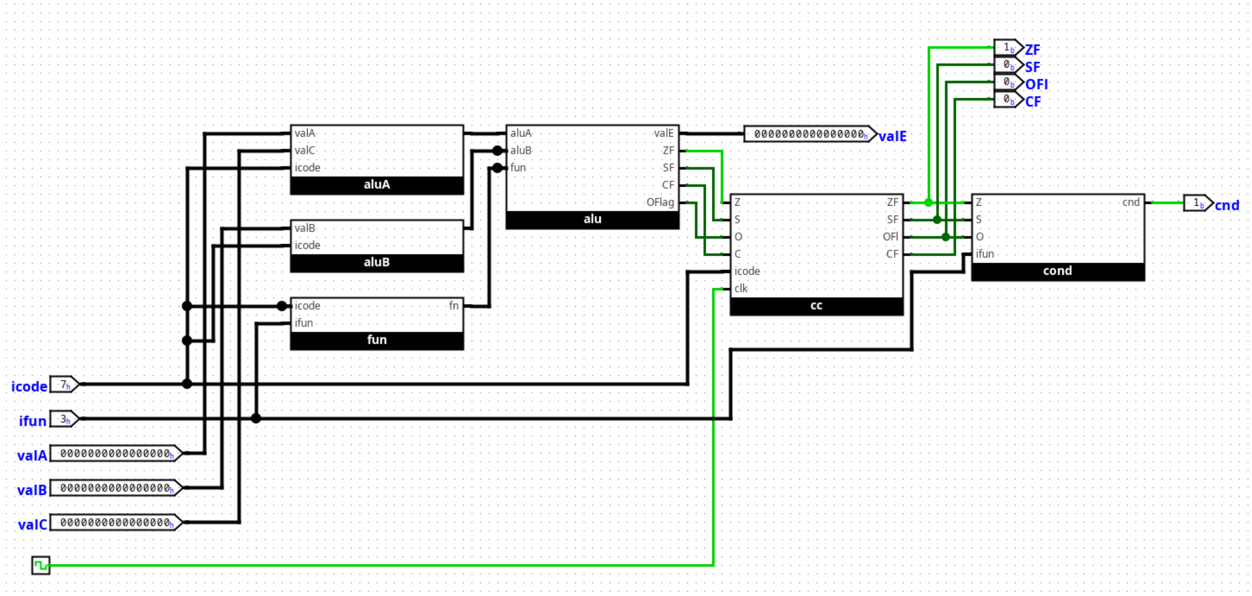
When the RegisterFile's internal clock is at 0 (it is reading valA and valB) and the fetch_done signal is high, the sequential logic block (counter_decode) will start up and output a high signal for three clock cycles into the mux which selects icode.



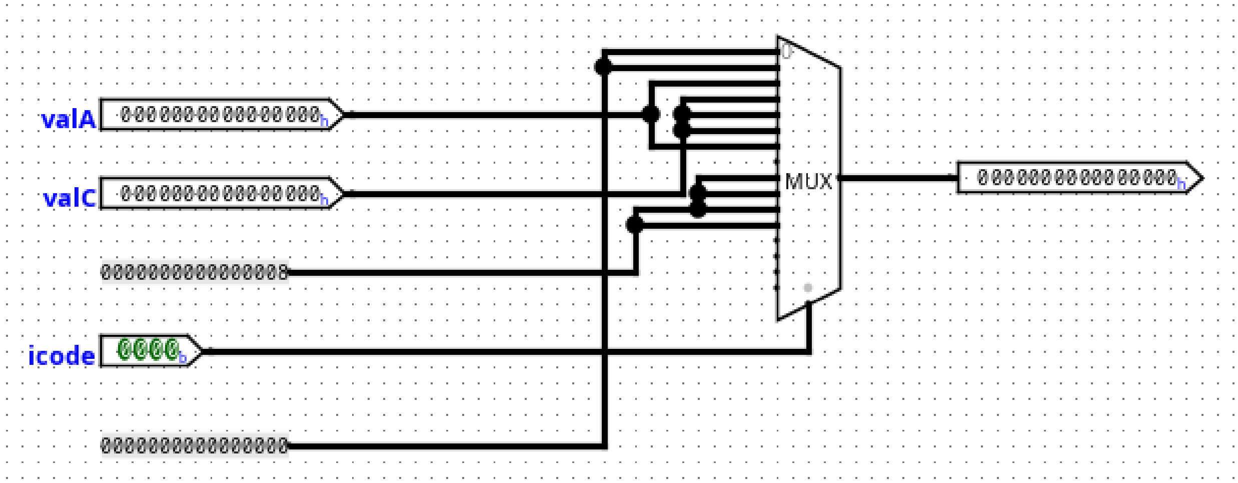
This design the execution stage and the memory stage to receive the correct icode signal instead of nop so everything can be synchronized. After three clock cycles, the memory and execution stages have completed their task and icode goes back to being 0 or nop so no values are changed.

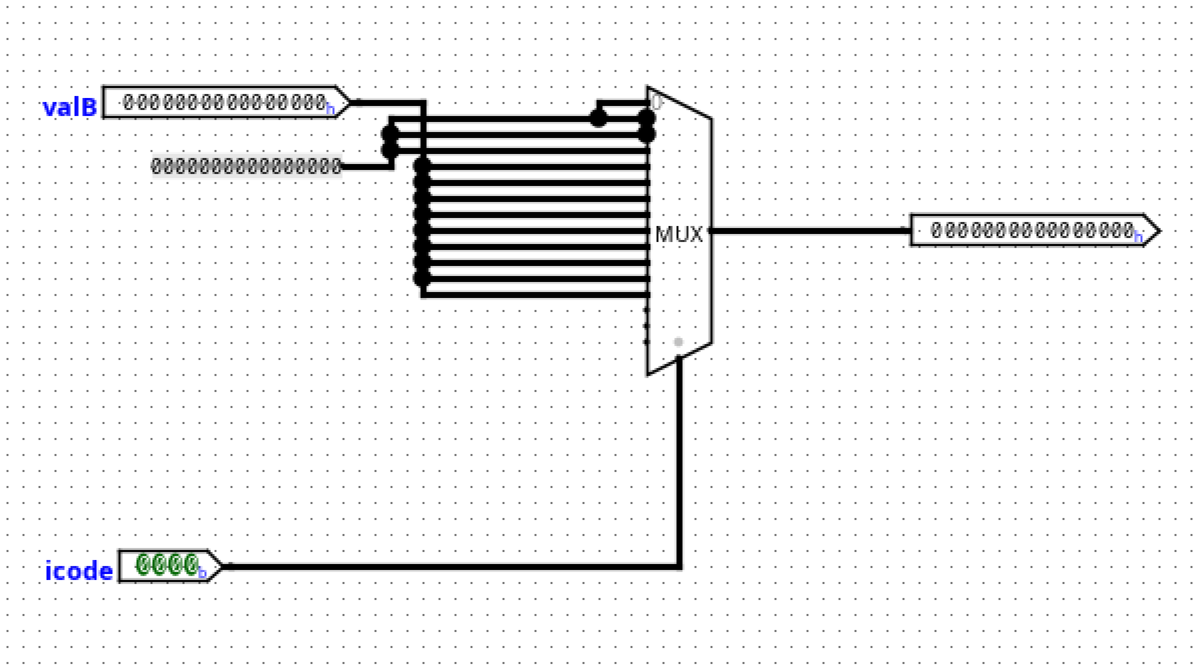
Execute

Execute takes in the values icode, ifun, valA, valB, valC, and clock. It outputs the zero flag (ZF), signed flag (SF), the overflow flag (OFl), carry flag (CF), valE (the arithmetic output), and the cnd (one bit output that tells whether or not a jmp is taken.

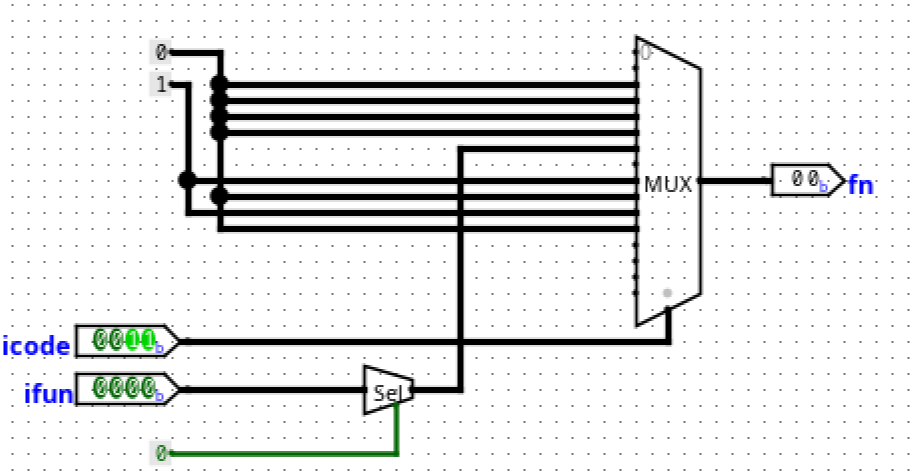


Depending on the icode, different values for aluA are selected. Operations such as pushq and popq will use 8 for aluA, but some operations such as opq will use valA for aluA. Similarly, aluB is selected based on the icode.

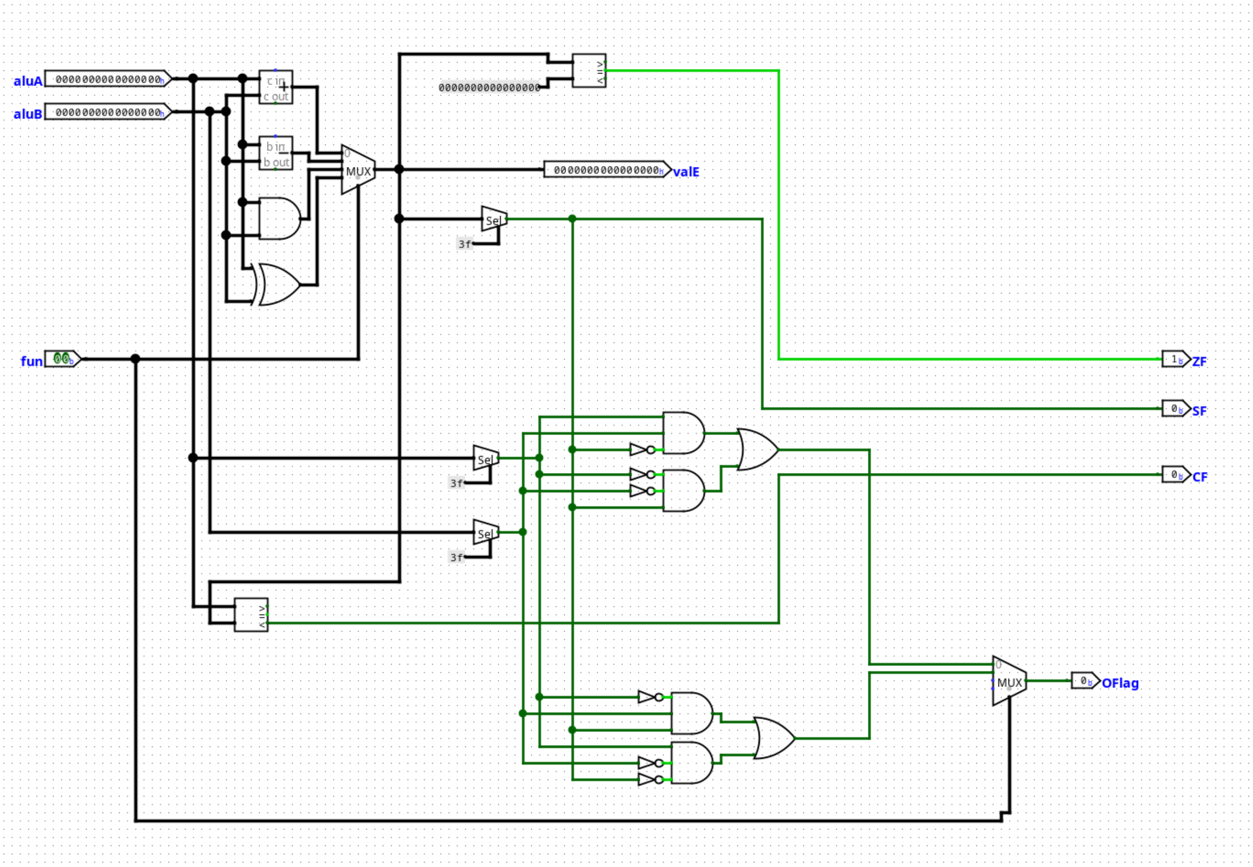




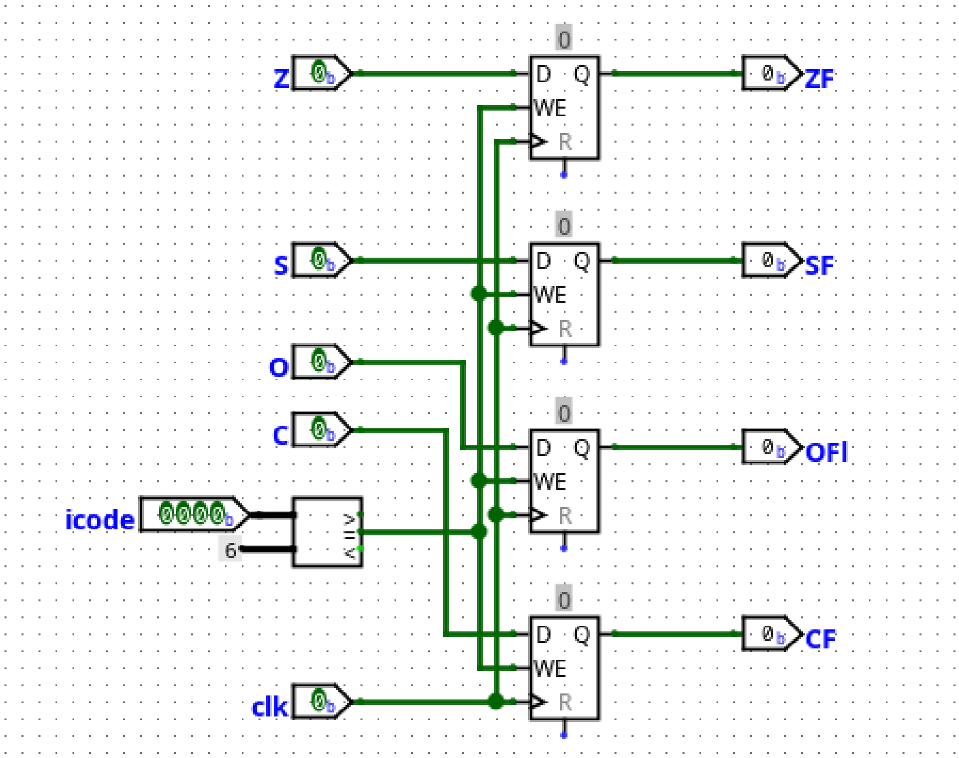
The ifun 0 in the ALU corresponds to addition, 1 corresponds to subtraction, 2 to AND, and 3 to XOR. This mux determines what operation the ALU needs to make by selecting from the icode.



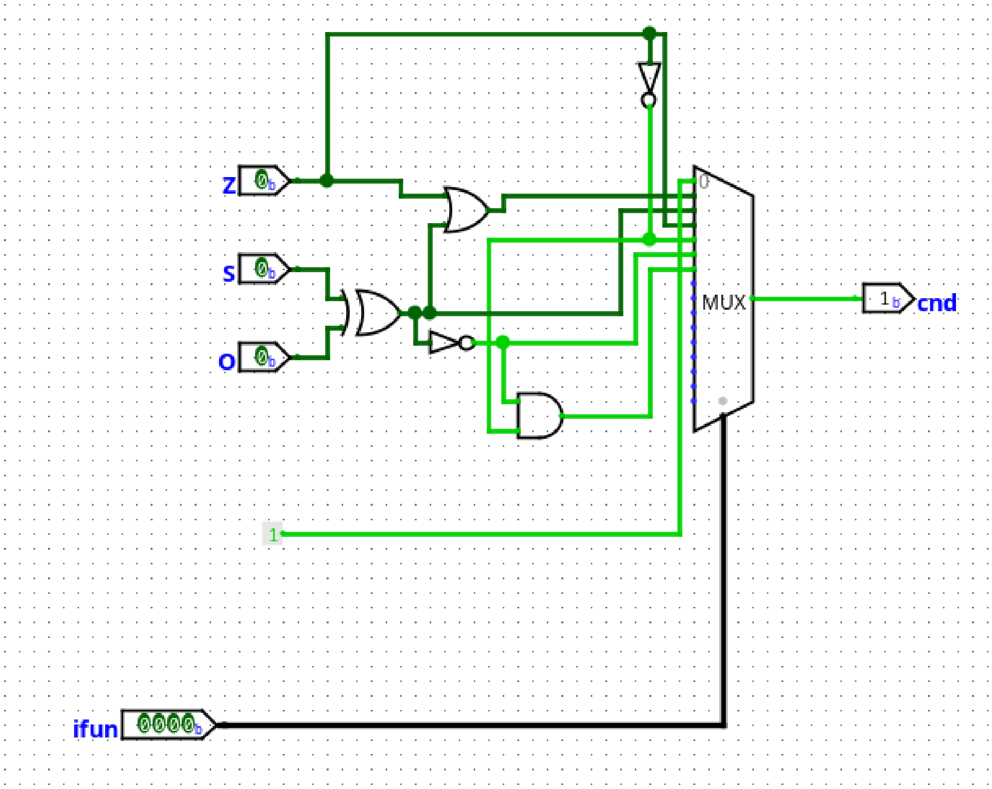
The ALU component takes in the inputs from fn, aluA, and aluB. It calculates the desired operation in valE. Moreover, it looks at the result from valE and determines the flags such as ZF (comparing to zero), SF (looking at the most significant bit for a signed number), CF (carry), and OF (determining overflow based on whether the operation is addition or subtraction).



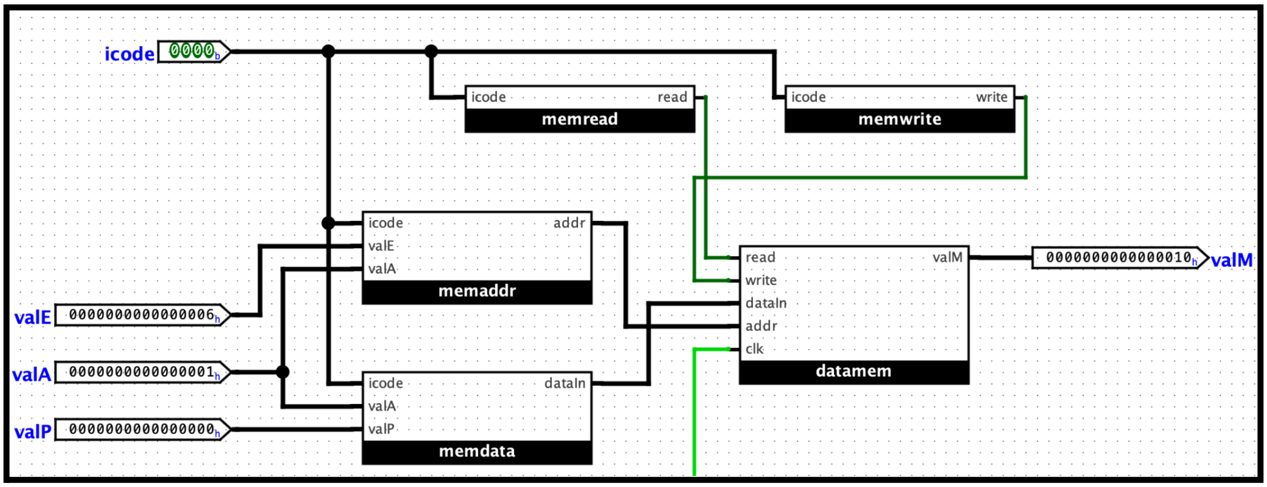
The calculated flags, however, are only written into the flags registers whenever there was an arithmetic operation, also known as icode==6 (determined by a comparator whose output feeds to the write enable of the flag registers).



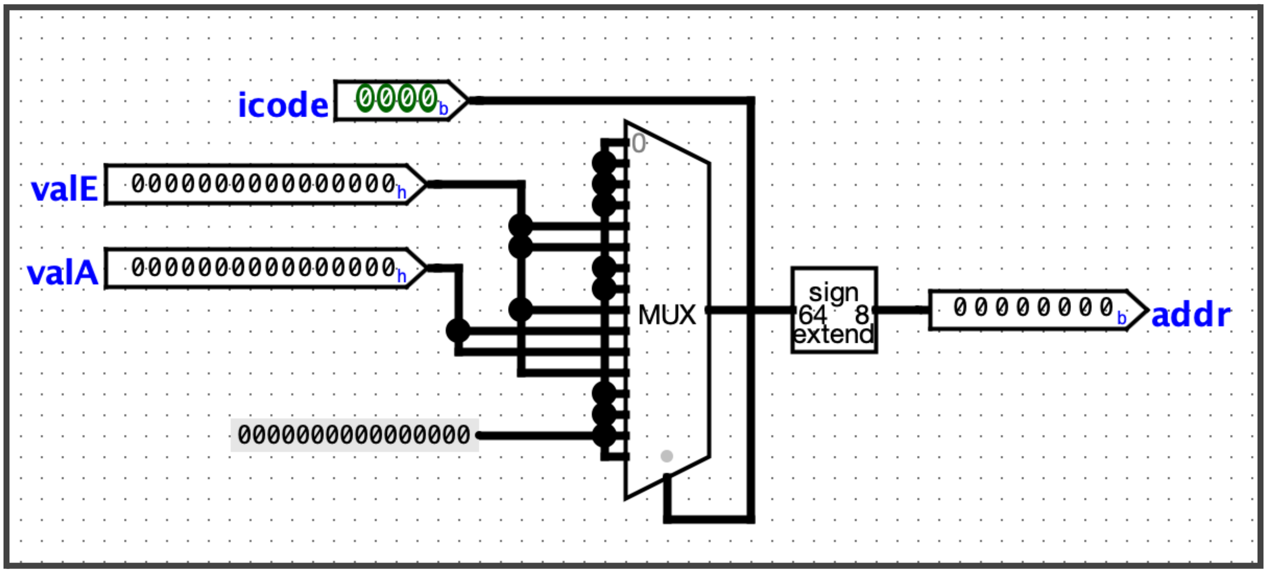
The cc module takes the outputs of these flag registers. When icode==7, there will be a jump, and which type of jump is determined by ifun. The mux selects (using the ifun) whether or not the condition of a jump is met (such as if ZF -> two inputs were equal), and outputs this to the cnd bit.



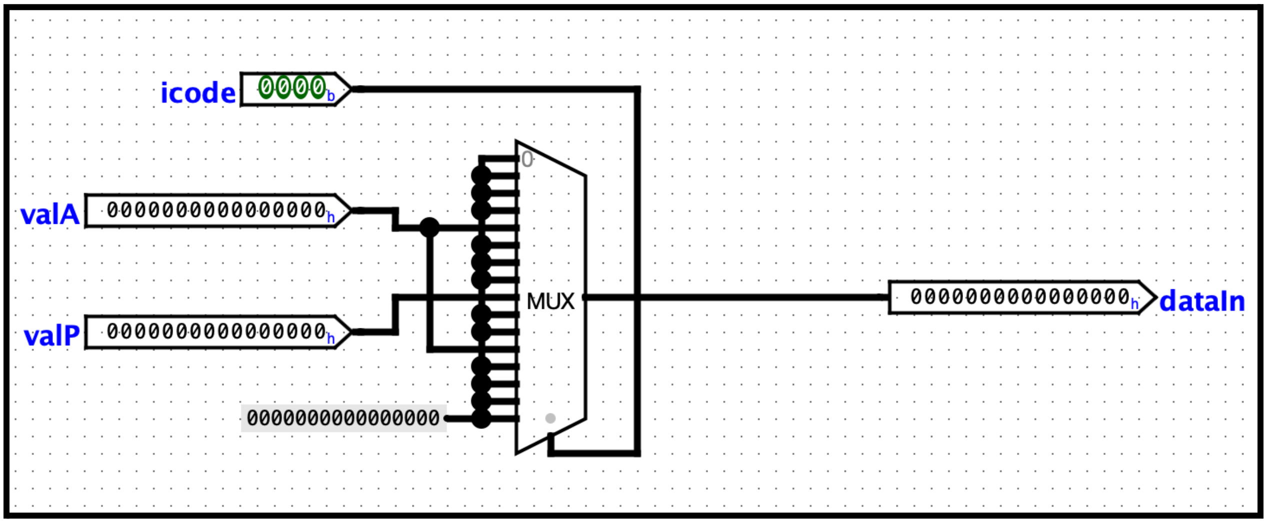
Memory



The block below is the memaddr block. It outputs the address (either valA or valE) that we will read/write from based on the icode. A mux is used to select valA or valE. For icode of 4, 5, 8, and b(11) valE will be used. For icode of 9, and a(10) valA will be used. Then an extender is used to translate it into an 8bit address.

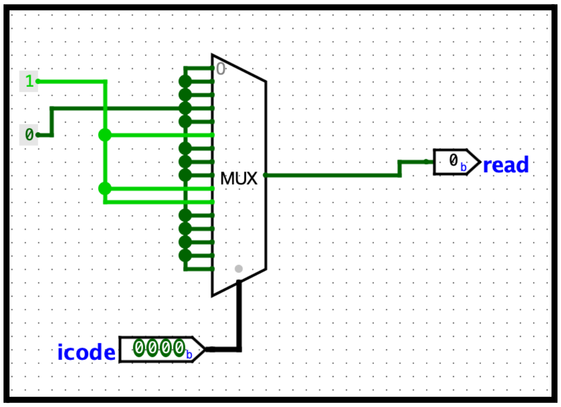


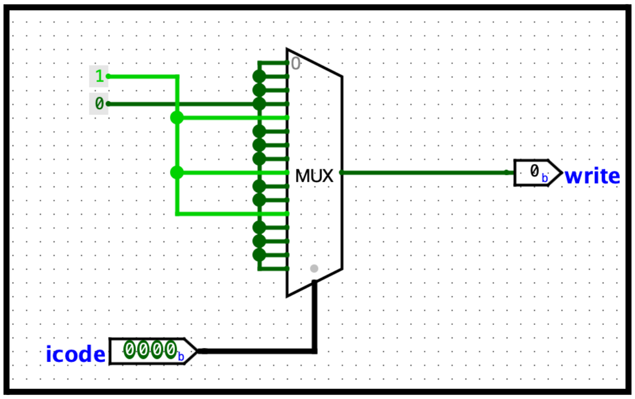
The block below is the memdata block, it works similar to the memaddr block in the way that it determines which data will be going in, based on icode. There will only be dataIn if it's writing, which will be icodes 4, 8, and b(11). For icodes 4 and b, the dataIn will be valA. For icode of 8, dataIn will be valP. For the remaining icodes, the dataIn will be zero.



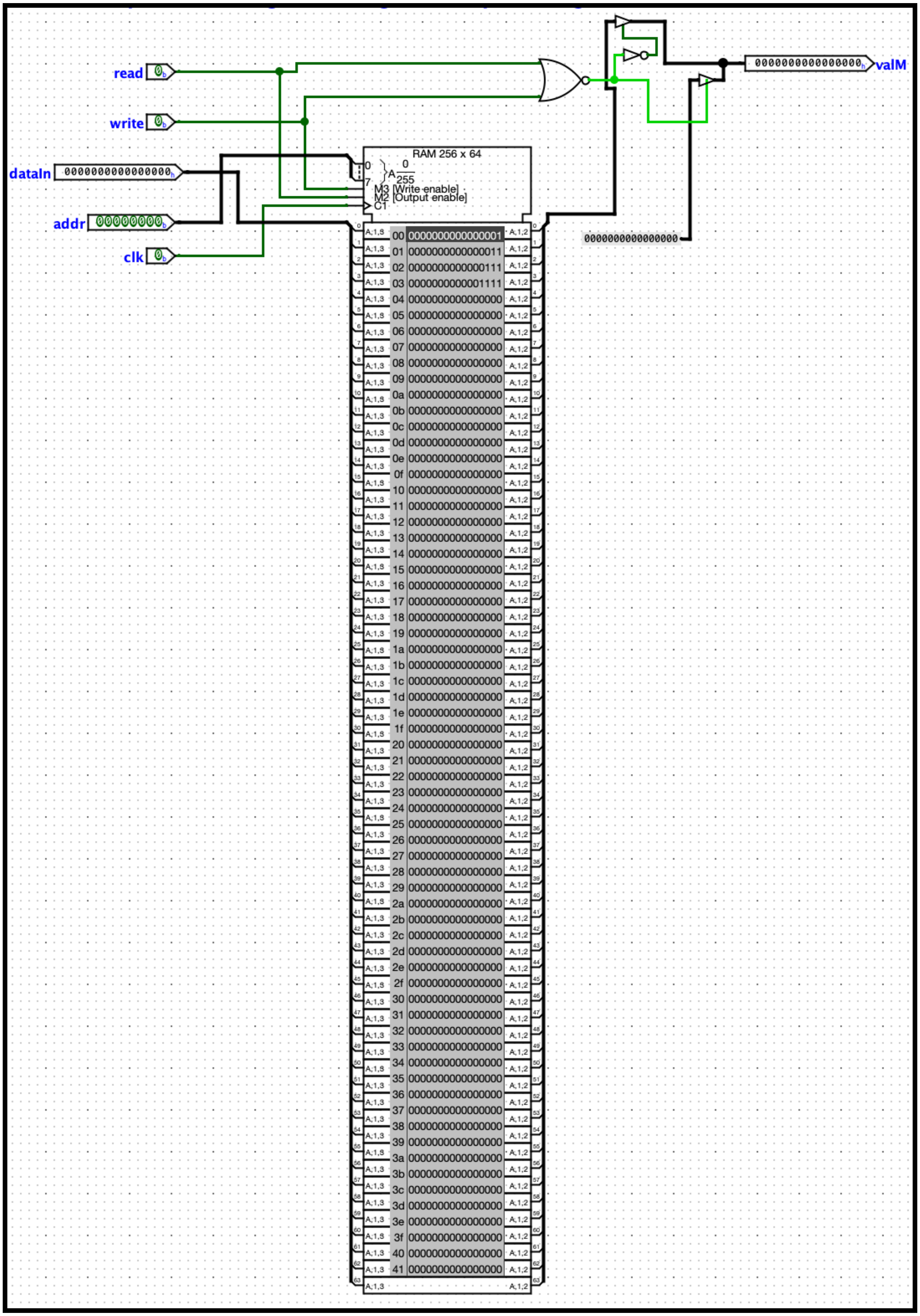
The 1st block below utilizes a mux to determine whether read will be enabled based on the icode. For icodes 5, 9, and a(10) it will be enabled, for the remaining icodes it will be disabled.

The 2nd block below utilizes a mux to determine whether write will be enabled based on the icode. For icodes 4, 8, and b(11) it will be enabled, for the remaining icodes it will be disabled.





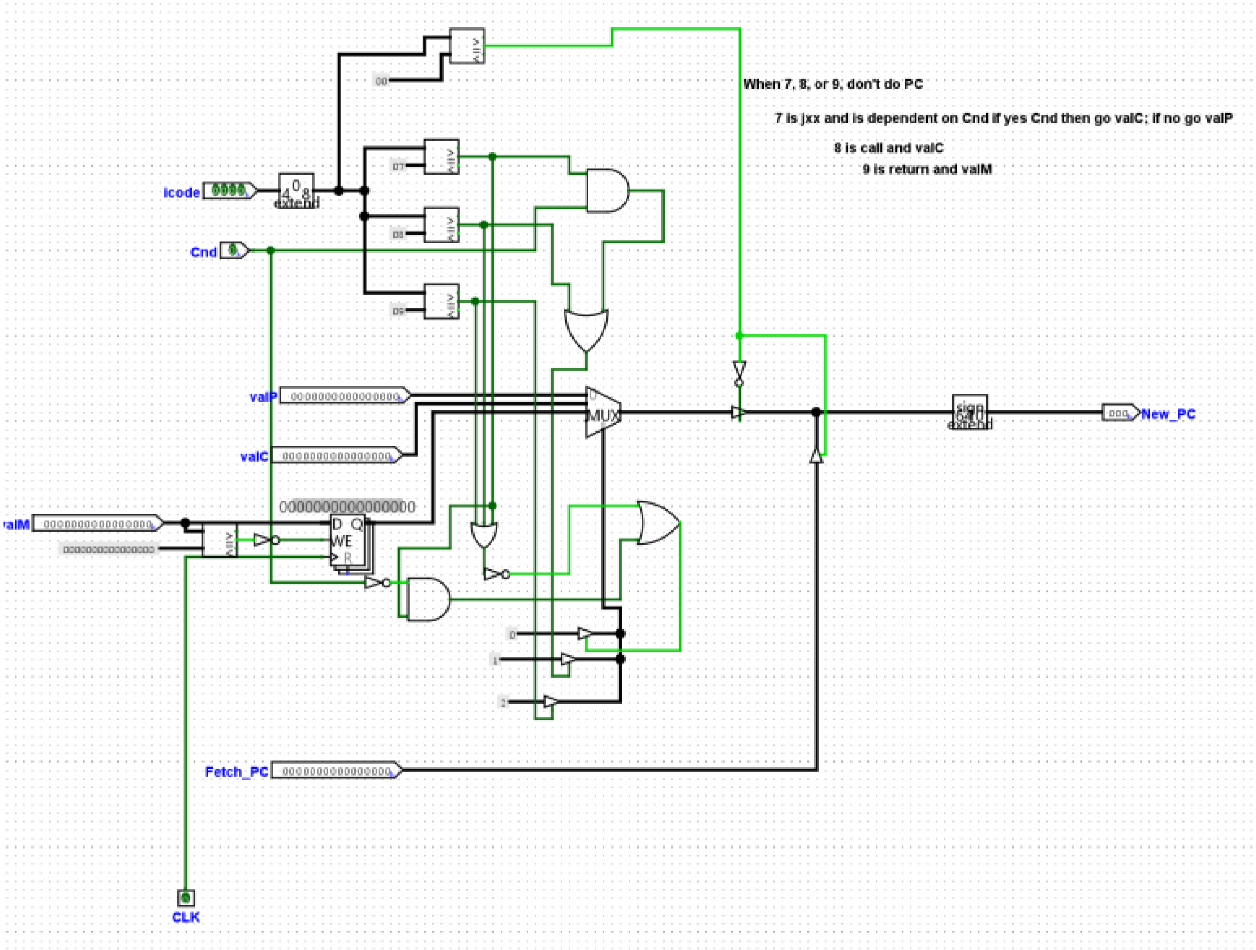
Below is the data memory block. It takes inputs of address, dataIn, read enable, write enable, and clk. It reads or writes to RAM based on the memory address given. If read and write are 0, valM is zero by default. When it reads, the value output is valM.



00	0000000000000001
01	0000000000000011
02	0000000000000111
03	0000000000001111
04	0000000000000000
05	0000000000000000
06	0000000000000000
07	0000000000000000
08	0000000000000000
09	0000000000000000
0a	0000000000000000
0b	0000000000000000
0c	0000000000000000
0d	0000000000000000
0e	0000000000000000
0f	0000000000000000
10	0000000000000000
11	0000000000000000
12	0000000000000000
13	0000000000000000
14	0000000000000000
15	0000000000000000
16	0000000000000000
17	0000000000000000
18	0000000000000000
19	0000000000000000
1a	0000000000000000
1b	0000000000000000
1c	0000000000000000
1d	0000000000000000
1e	0000000000000000
1f	0000000000000000
20	0000000000000000
21	0000000000000000
22	0000000000000000
23	0000000000000000
24	0000000000000000
25	0000000000000000
26	0000000000000000
27	0000000000000000
28	0000000000000000
29	0000000000000000
2a	0000000000000000
2b	0000000000000000
2c	0000000000000000
2d	0000000000000000
2e	0000000000000000
2f	0000000000000000
30	0000000000000000
31	0000000000000000
32	0000000000000000
33	0000000000000000
34	0000000000000000
35	0000000000000000
36	0000000000000000
37	0000000000000000
38	0000000000000000
39	0000000000000000
3a	0000000000000000
3b	0000000000000000
3c	0000000000000000
3d	0000000000000000
3e	0000000000000000
3f	0000000000000000
40	0000000000000000
41	0000000000000000

Write Back / PC update

Write back happens right after the decode stage. After valA and valB are read from the register file, Execution and Memory have the data needed to run outputting valE and valM. The first clock cycle after valA and valB is read, valE is written into the register file. On the next clock cycle, valM is written into the register file.



When 7, 8, or 9, don't do PC

7 is jxx and is dependent on Cnd if yes Cnd then go valC; if no go valP

8 is call and valC

9 is return and valM

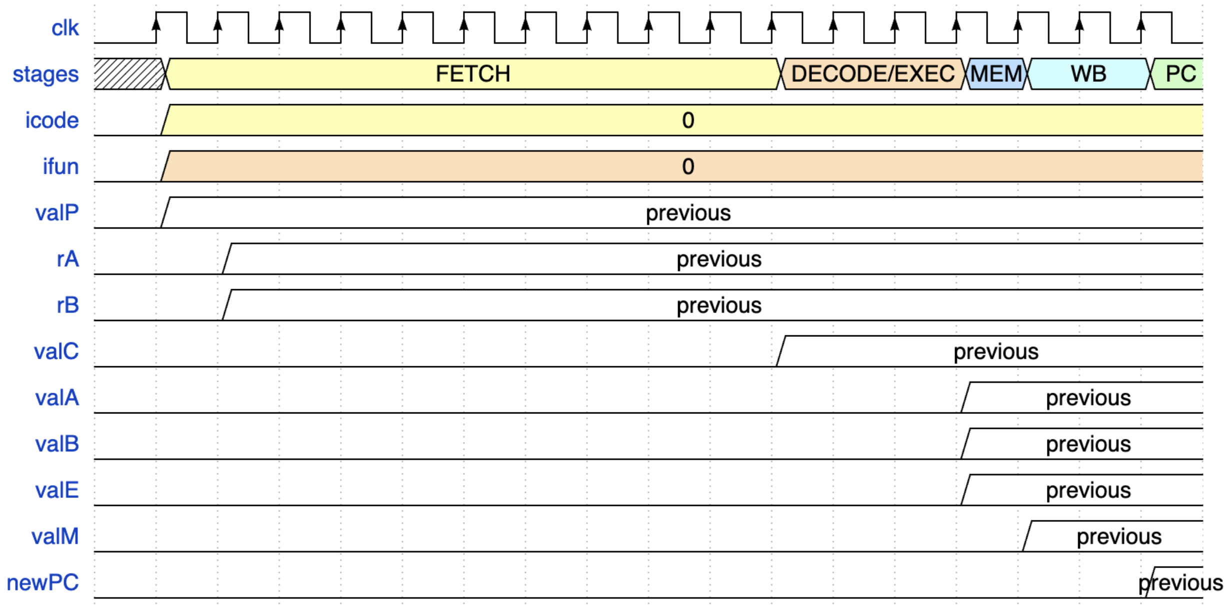
This is the PC update. It takes in 7 values, icode, Cnd, valP, valC, valM, clk, and fetch_pc to output a new_pc value. This circuit is mostly conditional logic which takes in icode and outputs either valP, valC, or valM. However, when icode is 0 (nop) the original PC value (fetch_pc) is outputted instead. Whenever the icode is 7, the output will be either valC or valP depending if Cnd is high or low. Whenever the icode is 8, the output is valC. Whenever icode is 9, the output is valM. However, valM is stored in a register and held until a new value for valM which is not 0 is inputted. This is to facilitate the return instruction.

Problem 2

The timing diagram for each output of each stage.
  Signals must include icode, ifun, rA, rB, valC, valP, valA, valB, valE, valM, new PC

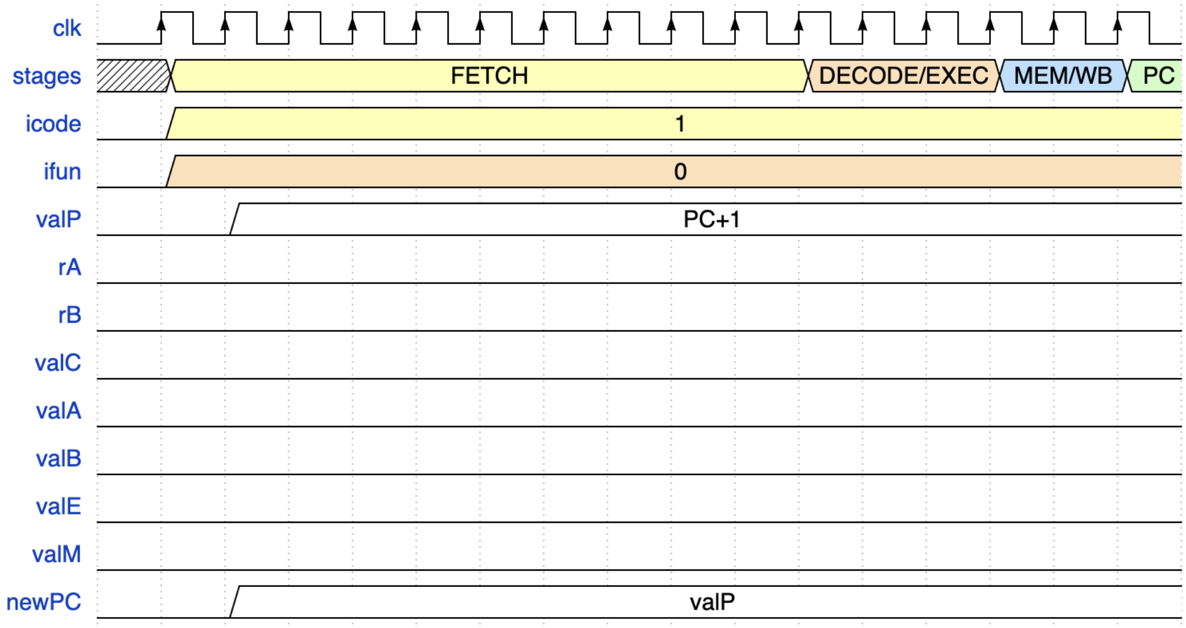
`halt`

0	0



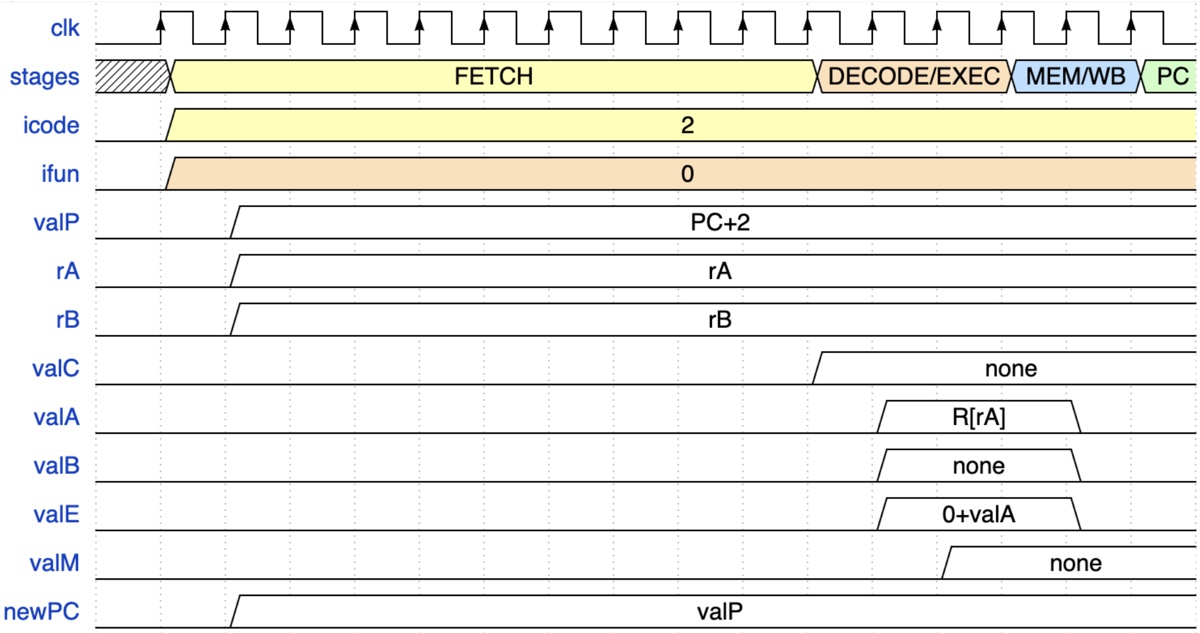
`nop`

1	0



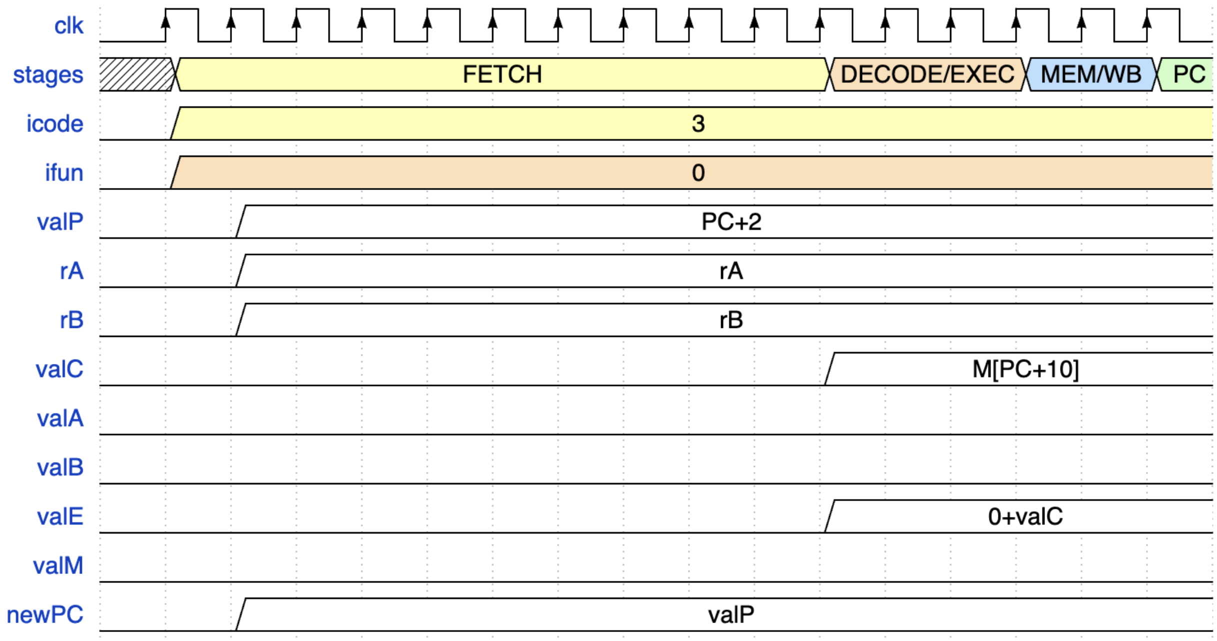
`rrmovl` rA, rB

2	0	rA	rB



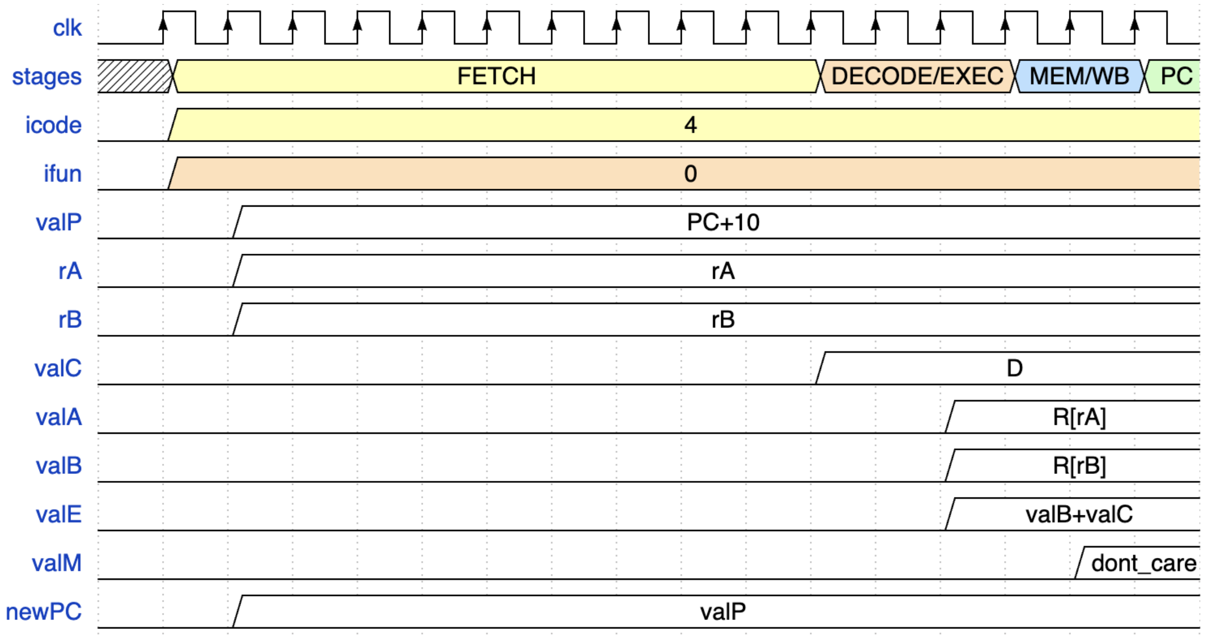
`irmovl` V, rB

3	0	F	rB	V



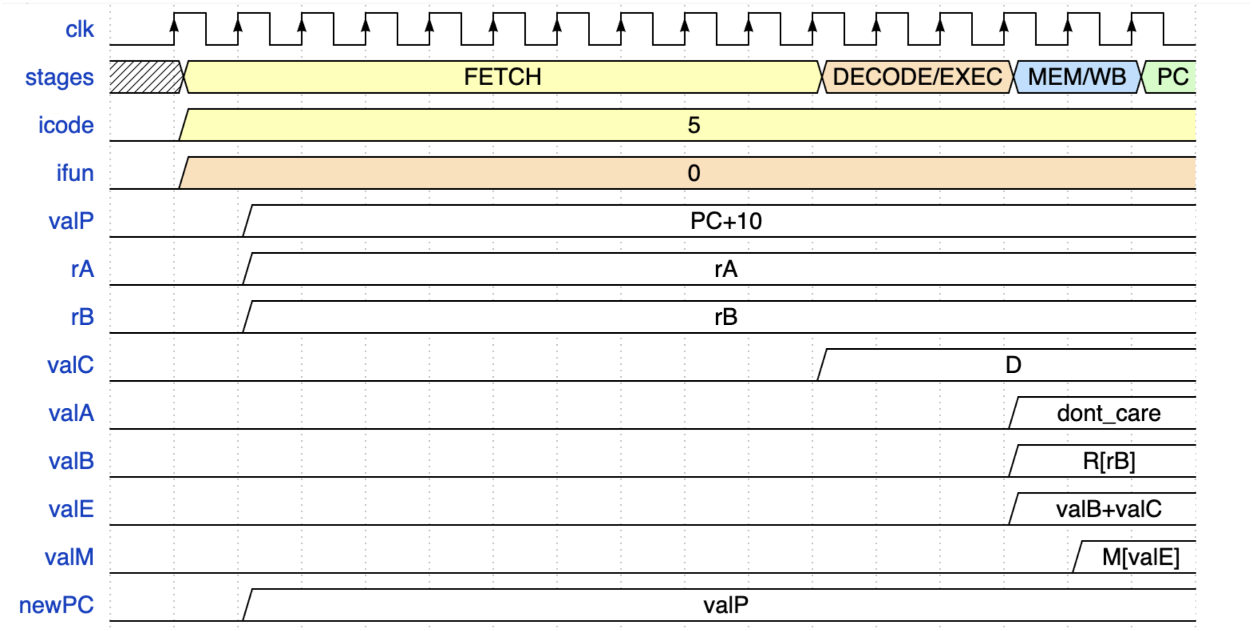
`rmmovl` rA, D(rB)

4	0	rA	rB	D



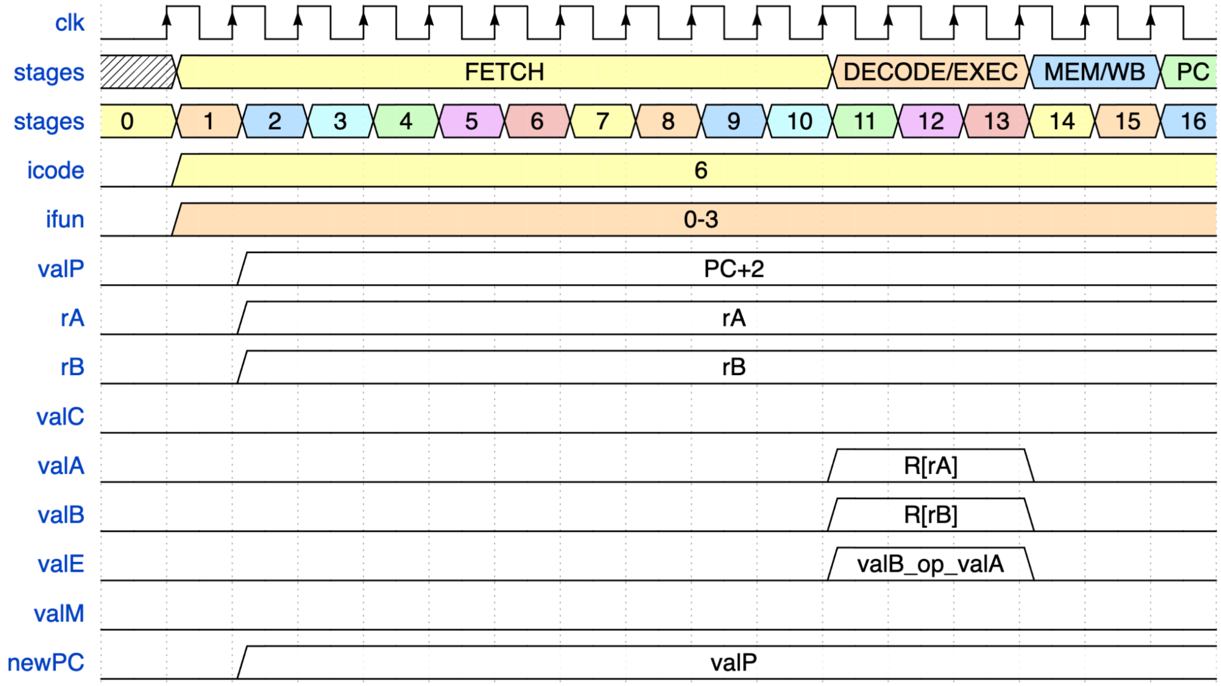
`mrmovl` D(rB), rA

5	0	rA	rB	D



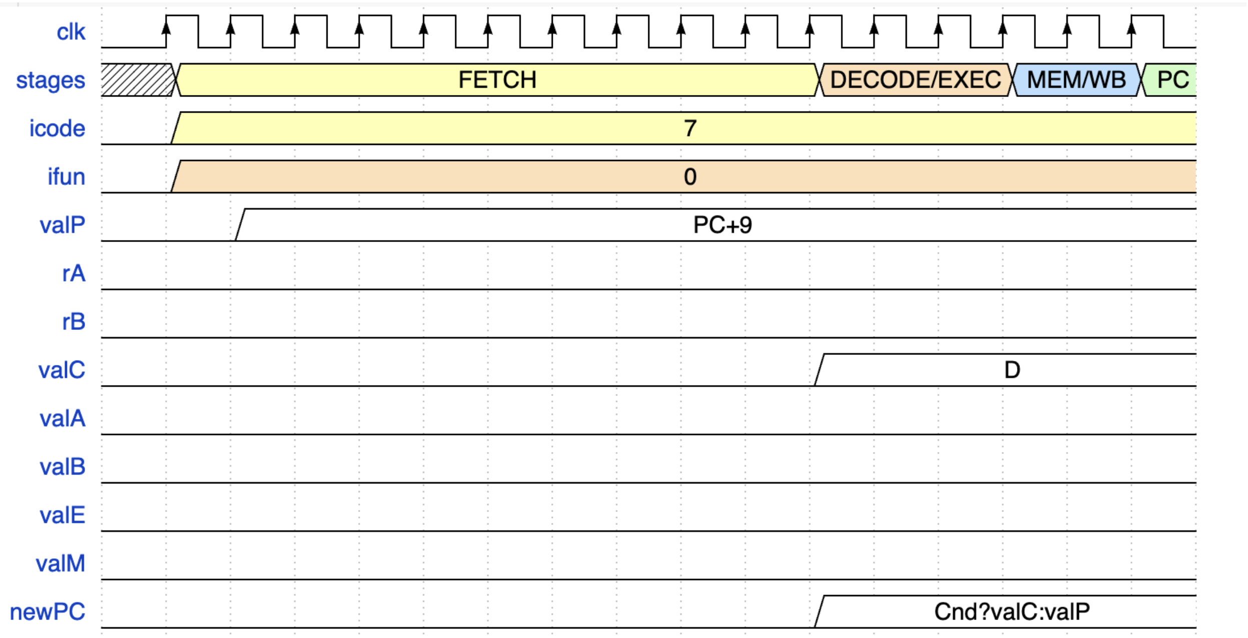
OP1 rA, rB

6	fn	rA	rB



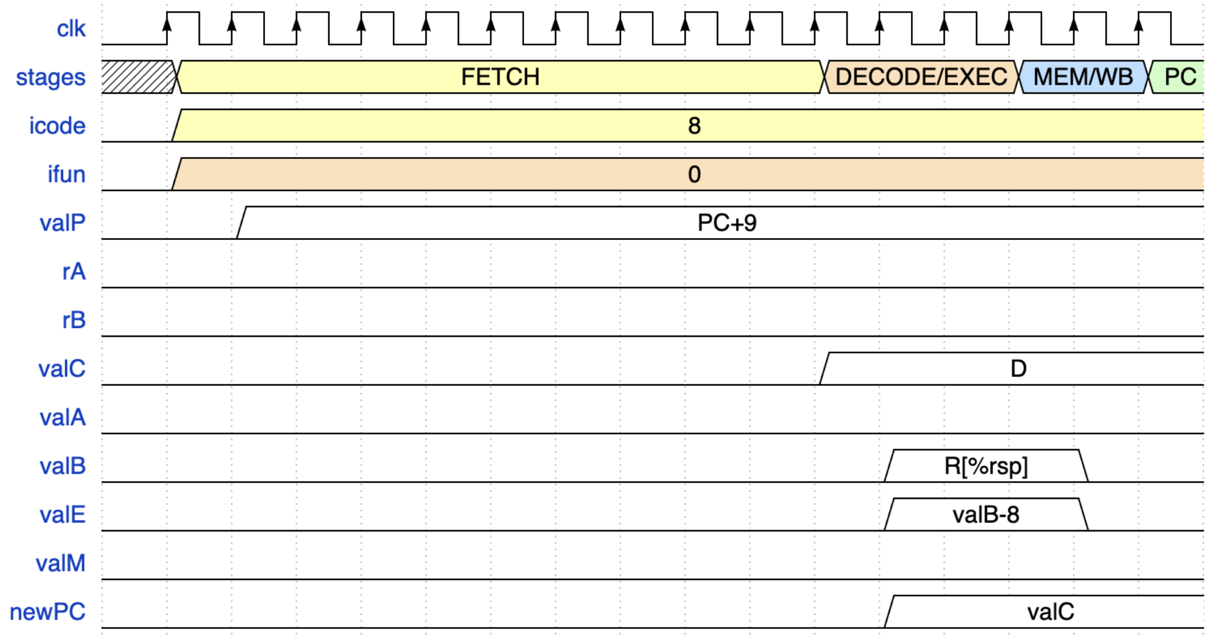
jXX Dest

7	fn	Dest



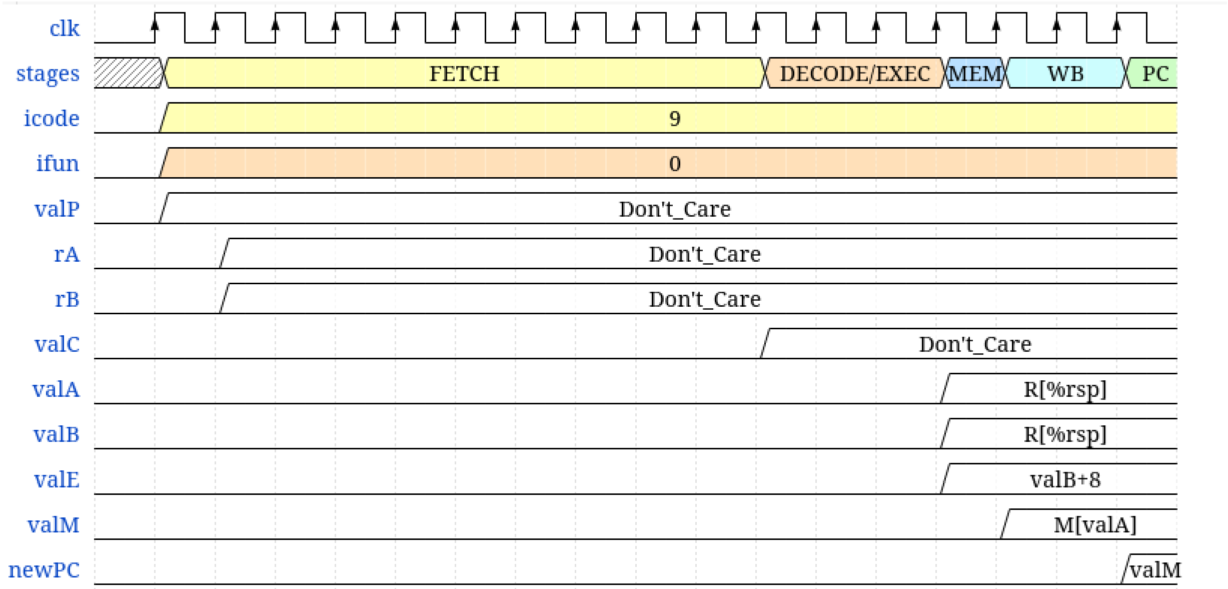
`call` **Dest**

8	0	**Dest**



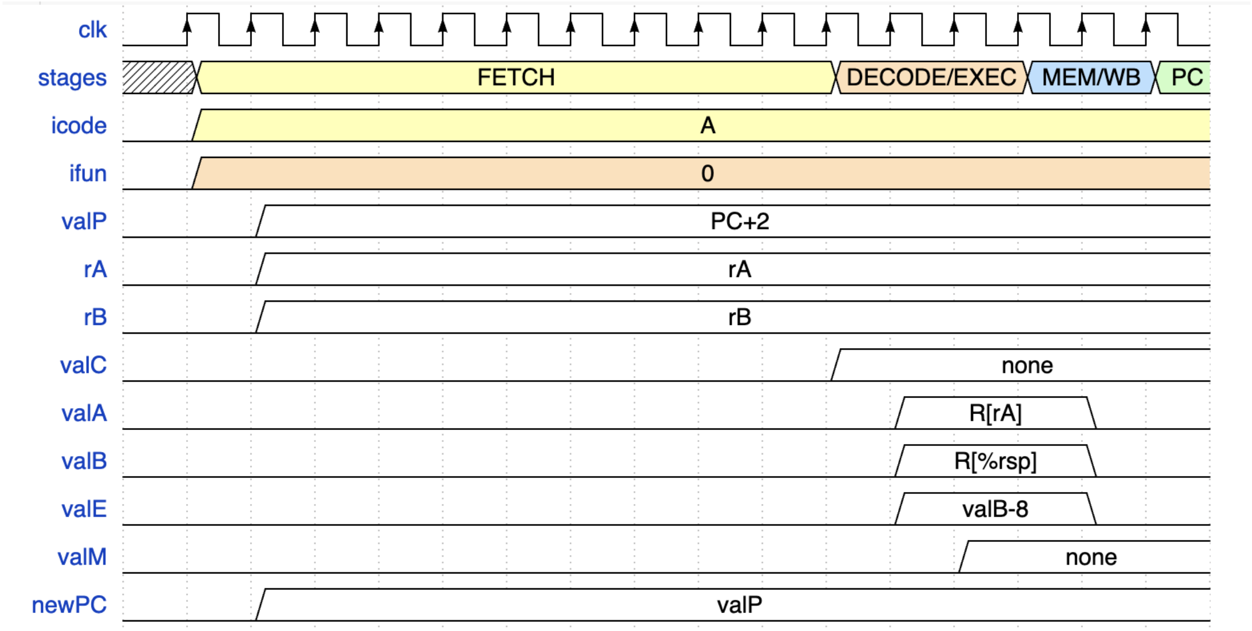
ret

9	0



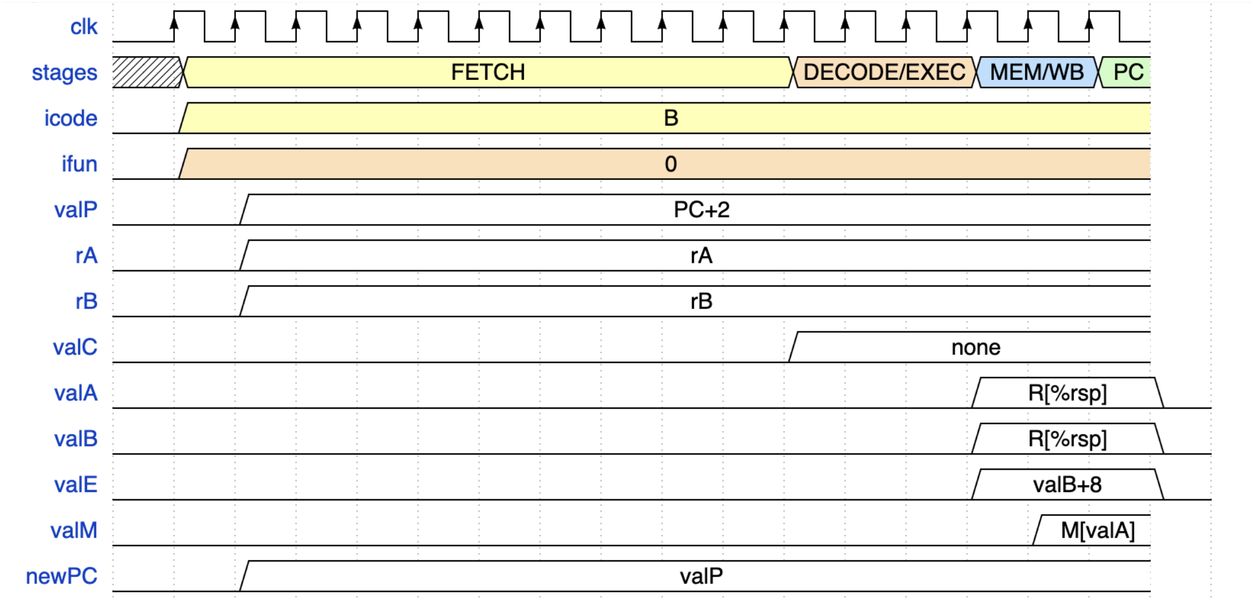
pushl rA

A	0	rA	F



`popl` **rA**

B	0	rA	F


clk
stages
FETCH
DECODE/EXEC
MEM/WB
PC
icode
B
ifun
0
valP
PC+2
rA
rA
rB
rB
valC
none
valA
R[%rsp]
valB
R[%rsp]
valE
valB+8
valM
M[valA]
newPC
valP