Name: Anh Ha

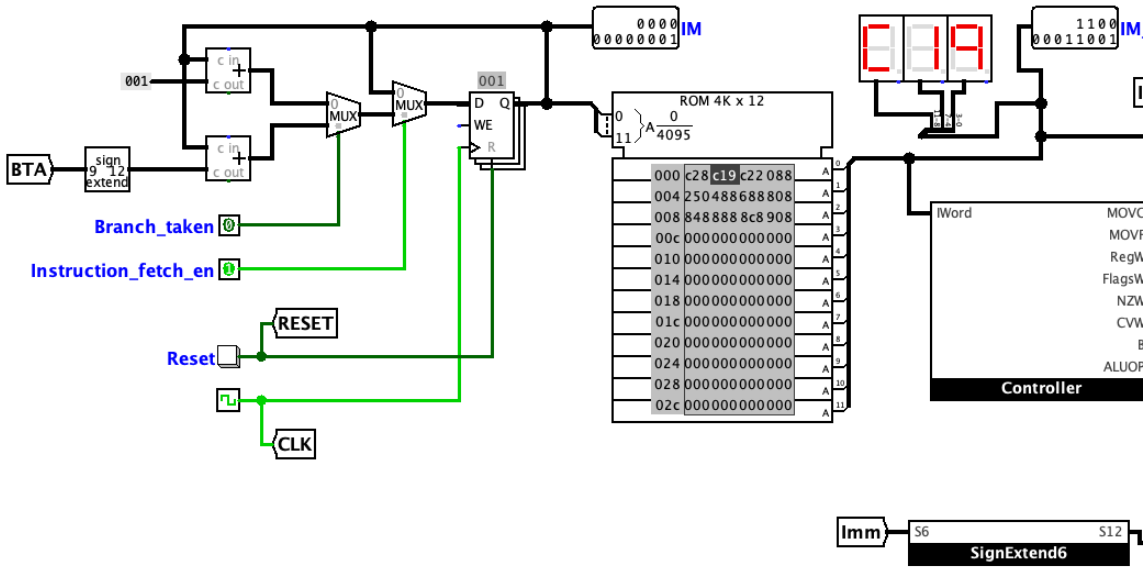
Student ID: 4603

FINAL PROJECT

This final project demonstrates the phased process of building a CPU. Initially, we will design each individual component of the CPU, including the Instruction Fetch Unit, the ALU, the Register File, the Flags Register, and the Controller. Following this, we will wire these components together and test them by executing small assembly language programs. After each test, we will not only verify the CPU's operation according to the previous instruction but also enhance the CPU's design to enable the execution of new instructions.

Build the Instruction Fetch Unit

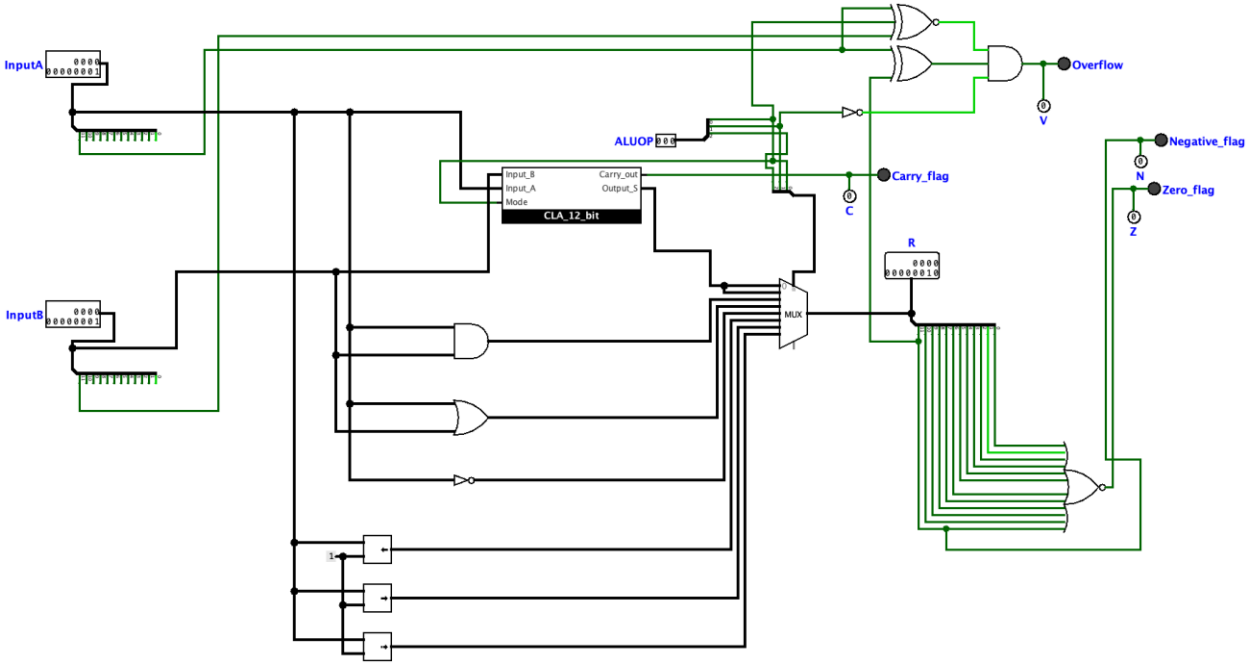
The main purpose of building the instruction fetch unit is to fetch the next instruction to execute. Because we are using word-aligned memory with 12-bit words, the next instruction is at the next location in memory. This means that the next instruction is at PC+1. So, I added the instruction memory with a constant 1 to move to the next instruction location. I also connected the **Reset** and **Clk** signals to the register so that we can reset the register to get back to instruction address zero. The branch instruction will be executed by adding the extended immediate to the instruction (bit 0-8), which is controlled or selected by the branch_taken signal. The next instruction can only go through if the instruction_fetch_en signal is high.



Build the ALU

- The ALUOP will determine which operation the ALU is going to implement. The logic is similar to the ALU that we built in the last assignment.
- Operation logic:
  - Add and Subtract Logic: The circuit contains the 12-bit adder/ subtractor
  - from problem 2 to operate addition and subtraction. Use the least significant
  - bit of the Opcode as a carry-in to the adder/ subtractor to determine which
  - operation is going to be implemented.
  - Bitwise AND: Perform bitwise AND between A and B using an AND gate.
  - Bitwise OR: Perform bitwise OR between A and B using an OR gate.
  - Bitwise NOT: Perform bitwise NOT on input A using a NOT gate.
  - LSL: Perform the logical shift left using the built-in shifter in Logisim.
  - LSR: Perform the logical shift right using the built-in shifter in Logisim.
  - ASR: Perform the arithmetic shift right using the built-in shifter in Logisim.

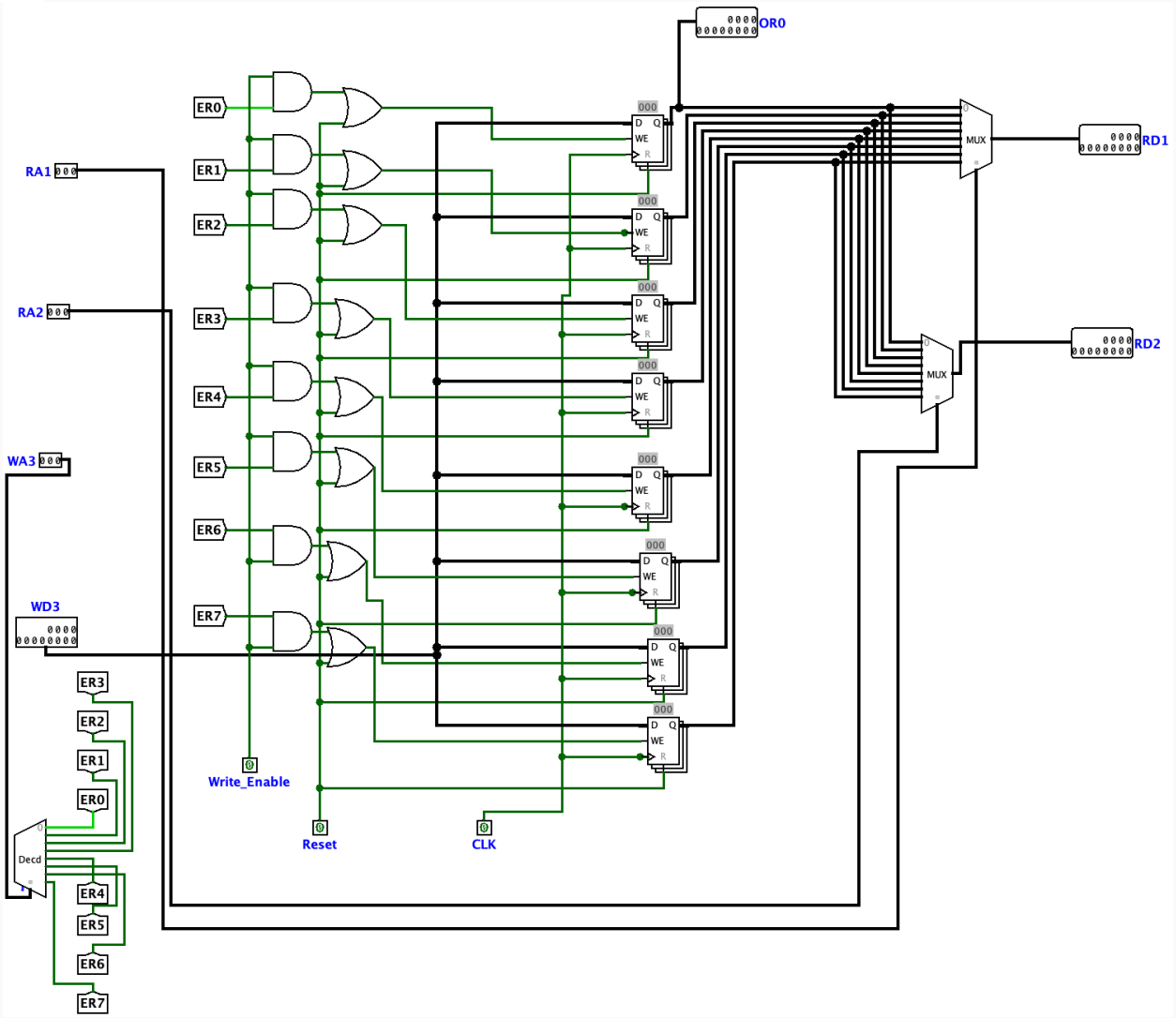
- Operations Mapping:
  - ALUOP 000: Add
  - ALUOP 001: Subtract
  - ALUOP 010: AND
  - ALUOP 011: OR
  - ALUOP 100: NOT (only applies to input A)
  - ALUOP 101: LSL
  - ALUOP 110: LSR
  - ALUOP 111: ASR
- The flags logic:
  - Negative (N): Check the most significant bit (MSB) of the result. If MSB is 1, set N high.
  - Zero (Z): Check if all bits of the result are zero. If true, set Z high.
  - Carry (C): Use the carry-out from the adder/subtractor for addition/subtraction operations.
  - Overflow (V): Detect signed overflow for addition/subtraction. Overflow occurs if all three conditions are satisfied:
    - (1) The ALU is performing addition or subtraction, which means the Opcode1 = 0.
    - (2) Input A and output S have opposite signs, which can be detected by the XOR gate or the XNOR gate.
    - (3) Either the two inputs A and B have the same sign and the adder is performing subtraction (Opcode0 = 1) have carry into the MSB differs from the carry out of the MSB.



Build the Register File

- The RA1 input specifies the three-bit address for the register that will output data on RD1. Similarly, the RA2 input provides the three-bit address for the register that will output data on RD2. The WA3 input designates the three-bit address for the register that will receive data via the WD3 input. Therefore, I use multiplexers (MUXes) to select the appropriate register for reading and writing operations based on the addresses provided by RA1, RA2, and WA3. Connect RA1 and RA2 to the multiplexers to select the registers for RD1 and RD2.
- I Implemented 8 registers to store the data. These registers will be accessed based on the addresses provided by RA1, RA2, and WA3.
- The RegW input controls the write operation to the register addressed by WA3. Therefore, to write data to the register specified by WA3, the data must be available on WD3, and the RegW input must be set to a logic high. register is generated by AND the decoder value.
- The Write data is always connected to every register; we control which register we get written to by changing the Write_en input on each register.
- WA3 is a function for the decoded value mapped to a pin[0, 8]. The write_en signal for each register is generated by AND the decoder value with RegW, then OR the result with the Reset signal. By doing it that way, we do not lose the stored value of the previous instruction when executing the next instruction. All registers will be reset when the Reset signal is on.

- Place a 12-bit output pin for OR0 to monitor the contents of R0.

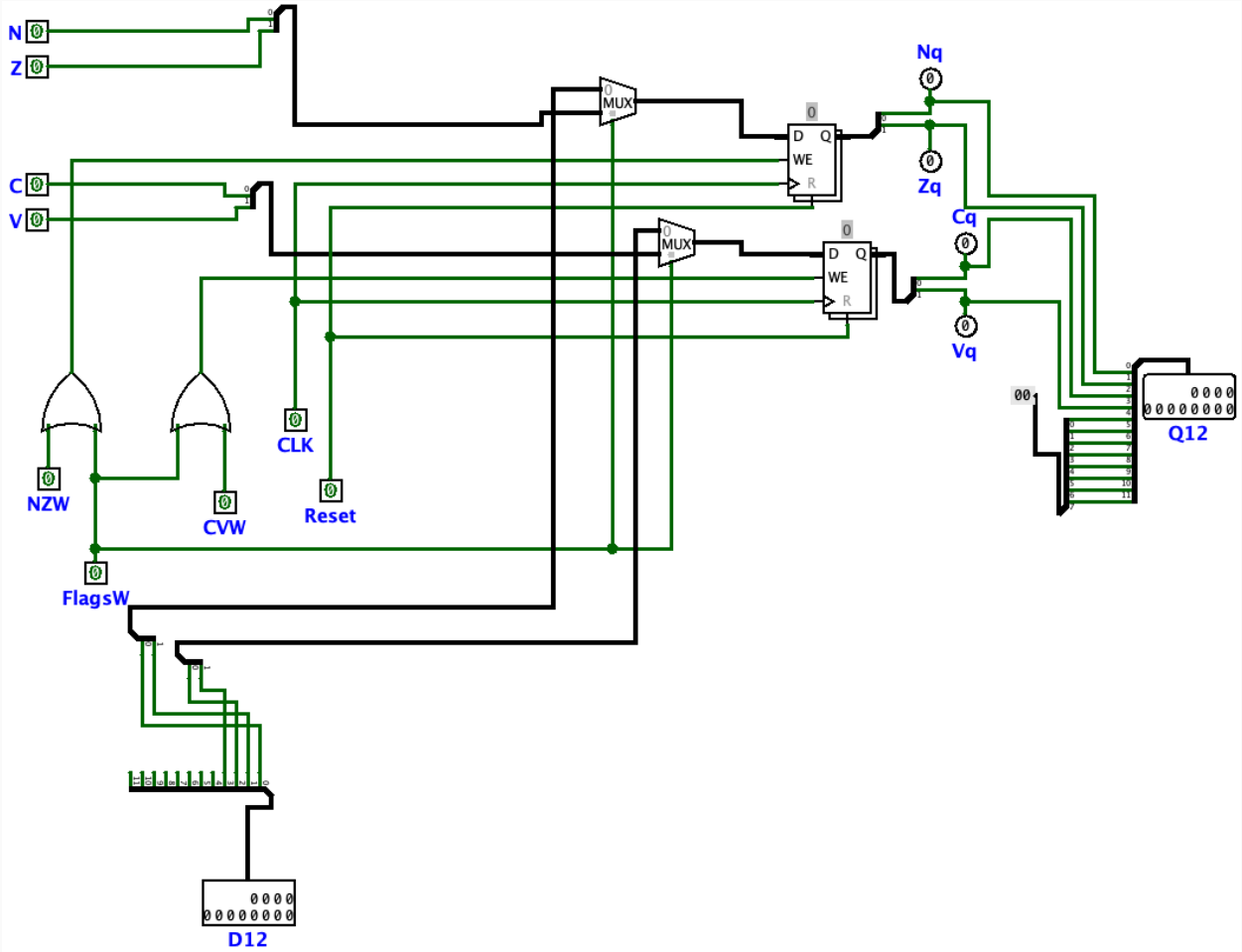


Thinking Through the FLAGS Register

- I used two two-bit registers: one for N and Z, and one for C and V.
- There are two **write enable signals,** one that enables updates to the NZ pair, and the other one for the CV pair. These come from the control units (NZW and CVW)
- The NZW controls flags N and Z. Similarly, CVW controls flags C and V. The FlagsW will control all the flags N, Z, C, V.
- The FlagsW is OR with each control unit (NZW and CVW) to enable the Register to output corresponding flags N, Z, C, V.
- The FlagsW is also a control signal that changes **where** the input comes from between the ALU output and a seperated 12-bit bus that is routed in through D12. I also added a

small MUX in front of each flag pair to choose between the ALU output and the external value.

- Connect the two registers with Reset and Clock.
- The lowest 4 bits of the output Q12 is reserved for the 4-bit flag N, Z, C, V. The left bits are ignored or reserved for other purposes.
- Testing it by executing the MOVRF which updates all flags.



Building the Decoder/controller

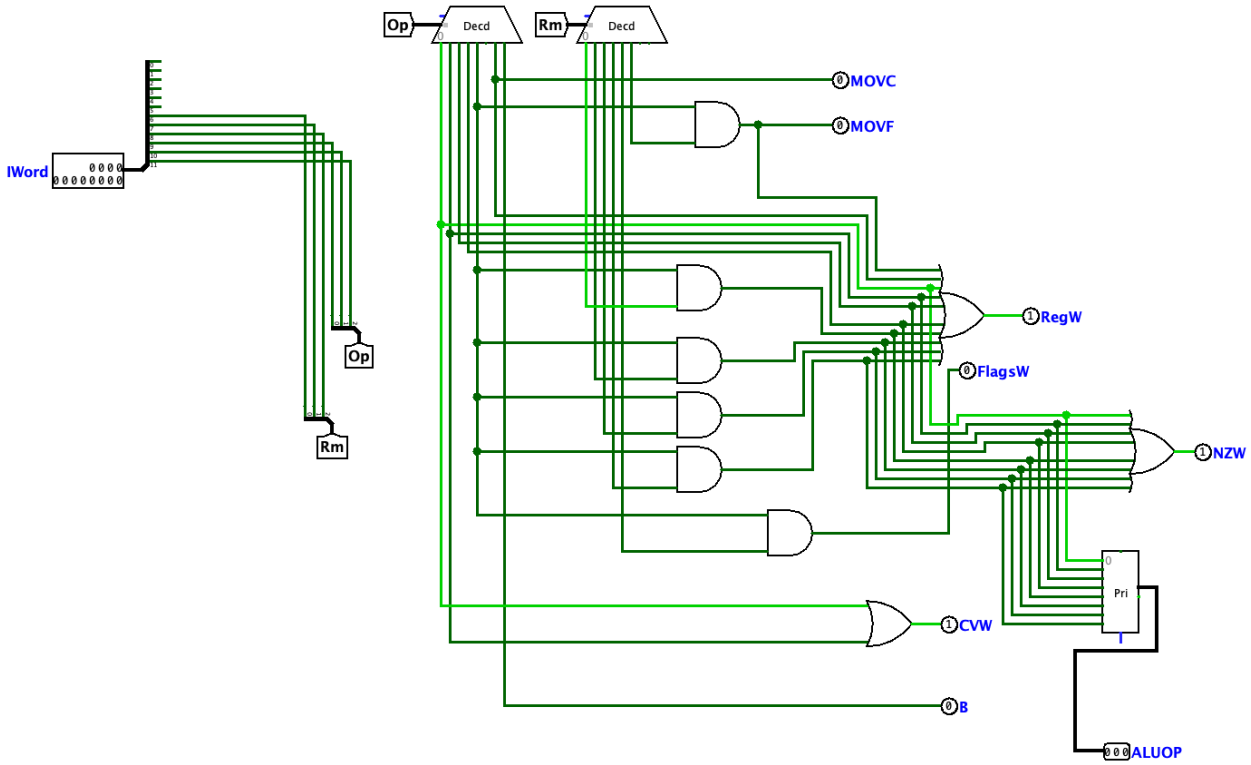
Control signal logic:

- MOVC: Asserted for the MOVC instruction.
- MOVF: Asserted for the MOVFR instruction.
- RegW: Asserted for instructions that write to a general-purpose register, including ADD, SUB, AND, OR,...
- FlagsW: Asserted for the MOVRF instruction.
- NZW: Asserted for ALU-based instructions that affect the N and Z flags.

- CVW: Asserted for ADD and SUB instructions to update the carry and overflow flags.
- B: Selects the source for the ALU's second input (immediate value for MOVC).
- ALUOP: 3-bit signal to select the ALU operation.

I used two decoders and all combinational circuits to match the logic with the control signals based on the truth table below. This truth table maps each instruction to its corresponding control signals

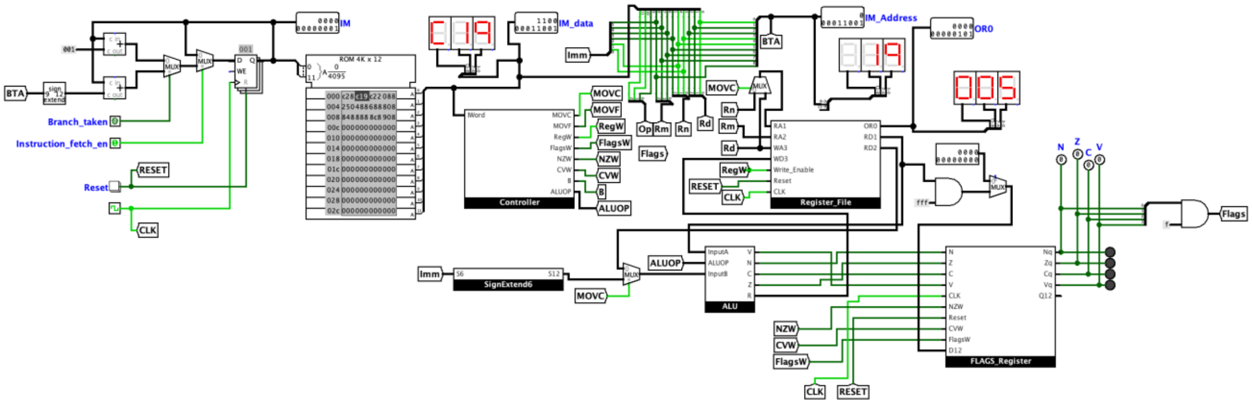
Instruction	MOVC	MOVF	RegW	FlagsW	NZW	CVW	B	ALUOP - SubOp
ADD	0	0	1	0	1	1	0	000
SUB	0	0	1	0	1	1	0	001
MOVC	1	0	1	0	0	0	1	110
MOVFR	0	1	1	0	0	0	0	100-101
MOVRF	0	0	0	1	0	0	0	100-100
BNZ	0	0	0	0	0	0	0	111
AND	0	0	1	0	1	0	0	010
OR	0	0	1	0	1	0	0	011
NOT	0	0	1	0	1	0	0	100-000
LSL	0	0	1	0	1	0	0	100-001
LSR	0	0	1	0	1	0	0	100-010
ASR	0	0	1	0	1	0	0	100-011



Developing the DataPATH

After completing the design of all individual components, I connected all components to form the complete DataPath. Ensure data flows correctly between register file, the ALU, multiplexers, and the FLAGS register. Verify that control signals are correctly routed to manage data flow. I used multiplexers when needed to determine which path to take based on the control signals.

Test and validate: Using the provided assembler and the Mac terminal to write assembly codes and convert it to hex file. Then, the hex file is loaded into the ROM. Simulate the DataPath to ensure it performs correctly for each instruction and then adding another instruction to the .asm file. Validate that data is routed correctly, operations are performed accurately, and control signals are asserted as expected.



IM
BTA
Branch_taken
Instruction_fetch_en
Reset
RESET
CLK
ROM 4K x 12
IM_data
Imm
Controller
MOVC
MOVF
RegW
FlagsW
NZW
CVW
B
ALUOP
Op
Rm
Rn
Rd
Flags
IM_Address
OR0
Register_File
SignExtend6
ALU
FLAGS_Register
N
Z
C
V