CS 240 Laboratory 5
Instruction Set Architecture and Microarchitecture

In computer science, an **Instruction Set Architecture** (**ISA**) is an abstract model of a computer.

At a minimum, the ISA specifies:

- **size of address bus:** specifies the size of the memory in the computer (and, therefore, the number of address bits required),
- **size of data bus:** specifies the size of an instruction and the size of a standard data value stored in a register (and, therefore, the **number of data bit**s required),
- the **number of registers** in the CPU: used to store values when executing instructions, and
- the **formats and meaning of the instructions** which can be executed in the CPU.

A **microarchitecture** is a concrete implementation of an ISA.

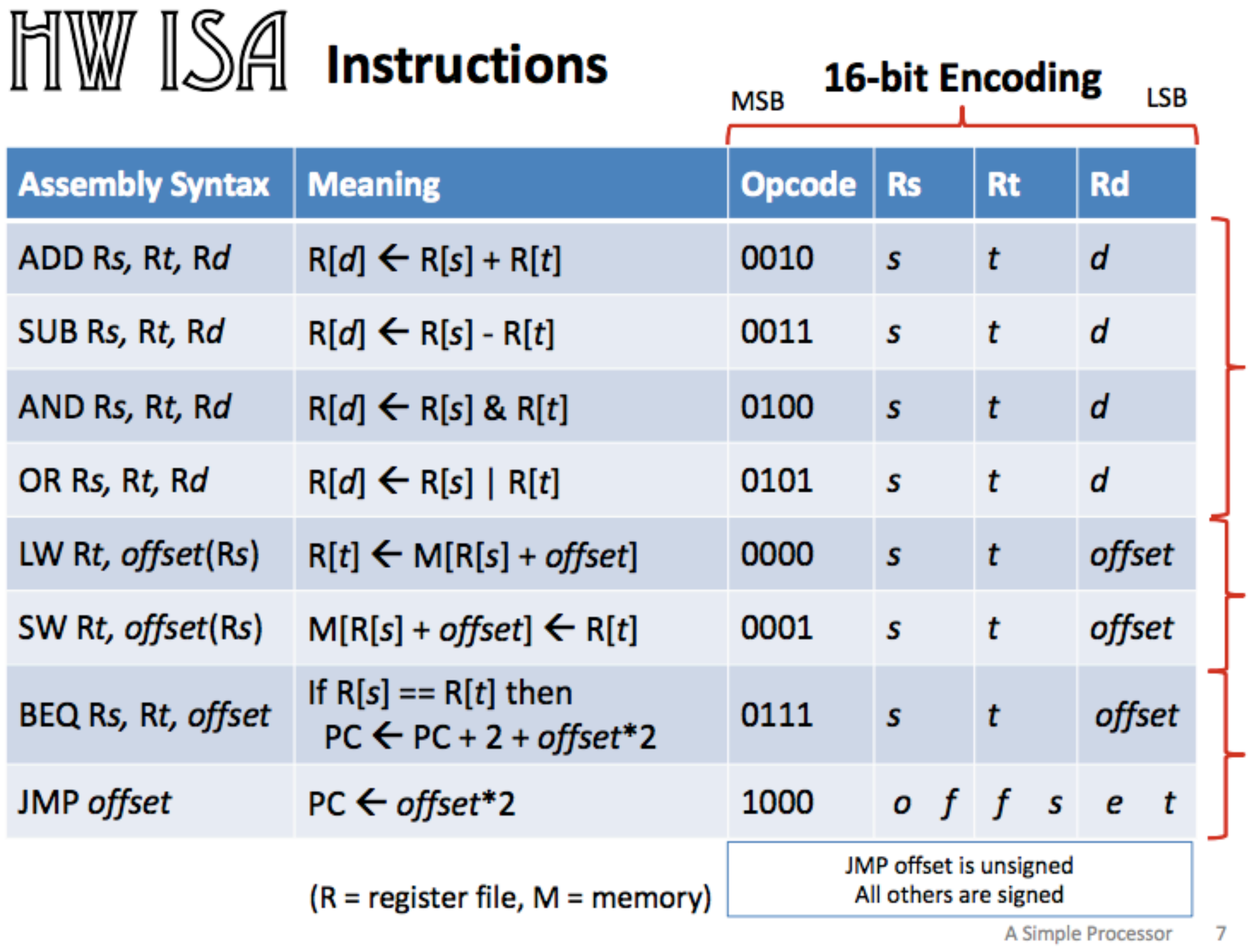
We will define an ISA for a machine we call the **HW** machine, and we will then use LogicWorks to implement a working microarchitecture that we can experiment with and test.

HW Instruction Set Architecture

- **8 bit address bus** so, instruction memory has $2^8 = 256$ bytes

- **16 bit data bus** so, each instruction is 16 bits (2 bytes), and each register will hold 16 bits of data

- **16 registers**
  R0 = 0 (constant)
  R1 =1 (constant)
  R2-R15 general purpose



HW ISA Instructions

16-bit Encoding

MSB LSB

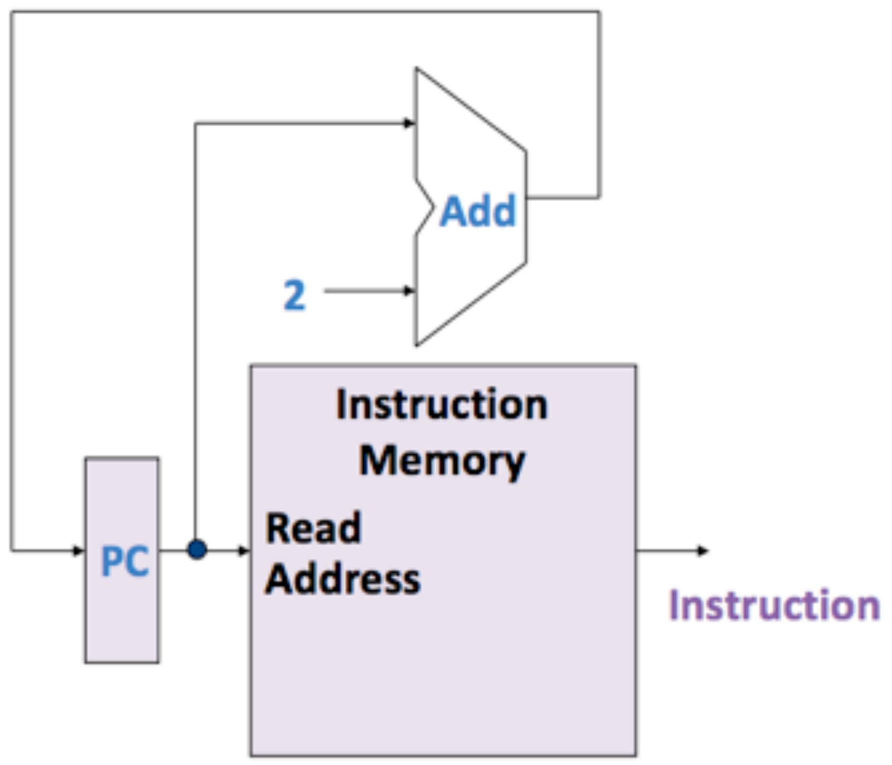
Assembly Syntax	Meaning	Opcode	Rs	Rt	Rd
ADD Rs, Rt, Rd	R[*d*] ← R[*s*] + R[*t*]	0010	*s*	*t*	*d*
SUB Rs, Rt, Rd	R[*d*] ← R[*s*] - R[*t*]	0011	*s*	*t*	*d*
AND Rs, Rt, Rd	R[*d*] ← R[*s*] & R[*t*]	0100	*s*	*t*	*d*
OR Rs, Rt, Rd	R[*d*] ← R[*s*] | R[*t*]	0101	*s*	*t*	*d*
LW Rt, *offset*(Rs)	R[*t*] ← M[R[*s*] + *offset*]	0000	*s*	*t*	*offset*
SW Rt, *offset*(Rs)	M[R[*s*] + *offset*] ← R[*t*]	0001	*s*	*t*	*offset*
BEQ Rs, Rt, *offset*	If R[*s*] == R[*t*] then PC ← PC + 2 + *offset**2	0111	*s*	*t*	*offset*
JMP *offset*	PC ← *offset**2	1000	*o f*	*f s*	*e t*

(R = register file, M = memory)

JMP offset is unsigned
All others are signed

Fetch Instruction from Memory

Remember the fetch instruction circuit from lecture?



Programs are stored as instructions in memory. The first instruction in the program is assumed to start at address 0 in memory.

The **Program Counter (PC)** register holds the address of the currently executing instruction.

The **PC** is initialized to 0 by a **reset** to begin execution of the program.

1. The instruction in memory at the address from **PC** is fetched (read at the outputs of the memory). It is then sent to the **CPU** to be executed.
2. The **PC** is incremented by 2.

Steps 1 and 2 are repeated until all instructions are executed.
This model assumes that all instructions in a program are always executed in sequential order.

However, the instruction in programs do not always execute in sequential order!

You must sometimes skip some instructions (like when you have to go to the **else** clause of an **if** statement, you skip the **if** clause). For example:

```
if (x < 0)
    y = 2;    // skip this statement when the condition is not met
else
   y = 3;
```

Or, you may need to go back to an earlier instruction (like when you get to the end of **loop**, you have to get back to the beginning of the loop to repeat it).

```
while (x > 0) {
   y = y+1;
   x = x – 1;     //after you execute this statement, you must
                 // go back to the beginning of the loop to test if
                  // the loop condition is still true to repeat the loop
}
```

BEQ (Branch-if -EQual)

In our **HW** machine, the **BEQ (Branch-if-EQual)** instruction allows us to *conditionally* change the sequential order of execution of instructions:

BEQ Rs Rt offset

This instruction compares the contents of **Rs** and **Rt** and calculates a new address to branch to if the registers' contents are equal.

The branch address is calculated relative to the address of the next sequential instruction in memory:

PC + 2

The **offset** is the number of instructions away that the branch address is from the usual next sequential instruction.

The **offset** is a *signed* value (it can be either positive or negative), and is specified as a 4-bit value. So, it can range from -8 to +7 in our **HW** machine.

Since each instruction is two bytes, the offset is multiplied by 2 and added to the address of the next sequential instruction to calculate the branch address. This formula can be expressed as:

PC + 2 + (2*offset)

For example, suppose you have executed the first few instructions of a program and the next ones are:

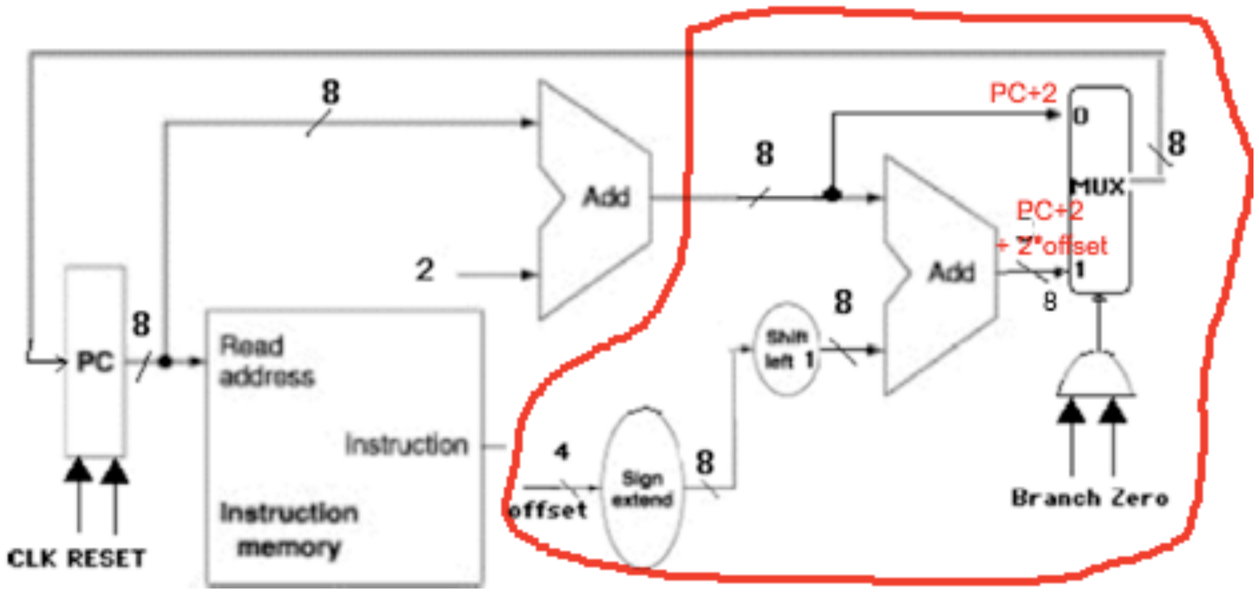
Address(from PC)	**Instruction at Address**
6:	**BEQ R3 R0 1**
8:	ADD R1 R2 R2
10:	AND R0 R0 R4

When executing **BEQ R3 R0 1**, **PC** contains address **6** and **offset** = 1.

If, at that point, **R3** contains a 0, then **R3** and **R0** contain equal values, and so the next value of the **PC** will be 6 + 2 + (2*1)= 10. Program execution will skip the instruction at address 8 and go straight to 10.

However, if **R3** does not contain a 0, the program will simply progress to the next address in memory, **PC + 2**, which will be 6 + 2 = 8.

We add the following logic (circled in red) to our instruction fetch circuit to implement **BEQ:**



The new **adder** computes **PC + 2 + 2*offset**.

The **MUX** (multiplexer) at the top right selects either:

PC+2 (address of next instruction in memory, from first adder), or
PC + 2 + 2*offset (branch address, from second adder)

to send back to the inputs of the **PC** as the address in memory of the next instruction to be executed.

The MUX selects which address to use based on two control bits, **Branch** and **Zero**.

When a **BEQ** instruction is executed, the ALU performs a subtraction (contents of **Rs** – contents of **Rt).**

If the two values are equal, the result is 0 (the **Zero** bit in the ALU is set).

The **Branch** bit simply indicates that a **BEQ** instruction is being executed (other instructions may also set the **Zero** bit in the **ALU**, so it is necessary that both **Branch** = 1 and **Zero** = 1 for the branch address to be selected.

If the values are not equal, the program simply continues execution at the next address in memory (**PC+2**).

We will construct this circuit in LogicWorks to verify correct operation.

JMP instruction

In the HW computer, there is also an unconditional branch instruction called **JMP** (Jump):

JMP offset

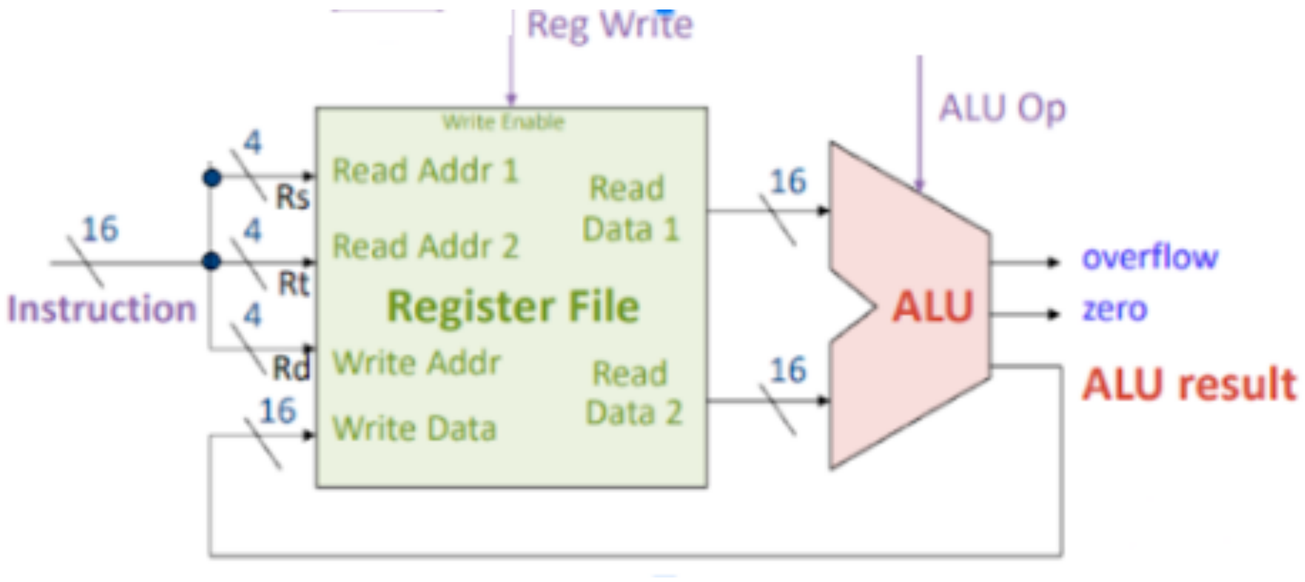
Here, the **offset** is a 12-bit value which specifies the number of instructions from the beginning of the program to jump to:

PC = **offset * 2**

For example, **JMP 3** sets the **PC** to 6, causing the instruction stored at address 6 (*i.e.*, the 3rd instruction in the program) to be executed next.

Basic Datapath

The following diagram describes the basic datapath for executing the arithmetic and logic instructions inside the CPU. It consists of a Register File and an ALU (you have studied both devices in previous labs).



This circuit can execute the arithmetic and logical operations including **ADD,SUB,AND,** and **OR**. The general form is:

op Rs Rt Rd

- read contents of **Rs** and **Rt** from Register File at **Read Data 1** and **Read Data 2** outputs
- perform an operation in the **ALU** on the contents of the registers (the **ALUOp** bits control which operation)
- write the **ALU result** back to register **Rd** in register file

The **ALUOp** bits are control signals that specify **AINV, BNEG**, and a 2-bit operation (00 = AND, 01=OR, 10=+). Remember those from our study of the ALU?

The **RegWrite** control signal indicates whether a particular instruction results in a change to a register (some instructions do not change any registers).

The **ALUop** and **RegWrite** control lines are functions of the **opcode**, so there is some simple logic which can be used to produce these signals.

The following table shows the opcode for each instruction, and the associated control signals **ALUop** and **RegWrite** to be used for each .

Instruction	**Opcode**	**ALUop**				**RegWrite**
		Ainv	**Bneg**	**Op1**	**Op0**	
ADD	0010	0	0	1	0	1
SUB	0011	0	1	1	0	1
AND	0100	0	0	0	0	1
OR	0101	0	0	0	1	1
BEQ	0111	0	1	1	0	0
JMP	1000	don't care				0

BEQ is accomplished by subtraction (which sets the **Zero** bit).

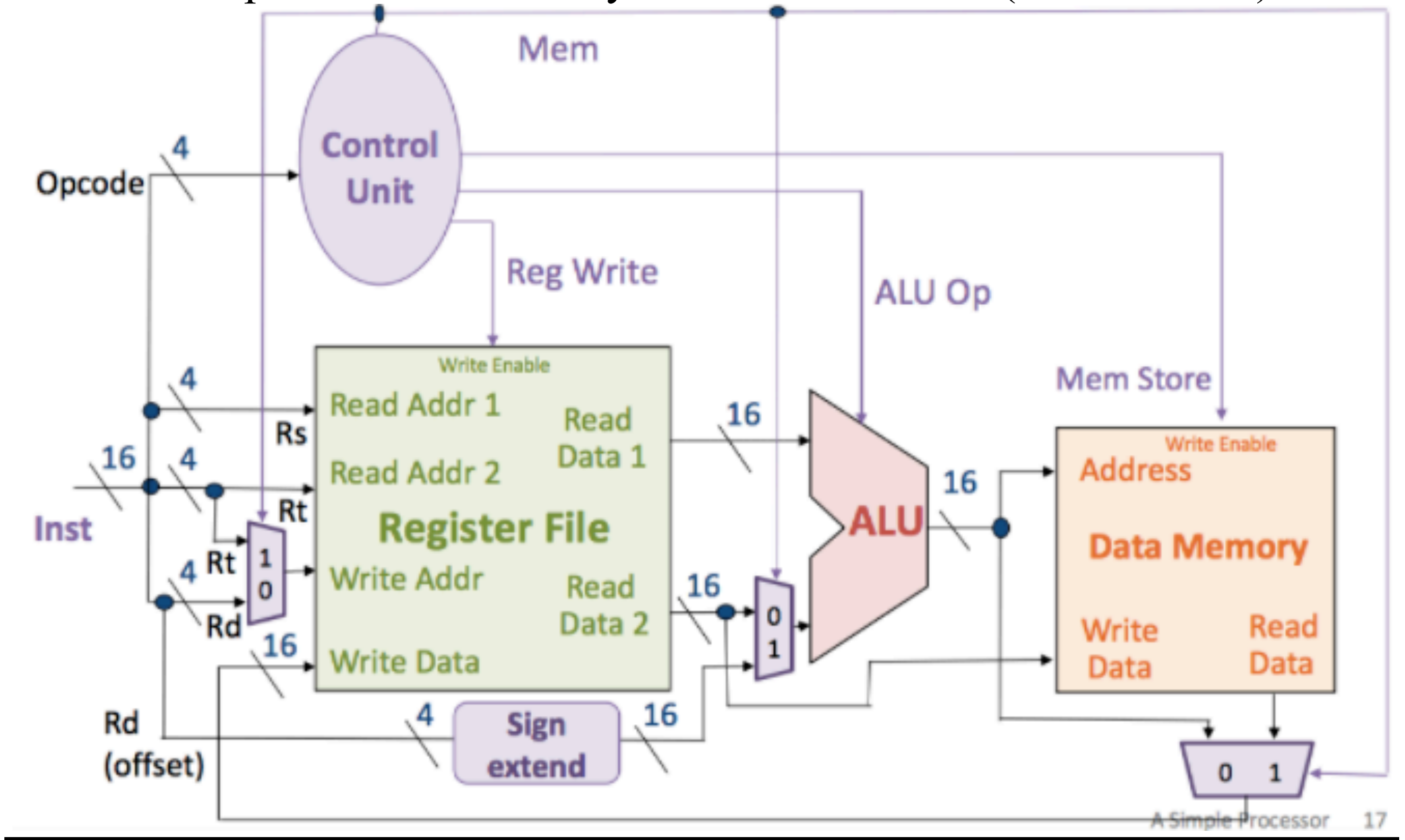
BEQ and **JMP** do not change the value of a register.

JMP does not even use the **ALU** (it is implemented as part of the instruction fetch circuit)

In lab, you will connect the Register File to the ALU, and test its operation.

Full Datapath with Memory Instructions

The following diagram adds the logic and the Data Memory device needed to implement the memory access instructions (**LW** and **SW**):



2x4 MUX at the input of the Register File selects either:

- **Rd** gets written to for arithmetic/logic operations, or
- **Rt** gets written to for **LW**

2x16 MUX at the second input to the ALU selects second ALU input:

- from **Rt** for arithmetic/logic operations, or
- **offset** (sign-extended to 16 bits) for **LW** or **SW**

ADD Rs Rt Rd ALU: **Rs + Rt → Rd**

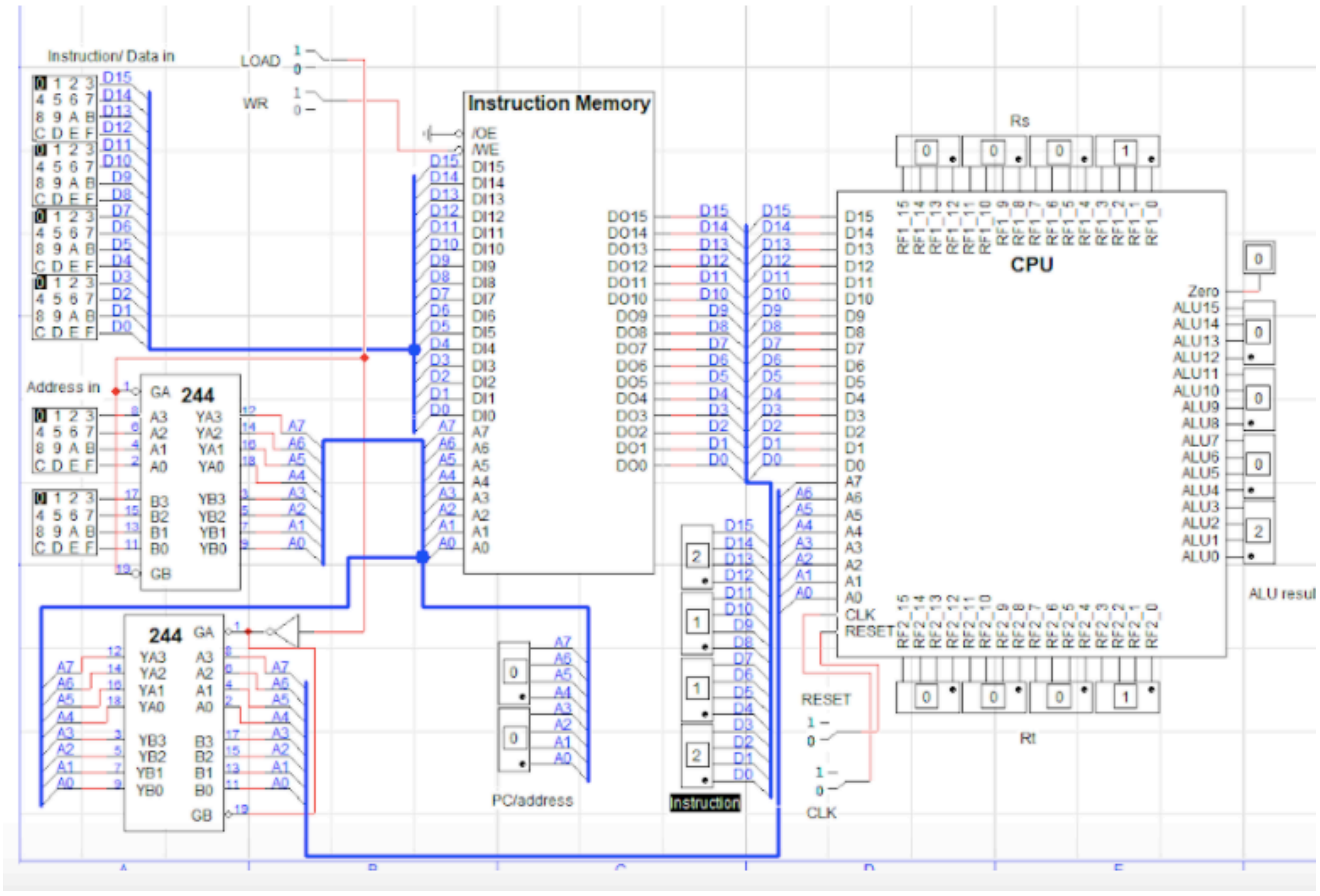
SW Rs Rt offset ALU: **Rs** + **offset → Address in Data Memory**
then value in **Rt** is stored to **Data Memory**

LW Rs Rt offset ALU: **Rs** + **offset → Address in Data Memory**
then value is loaded to **Rt** from **Data Memory**

Full Implementation

Finally, we abstract the **datapath** circuit into a **CPU** device, and connect it to our circuit for loading and fetching instructions from memory to execute a program.

You are given this circuit:



You will translate a short program from assembly language to machine code, load the instructions into memory, and then execute the instructions to run the program.