


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.

2




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.

3



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?

4

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:
 - **Anything that can be done with software can also be done with hardware, and anything that can be done with hardware can also be done with software.***

* Assuming speed is not a concern.

5

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.

6

1.3 An Example System

Consider this advertisement:

MHz??

L1 Cache??

MB??

PCI??

USB??

For Sale: Obsolete Computer – Cheap! Cheap! Cheap!



- Pentium III 667MHz
- 133MHz 64MB SDRAM
- 32KB L1 cache, 256KB L2
- 30GB EIDE hard drive (7200)
- 48X max variable CD-ROM
- 2 USB ports, 1 serial port, 1 parallel port
- Monitor, 19", 24mm AG, 1280x1024 at 85Hz
- Intel 3D AGP graphics card
- 56K PCI voice modem
- 64-bit PCI sound card

What does it all mean??

7

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}

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

8

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
 - Main memory (RAM) is measured in MB
 - Disk storage is measured in GB for small systems, TB for large systems.

9

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}

10

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.

11

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 ...

12

1.3 An Example System

The microprocessor is the “brain” of the system. It executes program instructions. This one is a Pentium III (Intel) running at 667MHz.

- Pentium III 667MHz
- 133MHz

A system bus moves data within the computer. The faster the bus the better. This one runs at 133MHz.

13

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.

14

1.3 An Example System

This system has 64MB of (fast) synchronous dynamic RAM (SDRAM) . . .

- 64MB SDRAM
- 32KB L1 cache, 256KB L2 cache

... 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.

15

1.3 An Example System

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

- 30GB EIDE hard drive (7200 RPM)

This one can store 30GB. 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.)

16

1.3 An Example System

EIDE stands for *enhanced integrated drive electronics*, which describes how the hard disk interfaces with (or connects to) other system components.

- EIDE
• 48X max variable CD-ROM

A CD-ROM can store about 650MB of data, making it an ideal medium for distribution of commercial software packages. 48x describes its speed.

17

1.3 An Example System

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

- 2 USB ports, 1 serial port, 1 parallel port

This system has four ports.

18

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.")

19

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 PCI devices: a sound card, and a modem for connecting to the Internet.

- 56K PCI voice modem
- 64-bit PCI sound card

20

1.3 An Example System

The number of times per second that the image on the monitor is repainted is its *refresh rate*. The *dot pitch* of a monitor tells us how clear the image is.

This monitor has a dot pitch of 0.24mm and a refresh rate of 85Hz.

- Monitor, 19", .24mm AG, 1280x1024 at 85Hz
- Intel 3D AGP graphics card

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

21

1.3 An Example System

Throughout the remainder of this 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?

22

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 . . .

23

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.

24

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)

25

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."

26

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.

27

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.

28

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.



29

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 first *general-purpose* computer.

30

1.5 Historical Development

- The First Generation: Vacuum Tube Computers (1945 - 1953)

- IBM 650 (1955)
- Phased out in 1969.



The first *mass-produced* computer.

31

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
- ... and many others.



DEC PDP-1

32

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.



IBM 360



Cray-1

33

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.”



34

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 ...

35

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.”

36

1.5 Historical Development

- Rock's Law
 - In 2003, a chip plant under construction will cost over \$2.5 billion.
- \$2.5 billion is more than the gross domestic product of some small countries, including Belize, Bhutan, and the Republic of Sierra Leone.**
- For Moore's Law to hold, Rock's Law must fall, or vice versa. But no one can say which will give out first.

37

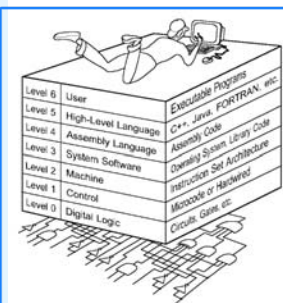
1.6 The Computer Level Hierarchy

- 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.

38

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.



39

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.

40

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.

41

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.

42

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.

43

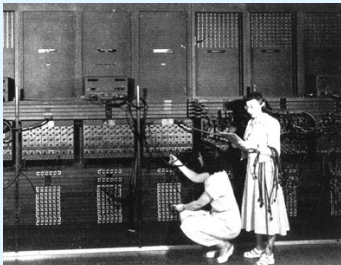
1.6 The Computer Level Hierarchy

- 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.

44

1.7 The von Neumann Model

- On the ENIAC, all programming was done at the digital logic level.
- Programming the computer involved moving plugs and wires.



45

1.7 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.

46

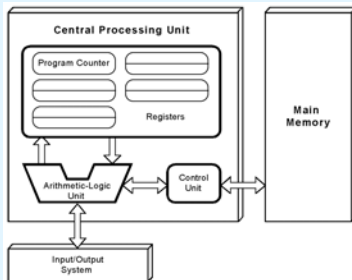
1.7 The von Neumann Model

- 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*.

47

1.7 The von Neumann Model

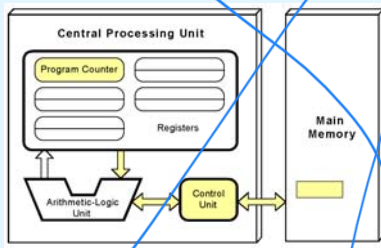
- This is a general depiction of a von Neumann system:
- These computers employ a fetch-decode-execute cycle to run programs as follows . . .



48

1.7 The von Neumann Model

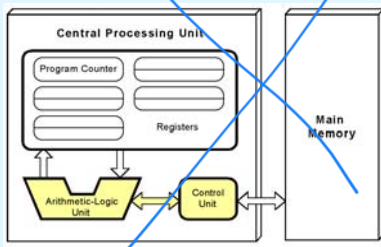
- The control unit fetches the next instruction from memory using the program counter to determine where the instruction is located.



49

1.7 The von Neumann Model

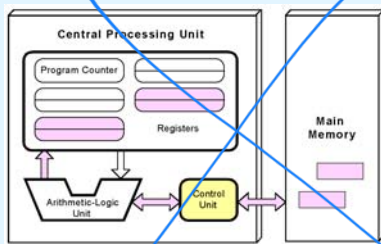
- The instruction is decoded into a language that the ALU can understand.



50

1.7 The von Neumann Model

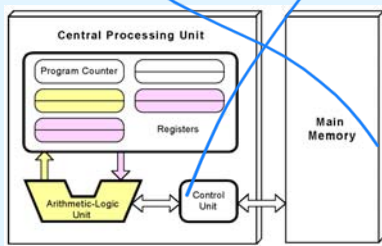
- Any data operands required to execute the instruction are fetched from memory and placed into registers within the CPU.



51

1.7 The von Neumann Model

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



52

1.8 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.

53

1.8 Non-von Neumann Models

- 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.

54

1.8 Non-von Neumann Models

- Parallel processing is only one method of providing increased computational power.
- More radical systems have reinvented the fundamental concepts of computation.
- These advanced systems include genetic computers, quantum computers, and dataflow systems.
- At this point, it is unclear whether any of these systems will provide the basis for the next generation of computers.

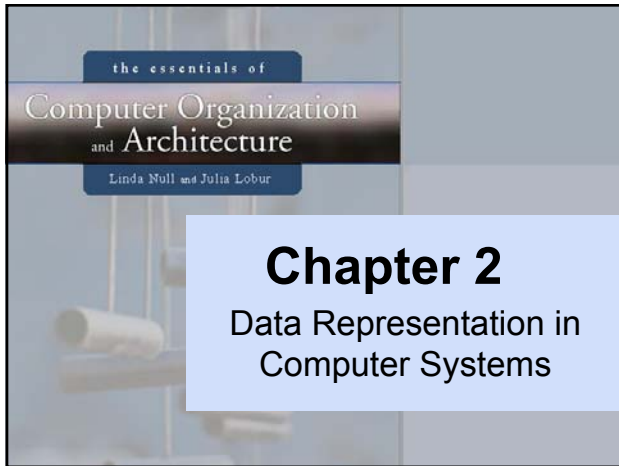
55

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.

56


57



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.


2



Chapter 2 Objectives

- Gain familiarity with the most popular character codes.
- Become aware of the differences between how data is stored in computer memory, how it is transmitted over telecommunication lines, and how it is stored on disks.
- Understand the concepts of error detecting and correcting codes.

3



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.

4

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* (or *nybble*).
 - Bytes, therefore, consist of two nibbles: a “high-order nibble,” and a “low-order” nibble.

5

2.2 Positional Numbering Systems



- Bytes store numbers when the position of each bit represents 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*).

6

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:

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

7

2.2 Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ = 16 + 8 + 0 + 0 + 1 = 25$$

- 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:

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

8

2.3 Decimal to Binary Conversions

- 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.

9

2.3 Decimal to Binary Conversions

- In a previous slide, we said that every integer value can be represented exactly using any radix system.
- You can use either of 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.

10

2.3 Decimal to Binary Conversions

- **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.

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

11



2.3 Decimal to Binary Conversions

- **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.

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

12

2.3 Decimal to Binary Conversions

- **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:
 $190_{10} = 21001_3$

$$\begin{array}{r} 190 \\ - 3 \\ \hline 187 \\ - 1 \\ \hline 186 \end{array}$$

$186 = 3^1 \times 0$
 $186 = 3^0 \times 1$

13

2.3 Decimal to Binary Conversions

- 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.

14

2.3 Decimal to Binary Conversions

- **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.

$$\begin{array}{r} 3 \overline{) 190} \\ \underline{189} \\ 1 \end{array}$$

63

15

2.3 Decimal to Binary Conversions

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

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

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

16

2.3 Decimal to Binary Conversions

- **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:

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

$$\begin{array}{r} 3 \overline{) 190} \quad 1 \\ 3 \overline{) 63} \quad 0 \\ 3 \overline{) 21} \quad 0 \\ 3 \overline{) 7} \quad 1 \\ 3 \overline{) 2} \quad 2 \\ \quad 0 \end{array}$$

17

2.3 Decimal to Binary Conversions

- 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.

18

2.3 Decimal to Binary Conversions

- 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:

$$\begin{aligned}0.47_{10} &= 4 \times 10^{-1} + 7 \times 10^{-2} \\0.11_2 &= 1 \times 2^{-1} + 1 \times 2^{-2} \\&= \frac{1}{2} + \frac{1}{4} \\&= 0.5 + 0.25 = 0.75\end{aligned}$$

19

2.3 Decimal to Binary Conversions

- As with whole-number conversions, you can use either of two methods: a subtraction method and 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.

20

2.3 Decimal to Binary Conversions

- 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.

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

21

2.3 Decimal to Binary Conversions

- 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.

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

22

2.3 Decimal to Binary Conversions

- Converting 0.8125 to binary . . .

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

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

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

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

23

2.3 Decimal to Binary Conversions

- 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.

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

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

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

$$\begin{array}{r} .5000 \\ \times 2 \\ \hline 1.0000 \end{array}$$

24

2.3 Decimal to Binary Conversions

- 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.

25

2.3 Decimal to Binary Conversions

- The hexadecimal numbering system uses the numerals 0 through 9 and the letters A through F.
 - The decimal number 12 is B_{16} .
 - The decimal number 26 is $1A_{16}$.
- 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**

26

2.3 Decimal to Binary Conversions

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

0011	0101	0001	1011
3	5	1	B

- 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.

27

2.4 Signed Integer Representation

- The conversions we have so far presented have involved only positive numbers.
- To represent negative values, computer systems allocate the high-order bit to indicate the sign of a value.
 - The high-order bit is the leftmost bit in a byte. It is also called the most significant bit.
- The remaining bits contain the value of the number.

28

2.4 Signed Integer Representation

- There are three ways in which signed binary numbers may be expressed:
 - Signed magnitude,
 - One's complement and
 - 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.

29

2.4 Signed Integer Representation

- For example, in 8-bit signed magnitude, positive 3 is: 00000011
- Negative 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.

30

2.4 Signed Integer Representation

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

$$\begin{array}{ll} 0 + 0 = 0 & 0 + 1 = 1 \\ 1 + 0 = 1 & 1 + 1 = 10 \end{array}$$

- 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 . . .

31

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.

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

32

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.

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

33

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.

```
      1
0  1001011
0 + 0101110
-----
    01
```

34

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.

```
      1 1 1
0  1001011
0 + 0101110
-----
    1001
```

35

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.

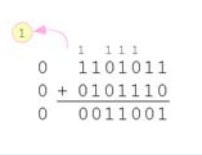
```
      1 1 1
0  1001011
0 + 0101110
-----
0  1111001
```

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

36

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**: $107 + 46 = 25$.


$$\begin{array}{r} 1 \quad 1 \quad 1 \quad 1 \quad 1 \\ 0 \quad 1101011 \\ 0 + 0101110 \\ \hline 0 \quad 0011001 \end{array}$$

37

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.

$$\begin{array}{r} 1 \quad 1 \\ 1 \quad 0101110 \\ 1 + 0011001 \\ \hline 1 \quad 1000111 \end{array}$$

38

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.


$$\begin{array}{r} 0 \quad 0 \quad 2 \quad 0 \quad 2 \\ 0 \quad 010110 \\ 1 + 0011001 \\ \hline 0 \quad 0010101 \end{array}$$

39

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.

40

2.4 Signed Integer Representation

- In complement systems, negative values are represented by some difference between a number and its base.
- In *diminished radix complement* systems, a negative value is given by the difference between the absolute value of a number and one less than its base.
- In the binary system, this gives us *one's complement*. It amounts to little more than flipping the bits of a binary number.

41

2.4 Signed Integer Representation

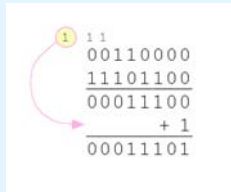
- For example, in 8-bit one's complement, positive 3 is: 00000011
- Negative 3 is: 11111100
 - In one's complement, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
- Complement systems are useful because they eliminate the need for special circuitry for subtraction.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

42

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



We note that 19 in one's complement is 00010011, so -19 in one's complement is: 11101100.

43

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.

44

2.4 Signed Integer Representation

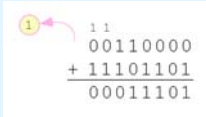
- To express a value in two's complement:
 - If the number is positive, just convert it to binary and you're done.
 - If the number is negative, find the one's complement of the number and then add 1.
- Example:
 - In 8-bit one's complement, positive 3 is: 00000011
 - Negative 3 in one's complement is: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

45

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.


$$\begin{array}{r} 11 \\ 00110000 \\ + 11101101 \\ \hline 00011101 \end{array}$$

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

46

2.4 Signed Integer Representation

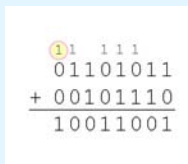
- When we use any finite number of bits to represent a number, we always run the risk of the result of our calculations becoming too large to be stored in the computer.
- While we can't always prevent overflow, we can always *detect* overflow.
- In complement arithmetic, an overflow condition is easy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



47

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: $107 + 46 = -103$.


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

Rule for detecting two's complement overflow: When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred.



48

2.5 Floating-Point Representation

- The signed magnitude, one's complement, and two's complement representation that we have just presented deal with 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.

49

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.

50

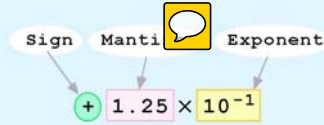
2.5 Floating-Point Representation

- 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:
 $0.125 = 1.25 \times 10^{-1}$
 $5,000,000 = 5.0 \times 10^6$

51

2.5 Floating-Point Representation

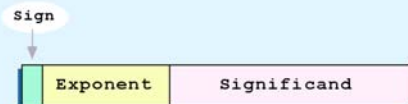
- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



52

2.5 Floating-Point Representation

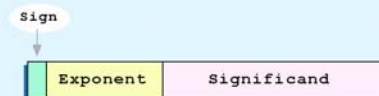
- Computer representation of a floating-point number consists of three fixed-size fields:



- This is the standard arrangement of these fields.

53

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.

54

2.5 Floating-Point Representation

Sign



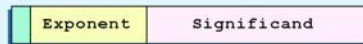
- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent and a 52-bit significand.

For illustrative purposes, we will use a 14-bit model with a 5-bit exponent and an 8-bit significand.

55

2.5 Floating-Point Representation

Sign

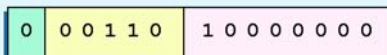


- The significand of a floating-point number is always preceded by an implied binary point.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 to which the significand is raised.

56

2.5 Floating-Point Representation

- Example:
 - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5 . So in (binary) scientific notation $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- Using this information, we put 110 ($= 6_{10}$) in the exponent field and 1 in the significand as shown.



57

2.5 Floating-Point Representation

- The illustrations shown at the right are *all* equivalent representations for 32 using our simplified model.

0	00110	10000000
---	-------	----------

0	00111	01000000
---	-------	----------

- Not only do these synonymous representations waste space, but they can also cause confusion.

0	01000	00100000
---	-------	----------

0	01001	00010000
---	-------	----------

58

2.5 Floating-Point Representation

Sign



- Another problem with our system is that we have made no allowances for negative exponents. We have no way to express $0.5 (=2^{-1})$! (Notice that there is no sign in the exponent field!)

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

59

2.5 Floating-Point Representation

- To resolve the problem of synonymous forms, we will establish a rule that the first digit of the significand must be 1. This results in a unique pattern for each floating-point number.
 - In the IEEE-754 standard, this 1 is implied meaning that a 1 is assumed after the binary point.
 - By using an implied 1, we increase the precision of the representation by a power of two. (Why?)

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

60

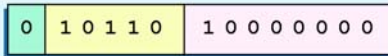
2.5 Floating-Point Representation

- To provide for negative exponents, we will use a *biased exponent*.
- A bias is a number that is approximately midway in the range of values expressible by the exponent. We subtract the bias from the value in the exponent to determine its true value.
 - In our case, we have a 5-bit exponent. We will use 16 for our bias. This is called *excess-16* representation.
- In our model, exponent values less than 16 are negative, representing fractional numbers.

61

2.5 Floating-Point Representation

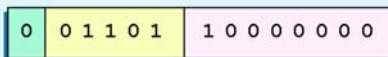
- Example:
 - Express 32_{10} in the revised 14-bit floating-point model.
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 16 biased exponent, we add 16 to 6, giving 22_{10} ($=10110_2$).
- Graphically:



62

2.5 Floating-Point Representation

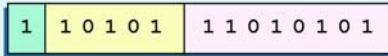
- Example:
 - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4} . So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 16 biased exponent, we add 16 to -3, giving 13_{10} ($=01101_2$).



63

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 16 biased exponent, we add 16 to 5, giving $21_{10} (=10101_2)$. We also need a 1 in the sign bit.



64

2.5 Floating-Point Representation

- The IEEE-754 single precision floating point standard uses bias of 127 over its 8-bit exponent.
 - An exponent of 255 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.
- The double precision standard has a bias of 1023 over its 11-bit exponent.
 - The “special” exponent value for a double precision number is 2047, instead of the 255 used by the single precision standard.



65

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.



66

2.5 Floating-Point Representation

- 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.



67

2.5 Floating-Point Representation

- Example:
 - Find the sum of 12_{10} and 1.25_{10} using the 14-bit 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 .

0	1	0	1	0	0	1	1	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	1	0	0
<hr/>													
0	1	0	1	0	0	1	1	0	1	0	1	0	0

+



68

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.



69

2.5 Floating-Point Representation

- Example:
 - Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model.
- We find $12_{10} = 0.1100 \times 2^4$. And $1.25_{10} = 0.101 \times 2^1$.
- Thus, our product is $0.01111100 \times 2^5 = 0.1111 \times 2^4$.
- The normalized product requires an exponent of $20_{10} = 10110_2$.

×

0	1	0	1	0	0	1	1	0	0	0	0	0	0
0	1	0	0	0	1	1	0	1	0	0	0	0	0
0	1	0	1	0	1	0	1	1	1	1	0	0	0

70

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.

71

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:

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

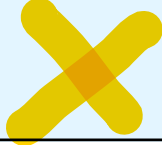
72

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:

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

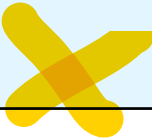
- If we had a procedure that repetitively added 0.5 to 128.5, we would have an error of nearly 2% after only four iterations.



73

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.



74

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.



75

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.

76

2.6 Character Codes

- As computers have evolved, character codes have evolved.
- Larger computer memories and storage devices permit richer character codes.
- The earliest computer coding systems used six bits.
- Binary-coded decimal (BCD) was one of these early codes. It was used by IBM mainframes in the 1950s and 1960s.

77

2.6 Character Codes

- 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 and BCD are still in use by IBM mainframes today.

78

2.6 Character Codes

- Other computer manufacturers chose the 7-bit ASCII (American Standard Code for Information Interchange) as a replacement for 6-bit codes.
- While BCD and EBCDIC were based upon punched card codes, ASCII was based upon telecommunications (Telex) codes.
- Until recently, ASCII was the dominant character code outside the IBM mainframe world.

79

2.6 Character Codes

- Many of today's systems embrace Unicode, a 16-bit system that can encode the characters of every language in the world.
 - The Java programming language, and some operating systems now use Unicode as their default character code.
- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

80

2.6 Character Codes

- The Unicode codespace allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

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

81

2.7 Codes for Data Recording And Transmission

- When character codes or numeric values are stored in computer memory, their values are unambiguous.
- This is not always the case when data is stored on magnetic disk or transmitted over a distance of more than a few feet.
 - Owing to the physical irregularities of data storage and transmission media, bytes can become garbled.
- Data errors are reduced by use of suitable coding methods as well as through the use of various error-detection techniques.

82

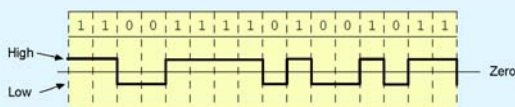
2.7 Codes for Data Recording And Transmission

- To transmit data, pulses of “high” and “low” voltage are sent across communications media.
- To store data, changes are induced in the magnetic polarity of the recording medium.
 - These polarity changes are called *flux reversals*.
- The period of time during which a bit is transmitted, or the area of magnetic storage within which a bit is stored is called a *bit cell*.

83

2.7 Codes for Data Recording And Transmission

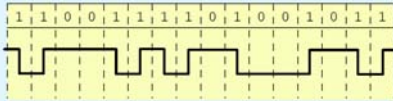
- The simplest data recording and transmission code is the non-return-to-zero (NRZ) code.
- NRZ encodes 1 as “high” and 0 as “low.”
- The coding of OK (in ASCII) is shown below.



84

2.7 Codes for Data Recording And Transmission

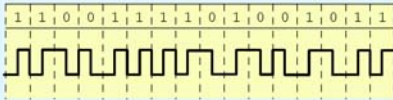
- The problem with NRZ code is that long strings of zeros and ones cause synchronization loss.
- Non-return-to-zero-invert (NRZI) reduces this synchronization loss by providing a transition (either low-to-high or high-to-low) for each binary 1.



85

2.7 Codes for Data Recording And Transmission

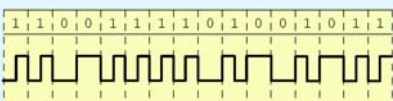
- Although it prevents loss of synchronization over long strings of binary ones, NRZI coding does nothing to prevent synchronization loss within long strings of zeros.
- Manchester coding (also known as phase modulation) prevents this problem by encoding a binary one with an “up” transition and a binary zero with a “down” transition.



86

2.7 Codes for Data Recording And Transmission

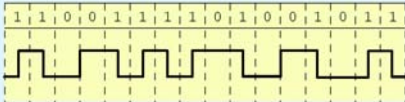
- For many years, Manchester code was the dominant transmission code for local area networks.
- It is, however, wasteful of communications capacity because there is a transition on every bit cell.
- A more efficient coding method is based upon the frequency modulation (FM) code. In FM, a transition is provided at each cell boundary. Cells containing binary ones have a mid-cell transition.



87

2.7 Codes for Data Recording And Transmission

- At first glance, FM is worse than Manchester code, because it requires a transition at each cell boundary.
- If we can eliminate some of these transitions, we would have a more economical code.
- Modified FM does just this. It provides a cell boundary transition only when adjacent cells contain zeros.
- An MFM cell containing a binary one has a transition in the middle as in regular FM.



88

2.7 Codes for Data Recording And Transmission

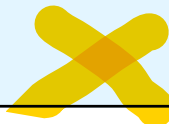
- The main challenge for data recording and transmission is how to retain synchronization without chewing up more resources than necessary.
- Run-length-limited, RLL, is a code specifically designed to reduce the number of consecutive ones and zeros.
 - Some extra bits are inserted into the code.
 - But even with these extra bits RLL is remarkably efficient.



89

2.7 Codes for Data Recording And Transmission

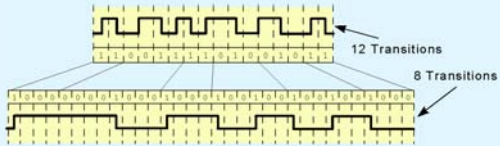
- An $RLL(d,k)$ code dictates a minimum of d and a maximum of k consecutive zeros between any pair of consecutive ones.
 - $RLL(2,7)$ has been the dominant disk storage coding method for many years.
- An $RLL(2,7)$ code contains more bit cells than its corresponding ASCII or EBCDIC character.
- However, the coding method allows bit cells to be smaller, thus closer together, than in MFM or any other code.



90

2.7 Codes for Data Recording And Transmission

- The RLL(2,7) coding for *OK* is shown below, compared to MFM. The RLL code (bottom) contains 25% fewer transitions than the MFM code (top).



91

2.8 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 increases--sometimes geometrically.
- Thus, error detection and correction is critical to accurate data transmission, storage and retrieval.

92

2.8 Error Detection and Correction

- Check digits**, appended to the end of a long number can provide some protection against data input errors.
 - The last character of UPC barcodes and ISBNs are check digits.
- Longer data streams require more economical and sophisticated error detection mechanisms.
- Cyclic redundancy checking (CRC)** codes provide error detection for large blocks of data.

93

2.8 Error Detection and Correction

- 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*.
- CRCs are 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.

94

2.8 Error Detection and Correction

- 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:

$$\begin{array}{ll} 0 + 0 = 0 & 0 + 1 = 1 \\ 1 + 0 = 1 & 1 + 1 = 0 \end{array}$$

You will fully understand why modulo 2 arithmetic is so handy after you study digital circuits in Chapter 3.

95

2.8 Error Detection and Correction

- 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.

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

96

2.8 Error Detection and Correction

- 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.

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

97

2.8 Error Detection and Correction

- 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.

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

98

2.8 Error Detection and Correction

- 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.

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

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

99

2.8 Error Detection and Correction

- Suppose we want to transmit the information string: 1111101.
- The receiver and sender decide to use the (arbitrary) polynomial pattern, 1101.
- The information string is shifted left by one position less than the number of positions in the divisor.
- The remainder is found through modulo 2 division (at right) and added to the information string: $1111101000 + 111 = 1111101111$.

```
      1011011
1101)1111101000
     1101
     001010
      1101
      01111
       1101
       001000
        1101
        01010
         1101
          0111
```

100

2.8 Error Detection and Correction

- If no bits are lost or corrupted, dividing the received information string by the agreed upon pattern will give a remainder of zero.
- We see this is so in the calculation at the right.
- Real applications use longer polynomials to cover larger information strings.
 - Some of the standard polynomials are listed in the text.

```
      1011011
1101)1111101111
     1101
     001010
      1101
      01111
       1101
       001011
        1101
        01101
         1101
          0000
```

101

2.8 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.

102

2.8 Error Detection and Correction

- Hamming codes and Reed-Soloman codes are two important error correcting codes.
- Reed-Soloman codes are particularly useful in correcting *burst errors* that occur when a series of adjacent bits are damaged.
 - Because CD-ROMs are easily scratched, they employ a type of Reed-Soloman error correction.
- Because the mathematics of Hamming codes is much simpler than Reed-Soloman, we discuss Hamming codes in detail.

103

2.8 Error Detection and Correction

- Hamming codes are code words formed by adding redundant check bits, or parity bits, to a data word.
- The *Hamming distance* between two code words is the number of bits in which two code words differ.

This pair of bytes has a Hamming distance of 3:

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.

104

2.8 Error Detection and Correction

- The minimum Hamming distance for a code, $D(\min)$, determines its error detecting and error correcting capability.
- 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 $D(\min) = k + 1$.

105

2.8 Error Detection and Correction

- Hamming codes can *detect* $D(\min) - 1$ errors and *correct* $\left\lfloor \frac{D(\min) - 1}{2} \right\rfloor$ errors
- Thus, a Hamming distance of $2k + 1$ is required to be able to correct k errors in any data word.
- Hamming distance is provided by adding a suitable number of parity bits to a data word.


106

2.8 Error Detection and Correction

- Suppose we have a set of n -bit code words consisting of m data bits and r (redundant) parity bits.
- An error could occur in any of the n bits, so each code word can be associated with n erroneous words at a Hamming distance of 1.
- Therefore, we have $n + 1$ bit patterns for each code word: one valid code word, and n erroneous words.

107

2.8 Error Detection and Correction

- With n -bit code words, we have 2^n possible code words consisting of 2^m data bits (where $m = n + r$). 

- This gives us the inequality:

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

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

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

- This inequality gives us a lower limit on the number of check bits that we need in our code words.

108

2.8 Error Detection and Correction

- Suppose we have data words of length $m = 4$. Then:
$$(4 + r + 1) \leq 2^r$$
implies that r must be greater than or equal to 3.
- This means to build a code with 4-bit data words that will correct single-bit errors, we must add 3 check bits.
- Finding the number of check bits is the hard part. The rest is easy.

109

2.8 Error Detection and Correction

- Suppose we have data words of length $m = 8$. Then:
$$(8 + r + 1) \leq 2^r$$
implies that r must be greater than or equal to 4.
- This means to build a code with 8-bit data words that will correct single-bit errors, we must add 4 check bits, creating code words of length 12.
- So how do we assign values to these check bits?

110

2.8 Error Detection and Correction

- With code words of length 12, we observe that each of the digits, 1 through 12, can be expressed in powers of 2. Thus:

$1 = 2^0$	$5 = 2^2 + 2^0$	$9 = 2^3 + 2^0$
$2 = 2^1$	$6 = 2^2 + 2^1$	$10 = 2^3 + 2^1$
$3 = 2^1 + 2^0$	$7 = 2^2 + 2^1 + 2^0$	$11 = 2^3 + 2^1 + 2^0$
$4 = 2^2$	$8 = 2^3$	$12 = 2^3 + 2^2$

 - 1 ($= 2^0$) contributes to all of the odd-numbered digits.
 - 2 ($= 2^1$) contributes to the digits, 2, 3, 6, 7, 10, and 11.
 - ... And so forth ...
- We can use this idea in the creation of our check bits.

111

2.8 Error Detection and Correction

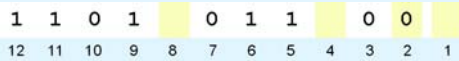
- Using our code words of length 12, number each bit position starting with 1 in the low-order bit.
- Each bit position corresponding to an even power of 2 will be occupied by a check bit.
- These check bits contain the parity of each bit position for which it participates in the sum.



112

2.8 Error Detection and Correction

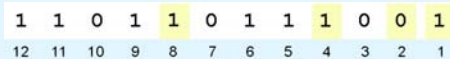
- Since 2 ($= 2^1$) contributes to the digits, 2, 3, 6, 7, 10, and 11. Position 2 will contain the parity for bits 3, 6, 7, 10, and 11.
- When we use even parity, this is the modulo 2 sum of the participating bit values.
- For the bit values shown, we have a parity value of 0 in the second bit position.



What are the values for the other parity bits?

113

2.8 Error Detection and Correction



- The completed code word is shown above.
 - Bit 1 checks the digits, 3, 5, 7, 9, and 11, so its value is 1.
 - Bit 4 checks the digits, 5, 6, 7, and 12, so its value is 1.
 - Bit 8 checks the digits, 9, 10, 11, and 12, so its value is also 1.
- Using the Hamming algorithm, we can not only detect single bit errors in this code word, but also correct them!

114

2.8 Error Detection and Correction

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

- Suppose an error occurs in bit 5, as shown above. Our parity bit values are:
 - Bit 1 checks digits, 3, 5, 7, 9, and 11. *Its value is 1, but should be zero.*
 - Bit 2 checks digits 2, 3, 6, 7, 10, and 11. The zero is correct.
 - Bit 4 checks digits, 5, 6, 7, and 12. *Its value is 1, but should be zero.*
 - Bit 8 checks digits, 9, 10, 11, and 12. This bit is correct.

115

2.8 Error Detection and Correction

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

- We have erroneous bits in positions 1 and 4.
- With *two* parity bits that don't check, we know that the error is in the data, and not in a parity bit.
- Which data bits are in error? We find out by adding the bit positions of the erroneous bits.
- Simply, $1 + 4 = 5$. This tells us that the error is in bit 5. If we change bit 5 to a 1, all parity bits check and our data is restored.

116

Chapter 2 Conclusion

- Computers store data in the form of bits, bytes, and words using the binary numbering system.
- Hexadecimal numbers are formed using four-bit groups called nibbles (or nybbles).
- Signed integers can be stored in one's complement, two's complement, or signed magnitude representation.
- Floating-point numbers are usually coded using the IEEE 754 floating-point standard.

117

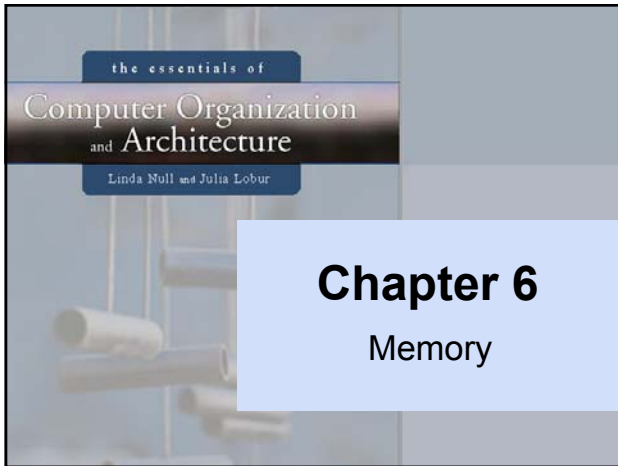
Chapter 2 Conclusion



- Character data is stored using ASCII, EBCDIC, or Unicode.
- Data transmission and storage codes are devised to convey or store bytes reliably and economically.
- Error detecting and correcting codes are necessary because we can expect no transmission or storage medium to be perfect.
- CRC, Reed-Soloman, and Hamming codes are three important error control codes.

118


119



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.


2



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.

3



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).
- Dynamic RAM 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.

4

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.

5

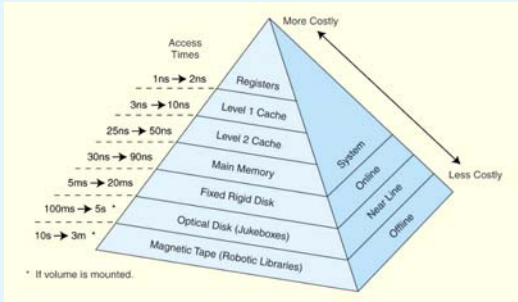
6.3 The Memory Hierarchy

- 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

6.3 The Memory Hierarchy

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



7

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.

8

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.

9

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.

10

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.

11

6.4 Cache Memory



- The "content" that is addressed in content addressable cache memory is a subset of the bits of a main memory address called a *field*.
- The fields into which a memory address is divided provide a many-to-one mapping between larger main memory and the smaller cache memory.
- Many blocks of main memory map to a single block of cache. A *tag* field in the cache block distinguishes one cached memory block from another.

12

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.
- Once a block of memory is copied into its slot in cache, a *valid* bit is set for the cache block to let the system know that the block contains valid data.

What could happen without having a valid bit?

13

6.4 Cache Memory

- The diagram below is a schematic of what cache looks like.

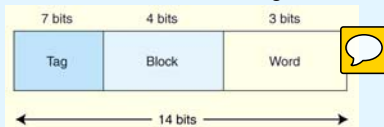
Block	Tag	Data	Valid
0	00000000	words A, B, C,...	1
1	11110101	words L, M, N,...	1
2		0
3		0

- Block 0 contains multiple words from main memory, identified with the tag 00000000. Block 1 contains words identified with the tag 11110101.
- The other two blocks are not valid.

14

6.4 Cache Memory

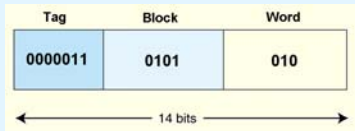
- The size of each field into which a memory address is divided depends on the size of the cache.
- Suppose our memory consists of 2^{14} words, cache has $16 = 2^4$ blocks, and each block holds 8 words.
 - Thus memory is divided into $2^{14} / 2^4 = 2^{10}$ blocks.
- For our field sizes, we know we need 4 bits for the block, 3 bits for the word, and the tag is what's left over:



15

6.4 Cache Memory

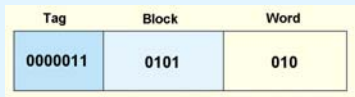
- As an example, suppose a program generates the address **1AA**. In 14-bit binary, this number is: 00000110101010.
- The first 7 bits of this address go in the tag field, the next 4 bits go in the block field, and the final 3 bits indicate the word within the block.



16

6.4 Cache Memory

- If subsequently the program generates the address **1AB**, it will find the data it is looking for in block 0101, word 011.



- However, if the program generates the address, **3AB**, instead, the block loaded for address **1AA** would be evicted from the cache, and replaced by the blocks associated with the **3AB** reference.

17

6.4 Cache Memory

- Suppose a program generates a series of memory references such as: **1AB**, **3AB**, **1AB**, **3AB**, The cache will continually evict and replace blocks.
- The theoretical advantage offered by the cache is lost in this extreme case.
- This is the main disadvantage of direct mapped cache.
- Other cache mapping schemes are designed to prevent this kind of thrashing.

18

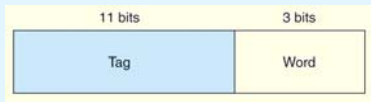
6.4 Cache Memory

- 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.

19

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.

20

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.

21

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.

22

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.

Set	Tag	Block 0 of set	Valid	Tag	Block 1 of set	Valid
0	00000000	Words A, B, C, ...	1	00000000		0
1	11110101	Words L, M, N, ...	1	00000000		0
2	00000000		0	10111011	P, Q, R, ...	1
3	00000000		0	11111100	T, U, V, ...	1

23

6.4 Cache Memory

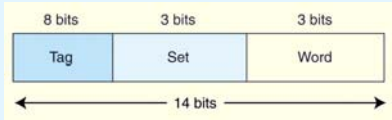
- In set associative cache mapping, a memory reference is divided into three fields: tag, set, and word, as shown below.
- As with direct-mapped cache, the word field chooses the word within the cache block, and the tag field uniquely identifies the memory address.
- The set field determines the set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



24

6.4 Cache Memory

- Suppose we have a main memory of 2^{14} bytes.
- This memory is mapped to a 2-way set associative cache having 16 blocks where each block contains 8 words.
- Since this is a 2-way cache, each set consists of 2 blocks, and there are 8 sets.
- Thus, we need 3 bits for the set, 3 bits for the word, giving 8 leftover bits for the tag:



25

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.

26

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 accessed 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.

27

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.

28

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.

29

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%.
- The EAT is:
$$0.99(10\text{ns}) + 0.01(200\text{ns}) = 9.9\text{ns} + 2\text{ns} = 11\text{ns}.$$
- This equation for determining the effective access time can be extended to any number of memory levels, as we will see in later sections.

30

6.4 Cache Memory

- Cache replacement policies must also 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.

31

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.

32

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.

33

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.

34

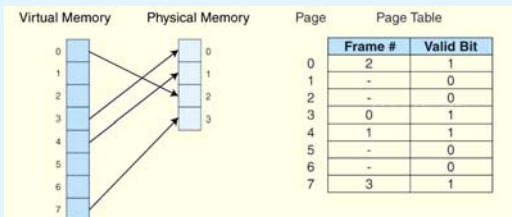
6.5 Virtual Memory

- 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.

35

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.



36

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.

37

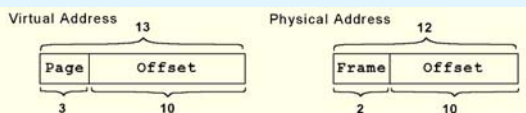
6.5 Virtual Memory

- 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.

38

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.



39

6.5 Virtual Memory

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

	Frame	Valid Bit	Addresses
Page 0	–	0	Page 0 : 0 – 1023
1	3	1	1 : 1024 – 2047
2	0	1	2 : 2048 – 3071
3	–	0	3 : 3072 – 4095
4	–	0	4 : 4096 – 5119
5	1	1	5 : 5120 – 6143
6	2	1	6 : 6144 – 7167
7	–	0	7 : 7168 – 8191

40

6.5 Virtual Memory

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

	Frame	Valid Bit	Addresses
Page 0	–	0	Page 0 : 0 – 1023
1	3	1	1 : 1024 – 2047
2	0	1	2 : 2048 – 3071
3	–	0	3 : 3072 – 4095
4	–	0	4 : 4096 – 5119
5	1	1	5 : 5120 – 6143
6	2	1	6 : 6144 – 7167
7	–	0	7 : 7168 – 8191

41

6.5 Virtual Memory

- What happens when the CPU generates address 1000000000100_2 ?

	Frame	Valid Bit	Addresses
Page 0	–	0	Page 0 : 0 – 1023
1	3	1	1 : 1024 – 2047
2	0	1	2 : 2048 – 3071
3	–	0	3 : 3072 – 4095
4	–	0	4 : 4096 – 5119
5	1	1	5 : 5120 – 6143
6	2	1	6 : 6144 – 7167
7	–	0	7 : 7168 – 8191

42

-
-
-
-
-
-

43

-
-
-
-
-
-

44



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.

46

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.

47

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.

48

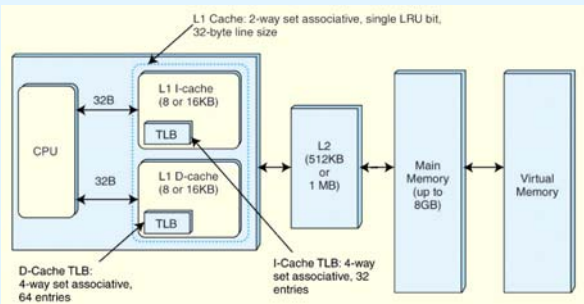
6.6 A Real-World Example

- 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.

49

6.6 A Real-World Example



50

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.

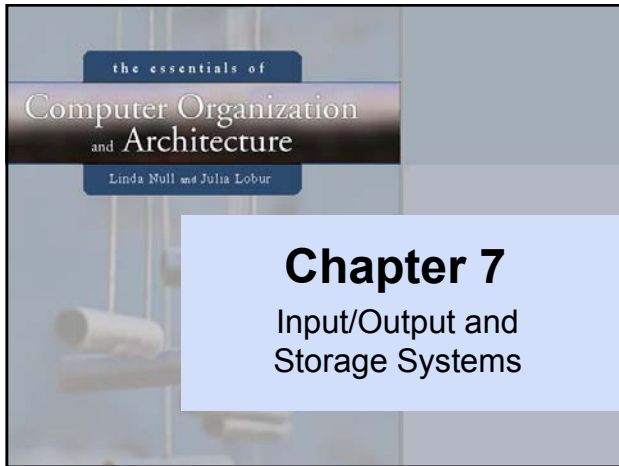
51

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.

52


53



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.
- Become familiar with the concepts of data compression and applications suitable for each type of compression algorithm.


2



7.1 Introduction

- Data storage and retrieval is one of the primary functions of computer systems.
- Sluggish I/O performance 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.

3



7.2 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.

4

7.2 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 250% 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?

5

7.2 Amdahl's Law

- The processor option offers a 130% speedup:

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

- And the disk drive option gives a 122% speedup:

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

- 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?

6

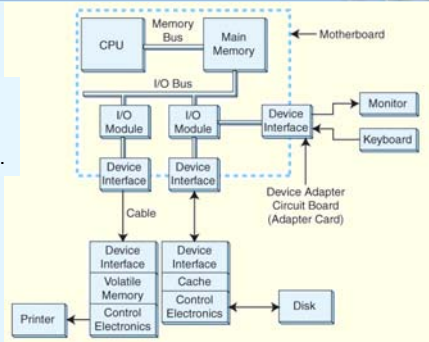
7.3 I/O Architectures

- 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

7.3 I/O Architectures

This is a model I/O configuration.



7.3 I/O Architectures

- I/O can be controlled in four 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.
- 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.

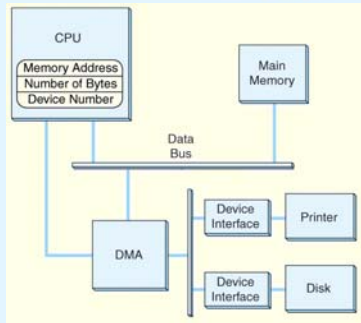
9

7.3 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.



10

7.3 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*.

11

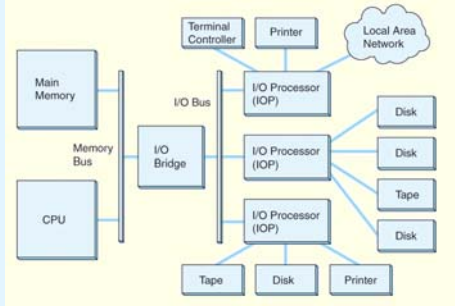
7.3 I/O Architectures

- 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.

12

7.3 I/O Architectures

- This is a channel I/O configuration.



13

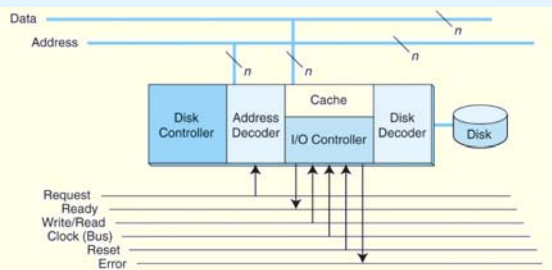
7.3 I/O Architectures

- 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.

14

7.3 I/O Architectures

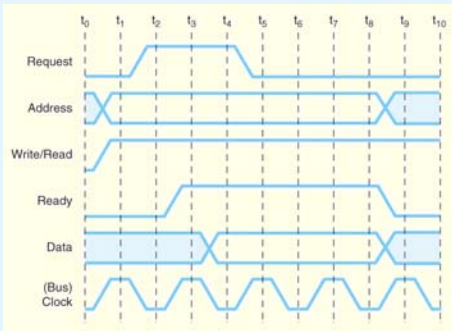
This is how a bus connects to a disk drive.



15

7.3 I/O Architectures

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



16

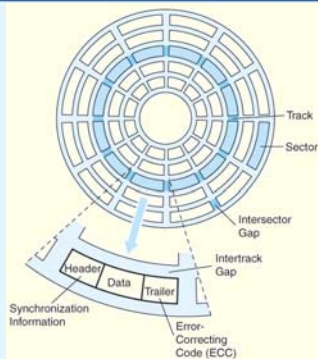
7.4 Magnetic Disk Technology

- 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.

17

7.4 Magnetic Disk Technology

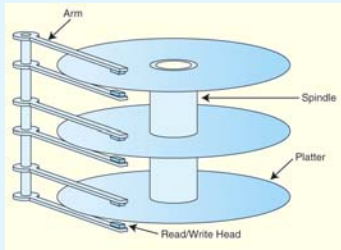
Disk tracks are numbered from the outside edge, starting with zero.



18

7.4 Magnetic Disk Technology

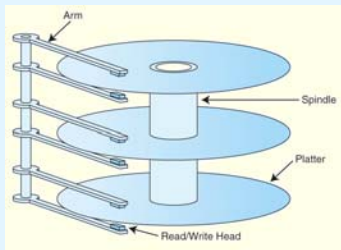
- Hard disk platters are mounted on spindles.
- Read/write heads are mounted on a comb that swings radially to read the disk.



19

7.4 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.



20

7.4 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.
- $\text{Seek time} + \text{rotational delay} = \text{access time}.$

21

7.4 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:
$$\frac{60 \text{ seconds}}{\text{disk rotation speed}} \times \frac{1000 \text{ ms}}{\text{second}}$$

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 7.11 in the text shows a sample disk specification.

22

7.4 Magnetic Disk Technology



- Floppy (flexible) disks are organized in the same way as hard disks, with concentric tracks that are divided into sectors.
- Physical and logical limitations restrict floppies to much lower densities than hard disks.
- A major logical limitation of the DOS/Windows floppy diskette is the organization of its file allocation table (FAT).
 - The FAT gives the status of each sector on the disk: Free, in use, damaged, reserved, etc.

23

7.4 Magnetic Disk Technology



- On a standard 1.44MB floppy, the FAT is limited to nine 512-byte sectors.
 - There are two copies of the FAT.
- There are 18 sectors per track and 80 tracks on each surface of a floppy, for a total of 2880 sectors on the disk. So each FAT entry needs at least 14 bits ($2^{14}=4096 < 2^{13} = 2048$).
 - FAT entries are actually 16 bits, and the organization is called FAT16.

24

7.4 Magnetic Disk Technology

- The disk directory associates logical file names with physical disk locations.
- Directories contain a file name and the file's first FAT entry.
- If the file spans more than one sector (or cluster), the FAT contains a pointer to the next cluster (and FAT entry) for the file.
- The FAT is read like a linked list until the <EOF> entry is found.

25

7.4 Magnetic Disk Technology

- A directory entry says that a file we want to read starts at sector 121 in the FAT fragment shown below.

FAT Index →	120	121	122	123	124	125	126	127
FAT Contents	97	124	<EOF>	1258	126	<BAD>	122	577

- Sectors 121, 124, 126, and 122 are read. After each sector is read, its FAT entry is to find the next sector occupied by the file.
- At the FAT entry for sector 122, we find the end-of-file marker <EOF>.

How many disk accesses are required to read this file?

26

7.5 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.

27

7.5 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!

28

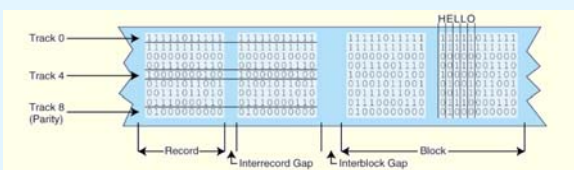
7.5 Optical Disks

- 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.
- DVDs can be thought of as quad-density CDs.
- Where a CD-ROM can hold at most 650MB of data, DVDs can hold as much as 8.54GB.
- It is possible that someday DVDs will make CDs obsolete.

29

7.6 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:



30

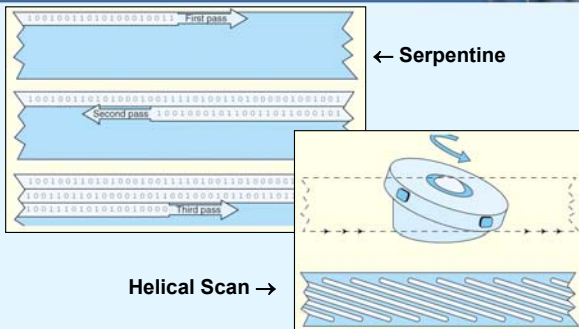
7.6 Magnetic Tape

- 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.

31

7.6 Magnetic Tape



32

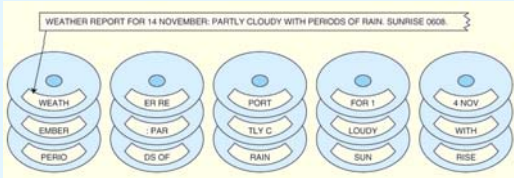
7.7 RAID

- 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.

33

7.7 RAID

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

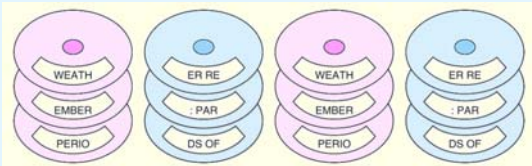


- The disadvantage of RAID 0 is in its low reliability.

34

7.7 RAID

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

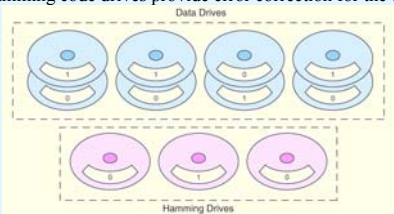


- The disadvantage of RAID 1 is cost.

35

7.7 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.

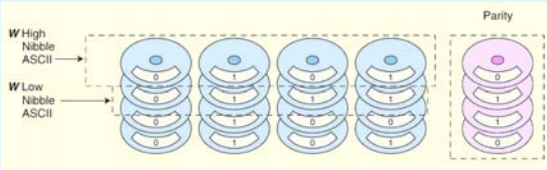


- RAID 2 performance is poor and the cost is relatively high.

36

7.7 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.

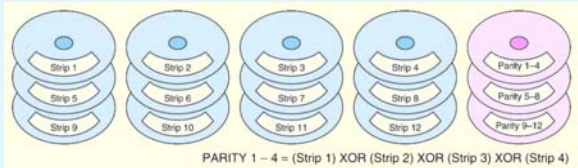


- RAID 3 is not suitable for commercial applications, but is good for personal systems.

37

7.7 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.

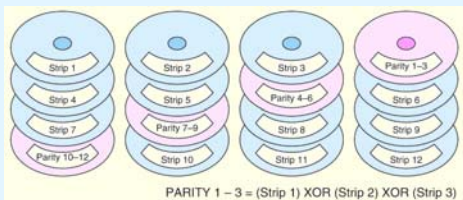


- RAID 4 would be feasible if all record blocks were the same size.

38

7.7 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.

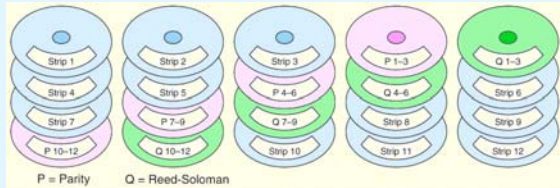


- RAID 5 is used in many commercial systems.

39

7.7 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.

40

7.7 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.
- Keep in mind that a higher RAID level does not necessarily mean a “better” RAID level. It all depends upon the needs of the applications that use the disks.

41

7.8 Data Compression

- Data compression is important to storage systems because it allows more bytes to be packed into a given storage medium than when the data is uncompressed.
- Some storage devices (notably tape) compress data automatically as it is written, resulting in less tape consumption and significantly faster backup operations.
- Compression also reduces Internet file transfer time, saving time and communications bandwidth.

42

7.8 Data Compression

- A good metric for compression is the *compression factor* (or *compression ratio*) given by:

$$\text{Compression factor} = 1 - \left[\frac{\text{compressed size}}{\text{uncompressed size}} \right] \times 100\%$$

- If we have a 100KB file that we compress to 40KB, we have a compression factor of:

$$1 - \left[\frac{40\text{KB}}{100\text{KB}} \right] \times 100\% = 60\%$$

43

7.8 Data Compression

- Compression is achieved by removing data redundancy while preserving information content.
- The information content of a group of bytes (a message) is its *entropy*.
 - Data with low entropy permit a larger compression ratio than data with high entropy.
- Entropy, H , is a function of symbol frequency. It is the weighted average of the number of bits required to encode the symbols of a message:

$$H = -P(x) \times \log_2 P(x_i)$$

44

7.8 Data Compression

- The entropy of the entire message is the sum of the individual symbol entropies.

$$\sum -P(x) \times \log_2 P(x_i)$$

- The average redundancy for each character in a message of length l is given by:

$$\sum P(x) \times l_i - \sum -P(x) \times \log_2 P(x_i)$$

45

7.8 Data Compression

- Consider the message: **HELLO WORLD !**
 - The letter **L** has a probability of $3/12 = 1/4$ of appearing in this message. The number of bits required to encode this symbol is $-\log_2(1/4) = 2$.
- Using our formula, $\sum -P(x) \times \log_2 P(x)$, the average entropy of the entire message is 3.022.
 - This means that the theoretical minimum number of bits per character is 3.022.
- Theoretically, the message could be sent using only 37 bits. ($3.022 \times 12 = 36.26$)

46

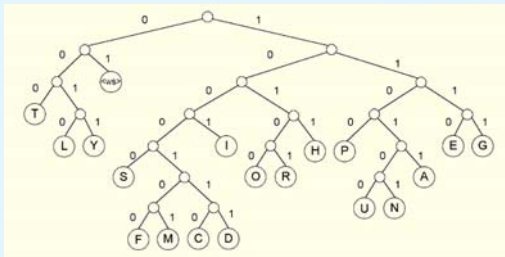
7.8 Data Compression

- The entropy metric just described forms the basis for statistical data compression.
- Two widely-used statistical coding algorithms are *Huffman coding* and *arithmetic coding*.
- Huffman coding builds a binary tree from the letter frequencies in the message.
 - The binary symbols for each character are read directly from the tree.
- Symbols with the highest frequencies end up at the top of the tree, and result in the shortest codes.

An example is shown on the next slide.

47

7.8 Data Compression



HIGGLETY PIGGLTY POP
THE DOG HAS EATEN THE MOP
THE PIGS IN A HURRY THE CATS IN A FLURRY
HIGGLETY PIGGLTY POP

48

7.8 Data Compression

- The second type of statistical coding, arithmetic coding, partitions the real number interval between 0 and 1 into segments according to symbol probabilities.
 - An abbreviated algorithm for this process is given in the text.
- Arithmetic coding is computationally intensive and it runs the risk of causing divide underflow.
- Variations in floating-point representation among various systems can also cause the terminal condition (a zero value) to be missed.

49

7.8 Data Compression

- For most data, statistical coding methods offer excellent compression ratios.
- Their main disadvantage is that they require two passes over the data to be encoded.
 - The first pass calculates probabilities, the second encodes the message.
- This approach is unacceptably slow for storage systems, where data must be read, written, and compressed within one pass over a file.

50

7.8 Data Compression

- *Ziv-Lempel* (LZ) dictionary systems solve the two-pass problem by using values in the data as a dictionary to encode itself.
- The LZ77 compression algorithm employs a text window in conjunction with a lookahead buffer.
 - The text window serves as the dictionary. If text is found in the lookahead buffer that matches text in the dictionary, the location and length of the text in the window is output.

STAR_LIGHT_ STAR_BRIGHT_FIRS
0, 5, B ←

51

7.8 Data Compression

- The LZ77 implementations include PKZIP and IBM's RAMAC RVA 2 Turbo disk array.
 - The simplicity of LZ77 lends itself well to a hardware implementation.
- LZ78 is another dictionary coding system.
- It removes the LZ77 constraint of a fixed-size window. Instead, it creates a trie as the data is read.
- Where LZ77 uses pointers to locations in a dictionary, LZ78 uses pointers to nodes in the trie.

52

7.8 Data Compression

- GIF compression is a variant of LZ78, called LZW, for Lempel-Ziv-Welsh.
- It improves upon LZ78 through its efficient management of the size of the trie.
- Terry Welsh, the designer of LZW, was employed by the Unisys Corporation when he created the algorithm, and Unisys subsequently patented it.
- Owing to royalty disputes, development of another algorithm PNG, was hastened.

53

7.8 Data Compression

- PNG employs two types of compression, first a Huffman algorithm is applied, which is followed by LZ77 compression.
- The advantage that GIF holds over PNG, is that GIF supports multiple images in one file.
- MNG is an extension of PNG that supports multiple images in one file.
- GIF, PNG, and MNG are primarily used for graphics compression. To compress larger, photographic images, JPEG is often more suitable.

54

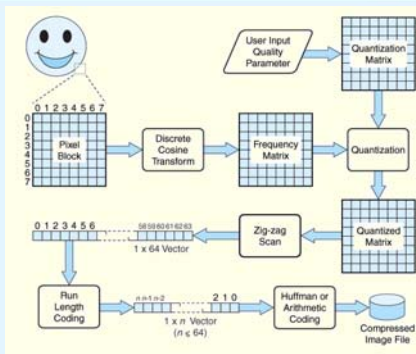
7.8 Data Compression

- Photographic images incorporate a great deal of information. However, much of that information can be lost without objectionable deterioration in image quality.
- With this in mind, JPEG allows user-selectable image quality, but even at the “best” quality levels, JPEG makes an image file smaller owing to its multiple-step compression algorithm.
- It's important to remember that JPEG is lossy, even at the highest quality setting. It should be used only when the loss can be tolerated.

The JPEG algorithm is illustrated on the next slide.

55

7.8 Data Compression



56

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.

57

Chapter 7 Conclusion

- Magnetic disk is the principal form of durable storage.
- Disk performance metrics include seek time, rotational delay, and reliability estimates.
- Optical disks provide long-term storage for large amounts of data, although access is slow.
- Magnetic tape is also an archival medium. Recording methods are track-based, serpentine, and helical scan.

58

Chapter 7 Conclusion

- RAID gives disk systems improved performance and reliability. RAID 3 and RAID 5 are the most common.
- Many storage systems incorporate data compression.
- Two approaches to data compression are statistical data compression and dictionary systems.
- GIF, PNG, MNG, and JPEG are used for image compression.

59

60