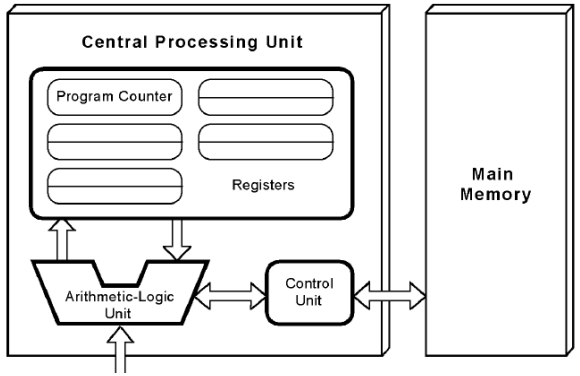
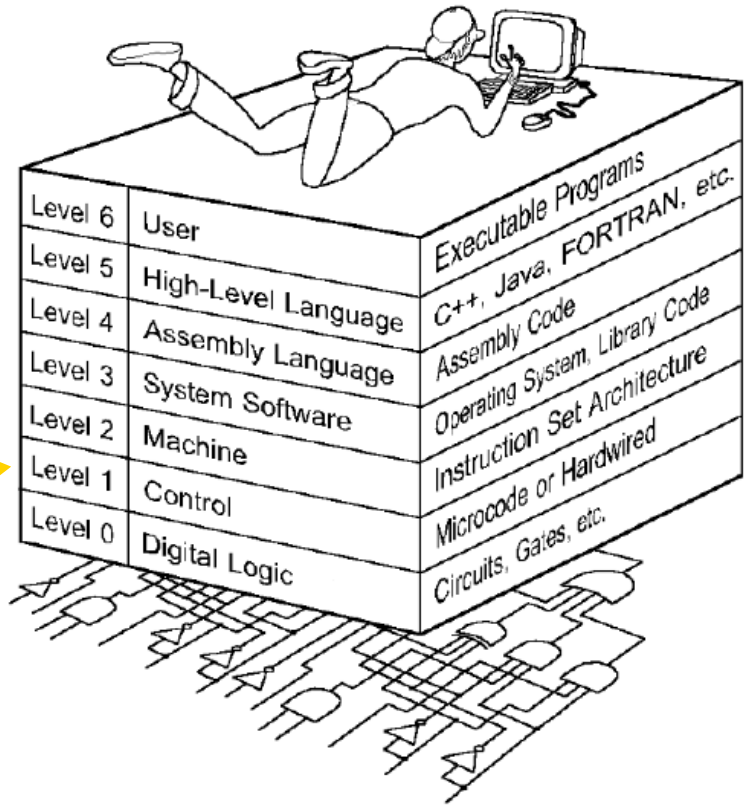
Fetch-Decode-Execute CYCLE
for an instruction's execution



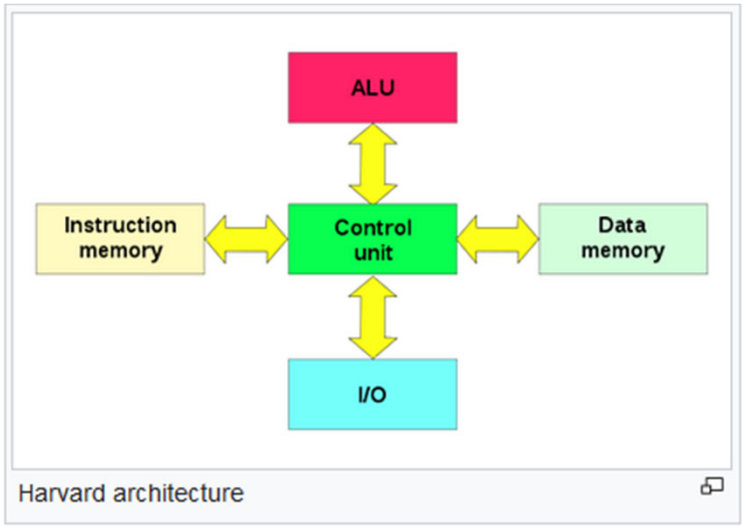


Ch-4 leads (provides middle step for easier understanding) to the Ch-5 (LO-3)

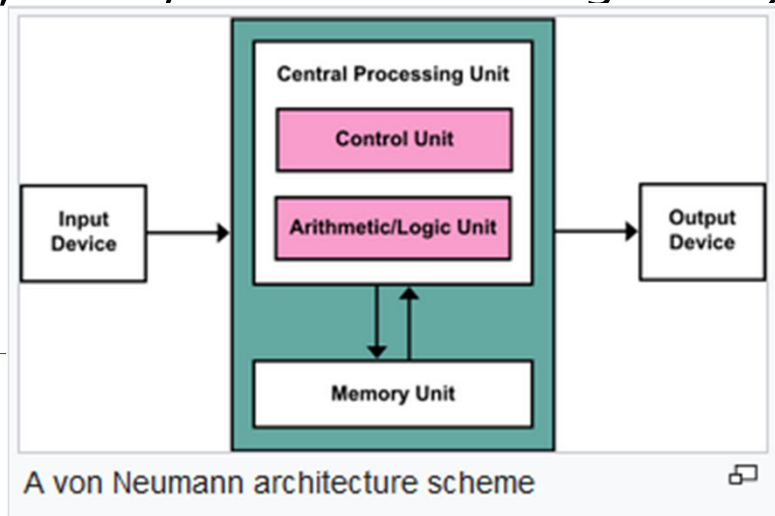
Chapter 4– MARIE: An Intro to Simple Computer

MARIE: Machine Architecture that is Really Intuitive and Easy

- An architecture consisting of **memory** (to store programs and data) and a **CPU** (consisting of an ALU, a CU, and several registers)



Modified-Harvard using dual-cache



[Wikipedia]

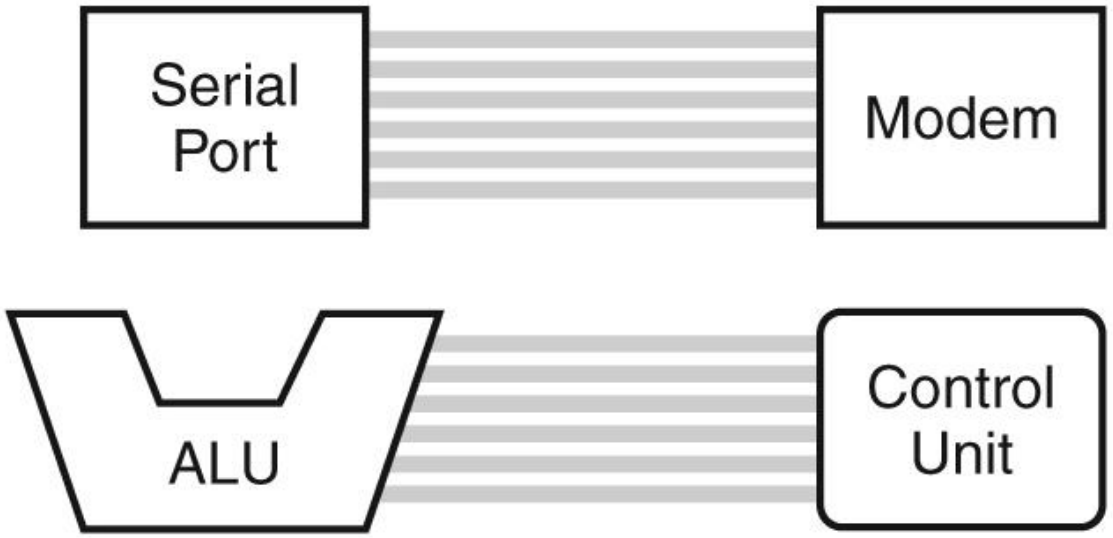
Chapter 4 Objectives

- Learn the components common to every modern computer system.
  - Memory organization and addressing
- Be able to explain how each component contributes to program execution.
- Understand a simple architecture invented to illuminate these basic concepts, and how it relates to some real architectures.
  - MARIE
- Know how the program assembly process works.

4.3 The Bus

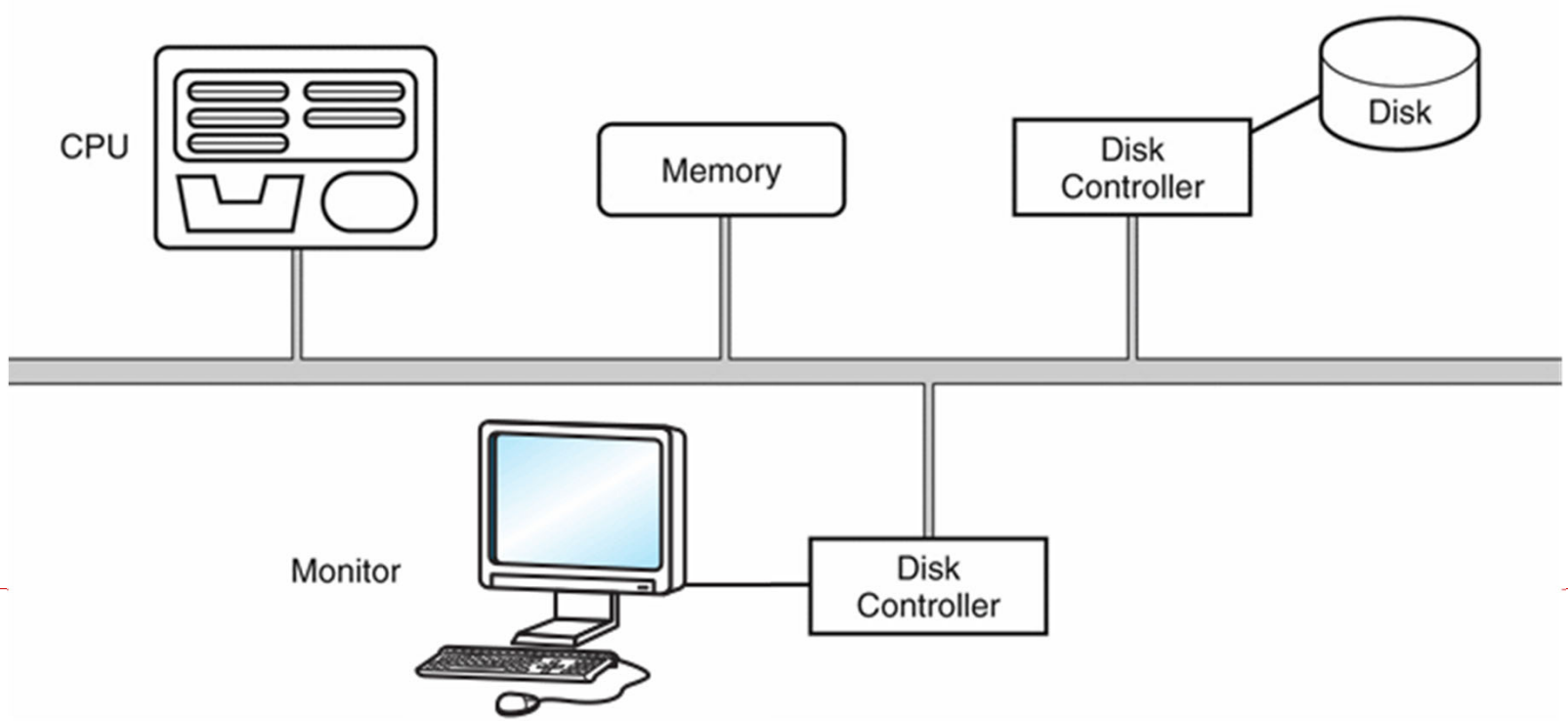
- The CPU shares data with other system components by way of a data bus.
  - A bus is a set of wires that simultaneously convey a single bit along each line.
- Two types of buses are commonly found in computer systems: *point-to-point*, and *multipoint* buses.

These are point-to-point buses:

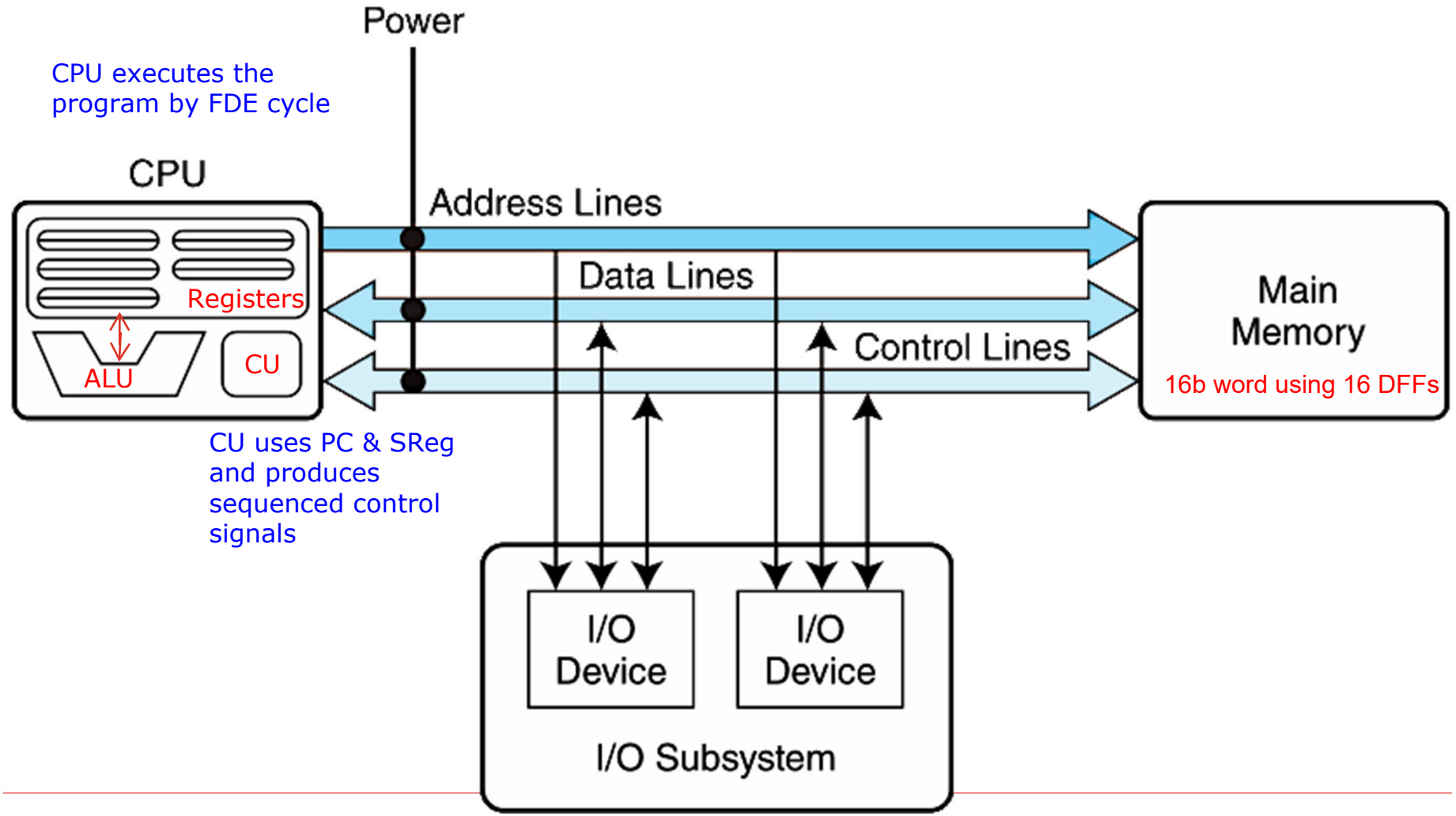


4.3 The Bus

- A multipoint bus is shown below.
- Because a multipoint bus is a **shared** resource, access to it is **controlled** through **protocols**, which are **built into the hardware**.



4.3 The Bus


Power
CPU executes the program by FDE cycle
CPU
Address Lines
Data Lines
Control Lines
Registers
ALU
CU
Main Memory
16b word using 16 DFFs
CU uses PC & SReg and produces sequenced control signals
I/O Device
I/O Device
I/O Subsystem

4.4 Clocks

- Every computer contains **at least one** clock that synchronizes the activities of its components.
- A fixed number of clock cycles are required to carry out each data **movement** or computational **operation.**
- The clock frequency, measured in megahertz or gigahertz, determines the speed with which all operations are carried out.
- Clock cycle time is the reciprocal of clock frequency.
  - An 800 MHz clock has a cycle time of 1.25 ns.

4.4 Clocks

- Clock speed should not be confused with CPU performance.
- The CPU time required to run a program is given by the general performance equation:

SW/Language ISA Clock

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- We see that we can **improve** CPU **throughput**
  - Reduce the number of instructions in a program
  - Reduce the number of cycles per instruction
  - Reduce the number of nanoseconds per clock cycle

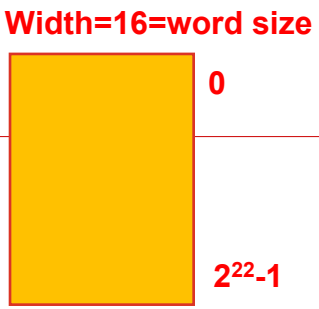
4.5 The Input/Output Subsystem

- A computer communicates with the outside world through its input/output (I/O) subsystem.
- I/O devices connect to the CPU through various interfaces.
  - I/O can be memory-mapped, where the I/O device behaves like main memory from the CPU's point of view.
  - I/O can be instruction-based, where the CPU has a specialized I/O instruction set.

4.6 Memory Organization

- Computer memory consists of a linear array of addressable storage cells that are similar to registers.
- Memory can be byte-addressable, or word-addressable, where a word typically consists of two or more bytes.
- Memory is constructed of RAM chips, often referred to in terms of length × width.
- If the memory word size of the machine is 16 bits, then a 4M ×16 RAM chip gives us 4 megawords of 16-bit at each memory location.

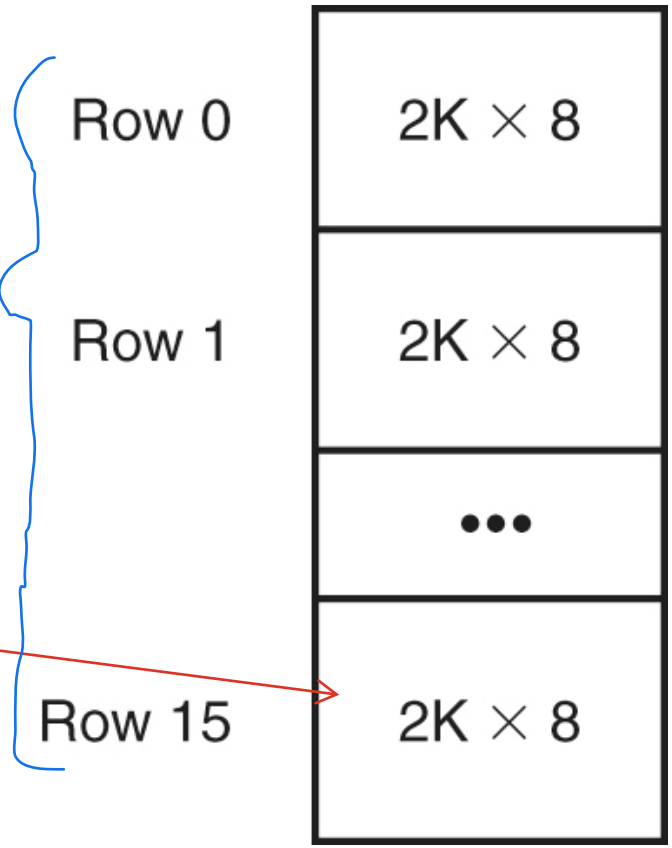
locations word size



$4M=2^2 \times 2^{20} = 2^{22}$

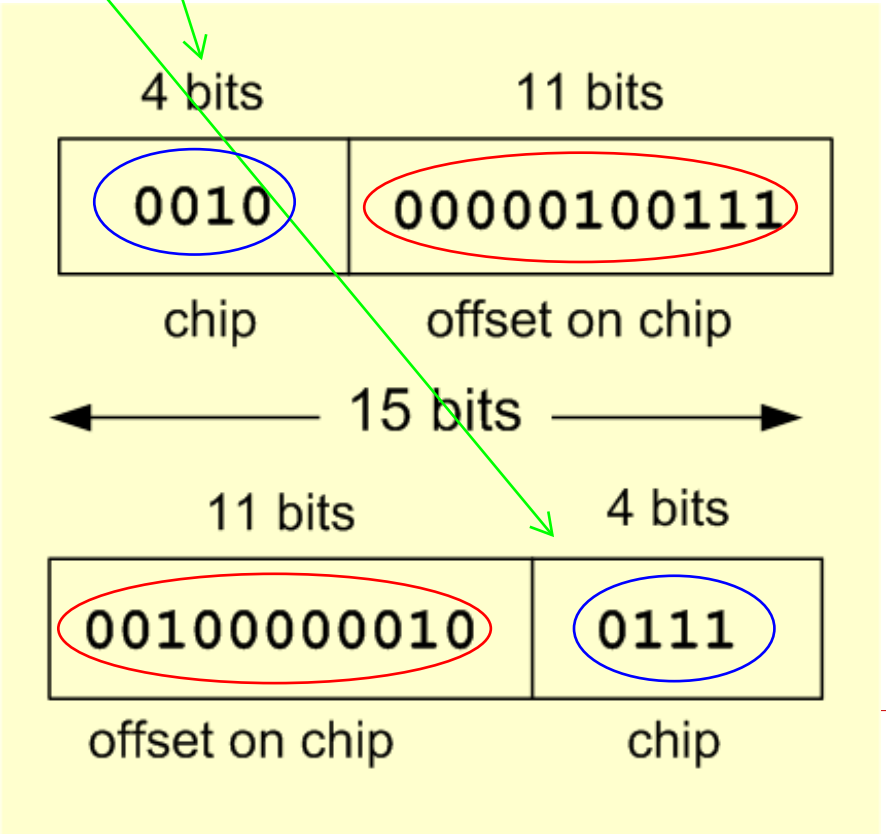
4.6 Memory Organization

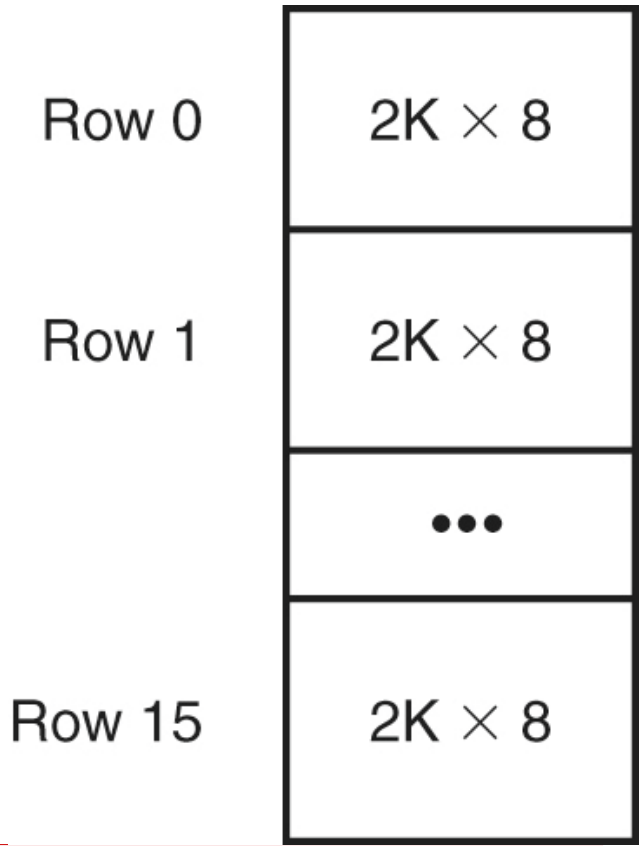
- Example: Suppose we have a memory consisting of 16 chips of 2K x 8 bit each.
  - Memory is 32K = $2^5 \times 2^{10} = 2^{15}$
  - 15 bits are needed for each address.
  - We need 4 bits to select the chip, and 11 bits for the offset into the chip that selects the byte.



4.6 Memory Organization

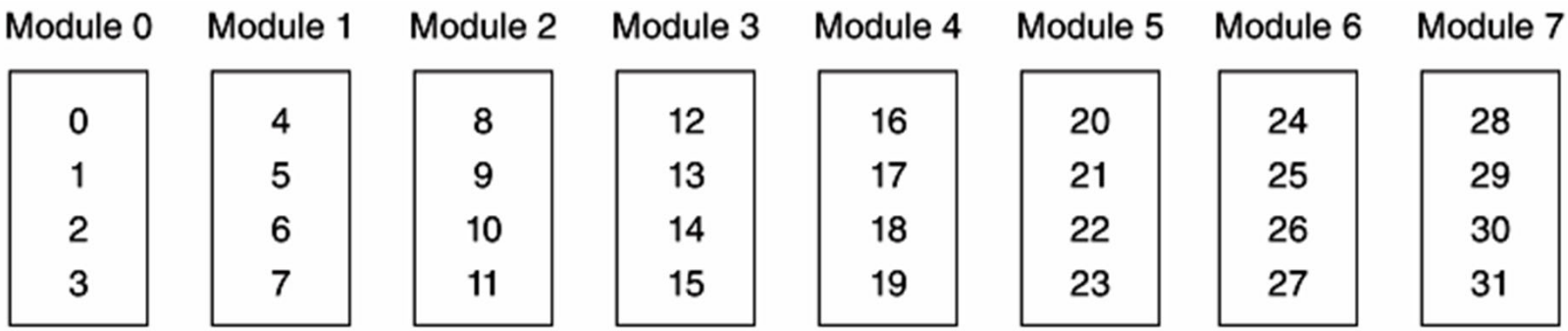
- In high-order interleaving the high-order 4 bits **select the chip**.
- In low-order interleaving the low-order 4 bits **select the chip**.



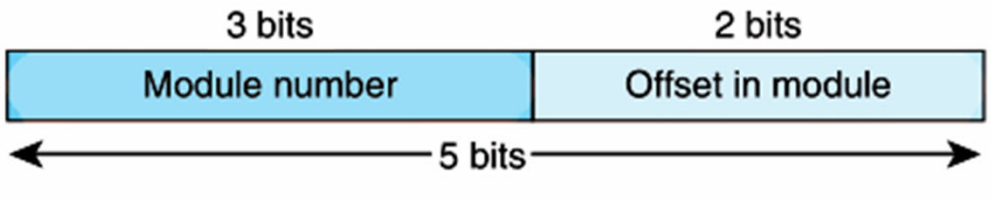


4.6 Memory Organization

a)



b)



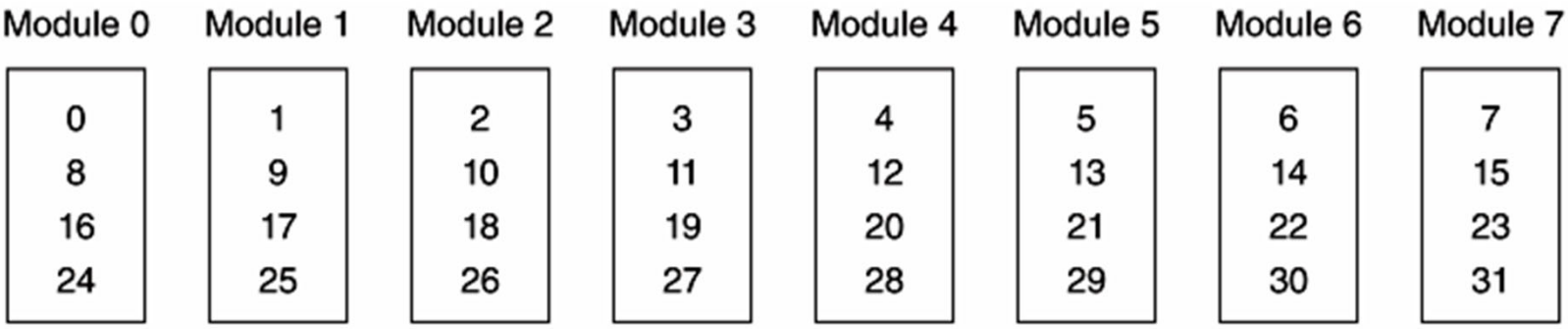
c)

Module	Decimal Word Address	Binary Address	Address Split per Given Structure	Module Number	Offset in Module
Module 0	0	00000	000 00	0	0
	1	00001	000 01	0	1
	2	00010	000 10	0	2
	3	00011	000 11	0	3
Module 1	4	00100	001 00	1	0
	5	00101	001 01	1	1
	6	00110	001 10	1	2
	7	00111	001 11	1	3

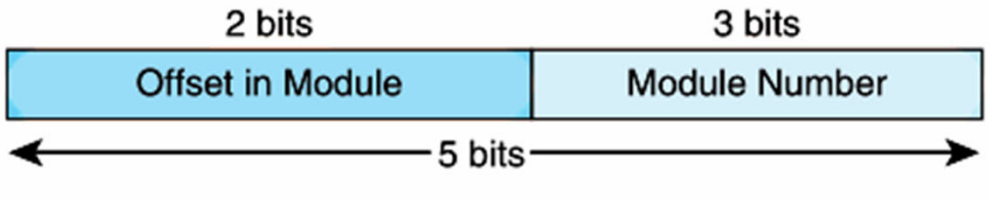
a) High-Order Memory Interleaving b) Address Structure c) First Two Modules

4.6 Memory Organization

a)



b)



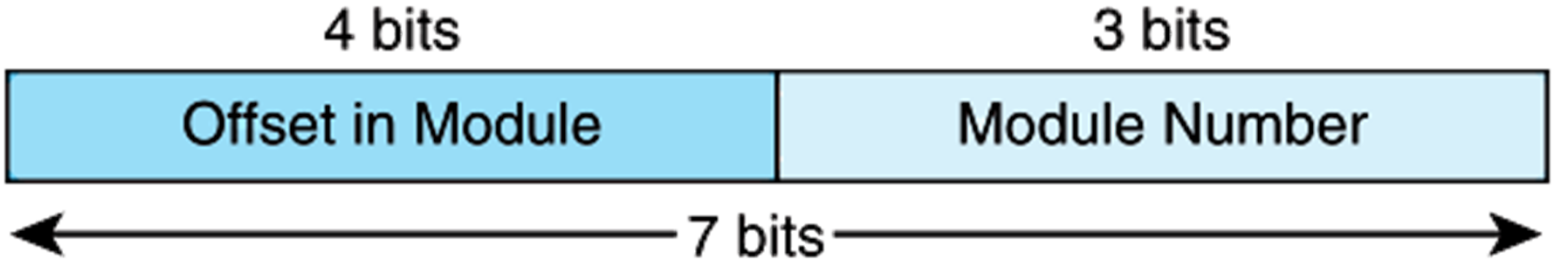
c)

Module	Decimal Word Address	Binary Address	Address Split per Given Structure	Offset in Module	Module Number
Module 0	0	00000	00 000	0	0
	8	01000	01 000	1	0
	16	10000	10 000	2	0
	24	11000	11 000	3	0
Module 1	1	00001	00 001	0	1
	9	01001	01 001	1	1
	17	10001	10 001	2	1
	25	11001	11 001	3	1

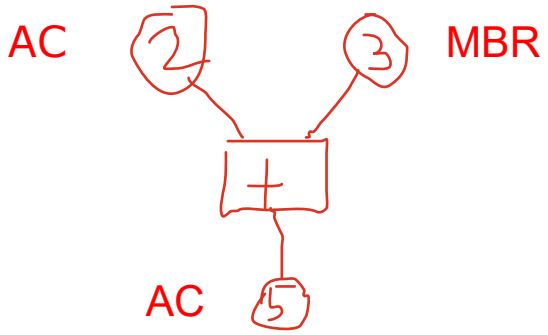
a) Low-Order Memory Interleaving b) Address Structure c) First Two Modules

4.6 Memory Organization

- EXAMPLE 4.1 Suppose we have a 128-word memory that is 8-way low-order interleaved
  - which means it uses 8 memory banks; $8 = 2^3$
- So we use the low-order 3 bits to identify the bank.
- Because we have 128 words, we need 7 bits for each address ($128 = 2^7$).



4.8 MARIE

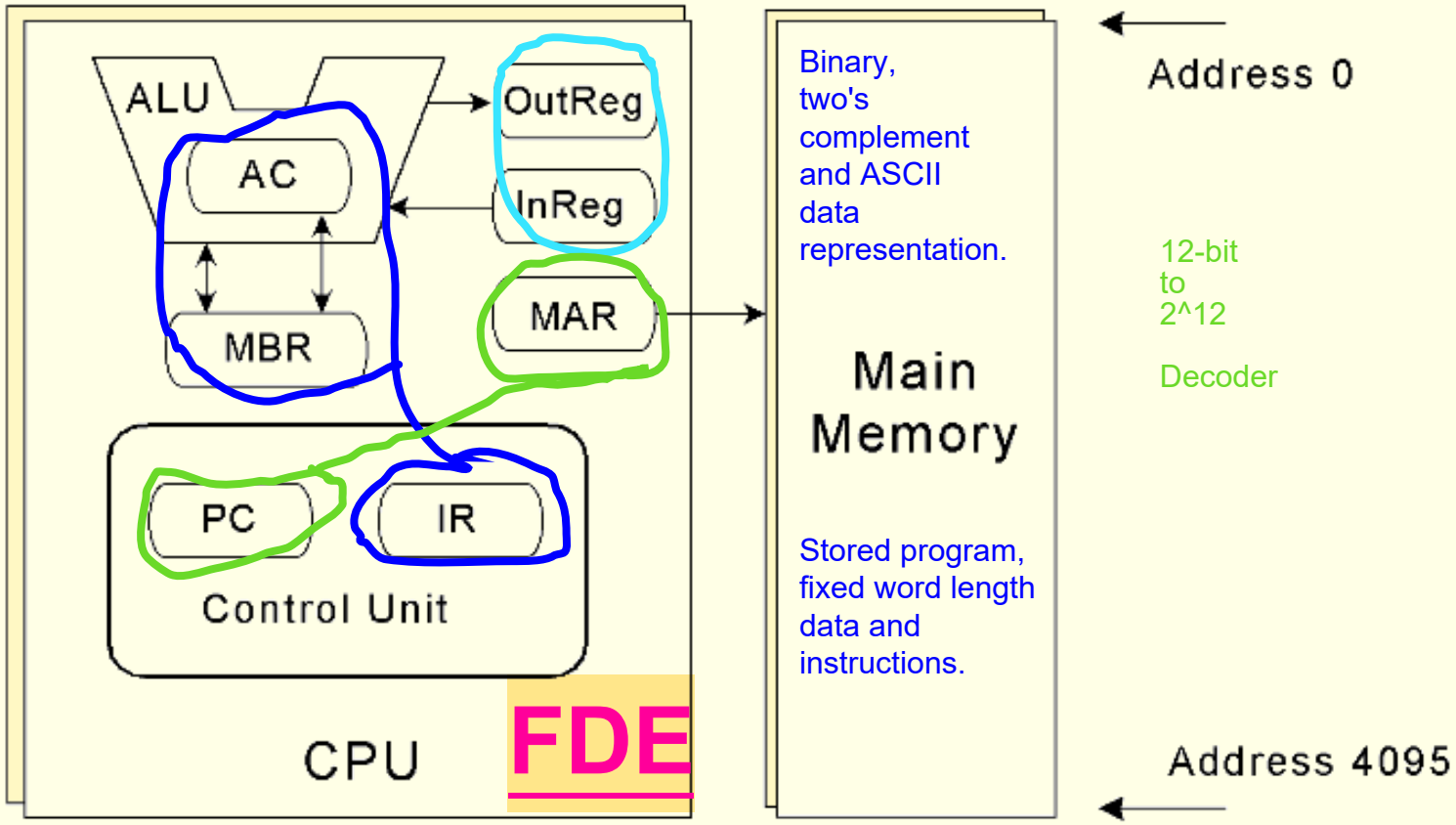


This is the MARIE architecture shown graphically.

16bit Data/ Instruction

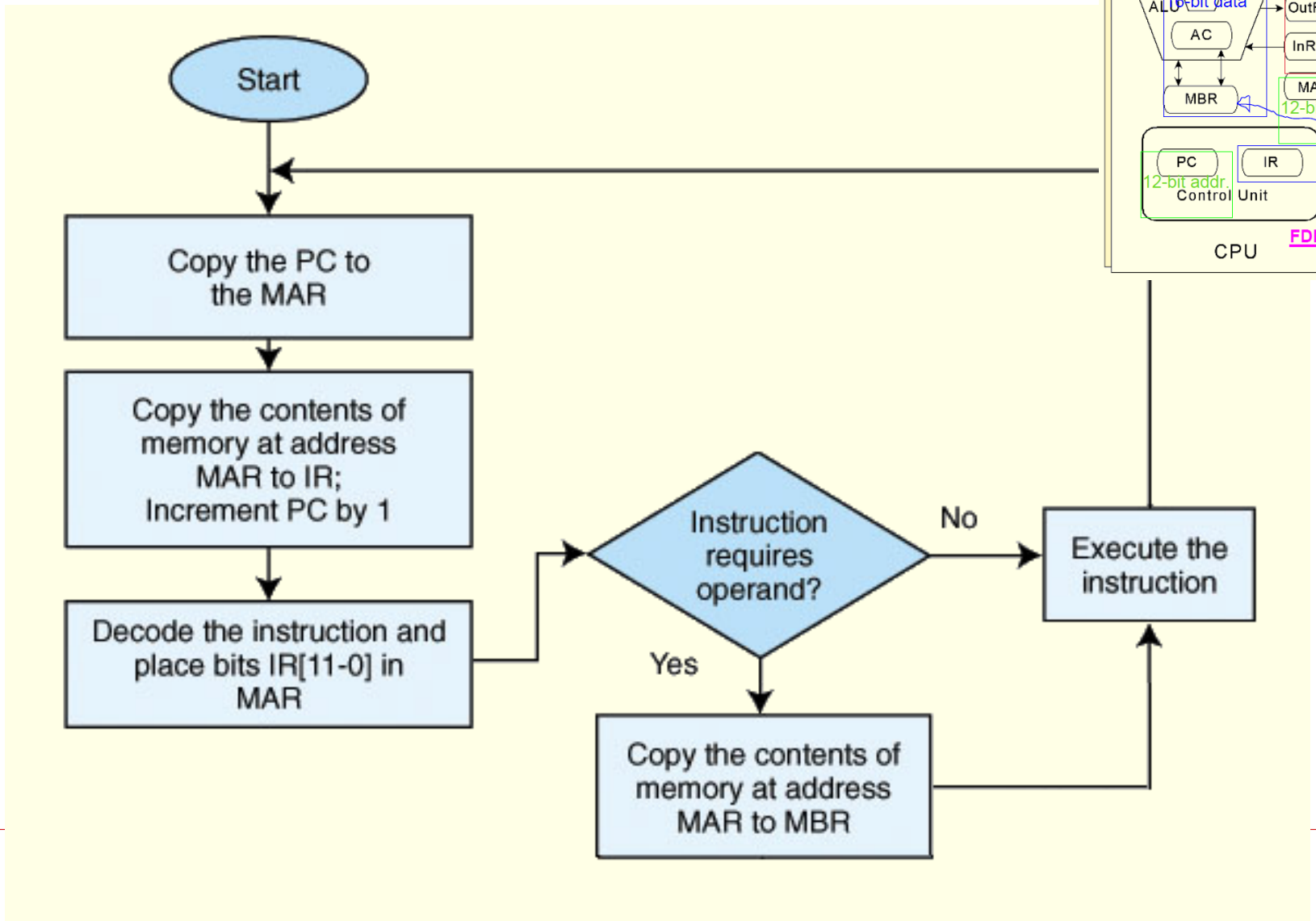
8bit Data

12bit address



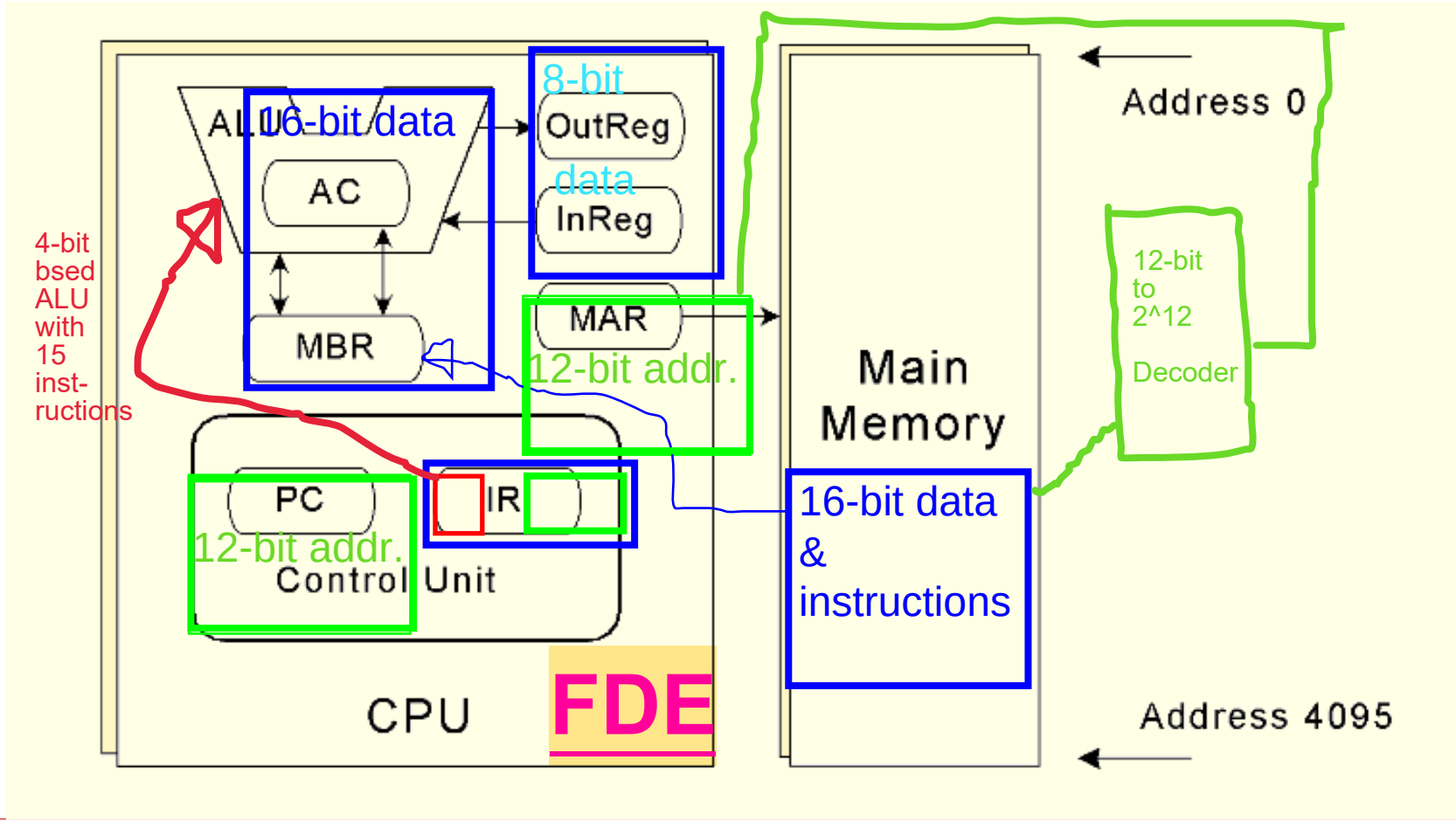
FDE

4.9 Instruction Processing



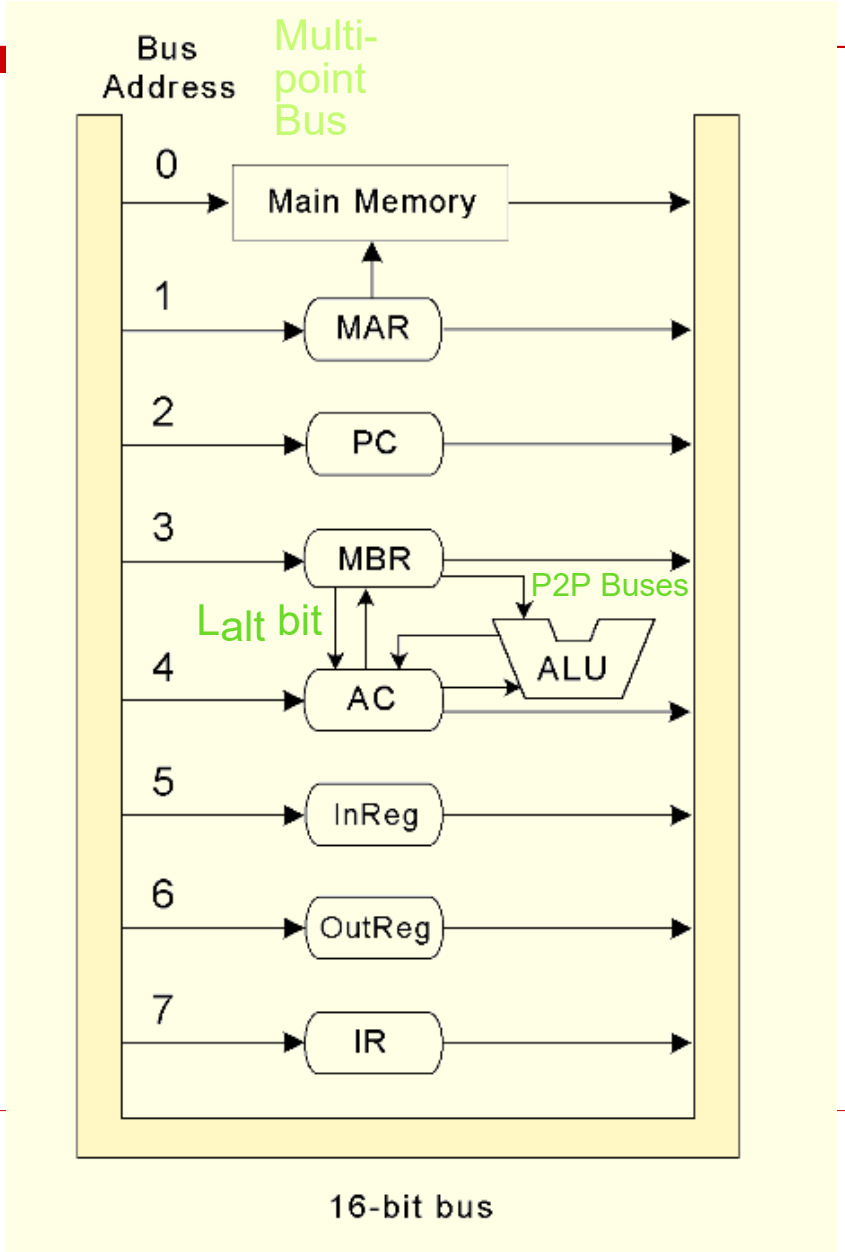
4.8 MARIE

This is the MARIE architecture shown graphically.



4.8 MARIE

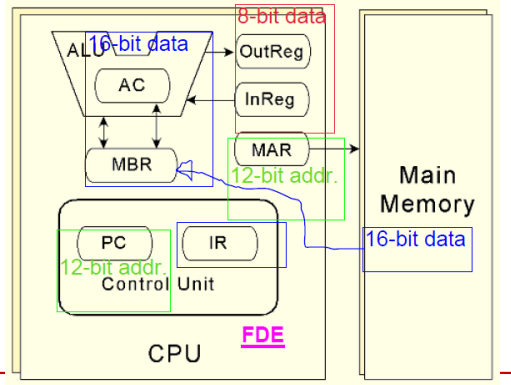
This is the MARIE data path shown graphically.



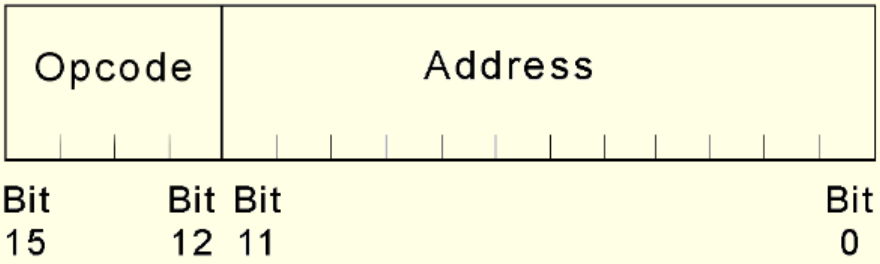
4.8 MARIE

- 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 **manipulating data** and controlling **program execution**.
- The MARIE ISA consists of only fifteen **instructions**.

4.8 MARIE



- This is the format of a MARIE instruction:



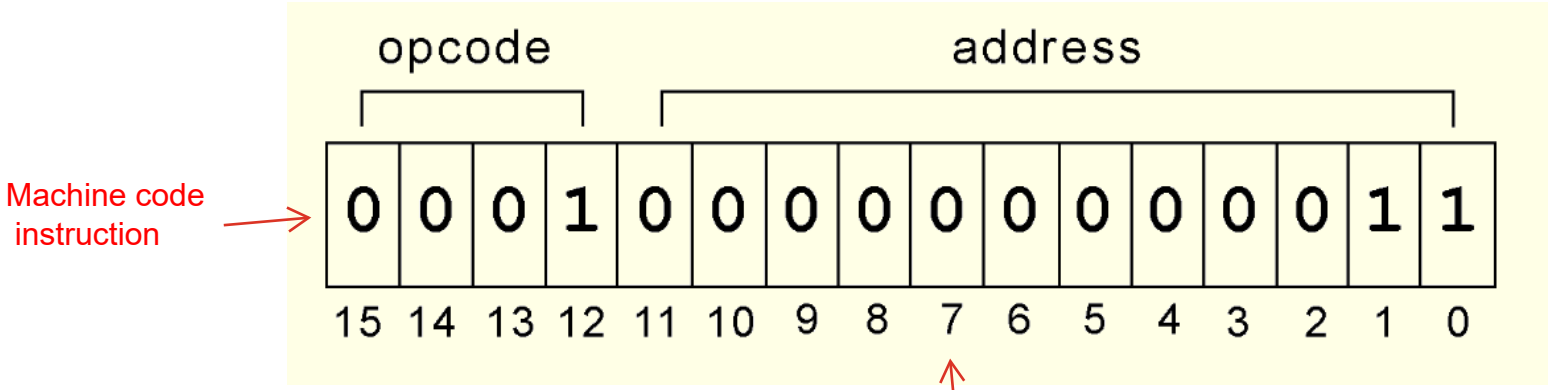
- The **fundamental MARIE instructions** are:

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.

4.8 MARIE

Binary	Hex	Instruction
0001	1	Load X

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

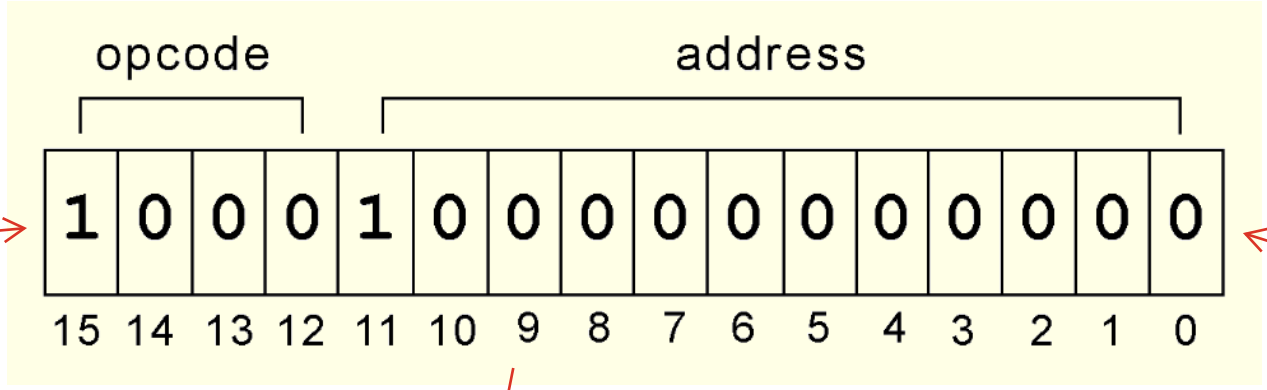


- We see that the opcode is 1 and the address from which to load the data is 3.

4.8 MARIE

1000	8	Skipcond

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



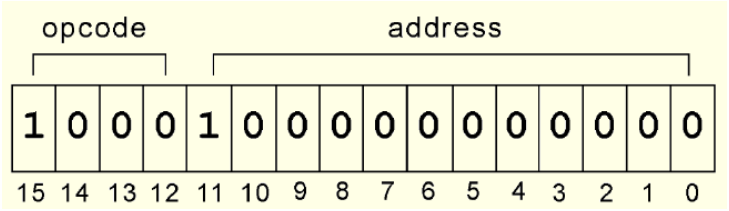
- We see that the opcode is 8 and **bits 11 and 10 are 10**, meaning that the **next instruction will be skipped if** the value in the AC is greater than zero.

What is the hexadecimal representation of this instruction?

8800

1000	8	Skipcond

4.8 MARIE



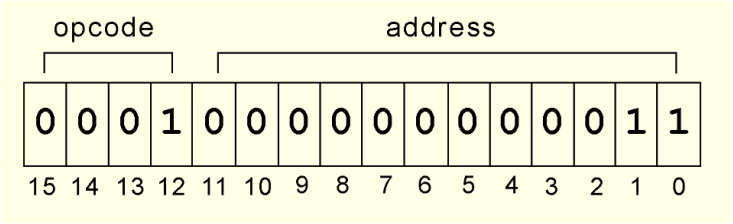
- Recall that **SKIPCOND** skips the next instruction according to the value of the AC.
- The RTL for the this instruction is the most complex in our instruction set:

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

e.g. Compare instruction in Assembly programming language: (which helps implement "if-else" like condition structure)
if x>y {skip next instruction}, else {dont skip or continue execution}
CMP instruction carries out AC=x-y

4.8 MARIE

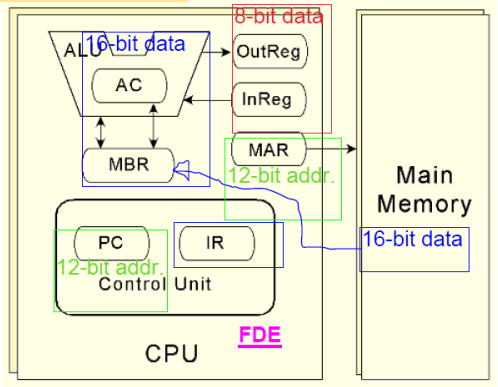
Load 3



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

sequence of microoperations executed by CU during FDE cycle

```
MAR ← X
MBR ← M[MAR]
AC ← MBR
```



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

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

4.7 Interrupts

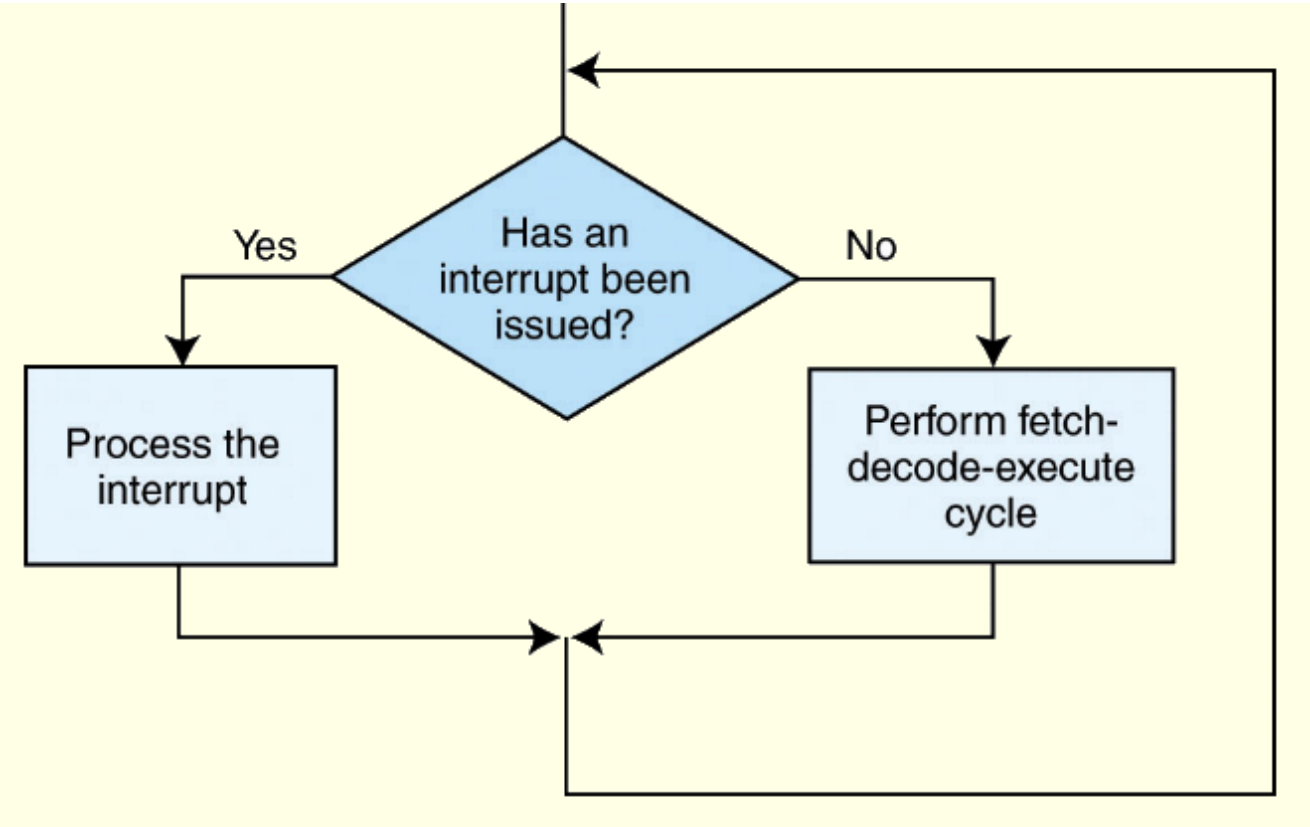
- The normal **execution** of a program is altered when an event of higher-priority occurs. The CPU is alerted to such an event through an interrupt.
- Interrupts can be triggered by
  - I/O **requests**
  - arithmetic **errors** (such as division by zero)
  - when an **invalid** instruction is encountered
- Each interrupt is associated with a procedure that directs the actions of the CPU when an interrupt occurs.
  - Nonmaskable interrupts are high-priority interrupts that cannot be ignored.

4.9 Interrupt Processing

- All computers provide a way of interrupting **the FDE** fetch-decode-execute cycle.
- 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*.

4.9 Interrupt Processing

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

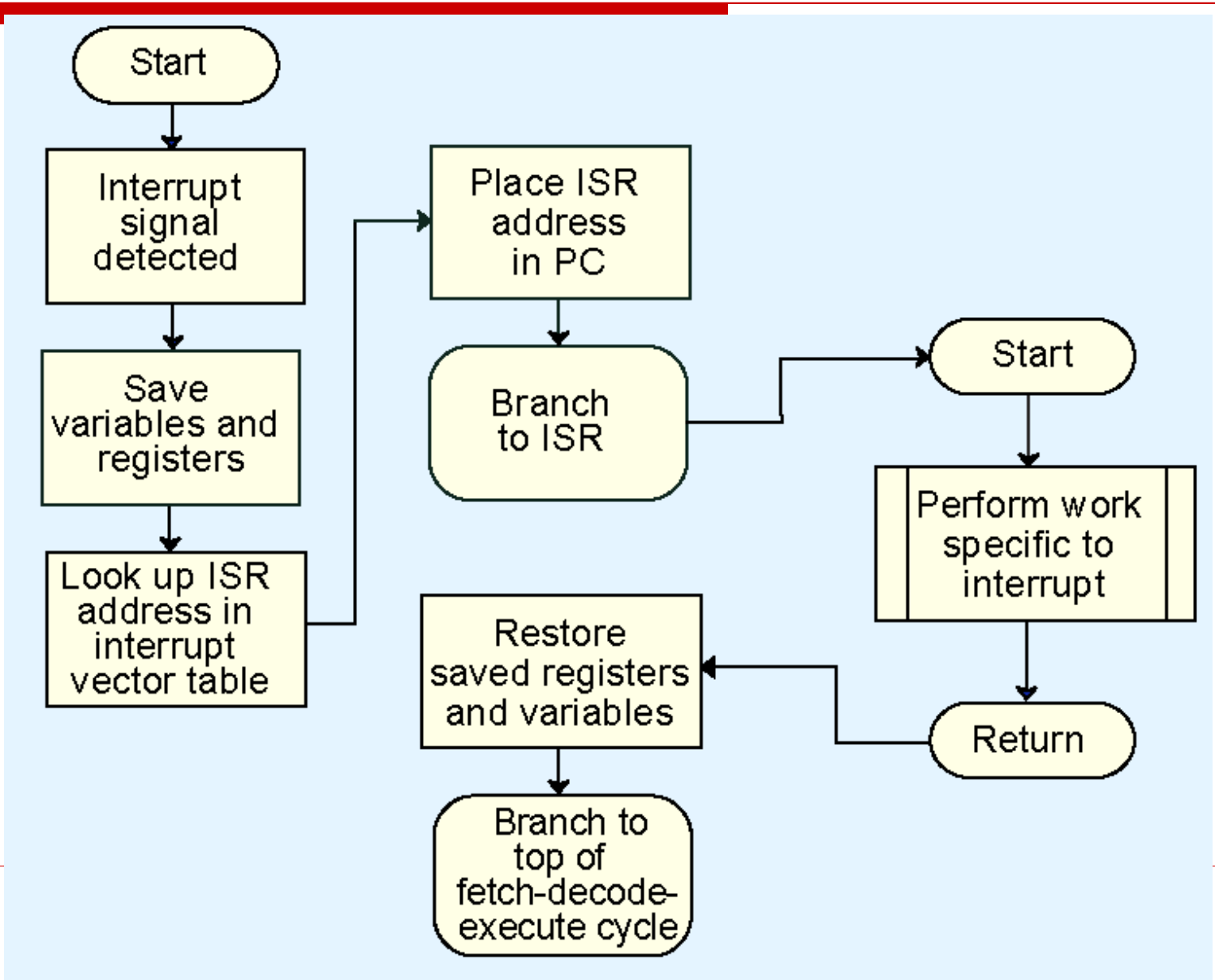


The next slide shows a flowchart of "Process the interrupt."

4.9 Interrupt 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 the I-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.

4.9 Interrupt Processing



4.9 I/O 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.
  - Greater detail in Chapter 7 (will be covered later).
- MARIE, 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**.

4.10 A Simple Pro

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.

- Consider the simple MARIE 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	0010000100000110	2106	
103	Halt	0111000000000000	7000	
104	0023	0000000000100011	0023	X
105	FFE9	1111111111101001	FFE9	Y
106	0000	0000000000000000	0000	Z

0010

2

Program: 0x0023 + 0xFFE9 = Z @ 0x106 ==> 35+(-23)=Z

4.10 A Simple Program

Address	Instruction	Hex Contents of Memory
100	Load 104	1104
101	Add 105	3105
102	Store 106	4106
103	Halt	7000
104	0023	0023
105	FFE9	FFE9
106	0000	0000

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

MAR ← X
MBR ← M[MAR]
AC ← MBR

Step		RTN	PC	IR	MAR	MBR	AC
(initial values)			100	------	------	------	------
Fetch	T0	MAR ← PC	100	------	100	------	------
	T1	IR ← M[MAR]	100	1104	100	------	------
	T2	PC ← PC + 1	101	1104	100	------	------
Decode	T3	MAR ← IR[11–0]	101	1104	104	------	------
		(Decode IR[15–12])	101	1104	104	------	------
Get operand	T4	MBR ← M[MAR]	101	1104	104	0023	------
Execute	T5	AC ← MBR	101	1104	104	0023	0023

4.10 A Simple Program

Address	Instruction	Hex Contents of Memory
100	Load 104	1104
101	Add 105	3105
102	Store 106	4106
103	Halt	7000
104	0023	0023
105	FFE9	FFE9
106	0000	0000

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

MAR ← X
MBR ← M[MAR]
AC ← AC + MBR

Step		RTN	PC	IR	MAR	MBR	AC
(initial values)			101	1104	104	0023	0023
Fetch	T0	MAR ← PC	101	1104	101	0023	0023
	T1	IR ← M[MAR]	101	3105	101	0023	0023
	T2	PC ← PC + 1	102	3105	101	0023	0023
Decode	T3	MAR ← IR[11–0]	102	3105	105	0023	0023
		(Decode IR[15–12])	102	3105	105	0023	0023
Get operand	T4	MBR ← M[MAR]	102	3105	105	FFE9	0023
Execute	T5	AC ← AC + MBR	102	3105	105	FFE9	000C

4.11 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**. Usually, this is **not the case with compilers.**

Address	Mnemonic Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0010000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	1111111111101001	FFE9
106	0000	0000000000000000	0000

4.11 A Discussion

Mnemonic

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

- 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.

Mnemonic Source Code

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

Instructions

Data

Address	Instruction	Hex Contents of Memory
100	Load 104	1104
101	Add 105	3105
102	Store 106	4106
103	Halt	7000
104	0023	0023
105	FFE9	FFE9
106	0000	0000

X	104
Y	105
Z	106

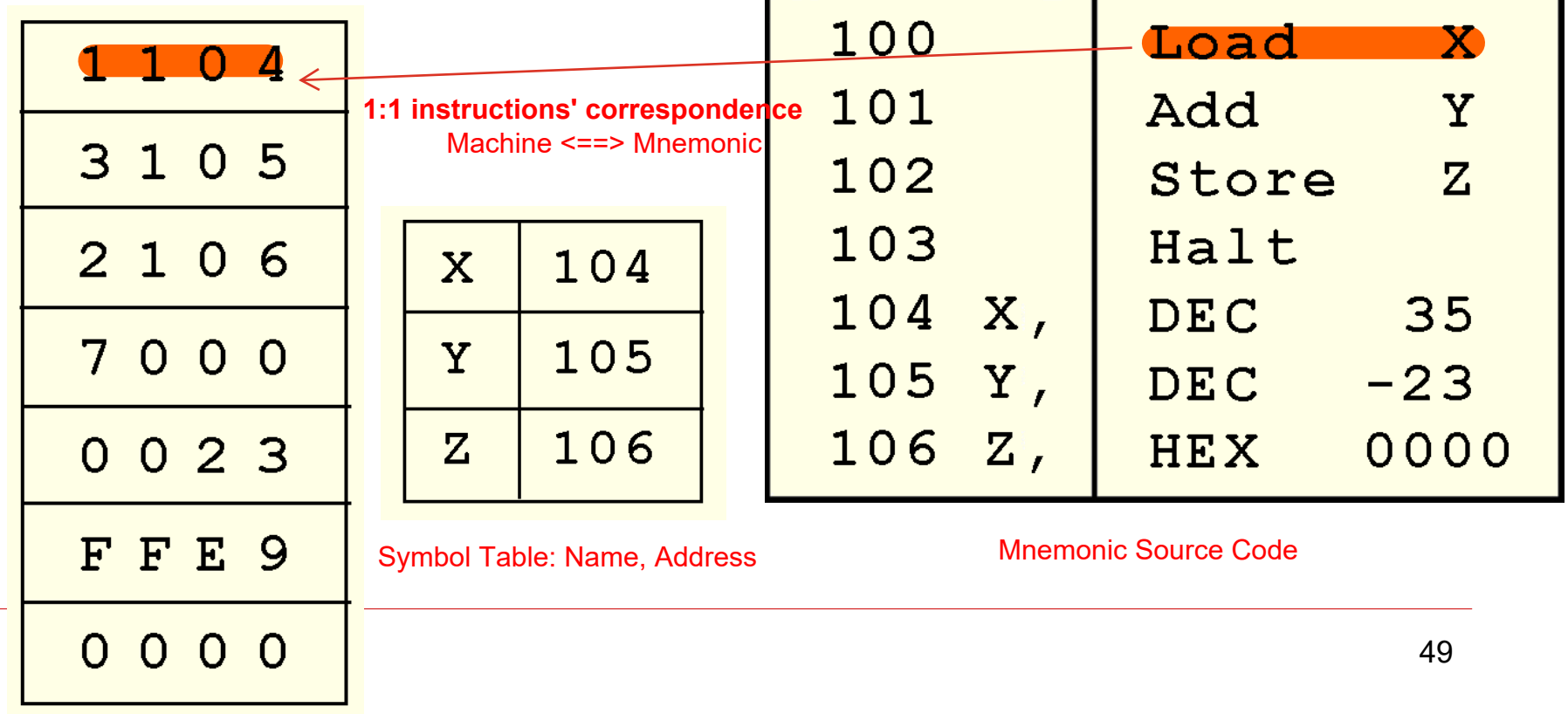
Symbol Table:
Symbol w/ memory reference

1	X
3	Y
2	Z
7	0 0 0

Partially-assembled instructions

4.11 A Discussion on Assemblers

- After the **second pass**, the assembly is complete.



Load X has Memory Reference 'X' ---> Direct addressing

LoadI X has a pointer 'X'

4.12 Extending Our Instruction Set

- We have included three **indirect addressing mode instructions in the MARIE instruction set**.
- The first two are **`LOADI X`** and **`STOREI X`** where '**I**' specifies the address of the operand to be loaded or stored.
  - In RTL :

RTL for the `LOAD 'X'`

```
MAR ← X
MBR ← M[MAR]
AC ← MBR
```

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

`LOADI X`

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

`STOREI X`

4.12 Extending Our Instruction Set

- The **ADDI** instruction is a combination of **LOADI X** and **ADD X**:
- In RTL:

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

ADDI X

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

LOADI X

arly, the RTL for the **ADD 'X'**

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

4.12 Extending Our Instruction Set

- Another helpful programming tool is the use of subroutines (like functions).

1001	9	Jump X	Load the value of X into PC.

- The jump-and-store instruction, **JNS, gives us limited subroutine functionality**. The details of the **JNS** instruction are given by the following RTL:

First, stores PC @ X, then fetches what is at X+1 as the next instruction for the program execution, i.e.

```
MBR ← PC
MAR ← X
M[MAR] ← MBR
MBR ← X
AC ← 1
AC ← AC + MBR
PC ← AC
```

X=PC

PC=X+1

Does JNS permit recursive calls?

No, cant repeat, will jumble up the original execution when it is "rollling-back" the recursion.

4.12 Extending Our Instruction Set

- Our first new instruction is the **`CLEAR`** instruction.
- All it does is set the contents of the **accumulator to all zeroes.**
- This is the RTL for **`CLEAR`**:

`AC` ← `0`

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.

Instruction Number (hex)	Instruction	Meaning
0	JnS *X*	Store the PC at address X and jump to X + 1.
A	Clear	Put all zeros in AC.
B	AddI X	Add indirect: Go to address X. Use the value at X as the actual address of the data operand to add to AC.
C	JumpI X	Jump indirect: Go to address X. Use the value at X as the actual address of the location to jump to.
D	LoadI X	Load indirect: Go to address X. Use the value at X as the actual address of the operand to load into the AC.
E	StoreI X	Store indirect: Go to address X. Use the value at X as the destination address for storing the value in the accumulator.

MARIE's Full Instruction Set

Any instruction's RTL defines the operation of MARIE's control unit.

A microoperation consists of a distinctive signal pattern that is interpreted by the control unit and results in the part-execution of the instruction.

Opcode	Instruction	RTN
0000	JnS *X*	MBR ← PC MAR ← *X* M[MAR] ← MBR MBR ← *X* AC ← 1 AC ← AC + MBR PC ← AC
0001	Load *X*	MAR ← *X* MBR ← M[MAR] AC ← MBR
0010	Store *X*	MAR ← *X*, MBR ← AC M[MAR] ← MBR
0011	Add *X*	MAR ← *X* MBR ← M[MAR] AC ← AC + MBR
0100	Subt *X*	MAR ← *X* MBR ← M[MAR] AC ← AC - MBR
0101	Input	AC ← InREG
0110	Output	OutREG ← AC
0111	Halt	
1000	Skipcond	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
1001	Jump *X*	PC ← IR[11-0]
1010	Clear	AC ← 0
1011	AddI *X*	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← M[MAR] AC ← AC + MBR
1100	JumpI *X*	MAR ← *X* MBR ← M[MAR] PC ← MBR
1101	LoadI *X*	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← M[MAR] AC ← MBR
1110	StoreI *X*	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← AC M[MAR] ← MBR

4.13 A Discussion on Decoding

- A computer's control unit **keeps things synchronized**, making sure that bits flow to the correct components as the components are needed.
- There are two general ways in which a control unit can be implemented: *hardwired control* and *microprogrammed control*.

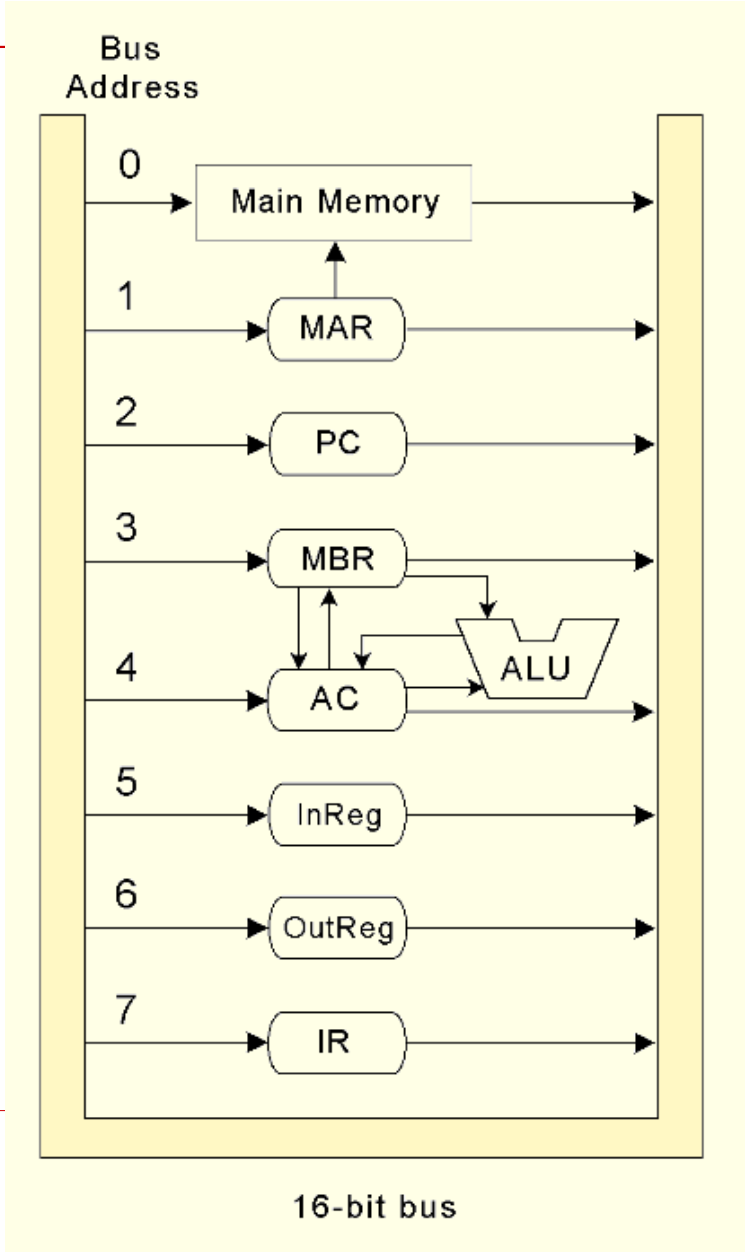
RISC-->
  - Hardwired controllers implement this program using digital logic components.

CISC-->
  - With microprogrammed control, a small program is placed into read-only memory in the microcontroller.

4.13 A Discussion on Decoding

- Each of MARIE's **registers** and main **memory** have a **unique address along the datapath**.
- The addresses take the form of **signals issued by the control unit.** combination

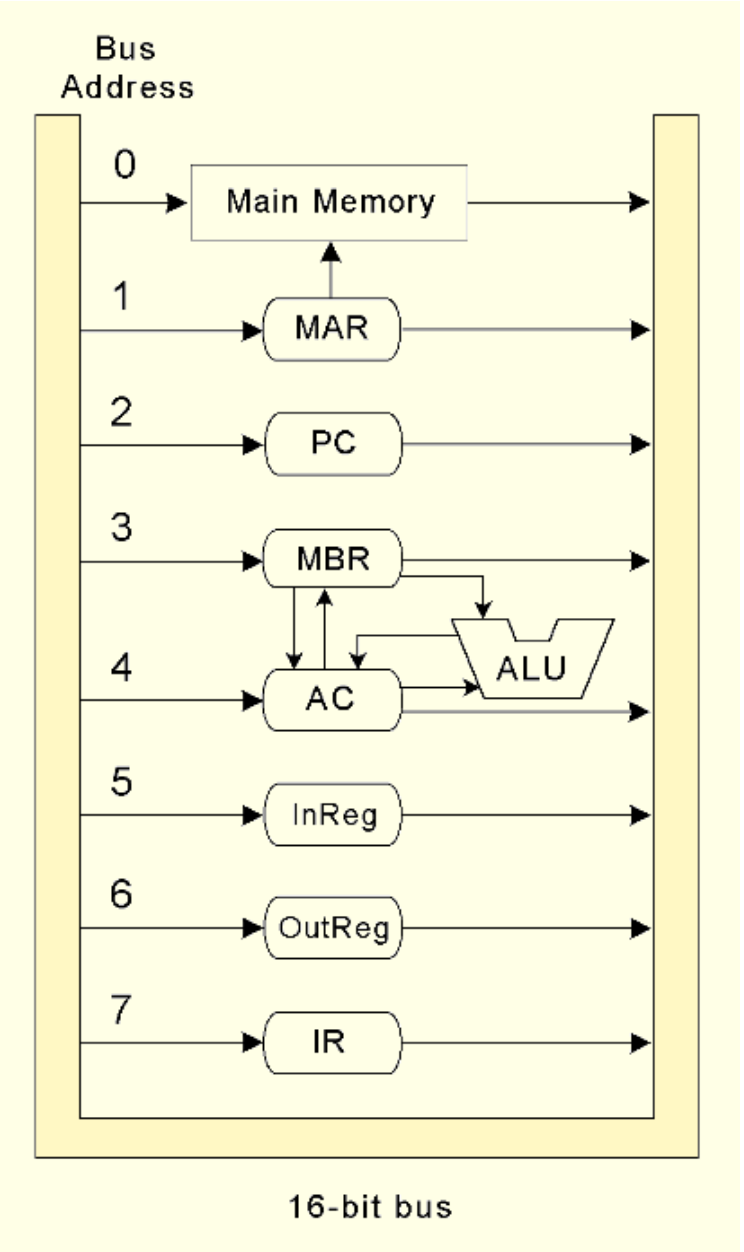
How many signal lines does MARIE's control unit need?



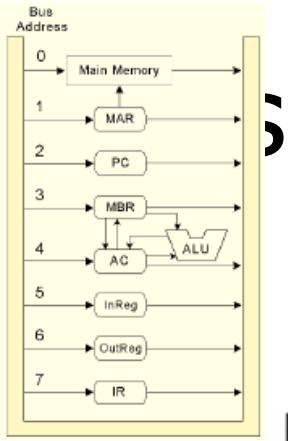
4.13 A Discussion on Decoding

- Let us define two sets of three signals.
- One set, **P_2, P_1, P_0,** controls **reading from** memory or a register, and the other set consisting of **P_5, P_4, P_3,** controls **writing to** memory or a register.

The next slide shows a close up view of MARIE's MBR.



4.13 A Dis sion on Decoding



D_{15} D_{14} • • • D_0

D Q

16
D Flip-Flops

MBR

This register is enabled for reading when P0 and P1 are high, and enabled for writing when P3 and P4 are high

P_5 P_4 P_3

Control Unit

P_2 P_1 P_0

4.13 A Discussion on Decoding

- Careful inspection of MARIE's RTL reveals that the **ALU has only three operations**: add, subtract, and clear. ARE WE SURE?
  - We will also define a **fourth "do nothing"** state.

- The entire set of MARIE's control signals consists of:
  - Register controls: P_0 through P_5, M_R , and M_W.
  - ALU controls: A_0 and A1
  - MBR and AC: L_{ALT} to control the data source. used in Load/Store - Data source is not the bus
  - Timing: T_0 through T_7 and counter reset C_r

ALU Control Signals		ALU Response
A_1	A_0	
0	0	Do Nothing
0	1	AC ← AC + MBR
1	0	AC ← AC – MBR
1	1	AC ← 0 (Clear)

4.13 A

control signals consists of:

- Register controls: P_0 through P_5, M_R , and M_W.
- ALU controls: A_0 and A1
- MBR and AC: L_{ALT} to control the data source.
- Timing: T_0 through T_7 and counter reset C_r

Step		RTN	PC	IR	MAR	MBR	AC
(initial values)			101	1104	104	0023	0023
Fetch	T0	MAR ← PC	101	1104	101	0023	0023
	T1	IR ← M[MAR]	101	3105	101	0023	0023
	T2	PC ← PC + 1	102	3105	101	0023	0023
Decode	T3	MAR ← IR[11–0]	102	3105	105	0023	0023
		(Decode IR[15–12])	102	3105	105	0023	0023
Get operand	T4	MBR ← M[MAR]	102	3105	105	FFE9	0023
Execute	T5	AC ← AC + MBR	102	3105	105	FFE9	000C

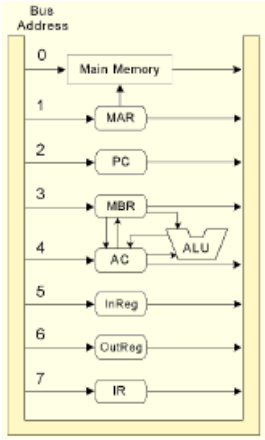
- Here is the complete signal sequence for MARIE's Add instruction:

$$P_3\ P_2\ P_1\ P_0\ T_3 \quad : \quad MAR \leftarrow X$$

$$P_4\ P_3\ T_4\ M_R \quad : \quad MBR \leftarrow M[MAR]$$

$$C_r\ A_0\ P_5\ T_5\ L_{ALT} \quad : \quad AC \leftarrow AC + MBR$$

$$[Reset\ counter]$$

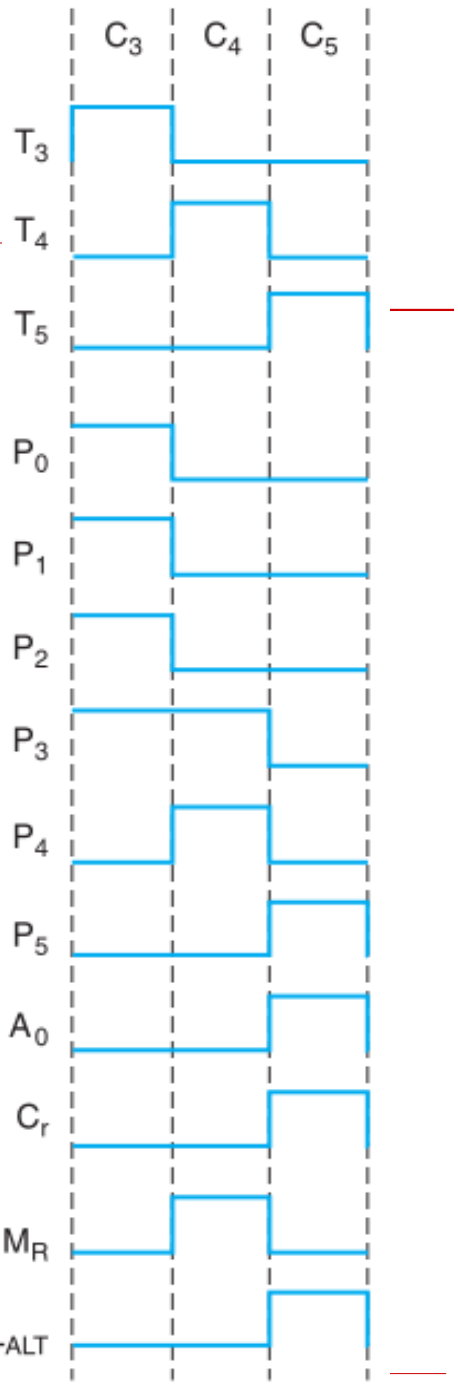


- These signals are ANDed with combinational logic to bring about the desired machine behavior.
- The next slide shows the timing diagram for this instruction.

4.13 Decoding

control signals consists of:

- Register controls: P_0 through P_5, M_R , and M_W.
- ALU controls: A_0 and A1
- MBR and AC: L_{ALT} to control the data source.
- Timing: T_0 through T_7 and counter reset C_r



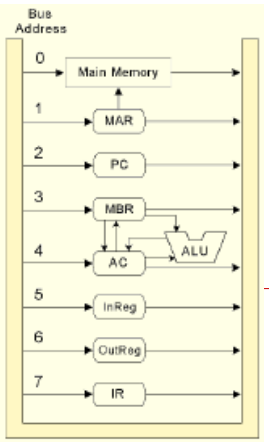
- Notice the concurrent signal states during each machine cycle: C_3 through C_5.

$P_3\ P_2\ P_1\ P_0\ T_3$: MAR ← X

$P_4\ P_3\ T_4\ M_R$: MBR ← M[MAR]

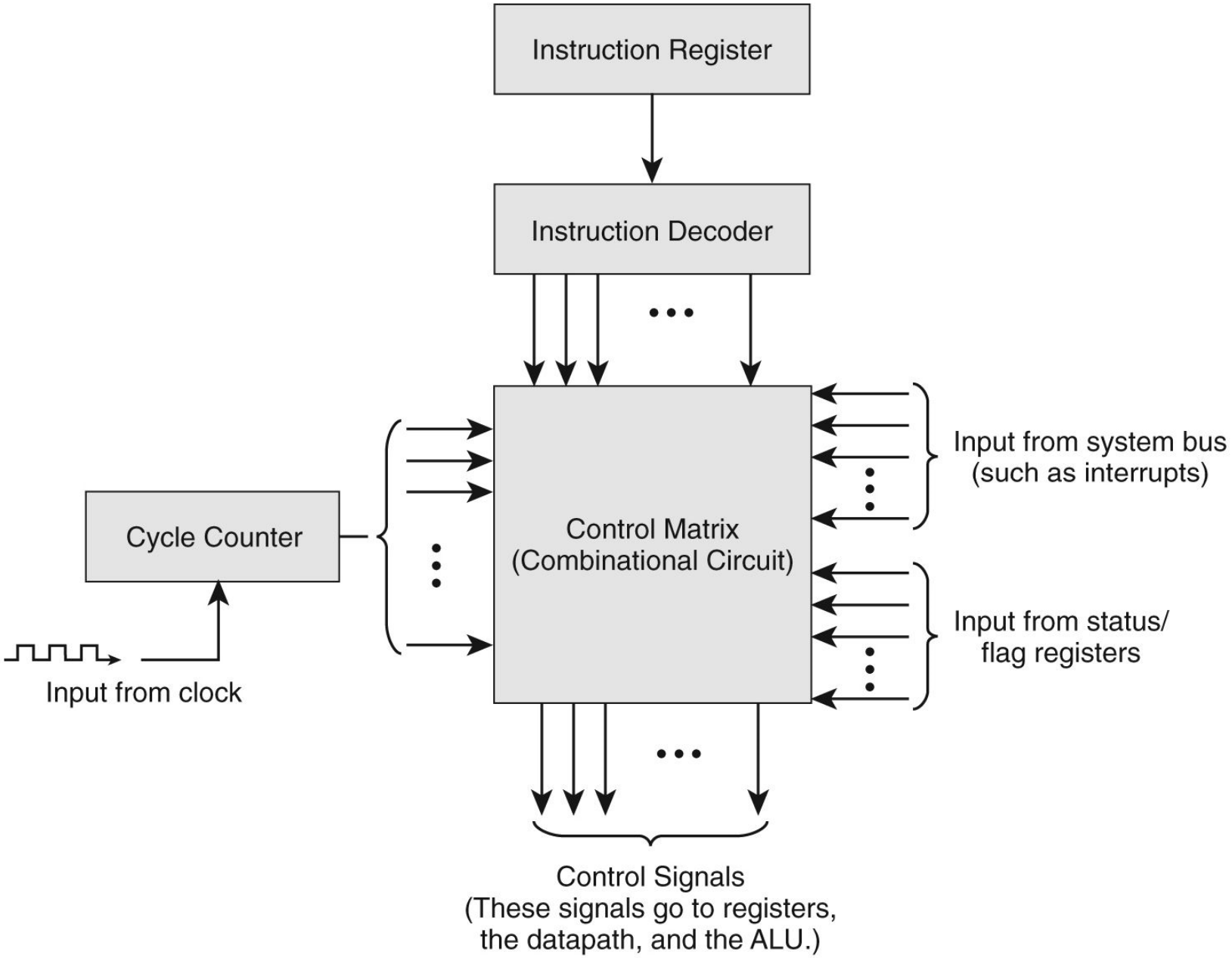
$C_r\ A_0\ P_5\ T_5\ L_{ALT}$: AC ← AC + MBR

[Reset counter]



Step		RTN	PC	IR	MAR	MBR	AC
(initial values)			101	1104	104	0023	0023
Fetch	T0	MAR ← PC	101	1104	101	0023	0023
	T1	IR ← M[MAR]	101	3105	101	0023	0023
	T2	PC ← PC + 1	102	3105	101	0023	0023
Decode	T3	MAR ← IR[11–0]	102	3105	105	0023	0023
		(Decode IR[15–12])	102	3105	105	0023	0023
Get operand	T4	MBR ← M[MAR]	102	3105	105	FFE9	0023
Execute	T5	AC ← AC + MBR	102	3105	105	FFE9	000C

4.13 Hardwired CU: Decoding



4.14 Real World Architectures

- MARIE **shares** many features with modern architectures but it is **not an accurate depiction** of them.
- Two machine architectures:
  - Intel architecture is a CISC machine. CISC is an acronym for complex instruction set computer.
  - MIPS, which is a RISC machine. RISC stands for reduced instruction set computer.

4.14 Real World Architectures

- The classic Intel architecture, the 8086, was born in 1979. It is a CISC architecture.
- It was adopted by IBM for its famed PC, which was released in 1981.
- The 8086 operated on 16-bit data words and supported 20-bit memory addresses.
- Later, to lower costs, the 8-bit 8088 was introduced. Like the 8086, it used 20-bit memory addresses.

What was the largest memory that the 8086 could address?

4.14 Real World Architectures

- In 1985, Intel introduced the 32-bit 80386.
- It also had no built-in floating-point unit.
- The 80486, introduced in 1989, was an 80386 that had built-in floating-point processing and cache memory.
- The 80386 and 80486 offered downward compatibility with the 8086 and 8088.
- Software written for the smaller-word systems was directed to use the lower 16 bits of the 32-bit registers.

4.14 Real World Architectures

- Intel's Pentium 4 introduced a brand new NetBurst architecture.
- Speed enhancing features include:
  - Hyperthreading
  - Hyperpipelining
  - Wider instruction pipeline
  - Execution trace cache (holds decoded instructions for possible reuse) multilevel cache and instruction pipelining.
- Intel, along with many others, is marrying many of the ideas of RISC architectures with microprocessors that are largely CISC.

4.14 Real World Architectures

- MIPS was one of the first RISC microprocessors.
- The original MIPS architecture had only 55 different instructions, as compared with the 8086 which had over 100.
- MIPS was designed with performance in mind: It is a *load/store* architecture, meaning that only the load and store instructions can access memory.
- The large number of registers in the MIPS architecture keeps bus traffic to a minimum.

How does this design affect performance?