SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

21CSS201T
COMPUTER ORGANIZATION AND ARCHITECTURE

UNIT-1

Computer Architecture 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.

COMPUTER ARCHITECTURE AND ORGANIZATION OVERVIEW

- A modern computer is an electronic, digital, general purpose computing machine that automatically follows a step-by-step list of instructions to solve a problem. This step-by step list of instructions that a computer follows is also called an algorithm or a computer program.
- Why to 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.
- Computer organization
  - Encompasses all physical aspects of computer systems.
  - E.g., circuit design, control signals, memory types.
  - *How does a computer work?*

COMPUTER ARCHITECTURE AND ORGANIZATION OVERVIEW

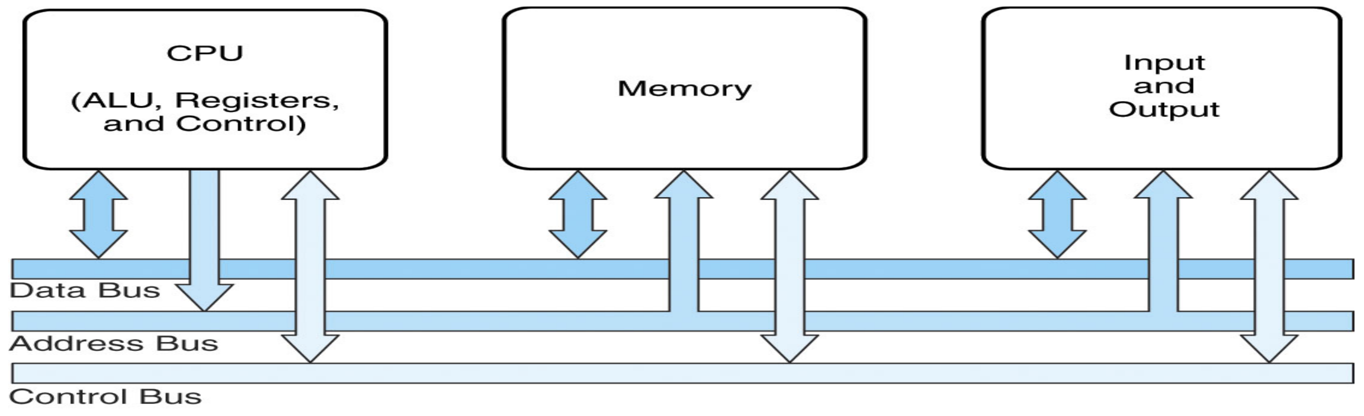
- Focuses on the **structure(**the way in which the components are interrelated**)** and behavior of the computer system and refers to the logical aspects of system implementation as seen by the programmer
- Computer architecture includes many elements such as

  instruction sets and formats, operation codes, data types, the number and types of registers, addressing modes, main memory access methods, and various I/O mechanisms.
- The architecture of a system directly affects the logical execution of programs.
- The computer architecture for a given machine is the combination of its hardware components plus its instruction set architecture (ISA).
- The ISA is the interface between all the software that runs on the machine and the hard
- **Studying computer architecture helps us to answer the question: How do I design a computer?**

COMPUTER ARCHITECTURE AND ORGANIZATION OVERVIEW

- In the case of the IBM, SUN and Intel ISAs, it is possible to purchase processors which execute the same instructions from more than one manufacturer
- All these processors may have quite different internal organizations but they all appear identical to a programmer, because their **instruction sets** are the same
- Organization & Architecture enables a family of computer models
  - Same Architecture, but with differences in Organization
  - Different price and performance characteristics
- When technology changes, only organization changes.
- This gives code compatibility (backwards)

COMPUTER COMPONENTS



At the most basic level, a computer is a device consisting of 3 pieces

A processor to interpret and execute programs

A memory (Includes Cache, RAM, ROM) to **store both data and program instructions**

A mechanism for transferring data to and from the outside world.

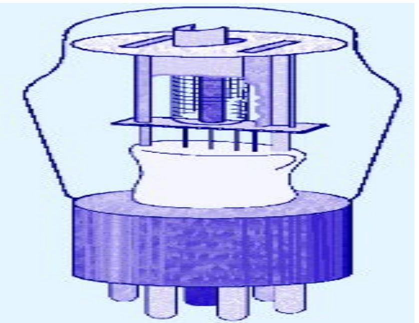
- I/O to communicate between computer and the world
- Bus to move info from one computer component to another

Contd..

- 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.

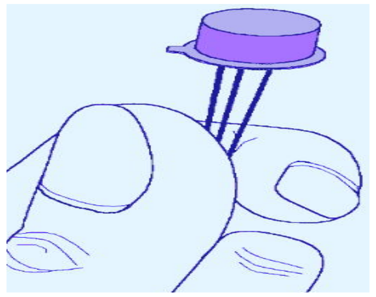
1ST GENERATION COMPUTERS

- Used vacuum tubes for logic and storage (very little storage available)
- A vacuum-tube circuit storing 1 byte
- Programmed in machine language
- Often programmed by physical connection (hardwiring)
- Slow, unreliable, expensive
- The ENIAC – often thought of as the first programmable electronic computer – 1946
- 17468 vacuum tubes, 1800 square feet, 30 tons



2nd Generation Computers

- Transistors replaced vacuum tubes
- Magnetic core memory introduced
- Changes in technology brought about cheaper and more reliable computers (vacuum tubes were very unreliable)
- Because these units were smaller, they were closer together providing a speedup over vacuum tubes
- Various programming languages introduced (assembly, high-level)
- Rudimentary OS developed
- The first supercomputer was introduced, CDC 6600 ($10 million)



3RD GENERATION COMPUTERS
INTEGRATED CIRCUIT (IC)

The ability to place circuits onto silicon chips

- Replaced both transistors and magnetic core memory
- Result was easily mass-produced components reducing the cost of computer manufacturing significantly
- Also increased speed and memory capacity
- Computer families introduced
- Minicomputers introduced
- More sophisticated programming languages and OS developed
- Popular computers included PDP-8, PDP-11, IBM 360 and Cray produced their first supercomputer, Cray-1
- Silicon chips now contained both logic (CPU) and memory
- Large-scale computer usage led to time-sharing OS

4th Generation Computers 1971-Present: Microprocessors

- Miniaturization took over
  - From SSI (10-100 components per chip) to
  - MSI (100-1000), LSI (1,000-10,000), **VLSI** (10,000+)

- Thousands of ICs were built onto a single silicon chip(VLSI), which allowed Intel, in 1971, to
  - create the world's first microprocessor, the 4004, which was a fully functional, 4-bit system that ran at 108KHz.
  - Intel also introduced the RAM chip, accommodating 4Kb of memory on a single chip. This allowed computers of the **4th** generation to become smaller and faster than their solid-state predecessors
  - Computers also saw the development of GUIs, the mouse and handheld devices

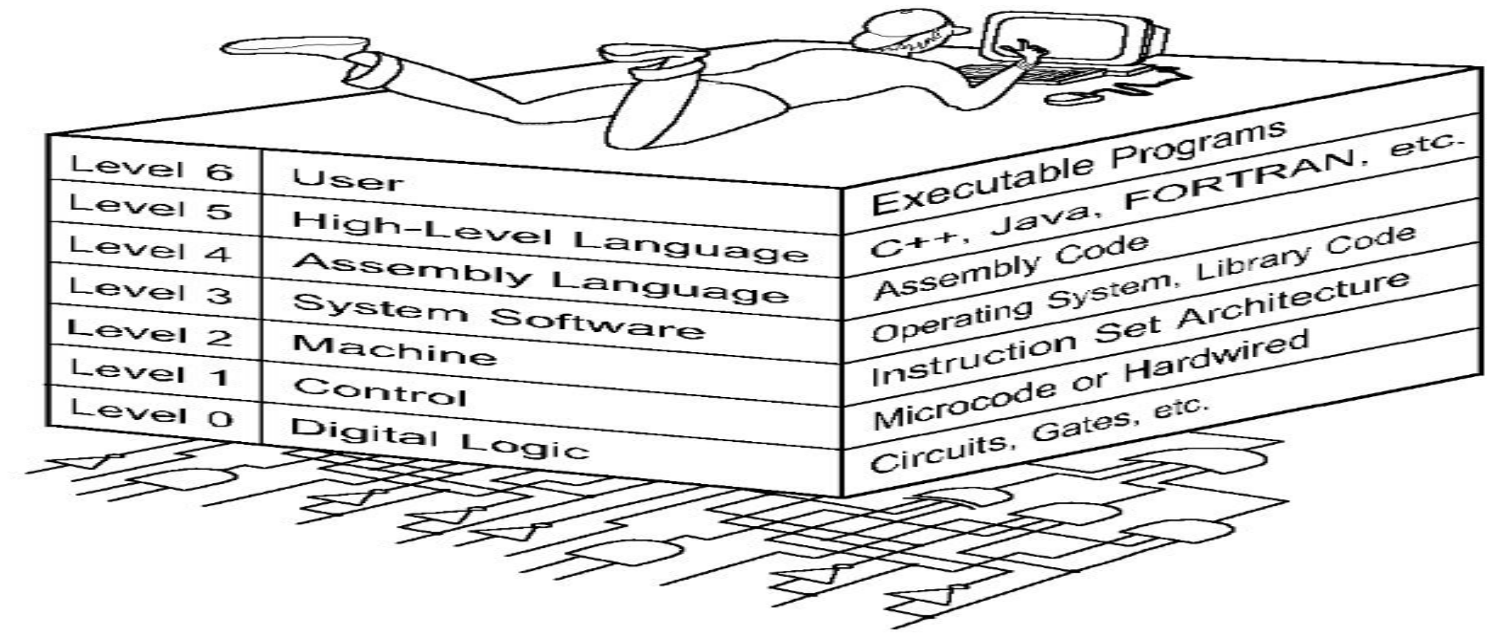
The Computer Level Hierarchy

- Through the principle of abstraction, we can imagine the machine to be built from a hierarchy of levels, in which each level has a specific function and exists as a distinct hypothetical Machine
- Abstraction is the ability to focus on important aspects of a situation at a higher level while ignoring the underlying complex details
- We call the hypothetical computer at **each level** a ***virtual machine***.
- Each level's virtual machine executes its own particular set of instructions, calling upon machines at lower levels to carry out the tasks when necessary

The Computer Level Hierarchy

Level 6: The User Level

- Composed of **applications** and is the level with which everyone is most familiar.
- At this level, we run programs such as word processors, graphics packages, or games.
- The lower levels are nearly invisible from the User Level.



The Computer Level Hierarchy

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
- These languages must be translated to a language the machine can understand. (using compiler / interpreter)
- **Compiled languages are translated into assembly** language and then assembled into machine code. (They are translated to the next lower level.)
- The user at this level sees very little of the lower levels

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
- As previously mentioned, compiled higher-level languages are first translated to assembly, which is then directly translated to machine language. This is a one-to-one translation, meaning that one assembly language instruction is translated to exactly one machine language instruction.
- By having separate levels, we reduce the semantic gap between a high-level language and the actual machine language

The Computer Level Hierarchy

Level 3: System Software Level

- deals with **operating system instructions**.
- This level is responsible for multiprogramming, protecting memory, synchronizing processes, and various other important functions.
- Often, instructions translated from assembly language to machine language are passed through this level unmodified

The Computer Level Hierarchy

Level 2: Machine Level

- Consists of instructions (ISA)that are particular to the architecture of the machine
- Programs written in machine language need no compilers, interpreters, or assemblers

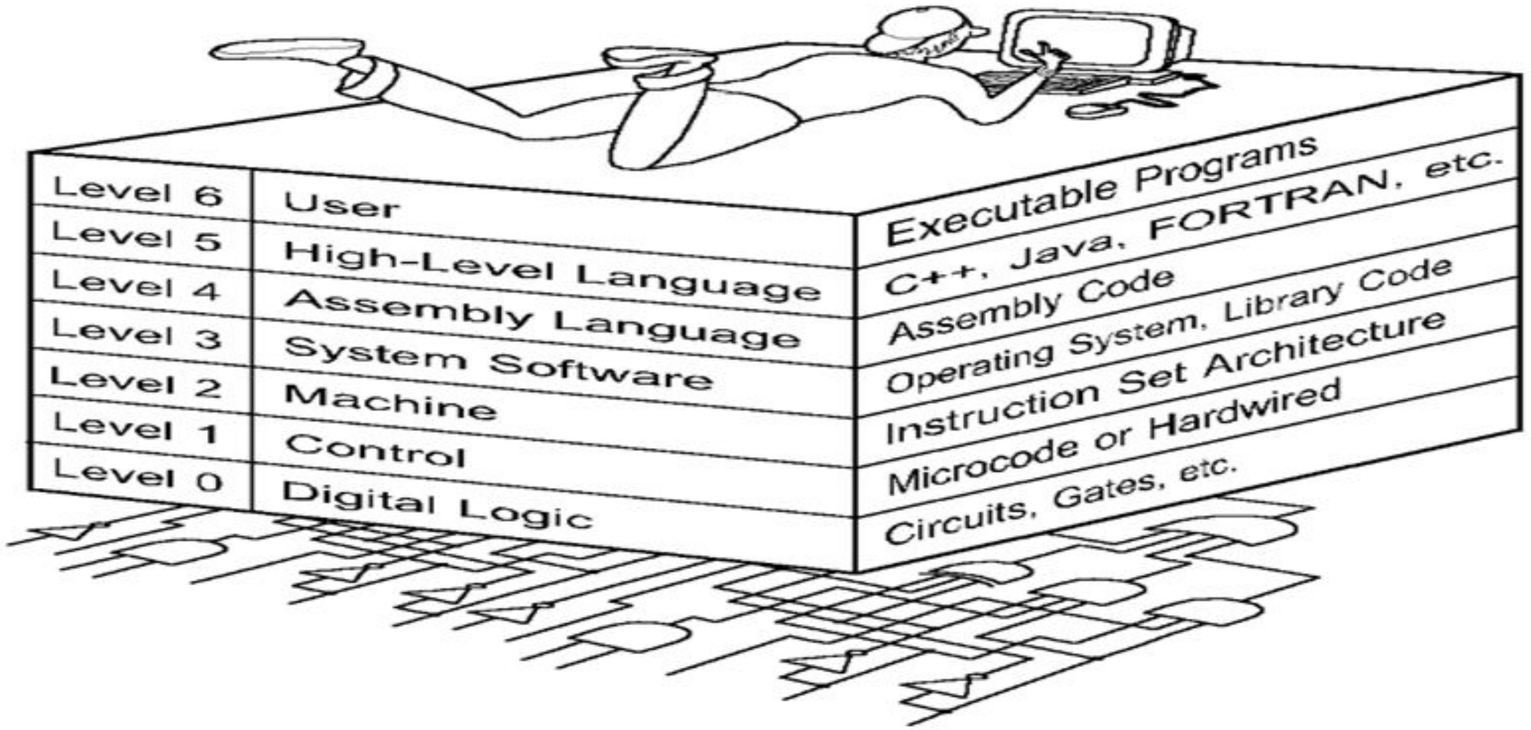
Level 1: Control Level

- 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 instruction

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



Level 6 User Executable Programs
Level 5 High-Level Language C++, Java, FORTRAN, etc.
Level 4 Assembly Language Assembly Code
Level 3 System Software Operating System, Library Code
Level 2 Machine Instruction Set Architecture
Level 1 Control Microcode or Hardwired
Level 0 Digital Logic Circuits, Gates, etc.

The Von Neumann Architecture

- Named after John von Neumann, Princeton, he designed a computer architecture whereby data and instructions would be retrieved from memory, operated on by an ALU, and moved back to memory (or I/O)
- This architecture is the basis for most modern computers (only parallel processors and a few other unique architectures use a different model)

The Von Neumann Architecture

- There is a single pathway used to move both data and instructions between memory, I/O and CPU
    - the pathway is implemented as a bus
    - the single pathway creates a bottleneck
        - known as the *von Neumann bottleneck*

- A **variation** of this architecture is the *Harvard architecture* which separates data and instructions into **two pathways**
- **Another variation**, used in most computers, **is the system bus version** in which there are different buses between CPU and memory and memory and I/O

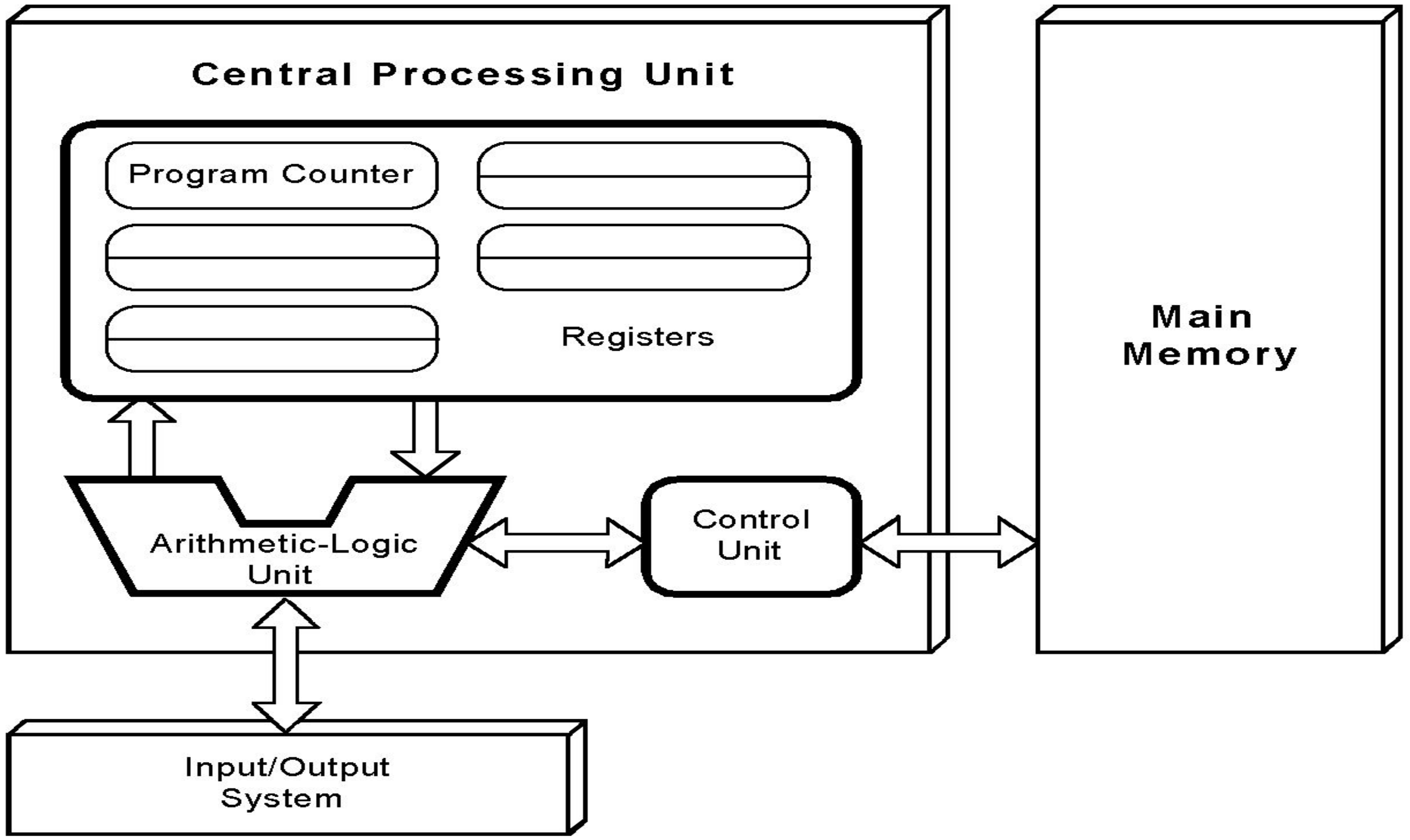
Fetch-execute cycle

- The von Neumann architecture operates on the *fetch-execute cycle*
  - Fetch an instruction from memory as indicated by the Program Counter register
  - Decode the instruction in the control unit
  - Data operands needed for the instruction are fetched from memory
  - Execute the instruction in the ALU storing the result in a register
  - Move the result back to memory if needed

THE VON NEUMANN MODEL

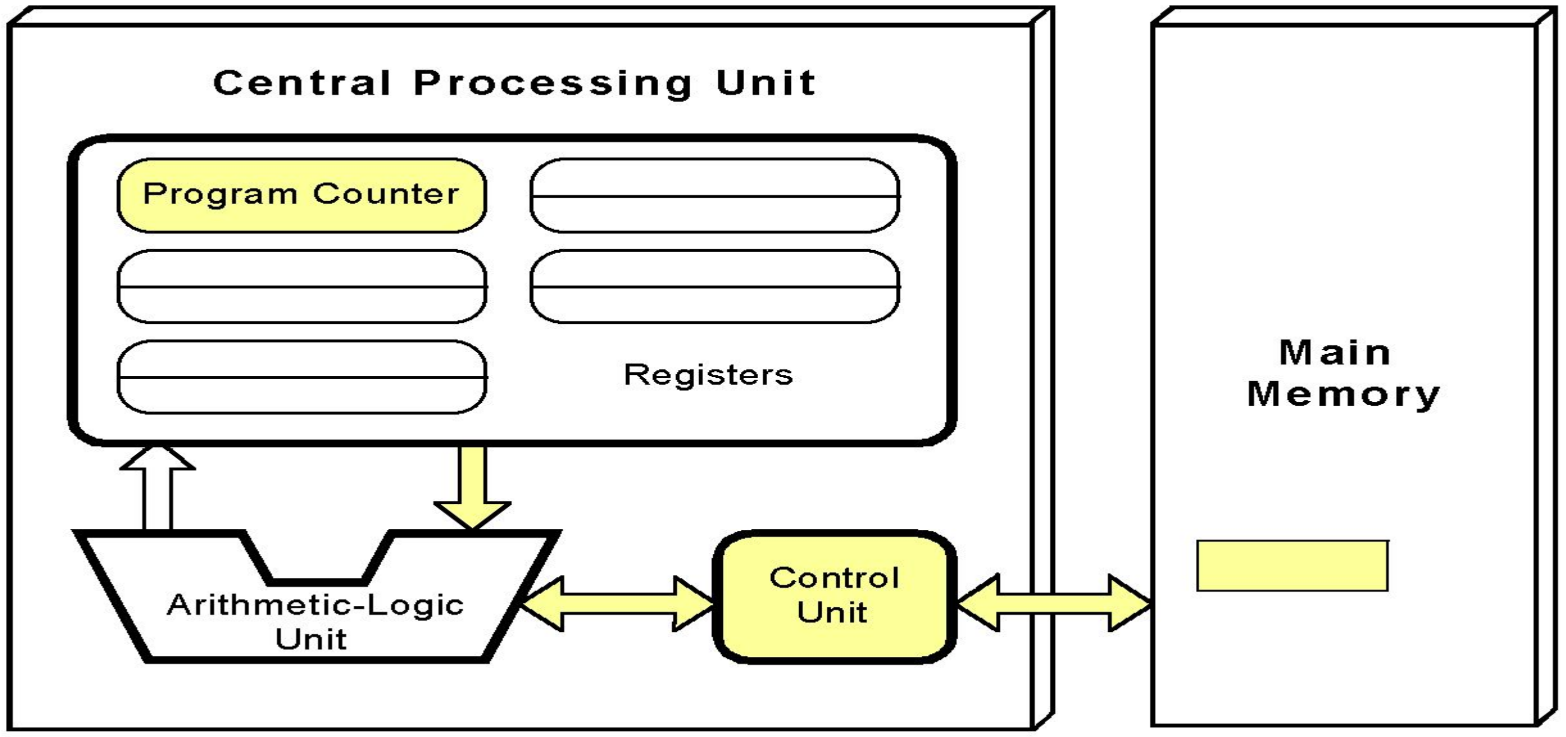
- This is a general depiction of a von Neumann system:

- These computers employ a fetch-decode-execute cycle to run programs as follows . . .



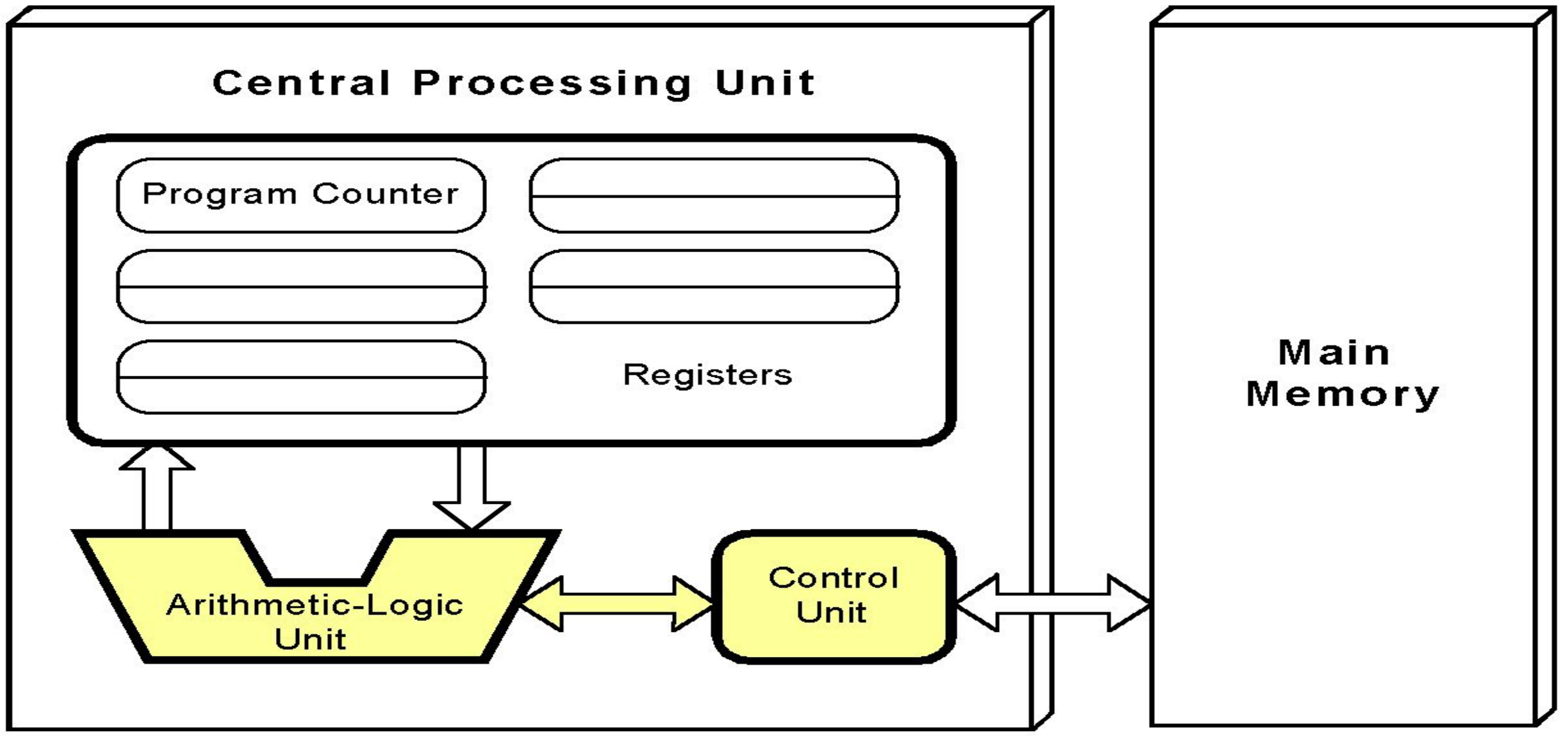
THE VON NEUMANN MODEL

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



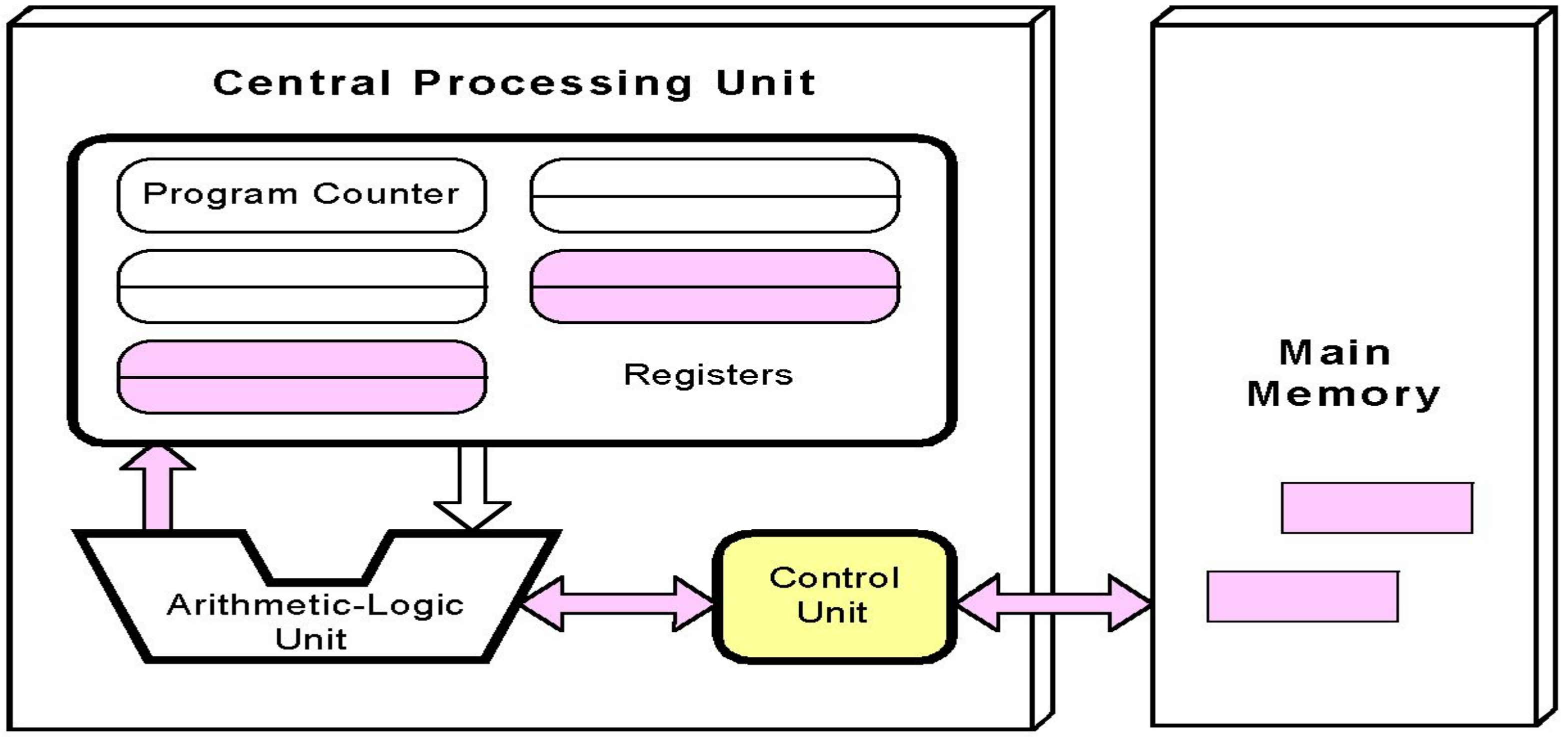
THE VON NEUMANN MODEL

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



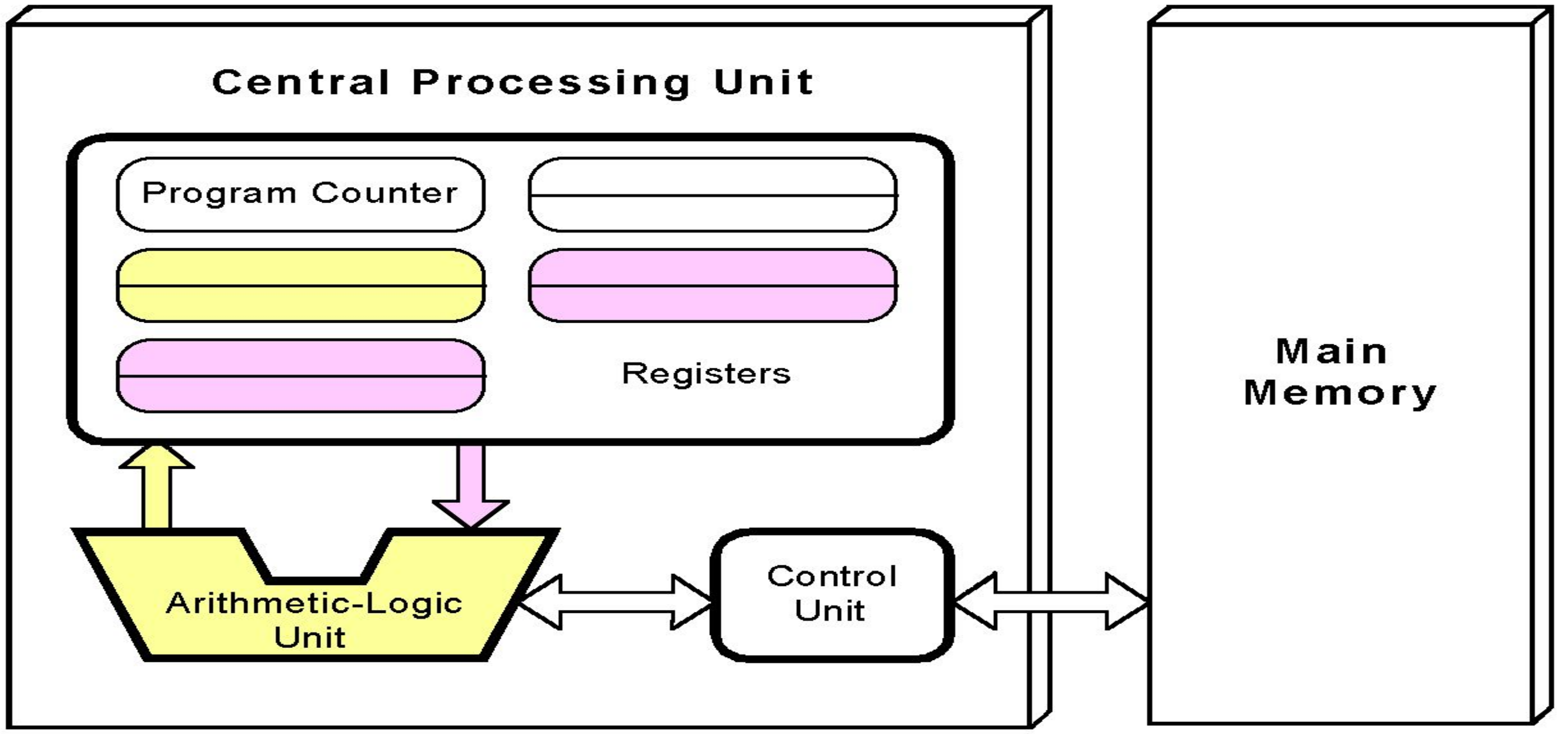
THE VON NEUMANN MODEL

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



THE VON NEUMANN MODEL

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



Non-Von Neumann Models

- Conventional stored-program computers have undergone many incremental improvements over the years
  - **specialized buses**
  - **floating-point units**
  - **cache memories**
- But enormous improvements in computational power require departure from the classic von Neumann architecture
- **Adding processors is one approach**

Non-Von Neumann Models

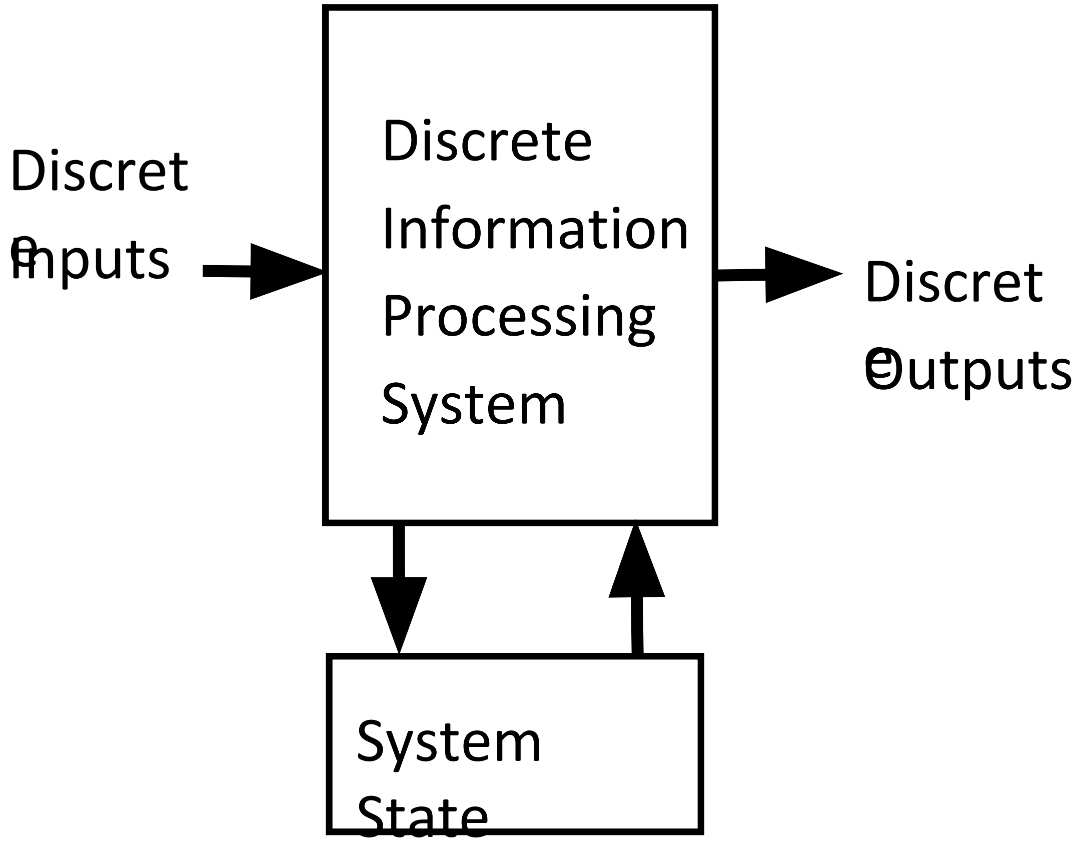
- In the late 1960s, high-performance computer systems were equipped with dual processors to increase computational throughput.
- In the 1970s supercomputer systems were introduced with 32 processors.
- Supercomputers with 1,000 processors were built in the 1980s.
- In 1999, IBM announced its Blue Gene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

Introduction to Number System and Logic Gates

- Number Systems- Binary, Decimal, Octal, Hexadecimal
- Codes- Grey, BCD,Excess-3,
- ASCII, Parity
- Binary Arithmetic- Addition, Subtraction, Multiplication, Division using Sign Magnitude
- 1’s compliment, 2’s compliment,
- BCD Arithmetic;
- Logic Gates-AND, OR, NOT, NAND,
- NOR, EX-OR, EX-NOR

Digital System

- Takes a set of discrete information inputs and discrete internal information (system state) and generates a set of discrete information outputs.

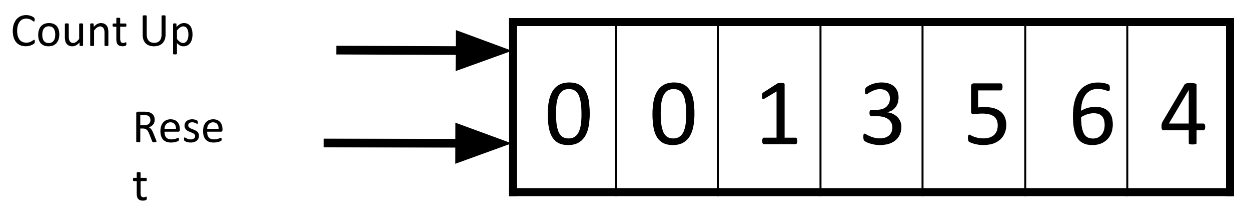


Types of Digital Systems

- No state present
  - Combinational Logic System
  - Output = Function(Input)
- State present
  - State updated at discrete times
    => Synchronous Sequential System
  - State updated at any time
    =>Asynchronous Sequential System
  - State = Function (State, Input)
  - Output = Function (State)
    or Function (State, Input)

Digital System Example

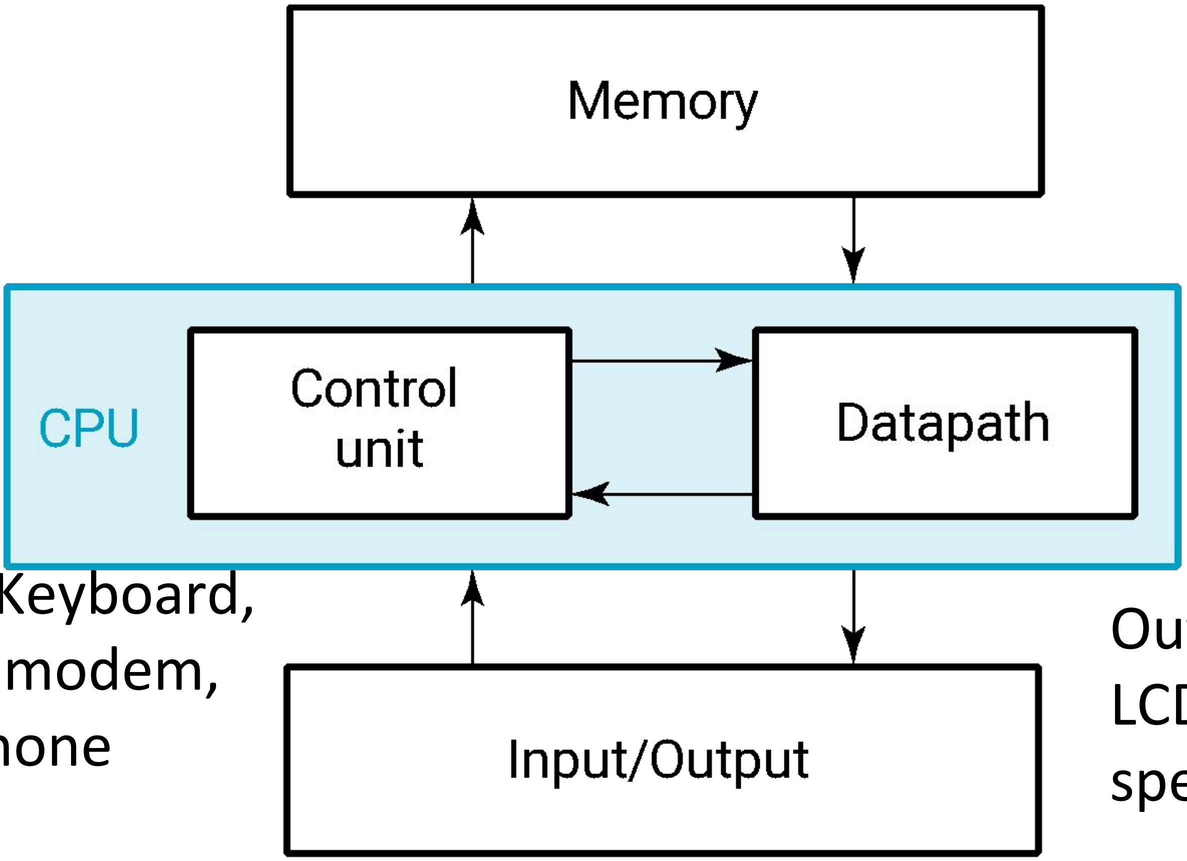
A Digital Counter (e. g., odometer):



Inputs:	Count Up, Reset
Outputs:	Visual Display
State :	"Value" of stored digits

Synchronous or Asynchronous?

A Digital Computer Example



Inputs: Keyboard, mouse, modem, microphone

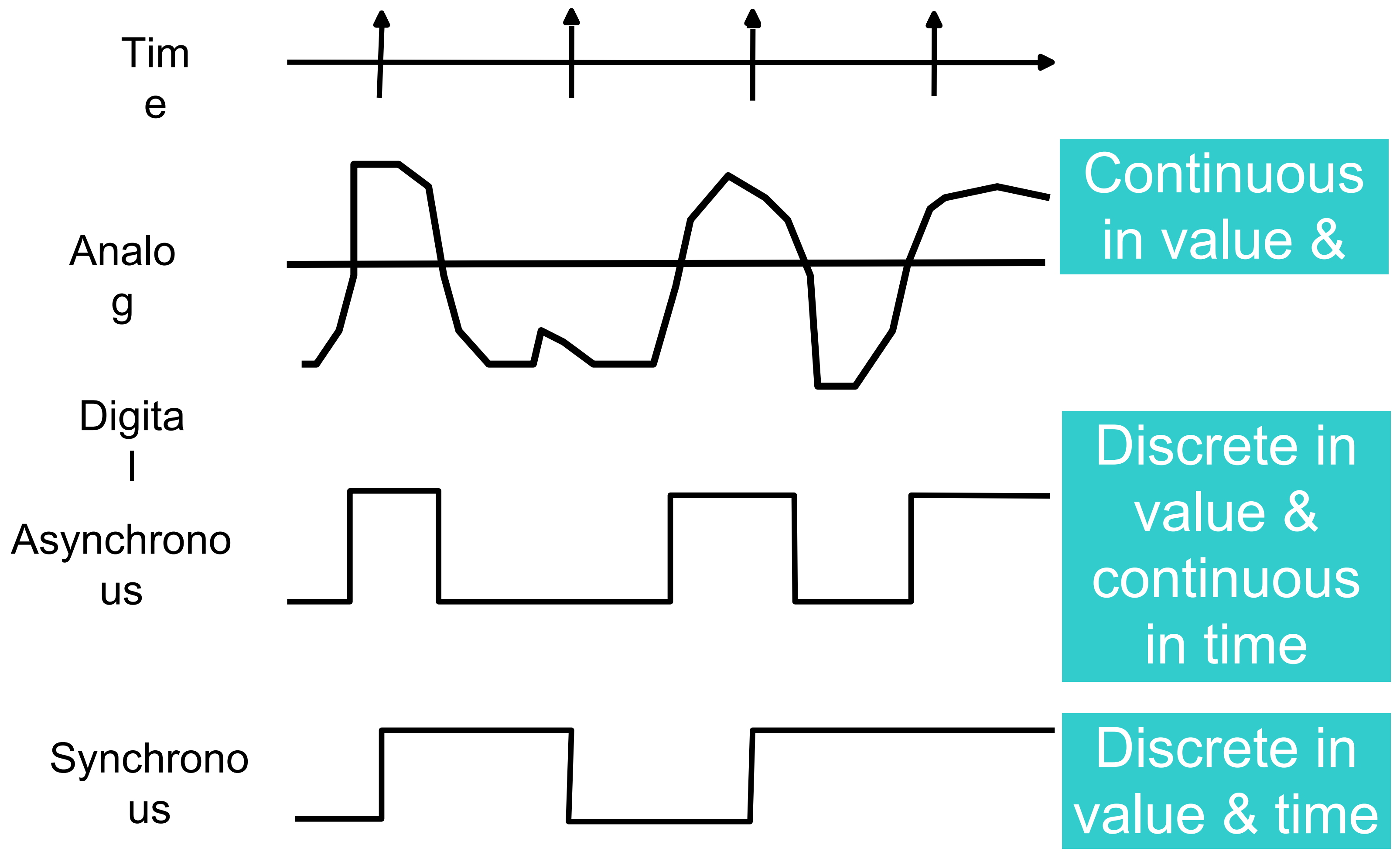
Outputs: CRT, LCD, modem, speakers

Synchronous or Asynchronous?

Signal

- An information variable represented by physical quantity.
- For digital systems, the variable takes on discrete values.
- Two level, or binary values are the most prevalent values in digital systems.
- Binary values are represented abstractly by:
  - digits 0 and 1
  - words (symbols) False (F) and True (T)
  - words (symbols) Low (L) and High (H)
  - and words On and Off.
- Binary values are represented by values or ranges of values of physical quantities

Signal Examples Over Time



Binary Values: Other Physical Quantities

- What are other physical quantities represent 0 and 1?
  - CPU Voltage
  - Disk Magnetic Field Direction
  - CD Surface Pits/Light
  - Dynamic RAM Electrical Charge

Number Systems Representation

- Positive radix, positional number systems
- A number with *radix* ***r*** is represented by a string of digits:

  $$\boldsymbol{A_{n-1}A_{n-2} \ldots A_1A_0 \,.\, A_{-1}A_{-2} \ldots A_{-m+1}A_{-m}}$$

  in which $\mathbf{0} \leq \boldsymbol{A_i} < \boldsymbol{r}$ and **.** is the *radix point*.
- The string of digits represents the power series:

$$(\text{Number})_r = \left(\sum_{i=0}^{i=n-1} A_i \cdot r^i\right) + \left(\sum_{j=-m}^{j=-1} A_j \cdot r^j\right)$$

(Integer Portion) + (Fraction Portion)

Number Systems – Examples

		General	Decimal	Binary
Radix (Base)		r	10	2
Digits		0 => r - 1	0 => 9	0 => 1
	0	r^0	1	1
	1	r^1	10	2
	2	r^2	100	4
	3	r^3	1000	8
Powers of	4	r^4	10,000	16
Radix	5	r^5	100,000	32
	-1	r^{-1}	0.1	0.5
	-2	r^{-2}	0.01	0.25
	-3	r^{-3}	0.001	0.125
	-4	r^{-4}	0.0001	0.0625
	-5	r^{-5}	0.00001	0.03125

Special Powers of 2

- 2^{10} (1024) is Kilo, denoted "K"
- 2^{20} (1,048,576) is Mega, denoted "M"
- 2^{30} (1,073, 741,824)is Giga, denoted "G"

Positive Powers of 2

- Useful for Base Conversion

Exponent	Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Exponent	Value
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536
17	131,072
18	262,144
19	524,288
20	1,048,576
21	2,097,152

Converting Binary to Decimal

- **To convert to decimal, use decimal arithmetic to form Σ (digit × respective power of 2).**
- **Example:Convert 11010_2 to N_{10}:**

Converting Decimal to Binary

- **Method 1**
  - **Subtract the largest power of 2 (see slide 14) that gives a positive remainder and record the power.**
  - **Repeat, subtracting from the prior remainder and recording the power, until the remainder is zero.**
  - **Place 1's in the positions in the binary result corresponding to the powers recorded; in all other positions place 0's.**
- **Example: Convert 625_{10} to N_2**

Commonly Occurring Bases

Name	Radix	Digits
Binary	2	0,1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

- The six letters (in addition to the 10 integers) in hexadecimal represent:

Numbers in Different Bases

- **Good idea to memorize!**

Decimal (Base 10)	Binary (Base 2)	Octal (Base 8)	Hex decimal (Base 16)
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10

Conversion Between Bases

- Method 2
  - To convert from one base to another:

1) Convert the Integer Part

2) Convert the Fraction Part

3) Join the two results with a radix point

Conversion Details

- **To Convert the Integral Part:**

  Repeatedly divide the number by the new radix and save the remainders. The digits for the new radix are the remainders in *reverse order* of their computation. If the new radix is > 10, then convert all remainders > 10 to digits A, B, …

- **To Convert the Fractional Part:**

  Repeatedly multiply the fraction by the new radix and save the integer digits that result. The digits for the new radix are the integer digits in *order* of their computation. If the new radix is > 10, then convert all integers > 10 to digits A, B, …

Example: Convert 46.6875_{10} To Base 2

- **Convert 46 to Base 2**

- **Convert 0.6875 to Base 2:**

- **Join the results together with the radix point:**

Additional Issue - Fractional Part

- **Note that in this conversion, the fractional part became 0 as a result of the repeated multiplications.**
- **In general, it may take many bits to get this to happen or it may never happen.**
- **Example: Convert 0.65_{10} to N_2**
  - **0.65 = 0.1010011001001 …**
  - **The fractional part begins repeating every 4 steps yielding repeating 1001 forever!**
- **Solution: Specify number of bits to right of radix point and round or truncate to this number.**

Checking the Conversion

- To convert back, sum the digits times their respective powers of r.
- From the prior conversion of 46.6875_{10}

$$101110_2 = 1\cdot32 + 0\cdot16 + 1\cdot8 + 1\cdot4 + 1\cdot2 + 0\cdot1$$

$$= 32 + 8 + 4 + 2$$

$$= 46$$

$$0.1011_2 = 1/2 + 1/8 + 1/16$$

$$= 0.5000 + 0.1250 + 0.0625$$

$$= 0.6875$$

A Final Conversion Note

- **You can use arithmetic in other bases if you are careful:**
- **Example: Convert 101110_2 to Base 10 using binary arithmetic:**

  Step 1 101110 / 1010 = 100 r 0110

  Step 2 100 / 1010 = 0 r 0100

  Converted Digits are 0100_2 | 0110_2

  or 4 6 $_{10}$

Binary Numbers and Binary Coding

- **Flexibility of representation**
  - **Within constraints below, can assign any binary combination (called a code word) to any data as long as data is uniquely encoded.**
- **Information Types**
  - **Numeric**
    - **Must represent range of data needed**
    - **Very desirable to represent data such that simple, straightforward computation for common arithmetic operations permitted**
    - **Tight relation to binary numbers**
  - **Non-numeric**
    - **Greater flexibility since arithmetic operations not applied.**
    - **Not tied to binary numbers**

Non-numeric Binary Codes

- **Given *n* binary digits (called bits), a binary code is a mapping from a set of represented elements to a subset of the 2^n binary numbers.**
- **Example: A binary code for the seven colors of the rainbow**
- **Code 100 is not used**

Color	Binary Number
Red	000
Orange	001
Yello	010
Green	011
Blue	101
Indigo	110
Violet	111

Number of Bits Required

- **Given M elements to be represented by a binary code, the minimum number of bits, *n*, needed, satisfies the following relationships:**

  $2^n > M > 2^{(n-1)}$

  $n = \lceil \log_2 M \rceil$ where $\lceil x \rceil$, called the *ceiling function,* is the integer greater than or equal to *x*.

- **Example: How many bits are required to represent <u>decimal digits</u> with a binary code?**

Number of Elements Represented

- **Given n digits in radix r, there are r^n distinct elements that can be represented.**
- **But, you can represent m elements, $m < r^n$**
- **Examples:**
  - **You can represent 4 elements in radix $r = 2$ with $n = 2$ digits: (00, 01, 10, 11).**
  - **You can represent 4 elements in radix $r = 2$ with $n = 4$ digits: (0001, 0010, 0100, 1000).**
  - **This second code is called a "<u>one hot</u>" code.**

Binary Codes for Decimal Digits

- There are over 8,000 ways that you can chose 10 elements from the 16 binary numbers of 4 bits.   A few are useful:

Decimal	8,4,2,1	Excess3	8,4,-2,-1	Gray
0	0000	0011	0000	0000
1	0001	0100	0111	0100
2	0010	0101	0110	0101
3	0011	0110	0101	0111
4	0100	0111	0100	0110
5	0101	1000	1011	0010
6	0110	1001	1010	0011
7	0111	1010	1001	0001
8	1000	1011	1000	1001
9	1001	1100	1111	1000

Binary Coded Decimal (BCD)

- The BCD code is the 8,4,2,1 code.
- This code is the simplest, most intuitive binary code for decimal digits and uses the same powers of 2 as a binary number, but only encodes the first ten values from 0 to 9.
- Example: 1001 (9) = 1000 (8) + 0001 (1)
- How many "invalid" code words are there?
- What are the "invalid" code words?

Excess 3 Code and 8, 4, –2, –1 Code

Decimal	Excess 3	8, 4, –2, –1
0	0011	0000
1	0100	0111
2	0101	0110
3	0110	0101
4	0111	0100
5	1000	1011
6	1001	1010
7	1010	1001
8	1011	1000
9	1100	1111

- **What interesting property is common to these two codes?**

GRAY CODE

- Gray code is the arrangement of binary number system such that each incremental value can only differ by one bit.
- This code is also known as **Reflected Binary Code** (RBC), **Cyclic Code** and **Reflected Binary** (RB). The reason for calling this code as reflected binary code is the first N/2 values compared with those of the last N/2 values in reverse order.
- In gray code when transverse from one step to another step the only one bit will be change of the group. This means that the two adjacent code numbers differ from each other by only one bit.
- It is popular for unit distance code but it is not use from arithmetic operations. This code has some application like convert analog to digital, error correction in digital communication.

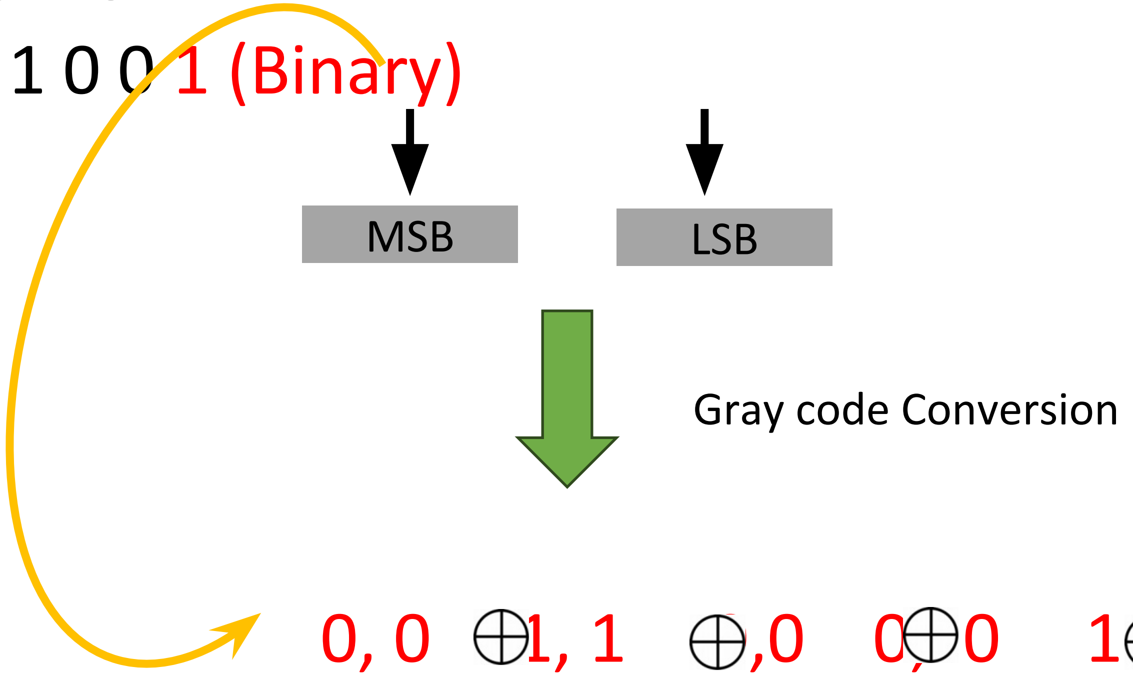
Binary- Gray code conversion

- **STEPS**
  - The most significant bit of gray code is equal to the first bit of the given binary bit.
  - The second bit of gray code will be exclusive-or (XOR) of the first and second bit of the given binary bit.
  - The third bit of gray code is equal to the exclusive-or (XOR) of the second and third binary bits. For father gray code result this process will be continuing.

Explanation

- The given binary digit is 01001

0 1 0 0 1 (Binary)



Truth Table of XOR

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

0, 0 $\oplus$ 1, 1 $\oplus$, 0 0, $\oplus$ 0 1 $\oplus$

0 , 1 , 1 , 0 , 1 (On concatening)

The gray code of the given binary code (01001)

=0 1 1 0 1 (One bit changed)

Example

- The gray code of the given binary code is $(010.01)_2$??
  - The first MSB bit of binary is same in the first bit of gray code. In this example the binary bit is “0”. So, gray bit also “0”.
  - Next gray bit is equal to the XOR of the first and the second binary bit. The first bit is 0, and the second bit is 1. The bits are different so resultant gray bit will be “1” (second gray codes bit)
  - The XOR of the second and third binary bit. The second bit is 1 and third is 0. These bits are again different so the resultant gray bit will be 1 (third gray codes bit)
  - Next we perform the XOR operation on third and fourth binary bit. The third bit is 0, and the fourth bit is 0. The both bits are same than resultant gray codes will be 0 (fourth gray codes bit).
  - Take the XOR of the fourth and fifth binary bit. The fourth bit is 0 and fifth bit is 1. These bits are different than resultant gray codes will be 1 (fifth gray code bit)
  - The result of binary to gray codes conversion is 01101.

:

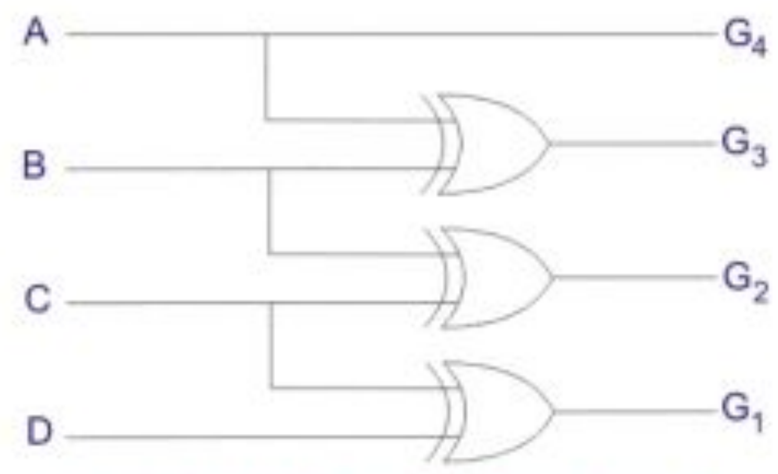
GRAY CODE TABLE

The conversion in between decimal to gray and binary to gray code is given below

Decimal Number	Binary Number	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

The gray code of the given binary code (01001) =0 1 1 0 1. We can see one bit change in the next incremental value.

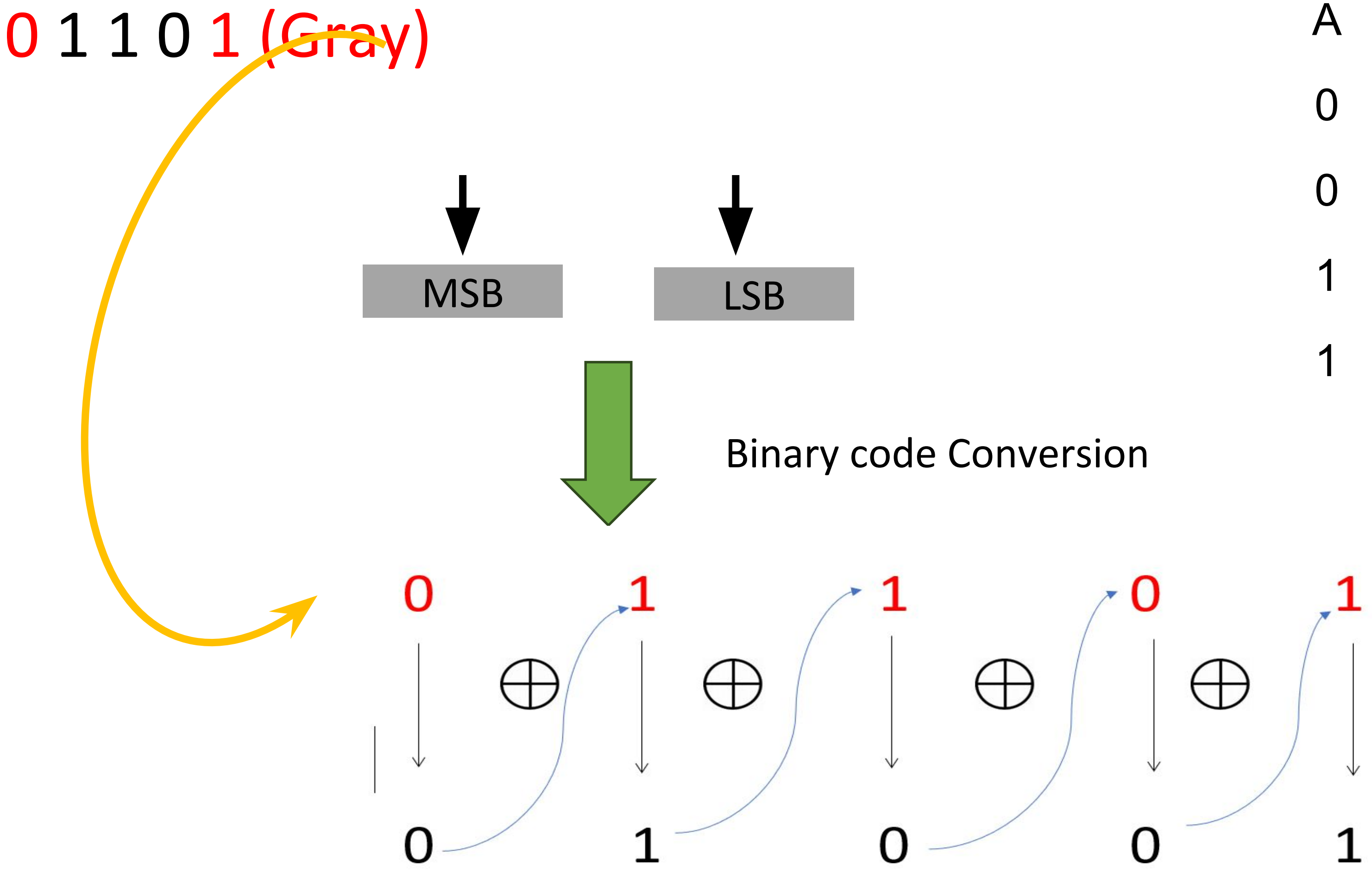
- You can convert n bit ($b_n b_{(n-1)} \ldots b_2 b_1 b_0$) binary number to gray code ($g_n g_{(n-1)} \ldots g_2 g_1 g_0$). For most significant bit $b_n = g_n$, and rest of the bit by XORing $b_{(n-1)} = g_{(n-1)} \oplus g_n$, ….



Logic Circuit for Binary to Gray Code Converter

Explanation

- The given gray code is 01101



Truth Table of XOR

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Gray-Binary code conversion

- **STEPS**
  1. The most significant bit of gray codes is equal in binary number.
  2. Now move to the next gray bit, as it is 1 the previous bit will be alter i.e it will be 1, thus the second binary bit will be 1.
  3. Next see the third bit, in this example the third bit is 1 again, the third binary bit will be alter of second binary bit and the third binary bit will be 0.
  4. Now fourth bit of the, here the fourth bit of gray code is 0. So the fourth bit will be same as a previous binary bit, i.e 4th binary bit will be 0.
  5. The last fifth bit of gray codes is 1; the fifth binary number is altering of fourth binary number.
  6. Therefore the gray code (01101) equivalent in binary number is (01001)

Merits & Demerits of Gray Code

Advantages of gray code

- It is best for error minimization in conversion of analog to digital signals.
- It is best for minimize a logic circuit
- Decreases the “Hamming Walls” which is undesirable state, when used in genetic algorithms
- It is useful in clock domain crossing

Disadvantages of gray code

- Not suitable for arithmetic operations
- It has limited use.

Decimal to BCD

- *Convert* $(324)_{10}$ *in BCD*
- *From truth table the BCD value of*
  - $(3)_{10} = (0011)_{BCD}$
  - $(2)_{10} = (0010)_{BCD}$
  - $(4)_{10} = (0100)_{BCD}$

So, $(324)_{10} = (0011\ 0010\ 0100)_{BCD}$ → (A)

$(324)_{10} = (101000100)_2 \rightarrow (B)$

- On comparing (A) and (B) we understand that binary and BCD value of the given decimal is not same.

Binary to BCD conversion

- Step 1: Convert binary to decimal number
- Step 2 : Convert decimal number to BCD
  - Consider a binary number $(11101)_2$ and convert it to BCD.

Step 1: Convert binary to decimal number

$$(11101)_2 = ((1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))$$
$$= 16 + 8 + 4 + 0 + 1 = (29)_{10}\ (Decimal)$$

Step 2 : Convert decimal number to BCD (in 4 digit) (Write the 4 digit BCD value for individual digit in decimal)

$$(11101)_2 = (29)_{10} = (0010\ 1001)_{BCD}$$

Advantages of BCD

- Easy to encode and Decode decimals into BCD and Vice-versa
- Easy to implement a hardware algorithm for BCD converter
- Very useful in digital systems whenever decimal information reqd.
  - Digital voltmeters, frequency converters and digital clocks all use BCD as they display output information in decimal.

Disadvantages of BCD Code

- Require more bits than straight binary code
- Difficult to be used in high speed digital computer when the size and capacity of their internal registers are restricted or limited.
- The arithmetic operations using BCD code require a complex design of Arithmetic and Logic Unit (ALU) than the straight binary number system.
- The speed of the arithmetic operations that can be realized using BCD code is naturally slow due to the complex hardware circuitry involved.
-

Excess-3 code

- The excess-3 code is also treated as **XS-3 code**. The excess-3 code is a non-weighted and self-complementary BCD code used to represent the decimal numbers.
- This code has a biased representation. This code plays an important role in arithmetic operations because it resolves deficiencies encountered when we use the 8421 BCD code for adding two decimal digits whose sum is greater than 9.
- The Excess-3 code uses a special type of algorithm, which differs from the binary positional number system or normal non-biased BCD.

Decimal to Excess-3 code conversion

Step-1: We find the decimal number of the given binary number.
Step-2: Then we add 3 in each digit of the decimal number.

Step-3: Now, we find the binary code of each digit of the newly generated decimal number.

Alternatively,

- We can also add 0011 in each 4-bit BCD code of the decimal number for getting excess-3 code.

Ex-1

- Convert decimal $(5)_{10}$ *to* Excess-3.
- Step 1: Add '3' to the given decimal → 5+3=8
- Step 2: Find binary digit of a new number '8' which is $(8)_2 = (1000)_2$.
- The Excess-3 code of given decimal $(5)_{10}$ is $(1000)_{Excess-3}$.

Ex-2

- Convert BCD $(0101)_{BCD}$ *to* Excess-3.

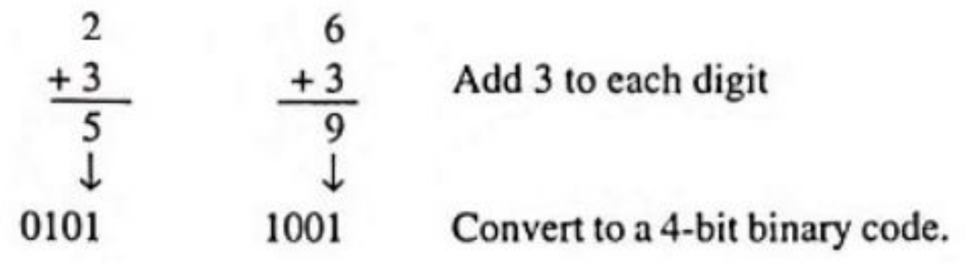
Step 1: Add BCD value of 3 (0011) to the given number.

$$\begin{array}{l} 0\,1\,0\,1 \\ \underline{0\,0\,1\,1}(+) \\ \underline{1\,0\,0\,0} \end{array}$$

$(0101)_{BCD} = (1000)_{Excess-3}$

Ex-3

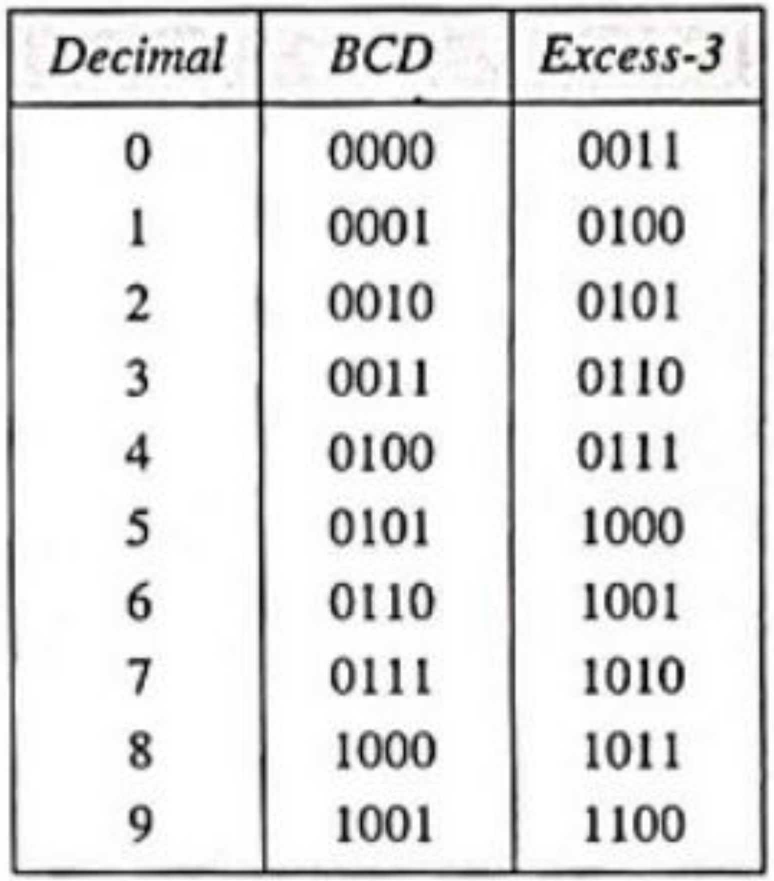
- Convert decimal $(26)_{10}$ to Excess-3.
- Step 1: Add ‘3’ to the individual given decimal



$$\begin{array}{ccl} 2 & 6 & \\ +3 & +3 & \text{Add 3 to each digit} \\ \hline 5 & 9 & \\ \downarrow & \downarrow & \\ 0101 & 1001 & \text{Convert to a 4-bit binary code.} \end{array}$$

- The Excess-3 code of given decimal $(26)_{10}$ is $(0101\ 1001)_{Excess-3}$.

Excess-3 code



Decimal	*BCD*	*Excess-3*
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

Ex-4

- Convert decimal $(81.61)_{10}$ *to* Excess-3.
- Step 1:

Decimal	BCD
8	1000
1	0001
6	0110
1	0001

Step 2:

BCD+ 0011	Excess-3
1000+0011	1011
0001+0011	0100
0110+0011	1001
0001+0011	0100

$$(81.61)_{10} = (1011\ 0100.1001\ 0100)_{Excess-3}$$

Ex 5

- **Convert $(11110)_2$ to Excess-3 using binary**
- **Step 1: Convert binary to Decimal. $(11110)_2 = (30)_{10}$**
- **Step 2: Add '3' to individual digits to decimal number**

$$\begin{array}{cc} 3 & 0 \\ \underline{3} & \underline{3} \\ 6 & 3 \end{array}$$

Step 3: Find binary values of $(63)_{10} = (01100011)_{\text{Excess-3}}$

Ex 6

- Convert $(01100011)_{\text{Excess-3}}$ to binary.
- **Step- 1 : Find the decimal by dividing it four digits**
  - $(01100011)_{\text{Excess-3}} = (0110\ 0011)_{\text{Excess-3}} = (30)_{10}$
- **Step-2 : Find the binary value of the decimal thorugh division method.**

$$(01100011)_{\text{Excess-3}} = (11110)_2$$

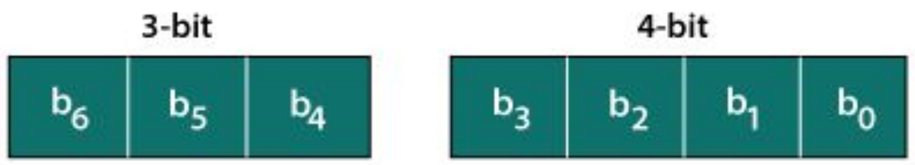
Advantages

1. These codes are self-complementary.
2. These codes use biased representation.
3. The excess-3 code has no limitation, so that it considerably simplifies arithmetic operations.
4. The codes 0000 and 1111 can cause a fault in the transmission line. The excess-3 code doesn't use these codes and gives an advantage for memory organization.
5. These codes are usually unweighted binary decimal codes.
6. This code has a vital role in arithmetic operations. It is because it resolves deficiencies which are encountered when we use the 8421 BCD code for adding two decimal digits whose sum is greater than 9.

ASCII

- The ASCII stands for American Standard Code for Information Interchange. The ASCII code is an alphanumeric code used for data communication in digital computers.
- The ASCII is a 7-bit code capable of representing 2^7 or 128 number of different characters. The ASCII code is made up of a three-bit group, which is followed by a four-bit code.

Representation of ASCII Code



ASCII Characters

- Control Characters-0 to 31 and 127
- Special Characters- 32 to 47, 58 to 64, 91 to 96, and 123 to 126
- Numbers Characters- 0 to 9
- Letters Characters - 65 to 90 and 97 to 122
- ASCII Character Set

Ex 1

- Encode (10010101100001111011011000011010100111000011011111 101001 110111011101001000000011000101100100110011)$_2$ to ASCII.

- Step 1:
- The given binary data is grouped into 7-bits because the ASCII code is 7 bit.
- 1001010 1100001 1110110 1100001 1010100 1110000 1101111 1101001 1101110 1110100 1000000 0110001 0110010 0110011
- Step 2:
- Then, we find the equivalent decimal number of the binary digits either from the ASCII table or **64 32 16 8 4 2 1** scheme.

Cont’d

Binary	64	32	16	8	4	2	1	DECIMAL	ASCII
1100011	1	1	0	0	0	1	1	99	C
1101111	1	1	0	1	1	1	1	111	O
1101101	1	1	0	1	1	0	1	109	M
1110000	1	1	1	0	0	0	0	112	P
1110101	1	1	1	0	1	0	1	117	U
1110100	1	1	1	0	1	0	0	116	T
1100101	1	1	0	0	1	0	1	101	E
1110010	1	1	1	0	0	1	0	114	R

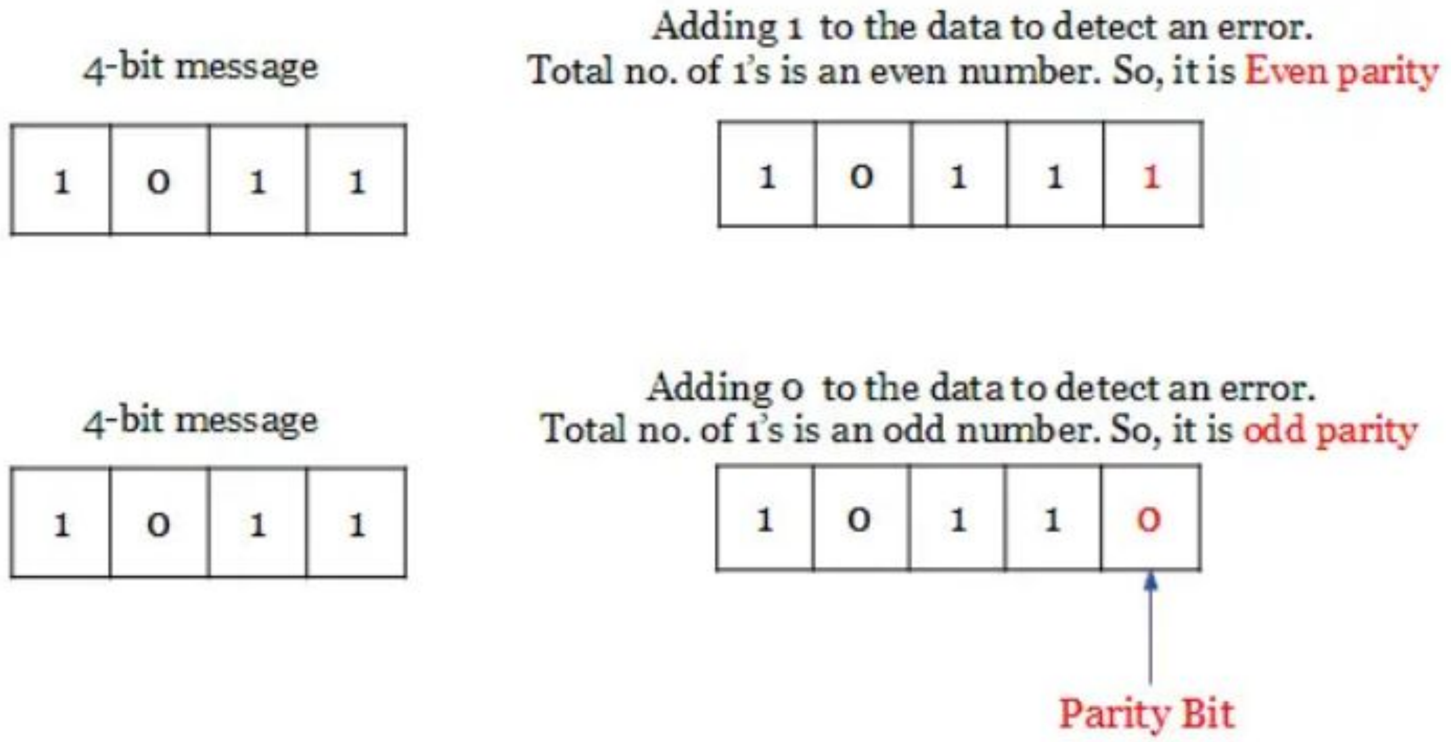
So the given binary digits results in ASCII Keyword COMPUTER

Parity Code

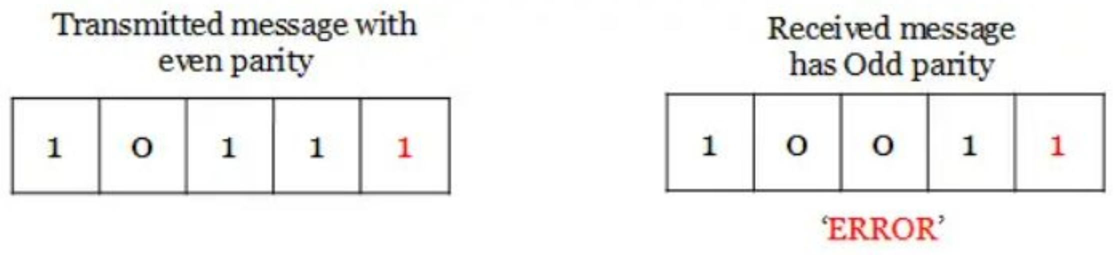
- The parity code is used for the purpose of detecting errors during the transmission of binary information. The parity code is a bit that is included with the binary data to be transmitted.
- The inclusion of a parity bit will make the number of 1’s either odd or even. Based on the number of 1’s in the transmitted data, the parity code is of two types.
    - Even parity code
    - Odd parity code

- In even parity, the added parity bit will make the total number of 1’s an even number.
- If the added parity bit make the total number of 1’s as odd number, such parity code is said to be odd parity code.

Explanation


4-bit message
1 0 1 1
Adding 1 to the data to detect an error.
Total no. of 1's is an even number. So, it is Even parity
1 0 1 1 1
4-bit message
1 0 1 1
Adding 0 to the data to detect an error.
Total no. of 1's is an odd number. So, it is odd parity
1 0 1 1 0
Parity Bit

- On the receiver side, if the received data is other than the sent data, then it is an error. If the sent date is even parity code and the received data is odd parity, then there is an error.


Transmitted message with even parity
1 0 1 1 1
Received message has Odd parity
1 0 0 1 1
'ERROR'

- So, both even and odd parity codes are used only for the detection of error and not for the correction in the transmitted data. Even parity is commonly used and it has almost become a convention.

- Advantages
  - Easy to implement
  - Simple method
  - Used error detection
- Demerits
  - No error correction

Warning: Conversion or Coding?

- **Do NOT mix up conversion of a decimal number to a binary number with coding a decimal number with a BINARY CODE.**
- **$13_{10} = 1101_2$ (This is conversion)**
- **13 $\Leftrightarrow$ 0001|0011 (This is coding)**

Binary Arithmetic

- **Single Bit Addition with Carry**
- **Multiple Bit Addition**
- **Single Bit Subtraction with Borrow**
- **Multiple Bit Subtraction**
- **Multiplication**
- **BCD Addition**

Multiple Bit Binary Addition

- **Extending this to two multiple bit examples:**

Carries	**<u>0</u>**	**<u>0</u>**
Augend	**01100**	**10110**
Addend	**<u>+10001</u>**	**<u>+10111</u>**
Sum		

- **Note: The <u>0</u> is the default Carry-In to the least significant bit.**

Single Bit Binary Subtraction with Borrow

- **Given two binary digits (X,Y), a borrow in (Z) we get the following difference (S) and borrow (B):**
- **Borrow in (Z) of 0:**

Z	0	0	0	0
X	0	0	1	1
- Y	-0	-1	-0	-1
BS	0 0	1 1	0 1	0 0

- **Borrow in (Z) of 1:**

Z	1	1	1	1
X	0	0	1	1
- Y	-0	-1	-0	-1
BS	11	1 0	0 0	1 1

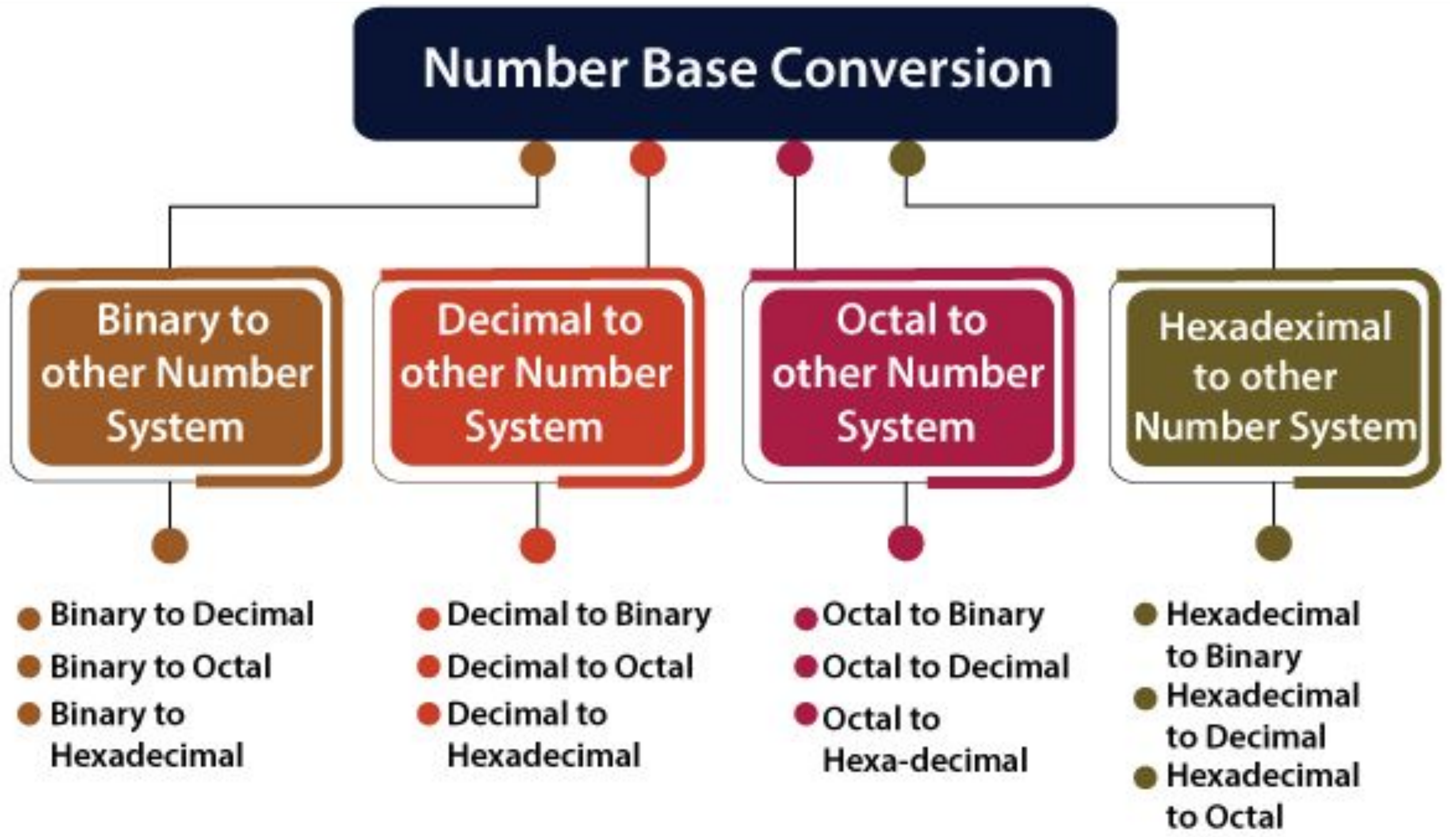
Multiple Bit Binary Subtraction

- **Extending this to two multiple bit examples:**

Borrows	$\underline{\mathbf{0}}$	$\underline{\mathbf{0}}$
Minuend	**10110**	**10110**
Subtrahend	$\underline{\mathbf{-\ 10010}}$	$\underline{\mathbf{-\ 10011}}$
Difference		

- **Notes: The $\underline{\mathbf{0}}$ is a Borrow-In to the least significant bit. If the Subtrahend > the Minuend, interchange and append a – to the result.**

Number System

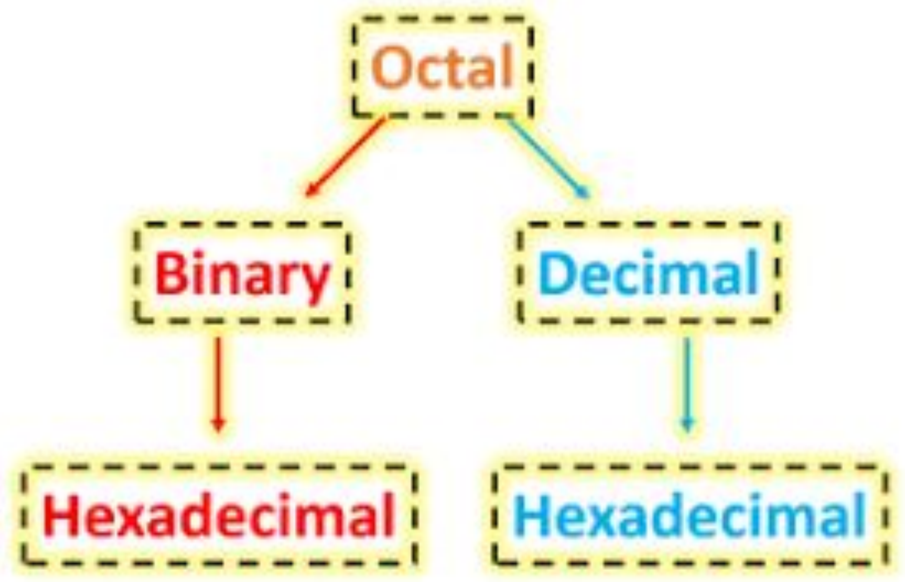


Octal Number System

Octal to Binary

Octal to Decimal

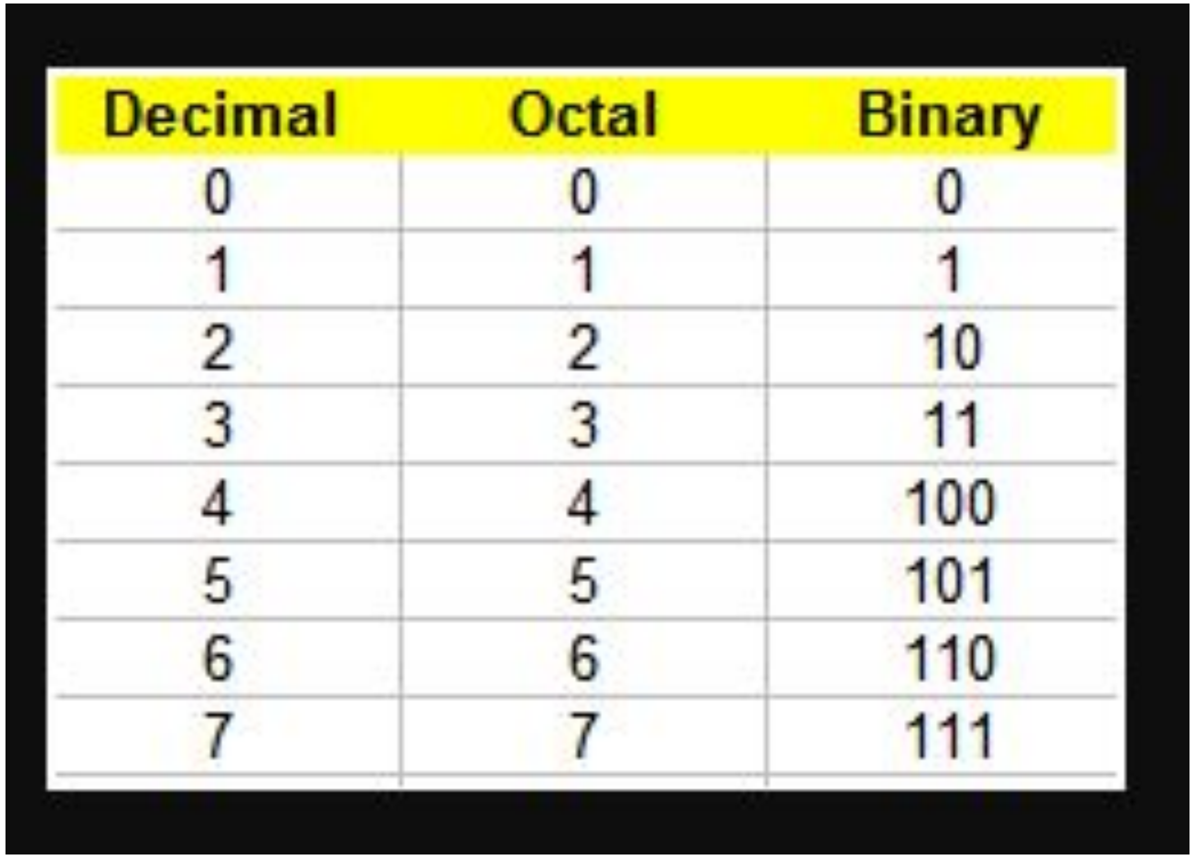
Octal to Hexadecimal



Octal to Binary

- **Octal** number is one of the number systems which has value of base is 8, that means there only 8 symbols – 0, 1, 2, 3, 4, 5, 6, and 7.
- Whereas **Binary** number is most familiar number system to the digital systems, networking, and computer professionals.

Octal to Binary Equivalent Table



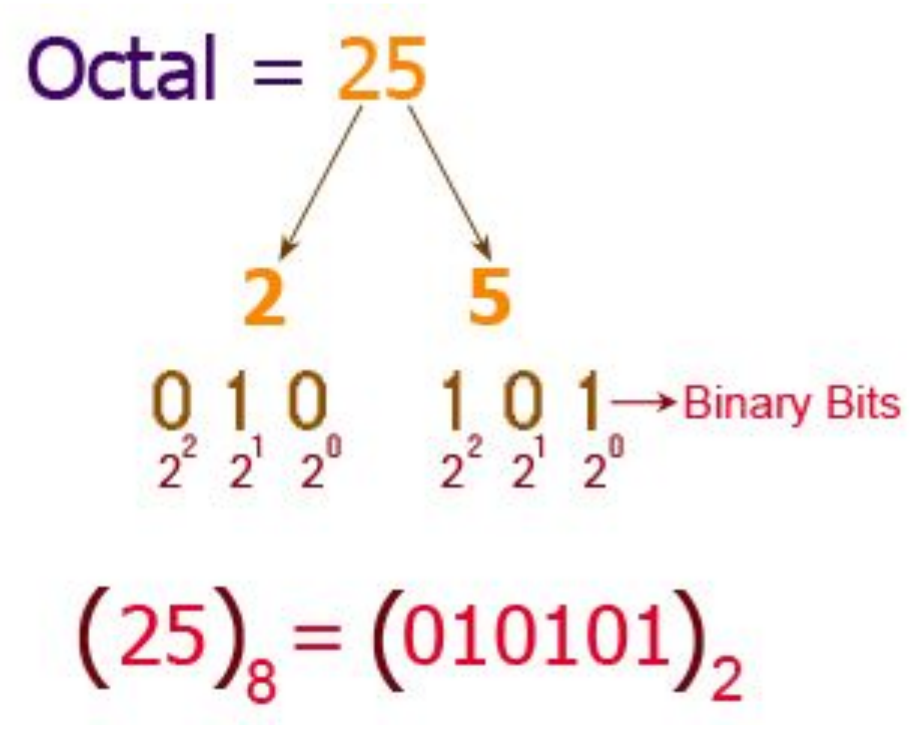
Decimal	Octal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111

Octal to Binary

- This method is simple and also works as reverse of Binary to Octal Conversion. The algorithm is explained as following below.
- Take Octal number as input
- Convert each digit of octal into binary.
- That will be output as binary number.

Octal to Binary

- **Example-1** Convert octal number 25 into binary number.

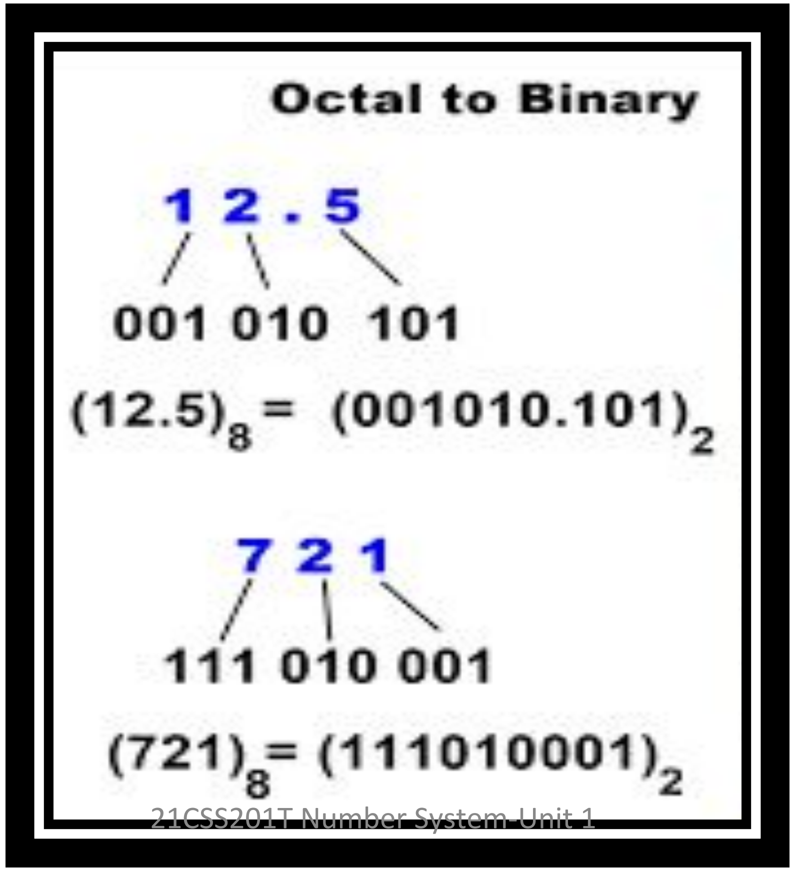


Octal to Binary

- **Example-2** Convert octal number 540 into binary number.
- According to above algorithm, equivalent binary number will be,
- $= (540)_8 = (101\ 100\ 000)_2 = (101100000)_2$

Octal to Binary

- Example 3: Convert octal number 12.5 & 721 into binary number.



$(12.5)_8 = (001010.101)_2$

$(721)_8 = (111010001)_2$

Octal to Binary

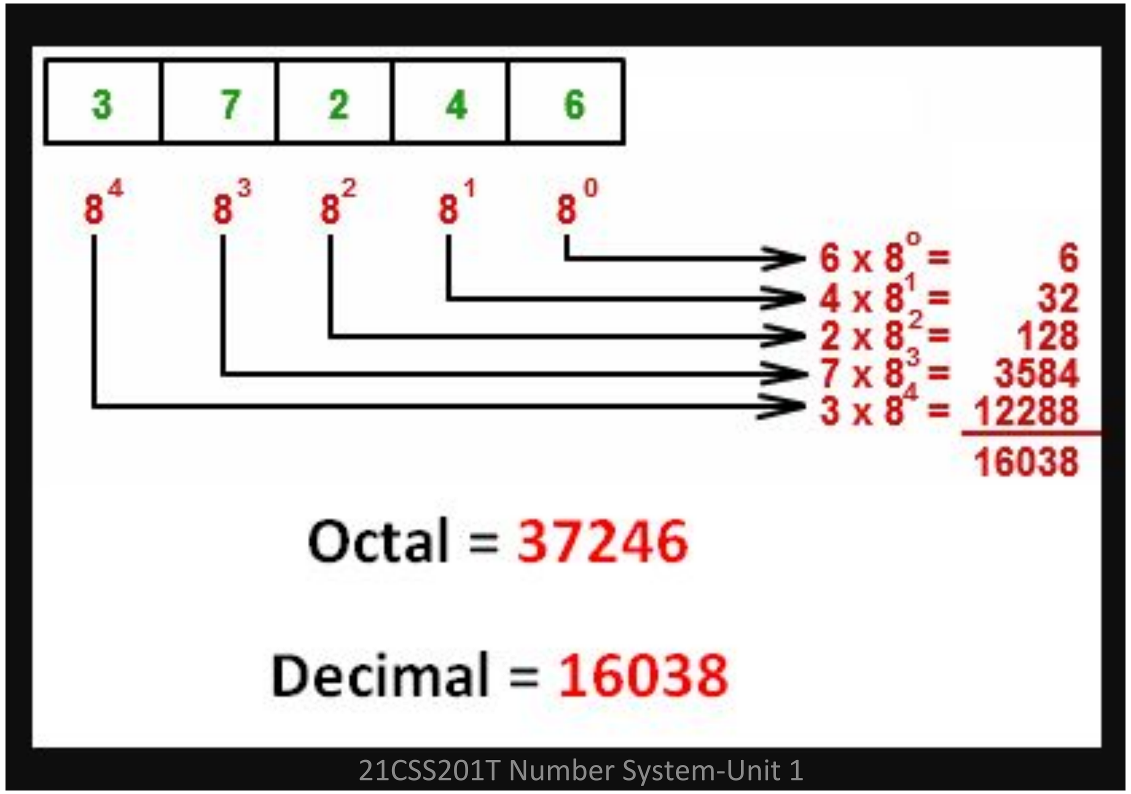
- Example 4: Convert octal number 352.563 into binary number.
- According to above algorithm, equivalent binary number will be,
- $= (352.563)_8 = (011\ 101\ 010\ .\ 101\ 110\ 011)_2 = (011101010.101110011)_2$

Octal to Decimal

- To convert an octal number to a decimal number we need to multiply each digit of the given octal with the reducing power of 8.
- Let us learn here, the conversion of Octal number to Decimal Number or base 8 to base 10.

Octal to Decimal

- **Example 1:** Convert octal number $37246_{(8)}$ into decimal form.

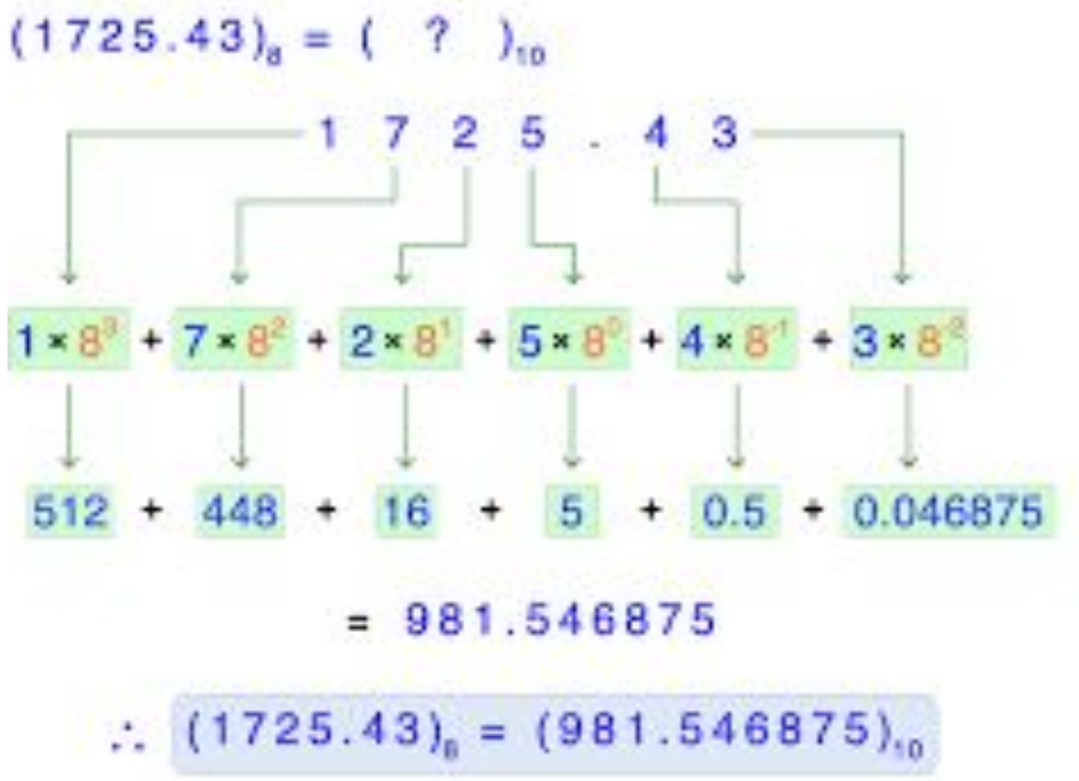


Octal = 37246

Decimal = 16038

Octal to Decimal

- **Example 2:** Convert octal number $1725.43_{(8)}$ into decimal form.



$(1725.43)_8 = (\ ?\ )_{10}$

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

$512 + 448 + 16 + 5 + 0.5 + 0.046875$

$= 981.546875$

$\therefore\ (1725.43)_8 = (981.546875)_{10}$

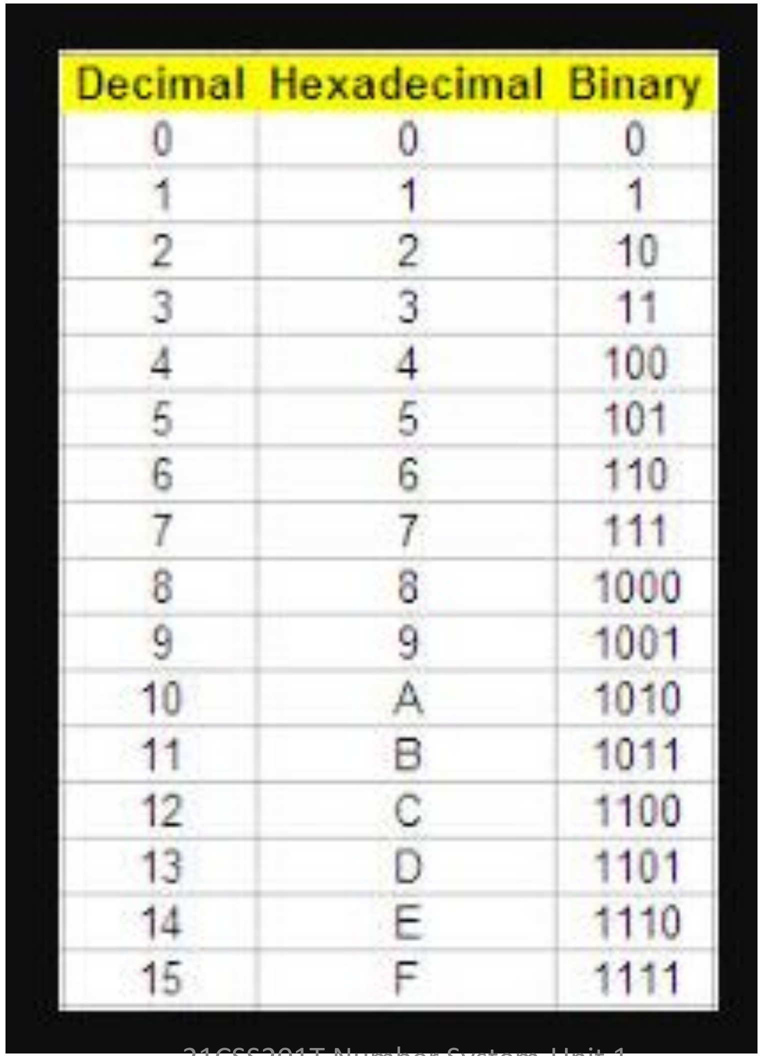
Octal to Decimal

- **Example 3:** Convert octal number $7.12172_{(8)}$ into decimal form.
- $= 7x8^{0} + 1x8^{-1} + 2x8^{-2} + 1x8^{-3} + 7x8^{-4} + 2x8^{-5}$
- = 7 + 0.125 + 0.03125 + 0.001953125 + 0.001708984375 + 0.00006103515624
- = 10.1599

Octal to Hexadecimal

- First turning the octal number into a
- 1. Binary Digit
- 2. binary to Hexadecimal

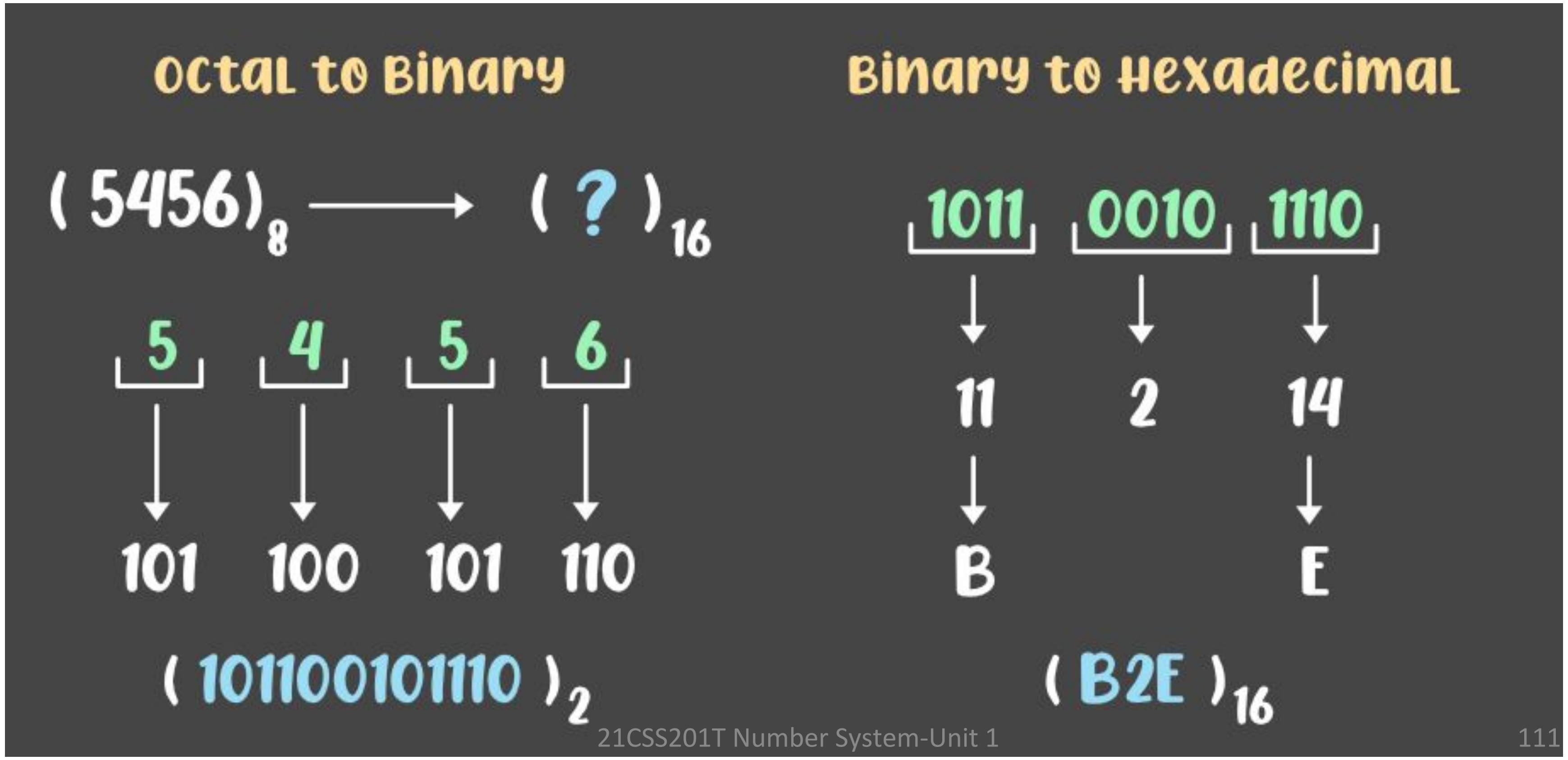
Octal to Hexadecimal



Decimal	Hexadecimal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

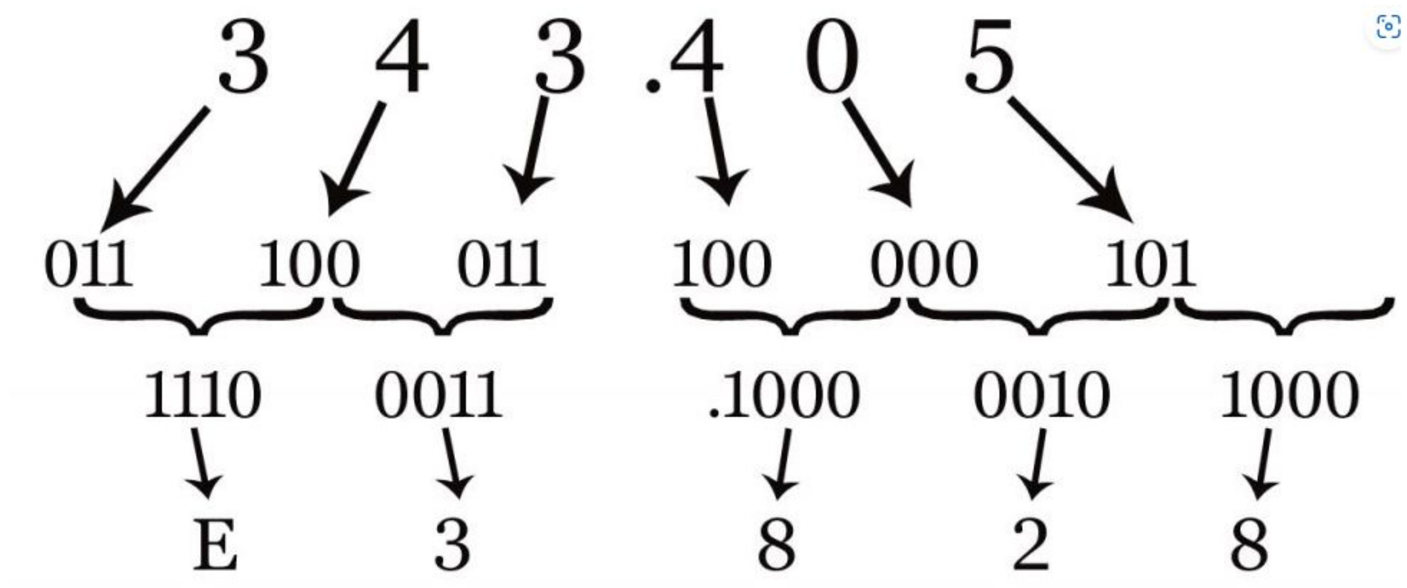
Octal to Hexadecimal

- **Example 1:** Convert octal number 5456 $_{(8)}$ into hexadecimal form.



Octal to Hexadecimal

- **Example 2:** Convert octal number 343.405$_{(8)}$ into hexadecimal form.



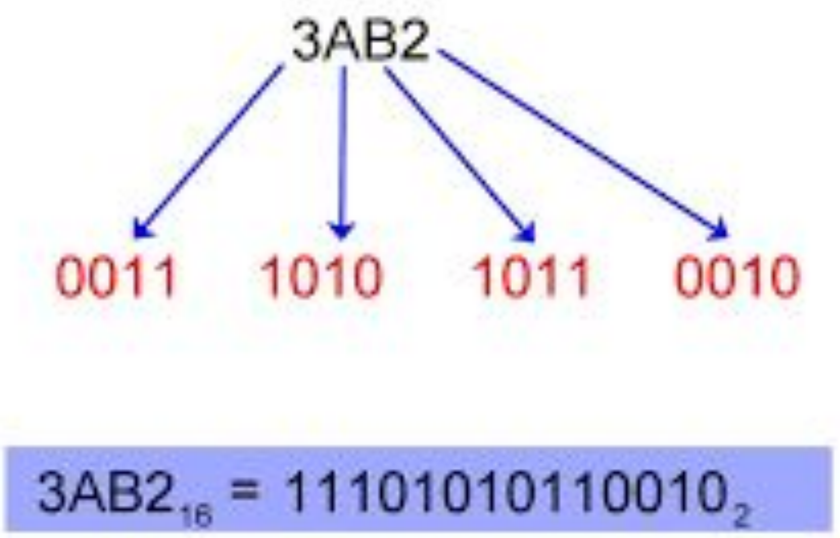
So, $(343.405)_8 = (E3.828)_{16}$

Hexadecimal Number System

- Hexadecimal to Binary
- Hexadecimal to decimal
- Hexadecimal to octal

Hexadecimal to Binary

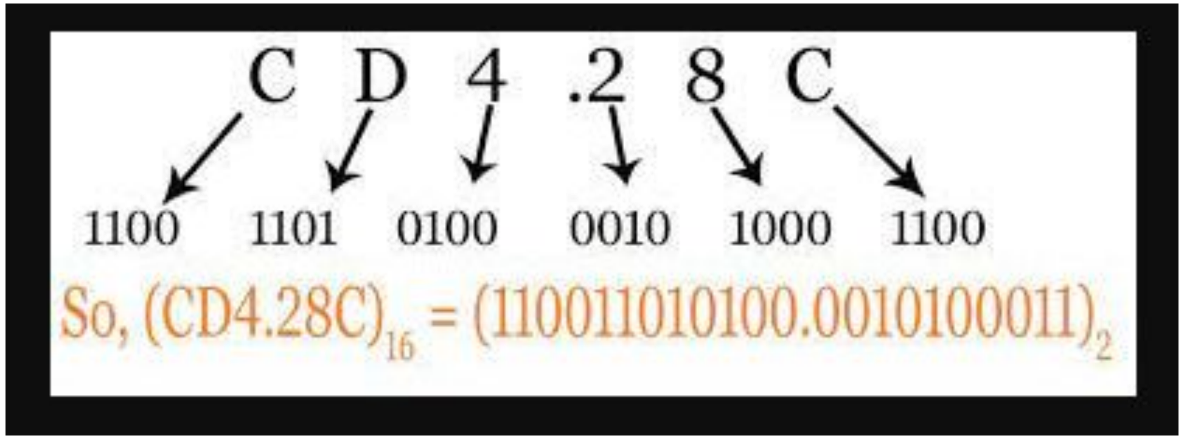
- **Example 1:** Convert hexadecimal number BAB2$_{(16)}$ into binary form.



$3AB2_{16} = 11101010110010_2$

Hexadecimal to Binary

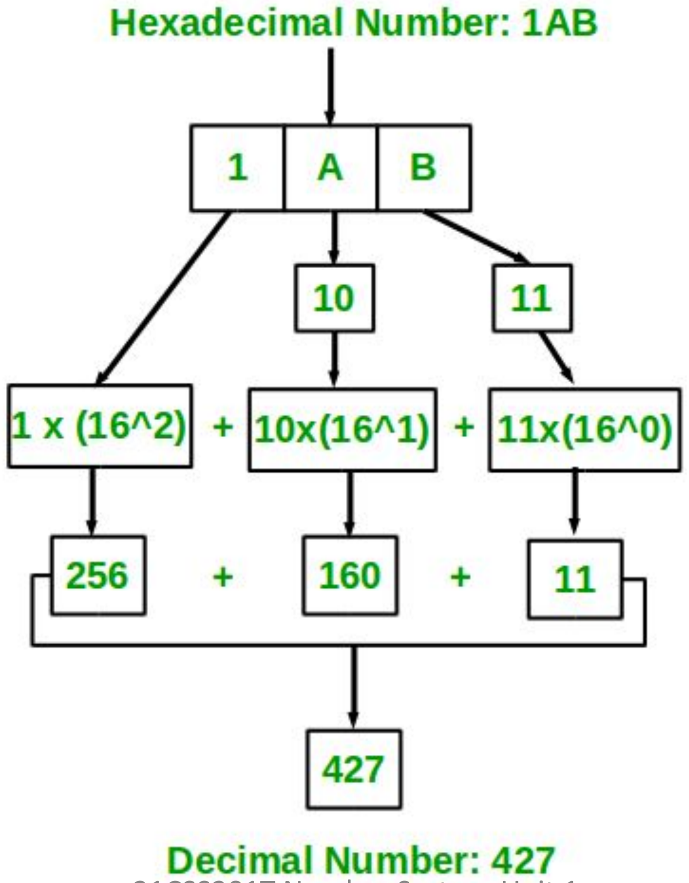
- **Example 2:** Convert hexadecimal number CD4.28C$_{(16)}$ into binary form.



So, $(CD4.28C)_{16} = (110011010100.0010100011)_2$

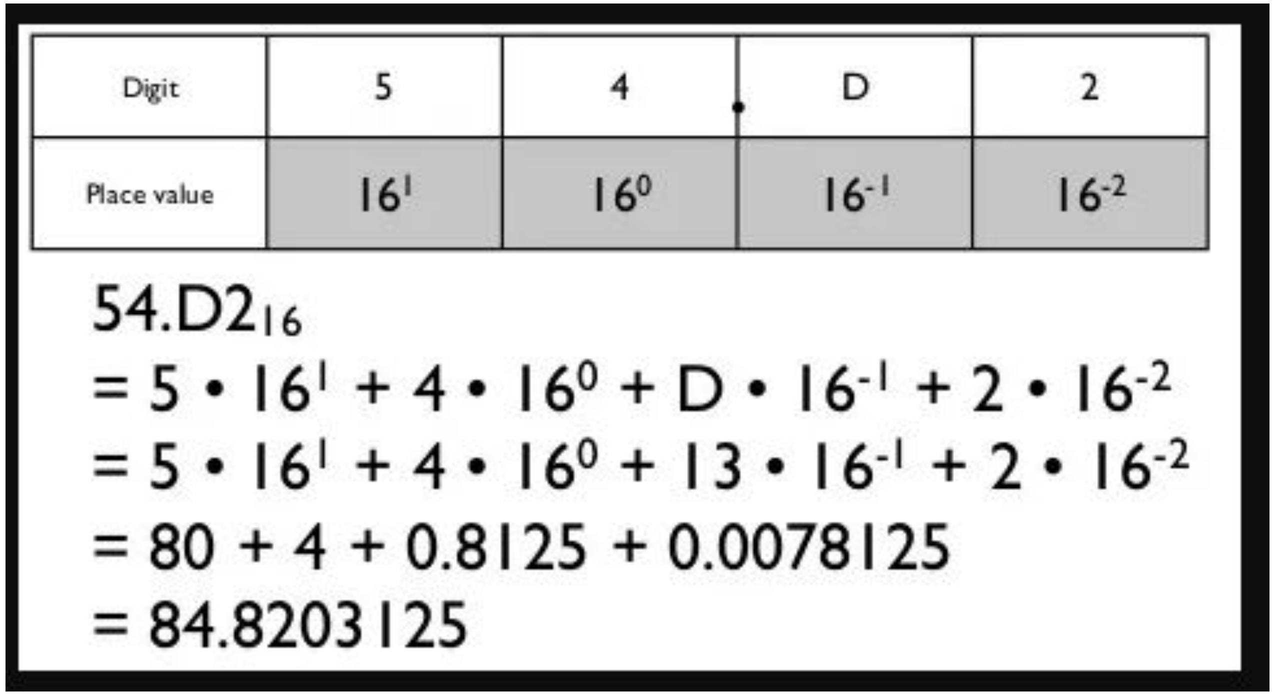
Hexadecimal to decimal

- **Example 1:** Convert hexadecimal number 1AB$_{(16)}$ into decim



Hexadecimal to decimal

- **Example 2:** Convert hexadecimal number 54.D2$_{(16)}$ into decimal form.



Digit	5	4	D	2
Place value	16^1	16^0	16^{-1}	16^{-2}

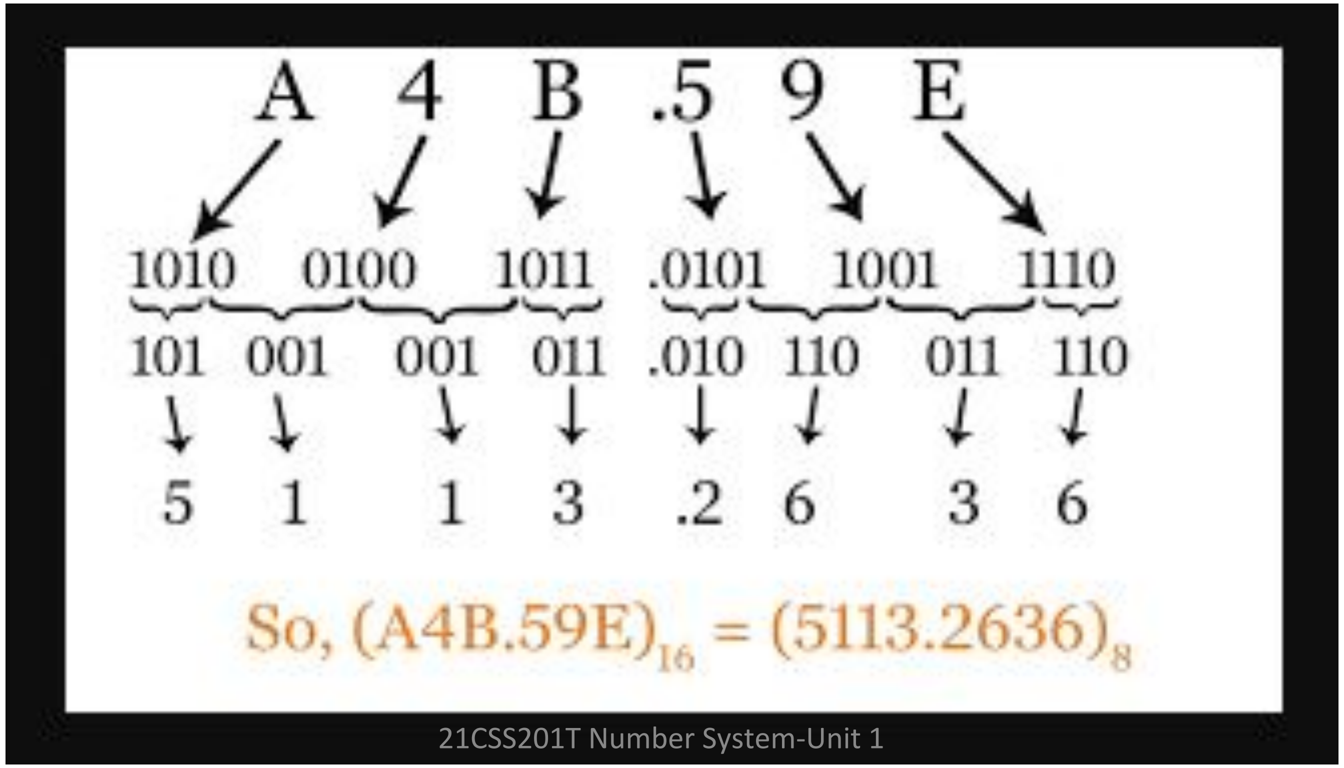
$$
\begin{aligned}
&54.D2_{16}\\
&= 5 \cdot 16^1 + 4 \cdot 16^0 + D \cdot 16^{-1} + 2 \cdot 16^{-2}\\
&= 5 \cdot 16^1 + 4 \cdot 16^0 + 13 \cdot 16^{-1} + 2 \cdot 16^{-2}\\
&= 80 + 4 + 0.8125 + 0.0078125\\
&= 84.8203125
\end{aligned}
$$

Hexadecimal to octal

- First turning the hexadecimal number into a
- 1. Binary Number
- 2. binary to octal

Hexadecimal to octal

- **Example 1:** Convert hexadecimal number A4B.59E$_{(16)}$ into decimal form.



So, $(A4B.59E)_{16} = (5113.2636)_8$

Signed Binary Numbers

- Two ways of representing signed numbers:
  - 1) Sign-magnitude form, 2) Complement form.
- Most of computers use complement form for negative number notation.
- 1’s complement and 2’s complement are two different methods in this type.

1’s Complement

- 1’s complement of a binary number is obtained by subtracting each digit of that binary number from 1.
- Example

$$\begin{array}{rcccc} & 1 & 1 & 1 & 1 \\ - & 1 & 1 & 0 & 1 \\ \hline & 0 & 0 & 1 & 0 \end{array} \qquad \begin{array}{rcccccc} & 1 & 1 & 1 & . & 1 & 1 \\ - & 1 & 0 & 1 & . & 0 & 1 \\ \hline & 0 & 1 & 0 & . & 1 & 0 \end{array}$$

(1’s complement of 1101) (1’s complement of 101.01)

Shortcut: Invert the numbers from 0 to 1 and 1 to 0

2’s Complement

- 2’s complement of a binary number is obtained by adding 1 to its 1’s complement.
- Example

```
    1 1 1 1            1 1 1 . 1 1
 -  1 1 0 0         -  1 0 1 . 0 1
 ----------         --------------
    0 0 1 1            0 1 0 . 1 0
 +        1         +            1
 ----------         --------------
    0 1 0 0            0 1 0 . 1 1
(2's complement of 1100)   (2's complement of 101.01)
```

Shortcut: Starting from right side, all bits are same till first 1 occurs and then invert rest of the bits

Subtraction using 1’s complement

- Using 1’s complement
  - Obtain 1’s complement of subtrahend
  - Add the result to minuend and call it intermediate result
  - If carry is generated then answer is positive and add the carry to Least Significant Digit (LSD)
  - If there is no carry then answer is negative and take 1’s complement of intermediate result and place negative sign to the result.

Subtraction using 2's complement

- Using 2's complement
  - Obtain 2's complement of subtrahend
  - Add the result to minuend
  - If carry is generated then answer is positive, ignore carry and result itself is answer
  - If there is no carry then answer is negative and take 2's complement of intermediate result and place negative sign to the result.

Subtraction using 1’s complement (Examples)

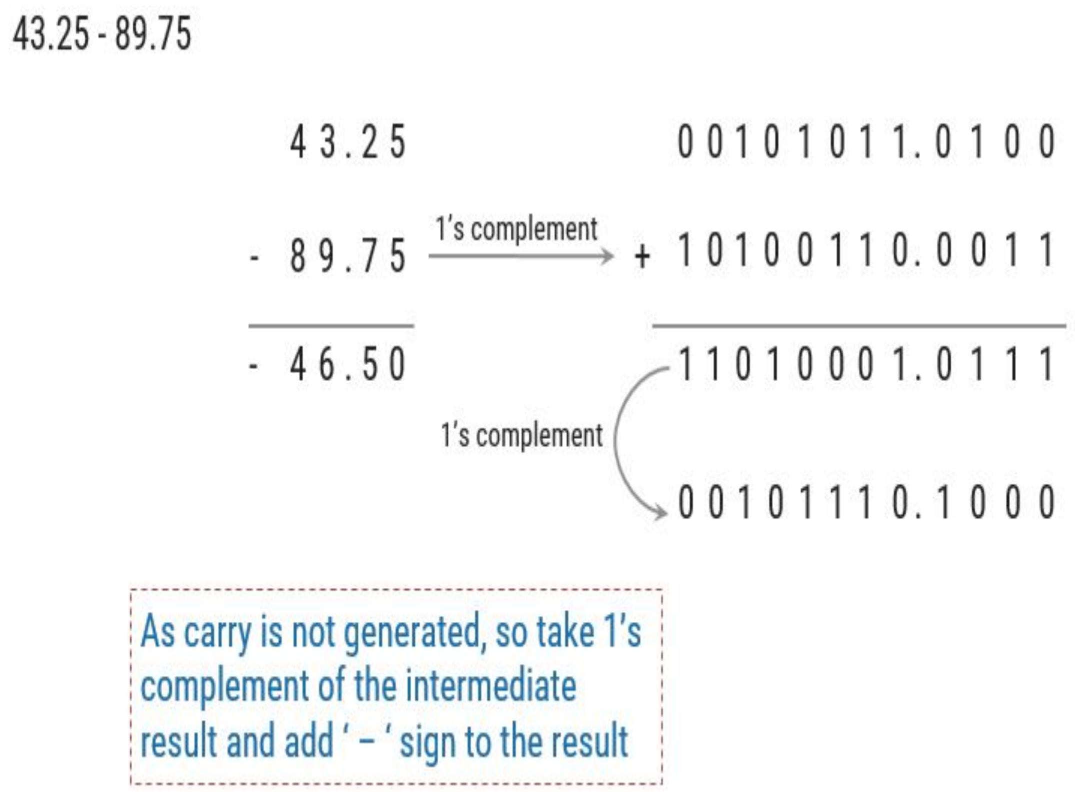
Example - 1

68.75 – 27.50

$$\begin{array}{rcr}
68.75 & & 01000100.1100 \\
-\ 27.50 & \xrightarrow{\text{1's complement}} & +\ 11100100.0111 \\
\hline
+\ 41.25 & & 100101001.0011 \\
 & & +1 \\
\hline
 & & 00101001.0100
\end{array}$$

Subtraction using 1’s complement (Examples)

Example - 2



43.25 - 89.75

$$\begin{array}{r} 43.25 \\ -\ 89.75 \\ \hline -\ 46.50 \end{array} \qquad \xrightarrow{\text{1's complement}} \qquad \begin{array}{r} 00101011.0100 \\ +\ 10100110.0011 \\ \hline 11010001.0111 \end{array}$$

$$11010001.0111 \xrightarrow{\text{1's complement}} 00101110.1000$$

As carry is not generated, so take 1’s complement of the intermediate result and add ‘ – ‘ sign to the result

Subtraction using 2’s complement (Examples)

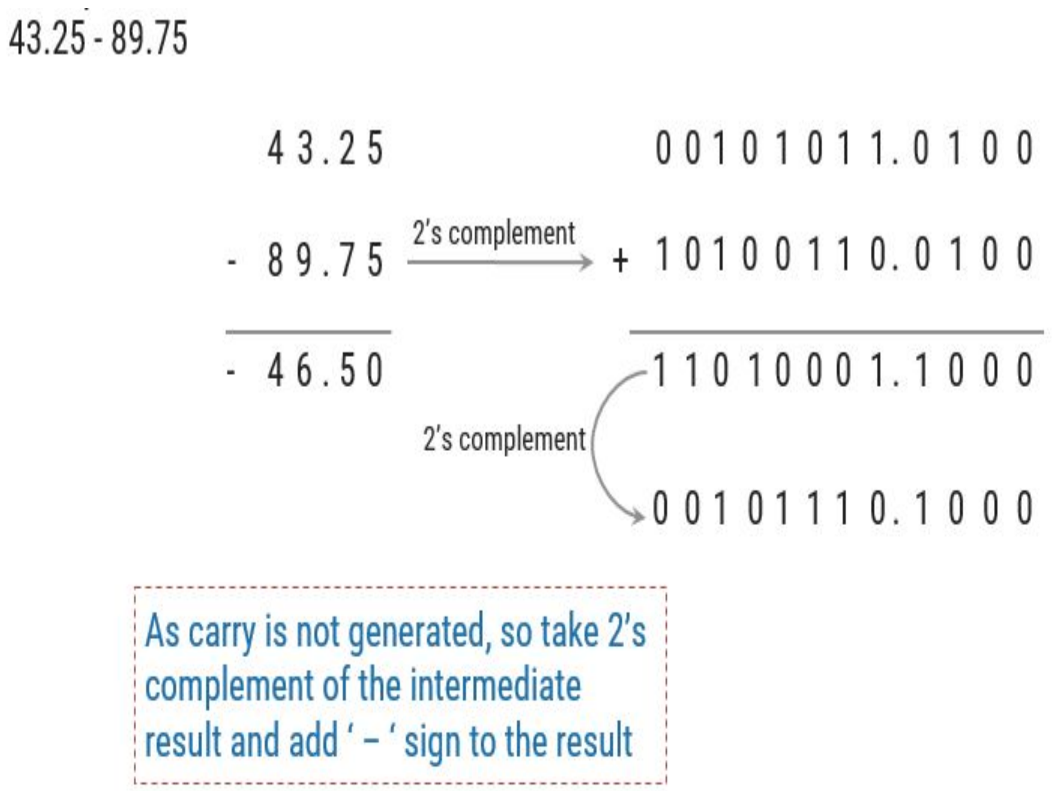
Example - 1

68.75 – 27.50

$$\begin{array}{rrcr} & 68.75 & & 0\,1\,0\,0\,0\,1\,0\,0\,.\,1\,1\,0\,0 \\ - & 27.50 & \xrightarrow{\text{2's complement}} + & 1\,1\,1\,0\,0\,1\,0\,0\,.\,1\,0\,0\,0 \\ \hline + & 41.25 & & \boxed{1}\,0\,0\,1\,0\,1\,0\,0\,1\,.\,0\,1\,0\,0 \\ & & \text{Ignore Carry bit} \leftarrow & \\ & & & 0\,0\,1\,0\,1\,0\,0\,1\,.\,0\,1\,0\,0 \end{array}$$

Subtraction using 2's complement (Examples)

Example - 2

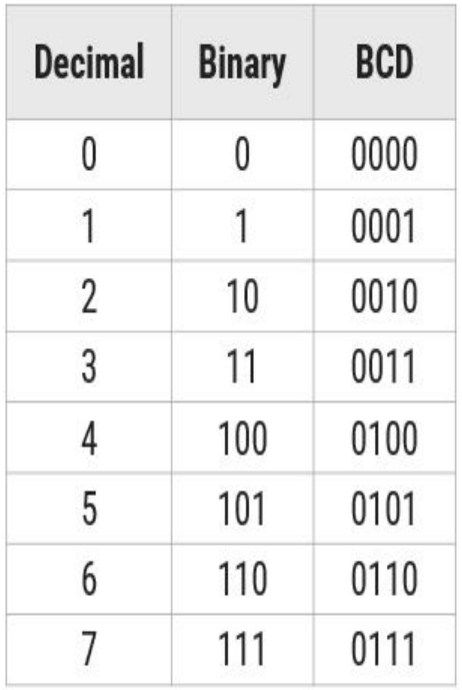


43.25 - 89.75

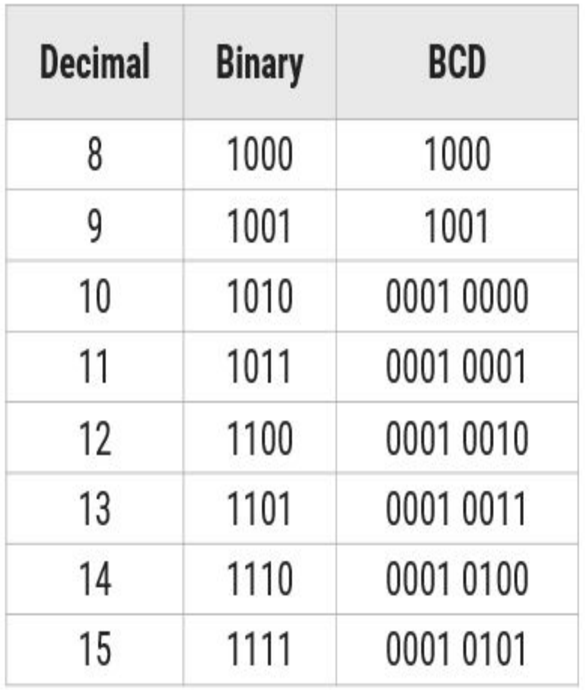
```
   43.25                          00101011.0100
                2's complement
 - 89.75        ────────────→   + 10100110.0100
 ────────                       ─────────────────
 - 46.50                          11010001.1000
                2's complement
                                  00101110.1000
```

As carry is not generated, so take 2's complement of the intermediate result and add ' – ' sign to the result

BCD ARITHMETIC



Decimal	Binary	BCD
0	0	0000
1	1	0001
2	10	0010
3	11	0011
4	100	0100
5	101	0101
6	110	0110
7	111	0111



Decimal	Binary	BCD
8	1000	1000
9	1001	1001
10	1010	0001 0000
11	1011	0001 0001
12	1100	0001 0010
13	1101	0001 0011
14	1110	0001 0100
15	1111	0001 0101

BCD Addition
Example - 1

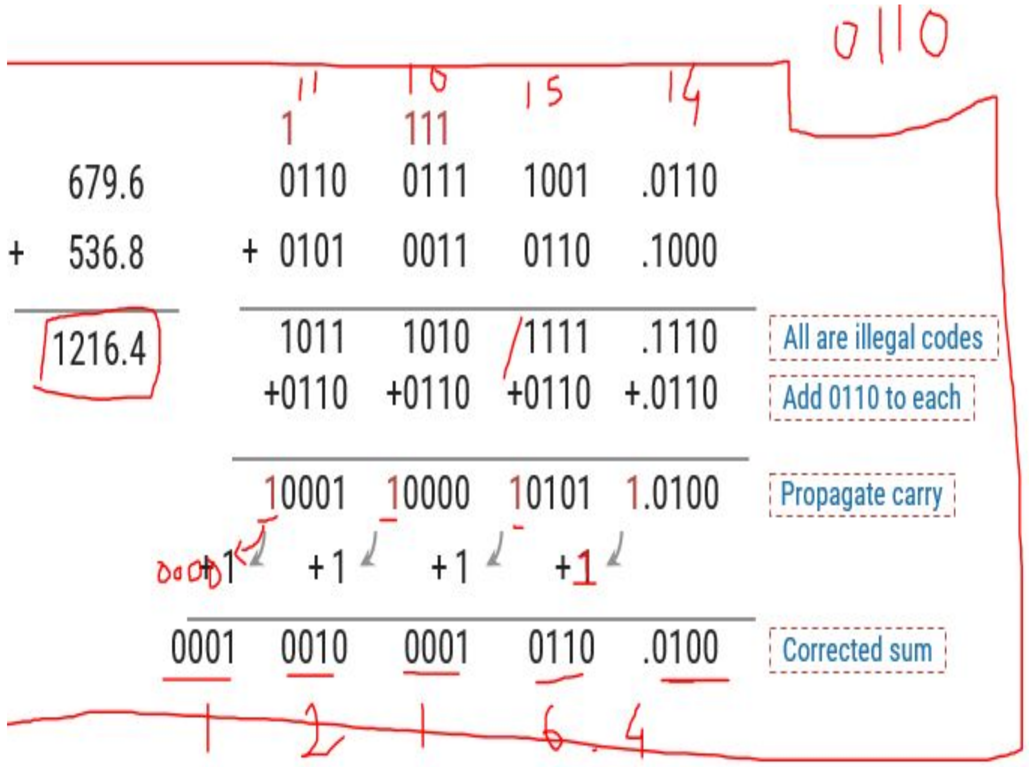
```
                        1   1   1
    2  5      0  0  1  0   0   1   0   1

  + 1  3    + 0  0  0  1   0   0   1   1
  ------    ---------------------------
    3  8      0  0  1  1   1   0   0   0
```

No carry, no illegal code. So, this is the correct sum.

Rule: If there is an illegal code or carry is generated as a result of addition, then add 0110 to particular that 4 bits of result.

BCD Addition

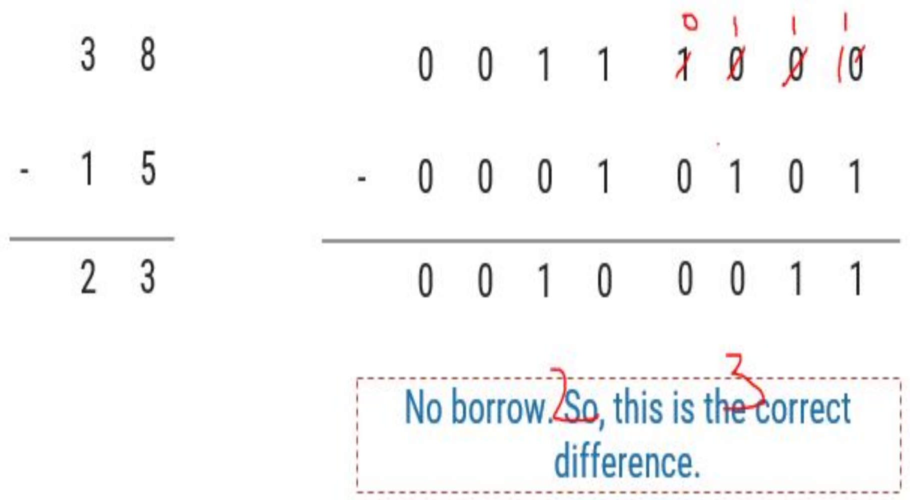
Example - 2



```
                                                              0110
                   11       10       15       14
                   1        111
      679.6        0110     0111     1001     .0110
  +   536.8      + 0101     0011     0110     .1000
     1216.4        1011     1010     1111     .1110    All are illegal codes
                  +0110    +0110    +0110    +.0110    Add 0110 to each
                   10001    10000    10101    1.0100   Propagate carry
        0001 +1      +1       +1       +1
        0001       0010     0001     0110     .0100    Corrected sum
          1          2        1        6  .     4
```

BCD Subtraction

Example - 1



$$\begin{array}{rcc} & 3 & 8 \\ - & 1 & 5 \\ \hline & 2 & 3 \end{array}$$

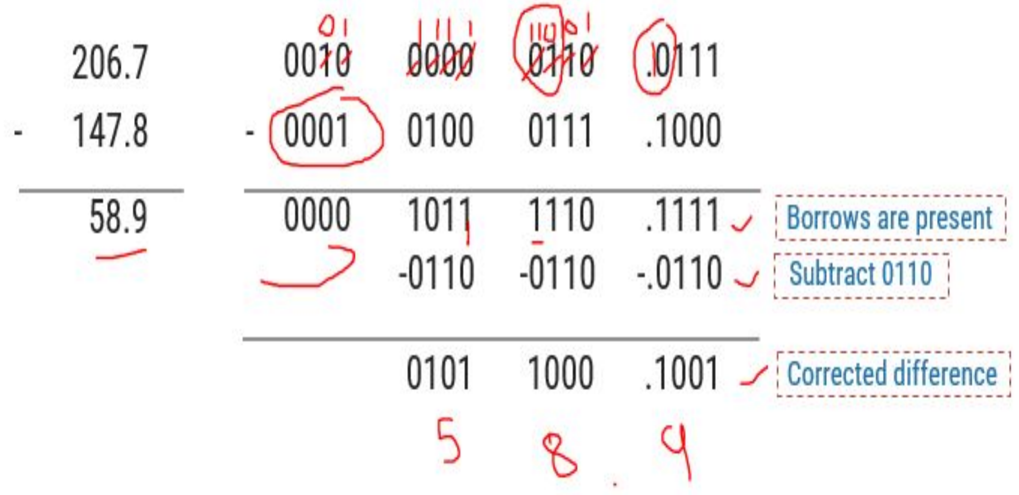
$$\begin{array}{ccccccccc} & 0 & 0 & 1 & 1 & \overset{0}{\cancel{1}} & \overset{1}{\cancel{0}} & \overset{1}{\cancel{0}} & \overset{1}{1\cancel{0}} \\ - & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ \hline & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{array}$$

No borrow. 2 So, this is the 3 correct difference.

Rule: If one 4-bit group needs to take borrow from neighbor, then subtract 0110 from the group which is receiving borrow.

BCD Subtraction

Example - 2



```
  206.7      0010   0000   0110   .0111
- 147.8    - 0001   0100   0111   .1000
  -----    ------------------------------
   58.9      0000   1011   1110   .1111   Borrows are present
                   -0110  -0110  -.0110   Subtract 0110
           ------------------------------
                    0101   1000   .1001   Corrected difference
                     5      8   .   9
```

Logic Gates

Goal:

To understand how digital a computer can work, at the lowest level.

To understand what is possible and the limitations of what is possible for a digital computer.

Logic Gates

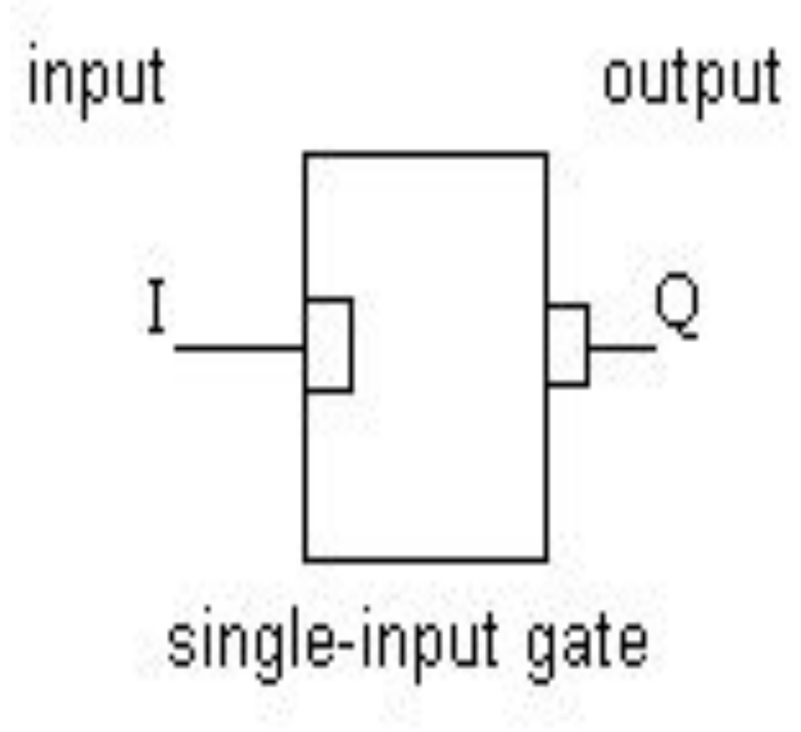
- All digital computers for the past 50 years have been constructed using the same type of components.
- These components are called logic gates.
- Logic gates have been implemented in many different ways.
- Currently, logic gates are most commonly implemented using electronic VLSI transistor logic.

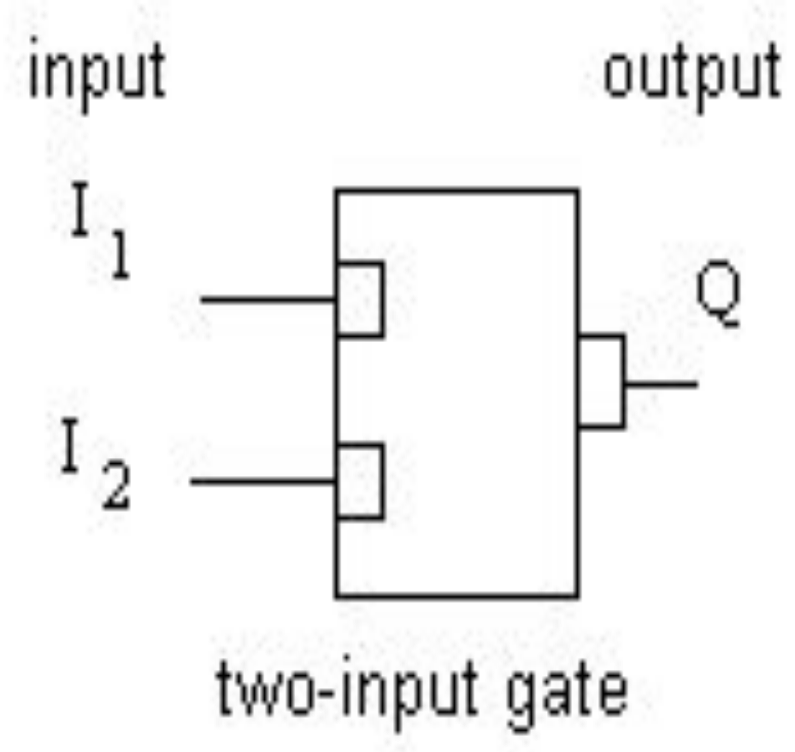
Logic Gates

- A logic gate is a simple switching circuit that determines whether an input pulse can pass through to the output in digital circuits.
- The building blocks of a digital circuit are logic gates, which execute numerous logical operations that are required by any digital circuit.
- These can take two or more inputs but only produce one output.
- The mix of inputs applied across a logic gate determines its output. Logic gates use Boolean algebra to execute logical processes.
- Logic gates are found in nearly every digital gadget we use on a regular basis.
- Logic gates are used in the architecture of our telephones, laptops, tablets, and memory devices.

Logic Gates

All basic logic gates have the ability to accept either one or two input signals (depending upon the type of gate) and generate one output signal.





Boolean Algebra

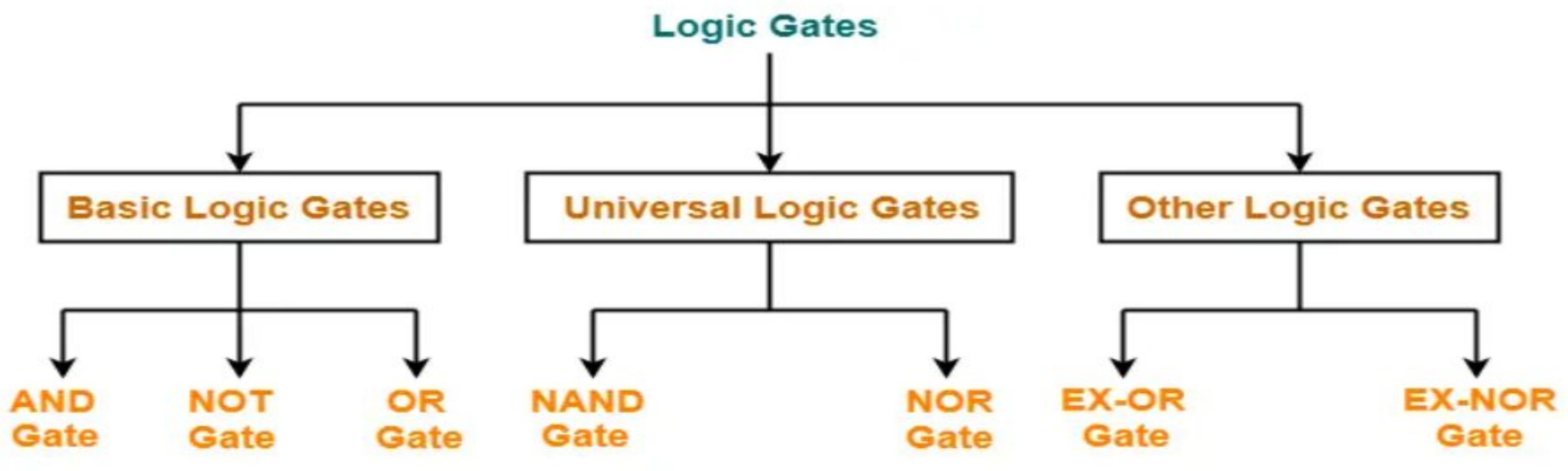
- Boolean algebra is a type of logical algebra in which symbols represent logic levels.
- The digits(or symbols) 1 and 0 are related to the logic levels in this algebra; in electrical circuits, logic 1 will represent a closed switch, a high voltage, or a device's "on" state.
- An open switch, low voltage, or "off" state of the device will be represented by logic 0.
- At any one time, a digital device will be in one of these two binary situations. A light bulb can be used to demonstrate the operation of a logic gate.
- When logic 0 is supplied to the switch, it is turned off, and the bulb does not light up.
- The switch is in an ON state when logic 1 is applied, and the bulb would light up.
- In integrated circuits (IC), logic gates are widely employed.

- Input and Output signals are binary.
    - binary:
        - always in one of two possible states;
        - typically treated as:
            - On / Off (electrically)
            - 1 / 0
            - True / False
- There is a delay between when a change happens at a logic gates inputs and when the output changes, called gate switching time.
- The True or False view is most useful for thinking about the meaning of the basic logic gates.

Truth Table

- The outputs for all conceivable combinations of inputs that may be applied to a logic gate or circuit are listed in a truth table.
- When we enter values into a truth table, we usually express them as 1 or 0, with 1 denoting True logic and 0 denoting False logic.

Classification



Types of Logic Gates

Basic Logic Gates-

- Basic Logic Gates are the fundamental logic gates using which universal logic gates and other logic gates are constructed.

They have the following properties-

- Basic logic gates are associative in nature.
- Basic logic gates are commutative in nature.

There are following three basic logic gates-

1. AND Gate
2. OR Gate
3. NOT Gate

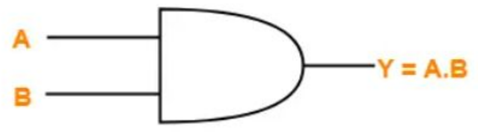
AND gate

- The output of AND gate is high (‘1’) if all of its inputs are high (‘1’).
- The output of AND gate is low (‘0’) if any one of its inputs is low (‘0’).

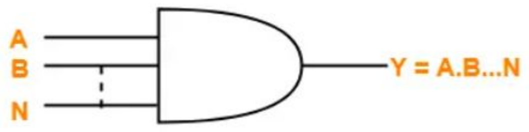
Logic Symbol

Logic Symbol-

The logic symbol for AND Gate is as shown below-

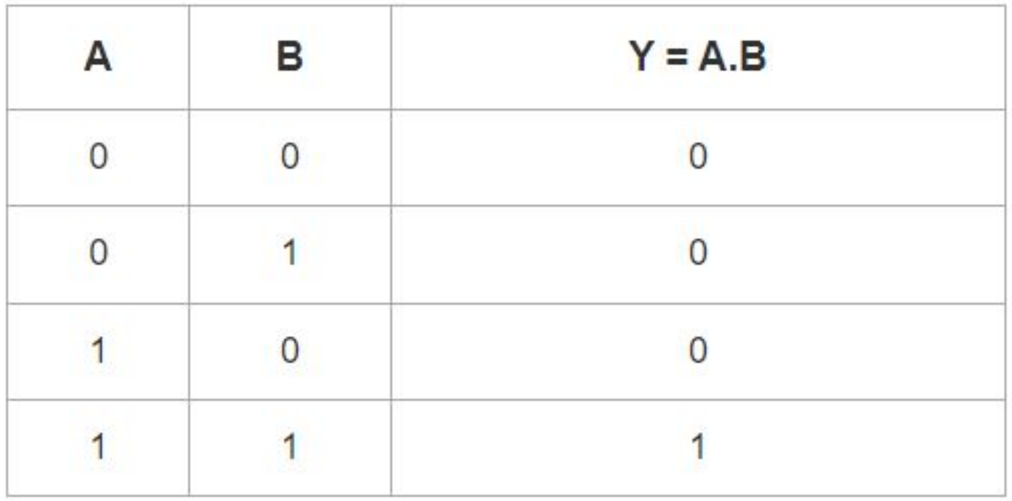


2-Input AND Gate



N-Input AND Gate

Truth Table



A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table

OR Gate

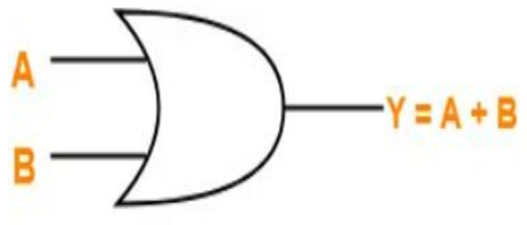
The output of OR gate is high (‘1’) if any one of its inputs is high (‘1’).

The output of OR gate is low (‘0’) if all of its inputs are low (‘0’).

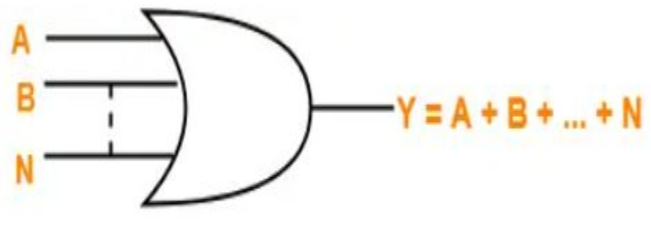
Logic Symbol

Logic Symbol-

The logic symbol for OR Gate is as shown below-

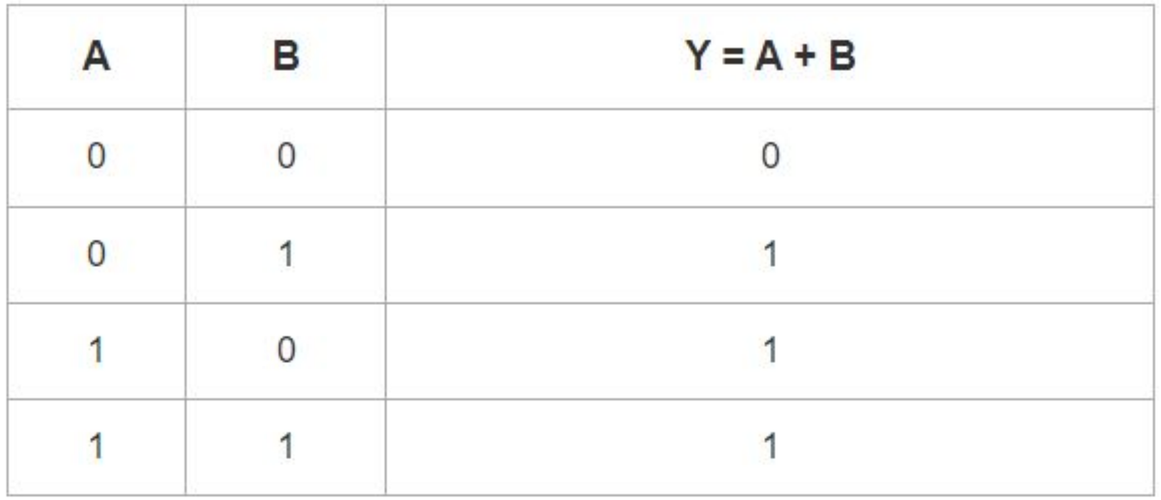


2-Input OR Gate



N-Input OR Gate

Truth Table



A	B	Y = A + B
0	0	0
0	1	1
1	0	1
1	1	1

Truth Table

NOT Gate

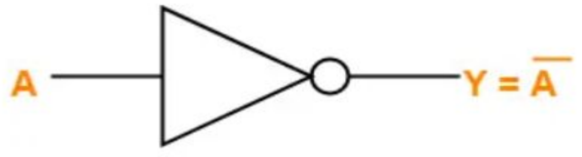
- The output of NOT gate is high (‘1’) if its input is low (‘0’).
- The output of NOT gate is low (‘0’) if its input is high (‘1’).

From here-

- It is clear that NOT gate simply inverts the given input.
- Since NOT gate simply inverts the given input, therefore it is also known as **Inverter Gate**.

Logic Symbol-

The logic symbol for NOT Gate is as shown below-



NOT Gate

A	Y = A’
0	1
1	0

Truth Table

Universal Logic Gates

Universal logic gates are the logic gates that are capable of implementing any Boolean function without requiring any other type of gate.

They are called as "**Universal Gates**" because-

- They can realize all the binary operations.
- All the basic logic gates can be derived from them.

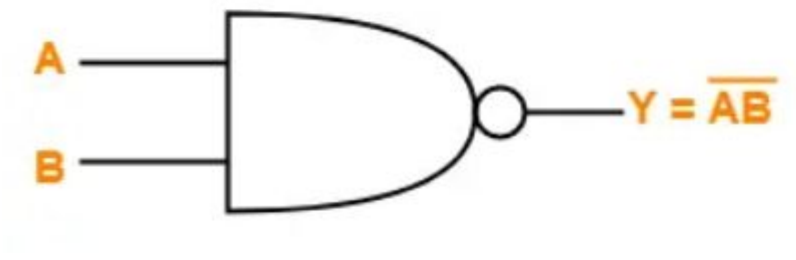
They have the following properties-

- Universal gates are not associative in nature.
- Universal gates are commutative in nature.

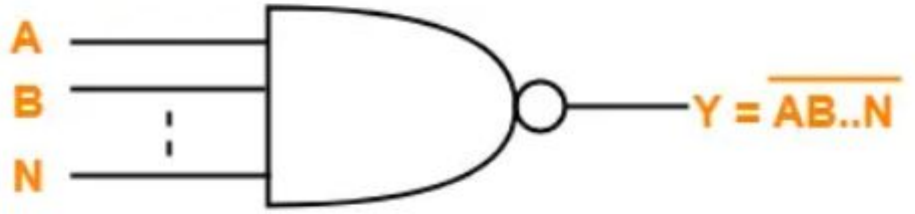
- A NAND Gate is constructed by connecting a NOT Gate at the output terminal of the AND Gate.
- The output of NAND gate is high (‘1’) if at least one of its inputs is low (‘0’).
- The output of NAND gate is low (‘0’) if all of its inputs are high (‘1’).

Logic Symbol-

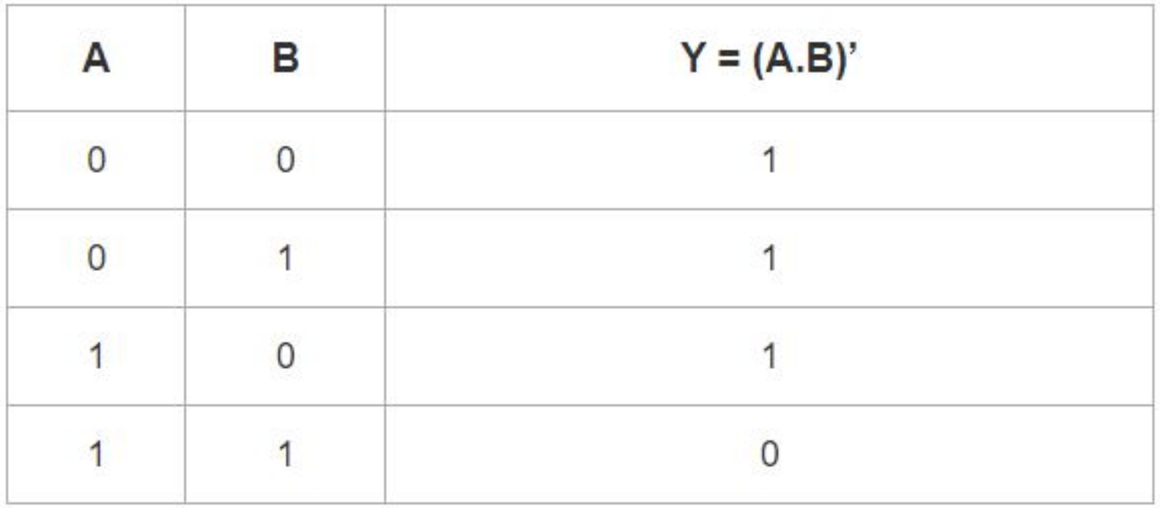
The logic symbol for NAND Gate is as shown below-



2-Input NAND Gate



N-Input NAND Gate



A	B	Y = (A.B)’
0	0	1
0	1	1
1	0	1
1	1	0

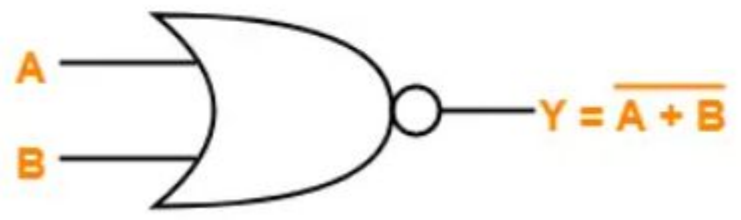
Truth Table

NOR Gate

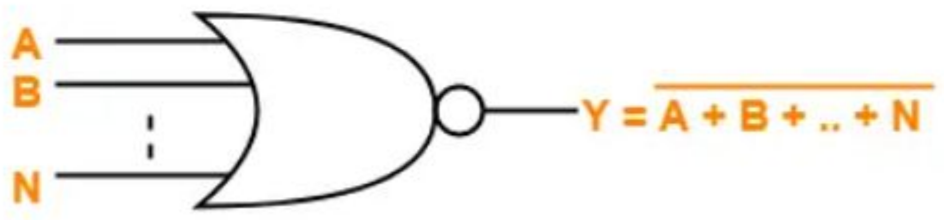
- A NOR Gate is constructed by connecting a NOT Gate at the output terminal of the OR Gate.
- The output of OR gate is high (‘1’) if all of its inputs are low (‘0’).
- The output of OR gate is low (‘0’) if any of its inputs is high (‘1’).

Logic Symbol-

The logic symbol for NOR Gate is as shown below-

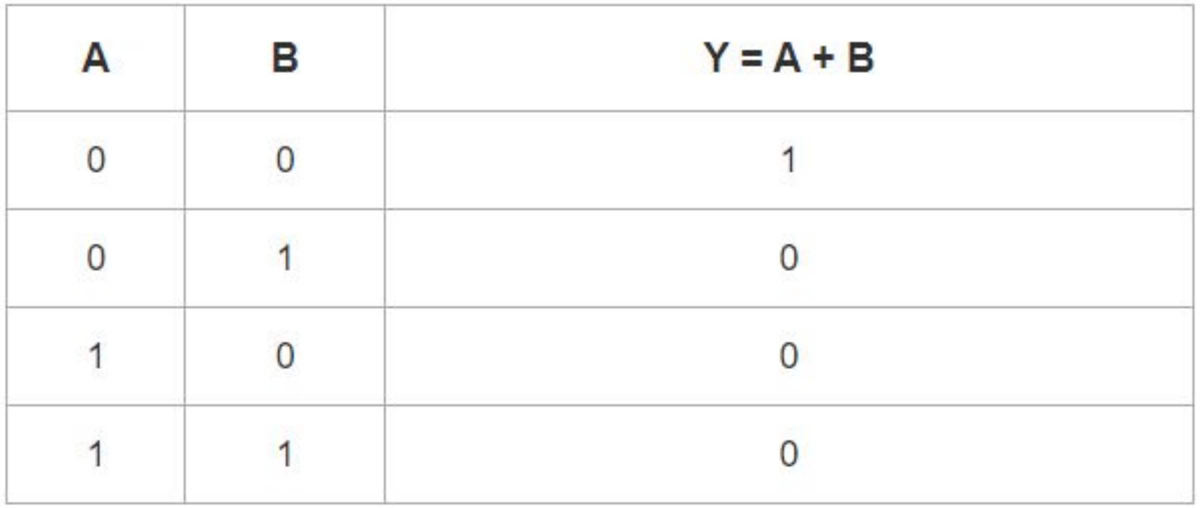


2-Input NOR Gate



N-Input NOR Gate

Truth Table



A	B	Y = A + B
0	0	1
0	1	0
1	0	0
1	1	0

Truth Table

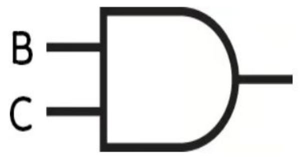
EX-OR & EX-NOR Gates

- One of the inputs of alternative gate will have a bubble (which represents NOT gate).
- For EX-OR structured original gate, alternative gate will be EX-NOR structured.
- For EX-NOR structured original gate, alternative gate will be EX-OR structured.
- If bubble is present at the output of original gate, then no bubble will be present at the output of alternative gate.
- If bubble is not present at the output of original gate, then a bubble will be present at the output of alternative gate.

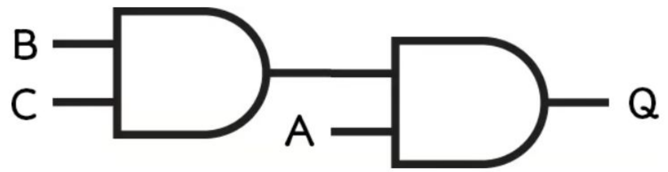
Example

1.Q = A AND (B AND C)

Step 1 – Start with the brackets, this is the “B AND C” part.



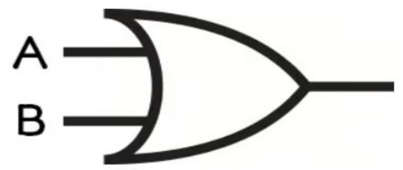
Step 2 – Add the outer expression, this is the “A AND” part.



Example

2.Q = NOT (A OR B)

Step 1 – Start with the brackets, this is the “A OR B” part



Step 2 – Add the outer expression, this is the “NOT” part.