Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Introduction to MARIE

Schedule

- **Today** – Introduce new machine architecture – MARIE – and assembly programming language

---

- **Next Tuesday**
  - Continue with MARIE intro
  - Exam review
- **Next Thursday – Exam 1**
  - Chapter 2 (Data representations)
  - Chapter 3 (Digital logic)
  - Part of Chapter 4 (basic organization and memory systems. Nothing on MARIE)

- **Following Tuesday** – Meet in **CTC 115** for assembly programming

Review Exercise – Memory Organization

- Exercise: Build a 1M x 16 word-addressable main memory using 128K x 4 RAM chips.
  1. **How many address bits are needed per RAM chip?**
  2. **How many RAM chips are there per word?**
  3. **How many RAM chips are necessary?**
  4. **How many address bits are needed for all memory?**
  5. **How many address bits would be needed if it were byte addressable?**
  6. **How many banks will there be?**
  7. **What bank would contain address 47129_{16} with (a) high-order interleaving or (b) low-order interleaving?**

Solution to Exercise

1. Each RAM chip has 128K locations: $2^7 * 2^{10}$ = **17 bits**

2. Each RAM chip location stores 4 bits, but we need 16:
   1. **4 chips needed per word**

3. Each RAM chip has 128K locations, but we need 1M locations:
   1. 1M/128K = 8 (times 4 chips per word) = **32 RAM chips** (8 rows, 4 columns)

4. Memory is 1M: 2^20 = **20 bits for all of memory**

5. Byte addressable adds 1 more bit here (to select either the lower 8 or upper 8 of the 16 bit long word): **21 bits**

6. **8 banks** of memory, where each bank has 4 chips

7. Address is 20 bits long, bank is upper 3 bits (2^3=8):
   47129(16) = 0100 0111 0001 0010 1001 (2)
   With high-order interleaving, bank is **#2**
   With low-order interleaving, bank is **#1**

MARIE

- Simple model computer used in this class
- **MARIE**
  - **M**achine **A**rchitecture that is **R**eally **I**ntuitive and **E**asy
  - Designed for education only
- While this system is too simple to do anything useful in the real world, a deep understanding of its functions will enable you to comprehend system architectures that are much more complex

MARIE Overview

- The MARIE architecture has the following characteristics:
  - Binary, two’s complement data representation
  - Stored program, fixed word length data and instructions
  - 4K 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

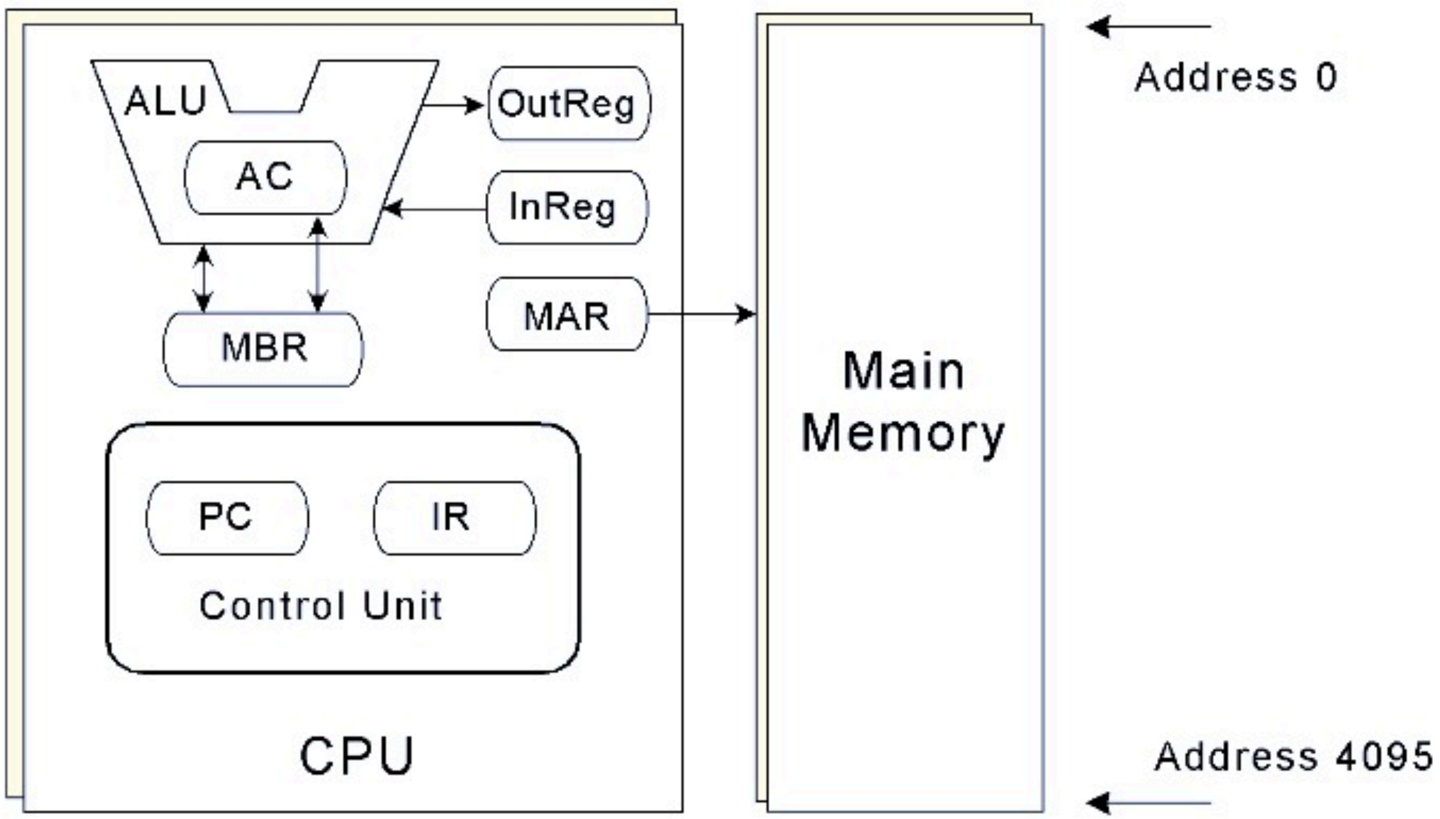
MARIE Registers (1-3)

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

MARIE Registers (4-7)

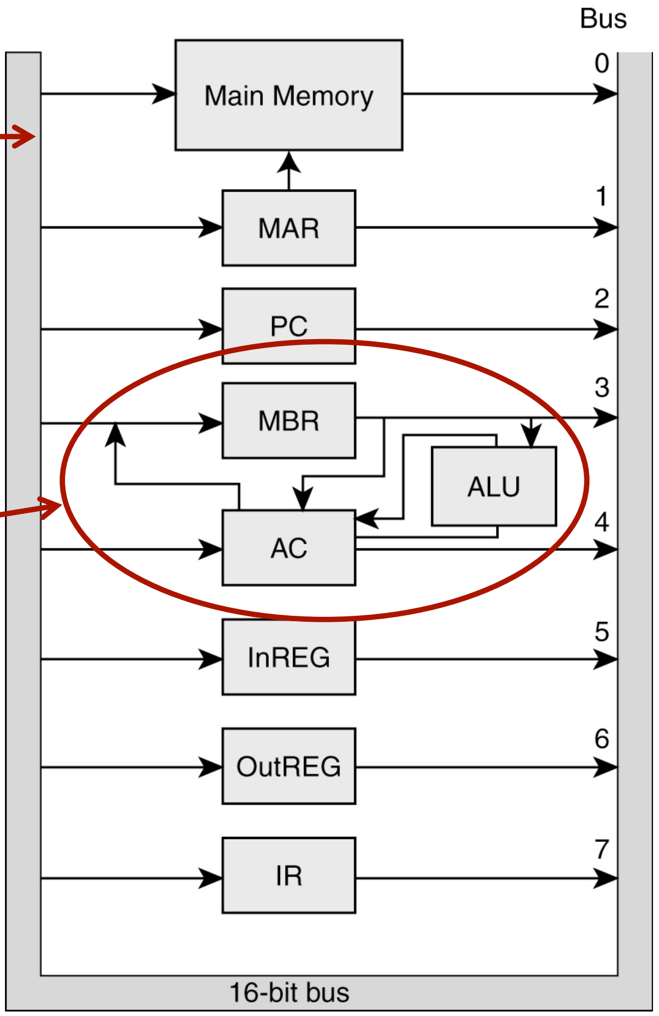
- **Program counter (PC)**
  - 12-bit register that holds the address of the next program instruction to be executed
- **Instruction register (IR)**
  - 16-bit register that holds an instruction immediately preceding its execution
- **Input register (InREG)**
  - 8-bit register that holds data read from an input device
- **Output register (OutREG)**
  - 8-bit register that holds data that is ready for the output device

MARIE Architecture



MARIE Data Path

- **Common data bus**
  - Links main memory and registers
  - Each device identified by unique number
  - Bus has control lines that identify device used in operation
- **Dedicated data paths**
  - Permits data transfer between accumulator (AC), memory buffer register (MBR), and ALU without using main data bus



MARIE Instruction Set Architecture

MARIE ISA

- **Instruction Set Architecture (ISA)** is the interface between hardware and software
  - Specifies the format of processor instructions
  - Specifies the primitive operations the processor can perform
- Real ISAs?
  - **Hundreds** of different instructions for processing data and controlling program execution
- MARIE ISA?
  - **Only thirteen instructions**

MARIE Instructions (Basic)

Instruction #

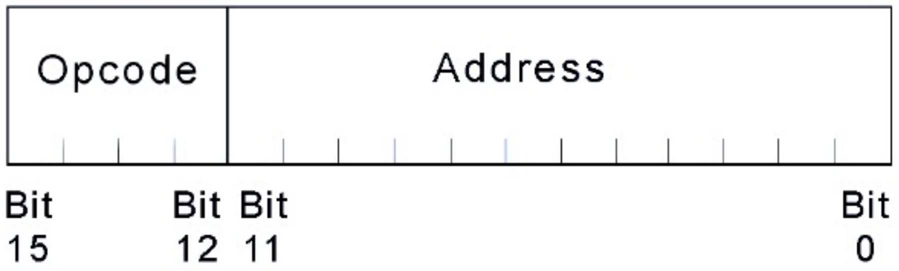
See table 4.7 in book!

Binary	Hex	Instruction	Meaning
0001	1	**LOAD X**	Load contents of address X into AC
0010	2	**STORE X**	Store contents of AC at address X
0011	3	**ADD X**	Add contents of address X to AC
0100	4	**SUBT X**	Subtract contents of address X from AC
0101	5	**INPUT**	Input value from keyboard into AC
0110	6	**OUTPUT**	Output value in AC to display
0111	7	**HALT**	Terminate program
1000	8	**SKIPCOND**	Skip next instruction on condition based on AC value
1001	9	**JUMP X**	Load value of X into PC

… plus a few more instructions we'll add later…

MARIE Instructions

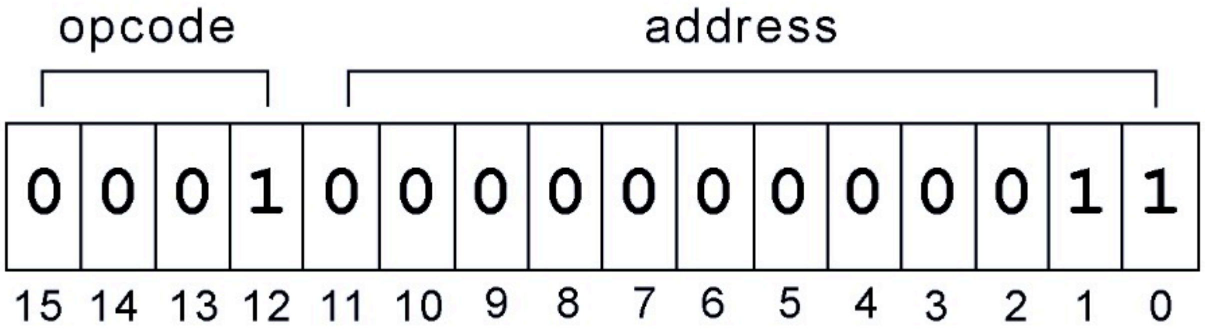
- How do we format these instructions in computer memory?



- Two fields
  - **Opcode** (4 bits) – Operation code
  - **Address** (12 bits) – Address to operate to/from

MARIE Instruction Example - LOAD

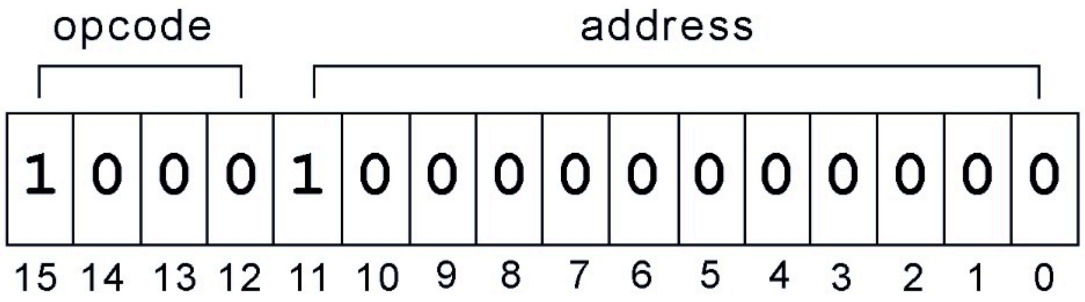
- Bit pattern for a `LOAD` instruction
  - Could be saved in memory or IR (if executing right now)



- **Decode this instruction**
  - Opcode is 1 – LOAD instruction
  - Address is 3 – Data will be loaded from here
- **What is the Hex representation of this instruction?**

MARIE Instruction Example - SKIPCOND

- Bit pattern for a `SKIPCOND` instruction
  - Could be saved in memory or IR (if executing right now)



- **Decode this instruction**
  - Opcode is 8 – SKIPCOND instruction
  - "Address" is not really an address here!
    - Upper two "address" bits are 10
    - Translation: The next instruction will be skipped if the value in the AC is **greater than** zero

MARIE RTL Design

MARIE RTL

- Each instruction can be somewhat complicated
  - Lots of individual processor elements must be precisely coordinated / scheduled
- Each **ISA instructions** is built from a sequence of *smaller instructions* called **micro-operations** (micro-ops)
  - Exact sequence of micro-ops is described to programmers / designers by a register transfer language (RTL)
- MARIE RTL notation
  - M[X] – indicates actual data stored in memory location X
  - $\leftarrow$ - indicates transfer of bytes to a register or memory location

MARIE RTL

- RTL for `LOAD X` instruction:

```
MAR ← X
MBR ← M[MAR]
AC ← MBR
```

- RTL for `ADD X` instruction:

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

MARIE RTL

- SKIPCOND skips the next instruction according to the value of the AC
- RTL for this instruction is complex!

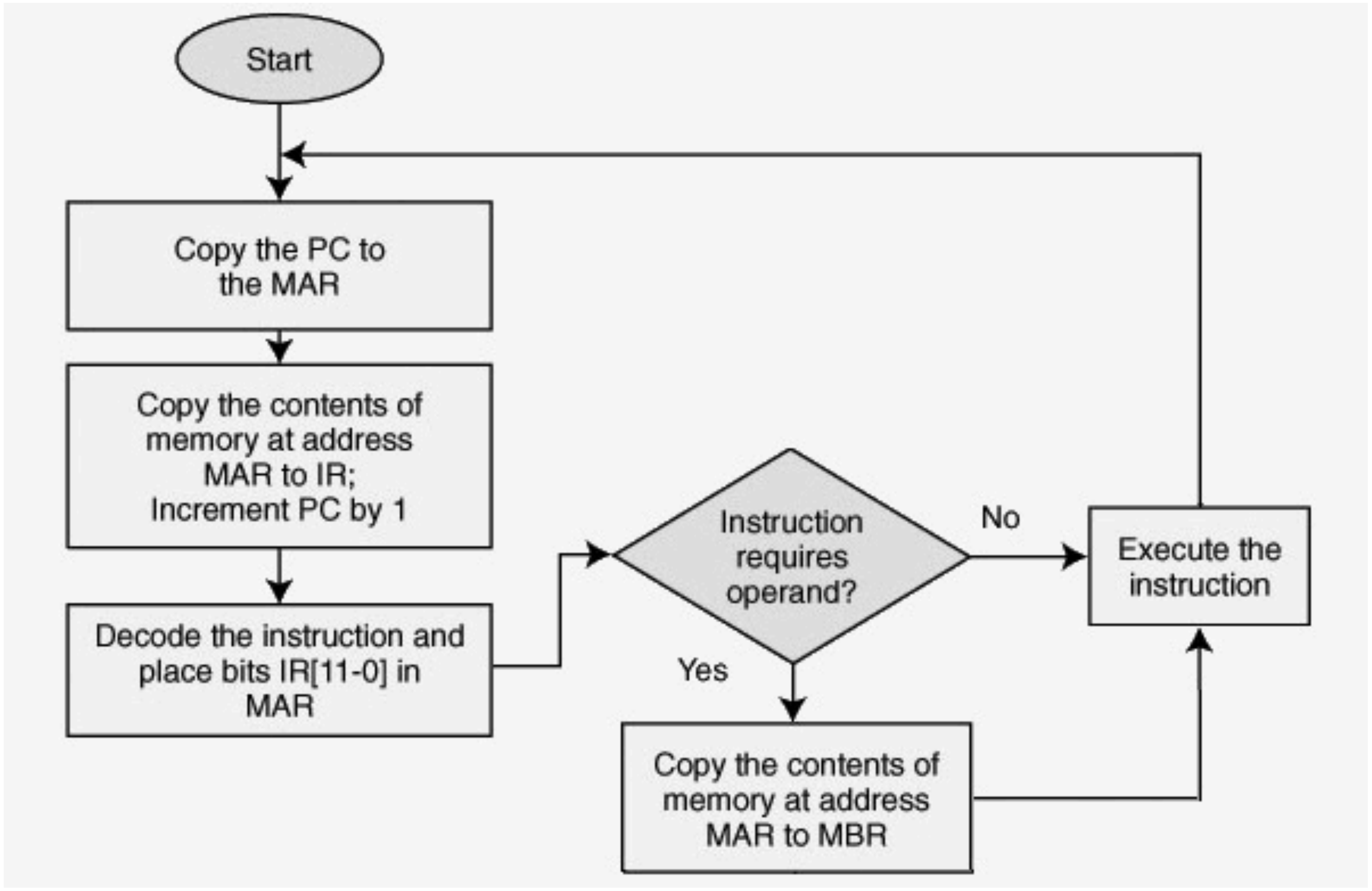
```
If IR[11 - 10] = 00 then
   If AC < 0 then PC ← PC + 1
else If IR[11 - 10] = 01 then
   If AC = 0 then PC ← PC + 1
else If IR[11 - 10] = 11 then
   If AC > 0 then PC ← PC + 1
```

MARIE Instruction Processing

Instruction Processing

- How does a computer run a program?
  - **Fetch-decode-execute cycle**
- Steps
  1. **Fetch** an instruction from memory and place it into the IR
  2. **Decode** instruction in IR to determine what needs to be done next.
     1. *If a memory value (operand) is involved in the operation, it is retrieved and placed into the MBR*
  3. **Execute** instruction (now that all data is ready)

Instruction Processing

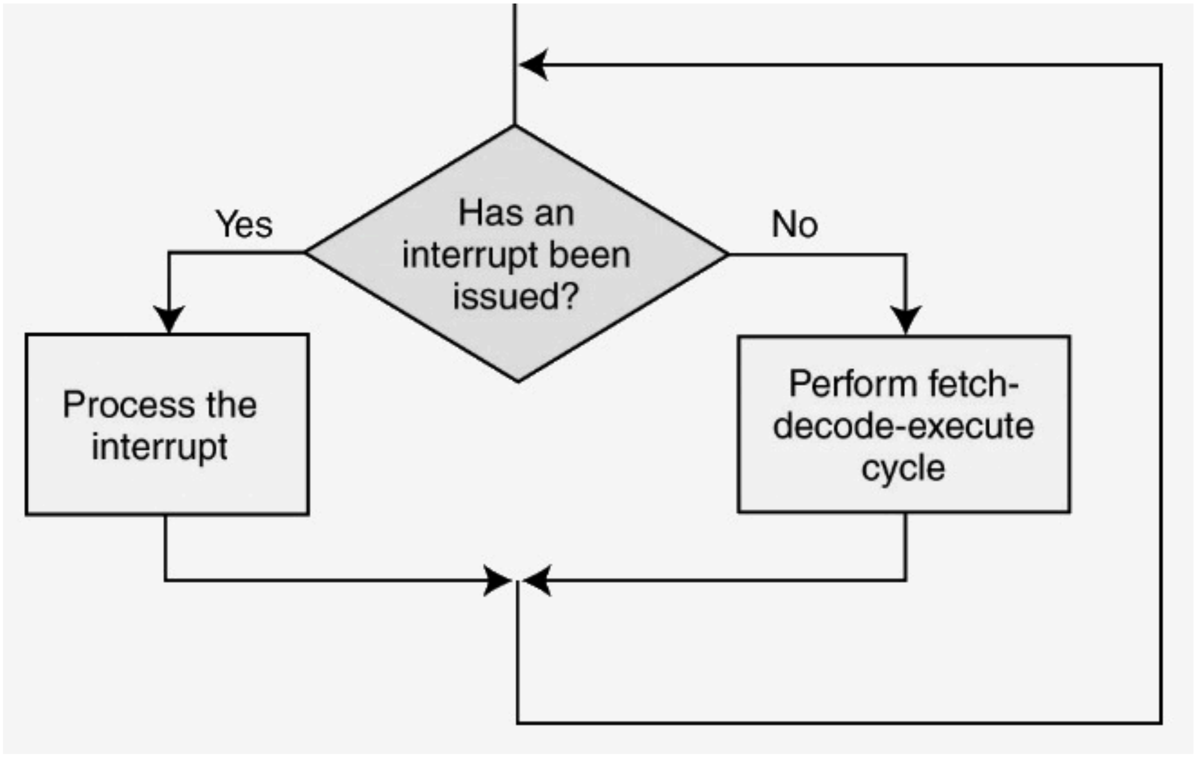

Start
Copy the PC to the MAR
Copy the contents of memory at address MAR to IR; Increment PC by 1
Decode the instruction and place bits IR[11-0] in MAR
Instruction requires operand?
No
Yes
Execute the instruction
Copy the contents of memory at address MAR to MBR

Interrupting the Cycle

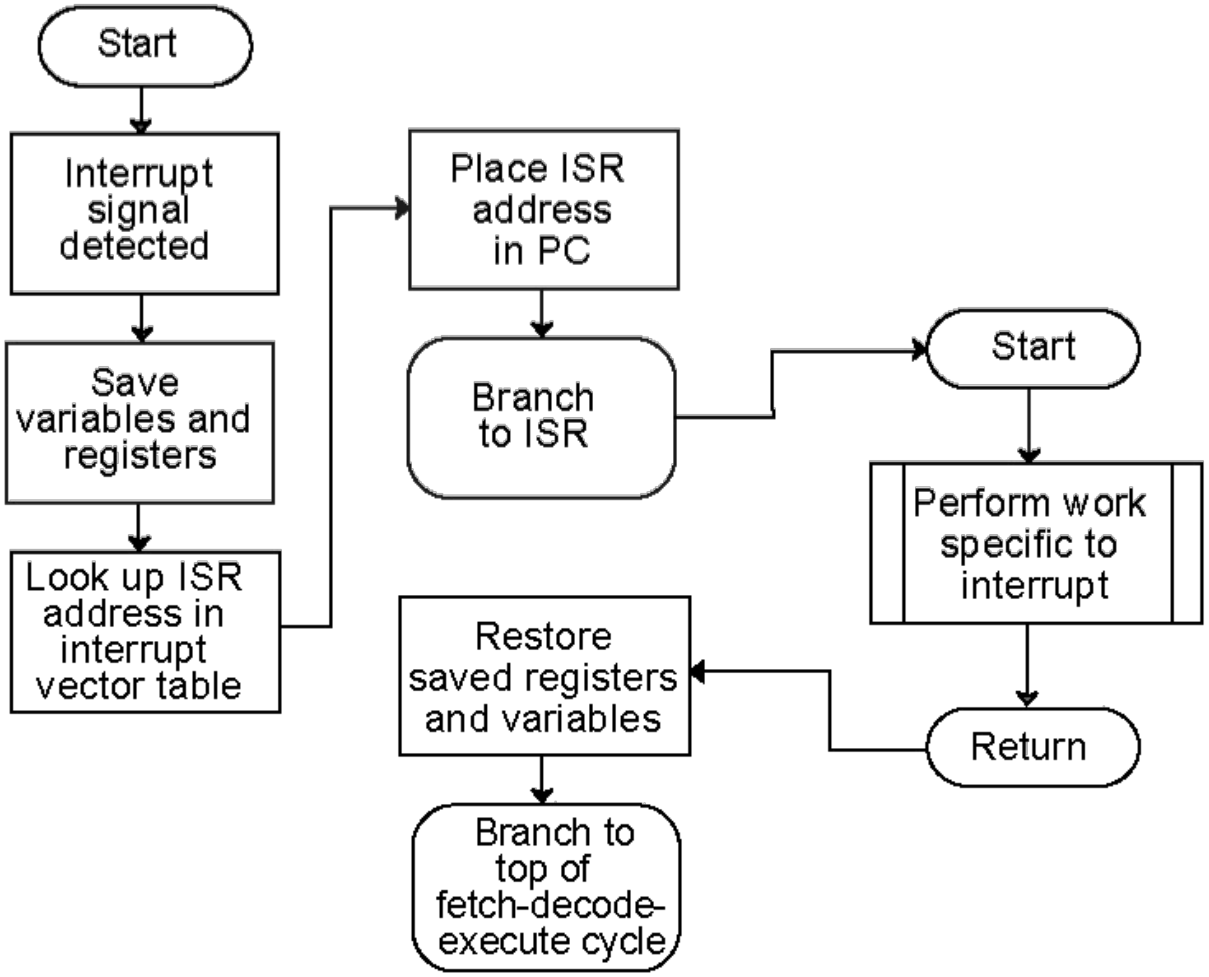
- The fetch-decode-execute cycle can be interrupted
- Interrupts occur when:
  - A user break (e.,g., Control+C) is issued
  - I/O is requested by the user or a program
  - A critical error occurs
- Interrupts can be caused by hardware or software.
  - Software interrupts are also called **traps**

Interrupting the Cycle

- Interrupt processing adds one step to the *fetch-decode-execute* cycle


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

Interrupt 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

Interrupt Processing

- For general-purpose systems, it is common to disable all interrupts during the time in which an interrupt is being processed
  - Set 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

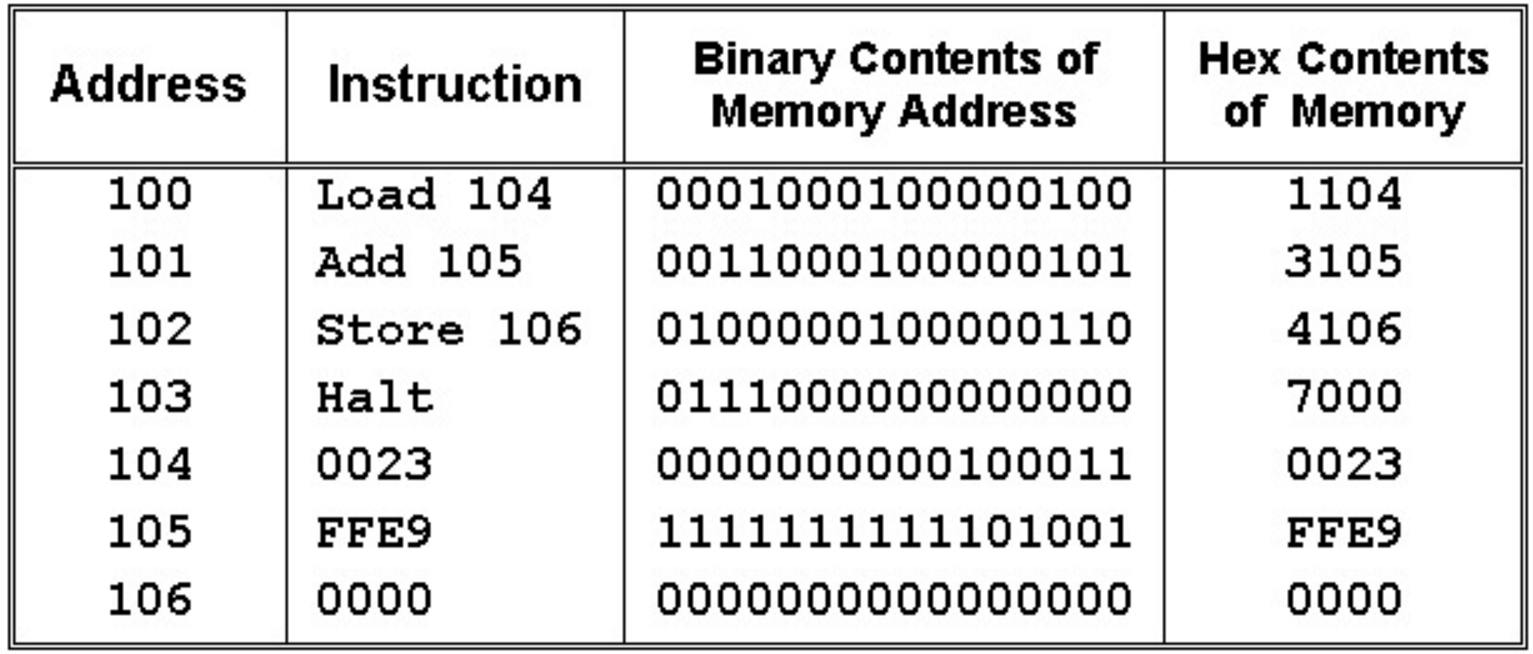
Instruction Processing

- Interrupts are very useful in processing I/O
  - Must modern I/O is interrupt driven (but complicated!)
  - *Will explore this further in Chapter 7*
- MARIE is simplified and only uses basic **programmed I/O**
  - All output is placed in an output register (OutREG)
  - CPU polls the input register (InREG) until input is sensed, and then input is copied into the accumulator

MARIE Programming

A Simple Program

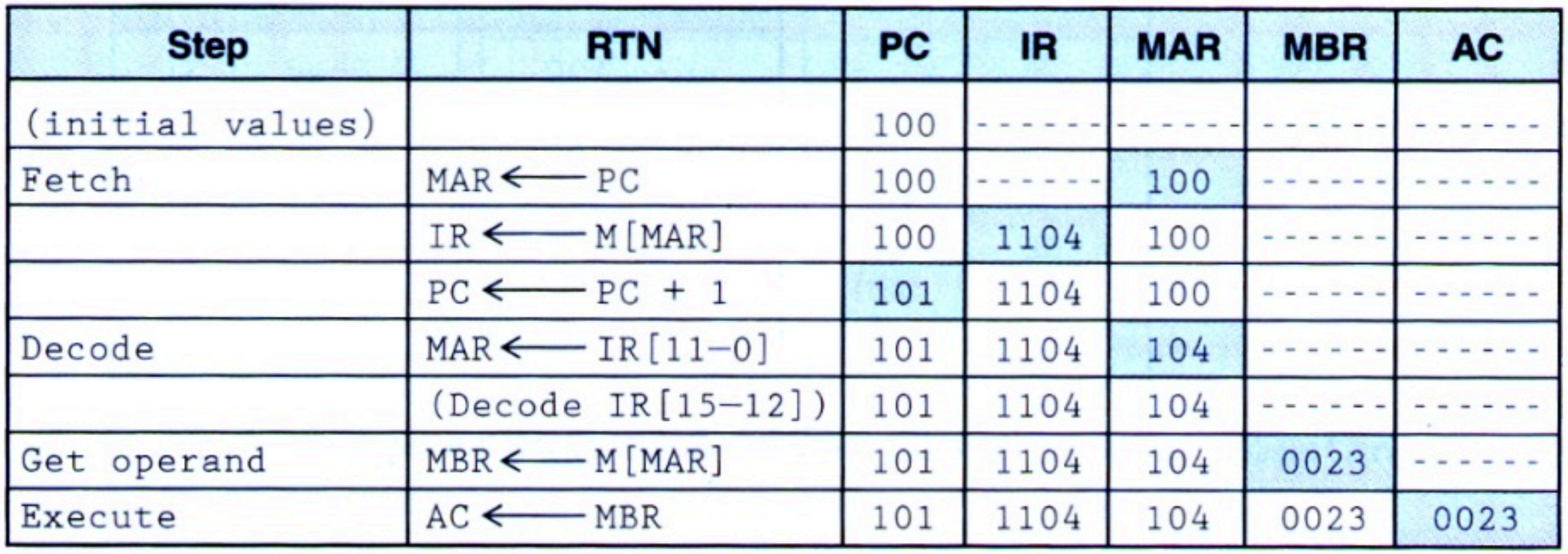
- Consider this simple MARIE program



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

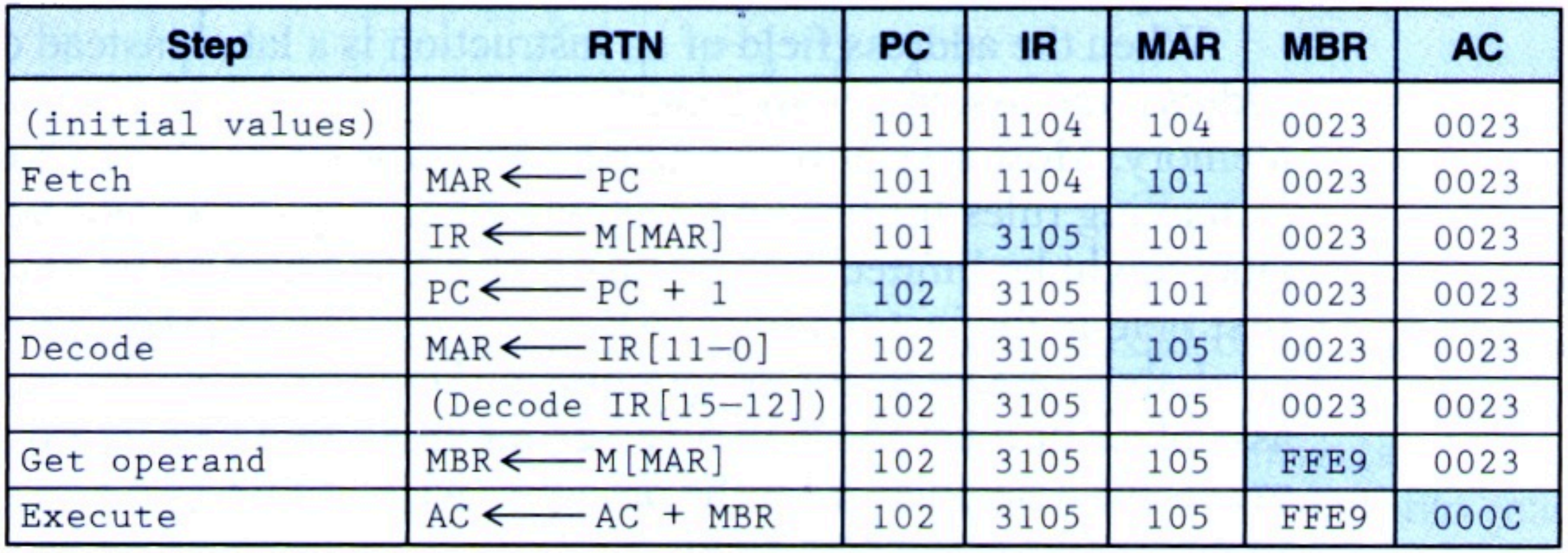
- What happens inside the computer when our program runs?
  - Instruction 1: **LOAD 104**



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

A Simple Program

- Instruction 2: **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