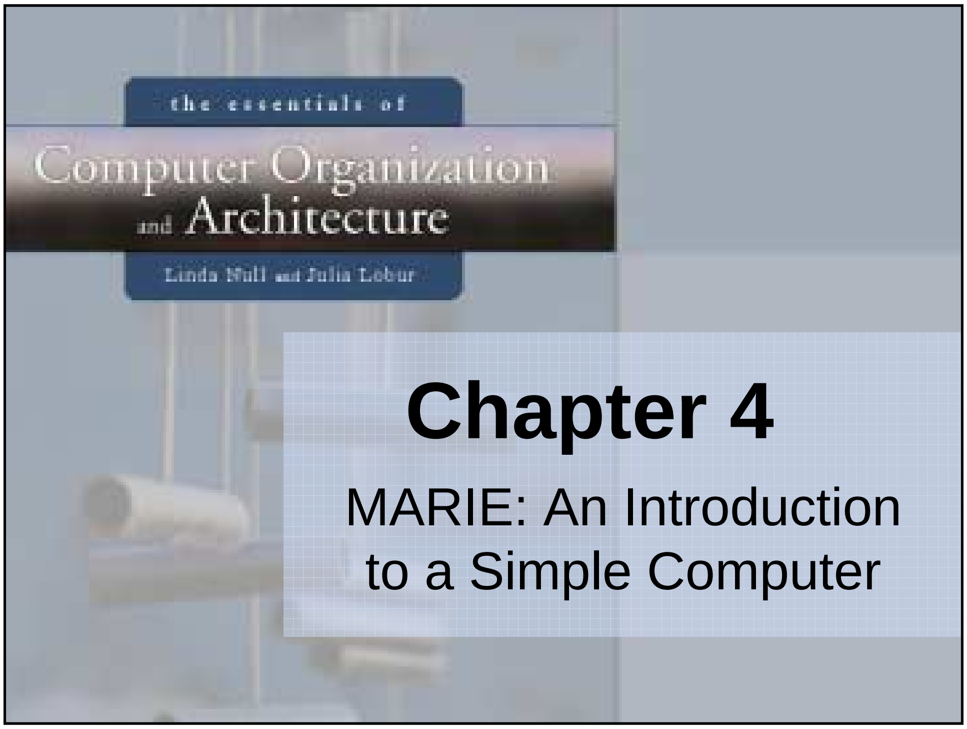


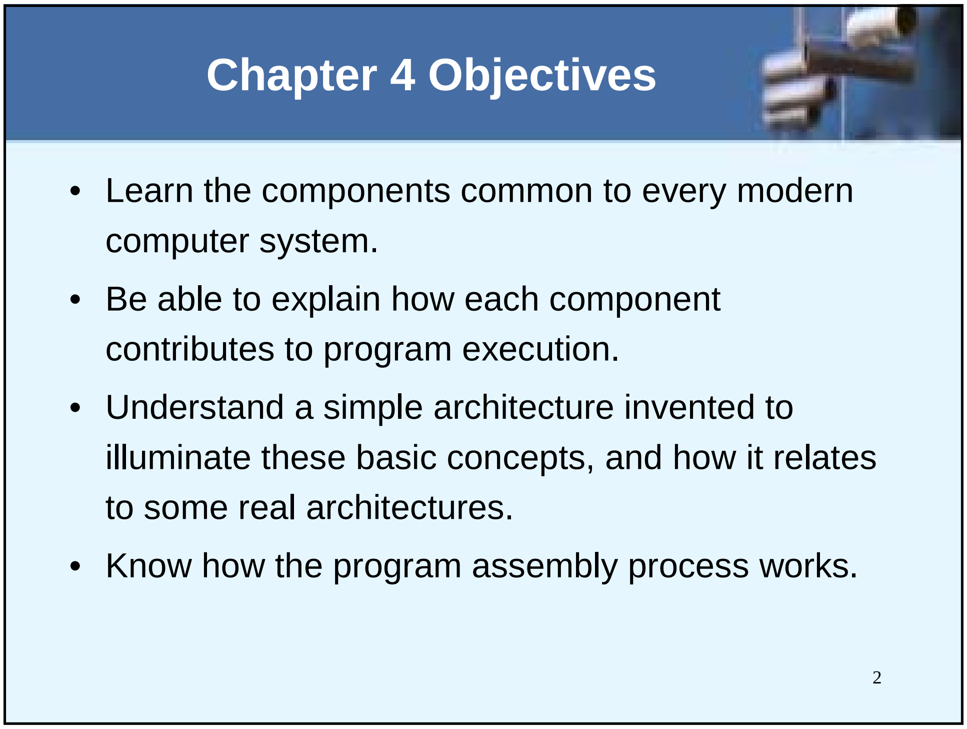
the essentials of

Computer Organization and Architecture

Linda Null and Julia Lobur

Chapter 4

MARIE: An Introduction to a Simple Computer



Chapter 4 Objectives

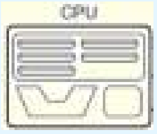
- Learn the components common to every modern computer system.
- Be able to explain how each component contributes to program execution.
- Understand a simple architecture invented to illuminate these basic concepts, and how it relates to some real architectures.
- Know how the program assembly process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

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.

4.1 Introduction

- The computer’s CPU fetches, decodes, and executes program instructions.
  - This is called the **Fetch/Decode/Execute Cycle** and the CPU repeats this cycle ad nauseum.
- The principal parts of the CPU are the *datapath, registers,* 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.



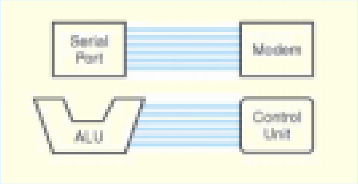
4.1 Introduction

- 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.

4.1 Introduction

- 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.

This is a point-to-point bus configuration:

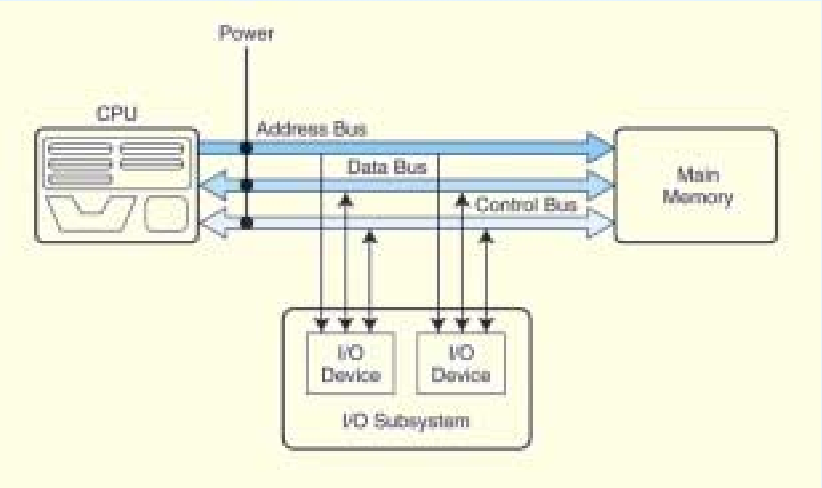


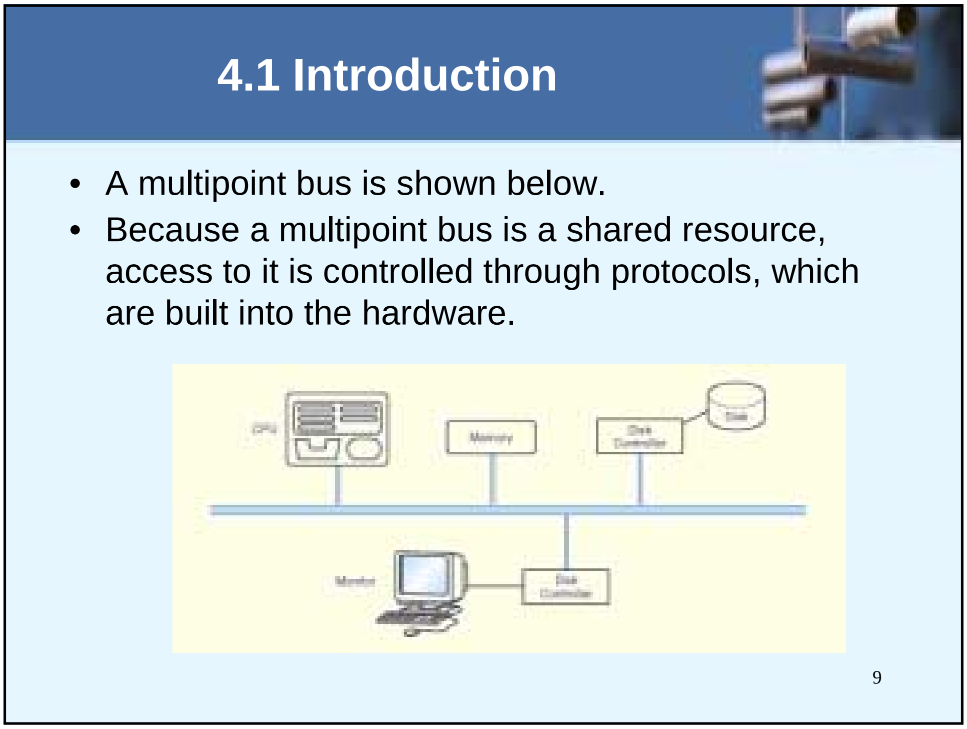
4.1 Introduction

- Buses consist of data lines, control lines, and address lines.
- While the 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.

The next slide shows a model bus configuration.

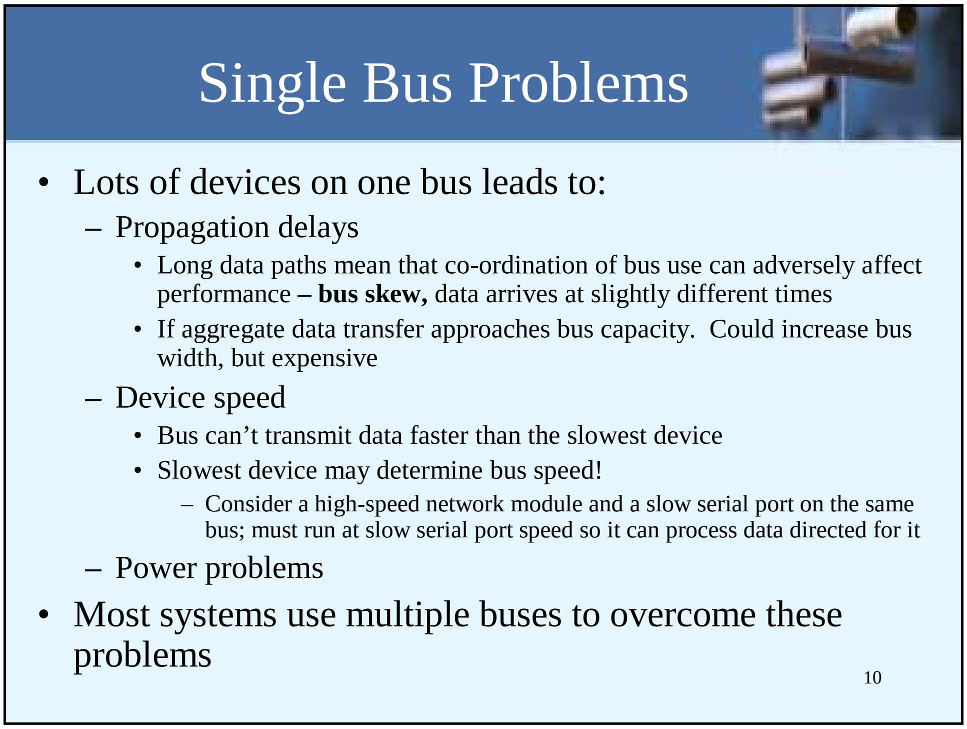
4.1 Introduction





4.1 Introduction

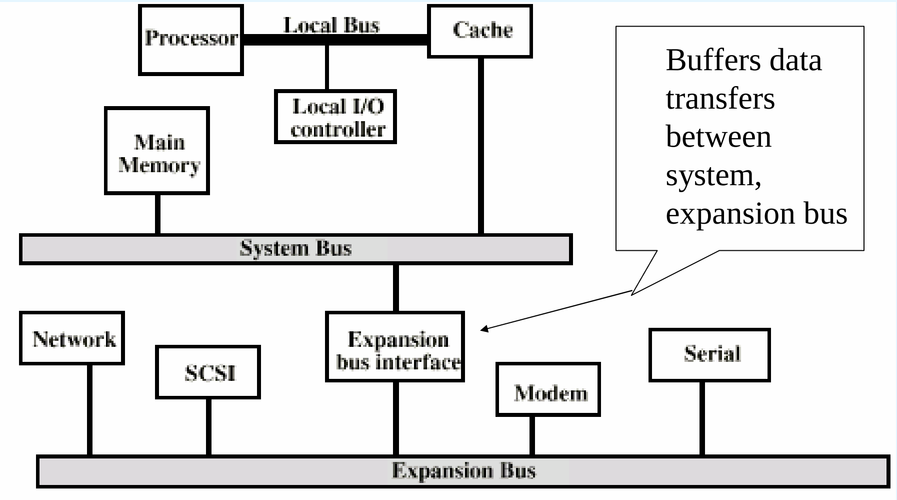
- A multipoint bus is shown below.
- Because a multipoint bus is a shared resource, access to it is controlled through protocols, which are built into the hardware.



Single Bus Problems

- Lots of devices on one bus leads to:
  - Propagation delays
    - Long data paths mean that co-ordination of bus use can adversely affect performance – **bus skew,** data arrives at slightly different times
    - If aggregate data transfer approaches bus capacity. Could increase bus width, but expensive
  - Device speed
    - Bus can’t transmit data faster than the slowest device
    - Slowest device may determine bus speed!
      - Consider a high-speed network module and a slow serial port on the same bus; must run at slow serial port speed so it can process data directed for it
  - Power problems
- Most systems use multiple buses to overcome these problems

Traditional (ISA) with cache

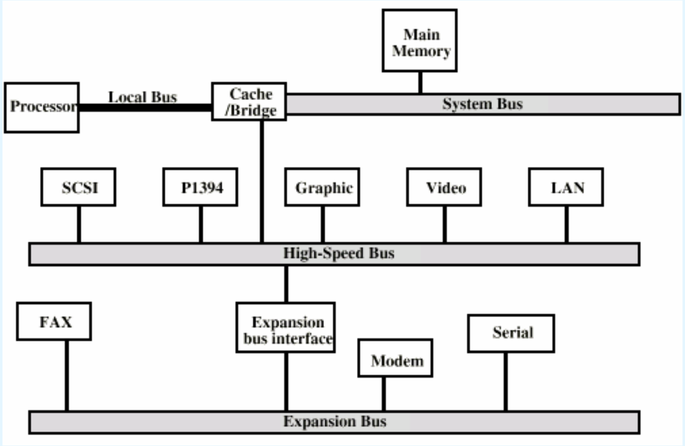


Buffers data transfers between system, expansion bus

This approach breaks down as I/O devices need higher performance

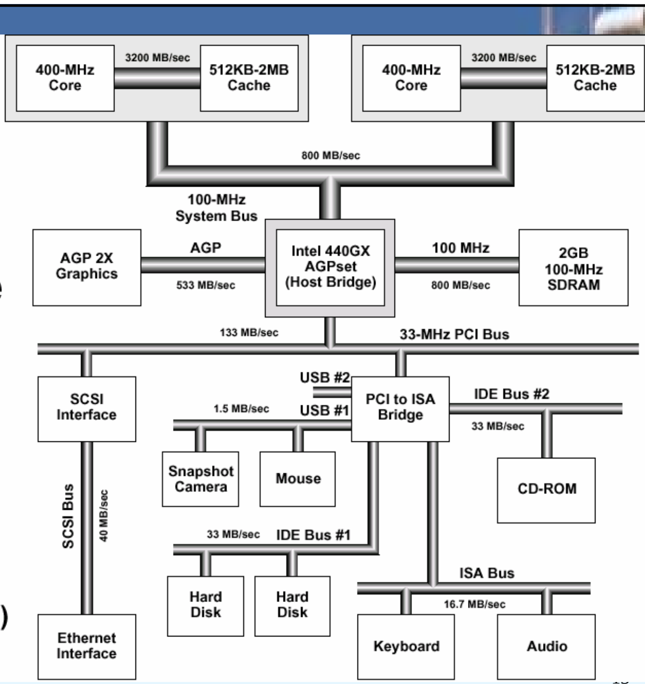
High Performance Bus – Mezzanine Architecture

Addresses higher speed I/O devices by moving up in the hierarchy



12

Bridge Based Bus Architecture



- Bridging with dual Pentium II Xeon processors on Slot 2.

(Source: http:// www.intel.com.)

4.1 Introduction

- In a master-slave configuration, where more than one device can be the bus master, concurrent bus master requests must be arbitrated.
- Four categories of bus arbitration are:
  - **Daisy chain:** Permissions are passed from the highest-priority device to the lowest.
  - **Centralized parallel:** Each device is directly connected to an arbitration circuit.
  - **Distributed using self-detection:** Devices decide which gets the bus among themselves.
  - **Distributed using collision-detection:** Any device can try to use the bus. If its data collides with the data of another device, it tries again.

4.1 Introduction

- 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.
  - Can't arbitrarily increase the clock frequency, constrained by the time it takes for data to travel from one register to the next or to perform some operation
- Clock cycle time is the reciprocal of clock frequency.
  - An 800 MHz clock has a cycle time of 1.25 ns.

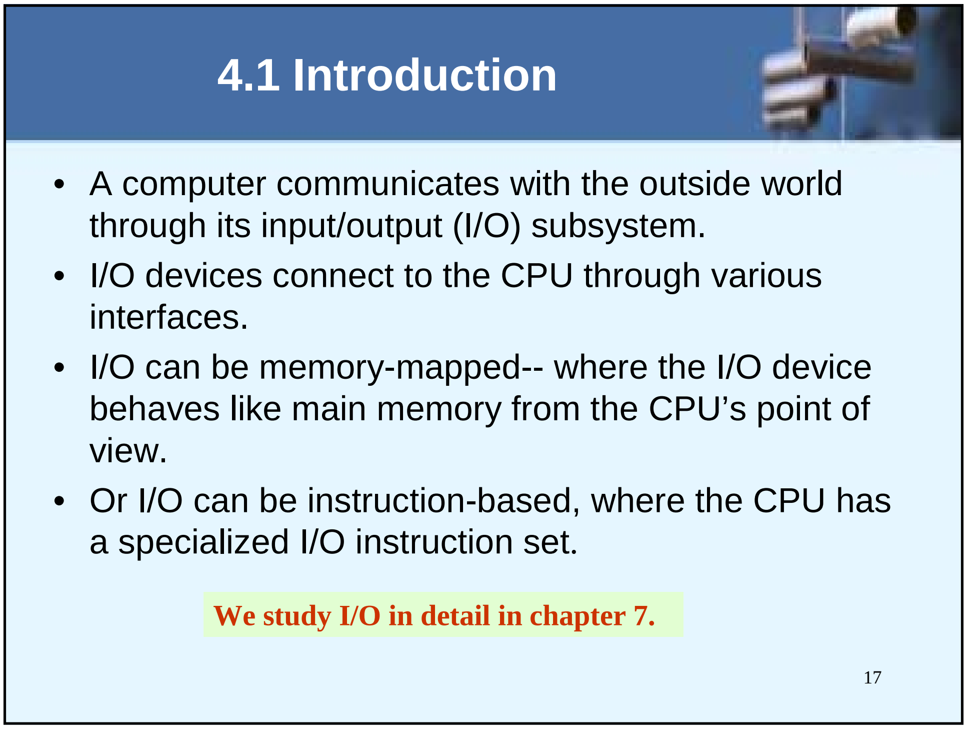
4.1 Introduction

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

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

  - We see that we can improve CPU throughput when we reduce the number of instructions in a program, reduce the number of cycles per instruction, or reduce the number of nanoseconds per clock cycle.

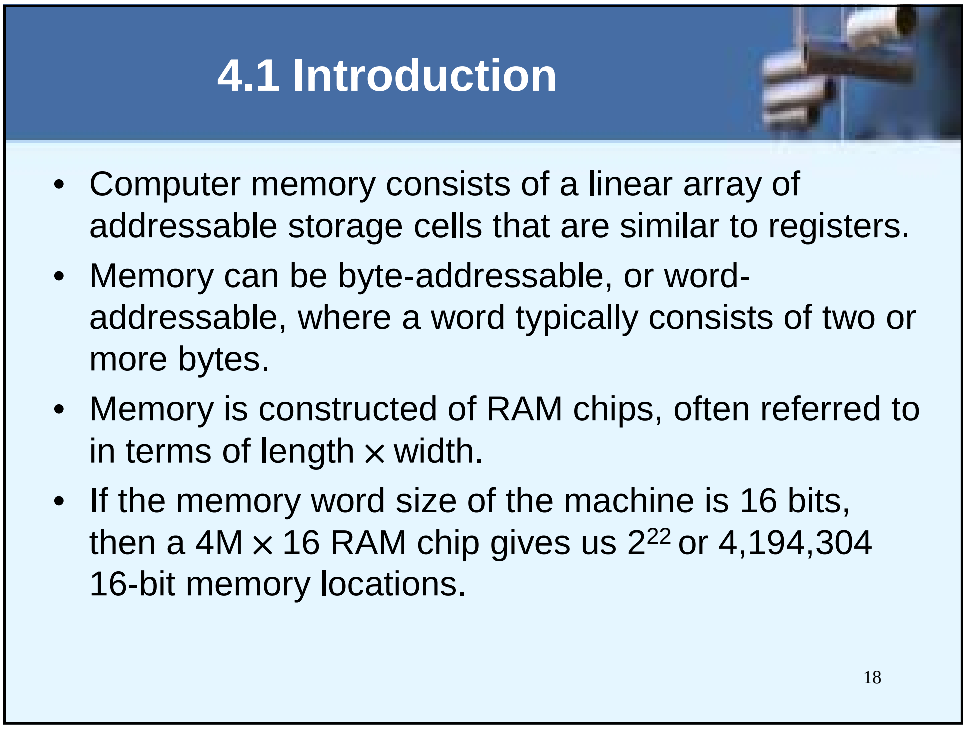
We will return to this important equation in later chapters.



4.1 Introduction

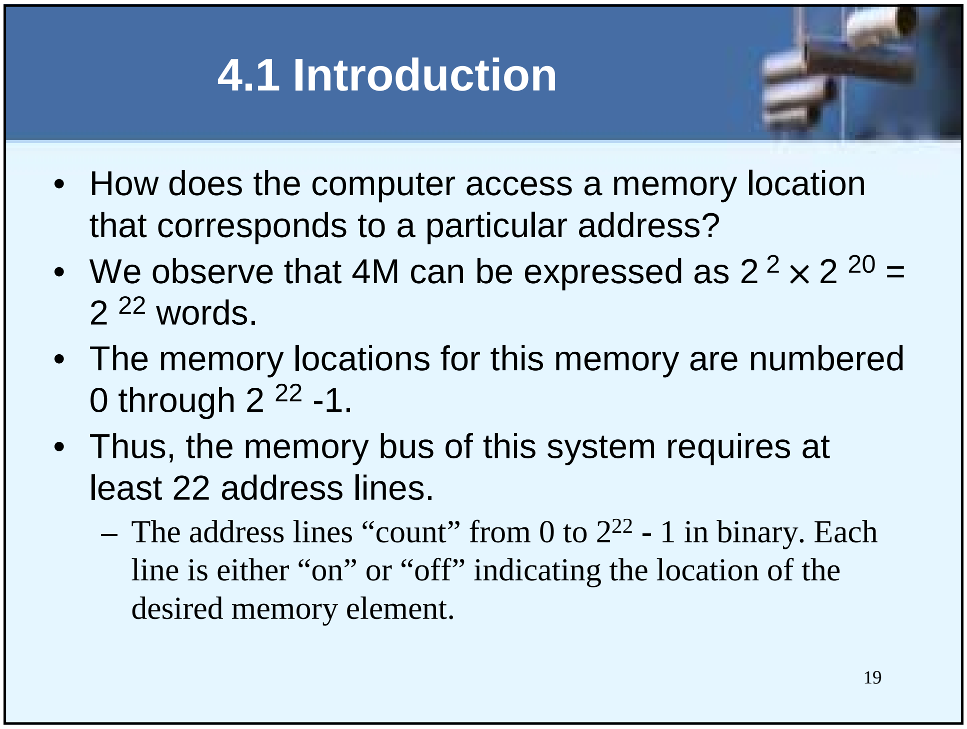
- A computer communicates with the outside world through its input/output (I/O) subsystem.
- I/O devices connect to the CPU through various interfaces.
- I/O can be memory-mapped-- where the I/O device behaves like main memory from the CPU's point of view.
- Or I/O can be instruction-based, where the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

We study I/O in detail in chapter 7.



4.1 Introduction

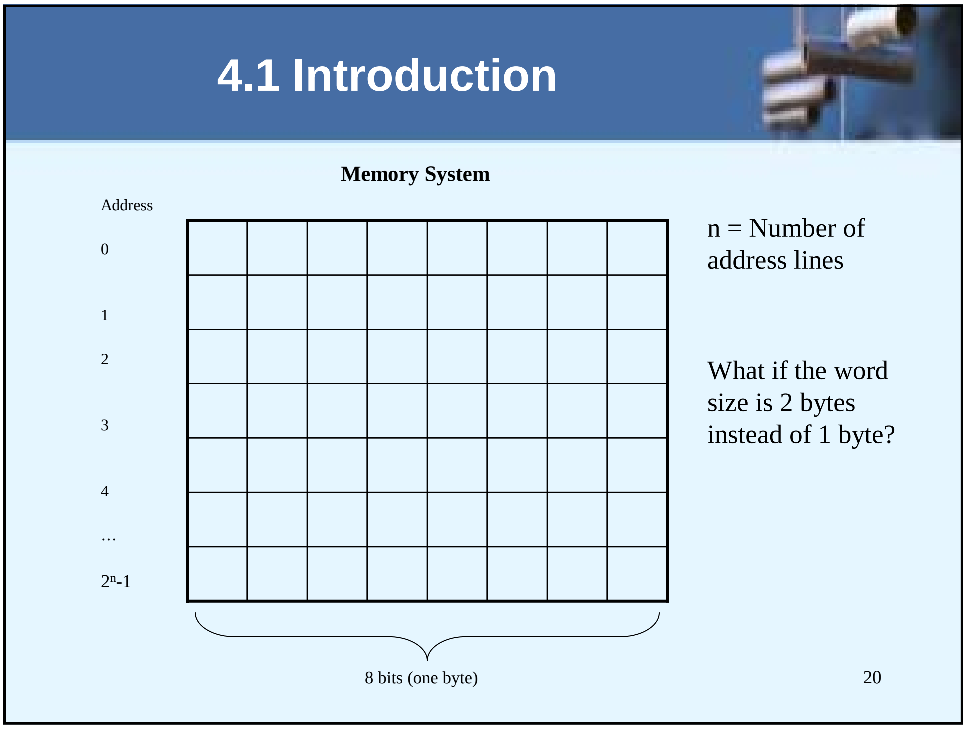
- Computer memory consists of a linear array of addressable storage cells that are similar to registers.
- Memory can be byte-addressable, or word-addressable, where a word typically consists of two or more bytes.
- Memory is constructed of RAM chips, often referred to in terms of length × width.
- If the memory word size of the machine is 16 bits, then a 4M × 16 RAM chip gives us 2^{22} or 4,194,304 16-bit memory locations.



4.1 Introduction

- How does the computer access a memory location that 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.

19



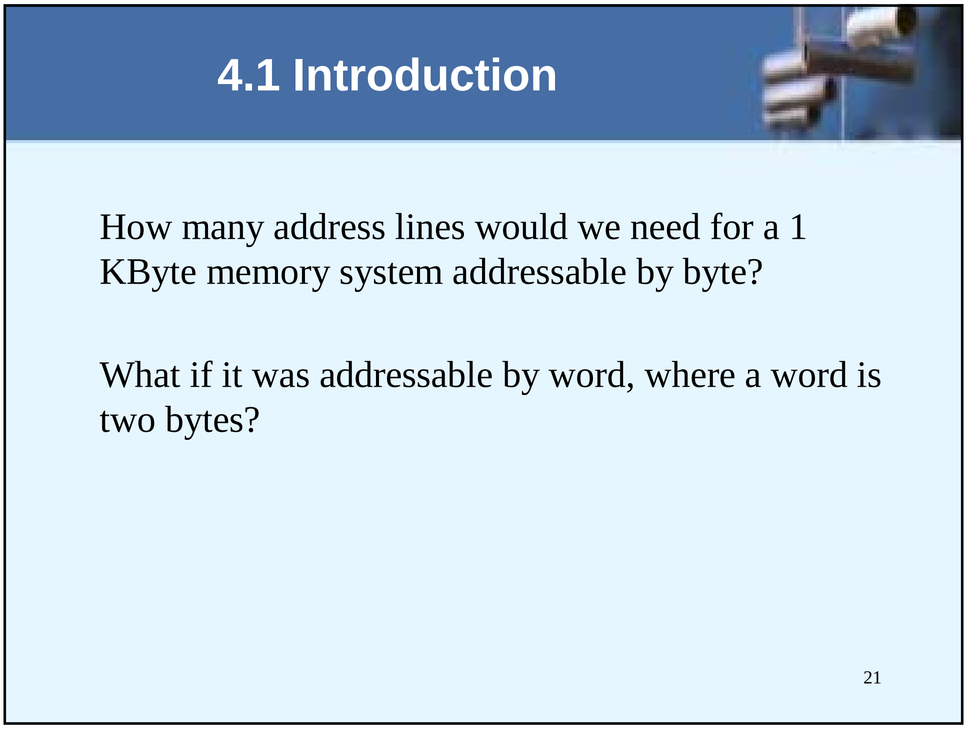
4.1 Introduction

Memory System

n = Number of address lines

What if the word size is 2 bytes instead of 1 byte?

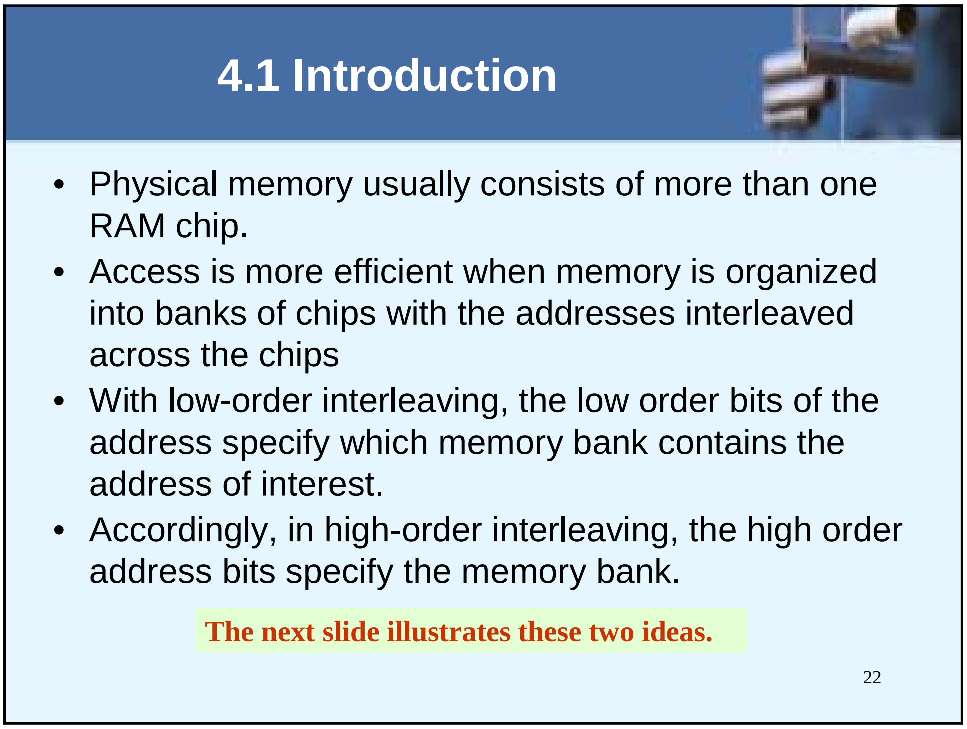
20



4.1 Introduction

How many address lines would we need for a 1 KByte memory system addressable by byte?

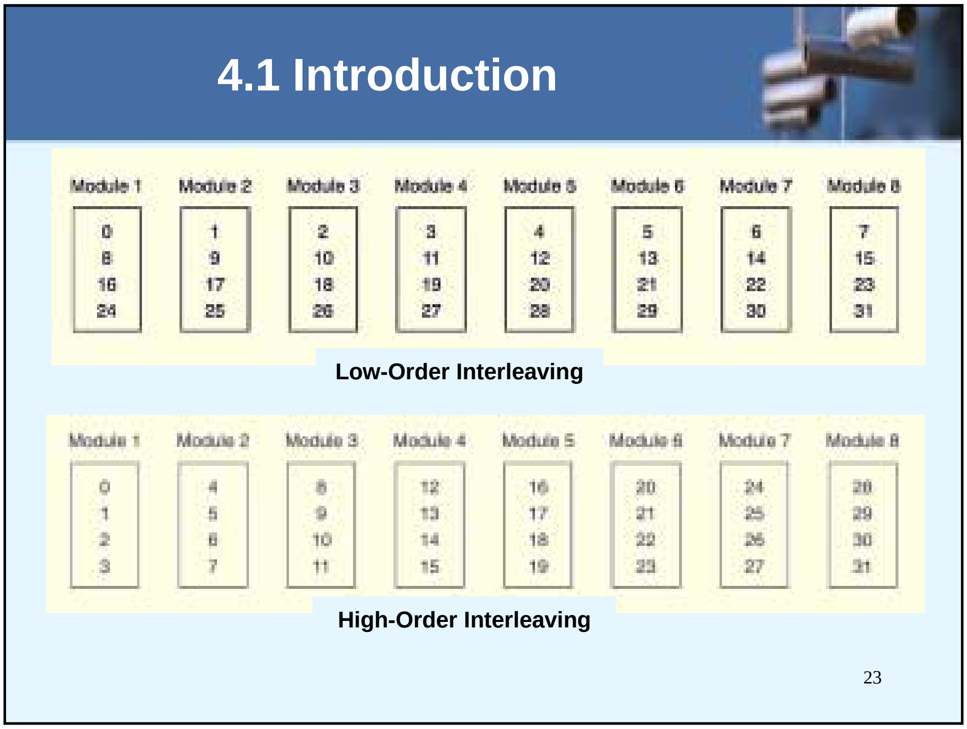
What if it was addressable by word, where a word is two bytes?



4.1 Introduction

- 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.

The next slide illustrates these two ideas.



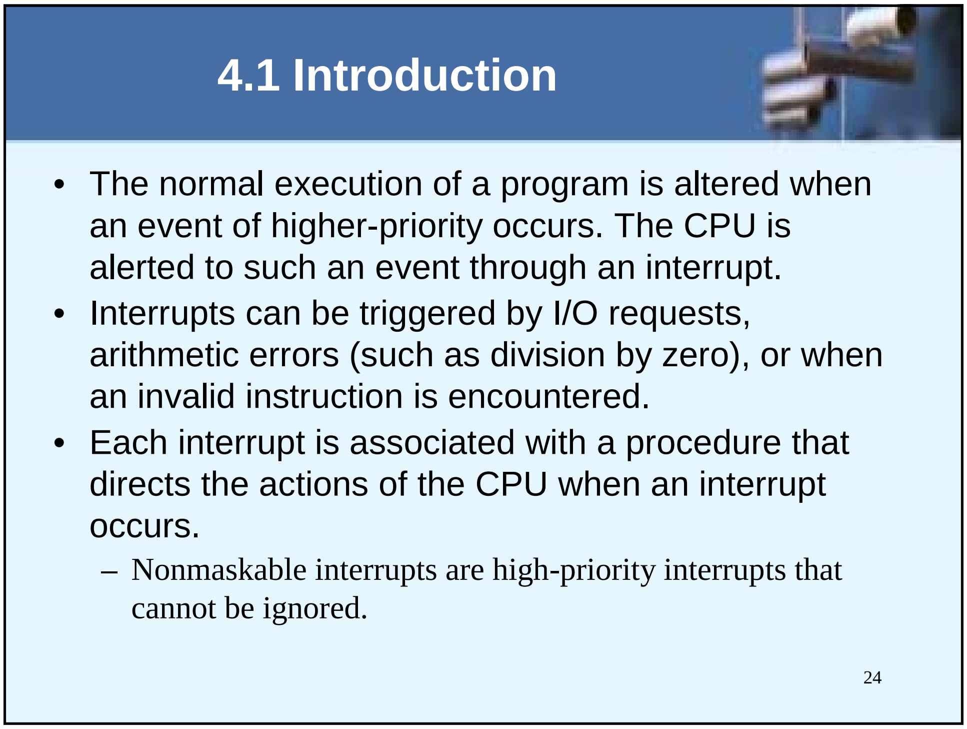
4.1 Introduction

Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7	Module 8
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

Low-Order Interleaving

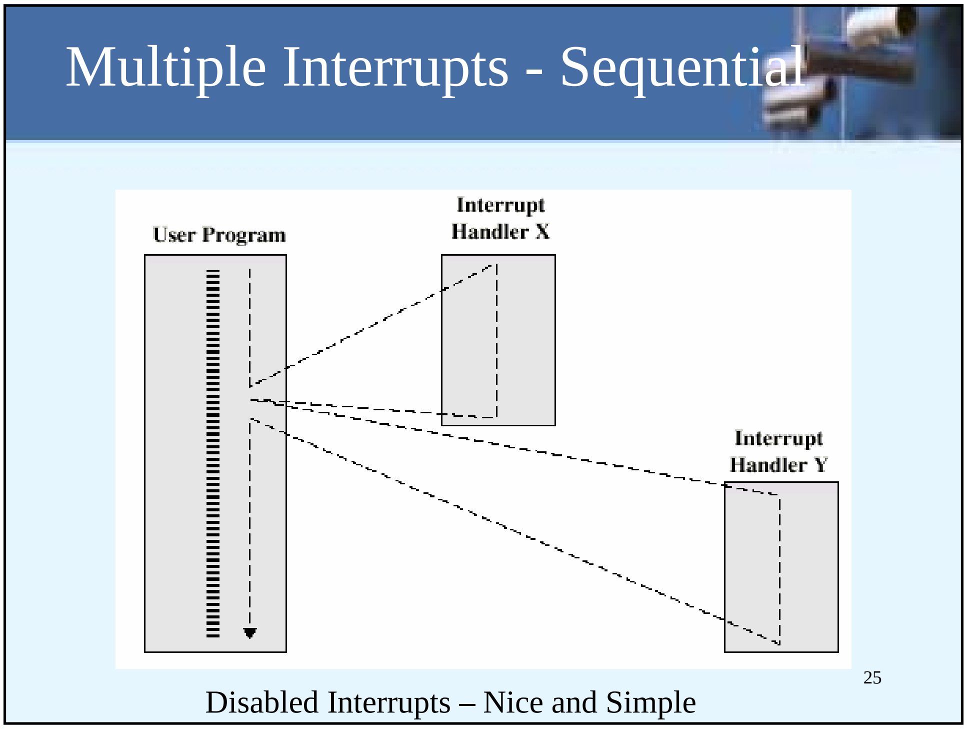
Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7	Module 8
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

High-Order Interleaving



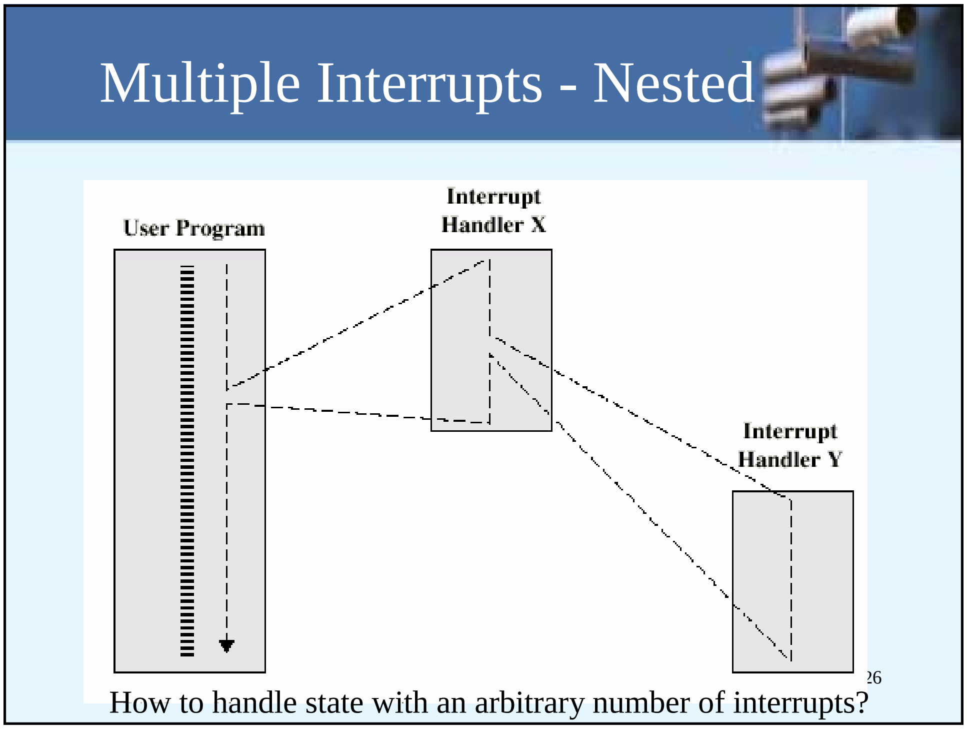
4.1 Introduction

- 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), or 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.



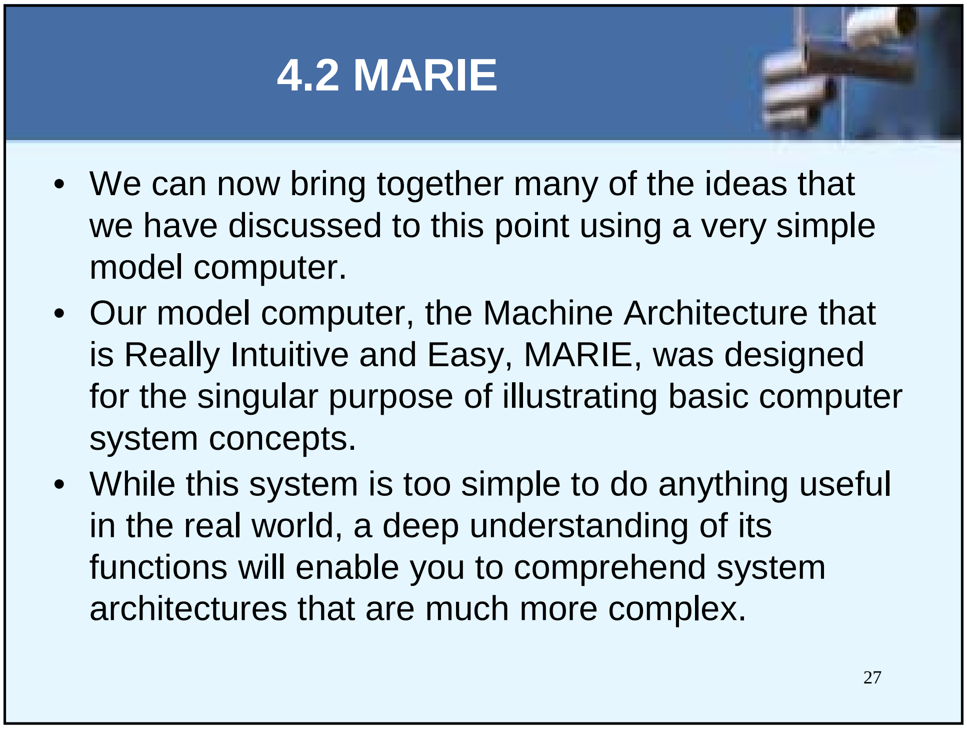
Multiple Interrupts - Sequential

Disabled Interrupts – Nice and Simple



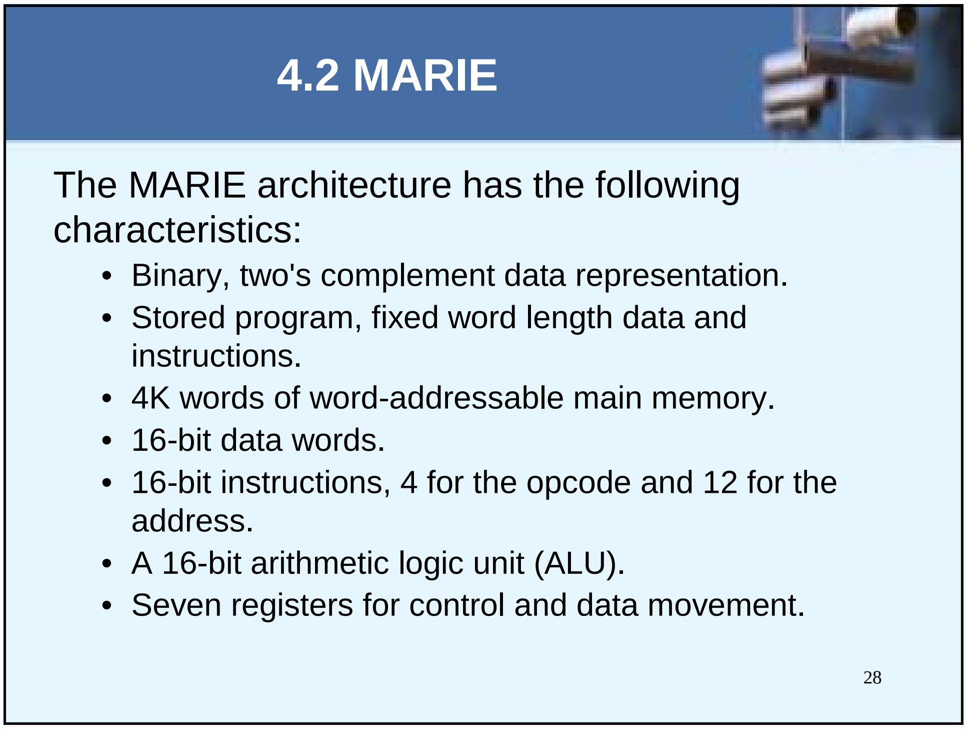
Multiple Interrupts - Nested

How to handle state with an arbitrary number of interrupts?



4.2 MARIE

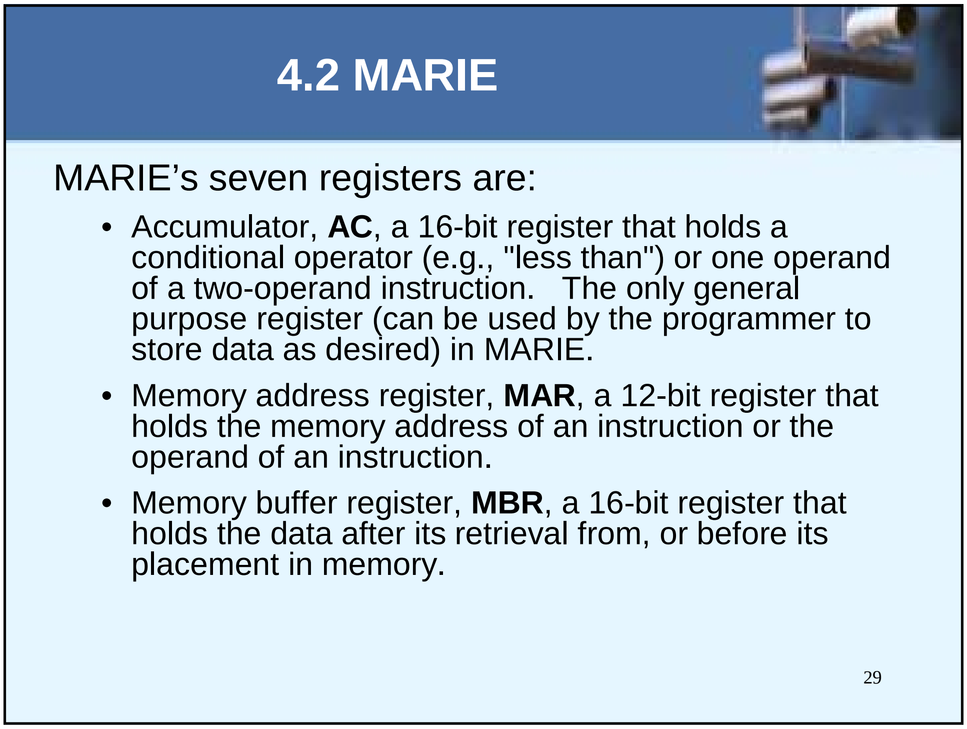
- We can now bring together many of the ideas that we have discussed to this point using a very simple model computer.
- Our model computer, the Machine Architecture that is Really Intuitive and Easy, MARIE, was designed for the singular purpose of illustrating basic computer system concepts.
- While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



4.2 MARIE

The MARIE architecture has the following characteristics:

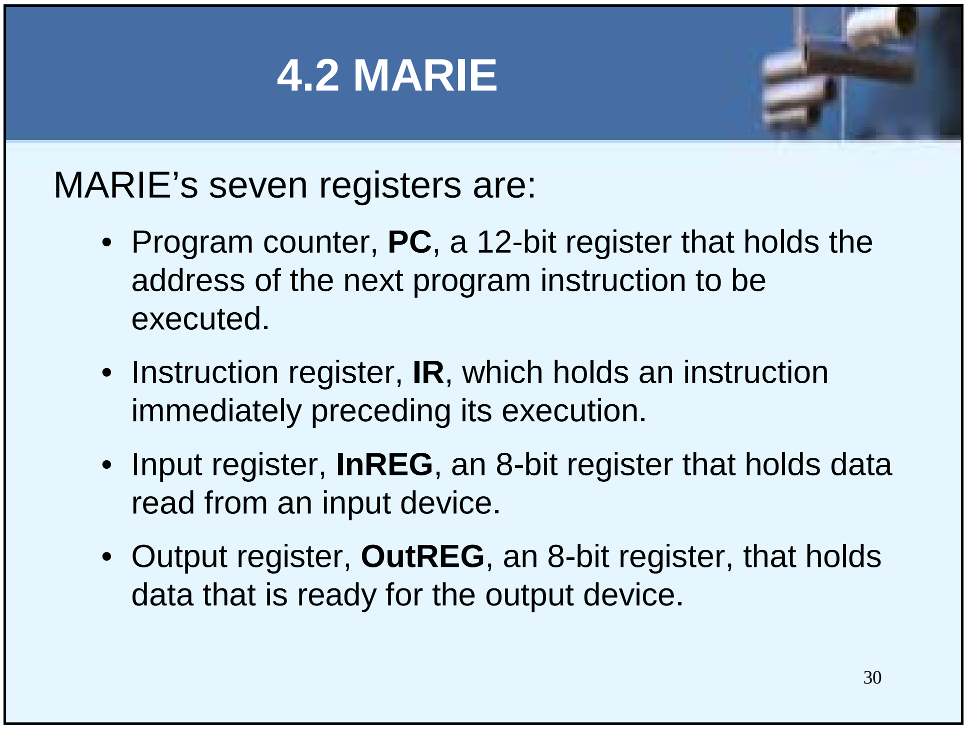
- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 4K words of word-addressable main memory.
- 16-bit data words.
- 16-bit instructions, 4 for the opcode and 12 for the address.
- A 16-bit arithmetic logic unit (ALU).
- Seven registers for control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



4.2 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.   The only general purpose register (can be used by the programmer to store data as desired) in MARIE.
- 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.



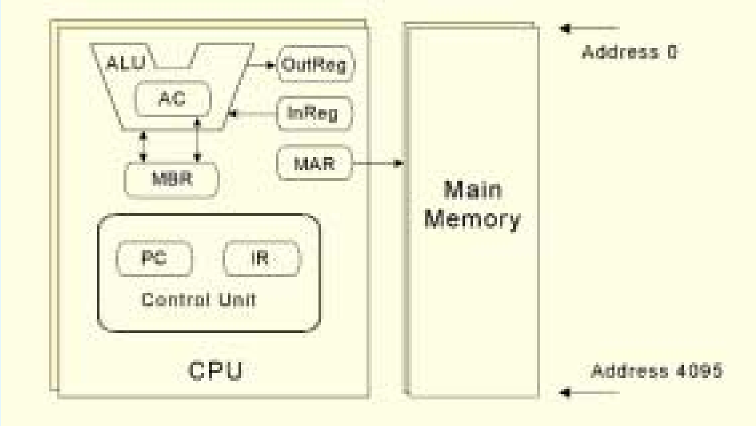
4.2 MARIE

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.

4.2 MARIE

This is the MARIE architecture shown graphically.

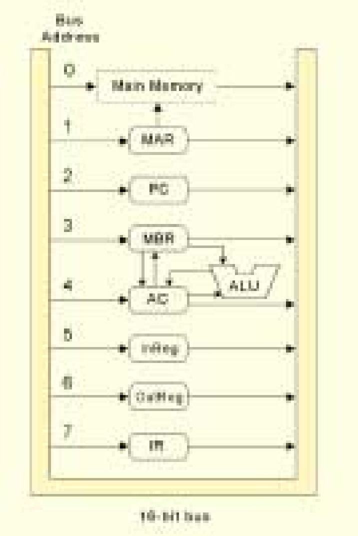


4.2 MARIE

- The registers are interconnected, and connected with main memory through a common data bus.
- Each device on the bus is identified by a unique number 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.

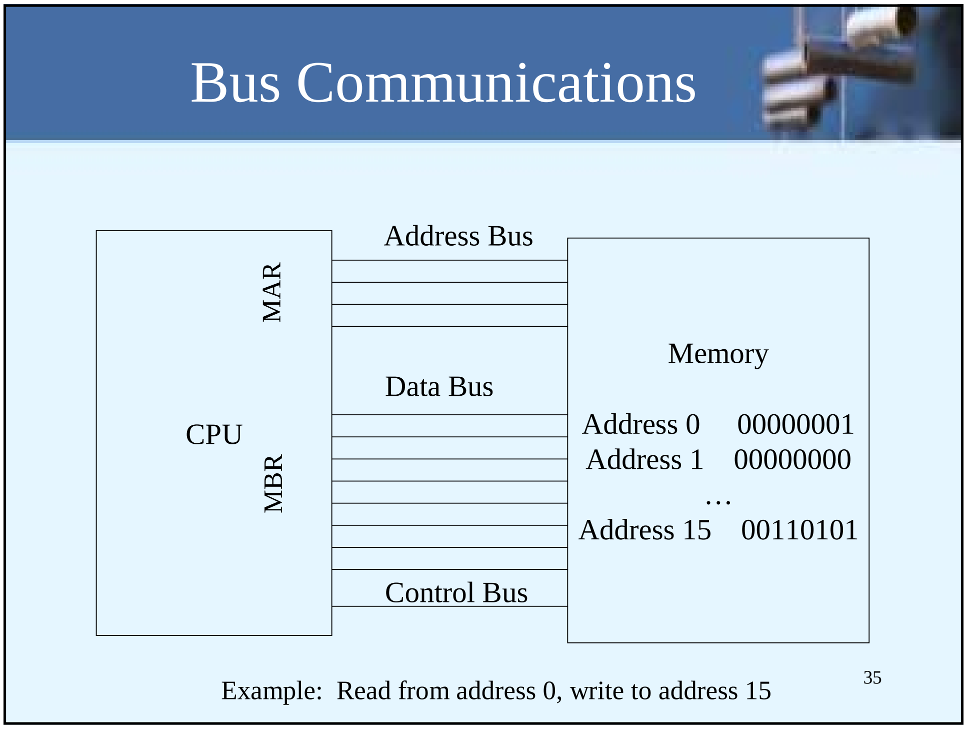
4.2 MARIE

This is the MARIE data path shown graphically.



MAR and MBR

- To get data from memory to the CPU
  - The address to read from is copied onto the MAR
  - The MAR sends its values on the address bus to memory
  - The control unit signals memory via the control bus that this is a “read” operation
  - Memory transmits the data at the address received on the address bus on the data bus
- To store data from the CPU to memory
  - The address to write to is copied onto the MAR
  - The data to write is copied onto the MBR
  - The MAR sends its values on the address bus to memory and the MBR sends its values on the data bus to memory
  - The control unit signals memory via the control bus that this is a “write” operation
  - Memory stores the data from the data bus into the address received from the address bus
- Transparent to the programmer
  - Since the MBR and MAR are intermediate steps to fetching and storing data, we will often leave off these details and just talk about writing directly from a register to memory, or from memory to a register



Bus Communications

Address Bus

MAR

CPU

MBR

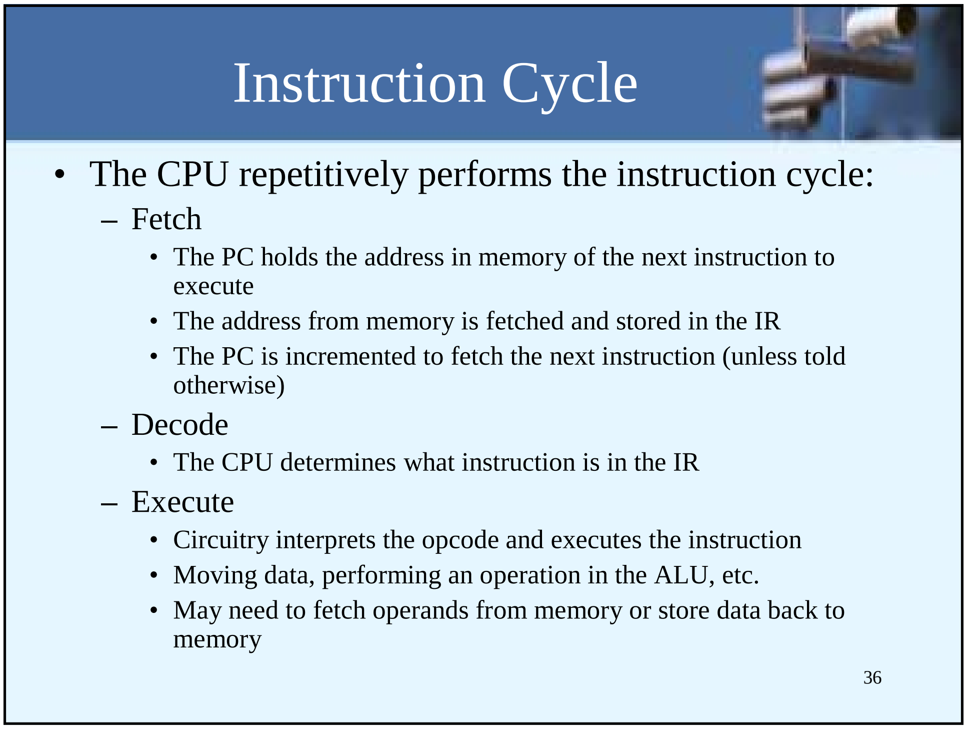
Data Bus

Control Bus

Memory

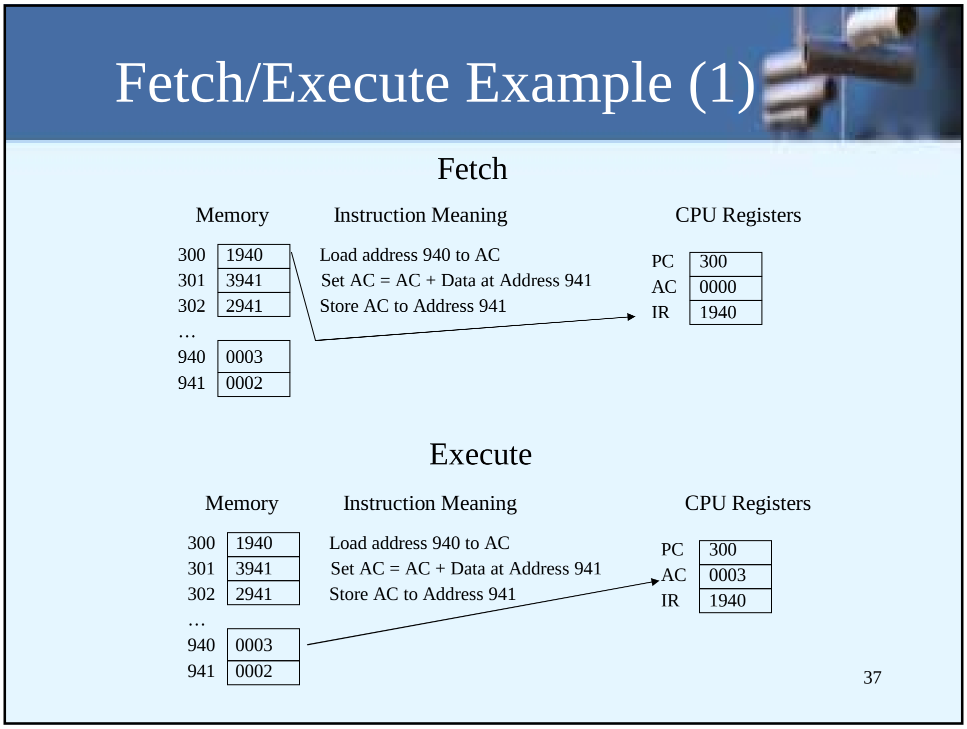
Address 0 00000001
Address 1 00000000
…
Address 15 00110101

Example: Read from address 0, write to address 15



Instruction Cycle

- The CPU repetitively performs the instruction cycle:
  - Fetch
    - The PC holds the address in memory of the next instruction to execute
    - The address from memory is fetched and stored in the IR
    - The PC is incremented to fetch the next instruction (unless told otherwise)
  - Decode
    - The CPU determines what instruction is in the IR
  - Execute
    - Circuitry interprets the opcode and executes the instruction
    - Moving data, performing an operation in the ALU, etc.
    - May need to fetch operands from memory or store data back to memory



Fetch/Execute Example (1)

Fetch

Memory

300	1940
301	3941
302	2941
…	
940	0003
941	0002

Instruction Meaning

Load address 940 to AC
Set AC = AC + Data at Address 941
Store AC to Address 941

CPU Registers

PC	300
AC	0000
IR	1940

Execute

Memory

300	1940
301	3941
302	2941
…	
940	0003
941	0002

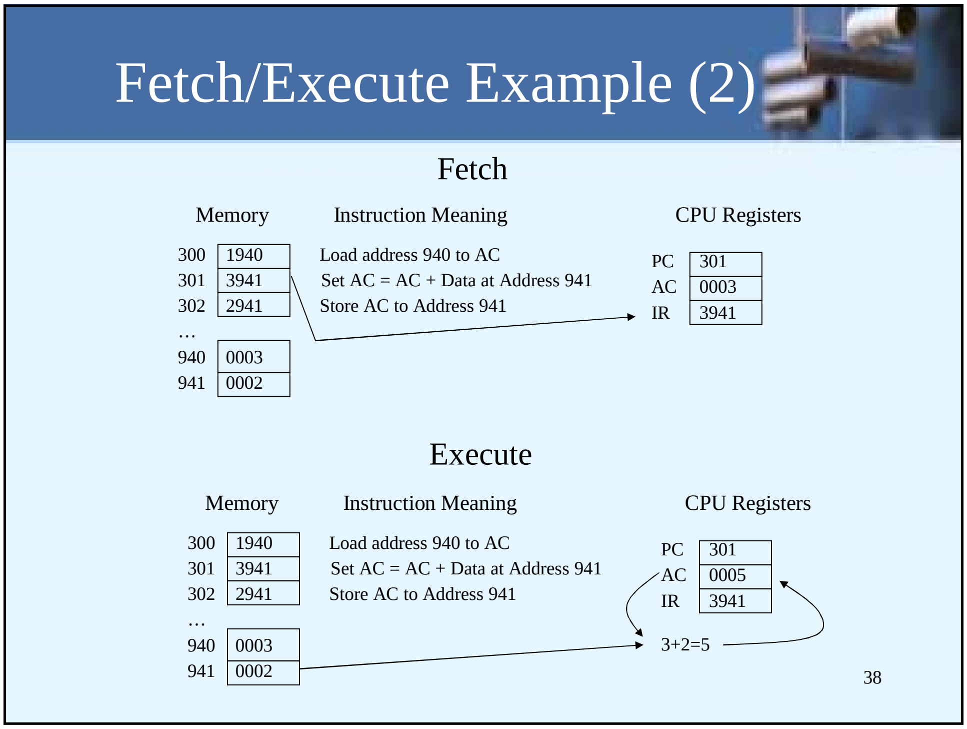
Instruction Meaning

Load address 940 to AC
Set AC = AC + Data at Address 941
Store AC to Address 941

CPU Registers

PC	300
AC	0003
IR	1940

37



Fetch/Execute Example (2)

Fetch

Memory

300	1940
301	3941
302	2941
…	
940	0003
941	0002

Instruction Meaning

Load address 940 to AC
Set AC = AC + Data at Address 941
Store AC to Address 941

CPU Registers

PC	301
AC	0003
IR	3941

Execute

Memory

300	1940
301	3941
302	2941
…	
940	0003
941	0002

Instruction Meaning

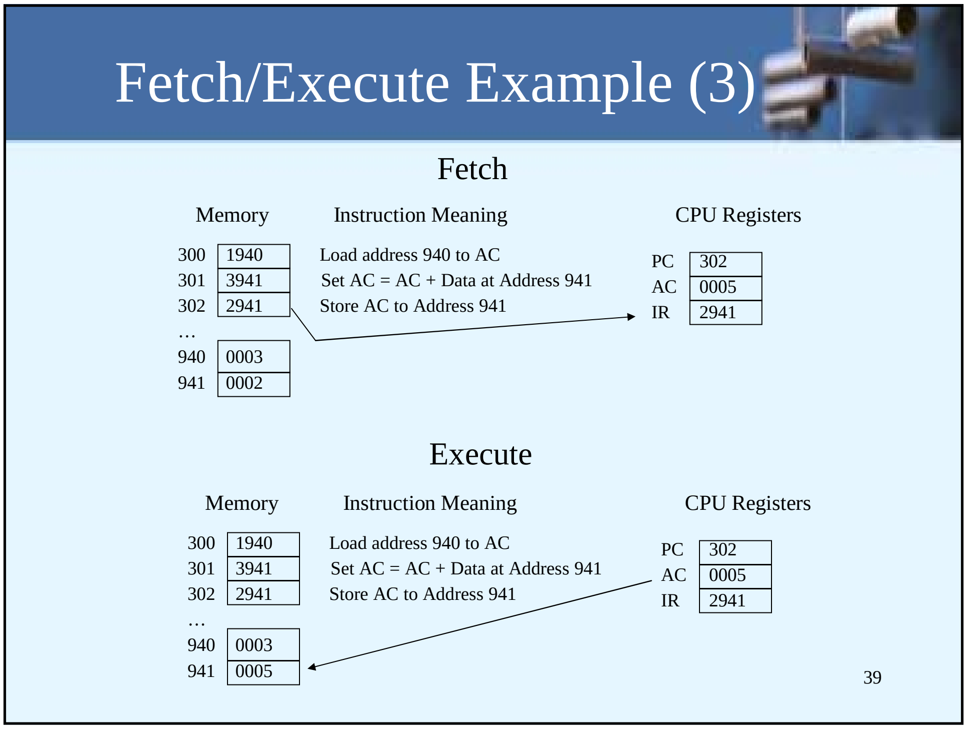
Load address 940 to AC
Set AC = AC + Data at Address 941
Store AC to Address 941

CPU Registers

PC	301
AC	0005
IR	3941

3+2=5

38



Fetch/Execute Example (3)

Fetch

Memory

300	1940
301	3941
302	2941
…	
940	0003
941	0002

Instruction Meaning

Load address 940 to AC
Set AC = AC + Data at Address 941
Store AC to Address 941

CPU Registers

PC	302
AC	0005
IR	2941

Execute

Memory

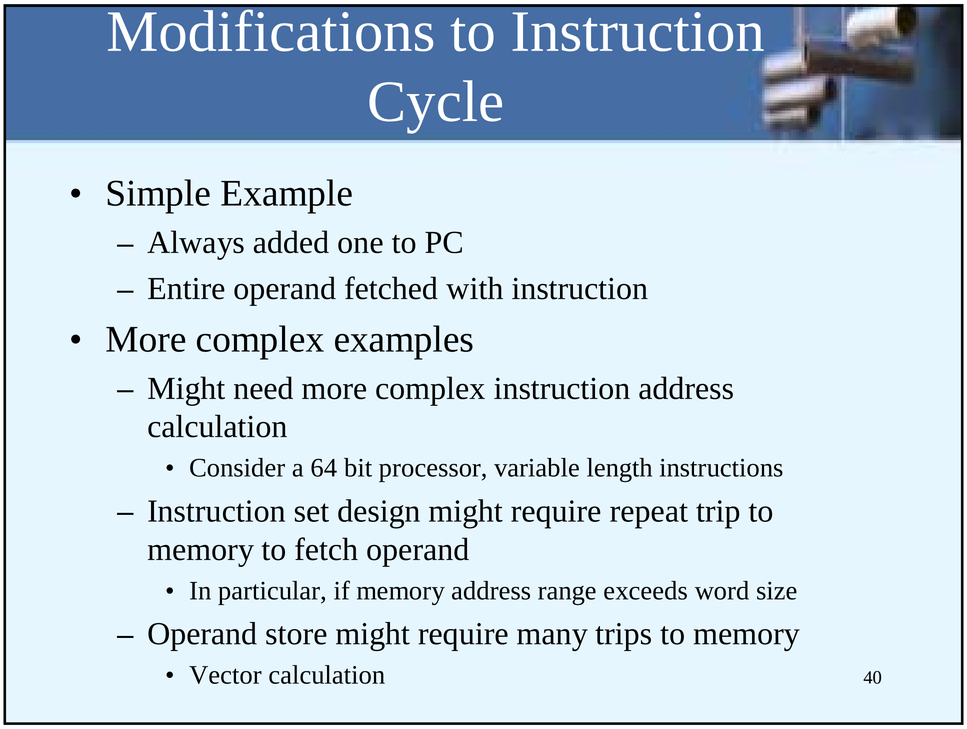
300	1940
301	3941
302	2941
…	
940	0003
941	0005

Instruction Meaning

Load address 940 to AC
Set AC = AC + Data at Address 941
Store AC to Address 941

CPU Registers

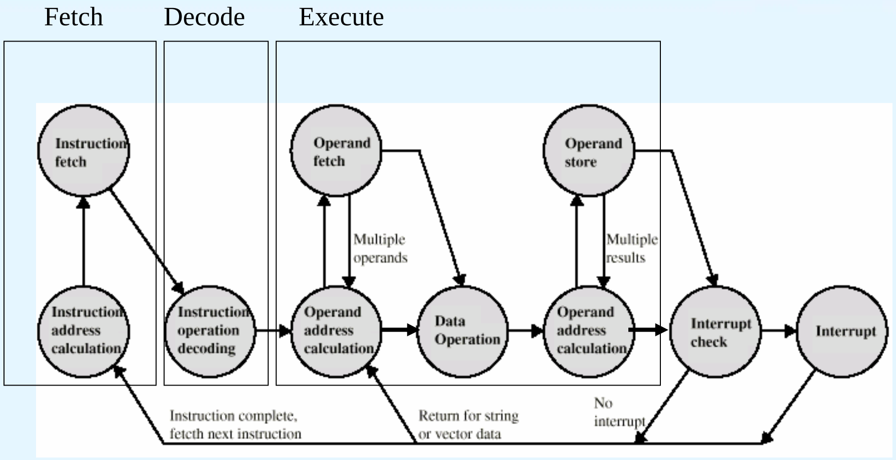
PC	302
AC	0005
IR	2941



Modifications to Instruction Cycle

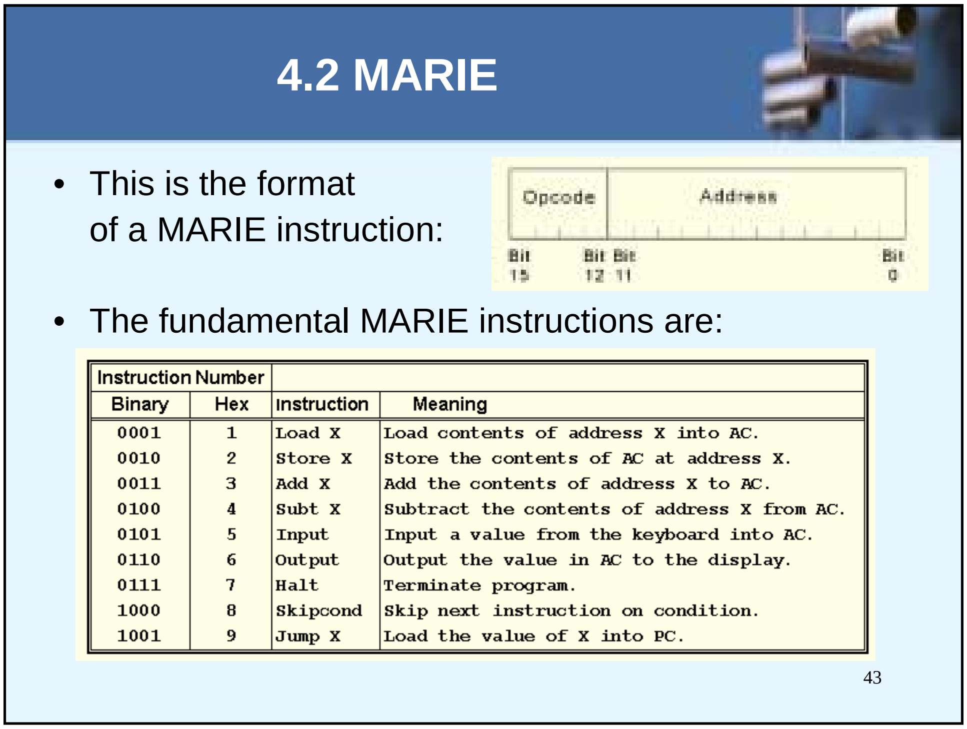
- Simple Example
  - Always added one to PC
  - Entire operand fetched with instruction
- More complex examples
  - Might need more complex instruction address calculation
    - Consider a 64 bit processor, variable length instructions
  - Instruction set design might require repeat trip to memory to fetch operand
    - In particular, if memory address range exceeds word size
  - Operand store might require many trips to memory
    - Vector calculation

Instruction Cycle (with Interrupts) - State Diagram


Fetch
Decode
Execute
Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data Operation
Operand address calculation
Interrupt check
Interrupt
Instruction complete, fetch next instruction
Return for string or vector data
No interrupt

4.2 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 thirteen instructions.



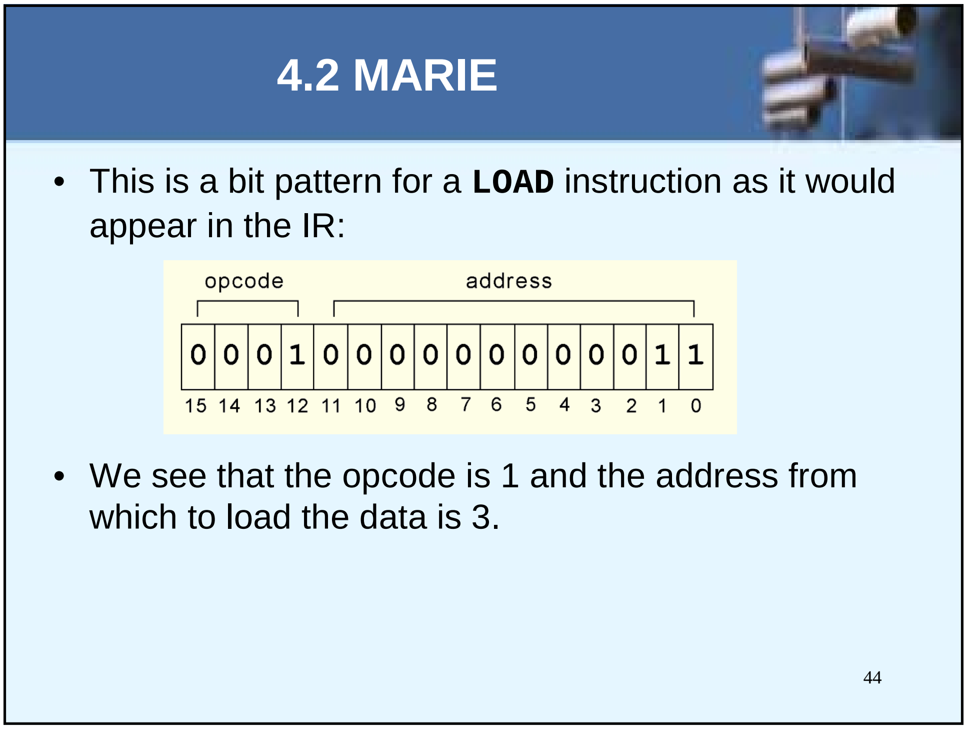
4.2 MARIE

- This is the format of a MARIE instruction:

Opcode
Address
Bit 15
Bit 12
Bit 11
Bit 0

- 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.



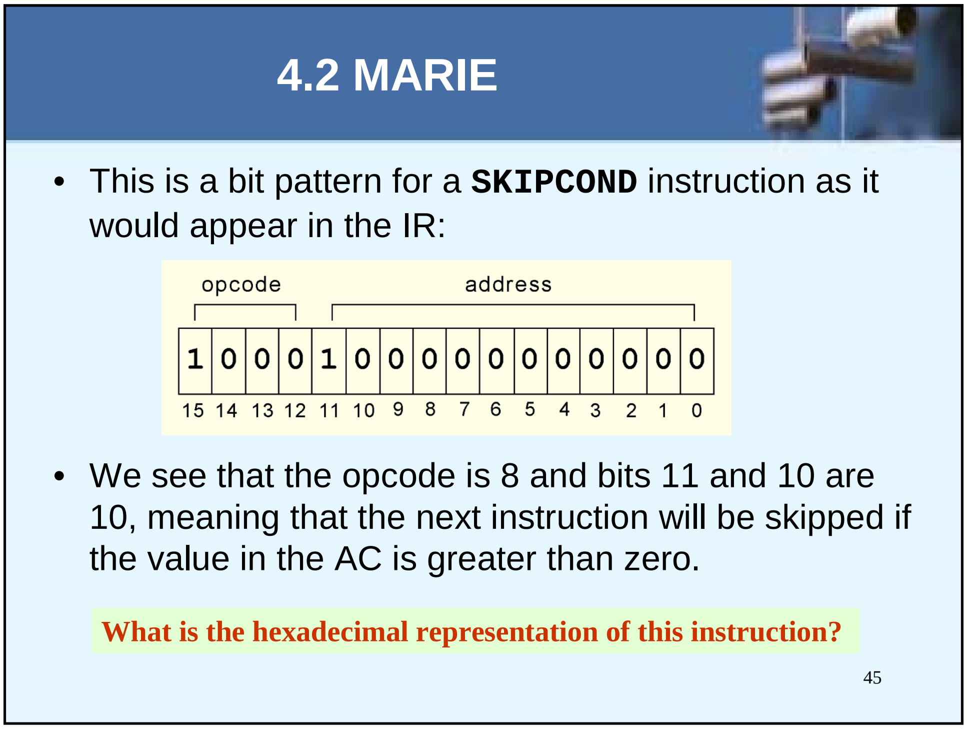
4.2 MARIE

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

opcode
address

0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

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



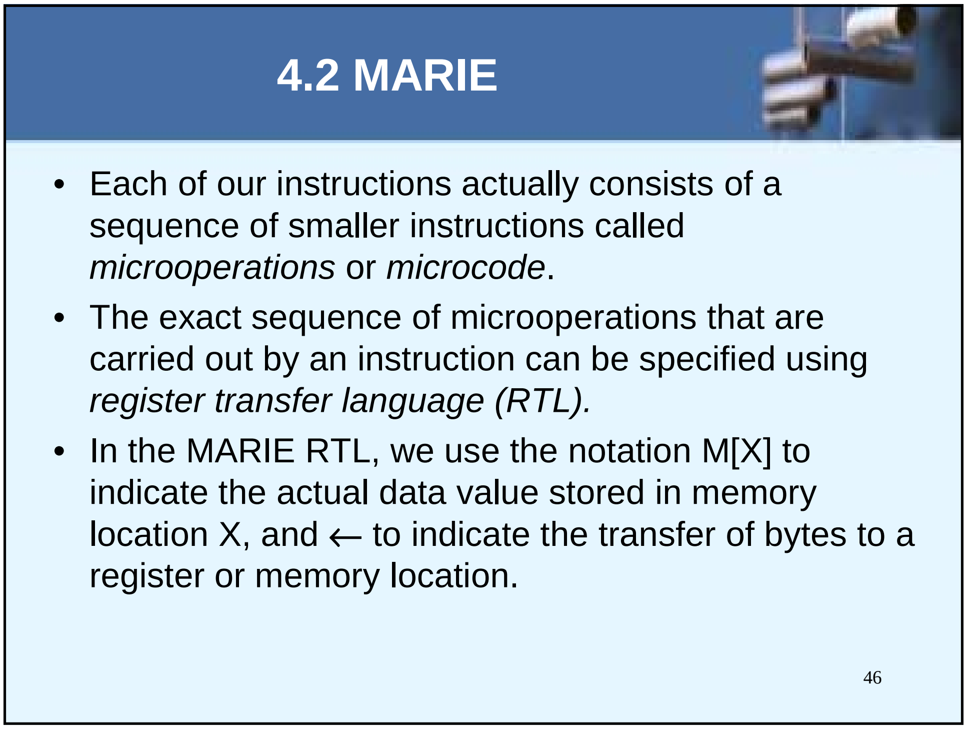
4.2 MARIE

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

opcode | | | | address | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

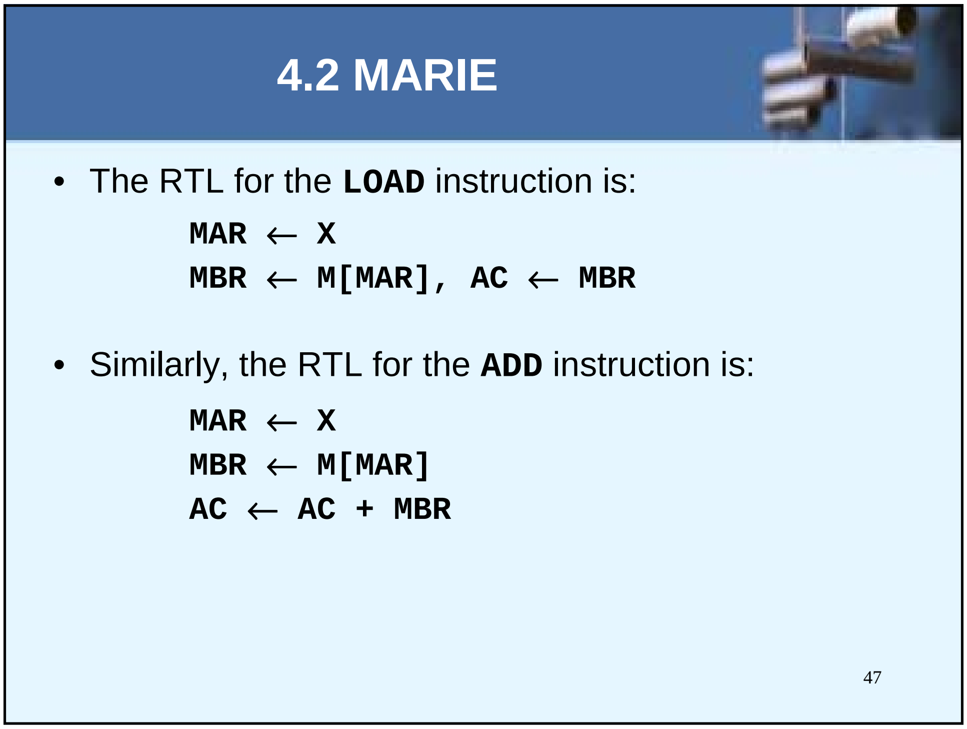
- We see that the opcode is 8 and bits 11 and 10 are 10, meaning that the next instruction will be skipped if the value in the AC is greater than zero.

What is the hexadecimal representation of this instruction?



4.2 MARIE

- Each of our instructions actually consists of a sequence of smaller instructions called *microoperations* or *microcode*.
- 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.



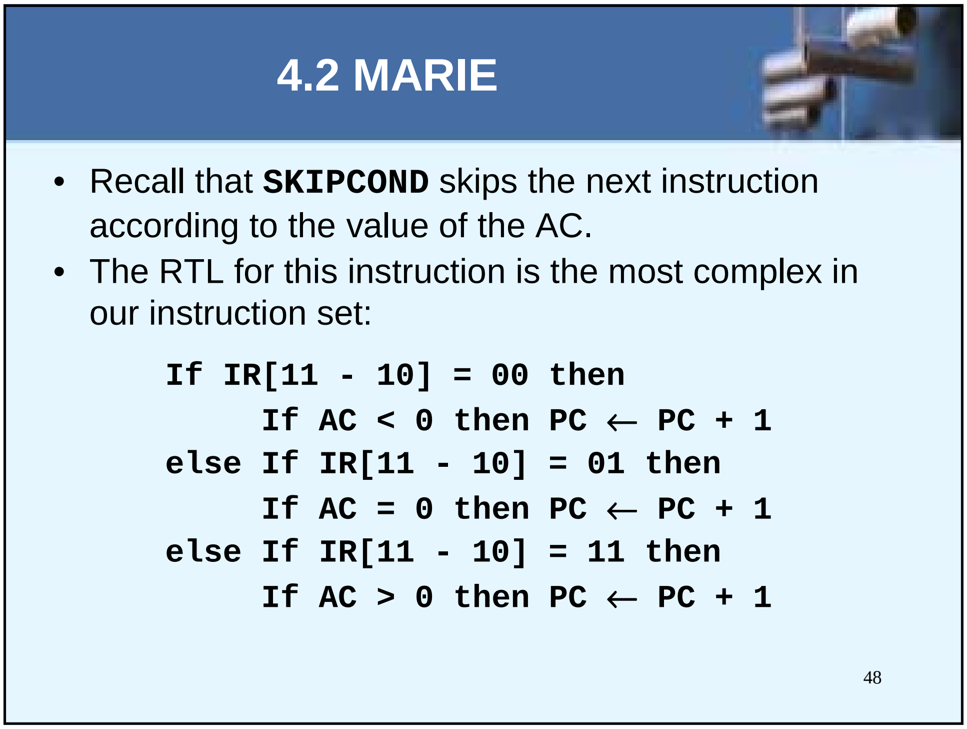
4.2 MARIE

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

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

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

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



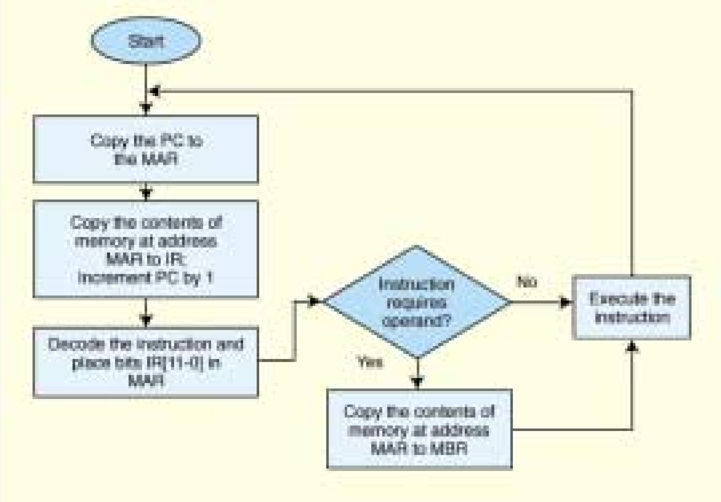
4.2 MARIE

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

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

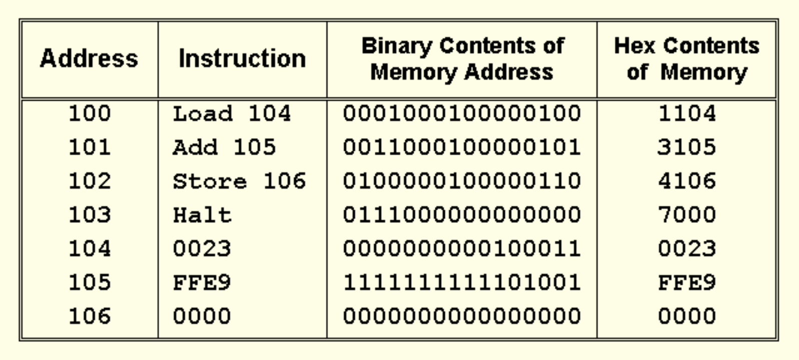
4.3 Instruction Processing

Another view of the Fetch/Execute Cycle

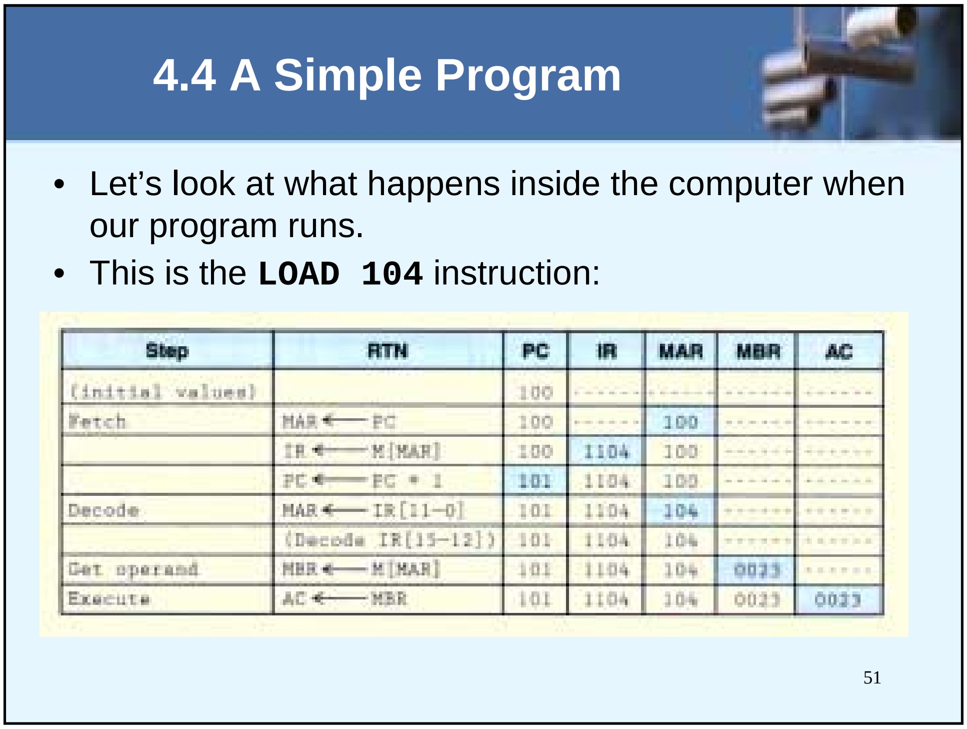

Start
Copy the PC to the MAR
Copy the contents of memory at address MAR to IR; Increment PC by 1
Decode the instruction and place bits IR[11-0] in MAR
Instruction requires operand?
No
Yes
Execute the instruction
Copy the contents of memory at address MAR to MBR

4.4 A Simple Program

- Consider the simple MARIE program given below. We show a set of mnemonic instructions stored at addresses 100 - 106 (hex):



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

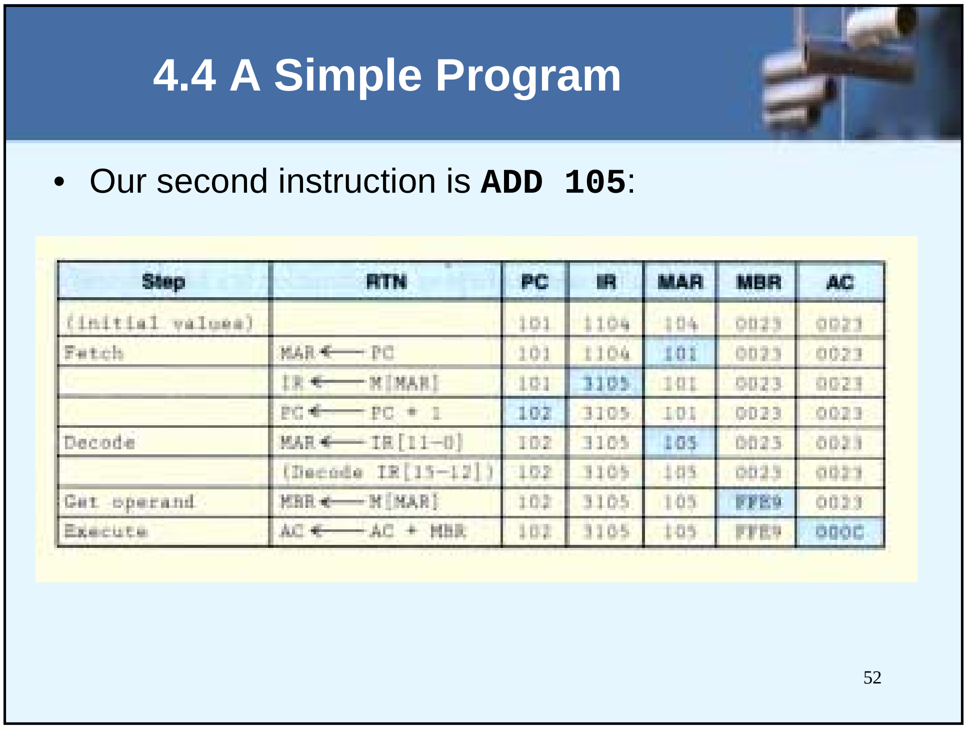


4.4 A Simple Program

- 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 ← 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

51

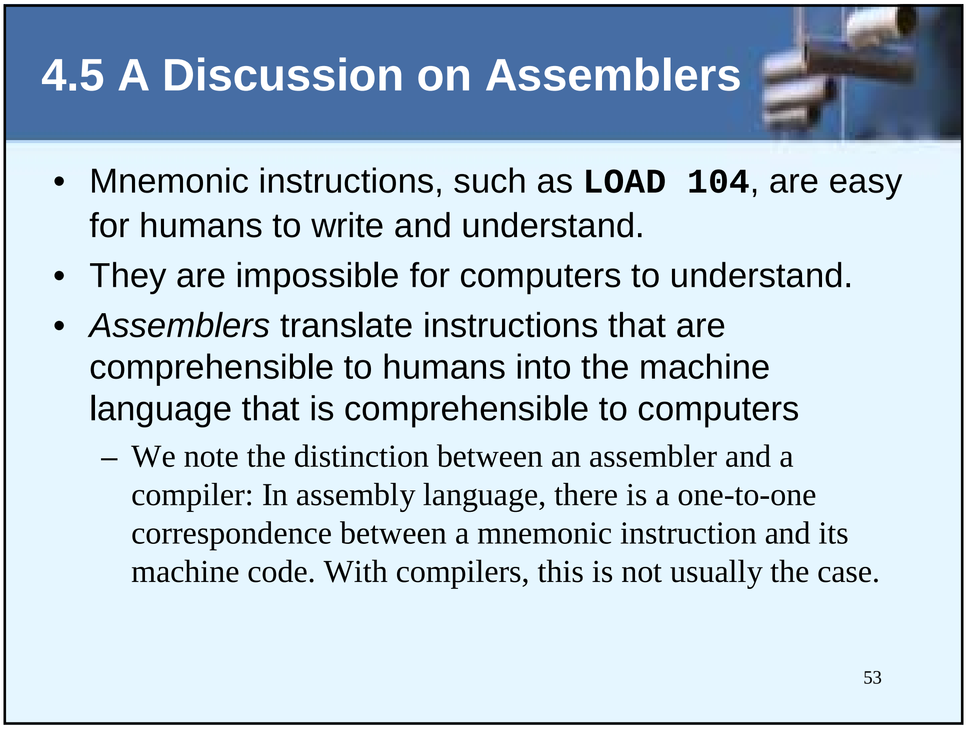


4.4 A Simple Program

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

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

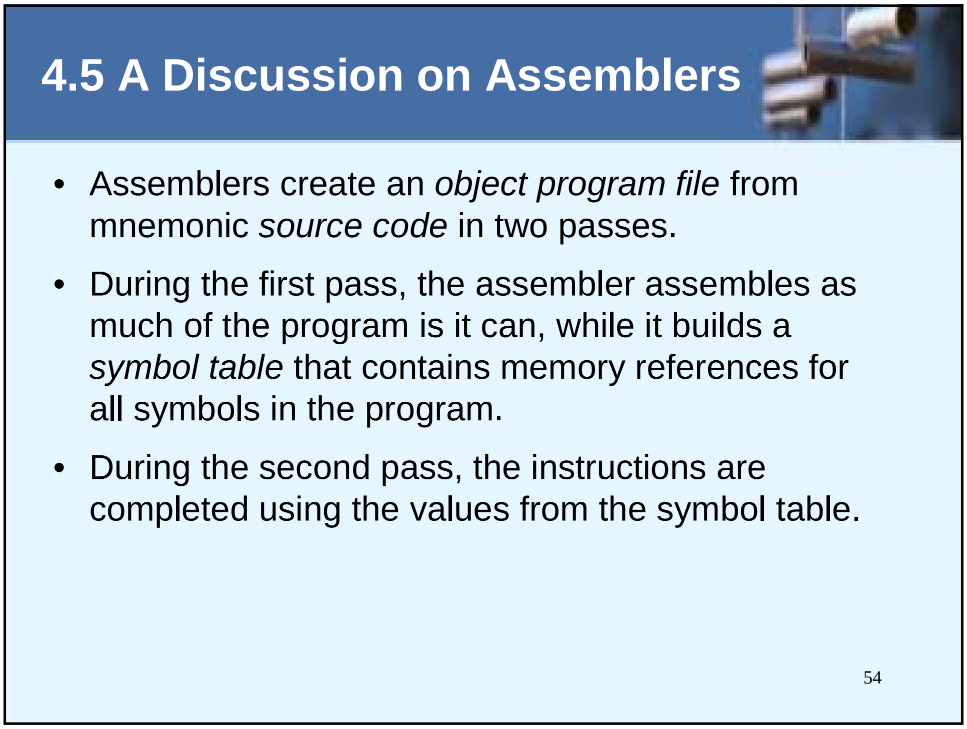
52



4.5 A Discussion on Assemblers

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

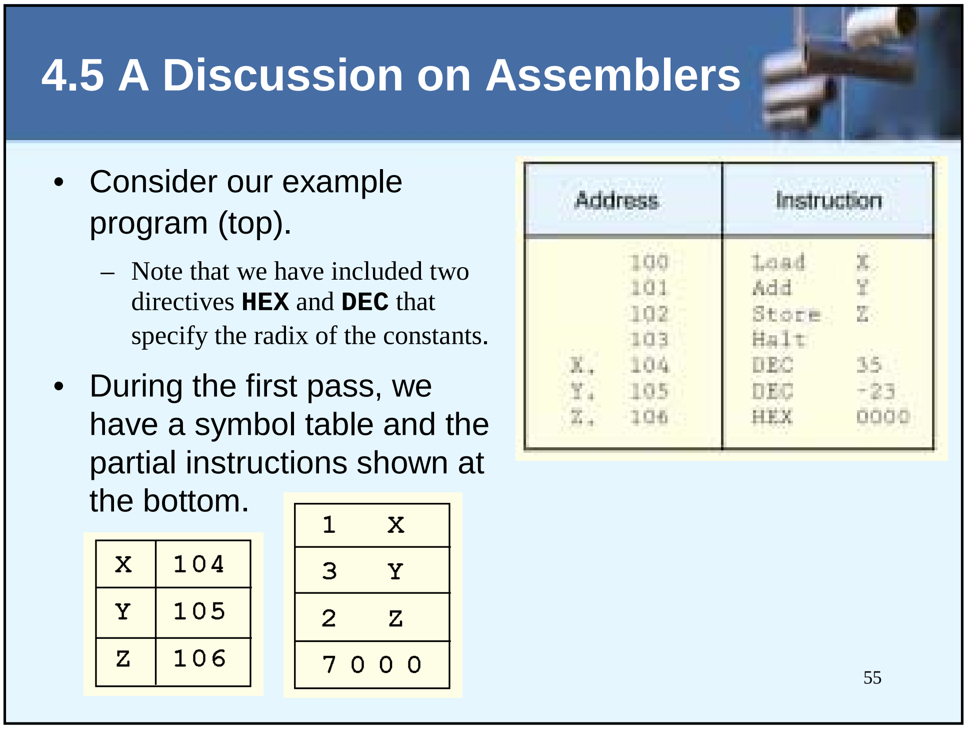
53



4.5 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 is 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.

54



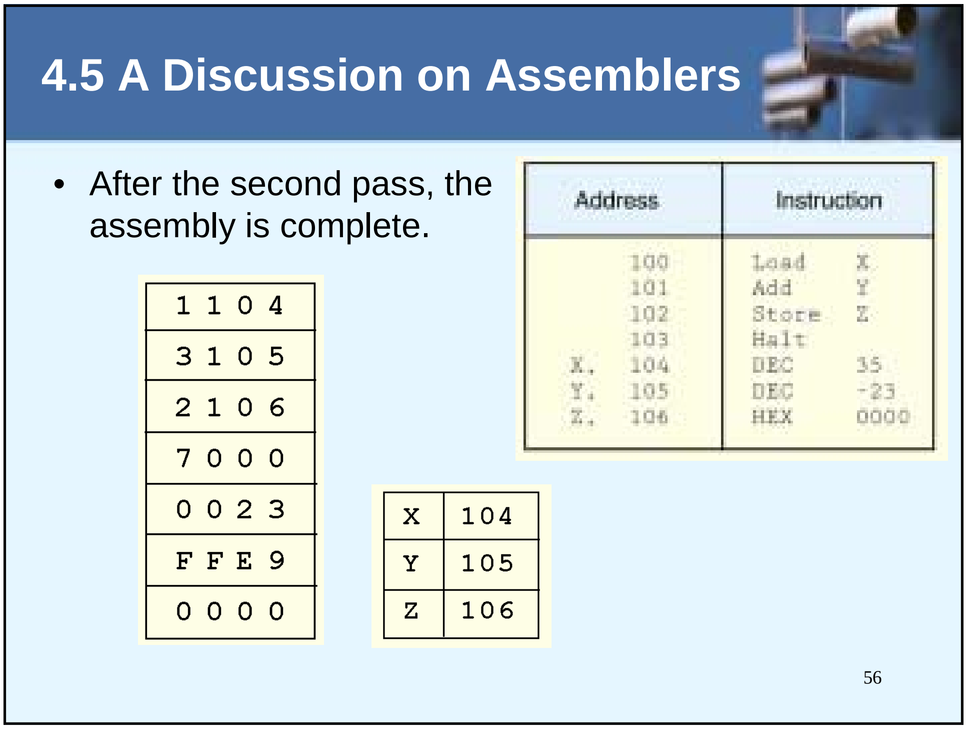
4.5 A Discussion on Assemblers

- Consider our example program (top).
  - Note that we have included two directives **HEX** and **DEC** that specify the radix of the constants.
- During the first pass, we have a symbol table and the partial instructions shown at the bottom.

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

X	104
Y	105
Z	106

1	X
3	Y
2	Z
7000	



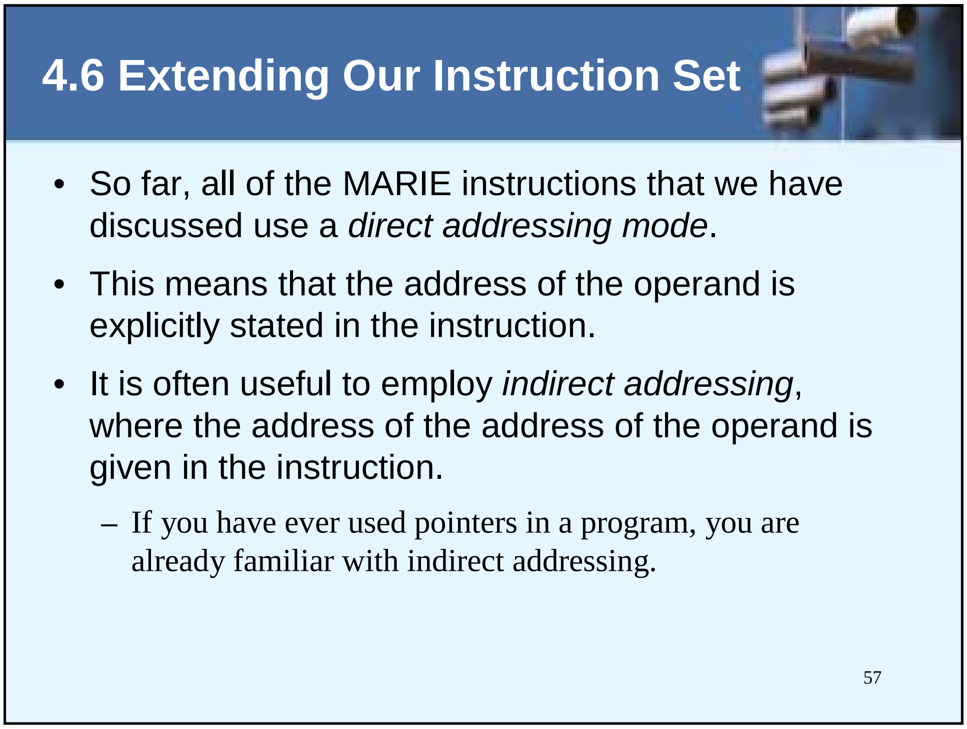
4.5 A Discussion on Assemblers

- After the second pass, the assembly is complete.

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

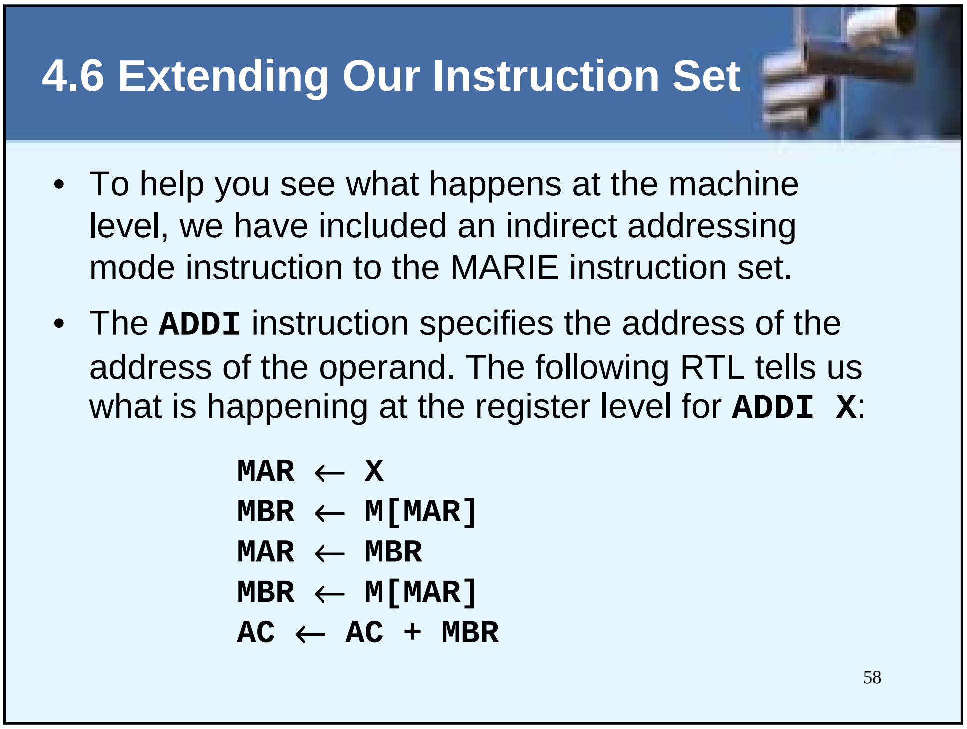
1104
3105
2106
7000
0023
FFE9
0000

X	104
Y	105
Z	106



4.6 Extending Our Instruction Set

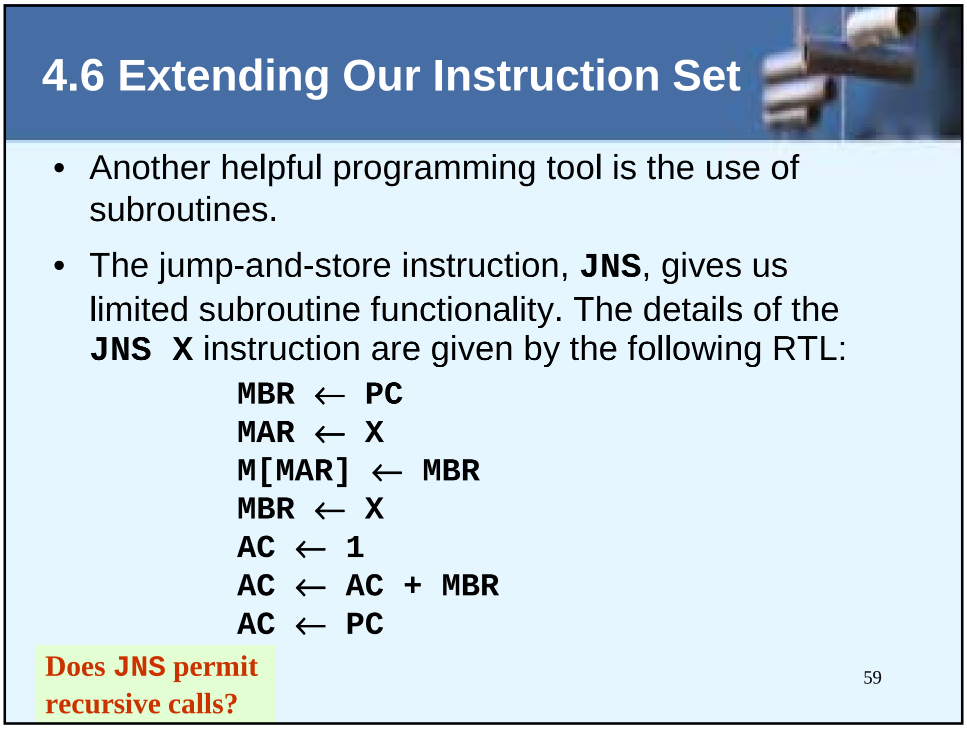
- So far, all of the MARIE instructions that we have discussed use a *direct addressing mode*.
- This means that the address of the operand is explicitly stated in the instruction.
- It is often useful to employ *indirect addressing*, where the address of the address of the operand is given in the instruction.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



4.6 Extending Our Instruction Set

- To help you see what happens at the machine level, we have included an indirect addressing mode instruction to the MARIE instruction set.
- The **`ADDI`** instruction specifies the address of the address of the operand. The following RTL tells us what is happening at the register level for **`ADDI X`**:

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

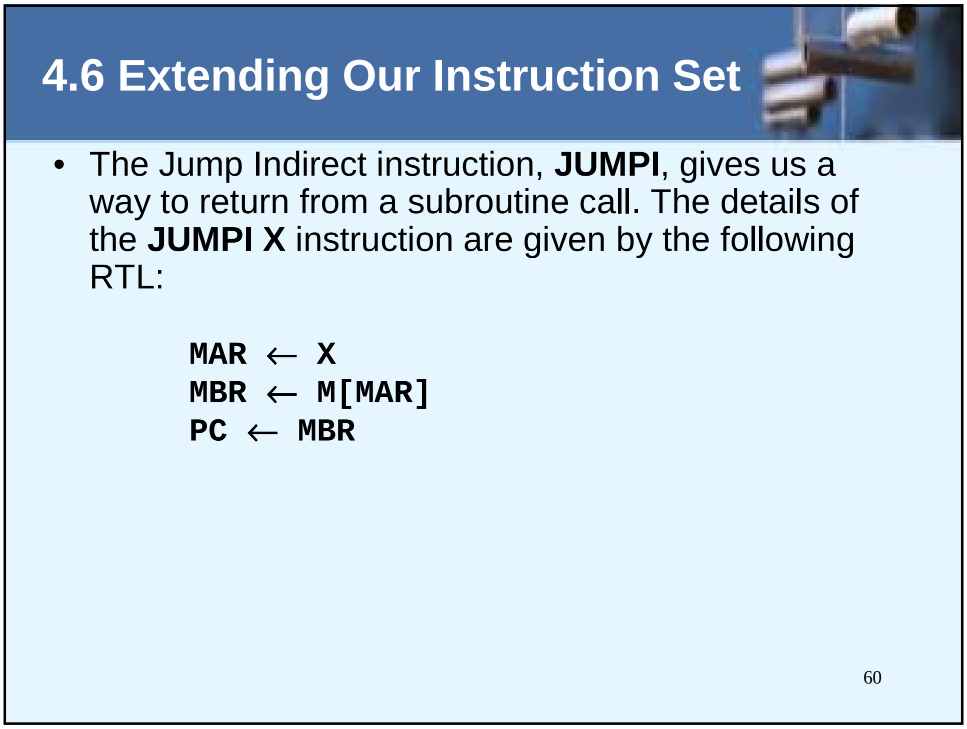


4.6 Extending Our Instruction Set

- Another helpful programming tool is the use of subroutines.
- The jump-and-store instruction, **JNS**, gives us limited subroutine functionality. The details of the **JNS X** instruction are given by the following RTL:

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

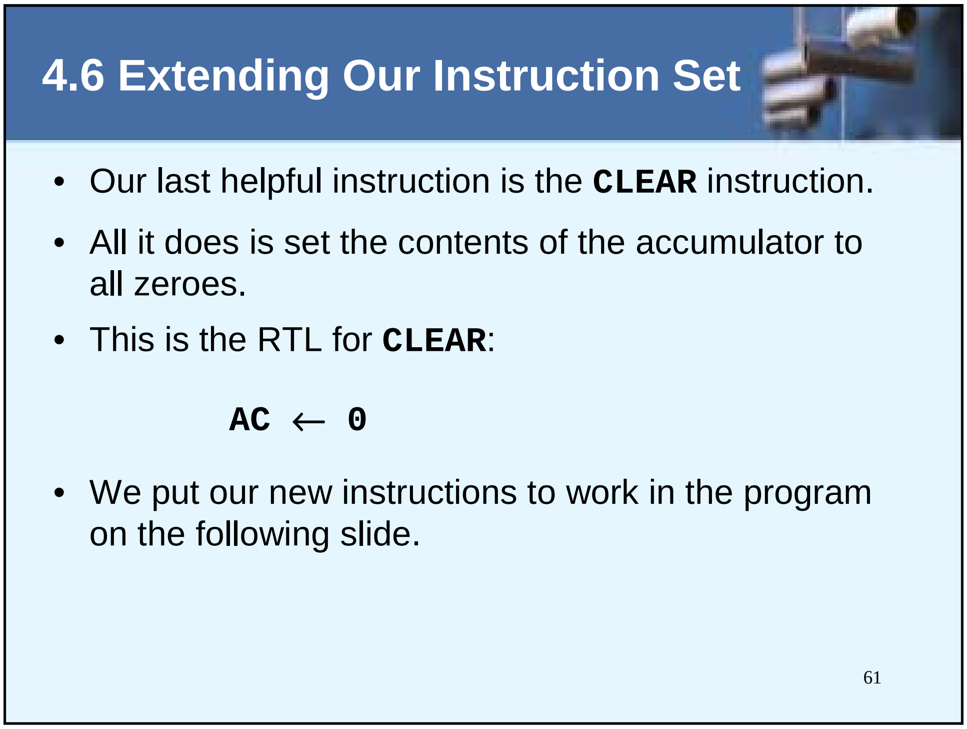
Does JNS permit recursive calls?



4.6 Extending Our Instruction Set

- The Jump Indirect instruction, **JUMPI**, gives us a way to return from a subroutine call. The details of the **JUMPI X** instruction are given by the following RTL:

```
MAR ← X
MBR ← M[MAR]
PC ← MBR
```

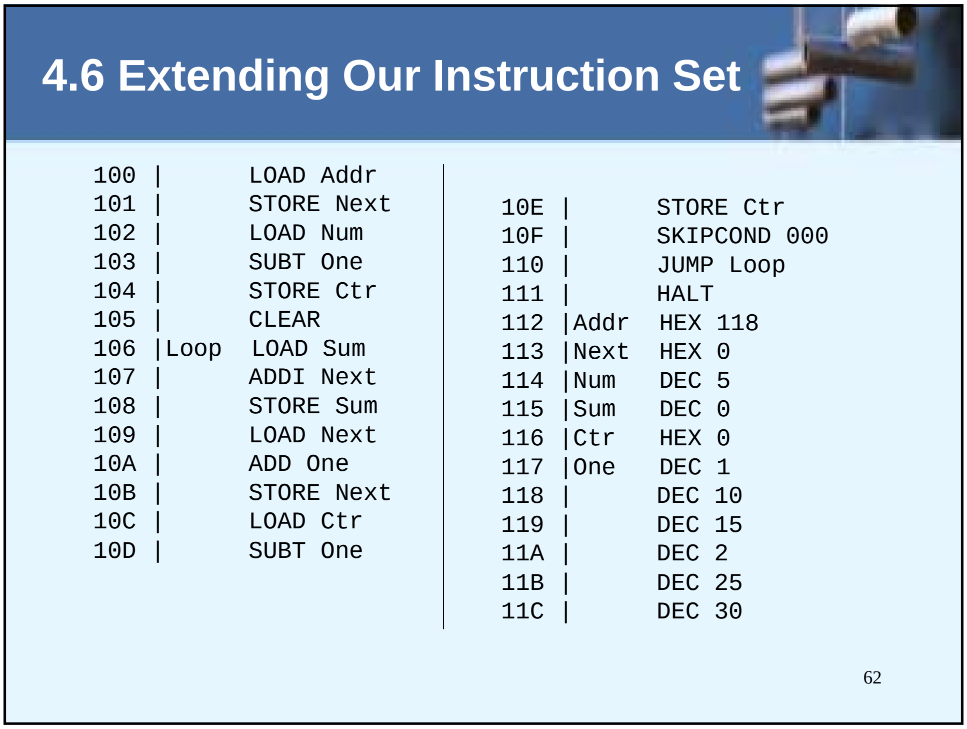


4.6 Extending Our Instruction Set

- Our last helpful 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

- We put our new instructions to work in the program on the following slide.



4.6 Extending Our Instruction Set

```
100 |      LOAD Addr
101 |      STORE Next
102 |      LOAD Num
103 |      SUBT One
104 |      STORE Ctr
105 |      CLEAR
106 |Loop  LOAD Sum
107 |      ADDI Next
108 |      STORE Sum
109 |      LOAD Next
10A |      ADD One
10B |      STORE Next
10C |      LOAD Ctr
10D |      SUBT One
```

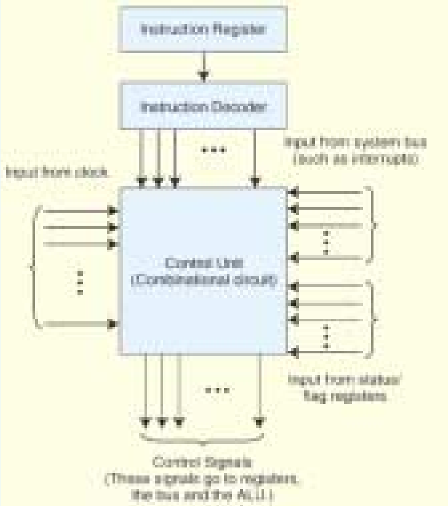
```
10E |      STORE Ctr
10F |      SKIPCOND 000
110 |      JUMP Loop
111 |      HALT
112 |Addr  HEX 118
113 |Next  HEX 0
114 |Num   DEC 5
115 |Sum   DEC 0
116 |Ctr   HEX 0
117 |One   DEC 1
118 |      DEC 10
119 |      DEC 15
11A |      DEC 2
11B |      DEC 25
11C |      DEC 30
```

4.7 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*.
  - With microprogrammed control, a small program is placed into read-only memory in the microcontroller.
  - Hardwired controllers implement this program using digital logic components.

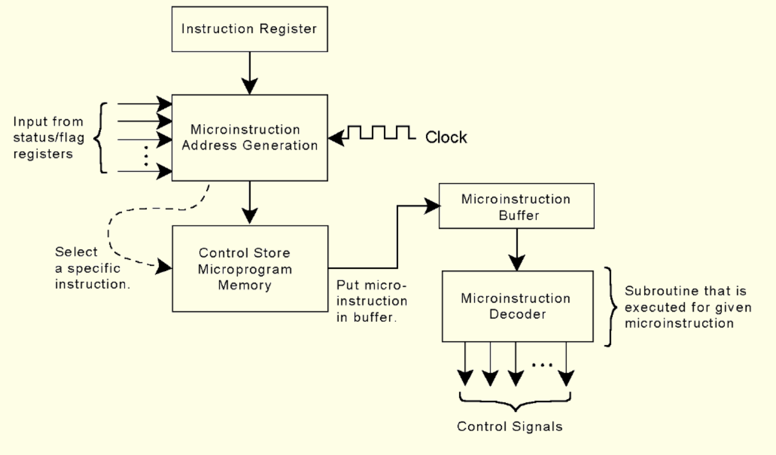
4.7 A Discussion on Decoding

- For example, a hardwired control unit for our simple system would need a 4-to-14 decoder to decode the opcode of an instruction.
- The block diagram at the right, shows a general configuration for a hardwired control unit.


Instruction Register
Instruction Decoder
Input from system bus (such as interrupts)
Input from clock
Control Unit (Combinational circuit)
Input from status/ flag registers
Control Signals (These signals go to registers, the bus and the ALU.)

4.7 A Discussion on Decoding

- In microprogrammed control, the *control store* is kept in ROM, PROM, or EPROM firmware, as shown below.



Instruction Register

Input from status/flag registers

Microinstruction Address Generation

Clock

Select a specific instruction.

Control Store Microprogram Memory

Put micro-instruction in buffer.

Microinstruction Buffer

Microinstruction Decoder

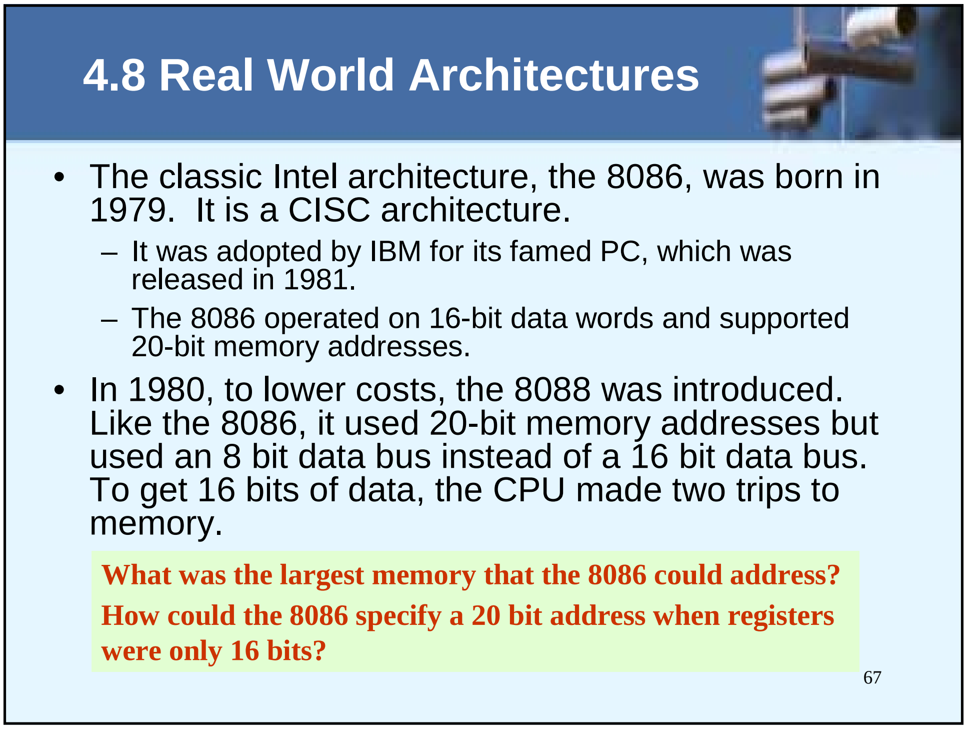
Subroutine that is executed for given microinstruction

Control Signals

4.8 Real World Architectures

- MARIE shares many features with modern architectures but it is not an accurate depiction of them.
- In the following slides, we briefly examine two machine architectures.
- We will look at an Intel architecture, which is a CISC machine and MIPS, which is a RISC machine.
  - CISC is an acronym for complex instruction set computer.
  - RISC stands for reduced instruction set computer.

We delve into the "RISC versus CISC" argument in Chapter 9.

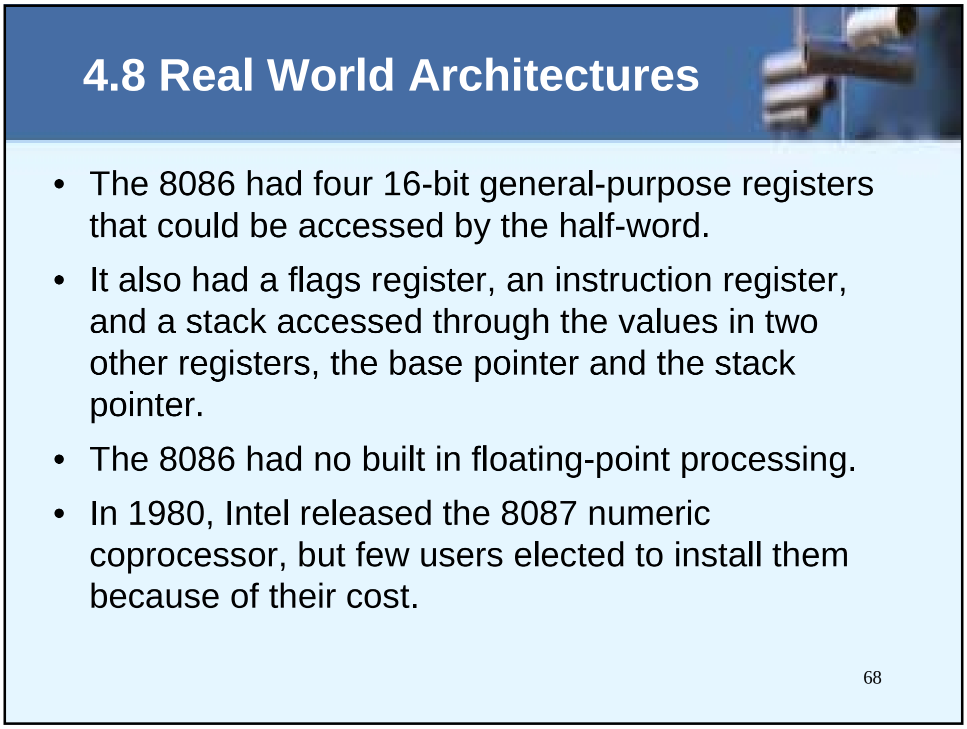


4.8 Real World Architectures

- 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.
- In 1980, to lower costs, the 8088 was introduced. Like the 8086, it used 20-bit memory addresses but used an 8 bit data bus instead of a 16 bit data bus. To get 16 bits of data, the CPU made two trips to memory.

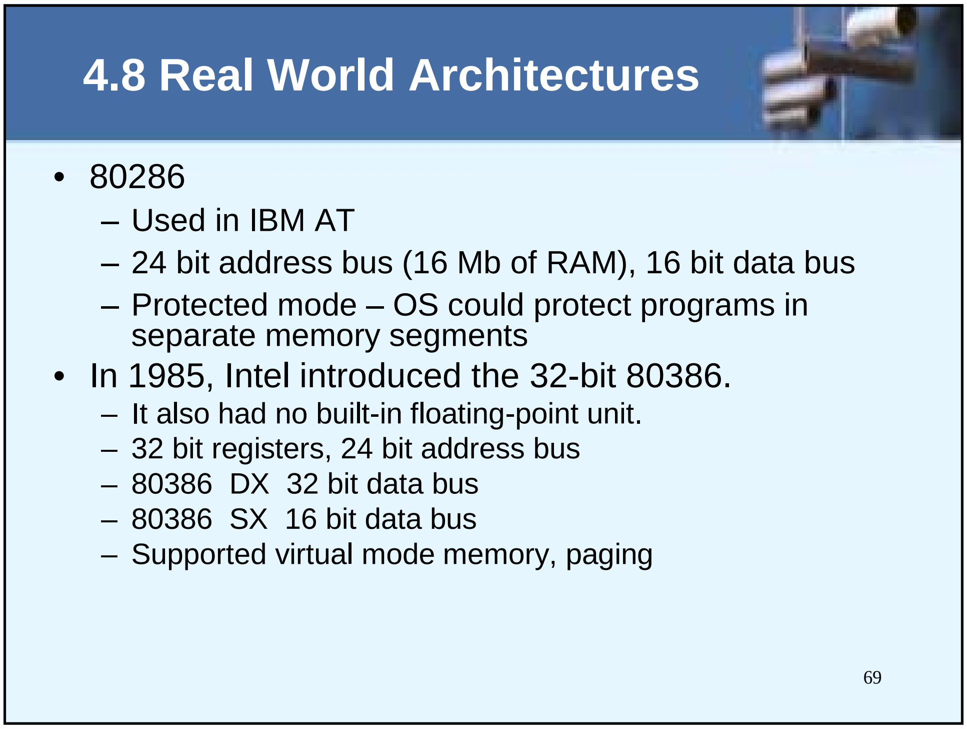
What was the largest memory that the 8086 could address?

How could the 8086 specify a 20 bit address when registers were only 16 bits?



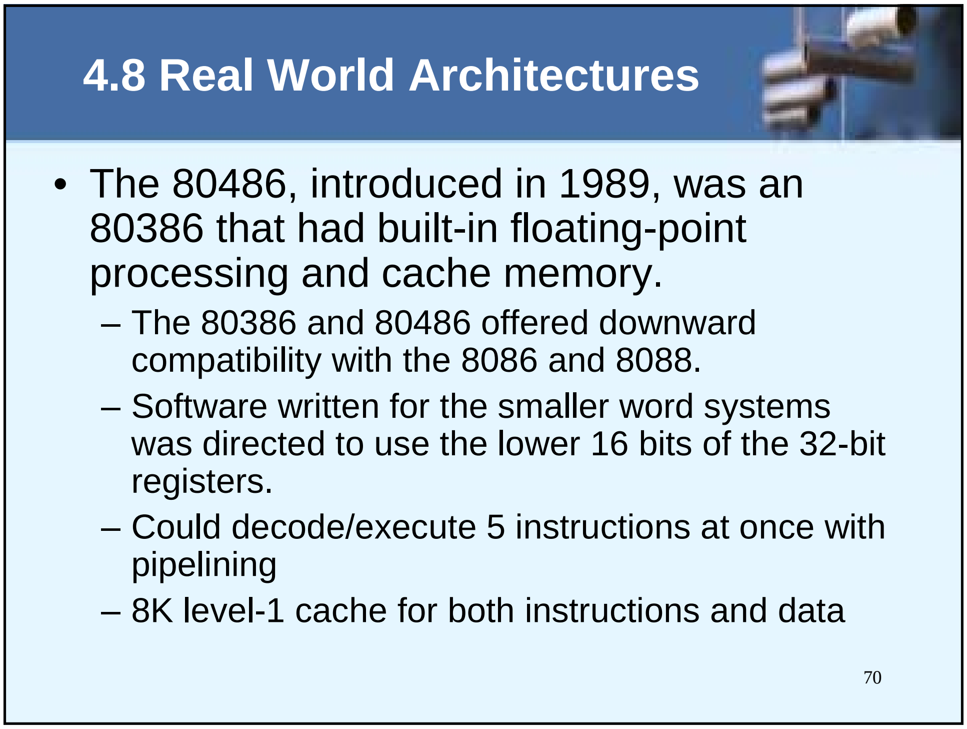
4.8 Real World Architectures

- 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 cost.



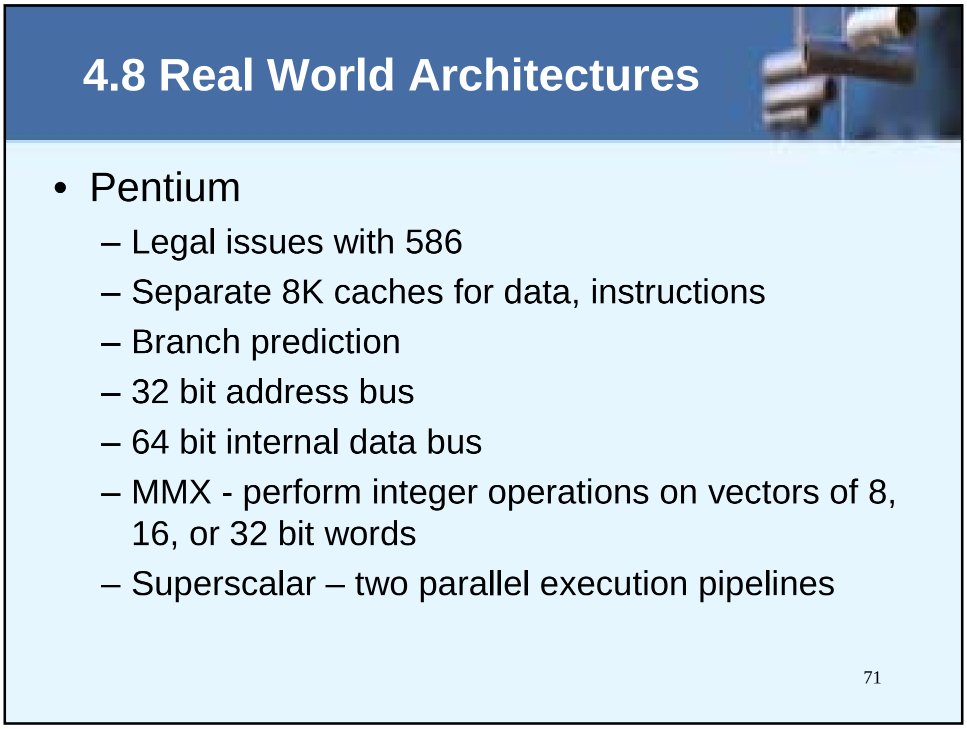
4.8 Real World Architectures

- 80286
  - Used in IBM AT
  - 24 bit address bus (16 Mb of RAM), 16 bit data bus
  - Protected mode – OS could protect programs in separate memory segments
- In 1985, Intel introduced the 32-bit 80386.
  - It also had no built-in floating-point unit.
  - 32 bit registers, 24 bit address bus
  - 80386 DX 32 bit data bus
  - 80386 SX 16 bit data bus
  - Supported virtual mode memory, paging



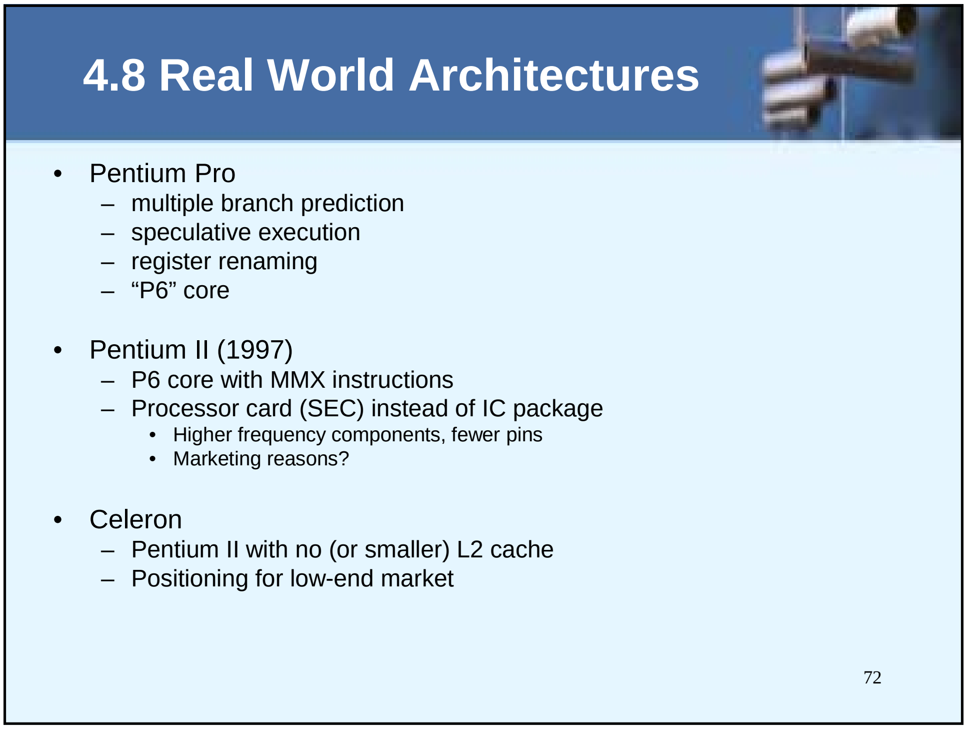
4.8 Real World Architectures

- 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.
  - Could decode/execute 5 instructions at once with pipelining
  - 8K level-1 cache for both instructions and data



4.8 Real World Architectures

- Pentium
  - – Legal issues with 586
  - – Separate 8K caches for data, instructions
  - – Branch prediction
  - – 32 bit address bus
  - – 64 bit internal data bus
  - – MMX - perform integer operations on vectors of 8, 16, or 32 bit words
  - – Superscalar – two parallel execution pipelines

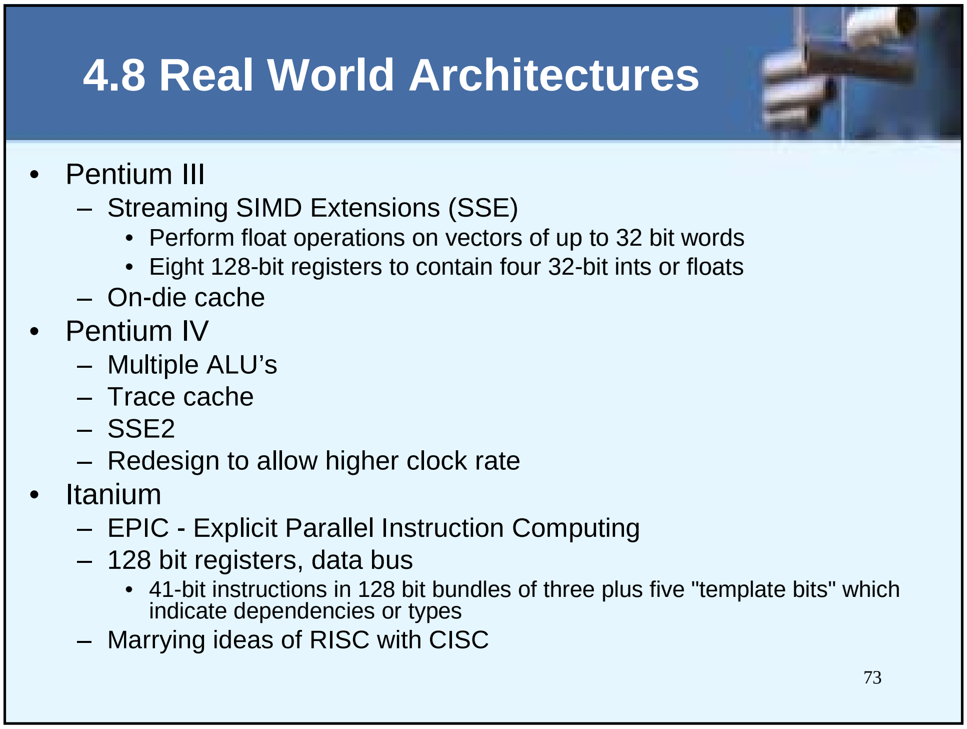


4.8 Real World Architectures

- Pentium Pro
  - – multiple branch prediction
  - – speculative execution
  - – register renaming
  - – “P6” core

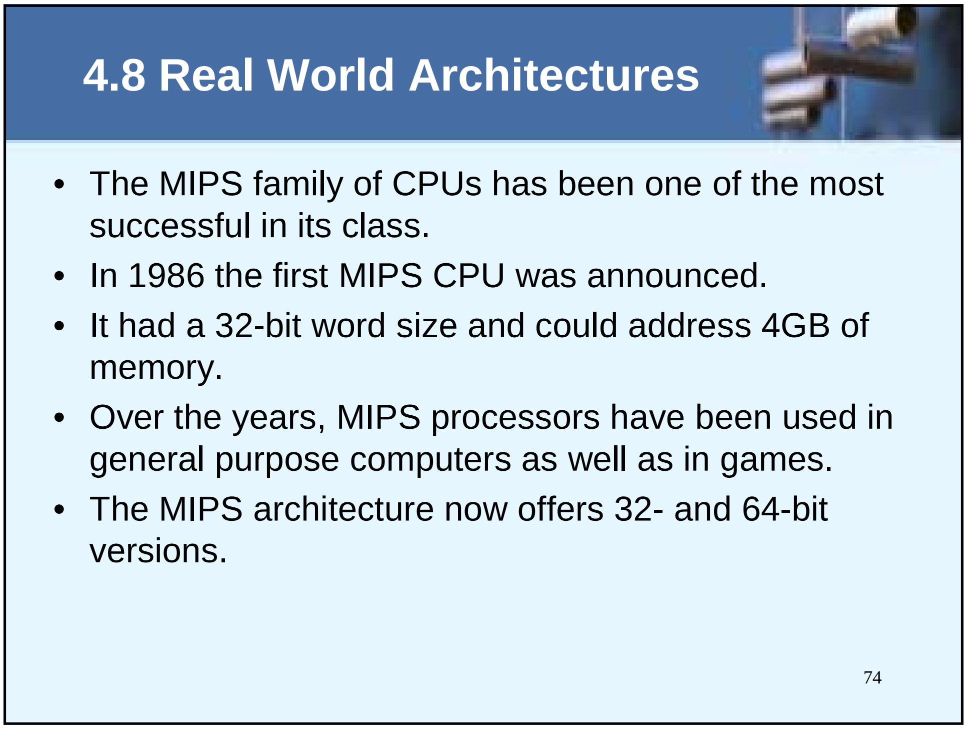
- Pentium II (1997)
  - – P6 core with MMX instructions
  - – Processor card (SEC) instead of IC package
    - Higher frequency components, fewer pins
    - Marketing reasons?

- Celeron
  - – Pentium II with no (or smaller) L2 cache
  - – Positioning for low-end market



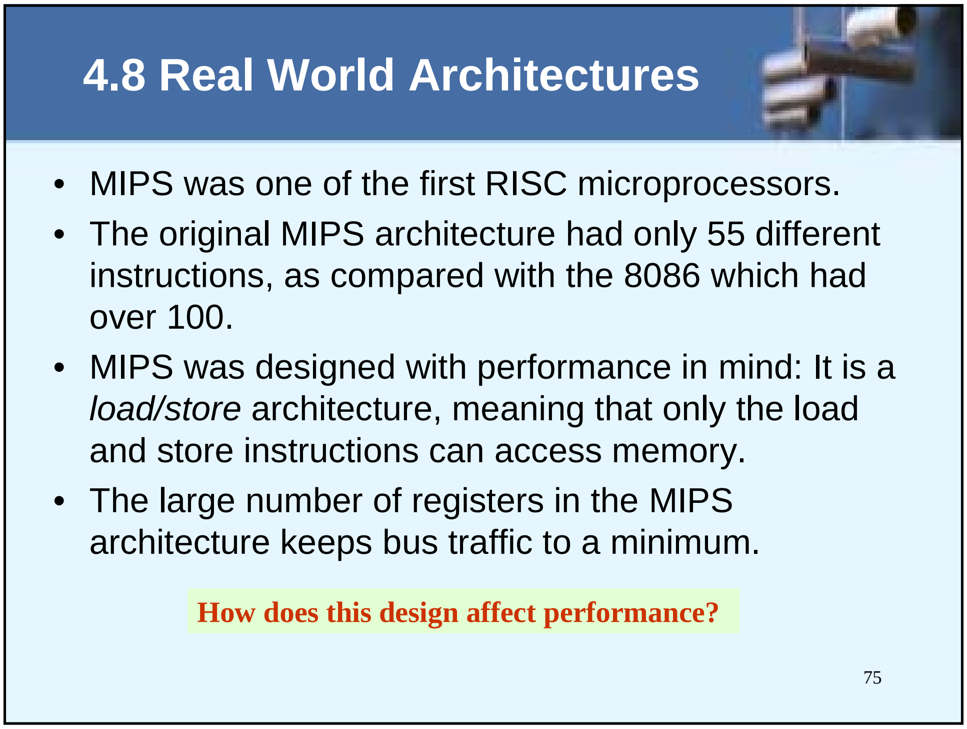
4.8 Real World Architectures

- Pentium III
  - Streaming SIMD Extensions (SSE)
    - Perform float operations on vectors of up to 32 bit words
    - Eight 128-bit registers to contain four 32-bit ints or floats
  - On-die cache
- Pentium IV
  - Multiple ALU’s
  - Trace cache
  - SSE2
  - Redesign to allow higher clock rate
- Itanium
  - EPIC - Explicit Parallel Instruction Computing
  - 128 bit registers, data bus
    - 41-bit instructions in 128 bit bundles of three plus five "template bits" which indicate dependencies or types
  - Marrying ideas of RISC with CISC



4.8 Real World Architectures

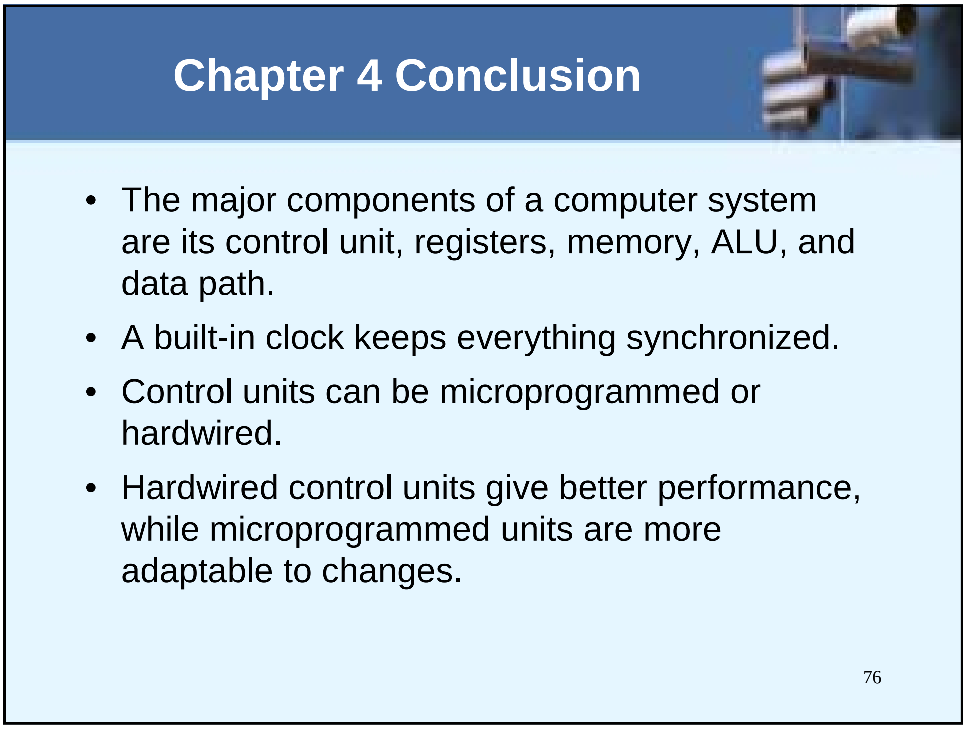
- The MIPS family of CPUs has been one of the most successful in its class.
- In 1986 the first MIPS CPU was announced.
- It had a 32-bit word size and could address 4GB of memory.
- Over the years, MIPS processors have been used in general purpose computers as well as in games.
- The MIPS architecture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



4.8 Real World Architectures

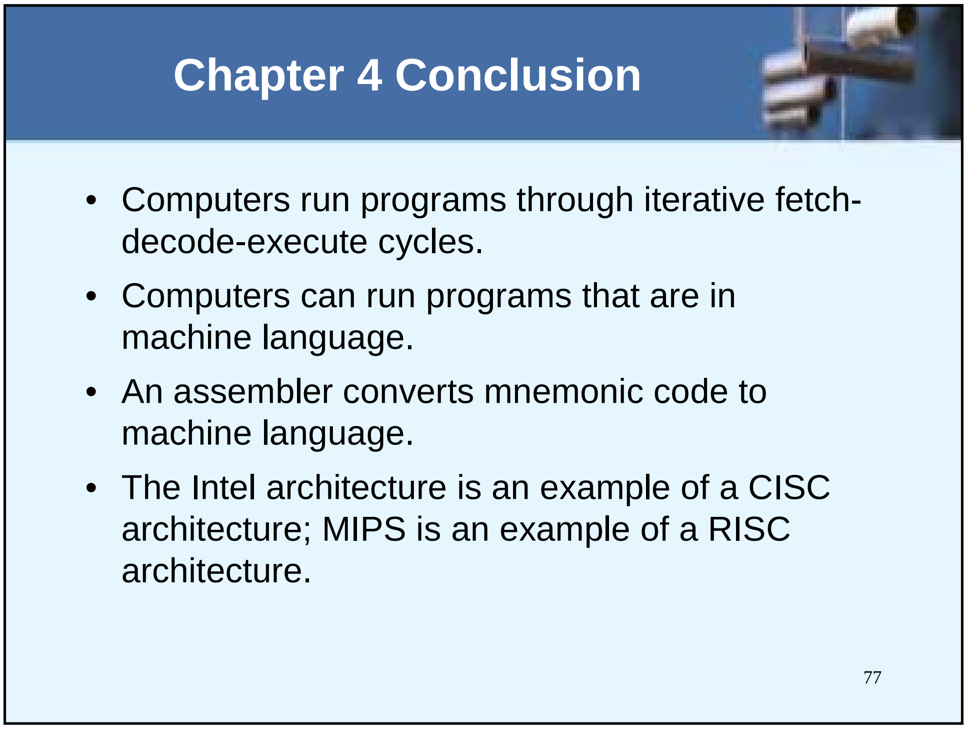
- MIPS was one of the first RISC microprocessors.
- The original MIPS architecture had only 55 different instructions, as compared with the 8086 which had over 100.
- MIPS was designed with performance in mind: It is a *load/store* architecture, meaning that only the load and store instructions can access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