

# Computer Organization

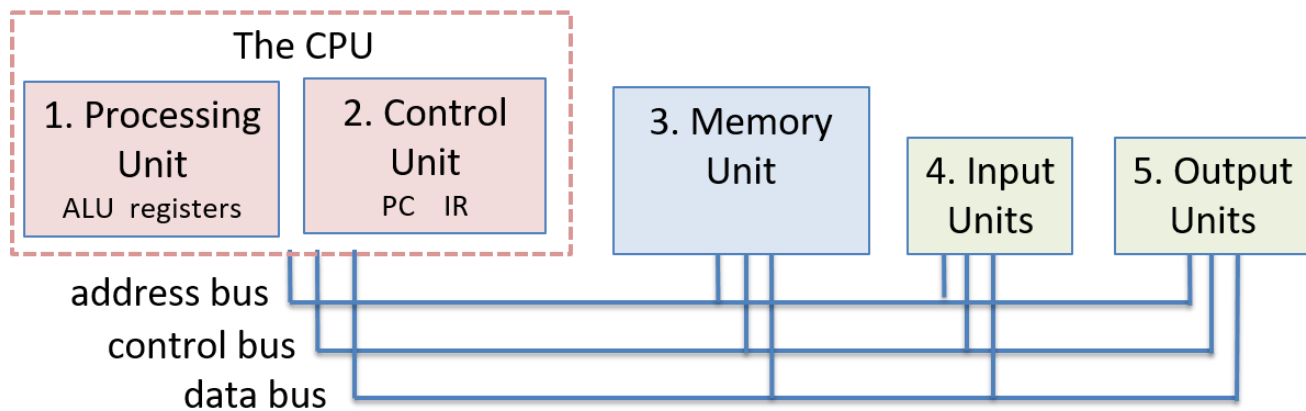
CMSC 313  
Raphael Elspas

# von Neumann architecture

- The von Neumann architecture is a scheme that includes the following components
- A processing unit with both an **arithmetic logic unit** and processor **registers**
- A control unit that includes an **instruction register** and a **program counter**
- **Memory** that stores data and instructions
- External mass storage
- Input and output mechanisms

# von Neumann architecture bus

- The **bus** in the von Neumann architecture is shared between all major components



# Busses

- There are 3 kinds of busses in the von Neumann architecture

Data Bus	A bidirectional communication path that transfers data between the CPU, memory, and I/O devices.
Address Bus	A unidirectional communication path used by the CPU to reference physical addresses in memory and I/O devices.
Control Bus	A set of communication paths that transmit control signals for managing system operations.

# Computer components

- CPU
  - Register
  - ALU
- Memory
- Inputs
- Outputs

# Special registers

- Program Counter (PC)
  - Keeps track of the memory address of the next instruction to be fetched and executed.
  - Essentially a number that gets incremented after every instruction is read.
  - Sometimes the number has to jump forwards or backwards if the outcome of an instructions execution relates to the control flow of the program

## Special registers (cont.)

- Instruction Register (IR)
  - The Instruction Register (IR) is a register inside the central processing unit (CPU) of a computer that holds the current instruction being executed or decoded. It is part of the CPU's control unit, which manages the execution of instructions and the flow of data within the processor.
  - The Instruction Register essentially serves as a temporary storage location for the instruction being processed. It allows the CPU to hold and manipulate the instruction data without repeatedly accessing the main memory during the execution of each stage in the Fetch-Decode-Execute cycle.

# Fetch Decode Execute

- **Fetch**

- The CPU fetches the next instruction from the memory location pointed to by the program counter (PC) and loaded into the Instruction Register (IR)
- The program counter (PC) is then incremented to point to the next instruction.

- **Decode**

- The fetched instruction is decoded to determine the operation to be performed and the operands involved.
- The control unit extracts information about the operation to be performed and the operands involved.

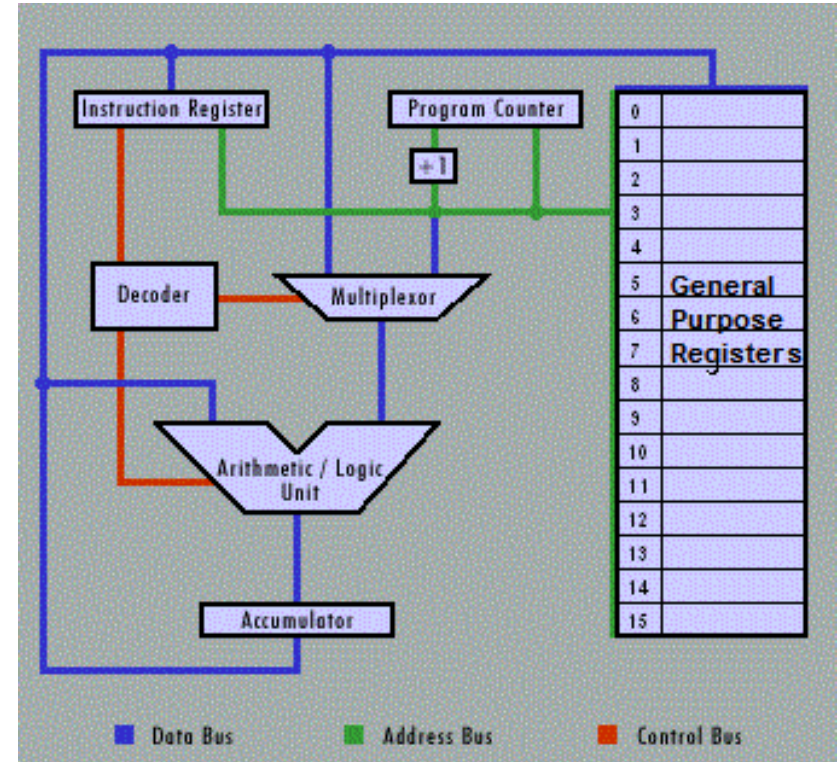
- **Execute**

- The CPU carries out the decoded instruction, performing the specified operation on the data.
- This stage may involve arithmetic or logical operations, data movement, control transfer, or other actions depending on the instruction.



# CPU architecture

- The PC indexes into the reg file and copies the next instruction from memory into the IR.
- The Instruction is decoded and the correct operation and data (or address parts depending on the ) are sent to the ALU
- The ALU will compute an output and store it in a register



# Machine code and assembly intro

**Machine code** are the  
1s and 0s

000 001 011
111 010000



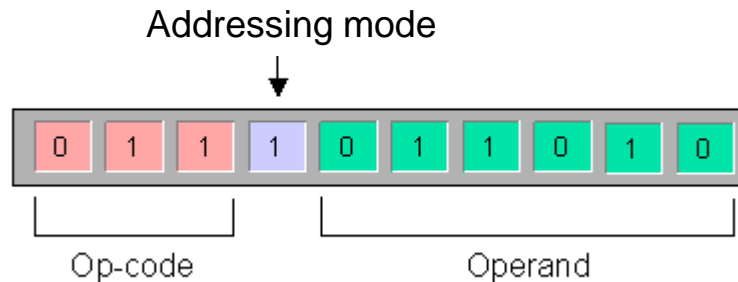
**Assembly** is the human readable  
version of the 1s and 0s

ADD R1, R3
JMP 32

The set of instructions available is called  
the **Instruction Set Architecture (ISA)**

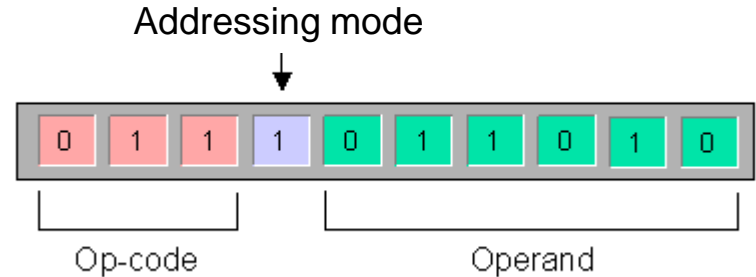
# Decoding Machine code instruction

- The bit pattern appearing in the op-code field indicates which of the elementary operations, such as STORE or JUMP, is requested by the instruction.
- The bit patterns found in the operands field provide more detailed information about the operation specified by the op-code.
- Another bit specifies whether the data in the operand should be treated as a register or an immediate value



# “addressing mode” bit

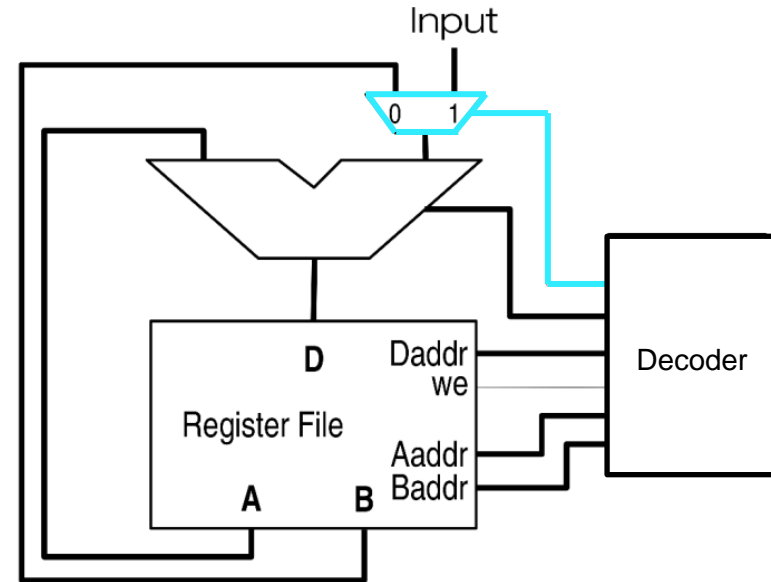
- Some architectures use an “addressing mode” bit, and others don’t.
- In this architecture, if addressing mode bit = 1, treat the operand as a **value**
- If the addressing bit = 0, treat the operand as a **register number**



<i>Machine code</i>	<i>Assembly code</i>	<i>Description</i>
011 1 000010	ADD #2	Load the <b>value 2</b> into the Accumulator
010 0 001101	SUB 13	Subtract <b>register 13</b> from the accumulator

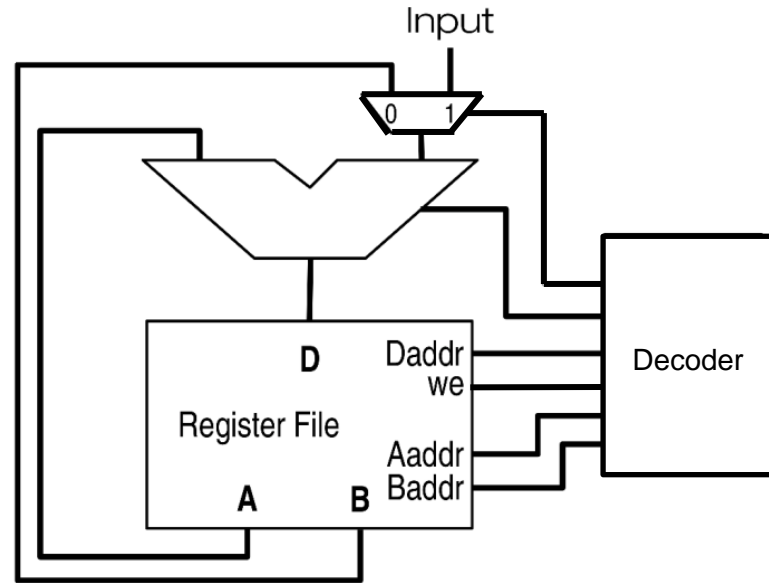
# Immediate input

- Let's assume I have this configuration:
- Let's also assume this decoder has access to all parts of a machine instruction:
  - a) the opcode
  - b) the addressing mode
  - b) the operand(s) [one of which is the destination]
- How does the addressing bit control immediate or non-immediate instructions?
- Use a MUX to control input.
- When MUX **select** = 1, choose immediate input
- When MUX **select** = 0, choose B output of register file



# Immediate input

- Let's assume I have this configuration:
- Let's also assume this decoder has access to all parts of a machine instruction:
  - a) the opcode
  - b) the addressing mode
  - b) the operand(s) [one of which is the destination]
- How does the addressing bit control immediate or non-immediate instructions?
- Use a MUX to control input.
- When MUX **select** = 1, choose immediate input
- When MUX **select** = 0, choose B output of register file

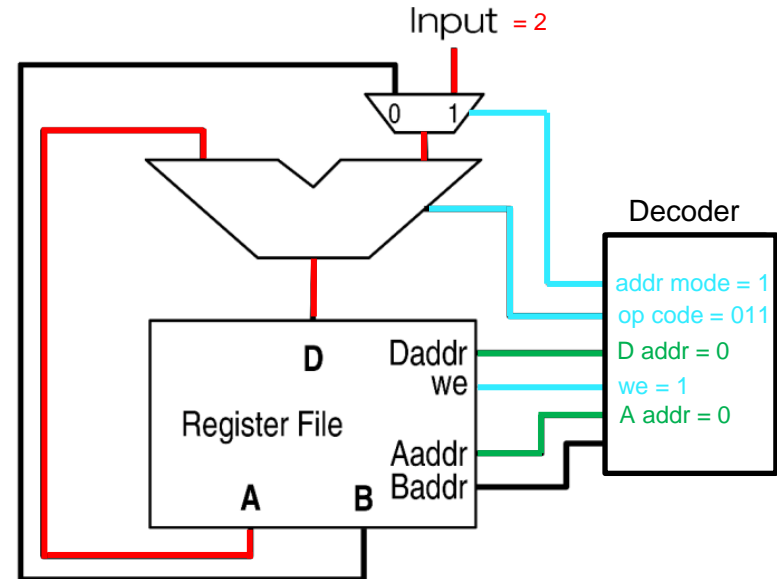


# Immediate input (cont.)

- Given this instruction:

Machine code	assembly	Meaning
011 1 000 010	ADD R0, #2	R0 = R0 + 2

- Input = 10 (immediate value 2)
- op code = 011(add)
- addr mode = 1
- D addr = 0 (reg 0)
- we = 1 (write enable because we are storing val in R0)
- A addr = 0 (reg 0)
- B addr = <nothing assigned>



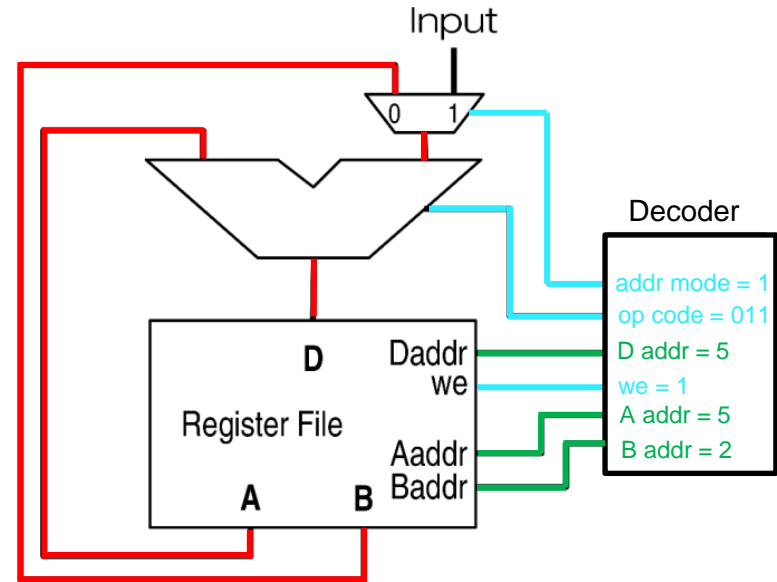
Data, Address, Control

# Immediate input (cont.)

- Given this instruction:

Machine code	assembly	Meaning
011 0 101 010	ADD R5, R2	$R5 = R5 + R2$

- op code** = 011(add)
- addr mode** = 0
- D addr** = 101 (R5)
- we** = 1 (write enable because we are storing val in R5)
- A addr** = 101 (R5)
- B addr** = 10 (R2)



Data, Address, Control

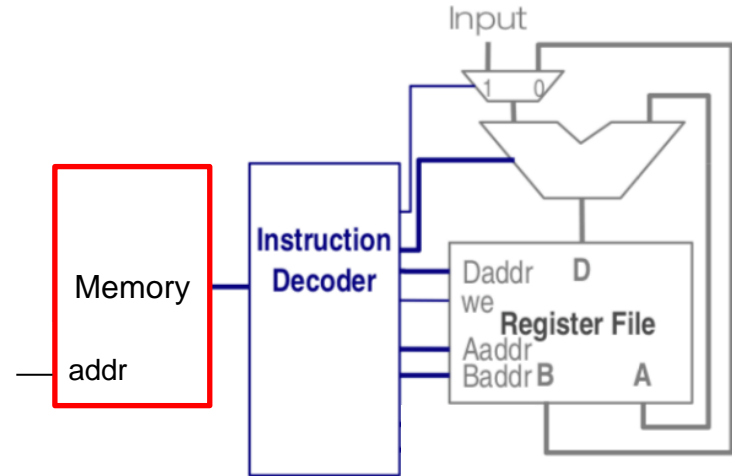


# Memory

- We can use memory to store the instructions that make up the program
- A memory is a set of storage elements with a read port to access the stored values and a way to initialize or write the values.
- Random Access Memory (RAM) has both read and write ports
- Read-Only Memory (ROM) had a read port and some way of initializing its contents
- ROM and RAM can be designed in a similar way to the register file (with a multiplexer).

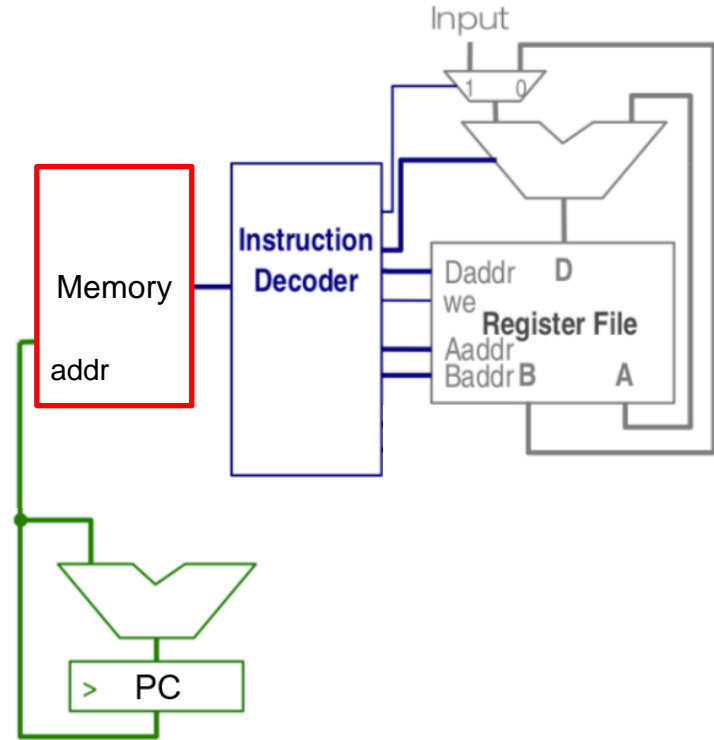
# Instruction Memory

- We can use memory to hold the processor instructions.
  - The memory would output the machine code instruction stored at the given address
  - How do we know what address to read the instruction from?
  - Use the PC



# Program counter (PC)

- A program counter is then used to provide the address of the instruction being executed.
- The PC is incremented on each clock edge.
- The combination of the PC and memory provides a generic control unit.
  - However, it is limited to repeatedly executing all stored instructions in memory.
- We could introduce a jump instruction (JMP) which could add some flexibility
  - A JMP instruction would set the PC to a different target point in the program
  - JMP 12 sets the PC = 12




# Jump (concept)

- Jumping allows you to change what instruction you are currently on.
- We can jump by modifying the Program Counter, so the next instruction is something other than the “next instruction”.
- Examples:

Simple Program

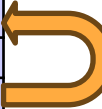
Inst#	assembly
0	Mov R0, PC
1	Add R0 #32
2	<b>JMP #5</b>
3	ADD R0, #1
4	MOV R1, R2
5	MUL R1, R0



Instructions 3 and 4 will always be skipped

Simple Program

Inst#	assembly
0	Mov R0, PC
1	Add R0 #32
2	ADD R0, #1
3	<b>JMP #1</b>
4	MOV R1, R2
5	MUL R1, R0



You can also use jumps to make loops

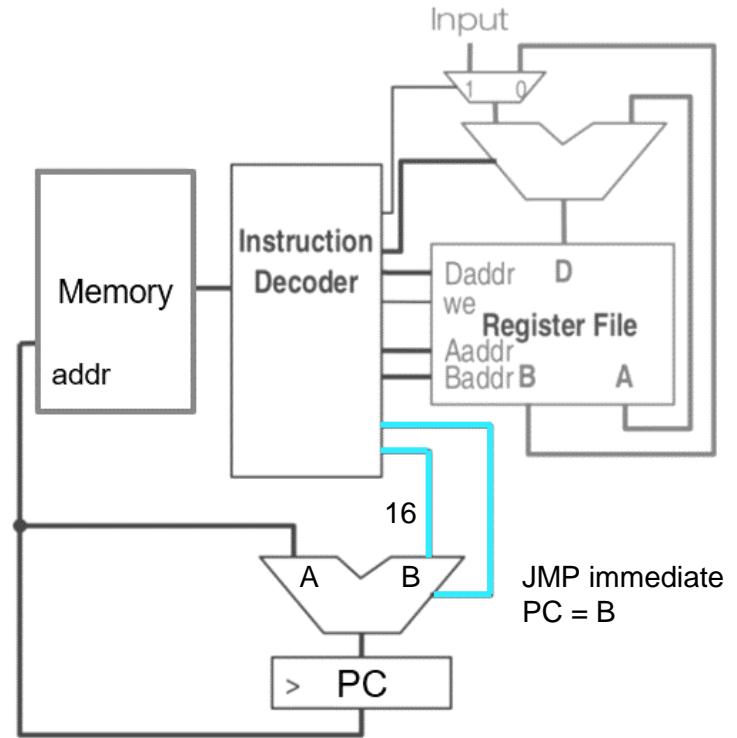
# Jump Implementation

- How would I design a circuit to change the PC to a different value?

Note: Made up machine instructions

Machine code	assembly	Meaning
110 010000	JMP #16	PC = 16

- I need another ALU that allows several type of jumps (we'll see some in a moment)
- Remember an ALU just performs variable operations based on an op code



# Flags

- Information about the last operation performed is stored as a **flag**.
- A flag is a 1 bit values that indicate a true or false statement about a past operation. Some examples include:
  - C = carry out, 1 if the last computation had a carry bit
  - Z = zero, 1 if the last computation resulted in a zero
  - OV = overflow, 1 if the last computation overflowed
  - N = negative, 1 if the last computation resulted in a negative number
- Sometime a flag has no meaning for an operation, i.e. JMP does not generate any flags, XOR generates Z & N flag, but not C & OV.
- Since these values are only 1 bit, some processors may combine each of these values into the same register. This register is sometimes called the status register (SR).

# Jump

- In C++ I can use if else statements to change my “location” in code, so I don’t have to run every instruction

```
if ( a > b ) {  
    foo()  
} else {  
    bar()  
}
```

- I can do the same thing in assembly using flags.
- First I perform  $c = a - b$ , then check the Negative and zero flag.
  - c negative:  $a < b \rightarrow N = 1, Z = 0$
  - c positive:  $a > b \rightarrow N = 0, Z = 0$
  - c zero:  $a = b \rightarrow Z = 1$ , Note that it doesn’t matter what N flag is equal to.

# Conditional Jump

- Therefore, I need to break every conditional jump into at least 2 instructions

C++

```
if ( a > b ) {  
    foo()  
} else {  
    bar()  
}
```

Instruction level code

```
1. SUB b, a  
2. BNC #5  
3. foo()  
4. JMP #6  
5. bar()
```

- Where SUB is the subtract instruction and BNC is branch (jump) if negative flag = 0.
- We call conditional jumps **branches**, since more than one path in the code can run depending on a condition.



# Branch names

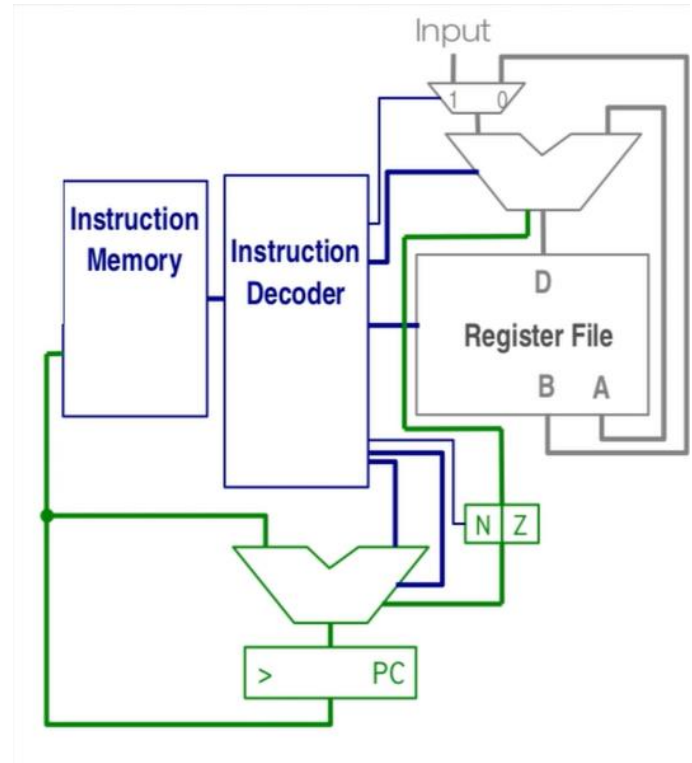
- Some example branch instruction shorthand we'll use is
  - BNC: Branch negative clear
    - Ex: BNC #12 means “if N flag = 0, PC = 12. else PC = PC +1”
  - BNS: Branch negative set
    - Ex: BNS #12 means “if N flag = 1, PC = 12. else PC = PC +1”
  - BCC: Branch carry clear
  - BCS: Branch carry set
  - BZC: Branch zero clear
  - BZS: Branch zero set

# Jump Conditions

- Other JMP type instructions like BZS (branch zero set) will jump only if the value of the previous operation is zero.

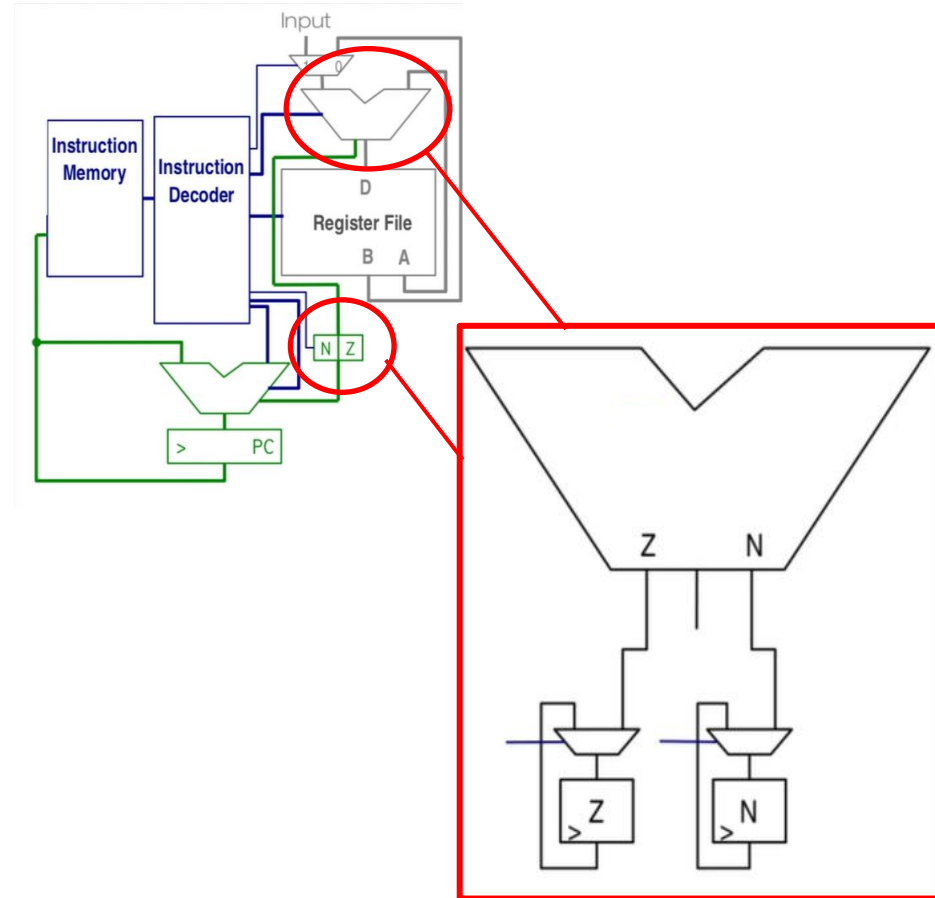
Note: Made up machine instructions

Machine code	assembly	Meaning
110 1 010000	BZS #32	If Z flag = 1: PC = 32 else: PC = PC + 1

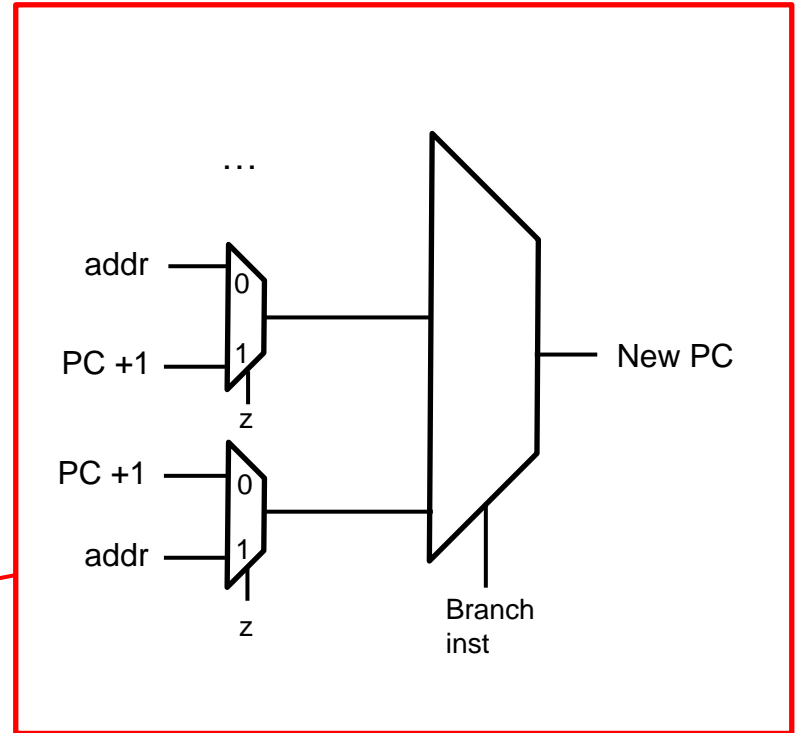
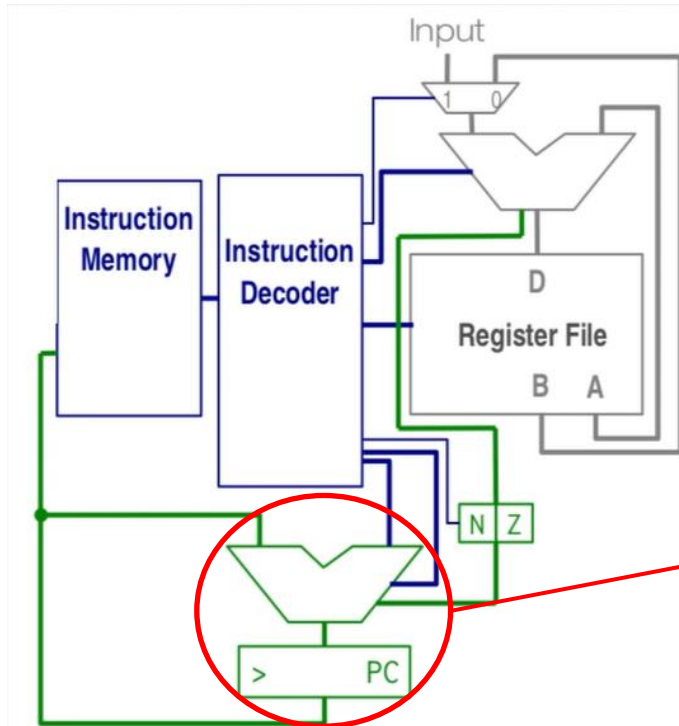


# Collecting Flags

- If an instruction doesn't generate a useful flag, we want to preserve the flag from a previous operation.
- Example: if an JMP operation happens, I don't want to edit Z or N, I want their values preserved.
- In implementation, I can use a MUX to guard values in our flip flop/register and select either the previous value or the new value



# Branch implementation



# Other operation patterns

- Load from memory
  - A value is copied from Memory to a register
  - Usually two operands: **memory addr** & **reg**
- Store to memory
  - A value is copied from a register to memory
  - Usually two operand: **reg** & **memory addr**
- Load from immediate
  - A value is copied directly from the machine instruction to a register
  - Two operands: **immediate value** & **reg**
- Mov from reg to reg
  - Two operands: **regA** & **regB**
  - Can be implemented as  $\text{regB} = \text{regA} + 0$  to save dedicated data wires. However, some flags will be triggered by add, but not by Mov

# References

- <https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html>
- <https://wou.edu/las/cs/csclasses/cs160/VTCS0/MachineArchitecture/Lessons/CPU/index.html>
- Ivan Sekyonda's slides