# Introduction to Telecommunications and Computer Engineering

## *Unit 2: Number Systems and Logic*

***Syedur Rahman***

***Lecturer, CSE Department***
***North South University***
***syedur.rahman@wolfson.oxon.org***

# *Acknowledgements*

These notes contain material from the following sources:

[1] *Number and Computer Systems*, by R. Palit, CSE Department, North South University, 2005.

[2] *Binary Numeral System, Binary Operations, etc.*, Wikipedia, http://www.wikipedia.org, 2007.

[3] *Basic Digital Logic*, by P. Godin, Western Canadian Robotics Society, 2004.

[4] *Discrete Mathematics and Its Applications* by K. Rosen, 5th Edition, Tata McGraw-Hill Ed., 2002.

# *Binary Numbers*

The **binary numeral system**, or base-2 number system, is a numeral system that represents numeric values using two symbols, usually ***0 and 1***. More specifically, the usual ***base-2 system*** is a positional notation with a ***radix of 2***.

Owing to its straightforward implementation in electronic circuitry, the binary system is used internally by virtually all modern computers.

# Binary to Decimal Comparison

$$500 = (5 \times 10^2) + (0 \times 10^1) + (0 \times 10^0)$$

$$5834 = (5 \times 10^3) + (8 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

From [2]

$$1_2 = 1 \times 2^0 = 1 \times 1 = 1_{10}$$

$$10_2 = (1 \times 2^1) + (0 \times 2^0) = 2 + 0 = 2_{10}$$

$$101_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 4 + 0 + 1 = 5_{10}$$

| $10^3$ x1000 | $10^2$ x100 | $10^1$ x10 | $10^0$ x1 |
|---|---|---|---|
| 5 | 8 | 4 | 3 |

| Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|
| $2^3$ x8 | $2^2$ x4 | $2^1$ x2 | $2^0$ x1 |
| 0 | 1 | 0 | 1 |

Most Significant Bit (MSB)

Least Significant Bit (LSB)

| Binary | Decimal |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 10 | 2 |
| 11 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | 10 |

# *Binary to Decimal Comparison*

## Addition Comparison

Decimal Addition of Digits Examples

- 5 + 3 = 8
- 7 + 9 = 16 (carry 1)
- and many more…

Binary Addition of Digits (all possible)

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 10 (0 carry:1)
- 1+1+1 = 11 (1 carry:1)

## Decimal to Binary Conversion

$$53_{10} = ?_2$$

First divide 53 repeatedly by 2

53 / 2 = 26 Rem 1

26 / 2 = 13 Rem 0

13 / 2 = 6 Rem 1

6 / 2 = 3 Rem 0

3 / 2 = 1 Rem 1

1 / 2 = 0 Rem 1

Write the remainders backward

$$53_{10} = 110101_2$$

# *Octal Numeral System*

The **octal** numeral system, or **oct** for short, is the base-8 number system, and uses the digits 0 to 7.

Octal numerals can be made from binary numerals by grouping consecutive digits into groups of three (starting from the right). For example, the binary representation for decimal 74 is 1001010, which groups into 001 001 010 — so the octal representation is 112.

Note: To ***convert a decimal number to octal or hexadecimal***, divide the number repeatedly by 8 or 16 respectively and the read the remainders backwards.

# *Hexadecimal Numeral System*

**Hexadecimal**, base-16, or simply **hex**, is a numeral system with a radix, or base, of 16, usually written using the symbols 0–9 and A–F (A=10, B=11, C=12, D=13, E=14, F=15).

Its primary purpose is to represent the binary code in a format easier for humans to read, and acts as a form of shorthand, in which one hexadecimal digit stands in place of four binary bits.

For example, the decimal numeral 79, whose binary representation is 01001111, is 4F in hexadecimal (4 = 0100, F = 1111).

# *Binary, Octal and Hex Conversion*

Since $8 = 2^3$ and $16 = 2^4$, it is very simple to convert between octal and binary or between hexadecimal and binary numbers, without having to convert the number into decimal first.

For converting a binary number into octal, the number can be divided into groups of three bits (starting from the right) and each group can be written as an octal digit. The reverse can be done to convert the number back to binary. Note that each octal digit must be converted into three bits (e.g. $7_8$ gives $111_2$, $2_8$ gives $010_2$ and not $10_2$)

For converting a binary number into hexadecimal, the number can be divided into groups of four bits (starting from the right) and each group can be written as an hexadecimal digit. The reverse can be done to convert the number back to binary, i.e. each hex digit gives four bits.

What is the easiest way to convert between octal and hexadecimal?

# *Digits Required for Numbers*

If $M$ is a number containing $n$ digits with radix $r$, we can write:

$$\left(r^{n}-1\right) \geq M$$

$$r^{n} \geq M+1$$

$$n \log r \geq \log(M+1)$$

$$n \geq \frac{\log(M+1)}{\log r}$$

Therefore, the minimum number of digits $n_{min}$, required to represent $M$ in radix $r$ is:

$$n_{\min} = \left\lceil \frac{\log(M+1)}{\log r} \right\rceil$$

From [1]

For example, to represent 15502 in hexadecimal (radix 16) we need a minimum of (log 15503)/(log 16) = 4.19/1.20 = 3.48 i.e. 4 digits

# *Negative Binary Numbers*

- Sign and Magnitude
- Bias/Excess-N
- One's Complement
- Two's Complement

# *Sign and Magnitude*

The sign and magnitude approach is to represent a number's sign by allocating one sign bit to represent the sign: set that bit (often the most significant bit) to 0 for a positive number, and set to 1 for a negative number. The remaining bits in the number indicate the magnitude (or absolute value).

Note that this number system has two representation of 0s.

It is not possible to simply “add” two s&m numbers to get their sum

| Binary value | S&M Intrp. |
|---|---|
| 00000000 | 0 |
| 00000001 | 1 |
| ... | ... |
| 01111111 | 127 |
| 10000000 | –0 |
| ... | ... |
| 11111111 | –127 |

# *Bias/Excess-N*

Excess-N, also called biased representation, uses a pre-specified number N as a biasing value. A value is represented by the unsigned number which is N greater than the intended value. Thus 0 is represented by N, and –N is represented by the all-zeros bit pattern.

The range of signed numbers using Excess-127 in a conventional eight-bit byte is $-127_{10}$ to $+128_{10}$.

| Binary value | Excess -127 intr |
|---|---|
| 00000000 | -127 |
| 00000001 | -126 |
| ... | ... |
| 01111111 | 0 |
| 10000000 | +1 |
| ... | ... |
| 11111111 | +128 |

# *Complements*



Given a number $a$ in radix $r$ having $n$ digits, **the $(r - 1)$'s complement of $a$ is defined as $(r^n - 1) - a$.**

**The $r$'s complement of $a$ is defined as $r^n - a$.**

Looking closely at the definition of $(r - 1)$'s complement we see that to obtain the $r$'s complement we need only add 1 to the $(r - 1)$'s complement

# *One's Complement (1C)*

The one's complement form of a negative binary number is the bitwise NOT applied to it — the complement of its positive counterpart. Ones' complement has two representations of 0: 00000000 (+0) and 11111111 (–0). The range of signed numbers with 8-bits in 1C is from $-127_{10}$ to $+127_{10}$, or from $-2^{k-1}$ to $(2^{k-1}-1)$ with k bits

To add two numbers represented in this system, one does a conventional binary addition, but it is then necessary to add any resulting carry back into the resulting sum. To see why this is necessary, consider the following example showing the case of the addition of –1 (11111110) to +2 (00000010).

| Binary value | 1'C Intr. |
|---|---|
| 00000000 | 0 |
| 00000001 | 1 |
| ... | ... |
| 01111111 | 127 |
| 11111111 | –0 |
| ... | ... |
| 10000000 | –127 |

# *Two's Complement (2C)*

The two's complement form of a negative binary number is the bitwise NOT applied to it — the complement of its positive counterpart, and then adding 1 to it. In 2C there is only representation of 0. To make a 2C negative number positive, one may bitwise NOT the bits again and add 1 to it.

Addition of a pair of two's-complement integers is the same as addition of a pair of unsigned numbers (except for detection of overflow, if that is done).

The range of signed numbers using two's complement in a conventional eight-bit byte is −12810 to +12710, or $-2^{k-1}$ to $(2^{k-1}-1)$ using k bits.

| Binary Number | 2'C Interpr |
|---|---|
| 00000000 | 0 |
| 00000001 | 1 |
| ... | ... |
| 01111101 | 125 |
| 01111110 | 126 |
| 01111111 | 127 |
| 10000000 | −128 |
| 10000001 | −127 |
| 10000010 | −126 |
| ... | ... |
| 11111110 | −2 |
| 11111111 | −1 |

From [2]

# *Two's Complement*

One may consider a *k*-bit 2C representation as having a negative weight for the most significant bit i.e. bit *k*-1. So in a 8-bit 2C system bit-7 has weight -128 ($-2^7$) instead of 128 ($2^7$). All –ve numbers have this bit as 1.

| x -128 | x 64 | x 32 | x 16 | x 8 | x 4 | x 2 | x 1 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

Computing $-15_{10}+27_{10}$

| x -128 | x 64 | x 32 | x 16 | x 8 | x 4 | x 2 | x 1 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

Computing $113_{10}+27_{10}$

# *Two's Complement*

One may consider a *k*-bit 2C representation as having a negative weight for the most significant bit i.e. bit *k*-1. So in a 8-bit 2C system bit-7 has weight -128 ($-2^7$) instead of 128 ($2^7$). All –ve numbers have this bit as 1.

| x -128 | x 64 | x 32 | x 16 | x 8 | x 4 | x 2 | x 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| | | | | | | | |

Computing $-15_{10}+27_{10}$

$= -128+64+32+16+1 = -15_{10}$

$= 16+8+2+1 = 27_{10}$

| x -128 | x 64 | x 32 | x 16 | x 8 | x 4 | x 2 | x 1 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

Computing $113_{10}+27_{10}$

# *Two's Complement*

One may consider a *k*-bit 2C representation as having a negative weight for the most significant bit i.e. bit *k*-1. So in a 8-bit 2C system bit-7 has weight -128 ($-2^7$) instead of 128 ($2^7$). All –ve numbers have this bit as 1.

| x -128 | x 64 | x 32 | x 16 | x 8 | x 4 | x 2 | x 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

1

Computing $-15_{10}+27_{10}$

$= -128+64+32+16+1 = -15_{10}$

$= 16+8+2+1 = 27_{10}$

$= 8 + 4 = 12_{10}$

| x -128 | x 64 | x 32 | x 16 | x 8 | x 4 | x 2 | x 1 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| | | | | | | | |

Computing $113_{10}+27_{10}$

$= 64+32+16+1 = 113_{10}$

$= 16+8+2+1 = 27_{10}$

# *Two's Complement*

One may consider a *k*-bit 2C representation as having a negative weight for the most significant bit i.e. bit *k*-1. So in a 8-bit 2C system bit-7 has weight -128 ($-2^7$) instead of 128 ($2^7$). All –ve numbers have this bit as 1.

| x -128 | x 64 | x 32 | x 16 | x 8 | x 4 | x 2 | x 1 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

1

Computing $-15_{10}+27_{10}$

$= -128+64+32+16+1 = -15_{10}$

$= 16+8+2+1 = 27_{10}$

$= 8 + 4 = 12_{10}$

| x -128 | x 64 | x 32 | x 16 | x 8 | x 4 | x 2 | x 1 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

Computing $113_{10}+27_{10}$

$= 64+32+16+1 = 113_{10}$

$= 16+8+2+1 = 27_{10}$

$= -116_{10}$!!! (overflow)

# Comparison using a 4-bit system

| *Deci-mal* | *Unsig-ned* | *S&M Repres.* | *Ones'C Repres.* | *Two's C Repres.* | *Excs-7 Repres* |
|---|---|---|---|---|---|
| +8 | 1000 | N/A | N/A | N/A | 1111 |
| +7 | 0111 | 0111 | 0111 | 0111 | 1110 |
| +6 | 0110 | 0110 | 0110 | 0110 | 1101 |
| +5 | 0101 | 0101 | 0101 | 0101 | 1100 |
| +4 | 0100 | 0100 | 0100 | 0100 | 1011 |
| +3 | 0011 | 0011 | 0011 | 0011 | 1010 |
| +2 | 0010 | 0010 | 0010 | 0010 | 1001 |
| +1 | 0001 | 0001 | 0001 | 0001 | 1000 |
| (+)0 | 0000 | 0000 | 0000 | 0000 | 0111 |

| *Deci-mal* | *Unsig-ned* | *S&M Repres.* | *One's C Repres.* | *Two's C Repres.* | *Excs-7 Repres* |
|---|---|---|---|---|---|
| (–)0 | N/A | 1000 | 1111 | N/A | N/A |
| –1 | N/A | 1001 | 1110 | 1111 | 0110 |
| –2 | N/A | 1010 | 1101 | 1110 | 0101 |
| –3 | N/A | 1011 | 1100 | 1101 | 0100 |
| –4 | N/A | 1100 | 1011 | 1100 | 0011 |
| –5 | N/A | 1101 | 1010 | 1011 | 0010 |
| –6 | N/A | 1110 | 1001 | 1010 | 0001 |
| –7 | N/A | 1111 | 1000 | 1001 | 0000 |
| –8 | N/A | N/A | N/A | 1000 | N/A |

From [2]

# *Fixed and Floating Point Binary*

Binary mixed numbers and fractions can be represented using a fixed-point or floating point representation.

A **fixed-point binary representation** is a real data type for a number that has a fixed number of bits before and after the radix point. In terms of binary numbers, each magnitude bit represents a power of two, while each fractional bit represents an inverse power of two.

| $2^4$ x16 | $2^3$ X8 | $2^2$ x4 | $2^1$ x2 | $2^0$ x1 | $2^{-1}$ x½ | $2^{-2}$ x¼ | $2^{-3}$ x⅛ |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |

The binary system on the left has 5 bits before the radix point and 3 bits after it

= 01110.101

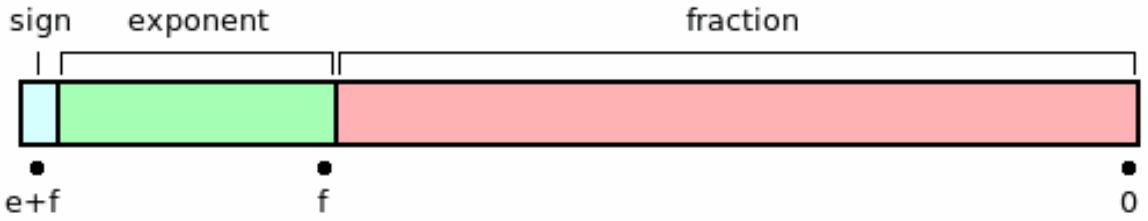
= 8 + 4 + 2 + ½ + ⅛ = 14.625

**Floating-point binary** refers to the fact that the radix point can be placed anywhere relative to the digits within the string. This position is indicated separately in the internal representation, and this representation can thus be thought of as a computer realization of scientific notation.

# *IEEE Standard 754 Floating Point Numbers*

IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms.

| | Sign | Exponent | Fraction | Bias |
|---|---|---|---|---|
| **Single Precision** | **1 [31]** | **8 [30-23]** | **23 [22-00]** | **127** |
| **Double Precision** | **1 [63]** | **11 [62-52]** | **52 [51-00]** | **1023** |

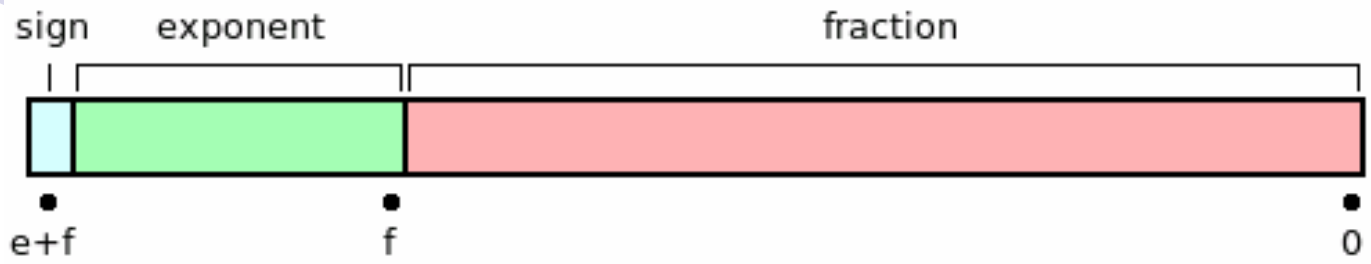



From [2]

# *Range of IEEE 754 Numbers*

| | Denormalized Range | Normalized Range | Approximate Decimal |
|---|---|---|---|
| **Single Precision** | $\pm 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$ | $\pm 2^{-126}$ to $(2-2^{-23})\times 2^{127}$ | $\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$ |
| **Double Precision** | $\pm 2^{-1074}$ to $(1-2^{-52})\times 2^{-1022}$ | $\pm 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$ | $\pm \sim 10^{-323.3}$ to $\sim 10^{308.3}$ |

# IEEE754 Single Precision




$$n = (-1)^s \times 2^e \times m$$

From [2]

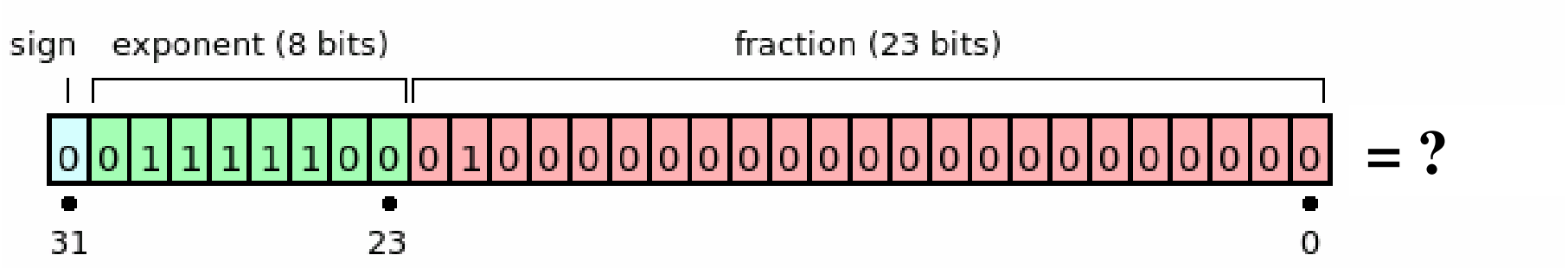
Where

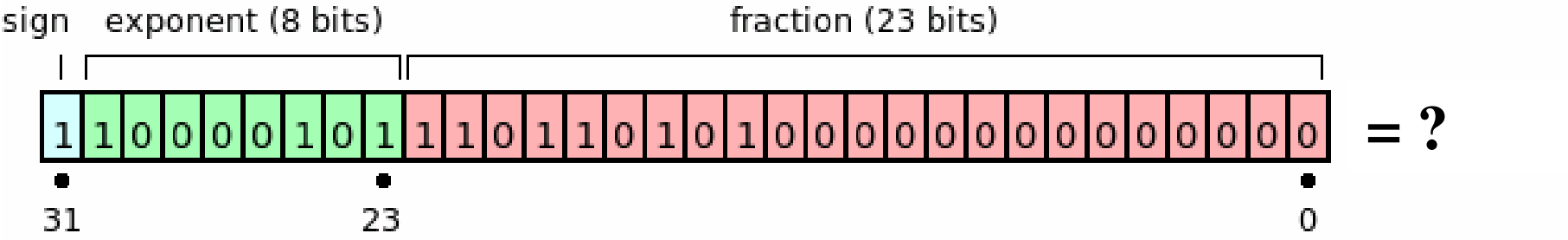
$s$ = the sign bit

$e$ = exp – 127 (in other words the exponent is stored with 127 added to it, also called "biased with 127" or excess-127)

$m$ = 1.fraction in binary (that is, the significand/mantissa is the binary number 1 followed by the radix point followed by the binary bits of the fraction). Therefore, $1 \leq m < 2$.

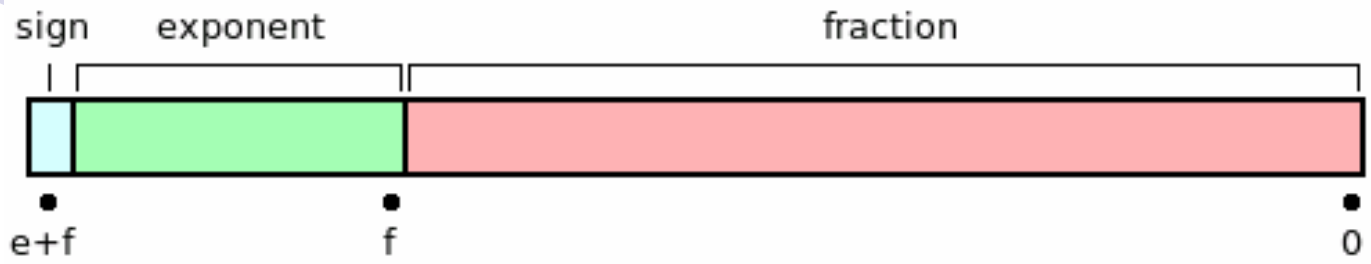
# *Single Precision Float Examples*




From [2]

# IEEE754 Double Precision




$$n = (-1)^s \times 2^e \times m$$

From [2]

Where

$s$ = the sign bit

$e$ = exp – 1023 (in other words the exponent is stored with 1023 added to it, also called excess-1023).

$m$ = 1.fraction in binary (that is, the significand/mantissa is the binary number 1 followed by the radix point followed by the binary bits of the fraction). Therefore, $1 \leq m < 2$.

# IEEE 754 Numbers - Details

| Type | Exponent | Fraction |
|---|---|---|
| Zeroes | 0 | 0 |
| Denormalized numbers | 0 | non zero |
| Normalized numbers | 1 to $2^k - 2$ | any |
| Infinities | $2^k - 1$ (all 1s) | 0 |
| NaNs | $2^k - 1$ (all 1s) | non zero |

*k* is the number of bits reserved for the exponent, i.e. 8 and 11 bits for single and double precision respectively

# *Denormalized Numbers in IEEE 754*

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a ***denormalized*** number, which does *not* have an assumed leading 1 before the binary point.

Thus for single precision, this represents a number

$$(-1)^s \times 0.f \times 2^{-126}$$

where $s$ is the sign bit and $f$ is the fraction.

For double precision, denormalized numbers represent

$$(-1)^s \times 0.f \times 2^{-1022}$$

From this you can interpret zero as a special type of denormalized number.

# *NaNs (Not a Number) in IEEE 754*

The value **NaN (*Not a Number*)** is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (***Quiet NaN***) and SNaN (***Signalling NaN***).

A **QNaN** is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An **SNaN** is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote ***indeterminate*** operations, while SNaN's denote ***invalid*** operations.

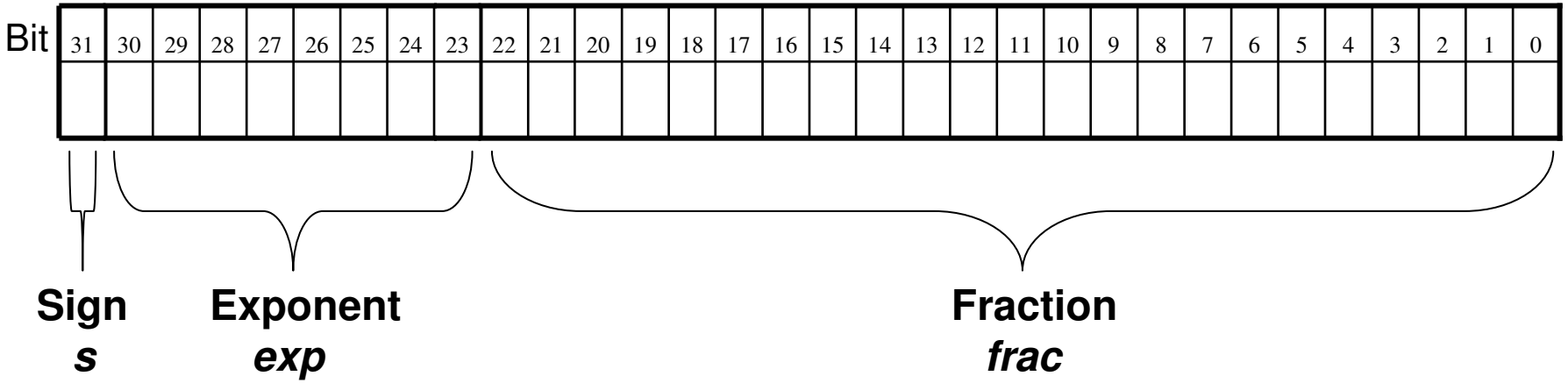
# IEEE 754 – Interpretation Summary

$n = (-1)^s \times 2^{exp-bias} \times 1.frac$ if no. is normalized ($exp \neq 0$)

$n = (-1)^s \times 2^{-bias+1} \times 0.frac$ if no. is denormalized ($exp = 0$)

*bias*=127 for single precision whereas *bias*=1023 for double precision

Example for Single Precision:



# *Special Operations*

Operations on special numbers are well-defined by IEEE. In the simplest case, any operation with a NaN yields a NaN result. Other operations are as follows:

| Operation | Result |
|---|---|
| n ÷ ±Infinity | 0 |
| ±Infinity × ±Infinity | ±Infinity |
| ±nonzero ÷ 0 | ±Infinity |
| Infinity + Infinity | Infinity |
| ±0 ÷ ±0 | NaN |
| Infinity - Infinity | NaN |
| ±Infinity ÷ ±Infinity | NaN |
| ±Infinity × 0 | NaN |

From [1]

# *BCD and ASCII*

The Binary Coded Decimal (**BCD**) system allows numbers to be stored in decimal form with each decimal digit being represented by 4 bits. So, 46 is represented by 01000110. It is however not of much use in binary arithmetic.

| Decimal Digit | BCD 8421 |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |

American Standard Code for Information Interchange (**ASCII**) is a seven-bit code, meaning it uses patterns of seven binary digits (a range of 0 to 127 decimal) to represent each character commonly used for text. The first 32 (0-31) are non-printable control characters. E.g. 13 is carriage return (enter) and 8 is backspace.


ASCII Printable Characters 32-126

# *Gray Code and Error Detecting Code*

**Gray Code**: The reflected binary code, also known as Gray code, is a binary numeral system where two successive values differ in only one digit. The reflected binary code was originally designed to prevent spurious output from electromechanical switches.

**Gray Code**

| 2-bit | 3-bit |
|---|---|
| 00 | 000 |
| 01 | 001 |
| 11 | 011 |
| 10 | 010 |
| | 110 |
| | 111 |
| | 101 |
| | 100 |

**Error Detecting Code**: To detect errors in data communication and processing, an extra bit is sometimes added to indicate its parity (e.g. an 8th bit is added to ASCII). An even or odd ***parity bit*** is an extra bit included with a message to make the total number of 1's either even or odd respectively

| | With even parity | With odd parity |
|---|---|---|
| ASCII A = 1000001 | 01000001 | 11000001 |
| ASCII T = 1010100 | 11010100 | 01010100 |

# *Basics of Digital Logic*

A **statement** is a collection of symbols that has a logic value – either **off/low** (**0**) or **on/high** (**1**).

**Variables** in boolean logic are symbols that have a certain meaning and take a binary value depending on the current situation.

**Connectives** or operators are symbols that are used to form larger statements out of smaller ones.

# *Logical Operations*

A **disjunction** is a compound statement in which two substatements are connected by + (**OR**), e.g. ***p+q***

A **conjunction** is a compound statement in which two substatements are connected by . (**AND**) e.g. ***p.q***

The **negation** (**NOT**) of statement *p* is ***p'*** or $\overline{p}$, meaning the complement of *p*

OR, AND and NOT are basic logical operations. Other logical operations include NAND, NOR, XOR etc.

# *Truth Tables*



A truth table is a mathematical table used in logic to compute the functional values of logical expressions on any of their functional arguments, that is, with respect to the various possible combinations of values that their logical variables may take.

Remember that with $n$ logical variables, the truth table will always have $2^n$ rows.

# Truth Tables: Examples

| $p$ | $q$ | $p.q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $p$ | $\overline{p}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

| $p$ | $q$ | $p+q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| a | b | a AND b<br>a . b | a OR b<br>a + b | a XOR b<br>a ⊕ b | NOT a<br>$\overline{a}$ | a NAND b<br>$\overline{(a \cdot b)}$ | b NOR b<br>$\overline{(a + b)}$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

From [4]

# *Bit-wise Logical Operations*

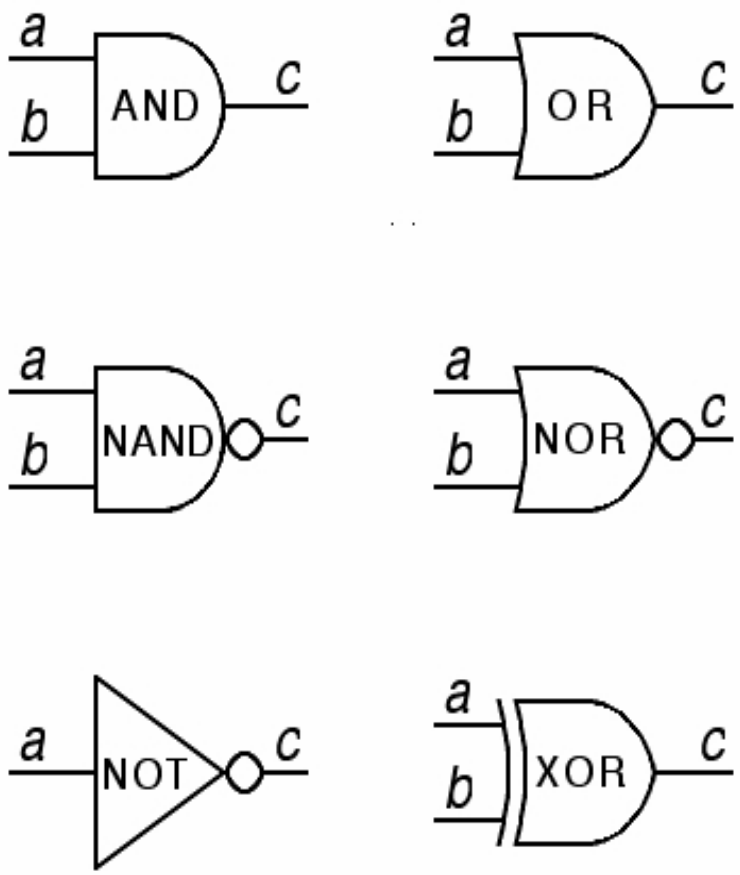
The bit-wise AND/OR operation between two *n*-bit numbers *X* and *Y* can be performed by computing the AND/OR of each bit of *X* with the corresponding bit of *Y*. Similarly the bit-wise NOT of a *n*-bit number *X* can be performed by NOT-ing each of its bits.

Example bit-wise AND, OR and NOT operations with 8 bits:

| AND | OR | |
|---|---|---|
| 00010111 | 01011000 | |
| 01111010 | 00010111 | NOT 11000010 |
| 00010010 | 01011111 | 00111101 |

# *Logic Gates*

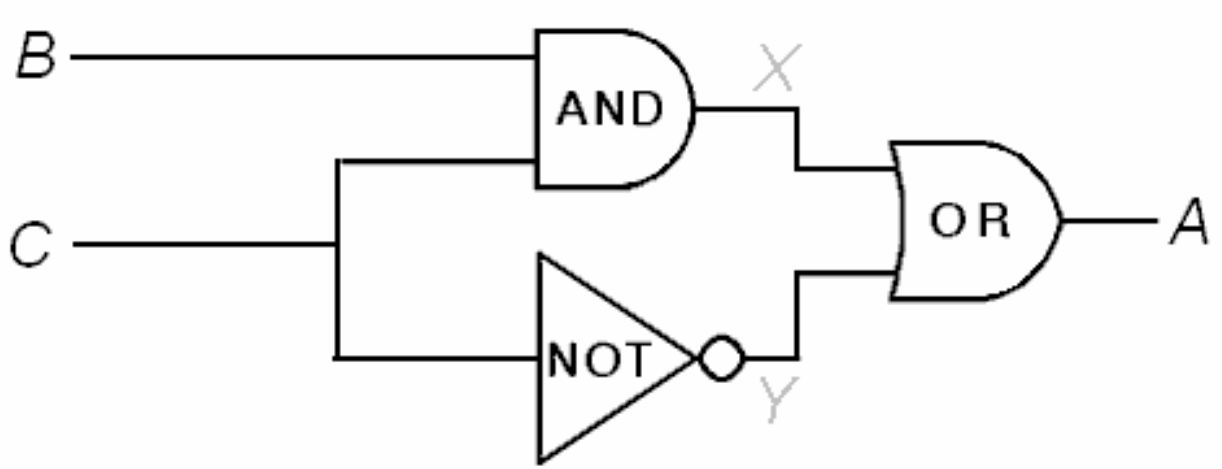
A **logic gate** performs a logical operation on one or more logic inputs and produces a single logic output. The logic normally performed is Boolean logic and is most commonly found in digital circuits.




From [2]

# An example of a logical circuit

$$A = B.C + \overline{C}$$




*Circuit Diagram*

*Truth Table*

| B | C | X | Y | A |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |

## Basic Identities of Boolean Algebra

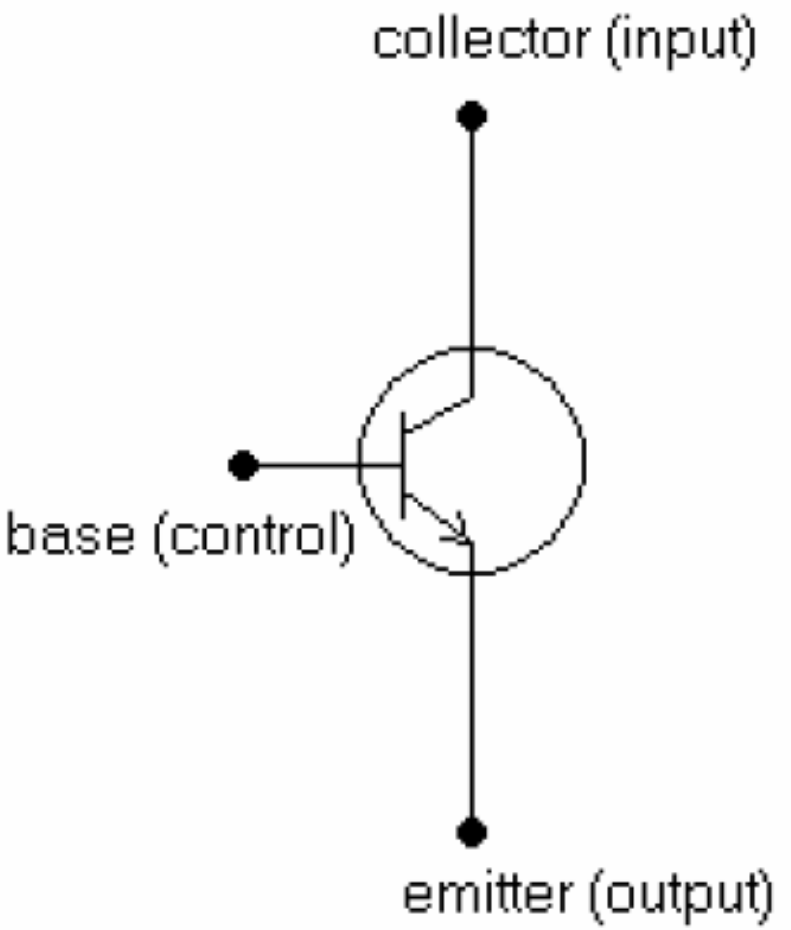
| | | | | |
|---|---|---|---|---|
| 1. | $X+0=X$ | 2. | $X\cdot 1=X$ | |
| 3. | $X+1=1$ | 4. | $X\cdot 0=0$ | |
| 5. | $X+X=X$ | 6. | $X\cdot X=X$ | |
| 7. | $X+\overline{X}=1$ | 8. | $X\cdot\overline{X}=0$ | |
| 9. | $\overline{\overline{X}}=X$ | | | |
| 10. | $X+Y=Y+X$ | 11. | $XY=YX$ | Commutative |
| 12. | $X+(Y+Z)=(X+Y)+Z$ | 13. | $X(YZ)=(XY)Z$ | Associative |
| 14. | $X(Y+Z)=XY+XZ$ | 15. | $X+YZ=(X+Y)(X+Z)$ | Distributive |
| 16. | $\overline{X+Y}=\overline{X}\cdot\overline{Y}$ | 17. | $\overline{X\cdot Y}=\overline{X}+\overline{Y}$ | DeMorgan's |

From [3]

# *Transistors*

**Transistors** are versatile three lead semiconductor devices whose applications include electronic switching and modulation (amplification). They are the building blocks of microcomputers.

Basically a transistors works as a closed switch when a certain voltage is applied to the base and as an open switch otherwise.




From [2]

# *Shift Operations*

Using shift operations the bits in a word/byte are moved, or *shifted*, to the left or right.

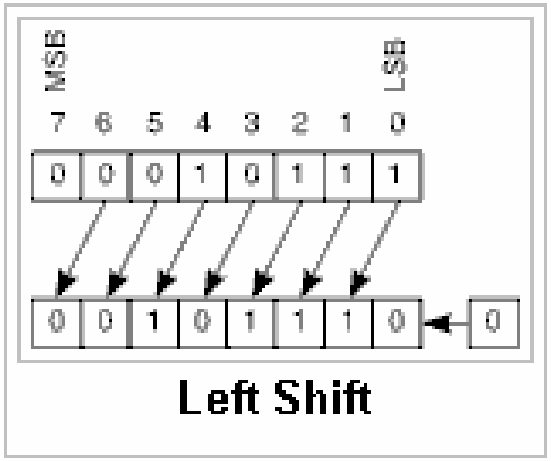
In a ***logical shift***, the bits that are shifted out are discarded, and zeros are shifted in (on either end).

In an ***arithmetic shift***, the bits that are shifted out of either end are discarded. In a left arithmetic shift, zeros are shifted in on the right; in a right arithmetic shift, the sign bit is shifted in on the left, thus preserving the sign of the operand.

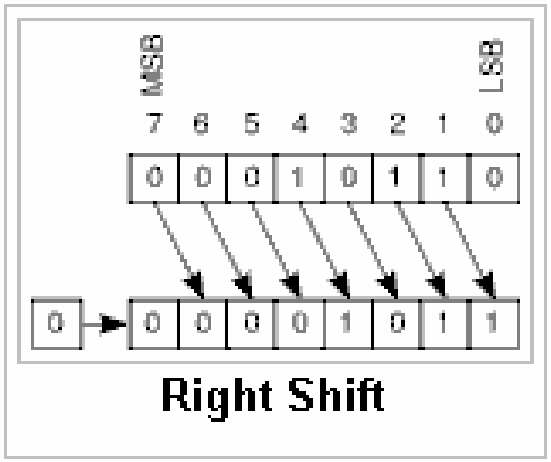
Other shift operations include ***rotate through carry*** **and** ***rotate no carry*** (with either does or does not respectively take into account the special carry bit reserved for arithmetic computations)

# *Shift and Multiplication/Division*

Arithmetic shift operations can be used for performing arithmetic operations such as multiplication by 2 using left shift and division by 2 using right shift.




Multiplications with other numbers can also be performed using shifts, addition and/or subtraction. Examples:
One can multiply *n* with 5, by left shifting *n* twice (i.e. multiplying by 4) and by adding *n* to the result.
One can multiply *n* with 7, by left shifting n thrice (i.e. multiplying by 8) and subtracting *n* from the result.




From [2]