COMPUTER ORGANIZATION AND ARCHITECTURE

Data Representation

Dr. Bore Gowda S B
Additional Professor
Dept. of ECE
MIT, Manipal

Introduction

- A *bit* is the most basic unit of information in a computer.
  - It is a state of “on” or “off” in a digital circuit.
  - Sometimes these states are “high” or “low” voltage instead of “on” or “off..”
- A *byte* is a group of eight bits.
  - A byte is the smallest possible *addressable* unit of computer storage.
  - The term, “addressable,” means that a particular byte can be retrieved according to its location in memory.

Positional Numbering Systems

- Bytes store numbers using the position of each bit to represent a power of 2.
  - The binary system is also called the base-2 system.
  - Our decimal system is the base-10 system. It uses powers of 10 for each position in a number.
  - Any integer quantity can be represented exactly using any **base (or *radix*).**

Positional Numbering Systems

- The decimal number 947 in powers of 10 is:

$$9 \times 10^{2} + 4 \times 10^{1} + 7 \times 10^{0}$$

- The decimal number 5836.47 in powers of 10 is:

$$5 \times 10^{3} + 8 \times 10^{2} + 3 \times 10^{1} + 6 \times 10^{0} + 4 \times 10^{-1} + 7 \times 10^{-2}$$

Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 16 + 8 + 0 + 0 + 1 = 25$$

- When the radix of a number is something other than 10, the base is denoted by a subscript.
  - Sometimes, the subscript 10 is added for emphasis:

$$11001_2 = 25_{10}$$

Converting Between Bases

- **Converting 190 to base 2**

$190_{10} = 10111110_2$

2	190	
2	95	0
2	47	1
2	23	1
2	11	1
2	5	1
2	2	1
2	1	0
	0	1

Converting Between Bases

- **Converting 0.8125 to binary . . .**
  - You are finished when the product is zero, or until you have reached the desired number of binary places.
  - Our result, reading from top to bottom is:

    $0.8125_{10} = 0.1101_2$
  - This method also works with any base. Just use the target radix as the multiplier.

```
  .8125
 ×   2
------
1.6250

  .6250
 ×   2
------
1.2500

  .2500
 ×   2
------
0.5000

  .5000
 ×   2
------
1.0000
```

Positional Numbering Systems

Decimal	Binary	Hexadecimal	Octal
0	00000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Numbering Systems

Numbers we can represent using binary representations

- **Positive numbers**
  - Unsigned binary
- **Negative numbers**
  - Two's complement
  - Sign/magnitude numbers

What about **fractions**?

Numbering Systems

Two common notations:

- **Fixed-point:** binary point fixed
- **Floating-point:** binary point floats to the right of the most significant 1

Numbering Systems

- 6.75 using 4 integer bits and 4 fraction bits:

  01101100

  0110.1100

  $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

  - Binary point is implied
  - The number of integer and fraction bits must be agreed upon beforehand

- Represent 7.5_{10} using 4 integer bits and 4 fraction bits.

01111000

Numbering Systems

Signed Integer Representation

- To represent signed integers, computer systems allocate the high-order bit to indicate the sign of a number.
  - The high-order bit is the leftmost bit. It is also called the most significant bit.
  - 0 is used to indicate a positive number;
  - 1 indicates a negative number.
- The remaining bits contain the value of the number (but this can be interpreted different ways)

Signed Integer Representation

- There are three ways in which signed binary integers may be expressed:
  - Signed magnitude
  - One's complement
  - Two's complement
- In an 8-bit word, *signed magnitude* representation places the absolute value of the number in the 7 bits to the right of the sign bit.

Signed Integer Representation

- For example, in 8-bit signed magnitude representation:

  +3 is: `00000011`

  - 3 is: `10000011`

- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
  - Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.

Signed Integer Representation

- Binary addition is as easy as it gets. You need to know only four rules:

```
0 + 0 =  0      0 + 1 =  1
1 + 0 =  1      1 + 1 = 10
```

- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.

- **Example:**
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- First, convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits.

```
0   1001011
0 + 0101110
```

Signed Integer Representation

Disadvantage of signed magnitude representation

- Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware.
- Another disadvantage of signed magnitude is that it allows two different representations for zero: positive zero and negative zero.
- For these reasons (among others) computers systems employ *complement systems* for numeric value representation

Signed Integer Representation

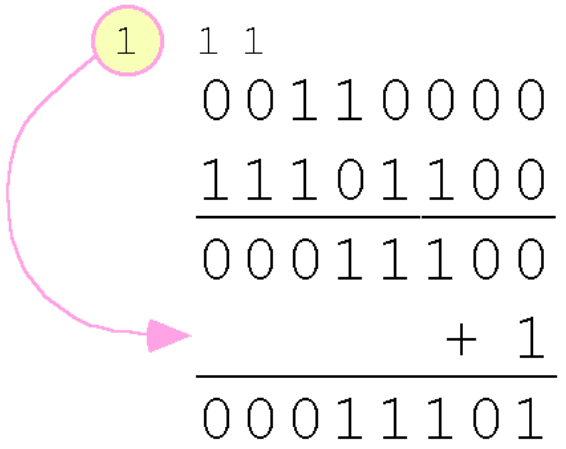
- In complement systems, negative values are represented by some difference between a number and its base.
- The *diminished radix complement* of a non-zero number *N* in base *r* with *d* digits is $(r^d - 1) - N$
- In the binary system, this gives us *one's complement*. It amounts to little more than flipping the bits of a binary number.

Signed Integer Representation

- For example, using 8-bit one's complement representation:
  + 3 is: `00000011`
  - 3 is: `11111100`
- In one's complement representation, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
- Complement systems are useful because they eliminate the need for subtraction. The difference of two values is found by adding the minuend to the complement of the subtrahend.

Signed Integer Representation

- With one's complement addition, the carry bit is "carried around" and added to the sum.
  - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19



We note that 19 in binary is `00010011`,
so -19 in one's complement is: `11101100`.

Signed Integer Representation

- Although the “end carry around” adds some complexity, one’s complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.
- Two’s complement solves this problem.
- Two’s complement is the radix complement of the binary numbering system; the *radix complement* of a non-zero number N in base r with d digits is $r^d - N$.

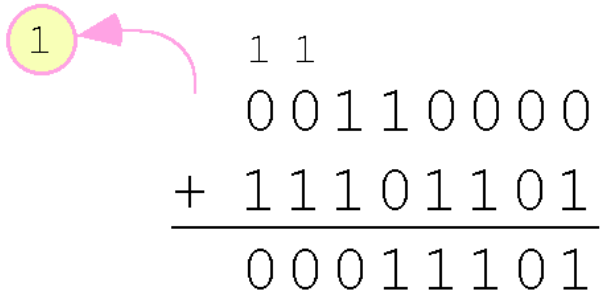
Signed Integer Representation

- To express a value in two's complement representation:
  - If the number is positive, just convert it to binary and you're done.
  - If the number is negative, find the one's complement of the number and then add 1.
- Example:
  - In 8-bit binary, 3 is: `00000011`
  - -3 using one's complement representation is: `11111100`
  - Adding 1 gives us -3 in two's complement form: `11111101`.

Signed Integer Representation

- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.

  – Example: Using two's complement binary arithmetic, find the sum of 48 and - 19.



We note that 19 in binary is: 00010011
so -19 using one's complement is: 11101100
and -19 using two's complement is: 11101101

Signed Integer Representation

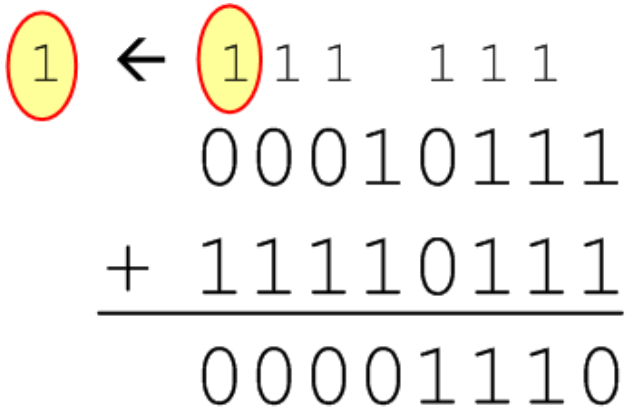
- When we use any finite number of bits to represent a number, we always run the risk of the result of our calculations becoming too large or too small to be stored in the computer.
- While we can't always prevent overflow, we can always *detect* overflow.
- In complement arithmetic, an overflow condition is easy to detect.
- **Example:**
  - Using two's complement binary arithmetic, find the sum of 107 and 46.
  - We see that the nonzero carry from the seventh bit *overflows* into the sign bit, giving us the erroneous result: 107 + 46 = -103.

```
  1 1   1 1 1
  01101011
+ 00101110
----------
  10011001
```

But overflow into the sign bit does not always mean that we have an error.

Signed Integer Representation

- Example:
  - Using two's complement binary arithmetic, find the sum of 23 and -9.
  - We see that there is carry into the sign bit and carry out. The final result is correct: 23 + (-9) = 14.



Rule for detecting signed two's complement overflow: When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred. If the carry into the sign bit equals the carry out of the sign bit, no overflow has occurred.

Floating-Point Representation

- The signed magnitude, one's complement, and two's complement representation that we have just presented deal with signed integer values only.
- **Without modification**, these formats are not useful in scientific or business applications that deal with real number values.
- Floating-point representation solves this problem.
- The Floating point representation is a way to the encode numbers in a format that can handle very large and very small values.
- It is based on scientific notation where numbers are represented as a fraction and an exponent.
- In computing, this representation allows for trade-off between *range* and *precision*.

Floating-Point Representation

- **Need for Floating Point Representation**
- The Floating point representation is crucial because:

  - ✓ **Range:** It can represent a wide range of values from the very large to very small numbers.
  - ✓ **Precision:** It provides a good balance between the precision and range, making it suitable for the scientific computations, graphics and other applications where exact values and wide ranges are necessary.
  - ✓ **Flexibility:** It adapts to different scales of numbers allowing for the efficient storage and computation of real numbers in the computer systems.

Floating-Point Representation

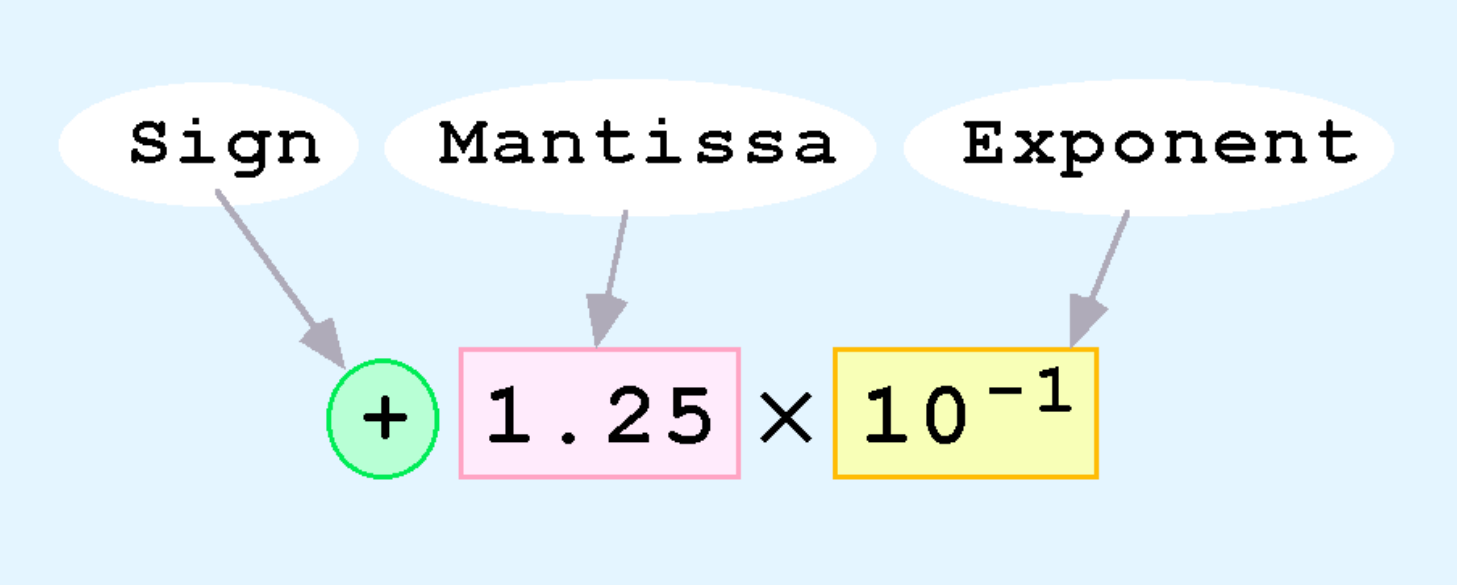
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

    $0.125 = 1.25 \times 10^{-1}$

    $5{,}000{,}000 = 5.0 \times 10^{6}$

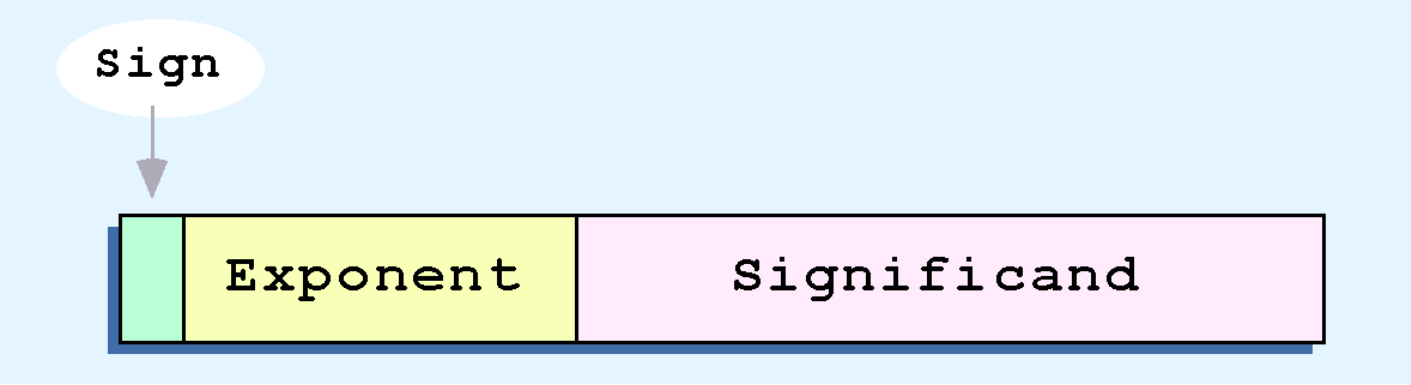
Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



Floating-Point Representation

- Computer representation of a floating-point number consists of three fixed-size fields:

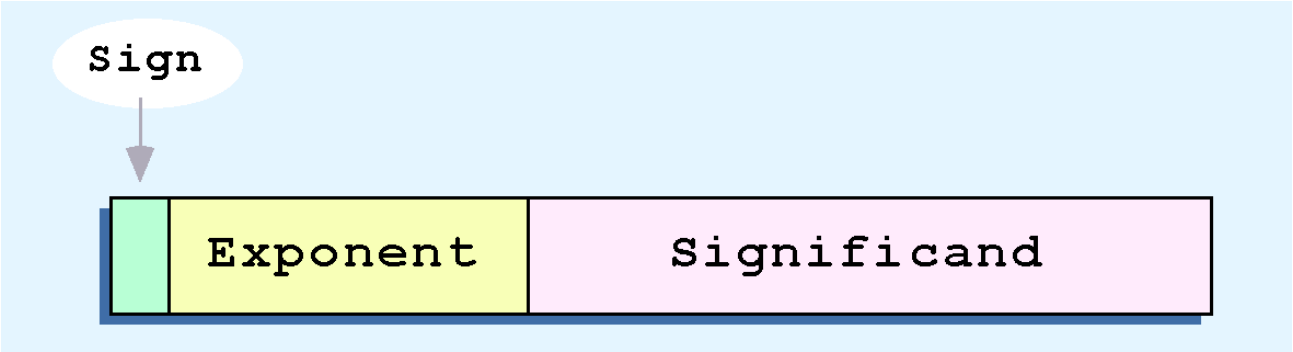


  - This is the standard arrangement of these fields

Note: Many people use "significand" and "mantissa" terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

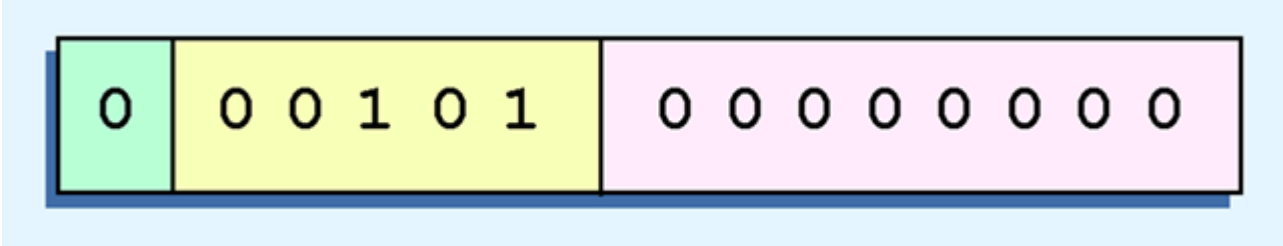
Floating-Point Representation

- a hypothetical "Simple Model" to explain the concepts
- In this model:
  - A floating-point number is 14 bits in length
  - The exponent field is 5 bits
  - The significand field is 8 bits



Floating-Point Representation

- **Example:**
  - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5. So in (binary) scientific notation $32 = 1.0 \times 2^5$
- Using this information, we put 101 (= 5_{10}) in the exponent field and 0 in the significand as shown.



IEEE Standard 754 Floating Point Numbers

- The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).
- The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability.
- IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.

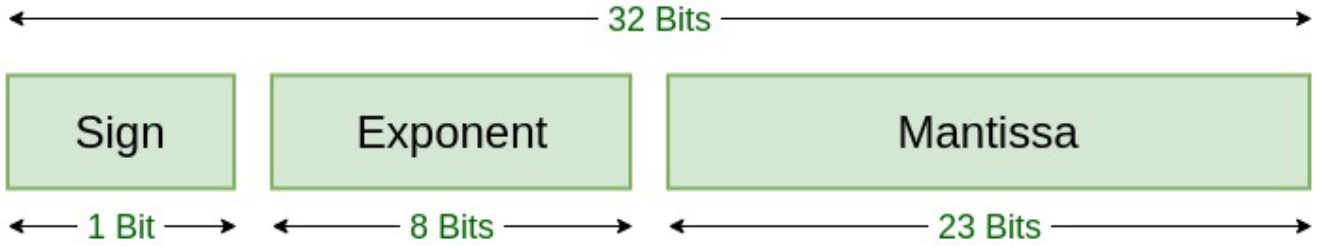
IEEE Standard 754 Floating Point Numbers

- **IEEE 754 has 3 basic components:**
- **The Sign of Mantissa –**
  - ✓ 0 represents a positive number while 1 represents a negative number.
- The Biased exponent –
  - ✓ The exponent field needs to represent both positive and negative exponents.
  - ✓ A bias is added to the actual exponent in order to get the stored exponent.
- The Normalised Mantissa –
  - ✓ The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits.
  - ✓ Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal..

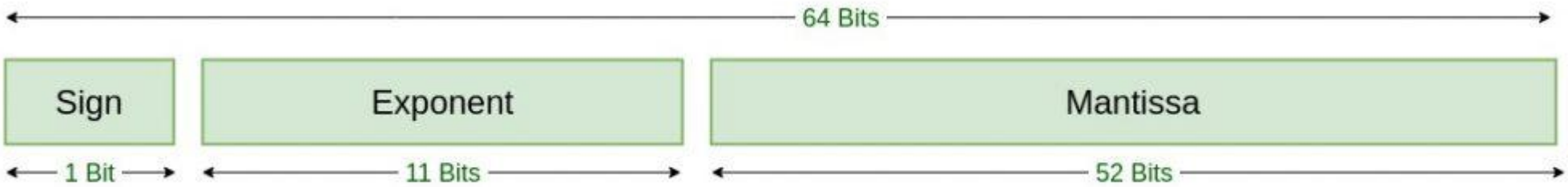
IEEE Standard 754 Floating Point Numbers

- **Types**

❑ 32-bit Single-Precision Floating-Point Numbers



❑ 64-bit Double-Precision Floating-Point Numbers



TYPES	SIGN	BIASED EXPONENT	NORMALISED MANTISA	BIAS
Single precision	1(31st bit)	8(30-23)	23(22-0)	127
Double precision	1(63rd bit)	11(62-52)	52(51-0)	1023

IEEE Standard 754 Floating Point Numbers

- Express the number $(85.125)_{10}$ in IEEE 754 single and double precision methods
  - 85 = 1010101
  - 0.125 = 001
  - 85.125 = 1010101.001
  - Normalized number =1.010101001 x 2^6
  - sign = 0

- **Single precision:**
  - biased exponent 127+6=133
  - 133 = 10000101
  - Normalised mantisa = 010101001
  - we will add 0's to complete the 23 bits

  - The IEEE 754 Single precision is: = **0 10000101 01010100100000000000000**
  - This can be written in hexadecimal form **42AA4000**

IEEE Standard 754 Floating Point Numbers

- **Express the number $(85.125)_{10}$ in IEEE 754 single and double precision methods**
  - 85 = 1010101
  - 0.125 = 001
  - 85.125 = 1010101.001
  - Normalized number =1.010101001×2^6
  - sign = 0

- **Double precision**
  - biased exponent 1023+6=1029
  - 1029 = 10000000101
  - Normalised mantisa = 010101001
  - we will add 0's to complete the 52 bits

  - The IEEE 754 Double precision is:
  - = **0 10000000101 0101010010000000000000000000000000000000000000000000**
  - This can be written in hexadecimal form 405548000000000

IEEE Standard 754 Floating Point Numbers

- **Example 1: Suppose that IEEE-754 32-bit floating-point representation pattern is**

0 10000000 110 0000 0000 0000 0000 0000.

Determine the number

```
Sign bit S = 0 ⇒ positive number
E = 1000 0000B = 128D (in normalized form)
Fraction is 1.11B (with an implicit leading 1) = 1 + 1×2^-1 + 1×2^-2 = 1.75D
The number is +1.75 × 2^(128-127) = +3.5D
```

- **Example 2: Suppose that IEEE-754 32-bit floating-point representation pattern is**

1 01111110 100 0000 0000 0000 0000 0000.

```
Sign bit S = 1 ⇒ negative number
E = 0111 1110B = 126D (in normalized form)
Fraction is 1.1B  (with an implicit leading 1) = 1 + 2^-1 = 1.5D
The number is -1.5 × 2^(126-127) = -0.75D
```

Numbering Systems