



Number Representation

Number System :: The Basics

- We are accustomed to using the so-called **decimal number system**
 - Ten digits :: 0,1,2,3,4,5,6,7,8,9
 - Every digit position has a weight which is a power of 10
 - **Base** or **radix** is 10

Example:

$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

$$250.67 = 2 \times 10^2 + 5 \times 10^1 + 0 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2}$$

Binary Number System

- Two digits:

- 0 and 1
- Every digit position has a weight which is a power of 2
- Base or radix is 2

- Example:

$$110 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$$

Positional Number Systems (General)

Decimal Numbers:

❖ 10 Symbols {0,1,2,3,4,5,6,7,8,9}, Base or Radix is 10

❖ $136.25 = 1 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$

Positional Number Systems (General)

Decimal Numbers:

- ❖ 10 Symbols {0,1,2,3,4,5,6,7,8,9}, Base or Radix is 10
- ❖ $136.25 = 1 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$

Binary Numbers:

- ❖ 2 Symbols {0,1}, Base or Radix is 2
- ❖ $101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$

Positional Number Systems (General)

Decimal Numbers:

- ❖ 10 Symbols {0,1,2,3,4,5,6,7,8,9}, Base or Radix is 10
- ❖ $136.25 = 1 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$

Binary Numbers:

- ❖ 2 Symbols {0,1}, Base or Radix is 2
- ❖ $101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$

Octal Numbers:

- ❖ 8 Symbols {0,1,2,3,4,5,6,7}, Base or Radix is 8
- ❖ $621.03 = 6 \times 8^2 + 2 \times 8^1 + 1 \times 8^0 + 0 \times 8^{-1} + 3 \times 8^{-2}$

Positional Number Systems (General)

Decimal Numbers:

- ❖ 10 Symbols {0,1,2,3,4,5,6,7,8,9}, Base or Radix is 10
- ❖ $136.25 = 1 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$

Binary Numbers:

- ❖ 2 Symbols {0,1}, Base or Radix is 2
- ❖ $101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$

Octal Numbers:

- ❖ 8 Symbols {0,1,2,3,4,5,6,7}, Base or Radix is 8
- ❖ $621.03 = 6 \times 8^2 + 2 \times 8^1 + 1 \times 8^0 + 0 \times 8^{-1} + 3 \times 8^{-2}$

Hexadecimal Numbers:

- ❖ 16 Symbols {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}, Base is 16
- ❖ $6AF.3C = 6 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 + 3 \times 16^{-1} + 12 \times 16^{-2}$

Binary-to-Decimal Conversion

- Each digit position of a binary number has a weight
 - Some power of 2
- A binary number:

$$B = b_{n-1} b_{n-2} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-m}$$

Corresponding value in decimal:

$$D = \sum_{i=-m}^{n-1} b_i 2^i$$

Examples

$$101011 \rightarrow 1x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 \\ = 43$$

$$(101011)_2 = (43)_{10}$$

$$.0101 \rightarrow 0x2^{-1} + 1x2^{-2} + 0x2^{-3} + 1x2^{-4} \\ = .3125$$

$$(.0101)_2 = (.3125)_{10}$$

$$101.11 \rightarrow 1x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2} \\ = 5.75$$

$$(101.11)_2 = (5.75)_{10}$$

Decimal to Binary: Integer Part

- Consider the integer and fractional parts separately.
- For the integer part:
 - Repeatedly divide the given number by 2, and go on accumulating the remainders, until the number becomes zero.
 - Arrange the remainders in reverse order.

| Base | Numb | Rem |
|------|------|-----|
|------|------|-----|

| | | |
|---|----|---|
| 2 | 89 | |
| 2 | 44 | 1 |
| 2 | 22 | 0 |
| 2 | 11 | 0 |
| 2 | 5 | 1 |
| 2 | 2 | 1 |
| 2 | 1 | 0 |
| | 0 | 1 |



$$(89)_{10} = (1011001)_2$$

Decimal to Binary: Integer Part

- Consider the integer and fractional parts separately.
- For the integer part:
 - Repeatedly divide the given number by 2, and go on accumulating the remainders, until the number becomes zero.
 - Arrange the remainders in reverse order.

| Base | Numb | Rem |
|------|------|-----|
|------|------|-----|

| | | |
|---|----|---|
| 2 | 89 | |
| 2 | 44 | 1 |
| 2 | 22 | 0 |
| 2 | 11 | 0 |
| 2 | 5 | 1 |
| 2 | 2 | 1 |
| 2 | 1 | 0 |
| | 0 | 1 |



$$(89)_{10} = (1011001)_2$$

| | | |
|---|----|---|
| 2 | 66 | |
| 2 | 33 | 0 |
| 2 | 16 | 1 |
| 2 | 8 | 0 |
| 2 | 4 | 0 |
| 2 | 2 | 0 |
| 2 | 1 | 0 |
| | 0 | 1 |

$$(66)_{10} = (1000010)_2$$

Decimal to Binary: Integer Part

- Consider the integer and fractional parts separately.
- For the integer part:
 - Repeatedly divide the given number by 2, and go on accumulating the remainders, until the number becomes zero.
 - Arrange the remainders in reverse order.

| Base | Numb | Rem |
|------|------|-----|
|------|------|-----|

| | | |
|---|----|---|
| 2 | 89 | |
| 2 | 44 | 1 |
| 2 | 22 | 0 |
| 2 | 11 | 0 |
| 2 | 5 | 1 |
| 2 | 2 | 1 |
| 2 | 1 | 0 |
| | 0 | 1 |



$$(89)_{10} = (1011001)_2$$

| | | |
|---|----|---|
| 2 | 66 | |
| 2 | 33 | 0 |
| 2 | 16 | 1 |
| 2 | 8 | 0 |
| 2 | 4 | 0 |
| 2 | 2 | 0 |
| 2 | 1 | 0 |
| | 0 | 1 |

$$(66)_{10} = (1000010)_2$$

| | | |
|---|-----|---|
| 2 | 239 | |
| 2 | 119 | 1 |
| 2 | 59 | 1 |
| 2 | 29 | 1 |
| 2 | 14 | 1 |
| 2 | 7 | 0 |
| 2 | 3 | 1 |
| 2 | 1 | 1 |
| | 0 | 1 |

$$(239)_{10} = (11101111)_2$$

Decimal to Binary: Fraction Part

- Repeatedly multiply the given fraction by 2.
 - Accumulate the integer part (0 or 1).
 - If the integer part is 1, chop it off.
- Arrange the integer parts in the order they are obtained.

Example: 0.634

$$.634 \times 2 = 1.268$$

$$.268 \times 2 = 0.536$$

$$.536 \times 2 = 1.072$$

$$.072 \times 2 = 0.144$$

$$.144 \times 2 = 0.288$$

:

:

$$(.634)_{10} = (.10100\dots)_2$$

Decimal to Binary: Fraction Part

- Repeatedly multiply the given fraction by 2.
 - Accumulate the integer part (0 or 1).
 - If the integer part is 1, chop it off.
- Arrange the integer parts in the order they are obtained.

Example: 0.634

$$.634 \times 2 = 1.268$$

$$.268 \times 2 = 0.536$$

$$.536 \times 2 = 1.072$$

$$.072 \times 2 = 0.144$$

$$.144 \times 2 = 0.288$$

:

:

$$(.634)_{10} = (.10100\dots)_2$$

Example: 0.0625

$$.0625 \times 2 = 0.125$$

$$.1250 \times 2 = 0.250$$

$$.2500 \times 2 = 0.500$$

$$.5000 \times 2 = 1.000$$

$$(.0625)_{10} = (.0001)_2$$

Decimal to Binary: Fraction Part

- Repeatedly multiply the given fraction by 2.
 - Accumulate the integer part (0 or 1).
 - If the integer part is 1, chop it off.
- Arrange the integer parts in the order they are obtained.

Example: 0.634

$$.634 \times 2 = 1.268$$

$$.268 \times 2 = 0.536$$

$$.536 \times 2 = 1.072$$

$$.072 \times 2 = 0.144$$

$$.144 \times 2 = 0.288$$

:

:

$$(.634)_{10} = (.10100\dots)_2$$

Example: 0.0625

$$.0625 \times 2 = 0.125$$

$$.1250 \times 2 = 0.250$$

$$.2500 \times 2 = 0.500$$

$$.5000 \times 2 = 1.000$$

$$(.0625)_{10} = (.0001)_2$$

$$(37)_{10} = (100101)_2$$

$$(.0625)_{10} = (.0001)_2$$

$$(37.0625)_{10} = (100101.0001)_2$$

Hexadecimal Number System

- A compact way of representing binary numbers
- 16 different symbols (radix = 16)

| | |
|----------|----------|
| 0 → 0000 | 8 → 1000 |
| 1 → 0001 | 9 → 1001 |
| 2 → 0010 | A → 1010 |
| 3 → 0011 | B → 1011 |
| 4 → 0100 | C → 1100 |
| 5 → 0101 | D → 1101 |
| 6 → 0110 | E → 1110 |
| 7 → 0111 | F → 1111 |

Binary-to-Hexadecimal Conversion

- For the integer part,
 - Scan the binary number from **right to left**
 - Translate each group of four bits into the corresponding hexadecimal digit
 - Add **leading** zeros if necessary
- For the fractional part,
 - Scan the binary number from **left to right**
 - Translate each group of four bits into the corresponding hexadecimal digit
 - Add **trailing** zeros if necessary

Example

1. $(\underline{1011} \ \underline{0100} \ \underline{0011})_2 = (B43)_{16}$
2. $(\underline{10} \ \underline{1010} \ \underline{0001})_2 = (2A1)_{16}$
3. $(\underline{.1000} \ \underline{010})_2 = (.84)_{16}$
4. $(\underline{101} \ . \ \underline{0101} \ \underline{111})_2 = (5.5E)_{16}$

Hexadecimal-to-Binary Conversion

- Translate every hexadecimal digit into its 4-bit binary equivalent

- Examples:

$$(3A5)_{16} = (0011\ 1010\ 0101)_2$$

$$(12.3D)_{16} = (0001\ 0010\ .\ 0011\ 1101)_2$$

$$(1.8)_{16} = (0001\ .\ 1000)_2$$

Unsigned Binary Numbers

- An n-bit binary number

$$B = b_{n-1}b_{n-2} \dots b_2b_1b_0$$

- 2^n distinct combinations are possible, 0 to 2^n-1 .
- For example, for $n = 3$, there are 8 distinct combinations
 - 000, 001, 010, 011, 100, 101, 110, 111
- Range of numbers that can be represented
 - $n=8 \rightarrow 0 \text{ to } 2^8-1 \text{ (255)}$
 - $n=16 \rightarrow 0 \text{ to } 2^{16}-1 \text{ (65535)}$
 - $n=32 \rightarrow 0 \text{ to } 2^{32}-1 \text{ (4294967295)}$

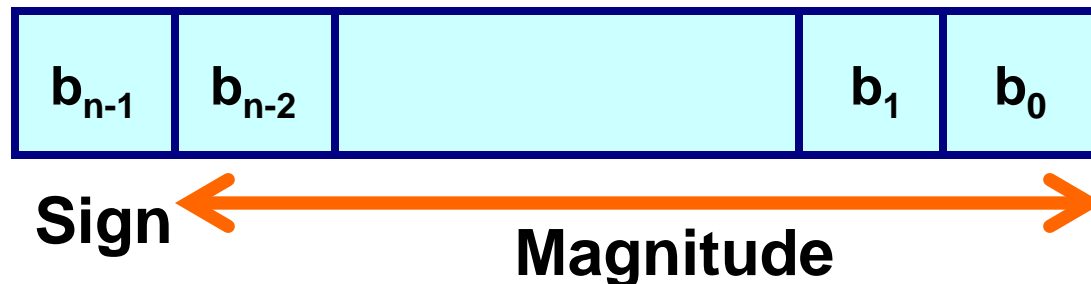


Signed Integer Representation

- Many of the numerical data items that are used in a program are signed (positive or negative)
 - Question:: How to represent sign?
- Three possible approaches:
 - Sign-magnitude representation
 - One's complement representation
 - Two's complement representation

Sign-magnitude Representation

- For an n-bit number representation
 - The most significant bit (MSB) indicates sign
 - 0 → positive
 - 1 → negative
 - The remaining n-1 bits represent magnitude



Contd.

- Range of numbers that can be represented:

Maximum $:: + (2^{n-1} - 1)$

Minimum $:: - (2^{n-1} - 1)$

- A problem:

Two different representations of zero

+0 \rightarrow 0 000....0

-0 \rightarrow 1 000....0

One's Complement Representation

- Basic idea:
 - Positive numbers are represented exactly as in sign-magnitude form
 - Negative numbers are represented in 1's complement form
- How to compute the 1's complement of a number?
 - Complement every bit of the number ($1 \rightarrow 0$ and $0 \rightarrow 1$)
 - MSB will indicate the sign of the number
 - 0 \rightarrow positive
 - 1 \rightarrow negative

Example :: $n=4$

0000 \rightarrow +0

0001 \rightarrow +1

0010 \rightarrow +2

0011 \rightarrow +3

0100 \rightarrow +4

0101 \rightarrow +5

0110 \rightarrow +6

0111 \rightarrow +7

1000 \rightarrow -7

1001 \rightarrow -6

1010 \rightarrow -5

1011 \rightarrow -4

1100 \rightarrow -3

1101 \rightarrow -2

1110 \rightarrow -1

1111 \rightarrow -0

To find the representation of, say, -4, first note that

$$+4 = 0100$$

$$-4 = 1\text{'s complement of } 0100 = 1011$$

Contd.

- Range of numbers that can be represented:

Maximum $:: + (2^{n-1} - 1)$

Minimum $:: - (2^{n-1} - 1)$

- A problem:

Two different representations of zero.

+0 \rightarrow 0 000....0

-0 \rightarrow 1 111....1

- Advantage of 1's complement representation

- ☐ Subtraction can be done using addition
- ☐ Leads to substantial saving in circuitry

Two's Complement Representation

- Basic idea:
 - Positive numbers are represented exactly as in sign-magnitude form
 - Negative numbers are represented in 2's complement form
- How to compute the 2's complement of a number?
 - Complement every bit of the number ($1 \rightarrow 0$ and $0 \rightarrow 1$), and then **add one** to the resulting number
 - MSB will indicate the sign of the number
 - 0 \rightarrow positive
 - 1 \rightarrow negative

Example : $n=4$

0000 \rightarrow +0

0001 \rightarrow +1

0010 \rightarrow +2

0011 \rightarrow +3

0100 \rightarrow +4

0101 \rightarrow +5

0110 \rightarrow +6

0111 \rightarrow +7

1000 \rightarrow -8

1001 \rightarrow -7

1010 \rightarrow -6

1011 \rightarrow -5

1100 \rightarrow -4

1101 \rightarrow -3

1110 \rightarrow -2

1111 \rightarrow -1

To find the representation of, say, -4, first note that

$$+4 = 0100$$

$$-4 = 2\text{'s complement of } 0100 = 1011 + 1 = 1100$$

Rule : Value = $-\text{msb} \cdot 2^{(n-1)} + [\text{unsigned value of rest}]$

Example: $0110 = 0 + 6 = 6$

$$1110 = -8 + 6 = -2$$

Contd.

- Range of numbers that can be represented:
 - Maximum $:: + (2^{n-1} - 1)$
 - Minimum $:: - 2^{n-1}$
- Advantage:
 - Unique representation of zero
 - Subtraction can be done using addition
 - Leads to substantial saving in circuitry
- Almost all computers today use the 2's complement representation for storing negative numbers

Contd.

■ In C

□ short int

■ 16 bits → $+(2^{15}-1)$ to -2^{15}

□ int or long int

■ 32 bits → $+(2^{31}-1)$ to -2^{31}

□ long long int

■ 64 bits → $+(2^{63}-1)$ to -2^{63}

Adding Binary Numbers

■ Basic Rules:

- $0+0=0$
- $0+1=1$
- $1+0=1$
- $1+1=0$ (carry 1)

■ Example:

```
01101001
00110100
-----
10011101
```

Subtraction Using Addition :: 1's Complement

■ How to compute $A - B$?

- Compute the 1's complement of B (say, B_1).
- Compute $R = A + B_1$
- If the carry obtained after addition is '1'
 - Add the carry back to R (called *end-around carry*)
 - That is, $R = R + 1$
 - The result is a positive number

Else

- The result is negative, and is in 1's complement form

Example 1 :: 6 – 2

1's complement of 2 = 1101

| | | | |
|----|----|---------------|----------------|
| 6 | :: | 0110 | A |
| -2 | :: | 1101 | B ₁ |
| | | <u>1 0011</u> | R |
| | | 1 | |
| | | <u>0100</u> | |

→ +4

End-around carry

Assume 4-bit representations

Since there is a carry, it is added back to the result

The result is positive

Example 2 :: 3 – 5

1's complement of 5 = 1010

$$\begin{array}{rcl} 3 & :: & 0011 & \text{A} \\ -5 & :: & \underline{1010} & \text{B}_1 \\ & & 1101 & \text{R} \\ & \downarrow & & \\ & -2 & & \end{array}$$

Assume 4-bit representations

Since there is no carry, the result is negative

1101 is the 1's complement of 0010, that is, it represents –2

Subtraction Using Addition :: 2's Complement

■ How to compute $A - B$?

- Compute the 2's complement of B (say, B_2)
- Compute $R = A + B_2$
- If the carry obtained after addition is '1'
 - Ignore the carry
 - The result is a positive number

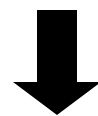
Else

- The result is negative, and is in 2's complement form

Example 1 :: 6 – 2

2's complement of 2 = 1101 + 1 = 1110

$$\begin{array}{rcl} 6 & :: & 0110 \quad \text{A} \\ -2 & :: & \underline{1110} \quad \text{B}_2 \\ & & 1 \quad 0100 \quad \text{R} \end{array}$$



+4

Assume 4-bit
representations

Presence of carry indicates
that the result is positive

No need to add the end-
around carry like in 1's
complement

Example 2 :: 3 – 5

2's complement of 5 = $1010 + 1 = 1011$

$$\begin{array}{rcl} 3 & :: & 0011 \quad \mathbf{A} \\ -5 & :: & \underline{1011} \quad \mathbf{B_2} \\ & & 1110 \quad \mathbf{R} \\ & & \downarrow \\ & & -2 \end{array}$$

Assume 4-bit representations

Since there is no carry, the result is negative

1110 is the 2's complement of 0010, that is, it represents –2

2's complement arithmetic: More Examples

- Example 1: $18 - 11 = ?$
- 18 is represented as 00010010
- 11 is represented as 00001011
 - 1's complement of 11 is 11110100
 - 2's complement of 11 is 11110101
- Add 18 to 2's complement of 11

```
00010010
+ 11110101
-----
00000111 (with a carry of 1
           which is ignored)
```

00000111 is 7

- Example 2: $7 - 9 = ?$
- 7 is represented as 00000111
- 9 is represented as 00001001
 - 1's complement of 9 is 11110110
 - 2's complement of 9 is 11110111
 - Add 7 to 2's complement of 9

```
00000111
+ 11110111
-----
11111110 (with a carry of 0
           which is ignored)
```

11111110 is -2

Overflow/Underflow:

Adding two +ve (-ve) numbers should not produce a -ve (+ve) number. If it does, overflow (underflow) occurs

Overflow/Underflow:

Adding two +ve (-ve) numbers should not produce a -ve (+ve) number. If it does, overflow (underflow) occurs

Another equivalent condition : carry in and carry out from Most Significant Bit (MSB) differ.

Overflow/Underflow:

Adding two +ve (-ve) numbers should not produce a -ve (+ve) number. If it does, overflow (underflow) occurs

Another equivalent condition : carry in and carry out from Most Significant Bit (MSB) differ.

(64) 01000000

(4) 00000100

(68) 01000100

carry (out)(in)

0 0

Overflow/Underflow:

Adding two +ve (-ve) numbers should not produce a -ve (+ve) number. If it does, overflow (underflow) occurs

Another equivalent condition : carry in and carry out from Most Significant Bit (MSB) differ.

```
(64) 01000000
( 4)  00000100
-----
(68) 01000100
```

carry (out)(in)
0 0

```
(64) 01000000
(96) 01100000
-----
(-96) 10100000
```

carry out in
0 1

differ:
overflow

Floating-point Numbers

- The representations discussed so far applies only to integers
 - Cannot represent numbers with fractional parts
- We can assume a decimal point before a signed number
 - In that case, pure fractions (without integer parts) can be represented
- We can also assume the decimal point somewhere in between
 - This lacks flexibility
 - Very large and very small numbers cannot be represented

Representation of Floating-Point Numbers

- A floating-point number F is represented by a doublet $\langle M, E \rangle$:

$$F = M \times B^E$$

- $B \rightarrow$ exponent base (usually 2)
 - $M \rightarrow$ mantissa
 - $E \rightarrow$ exponent
- M is usually represented in 2's complement form, with an implied binary point before it
- For example,
 - In decimal, 0.235×10^6
 - In binary, 0.101011×2^{0110}

Example :: 32-bit representation



- M represents a 2's complement fraction

$$1 > M > -1$$

- E represents the exponent (in 2's complement form)

$$127 > E > -128$$

- Points to note:

- The number of **significant digits** depends on the number of bits in M

- 6 significant digits for 24-bit mantissa

- The **range** of the number depends on the number of bits in E

- 10^{38} to 10^{-38} for 8-bit exponent.



A Warning

- The representation for floating-point numbers as shown is just for illustration
- The actual representation is a little more complex
- Example: IEEE 754 Floating Point format

IEEE 754 Floating-Point Format (Single Precision)

| S | E (Exponent) | M (Mantissa) |
|------|--------------|--------------|
| (31) | (30 ... 23) | (22 ... 0) |

S: Sign (0 is +ve, 1 is –ve)

E: Exponent (More bits gives a higher range)

M: Mantissa (More bits means higher precision)

[8 bytes are used for double precision]

Value of a Normal Number:

$$(-1)^S \times (1.0 + 0.M) \times 2^{(E - 127)}$$

An example

| S (31) | E (Exponent) (30 ... 23) | M (Mantissa) (22 ... 0) |
|-------------------------|---|--|
|-------------------------|---|--|

| | | |
|----------|-----------------|---------------------------------|
| 1 | 10001100 | 110110000000000000000000 |
|----------|-----------------|---------------------------------|

Value of a Normal Number:

$$= (-1)^S \times (1.0 + 0.M) \times 2^{(E - 127)}$$

$$= (-1)^1 \times (1.0 + 0.1101100) \times 2^{(10001100 - 1111111)}$$

$$= -1.1101100 \times 2^{1101} = -11101100000000$$

$$= -15104.0 \text{ (in decimal)}$$

Representing 0.3

| S (31) | E (Exponent) (30 ... 23) | M (Mantissa) (22 ... 0) |
|-----------|-----------------------------|----------------------------|
|-----------|-----------------------------|----------------------------|

0.3 (decimal)

= 0.0100100100100100100100100100...

= 1.00100100100100100100100100100... $\times 2^{-2}$

= 1.00100100100100100100100100100... $\times 2^{125-127}$

= $(-1)^S \times (1.0 + 0.M) \times 2^{(E-127)}$

| | | |
|----------|-----------------|-----------------------------------|
| 0 | 01111101 | 00100100100100100100100100 |
|----------|-----------------|-----------------------------------|

What are the largest and smallest numbers that can be represented in this scheme?

Representing 0

| S (31) | E (Exponent) (30 ... 23) | M (Mantissa) (22 ... 0) |
|-------------------------|---|--|
| 0 | 00000000 | 000000000000000000000000 |
| 1 | 00000000 | 000000000000000000000000 |

Representing Inf (∞)

| | | |
|----------|-----------------|---------------------------------|
| 0 | 11111111 | 000000000000000000000000 |
| 1 | 11111111 | 000000000000000000000000 |

Representing NaN (Not a Number)

| | | |
|----------|-----------------|-----------------|
| 0 | 11111111 | Non zero |
| 1 | 11111111 | Non zero |

Representation of Characters

- Many applications have to deal with non-numerical data.
 - Characters and strings
 - There must be a standard mechanism to represent alphanumeric and other characters in memory
- Three standards in use:
 - Extended Binary Coded Decimal Interchange Code (EBCDIC)
 - Used in older IBM machines
 - American Standard Code for Information Interchange (ASCII)
 - Most widely used today
 - UNICODE
 - Used to represent all international characters.
 - Used by Java

ASCII Code

- Each individual character is numerically encoded into a unique 7-bit binary code
 - A total of 2^7 or 128 different characters
 - A character is normally encoded in a byte (8 bits), with the MSB not been used.
- The binary encoding of the characters follow a regular ordering
 - Digits are ordered consecutively in their proper numerical sequence (0 to 9)
 - Letters (uppercase and lowercase) are arranged consecutively in their proper alphabetic order

Some Common ASCII Codes

'A' :: 41 (H) 65 (D)

'B' :: 42 (H) 66 (D)

.....

'Z' :: 5A (H) 90 (D)

'a' :: 61 (H) 97 (D)

'b' :: 62 (H) 98 (D)

.....

'z' :: 7A (H) 122 (D)

'0' :: 30 (H) 48 (D)

'1' :: 31 (H) 49 (D)

.....

'9' :: 39 (H) 57 (D)

'(' :: 28 (H) 40 (D)

'+' :: 2B (H) 43 (D)

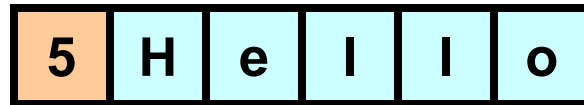
'?' :: 3F (H) 63 (D)

'\n' :: 0A (H) 10 (D)

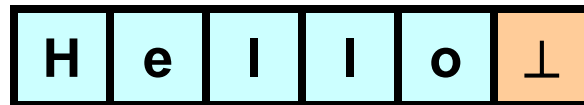
'\0' :: 00 (H) 00 (D)

Character Strings

- Two ways of representing a sequence of characters in memory



- The first location contains the number of characters in the string, followed by the actual characters



- The characters follow one another, and is terminated by a special delimiter

String Representation in C

- In C, the second approach is used
 - The `'\0'` character is used as the string delimiter

- Example:

"Hello"



- A null string "" occupies one byte in memory.
 - Only the `'\0'` character