BITs, Bytes & Number Systems

PDEU, Gandhinagar

BIT

- Bit is abbreviation of 'Binary Digit'.
- The smallest Unit in the Computer.
- It can store either 0 or 1 but not both simultaneously.
  i.e. they are mutually exclusive.
- In computer terminology, 1 means on and 0 means off.

BIT

No. of Bits	Different Combinations	No. of Different Combinations $= 2^{No.of\ Bits}$	Highest Value stored $= 2^{No.of\ Bits} - 1$
1	0 1	$2^1 = 2$	$2^1-1 = 1$
2	00 01 10 11	$2^2 = 4$	$2^2-1 = 3$
3	000 001 010 011 100 101 110 111	$2^3 = 8$	$2^3-1 = 7$

Byte

- The smallest unit inside the computer is bit.
- However, a single bit can't be used to store different numbers, alphabets or special symbols.
- So we require a series of bits.
- 8 bits together makes one byte.
- With one byte, we can store 256 different combinations, which include digits, alphabets and special symbols.

BIT-> Tera Byte

Relationship	
8 BITS	= 1 byte
4 BITS	= 1 Nibble
1 Byte	= 2 Nibbles
1024 Bytes	= 1 Kilo Byte (KB)
1024 KB	= 1 Mega Byte (MB)
1024 MB	= 1 Giga Byte (GB)
1024 GB	= 1 Tera Byte (TB)

Q.1. Why 1024 Bytes make 1 Kilo byte?
Q.2. What would be the highest number that we can store, if we are having 9 bits?

Different Number Systems

- Decimal
- Binary
- Octal
- Hexadecimal
- ASCII
- Unicode
- BCD
- EBCDIC

Decimal Number System

- Radix or Base 10
- Digits 0 – 9

- $(153.25)_{10}$ $= (1 * 10^2) + (5 * 10^1) + (3 * 10^0) + (2 * 10^{-1}) + (5 * 10^{-2})$

  or $(100 + 50 + 3 + .2 + .05)$

Binary Number System

- Radix or Base 2
- Digits 0 & 1

Conversion To Decimal

$(101.11)_2$ $= (1 * 2^2) + (0 * 2^1) + (1 * 2^0) + (1 * 2^{-1}) + (1 * 2^{-2})$

$= (4 + 0 + 1 + .5 + .25)$

$= (5.75)_{10}$

Octal Number System

- Radix or Base 8 (2^3)
- Digits 0 – 7

Conversion To Decimal

- $(250.14)_8 = (2 * 8^2) + (5 * 8^1) + (0 * 8^0) + (1 * 8^{-1}) + (4 * 8^{-2})$

$= (128 + 40 + 0 + 0.125 + 0.0625)$

$= (168.1875)_{10}$

Hexadecimal Number System

- Radix or Base 16 (2^4)
- Digits 0 – 9,A->10..F->15

Conversion To Decimal

- $(AB.75)_{16} = (A*16^1)+(B*16^0)+(7*16^{-1})+(5*16^{-2})$

$= (10*16)+(11*1)+(7/16)+(5/256)$

$= (171.45703125)_{10}$

Decimal->Binary Conversion

- $(35.25)_{10} = (?)_2$
- **(for integer part)**
- Keep on dividing 35 by 2 & then take remainder from bottom to top.
- **(For fractional part)**
- Keep on multiplying .25 by 2 unless you get .00. Take digits from top to bottom.

Decimal->Binary Conversion

- $(35.25)_{10} = (?)_2$

2	35	
2	17	1
2	8	1
2	4	0
2	2	0
2	1	0
	0	1

```
.25
*  2
--------
0   .50
*  2
---------
1   .00
```

Decimal->Binary Conversion

2	35	
2	17	1
2	8	1
2	4	0
2	2	0
2	1	0
	0	1

	.25
*	2

0	.50
*	2

1	.00

- $(35.25)_{10} = (100011.01)_2$

Conversion: Binary→Octal

- Two Ways
  - Convert Binary To Decimal;
  - And Decimal To Octal

  Or

  - Use Direct Conversion...

Direct Conversion Binary→Octal

- $(101110111)_2 = (\ ?\ )_8$
- Make a group of 3 bits from Right to Left. i.e. 101 110 111
- Compare them with 421

4	2	1	.
1	0	1	= 5 . . $(567)_8$
1	1	0	= 6
1	1	1	= 7

Conversion: Binary→Hexa

- Two Ways
  - Convert Binary To Decimal;
  - And Decimal To Hexa Decimal

  Or

  - Use Direct Conversion...

Direct Conversion Binary→Hexa

- $(10111111)_2 = (\ ?\ )_{16}$
- Make a group of 4 bits from Right to Left. i.e. 0001 0111 1111
- Compare them with 8421

8	4	2	1		
0	0	0	1	= 1	. . (1 7 F) 16
0	1	1	1	= 7	
1	1	1	1	= 15 i.e. F	

Practice Work

- Convert the following:

$(1110101)_2$ = (?)$_8$ (?)$_{10}$ (?)$_{16}$

$(ACoE)_{16}$ = (?)$_8$ (?)$_{10}$ (?)$_2$

$(007)_8$ = (?)$_2$ (?)$_{10}$ (?)$_{16}$

$(182.75)_{10}$ = (?)$_2$ (?)$_{16}$ (?)$_8$

ASCII

- American Standard Code For Information Interchange
- Most Widely Used Coding System To Represent Data.
- Two Types of ASCII
  - ASCII-7 (128 Diff. Combinations)
  - ASCII-8 (256 Diff. Combinations)

ASCII

- Out of 1 byte's 8 bits, ASCII-7 uses right most 7 bits while ASCII-8 uses all bits.
- Diff. combinations includes
  - 10 digits (0 - 9) (ASCII values 48-57)
  - 26 upper case alphabets (A – Z) (ASCII values 65-90)
  - 26 lower case alphabets (a-z) (ASCII values 97-122)

ASCII

- Diff.Combinations includes
  - 10 digits + 52 alphabets
  - Remaining are special characters and graphics characters.
- To store a single character, we require one byte.
- Example:
- To store 153, we require 3 bytes.
  - 1 = 49 = 00110001
  - 5 = 53 = 00110101
  - 3 = 51 = 00110011

  = $(00110001\ 00110101\ 00110011)_{ASCII}$

UNICODE

- Unicode defines a fully international character set from different languages.
- Character set includes Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, Hindi besides including characters from English, German, Spanish & French.

UNICODE

- It uses 16 bits (2 bytes).
- The diff. combinations are 65536.
- Java uses Unicode as its number system to represent data.
- 1st 256 combinations are same as that of ASCII.

BCD

- **B**inary **C**oded **D**ecimal
- Hybrid Of **Binary** And **Decimal**
- Each digit of A decimal number is converted into its 4-bit binary form
- 0 → 0000
- 1 → 0001
- 9 → 1001

BCD

- With 4 bits only digits can be stored. What about alphabets?
- So, a new method of BCD was developed.
- 2 zone bits were added to 4 bits combination

  00 0000

  Zone bits digit bits

BCD

- 6 BITS i.e. 64 diff. combinations.
- Includes
  - 10 Digits
  - 26 Upper Case Characters
  - 28 Special Characters
- What about 26 Lower Case Characters?

EBCDIC

- Extended Binary Coded Decimal Interchange Code
- Uses 4 Bits as Zone Bits

  0000 0000

  Zone Bits Digit Bits
- 8 Bits = 256 diff. combinations

EBCDIC

- Includes
  - 10 Digits
  - 26 Upper Case Characters
  - 26 Lower Case Characters
  - Rest Printable And Non-printable Control Characters and Special Symbols.
- When we store a number, all zone bits are on i.e. 1111

EBCDIC

- To store 153, we require 3 bytes as under:
  - 1 = 1111 0001
  - 5 = 1111 0101
  - 3 = 1111 0011
  - = (11110001 11110101 11110011) EBCDIC

2's Complement Method

- Most common number code for storing **integer values** inside the computer.
- It can store signed as well as unsigned numbers.
- The signs of all bits except the left most bit are +ve and the sign of leftmost bit is –ve.

Drawback of 1's Complement Method

Suppose we have only 2 bits and we want to store –ve numbers also, we will have to take one bit for storing sign. 0 means number is +ve and 1 means number -ve.

Sign bit	Digit Bit		
0	0	=	+0
0	1	=	+1
1	0	=	-0
1	1	=	-1

The Range will be -1 to +1 as there are two representations of zero.

Drawback of 1's Complement Method

Suppose we have only 3 bits and we want to store –ve numbers also, we will have to take one bit for storing sign. 0 means number is +ve and 1 means number -ve.

Sign bit	Digit Bit	Digit bit		
0	0	0	=	+0
0	1	1	=	+3
1	0	0	=	-0
1	1	1	=	-3

The Range will be -3 to +3 as there are two representations of zero.

2's Complement Method

$-2^{(2)}$	$2^{(1)}$	$2^{(0)}$		
-4	2	1		
0	0	0	=	0
0	1	1	=	3
1	0	0	=	-4

2's Complement Method

- **The Range of Numbers**
  $-2^{(\text{no.of bits-1})}$ to $+2^{(\text{no.of bits-1})}$ -1
  - With 3 BITS : -4 to +3
  - With 4 BITS : -8 to +7
  - With 8 BITS : -128 to +127
  - With 16 BITS : -32768 to +32767

  Here, only left most bit will have –ve weight.

2's Complement Method

- **Convert decimal to 2's complemented form (using 8 bits)**
  - **107**
  - **-107**
- **Convert following 2's complement numbers into decimal. (Using 8 bits)**
  - **10001101**
  - **01111111**

Binary Arithmetic-Addition

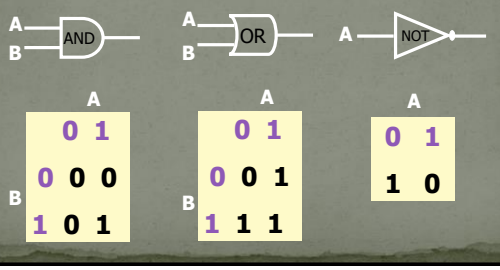
- **0 + 0 = 0**
- **0 + 1 = 1**
- **1 + 0 = 1**
- **1 + 1 = 0 with carry 1.**
  Therefore, 1 + 1 + 1 = 1 with carry 1

TRY THESE:

```
  10001001        10110 11
+10011011      +1110001
-----------    -----------
100100100      110011 00
```

Logic Circuits

Basic logic circuits: AND, OR, NOT

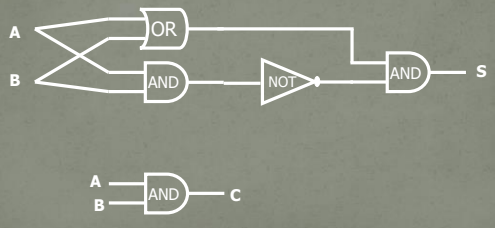


B \ A	0	1
0	0	0
1	0	1

B \ A	0	1
0	0	1
1	1	1

A	
0	1
1	0

Circuit for Addition



Binary Arithmetic-Multiplication

Binary multiplication

Equivalent decimal multiplication

```
0101        A
×1011       B
0101
0101   ——  Shift A left to multiply by B1 (= 2^1)
0000   ——  Since B2 = 0
0101
0110111
```

```
5
×11
55
```

- **Manual verification: 32 + 16 + 4 + 2 + 1 = 55**
- **Implemented in hardware using multiple shift-left and add steps**

Binary Arithmetic-Subtraction

- $(10101)_2 - (1010)_2 = (?)_2$

```
          C
1 0 1 0 1 ----->  0 1 0 1 0
- 0 1 0 1 0 ----->  + 0 1 0 1 0
                                C
                    1 0 1 0 0 ----> 0 1 0 1 1
```

Answer: $(1011)_2$ Here, C means Complement

Floating Point Representation

- Numbers having an Integer part and a Fractional part, is called a Real Number or Floating-Point Number.
- It can be either +ve or –ve.
- Every number can be represented in a Scientific Form i.e. $N = m\ r^{e}$
- N=no, m=mantissa, r=radix, e=exponent

Floating Point Representation

mantissa & exponent can be +ve or –ve

- 3.1415 = **.31415** x 10^{1}
- 3141.5 = .31415 x 10^{4}
- 0.0031415 = .31415 x $\mathbf{10^{-2}}$
- -31.415 = **-.31415** x $\mathbf{10^{2}}$

Note that

.31415 = $(3 \times 10^{-1}) + (1 \times 10^{-2}) + (4 \times 10^{-3}) + (1 \times 10^{-4}) + (5 \times 10^{-5})$

Binary Floating Point Representation

2-part number representation:

- **Mantissa: fractional part (in binary), with sign.**
- **Exponent: power of 2 (in binary), with sign.**

sign	mantissa	sign	exponent

Binary Floating Point Representation

using 10 bits mantissa & 6 bits for exponent, binary +1010.001 can be represented as

sign	mantissa	sign	exponent
0	101000100	0	00100

Binary Floating Point Representation

- IEEE 754 Floating Point Standards
  - Special codes for +/- infinity, NaN, +0, -0
  - SINGLE PRECISION (32 BITS)
  - DOUBLE PRECISION (64 BITS)

Binary Floating Point Representation

- IEEE 754 floating point standards
  - Single Precision (32 bits)
    - 1 bit to store sign (left most bit)
    - Exponent uses 8 bits "biased" representation
    - "biased" mean adding 127 to exponent
    - Exponent ranges from -126 to +127
    - 23 bits for mantissa

31	30.....23	22.....0
sign	exponent	mantissa

Example

- $(-118.625)_{10}$ = (?)$_{32\text{-BITS IEEE FORMAT}}$
- First we need to get the sign, the exponent and the fraction.
- The sign will be "1" as the whole number is a negative number.
- Now, we write the number (without the sign; i.e. unsigned, no two's complement) using binary notation.
- The result is 1110110.101.

Example

- **A normalized floating point number.**
- Next, let's move the radix point left, leaving only a 1 at its left:
- $1110110.101 = 1.110110101 \times 2^6$.
- The first 1 binary digit is dropped.
- The fraction is the part at the right of the radix point, filled with 0 on the right until we get all 23 bits.
- i.e. 11011010 00000000000000.

Example

- The exponent is 6, but we need to convert it to binary and bias it (so the most negative exponent is 0, and all exponents are non-negative binary numbers).
- For the 32-bit IEEE 754 format, the bias is 127
- So 6 + 127 = 133.
- In binary, this is written as 10000101.

Example

- The exponent is 6, but we need to convert it to binary and bias it (so the most negative exponent is 0, and all exponents are non-negative binary numbers).
- For the 32-bit IEEE 754 format, the bias is 127
- So 6 + 127 = 133.
- In binary, this is written as 10000101.

31	30 23	22 0
1	10000101	11011010100000000000000

Binary Floating Point Representation

- IEEE 754 Floating Point Standards
  - **Double Precision (64 Bits)**
    - **1 Bit To Store Sign (Left Most Bit)**
    - **Exponent Uses 11 Bits "Biased" Representation**
    - **"Biased" Mean Adding 1023 To Exponent**
    - **Exponent Ranges From -1022 To +1023**
    - **52 Bits For Mantissa**

63	62……52	51……0
sign	exponent	mantissa

What To Use? When?

- Trade-off between the range of numbers and accuracy.
- If we increase the exponent bits in 32-bit format, the range can be increased but accuracy of number goes down as size of mantissa will become smaller.
- Higher the no. of bits in mantissa, better will be precision.

What To Use? When?

- For increasing both precision and range, use double precision.

- in C/C++
- use **float** data type for single precision.
- use **double** data type for double precision.

Arithmetic on Real Nos.

- Arithmetic on real numbers are more complicated.
- Most ALU do only integer arithmetic
- Real (floating point) arithmetic is done
  - in software on some low-end processors.
  - in a floating-point unit (FPU) on most modern processors.
- Most processors today support single and double precision floating point arithmetic.