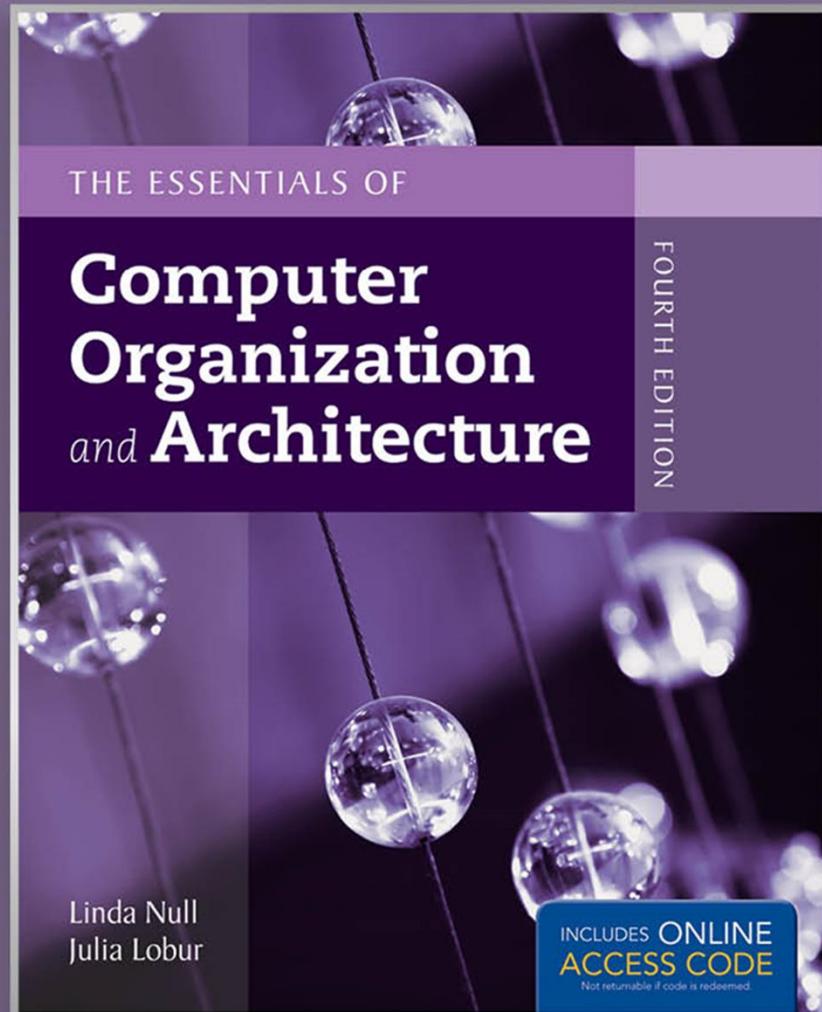


Chapter 3

Boolean Algebra and Digital Logic



Chapter 3 Objectives

- Understand the relationship between Boolean logic and digital computer circuits.
- Learn how to design simple logic circuits.
- Understand how digital circuits work together to form complex computer systems.

3.1 Introduction

- In the latter part of the nineteenth century, George Boole incensed philosophers and mathematicians alike when he suggested that logical thought could be represented through mathematical equations.
 - *How dare anyone suggest that human thought could be encapsulated and manipulated like an algebraic formula?*
- Computers, as we know them today, are implementations of Boole's *Laws of Thought*.
 - John Atanasoff and Claude Shannon were among the first to see this connection.

3.1 Introduction

- In the middle of the twentieth century, computers were commonly known as “thinking machines” and “electronic brains.”
 - Many people were fearful of them.
- Nowadays, we rarely ponder the relationship between electronic digital computers and human logic. Computers are accepted as part of our lives.
 - Many people, however, are still fearful of them.
- In this chapter, you will learn the simplicity that constitutes the essence of the machine.

3.2 Boolean Algebra

- Boolean algebra is a mathematical system for the manipulation of variables that can have one of two values.
 - In formal logic, these values are “true” and “false.”
 - In digital systems, these values are “on” and “off,” 1 and 0, or “high” and “low.”
- Boolean expressions are created by performing operations on Boolean variables.
 - Common Boolean operators include AND, OR, and NOT.

3.2 Boolean Algebra

- A Boolean operator can be completely described using a truth table.
- The truth table for the Boolean operators AND and OR are shown at the right.
- The AND operator is also known as a Boolean product. The OR operator is the Boolean sum.

X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

3.2 Boolean Algebra

- The truth table for the Boolean NOT operator is shown at the right.
- The NOT operation is most often designated by a prime mark (x'). It is sometimes indicated by an overbar (\bar{x}) or an “elbow” ($\neg x$).

NOT x	
x	x'
0	1
1	0

3.2 Boolean Algebra

- A Boolean function has:
 - At least one Boolean variable,
 - At least one Boolean operator, and
 - At least one input from the set {0,1}.
- It produces an output that is also a member of the set {0,1}.

Now you know why the binary numbering system is so handy in digital systems.

3.2 Boolean Algebra

- The truth table for the Boolean function:

$$F(x, y, z) = xz' + y$$

is shown at the right.

- To make evaluation of the Boolean function easier, the truth table contains extra (shaded) columns to hold evaluations of subparts of the function.

$$F(x, y, z) = xz' + y$$

x	y	z	z'	xz'	$xz' + y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

3.2 Boolean Algebra

- As with common arithmetic, Boolean operations have rules of precedence.
- The NOT operator has highest priority, followed by AND and then OR.
- This is how we chose the (shaded) function subparts in our table.

$$F(x, y, z) = xz' + y$$

x	y	z	z'	xz'	$xz' + y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

3.2 Boolean Algebra

- Digital computers contain circuits that implement Boolean functions.
- The simpler that we can make a Boolean function, the smaller the circuit that will result.
 - Simpler circuits are cheaper to build, consume less power, and run faster than complex circuits.
- With this in mind, we always want to reduce our Boolean functions to their simplest form.
- There are a number of Boolean identities that help us to do this.

3.2 Boolean Algebra

- Most Boolean identities have an AND (product) form as well as an OR (sum) form. We give our identities using both forms. Our first group is rather intuitive:

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$xx' = 0$	$x + x' = 1$

3.2 Boolean Algebra

- Our second group of Boolean identities should be familiar to you from your study of algebra:

Identity Name	AND Form	OR Form
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x + (y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$

3.2 Boolean Algebra

- Our last group of Boolean identities are perhaps the most useful.
- If you have studied set theory or formal logic, these laws are also familiar to you.

Identity Name	AND Form	OR Form
Absorption Law	$x(x+y) = x$	$x + xy = x$
DeMorgan's Law	$(xy)' = x' + y'$	$(x+y)' = x'y'$
Double Complement Law		$(x)'' = x$

3.2 Boolean Algebra

- We can use Boolean identities to simplify:

$$F(x, y, z) = xy + x'z + yz$$

$$\begin{aligned} F(x, y, z) &= xy + x'z + yz \\ &= xy + x'z + yz(1) && \text{(Identity)} \\ &= xy + x'z + yz(x + x') && \text{(Inverse)} \\ &= xy + x'z + (yz)x + (yz)x' && \text{(Distributive)} \\ &= xy + x'z + x(yz) + x'(zy) && \text{(Commutative)} \\ &= xy + x'z + (xy)z + (x'z)y && \text{(Associative twice)} \\ &= xy + (xy)z + x'z + (x'z)y && \text{(Commutative)} \\ &= xy(1 + z) + x'z(1 + y) && \text{(Distributive)} \\ &= xy(1) + x'z(1) && \text{(Null)} \\ &= xy + x'z && \text{(Identity)} \end{aligned}$$

3.2 Boolean Algebra

- Sometimes it is more economical to build a circuit using the complement of a function (and complementing its result) than it is to implement the function directly.
- DeMorgan's law provides an easy way of finding the complement of a Boolean function.
- Recall DeMorgan's law states:

$$(xy)' = x' + y' \quad \text{and} \quad (x + y)' = x'y'$$

3.2 Boolean Algebra

- DeMorgan's law can be extended to any number of variables.
- Replace each variable by its complement and change all ANDs to ORs and all ORs to ANDs.
- Thus, we find the the complement of:

$$F(x, y, z) = (xy) + (x'y) + (xz')$$

is:

$$\begin{aligned} F' (x, y, z) &= ((xy) + (x'y) + (xz'))' \\ &= (xy)' (x'y)' (xz')' \\ &= (x' + y') (x + y') (x' + z) \end{aligned}$$

3.2 Boolean Algebra

- Through our exercises in simplifying Boolean expressions, we see that there are numerous ways of stating the same Boolean expression.
 - These “synonymous” forms are *logically equivalent*.
 - Logically equivalent expressions have identical truth tables.
- In order to eliminate as much confusion as possible, designers express Boolean functions in *standardized* or *canonical* form.

3.2 Boolean Algebra

- There are two canonical forms for Boolean expressions: sum-of-products and product-of-sums.
 - Recall the Boolean product is the AND operation and the Boolean sum is the OR operation.
- In the sum-of-products form, ANDed variables are ORed together.
 - For example: $F(x, y, z) = xy + xz + yz$
- In the product-of-sums form, ORed variables are ANDed together:
 - For example: $F(x, y, z) = (x+y)(x+z)(y+z)$

3.2 Boolean Algebra

- It is easy to convert a function to sum-of-products form using its truth table.
- We are interested in the values of the variables that make the function true (=1).
- Using the truth table, we list the values of the variables that result in a true function value.
- Each group of variables is then ORed together.

$$F(x, y, z) = xz' + y$$

x	y	z	$xz' + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

3.2 Boolean Algebra

- The sum-of-products form for our function is:

$$\begin{aligned}F(x, y, z) = & (x'y'z') + (x'yz) \\& + (xy'z') + (xyz') \\& + (xyz)\end{aligned}$$

We note that this function is not in simplest terms. Our aim is only to rewrite our function in canonical sum-of-products form.

$$F(x, y, z) = xz' + y$$

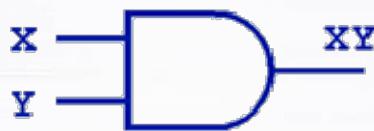
x	y	z	$xz' + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

3.3 Logic Gates

- We have looked at Boolean functions in abstract terms.
- In this section, we see that Boolean functions are implemented in digital computer circuits called gates.
- A gate is an electronic device that produces a result based on two or more input values.
 - In reality, gates consist of one to six transistors, but digital designers think of them as a single unit.
 - Integrated circuits contain collections of gates suited to a particular purpose.

3.3 Logic Gates

- The three simplest gates are the AND, OR, and NOT gates.



X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT X

X	X'
0	1
1	0

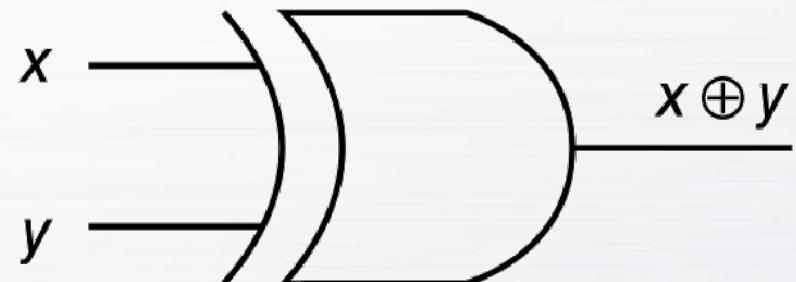
- They correspond directly to their respective Boolean operations, as you can see by their truth tables.

3.3 Logic Gates

- Another very useful gate is the exclusive OR (XOR) gate.
- The output of the XOR operation is true only when the values of the inputs differ.

$x \text{ XOR } y$

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0



Note the special symbol \oplus for the XOR operation.

3.3 Logic Gates

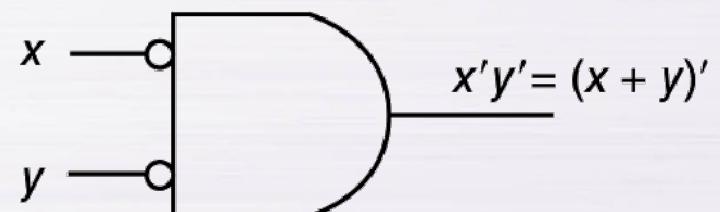
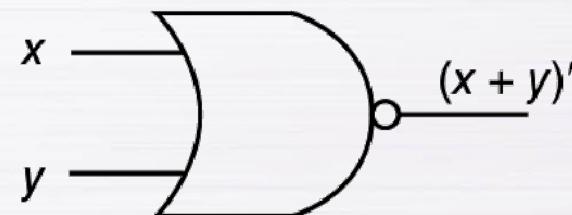
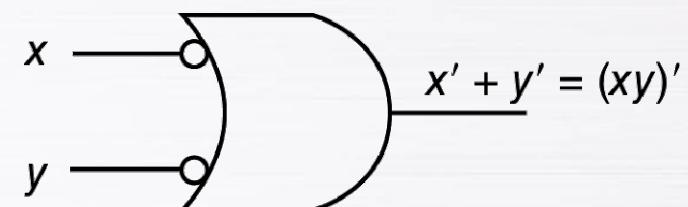
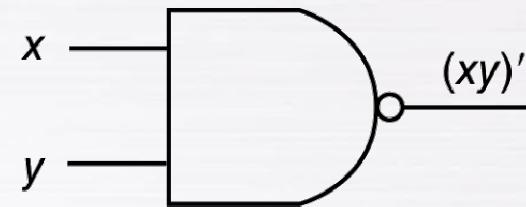
- NAND and NOR are two very important gates. Their symbols and truth tables are shown at the right.

X NAND Y

X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

X NOR Y

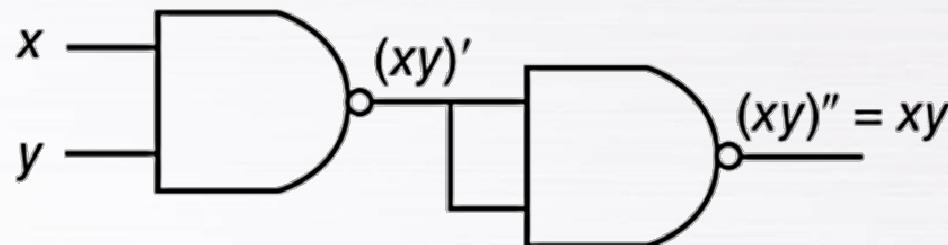
X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0



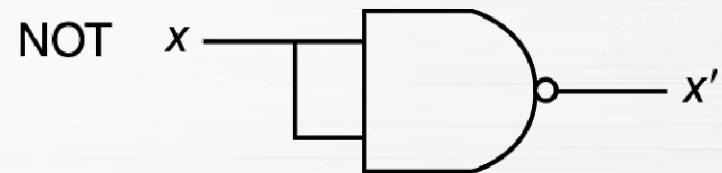
3.3 Logic Gates

- NAND and NOR are known as *universal gates* because they are inexpensive to manufacture and any Boolean function can be constructed using only NAND or only NOR gates.

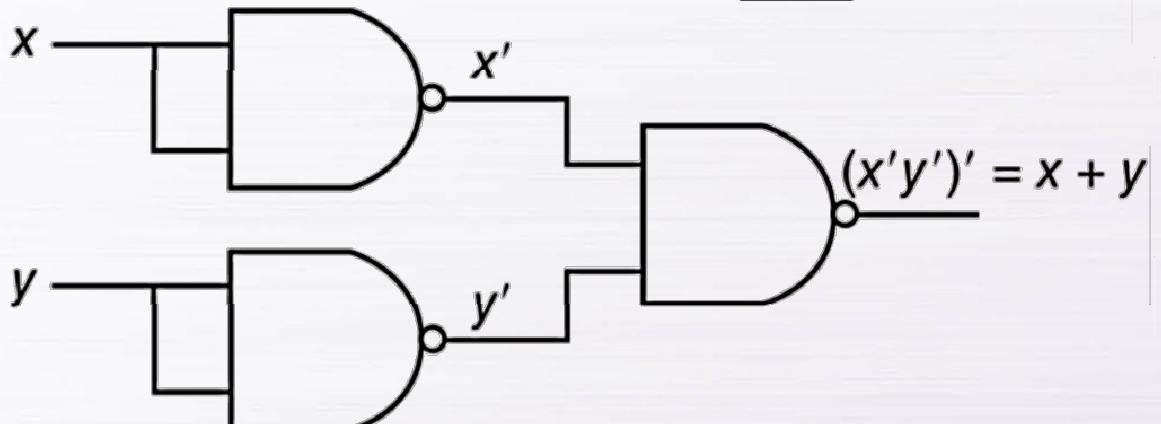
AND



NOT

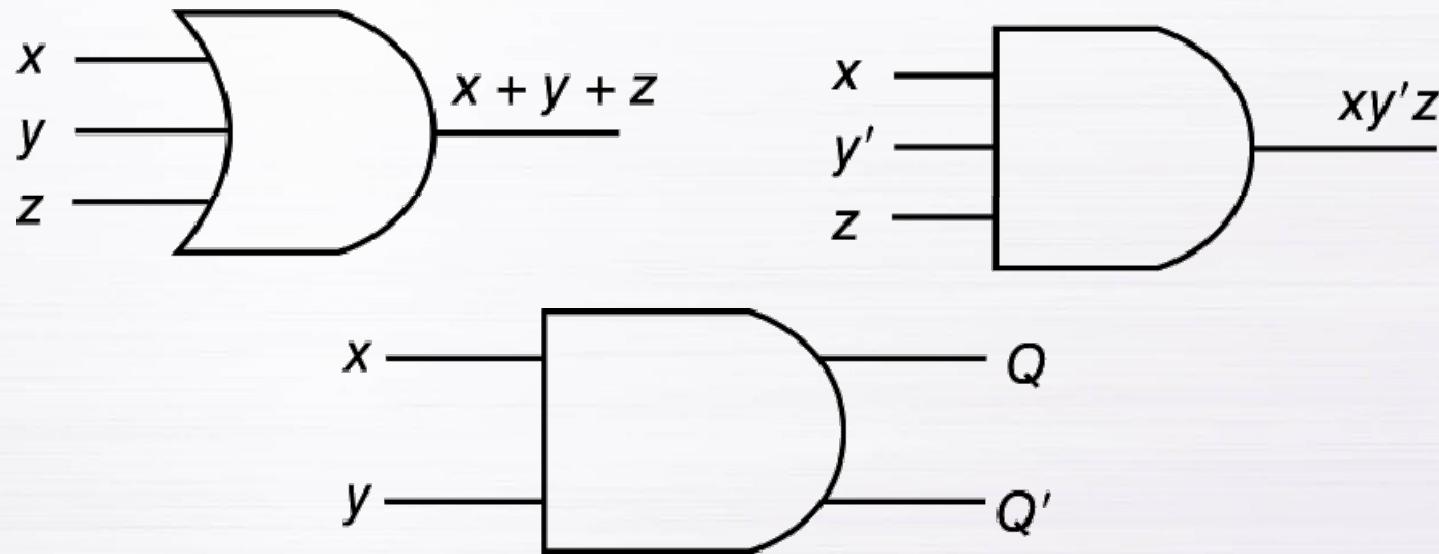


OR



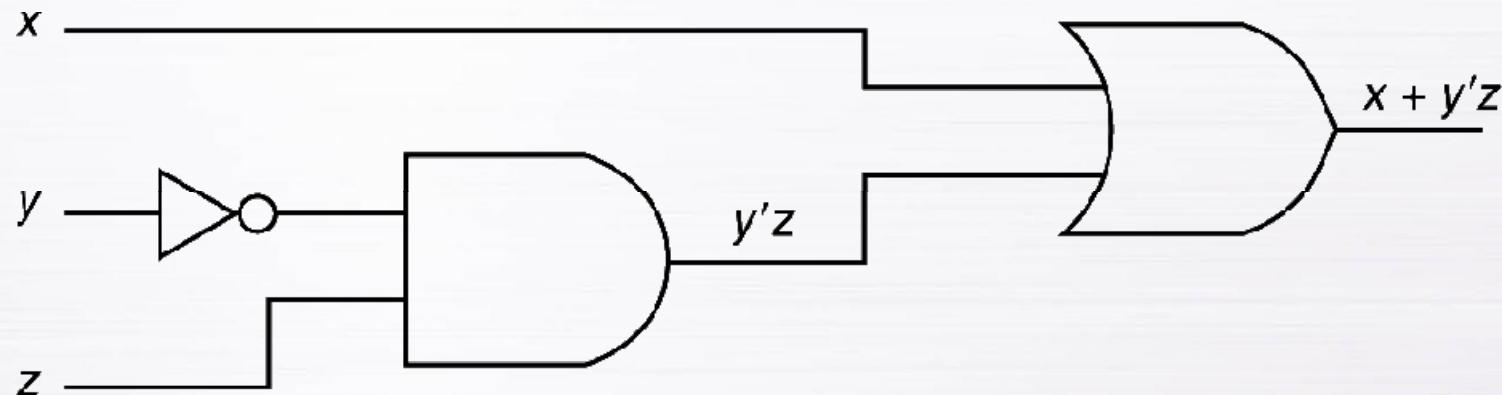
3.3 Logic Gates

- Gates can have multiple inputs and more than one output.
 - A second output can be provided for the complement of the operation.
 - We'll see more of this later.



3.3 Logic Gates

- The main thing to remember is that combinations of gates implement Boolean functions.
- The circuit below implements the Boolean function $F(x, y, z) = x + y'z$:



We simplify our Boolean expressions so that we can create simpler circuits.

3.5 Combinational Circuits

- We have designed a circuit that implements the Boolean function:

$$F(X, Y, Z) = X + \bar{Y}Z$$

- This circuit is an example of a *combinational logic* circuit.
- Combinational logic circuits produce a specified output (almost) at the instant when input values are applied.
 - In a later section, we will explore circuits where this is not the case.

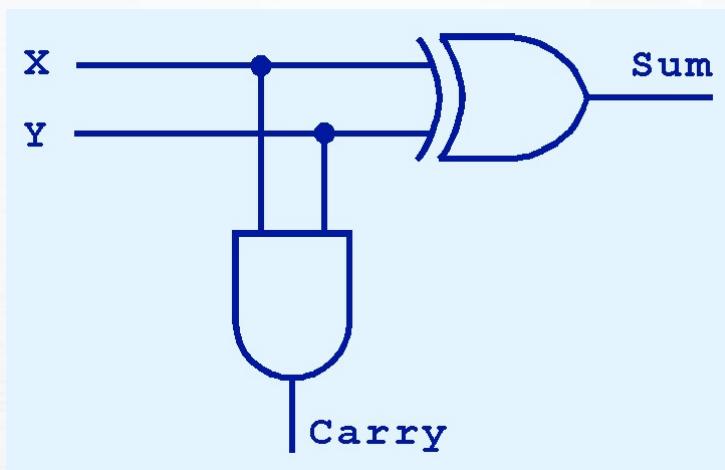
3.5 Combinational Circuits

- Combinational logic circuits give us many useful devices.
- One of the simplest is the *half adder*, which finds the sum of two bits.
- We can gain some insight as to the construction of a half adder by looking at its truth table, shown at the right.

Inputs		Outputs	
x	y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

3.5 Combinational Circuits

- As we see, the sum can be found using the XOR operation and the carry using the AND operation.



Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

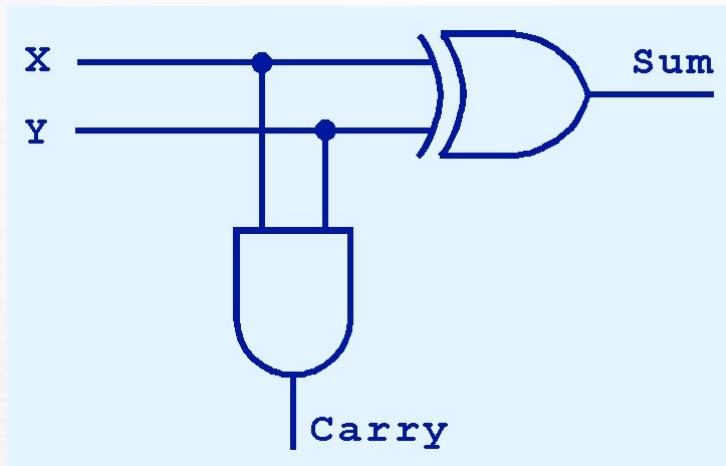
3.5 Combinational Circuits

- We can change our half adder into to a full adder by including gates for processing the carry bit.
- The truth table for a full adder is shown at the right.

Inputs			Outputs		
Carry In			Carry		
x	y	In	Sum	Out	
0	0	0	0	0	
0	0	1	1	0	
0	1	0	1	0	
0	1	1	0	1	
1	0	0	1	0	
1	0	1	0	1	
1	1	0	0	1	
1	1	1	1	1	

3.5 Combinational Circuits

- How can we change the half adder shown below to make it a full adder?

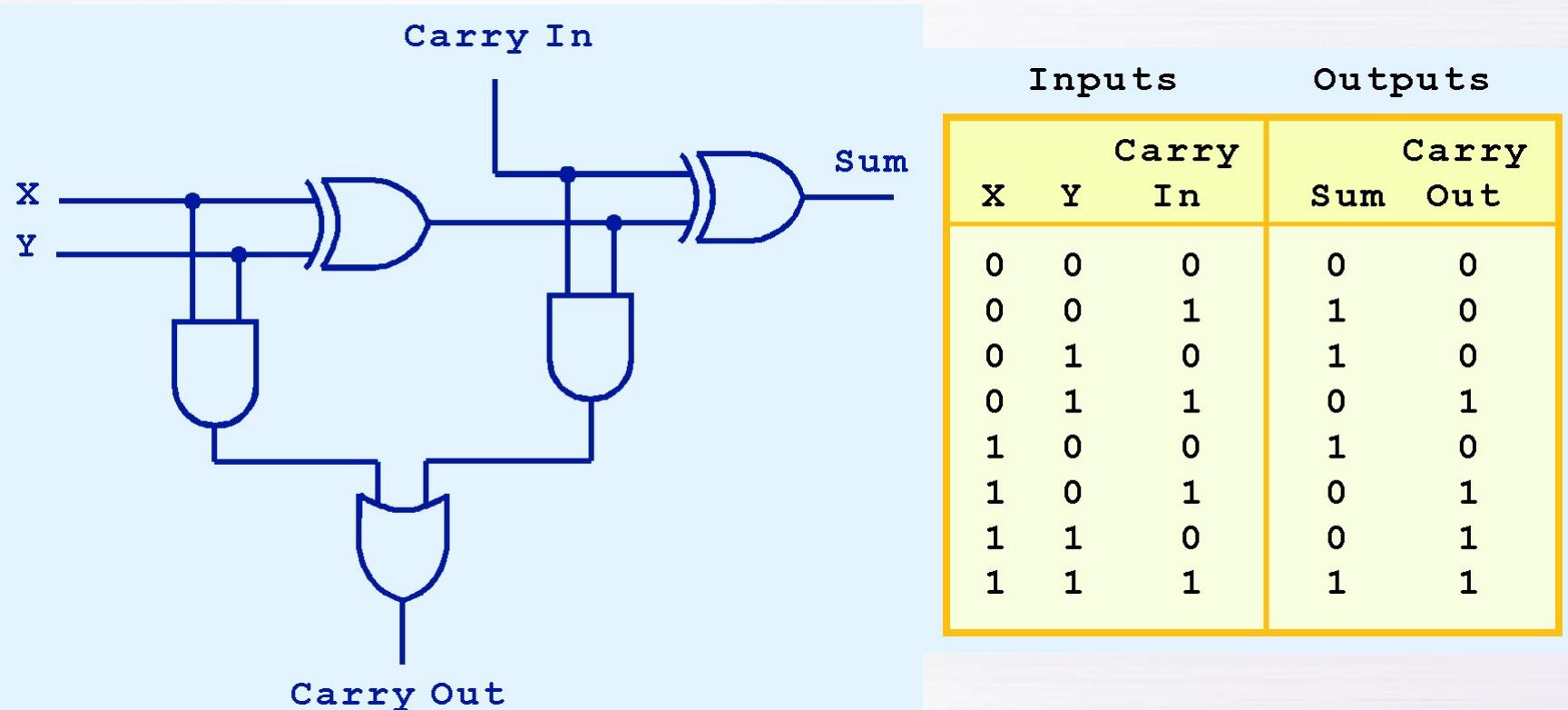


Inputs Outputs

Carry In			Carry Sum Out	
X	Y			
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

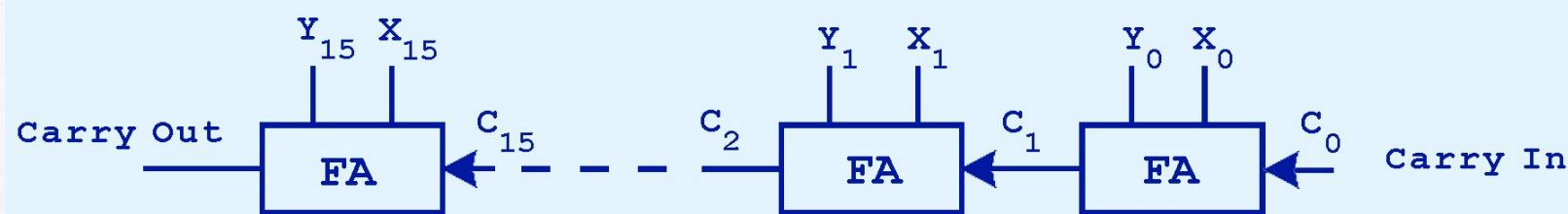
3.5 Combinational Circuits

- Here's our completed full adder.



3.5 Combinational Circuits

- Just as we combined half adders to make a full adder, full adders can connected in series.
- The carry bit “ripples” from one adder to the next; hence, this configuration is called a *ripple-carry adder*.



Today's systems employ more efficient adders.

3.5 Combinational Circuits

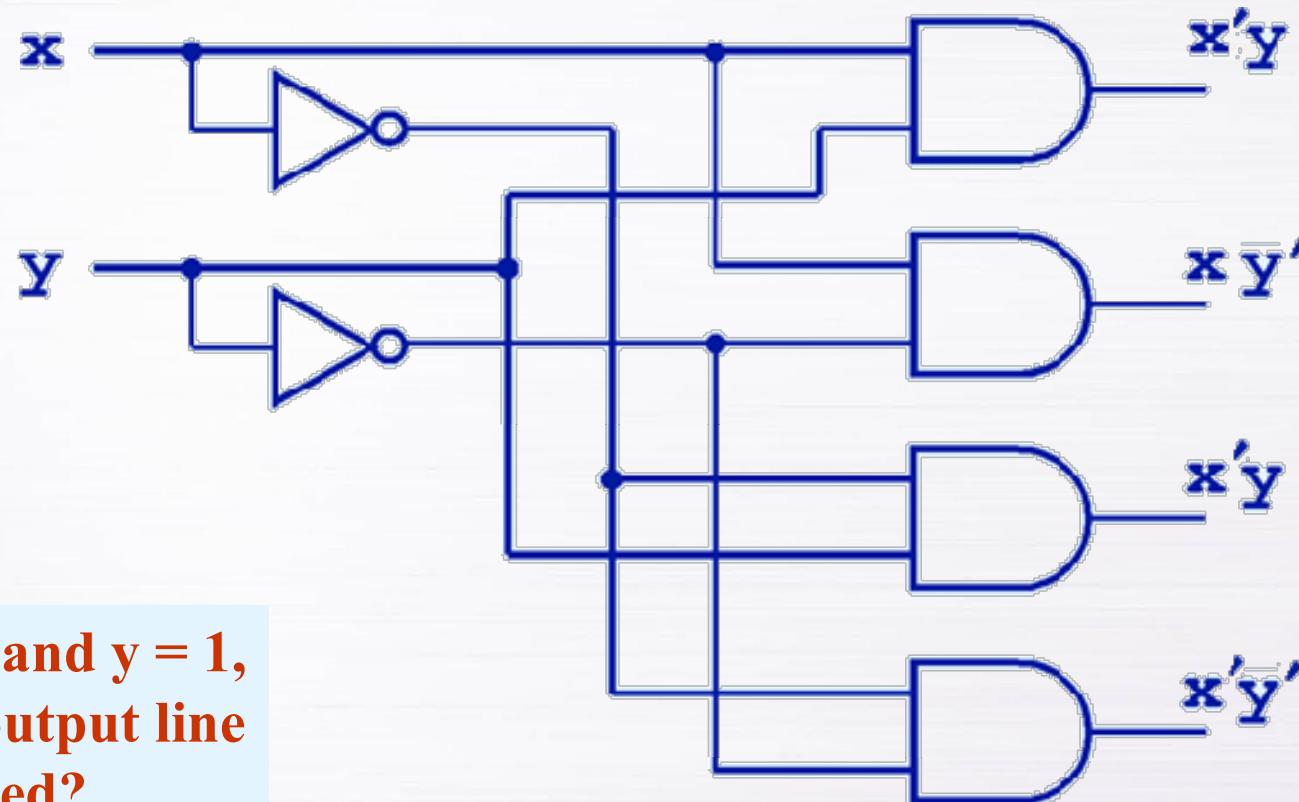
- Decoders are another important type of combinational circuit.
- Among other things, they are useful in selecting a memory location according a binary value placed on the address lines of a memory bus.
- Address decoders with n inputs can select any of 2^n locations.

This is a block diagram for a decoder.



3.5 Combinational Circuits

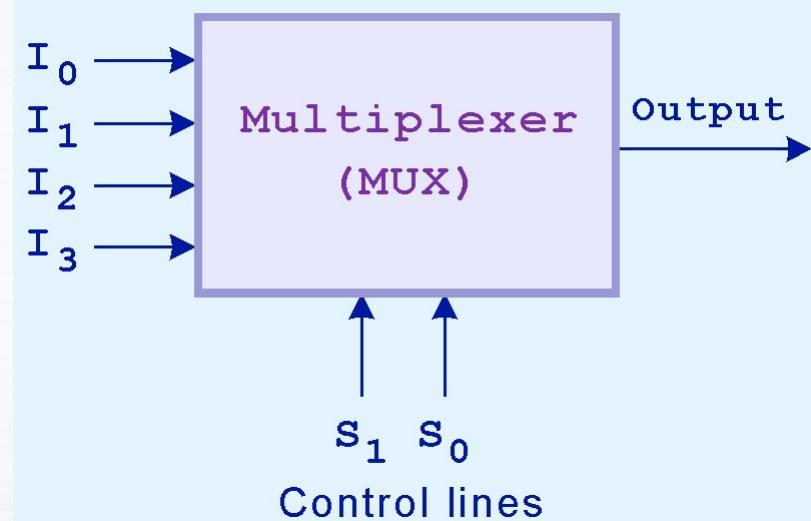
- This is what a 2-to-4 decoder looks like on the inside.



If $x = 0$ and $y = 1$,
which output line
is enabled?

3.5 Combinational Circuits

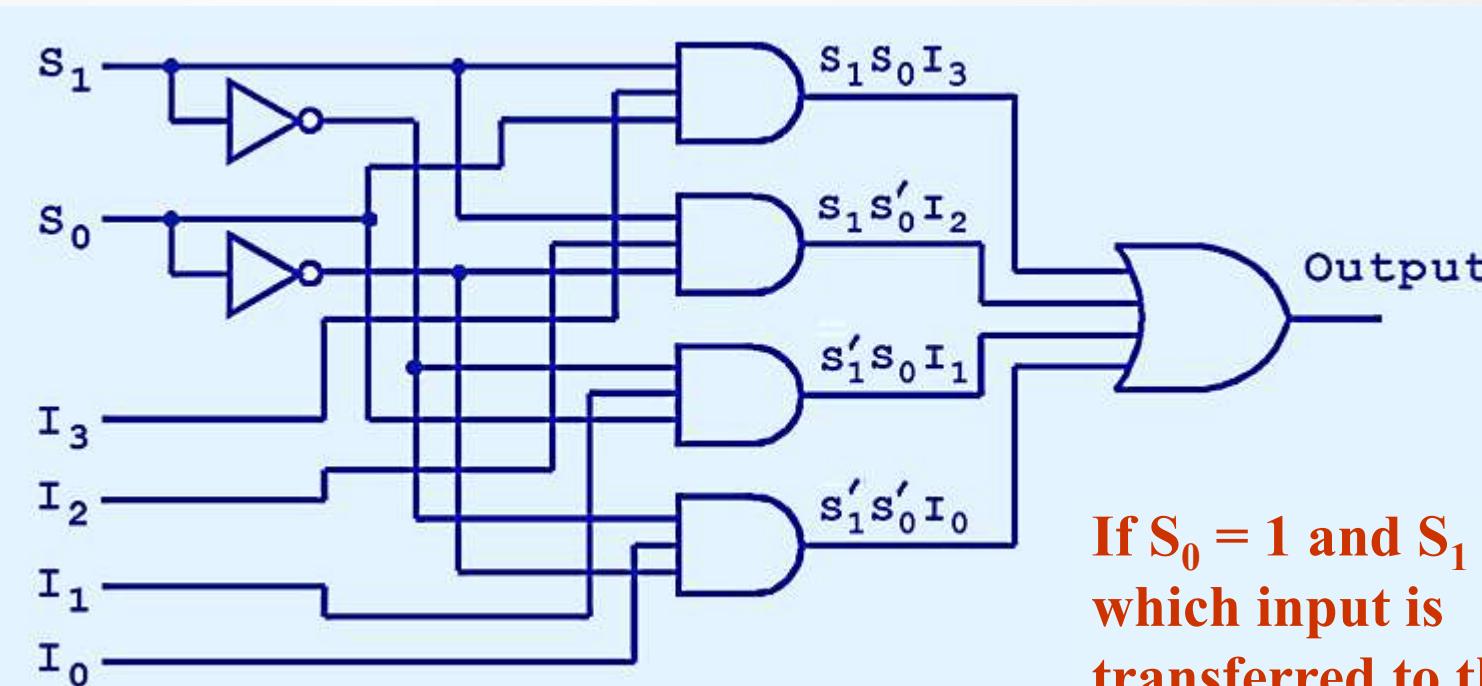
- A multiplexer does just the opposite of a decoder.
- It selects a single output from several inputs.
- The particular input chosen for output is determined by the value of the multiplexer's control lines.
- To be able to select among n inputs, $\log_2 n$ control lines are needed.



This is a block diagram for a multiplexer.

3.5 Combinational Circuits

- This is what a 4-to-1 multiplexer looks like on the inside.

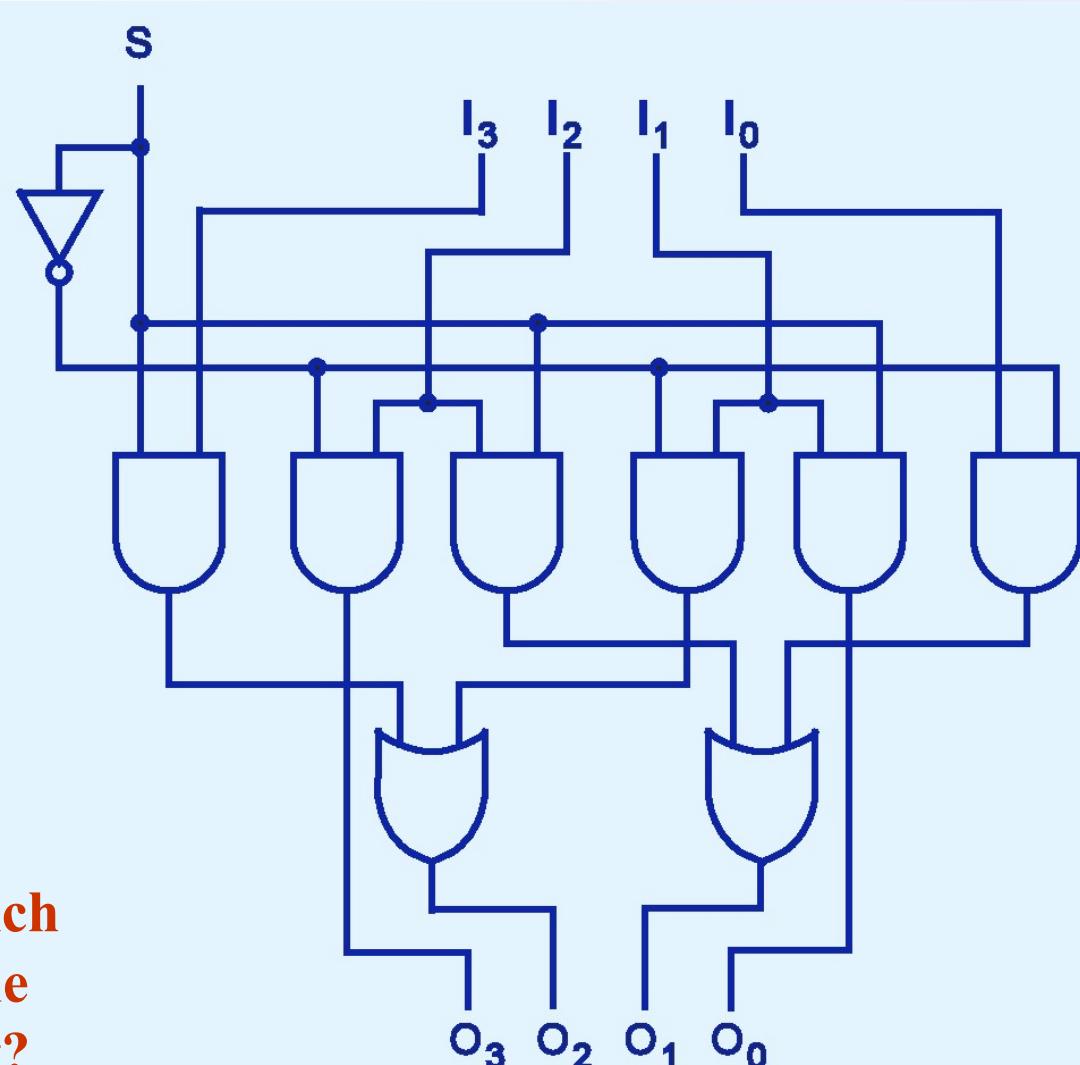


If $S_0 = 1$ and $S_1 = 0$,
which input is
transferred to the
output?

3.5 Combinational Circuits

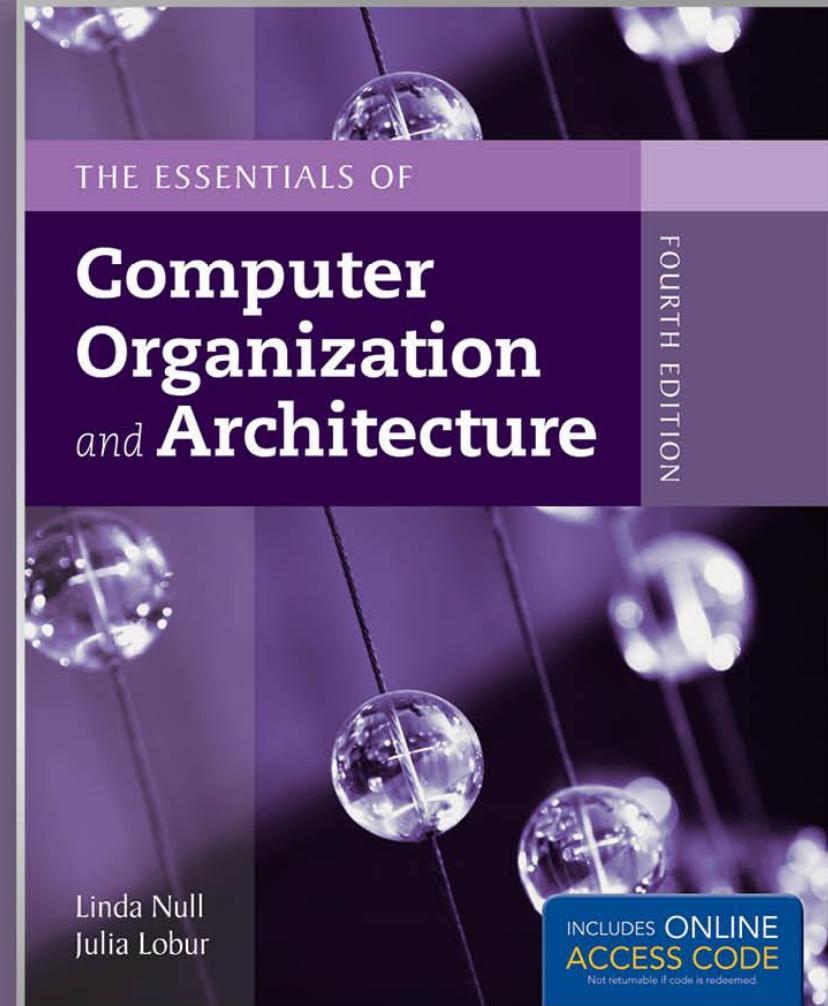
- This shifter moves the bits of a nibble one position to the left or right.

If $S = 0$, in which direction do the input bits shift?



Chapter 1

Introduction



Chapter 1 Objectives

- Know the difference between computer organization and computer architecture.
- Understand units of measure common to computer systems.
- Appreciate the evolution of computers.
- Understand the computer as a layered system.
- Be able to explain the von Neumann architecture and the function of basic computer components.

1.1 Overview

Why study computer organization and architecture?

- Design better programs, including system software such as compilers, operating systems, and device drivers.
- Optimize program behavior.
- Evaluate (benchmark) computer system performance.
- Understand time, space, and price tradeoffs.

1.1 Overview

- Computer organization
 - Encompasses all physical aspects of computer systems.
 - E.g., circuit design, control signals, memory types.
 - *How does a computer work?*
- Computer architecture
 - Logical aspects of system implementation as seen by the programmer.
 - E.g., instruction sets, instruction formats, data types, addressing modes.
 - *How do I design a computer?*

1.2 Computer Components

- There is no clear distinction between matters related to computer organization and matters relevant to computer architecture.
- Principle of Equivalence of Hardware and Software:
 - *Any task done by software can also be done using hardware, and any operation performed directly by hardware can be done using software.**

* Assuming speed is not a concern.

1.2 Computer Components

- At the most basic level, a computer is a device consisting of three pieces:
 - A processor to interpret and execute programs
 - A memory to store both data and programs
 - A mechanism for transferring data to and from the outside world.

1.3 An Example System

Consider this advertisement:

FOR SALE: OBSOLETE COMPUTER – CHEAP! CHEAP!

- Intel i7 Quad Core, 3.9GHz
- 1600MHz 32GB DDR3 SDRAM
- 128KB L1 cache, 2MB L2 cache
- 1TB SATA hard drive (7200 RPM)
- 10 USB ports, 1 serial port, 4 PCI express slots (1 PCI, 1 PCIe x16, 2 PCIe x1), Bluetooth, and...
- 24" widescreen LCD monitor, 16:10 aspect ratio, 1920x1200 WUXGA, 300 cd/m², active matrix, 1000:1 contrast, 8ms, 24-bit color (16.7 million colors), VGA/DVI, 2 USB ports
- 16x CD/DVD +/- RW drive
- 1GB PCIe video card
- PCIe sound card
- Integrated 10/100/1000 Ethernet

What does it all mean??

1.3 An Example System

Measures of capacity and speed:

- Kilo- (K) = 1 thousand = 10^3 and 2^{10}
- Mega- (M) = 1 million = 10^6 and 2^{20}
- Giga- (G) = 1 billion = 10^9 and 2^{30}
- Tera- (T) = 1 trillion = 10^{12} and 2^{40}
- Peta- (P) = 1 quadrillion = 10^{15} and 2^{50}
- Exa- (E) = 1 quintillion = 10^{18} and 2^{60}
- Zetta- (Z) = 1 sextillion = 10^{21} and 2^{70}
- Yotta- (Y) = 1 septillion = 10^{24} and 2^{80}

Whether a metric refers to a power of ten or a power of two typically depends upon what is being measured.

1.3 An Example System

- Hertz = clock cycles per second (frequency)
 - 1MHz = 1,000,000Hz
 - Processor speeds are measured in MHz or GHz.
- Byte = a unit of storage
 - 1KB = 2^{10} = 1024 Bytes
 - 1MB = 2^{20} = 1,048,576 Bytes
 - 1GB = 2^{30} = 1,099,511,627,776 Bytes
 - Main memory (RAM) is measured in GB
 - Disk storage is measured in GB for small systems, TB (2^{40}) for large systems.

1.3 An Example System

Measures of time and space:

- Milli- (m) = 1 thousandth = 10^{-3}
- Micro- (μ) = 1 millionth = 10^{-6}
- Nano- (n) = 1 billionth = 10^{-9}
- Pico- (p) = 1 trillionth = 10^{-12}
- Femto- (f) = 1 quadrillionth = 10^{-15}
- Atto- (a) = 1 quintillionth = 10^{-18}
- Zepto- (z) = 1 sextillionth = 10^{-21}
- Yocto- (y) = 1 septillionth = 10^{-24}

1.3 An Example System

- Millisecond = 1 thousandth of a second
 - Hard disk drive access times are often 10 to 20 milliseconds.
- Nanosecond = 1 billionth of a second
 - Main memory access times are often 50 to 70 nanoseconds.
- Micron (micrometer) = 1 millionth of a meter
 - Circuits on computer chips are measured in microns.

1.3 An Example System

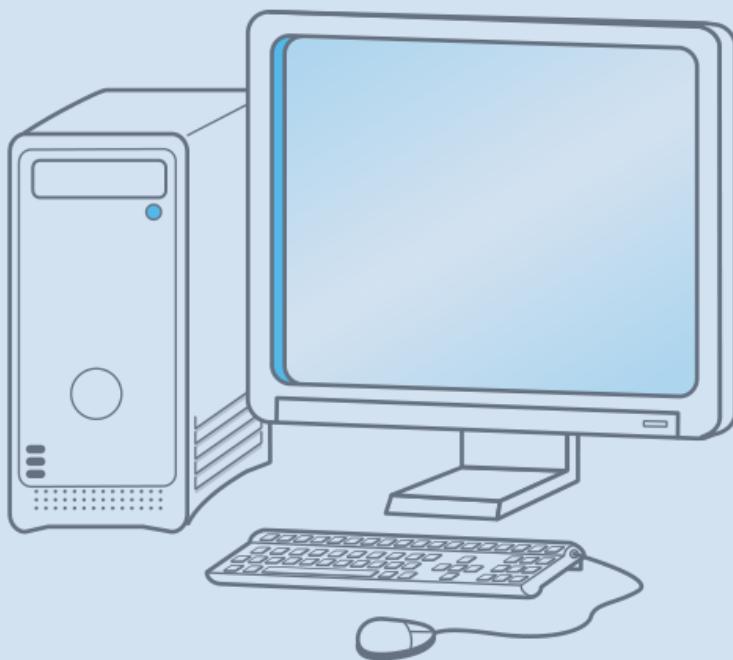
- We note that cycle time is the reciprocal of clock frequency.
- A bus operating at 133MHz has a cycle time of 7.52 nanoseconds:

$$133,000,000 \text{ cycles/second} = 7.52\text{ns/cycle}$$

Now back to the advertisement ...

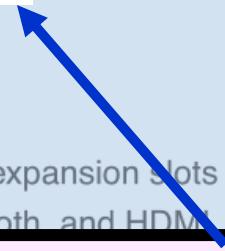
1.3 An Example System

FOR SALE: OBSOLETE COMPUTER – CHEAP! CHEAP! CHEAP!



- Intel i7 Quad Core, 3.9GHz
- 1600MHz 32GB DDR3 SDRAM
- 128KB L1 cache, 2MB L2 cache
- 1TB SATA hard drive (7200 RPM)
- 10 USB ports, 1 serial port, 4 PCI expansion slots
(1 PCI, 1 PCIx16, 2 PCIx1), Bluetooth, and HDMI
- 2
- 1
- (s
- in
- 1
- 1
- P
- I

The microprocessor is the “brain” of the system. It executes program instructions. This one is an Intel i7 running at 3.9GHz.

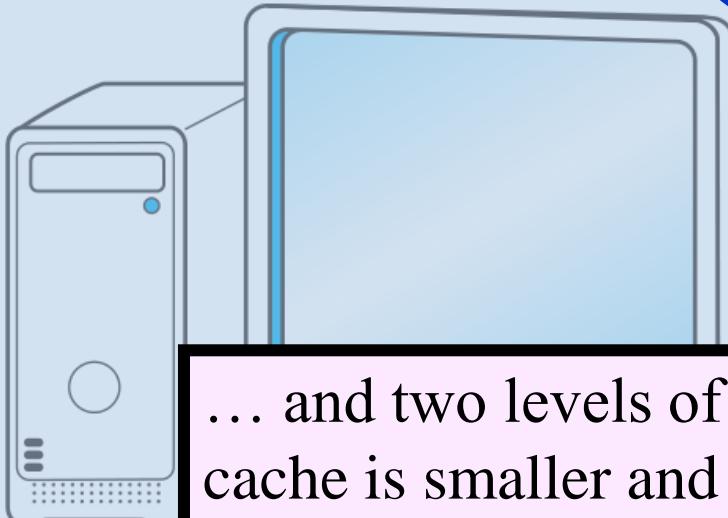


1.3 An Example System

- Computers with large main memory capacity can run larger programs with greater speed than computers having small memories.
- RAM is an acronym for *random access memory*. Random access means that memory contents can be accessed directly if you know its location.
- Cache is a type of temporary memory that can be accessed faster than RAM.

1.3 An Example System

This system has 32GB of (fast) synchronous dynamic RAM (SDRAM) . . .



CHEAP! CHEAP! CHEAP!

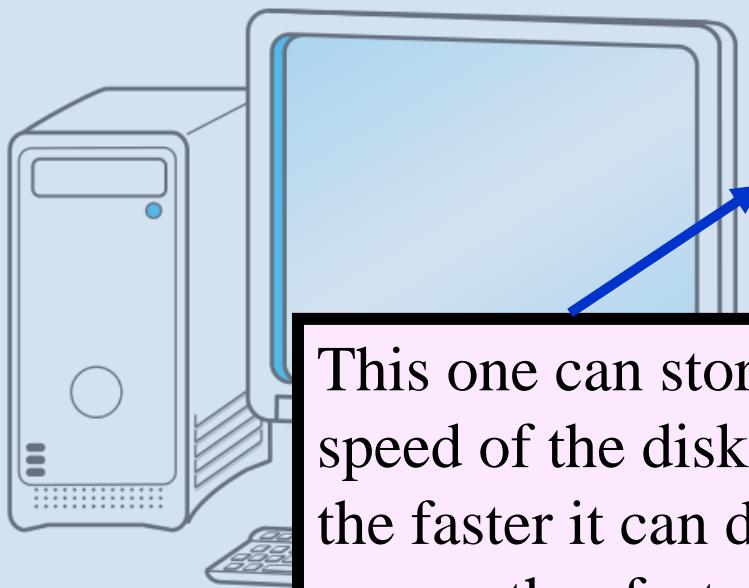
- Intel i7 Quad Core, 3.9GHz
- 1600MHz 32GB DDR3 SDRAM
- 128KB L1 cache, 2MB L2 cache
- 1TB SATA hard drive (7200 RPM)
- 10 USB ports, 1 serial port, 4 PCI expansion slots (1 PCI, 1 PCIx16, 2 PCIx1), Bluetooth, and HDMI
- 24" widescreen LCD monitor, 16:10 aspect ratio,

. . . and two levels of cache memory, the level 1 (L1) cache is smaller and (probably) faster than the L2 cache. Note that these cache sizes are measured in KB and MB.

- Integrated 10/100/1000 Ethernet

1.3 An Example System

Hard disk capacity determines the amount of data and size of programs you can store.



PUTER – CHEAP! CHEAP! CHEAP!

- Intel i7 Quad Core, 3.9GHz
- 1600MHz 32GB DDR3 SDRAM
- 128KB L1 cache, 2MB L2 cache
- 1TB SATA hard drive (7200 RPM)
- 10 USB ports, 1 serial port, 4 PCI expansion slots (1 PCI, 1 PCIx16, 2 PCIx1), Bluetooth, and HDMI
- 24" widescreen LCD monitor, 16:10 aspect ratio

This one can store 1TB. 7200 RPM is the rotational speed of the disk. Generally, the faster a disk rotates, the faster it can deliver data to RAM. (There are many other factors involved.)

- Integrated 10/100/1000 Ethernet

1.3 An Example System

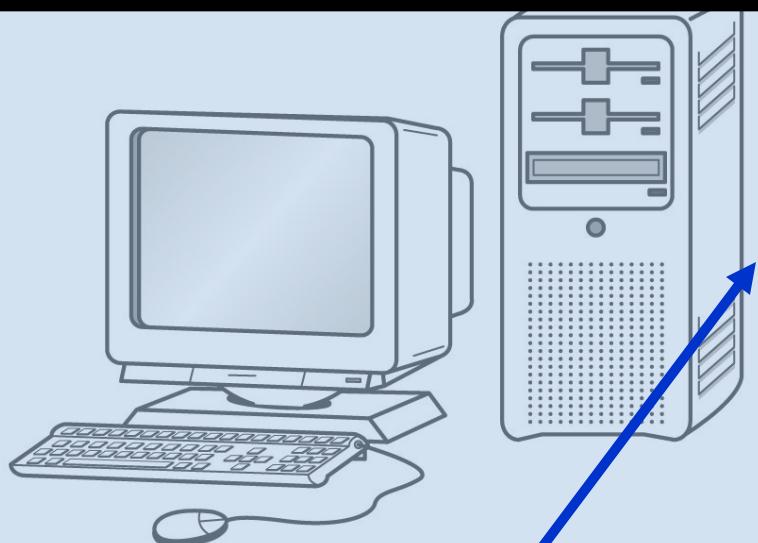
ATA stands for *advanced technology attachment*, which describes how the hard disk interfaces with (or connects to) other system components.

A DVD can store about 4.7GB of data. This drive supports rewritable DVDs, +/-RW, that can be written to many times.. 16x describes its speed.

- 1333MHz 4GB DDR SDRAM
- 128KB L1 cache, 2MB L2 cache
- 500GB serial ATA hard drive (7200 RPM)
- 4 USB ports, 1 serial port, 1 parallel port, 4 PCI expansion slots (1 PCI, 1 PCI x 16, 2 PCI x 1)
- Choice of monitor: 19", .24mm AG, 1280x1024 at 75Hz or 18.5", 1280x1024 SXGA, 250 cd/m², active matrix, 1000:1 (static), 5ms, 24-bit color (16.7 million colors), VGA/DVI input
- 16X DVD +/- RW Drive
- 1GB PCIe video card
- PCIe sound card
- Integrated 10/100/1000 Ethernet

1.3 An Example System

Ports allow movement of data between a system and its external devices.



This system has ten ports.

CHEAP! CHEAP! CHEAP!

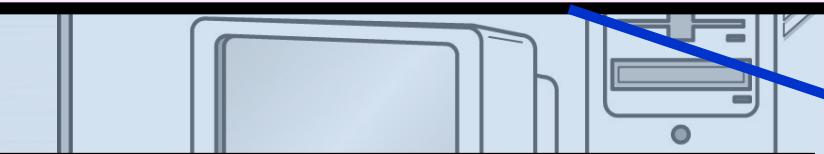
- Intel Pentium Dual Core, 3.06 GHz
- 1333MHz 4GB DDR SDRAM
- 128KB L1 cache, 2MB L2 cache
- 500GB serial ATA hard drive (7200 RPM)
- 4 USB ports, 1 serial port, 1 parallel port, 4 PCI expansion slots (1 PCI, 1 PCI x 16, 2 PCI x 1)
- Choice of monitor: 19", .24mm AG, 1280x1024 at 75Hz or 18.5", 1280x1024 SXGA, 250 cd/m², active matrix, 1000:1 (static), 5ms, 24-bit color (16.7 million colors), VGA/DVI input
- 16X DVD +/- RW Drive
- 1GB PCIe video card
- sound card
- integrated 10/100/1000 Ethernet

1.3 An Example System

- Serial ports send data as a series of pulses along one or two data lines.
- Parallel ports send data as a single pulse along at least eight data lines.
- USB, Universal Serial Bus, is an intelligent serial interface that is self-configuring. (It supports “plug and play.”)

1.3 An Example System

System buses can be augmented by dedicated I/O buses. PCI, *peripheral component interface*, is one such bus.



This system has two PCIe (*PCI express*) devices: a video card and a sound card.



HEAP! CHEAP!

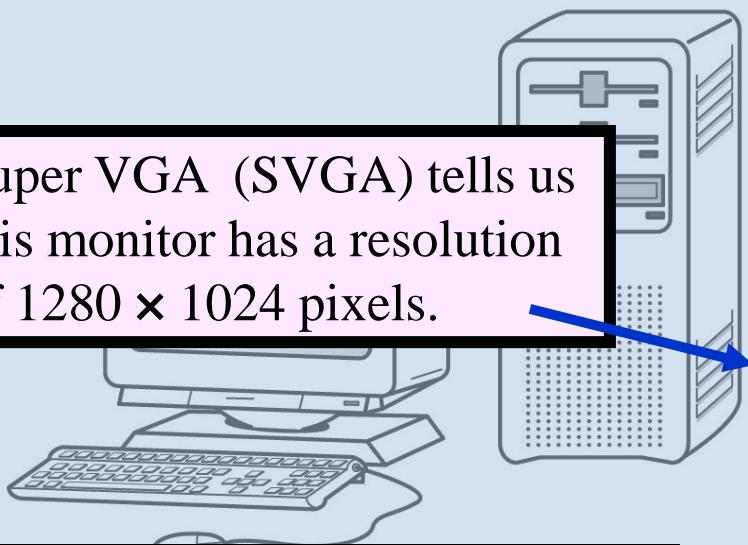
Dual Core, 3.06 GHz
4GB DDR SDRAM

- 128KB L1 cache, 2MB L2 cache
- 500GB serial ATA hard drive (7200 RPM)
- 4 USB ports, 1 serial port, 1 parallel port, 4 PCI expansion slots (1 PCI, 1 PCI x 16, 2 PCI x 1)
- Choice of monitor: 19", .24mm AG, 1280x1024 at 75Hz or 18.5", 1280x1024 SXGA, 250 cd/m², active matrix, 1000:1 (static), 5ms, 24-bit color (16.7 million colors), VGA/DVI input
- 16X DVD +/- RW Drive
- 1GB PCIe video card
- PCIe sound card
- Integrated 10/100/1000 Ethernet

1.3 An Example System

Active matrix technology uses one transistor per picture element (*pixel*). The *resolution* of a monitor determines the amount of text and graphics that the monitor can display.

Super VGA (SVGA) tells us this monitor has a resolution of 1280×1024 pixels.



The video card contains memory and programs that support the monitor.

- Intel Pentium Dual Core, 3.06 GHz
- 1333MHz 4GB DDR SDRAM
- 128KB L1 cache, 2MB L2 cache
- 500GB serial ATA hard drive (7200 RPM)
- 4 USB ports, 1 serial port, 1 parallel port, 4 PCI expansion slots (1 PCI, 1 PCI x 16, 2 PCI x 1)
- Choice of monitor: 19", .24mm AG, 1280x1024 at 75Hz or 18.5", 1280x1024 SXGA, 250 cd/m², active matrix, 1000:1 (static), 5ms, 24-bit color (16.7 million colors), VGA/DVI input
- 16X DVD +/- RW Drive
- 1GB PCIe video card
- PCIe sound card
- Integrated 10/100/1000 Ethernet

1.3 An Example System

Throughout the remainder of the book you will see how these components work and how they interact with software to make complete computer systems.

This statement raises two important questions:

What assurance do we have that computer components will operate as we expect?

And what assurance do we have that computer components will operate together?

1.4 Standards Organizations

- There are many organizations that set computer hardware standards-- to include the interoperability of computer components.
- Throughout this book, and in your career, you will encounter many of them.
- Some of the most important standards-setting groups are . . .

1.4 Standards Organizations

- The Institute of Electrical and Electronic Engineers (IEEE)
 - Promotes the interests of the worldwide electrical engineering community.
 - Establishes standards for computer components, data representation, and signaling protocols, among many other things.

1.4 Standards Organizations

- The International Telecommunications Union (ITU)
 - Concerns itself with the interoperability of telecommunications systems, including data communications and telephony.
- National groups establish standards within their respective countries:
 - The American National Standards Institute (ANSI)
 - The British Standards Institution (BSI)

1.4 Standards Organizations

- The International Organization for Standardization (ISO)
 - Establishes worldwide standards for everything from screw threads to photographic film.
 - Is influential in formulating standards for computer hardware and software, including their methods of manufacture.

Note: ISO is **not** an acronym. ISO comes from the Greek, *isos*, meaning “equal.”

1.5 Historical Development

- To fully appreciate the computers of today, it is helpful to understand how things got the way they are.
- The evolution of computing machinery has taken place over several centuries.
- In modern times computer evolution is usually classified into four generations according to the salient technology of the era.

We note that many of the following dates are approximate.

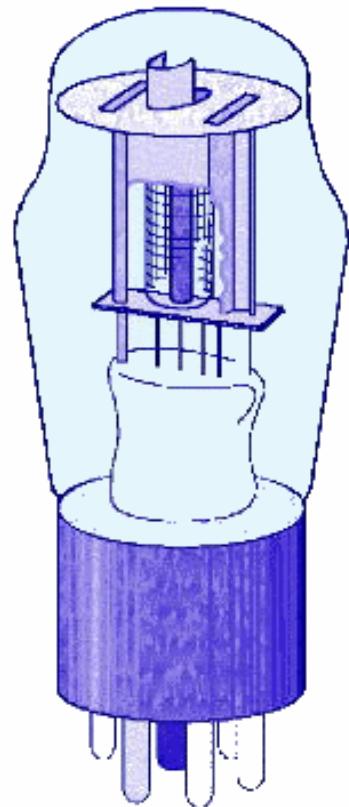
1.5 Historical Development

- Generation Zero: Mechanical Calculating Machines (1642 - 1945)
 - Calculating Clock - Wilhelm Schickard (1592 - 1635).
 - Pascaline - Blaise Pascal (1623 - 1662).
 - Difference Engine - Charles Babbage (1791 - 1871), also designed but never built the Analytical Engine.
 - Punched card tabulating machines - Herman Hollerith (1860 - 1929).

Hollerith cards were commonly used for computer input well into the 1970s.

1.5 Historical Development

- The First Generation: Vacuum Tube Computers (1945 - 1953)
 - Atanasoff Berry Computer (1937 - 1938) solved systems of linear equations.
 - John Atanasoff and Clifford Berry of Iowa State University.



1.5 Historical Development

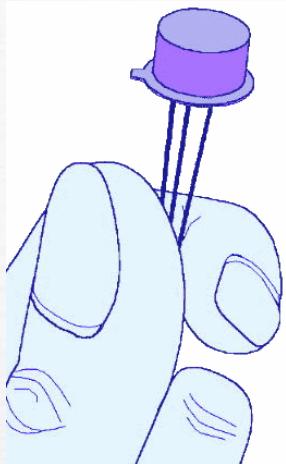
- The First Generation: Vacuum Tube Computers (1945 - 1953)
 - Electronic Numerical Integrator and Computer (ENIAC)
 - John Mauchly and J. Presper Eckert
 - University of Pennsylvania, 1946
- The ENIAC was the first general-purpose computer.

1.5 Historical Development

- The First Generation: Vacuum Tube Computers (1945 - 1953)
 - The IBM 650 first mass-produced computer. (1955)
 - It was phased out in 1969.
 - Other major computer manufacturers of this period include UNIVAC, Engineering Research Associates (ERA), and Computer Research Corporation (CRC).
 - UNIVAC and ERA were bought by Remington Rand, the ancestor of the Unisys Corporation.
 - CRC was bought by the Underwood (typewriter) Corporation, which left the computer business.

1.5 Historical Development

- The Second Generation: Transistorized Computers (1954 - 1965)
 - IBM 7094 (scientific) and 1401 (business)
 - Digital Equipment Corporation (DEC) PDP-1
 - Univac 1100
 - Control Data Corporation 1604.
 - . . . and many others.



These systems had few architectural similarities.

1.5 Historical Development

- The Third Generation: Integrated Circuit Computers (1965 - 1980)
 - IBM 360
 - DEC PDP-8 and PDP-11
 - Cray-1 supercomputer
 - . . . and many others.
- By this time, IBM had gained overwhelming dominance in the industry.
 - Computer manufacturers of this era were characterized as IBM and the BUNCH (Burroughs, Unisys, NCR, Control Data, and Honeywell).

1.5 Historical Development

- The Fourth Generation: VLSI Computers (1980 - ????)
 - Very large scale integrated circuits (VLSI) have more than 10,000 components per chip.
 - Enabled the creation of microprocessors.
 - The first was the 4-bit Intel 4004.
 - Later versions, such as the 8080, 8086, and 8088 spawned the idea of “personal computing.”

1.5 Historical Development

- Moore's Law (1965)
 - Gordon Moore, Intel founder
 - “The density of transistors in an integrated circuit will double every year.”
- Contemporary version:
 - “The density of silicon chips doubles every 18 months.”

But this “law” cannot hold forever ...

1.5 Historical Development

- Rock's Law
 - Arthur Rock, Intel financier
 - “The cost of capital equipment to build semiconductors will double every four years.”
 - In 1968, a new chip plant cost about \$12,000.

At the time, \$12,000 would buy a nice home in the suburbs.

An executive earning \$12,000 per year was “making a very comfortable living.”

1.5 Historical Development

- Rock's Law
 - In 2012, a chip plants under construction cost well over \$5 billion.

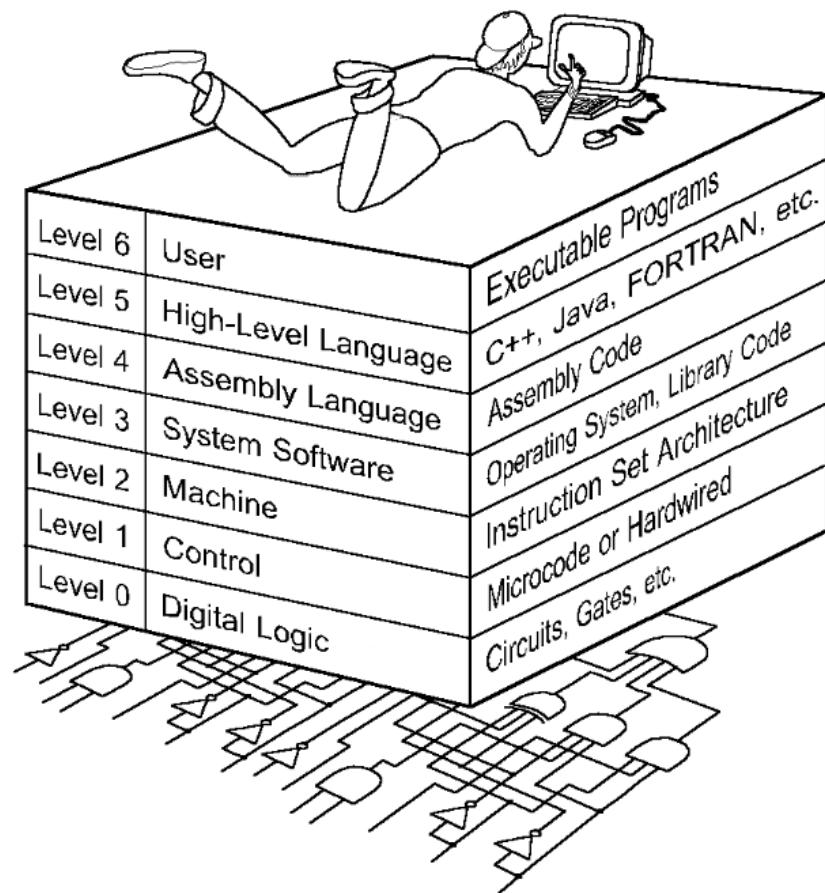
\$5 billion is more than the gross domestic product of some small countries, including Barbados, Mauritania, and Rwanda.
 - For Moore's Law to hold, Rock's Law must fall, or vice versa. But no one can say which will give out first.

1.6 The Computer Level Hierarchy

- Computers consist of many things besides chips.
- Before a computer can do anything worthwhile, it must also use software.
- Writing complex programs requires a “divide and conquer” approach, where each program module solves a smaller problem.
- Complex computer systems employ a similar technique through a series of virtual machine layers.

1.6 The Computer Level Hierarchy

- Each virtual machine layer is an abstraction of the level below it.
- The machines at each level execute their own particular instructions, calling upon machines at lower levels to perform tasks as required.
- Computer circuits ultimately carry out the work.



1.6 The Computer Level Hierarchy

- Level 6: The User Level
 - Program execution and user interface level.
 - The level with which we are most familiar.
- Level 5: High-Level Language Level
 - The level with which we interact when we write programs in languages such as C, Pascal, Lisp, and Java.

1.6 The Computer Level Hierarchy

- Level 4: Assembly Language Level
 - Acts upon assembly language produced from Level 5, as well as instructions programmed directly at this level.
- Level 3: System Software Level
 - Controls executing processes on the system.
 - Protects system resources.
 - Assembly language instructions often pass through Level 3 without modification.

1.6 The Computer Level Hierarchy

- Level 2: Machine Level
 - Also known as the Instruction Set Architecture (ISA) Level.
 - Consists of instructions that are particular to the architecture of the machine.
 - Programs written in machine language need no compilers, interpreters, or assemblers.

1.6 The Computer Level Hierarchy

- Level 1: Control Level
 - A *control unit* decodes and executes instructions and moves data through the system.
 - Control units can be *microprogrammed* or *hardwired*.
 - A microprogram is a program written in a low-level language that is implemented by the hardware.
 - Hardwired control units consist of hardware that directly executes machine instructions.

1.6 The Computer Level Hierarchy

- Level 0: Digital Logic Level
 - This level is where we find digital circuits (the chips).
 - Digital circuits consist of gates and wires.
 - These components implement the mathematical logic of all other levels.

1.7 Computing as a Service: Cloud Computing

- The ultimate aim of every computer system is to deliver functionality to its users.
- Computer users typically do not care about terabytes of storage and gigahertz of processor speed.
- Many companies outsource their data centers to third-party specialists, who agree to provide computing services for a fee.
- These arrangements are managed through *service-level agreements (SLAs)*.

1.7 Computing as a Service: Cloud Computing

- Rather than pay a third party to run a company-owned data center, another approach is to buy computing services from someone else's data center and connect to it via the Internet.
- This is the idea behind a collection of service models known as Cloud computing.

The “Cloud” is a visual metaphor traditionally used for the Internet. It is even more apt for service-defined computing.

1.7 Computing as a Service: Cloud Computing

- Cloud computing models:
 - *Software as a Service, or SaaS.* The consumer of this service buy application services
 - Well-known examples include Gmail, Dropbox, GoToMeeting, and Netflix.
 - *Platform as a Service, or PaaS.* Provides server hardware, operating systems, database services, security components, and backup and recovery services.
 - Well-known PaaS providers include Google App Engine and Microsoft Windows Azure Cloud Services.

1.7 Computing as a Service: Cloud Computing

- - *Infrastructure as a Service (IaaS)* provides only server hardware, secure network access to the servers, and backup and recovery services. The customer is responsible for all system software including the operating system and databases.
 - Well-known IaaS platforms include Amazon EC2, Google Compute Engine, Microsoft Azure Services Platform, Rackspace, and HP Cloud.
 - Cloud storage is a limited type of IaaS that includes services such as Dropbox, Google Drive, and Amazon.com's Cloud Drive.

1.7 Computing as a Service: Cloud Computing

- Cloud computing relies on the concept of *elasticity* where resources can be added and removed as needed.
- You pay for only what you use.
- *Virtualization* is an enabler of elasticity.
 - Instead of having a physical machine, you have a “logical” machine that may span several physical machines, or occupy only part of a single physical machine.

1.8 The von Neumann Model

- On the ENIAC, all programming was done at the digital logic level.
- Programming the computer involved moving plugs and wires.
- A different hardware configuration was needed to solve every unique problem type.

Configuring the ENIAC to solve a “simple” problem required many days labor by skilled technicians.

1.8 The von Neumann Model

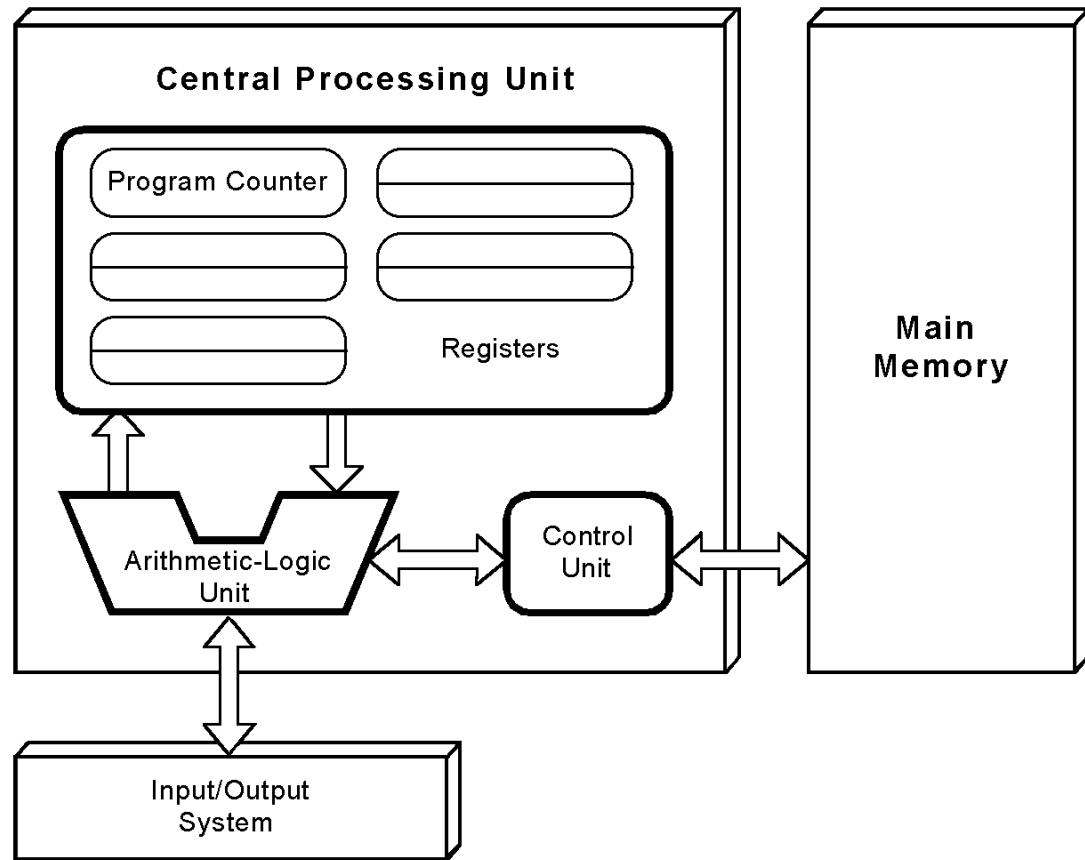
- Inventors of the ENIAC, John Mauchley and J. Presper Eckert, conceived of a computer that could store instructions in memory.
- The invention of this idea has since been ascribed to a mathematician, John von Neumann, who was a contemporary of Mauchley and Eckert.
- Stored-program computers have become known as von Neumann Architecture systems.

1.8 The von Neumann Model

- Today's stored-program computers have the following characteristics:
 - Three hardware systems:
 - A central processing unit (CPU)
 - A main memory system
 - An I/O system
 - The capacity to carry out sequential instruction processing.
 - A single data path between the CPU and main memory.
 - This single path is known as the *von Neumann bottleneck*.

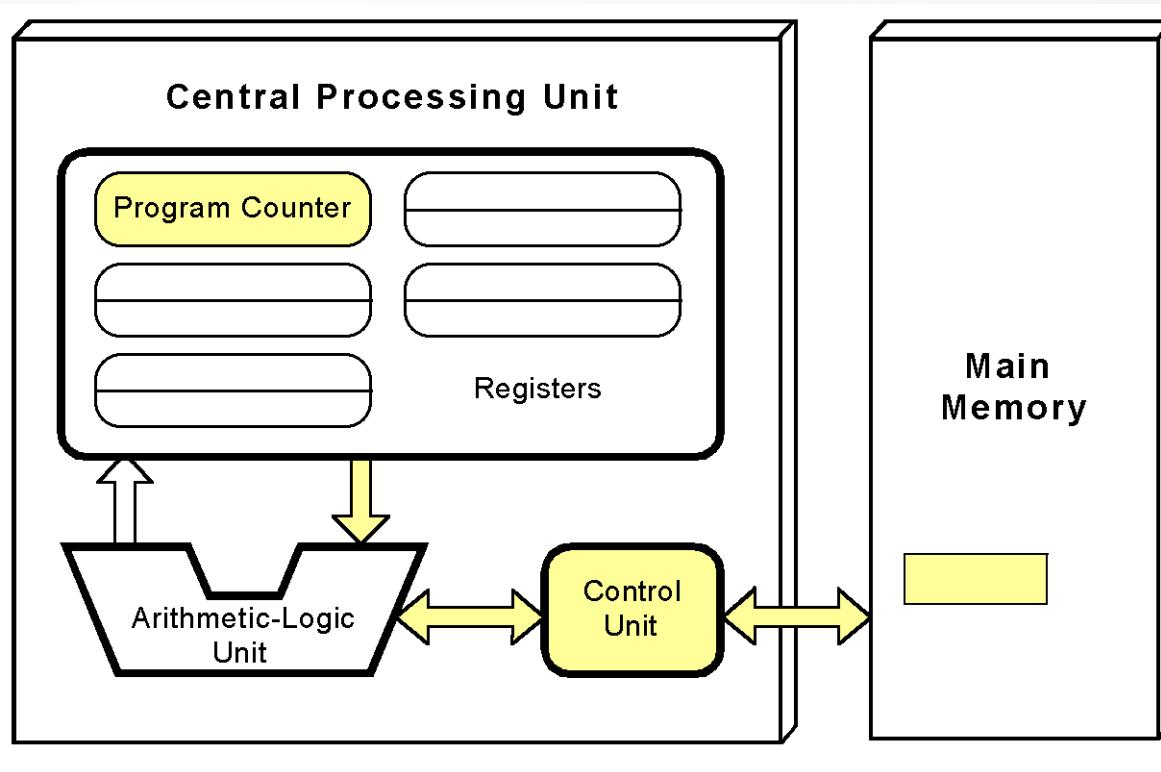
1.8 The von Neumann Model

- This is a general depiction of a von Neumann system:
- These computers employ a fetch-decode-execute cycle to run programs as follows . . .



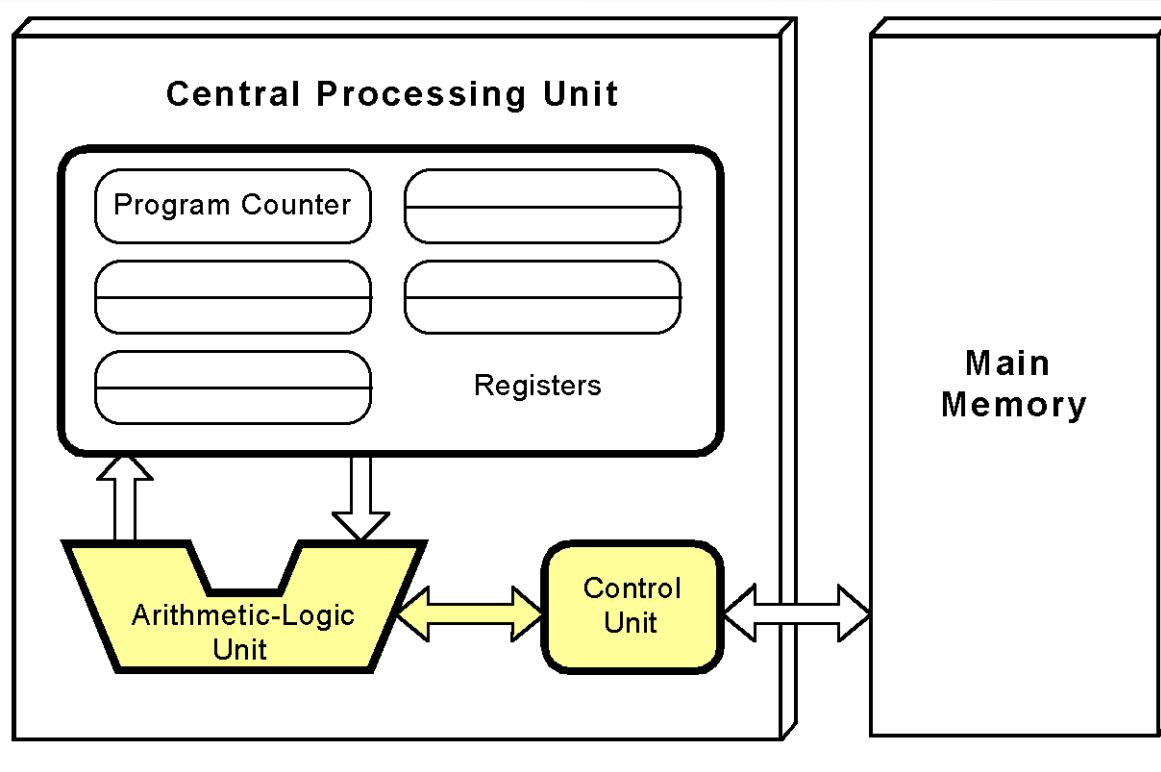
1.8 The von Neumann Model

- The control unit fetches the next instruction from memory using the program counter to determine where the instruction is located.



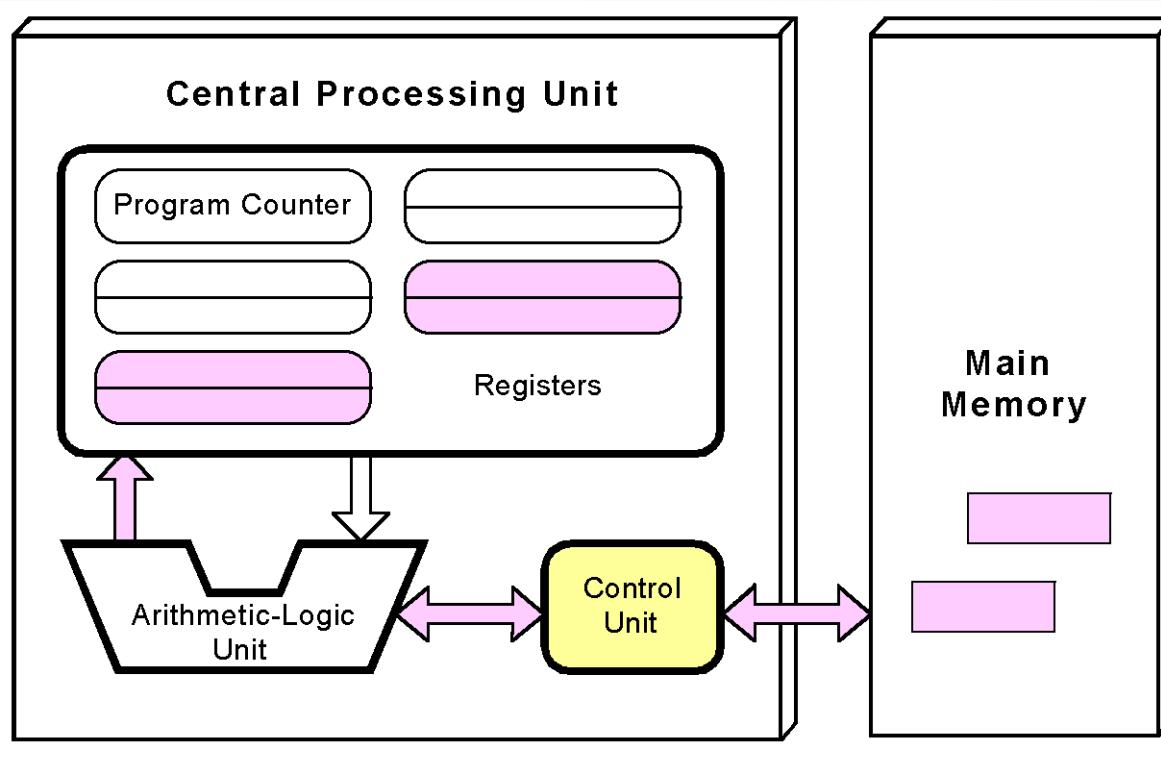
1.8 The von Neumann Model

- The instruction is decoded into a language that the ALU can understand.



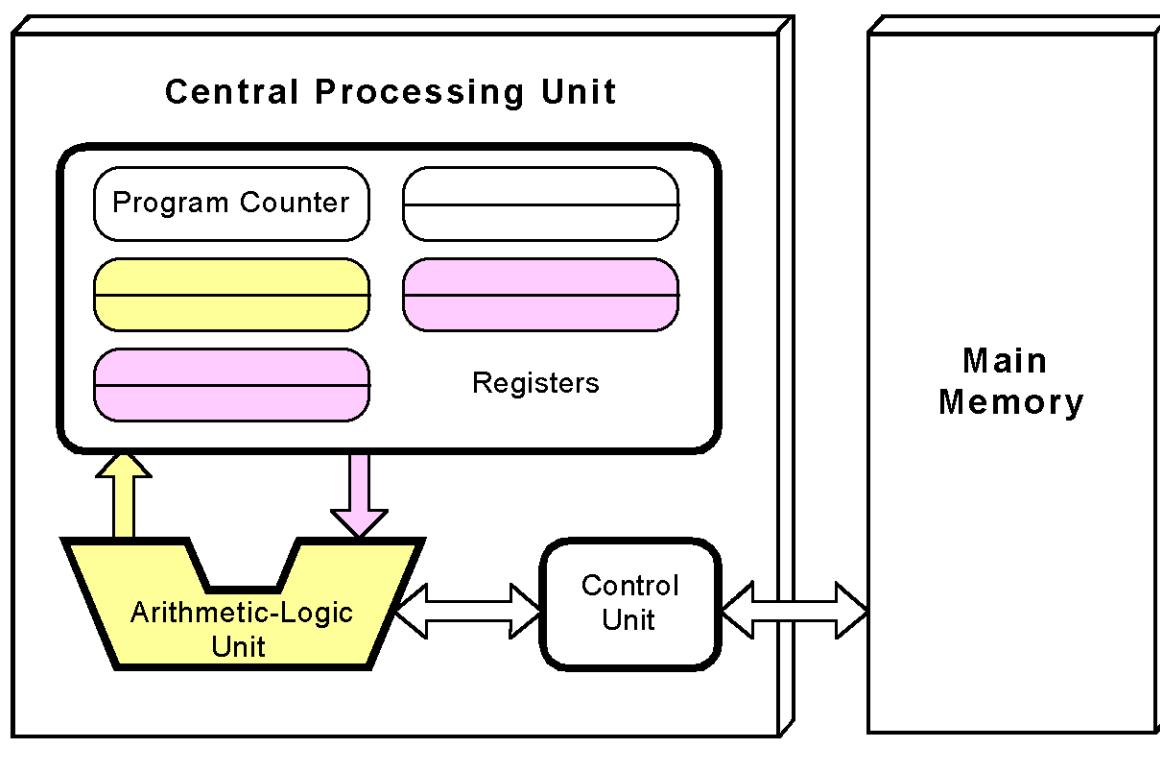
1.8 The von Neumann Model

- Any data operands required to execute the instruction are fetched from memory and placed into registers within the CPU.



1.8 The von Neumann Model

- The ALU executes the instruction and places results in registers or memory.



1.9 Non-von Neumann Models

- Conventional stored-program computers have undergone many incremental improvements over the years.
- These improvements include adding specialized buses, floating-point units, and cache memories, to name only a few.
- But enormous improvements in computational power require departure from the classic von Neumann architecture.
- Adding processors is one approach.

1.9 Non-von Neumann Models

- Some of today's systems have separate buses for data and instructions.
 - Called a Harvard architecture
- Other non-von Neumann systems provide special-purpose processors to offload work from the main CPU.
- More radical departures include dataflow computing, quantum computing, cellular automata, and parallel computing.

1.10 Parallel Computing

- In the late 1960s, high-performance computer systems were equipped with dual processors to increase computational throughput.
- In the 1970s supercomputer systems were introduced with 32 processors.
- Supercomputers with 1,000 processors were built in the 1980s.
- In 1999, IBM announced its Blue Gene system containing over 1 million processors.

1.10 Parallel Computing

- Parallel processing allows a computer to simultaneously work on subparts of a problem.
- Multicore processors have two or more processor cores sharing a single die.
- Each core has its own ALU and set of registers, but all processors share memory and other resources.
- “Dual core” differs from “dual processor.”
 - Dual-processor machines, for example, have two processors, but each processor plugs into the motherboard separately.

1.10 Parallel Computing

- Multi-core systems provide the ability to multitask
 - E.g., browse the Web while burning a CD
- Multithreaded applications spread mini-processes, *threads*, across one or more processors for increased throughput.
- New programming languages are necessary to fully exploit multiprocessor power.

1.11 Parallelism: Enabler of Machine Intelligence

- The quest for machine intelligence has been ongoing for over 300 years.
- The 20th Century witnessed the first machines that could beat human grandmasters at chess when Deep Blue beat Garry Kasparov in 1997.
- But the machine and the algorithm relied on a brute force solution, although impressive, hardly “intelligent” by any measure.

1.11 Parallelism: Enabler of Machine Intelligence

- Any definition of true machine “intelligence” would have to include the ability to acquire new knowledge independent of direct human intervention, and the ability to solve problems using incomplete and perhaps contradictory information.
- This is precisely what IBM achieved when it build the machine named Watson.
- Watson proved this when it beat two human *Jeopardy!* champions on February 16, 2011.

1.11 Parallelism: Enabler of Machine Intelligence

- Watson had a massively parallel architecture dubbed *DeepQA* (Deep Question and Answer).
- The system relied on 90 IBM POWER 750 servers.
- Each server was equipped with four POWER7 processors, and each POWER7 processor had eight cores, giving a total of 2880 processor cores.
- While playing Jeopardy!, each core had access to 16TB of main memory and 4TB of storage.

1.11 Parallelism: Enabler of Machine Intelligence

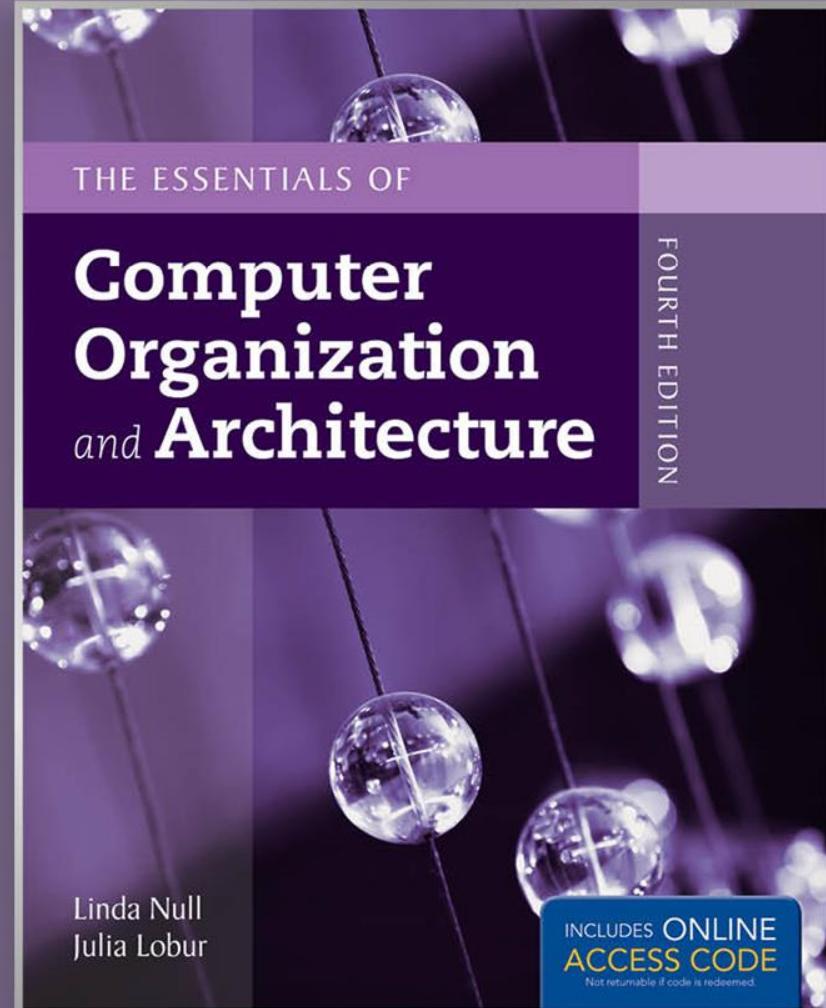
- Watson's technology has been put to work in treating cancer.
 - Commercial products based on Watson technology, including “Interactive Care Insights for Oncology” and “Interactive Care Reviewer,” are now available.
- Watson is also becoming more compact: Watson can now be run on a single POWER 750 server.
- Watson has surely given us a glimpse into the future of computing.

Conclusion

- This chapter has given you an overview of the subject of computer architecture.
- You should now be sufficiently familiar with general system structure to guide your studies throughout the remainder of this course.
- Subsequent chapters will explore many of these topics in great detail.

Chapter 3.6-

Sequential Circuits

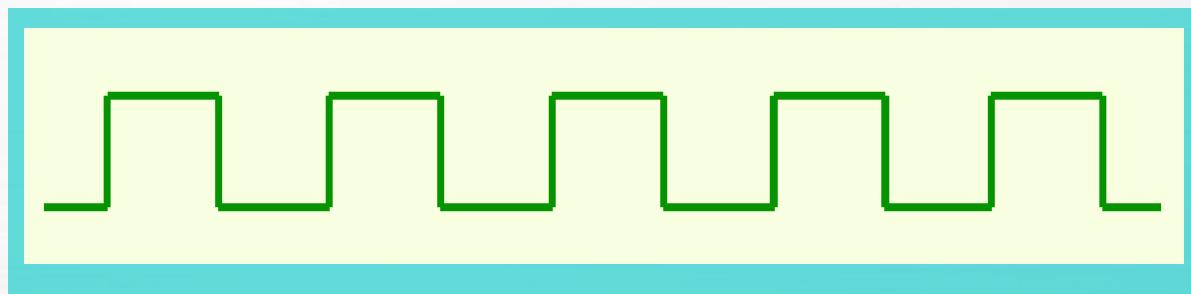


3.6 Sequential Circuits

- Combinational logic circuits are perfect for situations when we require the immediate application of a Boolean function to a set of inputs.
- There are other times, however, when we need a circuit to change its value with consideration to **its current state** as well as **its inputs**.
 - These circuits have to “remember” their current state.
- *Sequential logic circuits* provide this functionality for us.

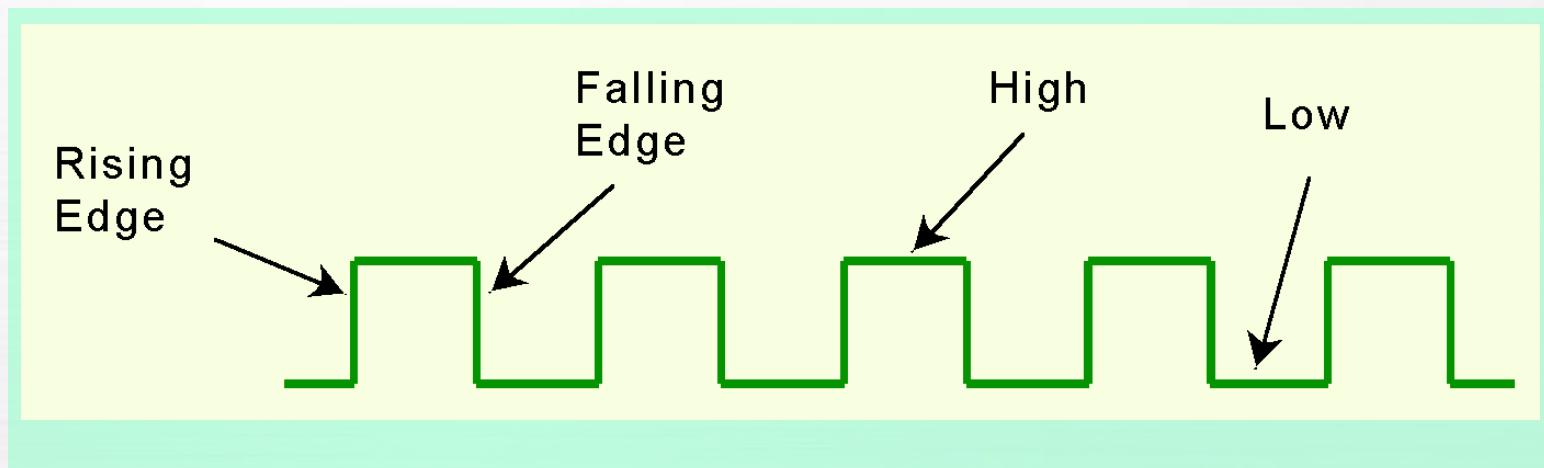
3.6 Sequential Circuits

- As the name implies, sequential logic circuits require a means by which events can be sequenced.
- State changes are controlled by clocks.
 - A “clock” is a special circuit that sends electrical pulses through a circuit.
- Clocks produce electrical waveforms such as the one shown below.



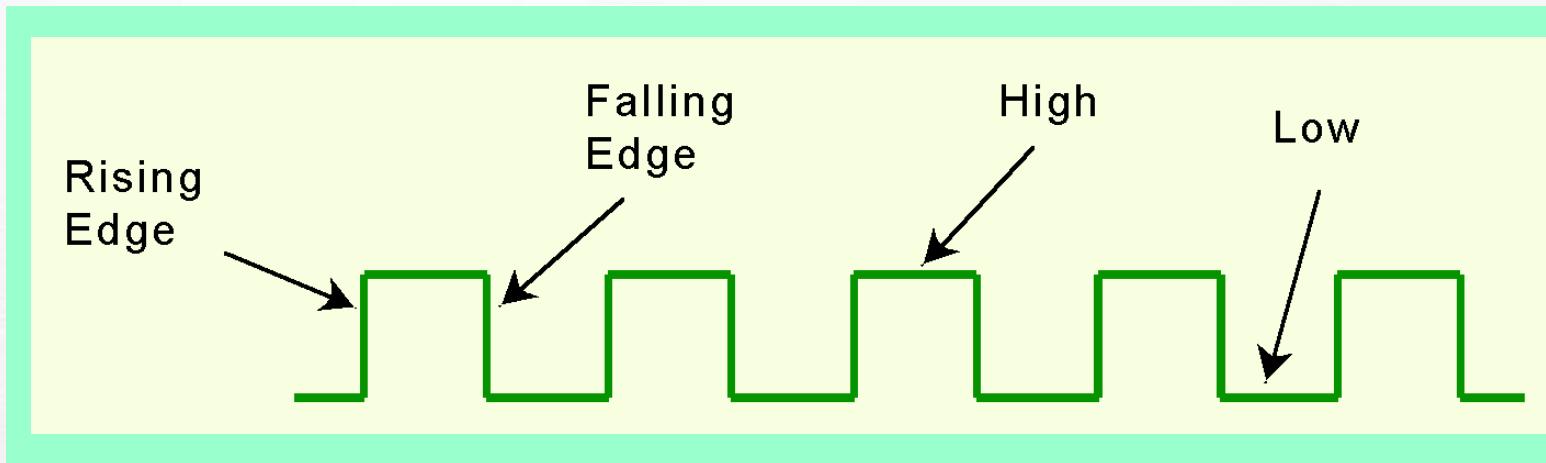
3.6 Sequential Circuits

- State changes occur in sequential circuits only when the clock ticks.
 - **Synchronous** sequential circuits
- Circuits can change state on the **rising edge**, **falling edge**, or when the clock pulse reaches its **highest voltage**.



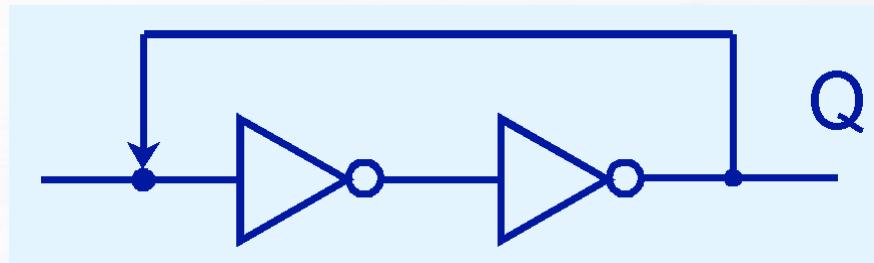
3.6 Sequential Circuits

- Circuits that change state on the rising edge, or falling edge of the clock pulse are called **edge-triggered**.
- **Level-triggered circuits** change state when the clock voltage reaches its highest or lowest level.



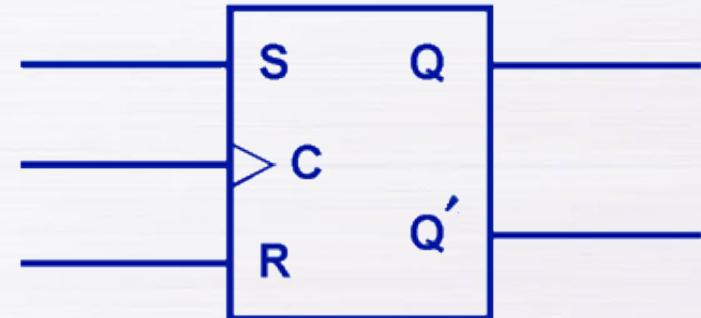
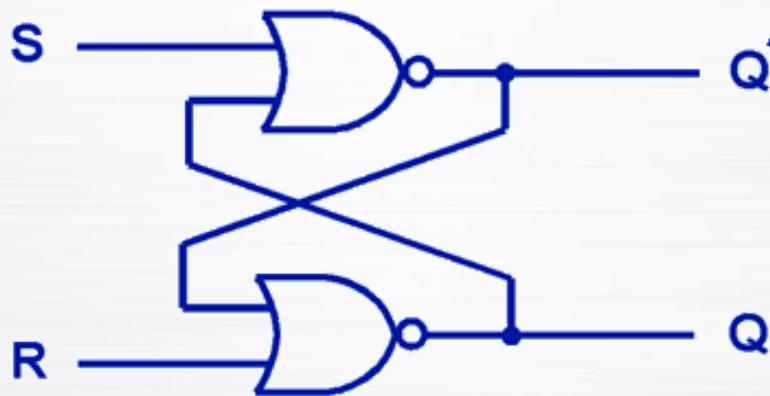
3.6 Sequential Circuits

- To retain their state values, sequential circuits rely on *feedback*.
- Feedback in digital circuits occurs when an output is looped back to the input.
- A simple example of this concept is shown below.
 - If Q is 0 it will always be 0, if it is 1, it will always be 1. Why?



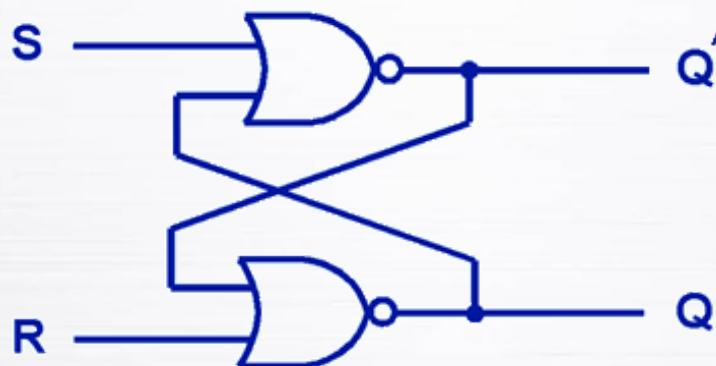
3.6 Sequential Circuits

- You can see how feedback works by examining the most basic sequential logic components, the SR flip-flop.
 - The “SR” stands for set/reset.
- The internals of an SR flip-flop are shown below, along with its block diagram.



3.6 Sequential Circuits

- The behavior of an SR flip-flop is described by a characteristic table.
- $Q(t)$ means the value of the output at time t .
- $Q(t+1)$ is the value of Q after the next clock pulse.



S	R	$Q(t+1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	undefined

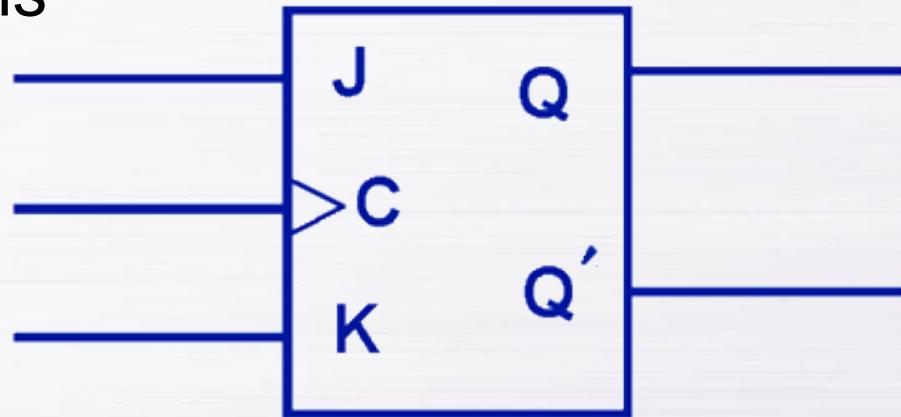
3.6 Sequential Circuits

- The SR flip-flop actually has three inputs: S, R, and its current output, Q.
- Thus, we can construct a truth table for this circuit, as shown at the right.
- Notice the two undefined values. When both S and R are 1, the SR flip-flop is unstable.

Present State			Next State
S	R	$Q(t)$	$Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	undefined
1	1	1	undefined

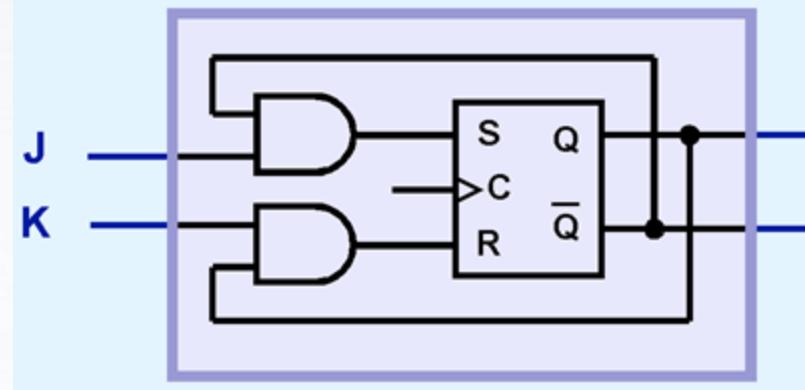
3.6 Sequential Circuits

- If we can be sure that the inputs to an SR flip-flop will never both be 1, we will never have an unstable circuit. This may not always be the case.
- The SR flip-flop can be modified to provide a stable state when both inputs are 1.
- This modified flip-flop is called a JK flip-flop, shown at the right.



3.6 Sequential Circuits

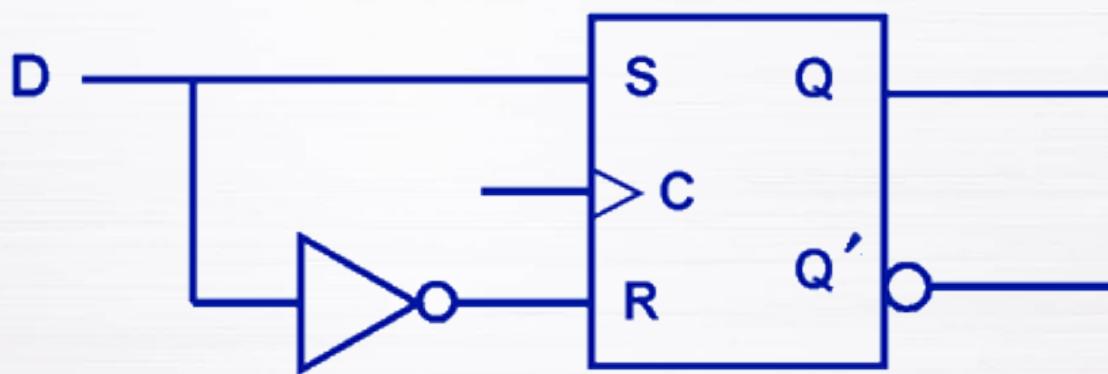
- At the right, we see how an SR flip-flop can be modified to create a JK flip-flop.
- The characteristic table indicates that the flip-flop is stable for all inputs.



J	K	$Q(t+1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}(t)$

3.6 Sequential Circuits

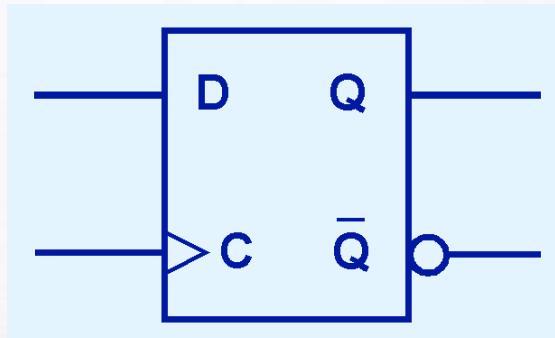
- Another modification of the SR flip-flop is the D flip-flop, shown below with its characteristic table.
- You will notice that the output of the flip-flop remains the same during subsequent clock pulses. The output changes only when the value of D changes.



D	$Q(t+1)$
0	0
1	1

3.6 Sequential Circuits

- The D flip-flop is the fundamental circuit of computer memory.
 - D flip-flops are usually illustrated using the block diagram shown below.
- The characteristic table for the D flip-flop is shown at the right.



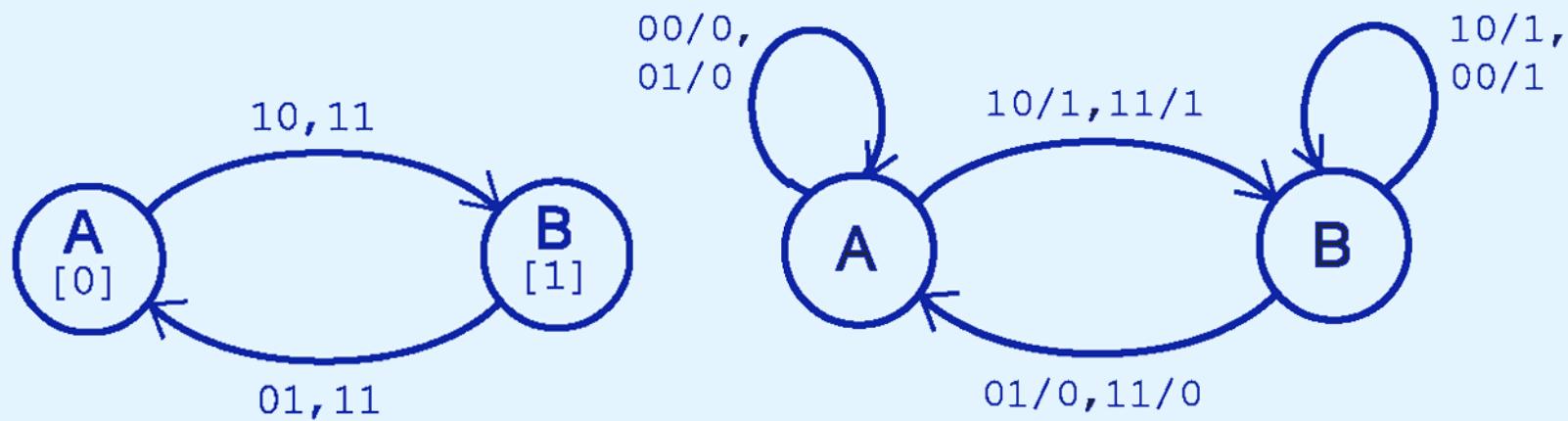
D	$Q(t+1)$
0	0
1	1

3.6 Sequential Circuits

- The behavior of sequential circuits can be expressed using characteristic tables or finite state machines (FSMs).
 - FSMs consist of a set of nodes that hold the states of the machine and a set of arcs that connect the states.
- Moore and Mealy machines are two types of FSMs that are equivalent.
 - They differ only in how they express the outputs of the machine.
- Moore machines place outputs on each node, while Mealy machines present their outputs on the transitions.

3.6 Sequential Circuits

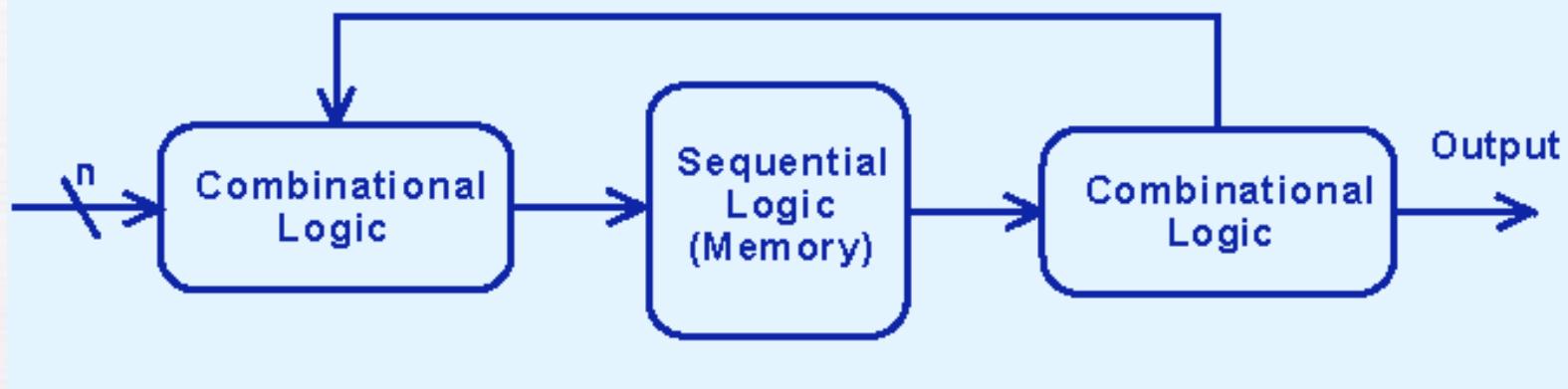
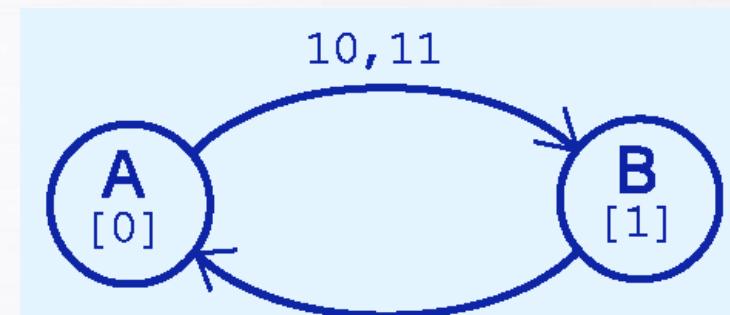
- The behavior of a JK flop-flop is depicted below by a Moore machine (left) and a Mealy machine (right).



3.6 Sequential Circuits

- Although the behavior of Moore and Mealy machines is identical, their implementations differ.

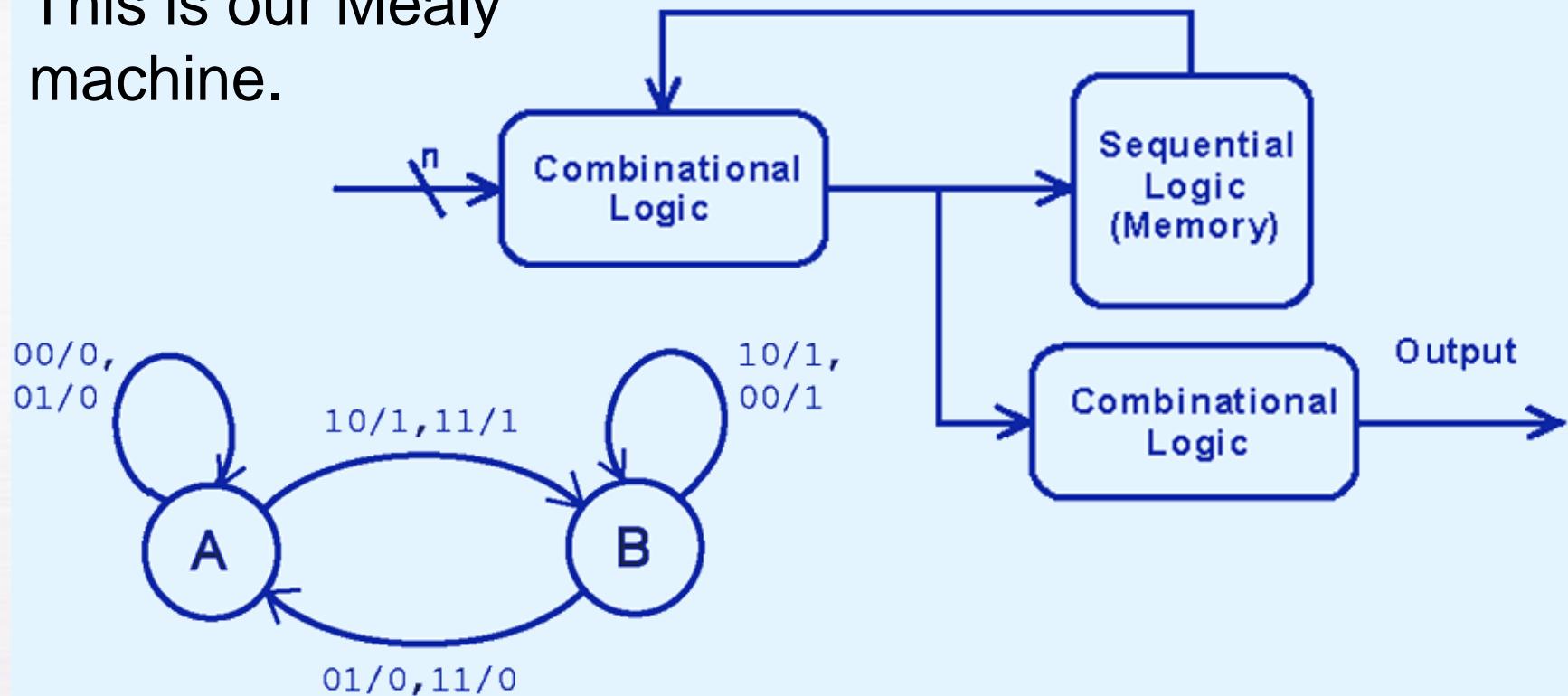
This is our Moore machine.



3.6 Sequential Circuits

- Although the behavior of Moore and Mealy machines is identical, their implementations differ.

This is our Mealy machine.

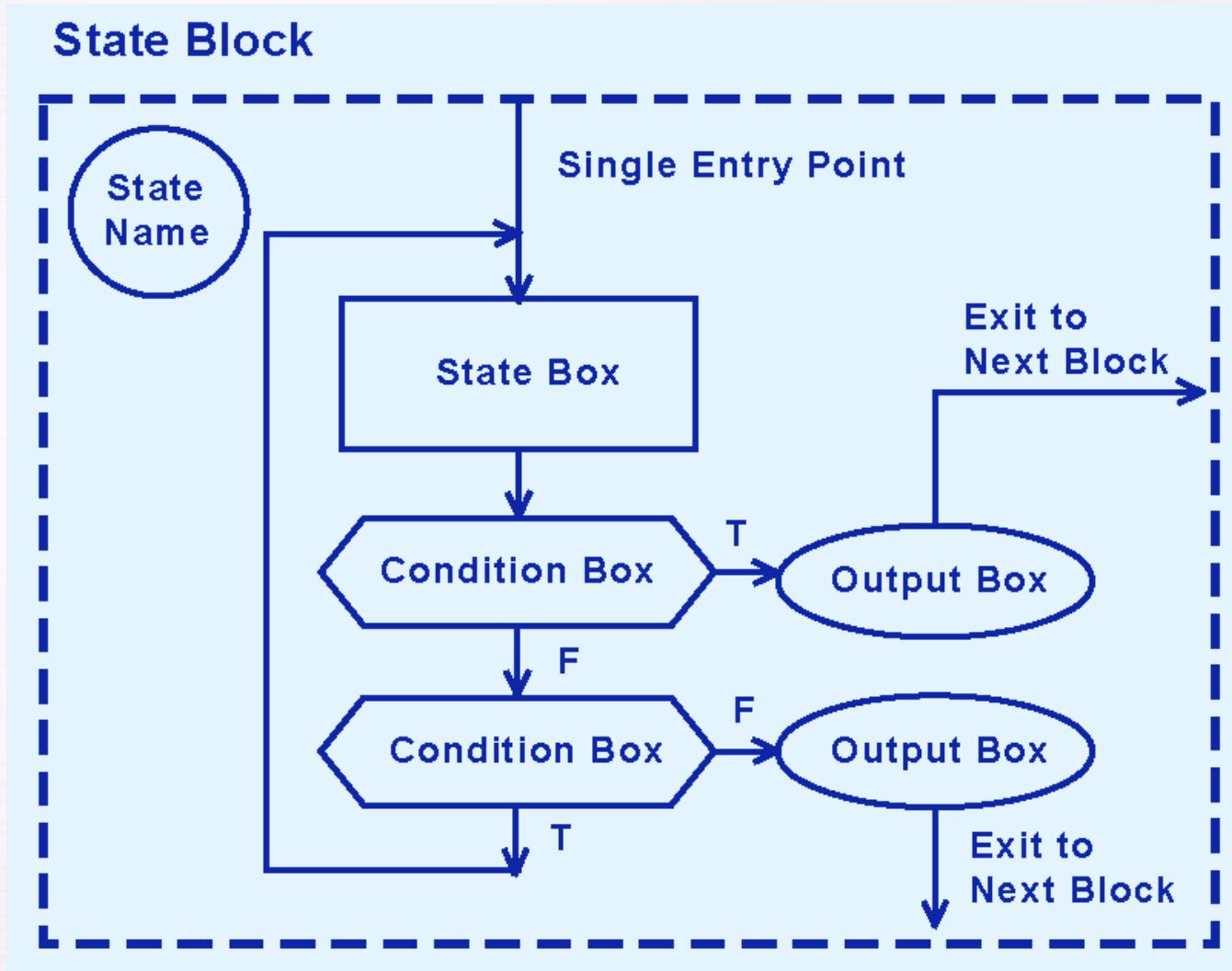


3.6 Sequential Circuits

- It is difficult to express the complexities of actual implementations using only Moore and Mealy machines.
 - For one thing, they do not address the intricacies of timing very well.
 - Secondly, it is often the case that an interaction of numerous signals is required to advance a machine from one state to the next.
- For these reasons, Christopher Clare invented the algorithmic state machine (ASM).

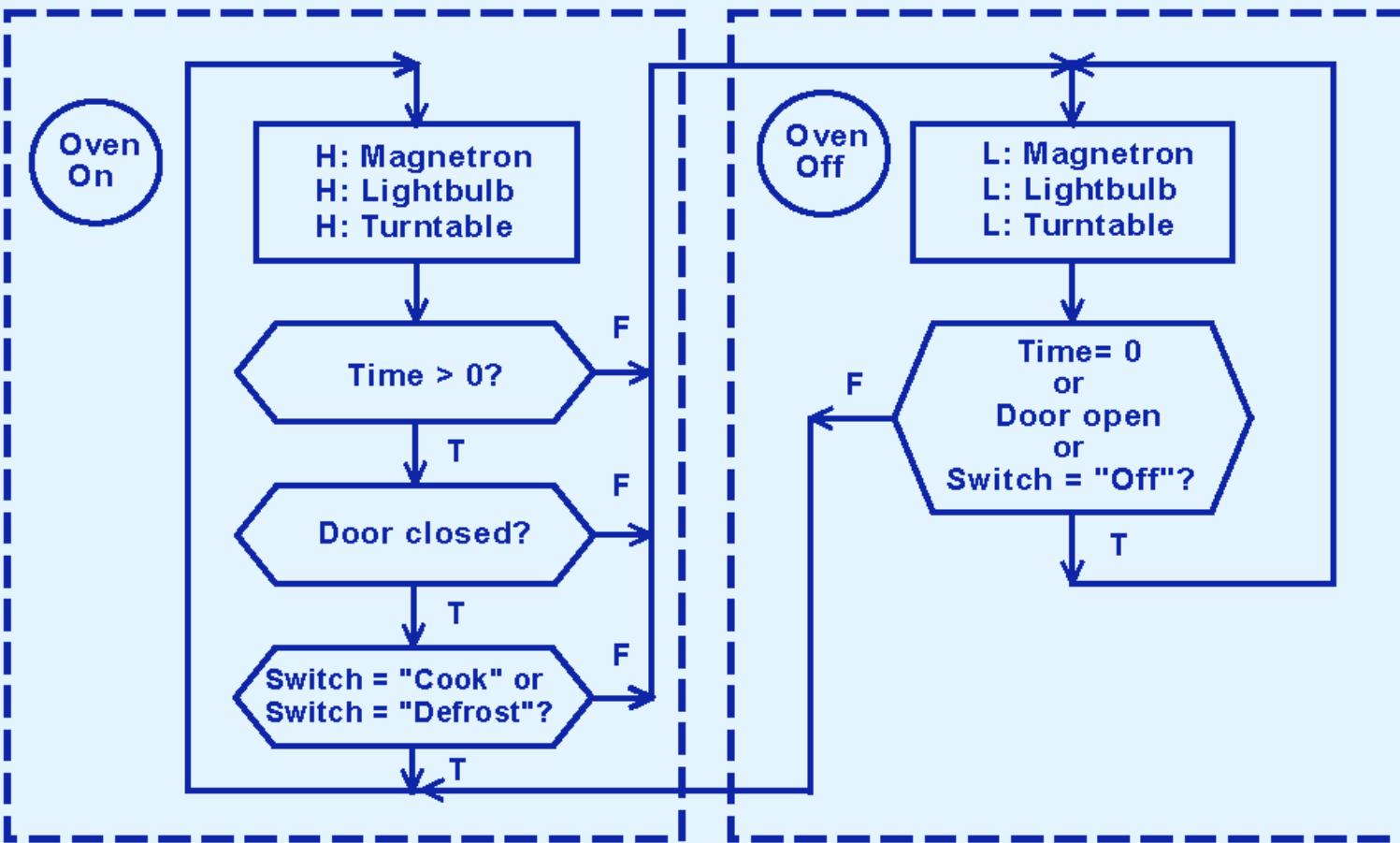
The next slide illustrates the components of an ASM.

3.6 Sequential Circuits



3.6 Sequential Circuits

- This is an ASM for a microwave oven.



3.6 Sequential Circuits

- Sequential circuits are used anytime that we have a “stateful” application.
 - A stateful application is one where the next state of the machine depends on the current state of the machine and the input.
- A stateful application requires both combinational and sequential logic.
- The following slides provide several examples of circuits that fall into this category.

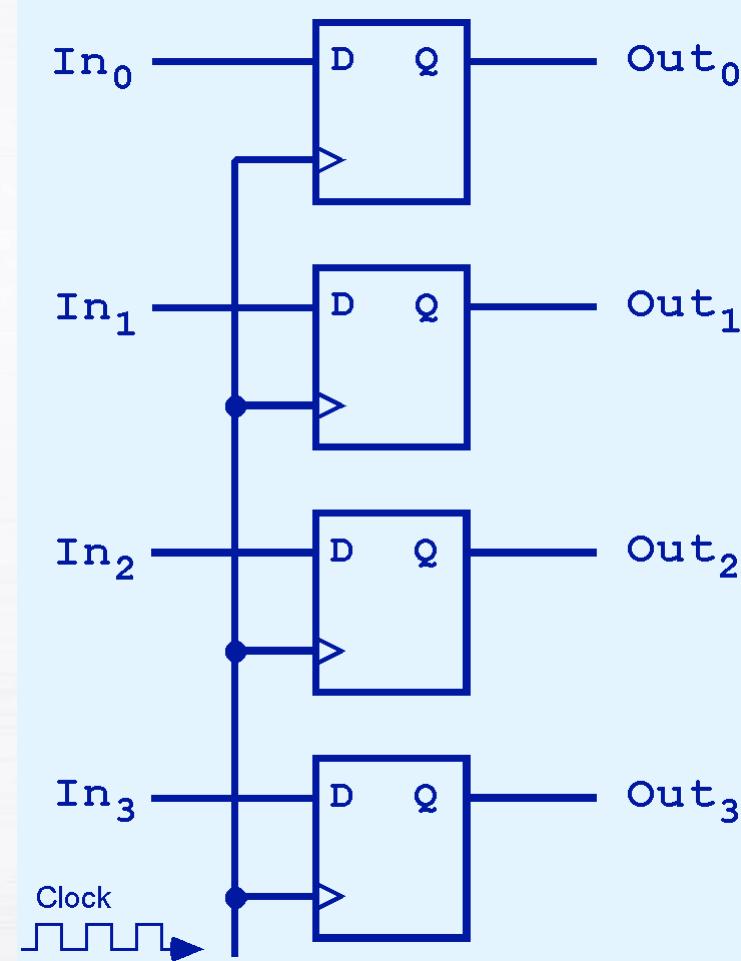
Can you think of others?

3.6 Sequential Circuits

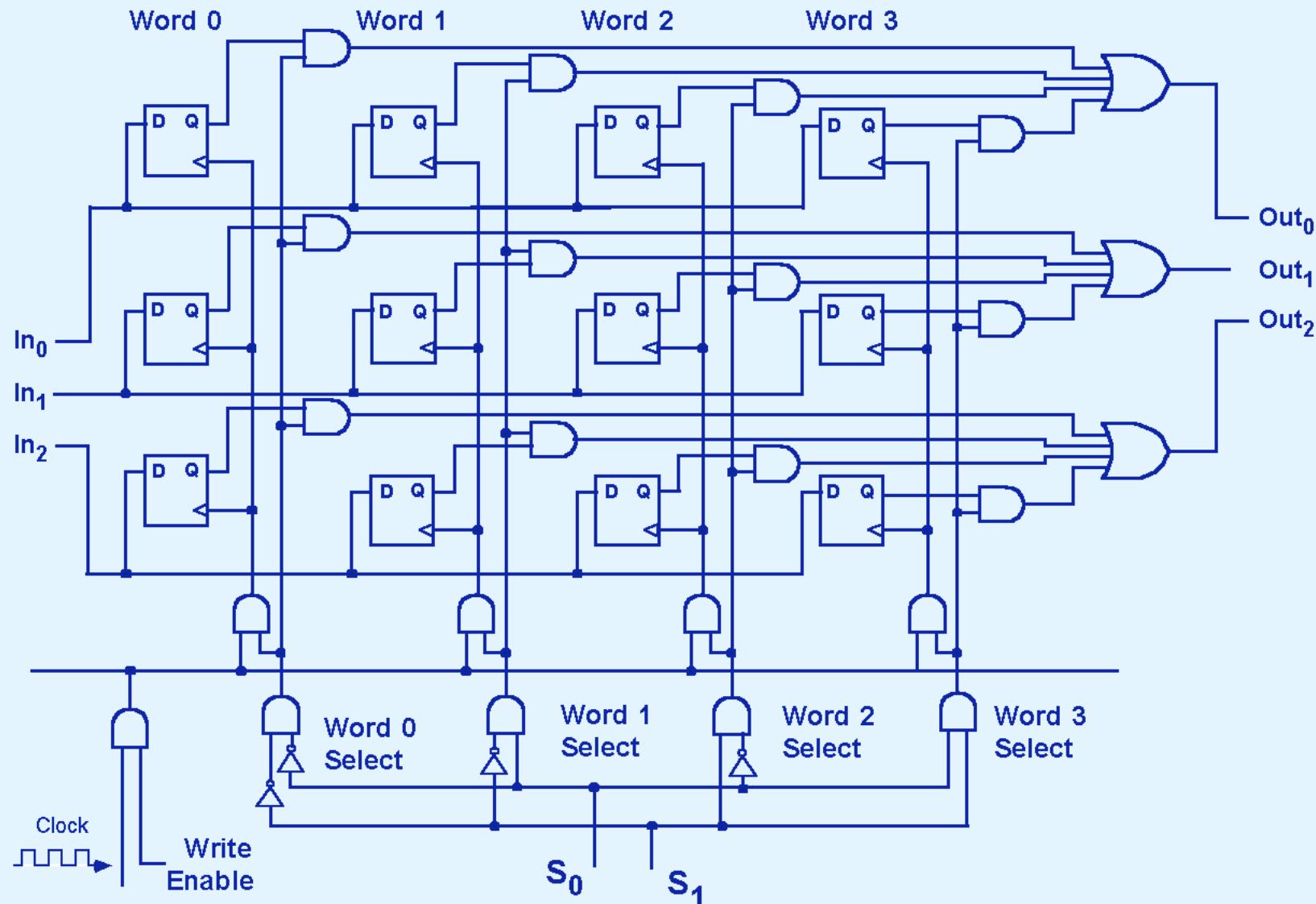
- This illustration shows a 4-bit register consisting of D flip-flops. You will usually see its block diagram (below) instead.



A larger memory configuration is shown on the next slide.

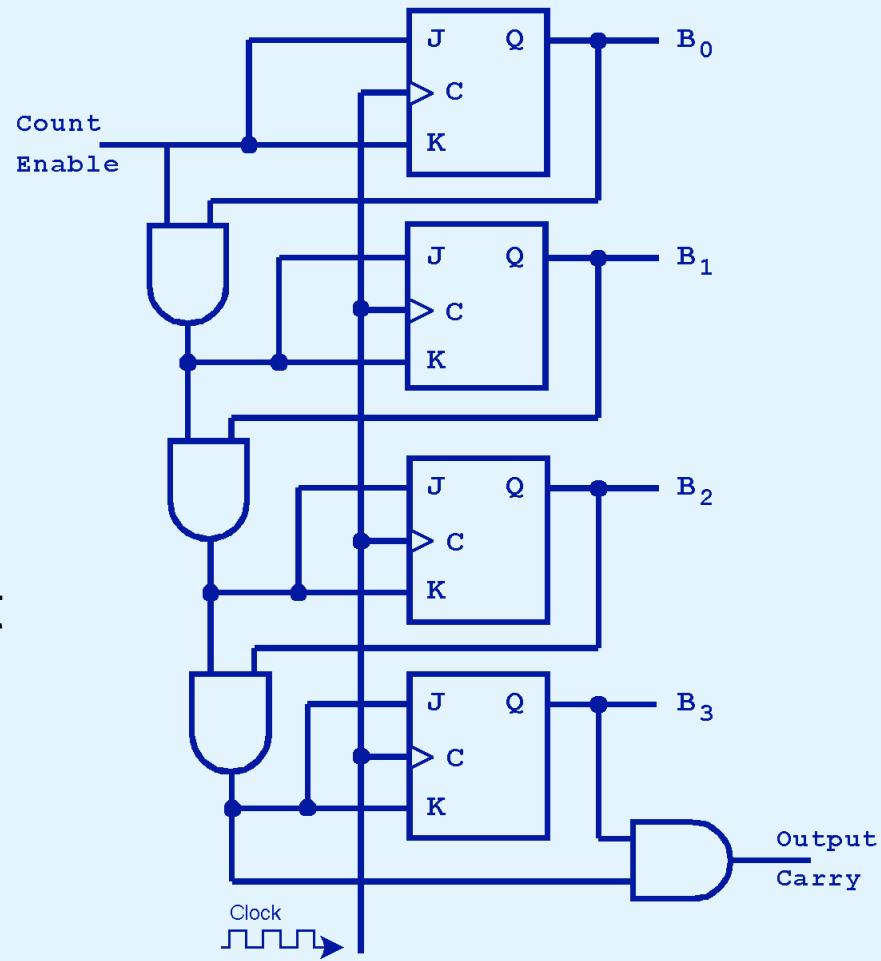


3.6 Sequential Circuits



3.6 Sequential Circuits

- A binary counter is another example of a sequential circuit.
- The low-order bit is complemented at each clock pulse.
- Whenever it changes from 0 to 1, the next bit is complemented, and so on through the other flip-flops.

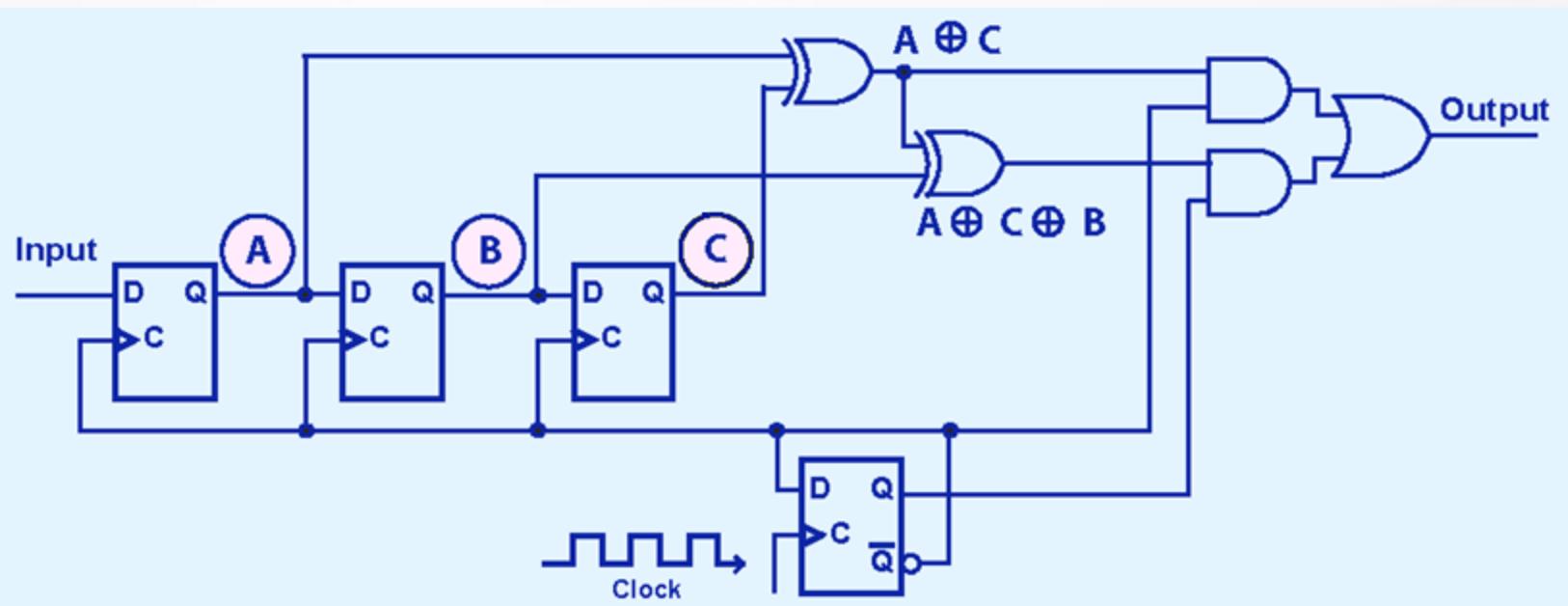


3.6 Sequential Circuits

- Convolutional coding and decoding requires sequential circuits.
- One important convolutional code is the $(2,1)$ convolutional code that underlies the PRML code that is briefly described at the end of Chapter 2.
- A $(2, 1)$ convolutional code is so named because two symbols are output for every one symbol input.
- A convolutional encoder for PRML with its characteristic table is shown on the next slide.

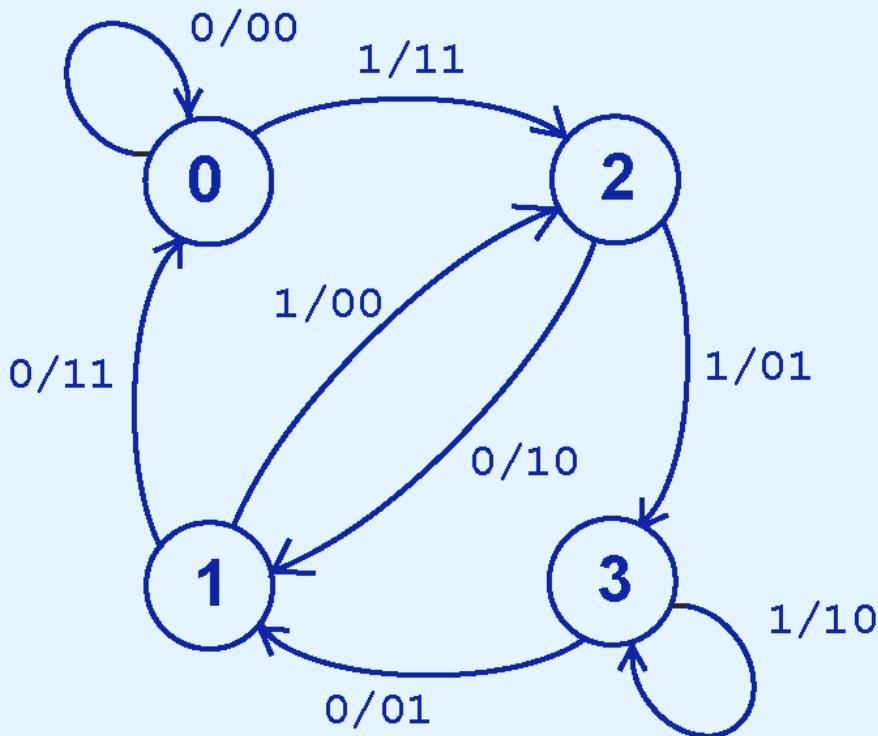
3.6 Sequential Circuits

Input A	Current State B C	Next State B C	Output	Input A	Current State B C	Next State B C	Output
0	00	00	00	0	10	01	10
1	00	10	11	1	10	11	01
0	01	00	11	0	11	01	01
1	01	10	00	1	11	11	10



3.6 Sequential Circuits

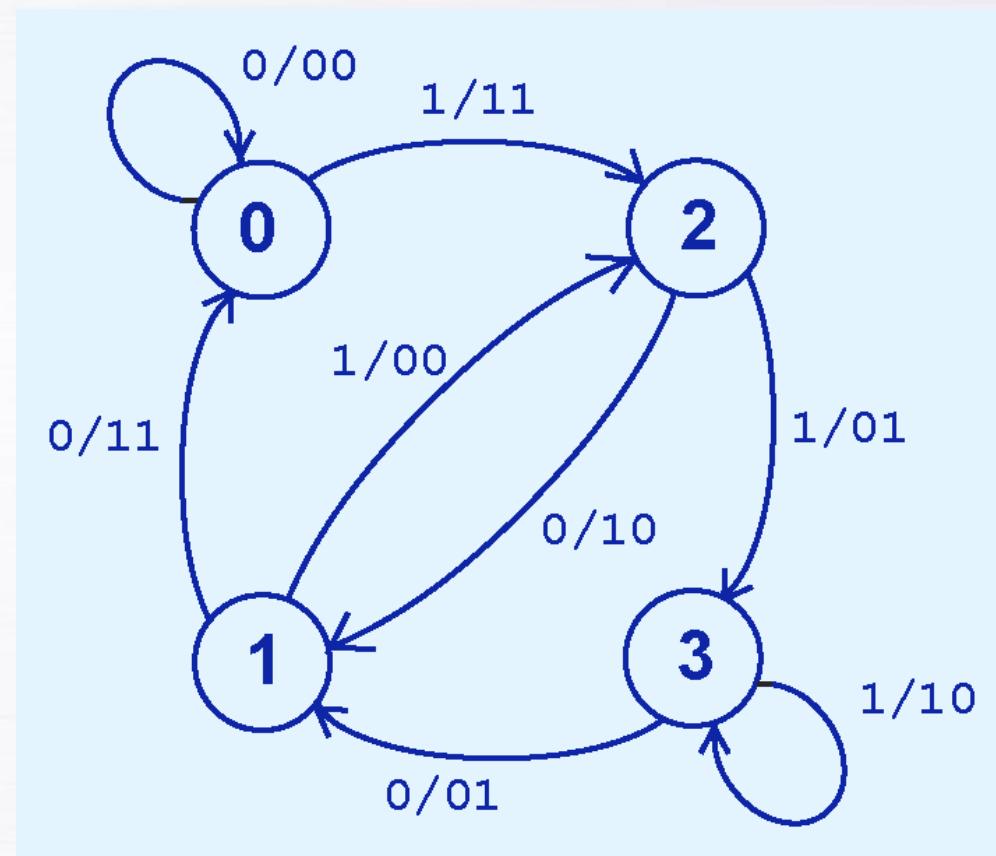
This is the Mealy machine for our encoder.



Input A	Current State B C	Next State B C	Output
0	00	00	00
1	00	10	11
0	01	00	11
1	01	10	00
0	10	01	10
1	10	11	01
0	11	01	01
1	11	11	10

3.6 Sequential Circuits

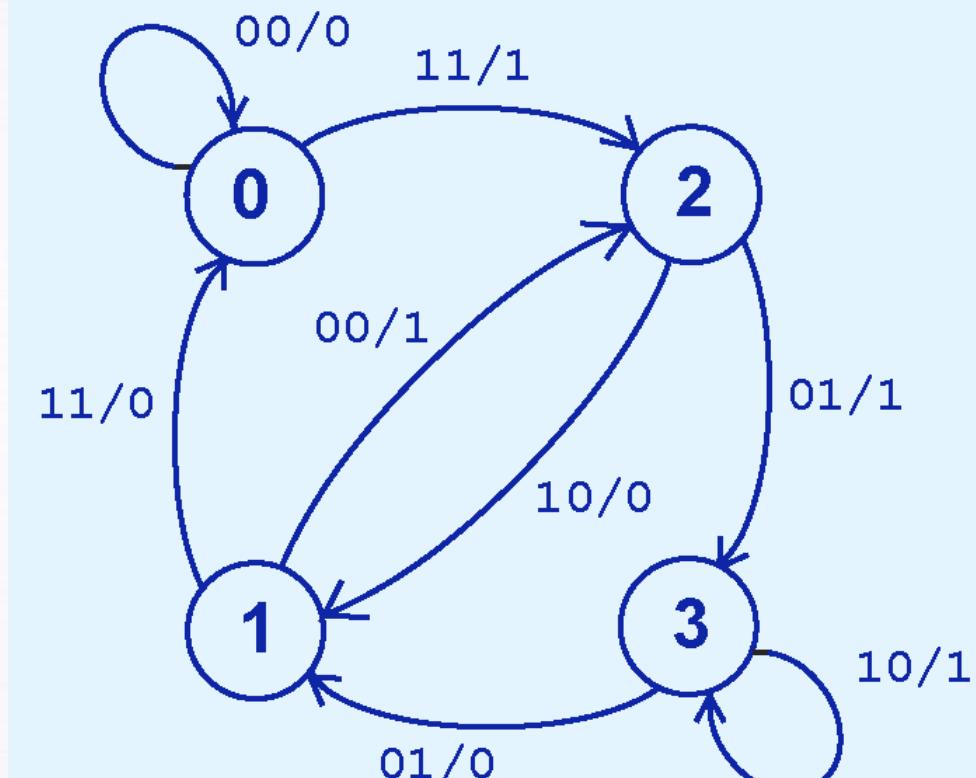
- The fact that there is a limited set of possible state transitions in the encoding process is crucial to the error correcting capabilities of PRML.
- You can see by our Mealy machine for encoding that:



$$F(1101 \ 0010) = 11 \ 01 \ 01 \ 00 \ 10 \ 11 \ 11 \ 10.$$

3.6 Sequential Circuits

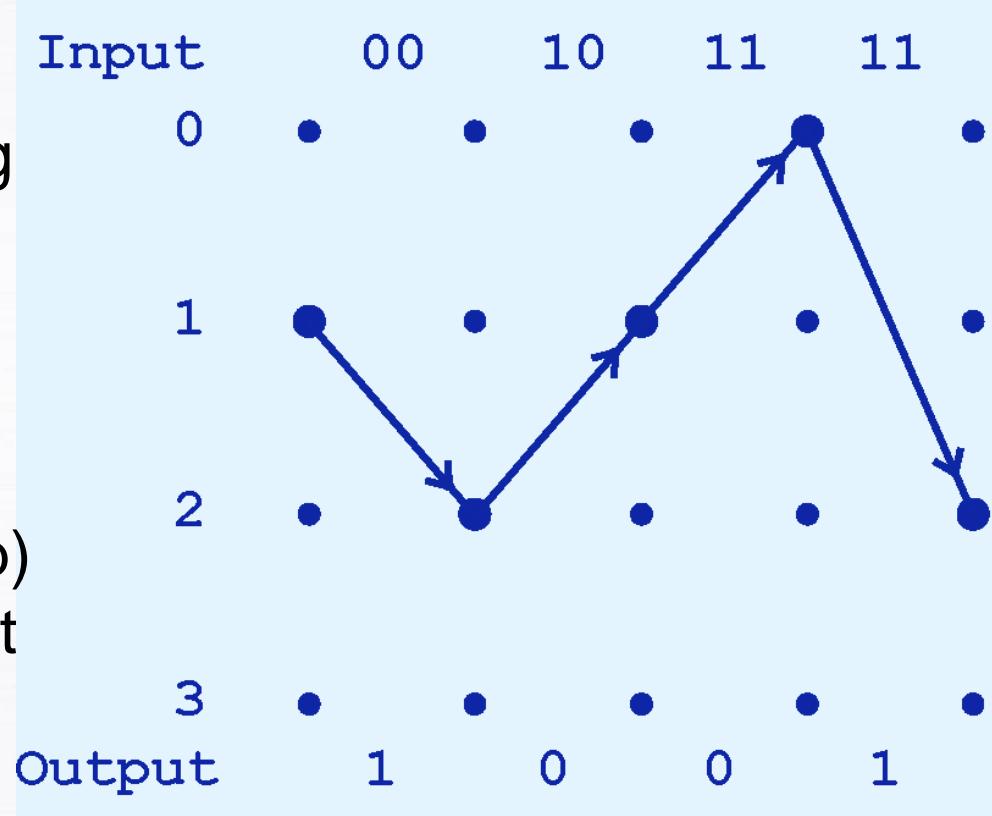
- The decoding of our code is provided by inverting the inputs and outputs of the Mealy machine for the encoding process.
- You can see by our Mealy machine for decoding that:



$$F(11 \ 01 \ 01 \ 00 \ 10 \ 11 \ 11 \ 10) = 1101 \ 0010$$

3.6 Sequential Circuits

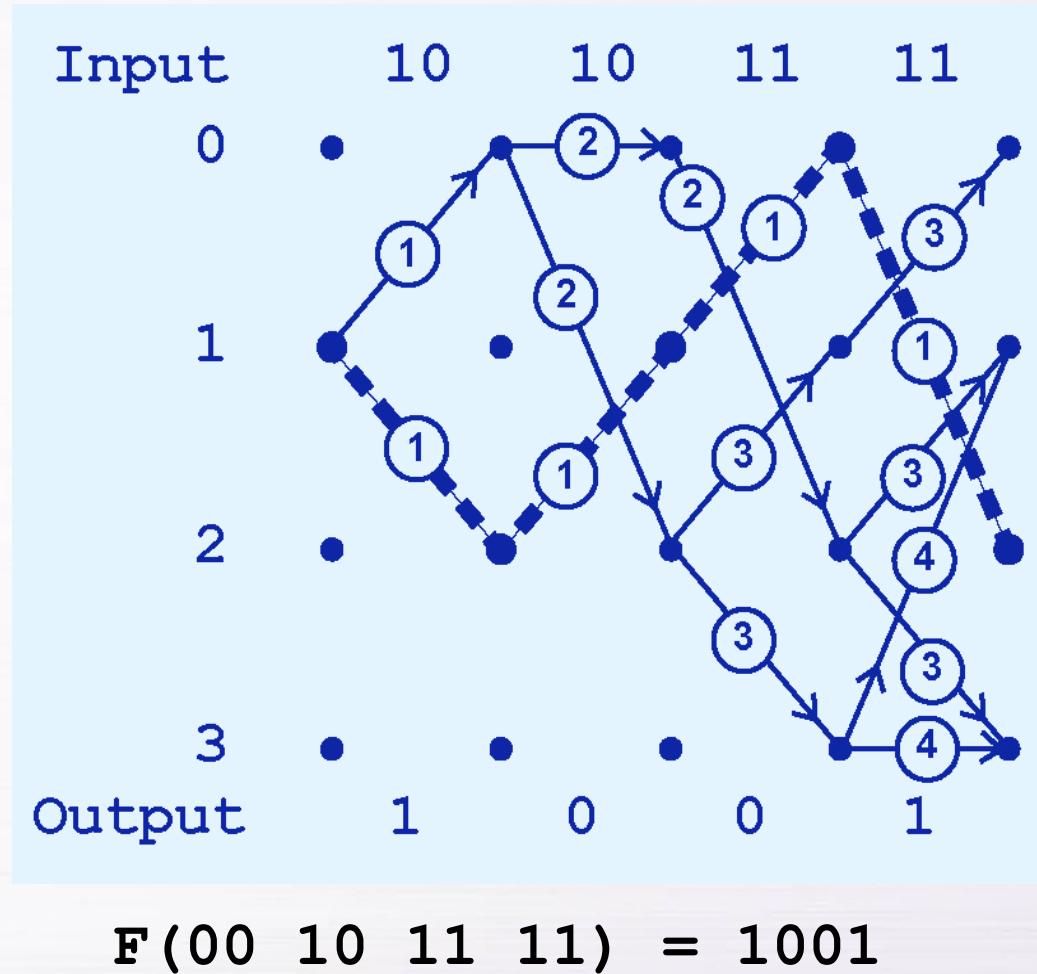
- Yet another way of looking at the decoding process is through a lattice diagram.
- Here we have plotted the state transitions based on the input (top) and showing the output at the bottom for the string 00 10 11 11.



$$F(00 \ 10 \ 11 \ 11) = 1001$$

3.6 Sequential Circuits

- Suppose we receive the erroneous string: 10 10 11 11.
- Here we have plotted the accumulated errors based on the allowable transitions.
- The path of least error outputs 1001, thus 1001 is the string of maximum likelihood.



3.7 Designing Circuits

- We have seen digital circuits from two points of view: digital analysis and digital synthesis.
 - *Digital analysis* explores the relationship between a circuit's inputs and its outputs.
 - *Digital synthesis* creates logic diagrams using the values specified in a truth table.
- Digital systems designers must also be mindful of the physical behaviors of circuits to include minute propagation delays that occur between the time when a circuit's inputs are energized and when the output is accurate and stable.

3.7 Designing Circuits

- Digital designers rely on specialized software to create efficient circuits.
 - Thus, software is an enabler for the construction of better hardware.
- Of course, software is in reality a collection of algorithms that could just as well be implemented in hardware.
 - Recall the Principle of Equivalence of Hardware and Software.

3.7 Designing Circuits

- When we need to implement a simple, specialized algorithm and its execution speed must be as fast as possible, a hardware solution is often preferred.
- This is the idea behind *embedded systems*, which are small special-purpose computers that we find in many everyday things.
- Embedded systems require special programming that demands an understanding of the operation of digital circuits, the basics of which you have learned in this chapter.

Chapter 3 Conclusion

- Computers are implementations of Boolean logic.
- Boolean functions are completely described by truth tables.
- Logic gates are small circuits that implement Boolean operators.
- The basic gates are AND, OR, and NOT.
 - The XOR gate is very useful in parity checkers and adders.
- The “universal gates” are NOR, and NAND.

Chapter 3 Conclusion

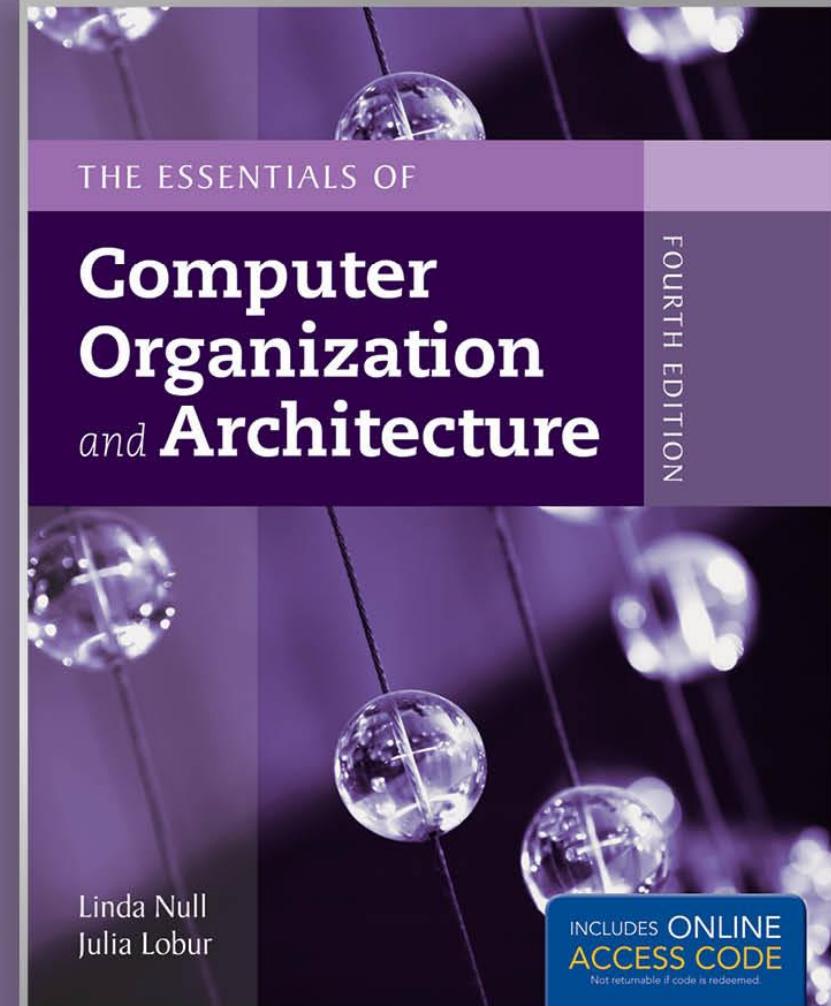
- Computer circuits consist of combinational logic circuits and sequential logic circuits.
- Combinational circuits produce outputs (almost) immediately when their inputs change.
- Sequential circuits require clocks to control their changes of state.
- The basic sequential circuit unit is the flip-flop: The behaviors of the SR, JK, and D flip-flops are the most important to know.

Chapter 3 Conclusion

- The behavior of sequential circuits can be expressed using characteristic tables or through various finite state machines.
- Moore and Mealy machines are two finite state machines that model high-level circuit behavior.
- Algorithmic state machines are better than Moore and Mealy machines at expressing timing and complex signal interactions.
- Examples of sequential circuits include memory, counters, and Viterbi encoders and decoders.

Chapter 4

MARIE: An Introduction to a Simple Computer



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.

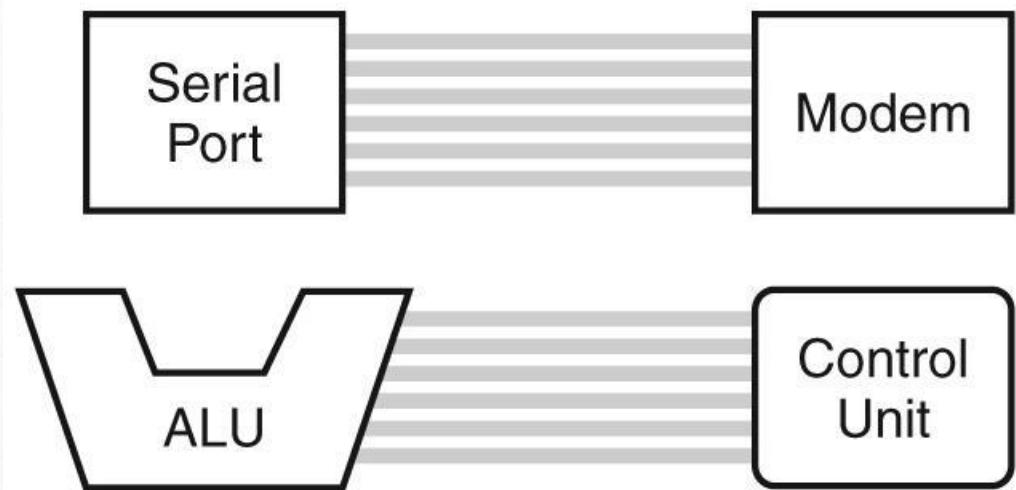
4.2 CPU Basics

- 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:

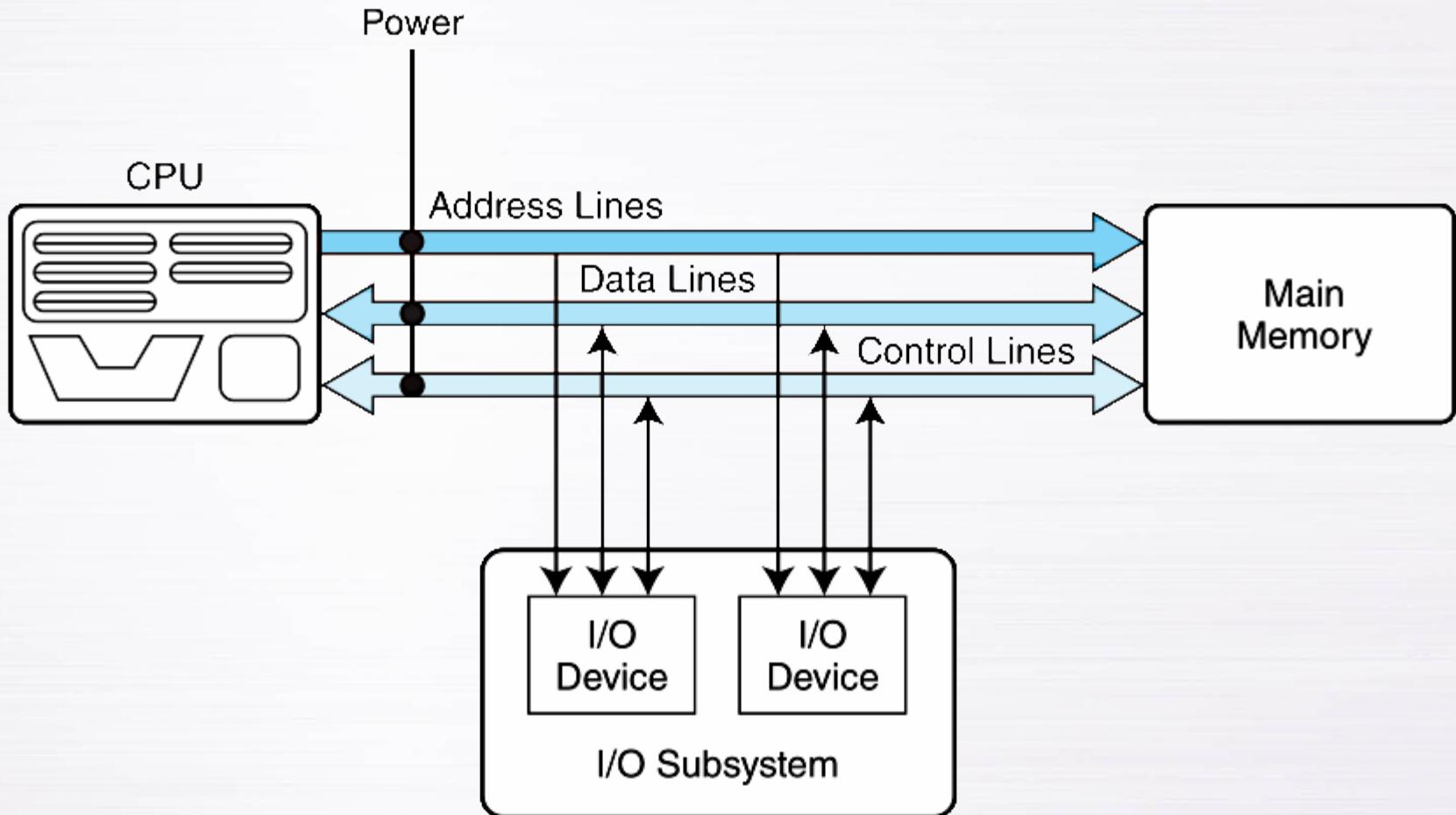


4.3 The Bus

- 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.

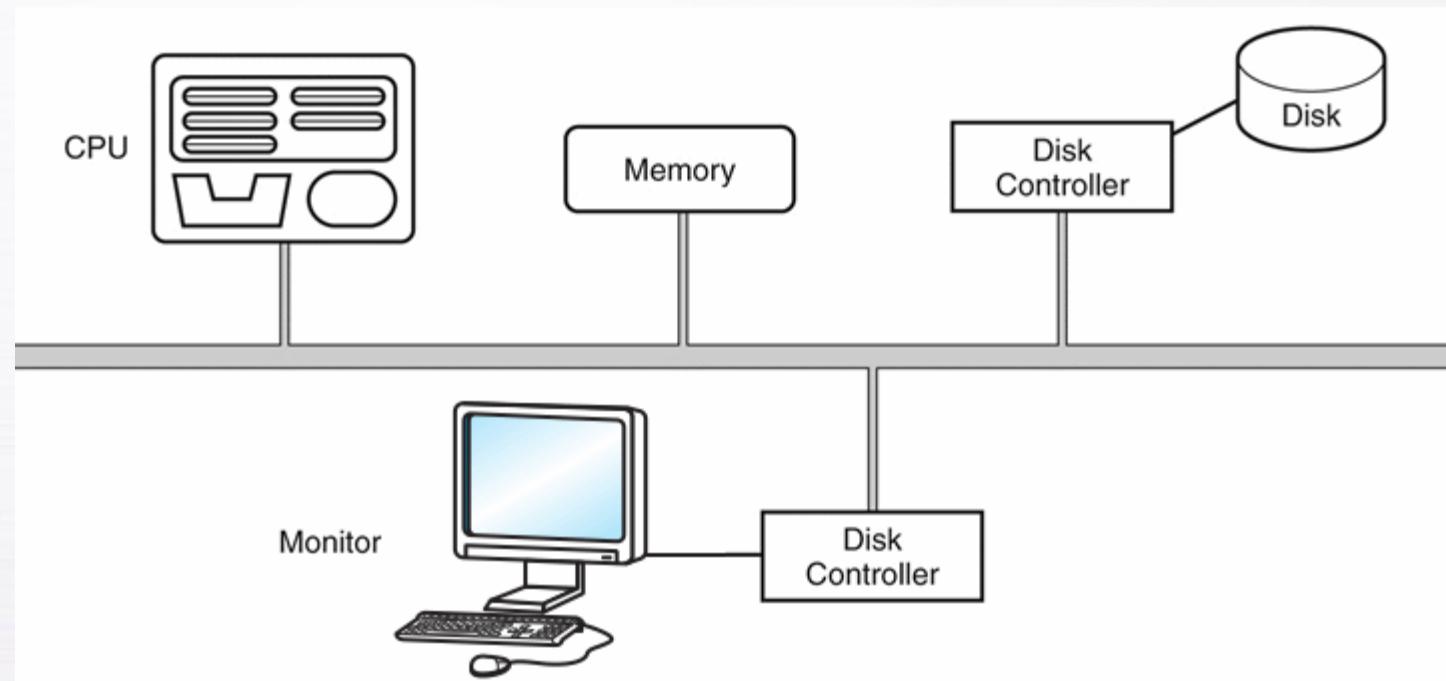
The next slide shows a model bus configuration.

4.3 The Bus



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

- 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.

4.4 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 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.

4.5 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.
- 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.

4.6 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 out 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 megabytes of 16-bit memory locations.

4.6 Memory Organization

- How does the computer access a memory location corresponding 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 bit address lines.

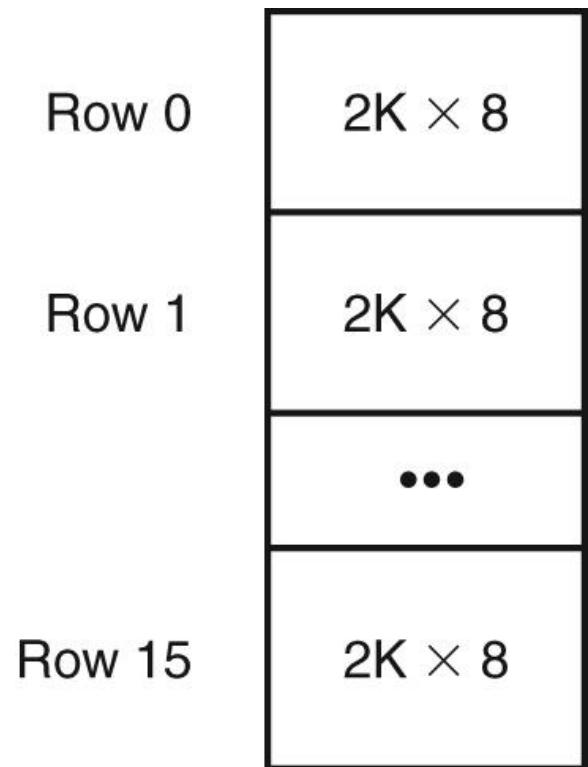
4.6 Memory Organization

- 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.

The next two slides illustrate these two ideas.

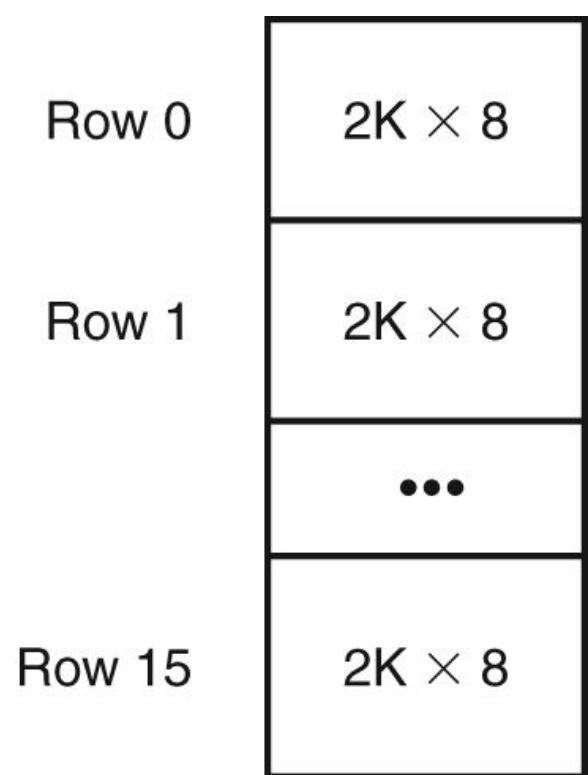
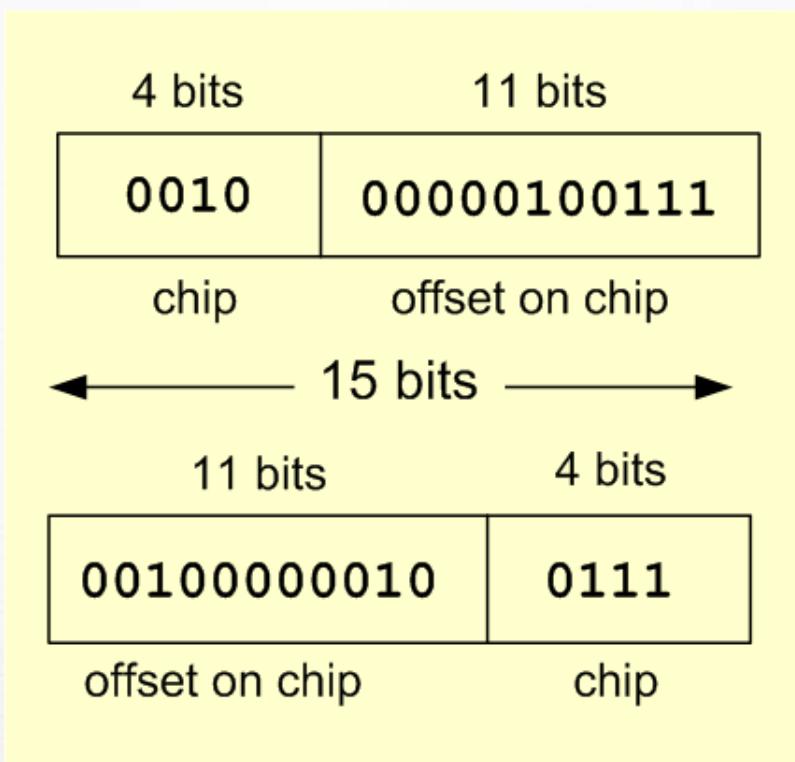
4.6 Memory Organization

- 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.

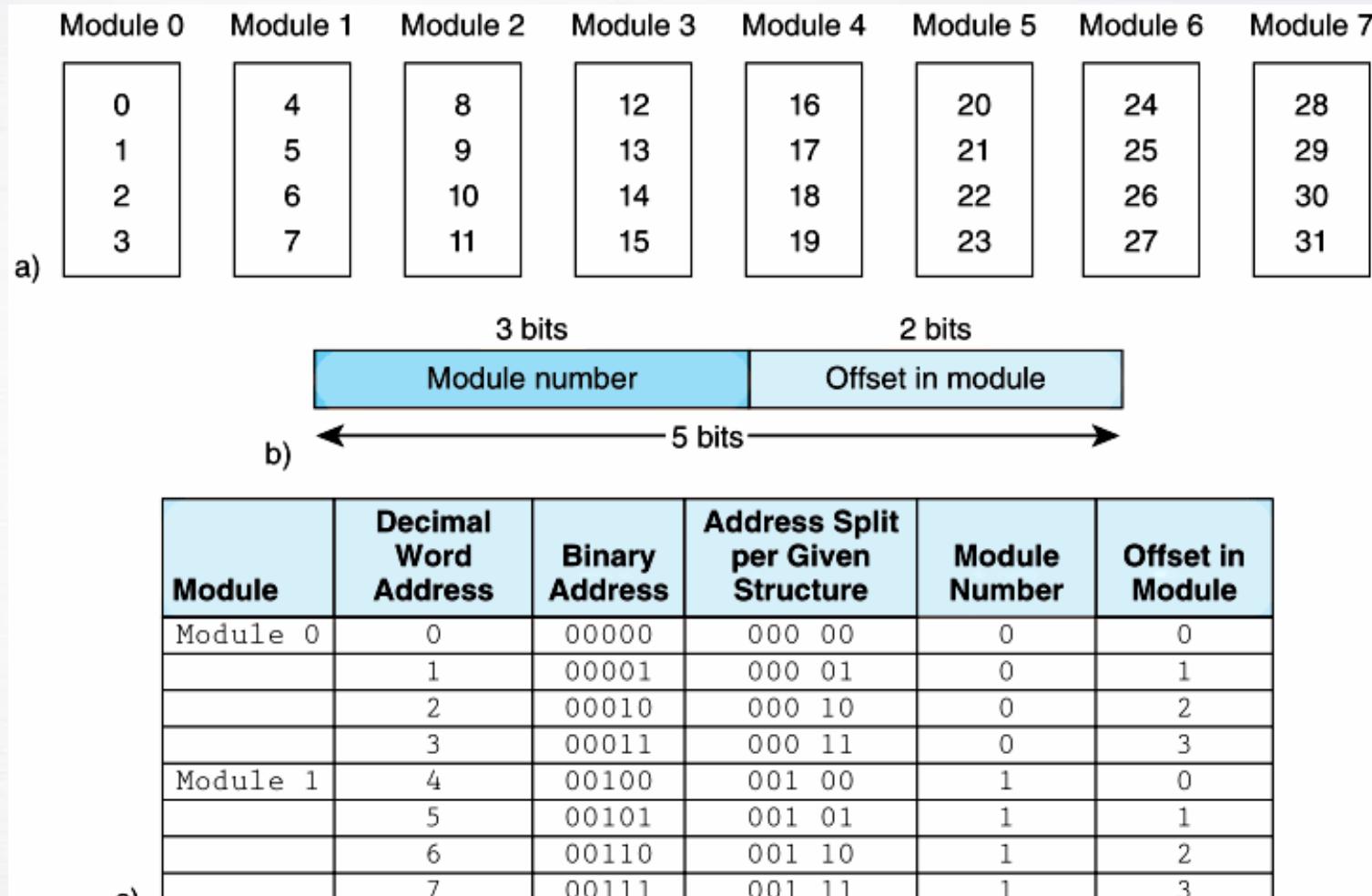


4.6 Memory Organization

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

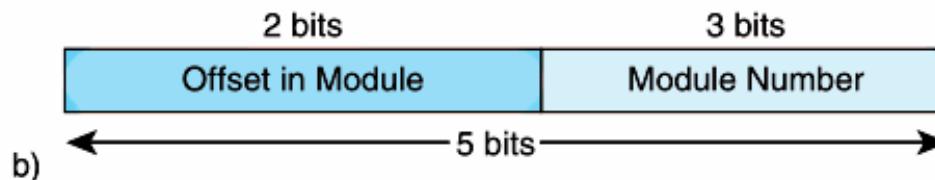
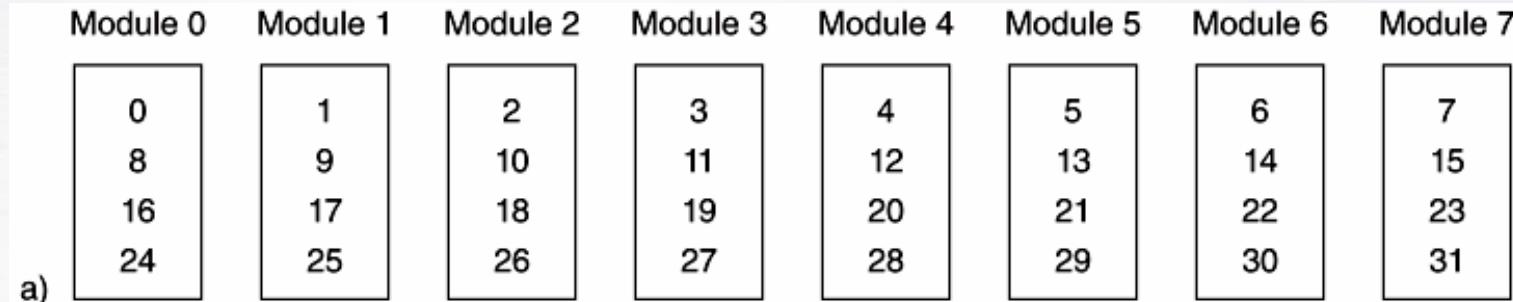


4.6 Memory Organization



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

4.6 Memory Organization



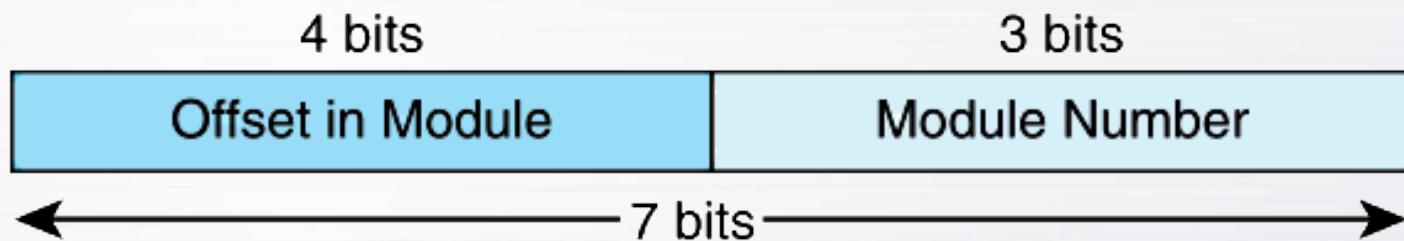
c)

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

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

4.6 Memory Organization

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



4.7 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), 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.

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.

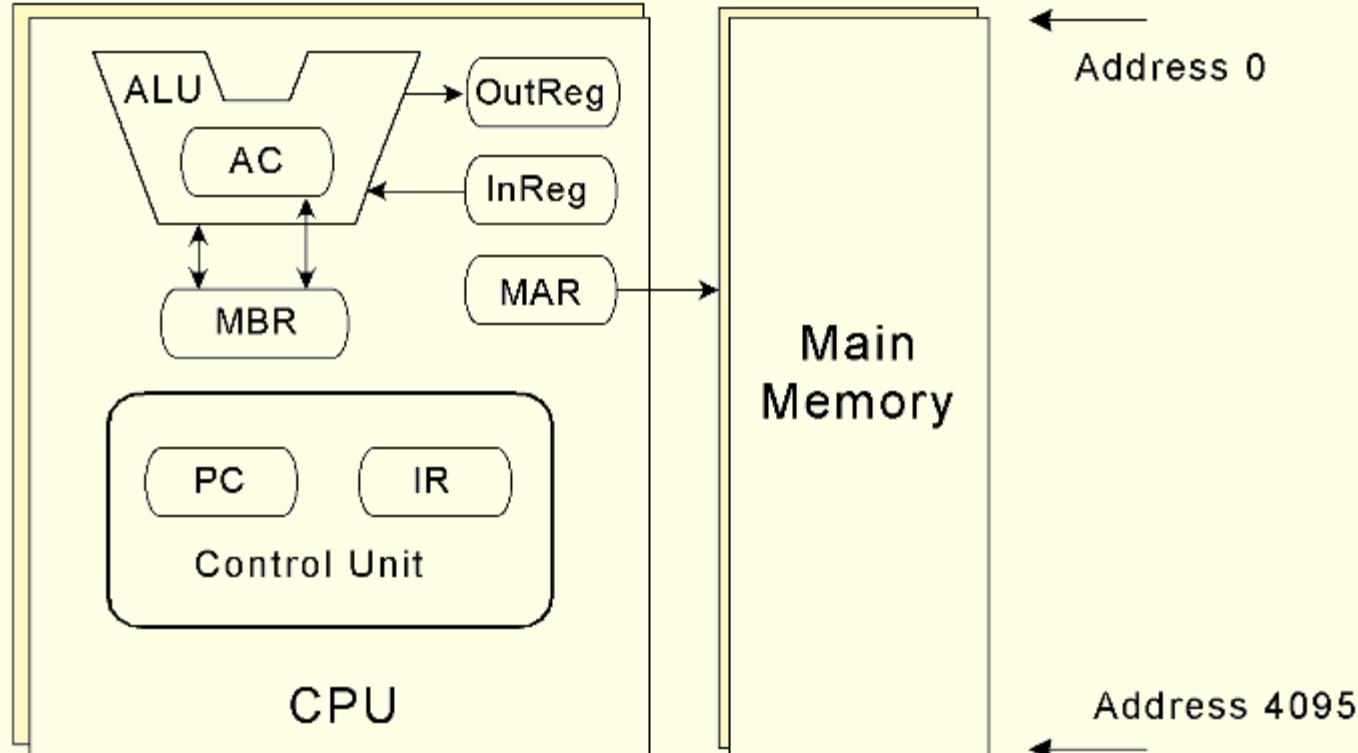
4.8 MARIE

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.

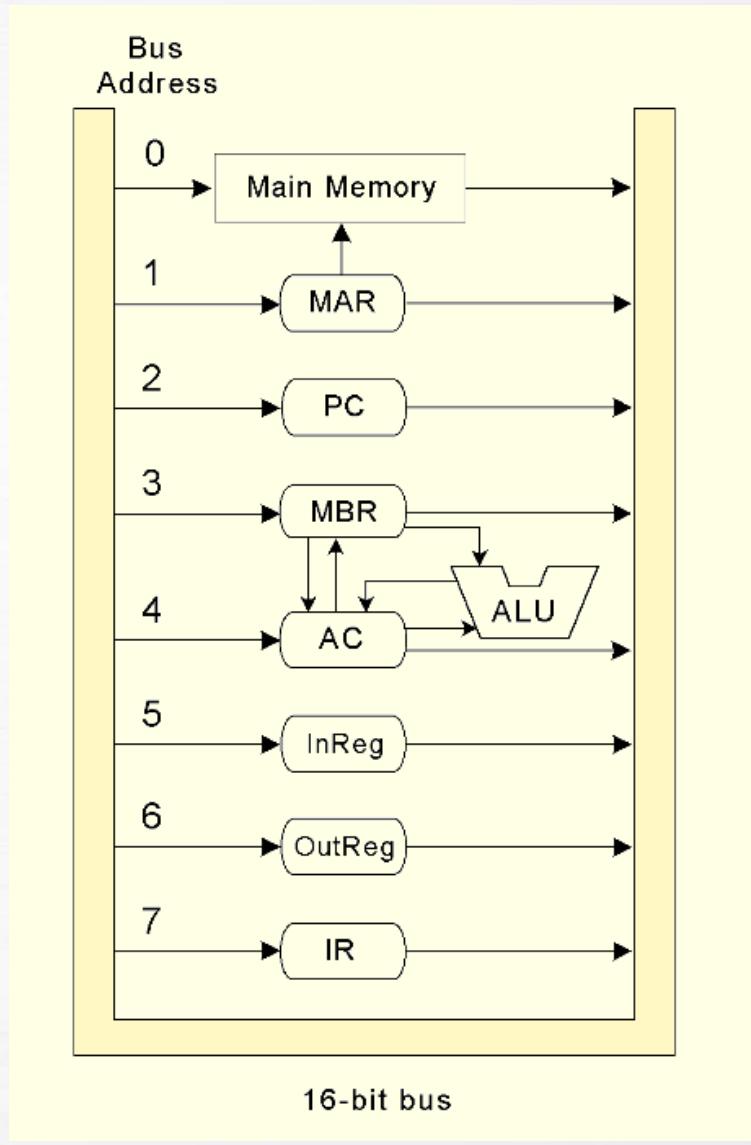


4.8 MARIE

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

4.8 MARIE

This is the MARIE data path shown graphically.

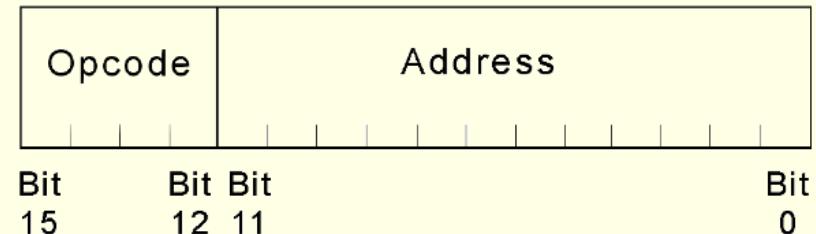


4.8 MARIE

- 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.
- We will be looking at nine instructions.

4.8 MARIE

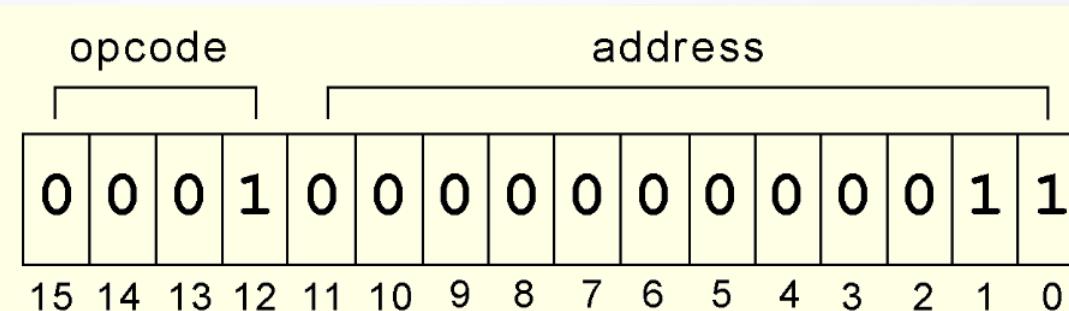
- This is the format of a MARIE instruction:
- The fundamental MARIE instructions are:



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

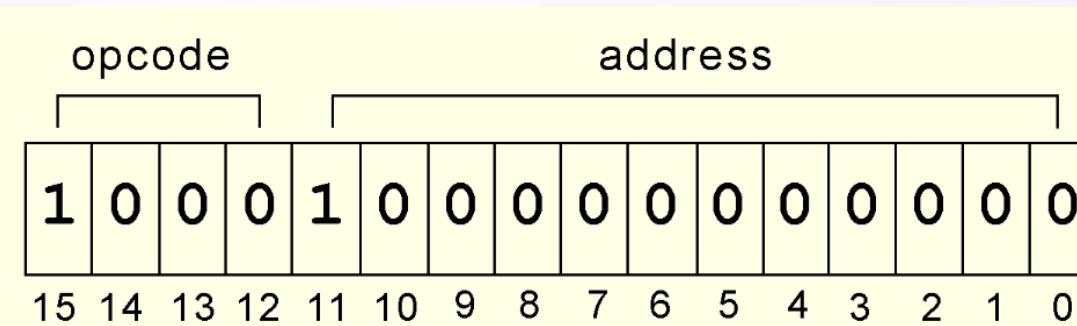
- This is a bit pattern for a **LOAD** instruction as it would appear in the IR:



- 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.

4.8 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 (RTL)*.
- In the MARIE RTL, we use the notation $M[X]$ to indicate the actual data value stored in memory location X , and \leftarrow to indicate the transfer of bytes to a register or memory location.

4.8 MARIE

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

MAR \leftarrow **X**

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

AC \leftarrow **MBR**

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

MAR \leftarrow **X**

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

AC \leftarrow **AC** + **MBR**

4.8 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:

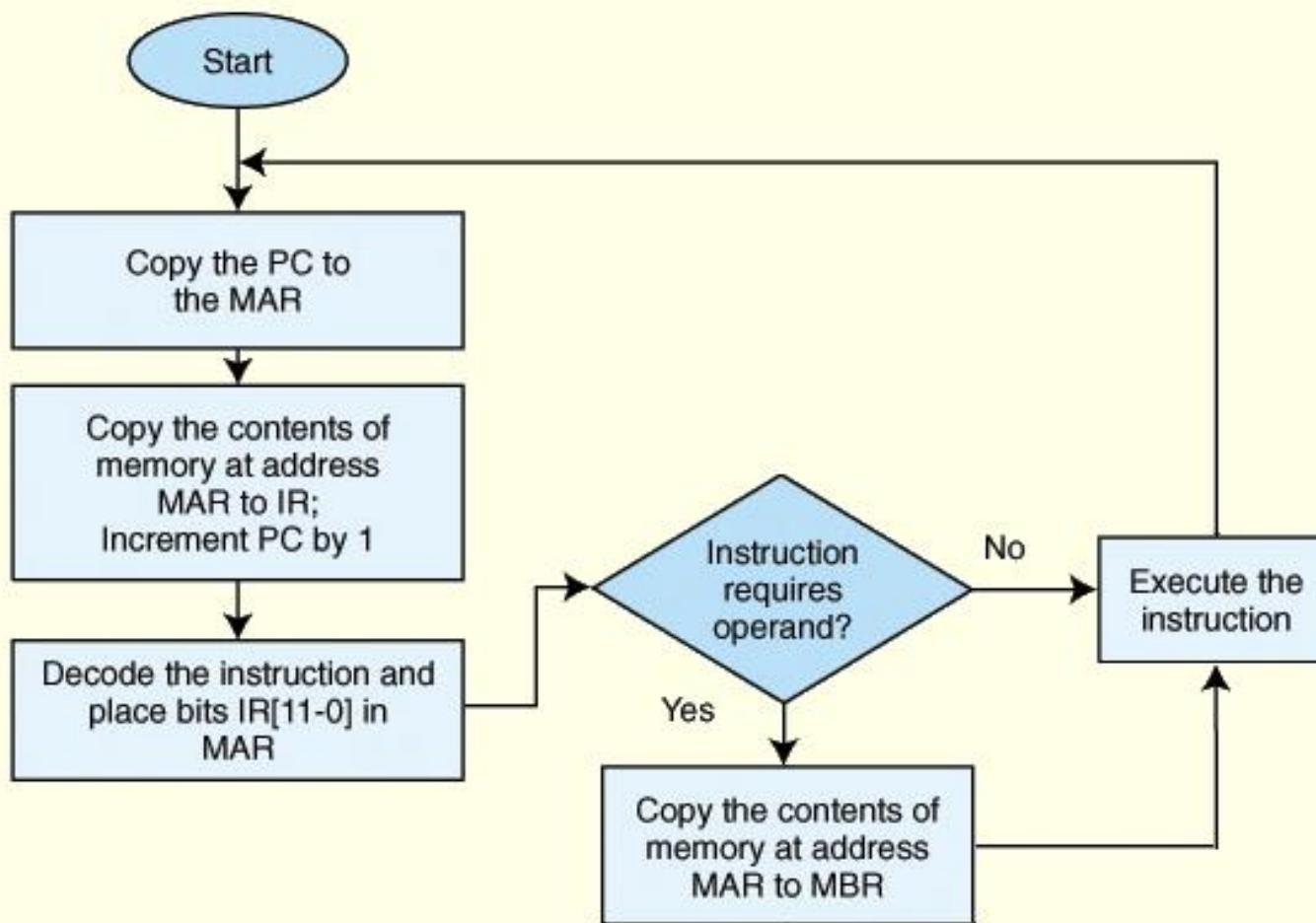
```
If IR[11 - 10] = 00 then  
    If AC < 0 then PC ← PC + 1  
else If IR[11 - 10] = 01 then  
    If AC = 0 then PC ← PC + 1  
else If IR[11 - 10] = 11 then  
    If AC > 0 then PC ← PC + 1
```

4.9 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

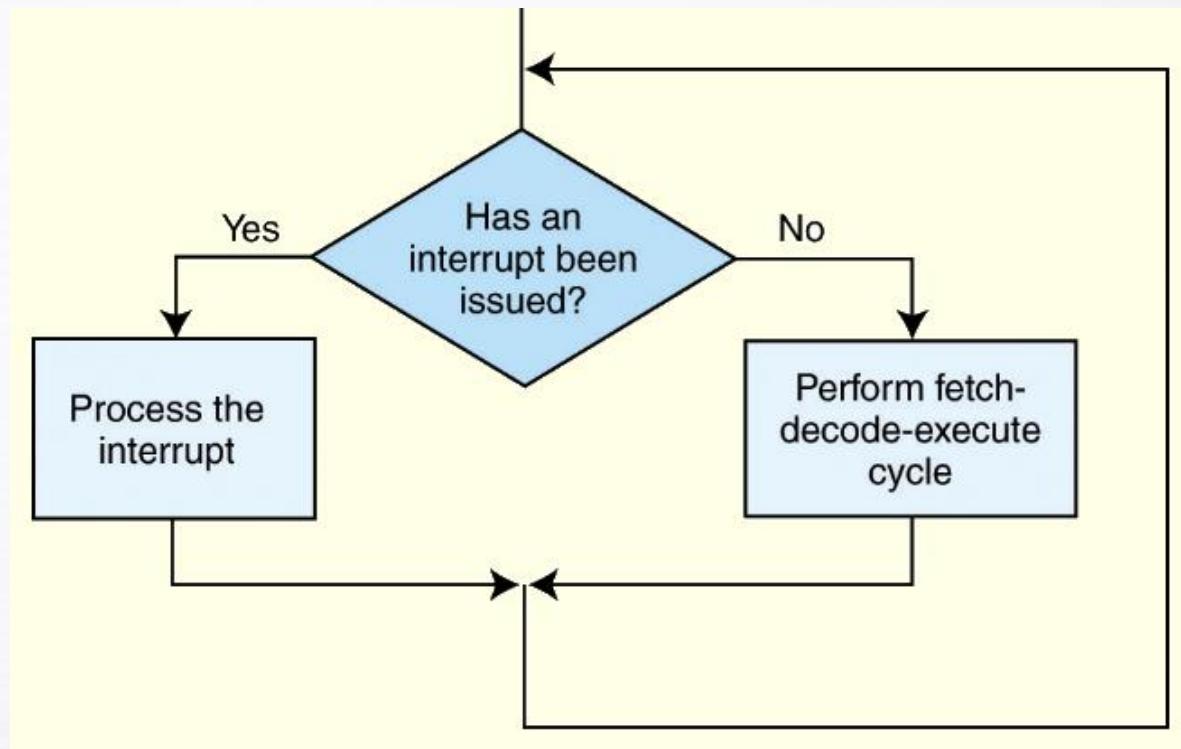


4.9 Instruction Processing

- All computers provide a way of interrupting the 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*.

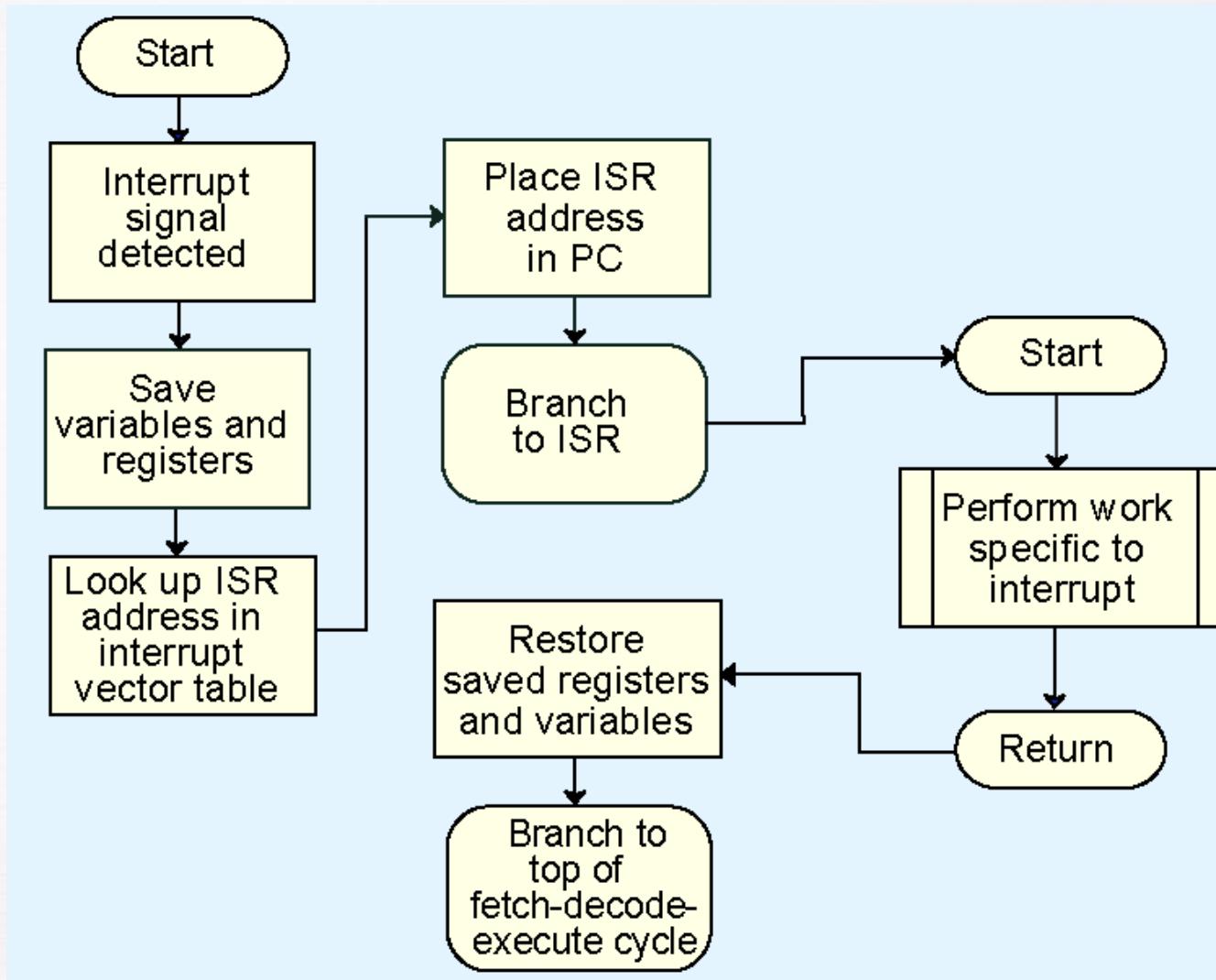
4.9 Instruction Processing

- 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.
 - We will look into this idea in greater detail in Chapter 7.
- 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
105	FFE9	111111111101001	FFE9
106	0000	0000000000000000	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 ← 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

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. With compilers, this is not usually the case.

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

X	104
Y	105
Z	106

1	X
3	Y
2	Z
7	0 0 0

4.11 A Discussion on Assemblers

- After the second pass, the assembly is complete.

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

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

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 **X** specifies the address of the operand to be loaded or stored.
- In RTL :

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
```

ADDI X

4.12 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
```

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

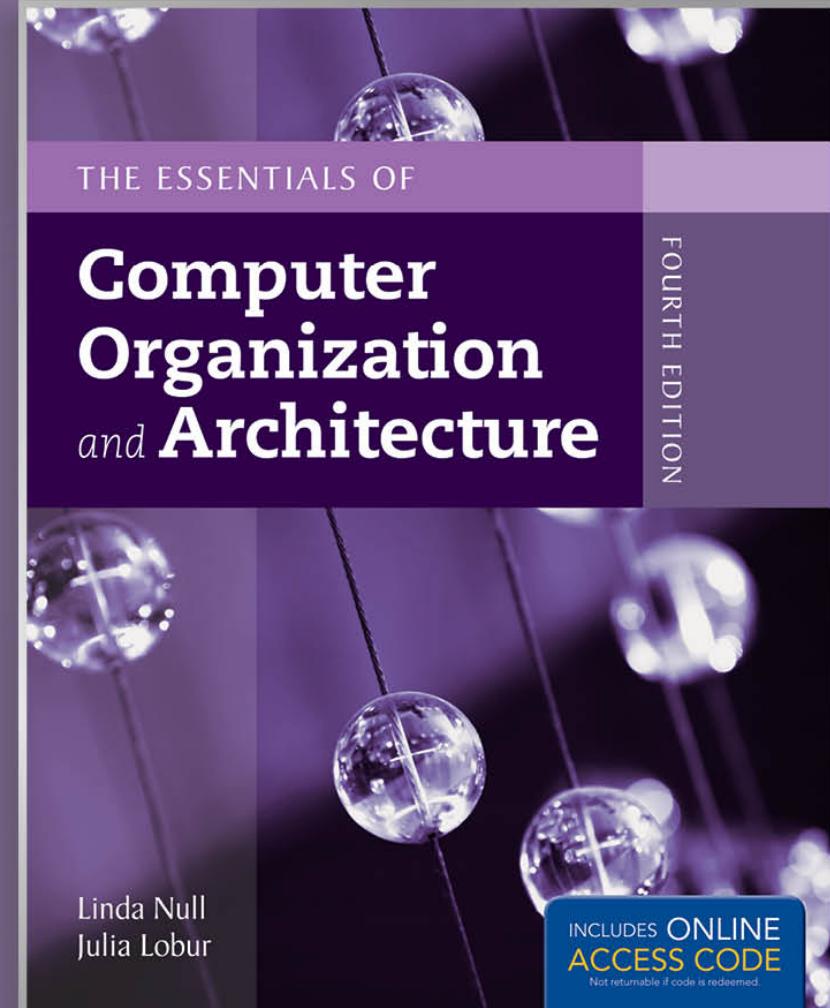
- We put our new instructions to work in the program on the following slide.

4.12 Extending Our Instruction Set

100	LOAD	Addr	10E	SKIPCOND
			000	
101	STORE	Next	10F	JUMP Loop
102	LOAD	Num	110	HALT
103	SUBT	One	111	Addr HEX 117
104	STORE	Ctr	112	Next HEX 0
105	Loop	LOAD Sum	113	Num DEC 5
106		ADDI Next	114	Sum DEC 0
			115	Ctr HEX 0
			116	One DEC 1
			117	DEC 10
107	STORE	Sum	118	DEC 15
108	LOAD	Next	119	DEC 2
109	ADD	One	11A	DEC 25
			11B	DEC 30
10A	STORE	Next		
10B	LOAD	Ctr		
10C	SUBT	One		
10D	STORE	Ctr		

Chapter 5

A Closer Look at Instruction Set Architectures



Chapter 5 Objectives

- Understand the factors involved in instruction set architecture design.
- Gain familiarity with memory addressing modes.
- Understand the concepts of instruction-level pipelining and its affect upon execution performance.

5.1 Introduction

- This chapter builds upon the ideas in Chapter 4.
- We present a detailed look at different instruction formats, operand types, and memory access methods.
- We will see the interrelation between machine organization and instruction formats.
- This leads to a deeper understanding of computer architecture in general.

5.2 Instruction Formats

Instruction sets are differentiated by the following:

- Number of bits per instruction.
- Stack-based or register-based.
- Number of explicit operands per instruction.
- Operand location.
- Types of operations.
- Type and size of operands.

5.2 Instruction Formats

Instruction set architectures are measured according to:

- Main memory space occupied by a program.
- Instruction complexity.
- Instruction length (in bits).
- Total number of instructions in the instruction set.

5.2 Instruction Formats

In designing an instruction set, consideration is given to:

- Instruction length.
 - Whether short, long, or variable.
- Number of operands.
- Number of addressable registers.
- Memory organization.
 - Whether byte- or word addressable.
- Addressing modes.
 - Choose any or all: direct, indirect or indexed.

5.2 Instruction Formats

- Byte ordering, or *endianness*, is another major architectural consideration.
- If we have a two-byte integer, the integer may be stored so that the least significant byte is followed by the most significant byte or vice versa.
 - In *little endian* machines, the least significant byte is followed by the most significant byte.
 - *Big endian* machines store the most significant byte first (at the lower address).

5.2 Instruction Formats

- As an example, suppose we have the hexadecimal number 0x12345678.
- The big endian and small endian arrangements of the bytes are shown below.

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

5.2 Instruction Formats

- A larger example: A computer uses 32-bit integers. The values 0xABCD1234, 0x00FE4321, and 0x10 would be stored sequentially in memory, starting at address 0x200 as below.

Address	Big Endian	Little Endian
0x200	AB	34
0x201	CD	12
0x202	12	CD
0x203	34	AB
0x204	00	21
0x205	FE	43
0x206	43	FE
0x207	21	00
0x208	00	10
0x209	00	00
0x20A	00	00
0x20B	10	00

5.2 Instruction Formats

- Big endian:
 - Is more natural.
 - The sign of the number can be determined by looking at the byte at address offset 0.
 - Strings and integers are stored in the same order.
- Little endian:
 - Makes it easier to place values on non-word boundaries.
 - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.

5.2 Instruction Formats

- The next consideration for architecture design concerns how the CPU will store data.
- We have three choices:
 1. A stack architecture
 2. An accumulator architecture
 3. A general purpose register architecture.
- In choosing one over the other, the tradeoffs are simplicity (and cost) of hardware design with execution speed and ease of use.

5.2 Instruction Formats

- In a stack architecture, instructions and operands are implicitly taken from the stack.
 - A stack cannot be accessed randomly.
- In an accumulator architecture, one operand of a binary operation is implicitly in the accumulator.
 - One operand is in memory, creating lots of bus traffic.
- In a general purpose register (GPR) architecture, registers can be used instead of memory.
 - Faster than accumulator architecture.
 - Efficient implementation for compilers.
 - Results in longer instructions.

5.2 Instruction Formats

- Most systems today are GPR systems.
- There are three types:
 - Memory-memory where two or three operands may be in memory.
 - Register-memory where at least one operand must be in a register.
 - Load-store where no operands may be in memory.
- The number of operands and the number of available registers has a direct affect on instruction length.

5.2 Instruction Formats

- Stack machines use one- and zero-operand instructions.
- **LOAD** and **STORE** instructions require a single memory address operand.
- Other instructions use operands from the stack implicitly.
- **PUSH** and **POP** operations involve only the stack's top element.
- Binary instructions (e.g., **ADD**, **MULT**) use the top two items on the stack.

5.2 Instruction Formats

- Stack architectures require us to think about arithmetic expressions a little differently.
- We are accustomed to writing expressions using *infix* notation, such as: $Z = X + Y$.
- Stack arithmetic requires that we use *postfix* notation: $Z = XY+$.
 - This is also called *reverse Polish notation*, (somewhat) in honor of its Polish inventor, Jan Lukasiewicz (1878 - 1956).

5.2 Instruction Formats

- The principal advantage of postfix notation is that parentheses are not used.
- For example, the infix expression,

$$Z = (X \times Y) + (W \times U) ,$$

becomes:

$$Z = X \ Y \times W \ U \times +$$

in postfix notation.

5.2 Instruction Formats

- Example: Convert the infix expression $(2+3) - 6/3$ to postfix:

$2\ 3+ -\ 6/3$

The sum $2 + 3$ in parentheses takes precedence; we replace the term with $2\ 3\ +.$

5.2 Instruction Formats

- Example: Convert the infix expression $(2+3) - 6/3$ to postfix:

2 3+ - 6 3/

The division operator takes next precedence; we replace $6/3$ with $6\ 3\ /$.

5.2 Instruction Formats

- Example: Convert the infix expression $(2+3) - 6/3$ to postfix:

2 3+ 6 3/ -

The quotient $6/3$ is subtracted from the sum of $2 + 3$, so we move the $-$ operator to the end.

5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression $2\ 3\ +\ 6\ 3\ /\ -$:

Scanning the expression from left to right, push operands onto the stack, until an operator is found

2	3	+	6	3	/	-
---	---	---	---	---	---	---



3
2

5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression $2\ 3\ +\ 6\ 3\ /\ -$:

Pop the two operands and carry out the operation indicated by the operator.
Push the result back on the stack.

2	3	+	6	3	/	-
---	---	---	---	---	---	---



5

5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression $2\ 3\ +\ 6\ 3\ /\ -$:

2	3	+	6	3	/	-
---	---	---	---	---	---	---

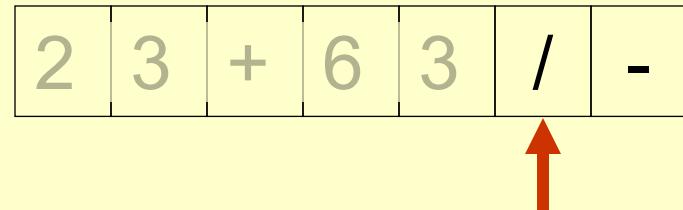


Push operands until another operator is found.

3
6
5

5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression $2\ 3\ +\ 6\ 3\ /\ -$:



Carry out the operation and push the result.

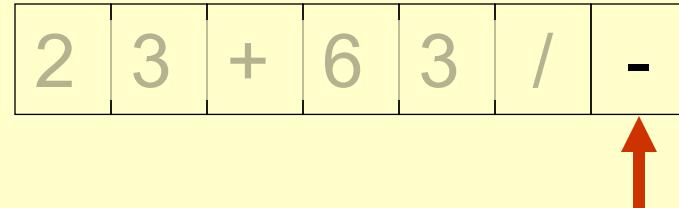


5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression $2\ 3\ +\ 6\ 3\ /\ -$:

Finding another operator,
carry out the operation and
push the result.

The answer is at the top of
the stack.



5.2 Instruction Formats

- Let's see how to evaluate an infix expression using different instruction formats.
- With a three-address ISA, (e.g.,mainframes), the infix expression,

$Z = X \times Y + W \times U$

might look like this:

```
MULT R1,X,Y  
MULT R2,W,U  
ADD Z,R1,R2
```

5.2 Instruction Formats

- In a two-address ISA, (e.g., Intel, Motorola), the infix expression,

$$Z = X \times Y + W \times U$$

might look like this:

```
LOAD R1,X  
MULT R1,Y  
LOAD R2,W  
MULT R2,U  
ADD R1,R2  
STORE Z,R1
```

Note: One-address ISAs usually require one operand to be a register.

5.2 Instruction Formats

- In a one-address ISA, like MARIE, the infix expression,

$$Z = X \times Y + W \times U$$

looks like this:

```
LOAD X
MULT Y
STORE TEMP
LOAD W
MULT U
ADD TEMP
STORE Z
```

5.2 Instruction Formats

- In a stack ISA, the postfix expression,

Z = X Y × W U × +

might look like this:

PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
PUSH Z

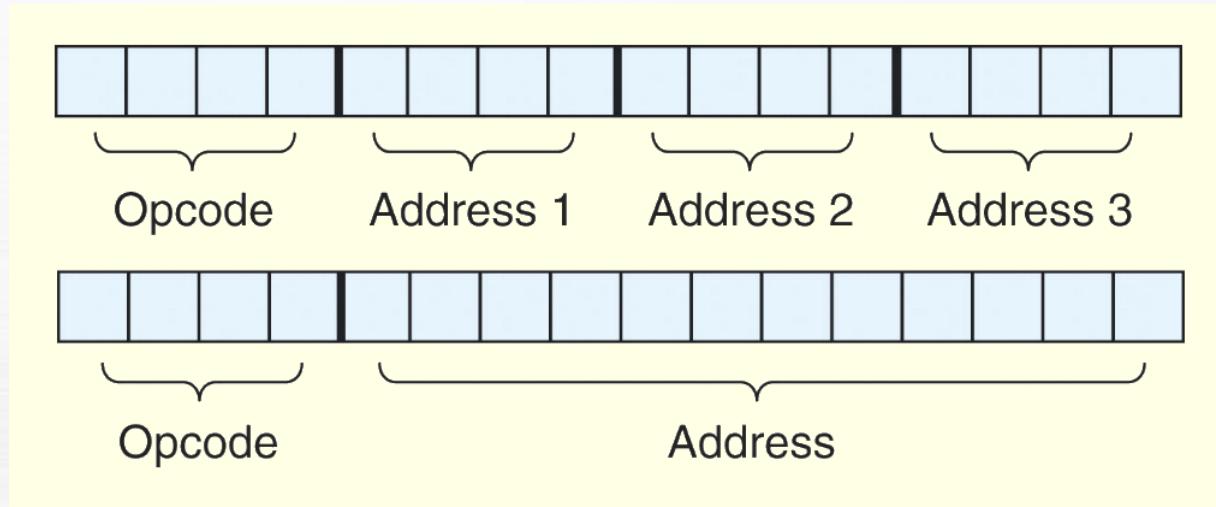
Would this program require more execution time than the corresponding (shorter) program that we saw in the 3-address ISA?

5.2 Instruction Formats

- We have seen how instruction length is affected by the number of operands supported by the ISA.
- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as `HALT`, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use expanding opcodes.

5.2 Instruction Formats

- A system has 16 registers and 4K of memory.
- We need 4 bits to access one of the registers. We also need 12 bits for a memory address.
- If the system is to have 16-bit instructions, we have two choices for our instructions:



5.2 Instruction Formats

- If we allow the length of the opcode to vary, we could create a very rich instruction set:

0000 R1 R2 R3	}	15 three-address codes
...		
1110 R1 R2 R3	}	14 two-address codes
1111 - escape opcode		
1111 0000 R1 R2	}	14 two-address codes
...		
1111 1101 R1 R2	}	14 two-address codes
1111 1110 - escape opcode		
1111 1110 0000 R1	}	31 one-address codes
...		
1111 1111 1110 R1	}	31 one-address codes
1111 1111 1111 - escape opcode		
1111 1111 1111 0000	}	16 zero-address codes
...		
1111 1111 1111 1111		

5.2 Instruction Formats

- Example: Given 8-bit instructions, is it possible to allow the following to be encoded?
 - 3 instructions with two 3-bit operands.
 - 2 instructions with one 4-bit operand.
 - 4 instructions with one 3-bit operand.

We need:

$$3 \times 2^3 = 192 \text{ bits for the 3-bit operands}$$

$$2 \times 2^4 = 32 \text{ bits for the 4-bit operands}$$

$$4 \times 2^3 = 32 \text{ bits for the 3-bit operands.}$$

Total: 256 bits.

5.2 Instruction Formats

- With a total of 256 bits required, we can exactly encode our instruction set in 8 bits!

We need:

$$3 \times 2^3 = 192 \text{ bits for the 3-bit operands}$$

$$2 \times 2^4 = 32 \text{ bits for the 4-bit operands}$$

$$4 \times 2^3 = 32 \text{ bits for the 3-bit operands.}$$

Total: 256 bits.

One such encoding is shown on the next slide.

5.2 Instruction Formats

00 xxx xxx	}	3 instructions with two 3-bit operands
01 xxx xxx		
10 xxx xxx		
11 - escape opcode	}	2 instructions with one 4-bit operand
1100 xxxx		
1101 xxxx		
1110 - escape opcode	}	4 instructions with one 3-bit operand
1111 - escape opcode		
11100 xxx		
11101 xxx		
11110 xxx		
11111 xxx		

5.3 Instruction types

Instructions fall into several broad categories that you should be familiar with:

- Data movement.
- Arithmetic.
- Boolean.
- Bit manipulation.
- I/O.
- Control transfer.
- Special purpose.

Can you think of
some examples
of each of these?

5.4 Addressing

- Addressing modes specify where an operand is located.
- They can specify a constant, a register, or a memory location.
- The actual location of an operand is its *effective address*.
- Certain addressing modes allow us to determine the address of an operand dynamically.

5.4 Addressing

- *Immediate addressing* is where the data is part of the instruction.
- *Direct addressing* is where the address of the data is given in the instruction.
- *Register addressing* is where the data is located in a register.
- *Indirect addressing* gives the address of the address of the data in the instruction.
- *Register indirect addressing* uses a register to store the address of the address of the data.

5.4 Addressing

- *Indexed addressing* uses a register (implicitly or explicitly) as an offset, which is added to the address in the operand to determine the effective address of the data.
- *Based addressing* is similar except that a base register is used instead of an index register.
- The difference between these two is that an index register holds an offset relative to the address given in the instruction, a base register holds a base address where the address field represents a displacement from this base.

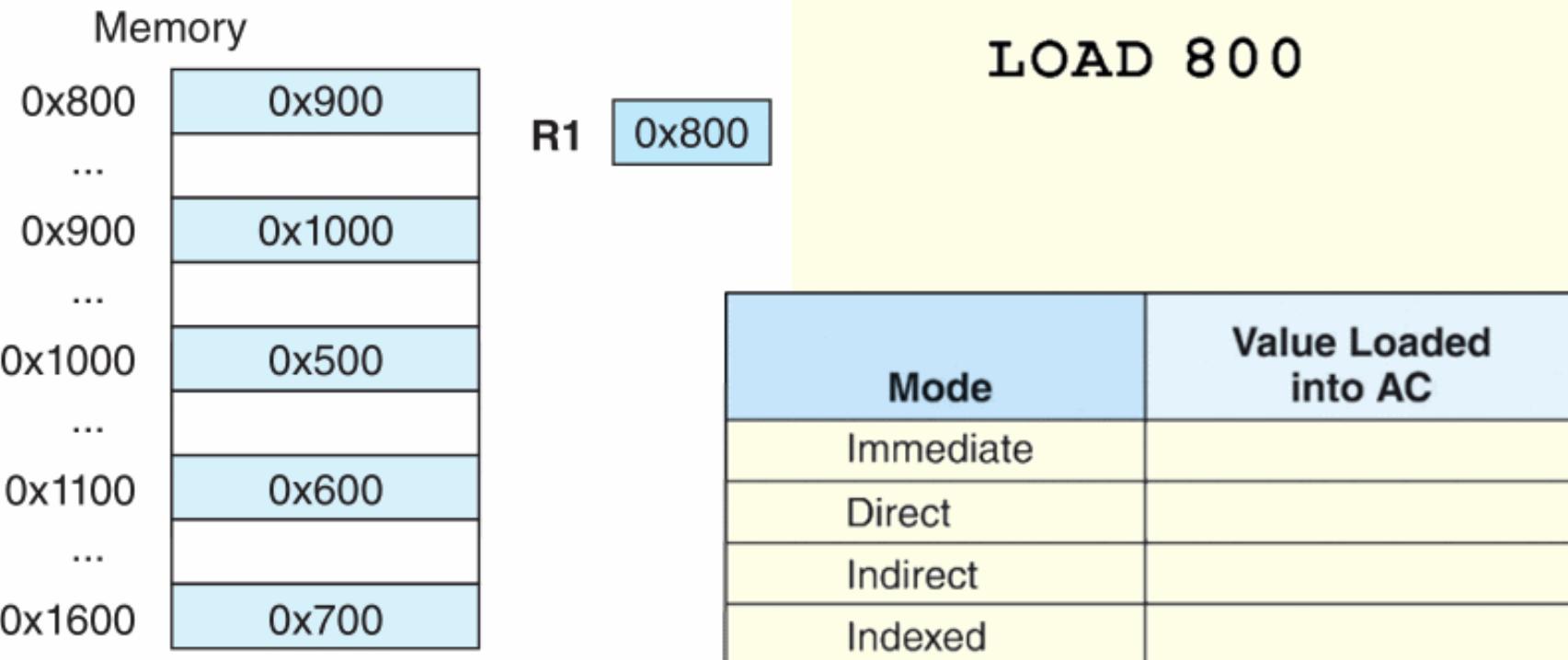
5.4 Addressing

- In *stack addressing* the operand is assumed to be on top of the stack.
- There are many variations to these addressing modes including:
 - Indirect indexed.
 - Base/offset.
 - Self-relative
 - Auto increment - decrement.
- We won't cover these in detail.

Let's look at an example of the principal addressing modes.

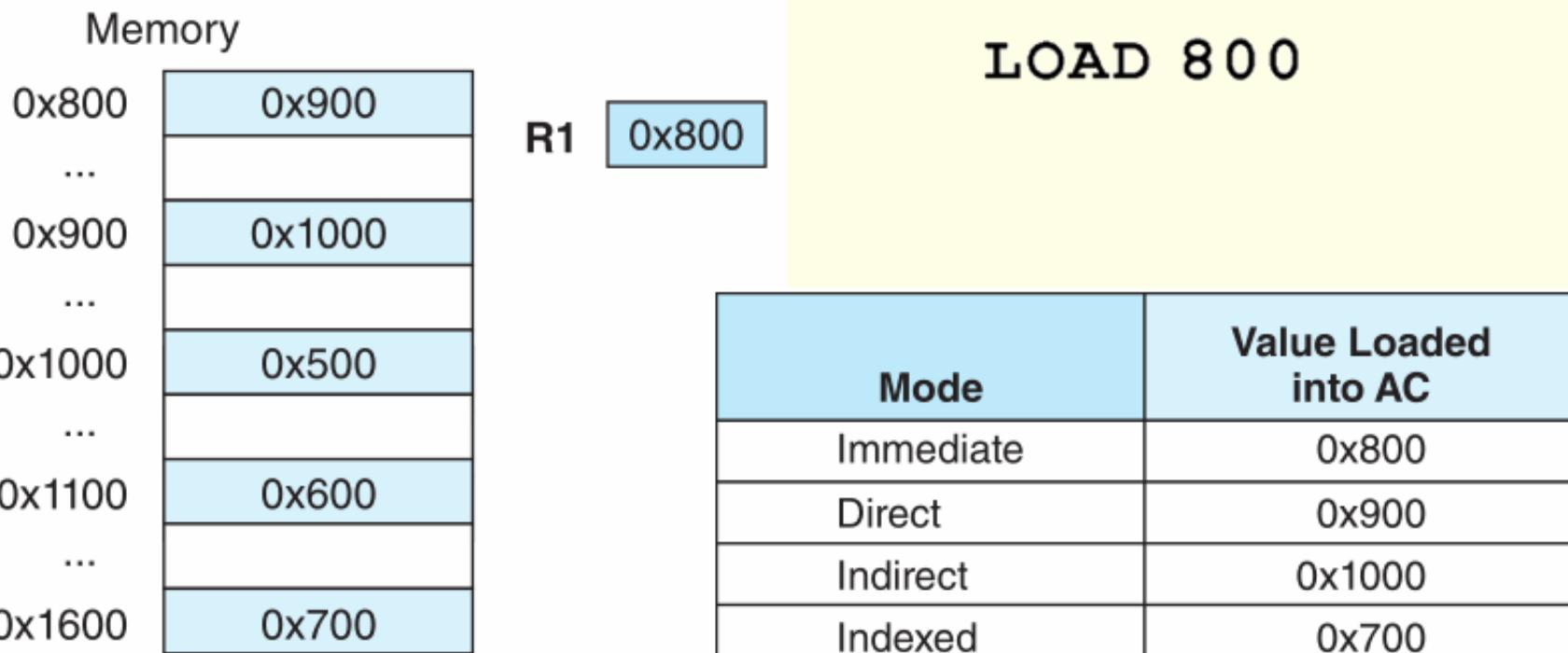
5.4 Addressing

- For the instruction shown, what value is loaded into the accumulator for each addressing mode?



5.4 Addressing

- For the instruction shown, what value is loaded into the accumulator for each addressing mode?



5.5 Instruction Pipelining

- Some CPUs divide the fetch-decode-execute cycle into smaller steps.
- These smaller steps can often be executed in parallel to increase throughput.
- Such parallel execution is called *instruction pipelining*.
- Instruction pipelining provides for *instruction level parallelism (ILP)*

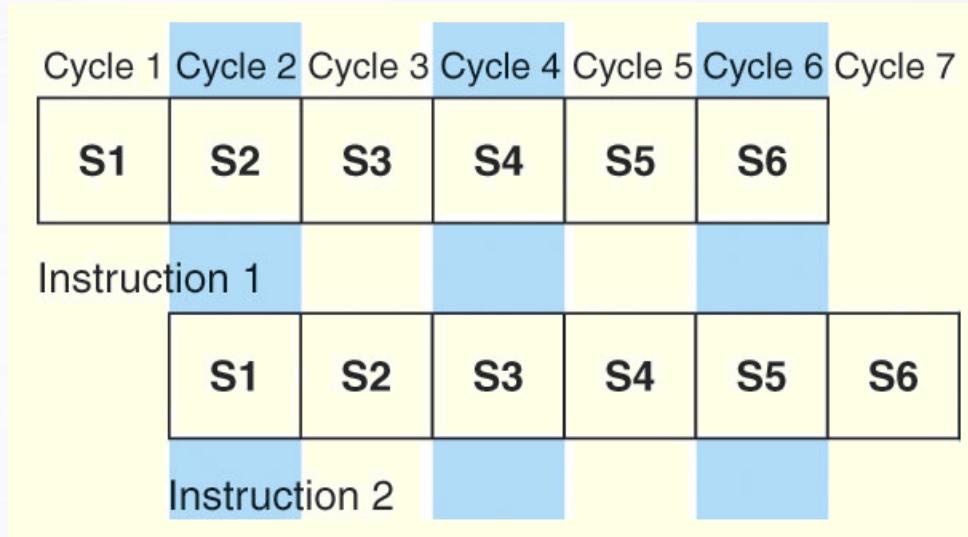
The next slide shows an example of instruction pipelining.

5.5 Instruction Pipelining

- Suppose a fetch-decode-execute cycle were broken into the following smaller steps:
 1. Fetch instruction.
 2. Decode opcode.
 3. Calculate effective address of operands.
 4. Fetch operands.
 5. Execute instruction.
 6. Store result.
- Suppose we have a six-stage pipeline. S1 fetches the instruction, S2 decodes it, S3 determines the address of the operands, S4 fetches them, S5 executes the instruction, and S6 stores the result.

5.5 Instruction Pipelining

- For every clock cycle, one small step is carried out, and the stages are overlapped.



S1. Fetch instruction.
S2. Decode opcode.
S3. Calculate effective address of operands.

S4. Fetch operands.
S5. Execute.
S6. Store result.

5.5 Instruction Pipelining

- The theoretical speedup offered by a pipeline can be determined as follows:

Let t_p be the time per stage. Each instruction represents a task, T , in the pipeline.

The first task (instruction) requires $k \times t_p$ time to complete in a k -stage pipeline. The remaining $(n - 1)$ tasks emerge from the pipeline one per cycle. So the total time to complete the remaining tasks is $(n - 1)t_p$.

Thus, to complete n tasks using a k -stage pipeline requires:

$$(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p.$$

5.5 Instruction Pipelining

- If we take the time required to complete n tasks without a pipeline and divide it by the time it takes to complete n tasks using a pipeline, we find:

$$\text{Speedup } S = \frac{n t_n}{(k + n - 1) t_p}$$

- If we take the limit as n approaches infinity, $(k + n - 1)$ approaches n , which results in a theoretical speedup of:

$$\text{Speedup } S = \frac{k t_p}{t_p} = k$$

5.5 Instruction Pipelining

- Our neat equations take a number of things for granted.
- First, we have to assume that the architecture supports fetching instructions and data in parallel.
- Second, we assume that the pipeline can be kept filled at all times. This is not always the case. Pipeline *hazards* arise that cause pipeline conflicts and stalls.

5.5 Instruction Pipelining

- An instruction pipeline may stall, or be flushed for any of the following reasons:
 - Resource conflicts.
 - Data dependencies.
 - Conditional branching.
- Measures can be taken at the software level as well as at the hardware level to reduce the effects of these hazards, but they cannot be totally eliminated.

5.6 Real-World Examples of ISAs

- We return briefly to the Intel and MIPS architectures from the last chapter, using some of the ideas introduced in this chapter.
- Intel introduced pipelining to their processor line with its Pentium chip.
- The first Pentium had two five-stage pipelines. Each subsequent Pentium processor had a longer pipeline than its predecessor with the Pentium IV having a 24-stage pipeline.
- The Itanium (IA-64) has only a 10-stage pipeline.

5.6 Real-World Examples of ISAs

- Intel processors support a wide array of addressing modes.
- The original 8086 provided 17 ways to address memory, most of them variants on the methods presented in this chapter.
- Owing to their need for backward compatibility, the Pentium chips also support these 17 addressing modes.
- The Itanium, having a RISC core, supports only one: register indirect addressing with optional post increment.

5.6 Real-World Examples of ISAs

- MIPS was an acronym for *Microprocessor Without Interlocked Pipeline Stages*.
- The architecture is little endian and word-addressable with three-address, fixed-length instructions.
- Like Intel, the pipeline size of the MIPS processors has grown: The R2000 and R3000 have five-stage pipelines.; the R4000 and R4400 have 8-stage pipelines.

5.6 Real-World Examples of ISAs

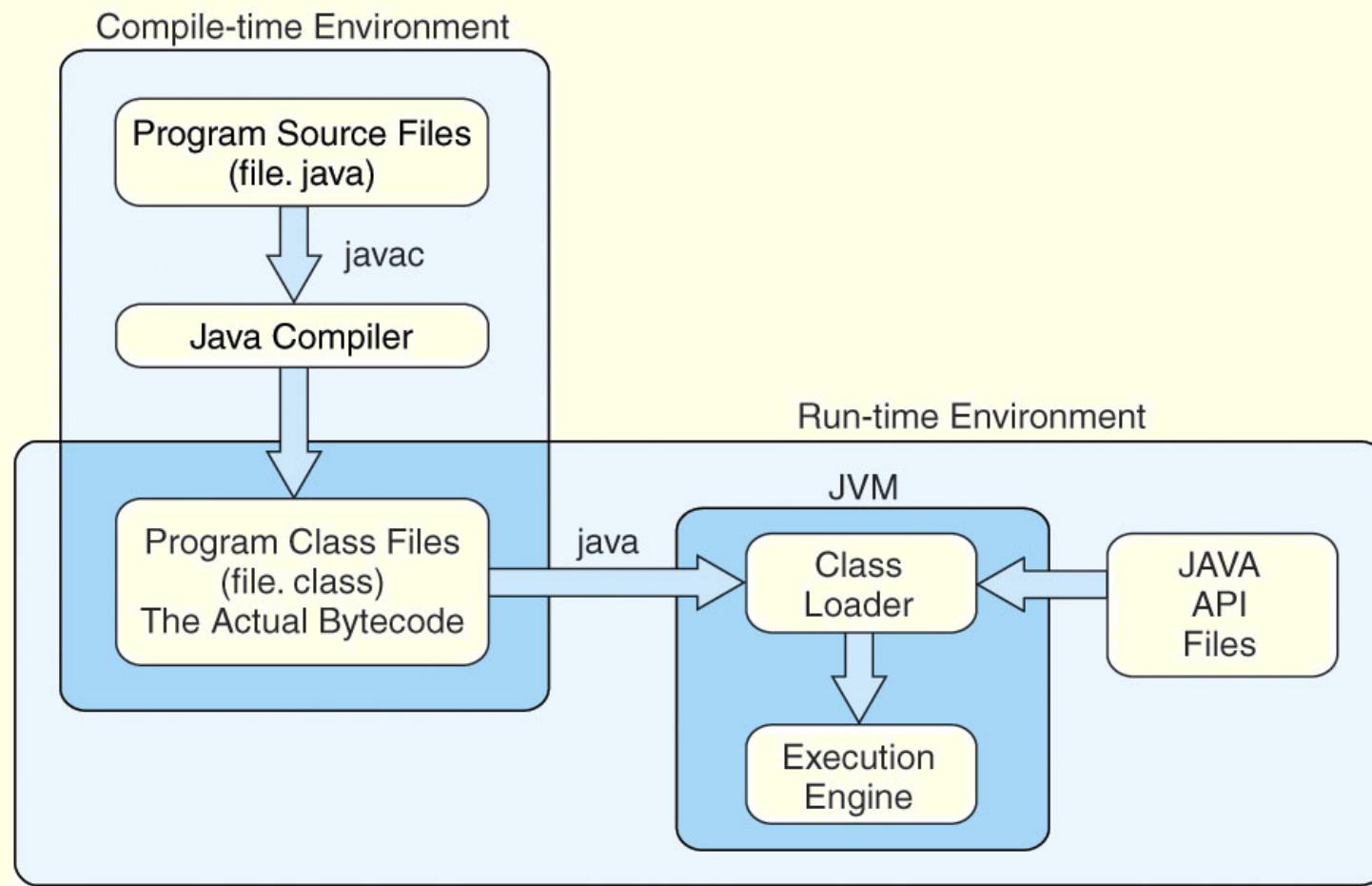
- The R10000 has three pipelines: A five-stage pipeline for integer instructions, a seven-stage pipeline for floating-point instructions, and a six-state pipeline for **LOAD/STORE** instructions.
- In all MIPS ISAs, only the **LOAD** and **STORE** instructions can access memory.
- The ISA uses only base addressing mode.
- The assembler accommodates programmers who need to use immediate, register, direct, indirect register, base, or indexed addressing modes.

5.6 Real-World Examples of ISAs

- The Java programming language is an interpreted language that runs in a software machine called the *Java Virtual Machine* (JVM).
- A JVM is written in a native language for a wide array of processors, including MIPS and Intel.
- Like a real machine, the JVM has an ISA all of its own, called *bytecode*. This ISA was designed to be compatible with the architecture of any machine on which the JVM is running.

The next slide shows how the pieces fit together.

5.6 Real-World Examples of ISAs



5.6 Real-World Examples of ISAs

- Java bytecode is a stack-based language.
- Most instructions are zero address instructions.
- The JVM has four registers that provide access to five regions of main memory.
- All references to memory are offsets from these registers. Java uses no pointers or absolute memory references.
- Java was designed for platform interoperability, not performance!

5.6 Real-World Examples of ISAs

- You may not have heard of ARM but most likely use an ARM processor every day. It is the most widely used 32-bit instruction architecture:
 - 95%+ of smartphones,
 - 80%+ of digital cameras
 - 40%+ of all digital television sets
- Founded in 1990, by Apple and others, ARM (Advanced RISC Machine) is now a British firm, ARM Holdings.
- ARM Holdings does not manufacture these processors; it sells licenses to manufacture.

5.6 Real-World Examples of ISAs

- ARM is a load/store architecture : all data processing must be performed on values in registers, not in memory.
- It uses fixed-length, three-operand instructions and simple addressing modes
- ARM processors have a minimum of a three-stage pipeline (consisting of fetch, decode, and execute);
 - Newer ARM processors have deeper pipelines (more stages). Some ARM8 implementations have 13-stage integer pipelines

5.6 Real-World Examples of ISAs

- ARM has 37 total registers but their visibility depends on the processor mode.
- ARM allows multiple register transfers.
 - It can simultaneously load or store any subset of the 16 general-purpose registers from/to sequential memory addresses.
- Control flow instructions include unconditional and conditional branching and procedure calls
- Most ARM instructions execute in a single cycle, provided there are no pipeline hazards or memory accesses.

Chapter 5 Conclusion

- ISAs are distinguished according to their bits per instruction, number of operands per instruction, operand location and types and sizes of operands.
- Endianness as another major architectural consideration.
- CPU can store data based on
 1. A stack architecture
 2. An accumulator architecture
 3. A general purpose register architecture.

Chapter 5 Conclusion

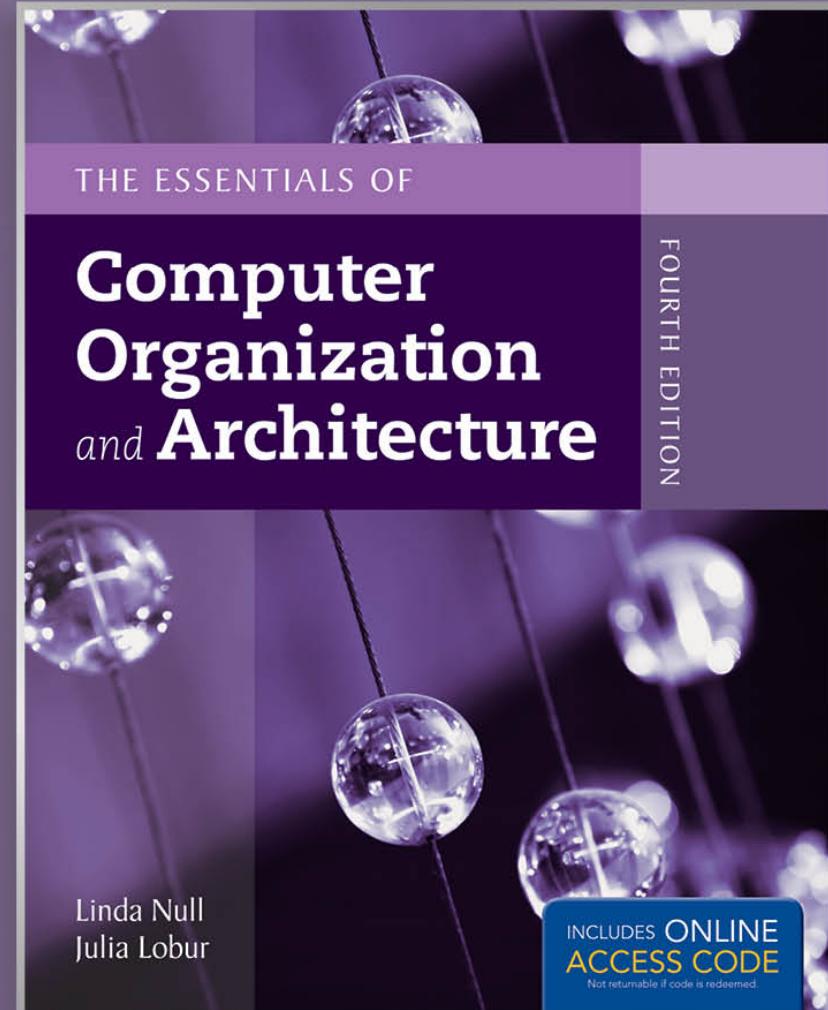
- Instructions can be fixed length or variable length.
- To enrich the instruction set for a fixed length instruction set, expanding opcodes can be used.
- The addressing mode of an ISA is also another important factor. We looked at:
 - Immediate
 - Register
 - Indirect
 - Based
 - Direct
 - Register Indirect
 - Indexed
 - Stack

Chapter 5 Conclusion

- A k -stage pipeline can theoretically produce execution speedup of k as compared to a non-pipelined machine.
- Pipeline hazards such as resource conflicts and conditional branching prevents this speedup from being achieved in practice.
- The Intel, MIPS, JVM, and ARM architectures provide good examples of the concepts presented in this chapter.

Chapter 6

Memory



Chapter 6 Objectives

- Master the concepts of hierarchical memory organization.
- Understand how each level of memory contributes to system performance, and how the performance is measured.
- Master the concepts behind cache memory, virtual memory, memory segmentation, paging and address translation.

6.1 Introduction

- Memory lies at the heart of the stored-program computer.
- In previous chapters, we studied the components from which memory is built and the ways in which memory is accessed by various ISAs.
- In this chapter, we focus on memory organization. A clear understanding of these ideas is essential for the analysis of system performance.

6.2 Types of Memory

- There are two kinds of main memory: *random access memory, RAM, and read-only-memory, ROM.*
- There are two types of RAM, dynamic RAM (DRAM) and static RAM (SRAM).
- DRAM consists of capacitors that slowly leak their charge over time. Thus, they must be refreshed every few milliseconds to prevent data loss.
- DRAM is “cheap” memory owing to its simple design.

6.2 Types of Memory

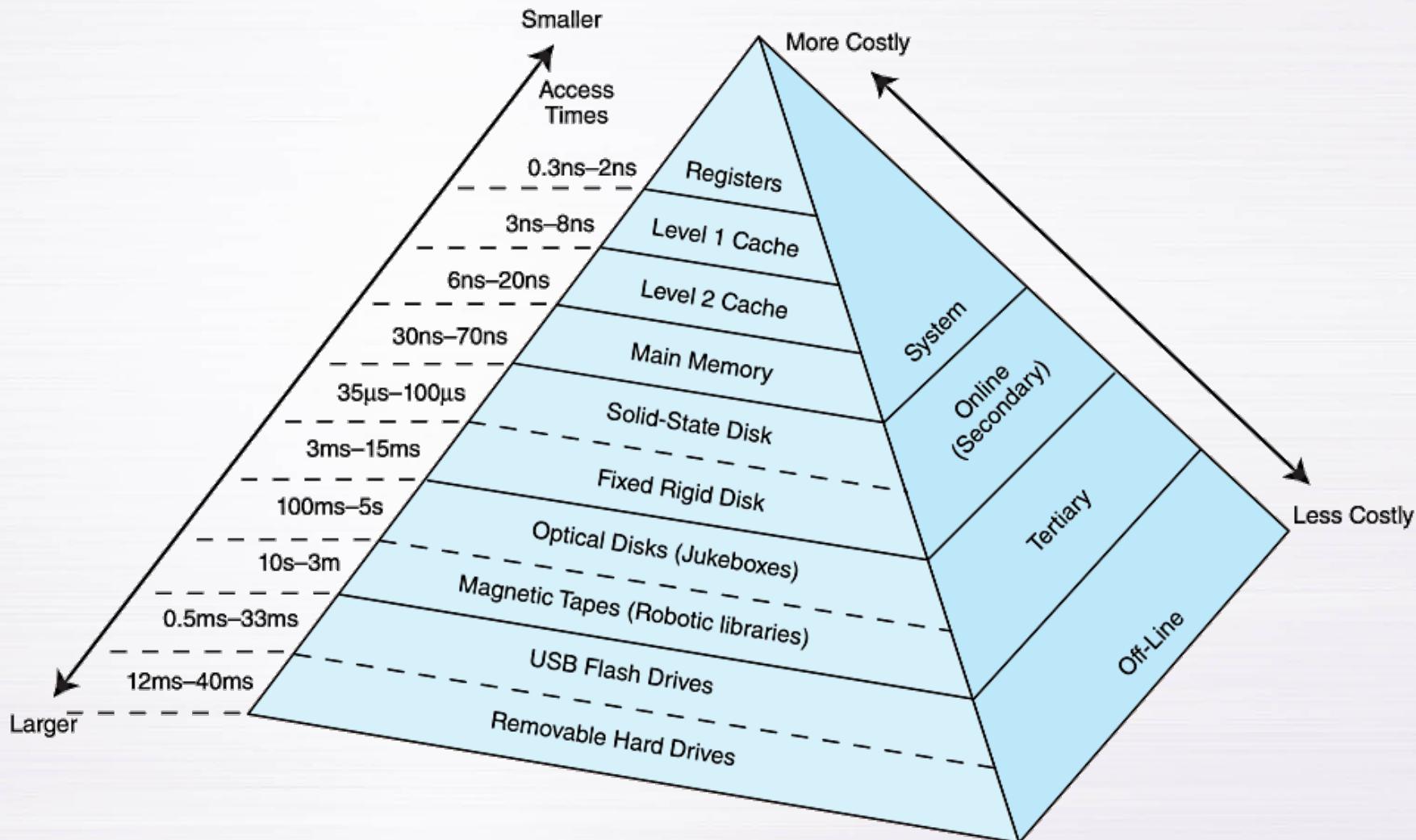
- SRAM consists of circuits similar to the D flip-flop that we studied in Chapter 3.
- SRAM is very fast memory and it doesn't need to be refreshed like DRAM does. It is used to build cache memory, which we will discuss in detail later.
- ROM also does not need to be refreshed, either. In fact, it needs very little charge to retain its memory.
- ROM is used to store permanent, or semi-permanent data that persists even while the system is turned off.

6.3 The Memory Hierarchy

- Generally speaking, faster memory is more expensive than slower memory.
- To provide the best performance at the lowest cost, memory is organized in a hierarchical fashion.
- Small, fast storage elements are kept in the CPU, larger, slower main memory is accessed through the data bus.
- Larger, (almost) permanent storage in the form of disk and tape drives is still further from the CPU.

6.3 The Memory Hierarchy

- This storage organization can be thought of as a pyramid:



6.3 The Memory Hierarchy

- We are most interested in the memory hierarchy that involves registers, cache, main memory, and virtual memory.
- Registers are storage locations available on the processor itself.
- Virtual memory is typically implemented using a hard drive; it extends the address space from RAM to the hard drive.
- Virtual memory provides more space: Cache memory provides speed.

6.3 The Memory Hierarchy

- To access a particular piece of data, the CPU first sends a request to its nearest memory, usually cache.
- If the data is not in cache, then main memory is queried. If the data is not in main memory, then the request goes to disk.
- Once the data is located, then the data, and a number of its nearby data elements are fetched into cache memory.

6.3 The Memory Hierarchy

- This leads us to some definitions.
 - A *hit* is when data is found at a given memory level.
 - A *miss* is when it is not found.
 - The *hit rate* is the percentage of time data is found at a given memory level.
 - The *miss rate* is the percentage of time it is not.
 - Miss rate = 1 - hit rate.
 - The *hit time* is the time required to access data at a given memory level.
 - The *miss penalty* is the time required to process a miss, including the time that it takes to replace a block of memory plus the time it takes to deliver the data to the processor.

6.3 The Memory Hierarchy

- An entire blocks of data is copied after a hit because the *principle of locality* tells us that once a byte is accessed, it is likely that a nearby data element will be needed soon.
- There are three forms of locality:
 - *Temporal locality*- Recently-accessed data elements tend to be accessed again.
 - *Spatial locality* - Accesses tend to cluster.
 - *Sequential locality* - Instructions tend to be accessed sequentially.

6.4 Cache Memory

- The purpose of cache memory is to speed up accesses by storing recently used data closer to the CPU, instead of storing it in main memory.
- Although cache is much smaller than main memory, its access time is a fraction of that of main memory.
- Unlike main memory, which is accessed by address, cache is typically accessed by content; hence, it is often called *content addressable memory*.
- Because of this, a single large cache memory isn't always desirable-- it takes longer to search.

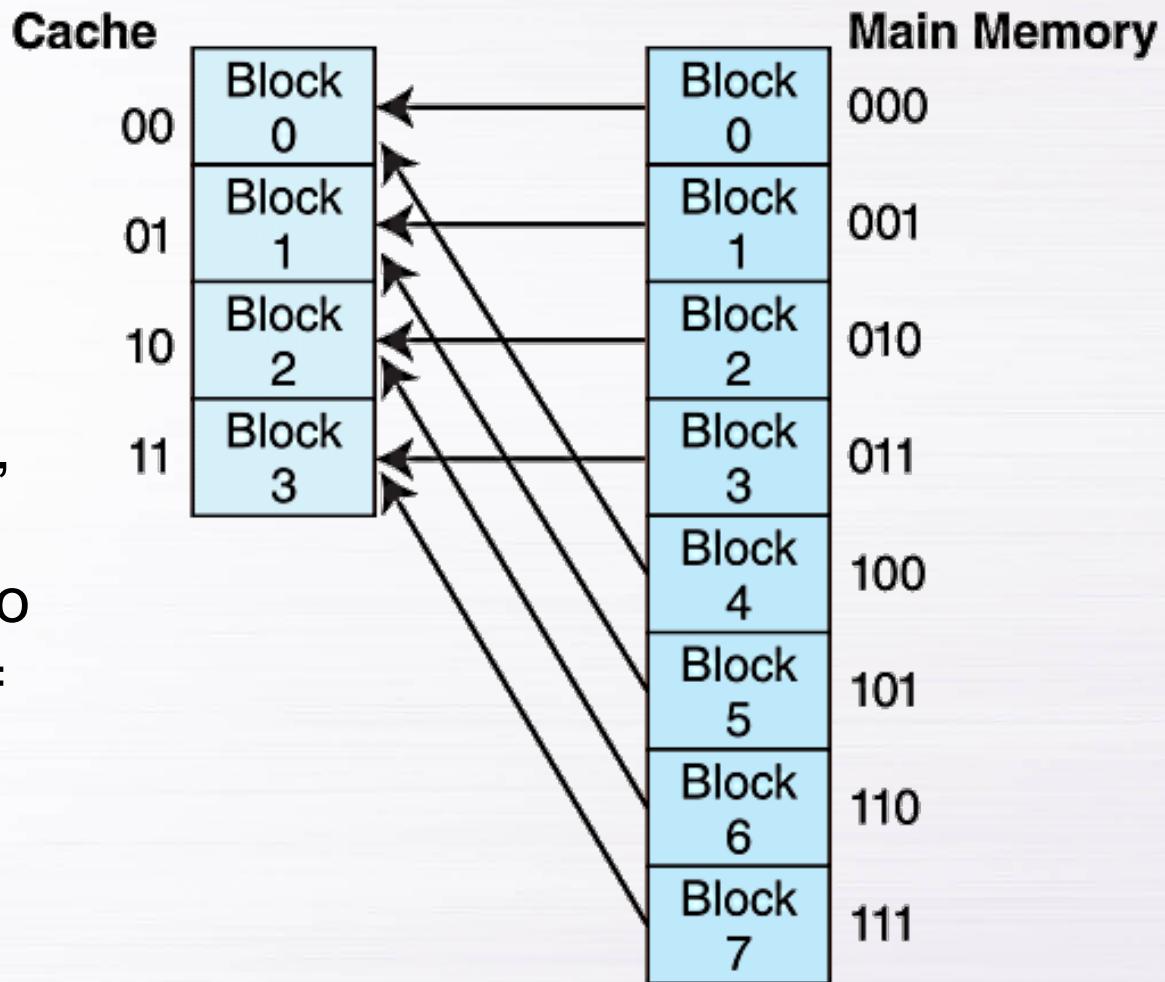
6.4 Cache Memory

- The simplest cache mapping scheme is *direct mapped cache*.
- In a direct mapped cache consisting of N blocks of cache, block X of main memory maps to cache block $Y = X \bmod N$.
- Thus, if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, . . . of main memory.

The next slide illustrates this mapping.

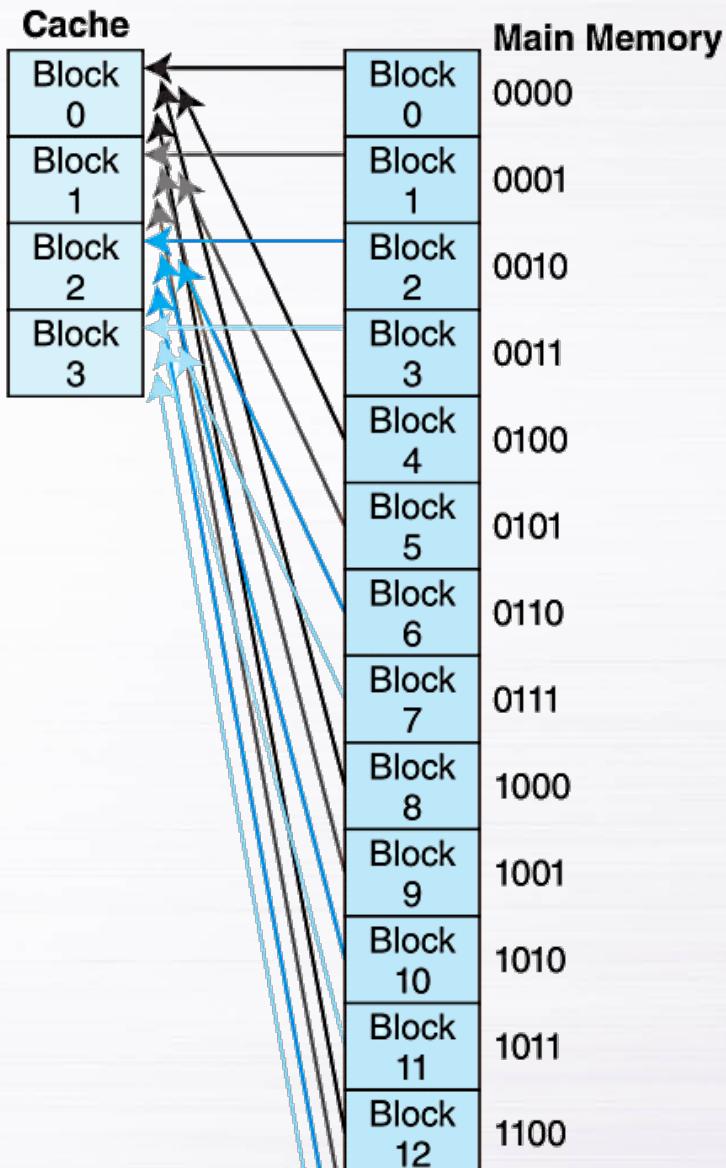
6.4 Cache Memory

- With direct mapped cache consisting of N blocks of cache, block X of main memory maps to cache block $Y = X \bmod N$.



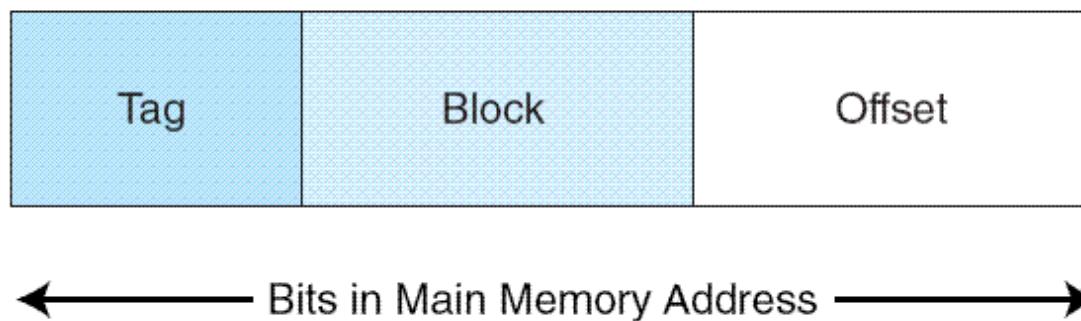
6.4 Cache Memory

- A larger example.



6.4 Cache Memory

- To perform direct mapping, the binary main memory address is partitioned into the *fields* shown below.
 - The *offset* field uniquely identifies an address within a specific block.
 - The *block* field selects a unique block of cache.
 - The *tag* field is whatever is left over.



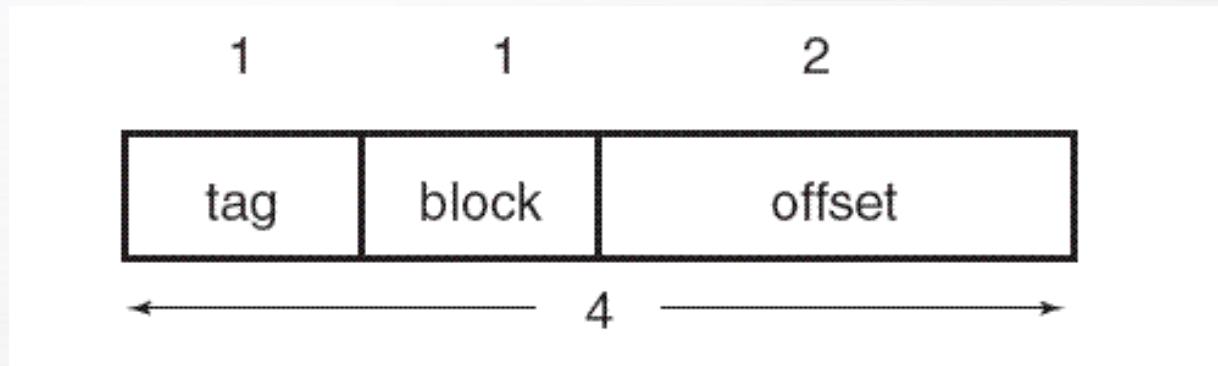
- The sizes of these fields are determined by characteristics of both memory and cache.

6.4 Cache Memory

- EXAMPLE 6.1 Consider a byte-addressable main memory consisting of 4 blocks, and a cache with 2 blocks, where each block is 4 bytes.
- This means Block 0 and 2 of main memory map to Block 0 of cache, and Blocks 1 and 3 of main memory map to Block 1 of cache.
- Using the tag, block, and offset fields, we can see how main memory maps to cache as follows.

6.4 Cache Memory

- EXAMPLE 6.1 Cont'd Consider a byte-addressable main memory consisting of 4 blocks, and a cache with 2 blocks, where each block is 4 bytes.
 - First, we need to determine the address format for mapping. Each block is 4 bytes, so the offset field must contain 2 bits; there are 2 blocks in cache, so the block field must contain 1 bit; this leaves 1 bit for the tag (as a main memory address has 4 bits because there are a total of $2^4=16$ bytes).

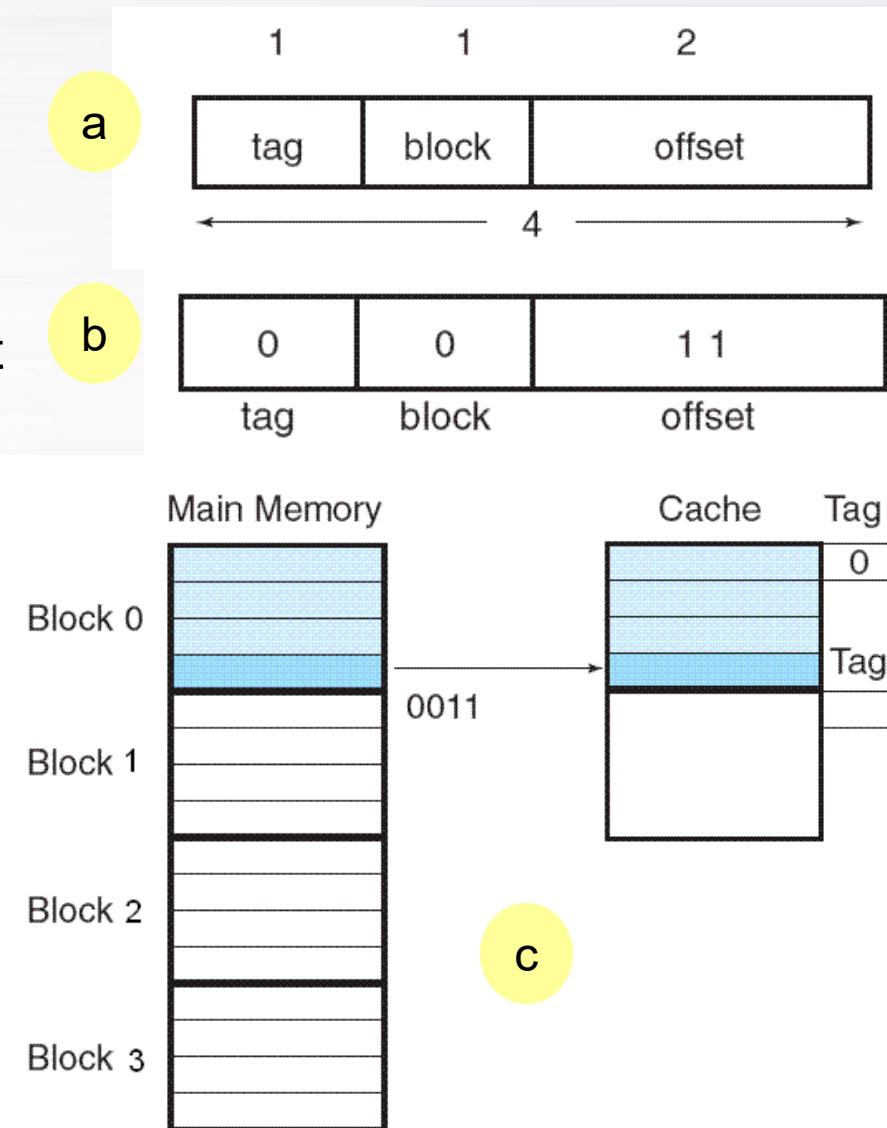


6.4 Cache Memory

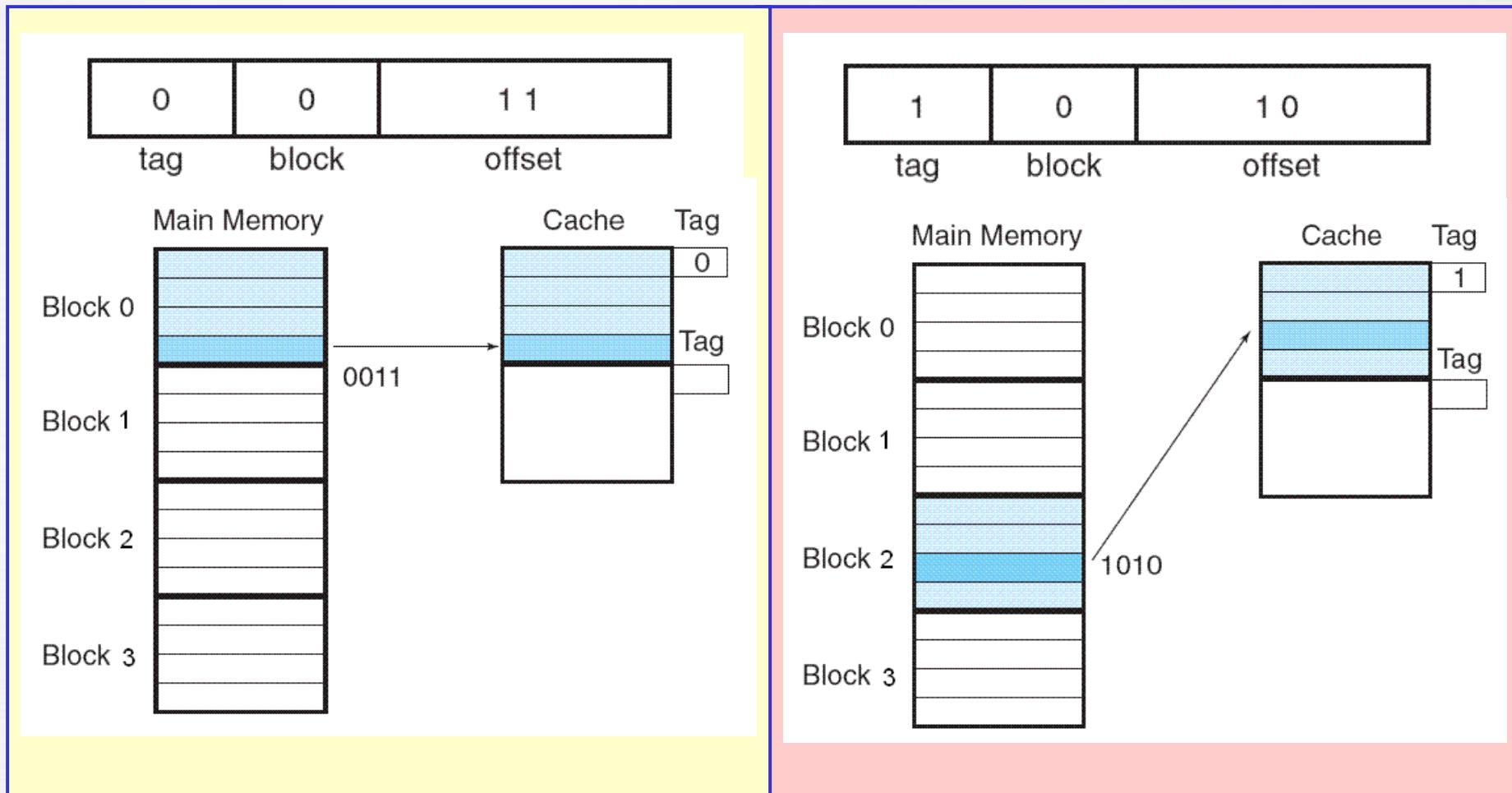
- EXAMPLE 6.1 Cont'd

- Suppose we need to access main memory address 3_{16} (0x0011 in binary). If we partition 0x0011 using the address format from Figure a, we get Figure b.
- Thus, the main memory address 0x0011 maps to cache block 0.
- Figure c shows this mapping, along with the tag that is also stored with the data.

The next slide illustrates another mapping.

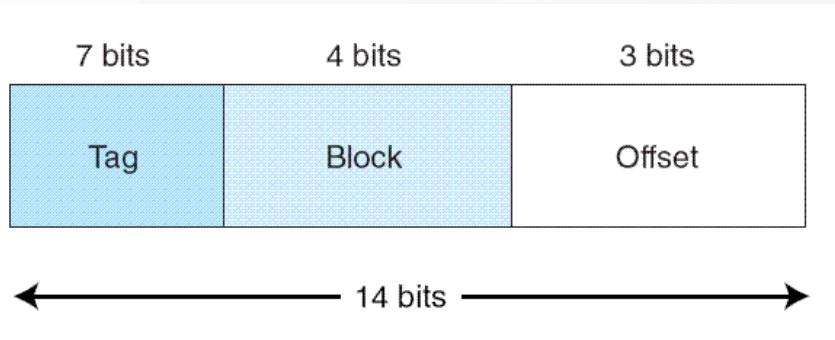


6.4 Cache Memory



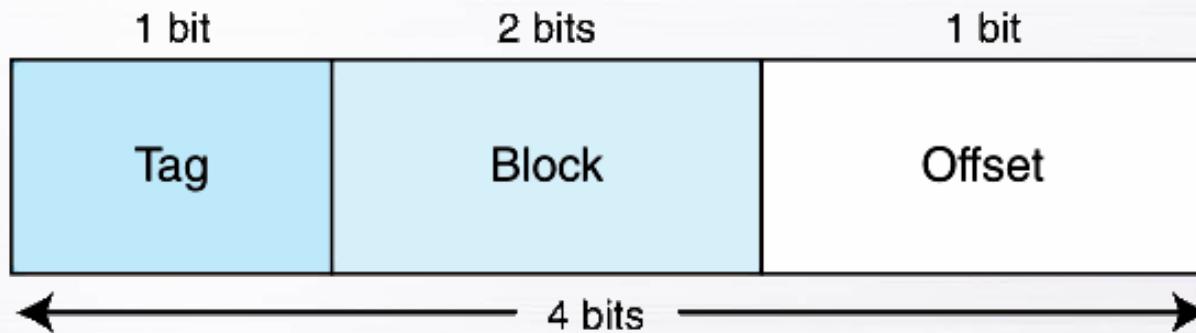
6.4 Cache Memory

- **EXAMPLE 6.2** Assume a byte-addressable memory consists of 2^{14} bytes, cache has 16 blocks, and each block has 8 bytes.
 - The number of memory blocks are: $\frac{2^{14}}{2^3} = 2^{11}$
 - Each main memory address requires 14 bits. Of this 14-bit address field, the rightmost 3 bits reflect the offset field
 - We need 4 bits to select a specific block in cache, so the block field consists of the middle 4 bits.
 - The remaining 7 bits make up the tag field.



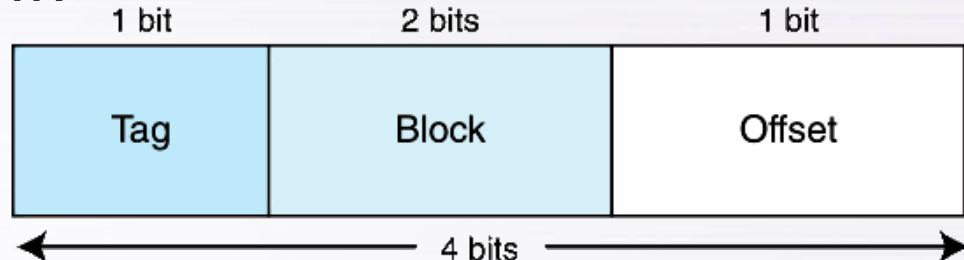
6.4 Cache Memory

- **EXAMPLE 6.3** Assume a byte-addressable memory consisting of 16 bytes divided into 8 blocks. Cache contains 4 blocks. We know:
 - A memory address has 4 bits.
 - The 4-bit memory address is divided into the fields below.



6.4 Cache Memory

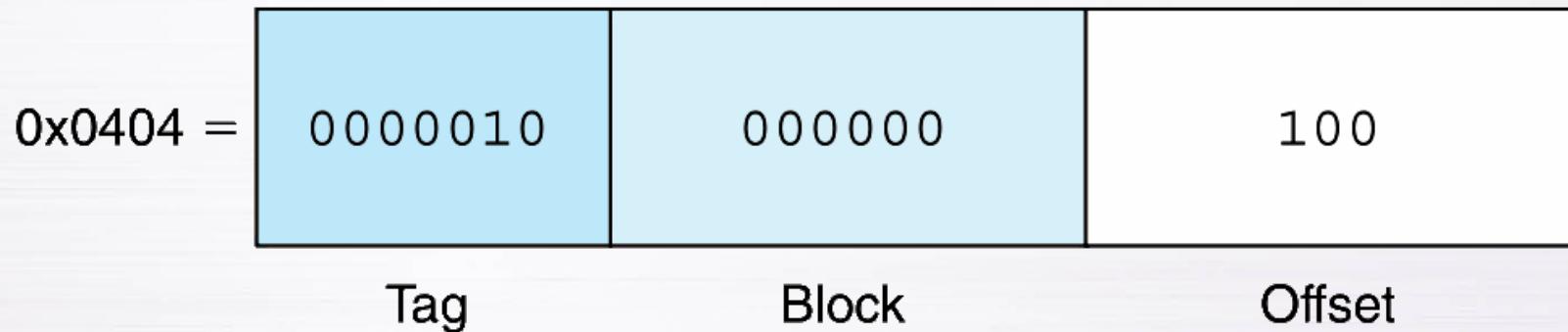
- EXAMPLE 6.3 Cont'd The mapping for memory references is shown below:



Main Memory	Maps To	Cache
(000) Block 0 (addresses 0x0, 0x1)	→	Block 0 (00)
(001) Block 1 (addresses 0x2, 0x3)	→	Block 1 (01)
(010) Block 2 (addresses 0x4, 0x5)	→	Block 2 (10)
(011) Block 3 (addresses 0x6, 0x7)	→	Block 3 (11)
(100) Block 4 (addresses 0x8, 0x9)	→	Block 0 (00)
(101) Block 5 (addresses 0xA, 0xB)	→	Block 1 (01)
(110) Block 6 (addresses 0xC, 0xD)	→	Block 2 (10)
(011) Block 7 (addresses 0xE, 0xF)	→	Block 3 (11)

6.4 Cache Memory

- EXAMPLE 6.4 Consider 16-bit memory addresses and 64 blocks of cache where each block contains 8 bytes. We have:
 - 3 bits for the offset
 - 6 bits for the block
 - 7 bits for the tag.
- A memory reference for 0x0404 maps as follows:



6.4 Cache Memory

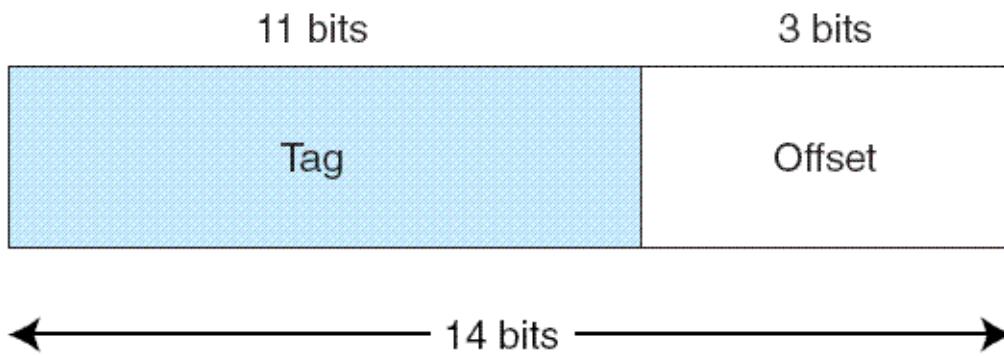
- In summary, direct mapped cache maps main memory blocks in a modular fashion to cache blocks. The mapping depends on:
- The number of bits in the main memory address (how many addresses exist in main memory)
- The number of blocks are in cache (which determines the size of the block field)
- How many addresses (either bytes or words) are in a block (which determines the size of the offset field)

6.4 Cache Memory

- Suppose instead of placing memory blocks in specific cache locations based on memory address, we could allow a block to go anywhere in cache.
- In this way, cache would have to fill up before any blocks are evicted.
- This is how *fully associative* cache works.
- A memory address is partitioned into only two fields: the tag and the word.

6.4 Cache Memory

- Suppose, as before, we have 14-bit memory addresses and a cache with 16 blocks, each block of size 8. The field format of a memory reference is:



- When the cache is searched, all tags are searched in parallel to retrieve the data quickly.
- This requires special, costly hardware.

6.4 Cache Memory

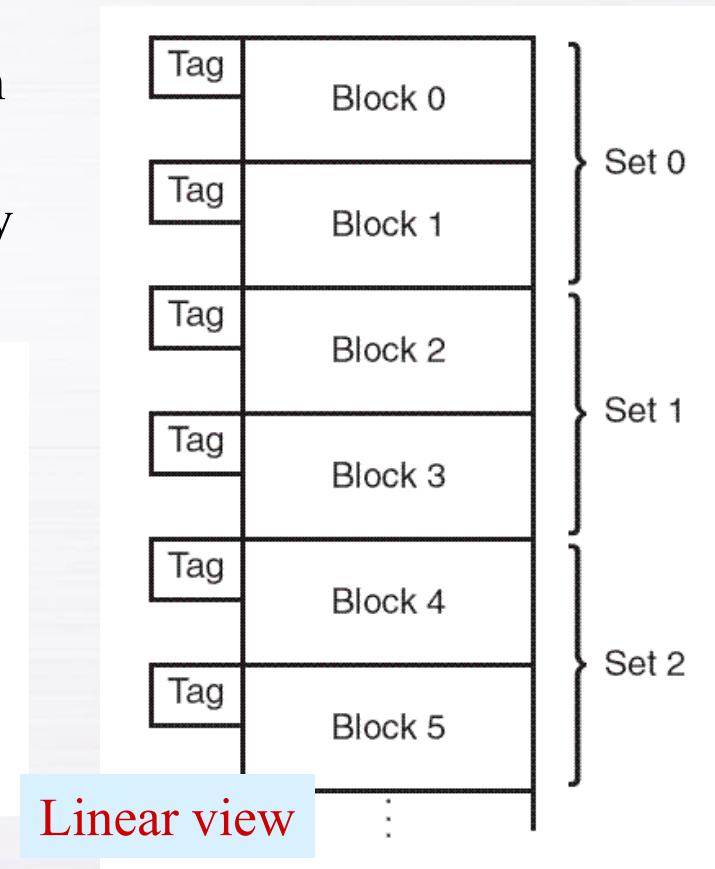
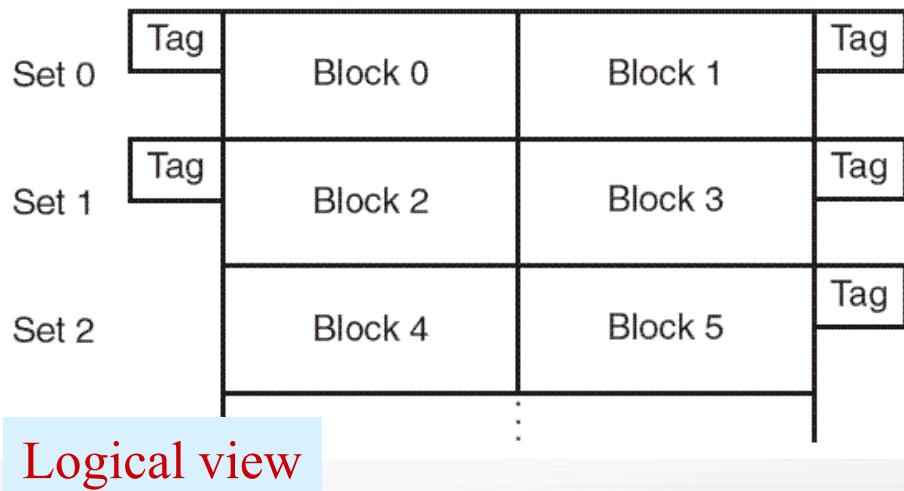
- You will recall that direct mapped cache evicts a block whenever another memory reference needs that block.
- With fully associative cache, we have no such mapping, thus we must devise an algorithm to determine which block to evict from the cache.
- The block that is evicted is the *victim block*.
- There are a number of ways to pick a victim, we will discuss them shortly.

6.4 Cache Memory

- Set associative cache combines the ideas of direct mapped cache and fully associative cache.
- An N -way set associative cache mapping is like direct mapped cache in that a memory reference maps to a particular location in cache.
- Unlike direct mapped cache, a memory reference maps to a set of several cache blocks, similar to the way in which fully associative cache works.
- Instead of mapping anywhere in the entire cache, a memory reference can map only to the subset of cache slots.

6.4 Cache Memory

- The number of cache blocks per set in set associative cache varies according to overall system design.
 - For example, a 2-way set associative cache can be conceptualized as shown in the schematic below.
 - Each set contains two different memory blocks.

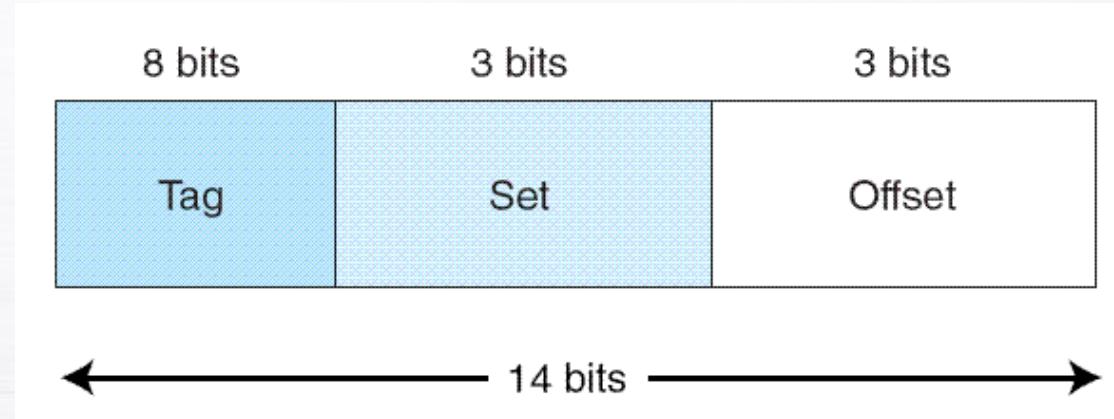


6.4 Cache Memory

- In set associative cache mapping, a memory reference is divided into three fields: tag, set, and offset.
- As with direct-mapped cache, the offset field chooses the word within the cache block, and the tag field uniquely identifies the memory address.
- The set field determines the set to which the memory block maps.

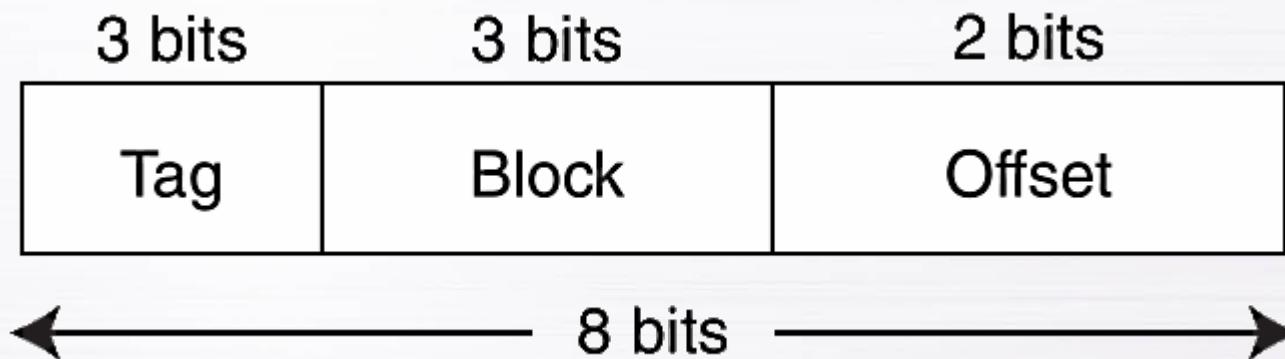
6.4 Cache Memory

- EXAMPLE 6.5 Suppose we are using 2-way set associative mapping with a word-addressable main memory of 2^{14} words and a cache with 16 blocks, where each block contains 8 words.
 - Cache has a total of 16 blocks, and each set has 2 blocks, then there are 8 sets in cache.
 - Thus, the set field is 3 bits, the offset field is 3 bits, and the tag field is 8 bits.



6.4 Cache Memory

- EXAMPLE 6.7 A byte-addressable computer with an 8-block cache of 4 bytes each, trace memory accesses: 0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01 for each mapping approach.
- The address format for direct mapped cache is:



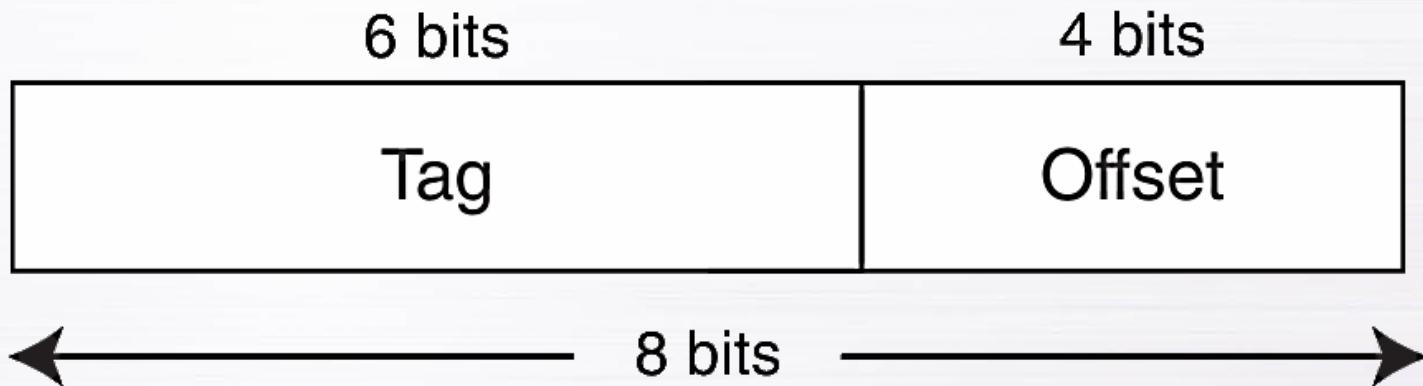
Our trace is on the next slide.

6.4 Cache Memory

Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	000 000 01	Miss	If we check cache block 000 for the tag 000, we find that it is not there. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into cache block 0 and store the tag 000 for that block.
0x04	000 001 00	Miss	We check cache block 001 for the tag 000, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into cache block 1 and store the tag 000 for that block.
0x09	000 010 01	Miss	A check of cache block 010 (2) for the tag 000 reveals a miss, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into cache block 2 and store the tag 000 for that block.
0x05	000 001 01	Hit	We check cache block 001 for the tag 000, and we find it. We then use the offset value 01 to get the exact byte we need.
0x14	000 101 00	Miss	We check cache block 101 (5) for the tag 000, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to cache block 5 and store the tag 000 with that block.
0x21	001 000 01	Miss	We check cache block 000 for the tag 001; we find tag 000 (which means this is not the correct block), so we overwrite the existing contents of this cache block by copying the data from addresses 0x20, 0x21, 0x22, and 0x23 into cache block 0 and storing the tag 001.
0x01	000 000 01	Miss	Although we have already fetched the block that contains address 0x01 once, it was overwritten when we fetched the block containing address 0x21 (if we look at block 000 in cache, we can see that its tag is 001, not 000). Therefore, we must overwrite the contents of block 0 in cache with the data from addresses 0x00, 0x01, 0x02, and 0x03, and store a tag of 000.

6.4 Cache Memory

- EXAMPLE 6.7 cont'd: A byte-addressable computer with an 8-block cache of 4 bytes each, trace memory accesses: 0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01 for each mapping approach.
- The address format for fully associative cache is:



Our trace is on the next slide.

6.4 Cache Memory

Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	000000 01	Miss	We search all of cache for the tag 000000, and we don't find it. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into cache block 0 and store the tag 000000 for that block.
0x04	000001 00	Miss	We search all of cache for the tag 000001, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into cache block 1 and store the tag 000001 for that block.
0x09	000010 01	Miss	We don't find the tag 000010 in cache, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into cache block 2 and store the tag 000010 for that block.
0x05	000001 01	Hit	We search all of cache for the tag 000001, and we find it stored with cache block 1. We then use the offset value 01 to get the exact byte we need.
0x14	000101 00	Miss	We search all of cache for the tag 000101, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to cache block 3 and store the tag 000101 with that block.
0x21	001000 01	Miss	We search all of cache for the tag 001000; we don't find it, so we copy the data from addresses 0x20, 0x21, 0x22, and 0x23 into cache block 4 and store the tag 001000.
0x01	000000 01	Hit	We search cache for the tag 000000 and find it with cache block 0. We use the offset of 1 to find the data we want.

6.4 Cache Memory

- EXAMPLE 6.7 cont'd: A byte-addressable computer with an 8-block cache of 4 bytes each, trace memory accesses: 0x01, 0x04, 0x09, 0x05, 0x14, 0x21, and 0x01 for each mapping approach.
- The address format for 2-way set-associative cache is:



Our trace is on the next slide.

6.4 Cache Memory

Address Reference	Binary Address (divided into fields)	Hit or Miss	Comments
0x01	0000 00 01	Miss	We search in set 0 of cache for a block with the tag 0000, and we find it is not there. So we copy the data from addresses 0x00, 0x01, 0x02, and 0x03 into set 0 (so now set 0 has one used block and one free block) and store the tag 0000 for that block. It does not matter which set we use; for consistency, we put the data in the first set.
0x04	0000 01 00	Miss	We search set 1 for a block with the tag 0000, and on finding it missing, we copy the data from addresses 0x04, 0x05, 0x06, and 0x07 into set 1, and store the tag 0000 for that block.
0x09	0000 10 01	Miss	We search set 2 (10) for a block with the tag 0000, but we don't find one, so we copy the data from addresses 0x08, 0x09, 0x0A, and 0x0B into set 2 and store the tag 0000 for that block.
0x05	0000 01 01	Hit	We search set 1 for a block with the tag 0000, and we find it. We then use the offset value 01 within that block to get the exact byte we need.
0x14	0001 01 00	Miss	We search set 1 for a block with the tag 0001, but it is not present. We copy addresses 0x14, 0x15, 0x16, and 0x17 to set 1 and store the tag 0001 with that block. Note that set 1 is now full.
0x21	0010 00 01	Miss	We search cache set 0 for a block with the tag 0010; we don't find it, so we copy the data from addresses 0x20, 0x21, 0x22, and 0x23 into set 0 and store the tag 0010. Note that set 0 is now full.
0x01	0000 00 01	Hit	We search cache set 0 for a block with the tag 0000, and we find it. We use the offset of 1 within that block to find the data we want.

6.4 Cache Memory

- With fully associative and set associative cache, a *replacement policy* is invoked when it becomes necessary to evict a block from cache.
- An *optimal* replacement policy would be able to look into the future to see which blocks won't be needed for the longest period of time.
- Although it is impossible to implement an optimal replacement algorithm, it is instructive to use it as a benchmark for assessing the efficiency of any other scheme we come up with.

6.4 Cache Memory

- The replacement policy that we choose depends upon the locality that we are trying to optimize-- usually, we are interested in temporal locality.
- A *least recently used* (LRU) algorithm keeps track of the last time that a block was assessed and evicts the block that has been unused for the longest period of time.
- The disadvantage of this approach is its complexity: LRU has to maintain an access history for each block, which ultimately slows down the cache.

6.4 Cache Memory

- *First-in, first-out* (FIFO) is a popular cache replacement policy.
- In FIFO, the block that has been in the cache the longest, regardless of when it was last used.
- A *random* replacement policy does what its name implies: It picks a block at random and replaces it with a new block.
- Random replacement can certainly evict a block that will be needed often or needed soon, but it never thrashes.

6.4 Cache Memory

- The performance of hierarchical memory is measured by its *effective access time* (EAT).
- EAT is a weighted average that takes into account the hit ratio and relative access times of successive levels of memory.
- The EAT for a two-level memory is given by:

$$\text{EAT} = H \times \text{Access}_C + (1-H) \times \text{Access}_{MM}$$

where H is the cache hit rate and Access_C and Access_{MM} are the access times for cache and main memory, respectively.

6.4 Cache Memory

- For example, consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit rate of 99%.
- Suppose access to cache and main memory occurs concurrently. (The accesses overlap.)
- The EAT is:

$$0.99(10\text{ns}) + 0.01(200\text{ns}) = 9.9\text{ns} + 2\text{ns} = 11\text{ns}.$$

6.4 Cache Memory

- For example, consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit rate of 99%.
- If the accesses do not overlap, the EAT is:

$$\begin{aligned}0.99(10\text{ns}) + 0.01(10\text{ns} + 200\text{ns}) \\= 9.9\text{ns} + 2.01\text{ns} = 12\text{ns}.\end{aligned}$$

- This equation for determining the effective access time can be extended to any number of memory levels, as we will see in later sections.

6.4 Cache Memory

- Caching depends upon programs exhibiting good locality.
 - Some object-oriented programs have poor locality owing to their complex, dynamic structures.
 - Arrays stored in column-major rather than row-major order can be problematic for certain cache organizations.
- With poor locality, caching can actually cause performance degradation rather than performance improvement.

6.4 Cache Memory

- Cache replacement policies must take into account *dirty blocks*, those blocks that have been updated while they were in the cache.
- Dirty blocks must be written back to memory. A *write policy* determines how this will be done.
- There are two types of write policies, *write through* and *write back*.
- Write through updates cache and main memory simultaneously on every write.

6.4 Cache Memory

- Write back (also called *copyback*) updates memory only when the block is selected for replacement.
- The disadvantage of write through is that memory must be updated with each cache write, which slows down the access time on updates. This slowdown is usually negligible, because the majority of accesses tend to be reads, not writes.
- The advantage of write back is that memory traffic is minimized, but its disadvantage is that memory does not always agree with the value in cache, causing problems in systems with many concurrent users.

6.4 Cache Memory

- The cache we have been discussing is called a *unified* or *integrated* cache where both instructions and data are cached.
- Many modern systems employ separate caches for data and instructions.
 - This is called a *Harvard* cache.
- The separation of data from instructions provides better locality, at the cost of greater complexity.
 - Simply making the cache larger provides about the same performance improvement without the complexity.

6.4 Cache Memory

- Cache performance can also be improved by adding a small associative cache to hold blocks that have been evicted recently.
 - This is called a *victim cache*.
- A *trace cache* is a variant of an instruction cache that holds decoded instructions for program branches, giving the illusion that noncontiguous instructions are really contiguous.

6.4 Cache Memory

- Most of today's small systems employ multilevel cache hierarchies.
- The levels of cache form their own small memory hierarchy.
- Level1 cache (8KB to 64KB) is situated on the processor itself.
 - Access time is typically about 4ns.
- Level 2 cache (64KB to 2MB) may be on the motherboard, or on an expansion card.
 - Access time is usually around 15 - 20ns.

6.4 Cache Memory

- In systems that employ three levels of cache, the Level 2 cache is placed on the same die as the CPU (reducing access time to about 10ns)
- Accordingly, the Level 3 cache (2MB to 256MB) refers to cache that is situated between the processor and main memory.
- Once the number of cache levels is determined, the next thing to consider is whether data (or instructions) can exist in more than one cache level.

6.4 Cache Memory

- If the cache system used an *inclusive* cache, the same data may be present at multiple levels of cache.
- *Strictly inclusive* caches guarantee that all data in a smaller cache also exists at the next higher level.
- *Exclusive* caches permit only one copy of the data.
- The tradeoffs in choosing one over the other involve weighing the variables of access time, memory size, and circuit complexity.

6.5 Virtual Memory

- Cache memory enhances performance by providing faster memory access speed.
- Virtual memory enhances performance by providing greater memory capacity, without the expense of adding main memory.
- Instead, a portion of a disk drive serves as an extension of main memory.
- If a system uses paging, virtual memory partitions main memory into individually managed *page frames*, that are written (*or paged*) to disk when they are not immediately needed.

6.5 Virtual Memory

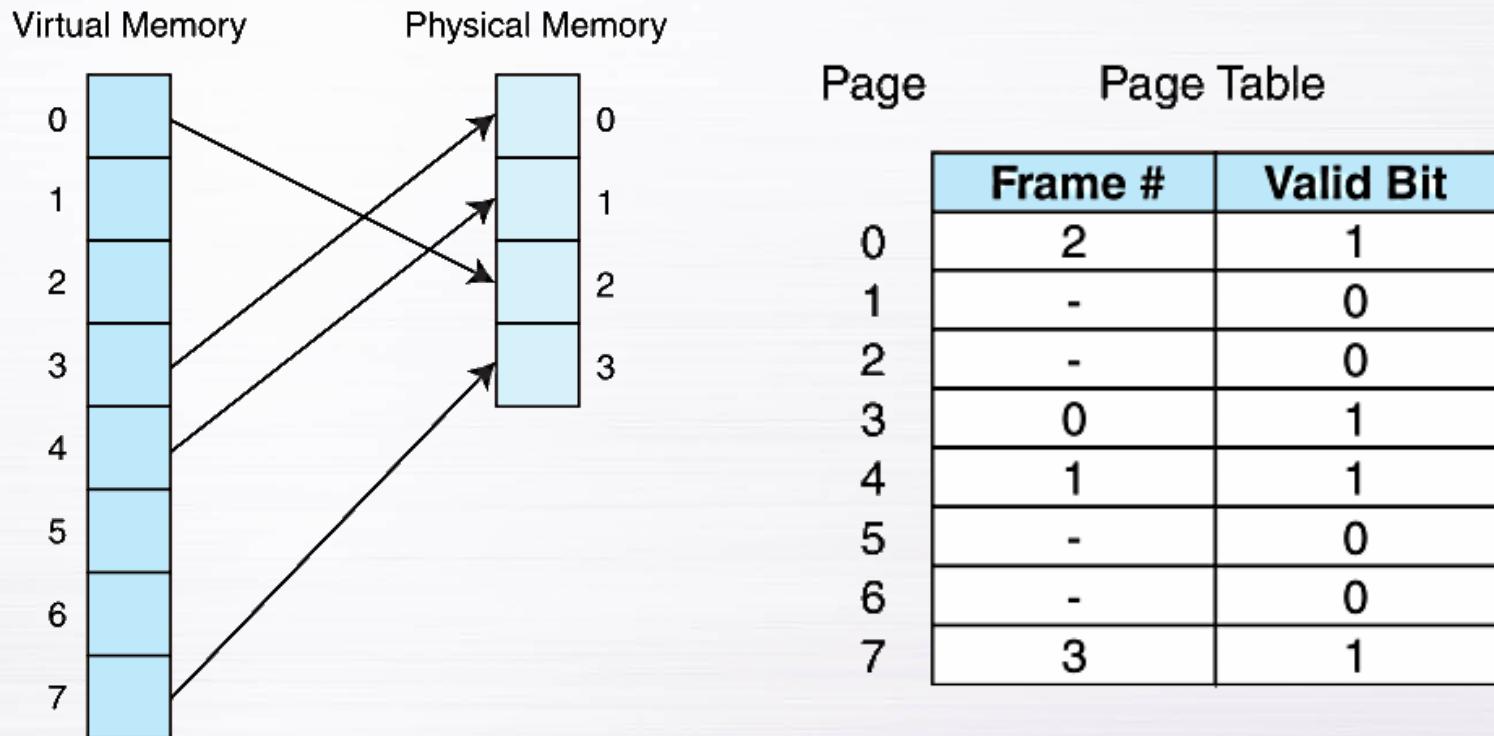
- A *physical address* is the actual memory address of physical memory.
- Programs create *virtual addresses* that are *mapped* to physical addresses by the memory manager.
- *Page faults* occur when a logical address requires that a page be brought in from disk.
- *Memory fragmentation* occurs when the paging process results in the creation of small, unusable clusters of memory addresses.

6.5 Virtual Memory

- Main memory and virtual memory are divided into equal sized pages.
- The entire address space required by a process need not be in memory at once. Some parts can be on disk, while others are in main memory.
- Further, the pages allocated to a process do not need to be stored contiguously-- either on disk or in memory.
- In this way, only the needed pages are in memory at any time, the unnecessary pages are in slower disk storage.

6.5 Virtual Memory

- Information concerning the location of each page, whether on disk or in memory, is maintained in a data structure called a *page table* (shown below).
- There is one page table for each active process.



6.5 Virtual Memory

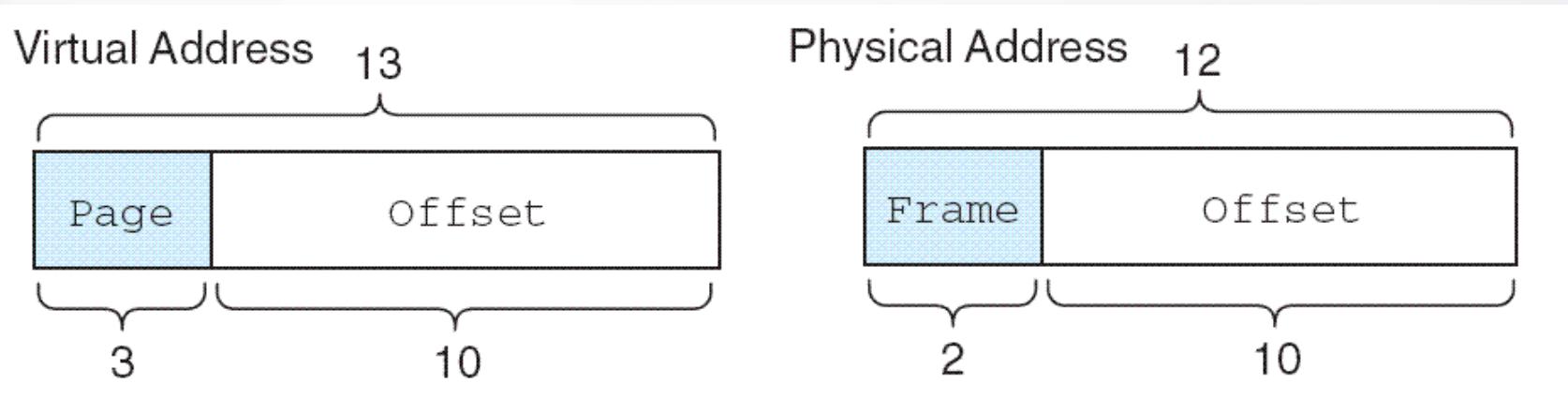
- When a process generates a virtual address, the operating system translates it into a physical memory address.
- To accomplish this, the virtual address is divided into two fields: A *page* field, and an *offset* field.
- The page field determines the page location of the address, and the offset indicates the location of the address within the page.
- The logical page number is translated into a physical page frame through a lookup in the page table.

6.5 Virtual Memory

- If the valid bit is zero in the page table entry for the logical address, this means that the page is not in memory and must be fetched from disk.
 - This is a page fault.
 - If necessary, a page is evicted from memory and is replaced by the page retrieved from disk, and the valid bit is set to 1.
- If the valid bit is 1, the virtual page number is replaced by the physical frame number.
- The data is then accessed by adding the offset to the physical frame number.

6.5 Virtual Memory

- As an example, suppose a system has a virtual address space of 8K and a physical address space of 4K, and the system uses byte addressing.
 - We have $2^{13}/2^{10} = 2^3$ virtual pages.
- A virtual address has 13 bits ($8K = 2^{13}$) with 3 bits for the page field and 10 for the offset, because the page size is 1024.
- A physical memory address requires 12 bits, the first two bits for the page frame and the trailing 10 bits the offset.



6.5 Virtual Memory

- Suppose we have the page table shown below.
- What happens when CPU generates address $5459_{10} = 1010101010011_2 = 0x1553$?

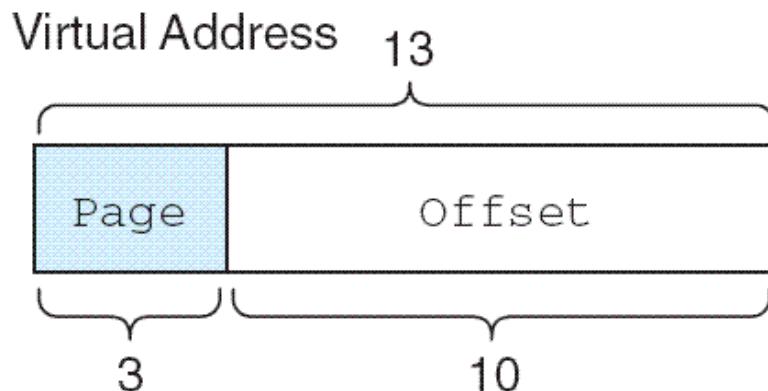
Page Table

Page	Frame	Valid Bit
0	-	0
1	3	1
2	0	1
3	-	0
4	-	0
5	1	1
6	2	1
7	-	0

Page	Addresses		
	Base 10	Base 16	
0	0 - 1023	0 -	3FF
1	1024 - 2047	400 -	7FF
2	2048 - 3071	800 -	BFF
3	3072 - 4095	C00 -	FFF
4	4096 - 5119	1000 -	13FF
5	5120 - 6143	1400 -	17FF
6	6144 - 7167	1800 -	1BFF
7	7168 - 8191	1C00 -	1FFF

6.5 Virtual Memory

- What happens when CPU generates address 5459_{10}
 $= 1010101010011_2 = 0x1553$?



The high-order 3 bits of the virtual address, 101 (5_{10}), provide the page number in the page table.

6.5 Virtual Memory

- The address 1010101010011_2 is converted to physical address $010101010011_2 = 0x1363$ because the page field 101 is replaced by frame number 01 through a lookup in the page table.

Page Table

Page	Frame	Valid Bit
0	-	0
1	3	1
2	0	1
3	-	0
4	-	0
5	1	1
6	2	1
7	-	0

Page	Addresses		
	Base 10	Base 16	
0 :	0 - 1023	0 -	3FF
1 :	1024 - 2047	400 -	7FF
2 :	2048 - 3071	800 -	BFF
3 :	3072 - 4095	C00 -	FFF
4 :	4096 - 5119	1000 -	13FF
5 :	5120 - 6143	1400 -	17FF
6 :	6144 - 7167	1800 -	1BFF
7 :	7168 - 8191	1C00 -	1FFF

6.5 Virtual Memory

- What happens when the CPU generates address 1000000000100_2 ?

Page Table

Page	Frame	Valid Bit
0	-	0
1	3	1
2	0	1
3	-	0
4	-	0
5	1	1
6	2	1
7	-	0

Page	Addresses		
	Base 10	Base 16	
0	0 - 1023	0 -	3FF
1	1024 - 2047	400 -	7FF
2	2048 - 3071	800 -	BFF
3	3072 - 4095	C00 -	FFF
4	4096 - 5119	1000 -	13FF
5	5120 - 6143	1400 -	17FF
6	6144 - 7167	1800 -	1BFF
7	7168 - 8191	1C00 -	1FFF

6.5 Virtual Memory

- We said earlier that effective access time (EAT) takes all levels of memory into consideration.
- Thus, virtual memory is also a factor in the calculation, and we also have to consider page table access time.
- Suppose a main memory access takes 200ns, the page fault rate is 1%, and it takes 10ms to load a page from disk. We have:

$$\text{EAT} = 0.99(200\text{ns} + 200\text{ns}) \cdot 0.01(10\text{ms}) = 100,396\text{ns}.$$

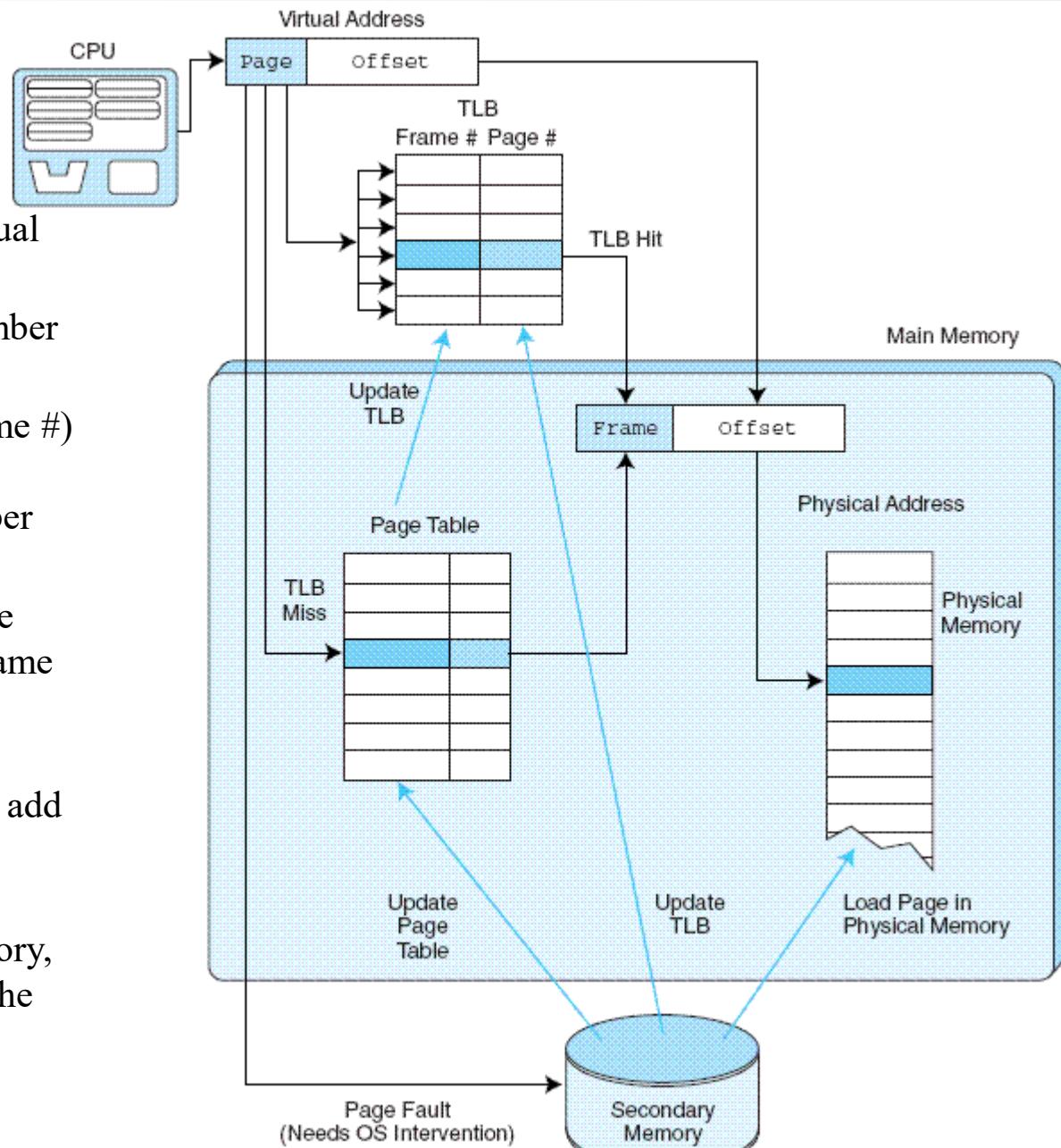
6.5 Virtual Memory

- Even if we had no page faults, the EAT would be 400ns because memory is always read twice: First to access the page table, and second to load the page from memory.
- Because page tables are read constantly, it makes sense to keep them in a special cache called a *translation look-aside buffer* (TLB).
- TLBs are a special associative cache that stores the mapping of virtual pages to physical pages.

The next slide shows address lookup steps when a TLB is involved.

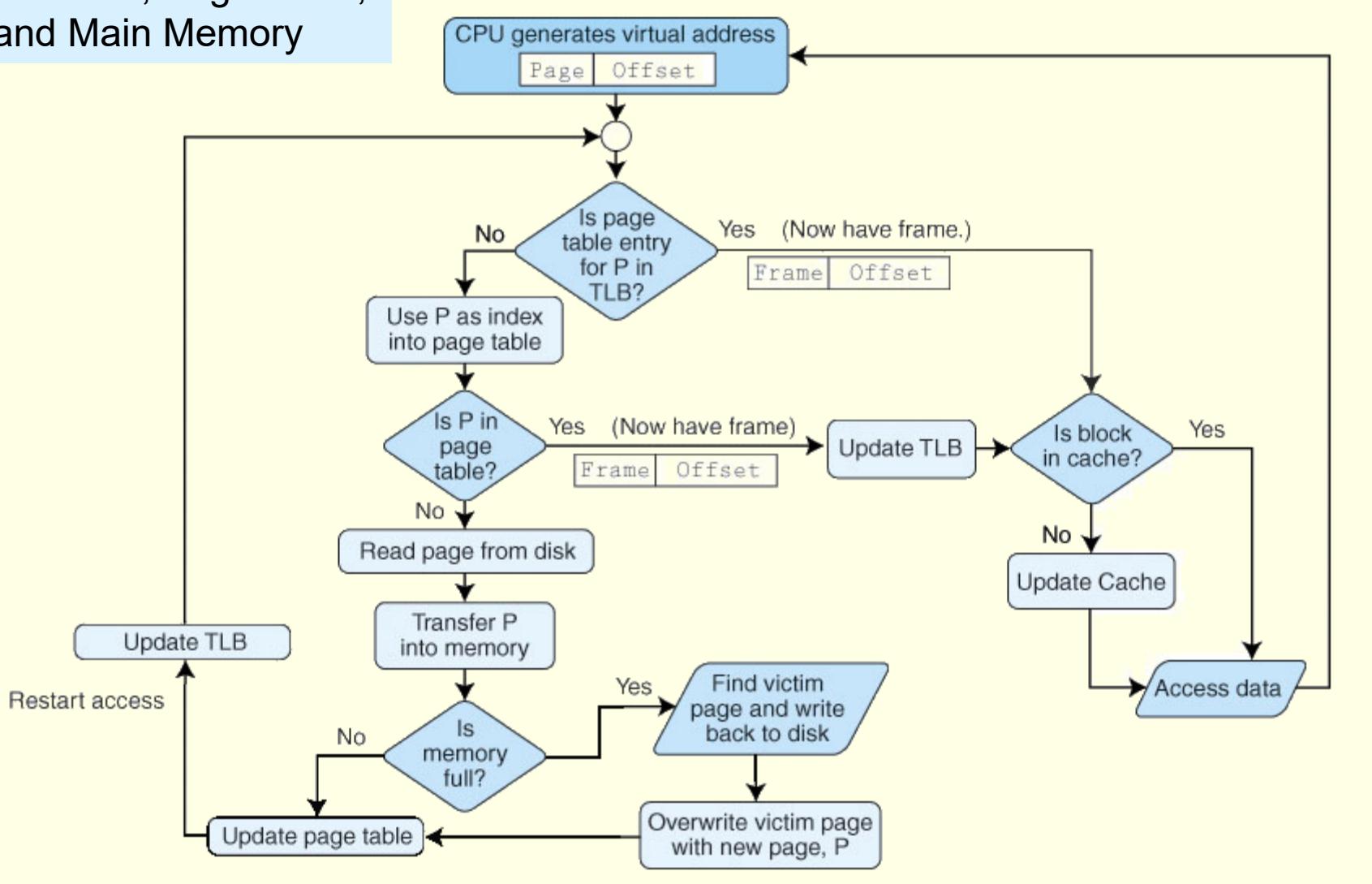
TLB lookup process

1. Extract the page number from the virtual address.
2. Extract the offset from the virtual address.
3. Search for the virtual page number in the TLB.
4. If the (virtual page #, page frame #) pair is found in the TLB, add the offset to the physical frame number and access the memory location.
5. If there is a TLB miss, go to the page table to get the necessary frame number.
If the page is in memory, use the corresponding frame number and add the offset to yield the physical address.
6. If the page is not in main memory, generate a page fault and restart the access when the page fault is complete.



Putting it all together:
The TLB, Page Table,
and Main Memory

6.5 Virtual Memory



6.5 Virtual Memory

- Another approach to virtual memory is the use of *segmentation*.
- Instead of dividing memory into equal-sized pages, virtual address space is divided into variable-length segments, often under the control of the programmer.
- A segment is located through its entry in a segment table, which contains the segment's memory location and a bounds limit that indicates its size.
- After a page fault, the operating system searches for a location in memory large enough to hold the segment that is retrieved from disk.

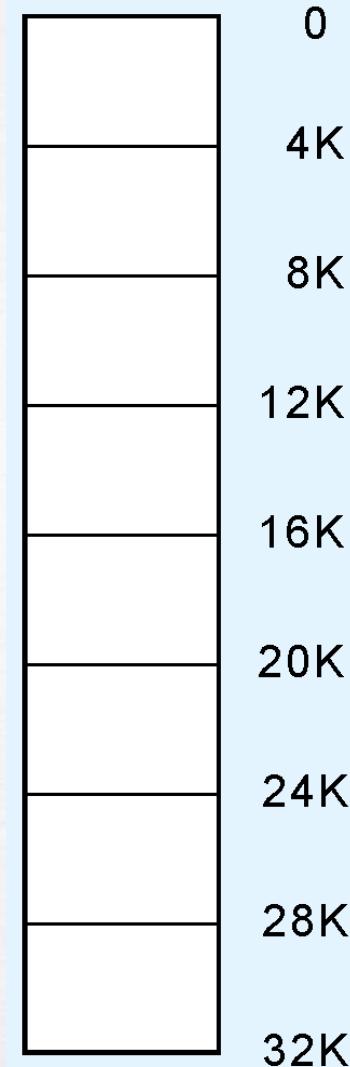
6.5 Virtual Memory

- Both paging and segmentation can cause fragmentation.
- Paging is subject to *internal* fragmentation because a process may not need the entire range of addresses contained within the page. Thus, there may be many pages containing unused fragments of memory.
- Segmentation is subject to *external* fragmentation, which occurs when contiguous chunks of memory become broken up as segments are allocated and deallocated over time.

The next slides illustrate internal and external fragmentation.

6.5 Virtual Memory

- Consider a small computer having 32K of memory.
- The 32K memory is divided into 8 page frames of 4K each.
- A schematic of this configuration is shown at the right.
- The numbers at the right are memory frame addresses.



6.5 Virtual Memory

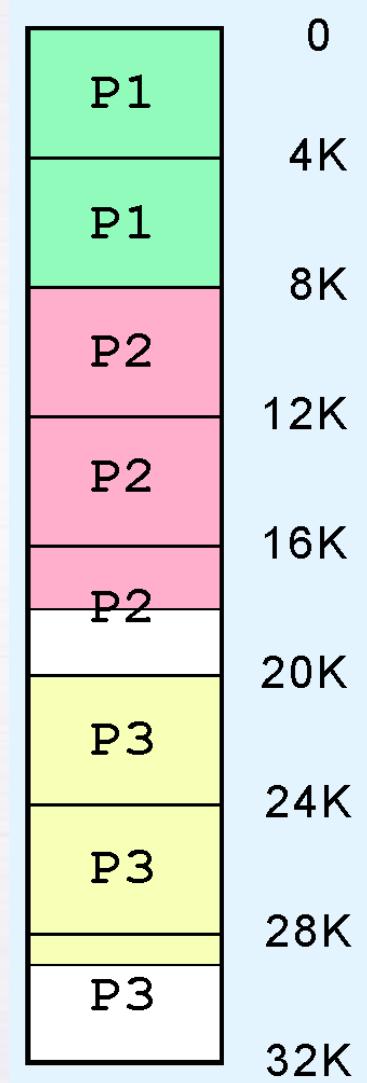
- Suppose there are four processes waiting to be loaded into the system with memory requirements as shown in the table.
- We observe that these processes require 31K of memory.

Process Name	Memory Needed
P1	8K
P2	10K
P3	9K
P4	4K

6.5 Virtual Memory

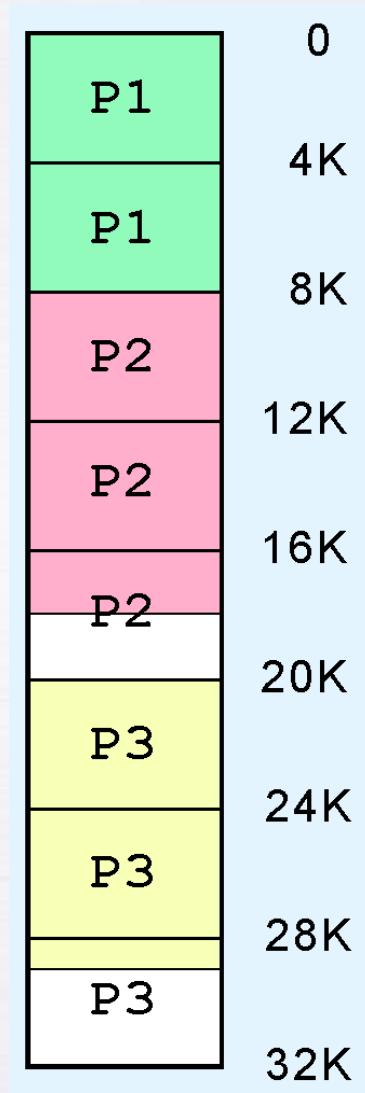
- When the first three processes are loaded, memory looks like this:
- All of the frames are occupied by three of the processes.

P1	8K
P2	10K
P3	9K
P4	4K



6.5 Virtual Memory

- Despite the fact that there are enough free bytes in memory to load the fourth process, P4 has to wait for one of the other three to terminate, because there are no unallocated frames.
- This is an example of *internal fragmentation*.



6.5 Virtual Memory

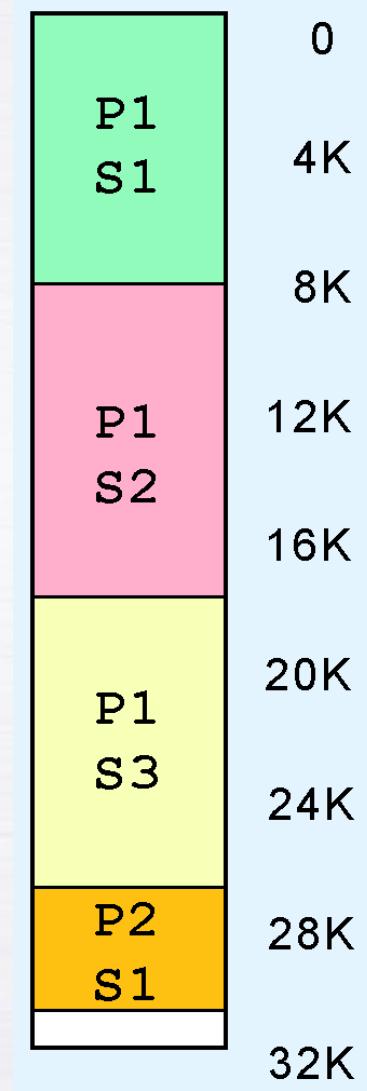
- Suppose that instead of frames, our 32K system uses segmentation.
- The memory segments of two processes is shown in the table at the right.
- The segments can be allocated anywhere in memory.

Process Name	Segment	Memory Needed
P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K

6.5 Virtual Memory

- All of the segments of P1 and one of the segments of P2 are loaded as shown at the right.
- Segment S2 of process P2 requires 11K of memory, and there is only 1K free, so it waits.

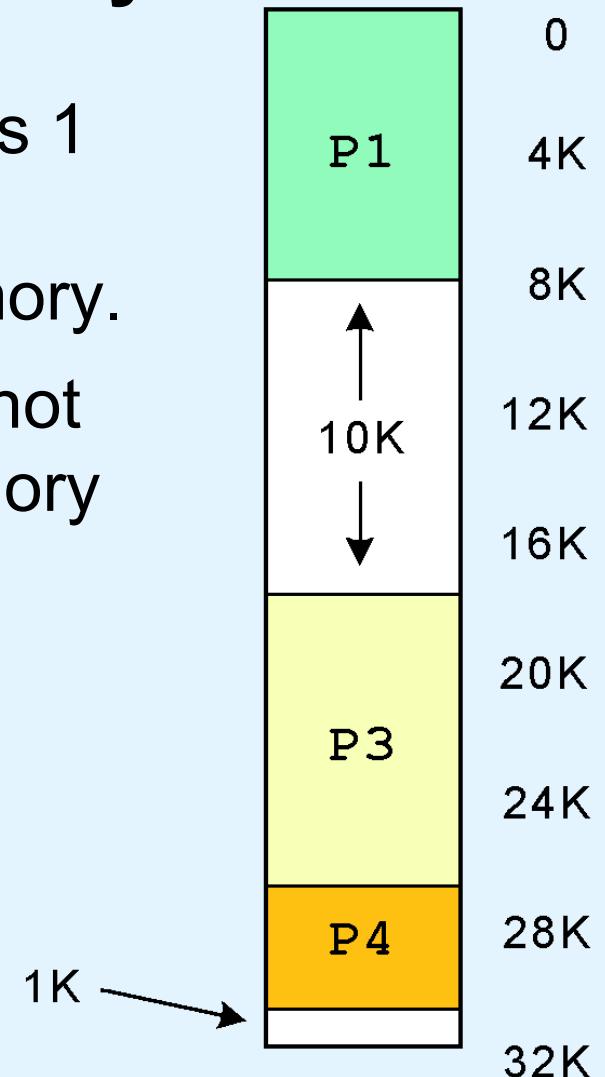
P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K



6.5 Virtual Memory

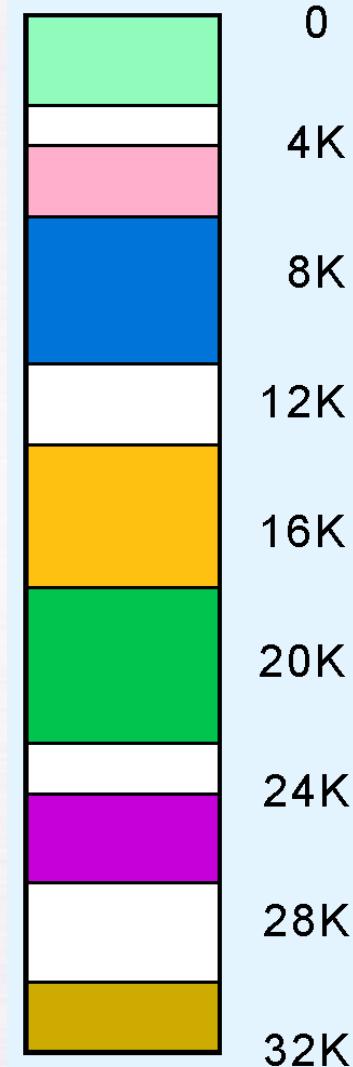
- Eventually, Segment 2 of Process 1 is no longer needed, so it is unloaded giving 11K of free memory.
- But Segment 2 of Process 2 cannot be loaded because the free memory is not contiguous.

P1	S1	8K
	S2	10K
	S3	9K
P2	S1	4K
	S2	11K



6.5 Virtual Memory

- Over time, the problem gets worse, resulting in small unusable blocks scattered throughout physical memory.
- This is an example of *external fragmentation*.
- Eventually, this memory is recovered through compaction, and the process starts over.



6.5 Virtual Memory

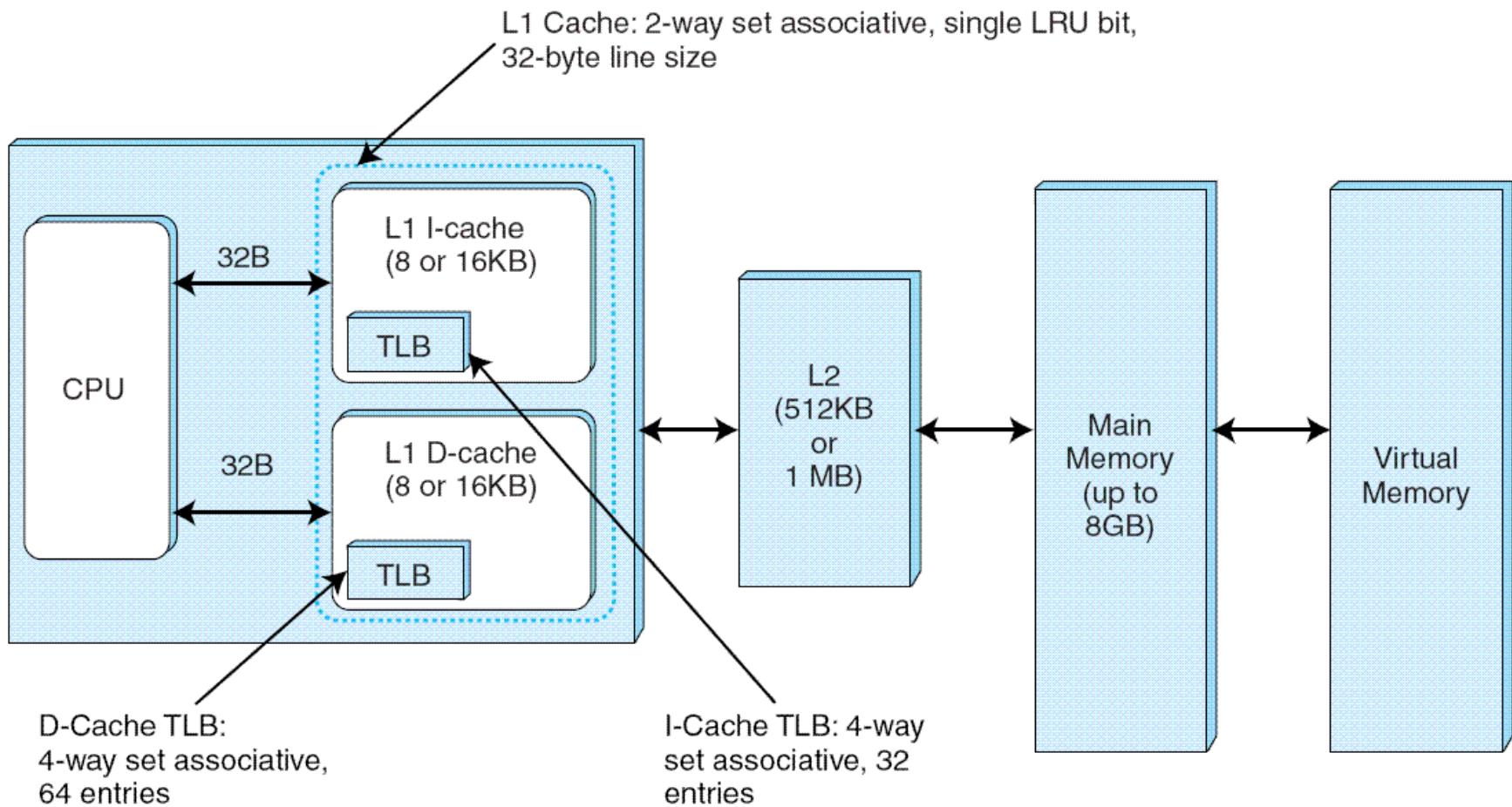
- Large page tables are cumbersome and slow, but with its uniform memory mapping, page operations are fast. Segmentation allows fast access to the segment table, but segment loading is labor-intensive.
- Paging and segmentation can be combined to take advantage of the best features of both by assigning fixed-size pages within variable-sized segments.
- Each segment has a page table. This means that a memory address will have three fields, one for the segment, another for the page, and a third for the offset.

6.6 A Real-World Example

- The Pentium architecture supports both paging and segmentation, and they can be used in various combinations including unpaged unsegmented, segmented unpaged, and unsegmented paged.
- The processor supports two levels of cache (L1 and L2), both having a block size of 32 bytes.
- The L1 cache is next to the processor, and the L2 cache sits between the processor and memory.
- The L1 cache is in two parts: an instruction cache (I-cache) and a data cache (D-cache).

The next slide shows this organization schematically.

6.6 A Real-World Example



Chapter 6 Conclusion

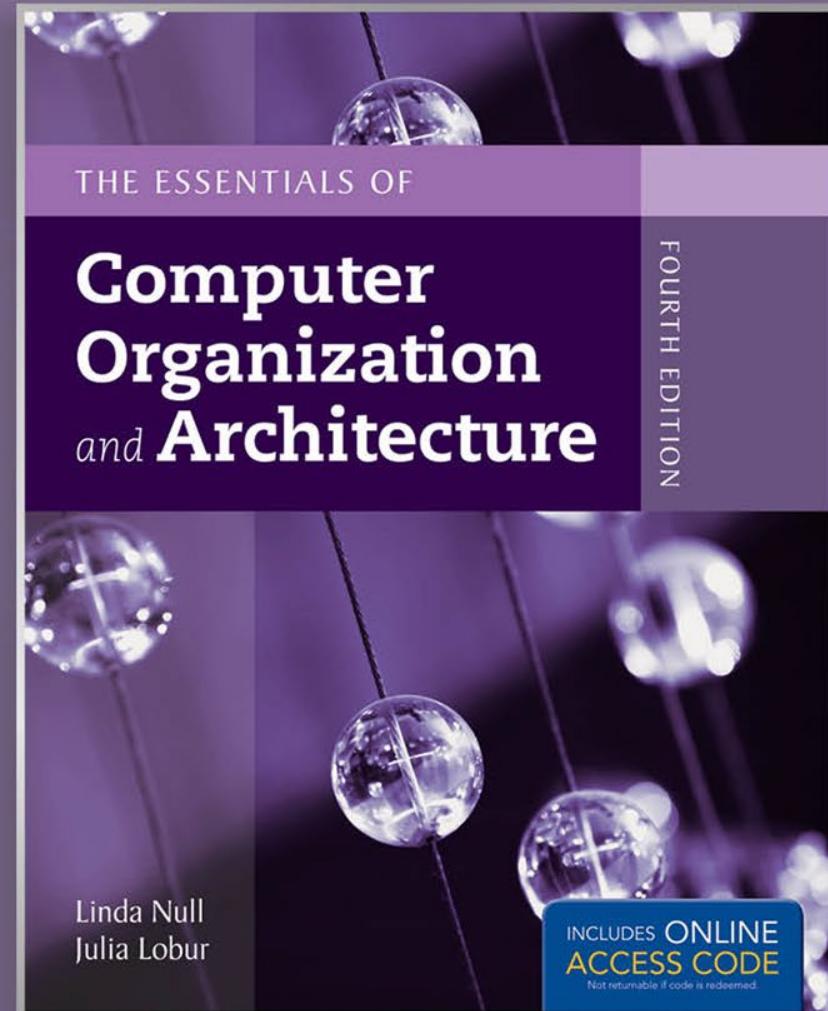
- Computer memory is organized in a hierarchy, with the smallest, fastest memory at the top and the largest, slowest memory at the bottom.
- Cache memory gives faster access to main memory, while virtual memory uses disk storage to give the illusion of having a large main memory.
- Cache maps blocks of main memory to blocks of cache memory. Virtual memory maps page frames to virtual pages.
- There are three general types of cache: Direct mapped, fully associative and set associative.

Chapter 6 Conclusion

- With fully associative and set associative cache, as well as with virtual memory, replacement policies must be established.
- Replacement policies include LRU, FIFO, or LFU. These policies must also take into account what to do with dirty blocks.
- All virtual memory must deal with fragmentation, internal for paged memory, external for segmented memory.

Chapter 7

Input/Output and Storage Systems



Chapter 7 Objectives

- Understand how I/O systems work, including I/O methods and architectures.
- Become familiar with storage media, and the differences in their respective formats.
- Understand how RAID improves disk performance and reliability, and which RAID systems are most useful today.
- Be familiar with emerging data storage technologies and the barriers that remain to be overcome.

7.1 Introduction

- Data storage and retrieval is one of the primary functions of computer systems.
 - One could easily make the argument that computers are more useful to us as data storage and retrieval devices than they are as computational machines.
- All computers have I/O devices connected to them, and to achieve good performance I/O should be kept to a minimum!
- In studying I/O, we seek to understand the different types of I/O devices as well as how they work.

7.2 I/O and Performance

- Sluggish I/O throughput can have a ripple effect, dragging down overall system performance.
 - This is especially true when virtual memory is involved.
- The fastest processor in the world is of little use if it spends most of its time waiting for data.
- If we really understand what's happening in a computer system we can make the best possible use of its resources.

7.3 Amdahl's Law

- The overall performance of a system is a result of the interaction of all of its components.
- System performance is most effectively improved when the performance of the most heavily used components is improved.
- This idea is quantified by Amdahl's Law:

$$S = \frac{1}{(1-f) + \frac{f}{k}}$$

where S is the overall speedup;
 f is the fraction of work performed
by a faster component; and
 k is the speedup of the faster
component.

7.3 Amdahl's Law

- Amdahl's Law gives us a handy way to estimate the performance improvement we can expect when we upgrade a system component.
- On a large system, suppose we can upgrade a CPU to make it 50% faster for \$10,000 or upgrade its disk drives for \$7,000 to make them 150% faster.
- Processes spend 70% of their time running in the CPU and 30% of their time waiting for disk service.
- An upgrade of which component would offer the greater benefit for the lesser cost?

7.3 Amdahl's Law

- The processor option offers a 30% speedup:

$$f = 0.70, \quad S = \frac{1}{(1 - 0.7) + 0.7/1.5}$$
$$k = 1.5$$

- And the disk drive option gives a 22% speedup:

$$f = 0.30, \quad S = \frac{1}{(1 - 0.3) + 0.3/2.5}$$
$$k = 2.5$$

- Each 1% of improvement for the processor costs \$333, and for the disk a 1% improvement costs \$318.

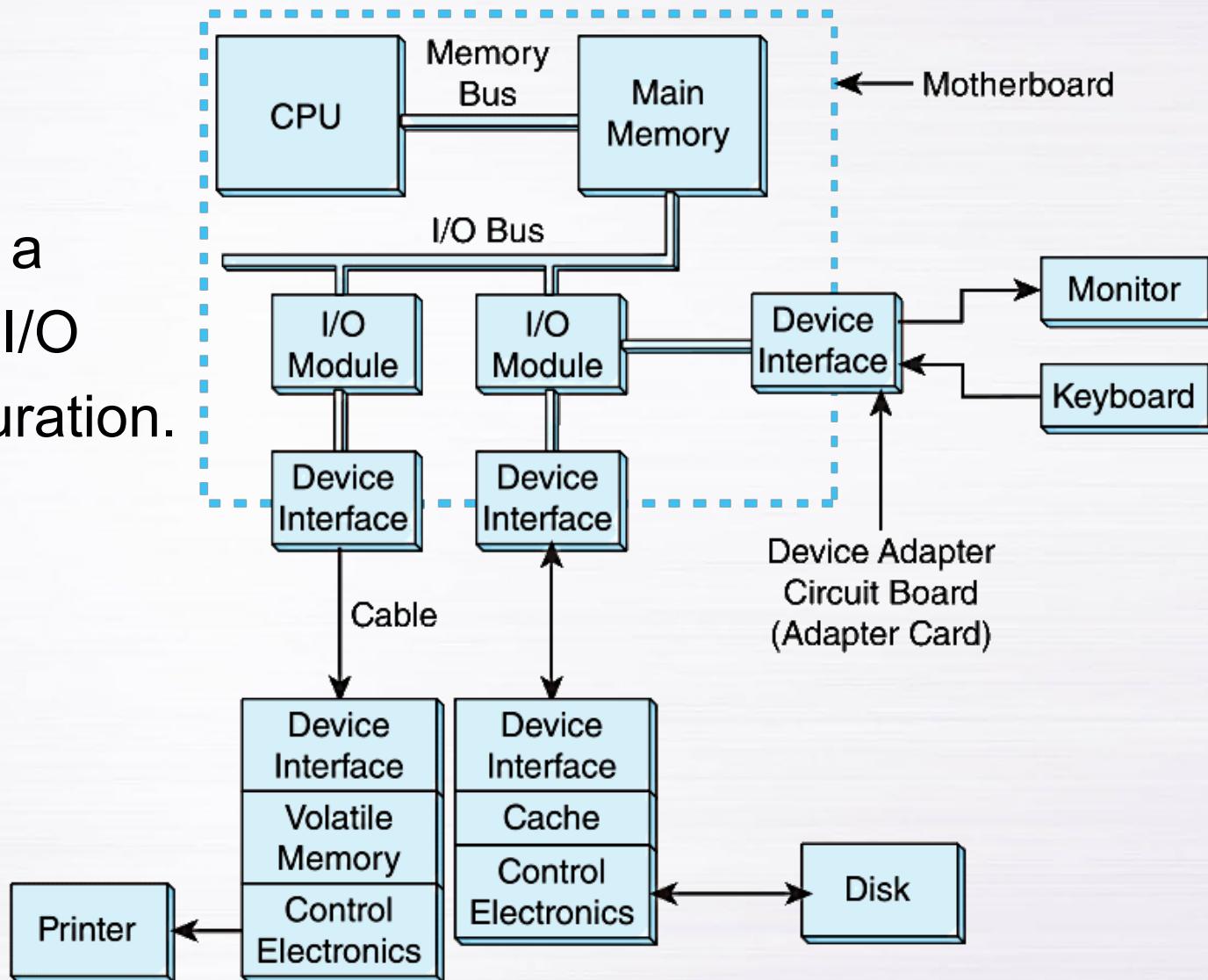
Should price/performance be your only concern?

7.4 I/O Architectures

- We define input/output as a subsystem of components that moves coded data between external devices and a host system.
- I/O subsystems include:
 - Blocks of main memory that are devoted to I/O functions.
 - Buses that move data into and out of the system.
 - Control modules in the host and in peripheral devices
 - Interfaces to external components such as keyboards and disks.
 - Cabling or communications links between the host system and its peripherals.

7.4 I/O Architectures

This is a model I/O configuration.



7.4 I/O Architectures

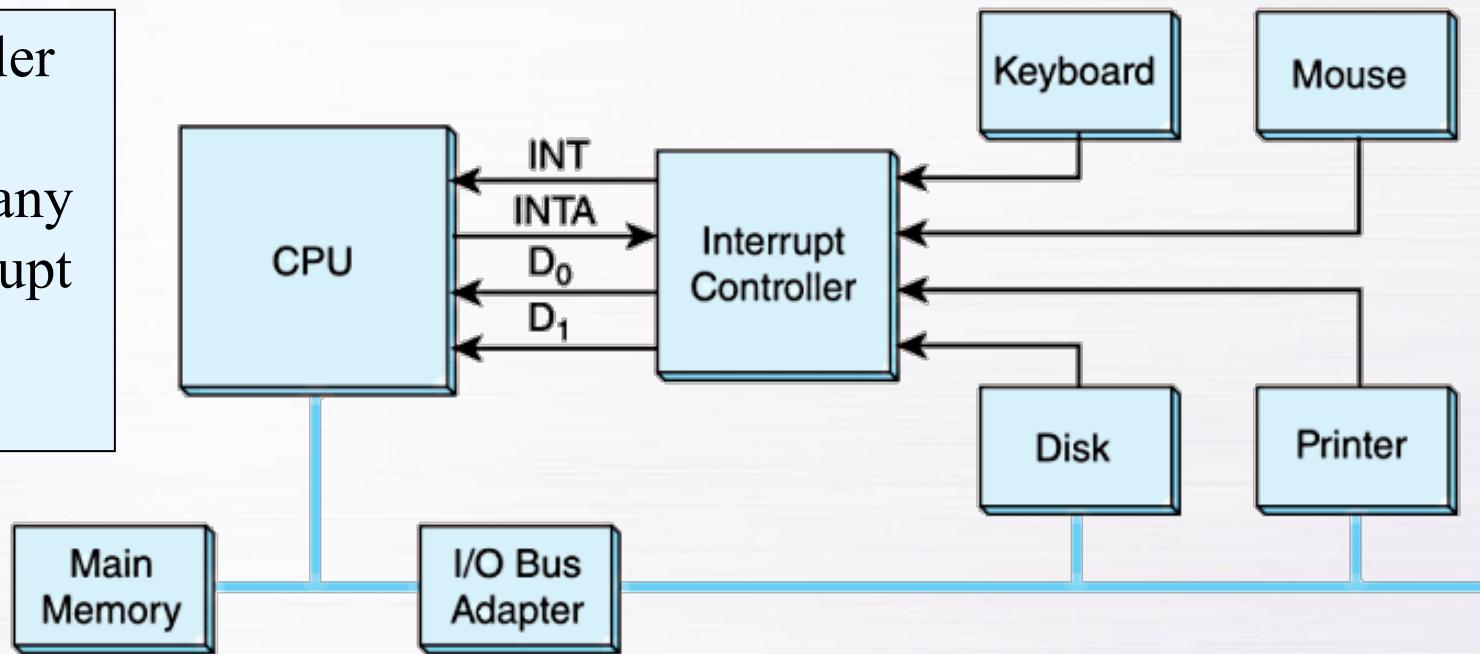
- I/O can be controlled in five general ways.
 - *Programmed I/O* reserves a register for each I/O device. Each register is continually polled to detect data arrival.
 - *Interrupt-Driven I/O* allows the CPU to do other things until I/O is requested.
 - *Memory-Mapped I/O* shares memory address space between I/O devices and program memory.
 - *Direct Memory Access (DMA)* offloads I/O processing to a special-purpose chip that takes care of the details.
 - *Channel I/O* uses dedicated I/O processors.

7.4 I/O Architectures

This is an idealized I/O subsystem that uses interrupts.

Each device connects its interrupt line to the interrupt controller.

The controller signals the CPU when any of the interrupt lines are asserted.

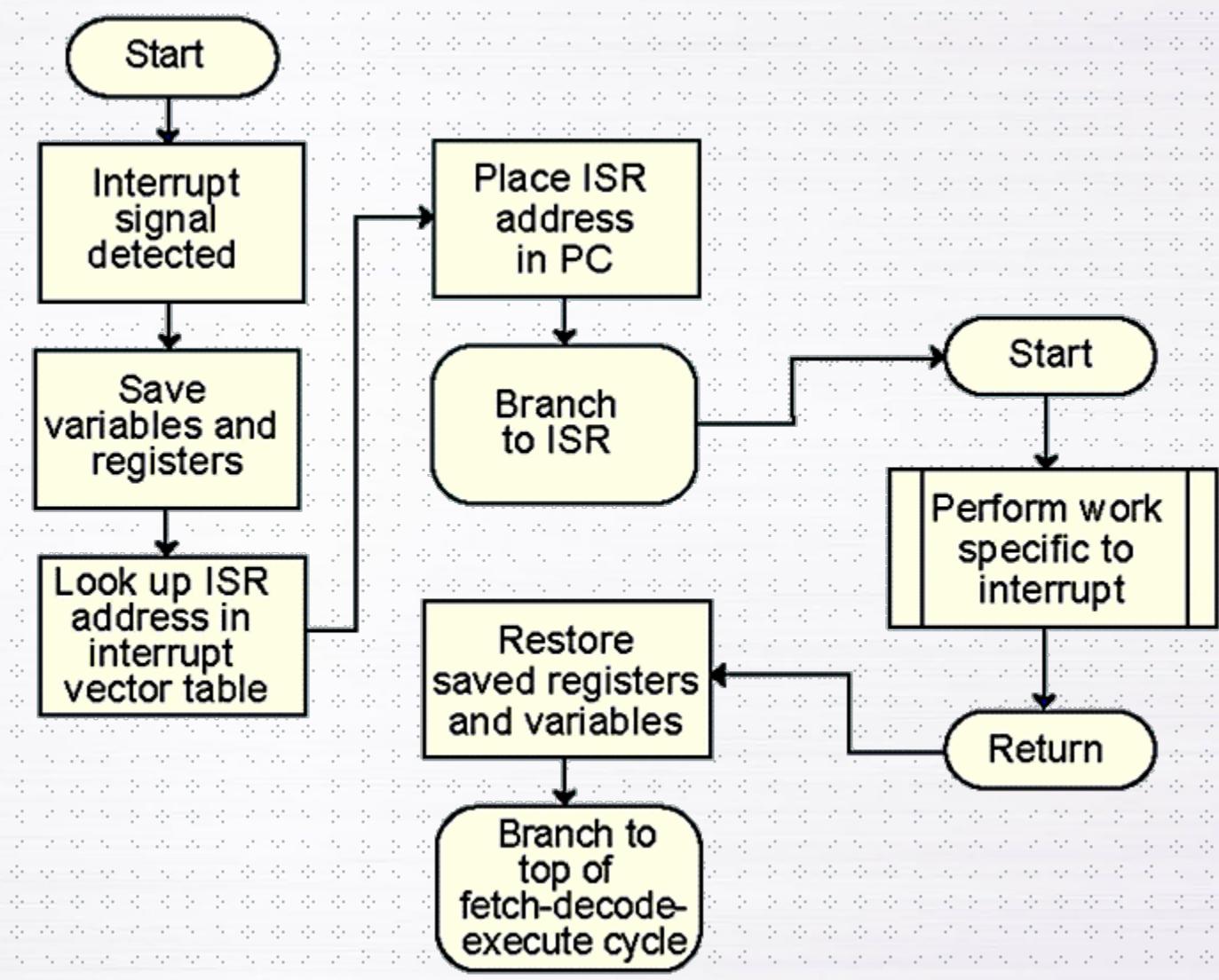


7.4 I/O Architectures

- Recall from Chapter 4 that in a system that uses interrupts, the status of the interrupt signal is checked at the top of the fetch-decode-execute cycle.
- The particular code that is executed whenever an interrupt occurs is determined by a set of addresses called *interrupt vectors* that are stored in low memory.
- The system state is saved before the interrupt service routine is executed and is restored afterward.

We provide a flowchart on the next slide.

7.4 I/O Architectures



7.4 I/O Architectures

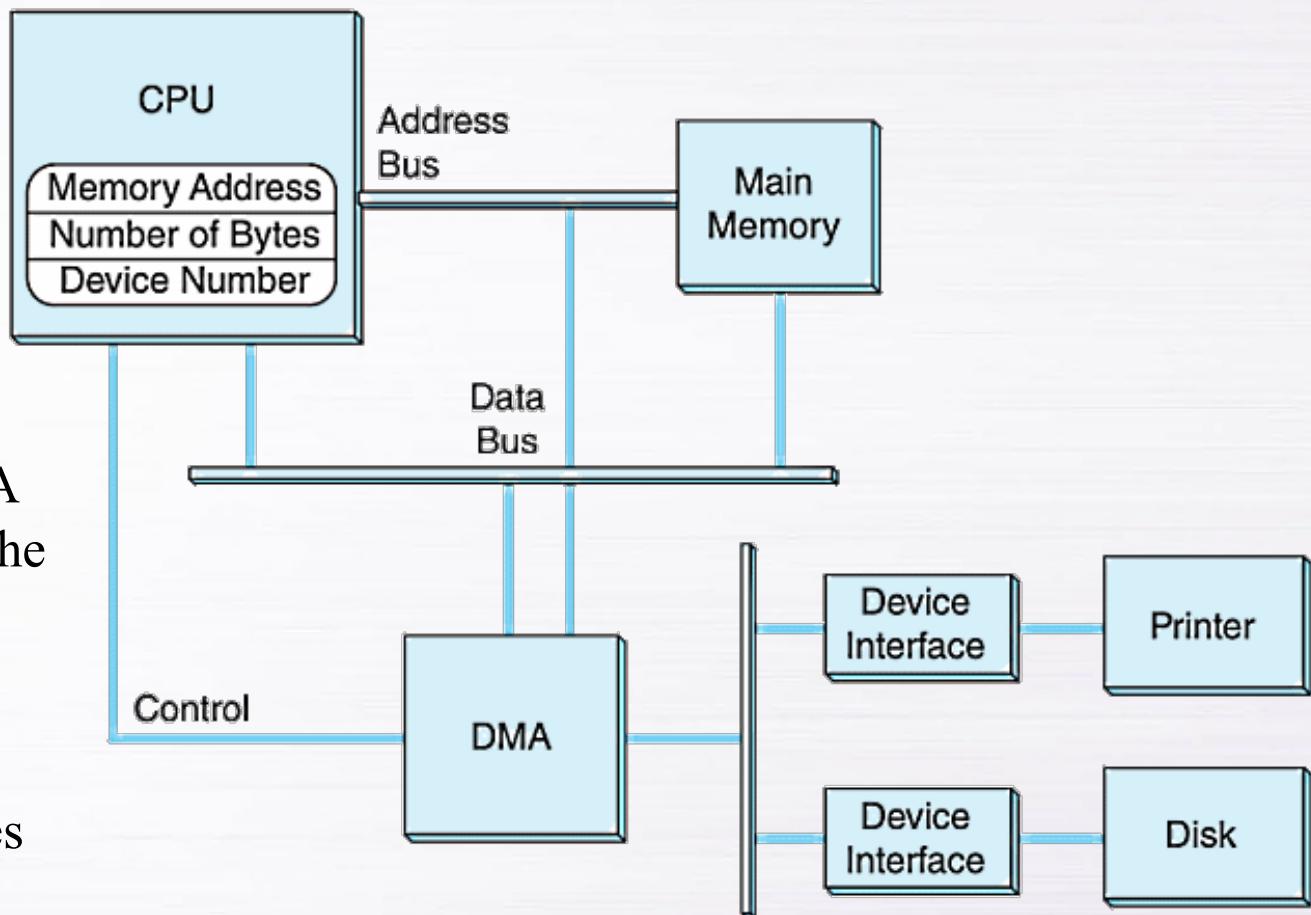
- In memory-mapped I/O devices and main memory share the same address space.
 - Each I/O device has its own reserved block of memory.
 - Memory-mapped I/O therefore looks just like a memory access from the point of view of the CPU.
 - Thus the same instructions to move data to and from both I/O and memory, greatly simplifying system design.
- In small systems the low-level details of the data transfers are offloaded to the I/O controllers built into the I/O devices.

7.4 I/O Architectures

This is a DMA configuration.

Notice that the DMA and the CPU share the bus.

The DMA runs at a higher priority and steals memory cycles from the CPU.



7.4 I/O Architectures

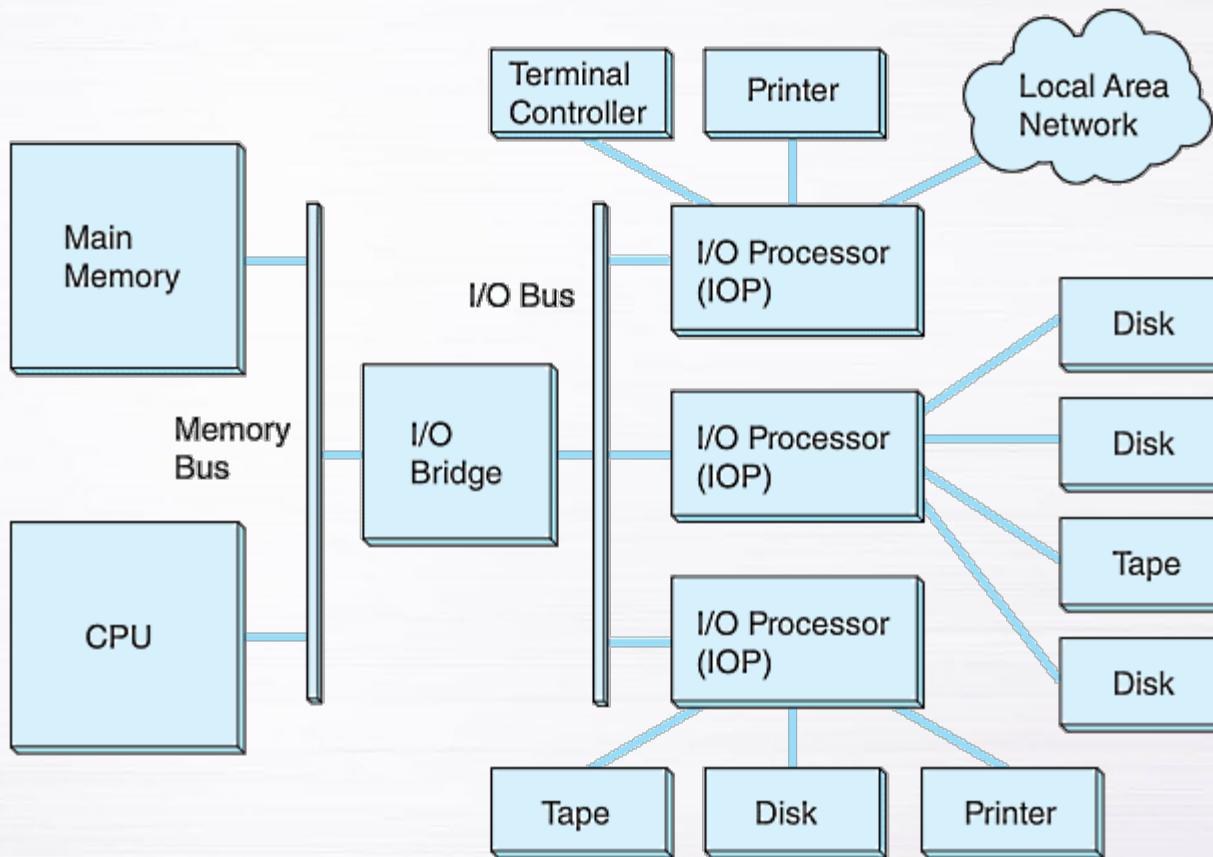
- Very large systems employ channel I/O.
- Channel I/O consists of one or more I/O processors (IOPs) that control various channel paths.
- Slower devices such as terminals and printers are combined (*multiplexed*) into a single faster channel.
- On IBM mainframes, multiplexed channels are called *multiplexor channels*, the faster ones are called *selector channels*.

7.4 I/O Architectures

- Channel I/O is distinguished from DMA by the intelligence of the IOPs.
- The IOP negotiates protocols, issues device commands, translates storage coding to memory coding, and can transfer entire files or groups of files independent of the host CPU.
- The host has only to create the program instructions for the I/O operation and tell the IOP where to find them.

7.4 I/O Architectures

- This is a channel I/O configuration.



7.4 I/O Architectures

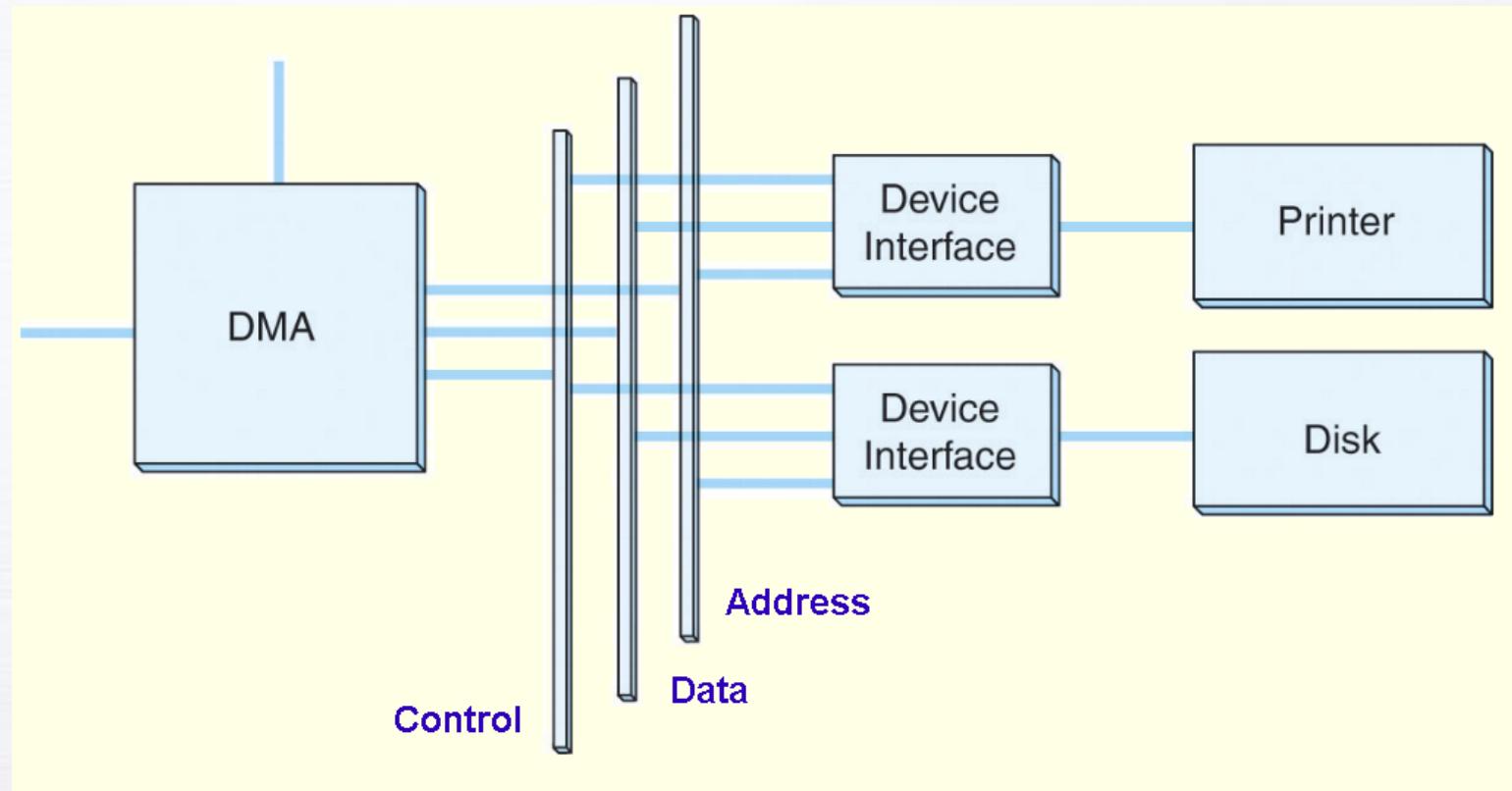
- Character I/O devices process one byte (or character) at a time.
 - Examples include modems, keyboards, and mice.
 - Keyboards are usually connected through an interrupt-driven I/O system.
- Block I/O devices handle bytes in groups.
 - Most mass storage devices (disk and tape) are block I/O devices.
 - Block I/O systems are most efficiently connected through DMA or channel I/O.

7.4 I/O Architectures

- I/O buses, unlike memory buses, operate asynchronously. Requests for bus access must be arbitrated among the devices involved.
- Bus control lines activate the devices when they are needed, raise signals when errors have occurred, and reset devices when necessary.
- The number of data lines is the *width* of the bus.
- A bus clock coordinates activities and provides bit cell boundaries.

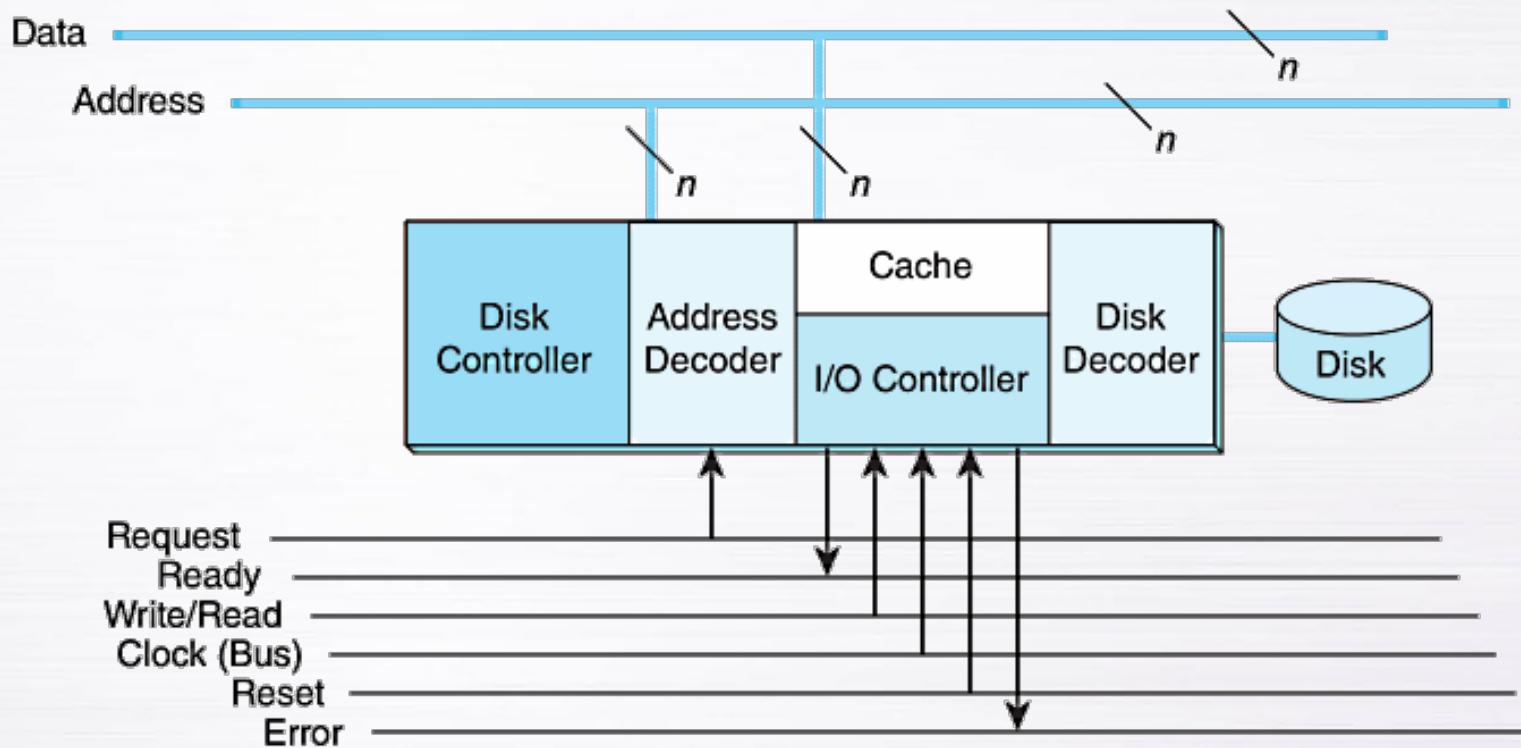
7.4 I/O Architectures

This is a generic DMA configuration showing how the DMA circuit connects to a data bus.



7.4 I/O Architectures

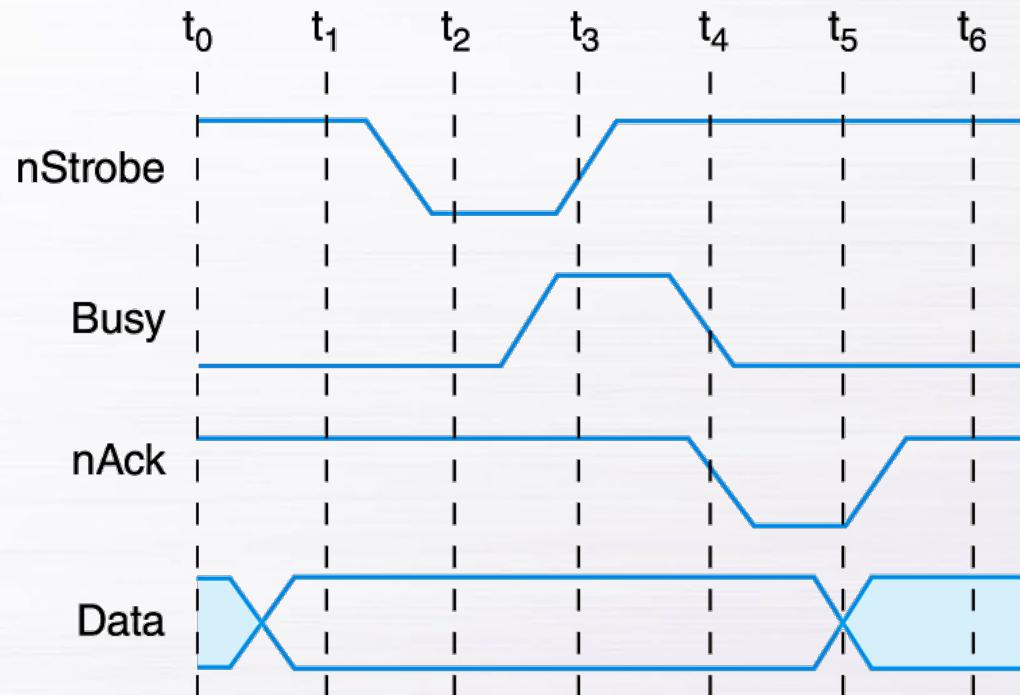
This is how a bus connects to a disk drive.



Timing
diagrams,
such as this
one, define
bus
operation in
detail.

7.5 Data Transmission Modes

- Bytes can be conveyed from one point to another by sending their encoding signals simultaneously using *parallel data transmission* or by sending them one bit at a time in *serial data transmission*.
 - Parallel data transmission for a printer resembles the signal protocol of a memory bus:



7.5 Data Transmission Modes

- In parallel data transmission, the interface requires one conductor for each bit.
- Parallel cables are fatter than serial cables.
- Compared with parallel data interfaces, serial communications interfaces:
 - Require fewer conductors.
 - Are less susceptible to attenuation.
 - Can transmit data farther and faster.

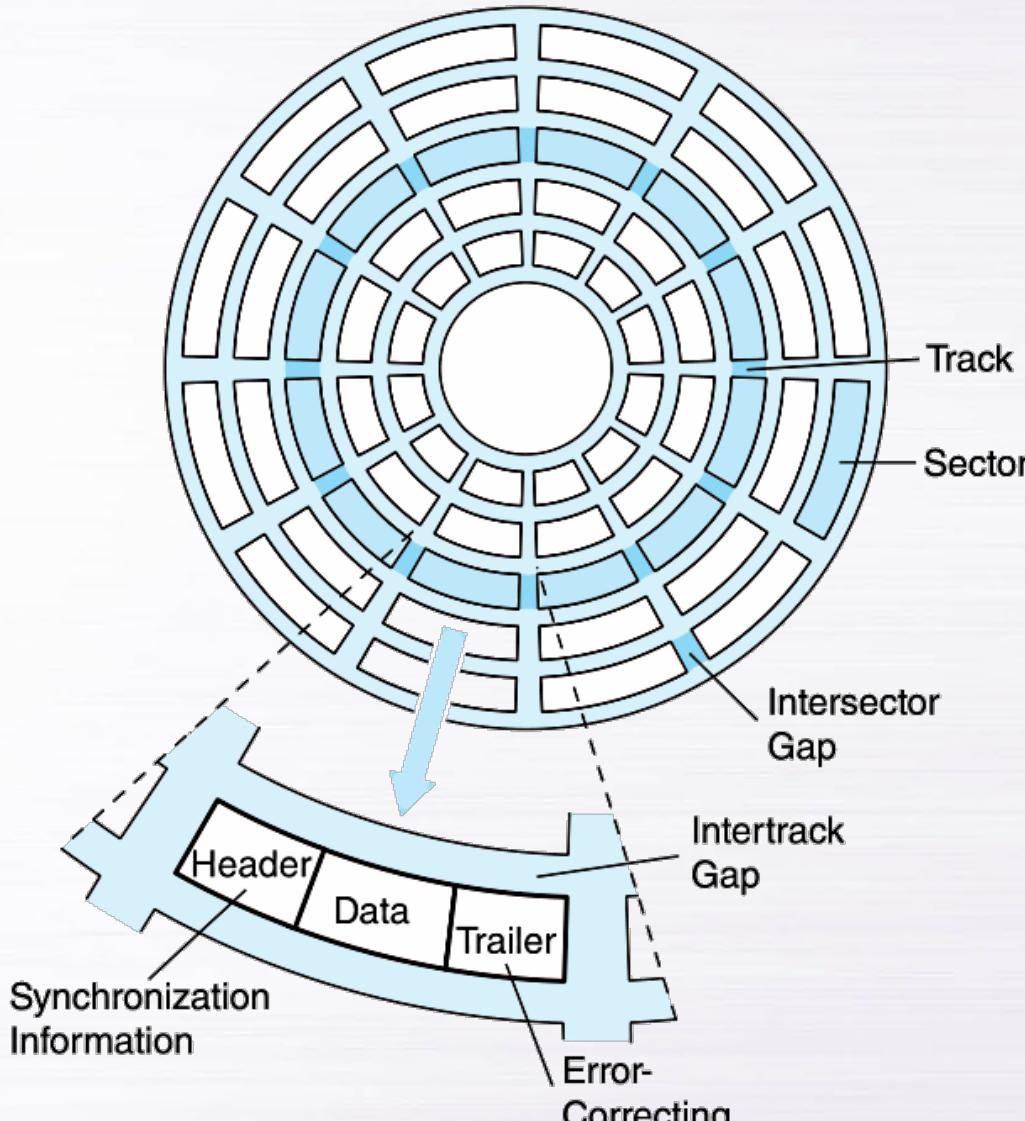
Serial communications interfaces are suitable for time-sensitive (*isochronous*) data such as voice and video.

7.6 Magnetic Disk Technology

- Magnetic disks offer large amounts of durable storage that can be accessed quickly.
- Disk drives are called *random* (or *direct*) access storage devices, because blocks of data can be accessed according to their location on the disk.
 - This term was coined when all other durable storage (e.g., tape) was sequential.
- Magnetic disk organization is shown on the following slide.

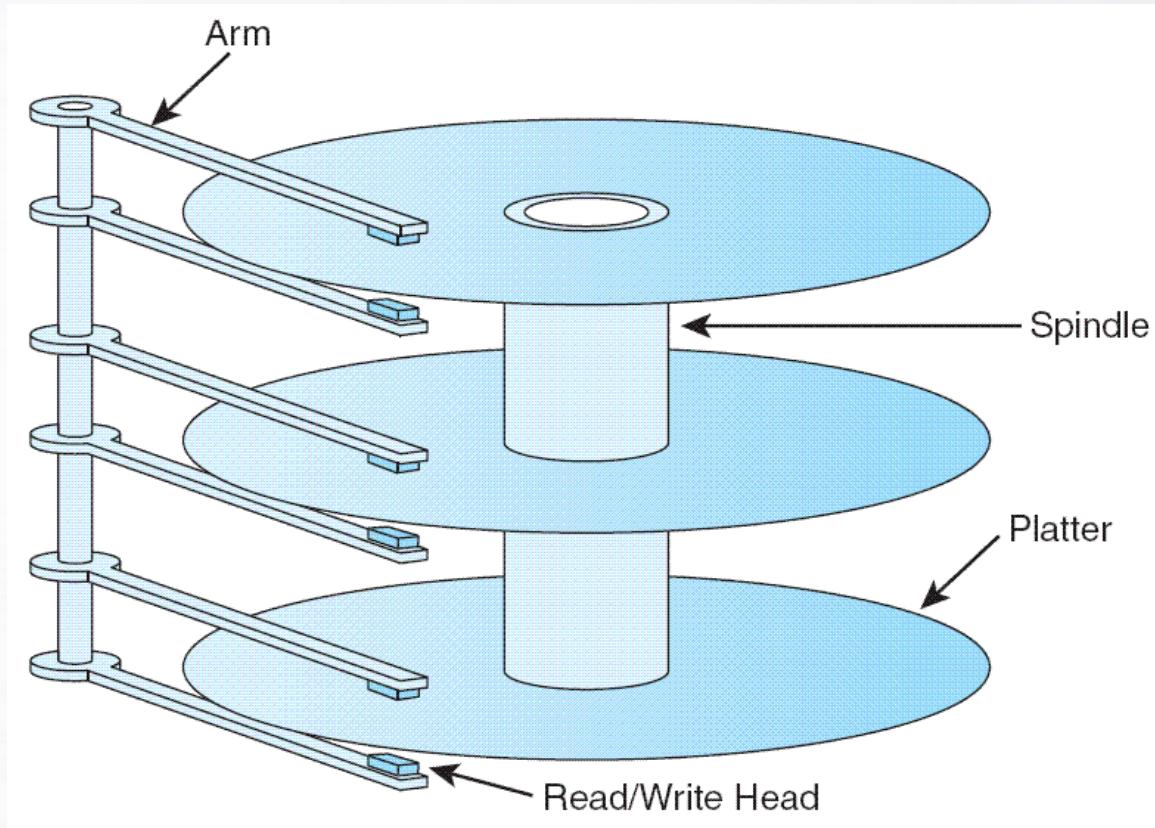
7.6 Magnetic Disk Technology

Disk tracks are numbered from the outside edge, starting with zero.



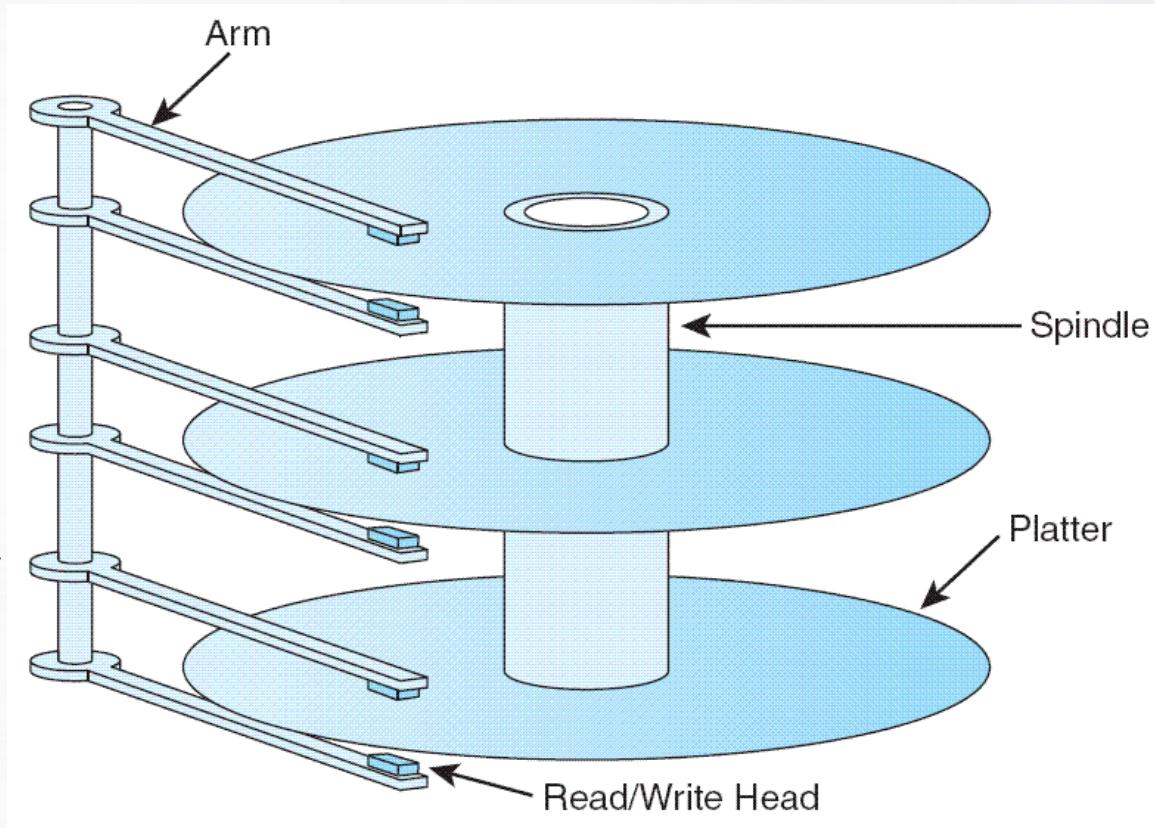
7.6 Magnetic Disk Technology

- Hard disk platters are mounted on spindles.
- Read/write heads are mounted on a comb that swings radially to read the disk.



7.6 Magnetic Disk Technology

- The rotating disk forms a logical cylinder beneath the read/write heads.
- Data blocks are addressed by their cylinder, surface, and sector.



7.6 Magnetic Disk Technology

- There are a number of electromechanical properties of hard disk drives that determine how fast its data can be accessed.
- *Seek time* is the time that it takes for a disk arm to move into position over the desired cylinder.
- *Rotational delay* is the time that it takes for the desired sector to move into position beneath the read/write head.
- Seek time + rotational delay = *access time*.

7.6 Magnetic Disk Technology

- *Transfer rate* gives us the rate at which data can be read from the disk.
- *Average latency* is a function of the rotational speed:
$$\frac{\frac{60 \text{ seconds}}{\text{disk rotation speed}} \times \frac{1000 \text{ ms}}{\text{second}}}{2}$$
- *Mean Time To Failure (MTTF)* is a statistically-determined value often calculated experimentally.
 - It usually doesn't tell us much about the actual expected life of the disk. *Design life* is usually more realistic.

Figure 7.15 in the text shows a sample disk specification.

7.6 Magnetic Disk Technology

- Low cost is the major advantage of hard disks.
- But their limitations include:
 - Very slow compared to main memory
 - Fragility
 - Moving parts wear out
- Reductions in memory cost enable the widespread adoption of *solid state drives, SSDs*.
 - Computers "see" SSDs as just another disk drive, but they store data in non-volatile *flash* memory circuits.
 - Flash memory is also found in memory sticks and MP3 players.

7.6 Magnetic Disk Technology

- SSD access time and transfer rates are *typically* 100 times faster than magnetic disk, but slower than onboard RAM by a factor of 100,000.
 - These numbers vary widely among manufacturers and interface methods.
- Unlike RAM, flash is block-addressable (like disk drives).
 - The duty cycle of flash is between 30,000 and 1,000,000 updates to a block.
 - Updates are spread over the entire medium through *wear leveling* to prolong the life of the SSD.

7.6 Magnetic Disk Technology

- SSD specifications share many common metrics with HDDs.
 - Clearly, there is no need for any metrics that concern spinning platters, such as rotational delay.
 - Compare Figs 7.15 with 7.17 in your text.
- Enterprise SSDs must maintain the highest degree of performance and reliability.
 - Onboard cache memories are backed up by capacitors that briefly hold a charge during a power failure, giving time to commit pending writes.

7.6 Magnetic Disk Technology

- The Joint Electron Devices Engineering Council (JEDEC) sets standards for SSD performance and reliability metrics. The most important are:
- Unrecoverable Bit Error Ratio (UBER) and terabytes written (TBW). TBW is a measure of disk endurance (or service life) and UBER is a measure of disk reliability.
 - UBER is calculated by dividing the number of data errors by the number of bits read using a simulated lifetime workload.
 - TBW is the number of terabytes that can be written to the disk before the disk fails to meet specifications for speed and error rates.

7.7 Optical Disks

- Optical disks provide large storage capacities very inexpensively.
- They come in a number of varieties including CD-ROM, DVD, and WORM.
- Many large computer installations produce document output on optical disk rather than on paper. This idea is called COLD-- *Computer Output Laser Disk*.
- It is estimated that optical disks can endure for a hundred years. Other media are good for only a decade-- at best.

7.7 Optical Disks

- CD-ROMs were designed by the music industry in the 1980s, and later adapted to data.
- This history is reflected by the fact that data is recorded in a single spiral track, starting from the center of the disk and spanning outward.
- Binary ones and zeros are delineated by bumps in the polycarbonate disk substrate. The transitions between pits and lands define binary ones.
- If you could unravel a full CD-ROM track, it would be nearly five miles long!

7.7 Optical Disks

- The logical data format for a CD-ROM is much more complex than that of a magnetic disk. (See the text for details.)
- Different formats are provided for data and music.
- Two levels of error correction are provided for the data format.
- Because of this, a CD holds at most 650MB of data, but can contain as much as 742MB of music.

7.7 Optical Disks

- DVDs can be thought of as quad-density CDs.
 - Varieties include single sided, single layer, single sided double layer, double sided double layer, and double sided double layer.
- Where a CD-ROM can hold at most 650MB of data, DVDs can hold as much as 17GB.
- One of the reasons for this is that DVD employs a laser that has a shorter wavelength than the CD's laser.
- This allows pits and lands to be closer together and the spiral track to be wound tighter.

7.7 Optical Disks

- A shorter wavelength light can read and write bytes in greater densities than can be done by a longer wavelength laser.
- This is one reason that DVD's density is greater than that of CD.
- The 405 nm wavelength of blue-violet light is much shorter than either red (750 nm) or orange (650 nm).
- The manufacture of blue-violet lasers can now be done economically, bringing about the next generation of laser disks.

7.7 Optical Disks

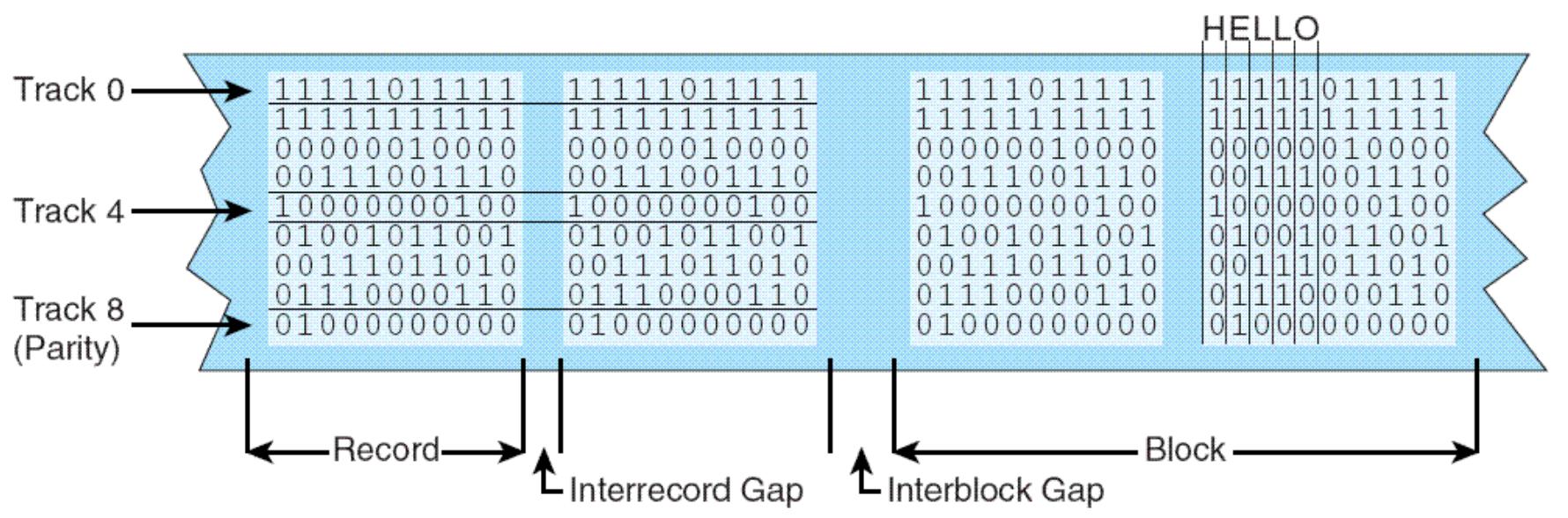
- The Blu-Ray disc format won market dominance over HD-CD owing mainly to the influence of Sony.
 - HD-CDs are backward compatible with DVD, but hold less data.
- Blu-Ray was developed by a consortium of nine companies that includes Sony, Samsung, and Pioneer.
 - Maximum capacity of a single layer Blu-Ray disk is 25GB.
 - Multiple layers can be "stacked" up to six deep.
 - Only double-layer disks are available for home use.

7.7 Optical Disks

- Blue-violet laser disks are also used in the data center.
- The intention is to provide a means for long term data storage and retrieval.
- Two types are now dominant:
 - Sony's Professional Disk for Data (PDD) that can store 23GB on one disk and
 - Plasmon's Ultra Density Optical (UDO) that can hold up to 30GB.
- It is too soon to tell which of these technologies will emerge as the winner.

7.8 Magnetic Tape

- First-generation magnetic tape was not much more than wide analog recording tape, having capacities under 11MB.
 - Data was usually written in nine vertical tracks:

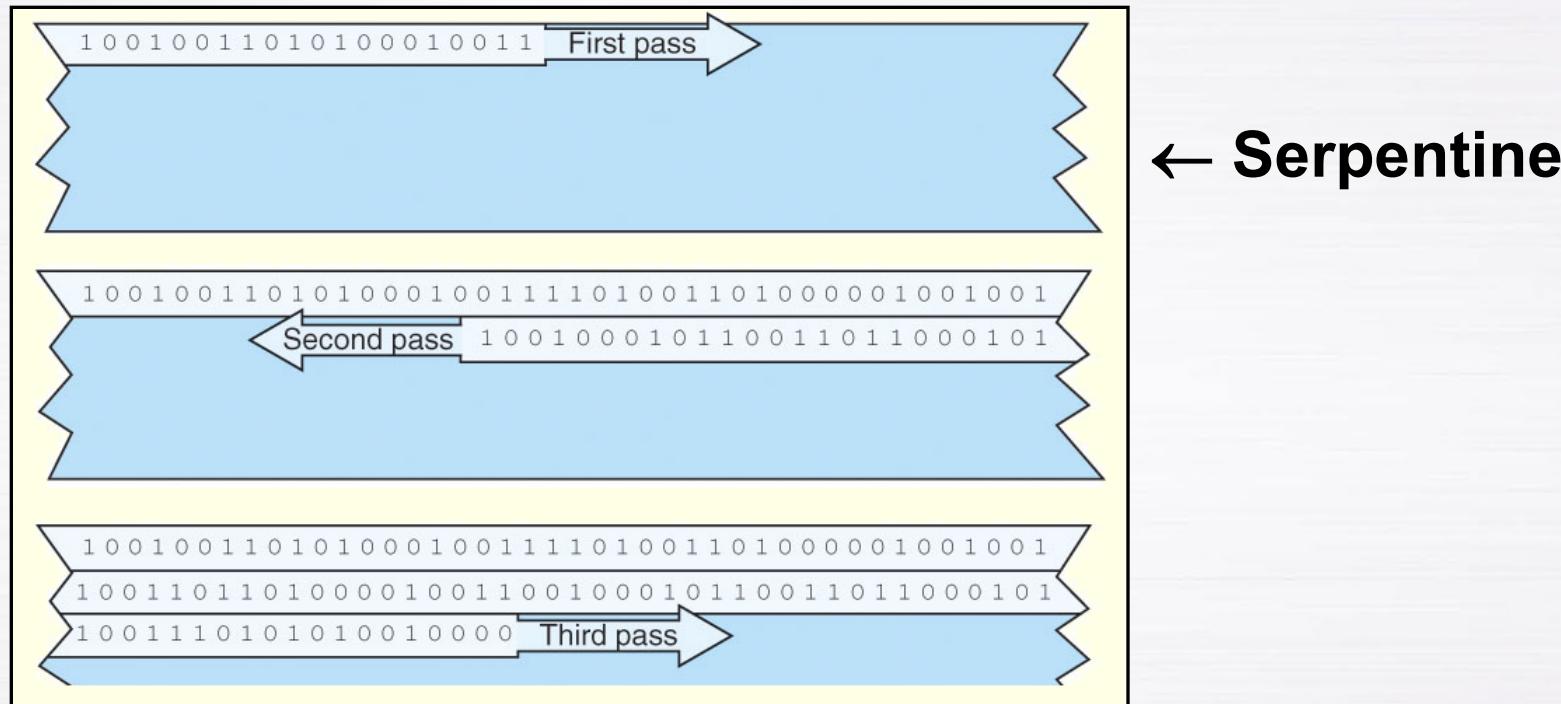


7.8 Magnetic Tape

- Today's tapes are digital, and provide multiple gigabytes of data storage.
- Two dominant recording methods are *serpentine* and *helical scan*, which are distinguished by how the read-write head passes over the recording medium.
- Serpentine recording is used in *digital linear tape* (DLT) and *Quarter inch cartridge* (QIC) tape systems.
- *Digital audio tape* (DAT) systems employ helical scan recording.

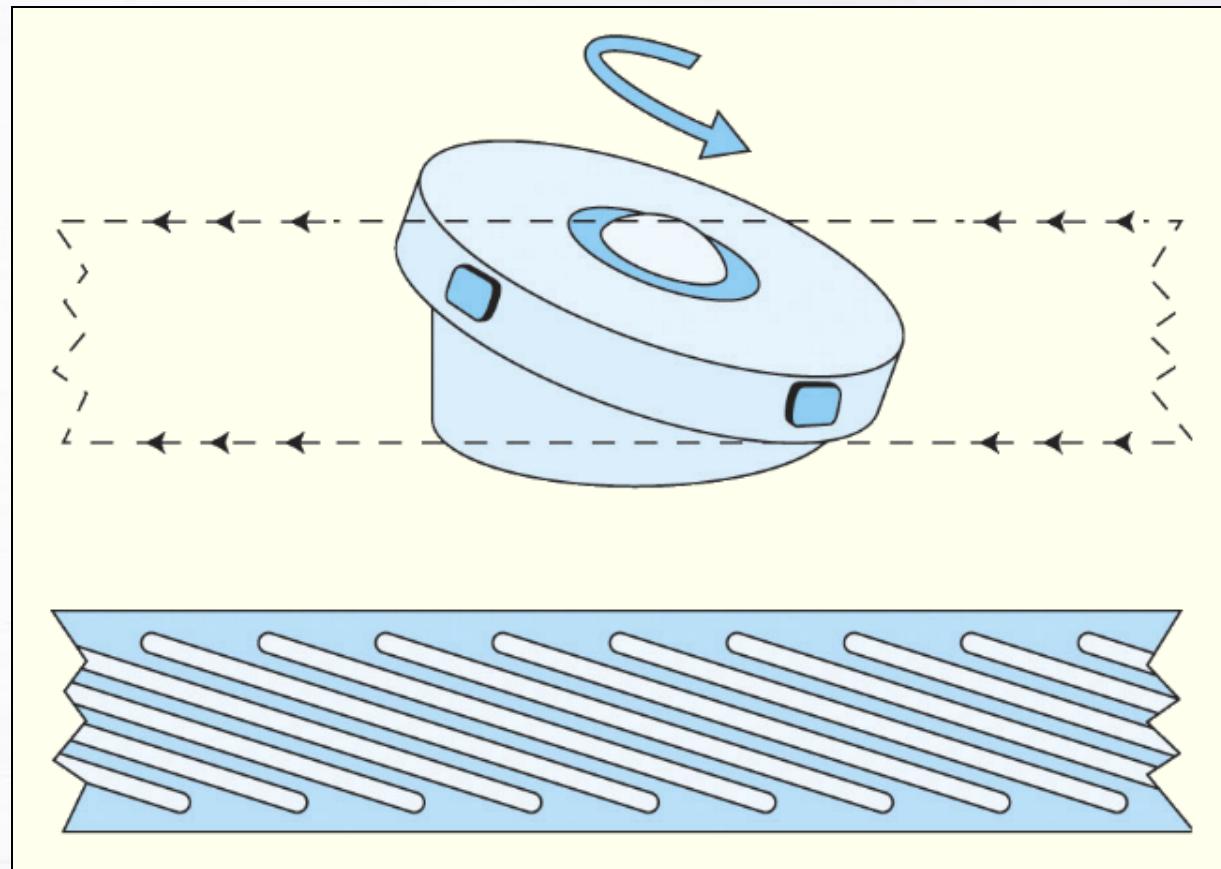
These two recording methods are shown on the next slide.

7.8 Magnetic Tape



7.8 Magnetic Tape

Helical Scan →



7.8 Magnetic Tape

- Numerous incompatible tape formats emerged over the years.
 - Sometimes even different models of the same manufacturer's tape drives were incompatible!
- Finally, in 1997, HP, IBM, and Seagate collaboratively invented a best-of-breed tape standard.
- They called this new tape format *Linear Tape Open* (*LTO*) because the specification is openly available.

7.8 Magnetic Tape

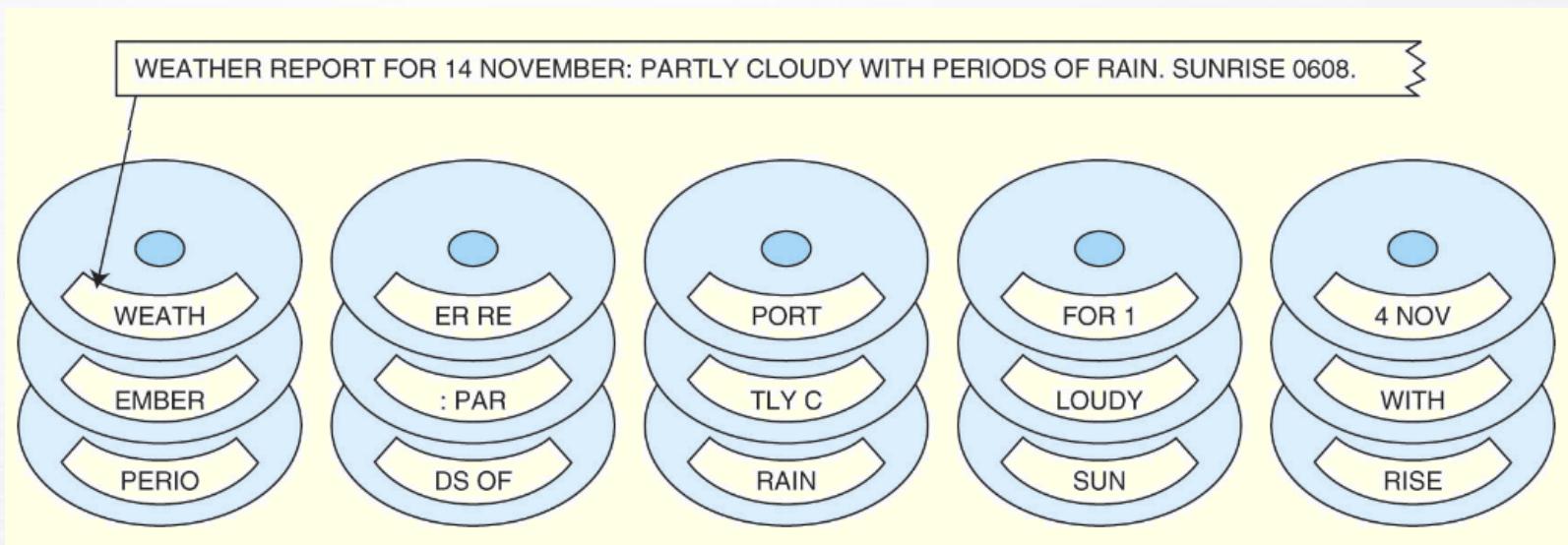
- LTO, as the name implies, is a linear digital tape format.
- The specification allowed for the refinement of the technology through four “generations.”
- Generation 5 was released in 2010.
 - Without compression, the tapes support a transfer rate of 208MB per second and each tape can hold up to 1.4TB.
- LTO supports several levels of error correction, providing superb reliability.
 - Tape has a reputation for being an error-prone medium.

7.9 RAID

- RAID, an acronym for *Redundant Array of Independent Disks* was invented to address problems of disk reliability, cost, and performance.
- In RAID, data is stored across many disks, with extra disks added to the array to provide error correction (redundancy).
- The inventors of RAID, David Patterson, Garth Gibson, and Randy Katz, provided a RAID taxonomy that has persisted for a quarter of a century, despite many efforts to redefine it.

7.9 RAID

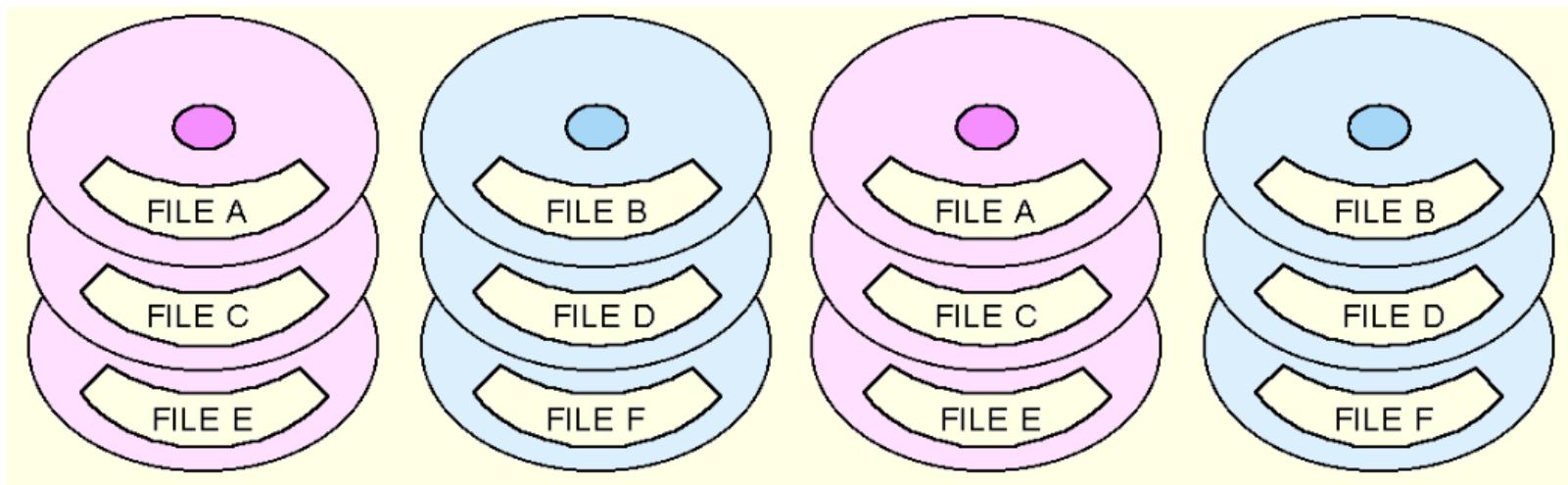
- RAID Level 0, also known as *drive spanning*, provides improved performance, but no redundancy.
 - Data is written in blocks across the entire array



- The disadvantage of RAID 0 is in its low reliability.

7.9 RAID

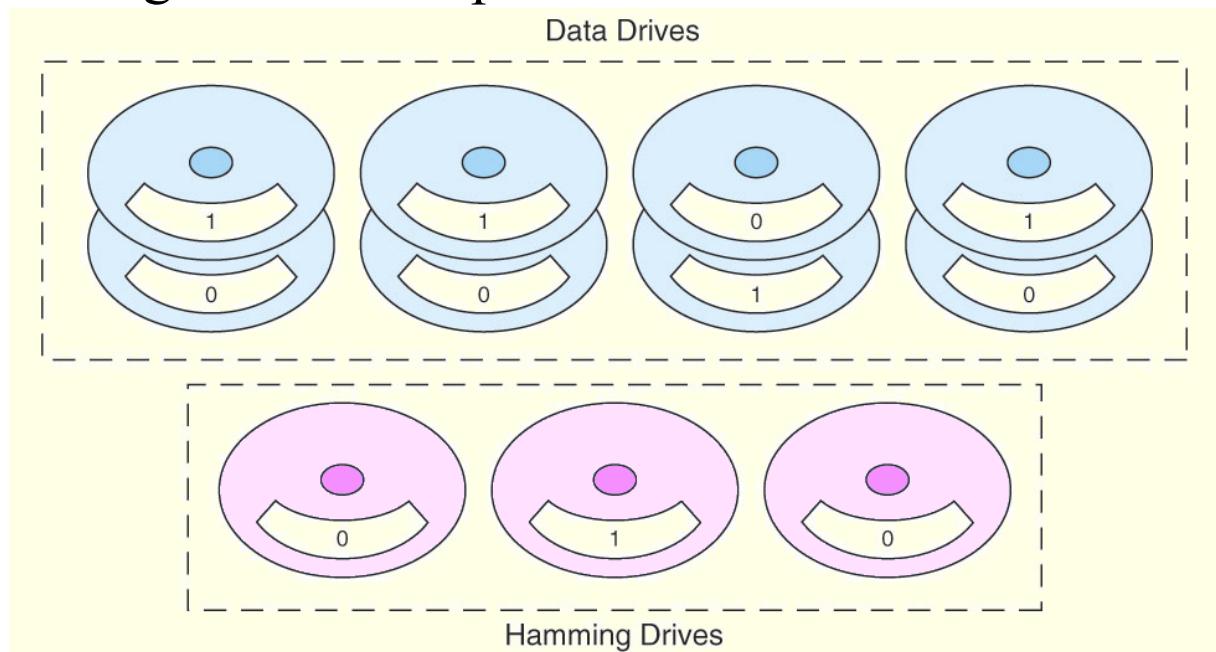
- RAID Level 1, also known as *disk mirroring*, provides 100% redundancy, and good performance.
 - Two matched sets of disks contain the same data.



- The disadvantage of RAID 1 is cost.

7.9 RAID

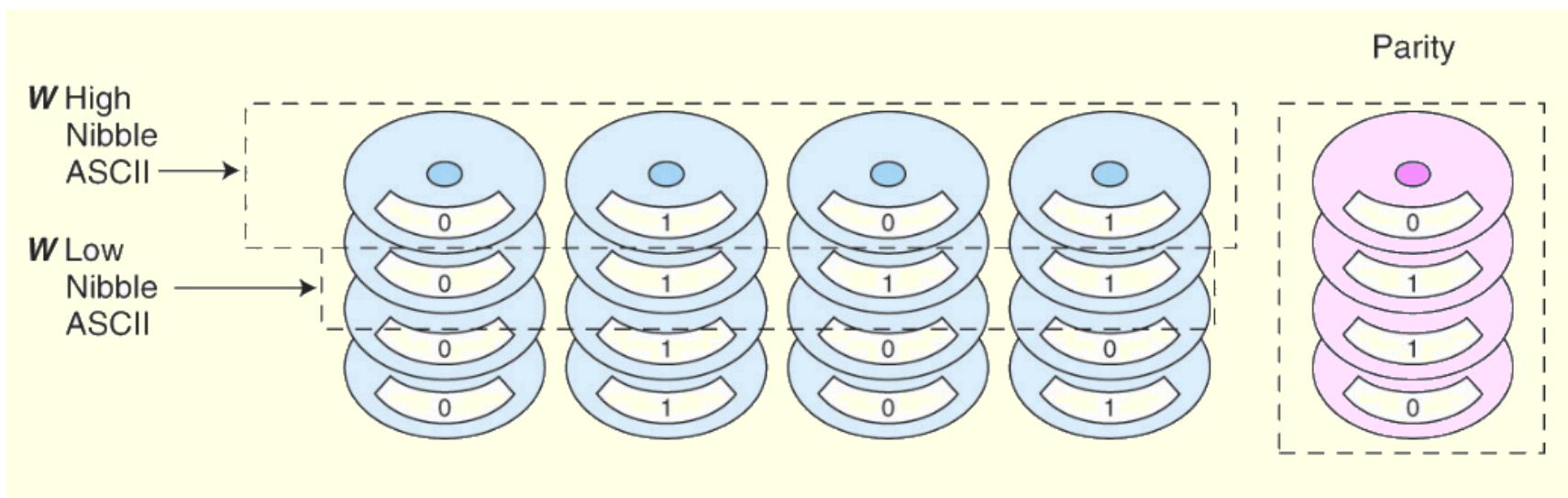
- A RAID Level 2 configuration consists of a set of data drives, and a set of Hamming code drives.
 - Hamming code drives provide error correction for the data drives.



- RAID 2 performance is poor and the cost is relatively high.

7.9 RAID

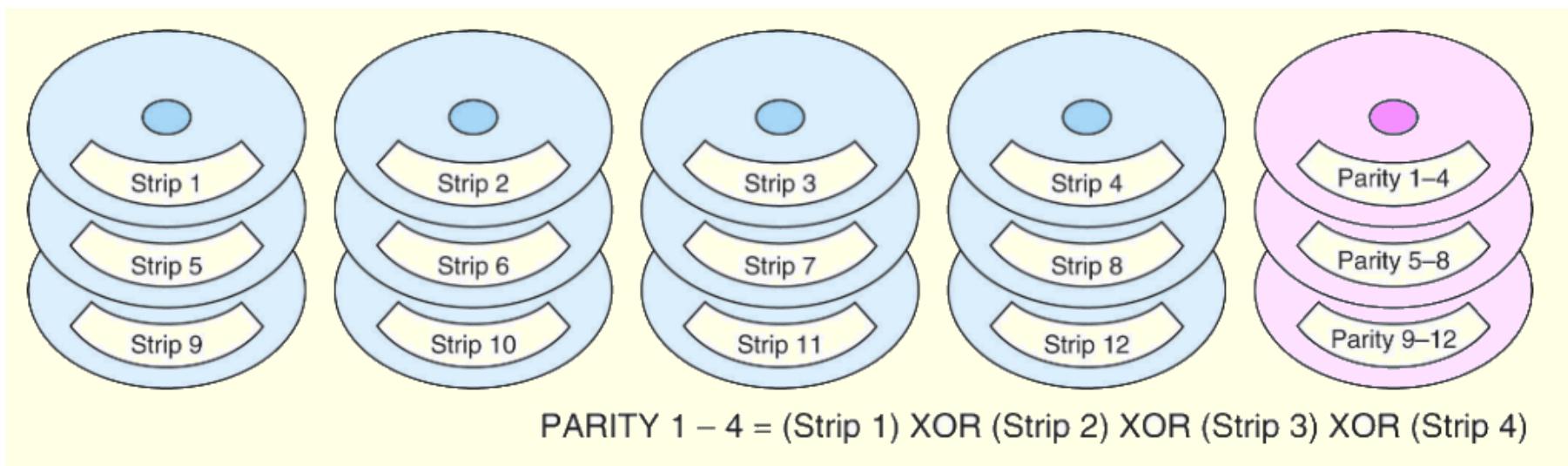
- RAID Level 3 stripes bits across a set of data drives and provides a separate disk for parity.
 - Parity is the XOR of the data bits.



- RAID 3 is not suitable for commercial applications, but is good for personal systems.

7.9 RAID

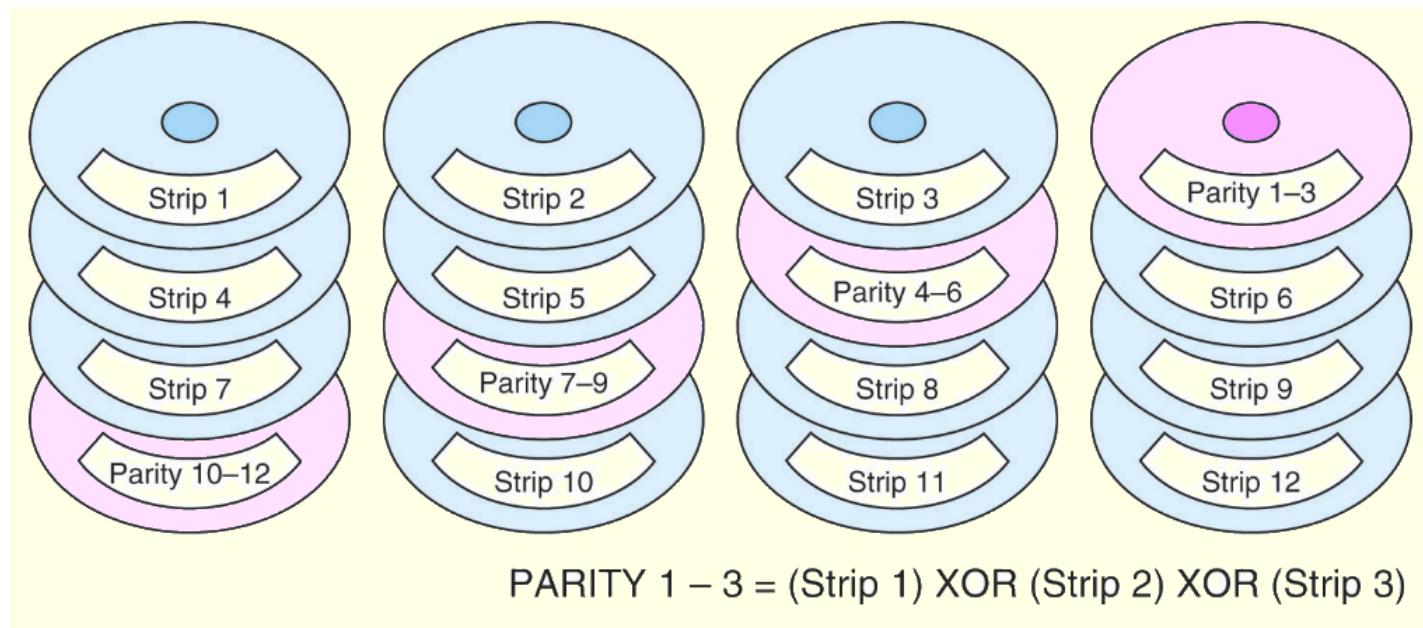
- RAID Level 4 is like adding parity disks to RAID 0.
 - Data is written in blocks across the data disks, and a parity block is written to the redundant drive.



- RAID 4 would be feasible if all record blocks were the same size.

7.9 RAID

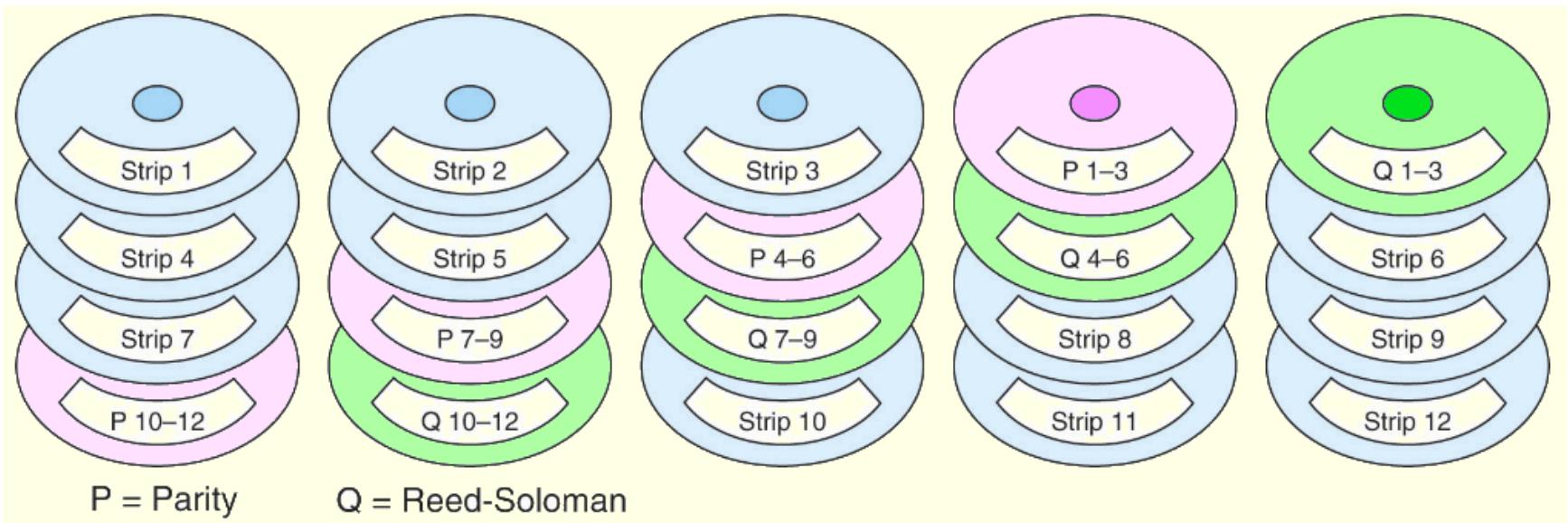
- RAID Level 5 is RAID 4 with distributed parity.
 - With distributed parity, some accesses can be serviced concurrently, giving good performance and high reliability.



- RAID 5 is used in many commercial systems.

7.9 RAID

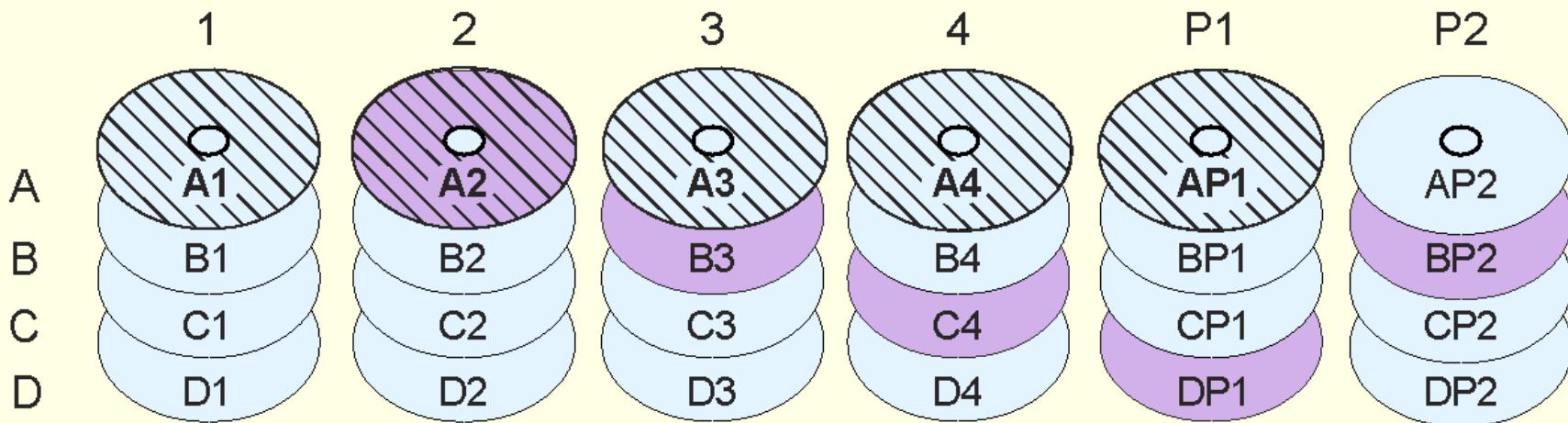
- RAID Level 6 carries two levels of error protection over striped data: Reed-Solomon and parity.
 - It can tolerate the loss of two disks.



- RAID 6 is write-intensive, but highly fault-tolerant.

7.9 RAID

- Double parity RAID (RAID DP) employs pairs of overlapping parity blocks that provide linearly independent parity functions.



$$AP1 = A1 \oplus A2 \oplus A3 \oplus A4$$

$$BP1 = B1 \oplus B2 \oplus B3 \oplus B4$$

$$CP1 = C1 \oplus C2 \oplus C3 \oplus C4$$

$$DP1 = D1 \oplus D2 \oplus D3 \oplus D4$$

$$AP2 = A1 \oplus B2 \oplus C3 \oplus D4$$

$$BP2 = A2 \oplus B3 \oplus C4 \oplus DP1$$

$$CP2 = A3 \oplus B4 \oplus CP1 \oplus D1$$

$$DP2 = A4 \oplus BP1 \oplus C1 \oplus D2$$

7.9 RAID

- Like RAID 6, RAID DP can tolerate the loss of two disks.
- The use of simple parity functions provides RAID DP with better performance than RAID 6.
- Of course, because two parity functions are involved, RAID DP's performance is somewhat degraded from that of RAID 5.
 - RAID DP is also known as EVENODD, diagonal parity RAID, RAID 5DP, advanced data guarding RAID (RAID ADG) and-- erroneously-- RAID 6.

7.9 RAID

- Large systems consisting of many drive arrays may employ various RAID levels, depending on the criticality of the data on the drives.
 - A disk array that provides program workspace (say for file sorting) does not require high fault tolerance.
- Critical, high-throughput files can benefit from combining RAID 0 with RAID 1, called RAID 10.
- RAID 50 combines striping and distributed parity. For good fault tolerance and high capacity.
 - Note: Higher RAID levels do not necessarily mean “better” RAID levels. It all depends upon the needs of the applications that use the disks.

7.10 The Future of Data Storage

- Advances in technology have defied all efforts to define the ultimate upper limit for magnetic disk storage.
 - In the 1970s, the upper limit was thought to be around 2Mb/in^2 .
 - Today's disks commonly support 20Gb/in^2 .
- Improvements have occurred in several different technologies including:
 - Materials science
 - Magneto-optical recording heads.
 - Error correcting codes.

7.10 The Future of Data Storage

- As data densities increase, bit cells consist of proportionately fewer magnetic grains.
- There is a point at which there are too few grains to hold a value, and a 1 might spontaneously change to a 0, or vice versa.
- This point is called the superparamagnetic limit.
 - In 2006, the superparamagnetic limit is thought to lie between 150Gb/in^2 and 200Gb/in^2 .
- Even if this limit is wrong by a few orders of magnitude, the greatest gains in magnetic storage have probably already been realized.

7.10 The Future of Data Storage

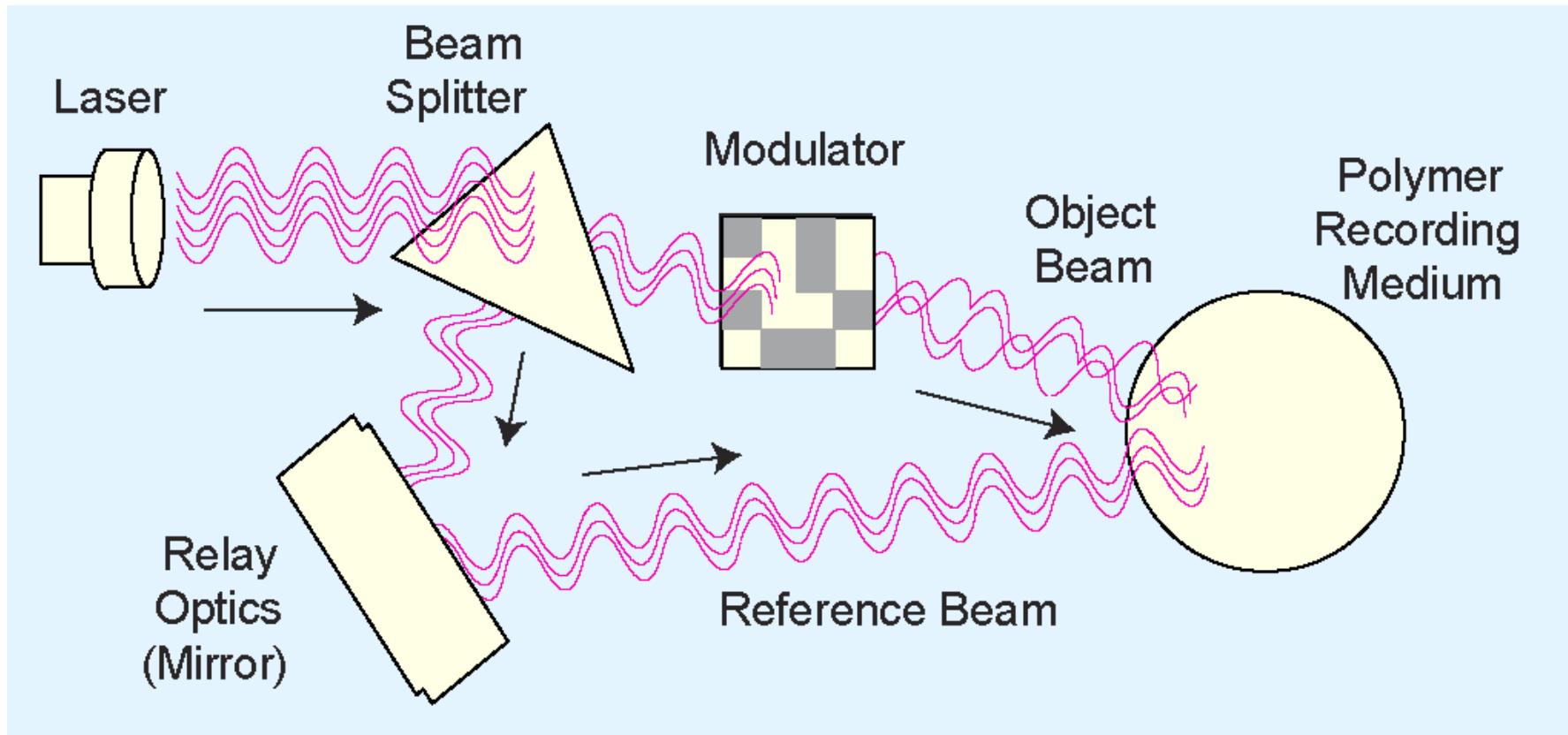
- Future exponential gains in data storage most likely will occur through the use of totally new technologies.
- Research into finding suitable replacements for magnetic disks is taking place on several fronts.
- Some of the more interesting technologies include:
 - Biological materials
 - Holographic systems
 - Micro-electro-mechanical devices
 - Carbon nanotubes
 - Memristors

7.10 The Future of Data Storage

- Present day biological data storage systems combine organic compounds such as proteins or oils with inorganic (magnetizable) substances.
- Early prototypes have encouraged the expectation that densities of 1 Tb/in^2 are attainable.
- Of course, the ultimate biological data storage medium is DNA.
 - Trillions of messages can be stored in a tiny strand of DNA.
- Practical DNA-based data storage is most likely decades away.

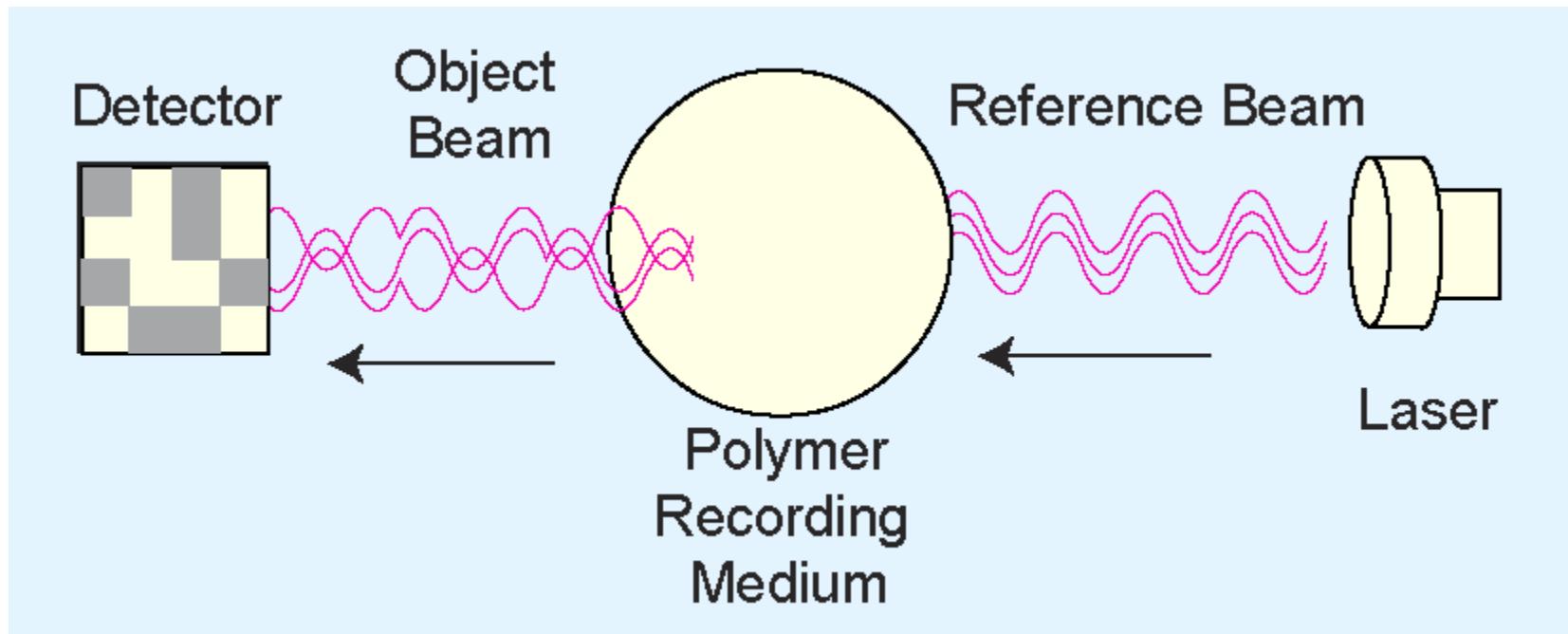
7.10 The Future of Data Storage

- Holographic storage uses a pair of laser beams to etch a three-dimensional hologram onto a polymer medium.



7.10 The Future of Data Storage

- Data is retrieved by passing the reference beam through the hologram, thereby reproducing the original coded object beam.



7.10 The Future of Data Storage

- Because holograms are three-dimensional, tremendous data densities are possible.
- Experimental systems have achieved over 30Gb/in^2 , with transfer rates of around 1GBps.
- In addition, holographic storage is content addressable.
 - This means that there is no need for a file directory on the disk. Accordingly, access time is reduced.
- The major challenge is in finding an inexpensive, stable, rewriteable holographic medium.

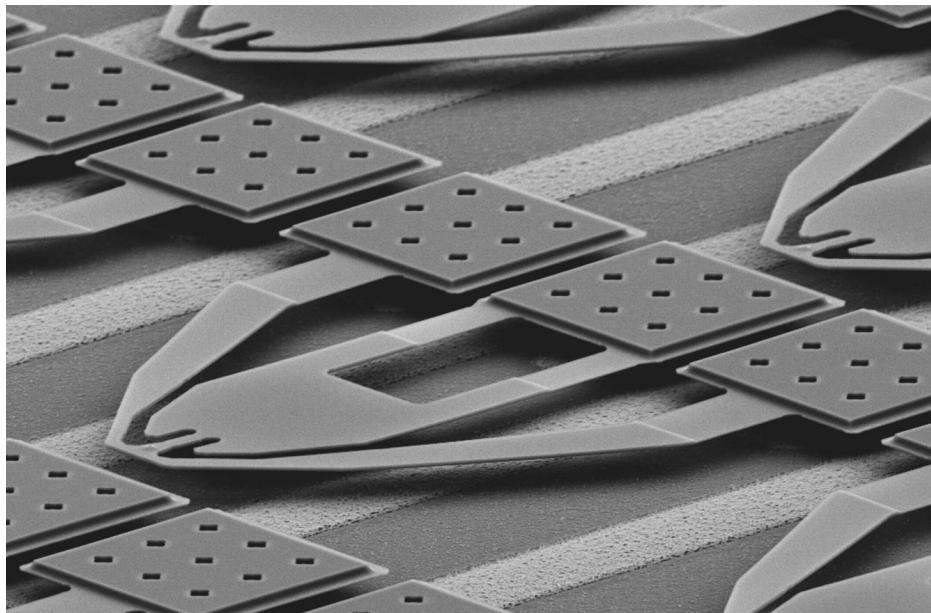
7.10 The Future of Data Storage

- Micro-electro-mechanical storage (MEMS) devices offer another promising approach to mass storage.
- IBM's Millipede is one such device.
- Prototypes have achieved densities of 100Gb/in^2 with 1Tb/in^2 expected as the technology is refined.

A photomicrograph of Millipede is shown on the next slide.

7.10 The Future of Data Storage

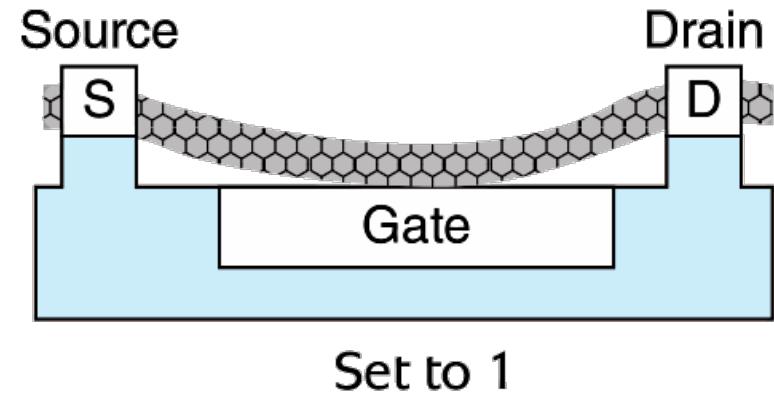
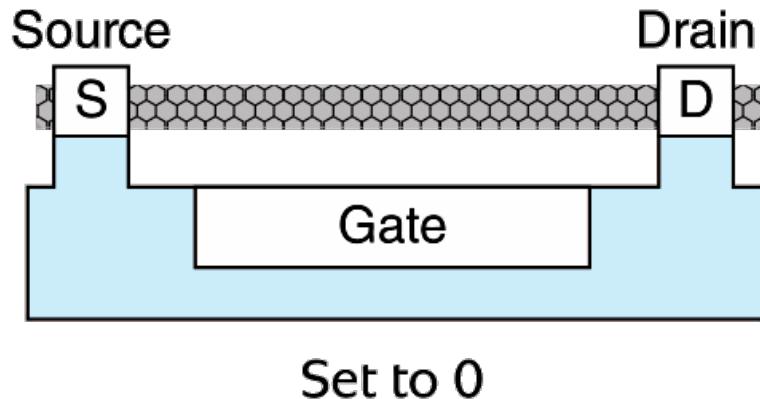
- Millipede consists of thousands of cantilevers that record a binary 1 by pressing a heated tip into a polymer substrate.
- The tip reads a binary 1 when it dips into the imprint in the polymer



**Photomicrograph courtesy
of the IBM Corporation.
© 2005 IBM Corporation**

7.10 The Future of Data Storage

- CNTs are a cylindrical form of elemental carbon: The walls of the cylinders are one atom thick.
- CNTs can act like switches, opening and closing to store bits.
- Once “set” the CNT stays in place until a release voltage is applied.



7.10 The Future of Data Storage

- Memristors are electronic components that combine the properties of a resistor with memory.
- Resistance to current flow can be controlled so that states of “high” and “low” store data bits.
- Like CNTs, memristor memories are non-volatile, holding their state until certain threshold voltages are applied.
- These non-volatile memories promise enormous energy savings and increased data access speeds in the very near future.

Chapter 7 Conclusion

- I/O systems are critical to the overall performance of a computer system.
- Amdahl's Law quantifies this assertion.
- I/O systems consist of memory blocks, cabling, control circuitry, interfaces, and media.
- I/O control methods include programmed I/O, interrupt-based I/O, DMA, and channel I/O.
- Buses require control lines, a clock, and data lines. Timing diagrams specify operational details.

Chapter 7 Conclusion

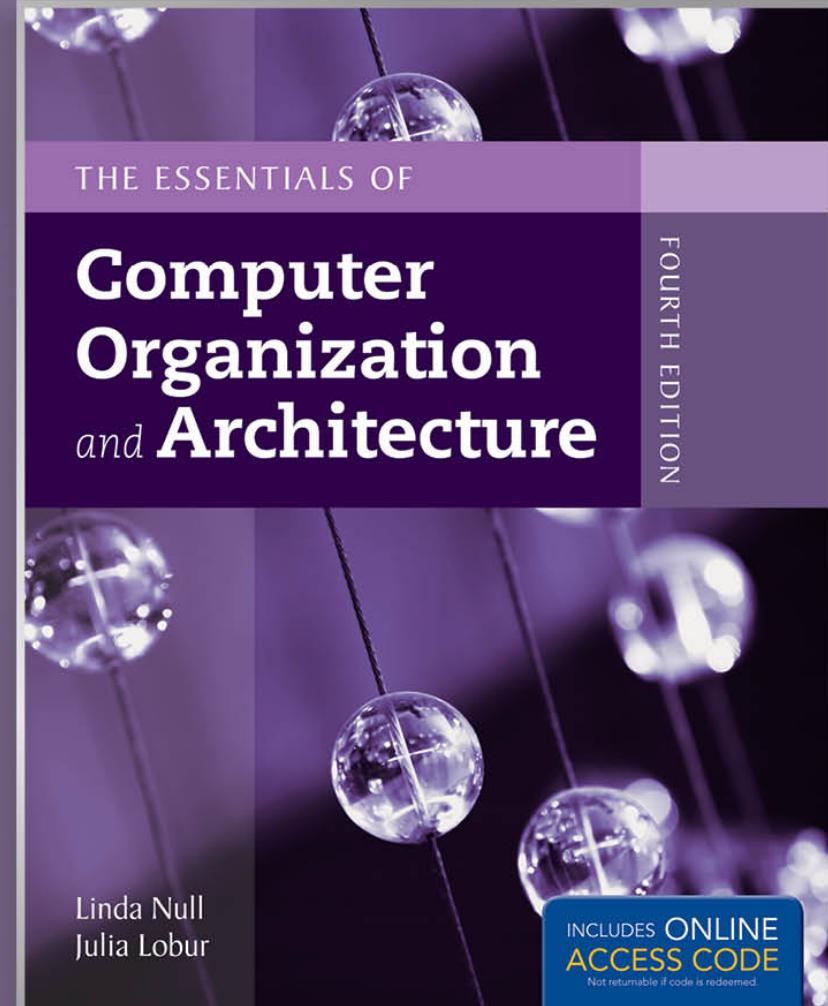
- Magnetic disk is the principal form of durable storage.
- Disk performance metrics include seek time, rotational delay, and reliability estimates.
- Enterprise SSDs save energy and provide improved data access for government and industry.
- Optical disks provide long-term storage for large amounts of data, although access is slow.
- Magnetic tape is also an archival medium still widely used.

Chapter 7 Conclusion

- RAID gives disk systems improved performance and reliability. RAID 3 and RAID 5 are the most common.
- RAID 6 and RAID DP protect against dual disk failure, but RAID DP offers better performance.
- Any one of several new technologies including biological, holographic, CNT, memristor, or mechanical may someday replace magnetic disks.
- The hardest part of data storage may be in locating the data after it's stored.

Chapter 9

Alternative Architectures



Chapter 9 Objectives

- Learn the properties that often distinguish RISC from CISC architectures.
- Understand how multiprocessor architectures are classified.
- Appreciate the factors that create complexity in multiprocessor systems.
- Become familiar with the ways in which some architectures transcend the traditional von Neumann paradigm.

9.1 Introduction

- We have so far studied only the simplest models of computer systems; classical single-processor von Neumann systems.
- This chapter presents a number of different approaches to computer organization and architecture.
- Some of these approaches are in place in today's commercial systems. Others may form the basis for the computers of tomorrow.

9.2 RISC Machines

- The underlying philosophy of RISC machines is that a system is better able to manage program execution when the program consists of only a few different instructions that are the same length and require the same number of clock cycles to decode and execute.
- RISC systems access memory only with explicit load and store instructions.
- In CISC systems, many different kinds of instructions access memory, making instruction length variable and fetch-decode-execute time unpredictable.

9.2 RISC Machines

- The difference between CISC and RISC becomes evident through the basic computer performance equation:

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

- RISC systems shorten execution time by reducing the clock cycles per instruction.
- CISC systems improve performance by reducing the number of instructions per program.

9.2 RISC Machines

- The simple instruction set of RISC machines enables control units to be hardwired for maximum speed.
- The more complex-- and variable-- instruction set of CISC machines requires microcode-based control units that interpret instructions as they are fetched from memory. This translation takes time.
- With fixed-length instructions, RISC lends itself to pipelining and speculative execution.

9.2 RISC Machines

- Consider the program fragments:

CISC

```
mov ax, 10  
mov bx, 5  
mul bx, ax
```

RISC

```
Begin    add ax, bx  
loop Begin
```

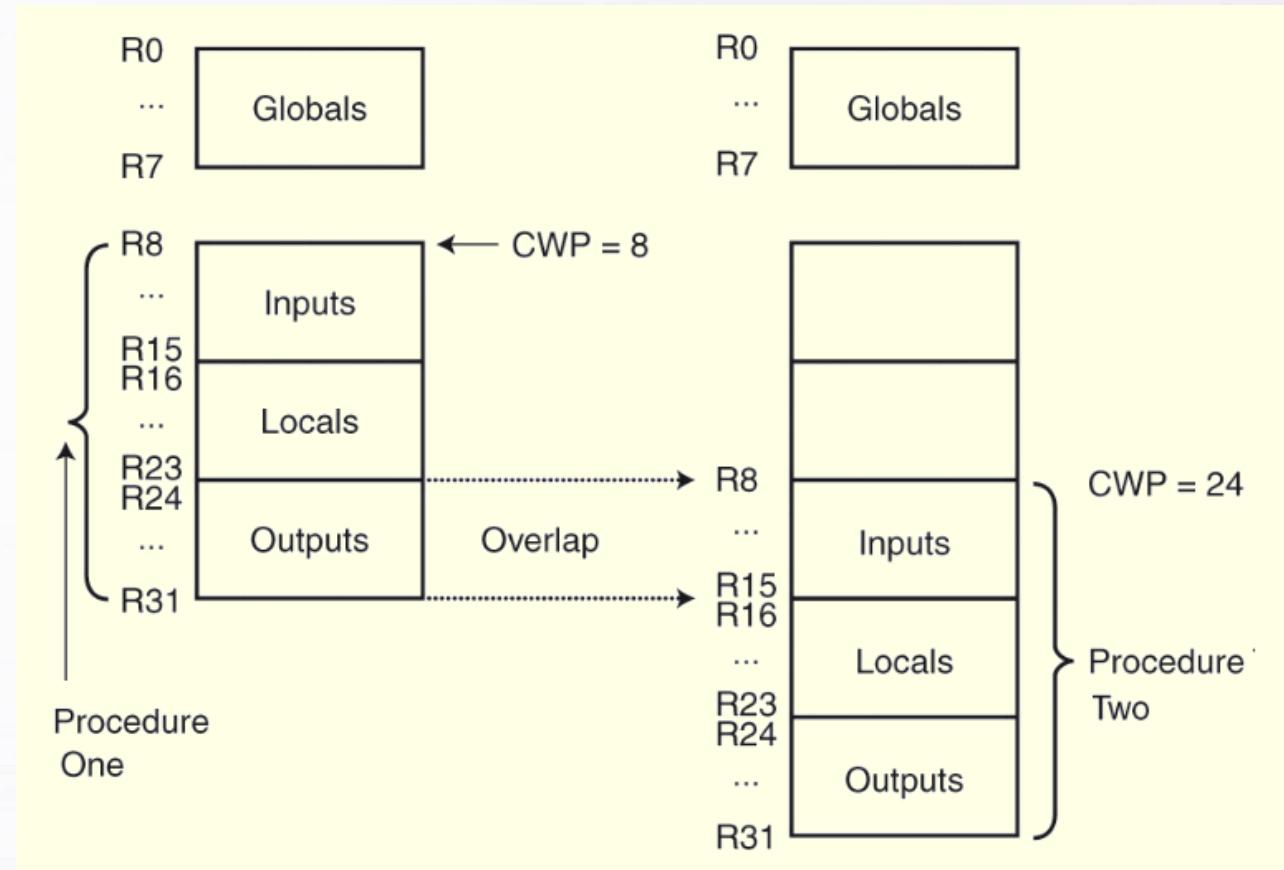
- The total clock cycles for the CISC version might be:
 $(2 \text{ movs} \times 1 \text{ cycle}) + (1 \text{ mul} \times 30 \text{ cycles}) = 32 \text{ cycles}$
- While the clock cycles for the RISC version is:
 $(3 \text{ movs} \times 1 \text{ cycle}) + (5 \text{ adds} \times 1 \text{ cycle}) + (5 \text{ loops} \times 1 \text{ cycle}) = 13 \text{ cycles}$
- With RISC clock cycle being shorter, RISC gives us much faster execution speeds.

9.2 RISC Machines

- Because of their load-store ISAs, RISC architectures require a large number of CPU registers.
- These register provide fast access to data during sequential program execution.
- They can also be employed to reduce the overhead typically caused by passing parameters to subprograms.
- Instead of pulling parameters off of a stack, the subprogram is directed to use a subset of registers.

9.2 RISC Machines

- This is how registers can be overlapped in a RISC system.
- The *current window pointer* (CWP) points to the active register window.



9.2 RISC Machines

- It is becoming increasingly difficult to distinguish RISC architectures from CISC architectures.
- Some RISC systems provide more extravagant instruction sets than some CISC systems.
- Some systems combine both approaches.
- The following two slides summarize the characteristics that traditionally typify the differences between these two architectures.

9.2 RISC Machines

- **RISC**

- Multiple register sets.
- Three operands per instruction.
- Parameter passing through register windows.
- Single-cycle instructions.
- Hardwired control.
- Highly pipelined.

- **CISC**

- Single register set.
- One or two register operands per instruction.
- Parameter passing through memory.
- Multiple cycle instructions.
- Microprogrammed control.
- Less pipelined.

Continued....

9.2 RISC Machines

- **RISC**

- Simple instructions, few in number.
- Fixed length instructions.
- Complexity in compiler.
- Only **LOAD/STORE** instructions access memory.
- Few addressing modes.

- **CISC**

- Many complex instructions.
- Variable length instructions.
- Complexity in microcode.
- Many instructions can access memory.
- Many addressing modes.

9.3 Flynn's Taxonomy

- Many attempts have been made to come up with a way to categorize computer architectures.
- *Flynn's Taxonomy* has been the most enduring of these, despite having some limitations.
- Flynn's Taxonomy takes into consideration the number of processors and the number of data paths incorporated into an architecture.
- A machine can have one or many processors that operate on one or many data streams.

9.3 Flynn's Taxonomy

- The four combinations of multiple processors and multiple data paths are described by Flynn as:
 - **SISD:** Single instruction stream, single data stream. These are classic uniprocessor systems.
 - **SIMD:** Single instruction stream, multiple data streams. Execute the same instruction on multiple data values, as in vector processors.
 - **MIMD:** Multiple instruction streams, multiple data streams. These are today's parallel architectures.
 - **MISD:** Multiple instruction streams, single data stream.

9.3 Flynn's Taxonomy

- Flynn's Taxonomy falls short in a number of ways:
- First, there appears to be no need for MISD machines.
- Second, parallelism is not homogeneous. This assumption ignores the contribution of specialized processors.
- Third, it provides no straightforward way to distinguish architectures of the MIMD category.
 - One idea is to divide these systems into those that share memory, and those that don't, as well as whether the interconnections are bus-based or switch-based.

9.3 Flynn's Taxonomy

- Symmetric multiprocessors (SMP) and massively parallel processors (MPP) are MIMD architectures that differ in how they use memory.
- SMP systems share the same memory and MPP do not.
- An easy way to distinguish SMP from MPP is:
 - MPP \Rightarrow many processors + distributed memory + communication via network
 - SMP \Rightarrow fewer processors + shared memory + communication via memory

9.3 Flynn's Taxonomy

- Other examples of MIMD architectures are found in distributed computing, where processing takes place collaboratively among networked computers.
 - A **network of workstations** (NOW) uses otherwise idle systems to solve a problem.
 - A **collection of workstations** (COW) is a NOW where one workstation coordinates the actions of the others.
 - A **dedicated cluster parallel computer** (DCPC) is a group of workstations brought together to solve a specific problem.
 - A **pile of PCs** (POPC) is a cluster of (usually) heterogeneous systems that form a dedicated parallel system.

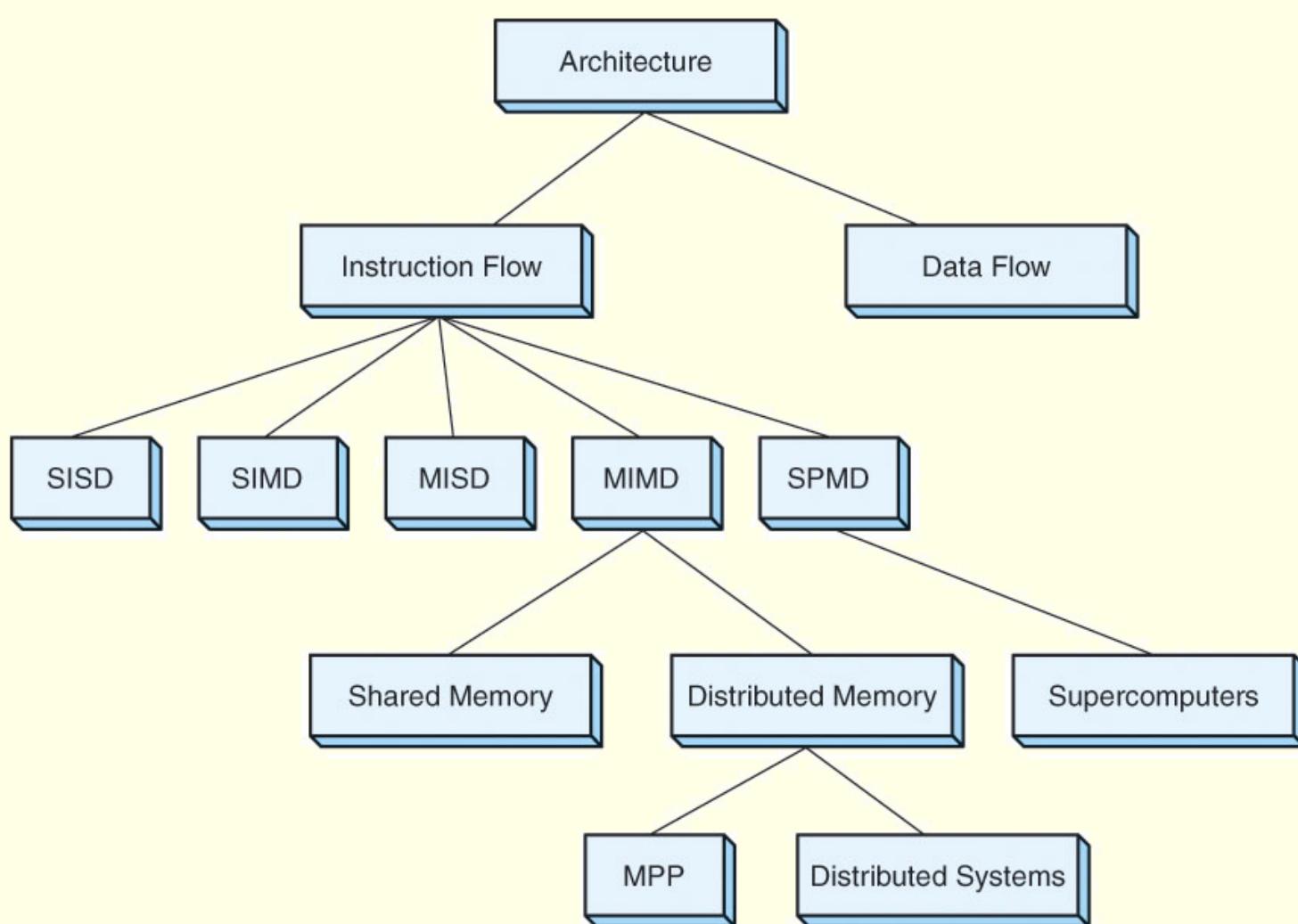
9.3 Flynn's Taxonomy

- Flynn's Taxonomy has been expanded to include SPMD (single program, multiple data) architectures.
- Each SPMD processor has its own data set and program memory. Different nodes can execute different instructions within the same program using instructions similar to:

If myNodeNum = 1 do this, else do that
- Yet another idea missing from Flynn's is whether the architecture is instruction driven or data driven.

The next slide provides a revised taxonomy.

9.3 Flynn's Taxonomy



9.4 Parallel and Multiprocessor Architectures

- Parallel processing is capable of economically increasing system throughput while providing better fault tolerance.
- The limiting factor is that no matter how well an algorithm is parallelized, there is always some portion that must be done sequentially.
 - Additional processors sit idle while the sequential work is performed.
- Thus, it is important to keep in mind that an n -fold increase in processing power does not necessarily result in an n -fold increase in throughput.

9.4 Parallel and Multiprocessor Architectures

- Recall that pipelining divides the fetch-decode-execute cycle into stages that each carry out a small part of the process on a set of instructions.
- Ideally, an instruction exits the pipeline during each tick of the clock.
- *Superpipelining* occurs when a pipeline has stages that require less than half a clock cycle to complete.
 - The pipeline is equipped with a separate clock running at a frequency that is at least double that of the main system clock.
- Superpipelining is only one aspect of superscalar design.

9.4 Parallel and Multiprocessor Architectures

- Superscalar architectures include multiple execution units such as specialized integer and floating-point adders and multipliers.
- A critical component of this architecture is the *instruction fetch unit*, which can simultaneously retrieve several instructions from memory.
- A *decoding unit* determines which of these instructions can be executed in parallel and combines them accordingly.
- This architecture also requires compilers that make optimum use of the hardware.

9.4 Parallel and Multiprocessor Architectures

- Very long instruction word (VLIW) architectures differ from superscalar architectures because the VLIW compiler, instead of a hardware decoding unit, packs independent instructions into one long instruction that is sent down the pipeline to the execution units.
- One could argue that this is the best approach because the compiler can better identify instruction dependencies.
- However, compilers tend to be conservative and cannot have a view of the run time code.

9.4 Parallel and Multiprocessor Architectures

- Vector computers are processors that operate on entire vectors or matrices at once.
 - These systems are often called supercomputers.
- Vector computers are highly pipelined so that arithmetic instructions can be overlapped.
- Vector processors can be categorized according to how operands are accessed.
 - **Register-register** vector processors require all operands to be in registers.
 - **Memory-memory** vector processors allow operands to be sent from memory directly to the arithmetic units.

9.4 Parallel and Multiprocessor Architectures

- A disadvantage of register-register vector computers is that large vectors must be broken into fixed-length segments so they will fit into the register sets.
- Memory-memory vector computers have a longer startup time until the pipeline becomes full.
- In general, vector machines are efficient because there are fewer instructions to fetch, and corresponding pairs of values can be prefetched because the processor knows it will have a continuous stream of data.

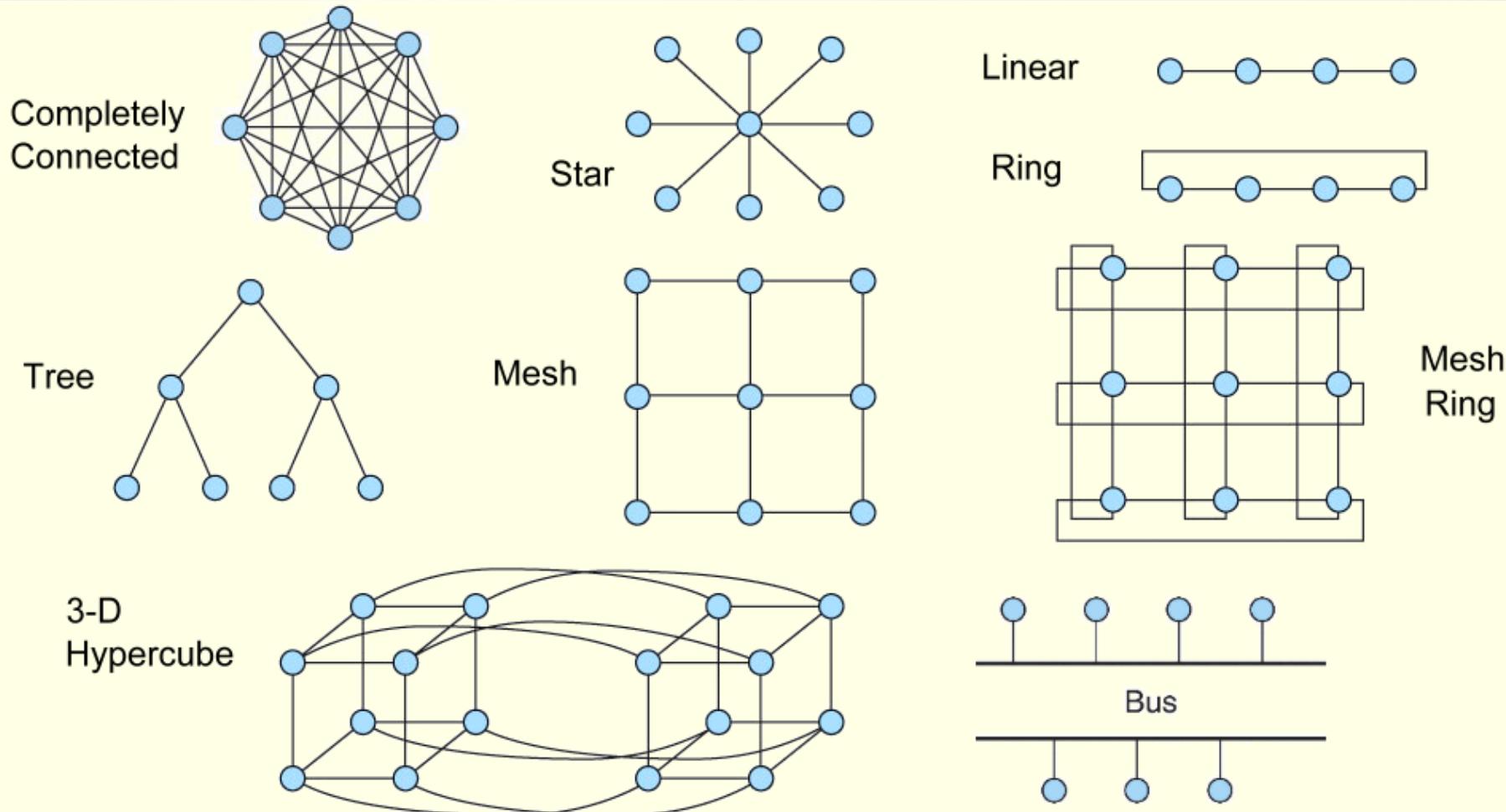
9.4 Parallel and Multiprocessor Architectures

- MIMD systems can communicate through shared memory or through an interconnection network.
- Interconnection networks are often classified according to their topology, routing strategy, and switching technique.
- Of these, the topology is a major determining factor in the overhead cost of message passing.
- Message passing takes time owing to network latency and incurs overhead in the processors.

9.4 Parallel and Multiprocessor Architectures

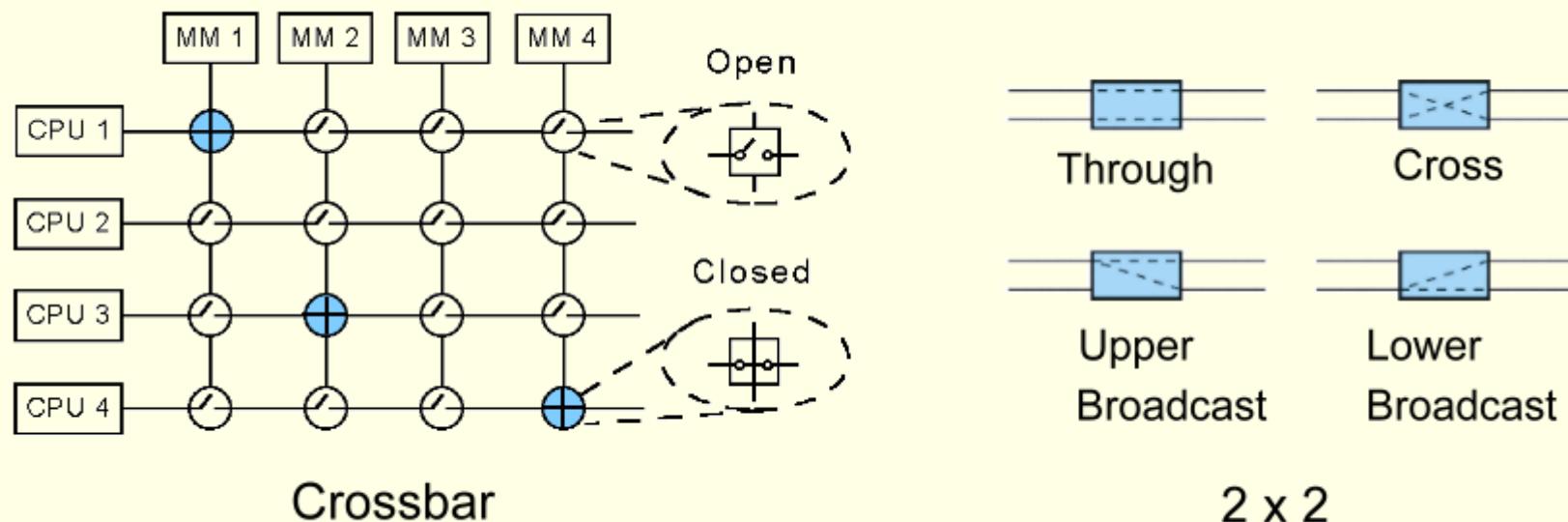
- Interconnection networks can be either static or dynamic.
- Processor-to-memory connections usually employ dynamic interconnections. These can be blocking or nonblocking.
 - Nonblocking interconnections allow connections to occur simultaneously.
- Processor-to-processor message-passing interconnections are usually static, and can employ any of several different topologies, as shown on the following slide.

9.4 Parallel and Multiprocessor Architectures



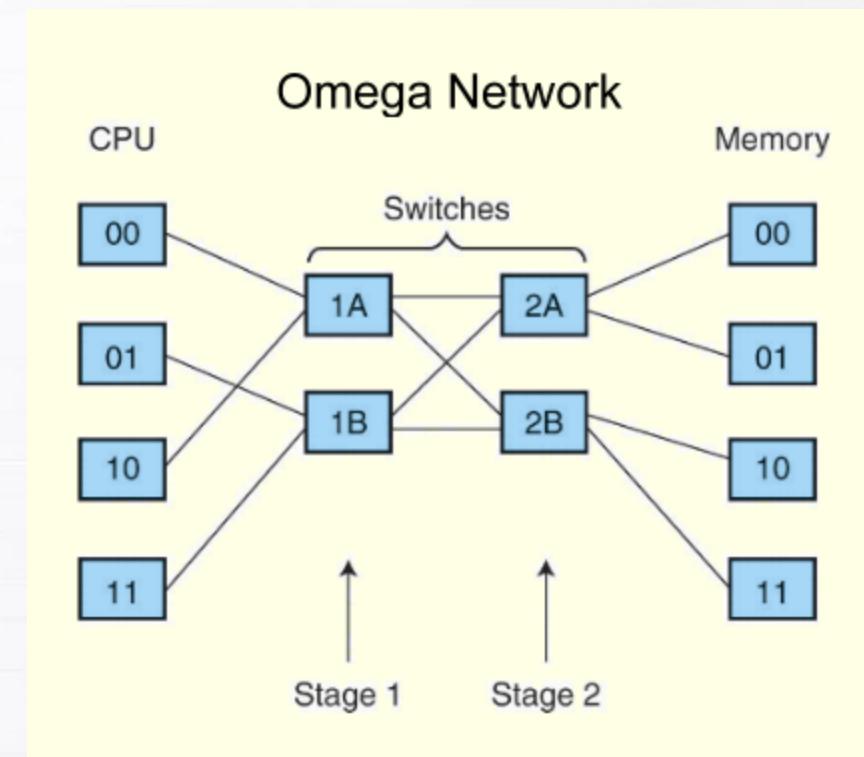
9.4 Parallel and Multiprocessor Architectures

- Dynamic routing is achieved through switching networks that consist of crossbar switches or 2×2 switches.



9.4 Parallel and Multiprocessor Architectures

- Multistage interconnection (or shuffle) networks are the most advanced class of switching networks.
- They can be used in loosely-coupled distributed systems, or in tightly-coupled processor-to-memory configurations.



9.4 Parallel and Multiprocessor Architectures

- There are advantages and disadvantages to each switching approach.
 - Bus-based networks, while economical, can be bottlenecks. Parallel buses can alleviate bottlenecks, but are costly.
 - Crossbar networks are nonblocking, but require n^2 switches to connect n entities.
 - Omega networks are blocking networks, but exhibit less contention than bus-based networks. They are somewhat more economical than crossbar networks, n nodes needing $\log_2 n$ stages with $n / 2$ switches per stage.

9.4 Parallel and Multiprocessor Architectures

- Tightly-coupled multiprocessor systems use the same memory. They are also referred to as shared memory multiprocessors.
- The processors do not necessarily have to share the same block of physical memory:
- Each processor can have its own memory, but it must share it with the other processors.
- Configurations such as these are called *distributed shared memory multiprocessors*.

9.4 Parallel and Multiprocessor Architectures

- Shared memory MIMD machines can be divided into two categories based upon how they access memory.
- In *uniform memory access* (UMA) systems, all memory accesses take the same amount of time.
- To realize the advantages of a multiprocessor system, the interconnection network must be fast enough to support multiple concurrent accesses to memory, or it will slow down the whole system.
- Thus, the interconnection network limits the number of processors in a UMA system.

9.4 Parallel and Multiprocessor Architectures

- The other category of MIMD machines are the *nonuniform memory access* (NUMA) systems.
- While NUMA machines see memory as one contiguous addressable space, each processor gets its own piece of it.
- Thus, a processor can access its own memory much more quickly than it can access memory that is elsewhere.
- Not only does each processor have its own memory, it also has its own cache, a configuration that can lead to *cache coherence* problems.

9.4 Parallel and Multiprocessor Architectures

- Cache coherence problems arise when main memory data is changed and the cached image is not. (We say that the cached value is *stale*.)
- To combat this problem, some NUMA machines are equipped with *snoopy cache controllers* that monitor all caches on the systems. These systems are called *cache coherent NUMA* (CC-NUMA) architectures.
- A simpler approach is to ask the processor having the stale value to either void the stale cached value or to update it with the new value.

9.4 Parallel and Multiprocessor Architectures

- When a processor's cached value is updated concurrently with the update to memory, we say that the system uses a *write-through* cache update protocol.
- If the *write-through with update* protocol is used, a message containing the update is broadcast to all processors so that they may update their caches.
- If the *write-through with invalidate* protocol is used, a broadcast asks all processors to invalidate the stale cached value.

9.4 Parallel and Multiprocessor Architectures

- Write-invalidate uses less bandwidth because it uses the network only the first time the data is updated, but retrieval of the fresh data takes longer.
- Write-update creates more message traffic, but all caches are kept current.
- Another approach is the *write-back* protocol that delays an update to memory until the modified cache block must be replaced.
- At replacement time, the processor writing the cached value must obtain exclusive rights to the data. When rights are granted, all other cached copies are invalidated.

9.4 Parallel and Multiprocessor Architectures

- Distributed computing is another form of multiprocessing. However, the term *distributed computing* means different things to different people.
- In a sense, all multiprocessor systems are distributed systems because the processing load is distributed among processors that work collaboratively.
- The common understanding is that a distributed system consists of very loosely-coupled processing units.
- Recently, NOWs have been used as distributed systems to solve large, intractable problems.

9.4 Parallel and Multiprocessor Architectures

- For general-use computing, the details of the network and the nature of the multiplatform computing should be transparent to the users of the system.
- Remote procedure calls (RPCs) enable this transparency. RPCs use resources on remote machines by invoking procedures that reside and are executed on the remote machines.
- RPCs are employed by numerous vendors of distributed computing architectures including the Common Object Request Broker Architecture (CORBA) and Java's Remote Method Invocation (RMI).

9.4 Parallel and Multiprocessor Architectures

- Cloud computing is distributed computing to the extreme.
- It provides *services* over the Internet through a collection of loosely-coupled systems.
- In theory, the service consumer has no awareness of the hardware, or even its location.
 - Your services and data may even be located on the same physical system as that of your business competitor.
 - The hardware might even be located in another country.
- Security concerns are a major inhibiting factor for cloud computing.

9.5 Alternative Parallel Processing Approaches

- Some people argue that real breakthroughs in computational power-- breakthroughs that will enable us to solve today's intractable problems-- will occur only by abandoning the von Neumann model.
- Numerous efforts are now underway to devise systems that could change the way that we think about computers and computation.
- In this section, we will look at three of these: dataflow computing, neural networks, and systolic processing.

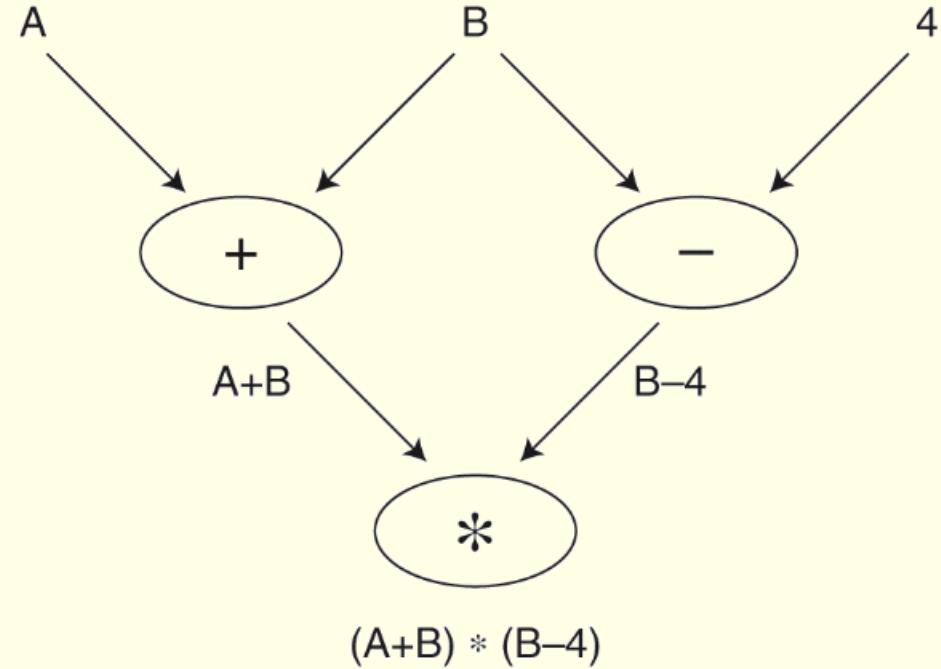
9.5 Alternative Parallel Processing Approaches

- Von Neumann machines exhibit sequential control flow: A linear stream of instructions is fetched from memory, and they act upon data.
- Program flow changes under the direction of branching instructions.
- In *dataflow* computing, program control is directly controlled by data dependencies.
- There is no program counter or shared storage.
- Data flows continuously and is available to multiple instructions simultaneously.

9.5 Alternative Parallel Processing Approaches

- A *data flow graph* represents the computation flow in a dataflow computer.

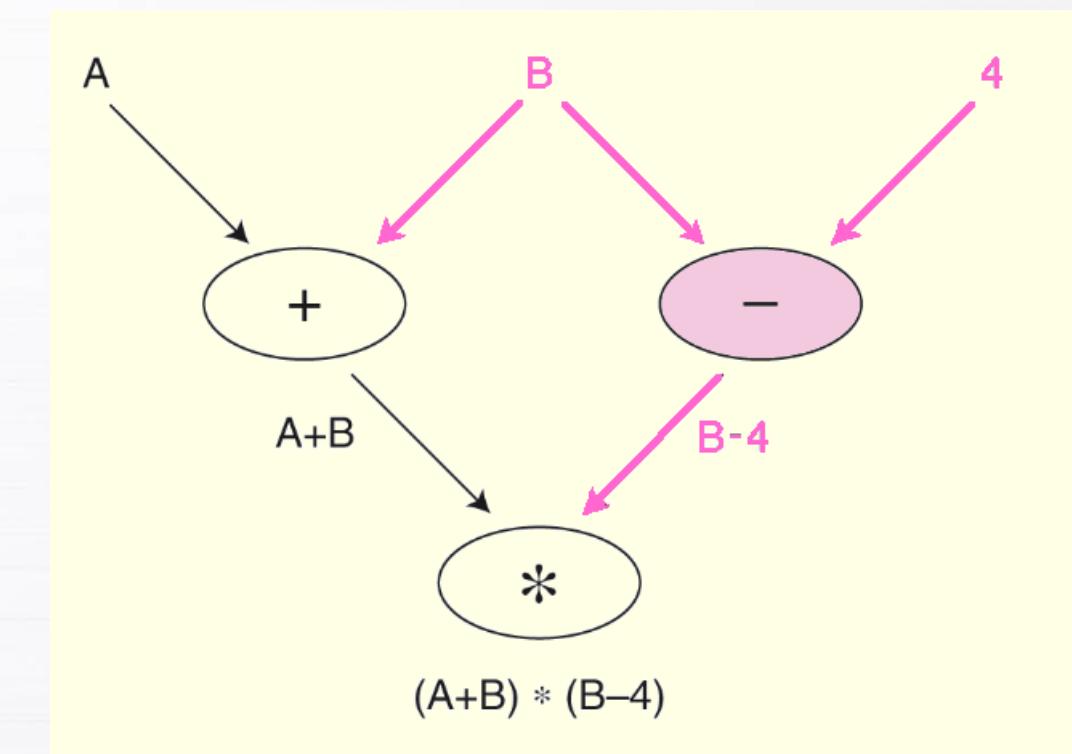
Its nodes contain the instructions and its arcs indicate the data dependencies.



9.5 Alternative Parallel Processing Approaches

- When a node has all of the data tokens it needs, it fires, performing the required operation, and consuming the token.

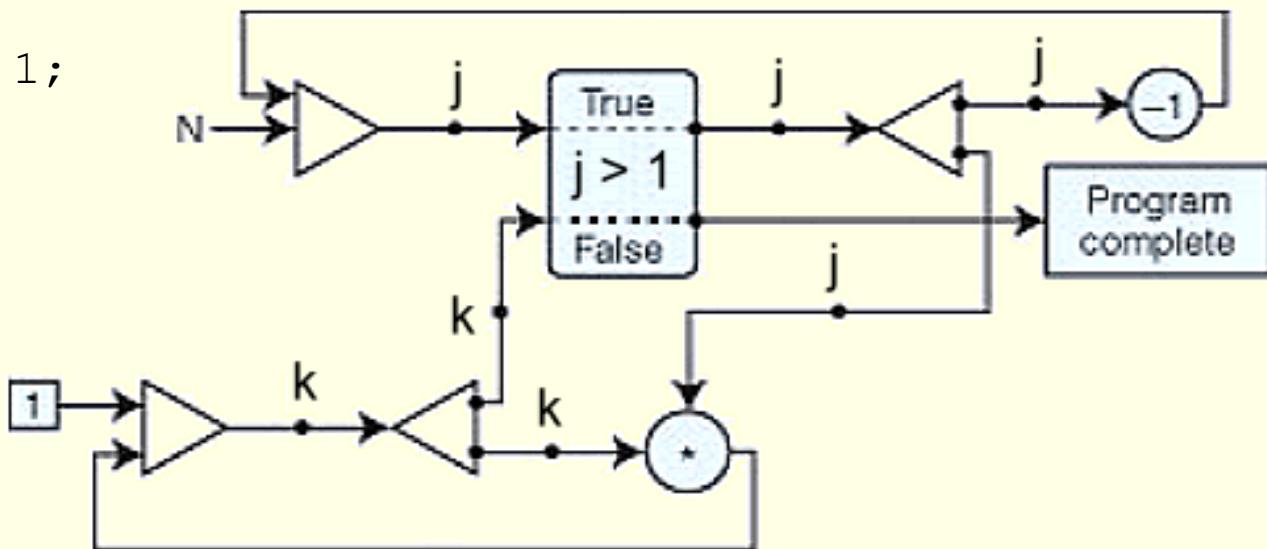
The result is placed on an output arc.



9.5 Alternative Parallel Processing Approaches

- A dataflow program to calculate $n!$ and its corresponding graph are shown below.

```
(initial j <- n; k <- 1  
  while j > 1 do  
    new k<- * j;  
    new j <- j - 1;  
return k)
```



9.5 Alternative Parallel Processing Approaches

- The architecture of a dataflow computer consists of processing elements that communicate with one another.
- Each processing element has an *enabling unit* that sequentially accepts tokens and stores them in memory.
- If the node to which this token is addressed fires, the input tokens are extracted from memory and are combined with the node itself to form an executable packet.

9.5 Alternative Parallel Processing Approaches

- Using the executable packet, the processing element's *functional unit* computes any output values and combines them with destination addresses to form more tokens.
- The tokens are then sent back to the enabling unit, optionally enabling other nodes.
- Because dataflow machines are data driven, multiprocessor dataflow architectures are not subject to the cache coherency and contention problems that plague other multiprocessor systems.

9.5 Alternative Parallel Processing Approaches

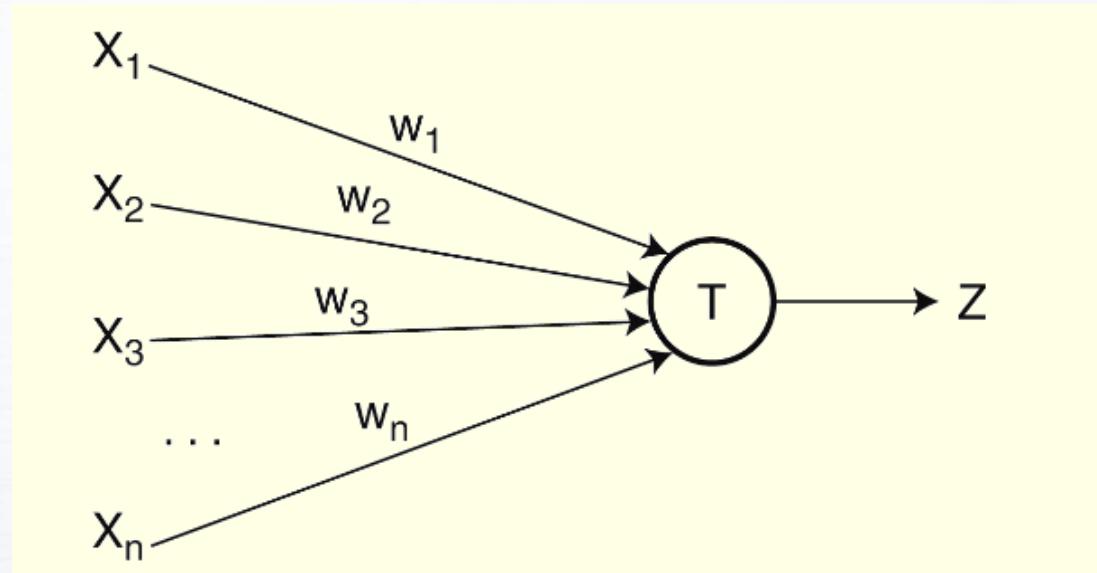
- *Neural network* computers consist of a large number of simple processing elements that individually solve a small piece of a much larger problem.
- They are particularly useful in dynamic situations that are an accumulation of previous behavior, and where an exact algorithmic solution cannot be formulated.
- Like their biological analogues, neural networks can deal with imprecise, probabilistic information, and allow for adaptive interactions.

9.5 Alternative Parallel Processing Approaches

- Neural network processing elements (PEs) multiply a set of input values by an adaptable set of weights to yield a single output value.
- The computation carried out by each PE is simplistic-- almost trivial-- when compared to a traditional microprocessor. Their power lies in their massively parallel architecture and their ability to adapt to the dynamics of the problem space.
- Neural networks learn from their environments. A built-in *learning algorithm* directs this process.

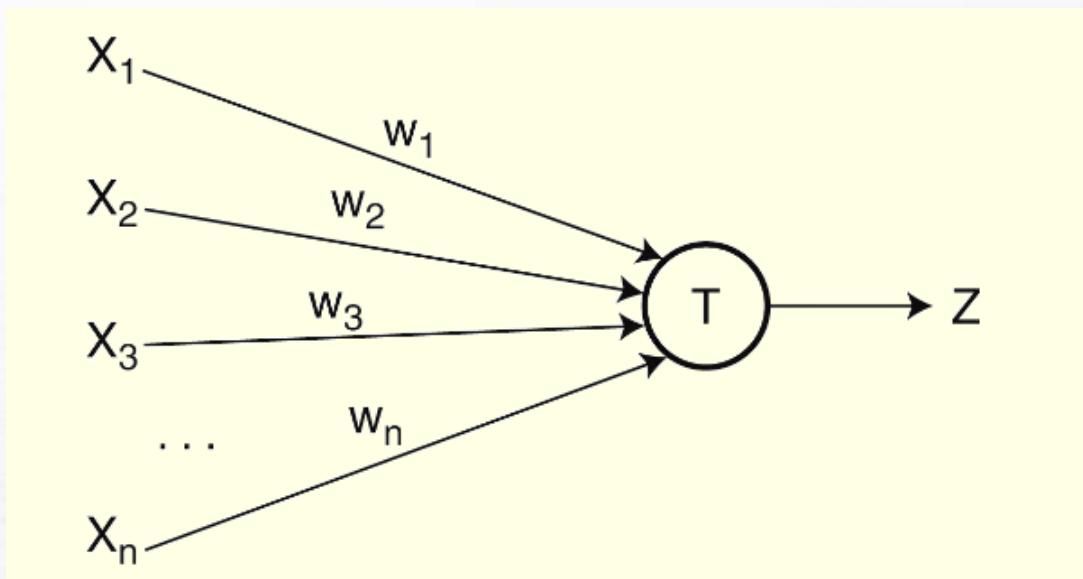
9.5 Alternative Parallel Processing Approaches

- The simplest neural net PE is the *perceptron*.
- Perceptrons are trainable neurons. A perceptron produces a Boolean output based upon the values that it receives from several inputs.



9.5 Alternative Parallel Processing Approaches

- Perceptrons are trainable because the threshold and input weights are modifiable.
- In this example, the output Z is true (1) if the net input, $w_1x_1 + w_2x_2 + \dots + w_nx_n$ is greater than the threshold T .



9.5 Alternative Parallel Processing Approaches

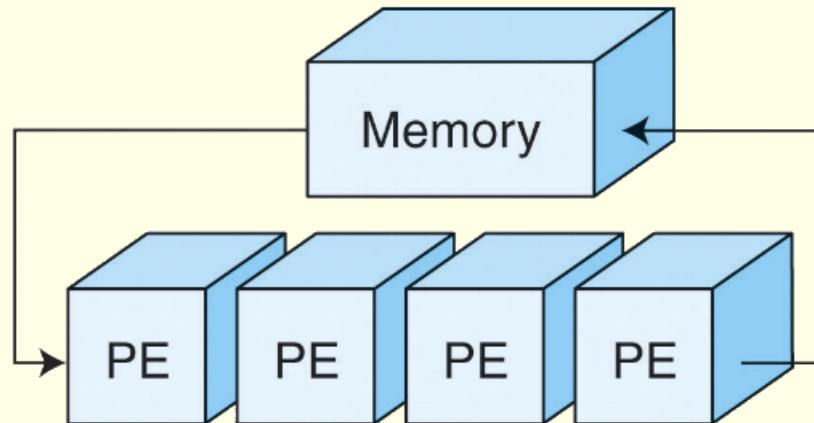
- Perceptrons are trained by use of supervised or unsupervised learning.
- Supervised learning assumes prior knowledge of correct results which are fed to the neural net during the training phase. If the output is incorrect, the network modifies the input weights to produce correct results.
- Unsupervised learning does not provide correct results during training. The network adapts solely in response to inputs, learning to recognize patterns and structure in the input sets.

9.5 Alternative Parallel Processing Approaches

- The biggest problem with neural nets is that when they consist of more than 10 or 20 neurons, it is impossible to understand how the net is arriving at its results. They can derive meaning from data that are too complex to be analyzed by people.
 - The U.S. military once used a neural net to try to locate camouflaged tanks in a series of photographs. It turned out that the nets were basing their decisions on the cloud cover instead of the presence or absence of the tanks.
- Despite early setbacks, neural nets are gaining credibility in sales forecasting, data validation, and facial recognition.

9.5 Alternative Parallel Processing Approaches

- Where neural nets are a model of biological neurons, *systolic array* computers are a model of how blood flows through a biological heart.
- Systolic arrays, a variation of SIMD computers, have simple processors that process data by circulating it through vector pipelines.



9.5 Alternative Parallel Processing Approaches

- Systolic arrays can sustain great throughput because they employ a high degree of parallelism.
- Connections are short, and the design is simple and scalable. They are robust, efficient, and cheap to produce. They are, however, highly specialized and limited as to the types of problems they can solve.
- They are useful for solving repetitive problems that lend themselves to parallel solutions using a large number of simple processing elements.
 - Examples include sorting, image processing, and Fourier transformations.

9.6. Quantum Computing

- Computers, as we know them are binary, transistor-based systems.
- But transistor-based systems strain to keep up with our computational demands.
- We increase the number of transistors for more power, and each transistor smaller to fit on the die.
 - Transistors are becoming so small that it is hard for them to hold electrons in the way in which we're accustomed to.
- Thus, alternatives to transistor-based systems are an active area of research.

9.6. Quantum Computing

- Computers are now being built based on:
 - Optics (photonic computing)
 - Biological neurons
 - DNA
- One of the most intriguing is quantum computers.
- Quantum computing uses quantum bits (qubits) that can be in multiple states at once.
- The "state" of a qubit is determined by the spin of an electron.

A thorough discussion of "spin" is under the domain of quantum physics.

9.6. Quantum Computing

- A qubit can be in multiple states at the same time.
 - This is called *superpositioning*.
- A 3-bit register can simultaneously hold the values 0 through 7
 - 8 operations can be performed at the same time.
- This phenomenon is called *quantum parallelism*.
 - A system with 600 qubits can superposition 2^{600} states

9.6. Quantum Computing

- D-Wave Computers is the first quantum computer manufacturer
- D-Wave computers having 512 qbits were purchased separately by University of Southern California and Google for research purposes.
- Quantum computers may be applied in the areas of cryptography, true random-number generation, and in the solution of other intractable problems.

9.6. Quantum Computing

- Making effective use of quantum computers requires rethinking our approach to problems and the development of new algorithms.
 - To break a cypher, the quantum machine simulates every possible state of the problem set (i.e., every possible key for a cipher) and it “collapses” on the correct solution.
- Examples include Schor’s algorithm for factoring products of prime numbers.
- Many others remain to be discovered.

9.6. Quantum Computing

- These systems are not constrained by a fetch-decode-execute cycle; however, quantum architectures have yet to settle on a definitive paradigm analogous to von Neumann systems.
- Rose's Law states that the number of qubits that can be assembled to successfully perform computations will double every 12 months; this has been precisely the case for the past nine years
 - This “law” is named after Geordie Rose, D-Wave’s founder and chief technology officer.

9.6. Quantum Computing

- One of the largest obstacles to the progress of quantum computation is the tendency for qubits to decay into a state of *decoherence*.
 - Decoherence causes uncorrectable errors.
- Advanced error-correction algorithms have been applied to this problem and show promise.
- Much research remains to be done, however.

9.6. Quantum Computing

- The realization of quantum computing has raised questions about technological singularity.
 - Technological singularity is the theoretical point when human technology has fundamentally and irreversibly altered human development
 - This is the point when civilization changes to an extent that its technology is incomprehensible to previous generations.
- Are we there, now?

Chapter 9 Conclusion

- The common distinctions between RISC and CISC systems include RISC's short, fixed-length instructions. RISC ISAs are load-store architectures. These things permit RISC systems to be highly pipelined.
- Flynn's Taxonomy provides a way to classify multiprocessor systems based upon the number of processors and data streams. It falls short of being an accurate depiction of today's systems.

Chapter 9 Conclusion

- Massively parallel processors have many processors, distributed memory, and computational elements communicate through a network. Symmetric multiprocessors have fewer processors and communicate through shared memory.
- Characteristics of superscalar design include superpipelining, and specialized instruction fetch and decoding units.

Chapter 9 Conclusion

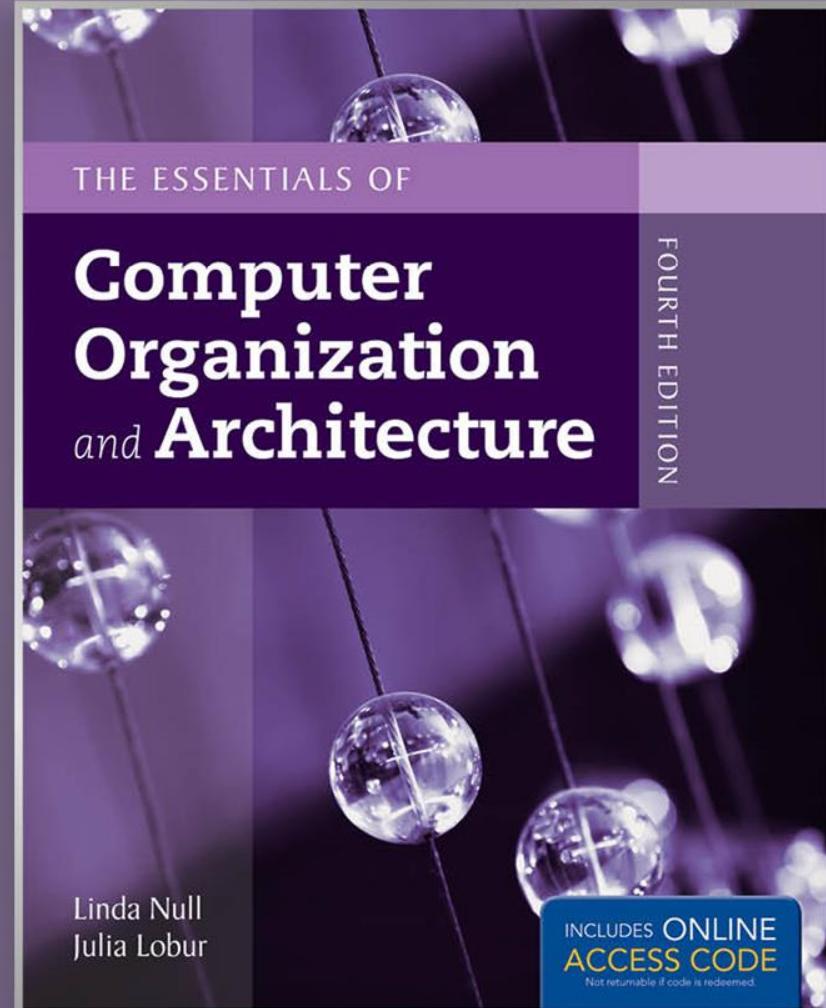
- Very long instruction word (VLIW) architectures differ from superscalar architectures because the compiler, instead of a decoding unit, creates long instructions.
- Vector computers are highly-pipelined processors that operate on entire vectors or matrices at once.
- MIMD systems communicate through networks that can be blocking or nonblocking. The network topology often determines throughput.

Chapter 9 Conclusion

- Multiprocessor memory can be distributed or exist in a single unit. Distributed memory brings to rise problems with cache coherency that are addressed using cache coherency protocols.
- New architectures are being devised to solve intractable problems. These new architectures include dataflow computers, neural networks, systolic arrays and quantum computers.

Chapter 2

Floating Point Numbers



2.5 Floating-Point Representation

- The signed magnitude, one's complement, and two's complement representation that we have just previously discussed deal with signed integer values only.
- Without modification, these formats are not useful in scientific or business applications that deal with real number values.
- Floating-point representation solves this problem.

2.5 Floating-Point Representation

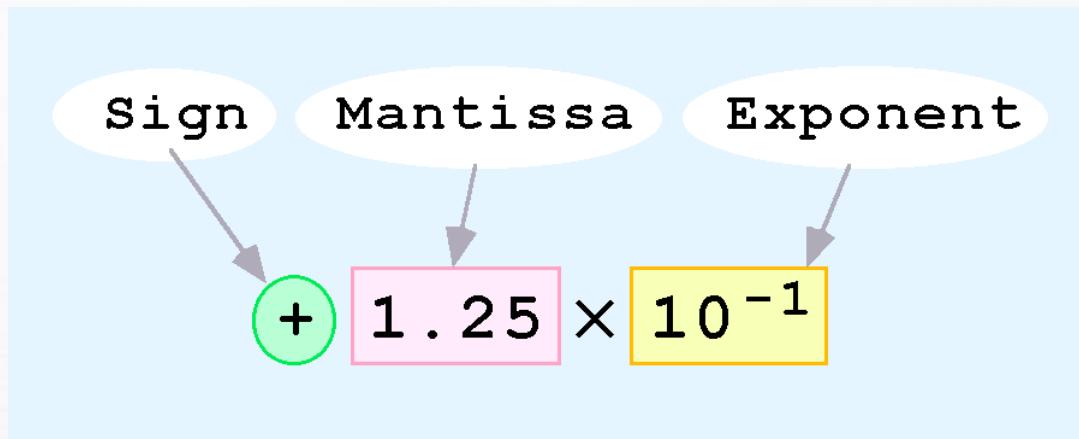
- If we are clever programmers, we can perform floating-point calculations using any integer format.
- This is called *floating-point emulation*, because floating point values aren't stored as such; we just create programs that make it seem as if floating-point values are being used.
- Most of today's computers are equipped with specialized hardware that performs floating-point arithmetic with no special programming required.
 - Other than using the provided instruction set of your CPU architecture

2.5 Floating-Point Representation

- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
 - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
 - For example:
$$0.125 = 1.25 \times 10^{-1}$$
$$5,000,000 = 5.0 \times 10^6$$

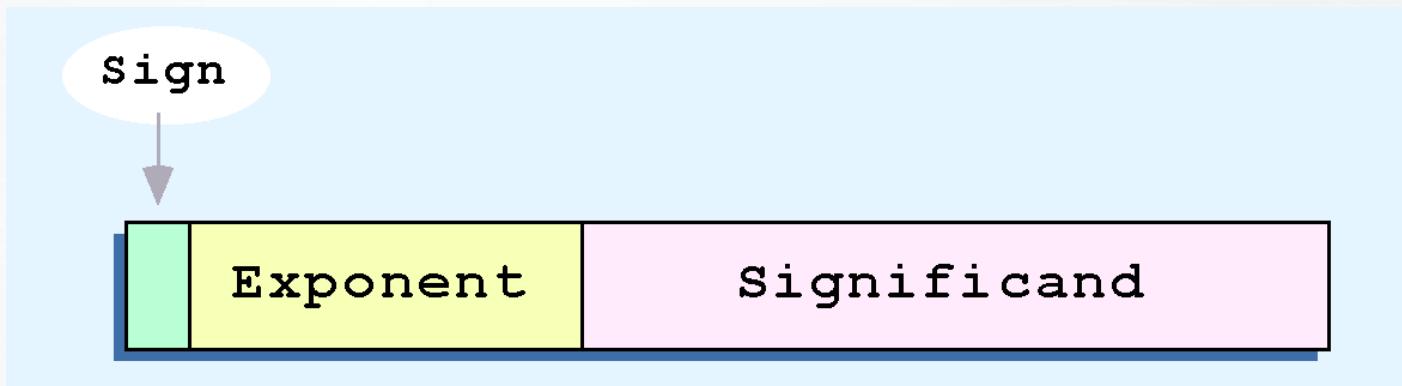
2.5 Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



2.5 Floating-Point Representation

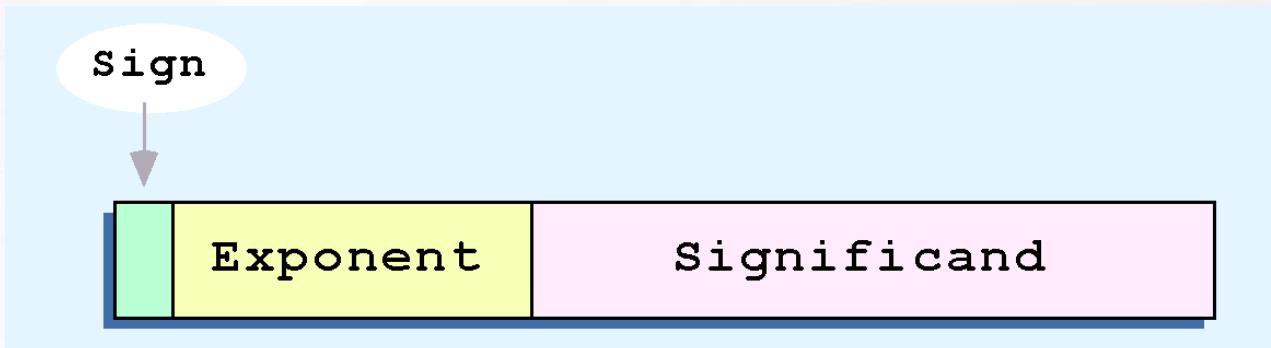
- Computer representation of a floating-point number consists of three fixed-size fields:



- This is the standard arrangement of these fields.

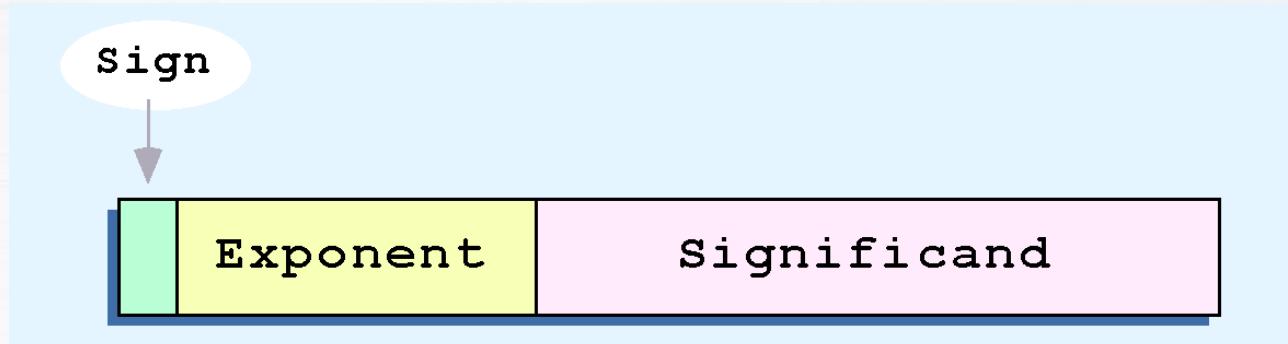
Note: Although “significand” and “mantissa” do not technically mean the same thing, many people use these terms interchangeably. We use the term “significand” to refer to the fractional part of a floating point number.

2.5 Floating-Point Representation



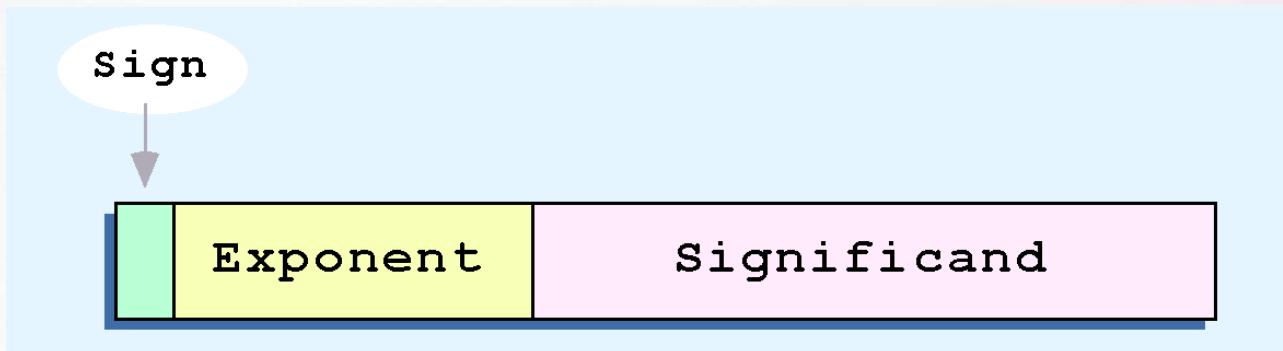
- The one-bit sign field is the sign of the stored value.
- The size of the exponent field determines the **range** of values that can be represented.
- The size of the significand determines the **precision** of the representation.

2.5 Floating-Point Representation



- We introduce a hypothetical “Model” to explain the concepts, after which we will discuss the IEEE-754 one.
- In this model:
 - A floating-point number is 14 bits in length
 - The exponent field is 5 bits
 - The significand field is 8 bits

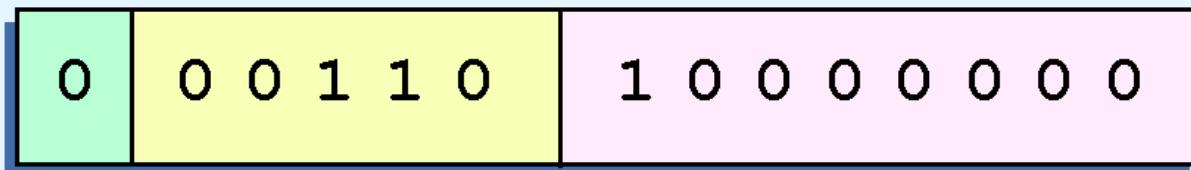
2.5 Floating-Point Representation



- The significand is always preceded by an implied binary point.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 by which the significand is multiplied.

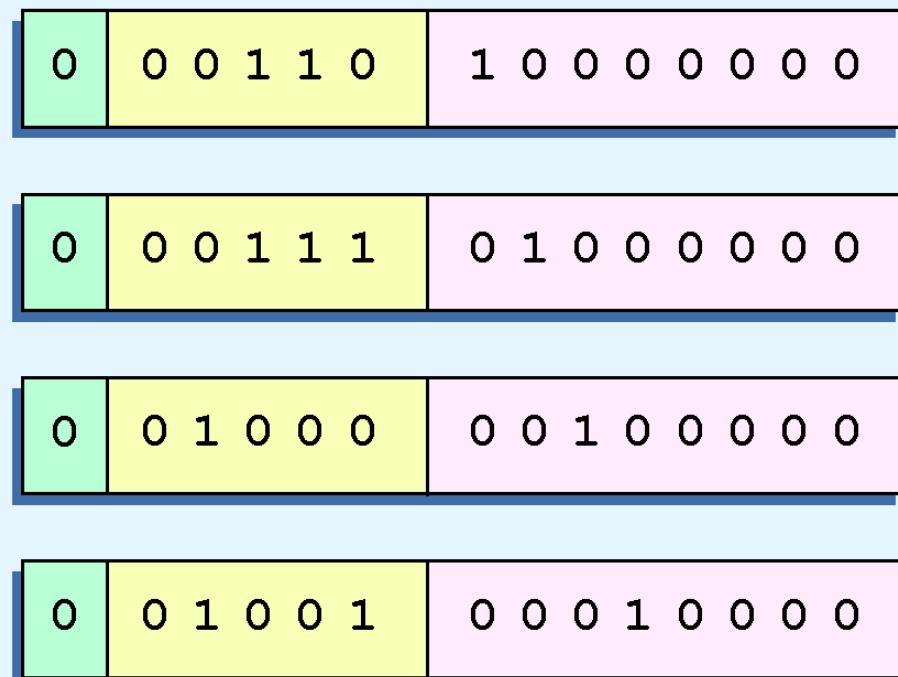
2.5 Floating-Point Representation

- Example:
 - Express 32_{10} in the simplified 14-bit floating-point model. (1-bit sign, 5-bit exponent, 9-bit significand)
- We know that 32 is 2^5 . So in (binary) scientific notation $32 = 100000_2 = 0.1_2 \times 2^6$.
- Using this information, we put 110 ($= 6_{10}$) in the exponent field and 1 in the significand as shown.



2.5 Floating-Point Representation

- The illustrations shown at the right are *all* equivalent representations for 32 using our simplified model.
- Not only do these synonymous representations waste space, but they can also cause confusion.



2.5 Floating-Point Representation

- To resolve the problem of synonymous forms, we establish a rule that the first digit of the significand must be 1, with no ones to the left of the radix point.
- This process, called *normalization*, results in a unique pattern for each floating-point number.
 - In our simple model, all significands must have the form 0.1xxxxxxxxx
 - For example, $4.5 = 100.1 \times 2^0 = 1.001 \times 2^2 = 0.1001 \times 2^3$. The last expression is correctly normalized.

In our simple instructional model, we use no implied bits.

2.5 Floating-Point Representation



- Another problem with our system is that we have made no allowances for negative exponents. We have no way to express 0.25! (Notice that there is no sign in the exponent field.)

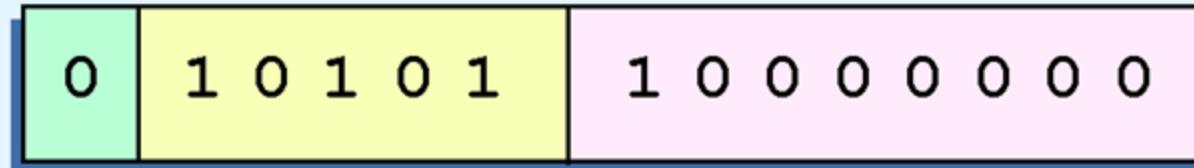
All of these problems can be fixed with no changes to our basic model.

2.5 Floating-Point Representation

- To provide for negative exponents, we will use a *biased exponent*.
 - In our case, we have a 5-bit exponent.
 - $2^{5-1} - 1 = 2^4 - 1 = 15$
 - Thus will use 15 for our bias: our exponent will use *excess-15* representation.
- In our model, exponent values less than 15 are negative, representing purely fractional numbers.

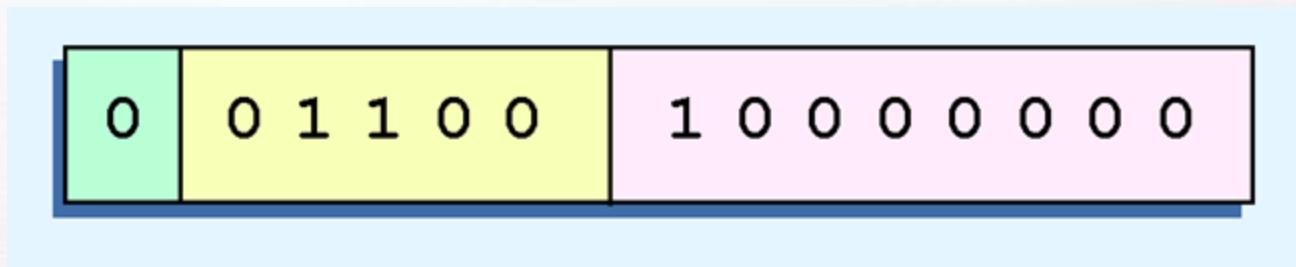
2.5 Floating-Point Representation

- Example:
 - Express 32_{10} in the revised 14-bit floating-point model.
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 15 biased exponent, we add 15 to 6, giving 21_{10} ($=10101_2$).
- So we have:



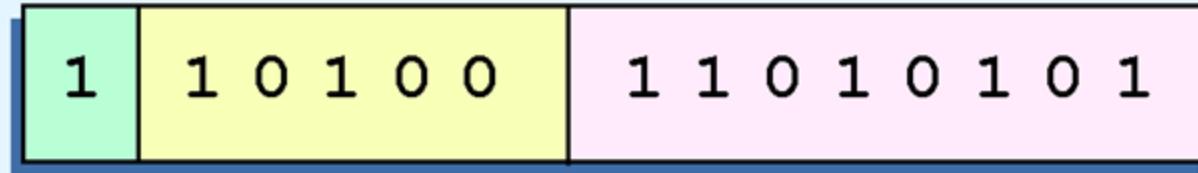
2.5 Floating-Point Representation

- Example:
 - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4} . So in (binary) scientific notation $0.0625 = 0.0001_2 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 15 biased exponent, we add 15 to -3, giving 12_{10} ($=01100_2$).



2.5 Floating-Point Representation

- Example:
 - Express -26.625_{10} in the revised 14-bit floating-point model.
- We find $26.625_{10} = 11010.101_2$. Normalizing, we have: $26.625_{10} = 0.11010101 \times 2^5$.
- To use our excess 15 biased exponent, we add 15 to 5, giving 20_{10} ($=10100_2$). We also need a 1 in the sign bit.



2.5 Floating-Point Representation

- The IEEE has established a standard for floating-point numbers
- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent (with a bias of 127) and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent (with a bias of 1023) and a 52-bit significand.

2.5 Floating-Point Representation

- In both the IEEE single-precision and double-precision floating-point standard, the significant has an implied 1 to the LEFT of the radix point.
 - The format for a significand using the IEEE format is:
1.xxx...
 - For example, $4.5 = .1001 \times 2^3$ in IEEE format is $4.5 = 1.001 \times 2^2$. The 1 is implied, which means it does not need to be listed in the significand (the significand would include only 001).

2.5 Floating-Point Representation

- Example: Express -3.75 as a floating point number using IEEE single precision.
- First, let's normalize according to IEEE rules:
 - $-3.75 = -11.11_2 = -1.111 \times 2^1$
 - The bias is 127, so we add $127 + 1 = 128$ (this is our exponent)



(implied)

- Since we have an implied 1 in the significand, this equates to
$$-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75.$$

2.5 Floating-Point Representation

- Using the IEEE-754 single precision floating point standard:
 - An exponent of 255(after adding the bias (all 1's)) indicates a special value.
 - If the significand is zero, the value is \pm infinity.
 - If the significand is nonzero, the value is NaN, “not a number,” often used to flag an error condition.
- Using the double precision standard:
 - The “special” exponent value for a double precision number is 2047, instead of the 255 used by the single precision standard.

2.5 Floating-Point Representation

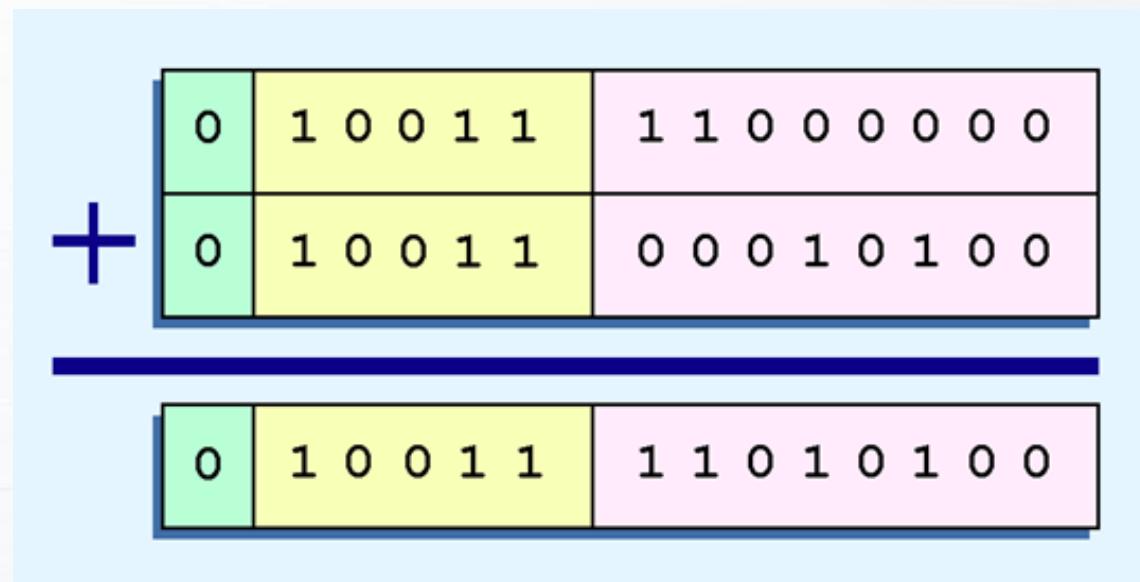
- Both the 14-bit model that we have presented and the IEEE-754 floating point standard allow two representations for zero.
 - Zero is indicated by all zeros in the exponent and the significand, but the sign bit can be either 0 or 1.
- This is why programmers should avoid testing a floating-point value for equality to zero.
 - Negative zero does not equal positive zero.

2.5 Floating-Point Representation

- IEEE Floating-point **addition** and **subtraction** are done using methods **analogous** to how we perform calculations using pencil and paper.
- The first thing that we do is express both operands in the same exponential power, then add the numbers, preserving the exponent in the sum.
- If the exponent requires adjustment, we do so at the end of the calculation.

2.5 Floating-Point Representation

- Example:
 - Find the sum of 12_{10} and 1.25_{10} using the 14-bit “simple” floating-point model.
- We find $12_{10} = 0.1100 \times 2^4$. And $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$.
- Thus, our sum is 0.110101×2^4 .



2.5 Floating-Point Representation

- Floating-point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper.
- We **multiply** the two operands and **add** their exponents.
- If the exponent requires adjustment, we do so at the end of the calculation.

2.5 Floating-Point Representation

- No matter how many bits we use in a floating-point representation, our model must be finite.
- The real number system is, of course, infinite, so our models can give nothing more than an approximation of a real value.
- At some point, every model breaks down, introducing errors into our calculations.
- By using a greater number of bits in our model, we can reduce these errors, but we can never totally eliminate them.

2.5 Floating-Point Representation

- Consider
- 0.1 in decimal.
- It cannot be perfectly represented in binary..
- $0.1_{10} =$
 $0.000110011001100110011 \dots {}_2$

2.5 Floating-Point Representation

- Our job becomes one of reducing error, or at least being aware of the possible magnitude of error in our calculations.
- We must also be aware that errors can compound through repetitive arithmetic operations.
- For example, our 14-bit model cannot exactly represent the decimal value 128.5. In binary, it is 9 bits wide:

$$10000000.1_2 = 128.5_{10}$$

2.5 Floating-Point Representation

- When we try to express 128.5_{10} in our 14-bit model, we lose the low-order bit, giving a relative error of:

$$\frac{128.5 - 128}{128.5} \approx 0.39\%$$

- If we had a procedure that repetitively added 0.5 to 128.5, we would have an error of nearly 2% after only four iterations.

2.5 Floating-Point Representation

- Floating-point errors can be reduced when we use operands that are similar in magnitude.
- If we were repetitively adding 0.5 to 128.5, it would have been better to iteratively add 0.5 to itself and then add 128.5 to this sum.
- In this example, the error was caused by loss of the low-order bit.
- Loss of the high-order bit is more problematic.

2.5 Floating-Point Representation

- Floating-point overflow and underflow can cause programs to crash.
- Overflow occurs when there is no room to store the high-order bits resulting from a calculation.
- Underflow occurs when a value is too small to store, possibly resulting in division by zero.

Experienced programmers know that it's better for a program to crash than to have it produce incorrect, but plausible, results.

2.5 Floating-Point Representation

- When discussing floating-point numbers, it is important to understand the terms *range*, *precision*, and *accuracy*.
- The range of a numeric integer format is the difference between the largest and smallest values that can be expressed.
- Accuracy refers to how closely a numeric representation approximates a true value.
- The precision of a number indicates how much information we have about a value

2.5 Floating-Point Representation

- Most of the time, greater precision leads to better accuracy, but this is not always true.
 - For example, 3.1333 is a value of pi that is accurate to two digits, but has 5 digits of precision.
- There are other problems with floating point numbers.
- Because of truncated bits, you cannot always assume that a particular floating point operation is commutative or distributive.

2.5 Floating-Point Representation

- This means that we cannot assume:

$$(a + b) + c = a + (b + c) \text{ or}$$

$$a * (b + c) = ab + ac$$

- Moreover, to test a floating point value for equality to some other number, it is best to declare a “nearness to x ” epsilon value. For example, instead of checking to see if floating point x is equal to 2 as follows:

if $x = 2$ then ...

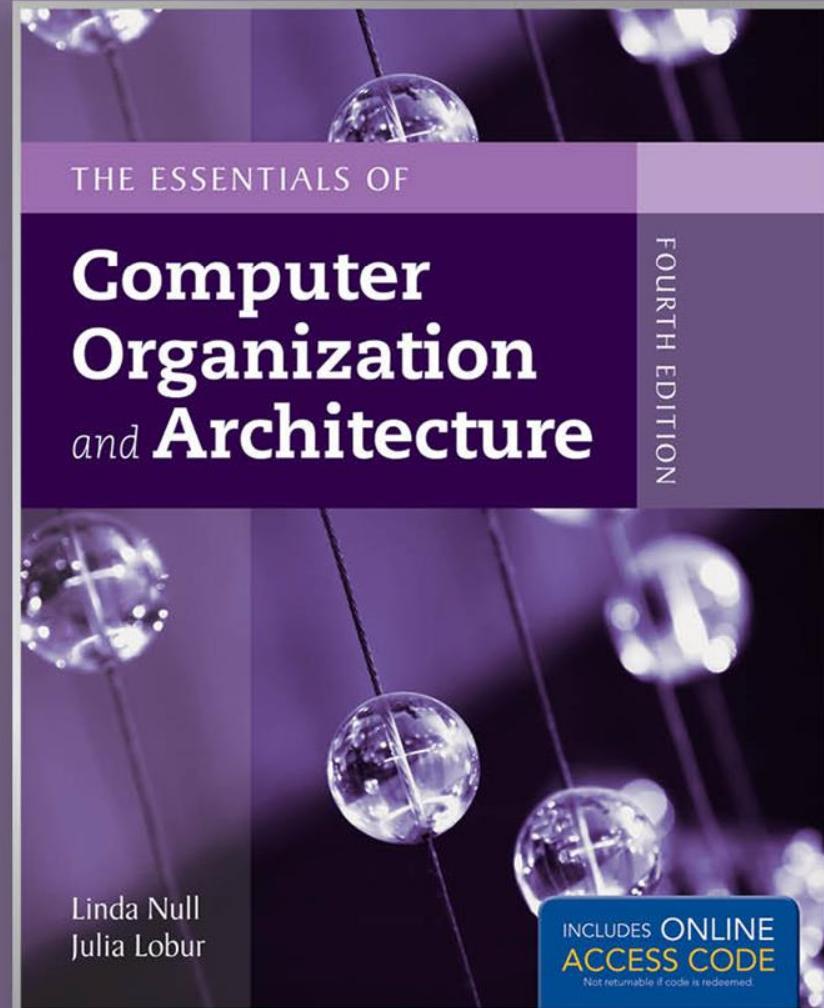
it is better to use:

if $(\text{abs}(x - 2) < \text{epsilon})$ then ...

(assuming we have epsilon defined correctly!)

Chapter 2.6

Character Codes



2.6 Character Codes

- Calculations aren't useful until their results can be displayed in a manner that is meaningful to people.
- We also need to store the results of calculations, and provide a means for data input.
- Thus, human-understandable characters must be converted to computer-understandable bit patterns using some sort of character encoding scheme.
- As computers have evolved, character codes have evolved.
- Larger computer memories and storage devices permit richer character codes.

2.6.1 Binary-Coded Decimal

- For many applications decimal digits need to be exactly represented, e.g. financial applications.
 - What is the binary equivalent of R 5.10?
- For these cases, we need an encoding for individual decimal digits (a form of character coding).
- Binary-coded decimal (BCD) is common in electronics that use numerical data, such as alarm clocks and calculators.
- BCD encodes each digit of a decimal number into a 4-bit binary form.
- E.g. 146 in BCD is

0001 0100 0110

Decimal Digit	BCD 8 4 2 1
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

2.6.1 Binary Coded Decimal

- Unpacked BCD: high order nibbles are padded with zeros.
 - Example: 146: 00000001 00000100 00000110
 - Unpacked BCD is wasteful.
- Packed BCD: stores two digits per byte.
 - Sign is stored at the end:
 - 1111: unsigned
 - 1100: positive
 - 1101: negative
 - Example: -146 would be stored as 00010100 01101101
 - Pad high-order nibble with zeros if necessary (e.g. -61)
- Disadvantage of BCD:
 - Many bits go to waste (1010 to 1011 not used)
- Advantage of BCD:
 - Can represent decimal numbers accurately.

2.6.1 Binary Coded Decimal

- Zoned decimal format BCD:
 - Same as for unpacked BCD, except that the high-order nibbles store a code High-order nibble of the last byte is the sign (1100: positive, 1101: negative)
 - Example: +146 in EBCDIC zoned decimal format is **11110001 11110100 11000110**
- Example: Represent -1265 using packed BCD and zoned BCD decimal.
 - Packed BCD: **00000001 00100110 01011101**
 - Zoned BCD: **11110001 11110010 11110110 11010101**

2.6.2 EBCDIC

- Early version of BCD on IBM systems used 6-bit representation for characters and numbers (very limited, e.g. no lowercase letters).
- In 1964, BCD was extended to an 8-bit code, **Extended Binary-Coded Decimal Interchange Code (EBCDIC)**.
- EBCDIC was one of the first widely-used computer codes that supported upper *and* lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
- EBCDIC still used by IBM mainframes today.
- Bytes in EBCDIC are in zone-digit form, e.g. character ‘a’ is 1000 0001 and digit ‘3’ is 1111 0011.

2.6.2 EBCDIC

Zone	Digit															
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NUL	SOH	STX	ETX	PF	HT	LC	DEL		RLF	SMM	VT	FF	CR	SO	SI
0001	DLE	DC1	DC2	TM	RES	NL	BS	IL	CAN	EM	CC	CU1	IFS	IGS	IRS	IUS
0010	DS	SOS	FS		BYP	LF	ETB	ESC			SM	CU2		ENQ	ACK	BEL
0011			SYN		PN	RS	UC	EOT				CU3	DC4	NAK		SUB
0100	SP									[.	<	(+	!	
0101	&]	\$	*)	;	^	
0110	-	/									,	%	-	>	?	
0111									:	#	@	'	=	"		
1000	a	b	c	d	e	f	g	h	i							
1001	j	k	l	m	n	o	p	q	r							
1010	~	s	t	u	v	w	x	y	z							
1011																
1100	{	A	B	C	D	E	F	G	H	I						
1101	}	J	K	L	M	N	O	P	Q	R						
1110	\	S	T	U	V	W	X	Y	Z							
1111	0	1	2	3	4	5	6	7	8	9						

2.6.3 ASCII

- Other computer manufacturers chose the 7-bit ASCII (American Standard Code for Information Interchange).
- Until recently, ASCII was the dominant character code outside the IBM mainframe world.
- The 8th bit was intended to be used for parity.

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	'
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	\$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[59	3B	;	91	5B	[123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

2.6.3 ASCII

- Parity: the most basic of all error-detection schemes.
- A parity bit is 1 or 0 depending on whether the sum of the other bits is odd or even.
- E.g. ASCII ‘A’ is 65_{10} or 100 0001, so the bit string transmitted would be 0100 0001 (Using even parity).
- Parity can be used to detect only single-bit errors.

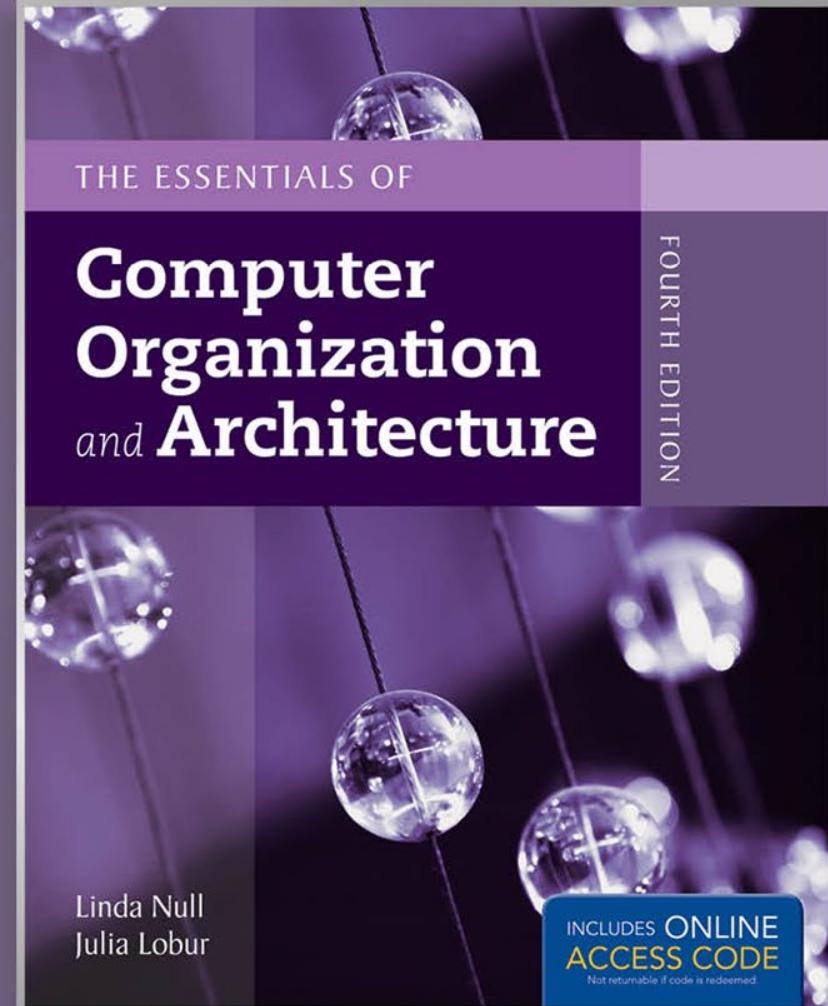
2.6.4 Unicode

- Many of today's systems embrace Unicode, a **16-bit system** that can encode the characters of every language in the world.
- Downward compatible with ASCII.
- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

Chapter 2.7

Error Detection and Correction



2.7 Error Detection and Correction

- It is physically impossible for any data recording or transmission medium to be 100% perfect 100% of the time over its entire expected useful life.
- As more bits are packed onto a square centimeter of disk storage, as communications transmission speeds increase, the likelihood of error inevitably increases.
- Thus, error detection and correction is critical to accurate data transmission, storage and retrieval.
- Check digits, appended to the end of a long number, can provide some protection against data input errors.
 - The last characters of ISBNs and UP student numbers are check digits.

2.7 Parity

- The simplest error detection method is that of parity.
 - Given a bit string of length N
 - We rather transmit a string of length N+1
 - The highest order bit, the parity bit, is used to store additional information about the lower order bit.
 - Two options
 - Odd parity:
 - If the number of 1's in the N-bits is odd then the parity bit is 0, else 1
 - Even Parity:
 - If the number of 1's in the N-bits is even then the parity bit is 0, else 1

2.7 Parity

- Assume even parity is being used:
- Original string: 111 0101
 - # of 1's is odd so parity bit is 1
- Sent string is: **1111 0101**
 - Observe that the number of 1's are now even.

2.7 Parity

- While the use of a parity bit is simple it has two major issues:
 - It can only usually detect at most 1 error!
 - It only tells that an error occurred, not where the error occurred.

2.7.1 Cyclic Redundancy Check

- A cyclic redundancy check (CRC) is a type of checksum that is used in error detection for large blocks of data.
- Checksums and CRCs are examples of ***systematic error detection***.
- In ***systematic error detection*** a group of error control bits is appended to the end of the block of transmitted data.
 - This group of bits is called a ***syndrome***.
 - Original information is unchanged by the addition of the error-checking bits.
- CRCs use polynomials over the modulo 2 arithmetic field.

The mathematical theory behind modulo 2 polynomials is beyond our scope. However, we can easily work with it without knowing its theoretical underpinnings.

2.7.1 Cyclic Redundancy Check

- Modulo 2 arithmetic works like clock arithmetic.
- In clock arithmetic, if we add 2 hours to 11:00, we get 1:00.
- In modulo 2 arithmetic if we add 1 to 1, we get 0. The addition rules couldn't be simpler:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0$$

- Example: Find the sum of 1011_2 and 110_2 modulo 2.

$$\begin{array}{r} 1011 \\ 0110 + (\text{mod } 2) \\ \hline 1101 \end{array}$$

2.7.1 Cyclic Redundancy Check

- Example: Find the **quotient** and **remainder** when **1111101** is divided by **1101** in modulo 2 arithmetic.
 - As with traditional division, we note that the dividend is divisible once by the divisor.
 - We place the divisor under the dividend and perform modulo 2 subtraction.

$$\begin{array}{r} & 1 \\ 1101) & \underline{1111101} \\ & \underline{1101} \\ & 0010 \end{array}$$

2.7.1 Cyclic Redundancy Check

- Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic...
 - Now we bring down the next bit of the dividend.
 - We see that 00101 is not divisible by 1101. So we place a zero in the quotient.

$$\begin{array}{r} & 10 \\ 1101) & 1111101 \\ & \underline{1101} \\ & 00101 \end{array}$$

2.7.1 Cyclic Redundancy Check

- Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic...
 - 1010 is divisible by 1101 in modulo 2.
 - We perform the modulo 2 subtraction.

$$\begin{array}{r} 101 \\ 1101) 1111101 \\ \underline{1101} \\ 001010 \\ \underline{1101} \\ 0111 \end{array}$$

2.7.1 Cyclic Redundancy Check

- Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic...
 - We find the quotient is 1011, and the remainder is 0010.
- This procedure is very useful to us in calculating CRC syndromes.

$$\begin{array}{r} 1011 \\ 1101 \) 1111101 \\ 1101 \\ \hline 001010 \\ 1101 \\ \hline 01111 \\ 1101 \\ \hline 0010 \end{array}$$

Note: The divisor in this example corresponds to a modulo 2 polynomial: $X^3 + X^2 + 1$.

2.7.1 Cyclic Redundancy Check

- How do we use this to perform error detection?
- Both parties decide on a CRC-Polynomial (P)
- The algorithm is then as follows for sender:
 - 1) Given a information block I , Left shift $|P| - 1$ bits.
 - 2) Perform modulo 2 division of I by P
 - Find the remainder, R
 - 3) Set $I = I + R$ (normal binary addition)
- The algorithm is the as follows for a receiver
 - 1) Receive I
 - 2) Perform modulo 2 division of I by P
 - 3) If $R = 0$ then transmission was successful
 - 4) Else there was an error.

2.7.1 Cyclic Redundancy Check (example)

- $I = 1001011_2$ and $P = 1011_2$
- $I = I \ll 3 = 1001011000_2$
- Module 2 divide I by P
 - $Q = 1010100$ and $R = 100$ (check at home)
- $I = I + R = 1001011100$
- Transmit.
 - Then if Module 2 divide I by P results in $R = 0$, transmission successful.
 - If $R \neq 0$ at least 1 bit error occurred.

2.7.1 Cyclic Redundancy Check

- Questions to think about
 - If P=11, how does CRC relate to parity?

2.7 Error Detection and Correction

- Data transmission errors are easy to fix once an error is detected.
 - Just ask the sender to transmit the data again.
- In computer memory and data storage, however, this cannot be done.
 - Too often the only copy of something important is in memory or on disk.
- Thus, to provide data integrity over the long term, error ***correcting codes*** are required.
 - We want to be able to detect and correct.
- **Hamming codes** and **Reed-Solomon** codes are two important error correcting codes.

2.7.2 Hamming Codes

- Hamming codes are code words formed by adding redundant check bits, or parity bits, to a data word.
- The memory word itself consists of m bits, but r redundant bits are added to allow for error detection and/or correction.
- The *Hamming distance* between two code words is the number of bits in which two code words differ (3 in example).

1	0	0	0	1	0	0	1
1	0	1	1	0	0	0	1
- The minimum Hamming distance for a code is the smallest Hamming distance between *all* pairs of words in the code.
- Example:
 - Assume memory with 2 data bits and 1 parity bit (appended at the end) that uses even parity (number of 1s in code word must be even).
 - Possible words in the code: 000, 011, 101, 110 (other possible bit patterns are invalid).
 - What is the minimum Hamming distance for this code?

2.7.2 Hamming Codes

- The minimum Hamming distance for a code, $D(\min)$, determines its error detecting and error correcting capability.
- For a single-parity bit code:
 - Only single-bit errors can be detected, because $D(\min)$ is 2 (two bit errors can convert a valid code into another valid code).
- In general (for detection):
 - For any code word, X , to be interpreted as a different valid code word, Y , at least $D(\min)$ single-bit errors must occur in X .
 - Thus, to detect k (or fewer) single-bit errors, the code must have a Hamming distance of $D(\min) = k + 1$.
 - Conversely, Hamming codes can always detect $D(\min) - 1$ errors.
- For correction:
 - Hamming codes can correct $\left\lfloor \frac{D(\min) - 1}{2} \right\rfloor$ errors.
 - Or, the Hamming distance of a code must be at least $2k + 1$ for it to be able to correct k errors.

Example 2.41

- Suppose we have the following code:

0 0 0 0 0

0 1 0 1 1

1 0 1 1 0

1 1 1 0 1

- What is $D(\min)$?
 - Answer: $D(\min) = 3$
- How many single-bit errors can be detected?
 - Answer: up to 2 single-bit errors
- How many single-bit errors can be corrected?
 - Answer: 1 single-bit errors can be corrected
- What is the correct code of the invalid code word 10000?
 - Difference vector to all code words: [1, 4, 2, 3], so correction is 00000 (closest code word). Assumption: the minimum number of possible errors has occurred. Correction might not be correct!
- What is the correct code of the invalid code word 11000?

2.7.2 Hamming Codes

Single bit correction

- Suppose we have a set of n -bit code words consisting of m data bits and r (redundant) check bits.
- How many legal code words?
 - Answer: 2^m
- How many illegal code words at a Hamming distance of 1 from each valid code word (i.e. bit strings with single-bit errors)?
 - Answer: n (an error can occur in any of the n bit positions)
- How many total bit patterns (valid and invalid) possible?
 - Answer: 2^n or $2^{(m+r)}$
- For each valid codeword, we have $(n+1)$ bit patterns (1 legal and n illegal). This gives us the inequality:

$$(n + 1) \times 2^m \leq 2^n$$

2.7.2 Hamming Codes

- From previous slide:

$$(n + 1) \times 2^m \leq 2^n$$

- Because $n = m + r$, we can rewrite the inequality as:

$$(m + r + 1) \times 2^m \leq 2^{m+r}$$

or $(m + r + 1) \leq 2^r$

- This inequality gives us a lower limit on the number of check bits that we need to construct a code with m data bits and r check bits that corrects all single-bit errors.
- For example, say we have data words of length 4. What is the minimum number of check bits needed for correcting single-bit errors?

$$(4 + r + 1) \leq 2^r$$

r must be greater than or equal to 3.

- We should always use the smallest value of r that makes the inequality true.

2.8 Error Detection and Correction

- For example, say we have data words of length 8. What is the minimum number of check bits needed for correcting single-bit errors?

$$(8 + r + 1) \leq 2^r$$

r must be greater than or equal to 4.

- How long will the resulting code words be?

Answer: 12

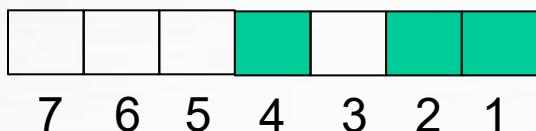
- So how do we assign values to check bits so that we can achieve detection and correction?

The Hamming Algorithm

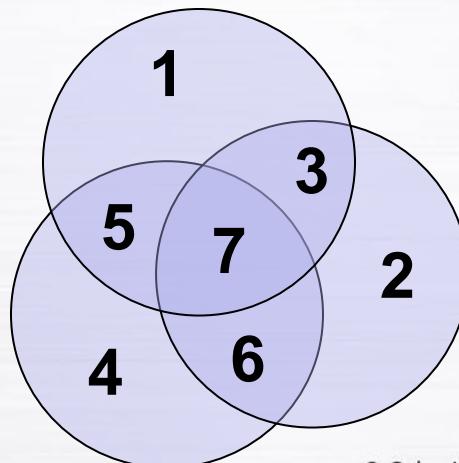
1. Determine the number of check bits, r , necessary for the code and then number the n bits, right to left, starting with 1 (not 0).
2. Each bit whose bit number is a power of 2 is a parity bit (positions 1,2,4,8,etc) and the others are data bits (positions 3,5,7,9,10,etc).
3. Each parity bit calculates the parity of some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately checks and skips:
Position 1: check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, etc. (1, 3, 5, 7, 9, ...)
Position 2: check 2 bits, skip 2 bits, check 2 bits, etc. (2,3,6,7,10,11,...)
Position 4: check 4 bits, skip 4 bits, check 4 bits, etc. (4,5,6,7,12,13,14,15,20,...)
...and so on.
4. Set each parity bit to 1 if the number of 1's in the positions it checks is odd, or to 0 if the number of 1's in the position it checks is even (even parity).

2.7.2 Hamming Codes

- Consider the example with words of length 4, with 3 check bits.
- Code words will be of length 7 (white cells are data and green cells are parity bits):



- The set of bits using parity bit 1: 1, 3, 5, 7 (check 1, skip 1, ...)
- The set of bits using parity bit 2: 2, 3, 6, 7 (check 2, skip 2, ...)
- The set of bits using parity bit 4: 4, 5, 6, 7 (check 4)



Notice that the parity bits (1, 2, 4) are each only in one set.
Bit position 7 is in all three sets.
Using this system, if a bit changes, it can be deduced which one changed.

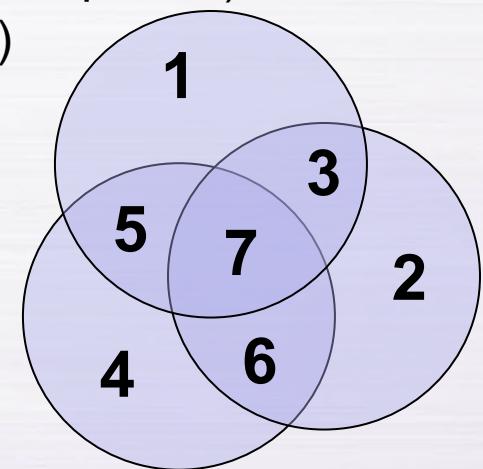
2.7.2 Hamming Codes

- Example: Determine the Hamming code word for 4-bit data 1101.
- Code words will be of length 7 (white cells are data and green cells are parity bits):

1	1	0		1		
7	6	5	4	3	2	1

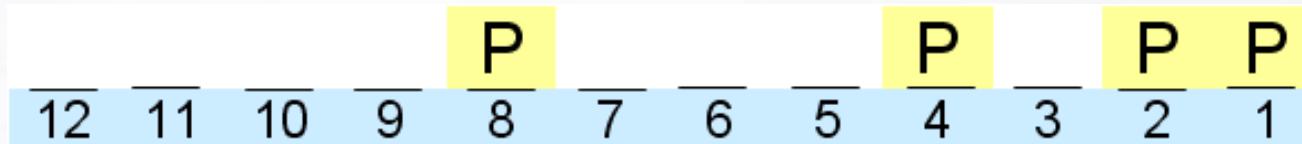
- The set of bits using parity bit 1: 1, 3, 5, 7 (check 1, skip 1, ...)
- The set of bits using parity bit 2: 2, 3, 6, 7 (check 2, skip 2, ...)
- The set of bits using parity bit 4: 4, 5, 6, 7 (check 4)

- Determine the parity bit values:
 - Parity bit 1: value is set to 0 (since 3=1, 5=0, 7=1)
 - Parity bit 2: value is set to 1 (since 3=1, 6=1, 7=1)
 - Parity bit 4: value is set to 0 (since 5=0, 6=1, 7=1)
- Hamming code word: 1100110
- How to determine where an error occurred?
 - Add the positions of parity bits that produce the error!



2.7.2 Hamming Codes

- What is the Hamming code of data word 1101 0110?
- Data word of length 8 requires 4 parity bits.
- Which will be the check (parity) bits?



- What is the full Hamming code?



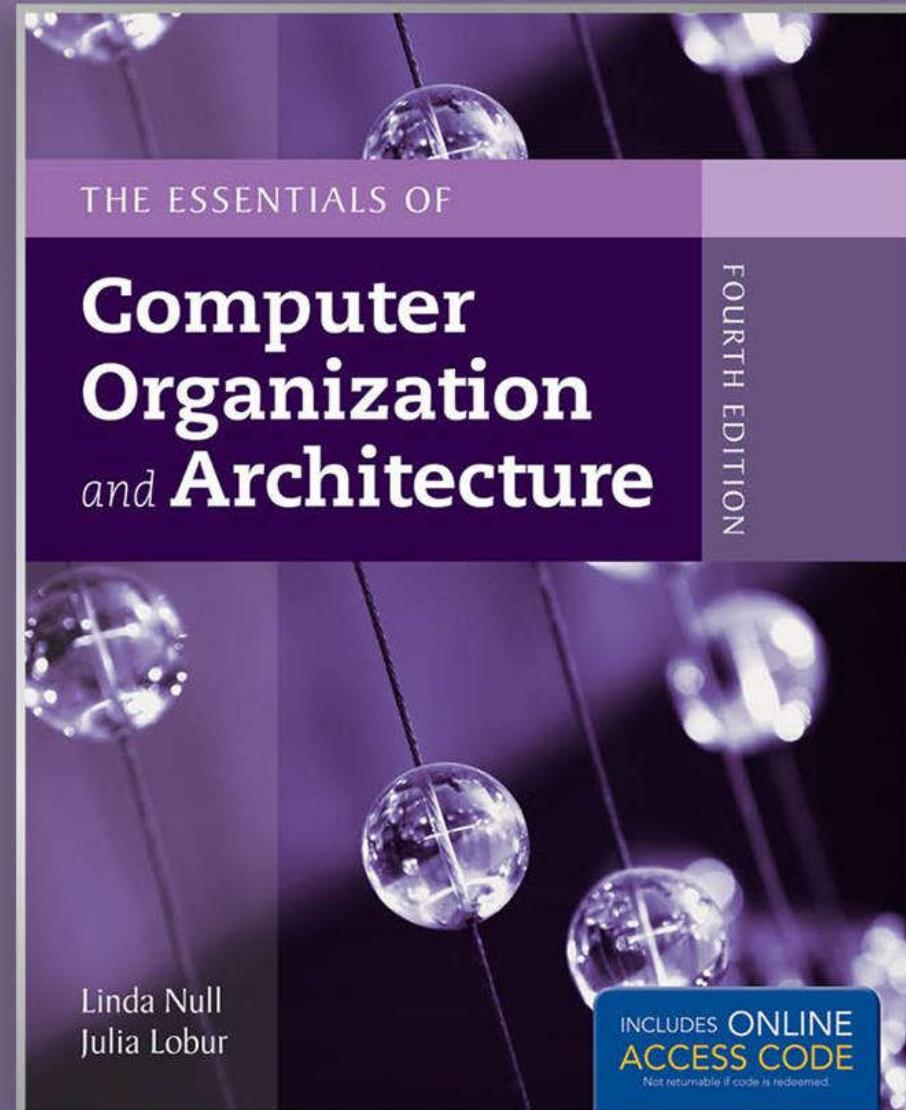
2.7.2 Hamming Codes

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

- There is an error in the code word above. Correct it.
 - Bit 1 checks 1, 3, 5, 7, 9, and 11. (1,0,0,0,1,1)
 - *This is incorrect as we have a total of 3 ones (which is not even parity).*
 - Bit 2 checks bits 2, 3, 6, 7, 10, and 11. (0,0,1,0,0,1)
 - The parity is correct.
 - Bit 4 checks bits 4, 5, 6, 7, and 12. (1,0,1,0,1)
 - *This parity is incorrect, as we 3 ones.*
 - Bit 8 checks bit 8, 9, 10, 11, and 12. (1,1,0,1,1)
 - This parity is correct.
 - The common elements in the two sets are 5 and 7, but 7 is part of bit 2 check, so cannot be the problem.
 - Therefore, the bit that changed was 5, so flip the bit in position 5.
 - Alternatively, simply add $1 + 4 = 5$, and flip bit 5.

Chapter 2

Data Representation in Computer Systems



Chapter 2 Objectives

- Understand the fundamentals of numerical data representation and manipulation in digital computers.
- Master the skill of converting between various radix systems.
- Understand how errors can occur in computations because of overflow and truncation.

Chapter 2 Objectives

- Understand the fundamental concepts of floating-point representation.
- Gain familiarity with the most popular character codes.
- Understand the concepts of error detecting and correcting codes.

2.1 Introduction

- A *bit* is the most basic unit of information in a computer.
 - It is a state of “on” or “off” in a digital circuit.
 - Sometimes these states are “high” or “low” voltage instead of “on” or “off..”
- A *byte* is a group of eight bits.
 - A byte is the smallest possible *addressable* unit of computer storage.
 - The term, “addressable,” means that a particular byte can be retrieved according to its location in memory.

2.1 Introduction

- A *word* is a contiguous group of bytes.
 - Words can be any number of bits or bytes.
 - Word sizes of 16, 32, or 64 bits are most common.
 - In a word-addressable system, a word is the smallest addressable unit of storage.
- A group of four bits is called a *nibble*.
 - Bytes, therefore, consist of two nibbles: a “high-order nibble,” and a “low-order” nibble.

2.2 Positional Numbering Systems

- Bytes store numbers using the position of each bit to represent a power of 2.
 - The binary system is also called the base-2 system.
 - Our decimal system is the base-10 system. It uses powers of 10 for each position in a number.
 - Any integer quantity can be represented exactly using any base (or *radix*).

2.2 Positional Numbering Systems

- The decimal number 947 in powers of 10 is:

$$9 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$$

- The decimal number 5836.47 in powers of 10 is:

$$\begin{aligned} & 5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 \\ & + 4 \times 10^{-1} + 7 \times 10^{-2} \end{aligned}$$

2.2 Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

$$\begin{aligned} & 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ = & 16 + 8 + 0 + 0 + 1 = 25 \end{aligned}$$

- When the radix of a number is something other than 10, the base is denoted by a subscript.
 - Sometimes, the subscript 10 is added for emphasis:

$$11001_2 = 25_{10}$$

2.3 Converting Between Bases

- Because binary numbers are the basis for all data representation in digital computer systems, it is important that you become proficient with this radix system.
- Your knowledge of the binary numbering system will enable you to understand the operation of all computer components as well as the design of instruction set architectures.

2.3 Converting Between Bases

- In an earlier slide, we said that every integer value can be represented exactly using any radix system.
- There are two methods for radix conversion: the subtraction method and the division remainder method.
- The subtraction method is more intuitive, but cumbersome. It does, however reinforce the ideas behind radix mathematics.

2.3 Converting Between Bases

- Suppose we want to convert the decimal number 190 to base 3.

- We know that $3^5 = 243$ so our result will be less than six digits wide. The largest power of 3 that we need is therefore $3^4 = 81$, and $81 \times 2 = 162$.
 - Write down the 2 and subtract 162 from 190, giving 28.

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \end{array} = 3^4 \times 2$$

2.3 Converting Between Bases

- **Converting 190 to base 3...**

- The next power of 3 is $3^3 = 27$. We'll need one of these, so we subtract 27 and write down the numeral 1 in our result.
- The next power of 3, $3^2 = 9$, is too large, but we have to assign a placeholder of zero and carry down the 1.

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \\ - 27 \\ \hline 1 \\ - 0 \\ \hline 1 \end{array} = 3^4 \times 2 \\ = 3^3 \times 1 \\ = 3^2 \times 0$$

2.3 Converting Between Bases

- **Converting 190 to base 3...**

- $3^1 = 3$ is again too large, so we assign a zero placeholder.
- The last power of 3, $3^0 = 1$, is our last choice, and it gives us a difference of zero.
- Our result, reading from top to bottom is:

$$190_{10} = 21001_3$$

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \\ - 27 \\ \hline 1 \\ - 0 \\ \hline 1 \\ - 0 \\ \hline 1 \\ - 1 \\ \hline 0 \end{array} = 3^4 \times 2 \\ = 3^3 \times 1 \\ = 3^2 \times 0 \\ = 3^1 \times 0 \\ = 3^0 \times 1$$

2.3 Converting Between Bases

- Another method of converting integers from decimal to some other radix uses division.
- This method is mechanical and easy.
- It employs the idea that successive division by a base is equivalent to successive subtraction by powers of the base.
- Let's use the division remainder method to again convert 190 in decimal to base 3.

2.3 Converting Between Bases

- **Converting 190 to base 3...**

- First we take the number that we wish to convert and divide it by the radix in which we want to express our result.
- In this case, 3 divides 190 63 times, with a remainder of 1.
- Record the quotient and the remainder.

$$\begin{array}{r} 3 \quad | \quad 190 \quad 1 \\ \hline 63 \end{array}$$

2.3 Converting Between Bases

- **Converting 190 to base 3...**

- 63 is evenly divisible by 3.
- Our remainder is zero, and the quotient is 21.

$$\begin{array}{r} 3 \overline{)190} & 1 \\ 3 \overline{)63} & 0 \\ & 21 \end{array}$$

2.3 Converting Between Bases

- **Converting 190 to base 3...**

- Continue in this way until the quotient is zero.
- In the final calculation, we note that 3 divides 2 zero times with a remainder of 2.
- Our result, reading from bottom to top is:

$$190_{10} = 21001_3$$

A handwritten long division diagram for converting 190 from base 10 to base 3. The divisor is 3. The dividend is 190. The quotient is 21001₃. The remainders are 1, 0, 0, 1, 2, and 0. The remainders are highlighted in yellow.

3	1	9	0	1
3	6	3	0	
3	2	1	0	
3	7			1
3	2			2
				0

2.3 Converting Between Bases

- Fractional values can be approximated in all base systems.
- Unlike integer values, fractions do not necessarily have exact representations under all radices.
- The quantity $\frac{1}{2}$ is exactly representable in the binary and decimal systems, but is not in the ternary (base 3) numbering system.

2.3 Converting Between Bases

- Fractional decimal values have nonzero digits to the right of the decimal point.
- Fractional values of other radix systems have nonzero digits to the right of the *radix point*.
- Numerals to the right of a radix point represent negative powers of the radix:

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$

$$\begin{aligned}0.11_2 &= 1 \times 2^{-1} + 1 \times 2^{-2} \\&= \frac{1}{2} + \frac{1}{4} \\&= 0.5 + 0.25 = 0.75\end{aligned}$$

2.3 Converting Between Bases

- As with whole-number conversions, you can use either of two methods: a subtraction method or an easy multiplication method.
- The subtraction method for fractions is identical to the subtraction method for whole numbers. Instead of subtracting positive powers of the target radix, we subtract negative powers of the radix.
- We always start with the largest value first, n^{-1} , where n is our radix, and work our way along using larger negative exponents.

2.3 Converting Between Bases

- The calculation to the right is an example of using the subtraction method to convert the decimal 0.8125 to binary.

- Our result, reading from top to bottom is:

$$0.8125_{10} = 0.1101_2$$

- Of course, this method works with any base, not just binary.

$$\begin{array}{r} 0.8125 \\ - 0.5000 \\ \hline 0.3125 \\ - 0.2500 \\ \hline 0.0625 \\ - 0 \\ \hline 0.0625 \\ - 0.0625 \\ \hline 0 \end{array} = 2^{-1} \times 1 \\ = 2^{-2} \times 1 \\ = 2^{-3} \times 0 \\ = 2^{-4} \times 1$$

2.3 Converting Between Bases

- **Using the multiplication method to convert the decimal 0.8125 to binary, we multiply by the radix 2.**
 - The first product carries into the units place.

$$\begin{array}{r} .8125 \\ \times 2 \\ \hline 1.6250 \end{array}$$

2.3 Converting Between Bases

- **Converting 0.8125 to binary . . .**

– Ignoring the value in the units place at each step, continue multiplying each fractional part by the radix.

$$\begin{array}{r} .8125 \\ \times 2 \\ \hline 1.6250 \end{array}$$

$$\begin{array}{r} .6250 \\ \times 2 \\ \hline 1.2500 \end{array}$$

$$\begin{array}{r} .2500 \\ \times 2 \\ \hline 0.5000 \end{array}$$

2.3 Converting Between Bases

- **Converting 0.8125 to binary . . .**

- You are finished when the product is zero, or until you have reached the desired number of binary places.
- Our result, reading from top to bottom is:

$$0.8125_{10} = 0.1101_2$$

- This method also works with any base. Just use the target radix as the multiplier.

$$\begin{array}{r} .8125 \\ \times 2 \\ \hline 1.6250 \\ .6250 \\ \times 2 \\ \hline 1.2500 \\ .2500 \\ \times 2 \\ \hline 0.5000 \\ .5000 \\ \times 2 \\ \hline 1.0000 \end{array}$$

2.3 Converting Between Bases

- The binary numbering system is the most important radix system for digital computers.
- However, it is difficult to read long strings of binary numbers -- and even a modestly-sized decimal number becomes a very long binary number.
 - For example: $11010100011011_2 = 13595_{10}$
- For compactness and ease of reading, binary values are usually expressed using the hexadecimal, or base-16, numbering system.

2.3 Converting Between Bases

- The hexadecimal numbering system uses the numerals 0 through 9 and the letters A through F.
 - The decimal number 12 is C₁₆.
 - The decimal number 26 is 1A₁₆.
- It is easy to convert between base 16 and base 2, because $16 = 2^4$.
- Thus, to convert from binary to hexadecimal, all we need to do is group the binary digits into groups of four.

A group of four binary digits is called a hextet

2.3 Converting Between Bases

- Using groups of hextets, the binary number 11010100011011_2 ($= 13595_{10}$) in hexadecimal is:

0011	0101	0001	1011
3	5	1	B

If the number of bits is not a multiple of 4, pad on the left with zeros.

- Octal (base 8) values are derived from binary by using groups of three bits ($8 = 2^3$):

011	010	100	011	011
3	2	4	3	3

Octal was very useful when computers used six-bit words.

2.4 Signed Integer Representation

- The conversions we have so far presented have involved only unsigned numbers.
- To represent signed integers, computer systems allocate the high-order bit to indicate the sign of a number.
 - The high-order bit is the leftmost bit. It is also called the most significant bit.
 - 0 is used to indicate a positive number; 1 indicates a negative number.
- The remaining bits contain the value of the number (but this can be interpreted different ways)

2.4 Signed Integer Representation

- There are three ways in which signed binary integers may be expressed:
 - Signed magnitude
 - One's complement
 - Two's complement
- In an 8-bit word, *signed magnitude* representation places the absolute value of the number in the 7 bits to the right of the sign bit.

2.4 Signed Integer Representation

- For example, in 8-bit signed magnitude representation:
 - +3 is: 00000011
 - 3 is: 10000011
- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
 - Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.

2.4 Signed Integer Representation

- Binary addition is as easy as it gets. You need to know only four rules:

$0 + 0 = 0$	$0 + 1 = 1$
$1 + 0 = 1$	$1 + 1 = 10$
- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.
 - We will describe these circuits in Chapter 3.

Let's see how the addition rules work with signed magnitude numbers . . .

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- First, convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits.

$$\begin{array}{r} 0 \quad 1001011 \\ 0 + 0101110 \\ \hline \end{array}$$

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Just as in decimal arithmetic, we find the sum starting with the rightmost bit and work left.

$$\begin{array}{r} 0 \quad 1001011 \\ 0 + 0101110 \\ \hline 1 \end{array}$$

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- In the second bit, we have a carry, so we note it above the third bit.

$$\begin{array}{r} & & 1 \\ 0 & 1001011 \\ 0 & + 0101110 \\ \hline & 01 \end{array}$$

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- The third and fourth bits also give us carries.

$$\begin{array}{r} & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & + & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ \hline & & & & & & 1 & 0 & 0 & 1 \end{array}$$

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Once we have worked our way through all eight bits, we are done.

$$\begin{array}{r} & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & + & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}$$

In this example, we were careful to pick two values whose sum would fit into seven bits. If that is not the case, we have a problem.

2.4 Signed Integer Representation

- Example:
 - Using signed magnitude binary arithmetic, find the sum of 107 and 46.
- We see that the carry from the seventh bit *overflows* and is discarded, giving us the erroneous result: $107 + 46 = 25$.

$$\begin{array}{r} & \textcircled{1} & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & + & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}$$

2.4 Signed Integer Representation

- The signs in signed magnitude representation work just like the signs in pencil and paper arithmetic.
 - Example: Using signed magnitude binary arithmetic, find the sum of - 46 and - 25.
- Because the signs are the same, all we do is add the numbers and supply the negative sign when we are done.

$$\begin{array}{r} & & 1 & 1 \\ & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ & 1 & + & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline & 1 & 1 & 0 & 0 & 1 & 1 & 1 \end{array}$$

2.4 Signed Integer Representation

- Mixed sign addition (or subtraction) is done the same way.
 - Example: Using signed magnitude binary arithmetic, find the sum of 46 and - 25.
- The sign of the result gets the sign of the number that is larger.
 - Note the “borrows” from the second and sixth bits.

$$\begin{array}{r} & \overset{0}{\cancel{2}} & \overset{0}{\cancel{2}} \\ 0 & \overset{0}{\cancel{1}}011\overset{1}{\cancel{1}}0 \\ 1 & + & 0011001 \\ \hline 0 & 0010101 \end{array}$$

2.4 Signed Integer Representation

- Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware.
- Another disadvantage of signed magnitude is that it allows two different representations for zero: positive zero and negative zero.
- For these reasons (among others) computers systems employ *complement systems* for numeric value representation.

2.4 Signed Integer Representation

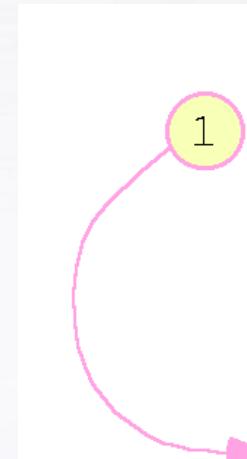
- In complement systems, negative values are represented by some difference between a number and its base.
- The *diminished radix complement* of a non-zero number N in base r with d digits is $(r^d - 1) - N$
- In the binary system, this gives us *one's complement*. It amounts to little more than flipping the bits of a binary number.

2.4 Signed Integer Representation

- For example, using 8-bit one's complement representation:
 - + 3 is: 00000011
 - 3 is: 11111100
- In one's complement representation, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
- Complement systems are useful because they eliminate the need for subtraction. The difference of two values is found by adding the minuend to the complement of the subtrahend.

2.4 Signed Integer Representation

- With one's complement addition, the carry bit is “carried around” and added to the sum.
 - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19


$$\begin{array}{r} \text{1 } \text{1} \\ 00110000 \\ 11101100 \\ \hline 00011100 \\ + \quad 1 \\ \hline 00011101 \end{array}$$

We note that 19 in binary is
so -19 in one's complement is:

00010011,
11101100.

2.4 Signed Integer Representation

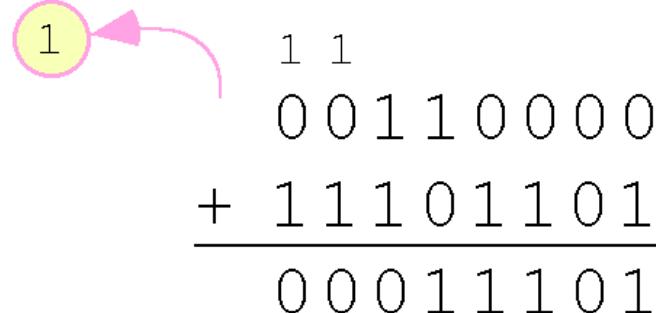
- Although the “end carry around” adds some complexity, one’s complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.
- Two’s complement solves this problem.
- Two’s complement is the radix complement of the binary numbering system; the *radix complement* of a non-zero number N in base r with d digits is $r^d - N$.

2.4 Signed Integer Representation

- To express a value in two's complement representation:
 - If the number is positive, just convert it to binary and you're done.
 - If the number is negative, find the one's complement of the number and then add 1.
- Example:
 - In 8-bit binary, 3 is:
00000011
 - -3 using one's complement representation is:
11111100
 - Adding 1 gives us -3 in two's complement form:
11111101.

2.4 Signed Integer Representation

- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.
 - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19.


$$\begin{array}{r} & \textcircled{1} \\ & \swarrow \\ 1 & 1 \\ 0 & 0 1 1 0 0 0 0 \\ + & 1 1 1 0 1 1 0 1 \\ \hline 0 & 0 0 1 1 1 0 1 \end{array}$$

We note that 19 in binary is: 00010011,
so -19 using one's complement is: 11101100,
and -19 using two's complement is: 11101101.

2.4 Signed Integer Representation

- Excess-M representation (also called offset binary representation) is another way for unsigned binary values to represent signed integers.
 - Excess-M representation is intuitive because the binary string with all 0s represents the smallest number, whereas the binary string with all 1s represents the largest value.
- An unsigned binary integer M (called the *bias*) represents the value 0, whereas all zeroes in the bit pattern represents the integer $-M$.
- The integer is interpreted as positive or negative depending on where it falls in the range.

2.4 Signed Integer Representation

- If n bits are used for the binary representation, we select the bias in such a manner that we split the range equally.
- Typically we choose a bias of $2^{n-1} - 1$.
 - For example, if we were using 4-bit representation, the bias should be $2^{4-1} - 1 = 7$.
- Just as with signed magnitude, one's complement, and two's complement, there is a specific range of values that can be expressed in n bits.

2.4 Signed Integer Representation

- The unsigned binary value for a signed integer using excess- M representation is determined simply by adding M to that integer.
 - For example, assuming that we are using excess-7 representation, the integer 0_{10} is represented as $0 + 7 = 7_{10} = 0111_2$.
 - The integer 3_{10} is represented as $3 + 7 = 10_{10} = 1010_2$.
 - The integer -7 is represented as $-7 + 7 = 0_{10} = 0000_2$.
 - To find the decimal value of the excess-7 binary number 1111_2 subtract 7: $1111_2 = 15_{10}$ and $15 - 7 = 8$; thus 1111_2 , in excess-7 is $+8_{10}$.

2.4 Signed Integer Representation

- Lets compare our representations:

Decimal	Binary (for absolute value)	Signed Magnitude	One's Complement
2	00000010	00000010	00000010
-2	00000010	10000010	11111101
100	01100100	01100100	01100100
-100	01100100	11100100	10011011

Decimal	Binary (for absolute value)	Two's Complement	Excess-127
2	00000010	00000010	10000001
-2	00000010	11111110	01111101
100	01100100	01100100	11100011
-100	01100100	10011100	00011011

2.4 Signed Integer Representation

- When we use any finite number of bits to represent a number, we always run the risk of the result of our calculations becoming too large or too small to be stored in the computer.
- While we can't always prevent overflow, we can always *detect* overflow.
- In complement arithmetic, an overflow condition is easy to detect.

2.4 Signed Integer Representation

- Example:
 - Using two's complement binary arithmetic, find the sum of 107 and 46.
- We see that the nonzero carry from the seventh bit *overflows* into the sign bit, giving us the erroneous result: $107 + 46 = -103$.

$$\begin{array}{r} \textcolor{yellow}{1}1111 \\ 01101011 \\ + 00101110 \\ \hline 10011001 \end{array}$$

But overflow into the sign bit does not always mean that we have an error.

2.4 Signed Integer Representation

- Example:
 - Using two's complement binary arithmetic, find the sum of 23 and -9.
 - We see that there is carry into the sign bit and carry out. The final result is correct: $23 + (-9) = 14$.

$$\begin{array}{r} 1 \leftarrow 1\ 1\ 1\ 1\ 1\ 1 \\ 00010111 \\ + 11110111 \\ \hline 00001110 \end{array}$$

Rule for detecting signed two's complement overflow: When the “carry in” and the “carry out” of the sign bit differ, overflow has occurred. If the carry into the sign bit equals the carry out of the sign bit, no overflow has occurred.

2.4 Signed Integer Representation

- Signed and unsigned numbers are both useful.
 - For example, memory addresses are always unsigned.
- Using the same number of bits, unsigned integers can express twice as many “positive” values as signed numbers.
- Trouble arises if an unsigned value “wraps around.”
 - In four bits: $1111 + 1 = 0000$.
- Good programmers stay alert for this kind of problem.

2.4 Signed Integer Representation

- Research into finding better arithmetic algorithms has continued for over 50 years.
- One of the many interesting products of this work is Booth's algorithm.
- In most cases, Booth's algorithm carries out multiplication faster and more accurately than pencil-and-paper methods.
- The general idea is to replace arithmetic operations with bit shifting to the extent possible.

2.4 Signed Integer Representation

In Booth's algorithm:

- If the current multiplier bit is 1 and the preceding bit was 0, subtract the multiplicand from the product
- If the current multiplier bit is 0 and the preceding bit was 1, we add the multiplicand to the product
- If we have a 00 or 11 pair, we simply shift.
- Assume a mythical “0” starting bit
- Shift after each step

$$\begin{array}{r} 0011 \quad (\text{multiplicand}) \\ \times 0110 \quad (\text{multiplier}) \\ \hline + 0000 \quad (\text{shift}) \\ - 0011 \quad (\text{subtract}) \\ + 0000 \quad (\text{shift}) \\ + 0011 \quad (\text{add}) \\ \hline 00010010 \end{array}$$

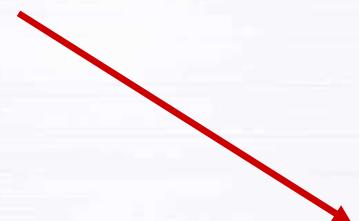
We see that $3 \times 6 = 18!$

2.4 Signed Integer Representation

- Here is a larger example.

$$\begin{array}{r} 00110101 \\ \times \quad 01111110 \\ \hline + \quad 0000000000000000 \\ + \quad 111111111001011 \\ + \quad 0000000000000000 \\ + \quad 000110101 \\ \hline 10001101000010110 \end{array}$$

Ignore all bits over $2n$.



$$53 \times 126 = 6678!$$

2.4 Signed Integer Representation

- Overflow and carry are tricky ideas.
- Signed number overflow means nothing in the context of unsigned numbers, which set a carry flag instead of an overflow flag.
- If a carry out of the leftmost bit occurs with an unsigned number, overflow has occurred.
- Carry and overflow occur independently of each other.

The table on the next slide summarizes these ideas.

2.4 Signed Integer Representation

Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010	0110	No	No	Yes
0100 + 0110	1010	No	Yes	No
1100 + 1110	1010	Yes	No	Yes
1100 + 1010	0110	Yes	Yes	No

2.4 Signed Integer Representation

- We can do binary multiplication and division by 2 very easily using an *arithmetic shift* operation
- A *left arithmetic shift* inserts a 0 in for the rightmost bit and shifts everything else left one bit; in effect, it multiplies by 2
- A *right arithmetic shift* shifts everything one bit to the right, but copies the sign bit; it divides by 2
- Let's look at some examples.

2.4 Signed Integer Representation

Example:

Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

00001011 (+11)

We shift left one place, resulting in:

00010110 (+22)

The sign bit has not changed, so the value is valid.

To multiply 11 by 4, we simply perform a left shift twice.

2.4 Signed Integer Representation

Example:

Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

00001100 (+12)

We shift left one place, resulting in:

00000110 (+6)

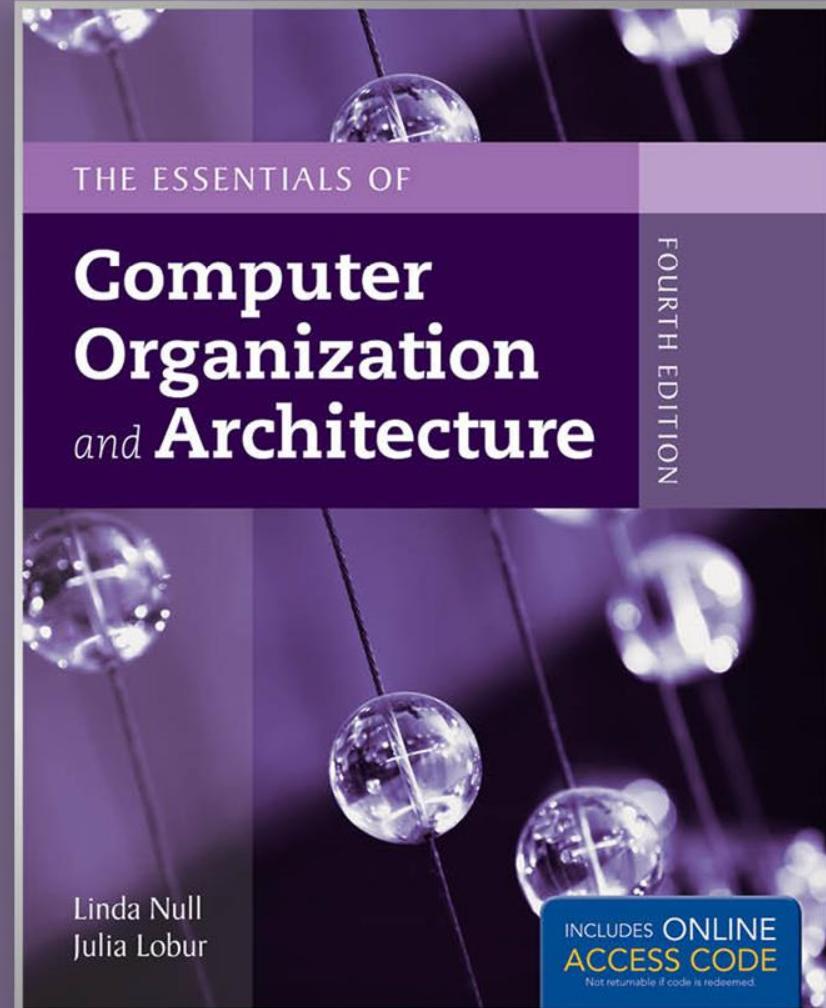
(Remember, we carry the sign bit to the left as we shift.)

To divide 12 by 4, we right shift twice.

Chapter 3

Special Section

Focus on Karnaugh Maps



3A.1 Introduction

- Simplification of Boolean functions leads to simpler (and usually faster) digital circuits.
- Simplifying Boolean functions using identities is time-consuming and error-prone.
 - No guarantee that an expression is as simple as possible.
- In todays lecture we will cover a systematic method for reducing Boolean expressions.

3A.1 Introduction

- In 1953, Maurice Karnaugh was a telecommunications engineer at Bell Labs.
- While exploring the new field of digital logic and its application to the design of telephone circuits, he invented a graphical way of visualizing and then simplifying Boolean expressions.
- This graphical representation, now known as a Karnaugh map, or Kmap, is named in his honor.



3A.2 Description of Kmaps and Terminology

- A Kmap is a matrix consisting of rows and columns that represent the output values of a Boolean function.
- The output values placed in each cell are derived from the minterms of a Boolean function.
- A *minterm* is a **product** term that contains all of the function's variables exactly once, either complemented or not complemented.

3A.2 Description of Kmaps and Terminology

- For example, the minterms for a function having the inputs x and y are xy , $x'y$, xy' , and $x'y'$.
- Consider the Boolean function, $F(x, y) = xy + xy'$
- Its minterms are:

Minterm	X	Y
$x'y'$	0	0
$x'y$	0	1
xy'	1	0
xy	1	1

3A.2 Description of Kmaps and Terminology

- Similarly, a function having three inputs, has the *minterms* that are shown in this table.

Minterm	X	Y	Z
$X'Y'Z'$	0	0	0
$X'Y'Z$	0	0	1
$X'Y Z'$	0	1	0
$X'Y Z$	0	1	1
$X Y'Z'$	1	0	0
$X Y'Z$	1	0	1
$X Y Z'$	1	1	0
$X Y Z$	1	1	1

3A.2 Description of Kmaps and Terminology

- A Kmap has a cell for each minterm.
- This means that it has a cell for each line for the truth table of a function.
- The truth table for the function $F(x,y) = xy$ is shown at the right along with its corresponding Kmap.

		$F(x, y) = xy$
x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

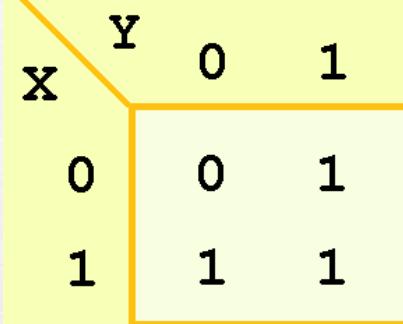
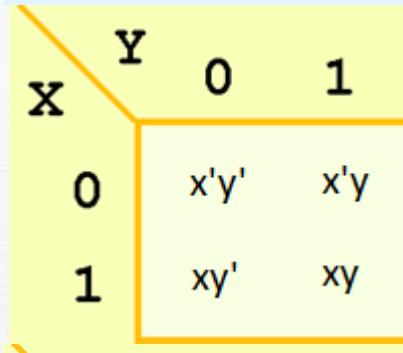
x	y	0	1
0	0	0	0
1	0	0	1

3A.2 Description of Kmaps and Terminology

- As another example, we give the truth table and KMap for the function, $F(x,y) = x + y$ at the right.
- This function is equivalent to the OR of all of the minterms that have a value of 1. Thus:

$$F(x, y) = x + y = x'y' + xy' + xy$$

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1



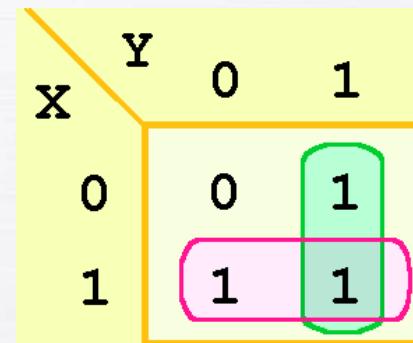
3A.3 Kmap Simplification for Two Variables

- Of course, the minterm function that we derived from our Kmap was not in simplest terms.
 - That's what we started with in this example.
- We can, however, reduce our complicated expression to its simplest terms by finding adjacent 1s in the Kmap that can be collected into groups that are **powers of two**.
- In our example, we have two such groups.
 - Can you find them?

x \ y	0	1
0	0	1
1	1	1

3A.3 Kmap Simplification for Two Variables

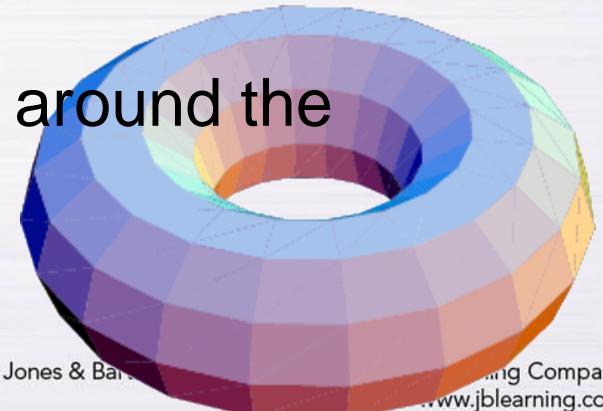
- The best way of selecting two groups of 1s from our simple Kmap is shown below.
- We see that both groups are powers of two and that the groups overlap.
- The next slide gives guidance for selecting Kmap groups.



3A.3 Kmap Simplification for Two Variables

The rules of Kmap simplification are:

- Groupings can contain **only** 1s; **no** 0s.
- Groups can be formed only at right angles; diagonal groups are not allowed.
- The number of 1s in a group must be a power of 2 – even if it contains a single 1.
- The groups must be **made as large as possible**.
- Groups can **overlap** and **wrap** around the sides of the Kmap.
- All 1's must be covered.



3A.3 Kmap Simplification for Three Variables

- A Kmap for three variables is constructed as shown in the diagram below.
- We have placed each minterm in the cell that will hold its value.
 - Notice that the values for the yz combination at the top of the matrix form a pattern that is not a normal binary sequence.

		yz	00	01	11	10
		x	00	01	11	10
		0	$x'y'z'$	$x'y'z$	$x'y z$	$x'y z'$
		1	$x'y'z'$	$x'y'z$	$x'y z$	$x'y z'$

3A.3 Kmap Simplification for Three Variables

- Thus, the first row of the Kmap contains all minterms where x has a value of zero.
- The first column contains all minterms where y and z both have a value of zero.

		yz	00	01	11	10
		x	00	01	11	10
0		$x'y'z'$	$x'y'z$	xyz	$x'y z'$	$x'y z'$
1		$xy'z'$	$x y'z$	$xy z$	$x y z'$	$x y z'$

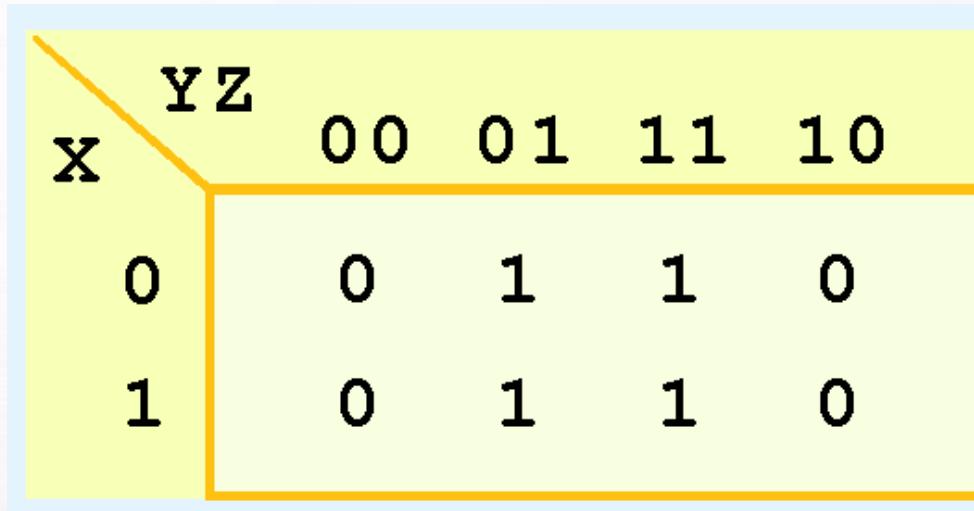
3A.3 Kmap Simplification for Three Variables

- Consider the function:

$$F(X, Y, Z) = X' Y' Z + X' Y Z + X Y' Z + X Y Z$$

- Its Kmap is given below.

- What is the largest group of 1s that is a power of 2?



3A.3 Kmap Simplification for Three Variables

- This grouping tells us that changes in the variables x and y have no influence upon the value of the function: They are irrelevant.
- This means that the function,
$$F(X, Y, Z) = X' Y' Z + X' Y Z + X Y' Z + X Y Z$$
reduces to $F(x) = z$.

You could verify
this reduction
with identities or
a truth table.

		Y	Z	00	01	11	10
		x	0	0	1	1	0
x	0	0	0	1	1	0	0
	1	0	0	1	1	0	0

3A.3 Kmap Simplification for Three Variables

- Systematic way to tell us that x and y aren't needed.
 - x y z
 - 0 0 1
 - 1 0 1
 - 0 1 1
 - 1 1 1
 - If Z=1 then the entry is 1 regardless of (x,y)

3A.3 Kmap Simplification for Three Variables

- Now for a more complicated Kmap. Consider the function:

$$F(X, Y, Z) = X'Y'Z' + X'Y'Z + X'YZ + X'YZ' + XY'Z' + XYZ'$$

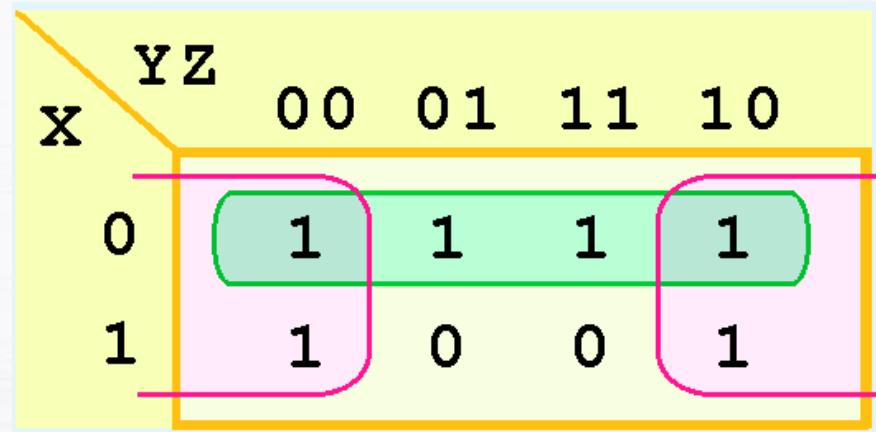
- Its Kmap is shown below. There are (only) two groupings of 1s.
 - Can you find them?

		YZ	00	01	11	10
		X	00	01	11	10
X	0	1	1	1	1	
	1	1	0	0	1	

3A.3 Kmap Simplification for Three Variables

- In this Kmap, we see an example of a group that wraps around the sides of a Kmap.
- This group tells us that the values of x and y are not relevant to the term of the function that is encompassed by the group.
 - What does this tell us about this term of the function?

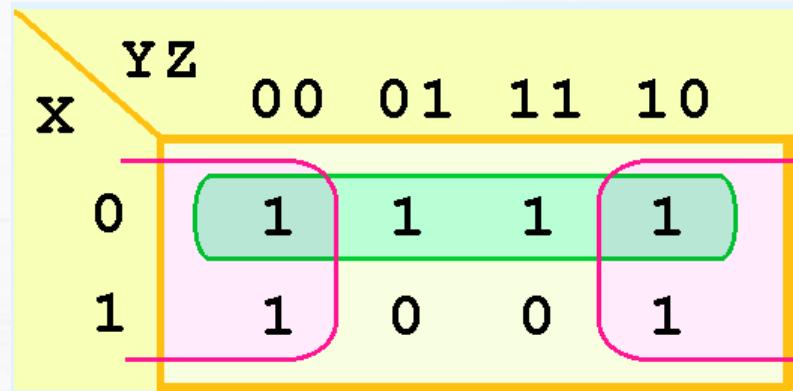
What about the green group in the top row?



3A.3 Kmap Simplification for Three Variables

- The green group in the top row tells us that only the value of x is significant in that group.
- We see that it is complemented in that row, so the other term of the reduced function is \mathbf{x}' .
- Our reduced function is $\mathbf{F(x, Y, Z)} = \mathbf{x'} + \mathbf{z'}$

Recall that we had
six minterms in our
original function!



3A.3 Kmap Simplification for Four Variables

- Our model can be extended to accommodate the 16 minterms that are produced by a four-input function.
- This is the format for a 16-minterm Kmap:

		yz	00	01	11	10
		wx	00	01	11	10
00	00	$w' x' y' z'$	$w' x' y' z$	$w' x' y z$	$w' x' y z'$	
	01	$w' x y' z'$	$w' x y' z$	$w' x y z$	$w' x y z'$	
	11	$w x y' z'$	$w x y' z$	$w x y z$	$w x y z'$	
	10	$w x' y' z'$	$w x' y' z$	$w x' y z$	$w x' y z'$	

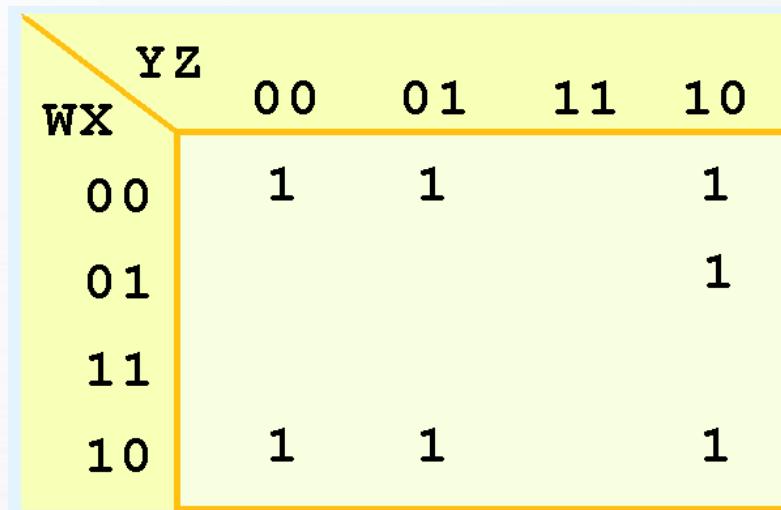
3A.3 Kmap Simplification for Four Variables

- We have populated the Kmap shown below with the nonzero minterms from the function:

$$\begin{aligned}F(W, X, Y, Z) = & W' X' Y' Z' + W' X' Y' Z + W' X' Y Z' \\& + W' X Y Z' + W X' Y' Z' + W X' Y' Z + W X' Y Z'\end{aligned}$$

- Can you identify (only) three groups in this Kmap?

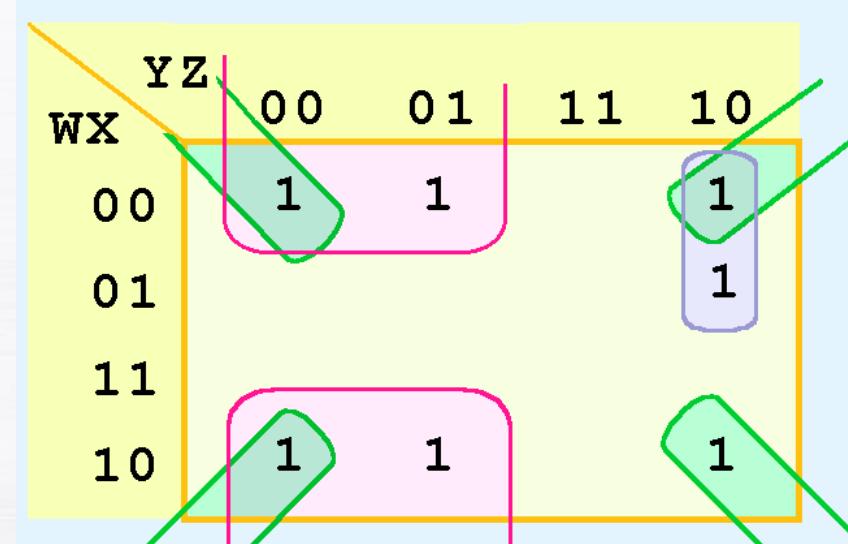
Recall that
groups can
overlap.



3A.3 Kmap Simplification for Four Variables

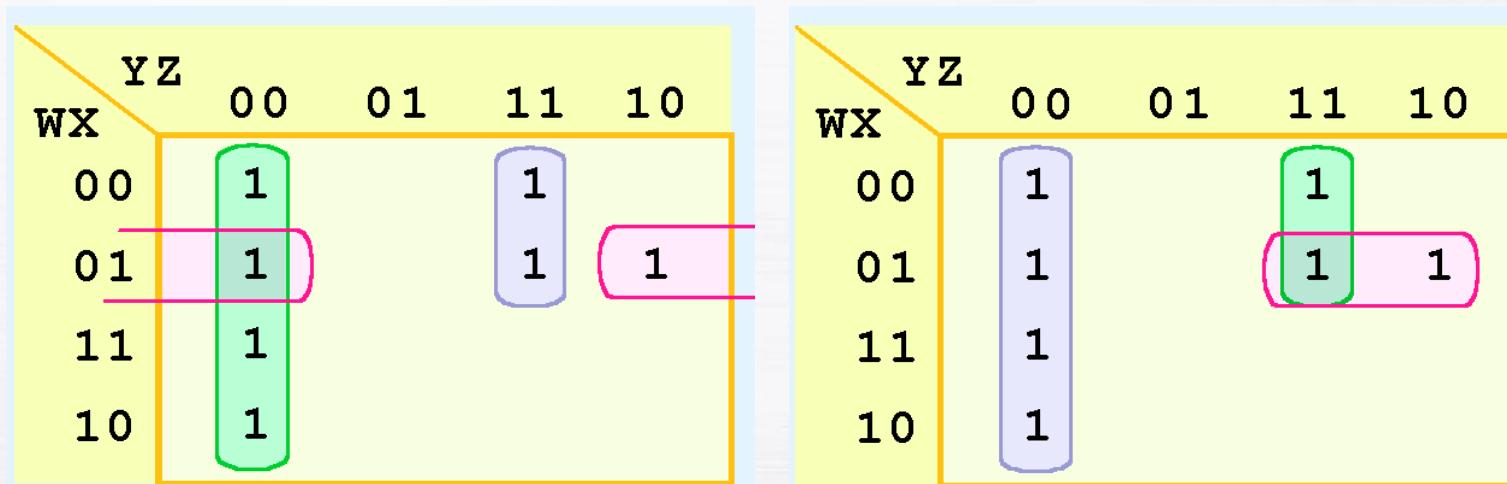
- Our three groups consist of:
 - A purple group entirely within the Kmap at the right.
 - A pink group that wraps the top and bottom.
 - A green group that spans the corners.
- Thus we have three terms in our final function:

$$F(W, X, Y, Z) = X' Y' + X' Z' + W' Y Z'$$



3A.3 Kmap Simplification for Four Variables

- It **is** possible to have a choice as to how to pick groups within a Kmap, while keeping the groups as large as possible.
- The (different) functions that result from the groupings below are logically equivalent.



3A.6 Don't Care Conditions

- Real circuits don't always need to have an output defined for every possible input.
 - For example, some calculator displays consist of 7-segment LEDs. These LEDs can display $2^7 - 1$ patterns, but only ten of them are useful.
- If a circuit is designed so that a particular set of inputs can never happen, we call this set of inputs a *don't care* condition.
- They are very helpful to us in Kmap circuit simplification.

3A.6 Don't Care Conditions

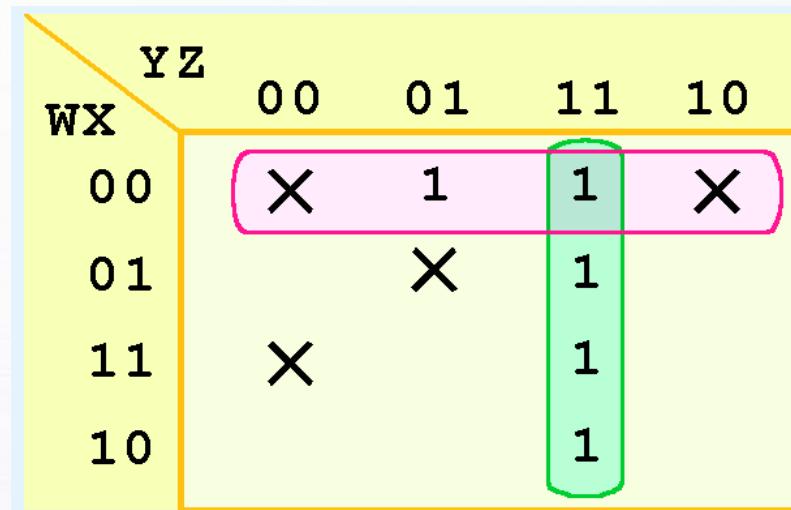
- In a Kmap, a don't care condition is identified by an X in the cell of the minterm(s) for the don't care inputs, as shown below.
- In performing the simplification, we are free to include or ignore the X 's when creating our groups.

w\nx	Y\nz	00	01	11	10
00	X	1	1	X	
01		X	1		
11	X		1		
10				1	

3A.6 Don't Care Conditions

- In one grouping in the Kmap below, we have the function:

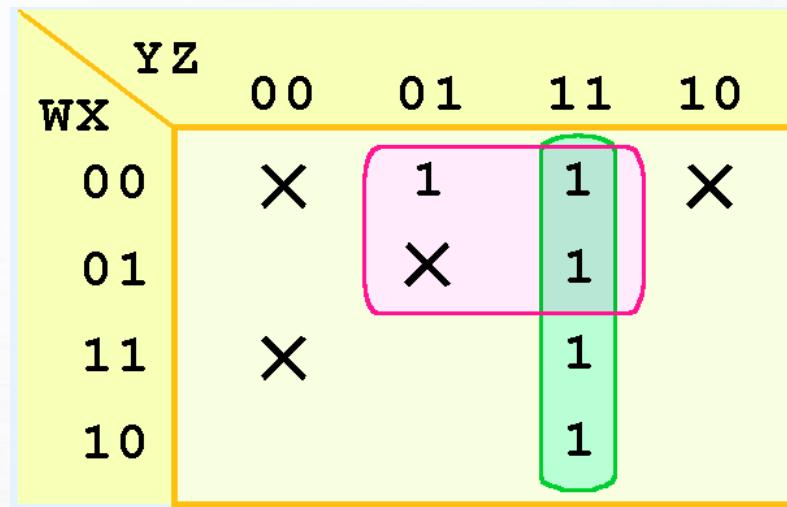
$$F(W, X, Y, Z) = W' X' + YZ$$



3A.6 Don't Care Conditions

- A different grouping gives us the function:

$$F(W, X, Y, Z) = W' Z + YZ$$



3A.6 Don't Care Conditions

- The truth table of:

$$F(W, X, Y, Z) = W' Y' + YZ$$

differs from the truth table of:

$$F(W, X, Y, Z) = W' Z + YZ$$

- However, the values for which they differ, are the inputs for which we have don't care conditions.

W\X	YZ	00	01	11	10
00	X	1	1	X	
01		X	1		
11	X		1		
10			1		

W\X	YZ	00	01	11	10
00	X	X	1	1	X
01		X	X	1	
11	X		X	1	
10			1	1	

3A Conclusion

- Kmaps provide an easy graphical method of simplifying Boolean expressions.
- A Kmap is a matrix consisting of the outputs of the minterms of a Boolean function.
- In this section, we have discussed 2- 3- and 4-input Kmaps. This method can be extended to any number of inputs through the use of multiple tables.
 - For 5 and 6 inputs it is basically a cube like structure viewed using 2 slices for 5 inputs and 4 slices for 6 inputs. You build groups of 1's that potentially go through layers

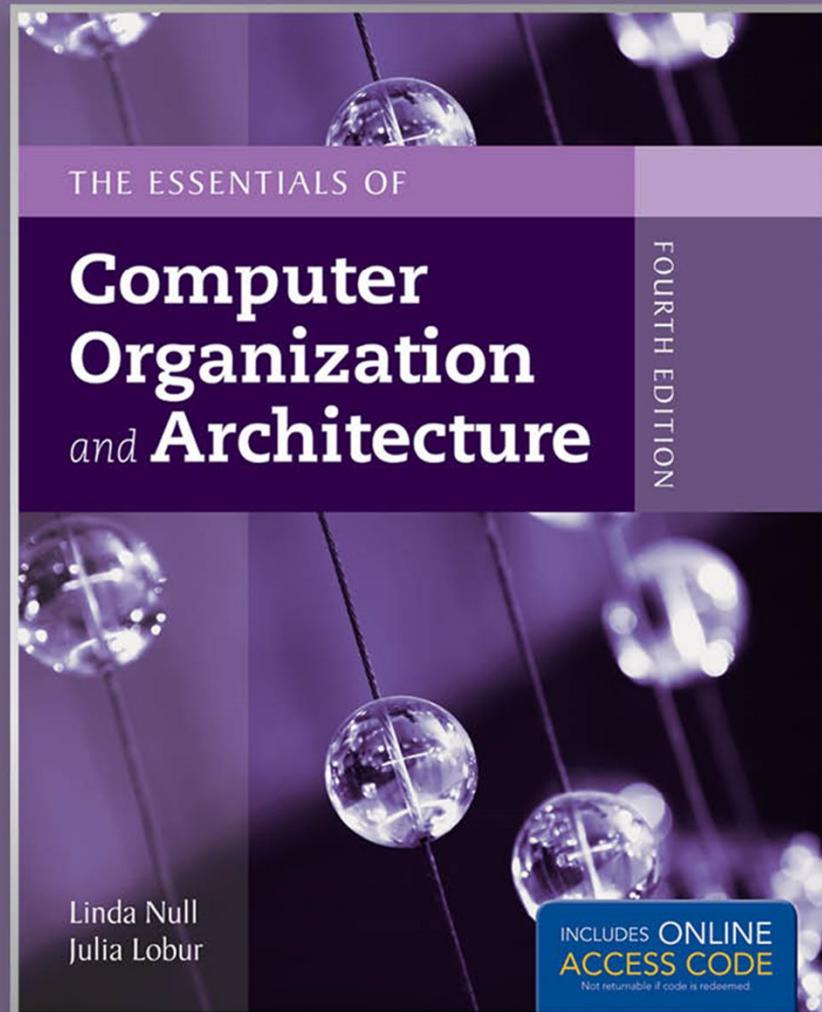
3A Conclusion

Recapping the rules of Kmap simplification:

- Groupings can contain only 1s; no 0s.
- Groups can be formed only at right angles; diagonal groups are not allowed.
- The number of 1s in a group must be a power of 2 – even if it contains a single 1.
- The groups must be made as large as possible.
- Groups can overlap and wrap around the sides of the Kmap.
- Use don't care conditions when you can.

Chapter 3

Boolean Algebra and Digital Logic



Chapter 3 Objectives

- Understand the relationship between Boolean logic and digital computer circuits.
- Learn how to design simple logic circuits.
- Understand how digital circuits work together to form complex computer systems.

3.1 Introduction

- In the latter part of the nineteenth century, George Boole incensed philosophers and mathematicians alike when he suggested that logical thought could be represented through mathematical equations.
 - *How dare anyone suggest that human thought could be encapsulated and manipulated like an algebraic formula?*
- Computers, as we know them today, are implementations of Boole's *Laws of Thought*.
 - John Atanasoff and Claude Shannon were among the first to see this connection.

3.1 Introduction

- In the middle of the twentieth century, computers were commonly known as “thinking machines” and “electronic brains.”
 - Many people were fearful of them.
- Nowadays, we rarely ponder the relationship between electronic digital computers and human logic. Computers are accepted as part of our lives.
 - Many people, however, are still fearful of them.
- In this chapter, you will learn the simplicity that constitutes the essence of the machine.

3.2 Boolean Algebra

- Boolean algebra is a mathematical system for the manipulation of variables that can have one of two values.
 - In formal logic, these values are “true” and “false.”
 - In digital systems, these values are “on” and “off,” 1 and 0, or “high” and “low.”
- Boolean expressions are created by performing operations on Boolean variables.
 - Common Boolean operators include AND, OR, and NOT.

3.2 Boolean Algebra

- A Boolean operator can be completely described using a truth table.
- The truth table for the Boolean operators AND and OR are shown at the right.
- The AND operator is also known as a Boolean product. The OR operator is the Boolean sum.

X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

3.2 Boolean Algebra

- The truth table for the Boolean NOT operator is shown at the right.
- The NOT operation is most often designated by a prime mark (x'). It is sometimes indicated by an overbar (\bar{x}) or an “elbow” ($\neg x$).

NOT x	
x	x'
0	1
1	0

3.2 Boolean Algebra

- A Boolean function has:
 - At least one Boolean variable,
 - At least one Boolean operator, and
 - At least one input from the set {0,1}.
- It produces an output that is also a member of the set {0,1}.

Now you know why the binary numbering system is so handy in digital systems.

3.2 Boolean Algebra

- The truth table for the Boolean function:

$$F(x, y, z) = xz' + y$$

is shown at the right.

- To make evaluation of the Boolean function easier, the truth table contains extra (shaded) columns to hold evaluations of subparts of the function.

$$F(x, y, z) = xz' + y$$

x	y	z	z'	xz'	$xz' + y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

3.2 Boolean Algebra

- As with common arithmetic, Boolean operations have rules of precedence.
- The NOT operator has highest priority, followed by AND and then OR.
- This is how we chose the (shaded) function subparts in our table.

$$F(x, y, z) = xz' + y$$

x	y	z	z'	xz'	$xz' + y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

3.2 Boolean Algebra

- Digital computers contain circuits that implement Boolean functions.
- The simpler that we can make a Boolean function, the smaller the circuit that will result.
 - Simpler circuits are cheaper to build, consume less power, and run faster than complex circuits.
- With this in mind, we always want to reduce our Boolean functions to their simplest form.
- There are a number of Boolean identities that help us to do this.

3.2 Boolean Algebra

- Most Boolean identities have an AND (product) form as well as an OR (sum) form. We give our identities using both forms. Our first group is rather intuitive:

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$xx' = 0$	$x + x' = 1$

3.2 Boolean Algebra

- Our second group of Boolean identities should be familiar to you from your study of algebra:

Identity Name	AND Form	OR Form
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x + (y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$

3.2 Boolean Algebra

- Our last group of Boolean identities are perhaps the most useful.
- If you have studied set theory or formal logic, these laws are also familiar to you.

Identity Name	AND Form	OR Form
Absorption Law	$x(x+y) = x$	$x + xy = x$
DeMorgan's Law	$(xy)' = x' + y'$	$(x+y)' = x'y'$
Double Complement Law		$(x)'' = x$

3.2 Boolean Algebra

- We can use Boolean identities to simplify:

$$F(x, y, z) = xy + x'z + yz$$

$$\begin{aligned} F(x, y, z) &= xy + x'z + yz \\ &= xy + x'z + yz(1) && \text{(Identity)} \\ &= xy + x'z + yz(x + x') && \text{(Inverse)} \\ &= xy + x'z + (yz)x + (yz)x' && \text{(Distributive)} \\ &= xy + x'z + x(yz) + x'(zy) && \text{(Commutative)} \\ &= xy + x'z + (xy)z + (x'z)y && \text{(Associative twice)} \\ &= xy + (xy)z + x'z + (x'z)y && \text{(Commutative)} \\ &= xy(1 + z) + x'z(1 + y) && \text{(Distributive)} \\ &= xy(1) + x'z(1) && \text{(Null)} \\ &= xy + x'z && \text{(Identity)} \end{aligned}$$

3.2 Boolean Algebra

- Sometimes it is more economical to build a circuit using the complement of a function (and complementing its result) than it is to implement the function directly.
- DeMorgan's law provides an easy way of finding the complement of a Boolean function.
- Recall DeMorgan's law states:

$$(xy)' = x' + y' \quad \text{and} \quad (x + y)' = x'y'$$

3.2 Boolean Algebra

- DeMorgan's law can be extended to any number of variables.
- Replace each variable by its complement and change all ANDs to ORs and all ORs to ANDs.
- Thus, we find the the complement of:

$$F(x, y, z) = (xy) + (x'y) + (xz')$$

is:

$$\begin{aligned} F' (x, y, z) &= ((xy) + (x'y) + (xz'))' \\ &= (xy)' (x'y)' (xz')' \\ &= (x' + y') (x + y') (x' + z) \end{aligned}$$

3.2 Boolean Algebra

- Through our exercises in simplifying Boolean expressions, we see that there are numerous ways of stating the same Boolean expression.
 - These “synonymous” forms are *logically equivalent*.
 - Logically equivalent expressions have identical truth tables.
- In order to eliminate as much confusion as possible, designers express Boolean functions in *standardized* or *canonical* form.

3.2 Boolean Algebra

- There are two canonical forms for Boolean expressions: sum-of-products and product-of-sums.
 - Recall the Boolean product is the AND operation and the Boolean sum is the OR operation.
- In the sum-of-products form, ANDed variables are ORed together.
 - For example: $F(x, y, z) = xy + xz + yz$
- In the product-of-sums form, ORed variables are ANDed together:
 - For example: $F(x, y, z) = (x+y)(x+z)(y+z)$

3.2 Boolean Algebra

- It is easy to convert a function to sum-of-products form using its truth table.
- We are interested in the values of the variables that make the function true (=1).
- Using the truth table, we list the values of the variables that result in a true function value.
- Each group of variables is then ORed together.

$$F(x, y, z) = xz' + y$$

x	y	z	$xz' + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

3.2 Boolean Algebra

- The sum-of-products form for our function is:

$$\begin{aligned}F(x, y, z) = & (x'y'z') + (x'yz) \\& + (xy'z') + (xyz') \\& + (xyz)\end{aligned}$$

We note that this function is not in simplest terms. Our aim is only to rewrite our function in canonical sum-of-products form.

$$F(x, y, z) = xz' + y$$

x	y	z	$xz' + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

3.3 Logic Gates

- We have looked at Boolean functions in abstract terms.
- In this section, we see that Boolean functions are implemented in digital computer circuits called gates.
- A gate is an electronic device that produces a result based on two or more input values.
 - In reality, gates consist of one to six transistors, but digital designers think of them as a single unit.
 - Integrated circuits contain collections of gates suited to a particular purpose.

3.3 Logic Gates

- The three simplest gates are the AND, OR, and NOT gates.



X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT X

X	X'
0	1
1	0

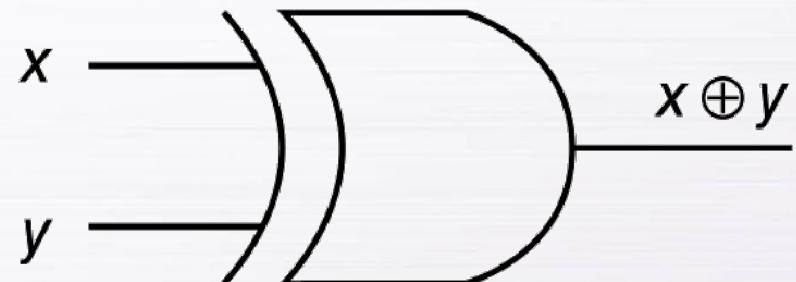
- They correspond directly to their respective Boolean operations, as you can see by their truth tables.

3.3 Logic Gates

- Another very useful gate is the exclusive OR (XOR) gate.
- The output of the XOR operation is true only when the values of the inputs differ.

$x \text{ XOR } y$

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0



Note the special symbol \oplus for the XOR operation.

3.3 Logic Gates

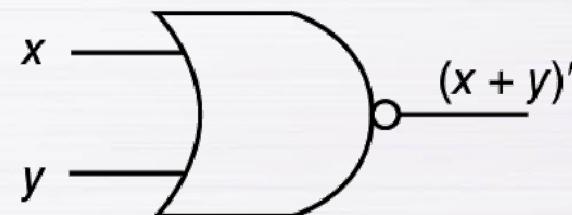
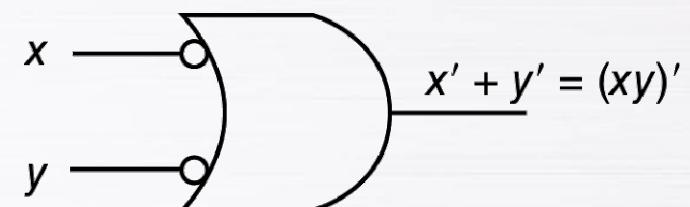
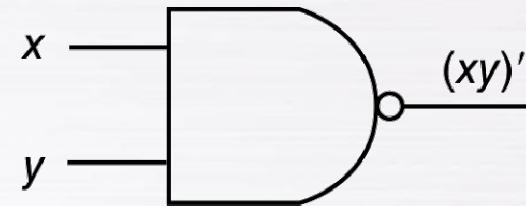
- NAND and NOR are two very important gates. Their symbols and truth tables are shown at the right.

X NAND Y

X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

X NOR Y

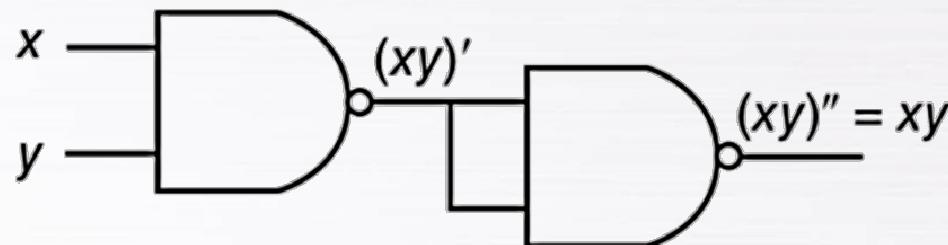
X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0



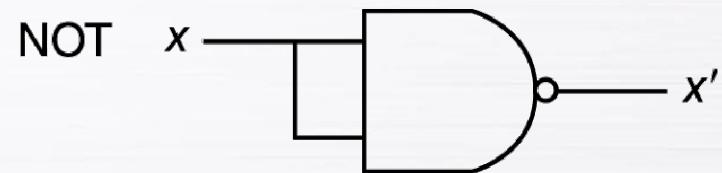
3.3 Logic Gates

- NAND and NOR are known as *universal gates* because they are inexpensive to manufacture and any Boolean function can be constructed using only NAND or only NOR gates.

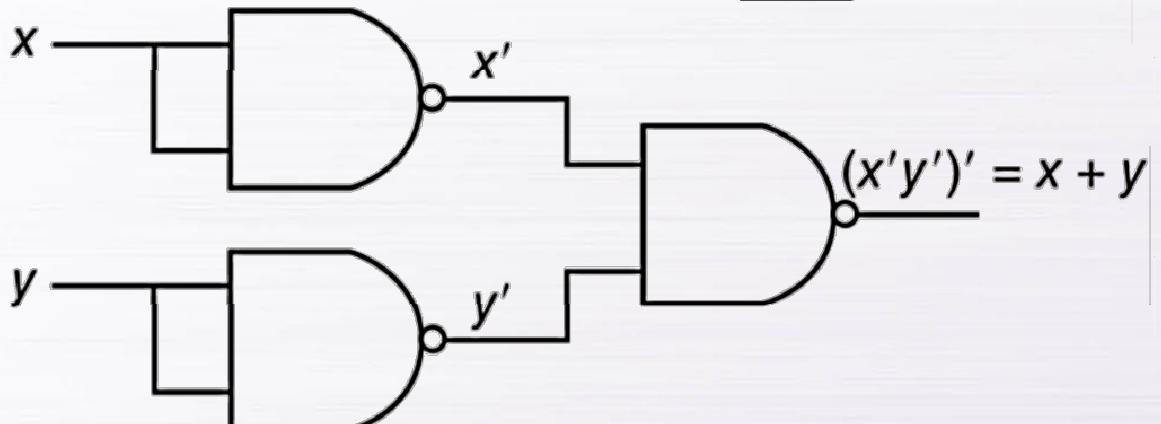
AND



NOT

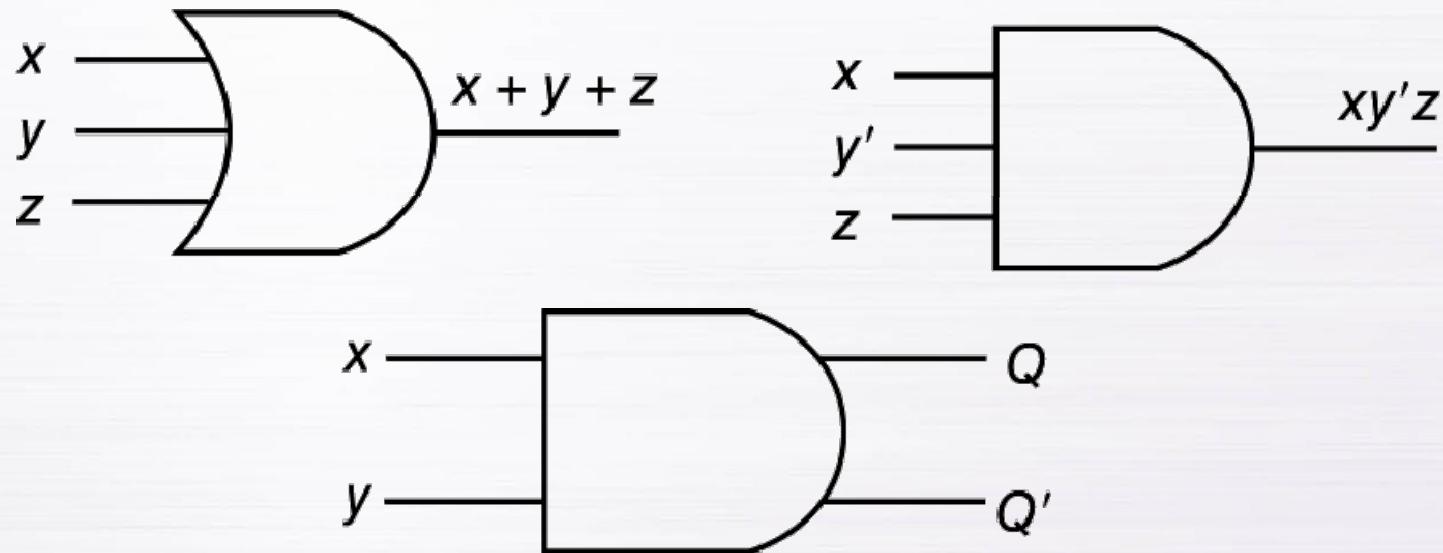


OR



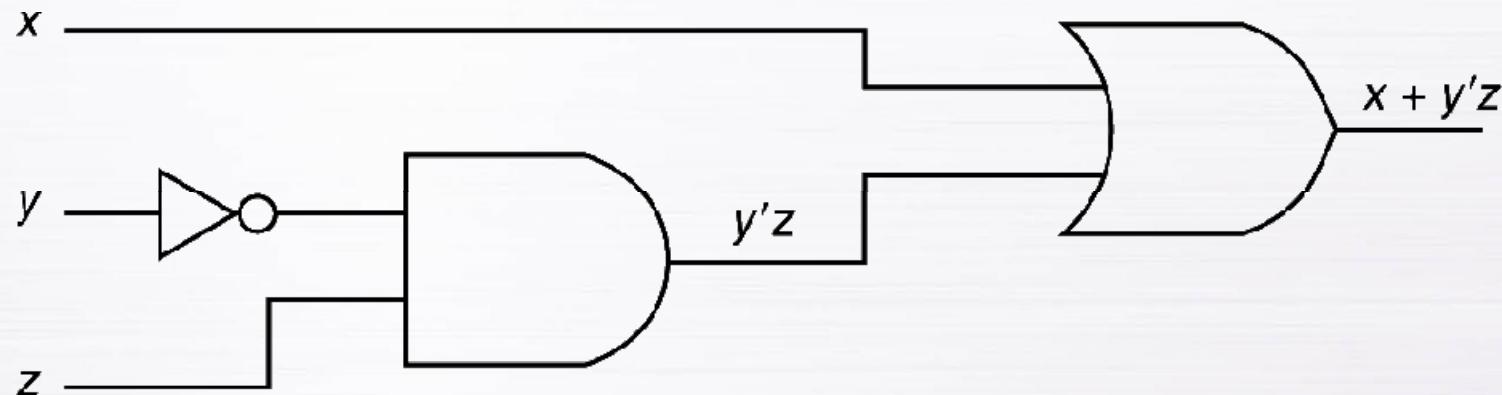
3.3 Logic Gates

- Gates can have multiple inputs and more than one output.
 - A second output can be provided for the complement of the operation.
 - We'll see more of this later.



3.3 Logic Gates

- The main thing to remember is that combinations of gates implement Boolean functions.
- The circuit below implements the Boolean function $F(x, y, z) = x + y'z$:



We simplify our Boolean expressions so that we can create simpler circuits.

3.5 Combinational Circuits

- We have designed a circuit that implements the Boolean function:

$$F(X, Y, Z) = X + \bar{Y}Z$$

- This circuit is an example of a *combinational logic* circuit.
- Combinational logic circuits produce a specified output (almost) at the instant when input values are applied.
 - In a later section, we will explore circuits where this is not the case.

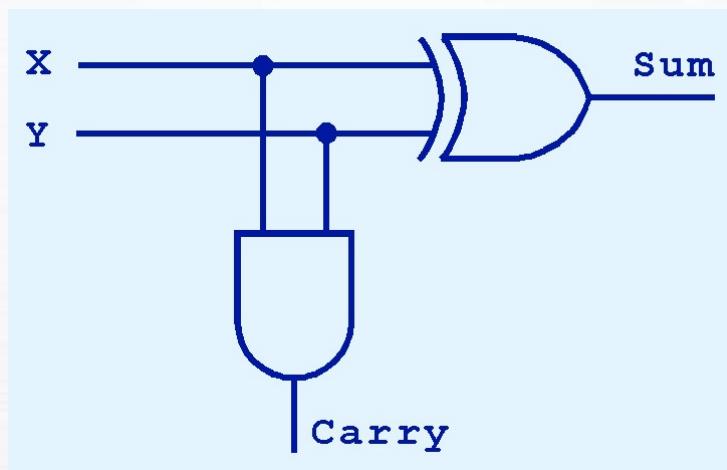
3.5 Combinational Circuits

- Combinational logic circuits give us many useful devices.
- One of the simplest is the *half adder*, which finds the sum of two bits.
- We can gain some insight as to the construction of a half adder by looking at its truth table, shown at the right.

Inputs		Outputs	
x	y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

3.5 Combinational Circuits

- As we see, the sum can be found using the XOR operation and the carry using the AND operation.



Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

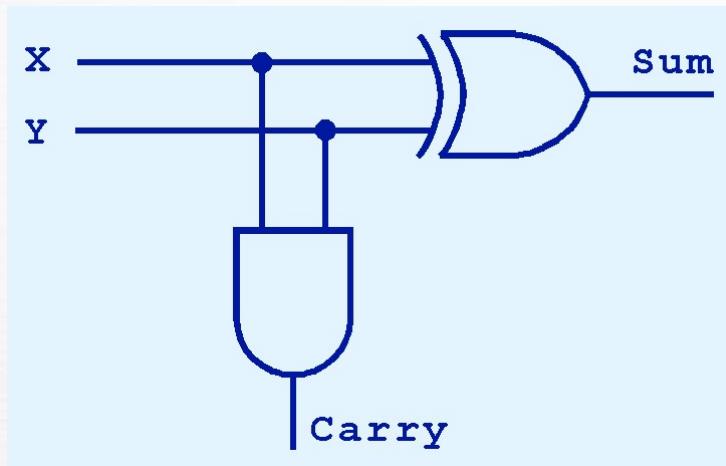
3.5 Combinational Circuits

- We can change our half adder into to a full adder by including gates for processing the carry bit.
- The truth table for a full adder is shown at the right.

Inputs			Outputs		
Carry In			Carry		
x	y	In	Sum	Out	
0	0	0	0	0	
0	0	1	1	0	
0	1	0	1	0	
0	1	1	0	1	
1	0	0	1	0	
1	0	1	0	1	
1	1	0	0	1	
1	1	1	1	1	

3.5 Combinational Circuits

- How can we change the half adder shown below to make it a full adder?

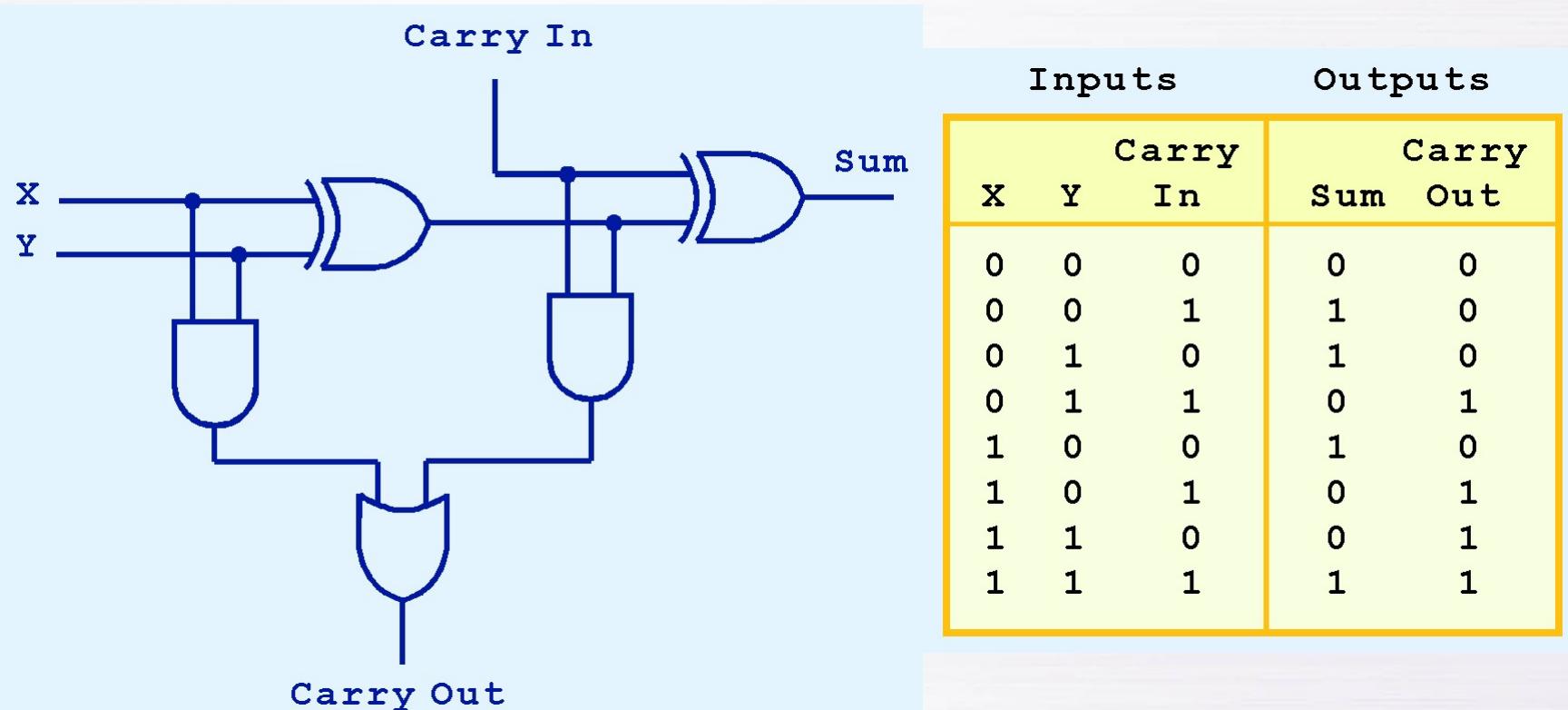


Inputs Outputs

Carry In			Carry Sum Out	
X	Y			
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

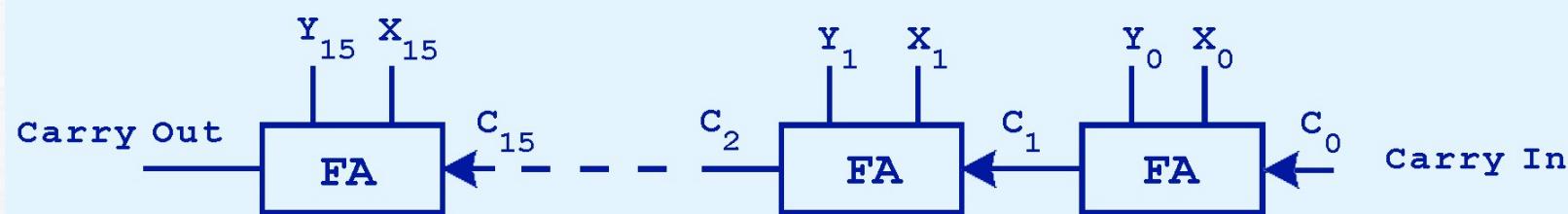
3.5 Combinational Circuits

- Here's our completed full adder.



3.5 Combinational Circuits

- Just as we combined half adders to make a full adder, full adders can connected in series.
- The carry bit “ripples” from one adder to the next; hence, this configuration is called a *ripple-carry adder*.



Today's systems employ more efficient adders.

3.5 Combinational Circuits

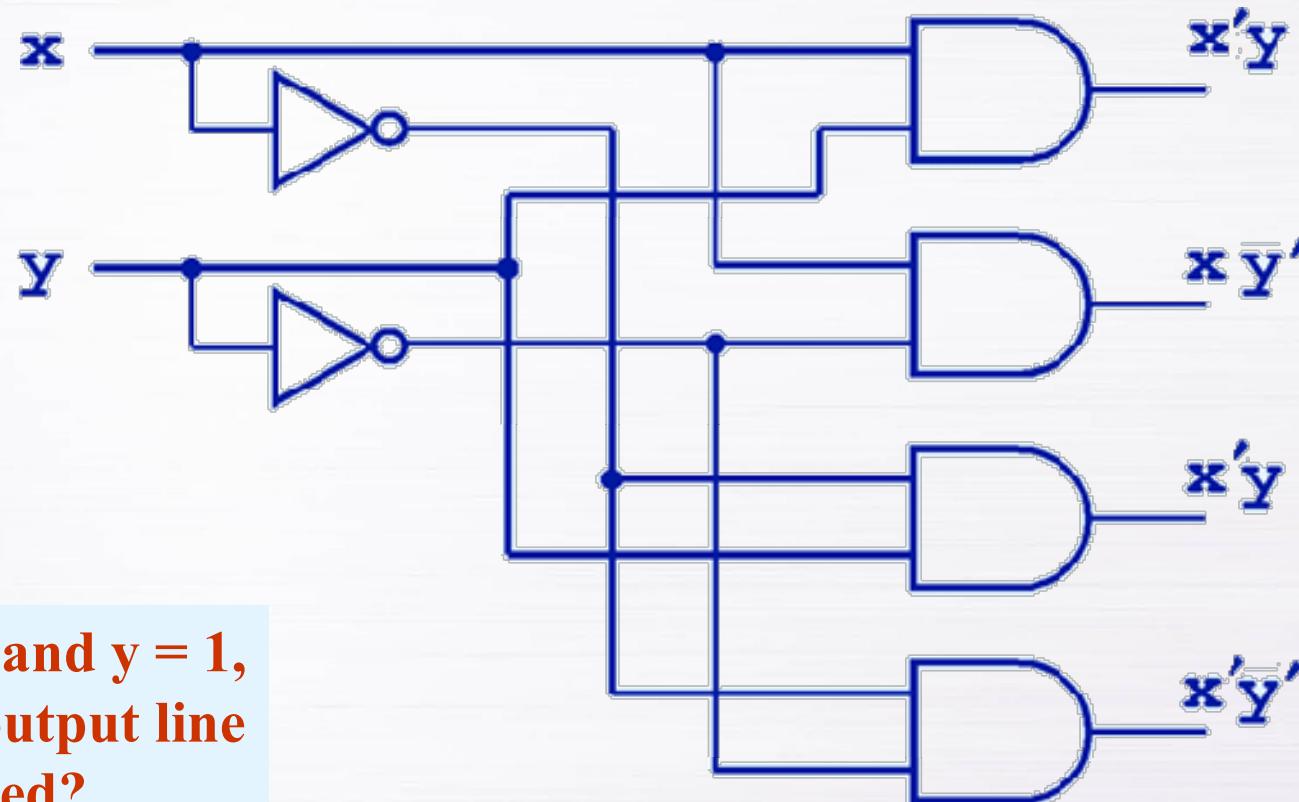
- Decoders are another important type of combinational circuit.
- Among other things, they are useful in selecting a memory location according a binary value placed on the address lines of a memory bus.
- Address decoders with n inputs can select any of 2^n locations.

This is a block diagram for a decoder.



3.5 Combinational Circuits

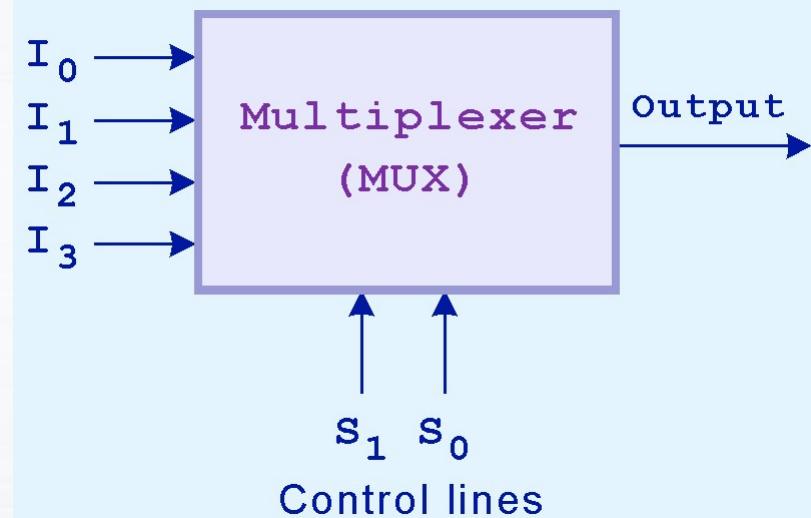
- This is what a 2-to-4 decoder looks like on the inside.



If $x = 0$ and $y = 1$,
which output line
is enabled?

3.5 Combinational Circuits

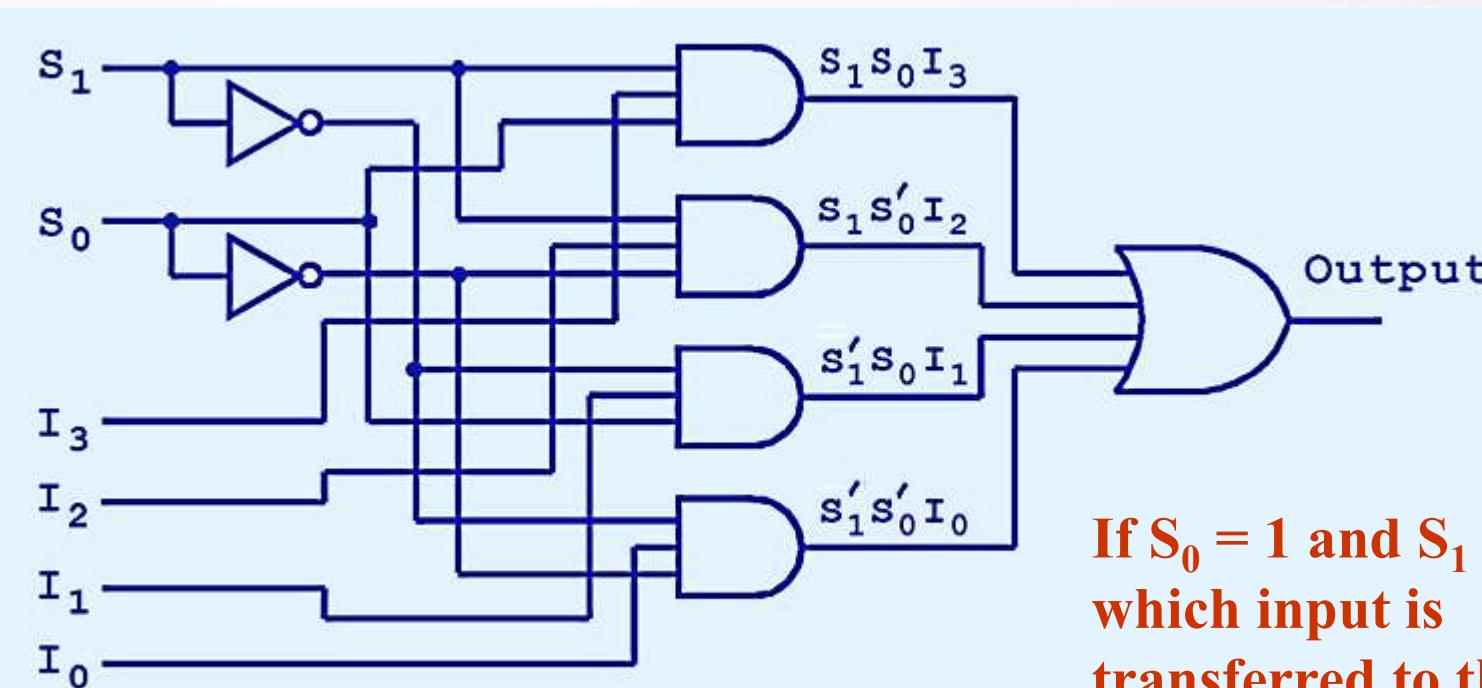
- A multiplexer does just the opposite of a decoder.
- It selects a single output from several inputs.
- The particular input chosen for output is determined by the value of the multiplexer's control lines.
- To be able to select among n inputs, $\log_2 n$ control lines are needed.



This is a block diagram for a multiplexer.

3.5 Combinational Circuits

- This is what a 4-to-1 multiplexer looks like on the inside.

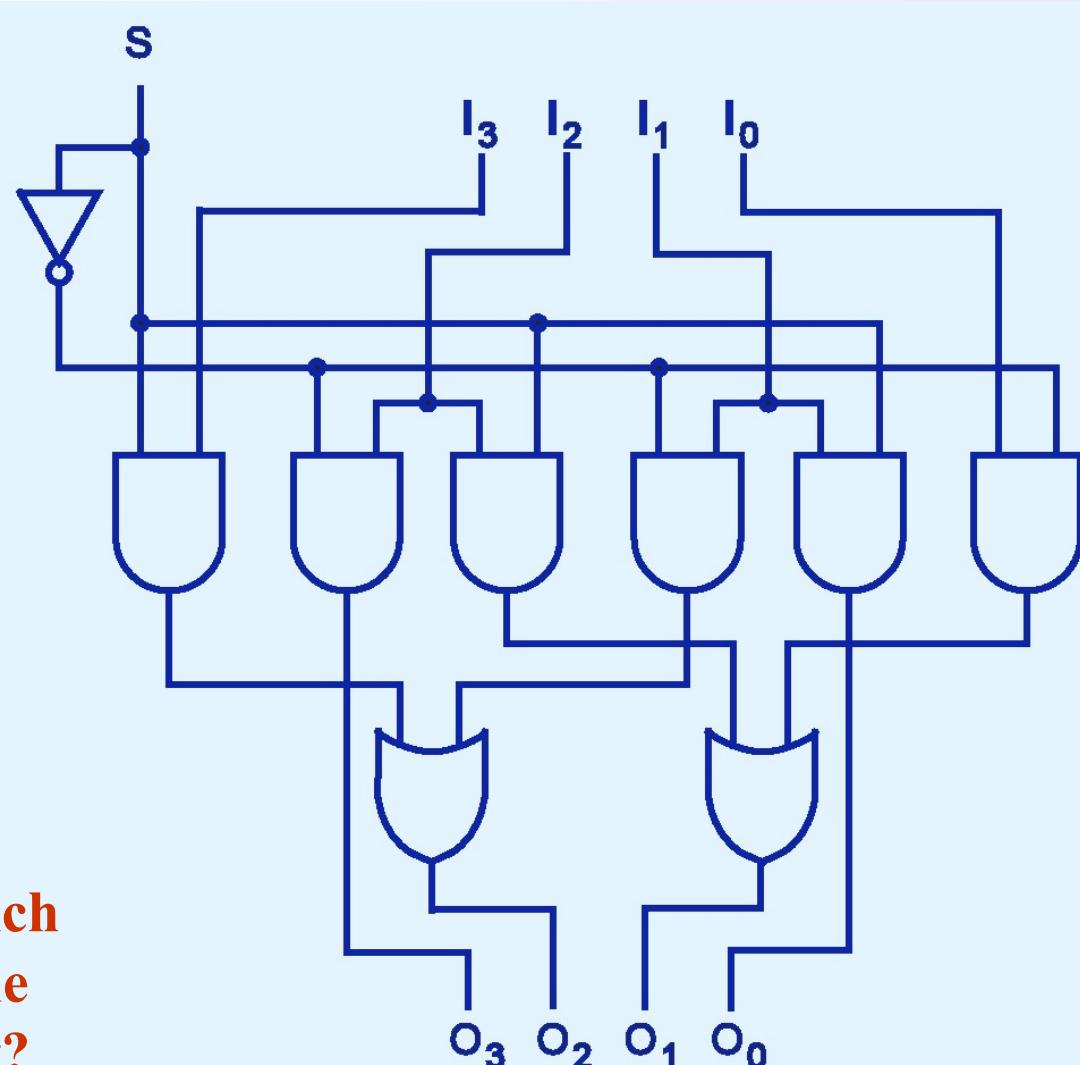


If $S_0 = 1$ and $S_1 = 0$,
which input is
transferred to the
output?

3.5 Combinational Circuits

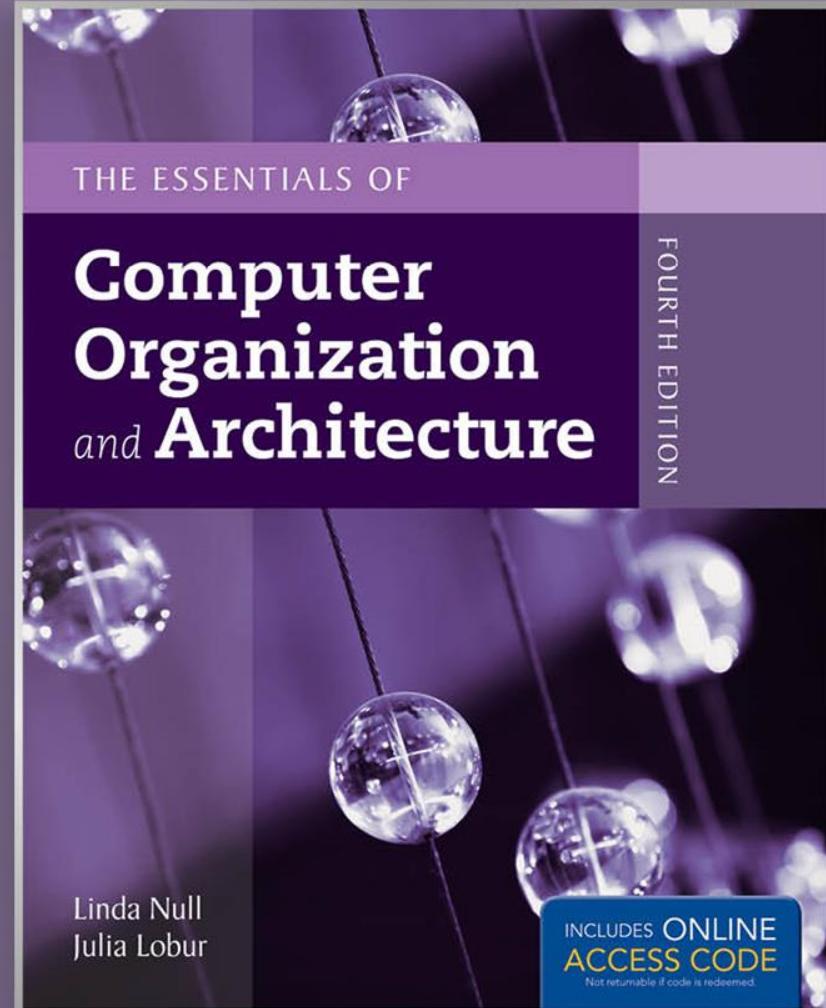
- This shifter moves the bits of a nibble one position to the left or right.

If $S = 0$, in which direction do the input bits shift?



Chapter 3.6-

Sequential Circuits

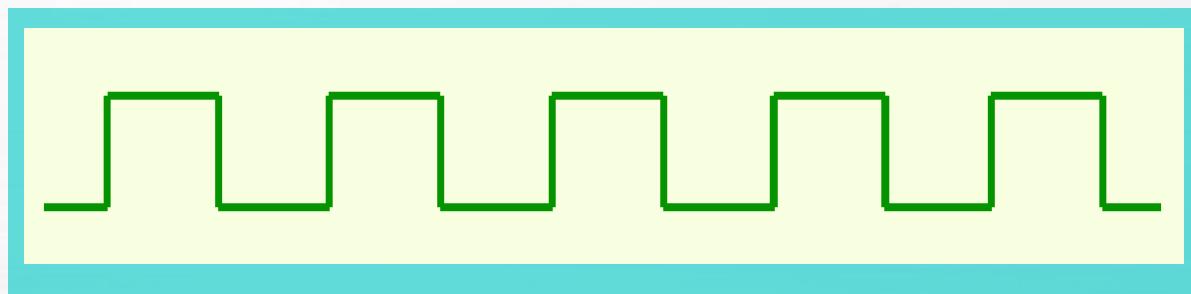


3.6 Sequential Circuits

- Combinational logic circuits are perfect for situations when we require the immediate application of a Boolean function to a set of inputs.
- There are other times, however, when we need a circuit to change its value with consideration to **its current state** as well as **its inputs**.
 - These circuits have to “remember” their current state.
- *Sequential logic circuits* provide this functionality for us.

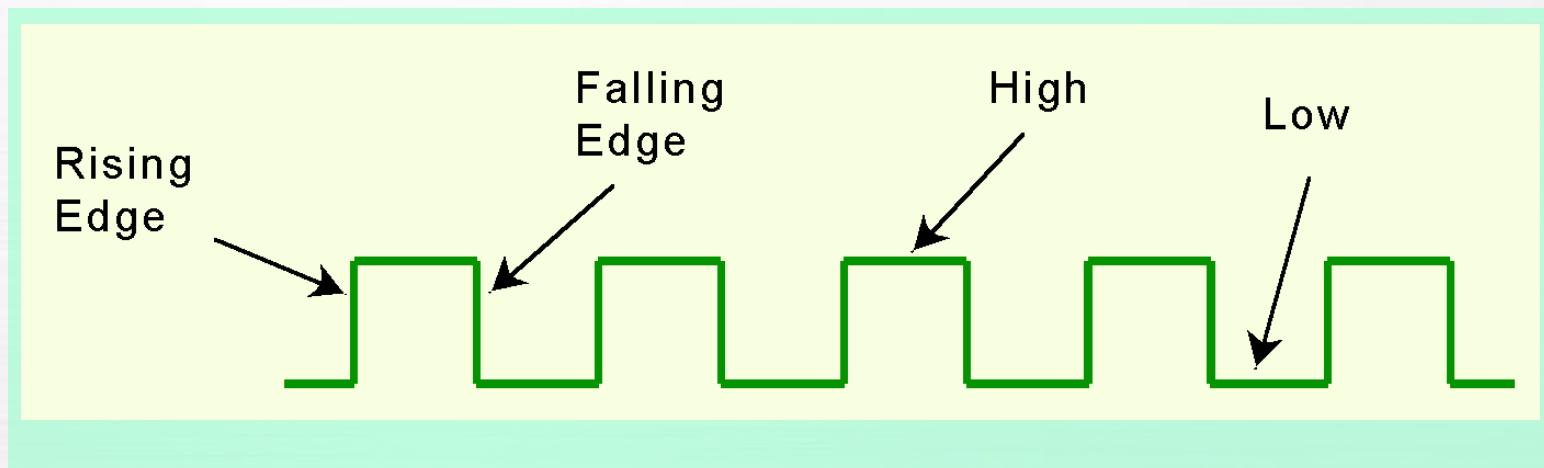
3.6 Sequential Circuits

- As the name implies, sequential logic circuits require a means by which events can be sequenced.
- State changes are controlled by clocks.
 - A “clock” is a special circuit that sends electrical pulses through a circuit.
- Clocks produce electrical waveforms such as the one shown below.



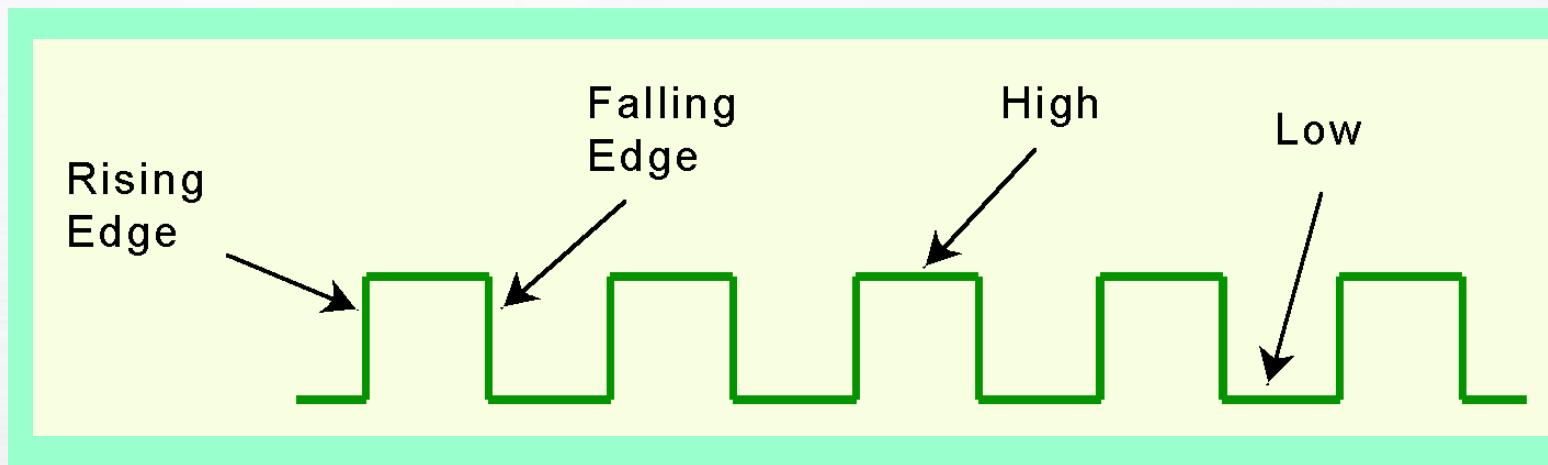
3.6 Sequential Circuits

- State changes occur in sequential circuits only when the clock ticks.
 - **Synchronous** sequential circuits
- Circuits can change state on the **rising edge**, **falling edge**, or when the clock pulse reaches its **highest voltage**.



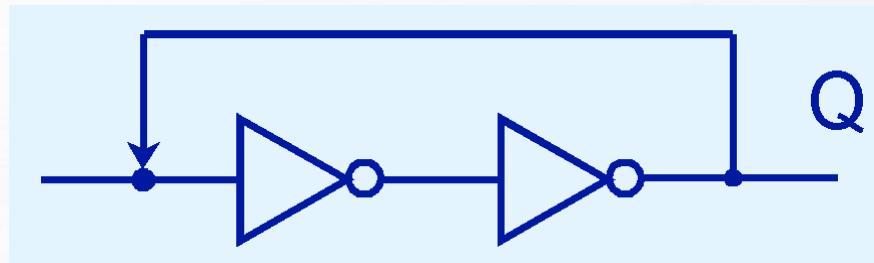
3.6 Sequential Circuits

- Circuits that change state on the rising edge, or falling edge of the clock pulse are called **edge-triggered**.
- **Level-triggered circuits** change state when the clock voltage reaches its highest or lowest level.



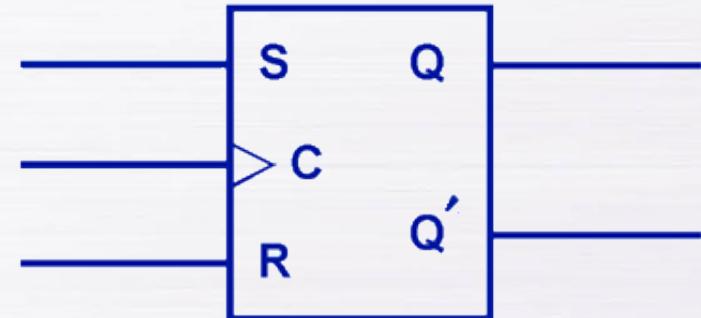
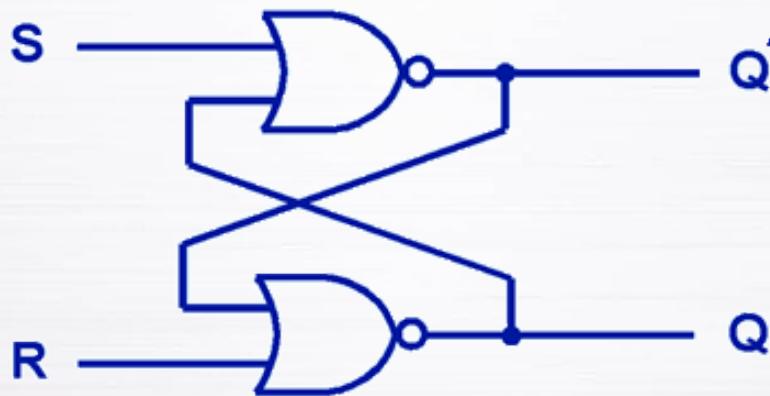
3.6 Sequential Circuits

- To retain their state values, sequential circuits rely on *feedback*.
- Feedback in digital circuits occurs when an output is looped back to the input.
- A simple example of this concept is shown below.
 - If Q is 0 it will always be 0, if it is 1, it will always be 1. Why?



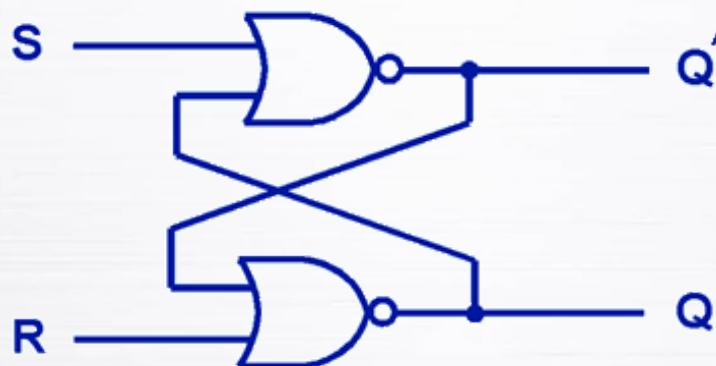
3.6 Sequential Circuits

- You can see how feedback works by examining the most basic sequential logic components, the SR flip-flop.
 - The “SR” stands for set/reset.
- The internals of an SR flip-flop are shown below, along with its block diagram.



3.6 Sequential Circuits

- The behavior of an SR flip-flop is described by a characteristic table.
- $Q(t)$ means the value of the output at time t .
- $Q(t+1)$ is the value of Q after the next clock pulse.



S	R	$Q(t+1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	undefined

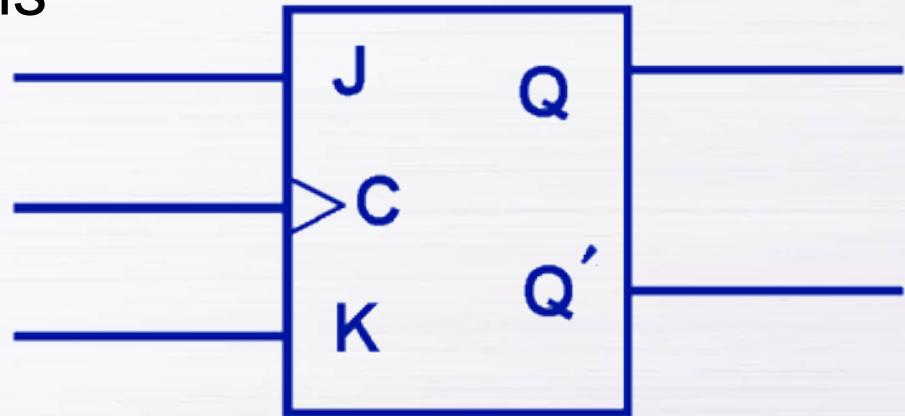
3.6 Sequential Circuits

- The SR flip-flop actually has three inputs: S, R, and its current output, Q.
- Thus, we can construct a truth table for this circuit, as shown at the right.
- Notice the two undefined values. When both S and R are 1, the SR flip-flop is unstable.

Present State			Next State
S	R	$Q(t)$	$Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	undefined
1	1	1	undefined

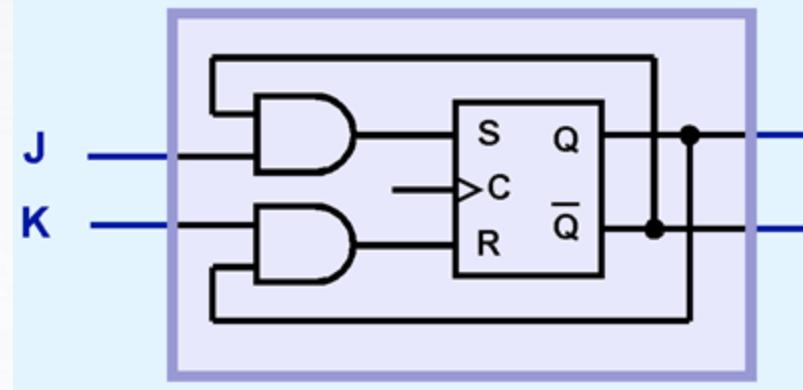
3.6 Sequential Circuits

- If we can be sure that the inputs to an SR flip-flop will never both be 1, we will never have an unstable circuit. This may not always be the case.
- The SR flip-flop can be modified to provide a stable state when both inputs are 1.
- This modified flip-flop is called a JK flip-flop, shown at the right.



3.6 Sequential Circuits

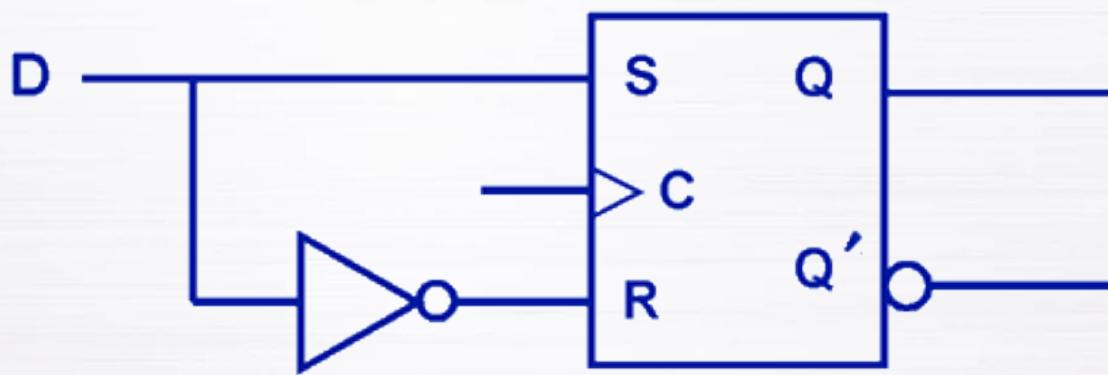
- At the right, we see how an SR flip-flop can be modified to create a JK flip-flop.
- The characteristic table indicates that the flip-flop is stable for all inputs.



J	K	$Q(t+1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}(t)$

3.6 Sequential Circuits

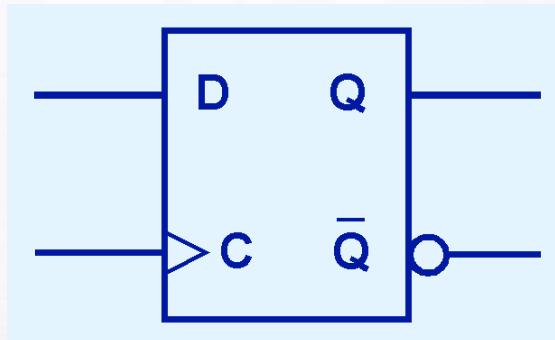
- Another modification of the SR flip-flop is the D flip-flop, shown below with its characteristic table.
- You will notice that the output of the flip-flop remains the same during subsequent clock pulses. The output changes only when the value of D changes.



D	$Q(t+1)$
0	0
1	1

3.6 Sequential Circuits

- The D flip-flop is the fundamental circuit of computer memory.
 - D flip-flops are usually illustrated using the block diagram shown below.
- The characteristic table for the D flip-flop is shown at the right.



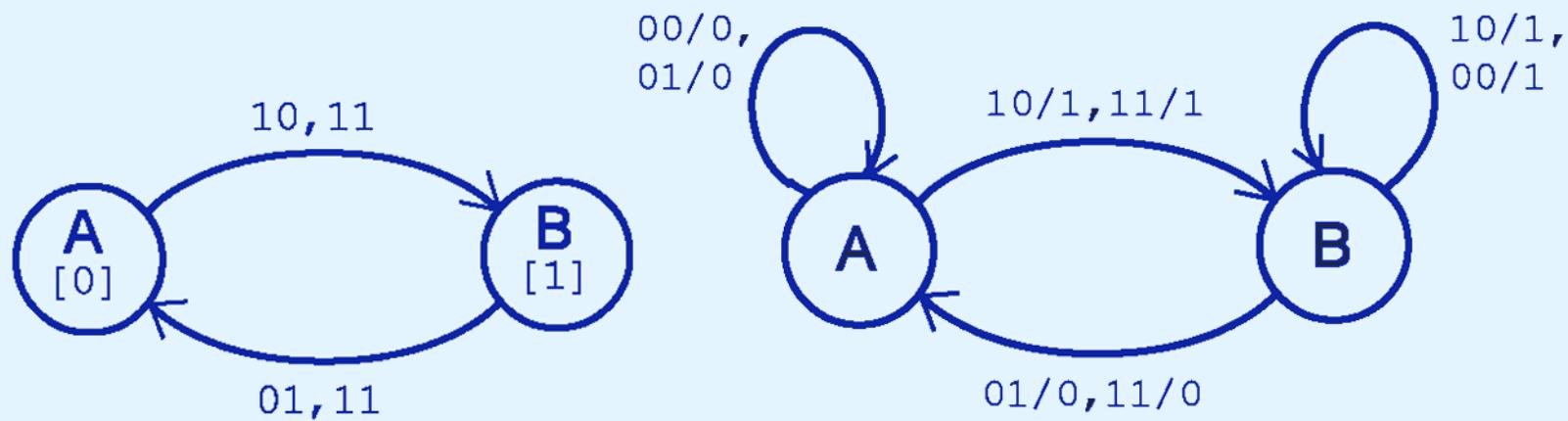
D	$Q(t+1)$
0	0
1	1

3.6 Sequential Circuits

- The behavior of sequential circuits can be expressed using characteristic tables or finite state machines (FSMs).
 - FSMs consist of a set of nodes that hold the states of the machine and a set of arcs that connect the states.
- Moore and Mealy machines are two types of FSMs that are equivalent.
 - They differ only in how they express the outputs of the machine.
- Moore machines place outputs on each node, while Mealy machines present their outputs on the transitions.

3.6 Sequential Circuits

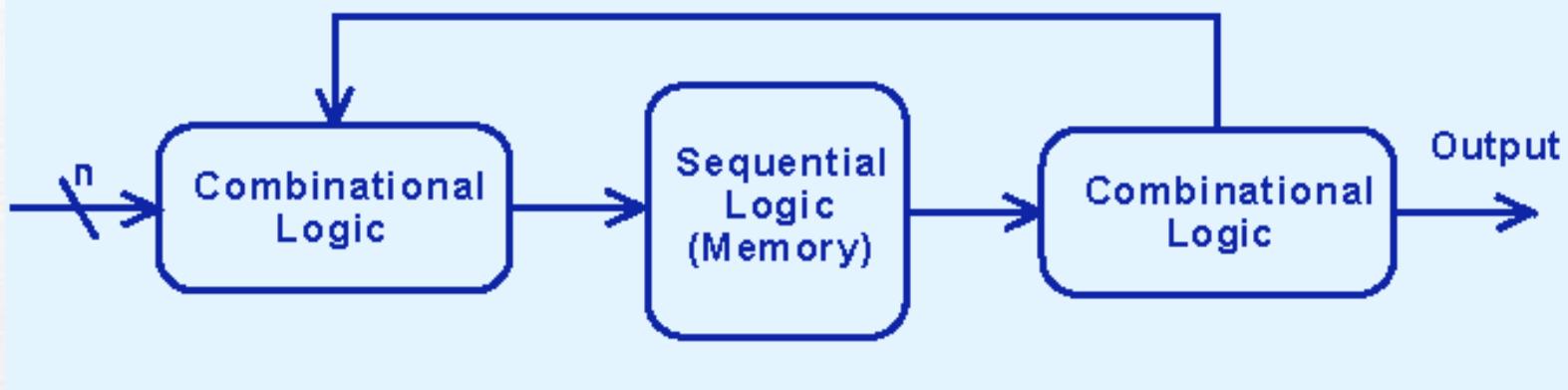
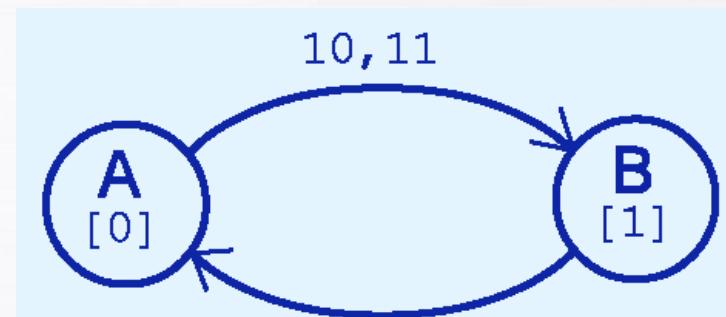
- The behavior of a JK flop-flop is depicted below by a Moore machine (left) and a Mealy machine (right).



3.6 Sequential Circuits

- Although the behavior of Moore and Mealy machines is identical, their implementations differ.

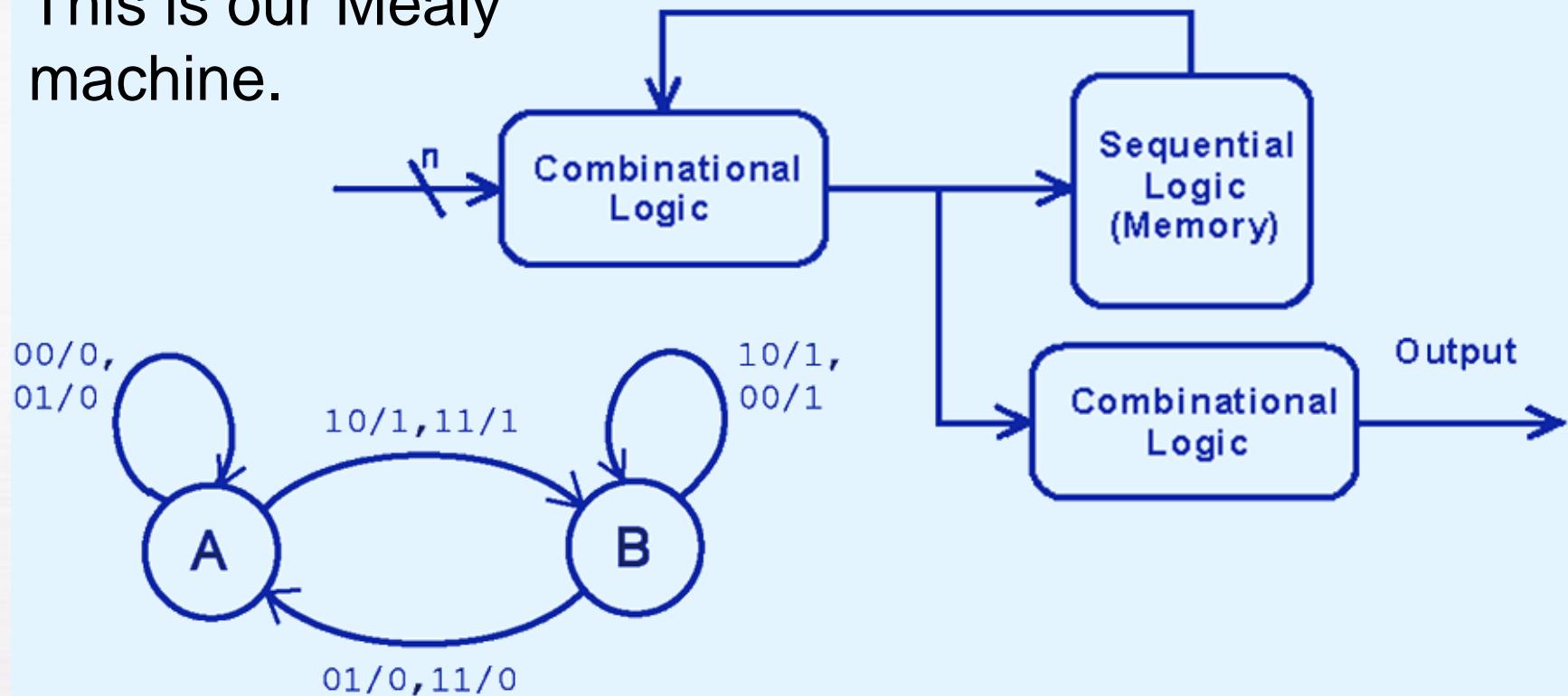
This is our Moore machine.



3.6 Sequential Circuits

- Although the behavior of Moore and Mealy machines is identical, their implementations differ.

This is our Mealy machine.

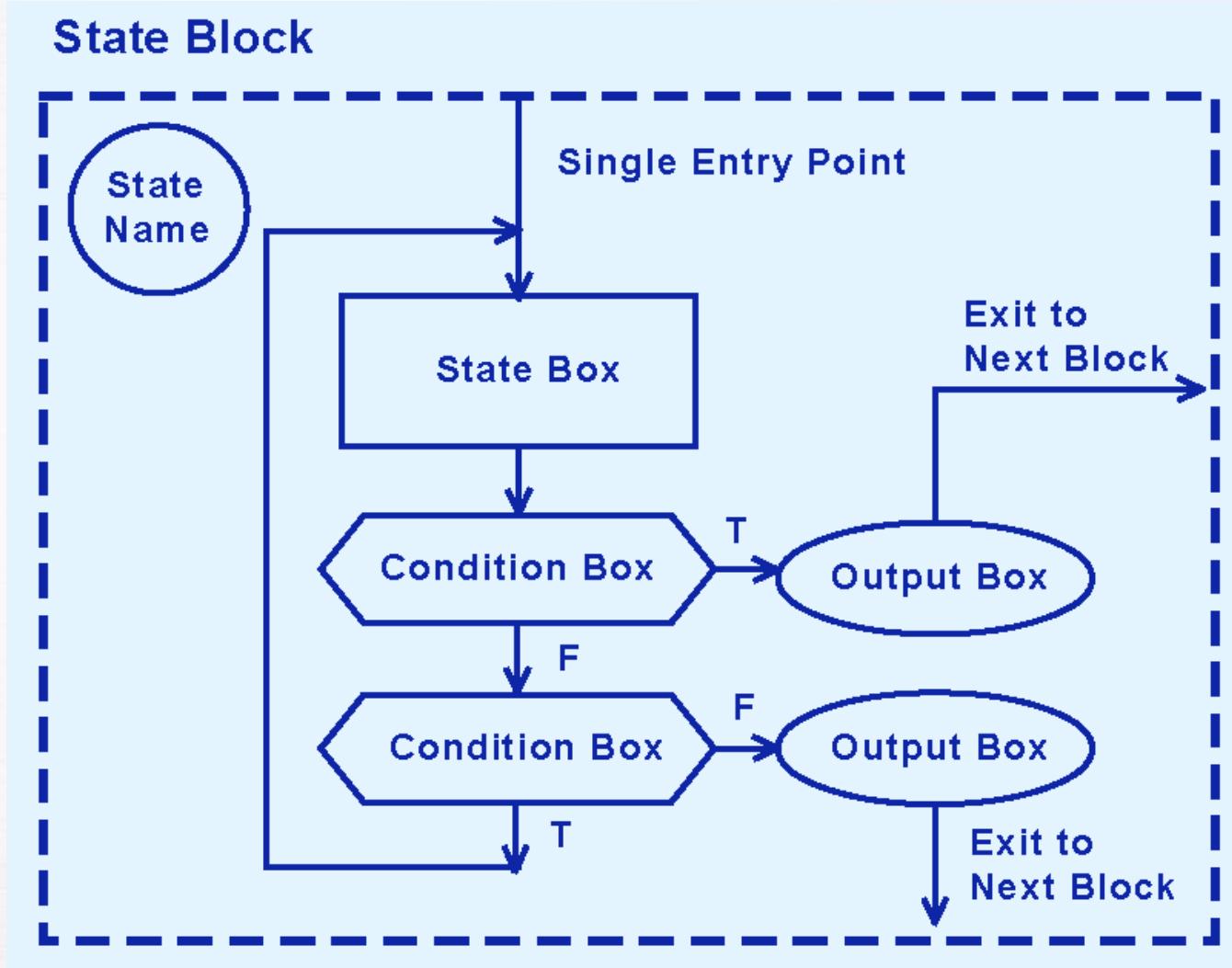


3.6 Sequential Circuits

- It is difficult to express the complexities of actual implementations using only Moore and Mealy machines.
 - For one thing, they do not address the intricacies of timing very well.
 - Secondly, it is often the case that an interaction of numerous signals is required to advance a machine from one state to the next.
- For these reasons, Christopher Clare invented the algorithmic state machine (ASM).

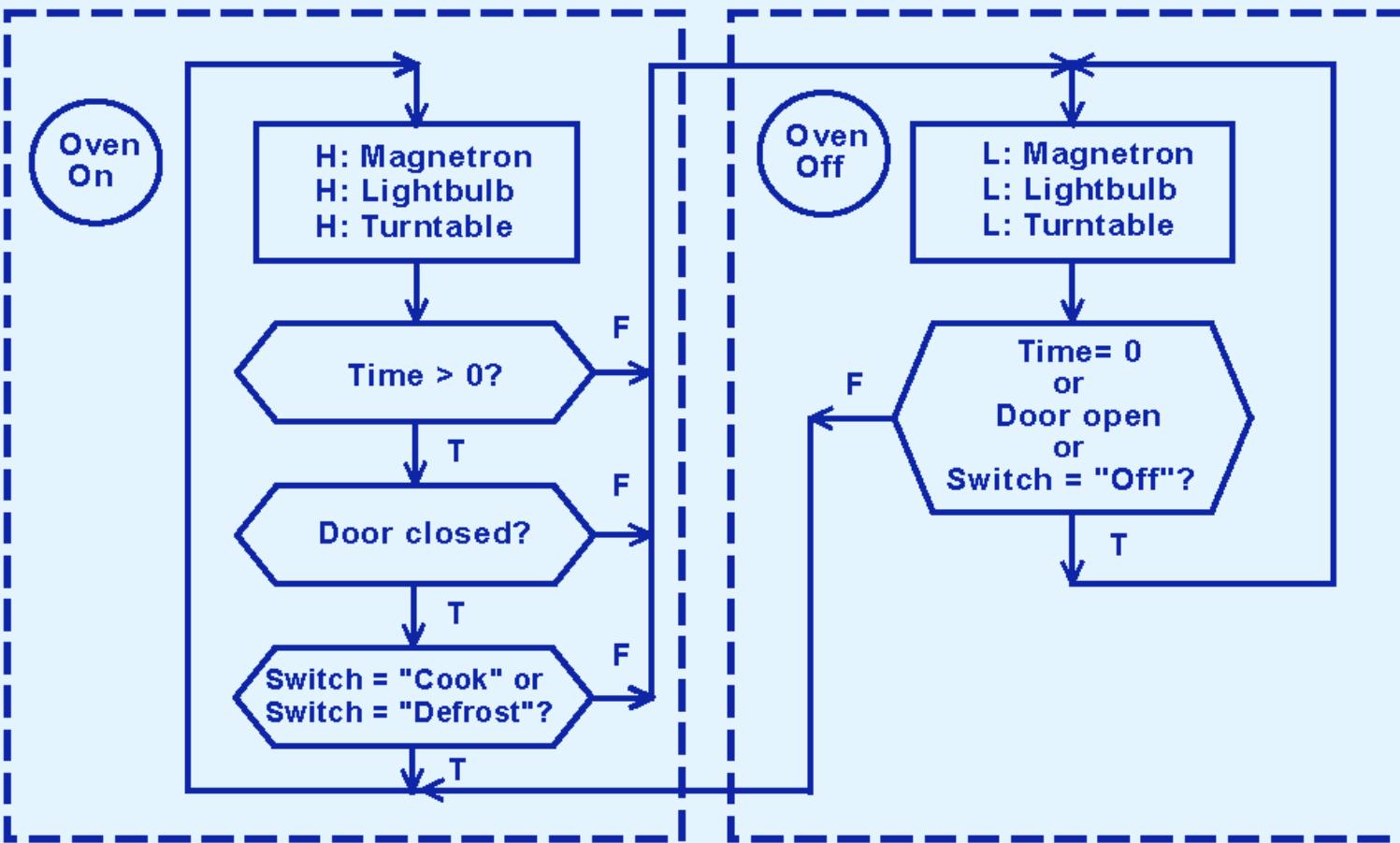
The next slide illustrates the components of an ASM.

3.6 Sequential Circuits



3.6 Sequential Circuits

- This is an ASM for a microwave oven.



3.6 Sequential Circuits

- Sequential circuits are used anytime that we have a “stateful” application.
 - A stateful application is one where the next state of the machine depends on the current state of the machine and the input.
- A stateful application requires both combinational and sequential logic.
- The following slides provide several examples of circuits that fall into this category.

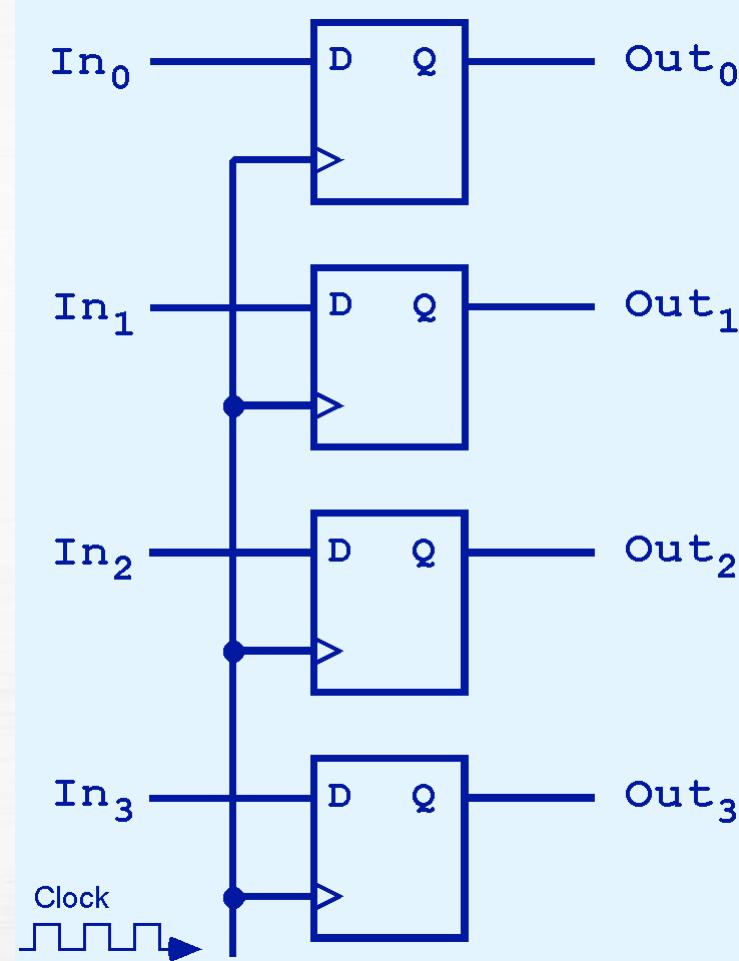
Can you think of others?

3.6 Sequential Circuits

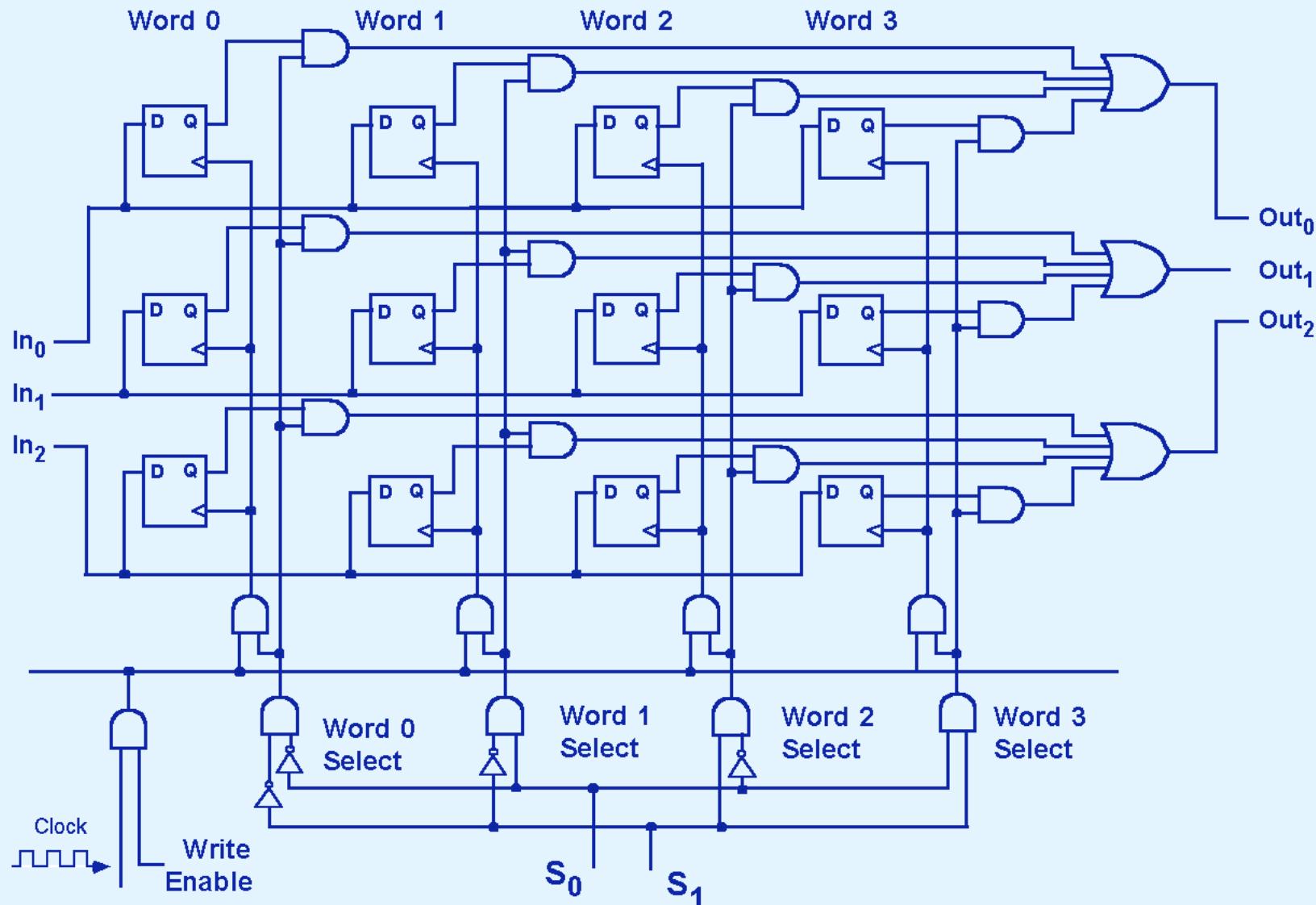
- This illustration shows a 4-bit register consisting of D flip-flops. You will usually see its block diagram (below) instead.



A larger memory configuration is shown on the next slide.

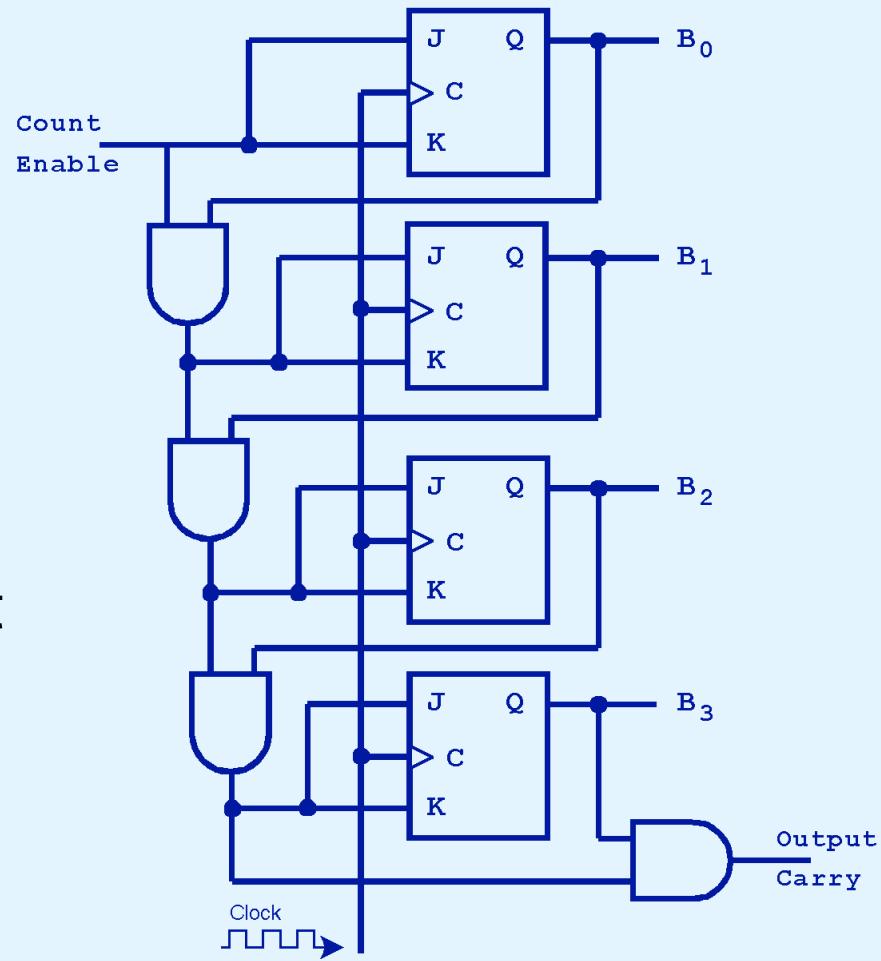


3.6 Sequential Circuits



3.6 Sequential Circuits

- A binary counter is another example of a sequential circuit.
- The low-order bit is complemented at each clock pulse.
- Whenever it changes from 0 to 1, the next bit is complemented, and so on through the other flip-flops.

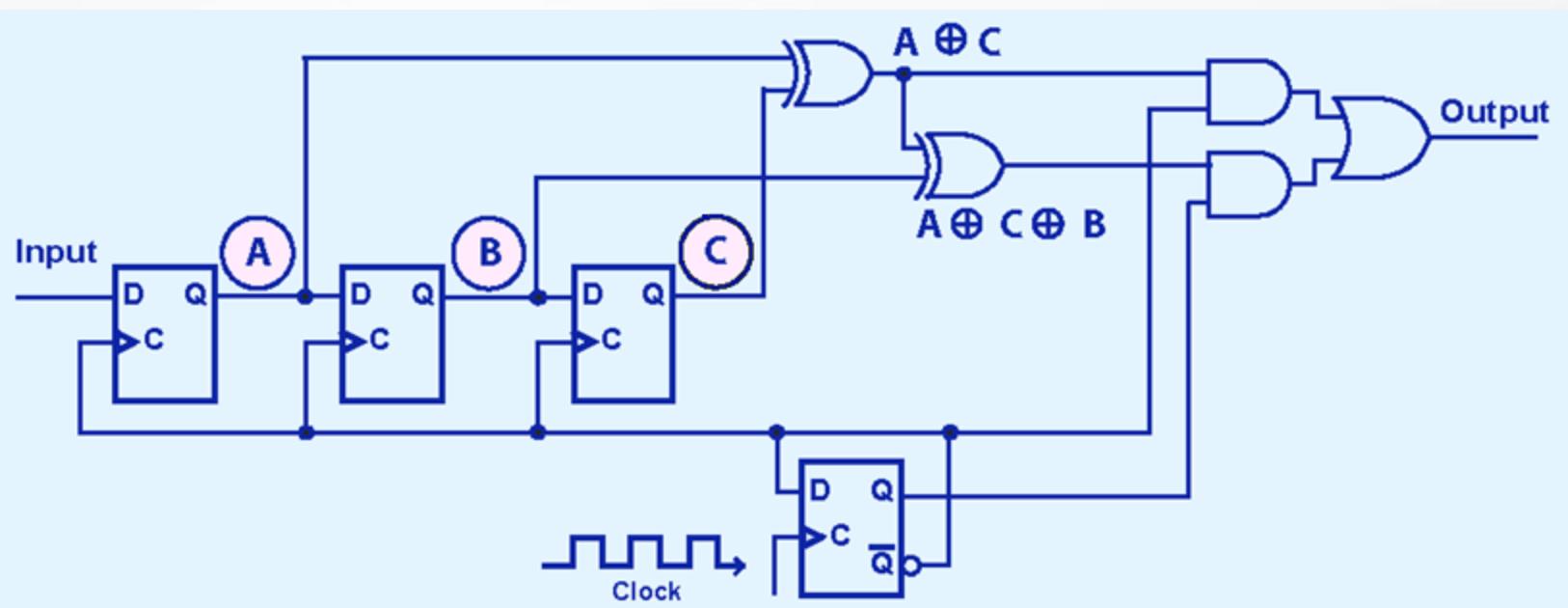


3.6 Sequential Circuits

- Convolutional coding and decoding requires sequential circuits.
- One important convolutional code is the $(2,1)$ convolutional code that underlies the PRML code that is briefly described at the end of Chapter 2.
- A $(2, 1)$ convolutional code is so named because two symbols are output for every one symbol input.
- A convolutional encoder for PRML with its characteristic table is shown on the next slide.

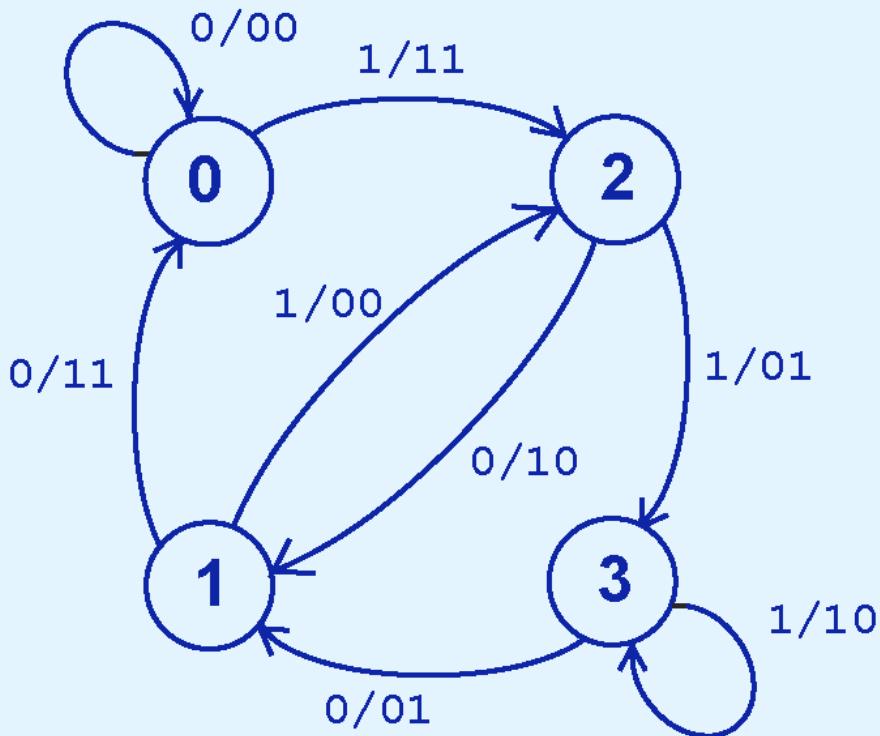
3.6 Sequential Circuits

Input A	Current State B C	Next State B C	Output	Input A	Current State B C	Next State B C	Output
0	00	00	00	0	10	01	10
1	00	10	11	1	10	11	01
0	01	00	11	0	11	01	01
1	01	10	00	1	11	11	10



3.6 Sequential Circuits

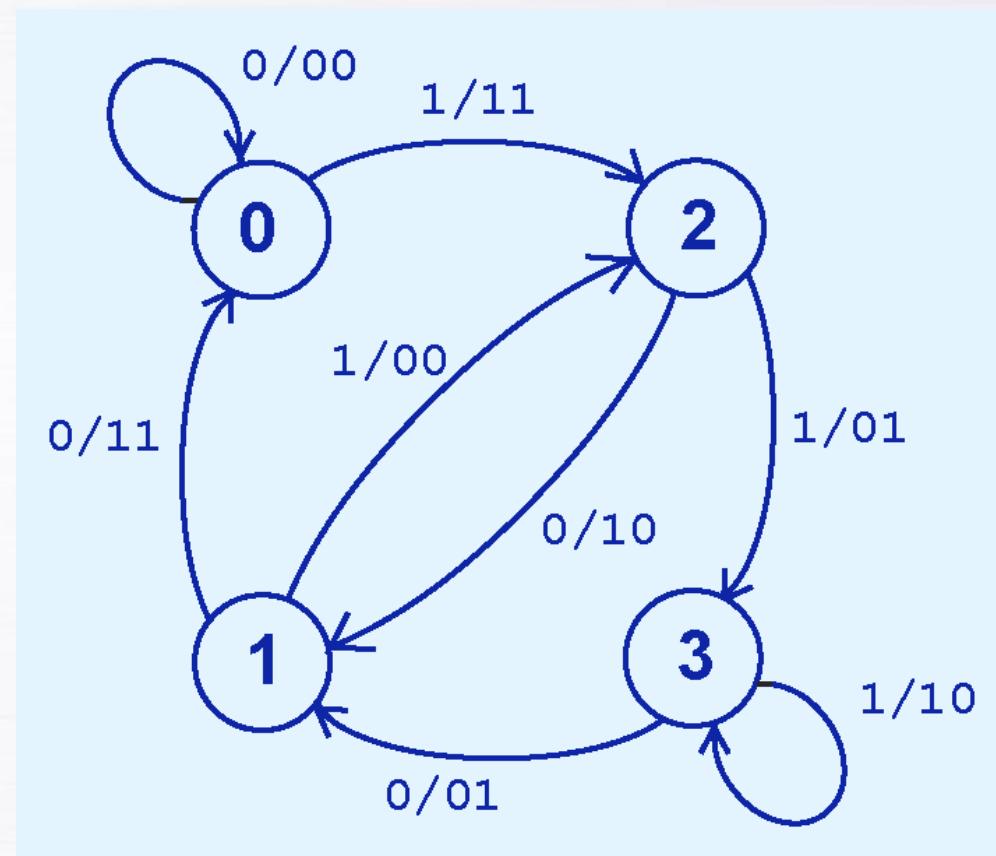
This is the Mealy machine for our encoder.



Input A	Current State B C	Next State B C	Output
0	00	00	00
1	00	10	11
0	01	00	11
1	01	10	00
0	10	01	10
1	10	11	01
0	11	01	01
1	11	11	10

3.6 Sequential Circuits

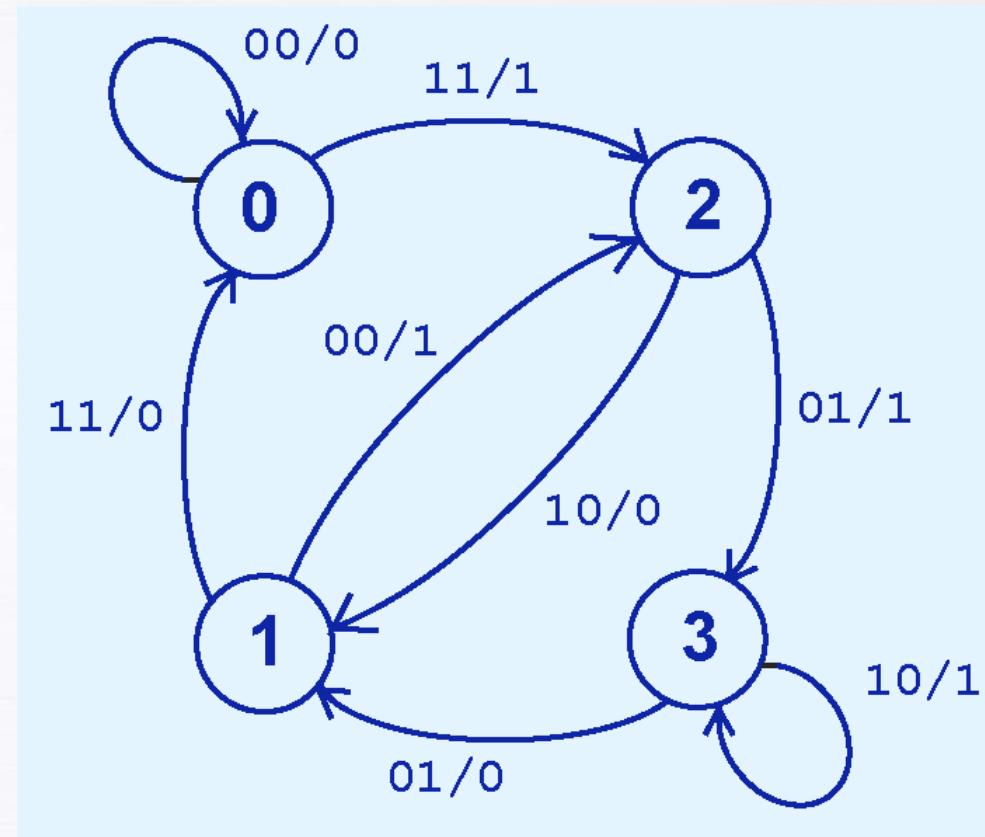
- The fact that there is a limited set of possible state transitions in the encoding process is crucial to the error correcting capabilities of PRML.
- You can see by our Mealy machine for encoding that:



$$F(1101 \ 0010) = 11 \ 01 \ 01 \ 00 \ 10 \ 11 \ 11 \ 10.$$

3.6 Sequential Circuits

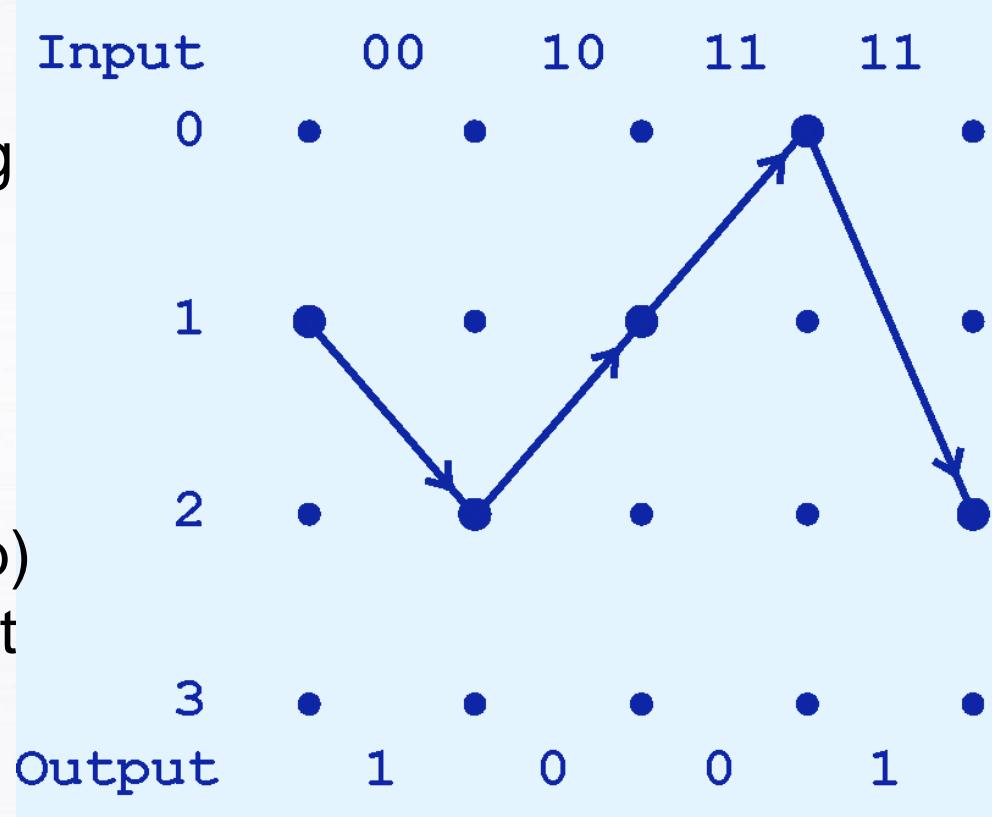
- The decoding of our code is provided by inverting the inputs and outputs of the Mealy machine for the encoding process.
- You can see by our Mealy machine for decoding that:



$$F(11 \ 01 \ 01 \ 00 \ 10 \ 11 \ 11 \ 10) = 1101 \ 0010$$

3.6 Sequential Circuits

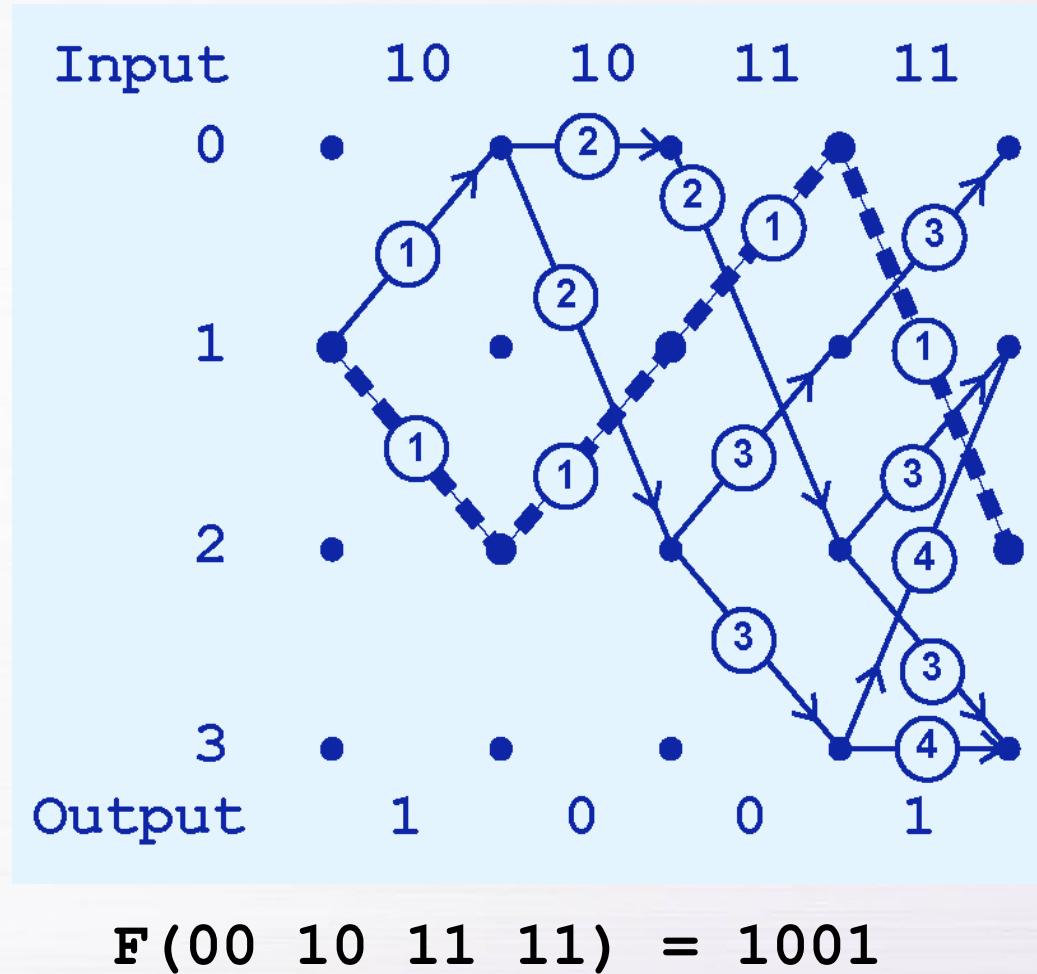
- Yet another way of looking at the decoding process is through a lattice diagram.
- Here we have plotted the state transitions based on the input (top) and showing the output at the bottom for the string 00 10 11 11.



$$F(00 \ 10 \ 11 \ 11) = 1001$$

3.6 Sequential Circuits

- Suppose we receive the erroneous string: 10 10 11 11.
- Here we have plotted the accumulated errors based on the allowable transitions.
- The path of least error outputs 1001, thus 1001 is the string of maximum likelihood.



3.7 Designing Circuits

- We have seen digital circuits from two points of view: digital analysis and digital synthesis.
 - *Digital analysis* explores the relationship between a circuit's inputs and its outputs.
 - *Digital synthesis* creates logic diagrams using the values specified in a truth table.
- Digital systems designers must also be mindful of the physical behaviors of circuits to include minute propagation delays that occur between the time when a circuit's inputs are energized and when the output is accurate and stable.

3.7 Designing Circuits

- Digital designers rely on specialized software to create efficient circuits.
 - Thus, software is an enabler for the construction of better hardware.
- Of course, software is in reality a collection of algorithms that could just as well be implemented in hardware.
 - Recall the Principle of Equivalence of Hardware and Software.

3.7 Designing Circuits

- When we need to implement a simple, specialized algorithm and its execution speed must be as fast as possible, a hardware solution is often preferred.
- This is the idea behind *embedded systems*, which are small special-purpose computers that we find in many everyday things.
- Embedded systems require special programming that demands an understanding of the operation of digital circuits, the basics of which you have learned in this chapter.

Chapter 3 Conclusion

- Computers are implementations of Boolean logic.
- Boolean functions are completely described by truth tables.
- Logic gates are small circuits that implement Boolean operators.
- The basic gates are AND, OR, and NOT.
 - The XOR gate is very useful in parity checkers and adders.
- The “universal gates” are NOR, and NAND.

Chapter 3 Conclusion

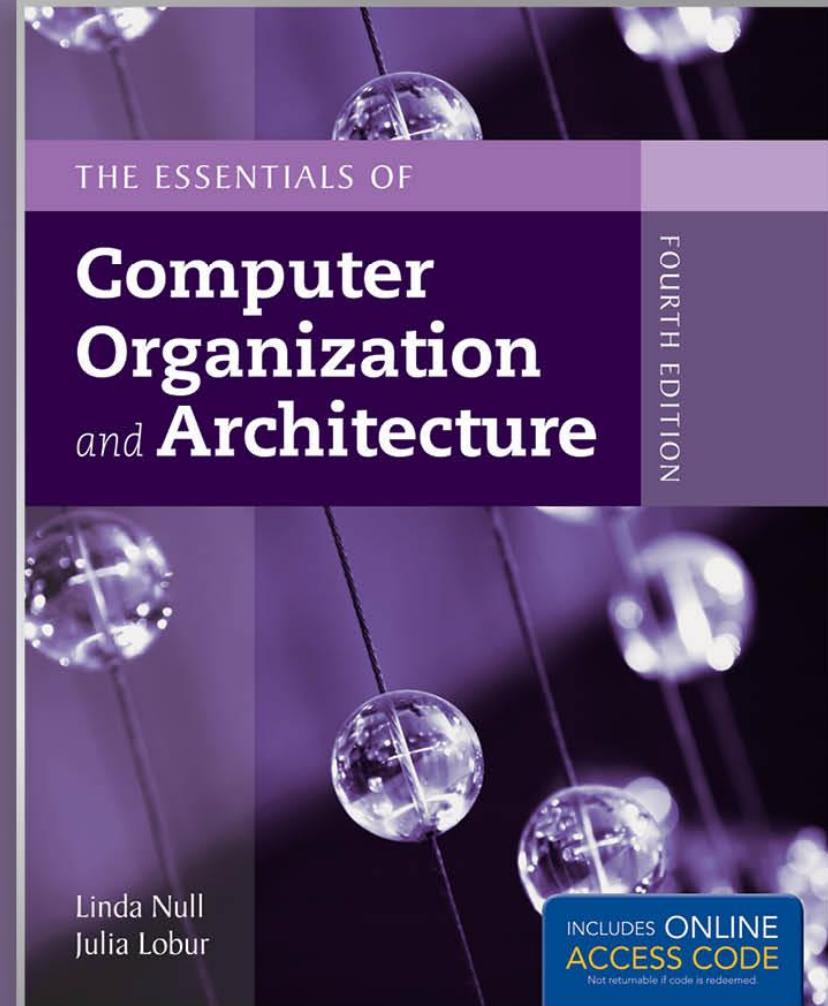
- Computer circuits consist of combinational logic circuits and sequential logic circuits.
- Combinational circuits produce outputs (almost) immediately when their inputs change.
- Sequential circuits require clocks to control their changes of state.
- The basic sequential circuit unit is the flip-flop: The behaviors of the SR, JK, and D flip-flops are the most important to know.

Chapter 3 Conclusion

- The behavior of sequential circuits can be expressed using characteristic tables or through various finite state machines.
- Moore and Mealy machines are two finite state machines that model high-level circuit behavior.
- Algorithmic state machines are better than Moore and Mealy machines at expressing timing and complex signal interactions.
- Examples of sequential circuits include memory, counters, and Viterbi encoders and decoders.

Chapter 4

MARIE: An Introduction to a Simple Computer



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.

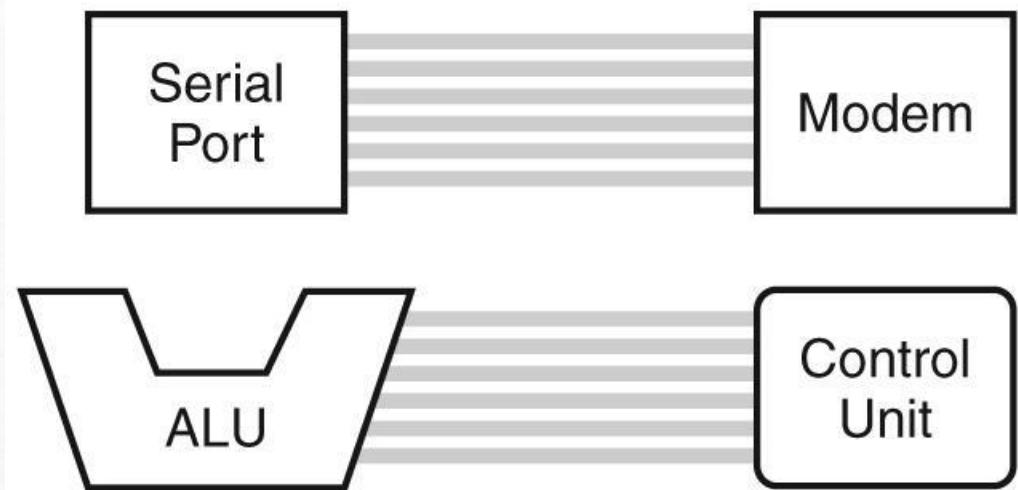
4.2 CPU Basics

- 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:

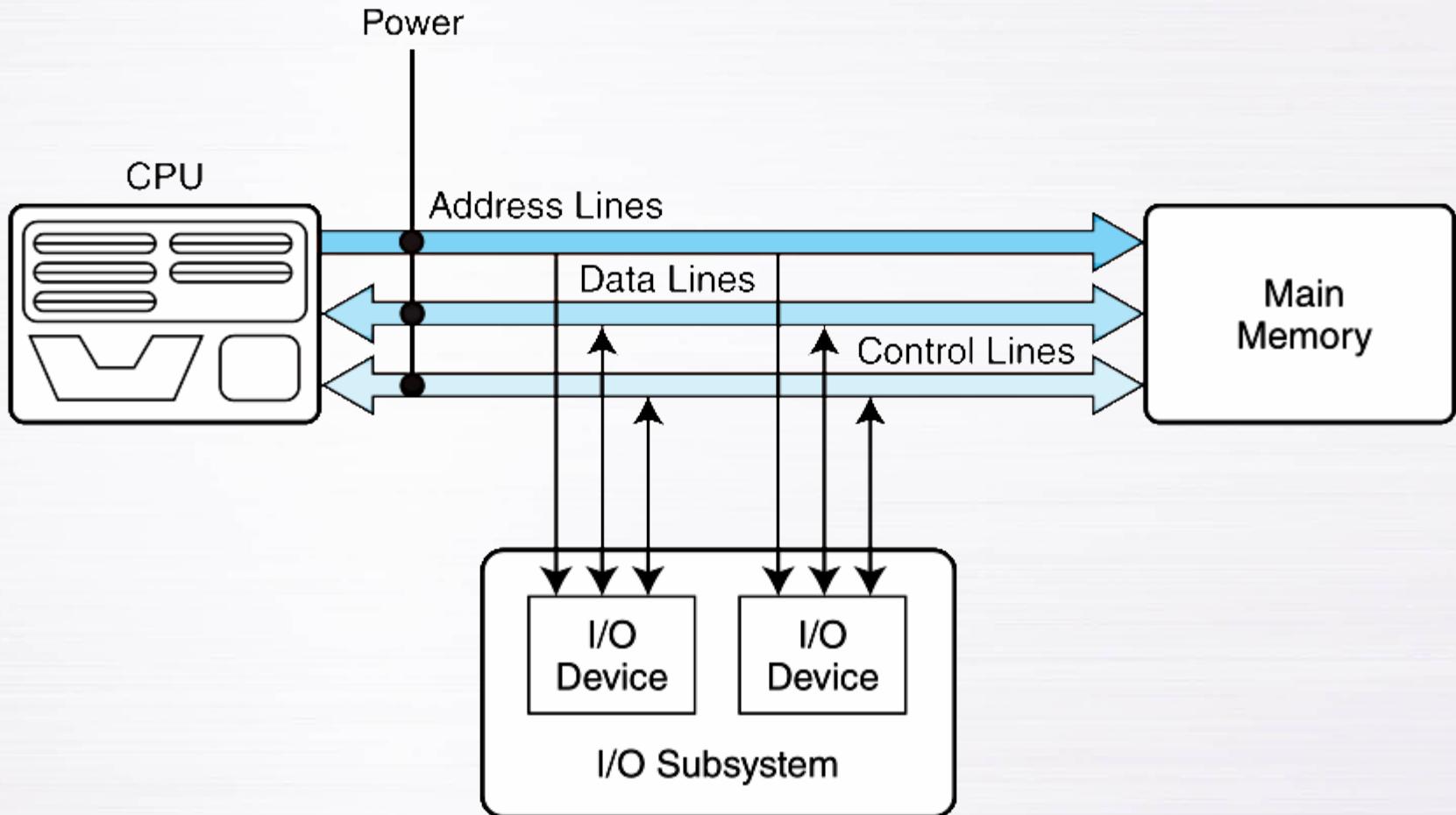


4.3 The Bus

- 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.

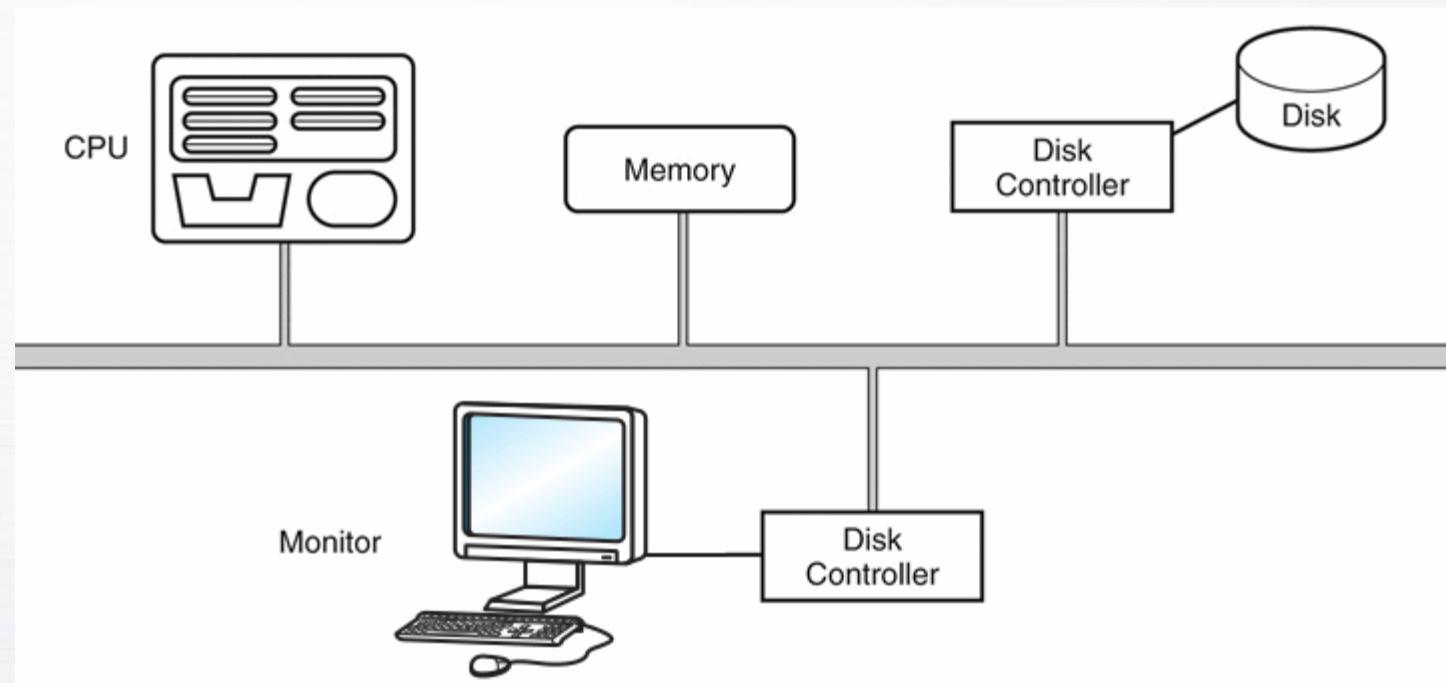
The next slide shows a model bus configuration.

4.3 The Bus



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

- 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.

4.4 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 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.

4.5 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.
- 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.

4.6 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 out 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 megabytes of 16-bit memory locations.

4.6 Memory Organization

- How does the computer access a memory location corresponding 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 bit address lines.

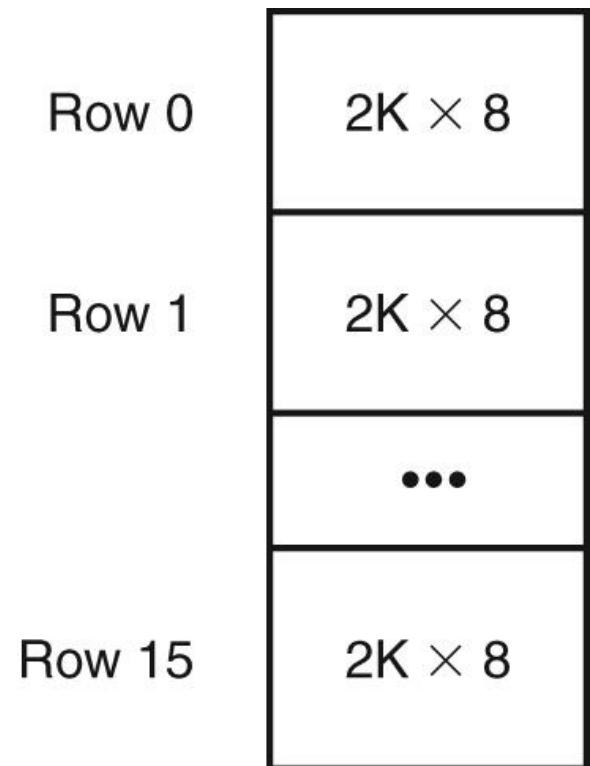
4.6 Memory Organization

- 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.

The next two slides illustrate these two ideas.

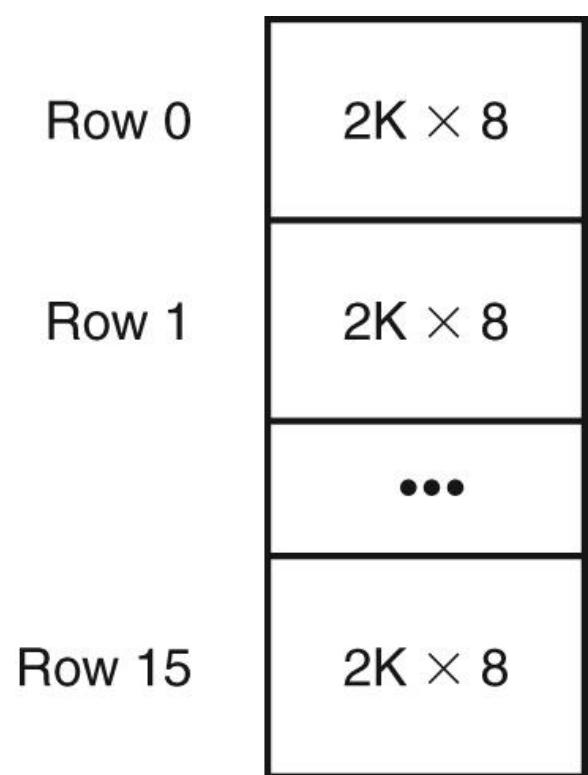
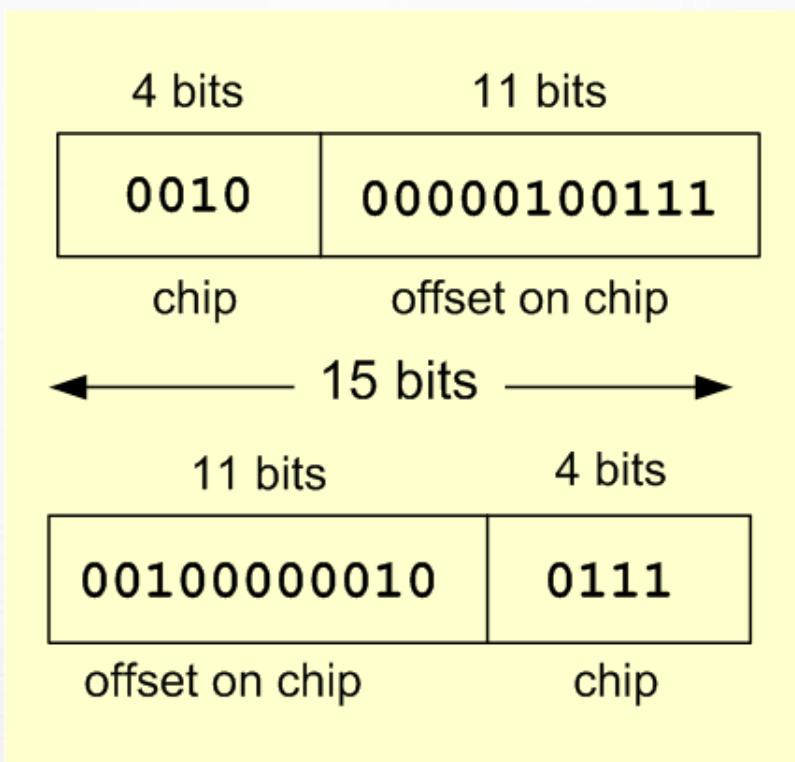
4.6 Memory Organization

- 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.

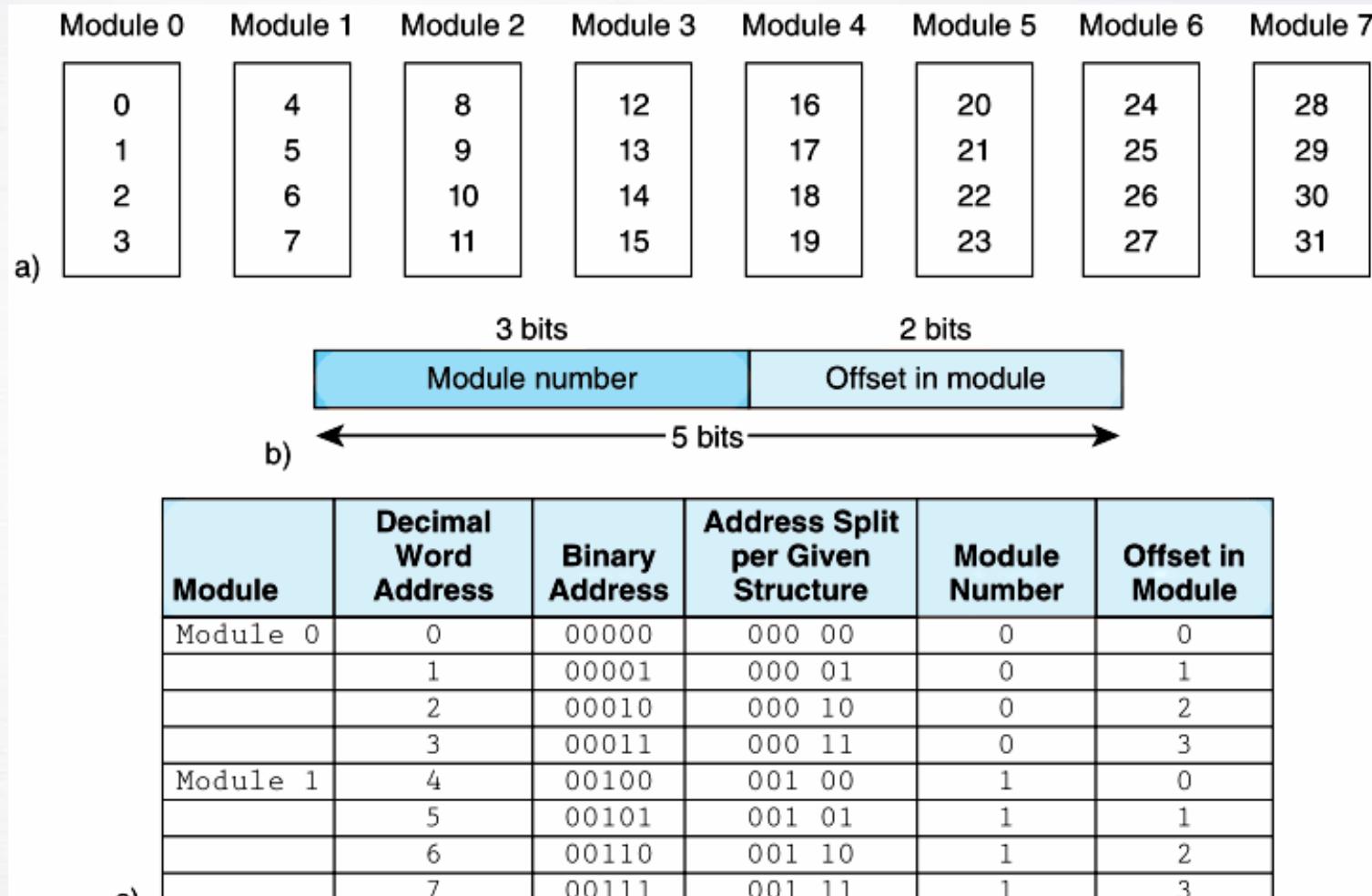


4.6 Memory Organization

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

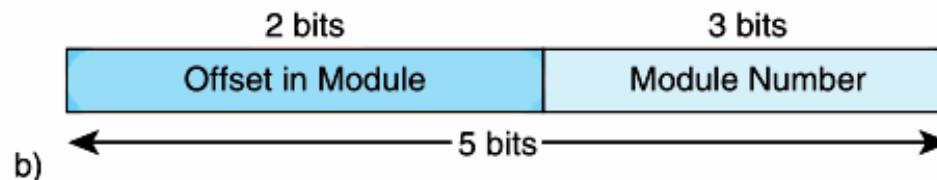
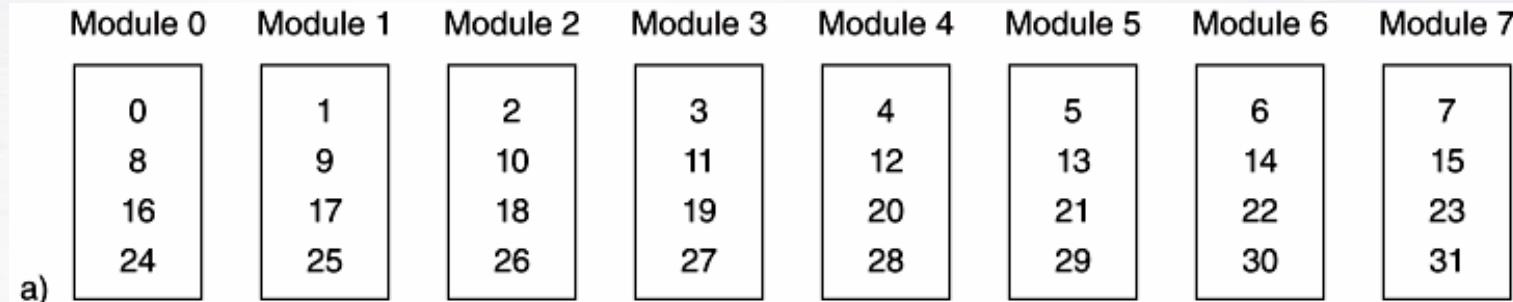


4.6 Memory Organization



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

4.6 Memory Organization



c)

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

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

4.6 Memory Organization

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



4.7 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), 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.

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.

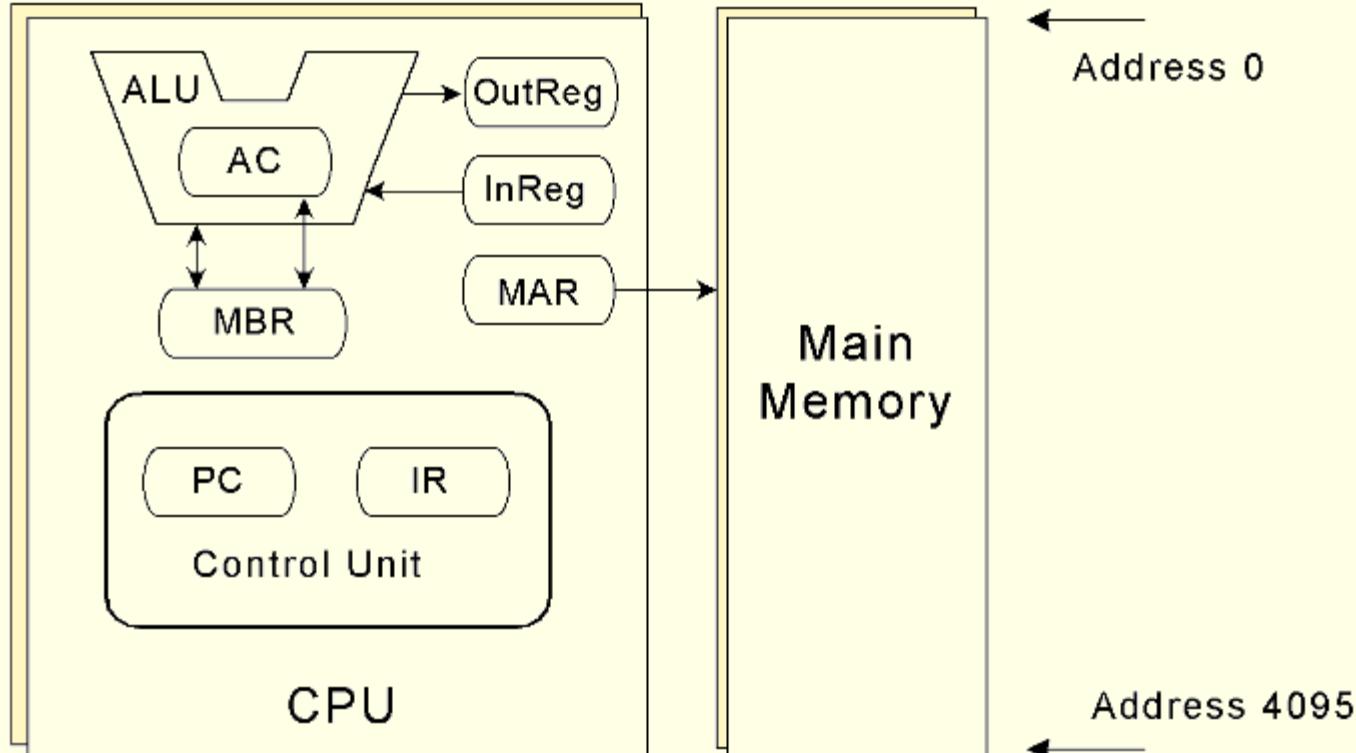
4.8 MARIE

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.

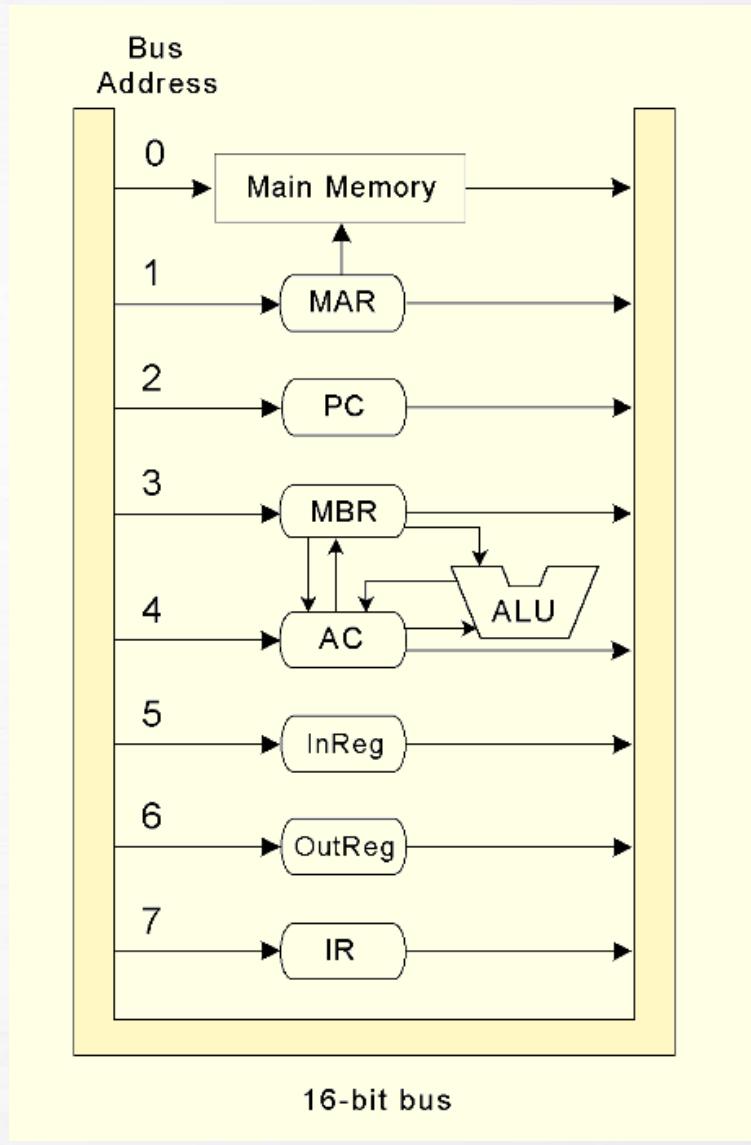


4.8 MARIE

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

4.8 MARIE

This is the MARIE data path shown graphically.

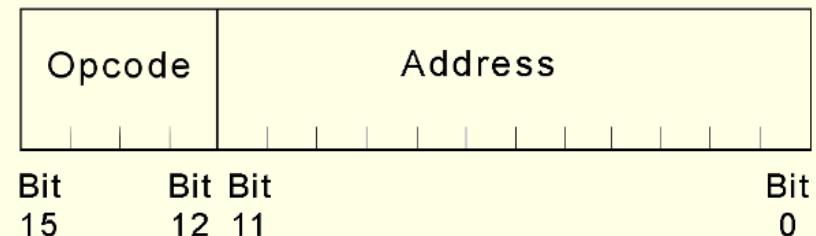


4.8 MARIE

- 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.
- We will be looking at nine instructions.

4.8 MARIE

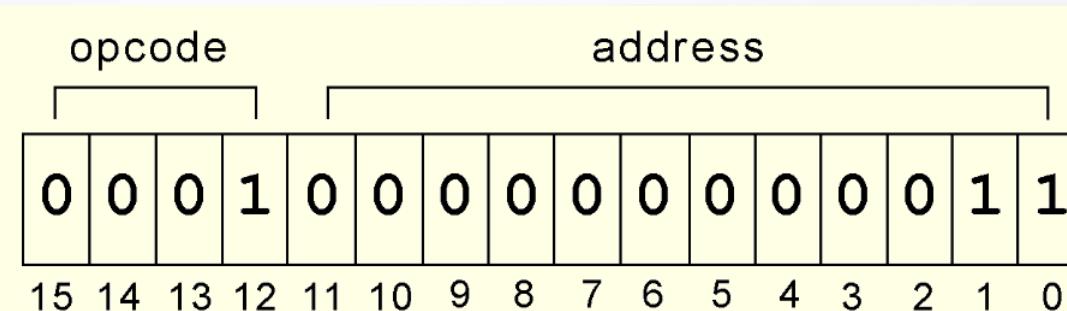
- This is the format of a MARIE instruction:
- The fundamental MARIE instructions are:



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

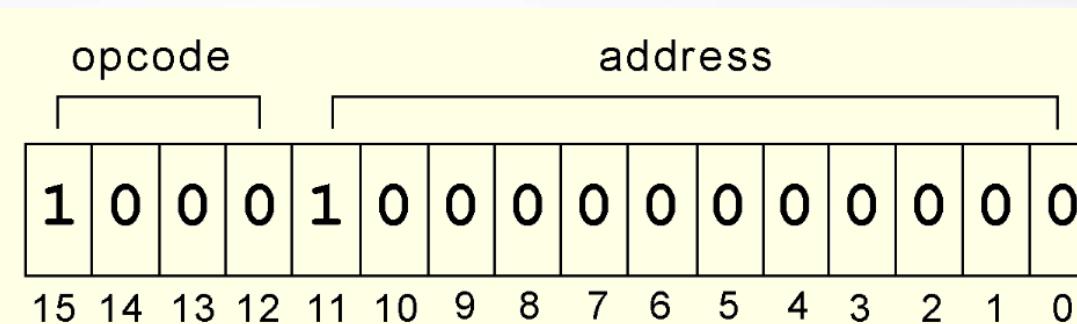
- This is a bit pattern for a **LOAD** instruction as it would appear in the IR:



- 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.

4.8 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 (RTL)*.
- In the MARIE RTL, we use the notation $M[X]$ to indicate the actual data value stored in memory location X , and \leftarrow to indicate the transfer of bytes to a register or memory location.

4.8 MARIE

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

MAR \leftarrow **X**

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

AC \leftarrow **MBR**

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

MAR \leftarrow **X**

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

AC \leftarrow **AC** + **MBR**

4.8 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:

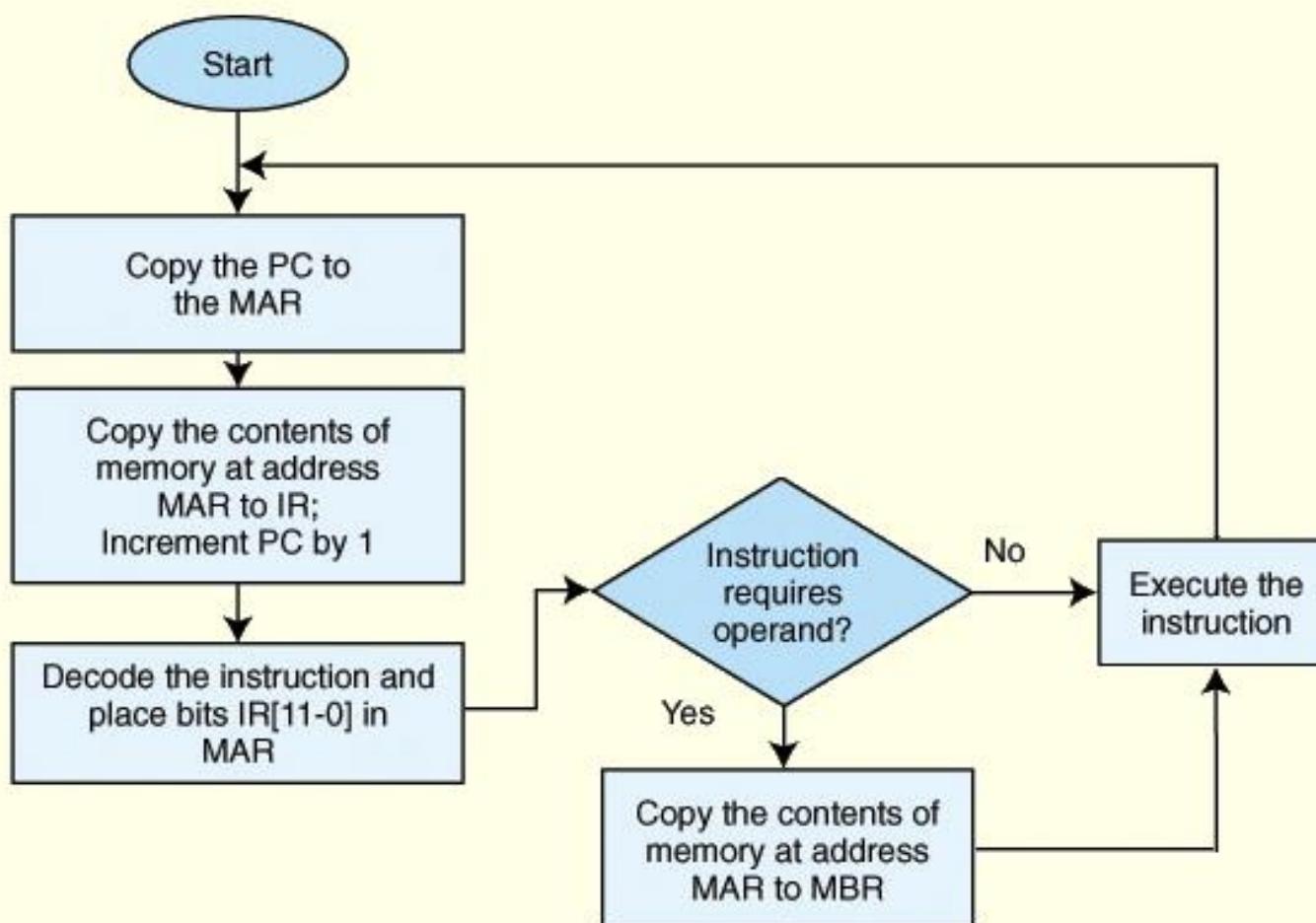
```
If IR[11 - 10] = 00 then
    If AC < 0 then PC ← PC + 1
else If IR[11 - 10] = 01 then
    If AC = 0 then PC ← PC + 1
else If IR[11 - 10] = 11 then
    If AC > 0 then PC ← PC + 1
```

4.9 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

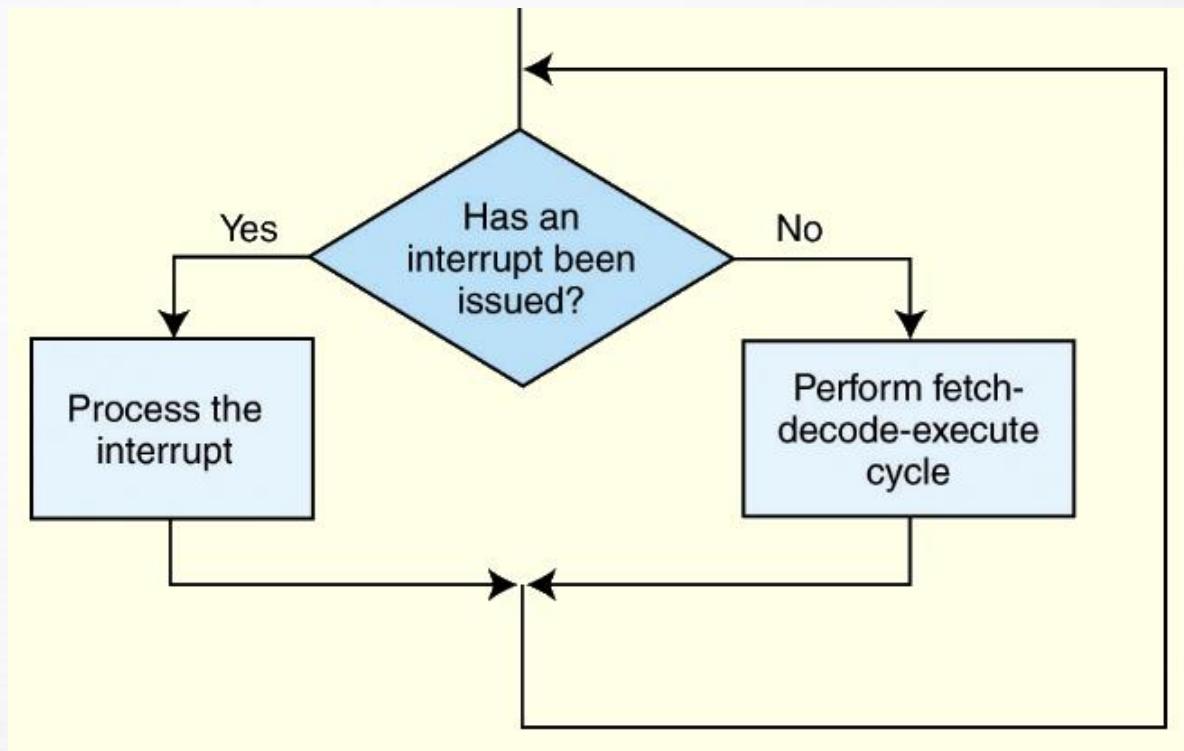


4.9 Instruction Processing

- All computers provide a way of interrupting the 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*.

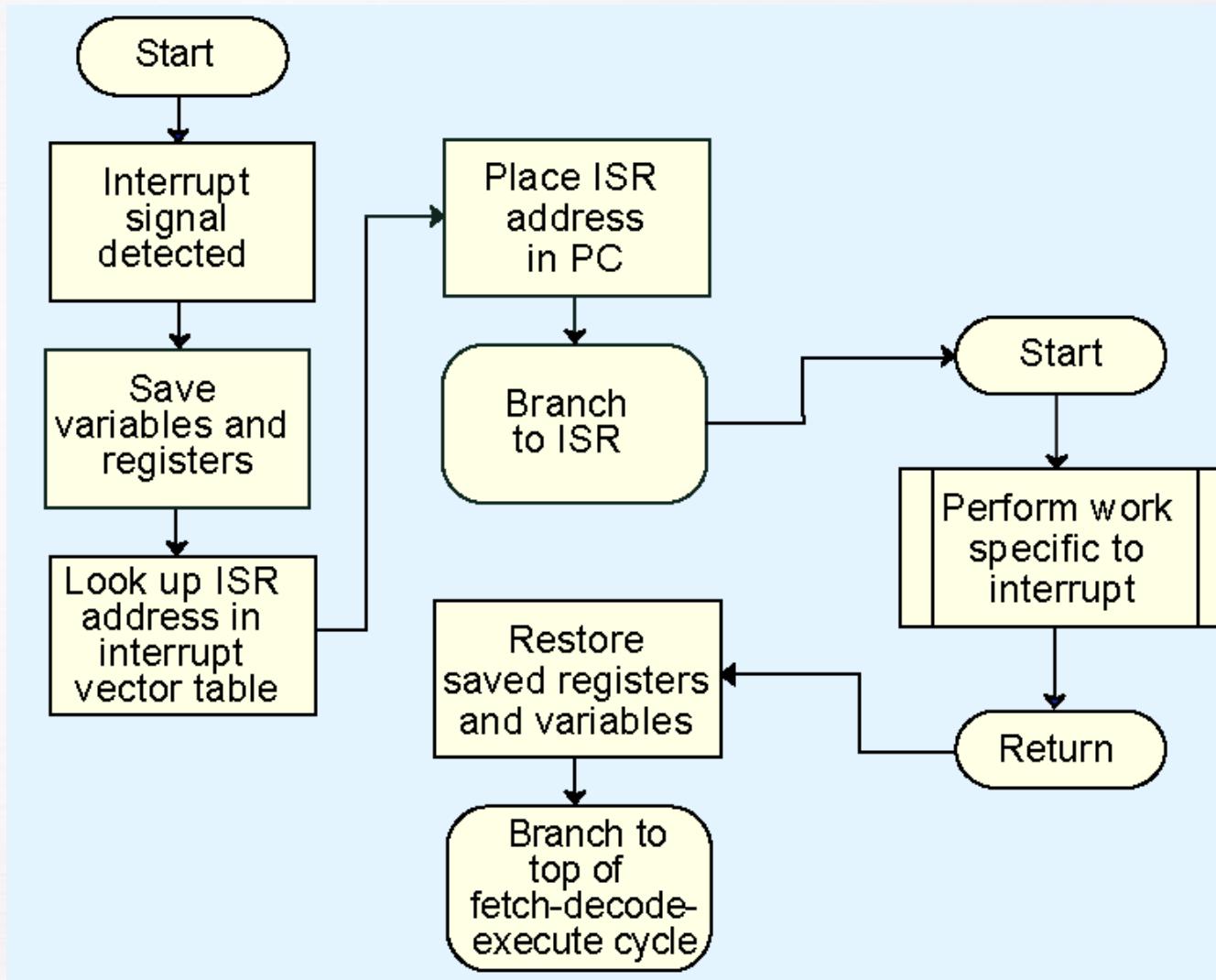
4.9 Instruction Processing

- 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.
 - We will look into this idea in greater detail in Chapter 7.
- 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
105	FFE9	111111111101001	FFE9
106	0000	0000000000000000	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 ← 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

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. With compilers, this is not usually the case.

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

X	104
Y	105
Z	106

1	X
3	Y
2	Z
7	0 0 0

4.11 A Discussion on Assemblers

- After the second pass, the assembly is complete.

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

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

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 **X** specifies the address of the operand to be loaded or stored.
- In RTL :

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
```

ADDI X

4.12 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
```

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

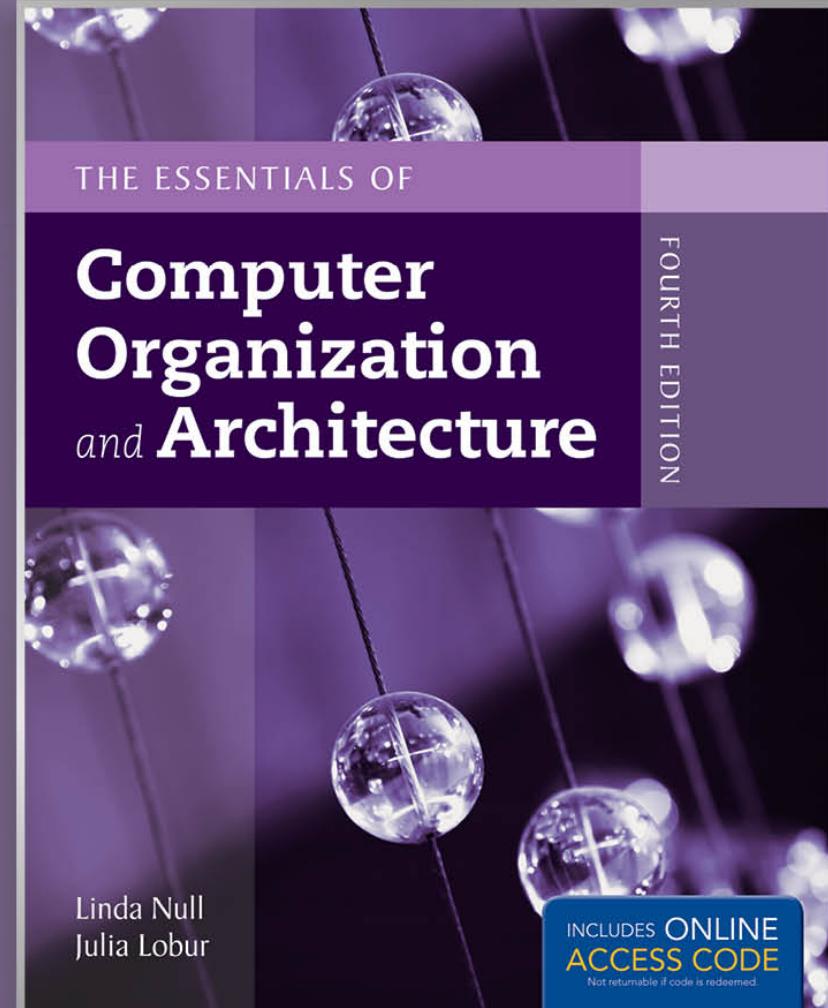
- We put our new instructions to work in the program on the following slide.

4.12 Extending Our Instruction Set

100	LOAD	Addr	10E	SKIPCOND
			000	
101	STORE	Next	10F	JUMP Loop
102	LOAD	Num	110	HALT
103	SUBT	One	111	Addr HEX 117
104	STORE	Ctr	112	Next HEX 0
105	Loop	LOAD Sum	113	Num DEC 5
106		ADDI Next	114	Sum DEC 0
			115	Ctr HEX 0
			116	One DEC 1
			117	DEC 10
107	STORE	Sum	118	DEC 15
108	LOAD	Next	119	DEC 2
109	ADD	One	11A	DEC 25
			11B	DEC 30
10A	STORE	Next		
10B	LOAD	Ctr		
10C	SUBT	One		
10D	STORE	Ctr		

Chapter 5

A Closer Look at Instruction Set Architectures



Chapter 5 Objectives

- Understand the factors involved in instruction set architecture design.
- Gain familiarity with memory addressing modes.
- Understand the concepts of instruction-level pipelining and its affect upon execution performance.

5.1 Introduction

- This chapter builds upon the ideas in Chapter 4.
- We present a detailed look at different instruction formats, operand types, and memory access methods.
- We will see the interrelation between machine organization and instruction formats.
- This leads to a deeper understanding of computer architecture in general.

5.2 Instruction Formats

Instruction sets are differentiated by the following:

- Number of bits per instruction.
- Stack-based or register-based.
- Number of explicit operands per instruction.
- Operand location.
- Types of operations.
- Type and size of operands.

5.2 Instruction Formats

Instruction set architectures are measured according to:

- Main memory space occupied by a program.
- Instruction complexity.
- Instruction length (in bits).
- Total number of instructions in the instruction set.

5.2 Instruction Formats

In designing an instruction set, consideration is given to:

- Instruction length.
 - Whether short, long, or variable.
- Number of operands.
- Number of addressable registers.
- Memory organization.
 - Whether byte- or word addressable.
- Addressing modes.
 - Choose any or all: direct, indirect or indexed.

5.2 Instruction Formats

- Byte ordering, or *endianness*, is another major architectural consideration.
- If we have a two-byte integer, the integer may be stored so that the least significant byte is followed by the most significant byte or vice versa.
 - In *little endian* machines, the least significant byte is followed by the most significant byte.
 - *Big endian* machines store the most significant byte first (at the lower address).

5.2 Instruction Formats

- As an example, suppose we have the hexadecimal number 0x12345678.
- The big endian and small endian arrangements of the bytes are shown below.

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

5.2 Instruction Formats

- A larger example: A computer uses 32-bit integers. The values 0xABCD1234, 0x00FE4321, and 0x10 would be stored sequentially in memory, starting at address 0x200 as below.

Address	Big Endian	Little Endian
0x200	AB	34
0x201	CD	12
0x202	12	CD
0x203	34	AB
0x204	00	21
0x205	FE	43
0x206	43	FE
0x207	21	00
0x208	00	10
0x209	00	00
0x20A	00	00
0x20B	10	00

5.2 Instruction Formats

- Big endian:
 - Is more natural.
 - The sign of the number can be determined by looking at the byte at address offset 0.
 - Strings and integers are stored in the same order.
- Little endian:
 - Makes it easier to place values on non-word boundaries.
 - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.

5.2 Instruction Formats

- The next consideration for architecture design concerns how the CPU will store data.
- We have three choices:
 1. A stack architecture
 2. An accumulator architecture
 3. A general purpose register architecture.
- In choosing one over the other, the tradeoffs are simplicity (and cost) of hardware design with execution speed and ease of use.

5.2 Instruction Formats

- In a stack architecture, instructions and operands are implicitly taken from the stack.
 - A stack cannot be accessed randomly.
- In an accumulator architecture, one operand of a binary operation is implicitly in the accumulator.
 - One operand is in memory, creating lots of bus traffic.
- In a general purpose register (GPR) architecture, registers can be used instead of memory.
 - Faster than accumulator architecture.
 - Efficient implementation for compilers.
 - Results in longer instructions.

5.2 Instruction Formats

- Most systems today are GPR systems.
- There are three types:
 - Memory-memory where two or three operands may be in memory.
 - Register-memory where at least one operand must be in a register.
 - Load-store where no operands may be in memory.
- The number of operands and the number of available registers has a direct affect on instruction length.

5.2 Instruction Formats

- Stack machines use one- and zero-operand instructions.
- **LOAD** and **STORE** instructions require a single memory address operand.
- Other instructions use operands from the stack implicitly.
- **PUSH** and **POP** operations involve only the stack's top element.
- Binary instructions (e.g., **ADD**, **MULT**) use the top two items on the stack.

5.2 Instruction Formats

- Stack architectures require us to think about arithmetic expressions a little differently.
- We are accustomed to writing expressions using *infix* notation, such as: $Z = X + Y$.
- Stack arithmetic requires that we use *postfix* notation: $Z = XY+$.
 - This is also called *reverse Polish notation*, (somewhat) in honor of its Polish inventor, Jan Lukasiewicz (1878 - 1956).

5.2 Instruction Formats

- The principal advantage of postfix notation is that parentheses are not used.
- For example, the infix expression,

$$Z = (X \times Y) + (W \times U) ,$$

becomes:

$$Z = X \ Y \times W \ U \times +$$

in postfix notation.

5.2 Instruction Formats

- Example: Convert the infix expression $(2+3) - 6/3$ to postfix:

2 3+ - 6/3

The sum $2 + 3$ in parentheses takes precedence; we replace the term with $2\ 3\ +$.

5.2 Instruction Formats

- Example: Convert the infix expression $(2+3) - 6/3$ to postfix:

2 3+ - 6 3/

The division operator takes next precedence; we replace $6/3$ with $6\ 3\ /$.

5.2 Instruction Formats

- Example: Convert the infix expression $(2+3) - 6/3$ to postfix:

2 3+ 6 3/ -

The quotient $6/3$ is subtracted from the sum of $2 + 3$, so we move the $-$ operator to the end.

5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression $2\ 3\ +\ 6\ 3\ /\ -$:

Scanning the expression from left to right, push operands onto the stack, until an operator is found

2	3	+	6	3	/	-
---	---	---	---	---	---	---



3
2

5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression $2\ 3\ +\ 6\ 3\ /\ -$:

Pop the two operands and carry out the operation indicated by the operator.
Push the result back on the stack.

2	3	+	6	3	/	-
---	---	---	---	---	---	---



5

5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression $2\ 3\ +\ 6\ 3\ /\ -$:

2	3	+	6	3	/	-
---	---	---	---	---	---	---

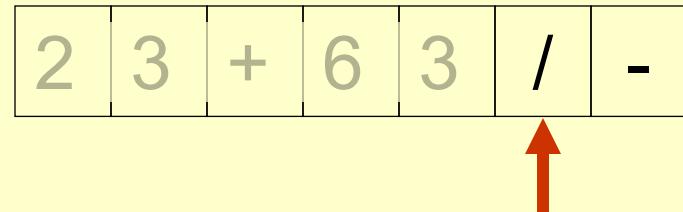


Push operands until another operator is found.

3
6
5

5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression $2\ 3\ +\ 6\ 3\ /\ -$:



Carry out the operation and push the result.

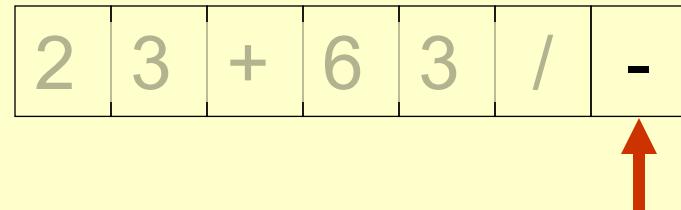


5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression $2\ 3\ +\ 6\ 3\ /\ -$:

Finding another operator,
carry out the operation and
push the result.

The answer is at the top of
the stack.



5.2 Instruction Formats

- Let's see how to evaluate an infix expression using different instruction formats.
- With a three-address ISA, (e.g.,mainframes), the infix expression,

$Z = X \times Y + W \times U$

might look like this:

```
MULT R1,X,Y  
MULT R2,W,U  
ADD Z,R1,R2
```

5.2 Instruction Formats

- In a two-address ISA, (e.g., Intel, Motorola), the infix expression,

$$Z = X \times Y + W \times U$$

might look like this:

```
LOAD R1,X  
MULT R1,Y  
LOAD R2,W  
MULT R2,U  
ADD R1,R2  
STORE Z,R1
```

Note: One-address ISAs usually require one operand to be a register.

5.2 Instruction Formats

- In a one-address ISA, like MARIE, the infix expression,

$$Z = X \times Y + W \times U$$

looks like this:

```
LOAD X
MULT Y
STORE TEMP
LOAD W
MULT U
ADD TEMP
STORE Z
```

5.2 Instruction Formats

- In a stack ISA, the postfix expression,

Z = X Y × W U × +

might look like this:

PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
PUSH Z

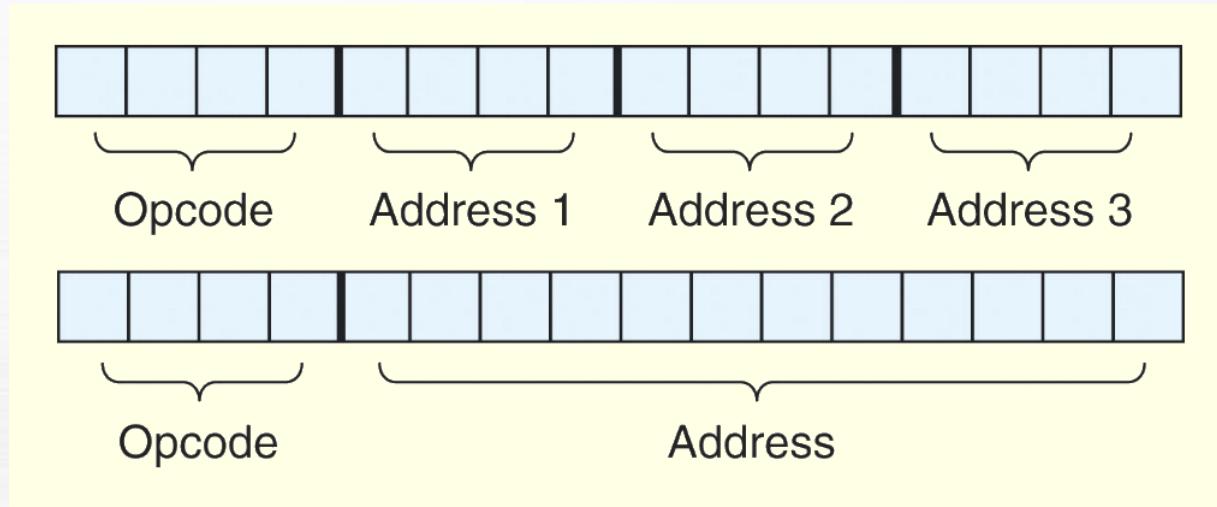
Would this program require more execution time than the corresponding (shorter) program that we saw in the 3-address ISA?

5.2 Instruction Formats

- We have seen how instruction length is affected by the number of operands supported by the ISA.
- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as `HALT`, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use expanding opcodes.

5.2 Instruction Formats

- A system has 16 registers and 4K of memory.
- We need 4 bits to access one of the registers. We also need 12 bits for a memory address.
- If the system is to have 16-bit instructions, we have two choices for our instructions:



5.2 Instruction Formats

- If we allow the length of the opcode to vary, we could create a very rich instruction set:

0000 R1 R2 R3	}	15 three-address codes
...		
1110 R1 R2 R3	}	14 two-address codes
1111 - escape opcode		
1111 0000 R1 R2	}	14 two-address codes
...		
1111 1101 R1 R2	}	14 two-address codes
1111 1110 - escape opcode		
1111 1110 0000 R1	}	31 one-address codes
...		
1111 1111 1110 R1	}	31 one-address codes
1111 1111 1111 - escape opcode		
1111 1111 1111 0000	}	16 zero-address codes
...		
1111 1111 1111 1111		

5.2 Instruction Formats

- Example: Given 8-bit instructions, is it possible to allow the following to be encoded?
 - 3 instructions with two 3-bit operands.
 - 2 instructions with one 4-bit operand.
 - 4 instructions with one 3-bit operand.

We need:

$$3 \times 2^3 = 192 \text{ bits for the 3-bit operands}$$

$$2 \times 2^4 = 32 \text{ bits for the 4-bit operands}$$

$$4 \times 2^3 = 32 \text{ bits for the 3-bit operands.}$$

Total: 256 bits.

5.2 Instruction Formats

- With a total of 256 bits required, we can exactly encode our instruction set in 8 bits!

We need:

$$3 \times 2^3 = 192 \text{ bits for the 3-bit operands}$$

$$2 \times 2^4 = 32 \text{ bits for the 4-bit operands}$$

$$4 \times 2^3 = 32 \text{ bits for the 3-bit operands.}$$

Total: 256 bits.

One such encoding is shown on the next slide.

5.2 Instruction Formats

00 xxx xxx	}	3 instructions with two 3-bit operands
01 xxx xxx		
10 xxx xxx		
11 - escape opcode	}	2 instructions with one 4-bit operand
1100 xxxx		
1101 xxxx		
1110 - escape opcode	}	4 instructions with one 3-bit operand
1111 - escape opcode		
11100 xxx		
11101 xxx		
11110 xxx		
11111 xxx		

5.3 Instruction types

Instructions fall into several broad categories that you should be familiar with:

- Data movement.
- Arithmetic.
- Boolean.
- Bit manipulation.
- I/O.
- Control transfer.
- Special purpose.

Can you think of some examples of each of these?

5.4 Addressing

- Addressing modes specify where an operand is located.
- They can specify a constant, a register, or a memory location.
- The actual location of an operand is its *effective address*.
- Certain addressing modes allow us to determine the address of an operand dynamically.

5.4 Addressing

- *Immediate addressing* is where the data is part of the instruction.
- *Direct addressing* is where the address of the data is given in the instruction.
- *Register addressing* is where the data is located in a register.
- *Indirect addressing* gives the address of the address of the data in the instruction.
- *Register indirect addressing* uses a register to store the address of the address of the data.

5.4 Addressing

- *Indexed addressing* uses a register (implicitly or explicitly) as an offset, which is added to the address in the operand to determine the effective address of the data.
- *Based addressing* is similar except that a base register is used instead of an index register.
- The difference between these two is that an index register holds an offset relative to the address given in the instruction, a base register holds a base address where the address field represents a displacement from this base.

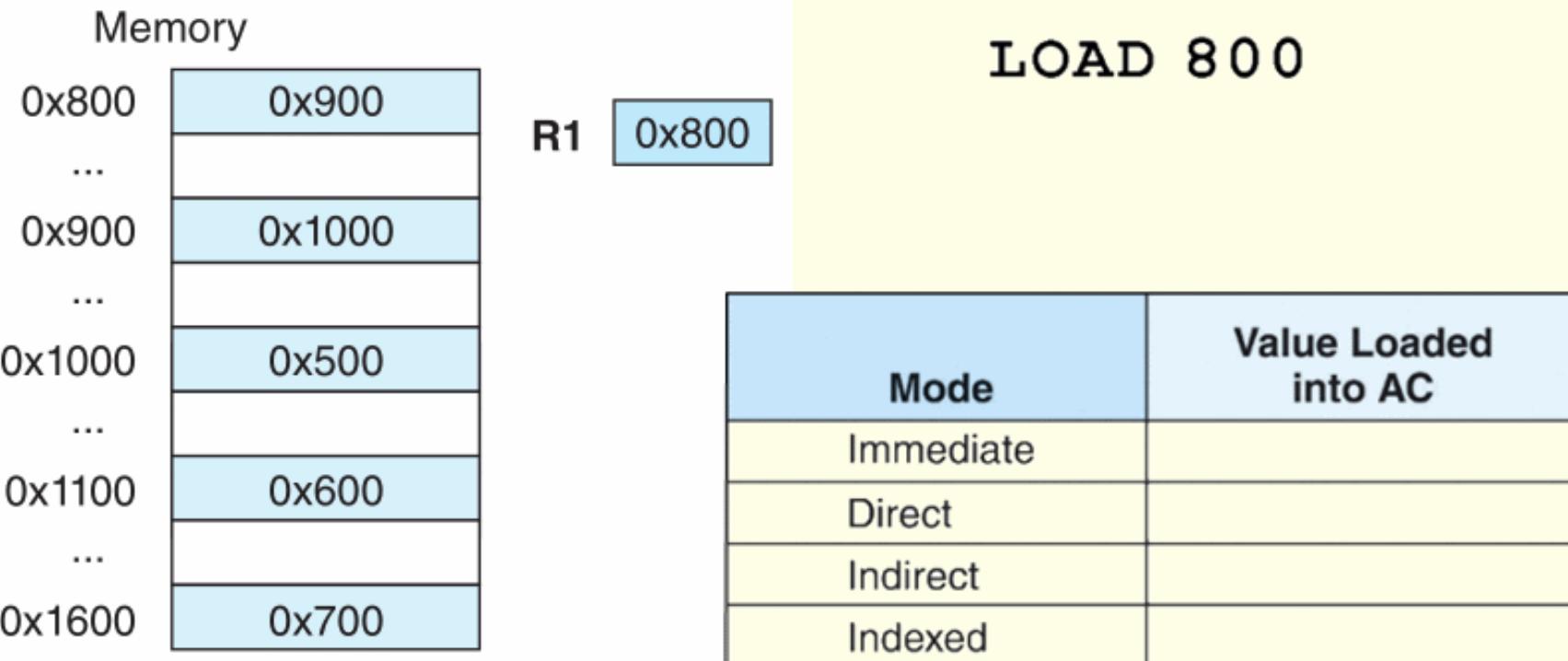
5.4 Addressing

- In *stack addressing* the operand is assumed to be on top of the stack.
- There are many variations to these addressing modes including:
 - Indirect indexed.
 - Base/offset.
 - Self-relative
 - Auto increment - decrement.
- We won't cover these in detail.

Let's look at an example of the principal addressing modes.

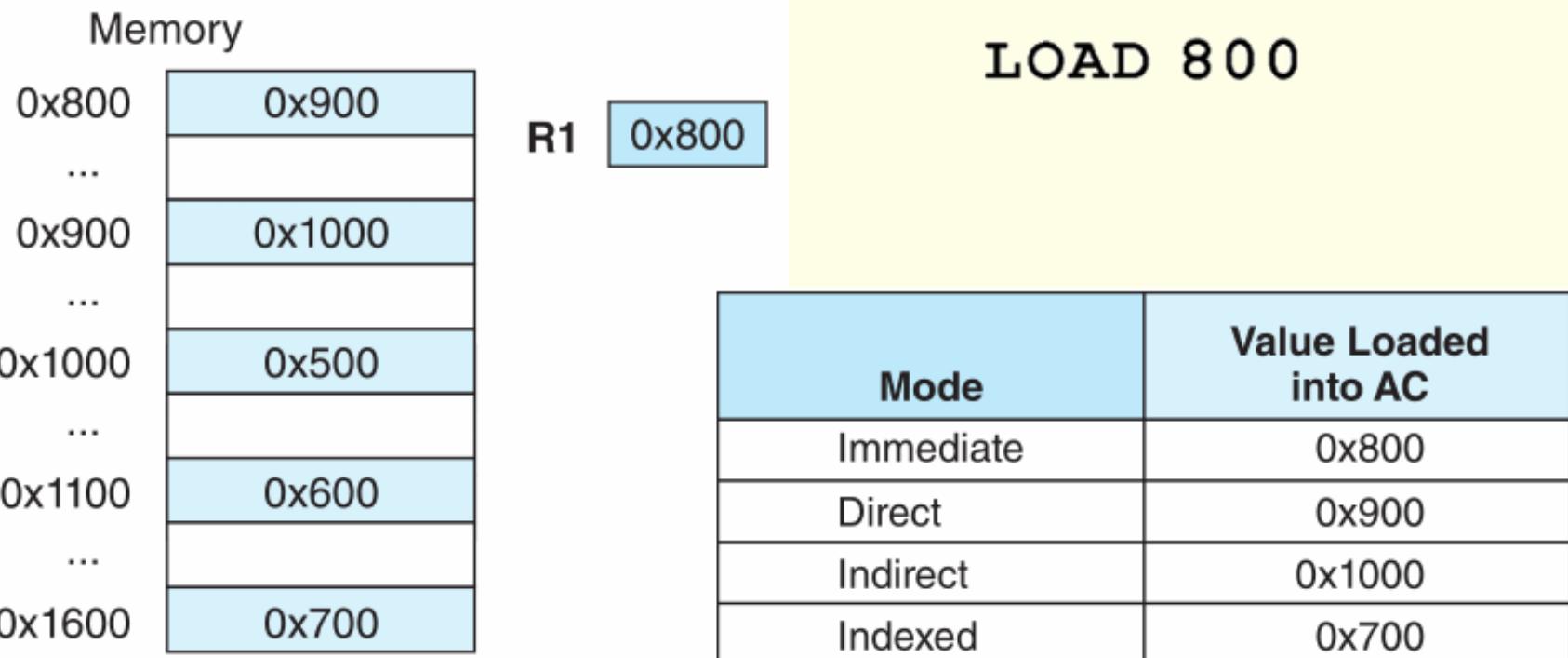
5.4 Addressing

- For the instruction shown, what value is loaded into the accumulator for each addressing mode?



5.4 Addressing

- For the instruction shown, what value is loaded into the accumulator for each addressing mode?



5.5 Instruction Pipelining

- Some CPUs divide the fetch-decode-execute cycle into smaller steps.
- These smaller steps can often be executed in parallel to increase throughput.
- Such parallel execution is called *instruction pipelining*.
- Instruction pipelining provides for *instruction level parallelism (ILP)*

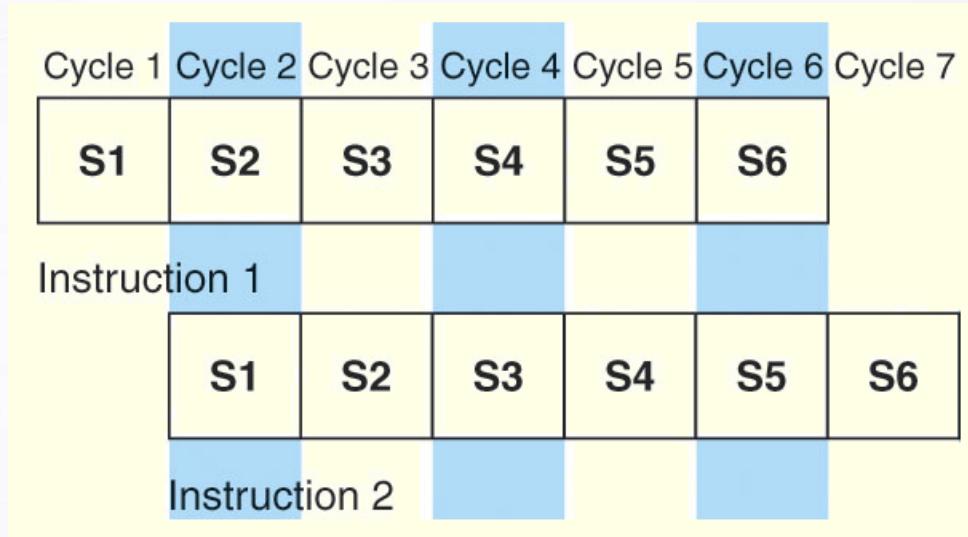
The next slide shows an example of instruction pipelining.

5.5 Instruction Pipelining

- Suppose a fetch-decode-execute cycle were broken into the following smaller steps:
 1. Fetch instruction.
 2. Decode opcode.
 3. Calculate effective address of operands.
 4. Fetch operands.
 5. Execute instruction.
 6. Store result.
- Suppose we have a six-stage pipeline. S1 fetches the instruction, S2 decodes it, S3 determines the address of the operands, S4 fetches them, S5 executes the instruction, and S6 stores the result.

5.5 Instruction Pipelining

- For every clock cycle, one small step is carried out, and the stages are overlapped.



S1. Fetch instruction.
S2. Decode opcode.
S3. Calculate effective address of operands.

S4. Fetch operands.
S5. Execute.
S6. Store result.

5.5 Instruction Pipelining

- The theoretical speedup offered by a pipeline can be determined as follows:

Let t_p be the time per stage. Each instruction represents a task, T , in the pipeline.

The first task (instruction) requires $k \times t_p$ time to complete in a k -stage pipeline. The remaining $(n - 1)$ tasks emerge from the pipeline one per cycle. So the total time to complete the remaining tasks is $(n - 1)t_p$.

Thus, to complete n tasks using a k -stage pipeline requires:

$$(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p.$$

5.5 Instruction Pipelining

- If we take the time required to complete n tasks without a pipeline and divide it by the time it takes to complete n tasks using a pipeline, we find:

$$\text{Speedup } S = \frac{n t_n}{(k + n - 1) t_p}$$

- If we take the limit as n approaches infinity, $(k + n - 1)$ approaches n , which results in a theoretical speedup of:

$$\text{Speedup } S = \frac{k t_p}{t_p} = k$$

5.5 Instruction Pipelining

- Our neat equations take a number of things for granted.
- First, we have to assume that the architecture supports fetching instructions and data in parallel.
- Second, we assume that the pipeline can be kept filled at all times. This is not always the case. Pipeline *hazards* arise that cause pipeline conflicts and stalls.

5.5 Instruction Pipelining

- An instruction pipeline may stall, or be flushed for any of the following reasons:
 - Resource conflicts.
 - Data dependencies.
 - Conditional branching.
- Measures can be taken at the software level as well as at the hardware level to reduce the effects of these hazards, but they cannot be totally eliminated.

5.6 Real-World Examples of ISAs

- We return briefly to the Intel and MIPS architectures from the last chapter, using some of the ideas introduced in this chapter.
- Intel introduced pipelining to their processor line with its Pentium chip.
- The first Pentium had two five-stage pipelines. Each subsequent Pentium processor had a longer pipeline than its predecessor with the Pentium IV having a 24-stage pipeline.
- The Itanium (IA-64) has only a 10-stage pipeline.

5.6 Real-World Examples of ISAs

- Intel processors support a wide array of addressing modes.
- The original 8086 provided 17 ways to address memory, most of them variants on the methods presented in this chapter.
- Owing to their need for backward compatibility, the Pentium chips also support these 17 addressing modes.
- The Itanium, having a RISC core, supports only one: register indirect addressing with optional post increment.

5.6 Real-World Examples of ISAs

- MIPS was an acronym for *Microprocessor Without Interlocked Pipeline Stages*.
- The architecture is little endian and word-addressable with three-address, fixed-length instructions.
- Like Intel, the pipeline size of the MIPS processors has grown: The R2000 and R3000 have five-stage pipelines.; the R4000 and R4400 have 8-stage pipelines.

5.6 Real-World Examples of ISAs

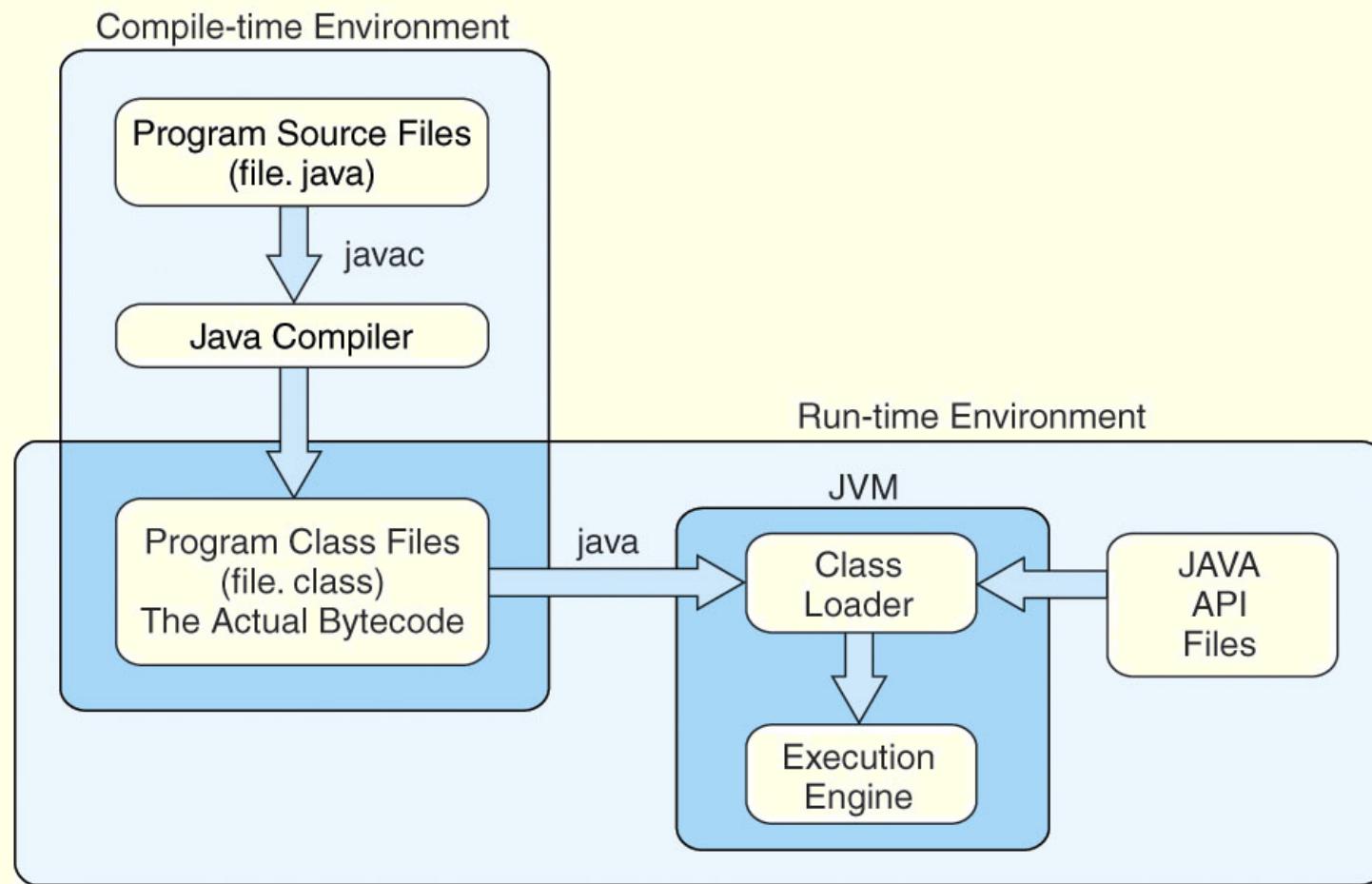
- The R10000 has three pipelines: A five-stage pipeline for integer instructions, a seven-stage pipeline for floating-point instructions, and a six-state pipeline for **LOAD/STORE** instructions.
- In all MIPS ISAs, only the **LOAD** and **STORE** instructions can access memory.
- The ISA uses only base addressing mode.
- The assembler accommodates programmers who need to use immediate, register, direct, indirect register, base, or indexed addressing modes.

5.6 Real-World Examples of ISAs

- The Java programming language is an interpreted language that runs in a software machine called the *Java Virtual Machine* (JVM).
- A JVM is written in a native language for a wide array of processors, including MIPS and Intel.
- Like a real machine, the JVM has an ISA all of its own, called *bytecode*. This ISA was designed to be compatible with the architecture of any machine on which the JVM is running.

The next slide shows how the pieces fit together.

5.6 Real-World Examples of ISAs



5.6 Real-World Examples of ISAs

- Java bytecode is a stack-based language.
- Most instructions are zero address instructions.
- The JVM has four registers that provide access to five regions of main memory.
- All references to memory are offsets from these registers. Java uses no pointers or absolute memory references.
- Java was designed for platform interoperability, not performance!

5.6 Real-World Examples of ISAs

- You may not have heard of ARM but most likely use an ARM processor every day. It is the most widely used 32-bit instruction architecture:
 - 95%+ of smartphones,
 - 80%+ of digital cameras
 - 40%+ of all digital television sets
- Founded in 1990, by Apple and others, ARM (Advanced RISC Machine) is now a British firm, ARM Holdings.
- ARM Holdings does not manufacture these processors; it sells licenses to manufacture.

5.6 Real-World Examples of ISAs

- ARM is a load/store architecture : all data processing must be performed on values in registers, not in memory.
- It uses fixed-length, three-operand instructions and simple addressing modes
- ARM processors have a minimum of a three-stage pipeline (consisting of fetch, decode, and execute);
 - Newer ARM processors have deeper pipelines (more stages). Some ARM8 implementations have 13-stage integer pipelines

5.6 Real-World Examples of ISAs

- ARM has 37 total registers but their visibility depends on the processor mode.
- ARM allows multiple register transfers.
 - It can simultaneously load or store any subset of the 16 general-purpose registers from/to sequential memory addresses.
- Control flow instructions include unconditional and conditional branching and procedure calls
- Most ARM instructions execute in a single cycle, provided there are no pipeline hazards or memory accesses.

Chapter 5 Conclusion

- ISAs are distinguished according to their bits per instruction, number of operands per instruction, operand location and types and sizes of operands.
- Endianness as another major architectural consideration.
- CPU can store data based on
 1. A stack architecture
 2. An accumulator architecture
 3. A general purpose register architecture.

Chapter 5 Conclusion

- Instructions can be fixed length or variable length.
- To enrich the instruction set for a fixed length instruction set, expanding opcodes can be used.
- The addressing mode of an ISA is also another important factor. We looked at:
 - Immediate
 - Register
 - Indirect
 - Based
 - Direct
 - Register Indirect
 - Indexed
 - Stack

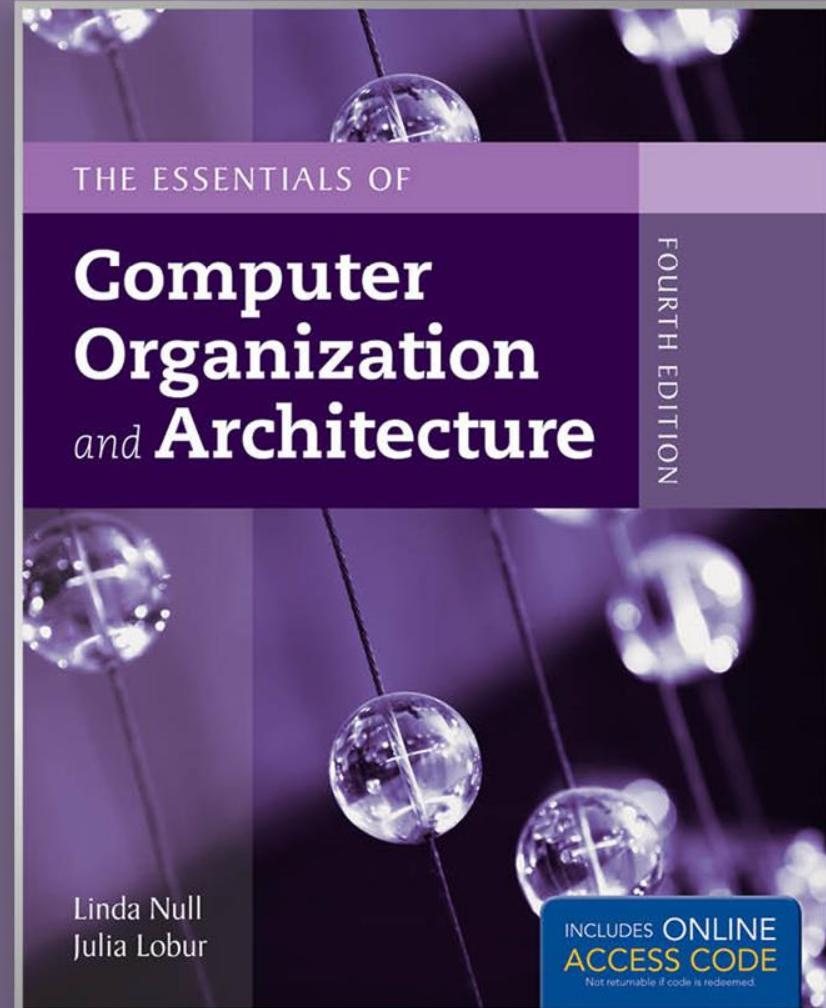
Chapter 5 Conclusion

- A k -stage pipeline can theoretically produce execution speedup of k as compared to a non-pipelined machine.
- Pipeline hazards such as resource conflicts and conditional branching prevents this speedup from being achieved in practice.
- The Intel, MIPS, JVM, and ARM architectures provide good examples of the concepts presented in this chapter.

Chapter 3

Special Section

Focus on Karnaugh Maps



3A.1 Introduction

- Simplification of Boolean functions leads to simpler (and usually faster) digital circuits.
- Simplifying Boolean functions using identities is time-consuming and error-prone.
 - No guarantee that an expression is as simple as possible.
- In todays lecture we will cover a systematic method for reducing Boolean expressions.

3A.1 Introduction

- In 1953, Maurice Karnaugh was a telecommunications engineer at Bell Labs.
- While exploring the new field of digital logic and its application to the design of telephone circuits, he invented a graphical way of visualizing and then simplifying Boolean expressions.
- This graphical representation, now known as a Karnaugh map, or Kmap, is named in his honor.



3A.2 Description of Kmaps and Terminology

- A Kmap is a matrix consisting of rows and columns that represent the output values of a Boolean function.
- The output values placed in each cell are derived from the minterms of a Boolean function.
- A *minterm* is a **product** term that contains all of the function's variables exactly once, either complemented or not complemented.

3A.2 Description of Kmaps and Terminology

- For example, the minterms for a function having the inputs x and y are xy , $x'y$, xy' , and $x'y'$.
- Consider the Boolean function, $F(x, y) = xy + xy'$
- Its minterms are:

Minterm	X	Y
$x'y'$	0	0
$x'y$	0	1
xy'	1	0
xy	1	1

3A.2 Description of Kmaps and Terminology

- Similarly, a function having three inputs, has the *minterms* that are shown in this table.

Minterm	X	Y	Z
$X'Y'Z'$	0	0	0
$X'Y'Z$	0	0	1
$X'Y Z'$	0	1	0
$X'Y Z$	0	1	1
$X Y'Z'$	1	0	0
$X Y'Z$	1	0	1
$X Y Z'$	1	1	0
$X Y Z$	1	1	1

3A.2 Description of Kmaps and Terminology

- A Kmap has a cell for each minterm.
- This means that it has a cell for each line for the truth table of a function.
- The truth table for the function $F(x,y) = xy$ is shown at the right along with its corresponding Kmap.

		$F(x, y) = xy$
x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

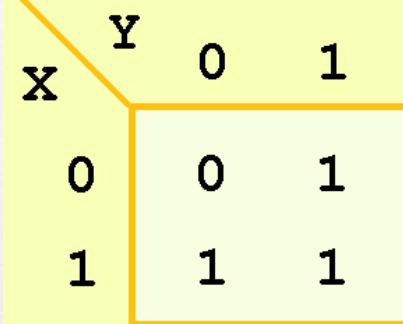
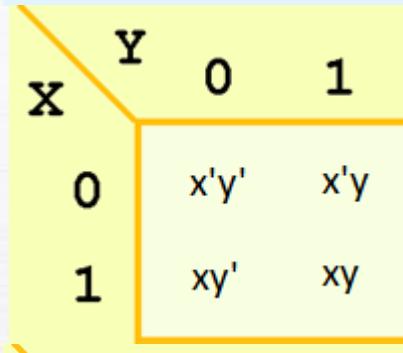
x	y	0	1
0	0	0	0
1	0	0	1

3A.2 Description of Kmaps and Terminology

- As another example, we give the truth table and KMap for the function, $F(x,y) = x + y$ at the right.
- This function is equivalent to the OR of all of the minterms that have a value of 1. Thus:

$$F(x, y) = x + y = x'y' + xy' + xy$$

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1



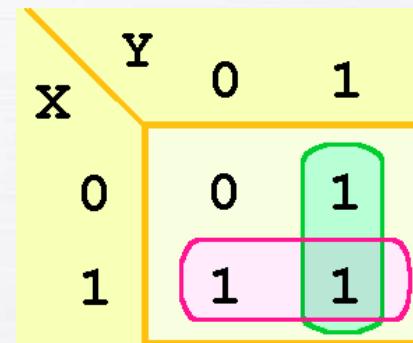
3A.3 Kmap Simplification for Two Variables

- Of course, the minterm function that we derived from our Kmap was not in simplest terms.
 - That's what we started with in this example.
- We can, however, reduce our complicated expression to its simplest terms by finding adjacent 1s in the Kmap that can be collected into groups that are **powers of two**.
- In our example, we have two such groups.
 - Can you find them?

x \ y	0	1
0	0	1
1	1	1

3A.3 Kmap Simplification for Two Variables

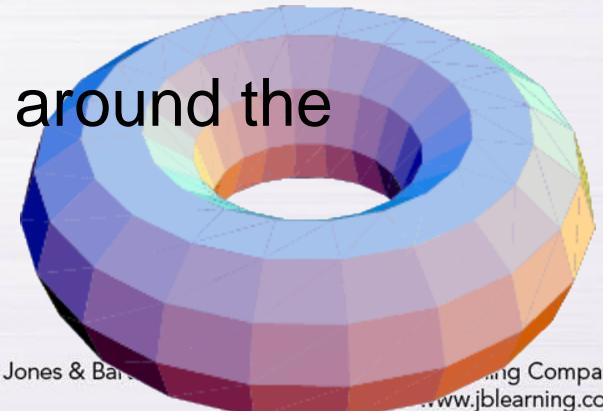
- The best way of selecting two groups of 1s from our simple Kmap is shown below.
- We see that both groups are powers of two and that the groups overlap.
- The next slide gives guidance for selecting Kmap groups.



3A.3 Kmap Simplification for Two Variables

The rules of Kmap simplification are:

- Groupings can contain **only** 1s; **no** 0s.
- Groups can be formed only at right angles; diagonal groups are not allowed.
- The number of 1s in a group must be a power of 2 – even if it contains a single 1.
- The groups must be **made as large as possible**.
- Groups can **overlap** and **wrap** around the sides of the Kmap.
- All 1's must be covered.



3A.3 Kmap Simplification for Three Variables

- A Kmap for three variables is constructed as shown in the diagram below.
- We have placed each minterm in the cell that will hold its value.
 - Notice that the values for the yz combination at the top of the matrix form a pattern that is not a normal binary sequence.

		yz	00	01	11	10
		x	00	01	11	10
		0	$x'y'z'$	$x'y'z$	$x'y z$	$x'y z'$
		1	$x'y'z'$	$x'y'z$	$x'y z$	$x'y z'$

3A.3 Kmap Simplification for Three Variables

- Thus, the first row of the Kmap contains all minterms where x has a value of zero.
- The first column contains all minterms where y and z both have a value of zero.

		yz	00	01	11	10
		x	00	01	11	10
0		$x'y'z'$	$x'y'z$	xyz	$x'y z'$	$x'y z'$
1		$xy'z'$	$x y'z$	$xy z$	$x y z'$	$x y z'$

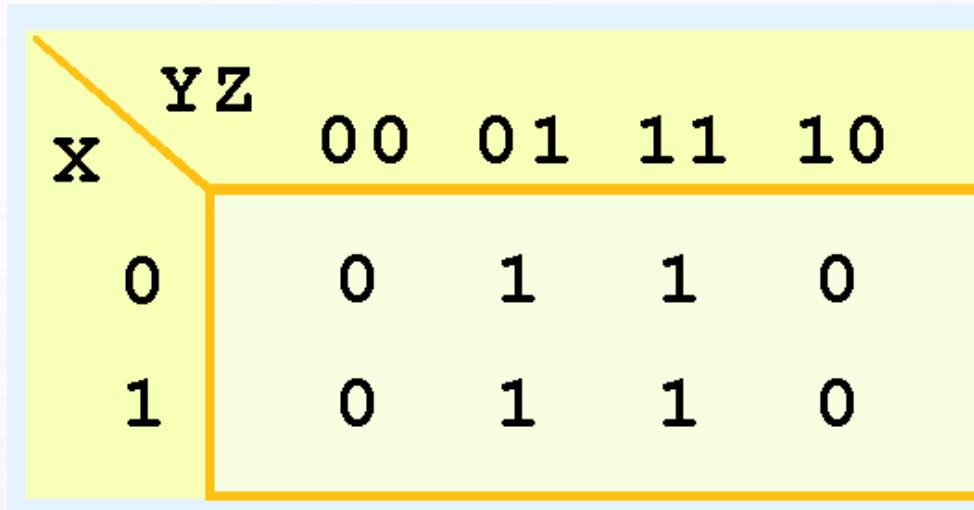
3A.3 Kmap Simplification for Three Variables

- Consider the function:

$$F(X, Y, Z) = X' Y' Z + X' Y Z + X Y' Z + X Y Z$$

- Its Kmap is given below.

- What is the largest group of 1s that is a power of 2?



3A.3 Kmap Simplification for Three Variables

- This grouping tells us that changes in the variables x and y have no influence upon the value of the function: They are irrelevant.
- This means that the function,
$$F(X, Y, Z) = X' Y' Z + X' Y Z + X Y' Z + X Y Z$$
reduces to $F(x) = z$.

You could verify
this reduction
with identities or
a truth table.

		Y	Z	00	01	11	10
		x	0	0	1	1	0
x	0	0	0	1	1	0	0
	1	0	1	1	1	0	0

3A.3 Kmap Simplification for Three Variables

- Systematic way to tell us that x and y aren't needed.
 - x y z
 - 0 0 1
 - 1 0 1
 - 0 1 1
 - 1 1 1
 - If Z=1 then the entry is 1 regardless of (x,y)

3A.3 Kmap Simplification for Three Variables

- Now for a more complicated Kmap. Consider the function:

$$F(X, Y, Z) = X'Y'Z' + X'Y'Z + X'YZ + X'YZ' + XY'Z' + XYZ'$$

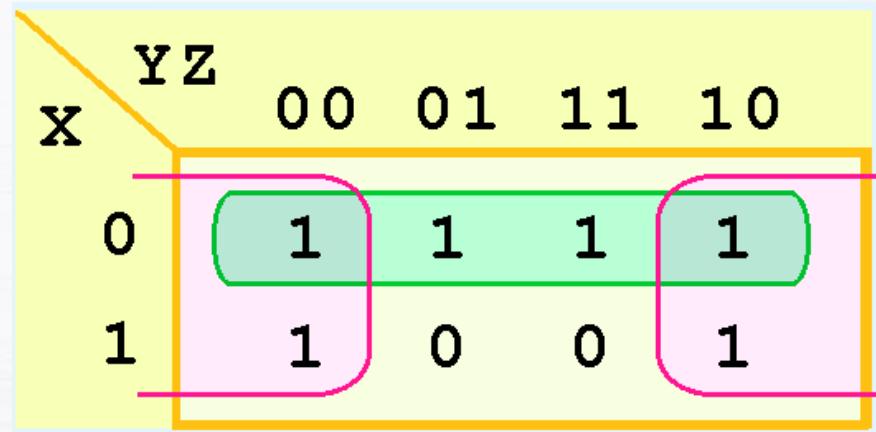
- Its Kmap is shown below. There are (only) two groupings of 1s.
 - Can you find them?

		YZ	00	01	11	10
		X	00	01	11	10
X	0	1	1	1	1	
	1	1	0	0	1	

3A.3 Kmap Simplification for Three Variables

- In this Kmap, we see an example of a group that wraps around the sides of a Kmap.
- This group tells us that the values of x and y are not relevant to the term of the function that is encompassed by the group.
 - What does this tell us about this term of the function?

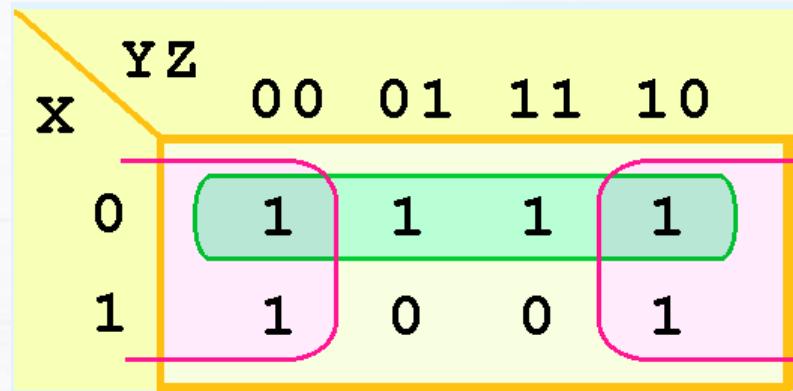
What about the green group in the top row?



3A.3 Kmap Simplification for Three Variables

- The green group in the top row tells us that only the value of x is significant in that group.
- We see that it is complemented in that row, so the other term of the reduced function is \mathbf{x}' .
- Our reduced function is $\mathbf{F}(\mathbf{x}, \mathbf{Y}, \mathbf{Z}) = \mathbf{x}' + \mathbf{z}'$

Recall that we had
six minterms in our
original function!



3A.3 Kmap Simplification for Four Variables

- Our model can be extended to accommodate the 16 minterms that are produced by a four-input function.
- This is the format for a 16-minterm Kmap:

		yz	00	01	11	10
		wx	00	01	11	10
00	00	$w' x' y' z'$	$w' x' y' z$	$w' x' y z$	$w' x' y z'$	
	01	$w' x y' z'$	$w' x y' z$	$w' x y z$	$w' x y z'$	
	11	$w x y' z'$	$w x y' z$	$w x y z$	$w x y z'$	
	10	$w x' y' z'$	$w x' y' z$	$w x' y z$	$w x' y z'$	

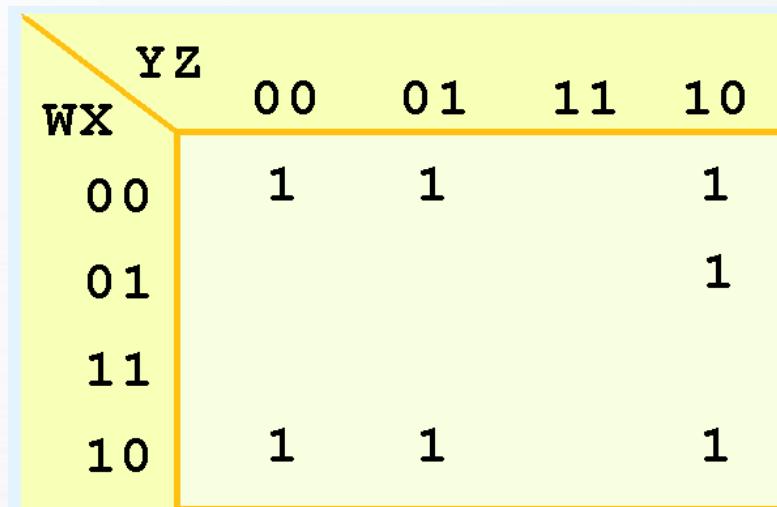
3A.3 Kmap Simplification for Four Variables

- We have populated the Kmap shown below with the nonzero minterms from the function:

$$\begin{aligned}F(W, X, Y, Z) = & W' X' Y' Z' + W' X' Y' Z + W' X' Y Z' \\& + W' X Y Z' + W X' Y' Z' + W X' Y' Z + W X' Y Z'\end{aligned}$$

- Can you identify (only) three groups in this Kmap?

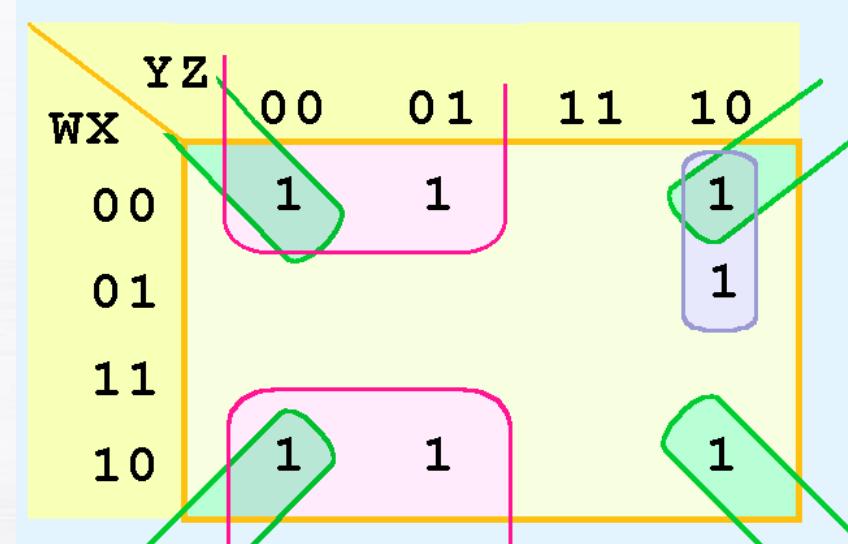
Recall that
groups can
overlap.



3A.3 Kmap Simplification for Four Variables

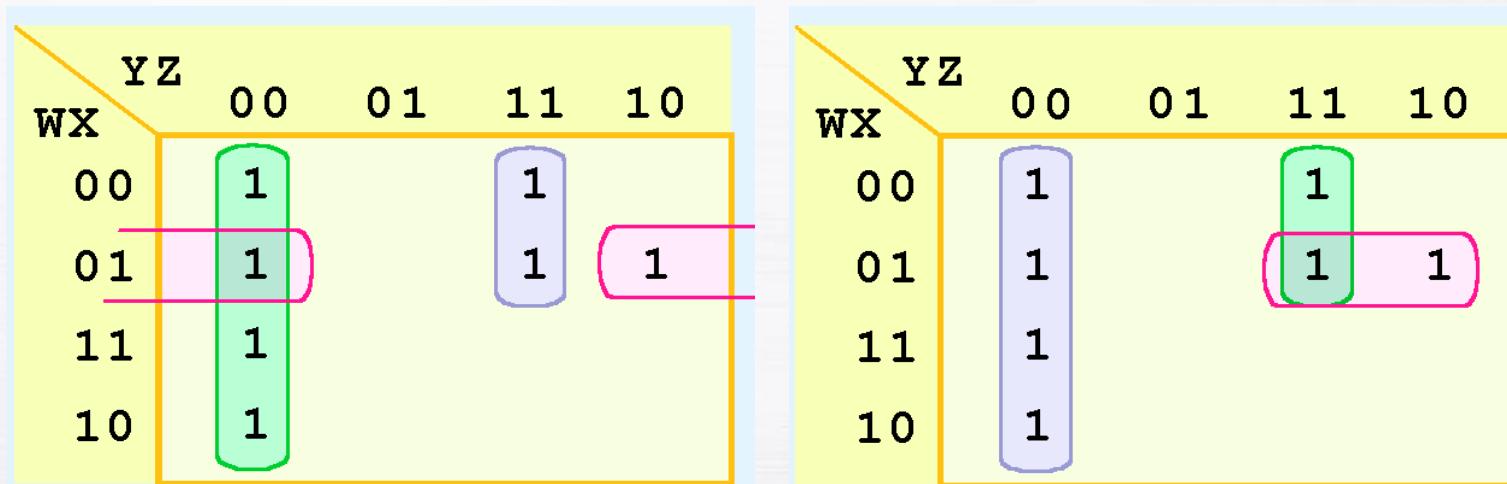
- Our three groups consist of:
 - A purple group entirely within the Kmap at the right.
 - A pink group that wraps the top and bottom.
 - A green group that spans the corners.
- Thus we have three terms in our final function:

$$F(W, X, Y, Z) = X' Y' + X' Z' + W' Y Z'$$



3A.3 Kmap Simplification for Four Variables

- It **is** possible to have a choice as to how to pick groups within a Kmap, while keeping the groups as large as possible.
- The (different) functions that result from the groupings below are logically equivalent.



3A.6 Don't Care Conditions

- Real circuits don't always need to have an output defined for every possible input.
 - For example, some calculator displays consist of 7-segment LEDs. These LEDs can display $2^7 - 1$ patterns, but only ten of them are useful.
- If a circuit is designed so that a particular set of inputs can never happen, we call this set of inputs a *don't care* condition.
- They are very helpful to us in Kmap circuit simplification.

3A.6 Don't Care Conditions

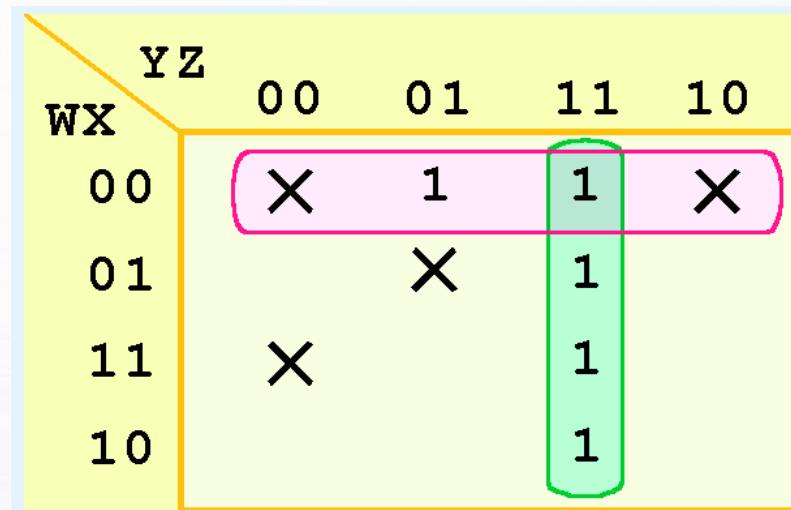
- In a Kmap, a don't care condition is identified by an X in the cell of the minterm(s) for the don't care inputs, as shown below.
- In performing the simplification, we are free to include or ignore the X 's when creating our groups.

w\nx	Y\nz	00	01	11	10
00	X	1	1	X	
01		X	1		
11	X		1		
10				1	

3A.6 Don't Care Conditions

- In one grouping in the Kmap below, we have the function:

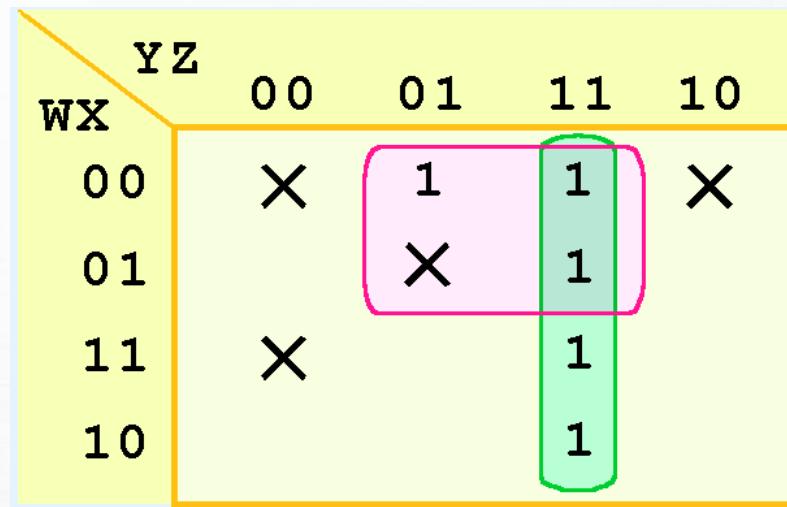
$$F(W, X, Y, Z) = W' X' + YZ$$



3A.6 Don't Care Conditions

- A different grouping gives us the function:

$$F(W, X, Y, Z) = W'Z + YZ$$



3A.6 Don't Care Conditions

- The truth table of:

$$F(W, X, Y, Z) = W' Y' + YZ$$

differs from the truth table of:

$$F(W, X, Y, Z) = W' Z + YZ$$

- However, the values for which they differ, are the inputs for which we have don't care conditions.

W\X	YZ	00	01	11	10
00	X	1	1	X	
01		X	1		
11	X		1		
10			1		

W\X	YZ	00	01	11	10
00	X	X	1	1	X
01		X	X	1	
11	X		X	1	
10			1	1	

3A Conclusion

- Kmaps provide an easy graphical method of simplifying Boolean expressions.
- A Kmap is a matrix consisting of the outputs of the minterms of a Boolean function.
- In this section, we have discussed 2- 3- and 4-input Kmaps. This method can be extended to any number of inputs through the use of multiple tables.
 - For 5 and 6 inputs it is basically a cube like structure viewed using 2 slices for 5 inputs and 4 slices for 6 inputs. You build groups of 1's that potentially go through layers

3A Conclusion

Recapping the rules of Kmap simplification:

- Groupings can contain only 1s; no 0s.
- Groups can be formed only at right angles; diagonal groups are not allowed.
- The number of 1s in a group must be a power of 2 – even if it contains a single 1.
- The groups must be made as large as possible.
- Groups can overlap and wrap around the sides of the Kmap.
- Use don't care conditions when you can.