
Chapter 4 – MARIE: An Introduction to a 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 and several registers)

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.
-
- 2

4.1 Introduction

- Chapter 1 presented a general **overview** of computer systems.
 - In Chapter 2, we discussed how **data** is stored and manipulated by various computer system components.
 - Chapter 3 described the fundamental components of **digital circuits**.
 - Having this background, we can now understand how computer **components** work, and how they fit **together** to create useful computer systems.
-

4.2 CPU Basics

- The computer's CPU fetches, decodes, and executes program instructions.
- The two principal parts of the CPU are the *datapath* and the *control unit*.
 - The **datapath** consists of an arithmetic-logic unit and storage units (registers) that are interconnected by a data **bus** that is also connected to main memory.
 - Various CPU components perform sequenced operations according to **signals** provided by its **control unit**.

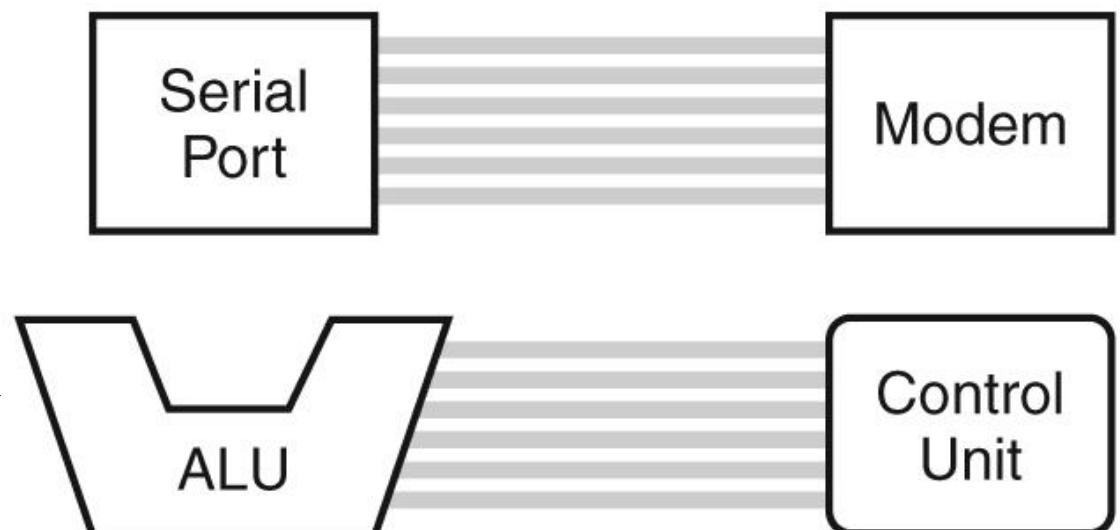
4.2 CPU Basics

- Registers hold data that can be readily accessed by the CPU.
- They can be implemented using D flip-flops.
 - A 32-bit register requires 32 D flip-flops.
- The arithmetic-logic unit (ALU) carries out logical and arithmetic operations as directed by the control unit.
- The control unit determines which actions to carry out according to the values in a program counter register and a status register.

4.3 The Bus

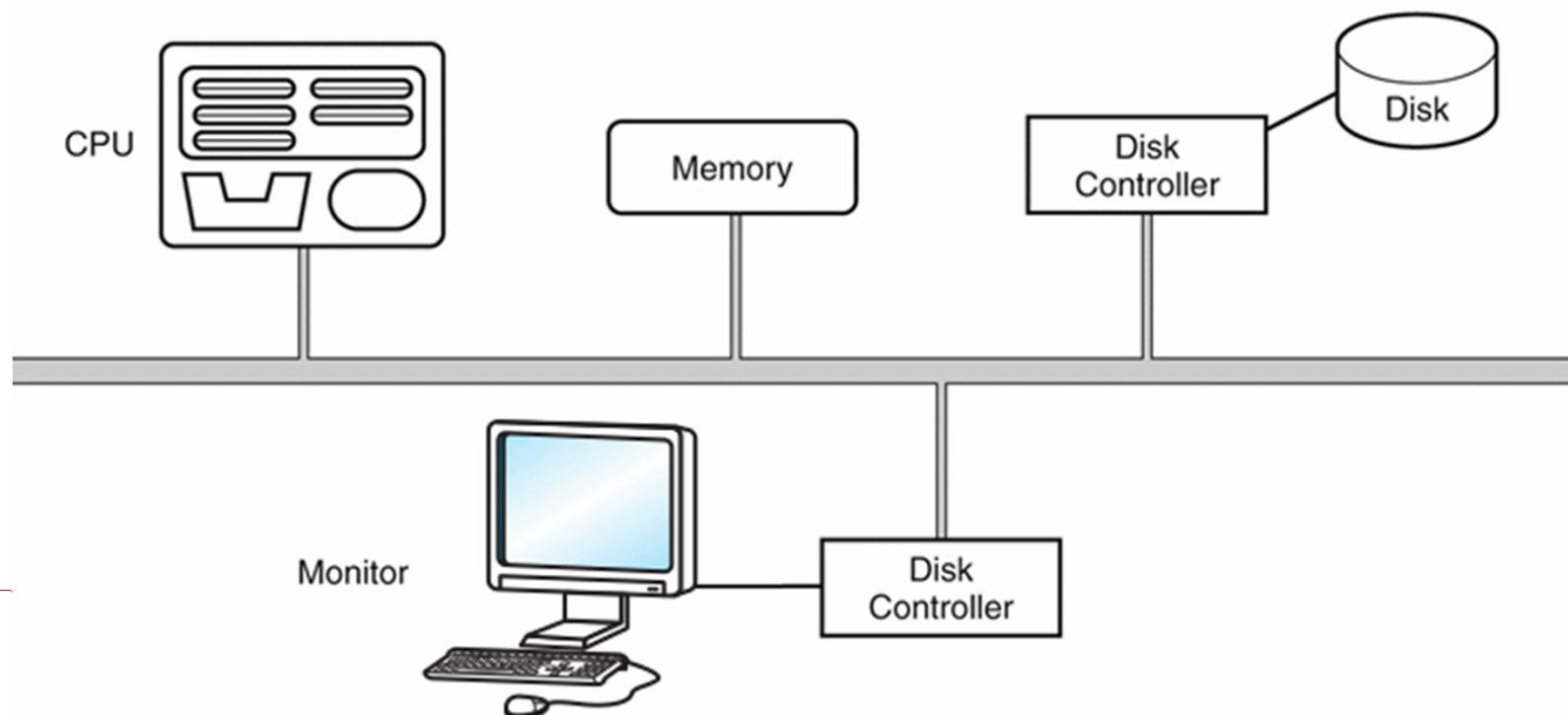
- The CPU shares data with other system components by way of a data bus.
 - A **bus** is a set of wires that simultaneously convey a single bit along each line.
- Two types of buses are commonly found in computer systems: *point-to-point*, and *multipoint* buses.

These are point-to-point buses:



4.3 The Bus

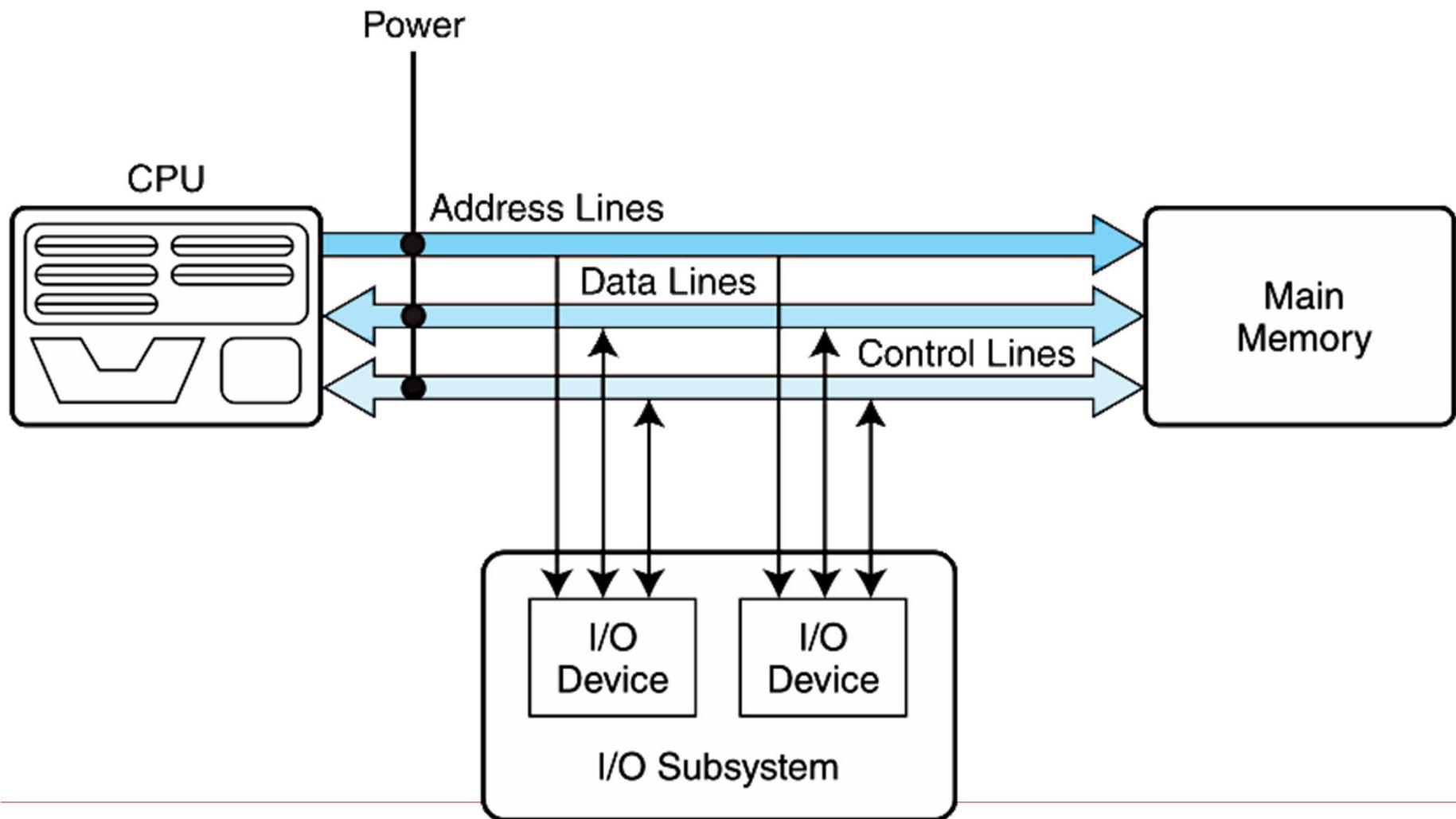
- A multipoint bus is shown below.
- Because a multipoint bus is a shared resource, access to it is controlled through protocols, which are built into the hardware.



4.3 The Bus

- Buses consist of **data lines**, **control lines**, and **address lines**.
 - **Data lines** convey bits from one device to another.
 - **Control lines** determine the direction of data flow, and when each device can access the bus.
 - **Address lines** determine the location of the source or destination of the data.

4.3 The Bus



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:

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

- 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 $4M \times 16$ RAM chip gives us $4 \frac{megabytes}{MW}$ of 16-bit memory locations.

4.6 Memory Organization

- How does the computer access a memory location corresponds to a particular address?
- We observe that 4M can be expressed as $2^2 \times 2^{20} = 2^{22}$ words.
- The memory locations for this memory are numbered 0 through $2^{22} - 1$.
- Thus, the memory bus of this system requires at least 22 address lines.
 - The address lines “count” from 0 to $2^{22} - 1$ in binary. Each line is either “on” or “off” indicating the location of the desired memory element.

4.6 Memory Organization

- Physical memory usually consists of more than one RAM chip.
- Access is more efficient when memory is organized into **banks** of chips with the addresses **interleaved** across the chips
- With **low-order interleaving**, the low order bits of the address specify which memory bank contains the address of interest.
- Accordingly, in **high-order interleaving**, the high order address bits specify the memory bank.

4.6 Memory Organization

- Example: Suppose we have a memory consisting of 16 2K x 8 bit chips.

- 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.

Row 0

2K × 8

Row 1

2K × 8

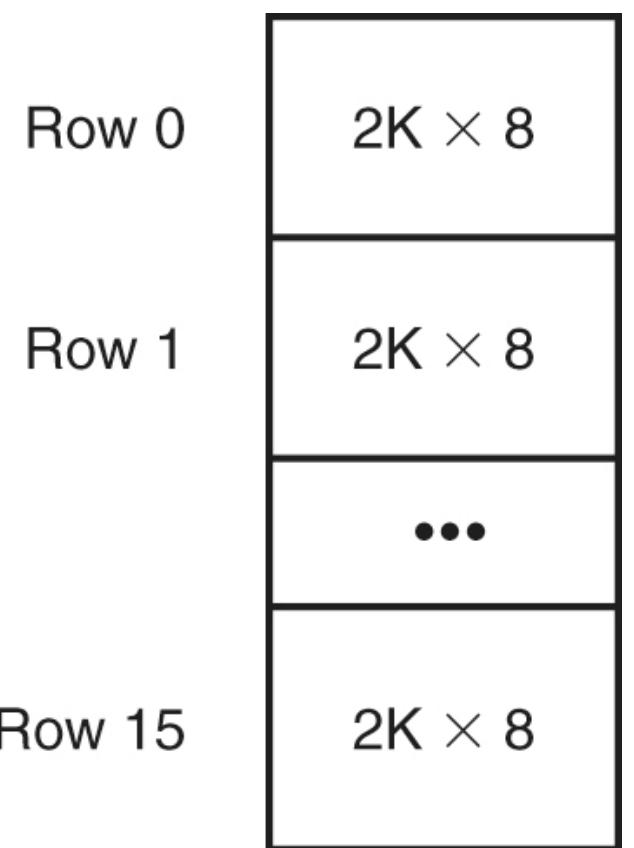
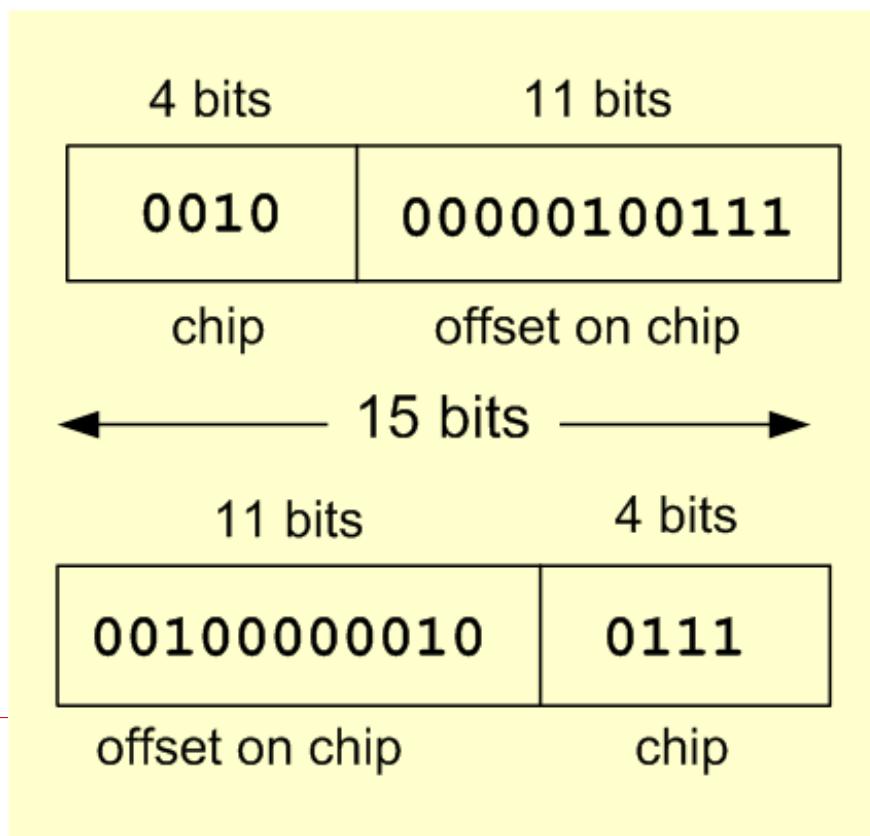
...

Row 15

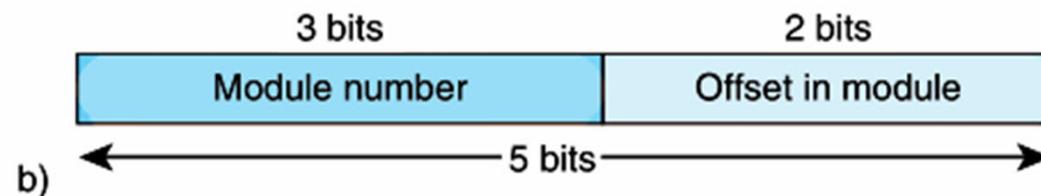
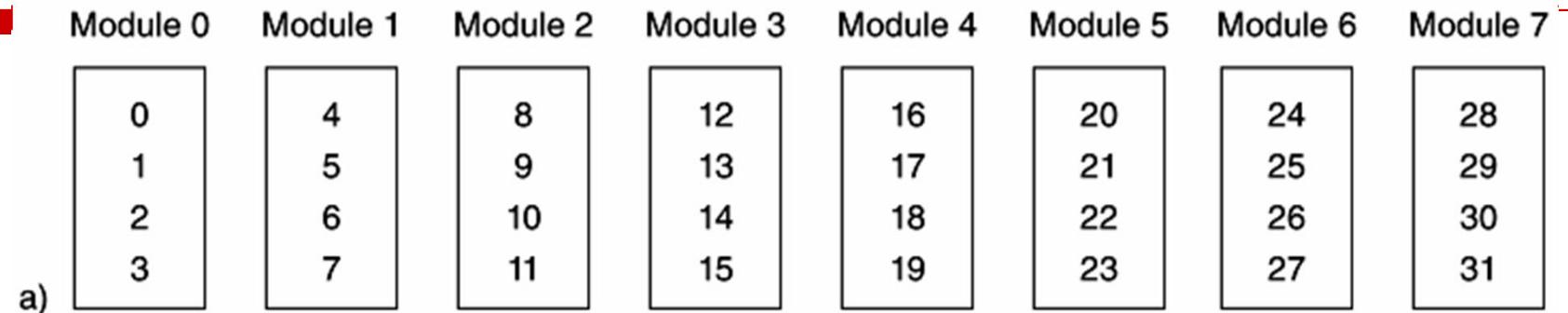
2K × 8

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



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

Module 0

0
8
16
24

Module 1

1
9
17
25

Module 2

2
10
18
26

Module 3

3
11
19
27

Module 4

4
12
20
28

Module 5

5
13
21
29

Module 6

6
14
22
30

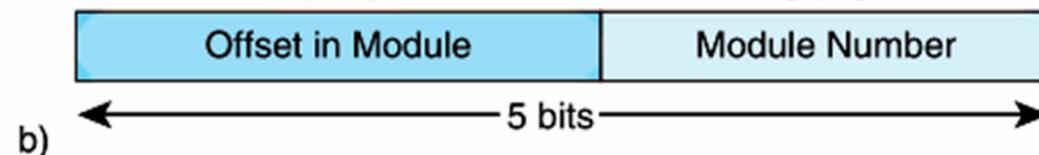
Module 7

7
15
23
31

a)

2 bits

3 bits



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 ($128 = 2^7$).



4.7 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.

4.8 MARIE

- We can now bring together many of the ideas that we have discussed to this point using a very **simple** model computer.
- Our model computer, the **Machine Architecture that is Really Intuitive and Easy**, **MARIE**, was designed for the singular purpose of illustrating basic computer system concepts.
- 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.

4.8 MARIE

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.

4.8 MARIE

MARIE'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.

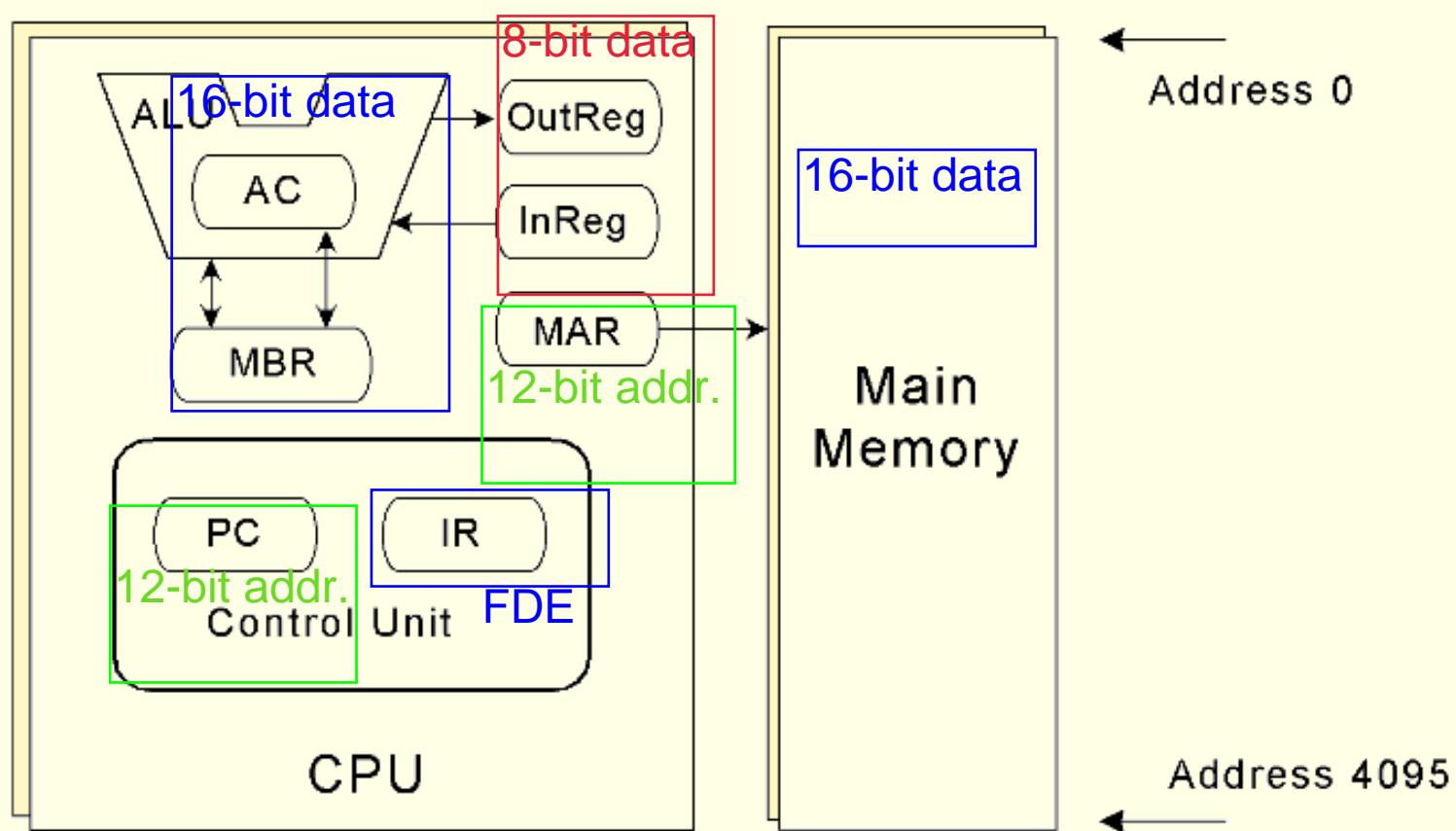
4.8 MARIE

MARIE'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, **InREG**, 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.

4.8 MARIE

This is the MARIE architecture shown graphically.

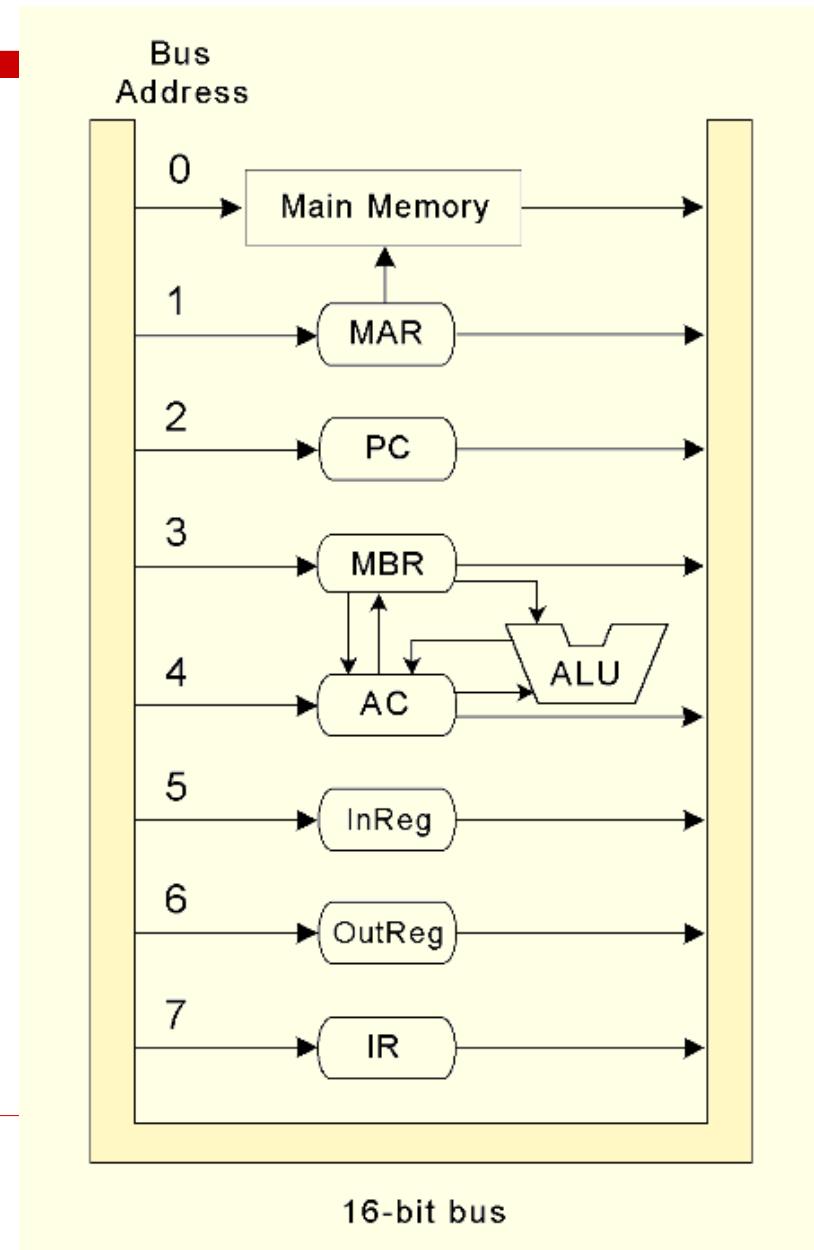


4.8 MARIE

- The **registers** are **interconnected**, and connected with **main memory** through a common data bus.
- Each **device** on the bus has a unique number based **identifier** (shown on the next slide) that is **set on the control lines** whenever that device is required to carry out an operation.
- **Separate** connections are also **provided between the accumulator** and the memory **buffer register**, and the **ALU** and the accumulator and memory buffer register.
- This **permits data transfer** between these devices **without use of the main data bus**.

4.8 MARIE

This is the MARIE data path shown graphically.

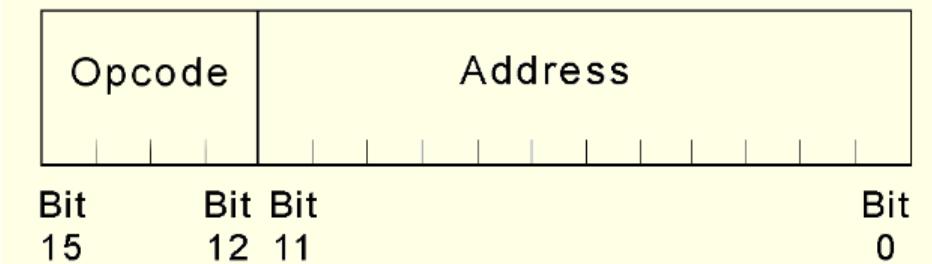


4.8 MARIE

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

4.8 MARIE

- This is the format of a MARIE instruction:

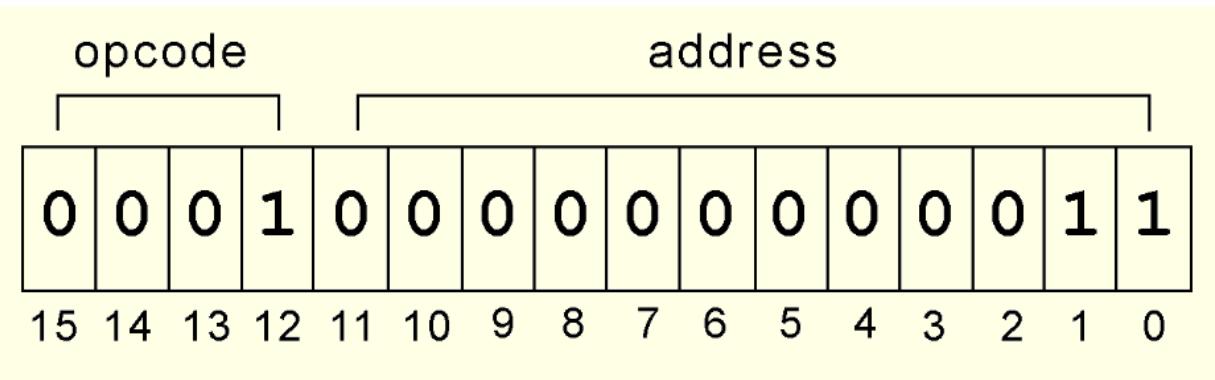


- The fundamental MARIE instructions are:

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

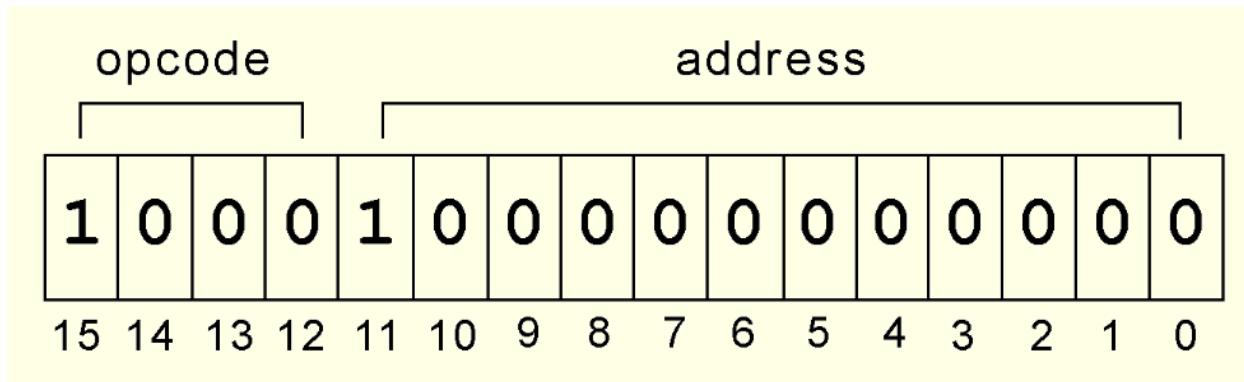
- This is a bit pattern for a **LOAD instruction** as it would appear in the IR:



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

4.8 MARIE

- 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?

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
```

4.8 MARIE

- Actually, each of our **instructions** 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 MARIE RTL, we **use the notation M[X]** to indicate the actual data value stored in memory location X, and ← to indicate the transfer of bytes to a register or memory location.

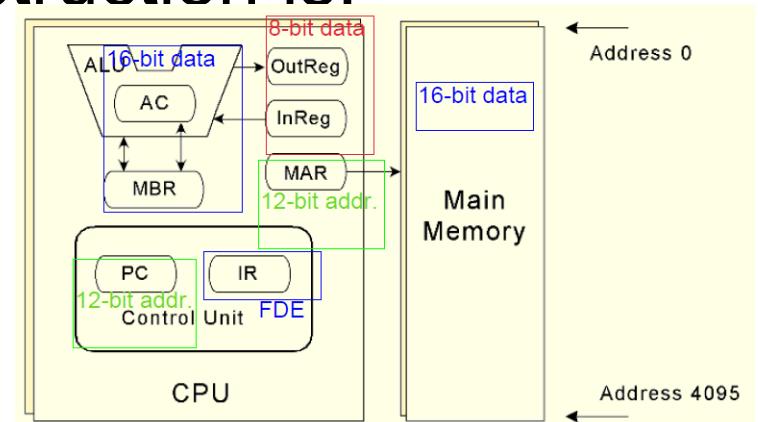
4.8 MARIE

- The RTL for the LOAD ' x ' instruction is:

MAR \leftarrow **x**

MBR \leftarrow **M[MAR]**

AC \leftarrow **MBR**



- Similarly, the RTL for the ADD ' x ' instruction is:

MAR \leftarrow **x**

MBR \leftarrow **M[MAR]**

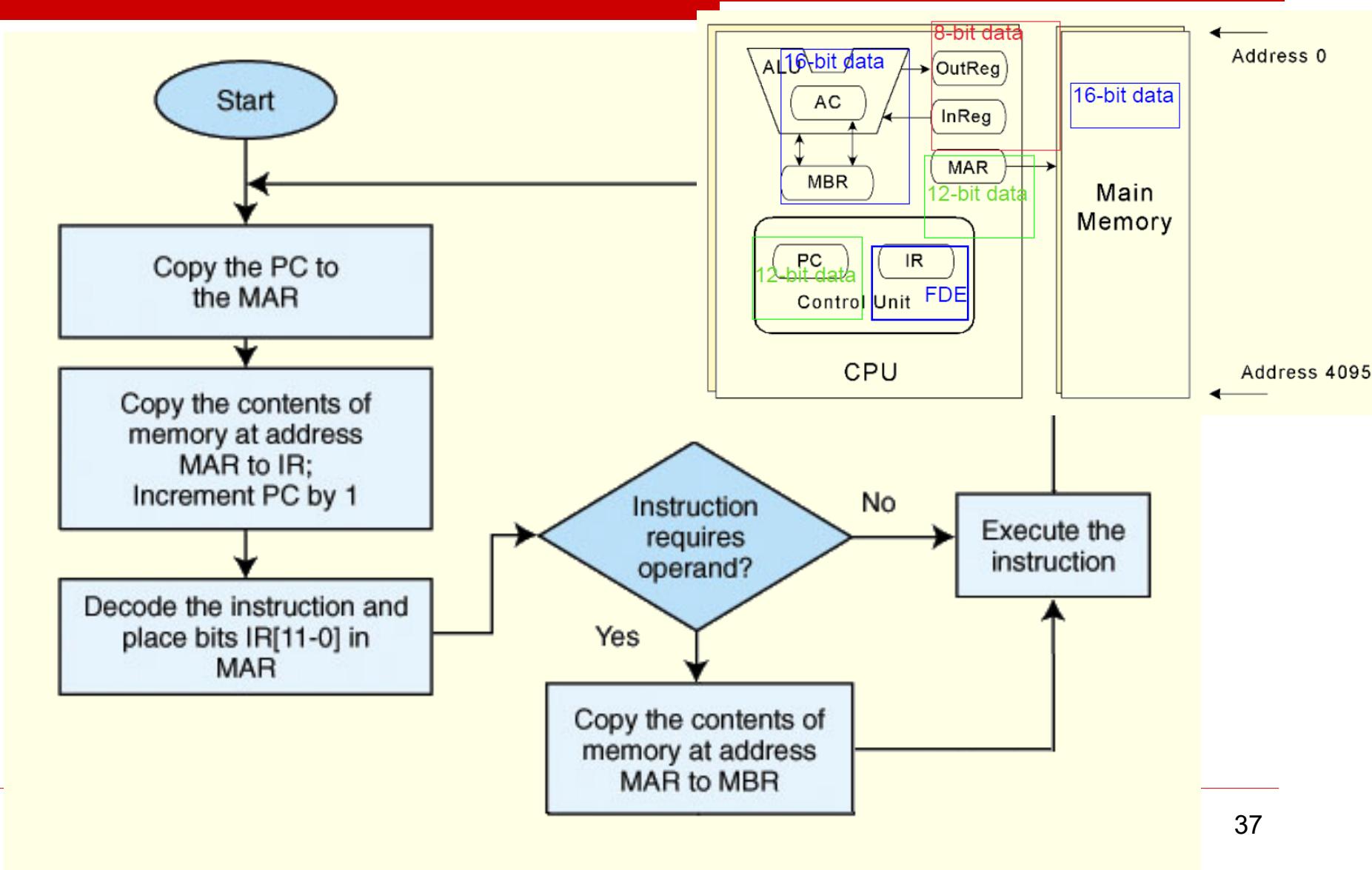
AC \leftarrow **AC + MBR**

4.9 Instruction Processing

- The *fetch-decode-execute cycle* is the series of steps that a computer carries out when it runs a program.
- We first have to *fetch* an instruction from memory, and place it into the IR.
- Once in the IR, it is *decoded* to determine what needs to be done next.
- If a memory value (**operand**) is involved in the operation, it is retrieved and placed into the **MBR**.
- With everything in place, the instruction is *executed*.

The next slide shows a flowchart of this process.

4.9 Instruction Processing

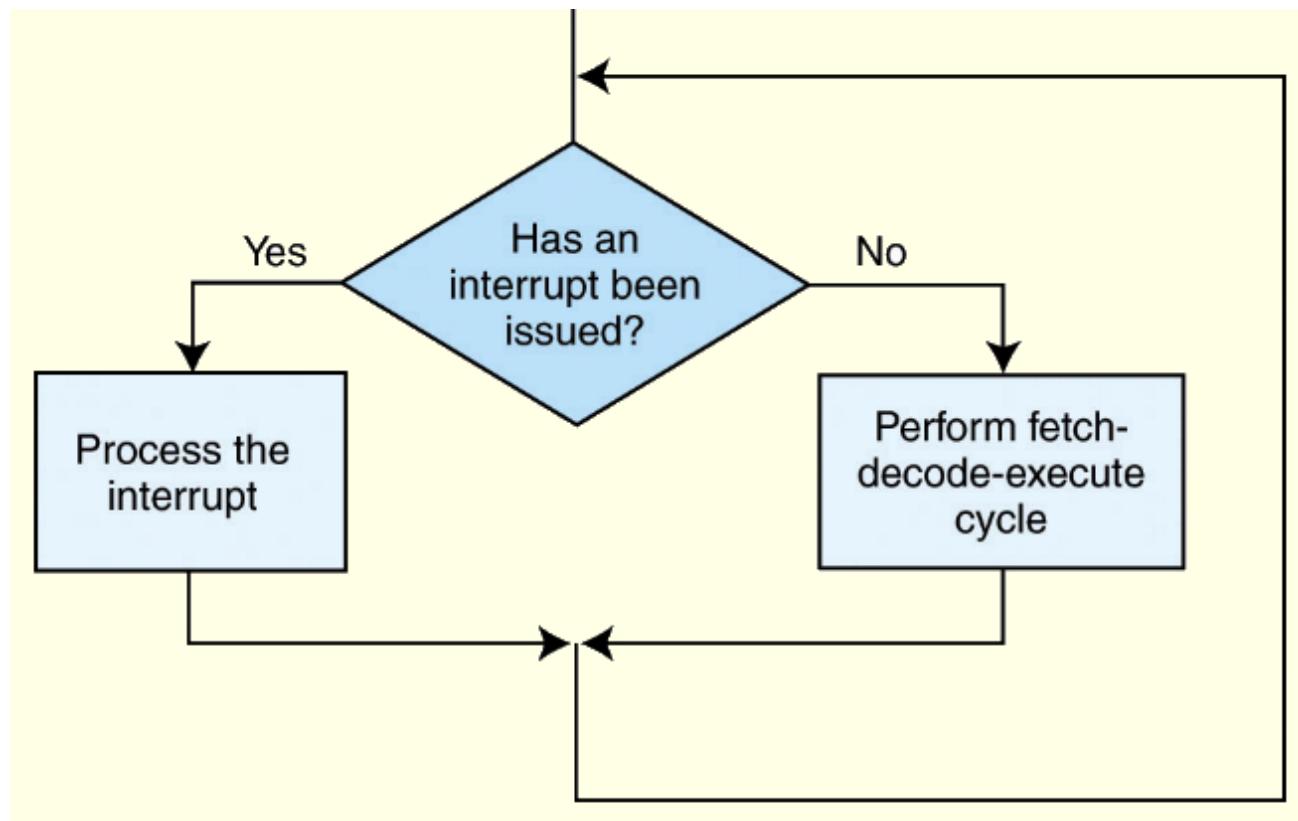


4.9 Instruction Processing

- 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*.

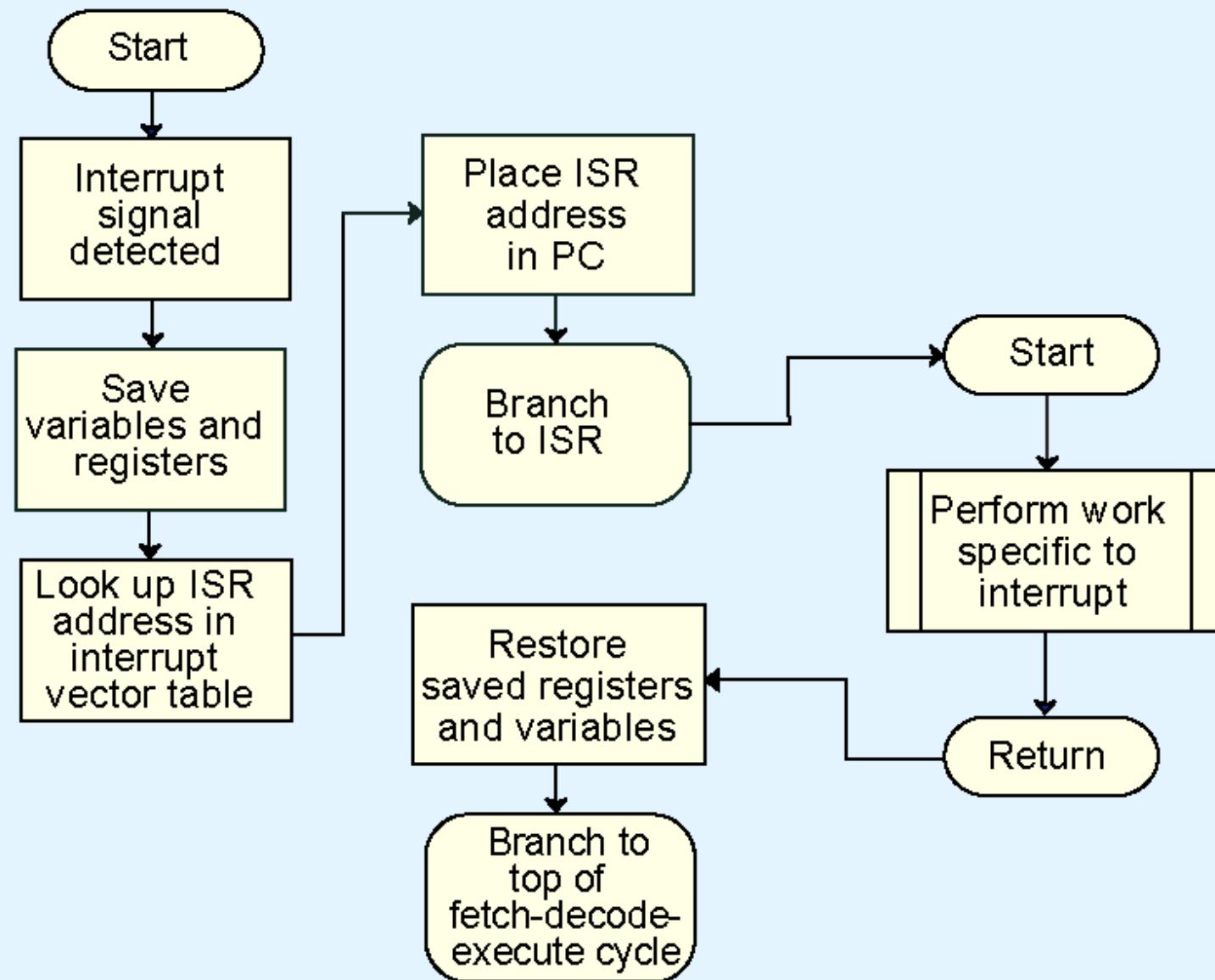
4.9 Instruction Processing

- ❑ 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 Instruction Processing



4.9 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.

4.9 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.
 - 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 Program

- 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	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023 X
105	FFE9	111111111101001	FFE9 Y
106	0000	0000000000000000	0000 Z

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

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

4.10 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 \leftarrow PC	100	- - - - -	100	- - - - -	- - - - -
	IR \leftarrow M [MAR]	100	1104	100	- - - - -	- - - - -
	PC \leftarrow PC + 1	101	1104	100	- - - - -	- - - - -
Decode	MAR \leftarrow IR[11-0]	101	1104	104	- - - - -	- - - - -
	(Decode IR[15-12])	101	1104	104	- - - - -	- - - - -
Get operand	MBR \leftarrow M [MAR]	101	1104	104	0023	- - - - -
Execute	AC \leftarrow MBR	101	1104	104	0023	0023

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	FPE9	FFE9
106	0000	0000

4.10 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 \leftarrow PC	101	1104	101	0023	0023
	IR \leftarrow M [MAR]	101	3105	101	0023	0023
	PC \leftarrow PC + 1	102	3105	101	0023	0023
Decode	MAR \leftarrow IR [11–0]	102	3105	105	0023	0023
	(Decode IR [15–12])	102	3105	105	0023	0023
Get operand	MBR \leftarrow M [MAR]	102	3105	105	FFE9	0023
Execute	AC \leftarrow 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**.

4.11 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.

4.11 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	Hex Contents of Memory
100	Load x	1104
101	Add y	3105
102	Store z	4106
103	Halt	7000
104	DEC 35	0023
105	DEC -23	FFB9
106	HEX 0000	0000

Address	Instruction
100	Load x
101	Add y
102	Store z
103	Halt
104	DEC 35
105	DEC -23
106	HEX 0000

X	104
Y	105
Z	106

Symbol Table: Name, Address

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.

Object Program File or Machine Code (hex)

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

Symbol Table: Name, Address

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

Partially-assembled instructions

4.12 Extending Our Instruction Set

- So far, all of the MARIE instructions that we have discussed use a *direct addressing mode*.
- This means that the address of the operand is **explicitly** stated in the instruction.
- It is often useful to employ a *indirect addressing*, where the address of the address of the operand is given in the instruction.
 - If you have ever used **pointers** in a program, you are already familiar with indirect addressing.

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 \leftarrow **x**

MBR \leftarrow **M** [**MAR**]

AC \leftarrow **MBR**

MAR \leftarrow **x**
MBR \leftarrow **M** [**MAR**]
MAR \leftarrow **MBR**
MBR \leftarrow **M** [**MAR**]
AC \leftarrow **MBR**

LOADI X

MAR \leftarrow **x**
MBR \leftarrow **M** [**MAR**]
MAR \leftarrow **MBR**
MBR \leftarrow **AC**
M [**MAR**] \leftarrow **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
```

earlier, the RTL for the **ADD 'x'**

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

RTL for the **LOAD 'x'**

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

ADDI X

4.12 Extending Our Instruction Set

- Another helpful programming tool is the use of **subroutines** (like functions).
- 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 Jumps to what is at X
+1 by having it as the
current program execution
(PC), i.e. $PC = X + 1$

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

Does JNS permit recursive calls?

No, can't repeat, will jumble up the original execution when it is "rolling-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 \leftarrow 0

- We put our new instructions to work in the programs (examples).

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

Opcode	Instruction	RTN
0000	JnS X	$\text{MBR} \leftarrow \text{PC}$ $\text{MAR} \leftarrow X$ $\text{M}[\text{MAR}] \leftarrow \text{MBR}$ $\text{MBR} \leftarrow X$ $\text{AC} \leftarrow 1$ $\text{AC} \leftarrow \text{AC} + \text{MBR}$ $\text{PC} \leftarrow \text{AC}$
0001	Load X	$\text{MAR} \leftarrow X$ $\text{MBR} \leftarrow \text{M}[\text{MAR}]$ $\text{AC} \leftarrow \text{MBR}$
0010	Store X	$\text{MAR} \leftarrow X, \text{MBR} \leftarrow \text{AC}$ $\text{M}[\text{MAR}] \leftarrow \text{MBR}$
0011	Add X	$\text{MAR} \leftarrow X$ $\text{MBR} \leftarrow \text{M}[\text{MAR}]$ $\text{AC} \leftarrow \text{AC} + \text{MBR}$
0100	Subt X	$\text{MAR} \leftarrow X$ $\text{MBR} \leftarrow \text{M}[\text{MAR}]$ $\text{AC} \leftarrow \text{AC} - \text{MBR}$
0101	Input	$\text{AC} \leftarrow \text{InREG}$
0110	Output	$\text{OutREG} \leftarrow \text{AC}$
0111	Halt	
1000	Skipcond	If $\text{IR}[11-10] = 00$ then If $\text{AC} < 0$ then $\text{PC} \leftarrow \text{PC} + 1$ Else If $\text{IR}[11-10] = 01$ then If $\text{AC} = 0$ then $\text{PC} \leftarrow \text{PC} + 1$ Else If $\text{IR}[11-10] = 10$ then If $\text{AC} > 0$ then $\text{PC} \leftarrow \text{PC} + 1$
1001	Jump X	$\text{PC} \leftarrow \text{IR}[11-0]$
1010	Clear	$\text{AC} \leftarrow 0$
1011	AddI X	$\text{MAR} \leftarrow X$ $\text{MBR} \leftarrow \text{M}[\text{MAR}]$ $\text{MAR} \leftarrow \text{MBR}$ $\text{MBR} \leftarrow \text{M}[\text{MAR}]$ $\text{AC} \leftarrow \text{AC} + \text{MBR}$
1100	JumpI X	$\text{MAR} \leftarrow X$ $\text{MBR} \leftarrow \text{M}[\text{MAR}]$ $\text{PC} \leftarrow \text{MBR}$
1101	LoadI X	$\text{MAR} \leftarrow X$ $\text{MBR} \leftarrow \text{M}[\text{MAR}]$ $\text{MAR} \leftarrow \text{MBR}$ $\text{MBR} \leftarrow \text{M}[\text{MAR}]$ $\text{AC} \leftarrow \text{MBR}$
1110	StoreI X	$\text{MAR} \leftarrow X$ $\text{MBR} \leftarrow \text{M}[\text{MAR}]$ $\text{MAR} \leftarrow \text{MBR}$ $\text{MBR} \leftarrow \text{AC}$ $\text{M}[\text{MAR}] \leftarrow \text{MBR}$

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*.
 - Hardwired controllers implement this program using digital logic components.
 - With microprogrammed control, a small program is placed into read-only memory in the microcontroller.

4.13 A Discussion on Decoding

- Your **text provides a complete list of the register transfer language for each of MARIE's instructions.**
- The **microoperations** given by each RTL **define the operation of MARIE's control unit.**
- Each microoperation consists of a **distinctive signal pattern** that is **interpreted by the control unit** and **results in the execution of an instruction.**
 - Recall, the RTL for the **Add** instruction is:

MAR \leftarrow **X**

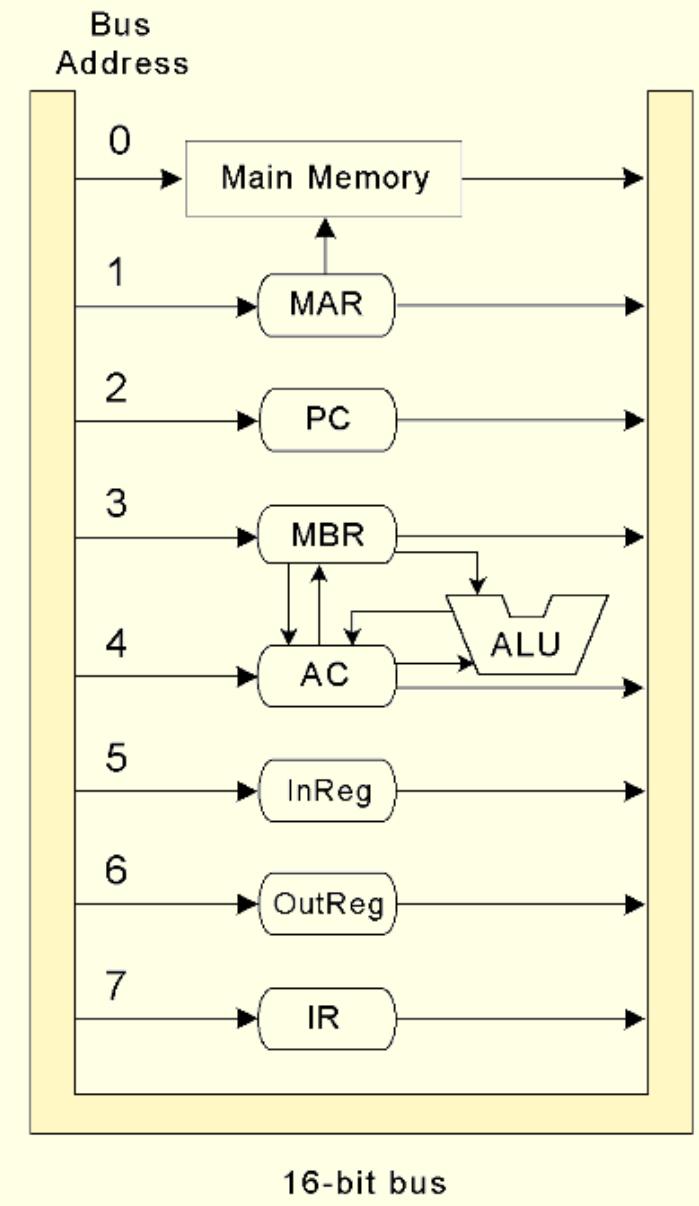
MBR \leftarrow **M[MAR]**

AC \leftarrow **AC + MBR**

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**.

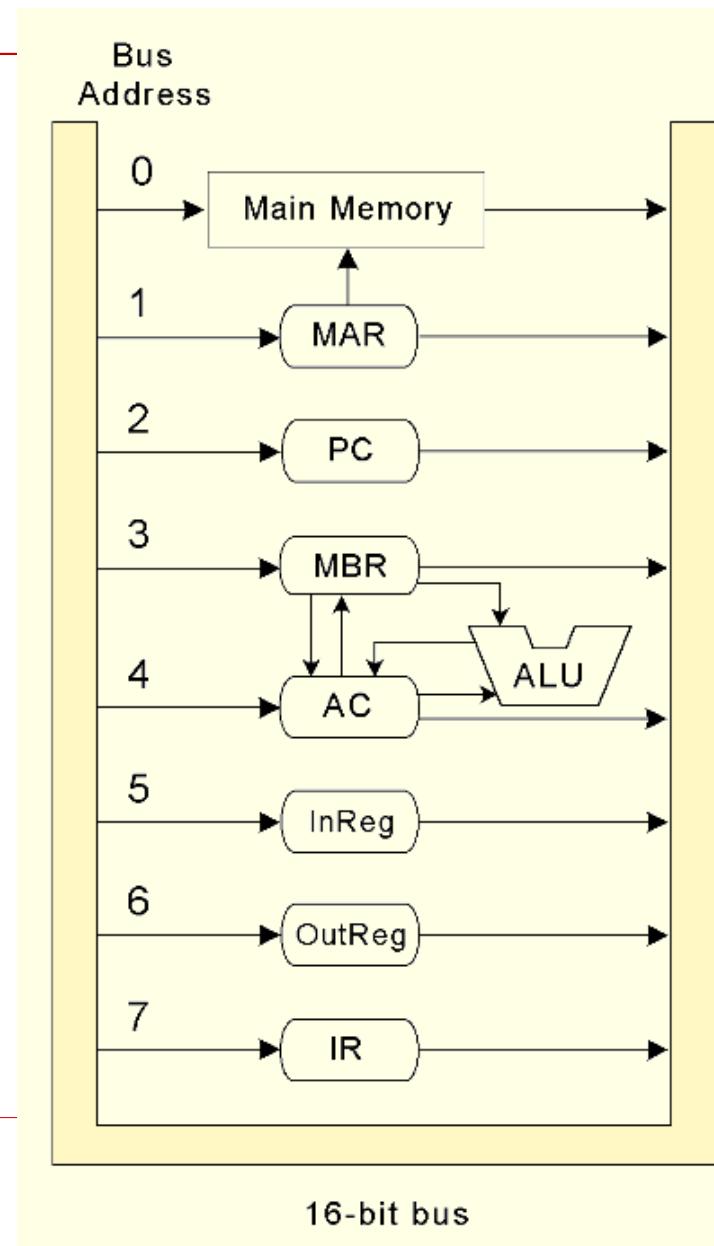
How many signal lines does MARIE's control unit need?



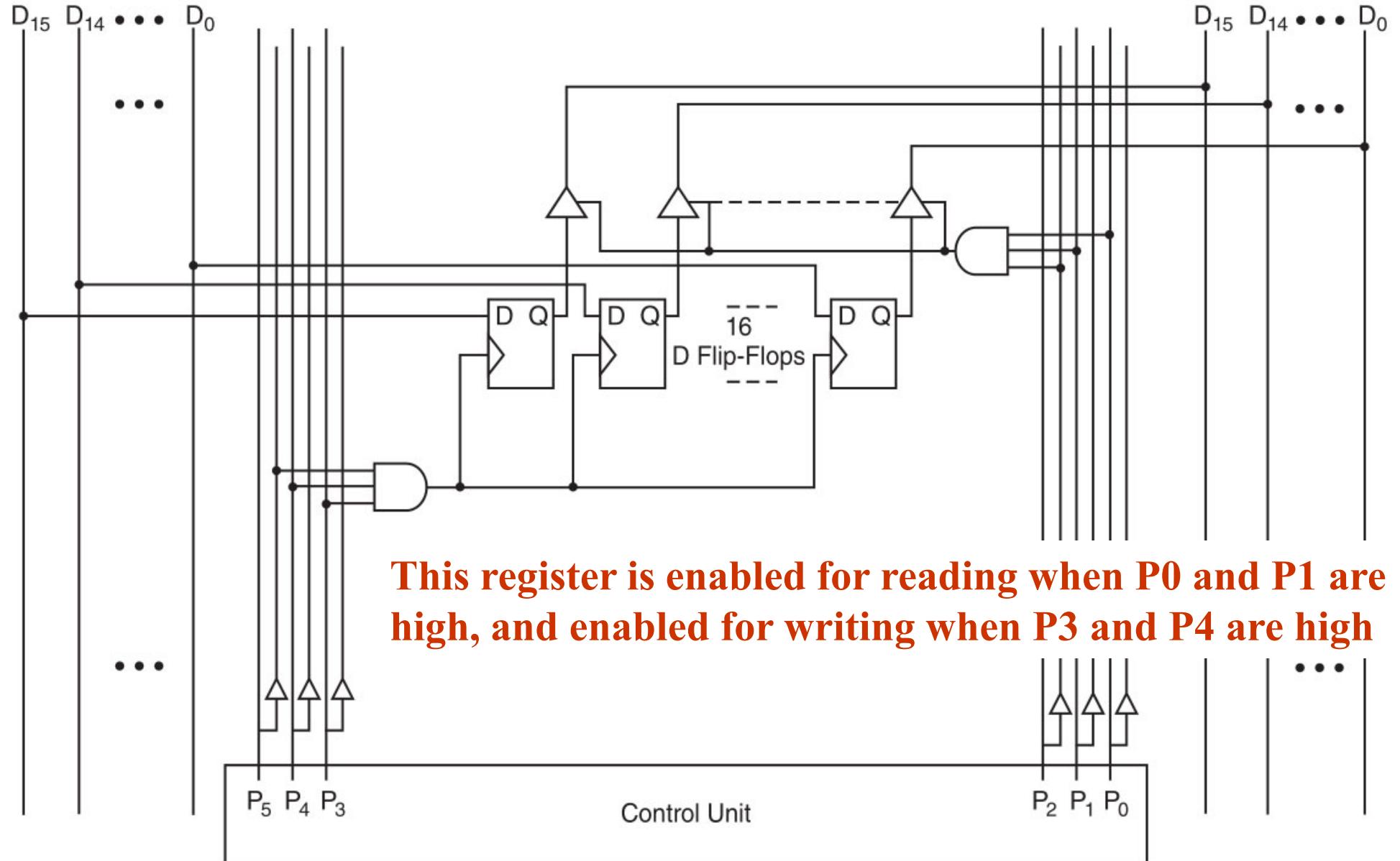
4.13 A Discussion on Decoding

- 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 Discussion on Decoding



4.13 A Discussion on Decoding

- Careful inspection of MARIE's RTL reveals that the **ALU has only three operations**: add, subtract, and clear.
 - 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 A_1
 - **MBR and AC**: L_{ALT} to control the data source.
 - **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	$AC \leftarrow AC + MBR$
1	0	$AC \leftarrow AC - MBR$
1	1	$AC \leftarrow 0$ (Clear)

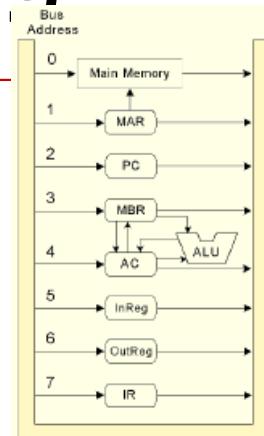
4.13 A Discussion on Decoding

- Consider MARIE's Add instruction. Its RTL is:

$$\text{MAR} \leftarrow X$$

$$\text{MBR} \leftarrow M[\text{MAR}]$$

$$\text{AC} \leftarrow \text{AC} + \text{MBR}$$



- After an Add instruction is fetched, the address, X, is in the rightmost 12 bits of the IR, which has a datapath address of 7.
- X is copied to the MAR, which has a datapath address of 1.
- Thus we need to raise signals P₀, P₁, and P₂ to read from the IR, and signal P₃ to write to the MAR.

4.13 A Discussion on Decoding

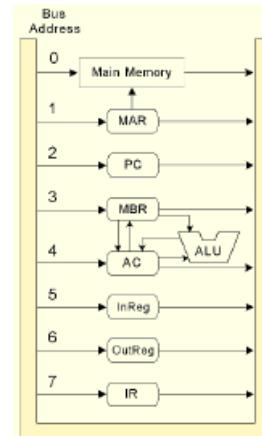
- Here is the complete signal sequence for MARIE's Add instruction:

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

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

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

[Reset counter]

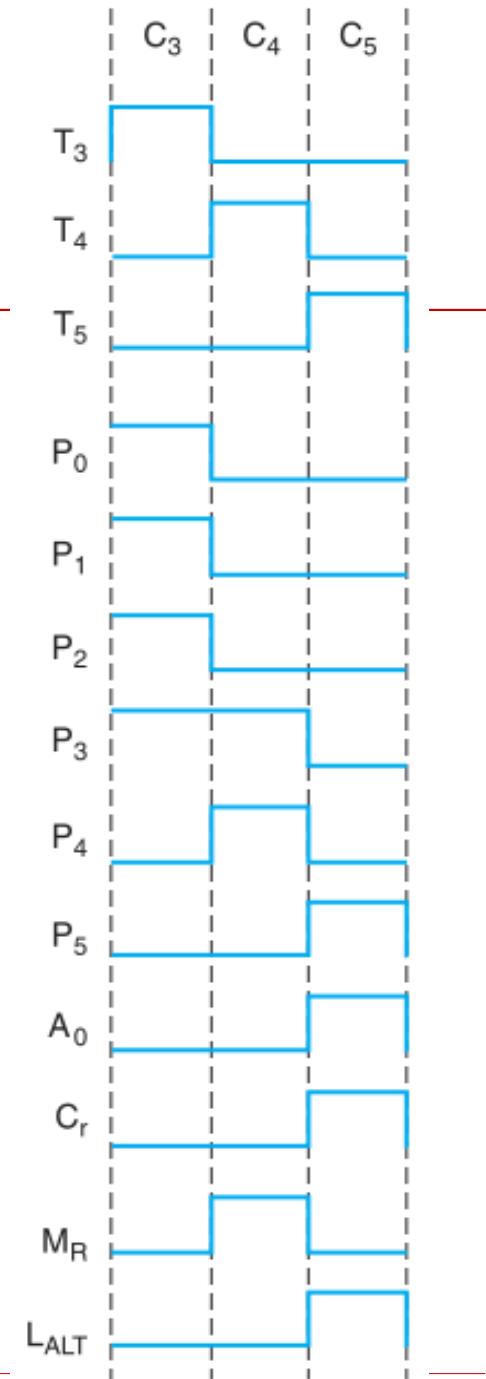
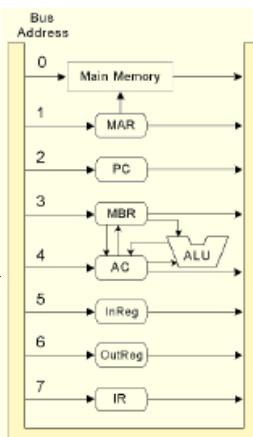


- These **signals** are ANDed with combinational logic to bring about the desired machine behavior.
- The next slide shows the **timing** diagram for this instruction.

4.13 Decoding

- Notice the concurrent signal states during each machine cycle: C₃ through C₅.

P₃ P₂ P₁ P₀ T₃ : MAR \leftarrow X
P₄ P₃ T₄ M_R : MBR \leftarrow M [MAR]
C_r A₀ P₅ T₅ L_{ALT} : AC \leftarrow AC + MBR
[Reset counter]

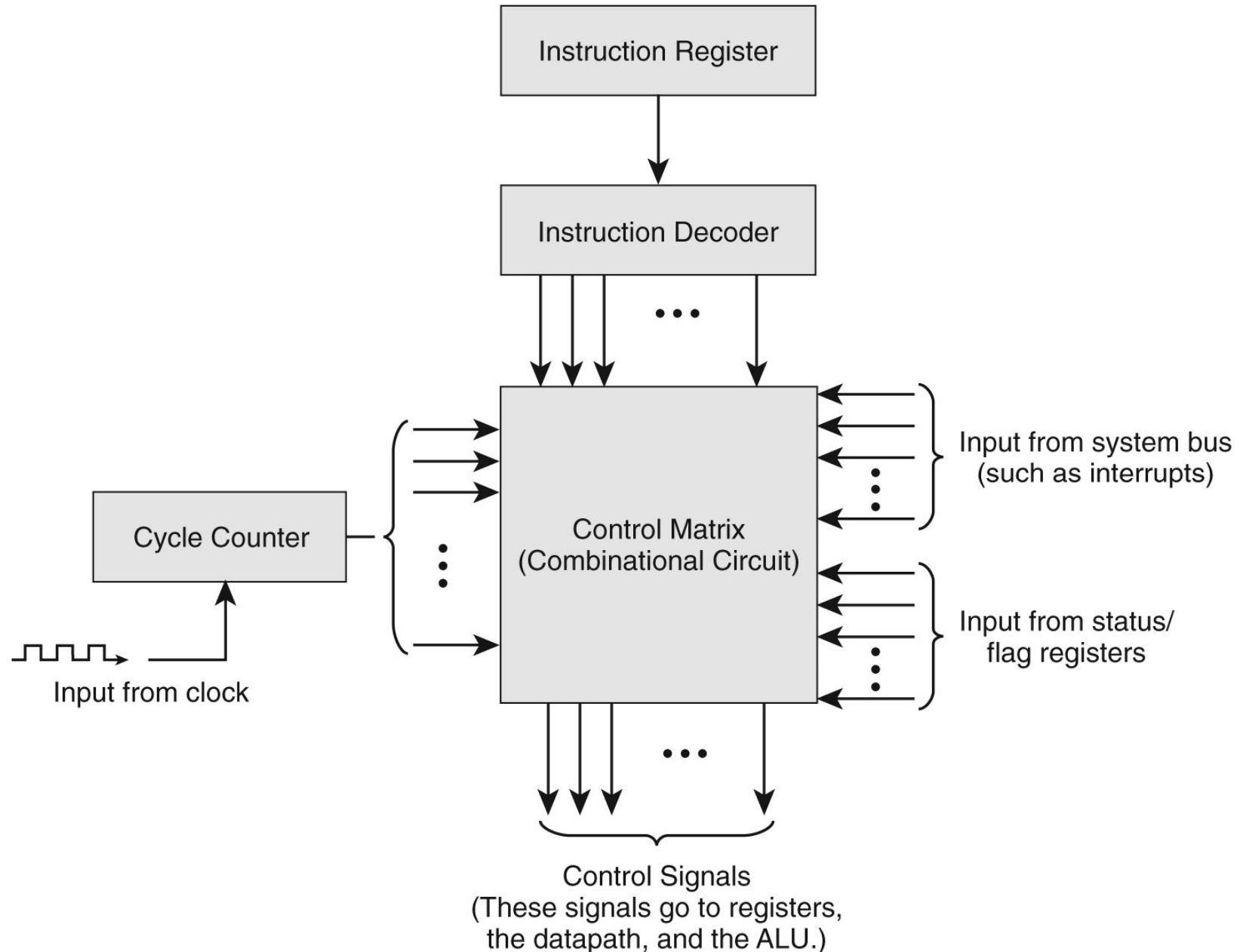


4.13 A Discussion on Decoding

- We note that the signal pattern just described is the **same** whether our machine used hardwired or microprogrammed control.
- In *hardwired control*, the bit pattern of machine instruction in the IR is decoded by **combinational logic**.
- The **decoder** output works with the control signals of the current system state to produce a new set of control signals.

A block diagram of a hardwired control unit is shown on the following slide.

4.13 A Discussion on 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

- The 8086 had four 16-bit general-purpose registers that could be accessed by the half-word.
 - It also had a flags register, an instruction register, and a stack accessed through the values in two other registers, the base pointer and the stack pointer.
 - The 8086 had no built in floating-point processing.
 - In 1980, Intel released the 8087 numeric coprocessor, but few users elected to install them because of their high cost.
-

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

- The MIPS family of CPUs has been one of the most successful in its class.
- In 1986 the first MIPS CPU was announced.
- It had a 32-bit word size and could address 4GB of memory.
- Over the years, MIPS processors have been used in general purpose computers as well as in games.
- The MIPS architecture now offers 32- and 64-bit versions.

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?

Chapter 4 Conclusion

- The **major** components of a computer system are its control unit, registers, memory, ALU, and data path.
- A built-in **clock** keeps everything synchronized.
- **Control units** can be microprogrammed or hardwired.
- **Hardwired control units** give better performance, while **microprogrammed** units are more adaptable to changes.

Chapter 4 Conclusion

- Computers run programs through iterative **fetch-decode-execute** cycles.
- Computers can run programs that are **in machine language**.
- An **assembler** converts mnemonic code to machine language.
- The **Intel** architecture is an example of a **CISC** architecture; **MIPS** is an example of a **RISC** architecture.