# UNIT 2 HIGHLIGHTS, DEFINITIONS

# ALGORITHM DISCOVERY AND DESIGN

## 3 Misconceptions of What CS is:

MISCONCEPTION 1: *Computer science is the study of computers.* INCOMPLETE.

Late 1950s & early 1960s. Initially considered a branch of logic & applied mathematics. Even today, branches of computer science quite distinct from the study of "real" machines. ***THEORETICAL COMPUTER SCIENCE***- study the logical and mathematical properties of problems and their solutions. Frequently, Researchers investigate problems not with actual computers but rather with ***formal models*** of computation, which are easier to study and analyze mathematically, involving pencil and paper, not circuit boards and disks.

---

MISCONCEPTION 2: *Computer science is the study of how to write computer programs.*

programming is a tool. When computer scientists design and analyze a new approach to solving a problem or create new ways to represent information, they often implement their ideas as programs to test them on an actual computer system. In computer science, it is not simply the construction of a high-quality program that is important but also the methods it embodies, the services it provides, and the results it produces.

MISCONCEPTION 3: *Computer science is the study of the uses and applications of computers and software.*

the computer scientist who is responsible for specifying, designing, building, and testing these software packages as well as the computer systems on which they run.

---

## History of Computers (Brief)

-**1930s** - Earliest theoretical work on the logical foundations of computer science.

-**1940–1946** The first general-purpose, electronic computers.

-First commercial machine, the **UNIVAC I**, did not make its appearance until **March 1951**, a date that marks the real beginning of the computer industry.

-**1947** first professional society for people in the field of computing, the Association for Computing Machinery (ACM)

-**1957** The first high-level Language – FORTRAN. Beginning of Software Industry.

- **October 1962**- First Department of Computer Science was established at Purdue University. ----

**-1964** Awarded its first M.Sc. degree **1966** First Ph.D. in computer science. **1967** Undergraduate program was established.

## 1.2The Definition of Computer Science

----- **Algorithm**- Ordered sequence of instructions that is guaranteed to solve a specific problem the central concept in computer science. Proposed by professors Norman Gibbs and Allen Tucker.

**Algorithm design process** includes the following operations:

1. Studying algorithm behavior to determine if they are **correct** and **efficient** (their **formal** & **mathematical properties**)
2. Designing / building computer systems that are **able to execute algorithms**

(their **hardware realizations**)

3. Designing programming languages / translating algorithms into these languages so that they **can be executed by the hardware** (their **linguistic realizations**)
4. Identifying important problems / designing correct and efficient software packages to solve these problems (their **applications**)

---

***All the operations used to construct algorithms belong to one of only three categories:***

1. **Sequential operations**- (Multiple) single well-defined task.
2. **Conditional operations-** Operations selected on the Boolean basis.
3. **Iterative operations** The "looping" instructions.

*If we can specify an algorithm to solve a problem, then we can automate its solution.*

---

**Abu Ja'far Muhammad ibn Musa Al-Khwarizmi (AD 780–850?)** Author of earliest mathematical textbooks ***Kitab al jabr w'al muqabala***, in English, **"The Concise Book of Calculation by Reduction." (AD 820).**, Arabic word ***al jabr*** *(algebra)* **means "reduction"**). Rendered as Algoritmi in Latin characters

**Computing Agent** the machine, robot, person, or thing carrying out the steps of an algorithm.

computer science can also be viewed as **the *science of algorithmic problem solving***.

**Unsolvable** - problems for which no generalized algorithmic solution can possibly exist.
Solution is too inefficient || Takes Too Much Time / Don't know how
***Brute Force* approach**- Iterating every possibility

---

**Ordering** - know which operation to do first and precisely which operation to do next as each step is successfully completed
algorithm must meet two criteria—must be ***unambiguous*** & ***effectively computable***

**unambiguous operation** is one that can be understood and carried out directly by the computing agent without further simplification or explanation. When an operation is unambiguous, we call it a *primitive operation*, or simply a **primitive** of the computing agent carrying out the algorithm. An algorithm must be composed entirely of primitives.

"doable" is **effectively computable**.

**infinite loop**, and it is a common error in the design of algorithms.

**Luddites**. The Luddites, named after their leader Ned Ludd of Nottingham, England, were violently opposed to this new manufacturing technology, and they burned down factories that attempted to use it.

> Babbage's proposed machine, called the **Analytical Engine**, is amazingly similar in design to a modern computer. The four components of the Analytical Engine are virtually identical in function to the four major components of today's computer systems:

| *Babbage's Term* | *Modern Terminology* |
|---|---|
| mill | arithmetic/logic unit |
| store | memory |
| operator | processor |
| output unit | input/output |

The machine, dubbed the **ENIAC** (Electronic Numerical Integrator and Calculator), was completed in 1946 (too late to assist in the war effort) and was the first fully electronic general-purpose programmable computer.

**stored program computer**. The memory unit stored only data, not instructions

**Von Neumann architecture**. instead of rewiring the machine, you would rewrite the sequence of instructions—that is, create a new program.

The period 1950–1957 (these dates are very rough approximations) is often called the *first generation* of computing.
The *second generation* of computing, roughly 1957–1965, heralded a major change in the size and complexity of computers.
This miniaturization process continued into the *third generation* of computing, which lasted from about 1965 to 1975. This was the era of the *integrated circuit*. desk-sized, and this period saw the birth of the first **minicomputer**
The *fourth generation*, roughly 1975–1985, saw the appearance of the first **microcomputer**. complete computer system could be contained on a single circuit board that you could hold in your hand.

Some of the major advancements in computing

| Generation | Approximate Dates | Major Advances |
|---|---|---|
| First | 1950–1957 | First commercial computers<br>First symbolic programming languages<br>Use of binary arithmetic, vacuum tubes for storage<br>Punched card input/output |
| Second | 1957–1965 | Transistors and core memories<br>First disks for mass storage<br>Size reduction, increased reliability, lower costs<br>First high-level programming languages<br>First operating systems |
| Third | 1965–1975 | Integrated circuits<br>Further reduction in size and cost, increased reliability<br>First minicomputers<br>Time-shared operating systems<br>Appearance of the software industry<br>First set of computing standards for compatibility between systems |
| Fourth | 1975–1985 | Large-scale and very-large-scale integrated circuits<br>Further reduction in size and cost, increased reliability |

| Generation | Approximate Dates | Major Advances |
|---|---|---|
| | | First microcomputers<br>Growth of new types of software and of the software industry<br>Computer networks<br>Graphical user interfaces |
| Fifth | 1985–? | Ultra-large-scale integrated circuits<br>Supercomputers and parallel processors<br>Laptops, tablets, smartphones, and handheld wireless devices<br>Mobile computing<br>Massive external data storage devices<br>Ubiquitous computing<br>High-resolution graphics, visualization, virtual reality<br>Worldwide networks and cloud computing<br>Multimedia user interfaces<br>Widespread use of digitized sound, images, and movies |

## Computer science

### Computer science is the study of algorithms, including

1. ***Their formal and mathematical properties.*** Level 1 of the text (Chapters 2 and 3) is titled "The Algorithmic Foundations of Computer Science." It continues the discussion of algorithmic problem solving begun in Sections 1.2 and 1.3 by introducing important mathematical and logical properties of algorithms. Chapters 2 presents the development of several algorithms that solve important technical problems—certainly more "technical" than shampooing your hair. It also looks at concepts related to the problem-solving process, such as how we discover and create good algorithms, what notation we can use to express our solutions, and how we can check to see whether our proposed algorithm correctly solves the desired problem.

   Our brute force chess example illustrates that it is not enough simply to develop a correct algorithm; we also want a solution that is efficient and that produces the desired result in a reasonable amount of time. (Would you want to market a chess-playing program that takes  years to make its first move?) Chapter 3 describes ways to compare the efficiency of different algorithms and select the best one to solve a given problem. The material in Level 1 provides the necessary foundation for a study of the discipline of computer science.
2. ***Their hardware realizations.*** Although our initial look at computer science investigated how an algorithm behaved when executed by some abstract "computing agent," we ultimately want to execute our algorithms on "real" machines to get "real" answers. Level 2 of the text (Chapters 4 and 5) is titled "The Hardware World," and it looks at how to design and construct computer systems. It approaches this topic from two quite different viewpoints.

   Chapter 4 presents a detailed discussion of the underlying hardware. It introduces the basic building blocks of computers—binary numbers, transistors, logic gates, and circuits—and shows how these elementary electronic devices can be used to construct components to perform arithmetic and logic functions such as addition, subtraction,

comparison, and sequencing. Although it is both interesting and important, this perspective produces a rather low-level view of a computer system. It is difficult to understand how a computer works by studying only these elementary components, just as it would be difficult to understand human behavior by investigating the behavior of individual cells. Therefore, Chapter 5 takes a higher-level view of computer hardware. It looks at computers not as a bunch of wires and circuits but as an integrated collection of subsystems called memory, processor, storage, input/output, and communications. It will explain in great detail the principles of the Von Neumann architecture introduced in Section 1.4.

A study of computer systems can be done at an even higher level. To understand how a computer works, we do not need to examine the functioning of every one of the thousands of components inside a machine. Instead, we need only be aware of a few critical pieces that are essential to our work. From the user's perspective, everything else is superfluous. This "user-oriented" view of a computer system and its resources is called a **virtual machine** or a virtual environment. A virtual machine is composed only of the resources that the user perceives rather than of all the hardware resources that actually exist.

This viewpoint is analogous to our level of understanding of what happens under the hood of a car. There may be thousands of mechanical components inside an automobile engine, but most of us concern ourselves only with the items reported on the dashboard—for example, oil pressure, fuel level, engine temperature. This is our "virtual engine," and that is all we need or want to know. We are all too happy to leave the remaining details about engine design to our friendly neighborhood mechanic.

Level 3 (Chapters 6, 7, and 8), titled "The Virtual Machine," describes how a virtual environment is created using a component called *system software*. Chapter 6 takes a look at the most important and widely used piece of system software on a modern computer system, the *operating system*, which controls the overall operation of a computer and makes it easier for users to access. Chapter 7 then goes on to describe how this virtual environment can extend beyond the boundaries of a single system as it examines how to interconnect individual machines into ***computer networks*** and ***distributed systems*** that provide users with access to a huge collection of computer systems and information as well as an enormous number of other users. It is the system software, and the virtual machine it creates, that makes computer hardware manageable and usable. Finally, Chapter 8 discusses a critically important component of a virtual machine—the ***security system*** that validates who you are and ensures that you are not attempting to carry out an improper, illegal, or unsafe operation. As computers become central to the management of such sensitive data as medical records, military information, and financial data, this aspect of system software is taking on even greater importance.

3. ***Their linguistic realizations***. After studying hardware design, computer organization, and virtual machines, you will have a good idea of the techniques used to design and build computers.

4. There are many programming languages, such as C++, Python, Java, and Perl, that can be used to encode algorithms. Chapter 10 provides an overview of a number of different languages and language models in current use, including the functional and parallel

models. Chapter 11 describes how a program written in a high-level programming language can be translated into the low-level machine language codes first described in Chapter 5. Finally, Chapter 12 shows that, even when we marshal all the powerful hardware and software ideas described in the first 11 chapters, problems exist that cannot be solved algorithmically. Chapter 12 demonstrates that there are, indeed, limits to computing.

5. ***Their applications***. Most people are concerned not with creating programs but with using programs, just as there are few automotive engineers but many, many drivers. Level 5, titled "Applications" (Chapters 13, 14, 15, 16), moves from *how* to write a program to *what* these programs can do.

   In Level 6, titled "Social Issues" (Chapter 17), we move to the highest level of abstraction—the view furthest removed from the computer itself—to discuss social, ethical, legal, and professional issues related to computer and information technology.

---

---

# UNIT 3 HIGHLIGHTS, DEFINITIONS

# THE EFFICIENCY OF ALGORITHMS

## 3.2 Attributes of Algorithms

First and foremost, we expect ***correctness*** from our algorithms. "Are we solving the right problem? Are we solving the problem right? Intended *use?*"

***program maintenance-*** The person who has to modify a program, either to correct errors or to expand its functionality, often is not the person who wrote the original program.

For easy Maintenance, the algorithm should be easy to understand. ***Ease of understanding***, clarity, "ease of handling", highly desirable of an algorithm.

***Elegance*** is the algorithmic equivalent of style.

**efficiency** -using fewest resources possible.

---

## 3.3 Measuring Efficiency

**benchmarking**- useful for rating one machine against another and for rating how sensitive a particular algorithm is with respect to variations in input on one particular machine.

**analysis of algorithms** - The study of the efficiency of algorithms

---

## 3.3.1 Sequential Search - the best approach available to search an unsorted list.

Number of comparisons to find NUMBER in a list of *n* numbers using sequential search

| Best Case | Worst Case | Average Case |
|---|---|---|
| 1 | *n* | *n/2* |

## 3.3.2 Order of Magnitude—Order *n*

**order of magnitude n** - Anything that varies as a constant times *n* is said to be of, written $\Theta(n)$ ,"order *n*." Algorithms are classified according in order of magnitude of their time efficiency.

## 3.3.3 Selection Sort - has O time complexity, making it inefficient on large lists. an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Very common problems: **Searching** and **Sorting**

**Comparison Formula: (n-1)/2 * n = $1/2n^2 - 1/2n$**

---

## 3.3.4 Order of Magnitude—Order $n^2$

it grows at approximately the *square* of that rate. An algorithm that does $cn^2$ work for any constant *c* is **order of magnitude** , or $\Theta(n^2)$.

One way to compare performance among different makes of computers is to give the number of arithmetic operations, such as additions or subtractions of real numbers, that each one can do in 1 second. These operations are called *floating-point operations*, and computers are often compared in terms of the number of **flops** (floating-point operations per second) they can crank out.

---

# 3.4 Analysis of Algorithms

## 3.4.1 Data Cleanup Algorithms while executing an algorithm, so we track space consumption more closely.

**The Shuffle-Left Algorithm.-** Finds non-**Legit** values and Deletes them Left-2-Right & moves Legit Values Left.

**The Copy-Over Algorithm. -** Finds non-**Legit** values and Copies Legit Values over them Right-2- Left & moves.

**The Converging-Pointers Algorithm.-** LEFT(Pointer) finds non-**Legit** values / RIGHT Points at the End of List & at Values to Copy into Non-Legit LEFT P Values. Right the decrements by 1, because Legit Value = -1

### Analysis of three data cleanup algorithms

| | 1. Shuffle-left | | 2. Copy-over | | 3. Converging-pointers | |
|---|---|---|---|---|---|---|
| | Time | Space | Time | Space | Time | Space |
| Best case | | *n* | | *n* | | *n* |
| Worst case | | *n* | | *2n* | | *n* |

| | 1. Shuffle-left | | 2. Copy-over | | 3. Converging-pointers | |
|---|---|---|---|---|---|---|
| Average case | | *n* | | | | *n* |

**If you have a very large data cleanup problem and you have average or possibly worst case data, but you also have no space concerns?** c

**Copy-over or Converging Pointers would be best. Remember that $\Theta(n^2)$ algorithms are not good choices if a $\Theta(n)$ algorithm is available.**

**If you have a very large data cleanup problem and you have average or possibly worst case data, but you also have space concerns?**

**Converging Pointers would be a good choice. See the comments on #2 on the previous slide.**

**If you know nothing about the data set--- i.e. neither its size nor its composition?**

**Converging Pointers** **is one choice for all the previous questions, it is probably the best choice.**

---

3.4.2 Binary Search- More Efficient than SEQUENTIAL SEARCH. Must be Sorted List. If list is to be searched repeatedly, it is more efficient to sort it and then use binary search

Divides List by 2. Checks if Value is Greater(Search Right) or Lesser(Search Left)

---

3.4.3 Pattern Matching forward-march algorithm performs quite well on text and patterns consisting of ordinary words.

Order-of-magnitude time efficiency summary

| Problem | Unit of Work | Algorithm | Best Case | Worst Case | Average Case |
|---|---|---|---|---|---|
| Searching | Comparisons | Sequential search | 1 | $\Theta(n)$ | $\Theta(n)$ |
| | | Binary search | 1 | $\Theta(\lg n)$ | $\Theta(\lg n)$ |
| Sorting Comparisons | and exchanges | Selection sort | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Data cleanup | Examinations and copies | Shuffle-left | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| | | Copy-over | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| | | Converging-pointers | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| Pattern matching | Character comparisons | Forward march | $\Theta(n)$ | $\Theta(m \times n)$ | |

---

## 3.5 When Things Get Out of Hand

**polynomially bounded** If the number of *nodes* and connecting *edges* is large, humans also might not "see" the solution immediately.

A collection of **nodes** and connecting **edges** is called a ***graph***.

***Hamiltonian circuit*** A path through a graph that begins and ends at the same node and goes through all other nodes exactly once.

**exponential algorithm**. Hence the trial-and-error approach to solving this Hamiltonian circuit problem is an exponential algorithm. A comparison of four orders of magnitude

| | n | | | |
|---|---|---|---|---|
| **Order** | **10** | **50** | **100** | **1,000** |
| lg *n* | 0.0003 sec | 0.0006 sec | 0.0007 sec | 0.001 sec |
| *n* | 0.001 sec | 0.005 sec | 0.01 sec | 0.1 sec |
| $n^2$ | 0.01 sec | 0.25 sec | 1 sec | 1.67 min |
| $2^n$ | 0.1024 sec | 3,570 years | 4 × 1016 centuries | *Too big to compute!!* |

Are there problems for which no polynomially bounded algorithm exists? Such problems are called **intractable**; they are solvable, but the solution algorithms all require so much work as to be virtually useless.

Problems for which no known polynomial solution algorithm exists are sometimes approached via **approximation algorithms**. These algorithms don't give the exact answer to the problem, but they provide a close approximation to a solution

---

# UNIT 4 HIGHLIGHTS, DEFINITIONS THE BUILDING BLOCKS: BINARY NUMBERS, BOOLEAN LOGIC, AND GATES

## 4.2 The Binary Numbering System

### 4.2.1 Binary Representation of Numeric and Textual Information

-virtually every computer ever built stores data—numbers, letters, graphics, images, sound-internally using the **binary numbering system**.

-Binary is a base-2 **positional numbering system**

-The two digits, 0 and 1, are frequently referred to as **bits**, a contraction of the two words *binary* digits.

*decimal-to-binary algorithm*, which is based on successive divisions by 2. Dividing the original decimal value by 2 produces a quotient and a remainder, which must be either a 0 or a 1. Record the

remainder digit and then divide the quotient by 2, getting a new quotient and a second remainder digit. The process of dividing by 2, saving the quotient, and writing down the remainder is repeated until the quotient equals 0.

-Any operation on this computer that produces an unsigned value greater than 65,535 results in the error condition called **arithmetic overflow**.

**-- Signed Numbers.** **sign/magnitude notation**-to represent signed integers, we can use the leftmost bit of a number to represent the sign, with 0 meaning positive (+) and 1 meaning negative (−).

--unwanted signed number: 10000 "negative zero,"

--computer designers tend to favor signed integer representations that do not suffer from the problem of two zeros.

--One of the most widely used is called **two's complement representation**. In the two's complement representation of signed integers, you can always represent one more negative number than positive.

--This is not as severe a problem as having two zeros, though, and two's complement is widely used for representing signed numbers inside a computer.

**--Fractional Numbers.** Shown Using **scientific notation**:

---

Initially, the most widely used code set for representing characters internally in a computer system was **ASCII**, an acronym for the American Standard Code for Information Interchange. ASCII is an international standard for representing textual information that uses 8 bits per character, so it is able to encode a total of  different symbols. These are assigned the integer values 0 to 255.

However, the code set called **Unicode**, developed in the early 1990s, has gained in popularity because it uses, at a minimum, a 16-bit representation for characters rather than the 8-bit format of ASCII. This means that it is able to represent at least  unique characters instead of the  of ASCII.

## 4.2.2 Binary Representation of Sound and Images

Sound is analog information, unlike the digital format used to represent text and numbers discussed in the previous section. In a **digital representation**, the allowable values for a given object are drawn from a finite set, such as letters {A, B, C, …, Z} or a subset of integers {0, 1, 2, 3, …, MAX}.

-In an **analog representation**, objects can take on any value.

-The **amplitude** (height) of the wave is a measure of its loudness—the greater the amplitude, the louder the sound. The **period** of the wave, designated as *T*, is the time it takes for the wave to make one complete cycle. The **frequency** *f* is the total number of cycles per unit time measured in cycles/second, also called *hertz*, and defined as . The frequency is a measure of the *pitch*, the highness or lowness of a sound.

--To store a waveform (such as the one in Figure 4.4) in a computer, the analog signal first must be **digitized**, that is, converted to a digital representation. This can be done using a technique known as **sampling**. At fixed time intervals, the amplitude of the signal is measured and stored as an integer value. The wave is thus represented in the computer in digital form as a sequence of sampled numerical amplitudes.

--The **sampling rate** measures how many times per second we sample the amplitude of the sound wave.

--The **bit depth** is the number of bits used to encode each sample.

-- The sampling process, often called ***scanning***, consists of measuring the intensity values of distinct points located at regular intervals across the image's surface. These points are called ***pixels***, short for picture elements,
--The easiest and most space-efficient approach is to mark each pixel as either white, stored as a binary 0, or black, stored as a binary 1. The only problem is that this produces a stark ***black-and-white image***
-- takes more storage, is to represent black-and-white images using a ***gray scale*** of varying intensity.
-- **raster graphics-** encode our image as a sequence of numerical pixel values, storing each row of pixels completely, from left to right, before moving down to store the next row. Each pixel is encoded as an unsigned binary value representing its gray scale intensity.
-- **RGB encoding scheme-**Most Common. To digitize color images, we still measure the intensity value of the image at a discrete set of points, but we need to store more information about each pixel. It uses one **byte**, or 8 bits, for each color, allowing us to represent an intensity range of 0 to 255 for each color. The value 0 means that there is no contribution from this color, whereas the value 255 means a full contribution of this color.
This 24-bit color-encoding scheme, ***True Color***, provides an enormous range of shades and an extremely accurate color image reproduction.
- reduce that value by using what is called a ***color palette***.
-- the storage of analog information, such as sound, images, voice, and video, is enormously space intensive, and an important area of computer science research—**data compression**
-- a simple compression technique that can be used on almost any form of data is
--***run-length encoding***- replaces a sequence of identical values , , ..., by a pair of values ($v$, $n$), which indicates that the value $v$ is replicated $n$ times
--Compression schemes are usually evaluated by their **compression ratio**, which measures how much they reduce the storage requirements of the data
--***Variable-length code sets***, which are often used to compress text but can also be used with other forms of data. a wasteful approach because some symbols occur much more frequently than others.
-**lossless compression** schemes-no information is lost in the compression, possible to exactly reproduce the original data. **Lossy compression** schemes compress data in a way that does not guarantee that all of the information in the original data can be fully and completely recreated.

## 4.2.3 The Reliability of Binary Representation

-Computers use binary representation not for any theoretical reasons but for reasons of **reliability**.
-internal representation of information must be implemented in terms of electronic quantities such as currents and voltage levels.
-**Drift**- change energy state over time. As electrical devices age, they become unreliable. Voltage Drop
**Bistable environment**- only 2 stable states separated by a huge energy barrier. Electrical systems tend to operate best this way

## 4.2.4 Binary Storage Devices- a binary computer and its internal components using any hardware device that meets the following four criteria:

1. The device has **two stable energy states** (one for a 0, one for a 1).
2. These two states are **separated by a large energy barrier** (so that a 0 does not accidentally become a 1, or vice versa).
3. It is **possible to sense state** the device is in (to see whether it is storing a 0 or a 1) **w/o destroying the stored value**.
4. Possible to **switch the state by applying a sufficient amount of energy**, 0 to a 1, or vice versa.

-- Light Switch MEETS Criteria.
--1955-1975- **Magnetic cores** used to construct computer memories. referred to as **core memory**
--**core** is a small, magnetizable, iron oxide-coated "doughnut," about 1/50 of an inch in inner diameter, with wires strung through its center hole.
**--direction** of the magnetic field of the core determined *States, 1 or 0*
When electrical current is sent through the wire in one specific direction->
left to right, the core is magnetized in a **counterclockwise direction**.* Could represent a '1'

1 **gigabyte** = 1 billion 8-bit bytes

***Transistor*** is like a light switch. OFF state- doesn't allow electricity to flow. ON state, electricity can flow. --Is a ***solid-state*** device-has no mechanical or moving parts. States are changed electronically. Allowing the transistor to be fast & extremely small.
--Transistors are made from special materials called **semiconductors**, Like: silicon & gallium arsenide.
-- **integrated circuit** or, more commonly, a **chip**--A large number of transistors & electrical conducting paths connecting them, printed photographically on a wafer of silicon to produce a device.
--MADE BY: The chip is mounted on a **circuit board**, interconnecting all the different chips (e.g., **memory, processor,** and **communications**) needed to run a computer system. Circuit board is then plugged into the computer using a set of connectors located on the end of the board.

**mask** - A standard template. Another advantage of photographic production techniques.

transistors contains three lines—two input lines (**control** and **collector**) and one output line (**emitter**), -Each line has the 1-state-high positive voltage, or the 0-state-voltage close to 0.
--The first input line, ***control*** or ***base***, is used to open or close the switch inside the transistor. If we set the control line to a 1 by applying a sufficiently high positive voltage, the switch closes and the transistor enters the ON state.
In this state, current from the input line called the ***collector*** can flow directly to the single output line called the ***emitter***, and the associated voltage can be detected by a measuring device. In the OFF state, the flow of current through the transistor is blocked and no voltage is detected on the emitter line.

***Moore's Law***- # of transistors on a circuit board has been doubling roughly every 24 months. But it has Limits, once transistors become the size of Atoms. Pertaining to: Speed & Power of Computers.

# 4.3 Boolean Logic and Gates

## 4.3.1 Boolean Logic

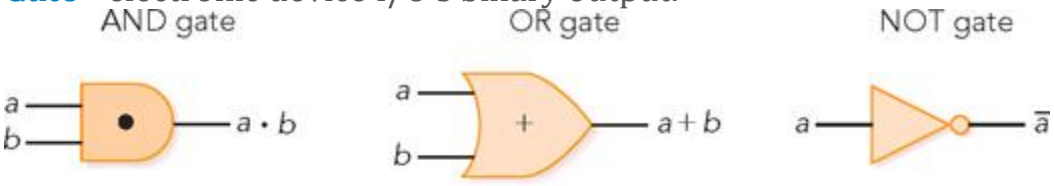
--**Boolean logic-** two logical values *true* OR *false.* The construction of computer circuits is based on the branch of mathematics and symbolic logic.
--expressed using a structure called a **truth table**,

--**hardware design**, called **logic design**-Used to construct circuits that perform operations such as adding numbers, comparing numbers, and fetching instructions.

--**Boolean expression** as any expression that evaluates to either true or false.
--**Boolean Operators:** AND, OR, NOT

## 4.3.2 Gates

**Gate-** electronic device I/O's binary output.




| a | b | a · b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| a | b | a + b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| a | ā |
|---|---|
| 0 | 1 |
| 1 | 0 |

A NOT gate can be constructed from a single transistor. ON/OFF
To construct an AND gate, connect two transistors *in* **series**. Both ON= INPUT 1
NAND gate(NOT AND)- Both ON= INPUT 0
OR Gate-Transistors connected *in* **parallel**. Either ON = INPUT 1
NOR gate(NOT OR)- Either ON= INPUT 0

# 4.4 Building Computer Circuits

## 4.4.1 Introduction

**Circuit**- Collection of logic gates that I/O's binary inputs. Outputs depend on Inputs. More properly called **combinational circuit**.

direct relationship between Boolean expressions and **circuit diagrams**
**sequential circuits-** Output depends on current input & *previous* inputs. Used to build memory units.
-Contain feedback loops in which the output of a gate is fed back as input to an earlier gate.

## 4.4.2 A Circuit Construction Algorithm

**sum-of-products algorithm** For circuit design

1. Construct the truth table describing the behavior of the desired circuit
2. While there is still an output column in the truth table, do Steps 3 through 6
3. Select an output column
4. Subexpression construction using AND and NOT gates
5. Subexpression combination using OR gates
6. Circuit diagram production
7. Done

**-Step 1: Truth Table Construction.** Determine how the circuit should behave under all possible circumstances. (binary value for each output line of the circuit for every possible combination of inputs) This information can be organized as a truth table.
-For each output of the circuit, we must specify the desired output value for every row in the truth table.

**-Step 2: Subexpression Construction Using AND and NOT Gates.** Choose any one output column of the truth table built in Step 1 and scan down that column. Every place that you find a 1 in that output column, you build a Boolean ***subexpression*** that produces the value 1 (i.e., is true) for exactly that combination of input values and no other. To build this subexpression, you examine the value of each input for this specific case. If the input is a 1, use that input value directly in your subexpression. If the input is a 0, first take the NOT of that input, changing it from a 0 to a 1, and then use that ***complemented*** input value in your subexpression. You now have an input sequence of all 1s, and if all of these modified inputs are ANDed together (two at a time, of course), then the output value is a 1.

**Step 3: Subexpression Combination Using OR Gates.** Take each of the subexpressions produced in Step 2 and combine them, two at a time, using OR gates. Each of the individual subexpressions produces a 1 for exactly one particular case where the truth table output is a 1, so the OR of the output of all of them produces a 1 in each case where the truth table has a 1 and in no other case. Consequently, the Boolean expression produced in Step 3 implements exactly the function described in the output column of the truth table on which we are working.

**Step 4: Circuit Diagram Production.** Construct the final circuit diagram. To do this, convert the Boolean expression produced at the end of Step 3 into a circuit diagram, using AND, OR, and NOT gates to implement the AND, OR, and NOT operators appearing in the Boolean expression. This circuit diagram produces an output,

Algorithms for **circuit optimization**—reduce # of gates needed to implement a circuit. For Speed.

---

## 4.4.3 Examples of Circuit Design and Construction

two circuits important to the operation of any real-world computer:

***compare-for-equality (CE)*** circuit and an ***addition (ADD)*** circuit.

**A Compare-for-Equality Circuit.** Tests two unsigned binary numbers for exact equality. If 2#'s are Equal, value 1(*true*). Used in many situations.

**An Addition Circuit.** addition circuit called ADD that performs binary addition on two unsigned *N*-bit integers. This circuit is a **full adder**.

## 4.5 Control Circuits

**control circuits**- determines the order of operations & selects correct data values to be processed.
-The sequencing and Decision-making circuits inside a computer.
-2 major types of control circuits: **multiplexers** and **decoders**, can be completely described in terms of gates and the rules of logic.

**multiplexer** is a circuit that has $2^n$ *input lines* and 1 *output line*. Its function is to select exactly one of its $2^n$ input lines and copy the binary value on that input line onto its single output line. A multiplexer chooses one specific input by using an additional set of *N* lines called **selector lines**. (Thus the total number of inputs to the multiplexer circuit is $2^n$.) The $2^n$ input lines of a multiplexer are numbered . Each of the *N* selector lines can be set to either a 0 or a 1, so we can use the *N* selector lines to represent all binary values from 000 … 0 (*N* zeros) to 111 … 1 (*N* ones), which represent all integer

values from 0 to . These numbers correspond exactly to the numbers of the input lines. Thus the binary number that appears on the selector lines can be interpreted as the identification number of the input line that is to be selected.

**decoder**- operates in the opposite way from a multiplexer. A decoder has *N* input lines numbered and $2^n$ output lines numbered.

---

---

# UNIT 5 HIGHLIGHTS, DEFINITIONS

# COMPUTER SYSTEMS ORGANIZATION

**CLASS NOTES**

---

**VON NEUMANN ARCHETECTURE** – Every Computer is built this way. CPU, Memory Subset, IO Device
**Memory**- Functional unite where data is stored/Retried
**RAM** – Organized into **Cells**, each given a unique **address**. Always take the same amount of time to access. Cell values can be read & changed. **Volatile Memory**(Gone w/o Power)
**ROM (Read-Only memory)**- Type of RAM with non-modifiable prerecorded info. BIOS ROM – an OS
**Cell size/Memory Width**- **8-bits** typically = **Byte** of Memory.
**Maximum Memory size/ Address space**- is $2^n$, where n = length of address.
*-Typically 4 Bytes (32 Bits) used for Ints.*
*-Typically 8 Bytes (64 Bits) used for Floats.*
MEMORY HOLDING CELLS- Memory Address Register (MAR)/ Memory Data Register (MDR)
**Fetch**: Retrieve from Memory **(Nondestructive Fetch)**
**Store**: Write to Memory. (**Destructive Store**)
Memory access time
-Time to Fetch/Store
-Modern RAM requires 5-10 **Nanoseconds**.
Memory System Circuits: **decoder & fetch / store controller**
Decoder converts MAR into signal to a specific memory cell
2D Model Memory (16 bits)= 8 bits = **a row**, 8 bits = **a column** INTERSECTING Searching
Fetch/Store Controller: Indicates Fetch/store Instructions

-RAM speeds increased more slowly than CPU speeds
-**Cache Memory**- Fast & Expensive.
-Built into CPU
**-Principle of Locality**-(Stack) Values close to recently accessed memory are more likely to be accessed
-Load neighbors into cache & Keeps recent values there
**Cache hit rate**: percentage of times values are found in cache.
-**Mass Storage Systems** = **Nonvolatile Memory**.
**-Direct Access Storage Device (DASDs)**
**-Sequential Access Storage Device (SASDs)**

## DASDs

– **Tracks**: Concentric rings around the disk surface
**-Sectors**- (Unit of Retrieval)Fixed size segments of Tracks.
-Time to retrieve data based on

-**Seek Time** (Record Player Needle looking for the right track.)
- **Latency** – Time it takes to get to the data
- **Transfer time**- Retrieving the Data.

-DASDs/SASDs- Orders of Magnitude slower than RAM
-**I/O Controller**- manages Data Transfer w/slow I/O devices.
-Controller sends an **interrupt signal** to processor when I/O task is done.

-ALU is part of the **processor**. –
----Contains circuits for Arithmetic / Comparison & Logic
--Contains **registers**: high speed dedicated memory connected to circuits for Holding Operands. Doesn't have an address. 5-10X faster than RAM
--**Data Path** - registers->circuits || circuits->registers.

---

--**functional units** or subsystems that perform tasks such as instruction processing, information storage, computation, and data transfer. The branch of computer science that studies computers in terms of their major functional units is **computer organization**

--a different viewpoint, a different perspective, a different **level of abstraction**. abstracting away" of unnecessary detail. **hierarchy of abstractions**

## 5.2 The Components of a Computer System

--The structure and organization of virtually all modern computational devices are based on a single theoretical model called the **Von Neumann architecture**
The Von Neumann architecture is based on the following three characteristics:

- Four major subsystems called **memory, input/output**, the **arithmetic/logic unit (ALU)**, and the **control unit**. **central processing unit** or CPU- The ALU and the control unit are often bundled
- The **stored program** concept, in which the instructions to be executed by the computer are represented as binary values and stored in memory.
* The **sequential execution of instructions**, in which one instruction at a time is fetched from memory and passed to the control unit, where it is decoded and executed.

### 5.2.1 Memory and Cache

--**Memory** is the functional unit of a computer that stores and retrieves instructions and data.

-- Computer memory uses an access technique called ***random access***, and the memory unit is frequently referred to as **random access memory (RAM)**. RAM has the following three characteristics:

- Memory is divided into fixed-size units called **cells**, and each cell is associated with a unique identifier called an **address**. These addresses are the unsigned integers 0, 1, 2, …, MAX.
- All accesses to memory are to a specified address, and we must always fetch or store a complete cell—that is, all the bits in that cell. The cell is the minimum unit of access.
- The time it takes to fetch or store the contents of a single cell is the same for all cells in memory.

--**Read-only memory (ROM)** (EX: OS) is a special type of random access memory into which information has been prerecorded during manufacture. This information cannot be modified or removed, only fetched. ROM is used to hold important system instructions and data in a place where a user cannot accidentally or intentionally overwrite them.)

-- the memory unit is made up of cells that contain a fixed number of binary digits. The number of bits per cell is called the cell size or the **memory width**, and it is usually denoted as W.

-- The two basic memory operations are *fetching* and *storing*, and they can be described formally as follows:

- Meaning: Fetch a copy of the contents of the memory cell with the specified *address* and return those contents as the result of the operation. The original contents of the memory cell that was accessed are unchanged. This is termed a **nondestructive fetch**. Given the preceding diagram, the operation Fetch(42) returns the number 1. The value 1 remains in address 42.
- *Store(address, value)*

Meaning: Store the specified *value* into the memory cell specified by *address*. The previous contents of the cell are lost. This is termed a **destructive store**. The operation Store(42, 2) stores the value 2 into cell 42, overwriting the previous value 1.

-- **fetch/store controller**. This unit determines whether we put the contents of a memory cell into the MDR (a fetch operation) or put the contents of the MDR into a memory cell (a store operation).

-- memory fetch operation described earlier. Instead, it carries out the following three steps:

1. Look first in cache memory to see whether the information is there. If it is, then the computer can access it at the higher speed of the cache.
2. If the desired information is not in the cache, then access it from RAM at the slower speed, using the fetch operation described earlier.
3. Copy the data just fetched into the cache along with the *k* immediately following memory locations. If the cache is full, then copy the new data into the cache, to overwrite some of the older items that have not recently been accessed.

-- Furthermore, assume that the information we need is in the cache 70% of the time, a value called the **cache hit rate**.

## 5.2.2 Input/Output and Mass Storage

--The **input/output (I/O)** units are the devices that allow a computer system to communicate and interact with the outside world as well as store information for the long term.

-- **volatile memory**—the information disappears when the power is turned off. Without some type of long-term, **nonvolatile memory**, information could not be saved between shutdowns of the machine.

--Nonvolatile storage is the role of **mass storage systems** such as disks, flash drives, and tapes. (Today, a good deal of long-term data storage is no longer on local I/O devices but on remote data servers in special locations called **data centers**. This type of *cloud storage*

Fundamental characteristics of *random* access:

1. Every memory cell has a unique address.
2. It takes the same amount of time to access every cell.

-- A magnetic disk stores information in units called **sectors**, each of which contains an address and a data block containing a fixed number of bytes

-- A fixed number of these sectors are placed in a concentric circle on the surface of the disk, called a **track**

-- **I/O controller**. An I/O controller is like a special-purpose computer whose responsibility is to handle the details of input/output and to compensate for any speed differences between I/O devices and other

parts of the computer. It has a small amount of memory, called an *I/O buffer*, and enough *I/O control and logic* processing capability to handle the mechanical functions of the I/O device, such as the read/write head, paper feed mechanism, and screen display. It is also able to transmit to the processor a special hardware signal, called an **interrupt signal**, when an I/O operation is done.

## 5.2.3 The Arithmetic/Logic Unit

The **arithmetic/logic unit (ALU)** is the subsystem that performs such mathematical and logical operations as addition, subtraction, and comparison for equality.
-- ALU and the control unit (discussed in the next section) have become fully integrated into a single component called the **processor** (the CPU).
-- The ALU is made up of three parts: the **registers**, the **interconnections between components**, and **ALU circuitry**. Together these components are called the **data path**.

-- A **register** is a special high-speed storage cell that holds the operands of an arithmetic operation and that, when the operation is complete, holds its result.

-- Registers & random access memory cells differences:

- They do not have a numeric memory address but are accessed by a special register designator such as A, X, or R0.
- They can be accessed much more quickly than regular memory cells. Because there are only a few registers (typically, a few dozen up to a few hundred), it is reasonable to utilize the expensive circuitry needed to make the fetch and store operations 5–10 times faster than regular memory cells, of which there will be billions.
- They are not used for general-purpose storage but for specific purposes such as holding operands for an upcoming arithmetic computation.

-- a path for electrical signals is termed a **bus**.

## -- 5.2.4 The Control Unit -

The most fundamental characteristic of the Von Neumann architecture is the **stored program**—a sequence of machine language instructions stored as binary values in memory. **control unit** tasks:

- (1) *fetch* from memory the next instruction to be executed,
- (2) *decode* it—that is, determine what is to be done, and

*(3) *execute* it by issuing the appropriate command to the ALU, memory, or I/O controllers.

--**Machine Language Instructions.** The instructions that can be decoded and executed by the control unit of a computer are represented in **machine language**.
--The set of all operations that can be executed by a processor is called its **instruction set**

--One approach to designing instruction sets is to make them as small, fast , & simple as possible, with perhaps as few as 30–50 instructions. R*educed instruction set computers* or **RISC machines**.

--The opposite philosophy is to include a much larger number, say 300–500, of very powerful instructions in the instruction set, called *complex instruction set computers*, or **CISC machines**, and they

are designed to directly provide a wide range of powerful features so that finished programs for these processors are shorter.

--Machine language instructions can be grouped into four basic classes called **data transfer, arithmetic, compare,** and **branch**.

--*Data transfer*. Operations move information between / within different components of the computer
--*Arithmetic*. These operations cause the arithmetic/logic unit to perform a computation.
--*Compare*. These operations compare two values and set an indicator on the basis of the results of the compare. Most Von Neumann machines have a special set of bits inside the processor called *condition codes* (or a special register called a *status register* or *condition register*); these bits are set by the compare operations.
--*Branch*. These operations alter the normal sequential flow of control. The normal mode of operation of a Von Neumann machine is *sequential*. Branch is based on the current settings of the condition codes & almost always preceded by either a compare instruction || other instructions to set the condition codes.

--**Control Unit Registers and Circuits.** It is the task of the control unit to fetch and execute instructions / To accomplish this task, the control unit relies on two special registers called the **program counter (PC)** and the **instruction register (IR)** and on an *instruction decoder circuit*.

# --5.3Putting the Pieces Together—the Von Neumann Architecture

- Memory (Figure 5.8)
- Input/output (Figure 5.15)
- ALU (Figure 5.19)
- Control unit (Figures 5.22 and 5.23)

--This repetition of the fetch/decode/execute phase is called the *Von Neumann cycle*.

1. --*Fetch phase*. During the fetch phase, the control unit gets the next instruction from memory and moves it into the IR. The fetch phase is the same for every instruction and consists of the following four steps.

   1. PC → MAR Send the address in the PC to the MAR register.
   2. FETCH Initiate a fetch operation using the address in the MAR. The contents of that cell are placed in the MDR.
   3. MDR → IR Move the instruction in the MDR to the instruction register so that we are ready to decode it during the next phase.
   4. PC + 1 → PC Send the contents of the PC to the incrementor and put it back. This points the PC to the next instruction.

2. The control unit now has the current instruction in the IR and has updated the program counter so that it will correctly fetch the next instruction when the execution of this instruction is completed. It is ready to begin decoding and executing the current instruction.

3. *Decode phase*. Decoding the instruction is simple because all that needs to be done is to send the op code portion of the IR to the instruction decoder, which determines its type. The op code is the 4-bit binary value in the first column of Figure 5.25.

1. → instruction decoder
   The instruction decoder generates the proper control signals to activate the circuitry to carry out the instruction.

4. *Execution phase*. The specific actions that occur during the execution phase are different for each instruction. Therefore, there will be a unique set of circuitry for each of the distinct instructions in the instruction set. The control unit circuitry generates the necessary sequence of control signals and data transfer signals to the other units (ALU, memory, and I/O) to carry out the intent of this instruction. The following examples show what signals and transfers take place during the execution phase of some of the instructions in Figure 5.25 using the Von Neumann model of Figure 5.24.

---

--The unit you might be most familiar with is *clock speed*, measured in billions of cycles per second, called gigahertz (GHz). A more accurate measure of machine speed is *instruction rate*, measured in MIPS, an acronym for millions of instructions per second, or GIPS, billions of instructions per second.
--For these machines, a better measure of speed might be the *floating-point instruction rate*, measured in GFLOPS—billions of floating-point operations per second, TFLOPS, trillions of floating point operations per second, or PFLOPS, quadrillions of floating point operations per second.

## ---5.4Non–Von Neumann Architectures

--The inability of the sequential one-instruction-at-a-time Von Neumann model to handle today's large-scale problems is called the **Von Neumann bottleneck**, and it is a major problem in computer organization.
--To solve this problem, computer engineers are rethinking many of the fundamental ideas presented in this chapter, and they are studying nontraditional approaches to computer organization called **non-Von Neumann architectures**.
--One of the most important areas of research in these non–Von Neumann architectures is based on the following fairly obvious principle: *If you cannot build something to work twice as fast, build it to do two things at once. The results will be identical.*

--From this comes the principle of **parallel processing**—building computers not with one processor, but with hundreds, thousands, or even tens of thousands.

--Most large-scale parallel processors use architecture called **MIMD parallel processing** (*m*ultiple *i*nstruction stream/*m*ultiple *d*ata stream), called **cluster computing**. It's a computer system has multiple, independent processors each with its own primary memory unit, and every processor is capable of executing its own separate program in its own private memory at its own rate.

--Each of the processors tackles a small part of the overall problem and then communicates its result to the other processors via the *interconnection network*, a communications system that allows processors to exchange messages and data.
--MIMD parallelism is also a scalable architecture. *Scalability* means that, at least theoretically, it is possible to match the number of processors to the size of the problem.

--it is possible to address and solve massive problems by utilizing the resources of idle computers located around the world, regardless of whom they belong to. This realization led to an exciting new form of large-scale parallelism called **grid computing**.
--The field of **parallel algorithms**, the study of techniques that make efficient use of parallel architectures, is an important branch of research in computer science.

--One of the most fascinating (and complex) models of non-Von Neumann computing is called **quantum computing**. In a "regular" (i.e., Von Neumann) machine an individual bit of data is always in a well-defined state–either a 0 or a 1. However, quantum computers are built using the quantum mechanical principle called *superposition*, in which a single bit of data, now called a *qubit*, can be either a 0 or a 1 or *both* a 0 and a 1 simultaneously!

--Component that creates this kind of friendly, problem-solving environment is called *system software*. It is an intermediary between the user and the hardware components of the Von Neumann machine.
--
--

---

**3 buses**- **Address** (1-Directional)/ **Data** / **Control**
Memory Unit -> Contains MAR/ Memory Decoder Circuit/ RAM / MDR / Fetch+Store Controller
ALU -> Contains Registers
I/O -> Contains I/O Controller / IO Device

**Control Unit** - (Fetch/Decode/Execute [Until HALT]) works with Store Programs.
Called **Von Neumann Cycle**.

| **CONTAINS: Program Counter (PC)** & **Instruction Register (IR)** & Instruction Decoder Circuit |
|---|
| **Operation Code**- Tells computer what operation<br>**Addresses** = Register Location<br>**Memory can be: $2^n - 1$**<br>Processors have different Languages(Instruction Sets)<br>RISC (Reduced Instruction Set Computing) – Optimized / Easy<br>CISC (Complex ^^) can do many jobs / Complex<br>Types of Instructions: Data Transfer / AL(U) / Comparison / Branch |
| **Instruction Register (IR)-** Gets OP Code & sends to Decoder Circuit |
| Made faster by **Cache** (Close Memory) Cache **Hit Rate** = 70-80% of time, has info needed. |

---

| EXAM 1 |
|---|

6 ?'s on Each Chapter
5 ?'s on Python (HW#3) / Unit Tests

CH1 – ALGORITHMS-Correct & Efficient
-Sequential- (Computing IO) /Conditional(IF)/Iterative(Loops)
-Write Pseudocode
-Searching Methods

CH3 – Big O Notation - Which Algorithm to use (Theta)-> Efficiency[Order of magnitude] / Space
BO Notation-> Crossing Point = O(0)

Algorithm Complexity – Not Time Complexity
BINARY: twos complement(How) = Invert Bits + 1
Data Compression- Images & Sounds {Lossy || Lossless}

Transistors(NOT Gate) – On / Off. Pulls Ground for 0.
AND(Multiply) & OR(Add) Gates & Truth Tables / Opposites -> NAND / NOR

Von Neumann Architecture = Memory/IO/ALU/Control Unit