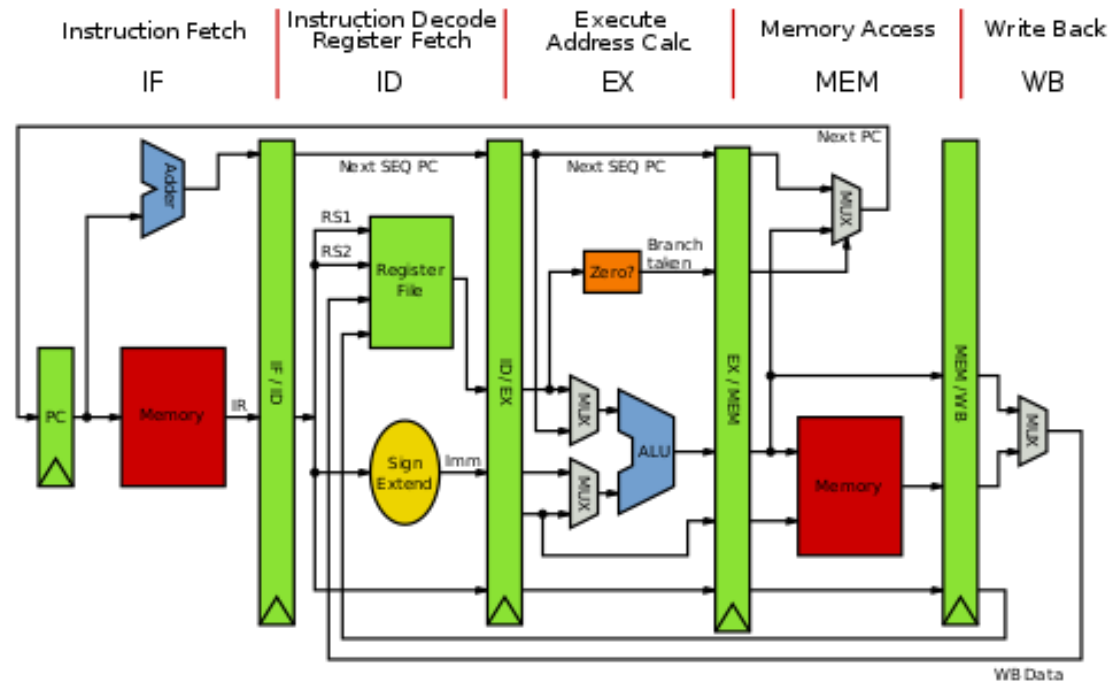


CHAPTER- 4

MARIE: AN INTRODUCTION TO A SIMPLE COMPUTER

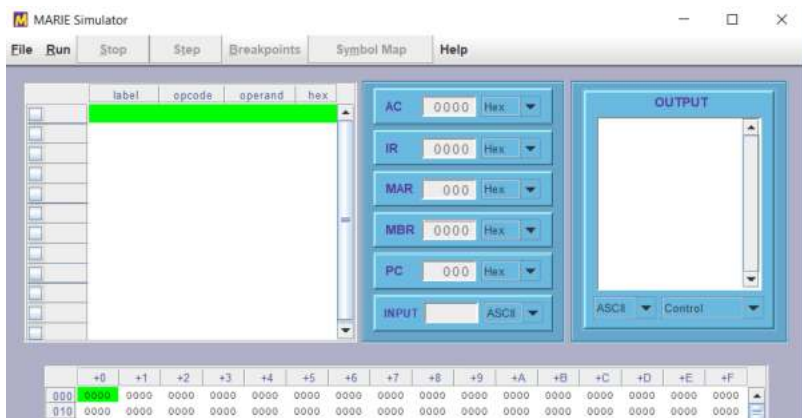


Dr. Rania Baashirah
r.baashirah@ubt.edu.sa



MARIE

- **M**achine **A**rchitecture that is **R**eally **I**ntuitive and **E**asy (**MARIE**).
- It has all the functional components necessary to be a real working computer.



Online Version:

<https://marie-js.github.io/>

<https://marie-js.github.io/MARIE.js/>

The Architecture of MARIE

- MARIE has the following characteristics:
 - Binary, two's complement.
 - Stored program, fixed word length.
 - Word (but not byte) addressable.
 - 4K words of main memory (this implies 12 bits per address (2^{12}))
 - 16-bit data (words have 16 bits)
 - 16-bit instructions: 4 for the opcode + 12 for the address
- **Registers:**
 - A 16-bit accumulator (AC)
 - A 16-bit instruction register (IR)
 - A 16-bit memory buffer register (MBR)
 - A 12-bit program counter (PC)
 - A 12-bit memory address register (MAR)
 - An 8-bit input register
 - An 8-bit output register

The Architecture of MARIE

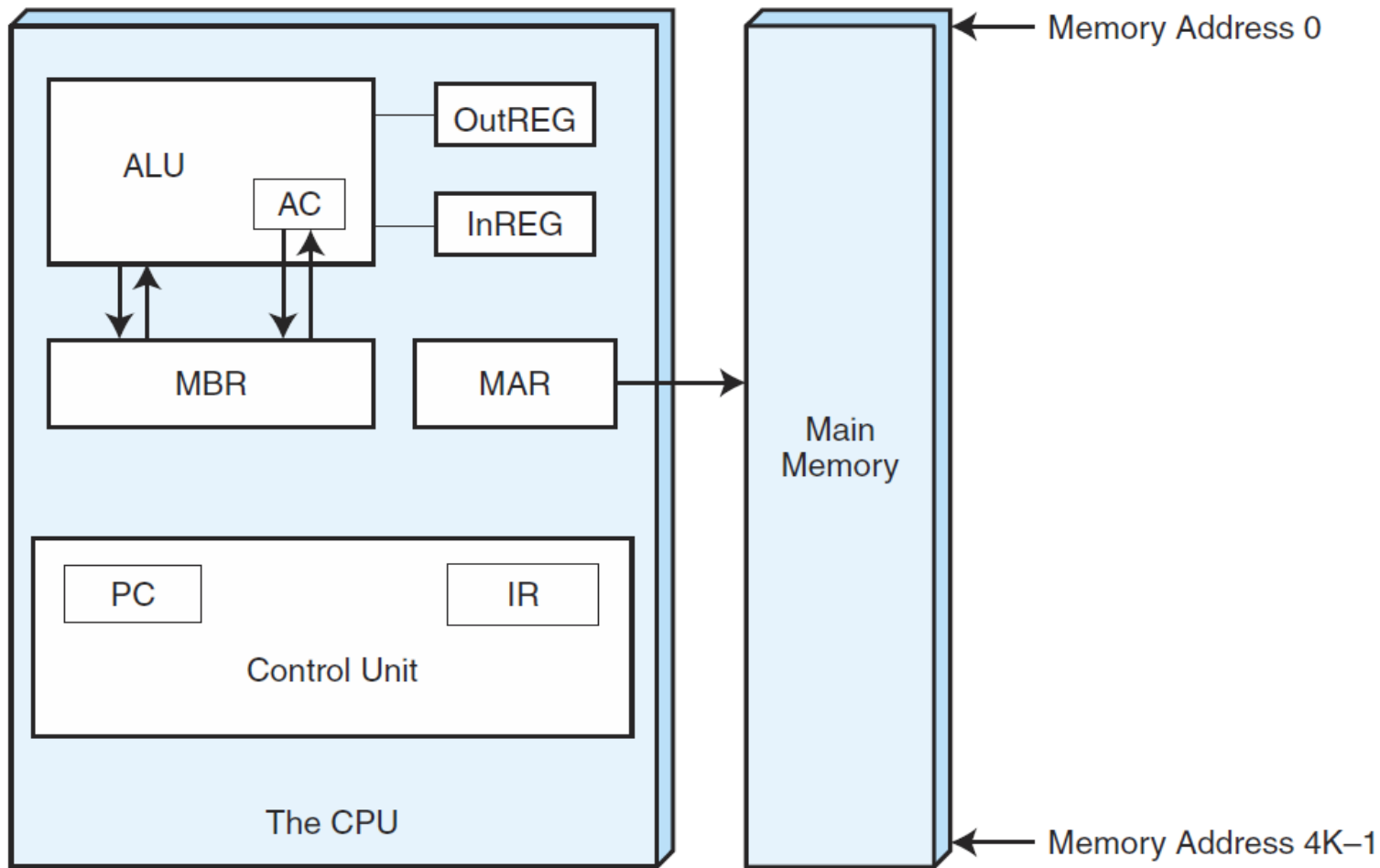


FIGURE 4.8 MARIE's Architecture



Registers and Buses



- In MARIE, there are seven registers, as follows:
- **AC (Accumulator)**, holds data value to be needed soon.
 - This is a general-purpose register.
 - Most computers today have multiple general-purpose registers.
- **MAR (Memory Address Register)** : holds the memory address of the location to for reading or writing.
- **MBR (Memory Buffer Register)**: holds either the data just read from memory or the data ready to be written to memory.
- **PC (Program Counter)**: holds the address of the next instruction to be executed in the program.
- **IR (instruction register)**: holds the next instruction to be executed.
- **InREG (Input Register)**: holds data from the input device.
- **OutREG (Output Register)**: holds data for the output device.

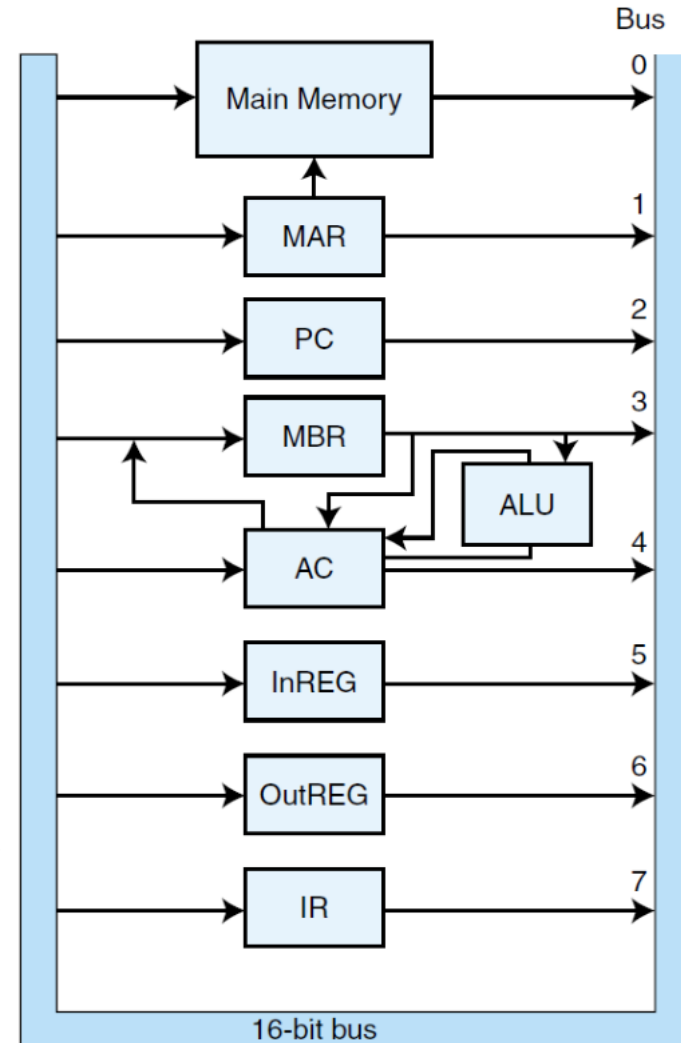
Registers and Buses

- Modern CPUs have multiple general-purpose registers, often called **user-visible registers**, that perform functions similar to those of the AC.
- for example, some computers have registers that shift data values and other registers that, if taken as a set, can be treated as a list of values.

Buses in MARIE

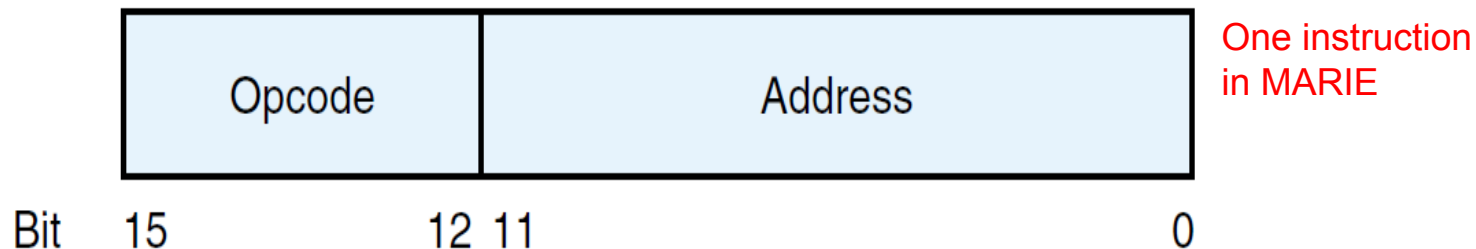
- Common Bus (16-bit) is shared between all registers.
- Direct pathway between MAR and RAM.
- AC directly put as an input of ALU and the result can be stored again in AC directly without using the common Bus.
- MBR directly connected to AC.

IMPORTANT



Instruction Set Architecture (ISA)

- ISA is a set of instructions, so each instruction asks CPU to do something. ISA is essentially an interface between the software and the hardware.
- **Opcode:** the part of operation to be done (add, multiply, read..etc).
- Since MARIE has 16-bit in its instruction (4-opcode + 12-address).
 - 4 bits can encode only $2^4 = 16$ different instructions.
 - 12 bits can access $2^{12} = 2K$ different memory locations.



Instruction Set Architecture (ISA)

Instruction Number		Instruction	Meaning
Bin	Hex		
0001	1	Load X	Load the 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 and store the result in AC.
0100	4	Subt X	Subtract the contents of address X from AC and store the result in 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 the program.
1000	8	Skipcond	Skip the next instruction on condition.
1001	9	Jump X	Load the value of X into PC.

- **Skipcond:** looks for bits 10 and 11,
 - If they are 00, this translates to "skip if the AC is negative".
 - If they are 01, this translates to "skip if the AC is equal to 0".
 - if they are 10, this translates to "skip if the AC is greater than 0".

[https://github.com/MARIE-js/MARIE.js/wiki/MARIE-Instruction-Set-\(with-Opcodes\)](https://github.com/MARIE-js/MARIE.js/wiki/MARIE-Instruction-Set-(with-Opcodes))

Example Program

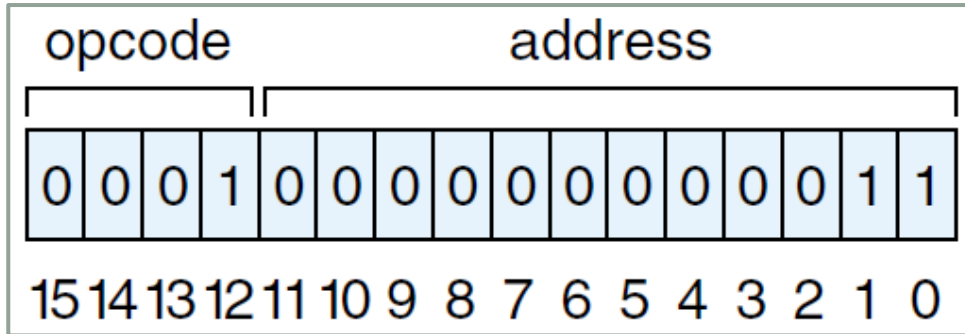
- This is an example program of adding two numbers and store the output in somewhere in memory.

Same code in Java:

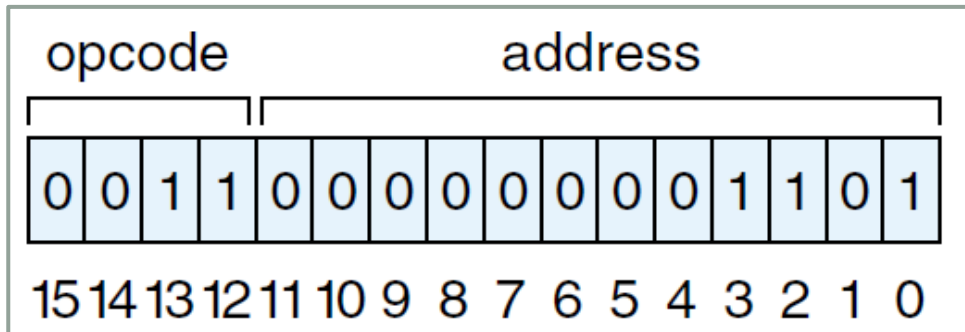
```
int X = 35;  
int Y = -23;  
int Z = X + Y;
```

Hex Address	Instruction	
100	Load	104
101	Add	105
102	Store	106
103	Halt	
104	0023	
105	FFE9	
106	0000	

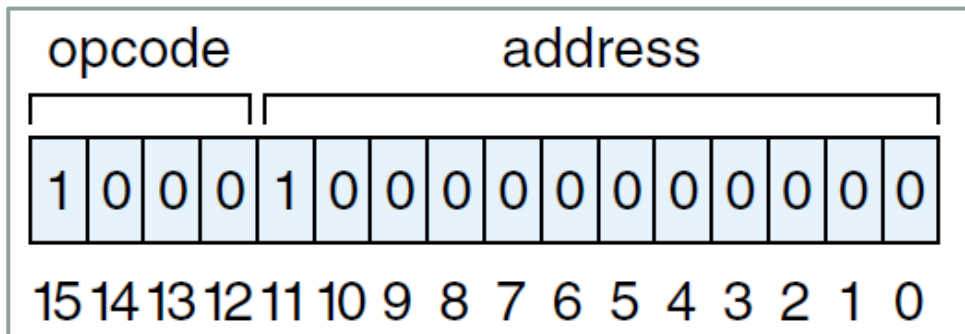
MARIE's Instruction Examples



Load the content of location 3 in the memory into the AC.



Add the content of location 13 to the AC and put the result in AC



Skip the next instruction if $AC > 0$

Register Transfer Notation (RTN)

- Each instruction appears to be very simple. However, each instruction involves multiple operations.
- For example, the LOAD instruction has *microoperations* are being executed.
 - First, load the address from the instruction into MAR.
 - Then, load the data in memory at this location into MBR.
 - Then, load the MBR into AC.

RTL

- The symbolic notation used to describe the microoperations is called **Register Transfer Notation (RTN)** or **Register Transfer Language (RTL)**.
- **M[X]** to indicate the actual data stored at location X in memory.
- The back arrow \leftarrow to indicate a transfer of information.
- Example: the microoperations (RTL) of "LOAD X" are:
 - $MAR \leftarrow X$ //using main bus (1 clock cycle)
 - $MBR \leftarrow M[MAR]$ //using main bus (1 clock cycle)
 - $AC \leftarrow MBR$ //using direct pathways (same previous clock)
- Review the bus architecture image.

Assuming that IR is already contains the instruction.

RTL for all instructions

Instruction	Microoperations (RTL)	Description	Clock cycles
Store X	$MAR \leftarrow X$ $MBR \leftarrow AC$ $M[MAR] \leftarrow MBR$	Put X into MAR Put the value of AC into MBR Write the location of MAR in memory with MBR value.	2
Add X	$MAR \leftarrow X$ $MBR \leftarrow M[MAR]$ $AC \leftarrow AC + MBR$	Put X into MAR Read the location of MAR and its value in MBR. Add AC to MBR and store the result in AC.	3
Subt X	$MAR \leftarrow X$ $MBR \leftarrow M[MAR]$ $AC \leftarrow AC - MBR$	Put X into MAR Read the location of MAR and its value in MBR. Subtract MBR from AC and store the result in AC.	3
Input	$AC \leftarrow InREG$	Read the input register and put it in AC	1
Output	$OutREG \leftarrow AC$	Place the AC value in the output.	1
Halt	No operations	the machine simply stop execution of the program	0
Jump X	$PC \leftarrow X$		1

RTL for all instructions

- Skipcond: it follows this algorithm.

```
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] = 10 then
    If AC > 0 then PC ← PC + 1
```

NOTE: If the IR[11-10] are both 1, an error condition results. However, an additional condition could also be defined using these bit values.

Instruction Processing

- Now that we have a basic language with which to communicate ideas to our computer.
- After studying each instruction separately, we need to discuss exactly how a whole program is executed.
- All computers follow a basic machine cycle: the *fetch*, *decode*, and *execute* cycle.

The Fetch–Decode–Execute Cycle

- The CPU do the following
 - **Fetches** an instruction.
 - transfers it from main memory to the instruction register.
 - **Decodes** it.
 - determines the opcode and fetches any data necessary.
 - **Executes** it.
 - performs the microoperation[s] indicated by the instruction.

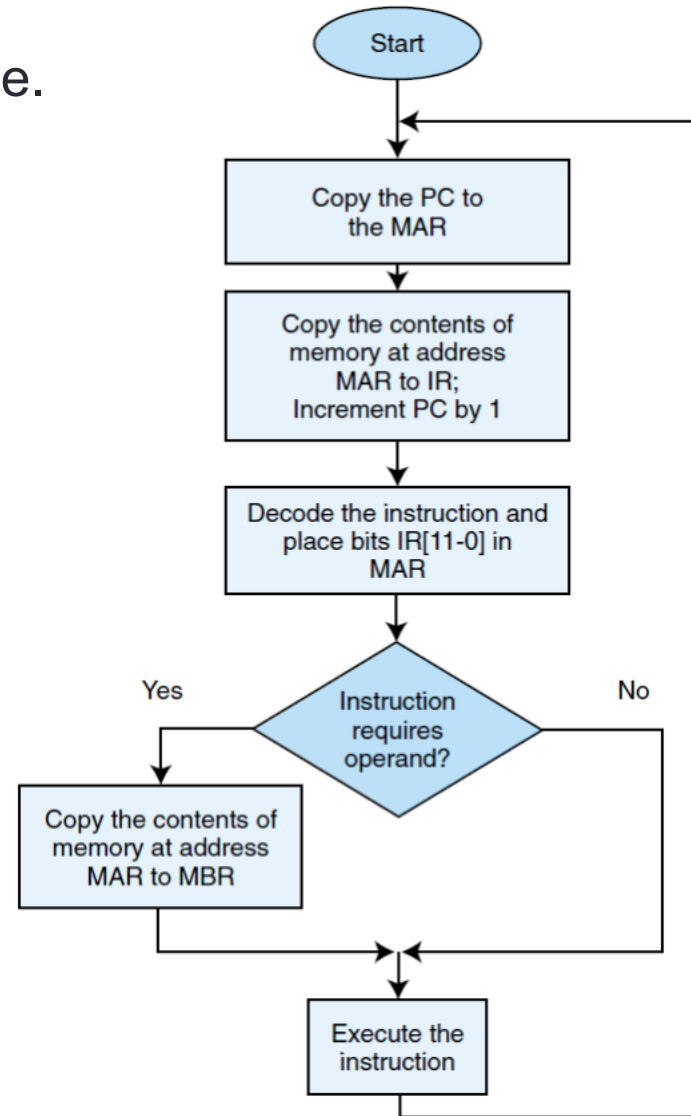
The Fetch–Decode–Execute Cycle

- Specifically, the CPU do the following operations, in order to execute one instruction in the program

	Steps	Operations	Description
Fetch	Step-1	$MAR \leftarrow PC$	Copy the contents of the PC to the MAR
Fetch	Step-2	$IR \leftarrow M[MAR]$ $PC \leftarrow PC + 1$	place the instruction in IR PC now points to the next instruction in the program
Decode	Step-3	$MAR \leftarrow IR[0-11]$ decode $IR[12-15]$	Copy the rightmost 12 bits of the IR decode the leftmost 4 bits to determine the opcode
Execute	Step-4	$MBR \leftarrow M[MAR]$ execute	If necessary, use the address in the MAR to go to memory to get data, placing the data in the MBR

The Fetch–Decode–Execute Cycle

- Flowchart of the Fetch-Decode-Execute Cycle.
- In this way any program can be executed.

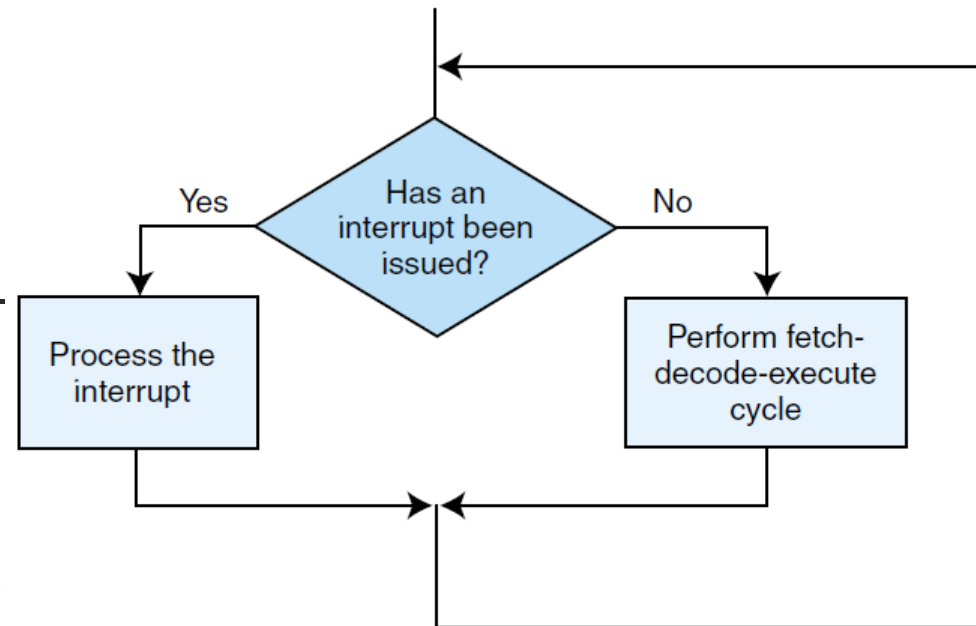


Interrupts and the Instruction Cycle

- **Problem:** The computer must be able to read each character that is put into the input register. if the keyboard typing is very fast, and another character is entered into InREG register before the computer has a chance to process the current character, the current character is lost!!
 - It is more likely, since the processor is very fast and keyboard input is very slow.
- **Solution:** by using *interrupt-driven I/O*.

Interrupt-Driven I/O

- A signal (interrupt) is submitted from I/O device to CPU indicating that input or output is complete.
- CPU to switch from the usual fetch-decode-execute cycle to "recognize" this interrupt.
- A **special bit** (flag) is set whenever interrupt is occurred.



Interrupt Examples:

- **External Interrupt:** When you press Ctrl-break, Ctrl-C to break up the program.
- **Internal Interrupt:** division by zero, stack overflow violations.

MARIE's I/O

- I/O processing is one of the most challenging aspects of computer system design and programming.
- The timing used by these two registers is very important.
 - **For example:** you may lose data if you press on a key in the keyboard and the CPU is still busy of registering the previous key.
- MARIE employs a modified type of programmed I/O that places all I/O under the direct control of the programmer.
 - output action is simply a matter of placing a value into the OutREG.
 - For input, CPU is put into a waiting state until a character is entered into InREG

A Simple Program

- For simplicity, let's assume our programs in MARIE are always loaded starting at address 100_{16}
- To avoid using a subscript of 16, we use the standard "0x" notation to distinguish a hexadecimal number.
 - **For example**, instead of saying 123_{16} , we write **0x123**.

Program of adding 2 numbers

- There is a one-to-one correspondence between the **assembly language** and the **machine language** instructions. This makes it easy to convert assembly language into machine code. Using the instruction tables given in this chapter, you should be able to hand assemble any of our example programs.

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	0010000100000110	2106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	1111111111101001	FFE9
106	0000	0000000000000000	0000

Program of Adding 2 Numbers

- Trace a sample instruction of the Program to Add Two Numbers.

Hex Address	Instruction
100	Load 104
101	Add 105
102	Store 106
103	Halt
104	0023
105	FFE9
106	0000

Add 105

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	$MAR \leftarrow PC$			101		
	$IR \leftarrow M[MAR]$	101	3105			
	$PC \leftarrow PC + 1$	102				
Decode	$MAR \leftarrow IR[11-0]$			105		
	(Decode $IR[15-12]$)	No change				
Get operand	$MBR \leftarrow M[MAR]$				FFE9	0023
Execute	$AC \leftarrow AC + MBR$					000C

A Discussion On Assemblers

- It is difficult for human beings to understand and program in sequences of 0s and 1s. We prefer words and symbols, so it seems a natural solution to use a special program that does this simple conversion for us.
- This program is called an **assembler**.
- Assembler reads a **source** file (assembly program) and produces an **object** file (the machine code).

Using Labels

- We substitute *labels* to identify or name particular memory addresses, making assembly program even simpler to write.
- Assembler reads the program twice:
 - On the first pass, it builds a set of labels called a *symbol table*.
 - Second pass is like "fill in the blanks".

Address	Instruction
100	Load X
101	Add Y
102	Store Z
103	Halt
X, 104	0023
Y, 105	FFE9
Z, 106	0000

An Example Using Labels

MARIE requires labels to be followed by a comma (,)

X	104
Y	105
Z	106

Symbol Table

Assembler Directive

- People are uncomfortable reading hexadecimal.
- **Assembler directive:** is an instruction is given to the assembler to specify which base is to be used to interpret the value.
 - specifically for the assembler that is not supposed to be translated into machine code.
- Comments also is considered as directive

- We use DEC for decimal and HEX for hexadecimal in MARIE's assembly language.
- MARIE's comment delimiter is a front slash ("/").

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

Program Example

/add tow numbers

```
Load X
Add Y
Store S
Output
Halt
S, DEC 0
X, DEC 5
Y, DEC 4
```

/add two numberes
/given from a user.

Input
store 77

input
store 88

Load 77
add 88

output
Halt

/ print "Hello"

```
LOAD X1
output
LOAD X2
output
LOAD X3
output
LOAD X4
output
LOAD X5
output
Halt
```

```
X1, dec 72
X2, dec 101
X3, dec 108
X4, dec 108
X5, dec 111
```

Multiplication (using loop)

- Multiplication is nothing but a repeated addition.
 $5 * 4 = 5 + 5 + 5 + 5$

Same code in Java:

```
int X = 5;
int Y = 4;
int Z = X * Y;
System.out.println(Z);
```

```
X, DEC 5
Y, DEC 4

/ Loop for performing iterative addition
loop,    Load num
          Add X
          Store num

          Load Y
          Subt 1
          Store Y

          Skipcond 400      / have we completed the
multiplication?
          Jump loop        / no; repeat loop
          / yes, so exit the loop

/ Output result and Stop
Load num
Output
Halt

num, DEC 0
```