Computer System

Taimur Shahzad

Computer Components

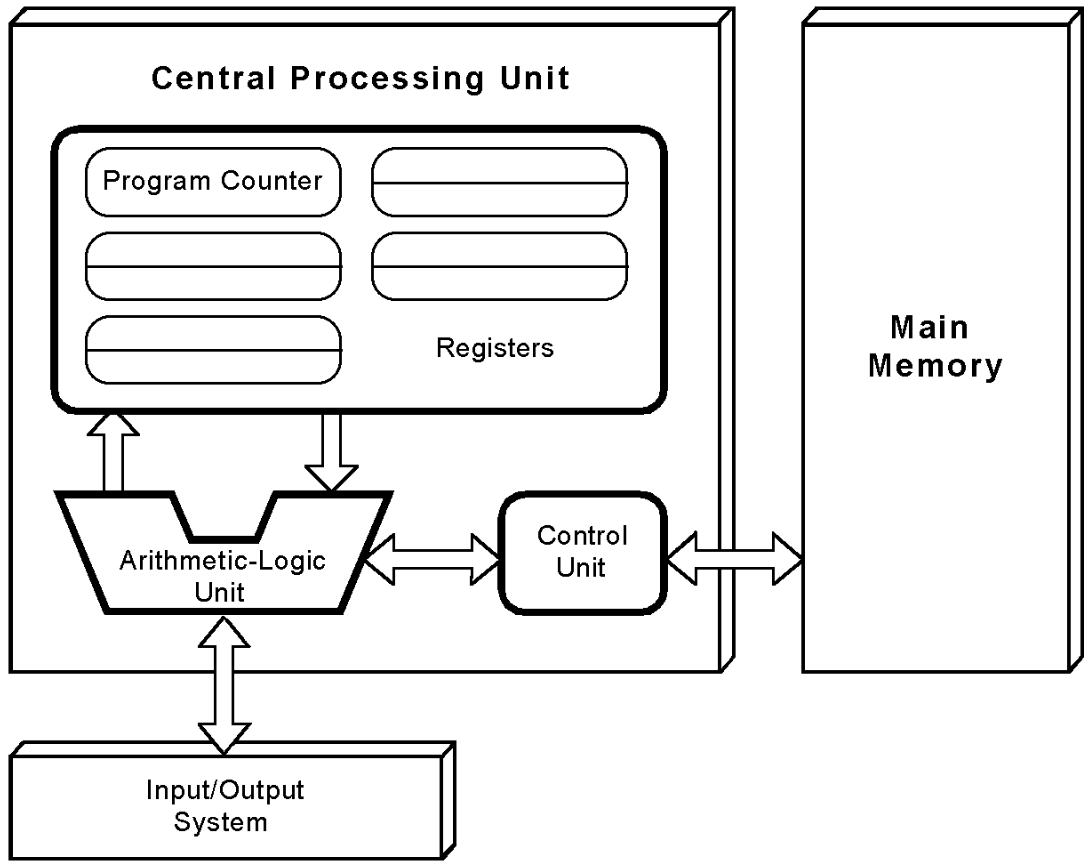
- Contemporary computer designs → Stored-program computers
    - *Based on von Neumann architecture*
        - Three key concepts:
            - Data and instructions are stored in a single read-write memory
            - The contents of this memory are addressable by location
            - Execution occurs in a sequential fashion (unless explicitly modified) from one instruction to the next

- Today's stored-program computers have the following characteristics:
    - Three hardware systems:
        - A central processing unit (CPU)
        - A main memory system
        - An I/O system
    - The capacity to carry out sequential instruction processing.
    - A single data path between the CPU and main memory.
        - This single path is known as the *von Neumann bottleneck*.

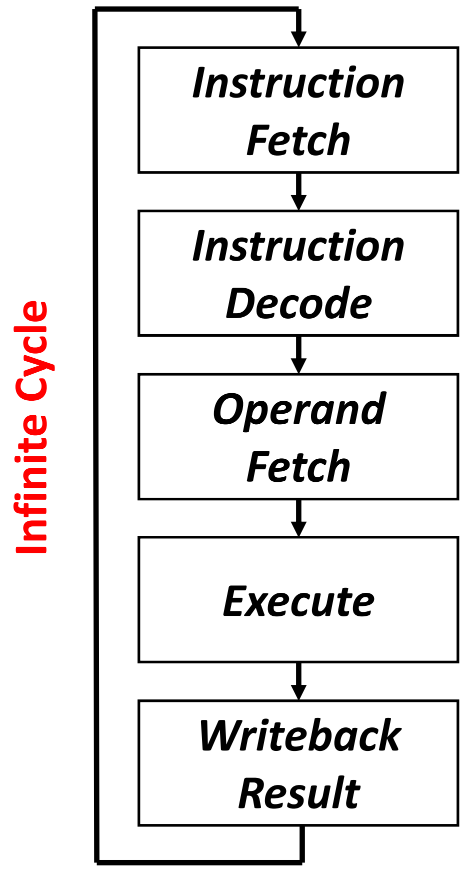
The von Neumann Model

- This is a general depiction of a von Neumann system:

- These computers employ a fetch-decode-execute cycle to run programs as follows . . .



Instruction Execute Cycle



Obtain instruction from program storage

Determine required actions and instruction size

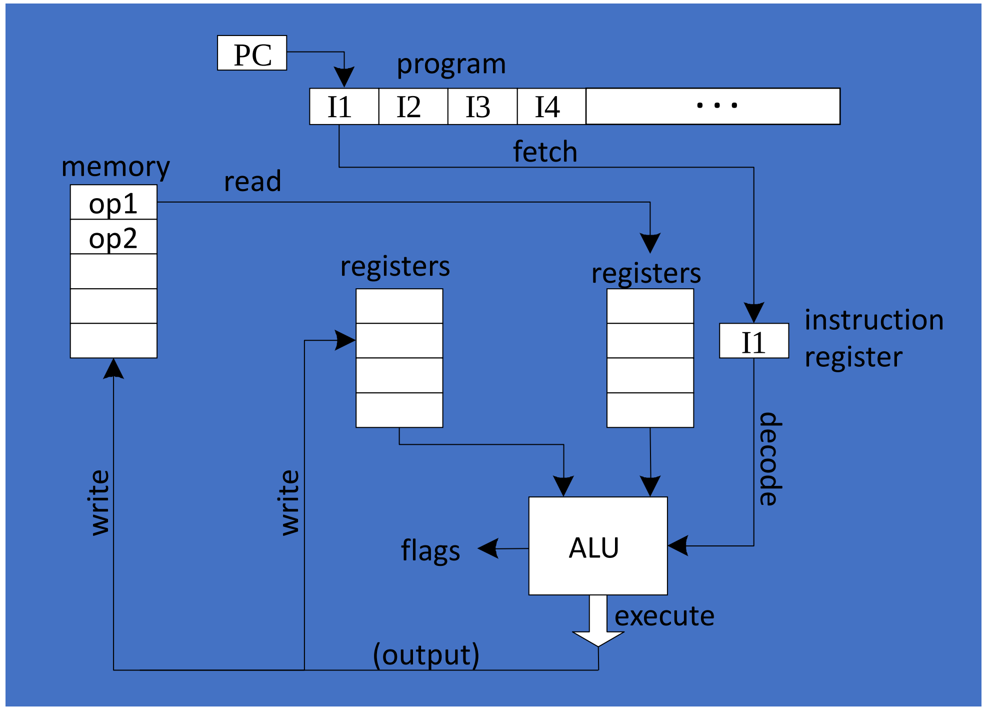
Locate and obtain operand data

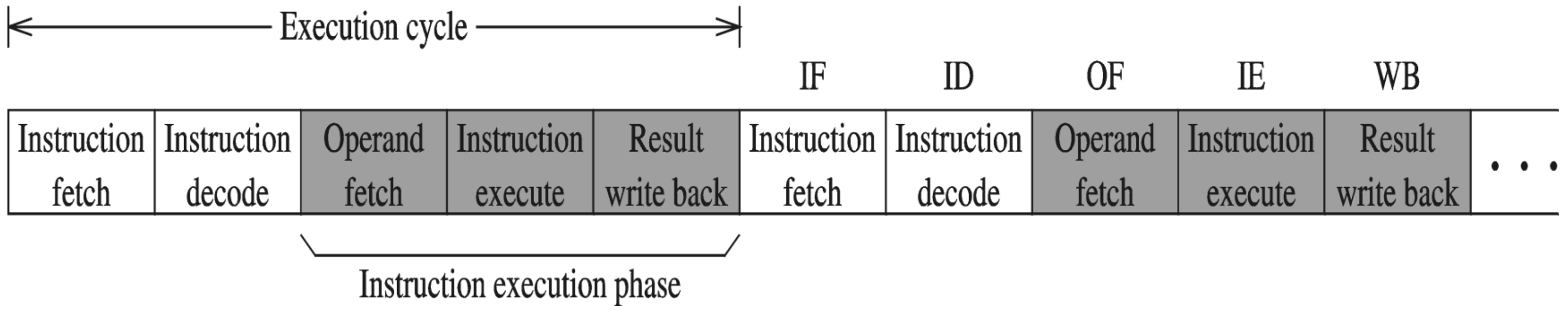
Compute result value and status

Deposit results in storage for later use

Instruction Execution Cycle – cont'd

- Instruction Fetch
- Instruction Decode
- Operand Fetch
- Execute
- Result Writeback





MARS (Machine Architecture Really Simple)

The MARS architecture has the following characteristics:

- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 4096 words of word-addressable main memory.
- 16-bit data words.
- 16-bit instructions, 4 for the opcode and 12 for the address.
- A 16-bit arithmetic logic unit (ALU).
- Seven registers for control and data movement.

MARS

MARS’s seven registers are:

- Accumulator, AC, a 16-bit register that holds a conditional operator (e.g., "less than") or one operand of a two-operand instruction.
- Memory address register, MAR, a 12-bit register that holds the memory address of an instruction or the operand of an instruction.
- Memory buffer register, MBR, a 16-bit register that holds the data after its retrieval from, or before its placement in memory.

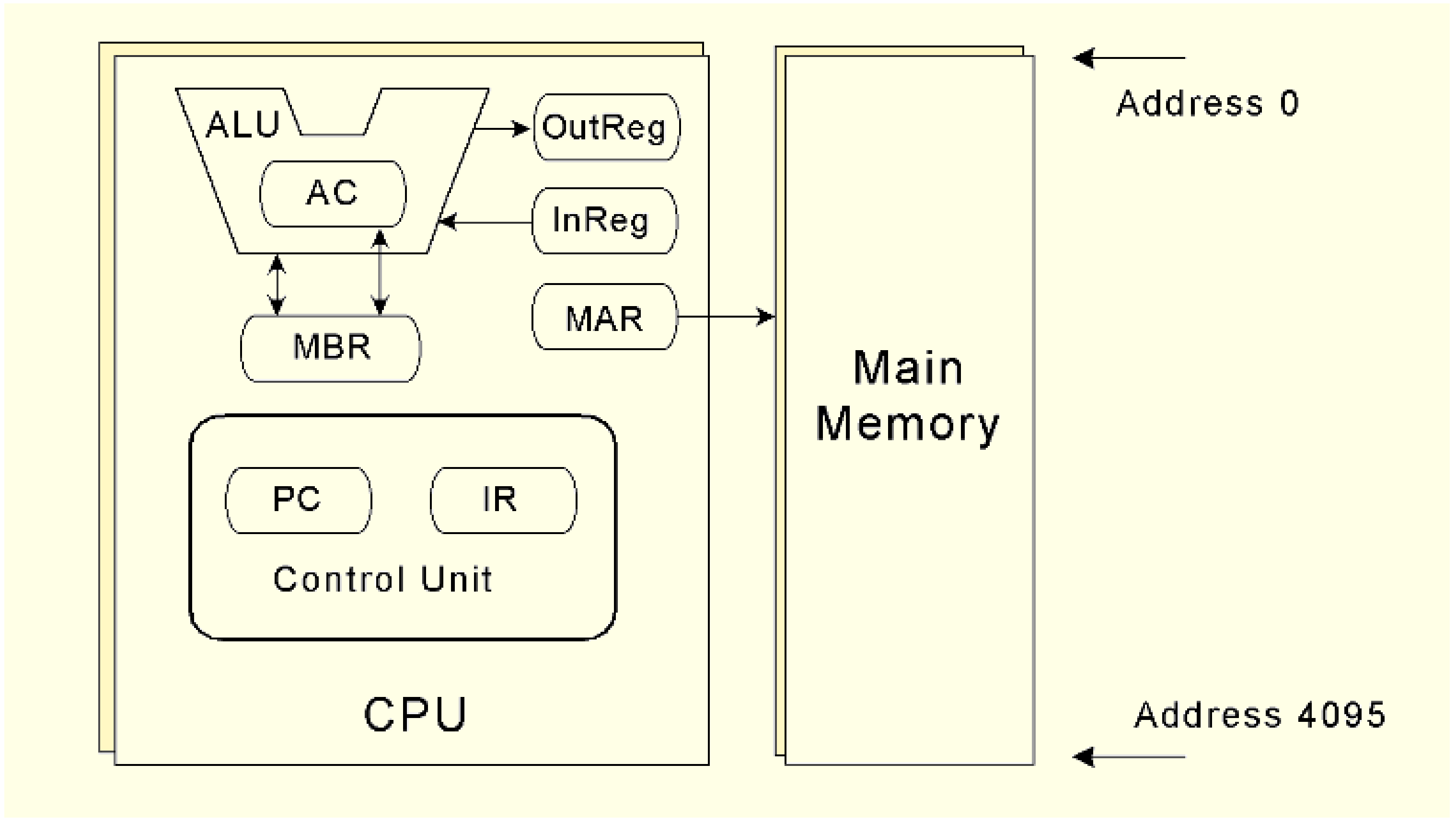
MARS

MARS’s seven registers are:

- Program counter, PC, a 12-bit register that holds the address of the next program instruction to be executed.
- Instruction register, IR, which holds an instruction immediately preceding its execution.
- Input register, h, an 8-bit register that holds data read from an input device.
- Output register, OutREG, an 8-bit register, that holds data that is ready for the output device.

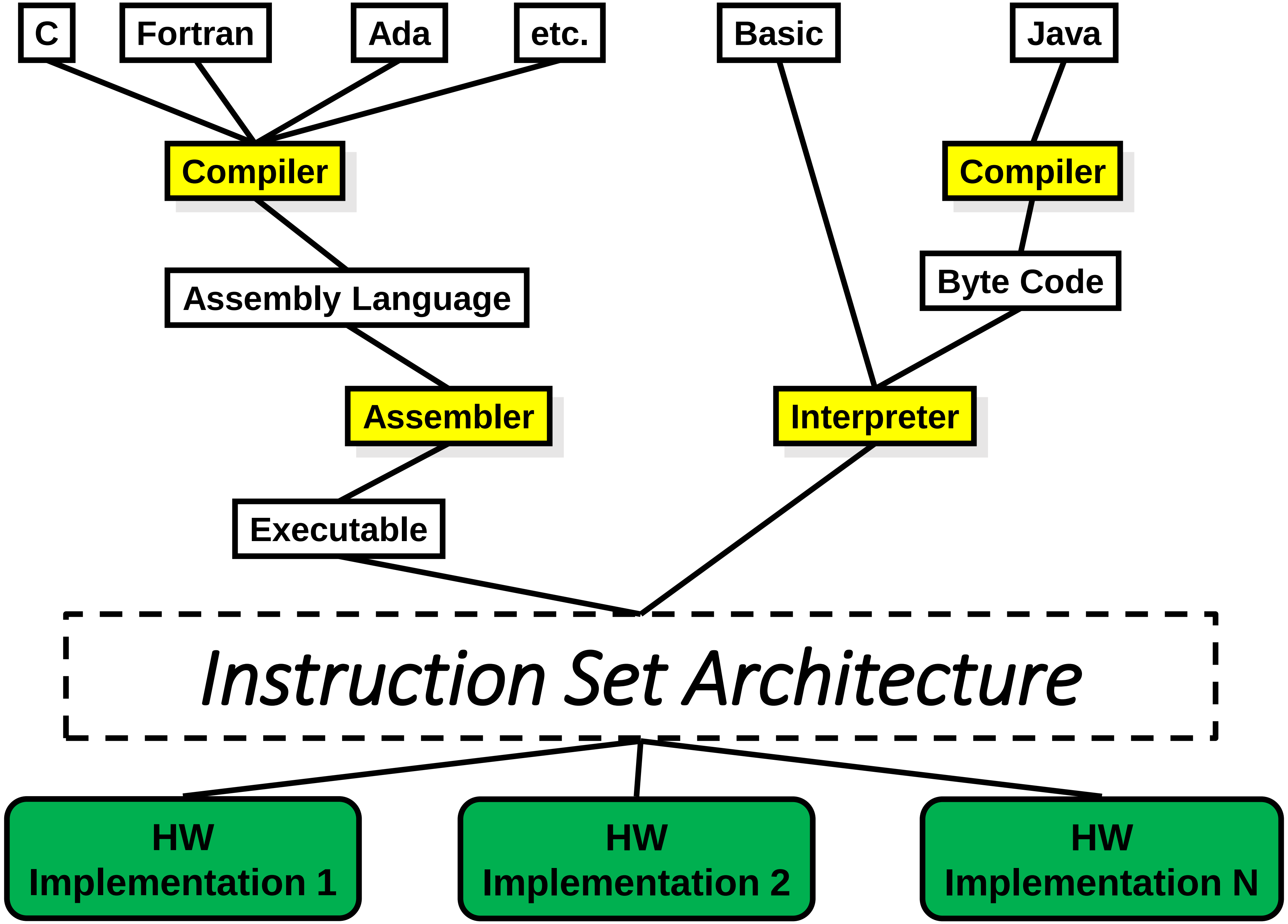
MARS

This is the MARS architecture shown graphically.



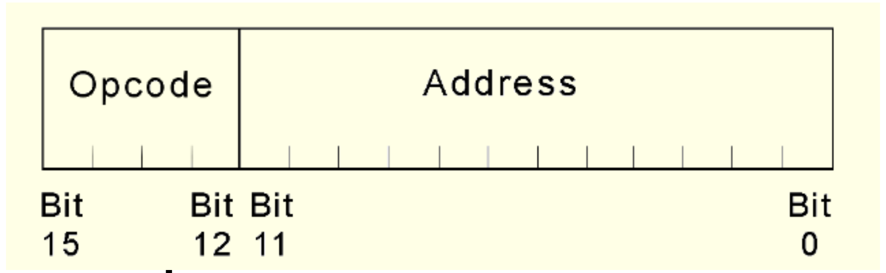
Instruction Set Architecture

- A computer’s instruction set architecture (ISA) specifies the format of its instructions and the primitive operations that the machine can perform.
- The ISA is an interface between a computer’s hardware and its software.
- Some ISAs include hundreds of different instructions for processing data and controlling program execution.
- The MARS ISA consists of only thirteen instructions.

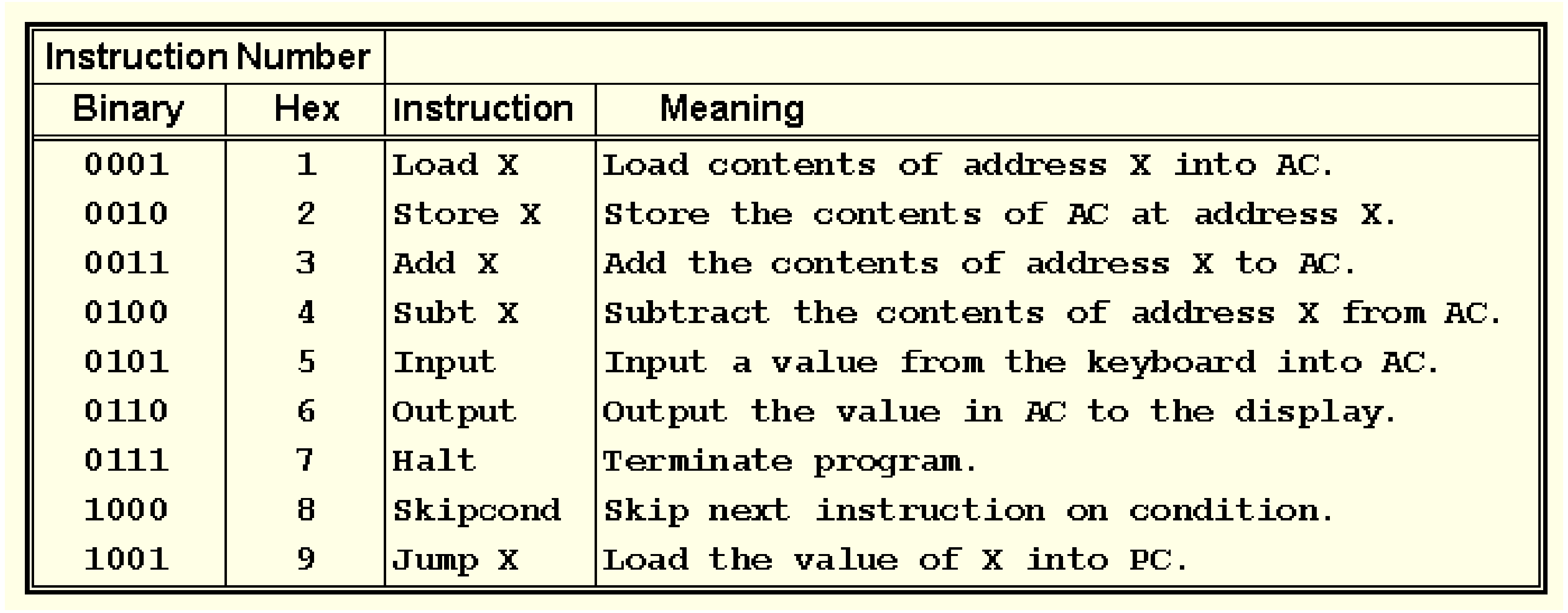

C
Fortran
Ada
etc.
Basic
Java
Compiler
Compiler
Assembly Language
Byte Code
Assembler
Interpreter
Executable
Instruction Set Architecture
HW Implementation 1
HW Implementation 2
HW Implementation N

MARS

- This is the format of a MARS instruction:



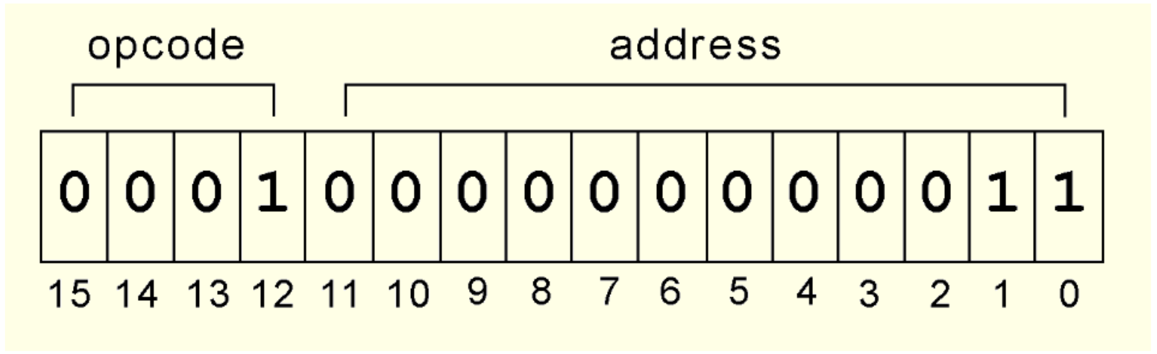
- The fundamental MARS instructions are:



Instruction Number			
Binary	Hex	Instruction	Meaning
0001	1	Load X	Load contents of address X into AC.
0010	2	Store X	Store the contents of AC at address X.
0011	3	Add X	Add the contents of address X to AC.
0100	4	Subt X	Subtract the contents of address X from AC.
0101	5	Input	Input a value from the keyboard into AC.
0110	6	Output	Output the value in AC to the display.
0111	7	Halt	Terminate program.
1000	8	Skipcond	Skip next instruction on condition.
1001	9	Jump X	Load the value of X into PC.

MARS

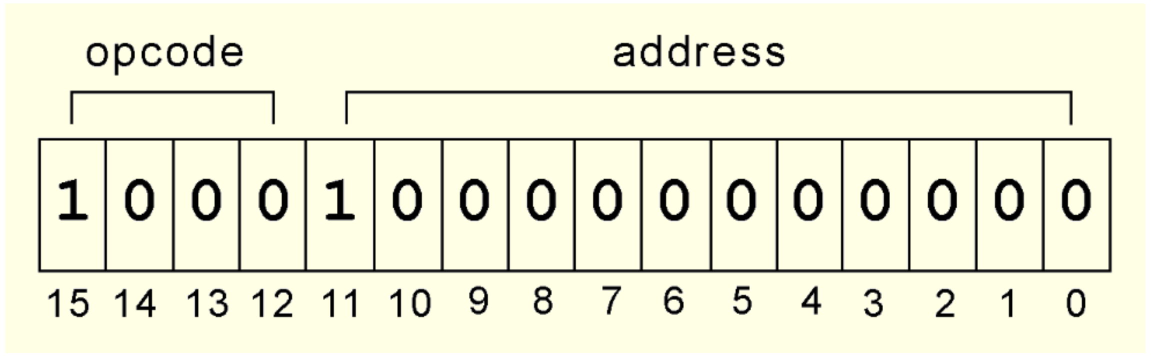
- This is a bit pattern for a **`LOAD`** instruction as it would appear in the IR:



- We see that the opcode is 1 and the address from which to load the data is 3.

MARS

- This is a bit pattern for a **SKIPCOND** instruction as it would appear in the IR:



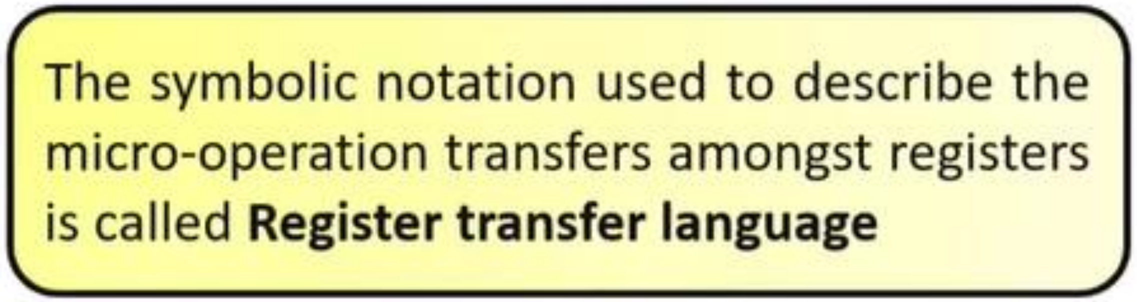
- We see that the opcode is 8 and bits 11 and 10 are 10, meaning that the next instruction will be skipped if the value in the AC is greater than zero.

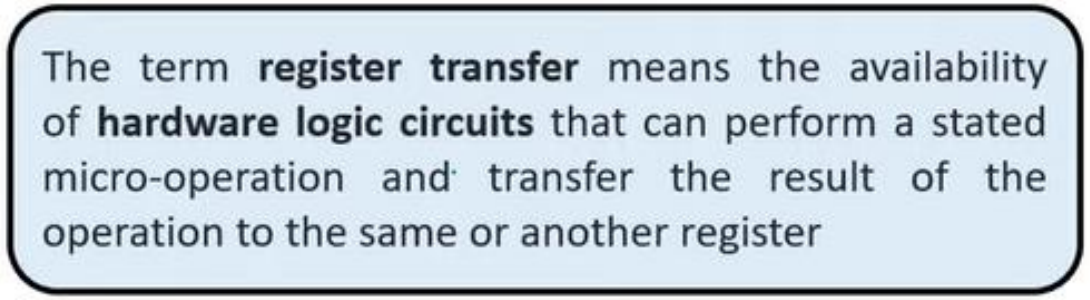
What is the hexadecimal representation of this instruction?

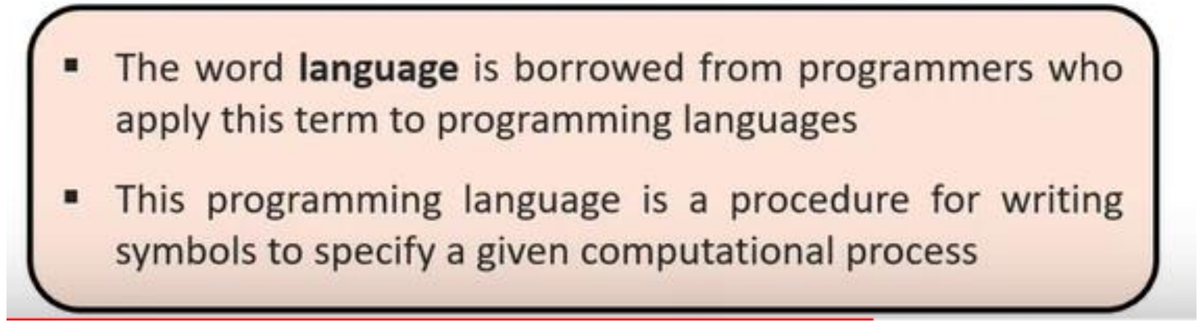
MARS

- Each of our instructions actually consists of a sequence of smaller instructions called *microoperations*.
- The exact sequence of microoperations that are carried out by an instruction can be specified using *register transfer language (RTL).*
- In the MARS RTL, we use the notation M[X] to indicate the actual data value stored in memory location X, and $\leftarrow$ to indicate the transfer of bytes to a register or memory location.

Register Transfer Language (RTL)


The symbolic notation used to describe the micro-operation transfers amongst registers is called **Register transfer language**


The term **register transfer** means the availability of **hardware logic circuits** that can perform a stated micro-operation and transfer the result of the operation to the same or another register


- The word **language** is borrowed from programmers who apply this term to programming languages
- This programming language is a procedure for writing symbols to specify a given computational process

MARS

- The RTL for the **LOAD** instruction is:

```
MAR ← X
MBR ← M[MAR]
AC ← MBR
```

- Similarly, the RTL for the **ADD** instruction is:

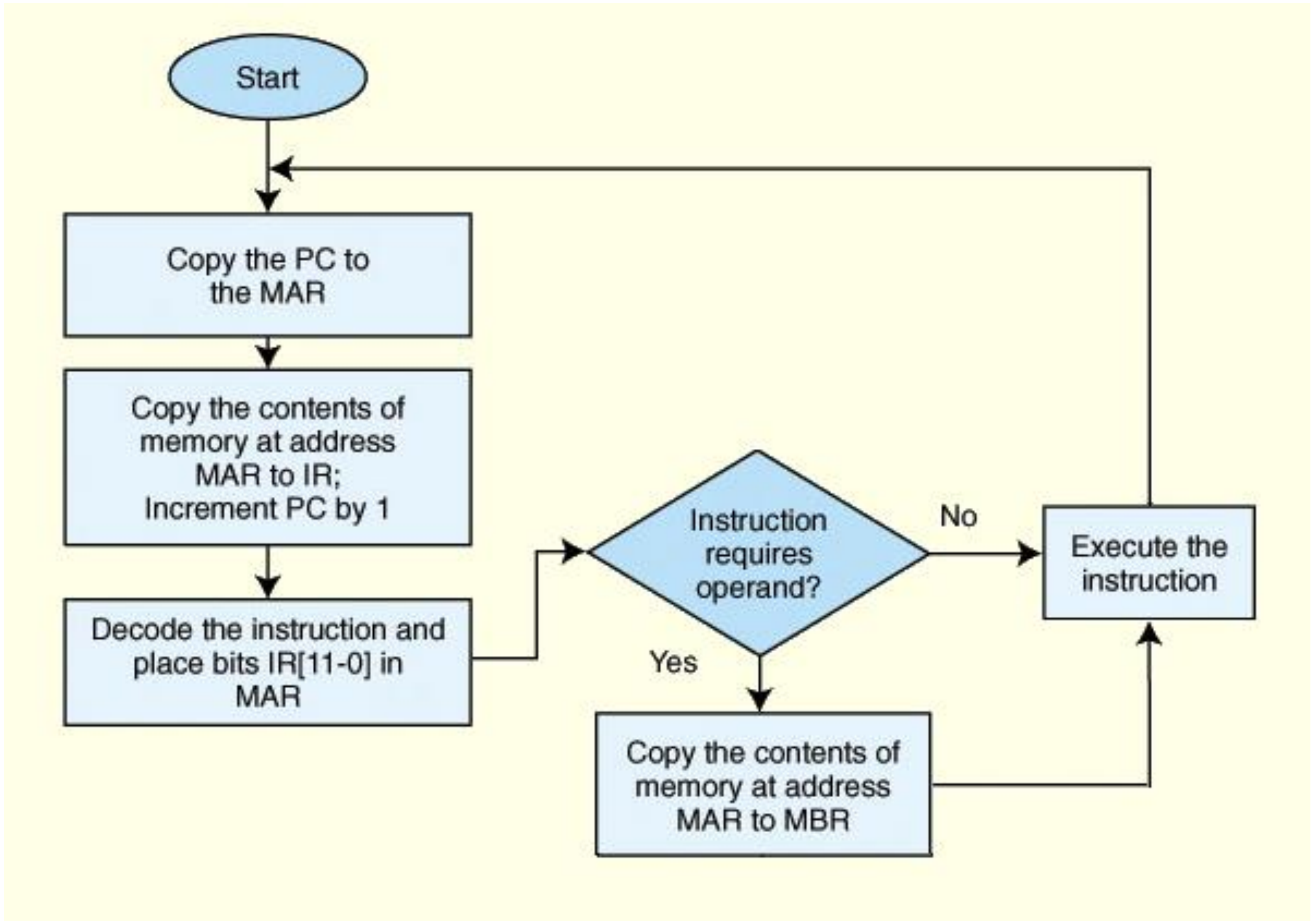
```
MAR ← X
MBR ← M[MAR]
AC ← AC + MBR
```

MARS

- Recall that **`SKIPCOND`** skips the next instruction according to the value of the AC.
- The RTL for this instruction is the most complex in our instruction set:

```
If IR[11 - 10] = 00 then
       If AC < 0 then PC ← PC + 1
else If IR[11 - 10] = 01 then
       If AC = 0 then PC ← PC + 1
else If IR[11 - 10] = 11 then
       If AC > 0 then PC ← PC + 1
```

Instruction Processing on MARS

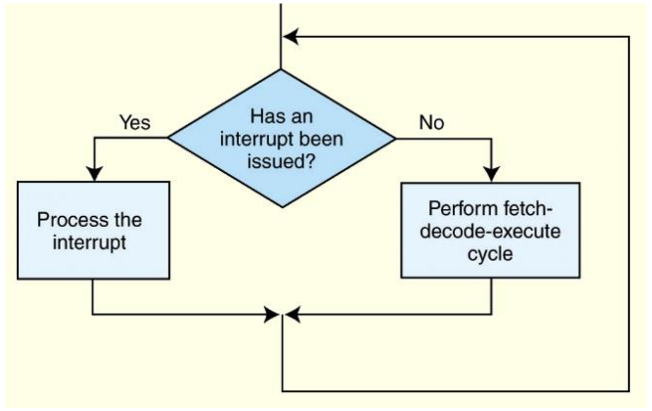

Start
Copy the PC to the MAR
Copy the contents of memory at address MAR to IR; Increment PC by 1
Decode the instruction and place bits IR[11-0] in MAR
Instruction requires operand?
No
Yes
Execute the instruction
Copy the contents of memory at address MAR to MBR

Instruction Processing

- All computers provide a way of interrupting the fetch-decode-execute cycle.
- Interrupts are asynchronous and indicate some type of service is required.
- Interrupts occur when:
  - A user break (e.g., Control+C) is issued
  - I/O is requested by the user or a program
  - A critical error occurs
- Interrupts can be caused by hardware or software.
  - Software interrupts are also called *traps*.

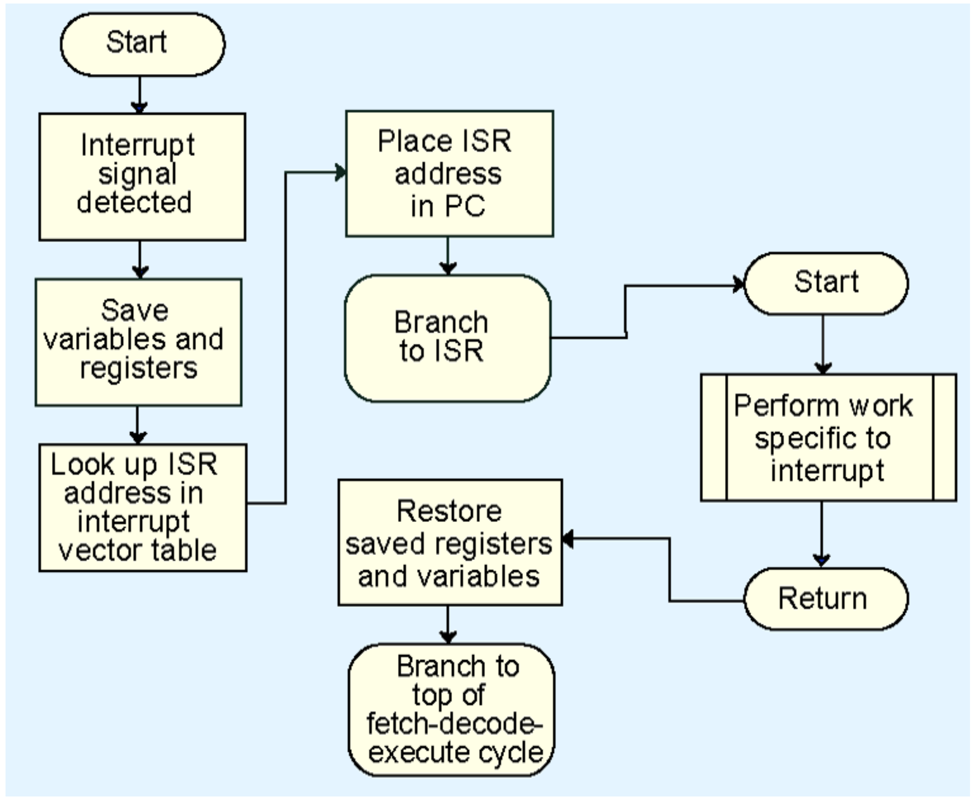
Instruction Processing …

- Interrupt processing involves adding another step to the fetch-decode-execute cycle as shown below.


Has an interrupt been issued?
Yes
No
Process the interrupt
Perform fetch-decode-execute cycle

The next slide shows a flowchart of “Process the interrupt.”

Instruction Processing…


Start
Interrupt signal detected
Save variables and registers
Look up ISR address in interrupt vector table
Place ISR address in PC
Branch to ISR
Start
Perform work specific to interrupt
Return
Restore saved registers and variables
Branch to top of fetch-decode-execute cycle

Instruction Processing …

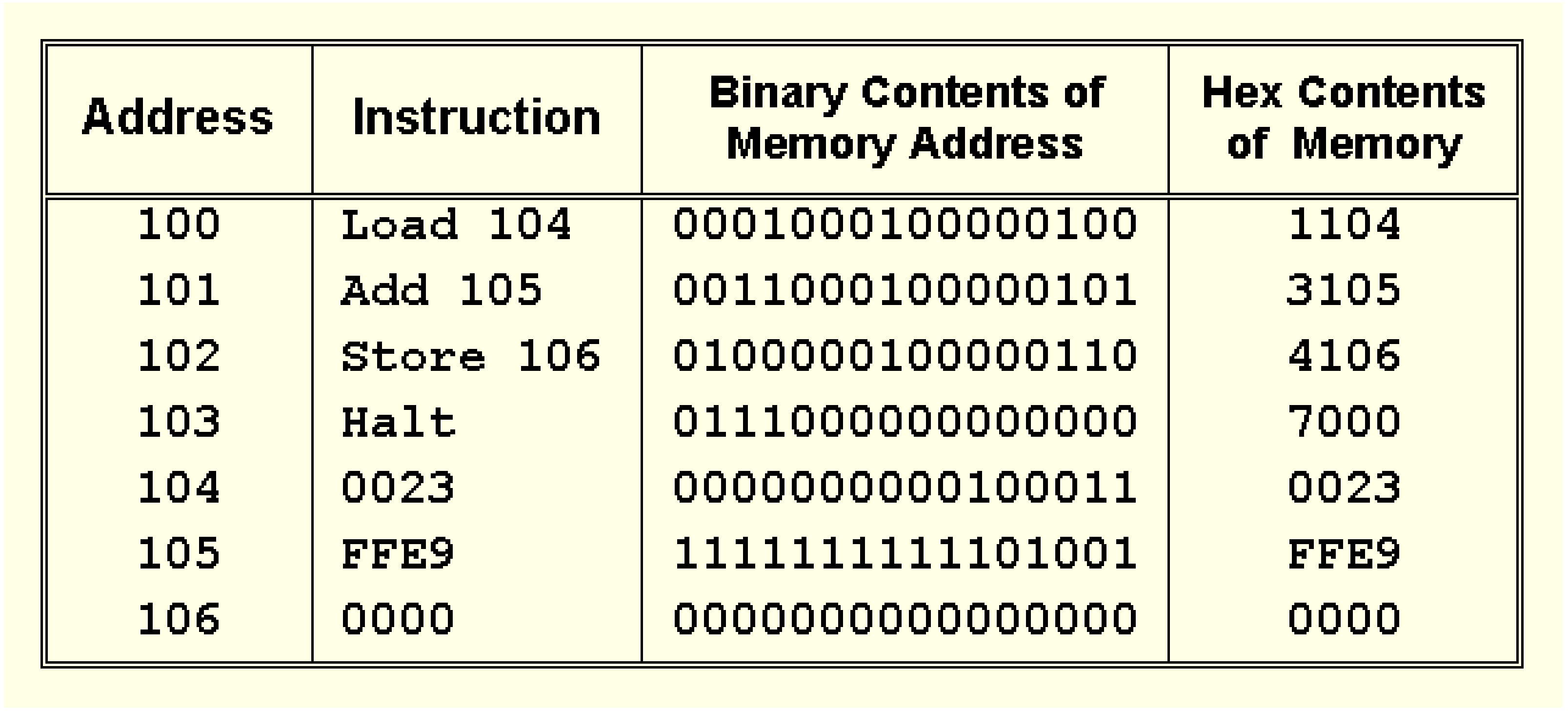
- For general-purpose systems, it is common to disable all interrupts during the time in which an interrupt is being processed.
  - Typically, this is achieved by setting a bit in the flags register.
- Interrupts that are ignored in this case are called *maskable*.
- *Nonmaskable* interrupts are those interrupts that must be processed in order to keep the system in a stable condition.

Instruction Processing

- Interrupts are very useful in processing I/O.
- However, interrupt-driven I/O is complicated, and is beyond the scope of our present discussion.
  - We will look into this idea in greater detail later on
- MARS, being the simplest of simple systems, uses a modified form of programmed I/O.
- All output is placed in an output register (OutREG) and the CPU polls the input register (InREG) until input is sensed, at which time the value is copied into the accumulator.

A Simple Program

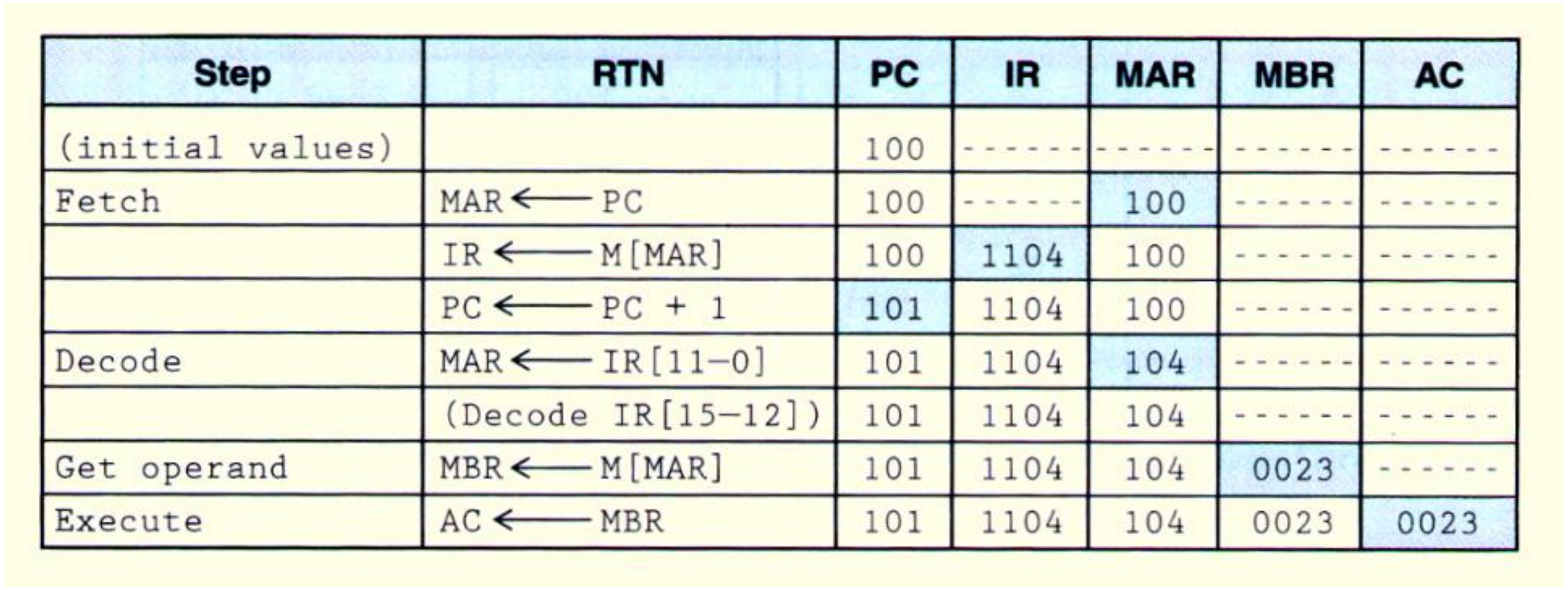
- Consider the simple MARS program given below. We show a set of mnemonic instructions stored at addresses 0x100 – 0x106 (hex):



Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	1111111111101001	FFE9
106	0000	0000000000000000	0000

A Simple Program

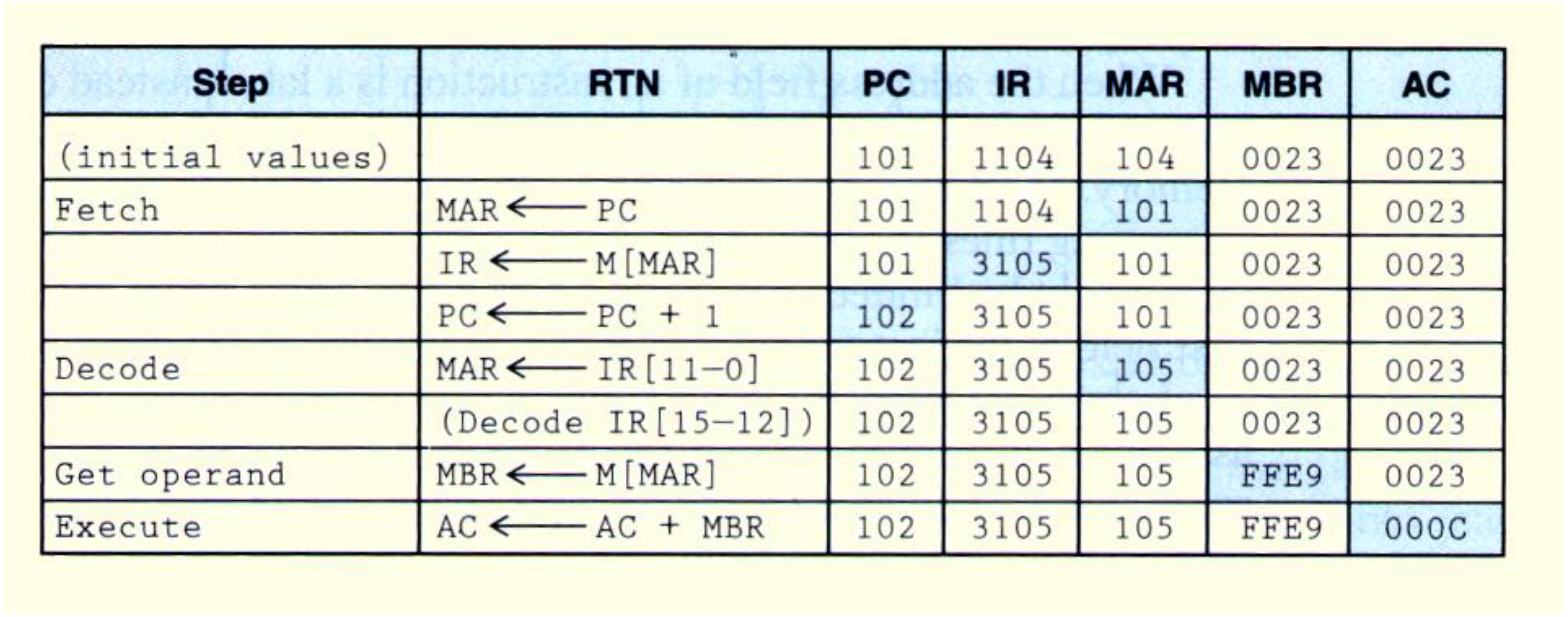
- Let’s look at what happens inside the computer when our program runs.
- This is the **`LOAD 104`** instruction:



Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100	------	------	------	------
Fetch	MAR $\longleftarrow$ PC	100	------	100	------	------
	IR $\longleftarrow$ M[MAR]	100	1104	100	------	------
	PC $\longleftarrow$ PC + 1	101	1104	100	------	------
Decode	MAR $\longleftarrow$ IR[11–0]	101	1104	104	------	------
	(Decode IR[15–12])	101	1104	104	------	------
Get operand	MBR $\longleftarrow$ M[MAR]	101	1104	104	0023	------
Execute	AC $\longleftarrow$ MBR	101	1104	104	0023	0023

A Simple Program

- Our second instruction is **`ADD 105`**:



Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR ← PC	101	1104	101	0023	0023
	IR ← M[MAR]	101	3105	101	0023	0023
	PC ← PC + 1	102	3105	101	0023	0023
Decode	MAR ← IR[11–0]	102	3105	105	0023	0023
	(Decode IR[15–12])	102	3105	105	0023	0023
Get operand	MBR ← M[MAR]	102	3105	105	FFE9	0023
Execute	AC ← AC + MBR	102	3105	105	FFE9	000C

```
public void run() {
      while (true) {

         // Fetch and parse
         int IR = mem[pc++];          // fetch next instruction
         int op-code   = (inst >> 12) &  15;    // get opcode (bits 12-15)
        int addr = (inst >>  0) & 255;    // get addr   (bits  0- 7)
```

```
// Execute
        switch (op-code) {
           case  1:                    break;
}
```

A Discussion on Assemblers

- Mnemonic instructions, such as **`LOAD 104`**, are easy for humans to write and understand.
- They are impossible for computers to understand.
- *Assemblers* translate instructions that are comprehensible to humans into the machine language that is comprehensible to computers
  - We note the distinction between an assembler and a compiler: In assembly language, there is a one-to-one correspondence between a mnemonic instruction and its machine code. With compilers, this is not usually the case.

A Discussion on Assemblers

- Assemblers create an *object program file* from mnemonic *source code* in two passes.
- During the first pass, the assembler assembles as much of the program as it can, while it builds a *symbol table* that contains memory references for all symbols in the program.
- During the second pass, the instructions are completed using the values from the symbol table.

A Discussion on Assemblers

- Consider our example program at the right.
  - Note that we have included two directives **HEX** and **DEC** that specify the radix of the constants.
- The first pass, creates a symbol table and the partially-assembled instructions as shown.

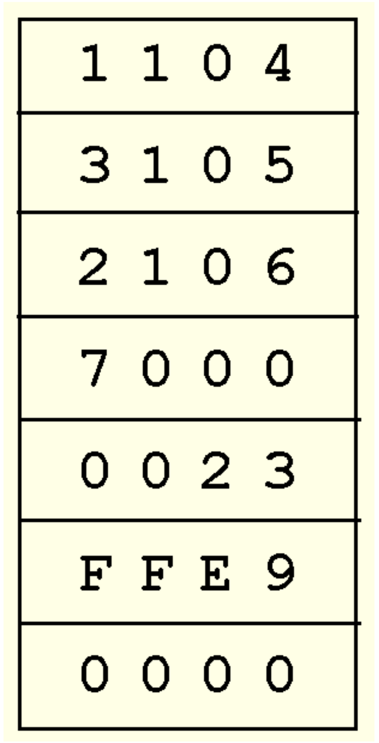
Address	Instruction
100	Load X
101	Add Y
102	Store Z
103	Halt
104 X,	DEC 35
105 Y,	DEC -23
106 Z,	HEX 0000

X	104
Y	105
Z	106

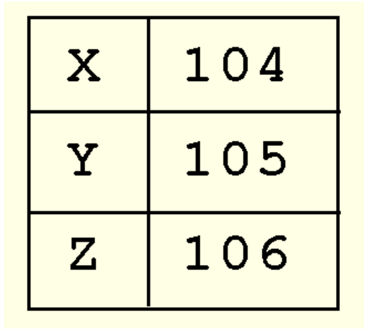
1	X
3	Y
2	Z
7 0 0 0	

A Discussion on Assemblers

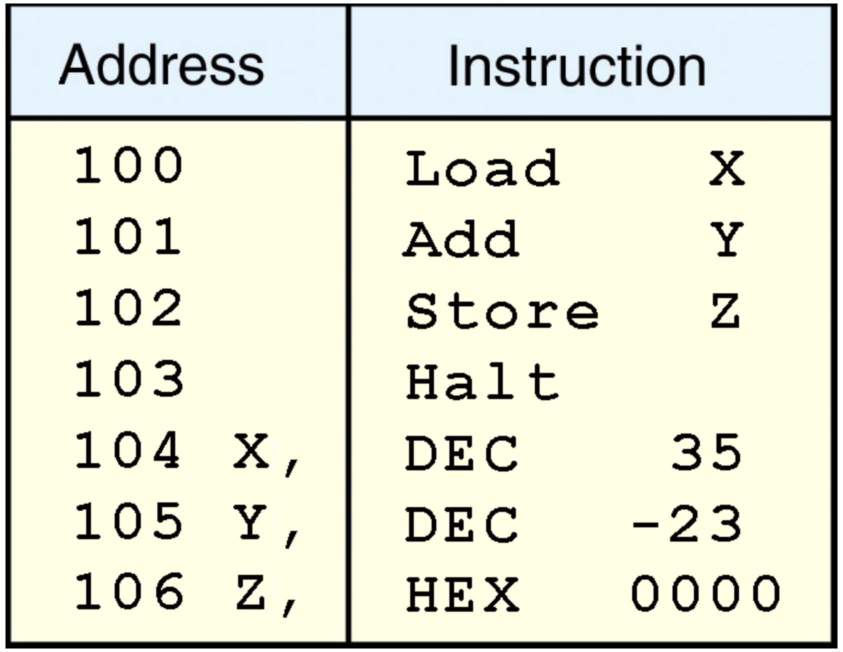
- After the second pass, the assembly is complete.



1 1 0 4
3 1 0 5
2 1 0 6
7 0 0 0
0 0 2 3
F F E 9
0 0 0 0



X	104
Y	105
Z	106



Address	Instruction
100	Load X
101	Add Y
102	Store Z
103	Halt
104 X,	DEC 35
105 Y,	DEC -23
106 Z,	HEX 0000

Thanks!