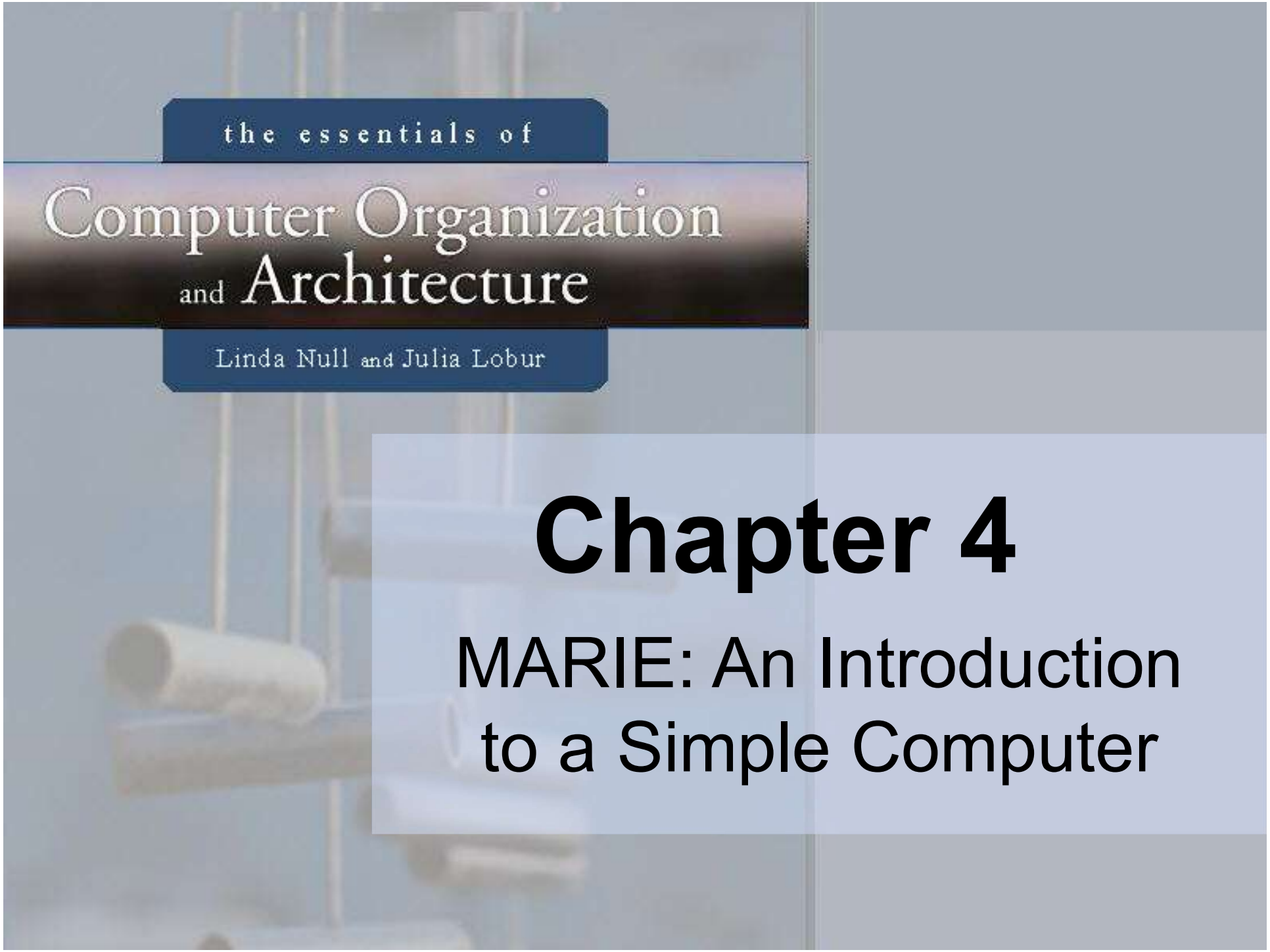


Chapter 4

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.

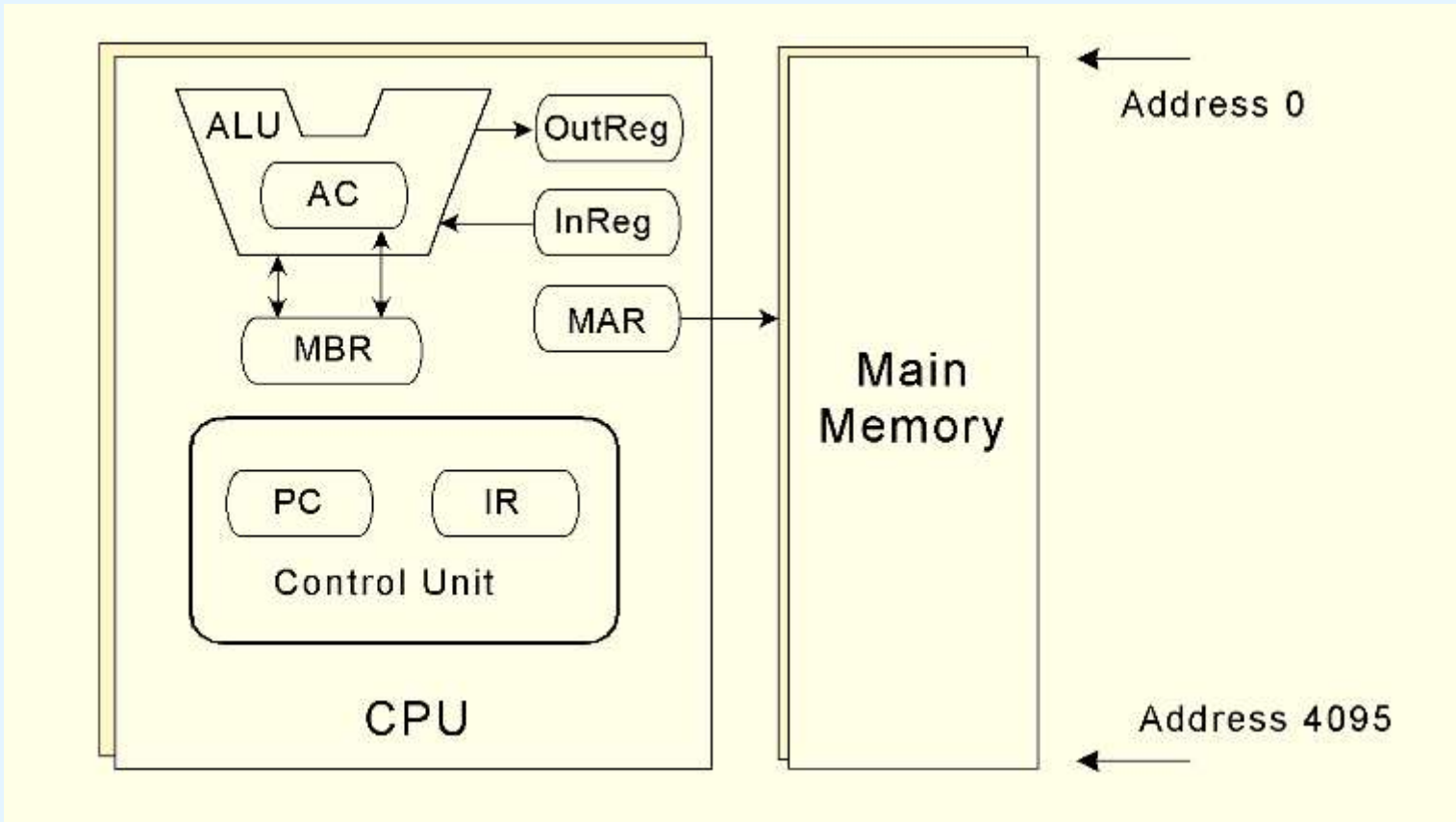
4.1 Introduction

- 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.
- 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.

4.1 Introduction



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.
  - Accessed very quickly
- Register's common size: 8,16,32,64
- Special purpose and General purpose register
  - Index register, stack register, status register
  - General register available to the programmer

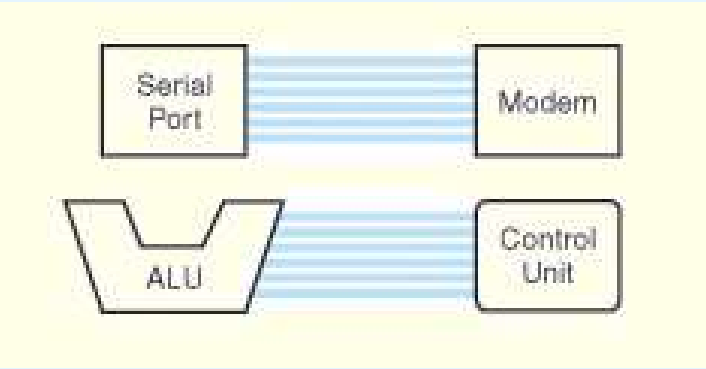
4.1 Introduction

- The arithmetic-logic unit (ALU) : carries out logical operations(comparison, bit and, or…) and arithmetic operations (add, multiply)
- Generally has two data inputs and one data output.
- ALU operations often affect status register(Z, C,V, N… flags)
- **The control unit** is the policeman/traffic manager of the CPU
- **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 : buses

- 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:

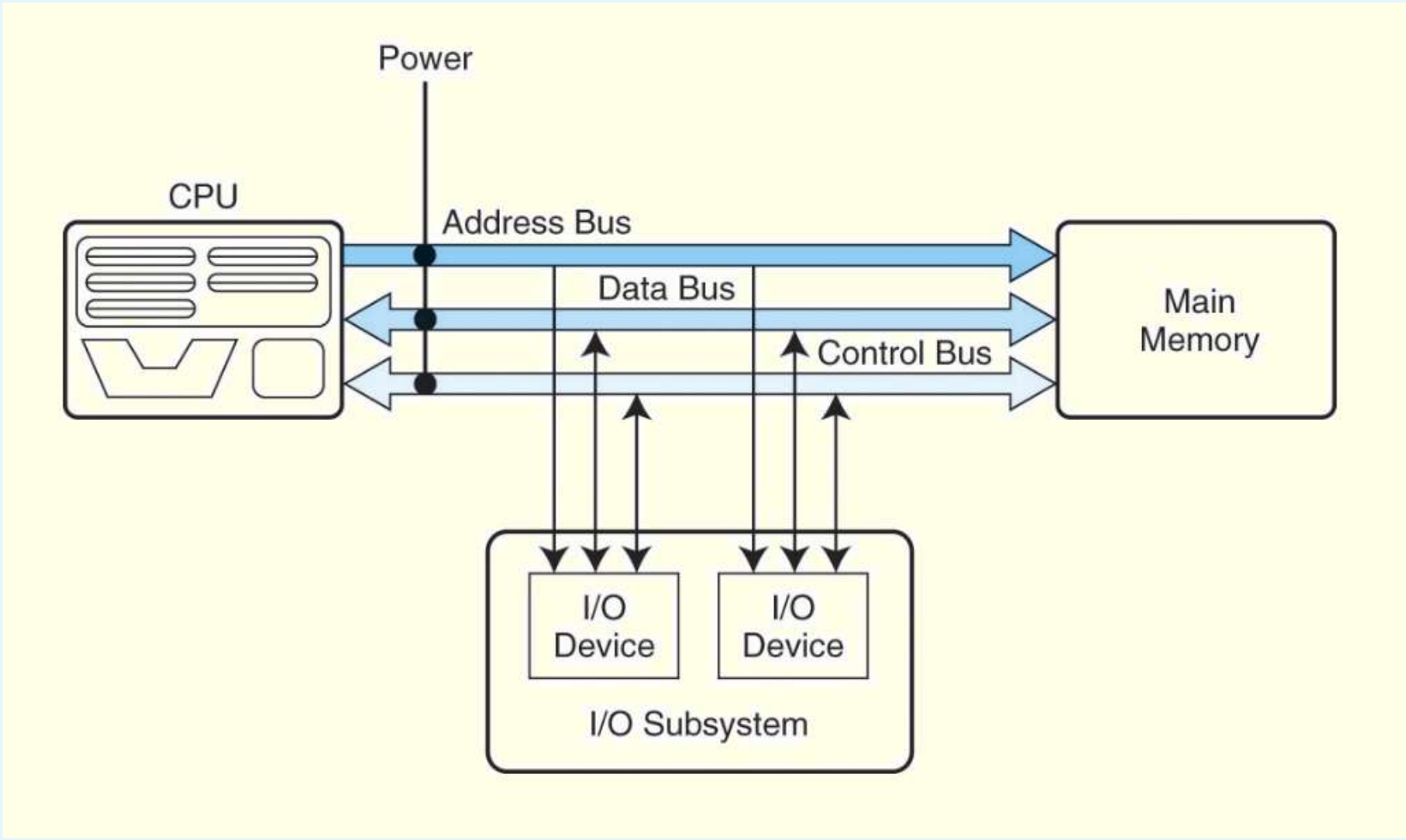


4.1 Introduction : buses

- Buses consist of data bus, control bus, and address bus.
- data bus: convey bits from one device to another,
- control bus: determine the direction of data flow(R/W), which device has permission to use the bus(CS), acknowledgments for bus requests, interrupt, …..
- Address bus determine the location of the source or destination of the data.

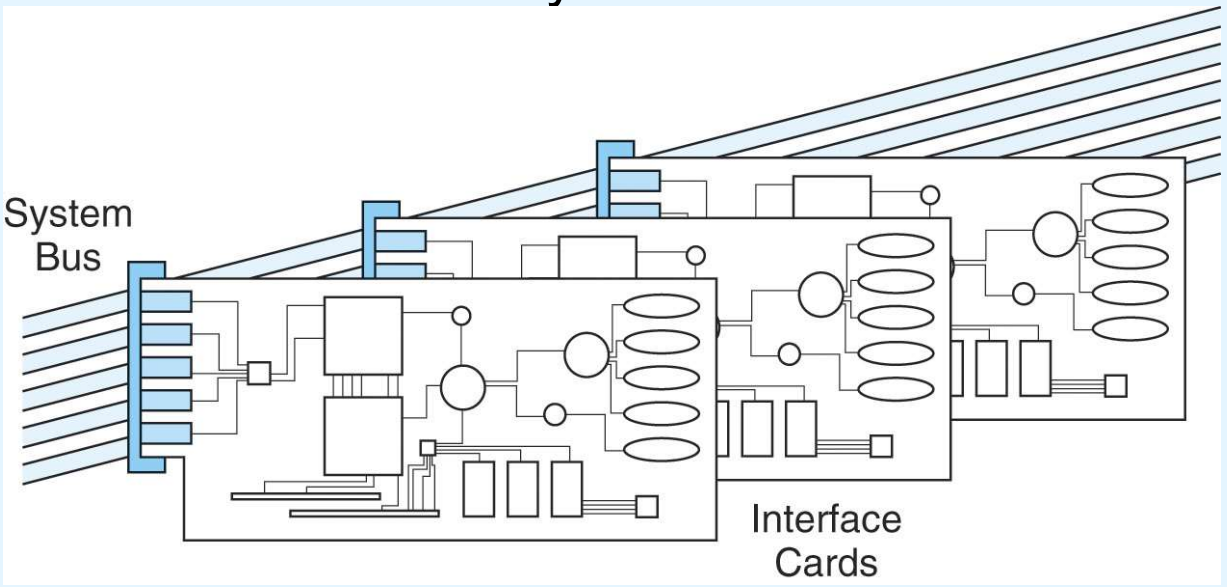
The next slide shows a model bus configuration.

4.1 Introduction : buses



4.1 Introduction: buses

- Buses have been divided into different types:
  - Processer-memory buses: short, high speed
  - I/O buses: longer, allowing for many types of devices with varying bandwidths
  - Backplane buses: in chassis, connects the processor, I/O devices and the memory



4.1 Introduction: buses

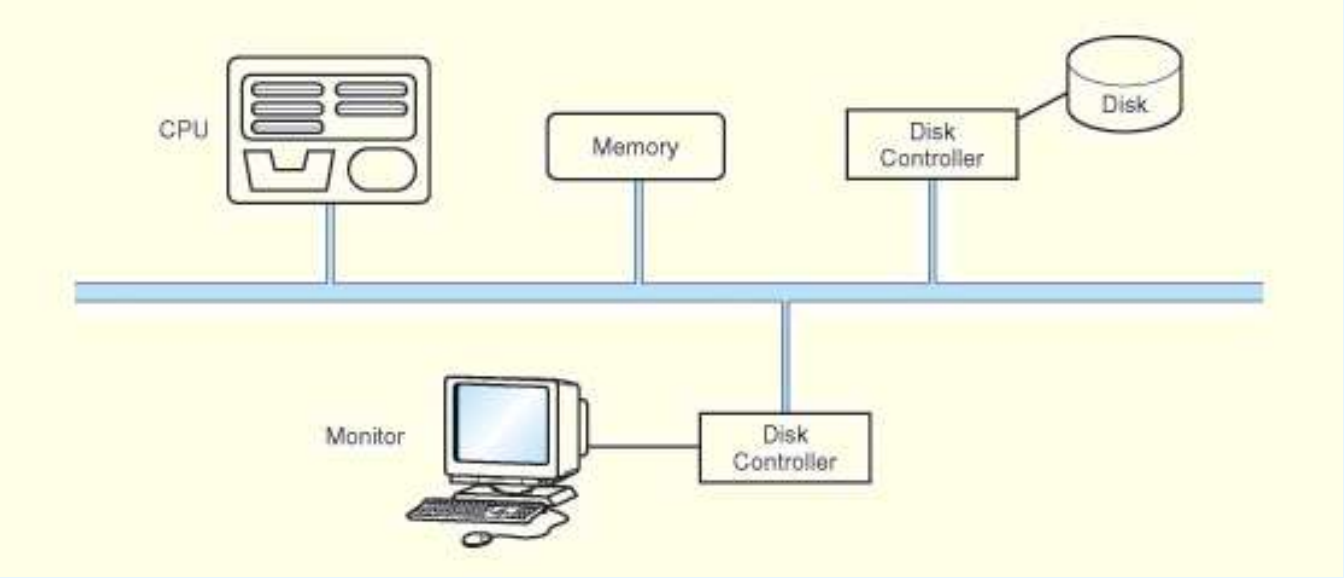
- hierarchy of buses: high performance systems often use all three types of buses.
- Internal bus v.s. External bus(expansion bus)
- Synchronous buses: clocked, thing happen only at the clock ticks
  - Clock rate
- Asynchronous buses: control lines coordinate the operations and a complex handshake protocol must be used.
  - CS, RD, OE,WR,….
  - ReqREAD, ReadyDATA, ACK

4.1 Introduction: buses

- **Synchronous vs. Asynchronous**
  - **Synchronous data transfer: sender and receiver use the same clock signal**
    - supports high data transfer rate
    - needs clock signal between the sender and the receiver
    - requires master/slave configuration
  - **Asynchronous data transfer: sender provides a synchronization signal to the receiver before starting the transfer of each message**
    - does not need clock signal between the sender and the receiver
    - slower data transfer rate

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.



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(菊花链):** Permissions are passed from the highest-priority device to the lowest. Simple but not fair. Low priority devices may be starved out
  - **Centralized parallel(集中并行):** Each device has a request line directly connected to an arbitration circuit.

4.1 Introduction

- **Distributed using self-detection(分布式自行检测):** similar to centralized arbitration, Devices decide which gets the bus among themselves.
- **Distributed using collision-detection(分布式冲突检测):** Any device can try to use the bus. If its data collides with the data of another device, it tries again. (Ethernet uses this type)

4.1 Introduction: 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.

4.1 Introduction: 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 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.

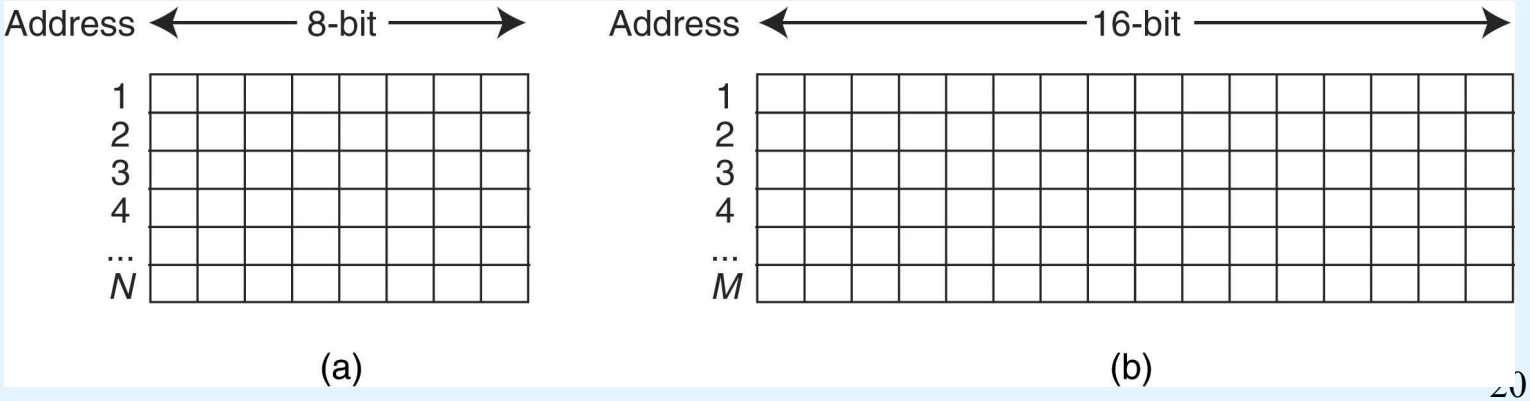
4.1 Introduction: I/O

- 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: memory

- 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.
- Normally, memory is byte-addressable. Each byte has a unique address



4.1 Introduction: memory

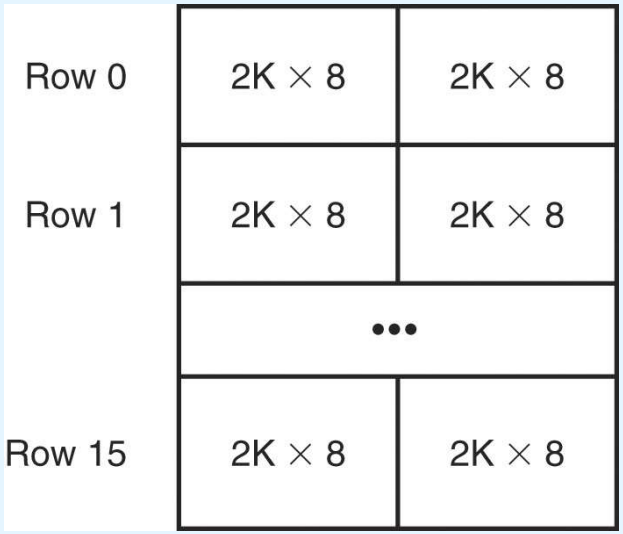
- The issue of alignment
  - 32-bit word on a byte-addressable machine
  - The word was stored on a natural alignment boundary
  - The access must start on that boundary
  - Some architectures allow unaligned accesses.
- Memory is constructed of RAM chips, often referred to in terms of length × width.
- If the memory word size of the machine is 16 bits, then a 4M × 16 RAM chip gives us 4 megawords of 16-bit memory locations.

4.1 Introduction: memory

- 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.

4.1 Introduction: memory

- Physical memory usually consists of more than one RAM chip.
  - Buid 32K*16 memory with 2K*8 RAM chips
  - 16 rows and 2 columns of chips
  - 32K has 15bits addresses
  - Each chip pair requires only 11bits
  - 4 bits to be decoded to select which row holds the desired address



4.1 Introduction: memory

- 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: memory

Module 0	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7
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 0	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7
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: interrupts

- Suppose that a 1M x 16 main memory is built using 128KBx8 RAM chips and memory is word-addressable.
  - a. How many RAM chips are necessary?
  - b. How many RAM chips are there per memory word?
  - c. How many address bits are needed for each RAM chip?
  - d. How many banks will this memory have?
  - e. How many address bits are needed for all of memory?
  - f. If high-order interleaving is used, where would address 14 (which is E in hex) be located?
  - g. Repeat Exercise f for low-order interleaving.

4.1 Introduction: interrupts

- Interrupts are events that alter the normal flow of execution in the system
- An interrupt can be triggered by
  - I/O requests
  - arithmetic errors (such as division by zero),
  - Arithmetic underflow or overflow
  - Hardware malfunction
  - User-defined break points
  - Page faults
  - Invalid instruction

4.1 Introduction: interrupts

- Each interrupt is associated with a procedure (interrupt handling, interrupt service routine) that directs the actions of the CPU when an interrupt occurs.
  - Interrupt priority
  - Maskable vs. Nonmaskable interrupts

4.2 MARIE

- 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：architecture

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:

- 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.

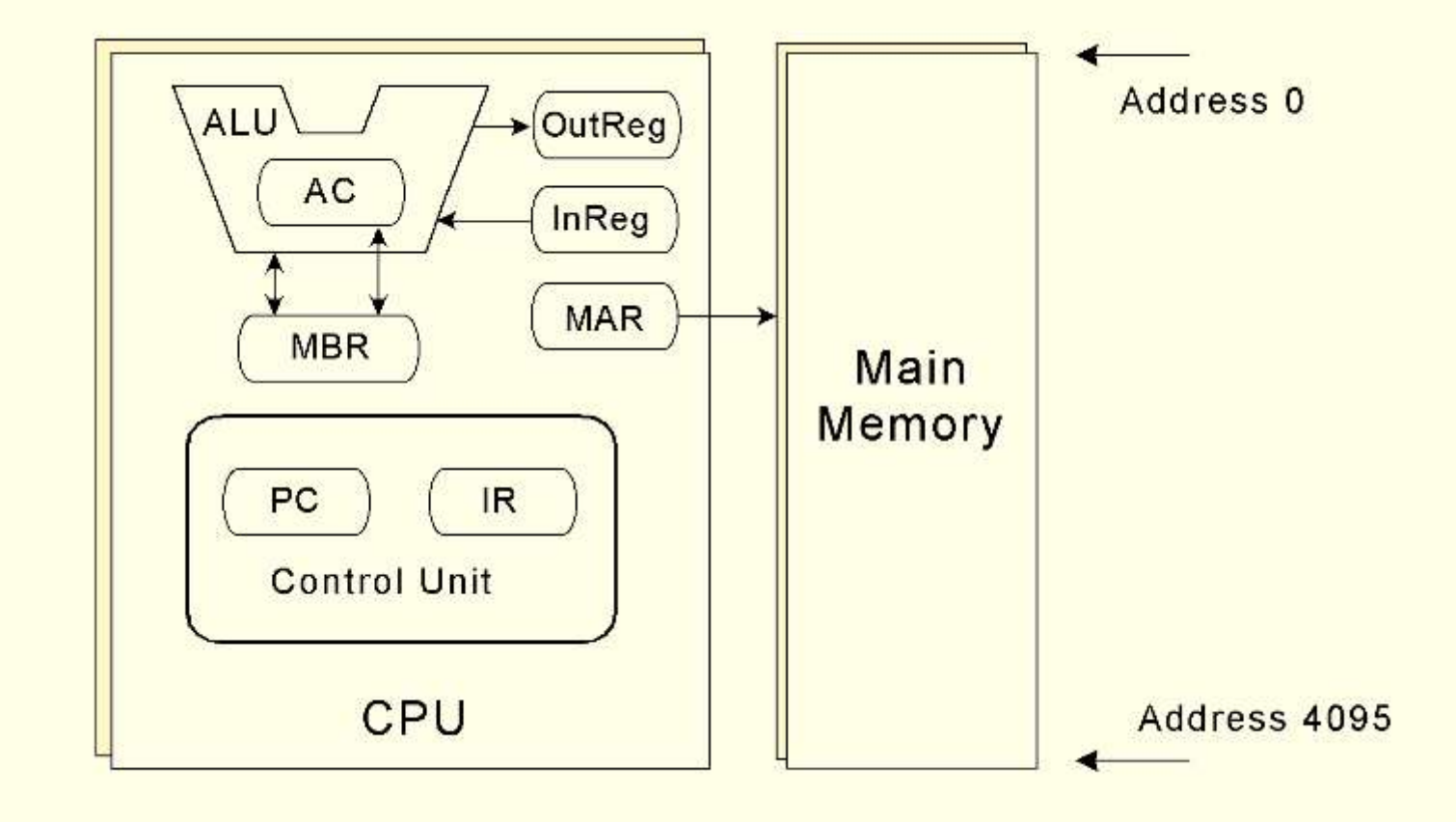
4.2 MARIE

MARIE's seven registers:

- 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.

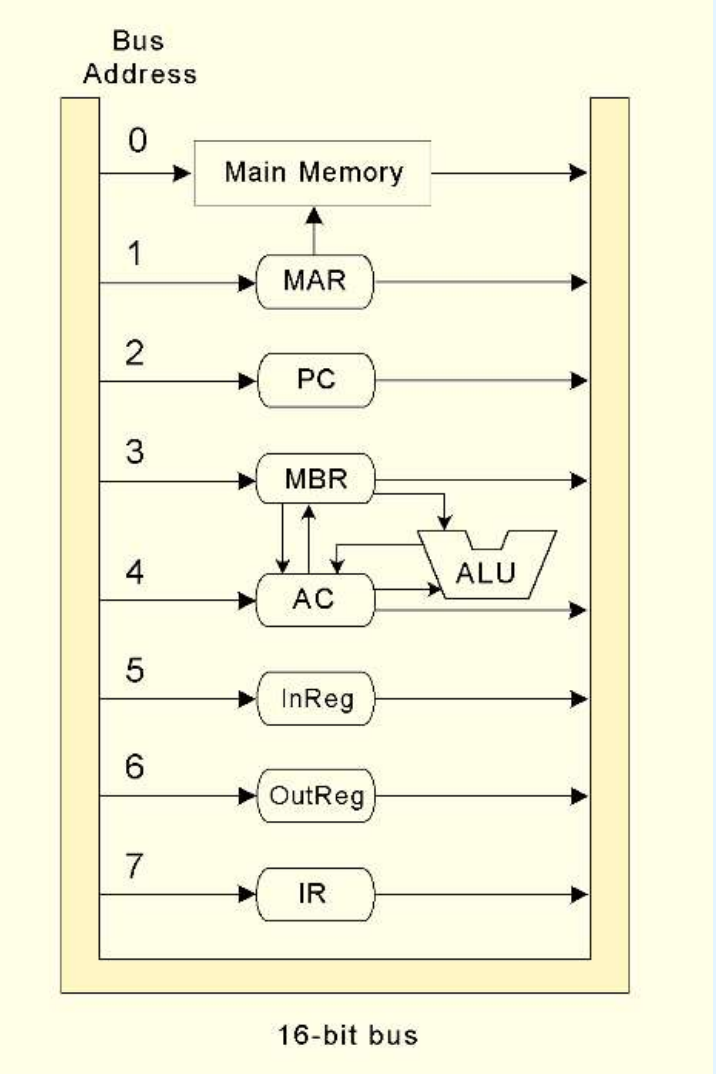


4.2 MARIE

- 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 ACC and the MBR, and the ALU and the ACC and MBR.
- 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.

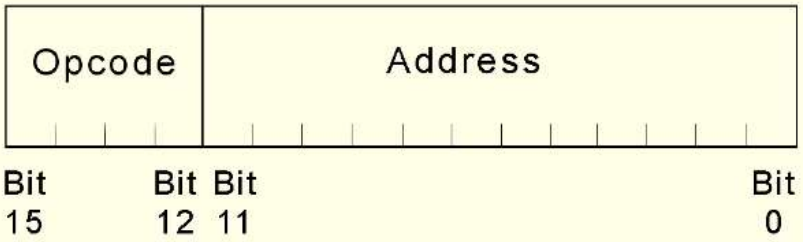


4.2 MARIE: ISA

- 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: ISA

- This is the format of a MARIE instruction:

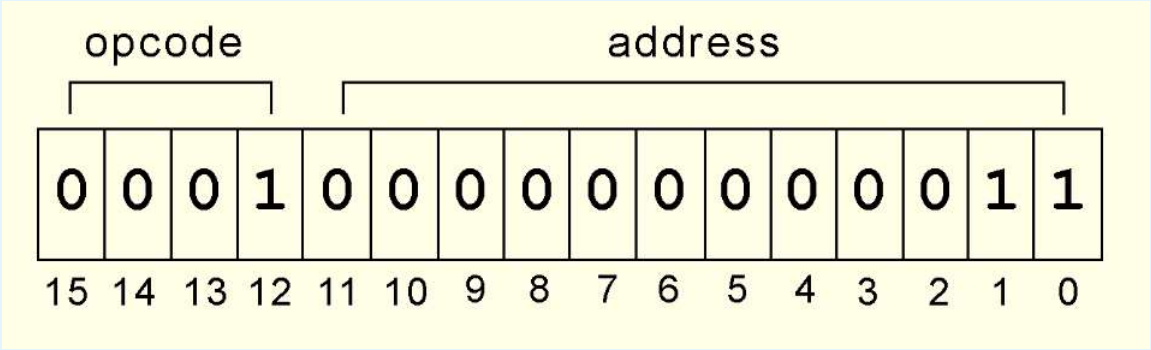


- The fundamental MARIE instructions are:

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.

4.2 MARIE

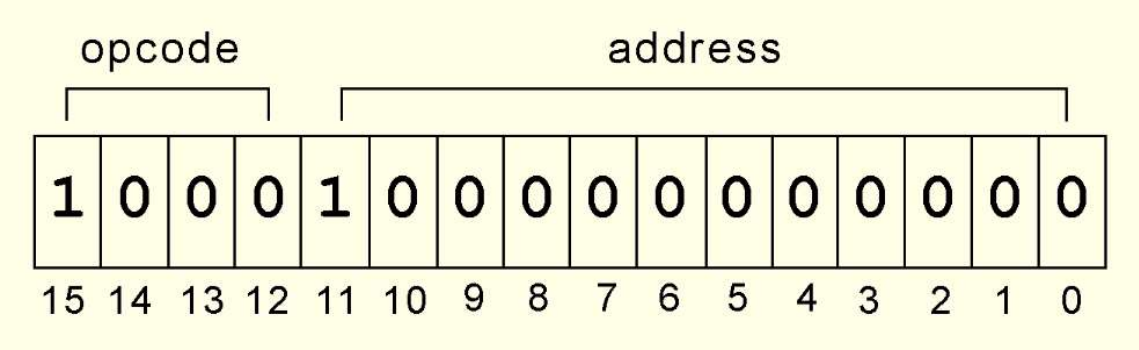
- This is a bit pattern for a **LOAD** instruction as it would appear in the IR:



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

4.2 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?

4.2 MARIE

- Each of our instructions actually 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/Notation (RTL/RTN).*
- In the MARIE RTL,
  - M[X]: indicate the actual data value stored in memory location X
  - ← : 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 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] = 11 then
      If AC > 0 then PC ← PC + 1
```

4.2 MARIE

- Store X
- Subt X
- Input
- Output
- Halt
- Jump X

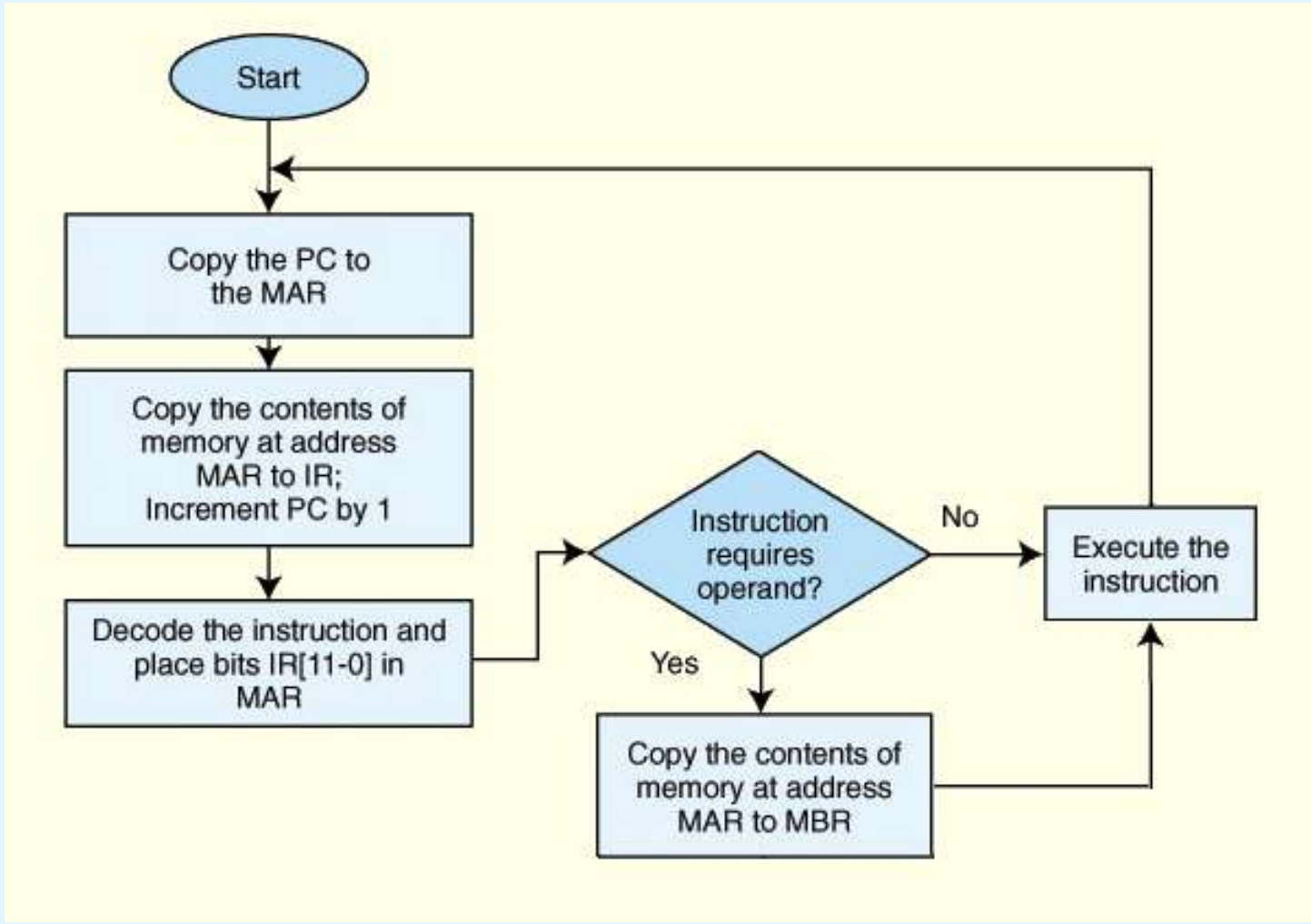
On Page 164

4.3 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*.

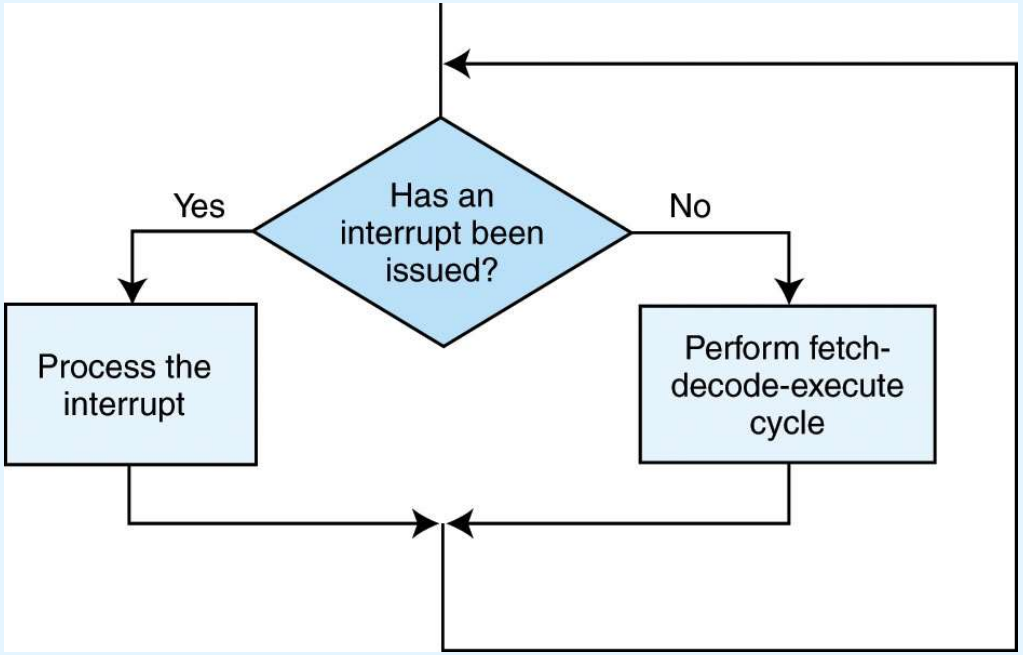
The next slide shows a flowchart of this process.

4.3 Instruction Processing



4.3 Instruction Processing

- Interrupts and I/O
  - I/O situations: data lost and data multiple times read
  - Interrupt driver I/O
  - Check to see if an interrupt is pending at the beginning of each fetch-decode-execute cycle.



4.3 Instruction Processing

- Interrupt flag
- Interrupt handling process
- How to determine the address of interrupt service routine
- CPU switch from running program to ISR and switch back.
- Context switch: save registers, returning address

4.3 Instruction Processing

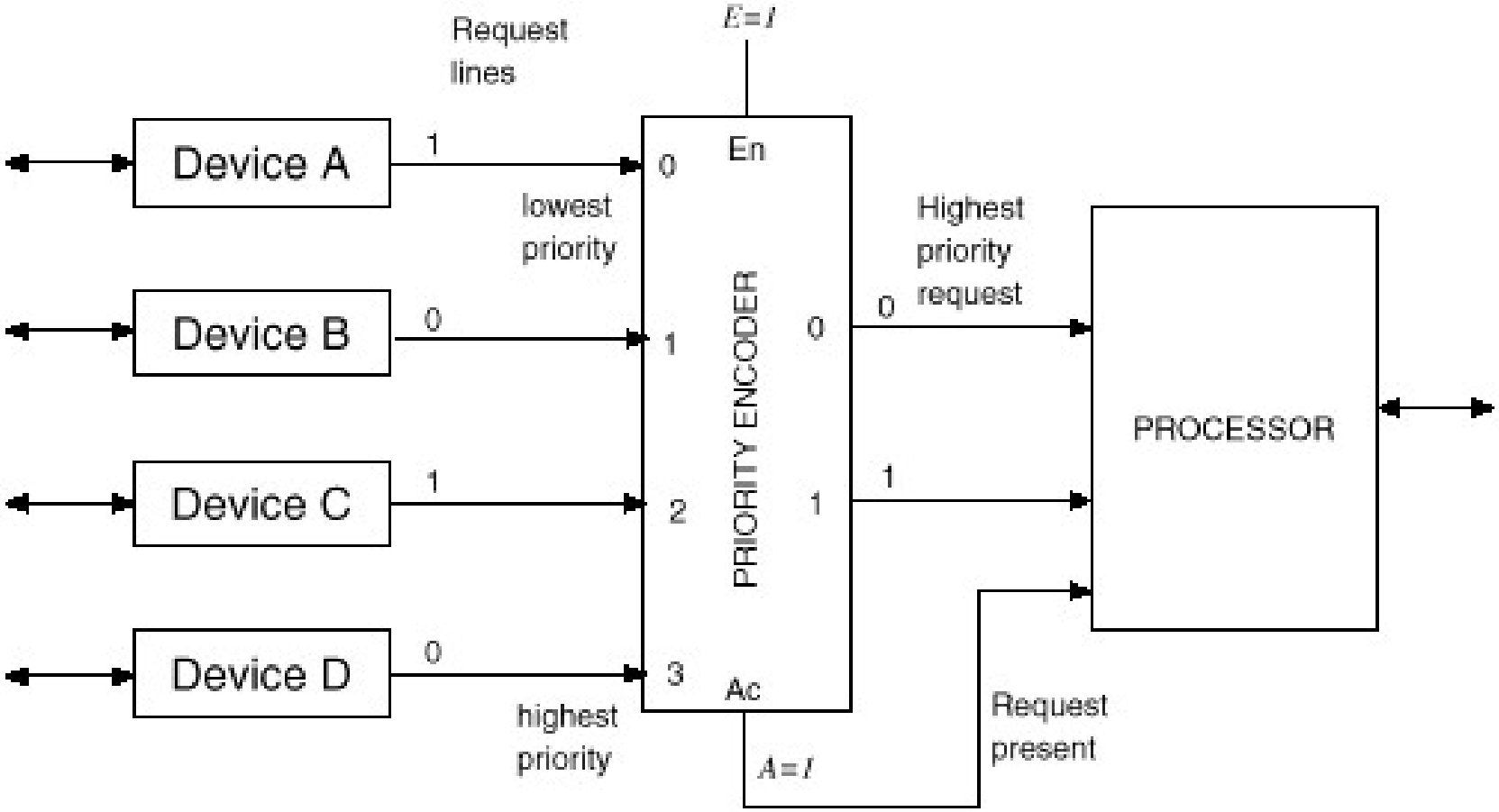


Figure 9.17: Resolving interrupt requests using a priority encoder.

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					
	IR ← M[MAR]					
	PC ← PC + 1					
Decode	MAR ← IR[11–0]					
	(Decode IR[15–12])					
Get operand	MBR ← M[MAR]					
Execute	AC ← MBR					

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					
	IR ← M[MAR]					
	PC ← PC + 1					
Decode	MAR ← IR[11–0]					
	(Decode IR[15–12])					
Get operand	MBR ← M[MAR]					
Execute	AC ← AC + MBR					

4.4 A Simple Program

- The third instruction is **`Store 106`**:

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		102	3105	105	FFE9	000C
Fetch	MAR ← PC					
	IR ← M[MAR]					
	PC ← PC + 1					
Decode	MAR ← IR[11–0]					
	(Decode IR[15–12])					
Get operand	(not necessary)					
Execute	MBR ← AC					
	M[MAR] ← MBR					

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
  - 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.

4.5 A Discussion on Assemblers

- Labels are nice for programmer, Assemblers need to do the translation
- 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.

4.5 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
7 0 0 0

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

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

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 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.

4.6 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:

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

4.6 Extending Our Instruction Set

- JumpI X: jump indirect, goto address X, use the value at X as the actual address of the location to jump to.
  - MAR ← X
  - MBR ← M[MAR]
  - PC ← 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** instruction are given by the following RTL:

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

Does JNS permit recursive calls?

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**:

$$\mathbf{AC} \leftarrow \mathbf{0}$$

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

4.6 Extending Our Instruction Set

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

4.6 Extending Our Instruction Set

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

4.6 Coding Example1

```
   Address   Instruction      Comments
If,    100   Load      X      /Load the first value
       101   Subt      Y      /Subtract value of Y and store result in AC
       102   Skipcond  01     /If AC = 0, skip the next instruction
       103   Jump      Else   /Jump to the Else part if AC is not equal to 0
Then,  104   Load      X      /Reload X so it can be doubled
       105   Add       X      /Double X
       106   Store     X      /Store the new value
       107   Jump      Endif  /Skip over Else part to end of If
Else,  108   Load      Y      /Start the Else part by loading Y
       109   Subt      X      /Subtract X from Y
       10A   Store     Y      /Store Y - X in Y
Endif, 10B   Halt             /Terminate program (it doesn't do much!)
X,     10C   Dec       12     /Load the loop control variable
Y,     10D   Dec       20     /Subtract one from the loop control variable
```

4.6 Coding Example2

```
        100  Load   X     /Load the first number to be doubled
        101  Store  Temp  /Use Temp as a parameter to pass value to Subr
        102  JnS    Subr  /Store return address, jump to procedure
        103  Store  X     /Store first number, doubled
        104  Load   Y     /Load the second number to be doubled
        105  Store  Temp  /Use Temp as a parameter to pass value to Subr
        106  JnS    Subr  /Store return address, jump to procedure
        107  Store  Y     /Store second number, doubled
        108  Halt         /End program
X,      109  Dec    20
Y,      10A  Dec    48
Temp,   10B  Dec    0
Subr,   10C  Hex    0     /Store return address here
        10D  Clear        /Clear AC as it was modified by JnS
        10E  Load   Temp  /Actual subroutine to double numbers
        10F  Add    Temp  /AC now holds double the value of Temp
        110  JumpI  Subr  /Return to calling code
             END
```

4.6 Coding Example3

```
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
```

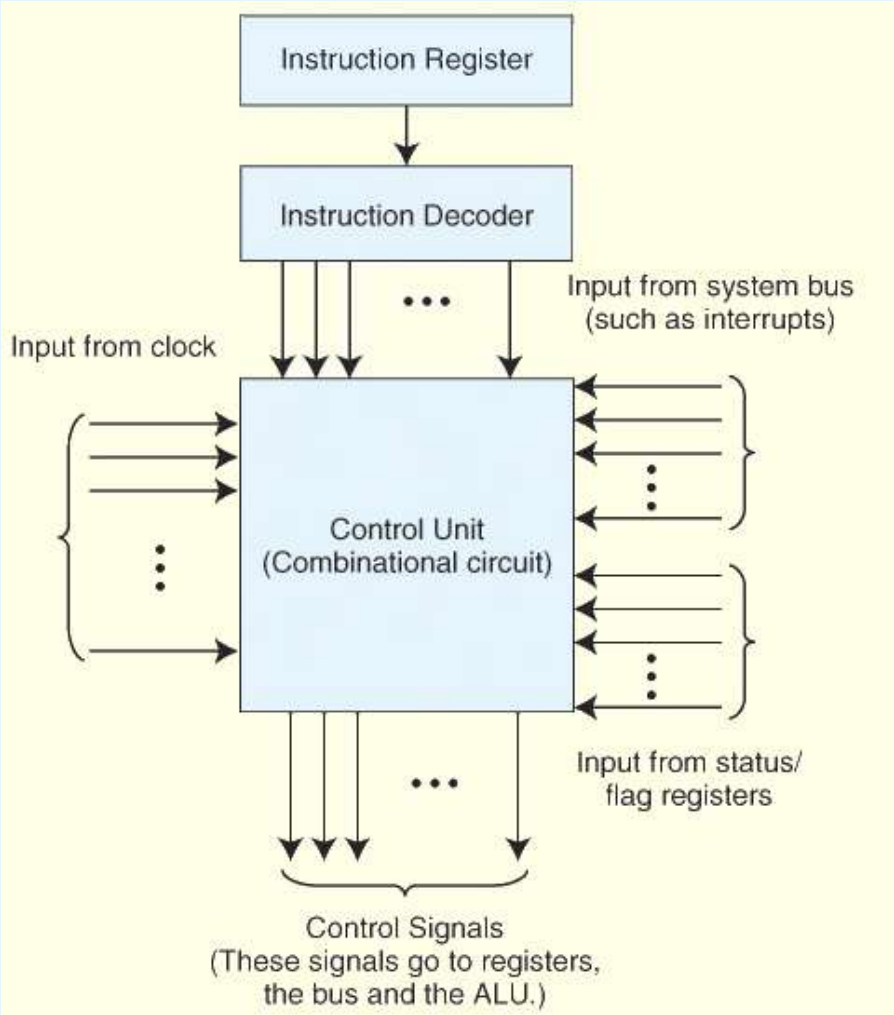
```
10E |       STORE Ctr
10F |       SKIPCOND 000
110 |       JUMP Loop
111 |       HALT
112 |Addr   HEX 118
113 |Next   HEX 0
114 |Num    DEC 5
115 |Sum    DEC 0
116 |Ctr    HEX 0
117 |One    DEC 1
118 |       DEC 10
119 |       DEC 15
11A |       DEC 2
11B |       DEC 25
11C |       DEC 30
```

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.

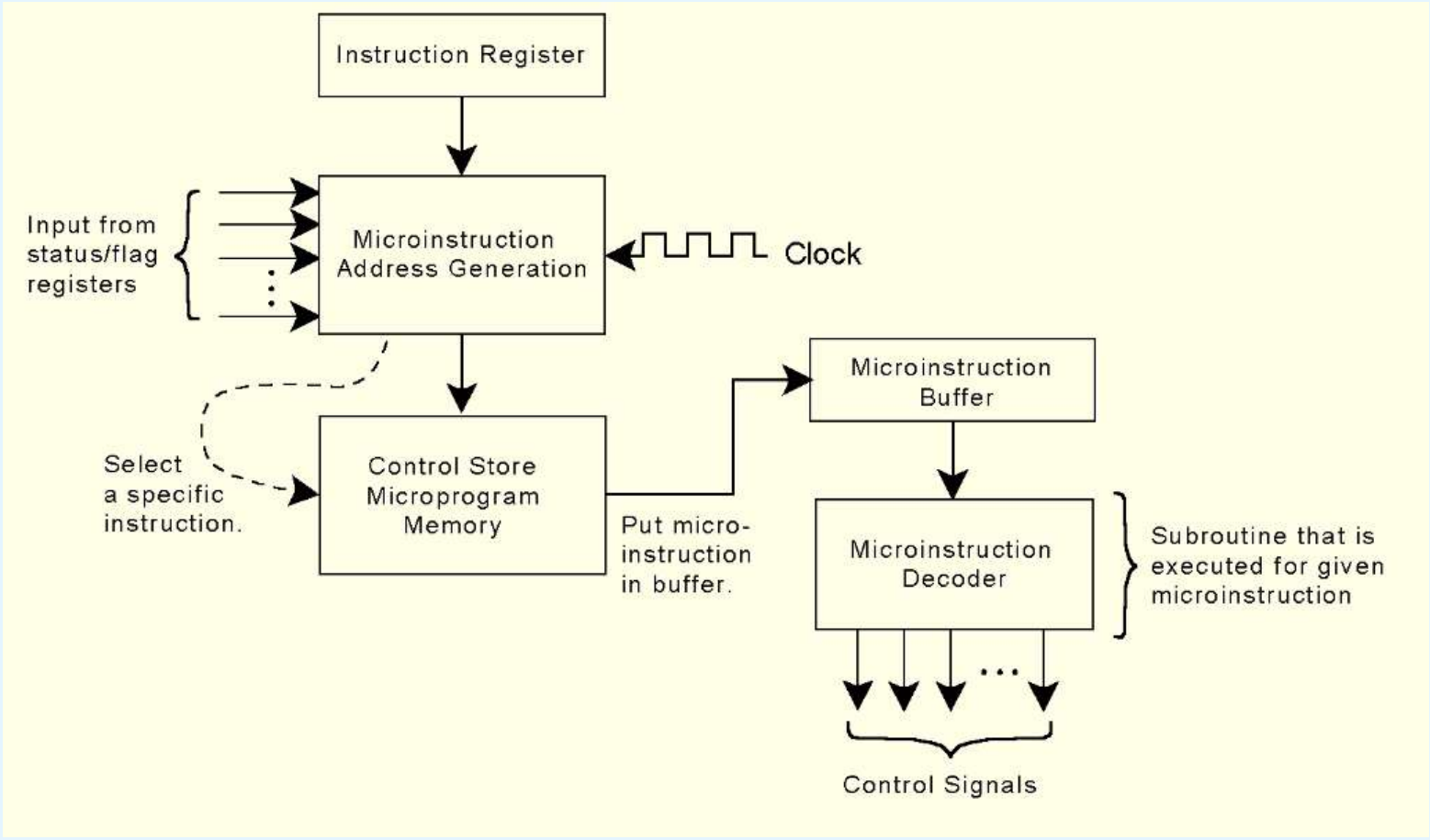
4.7 A Discussion on Decoding

- 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.



4.7 A Discussion on Decoding

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



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.

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.
- 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?

4.8 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

- 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.

4.8 Real World Architectures

- Currently, Intel's most advanced 32-bit microprocessor is the Pentium 4.
- It can run as fast as 3.06 GHz. This clock rate is over 350 times faster than that of the 8086.
- Speed enhancing features include 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.

4.8 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.

Chapter 4 Homework

- P193: 4, P194: 6, 8, P195: 11
- P195: 13, 14
- P196: 15.b, 15.c, 17, 18, 19,20