

# Chapter 2

## Data Representation

LO-1:

- **Represent data** in various **formats**, and **convert** between decimal, binary, octal, hexadecimal, sign-magnitude, and ones and twos-complement.
- Perform some **basic** binary **arithmetic**, multiplication and division.

>>> **Test-1**

# Chapter 2 Objectives

---

- Understand the *fundamentals* of numerical **data representation** and *manipulation* in computer systems.
- Master the skill of *converting* between different numeric-radix systems.
- Understand how *errors* can occur in computations because of overflow and truncation.
- Understand the *fundamental* concepts of floating-point representation.
- Gain familiarity with the most *popular character* codes.

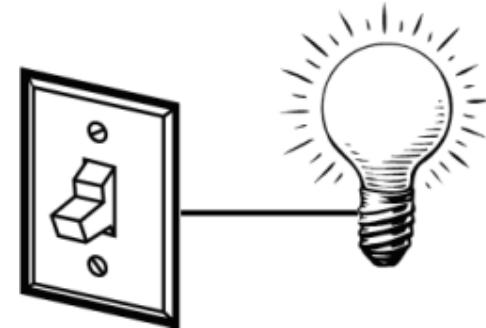
# Outline

---

## Data

- 1- numeric
  - i- integer (unsigned, signed)
  - ii- fraction
- 2- non-numeric (text, symbols, etc.)

- Converting between different **numeric-radix systems**
  - Binary addition and subtraction
  - Two's complement representation
  - Floating-point representation
  - Characters in computer
-



## 2.1 Introduction

- A *bit* is the most basic unit of information in a computer.
  - It is a state of either “on” or “off”, “high” or “low” voltage in a digital circuit.
  - In a computer, a bit could be either “1” or “0”.
- A *byte* is a group of 8 bits.
  - a byte is the *smallest* possible unit of storage in computer systems
- A group of 4 bits is called a *nibble*.
  - A byte consists of 2 nibbles:
    - The “*high-order*” nibble and the “*low-order*” nibble.

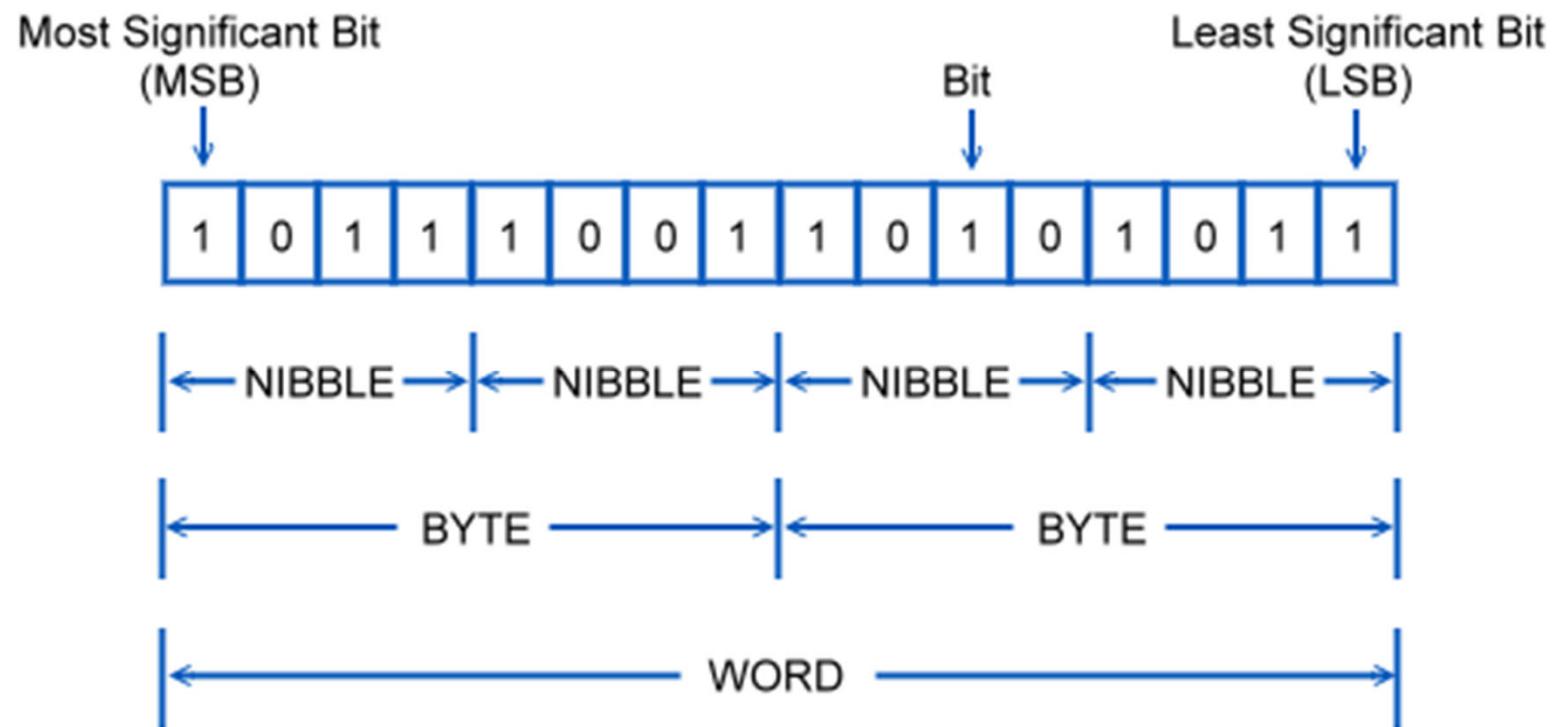
## 2.1 Introduction

---

- A *word* is a contiguous group of bytes.
  - Words can be any number of bits or bytes.
  - According to different computer systems, the size of word could be *2 bytes* (16 bits), *4 bytes* (32 bits), or *8 bytes* (64 bits).

## 2.1 Introduction

---



# Byte or Word Addressable

---

- A computer allows either a byte or a word to be addressable
  - Addressable: a particular unit of storage can be retrieved by CPU, according to its location in memory.
  - A byte is the *smallest* possible addressable unit of storage in a *byte-addressable* computer
  - A word is the smallest addressable unit of storage in a *word-addressable* computer

# 2.2 Positional Numbering Systems

---

- Bytes store numbers use the position of each bit to represent a *power of 2 (radix of 2)*.
  - The binary system is also called the *base-2* system.
  - Our decimal system is the *base-10* system, because the position of each number represents a *power of 10 (radix of 10)*.
- When the *radix* of a number is *other than 10*, the base is denoted as a *subscript*.
  - Sometimes, the subscript 10 is added for emphasis

## 2.2 Positional Numbering Systems

---

- Let's first look at numbers in base-10 number system

- The decimal number  $947_{10}$  (base-10) is:

$$947_{10} = 9 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$$

- The decimal number  $5836.47_{10}$  (base-10) is:

$$\begin{aligned}5836.47_{10} &= 5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \\&\quad \times 10^0 + 4 \times 10^{-1} + 7 \times 10^{-2}\end{aligned}$$

## 2.2 Positional Numbering Systems

---

- Then, look at numbers in base-2 number system
- The binary number  $11001_2$  (**base-2**) is:

$$\begin{aligned} & 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ = & 16 + 8 + 0 + 0 + 1 = 25_{10} \end{aligned}$$

- $11001_2 = 25_{10}$

# Practice

---

$(01111101)_2 = ?$

$(123)_8 = ?$

$(123)_3 = ?$

A digit in a numeral that is greater than or equal to the base of the number is not allowed.

Any problem?

# Practice

---

- $(01111101)_2 = 64+32+16+8+4+1=125$
  
- $(123)_8 = 1\times8^2+2\times8+3\times8^0=83$
  
- $(123)_3 \rightarrow 130_3 \rightarrow 200_3 = 2\times3^2 = 18$ 
  - $123_3 = 1\times3^2+2\times3^1+3\times3^0=9+6+3=18$

# Why Do I Have To Learn And Convert Binary Numbers?

---

- Because binary numbers are **the basis** for all **data representation** in computer systems
  - It is important that you become proficient with binary system.
- Your knowledge of the binary numbering system will help you **understand the operations** of all computer components
  - As well as the **instruction set** of computer.

## 2.3 Converting Between Bases

---

- How can any integer (base-10 number) be converted into any radix system?
- There are two methods of conversion:
  - The ***Subtraction (-)*** method, and
  - The ***Division (/) remainder*** method.
- Let's use the subtraction method to convert  $190_{10}$  to  $(x)_3$ .

# 2.3 Converting Between Bases: ***Subtraction***

---

## □ Converting the decimal number

$190_{10}$  to  $X_3$ .

- Let's try different integers  
 $i=3^{\textcolor{red}{n}}$
- $3^{\textcolor{red}{5}} = 243 > 190$
- The largest power of 3 that we need is  $3^{\textcolor{red}{4}} = 81 < 190$ , and  $81 \times 2 = 162$ .
- Write down the 2 and subtract 162 from 190, get the remainder 28.

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \end{array} = 3^4 \times \boxed{2}$$

# 2.3 Converting Between Bases: ***Subtraction***

## □ Converting $190_{10}$ to $X_3$ ...

- The next power of 3 is  $3^3 = 27 < 28$ , so we subtract 27 and write down the numeral 1 as our result.
- The next power of 3 is  $3^2 = 9 > 1$ , too large, so we skip  $3^2$

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \\ - 27 \\ \hline 1 \\ - 0 \\ \hline 1 \end{array} = 3^4 \times 2$$
$$= 3^3 \times 1$$
$$= 3^2 \times 0$$

# 2.3 Converting Between Bases: ***Subtraction***

## □ Converting $190_{10}$ to $X_3$ ...

- $3^1 = 3 > 1$ . too large, so we skip  $3^1$
- The last power of 3 =  $3^0 = 1$ , is our last choice, and it gives us a difference of 0.
- Our result, reading from top to bottom is:

$$190_{10} = 21001_3$$

$$\begin{array}{r} 190 \\ - 162 \\ \hline 28 \\ - 27 \\ \hline 1 \\ - 0 \\ \hline 1 \\ - 0 \\ \hline 1 \\ - 1 \\ \hline 0 \end{array} = 3^4 \times 2$$

A vertical red rectangle highlights the second column of digits (2, 1, 0, 0, 1) from top to bottom. A vertical yellow rectangle highlights the third column of digits (1, 0, 0, 1, 0) from top to bottom. A red arrow points downwards from the bottom of the yellow rectangle towards the final result of 0.

The subtraction method is more **intuitive**, but cumbersome.

## 2.3 Converting Between Bases: ***Division***

---

- Let's try another method of conversion:  
**Division.**
- The idea is that:
  - Successive division by a base is equivalent to successive subtraction by powers of the base!
- Let's use the division method to convert  $190_{10}$  to  $X_3$ , again.

## 2.3 Converting Between Bases

---

### □ Converting $190_{10}$ to base 3...

- First we take the number that we wish to convert and divide it by the radix in which we would like to convert to.
- In this case,  $190/3 \rightarrow$  the quotient is 63, and the remainder is 1.

$$\begin{array}{r} 3 \quad | \quad 190 \quad \boxed{1} \\ \hline \quad \quad \quad \boxed{63} \end{array}$$

## 2.3 Converting Between Bases

---

### □ Converting $190_{10}$ to base 3...

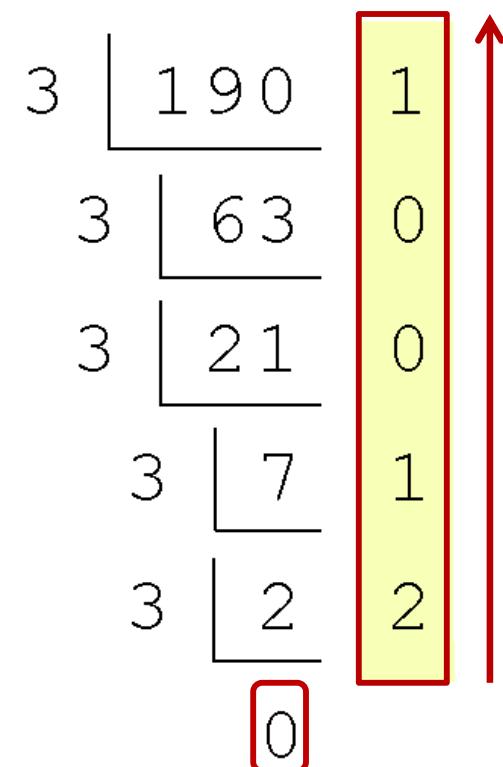
- 63 is evenly divisible by 3.
- The quotient is 21, the remainder is 0.

$$\begin{array}{r} 3 \longdiv{190} & 1 \\ 3 \longdiv{63} & 0 \\ \hline & 21 \end{array}$$

## 2.3 Converting Between Bases

- Converting  $190_{10}$  to base 3...
  - Continue in this way until the quotient is 0.
  - In the final calculation, we note that 3 divides 2 zero times with a remainder of 2.
  - Our result, reading from bottom to top is:

$$190_{10} = 21001_3$$



It is **algorithmic and easier!**

# Exercise

---

$458_{10} = \underline{\hspace{2cm}}_2$

$652_{10} = \underline{\hspace{2cm}}_2$

$458_{10} = \underline{\hspace{2cm}}_3$

$652_{10} = \underline{\hspace{2cm}}_5$

- Once you get the result, please verify your result by converting back!
  - Don't user calculator!
-

# Exercise

---

- $458_{10} = 1\ 1100\ 1010_2$
  - $652_{10} = 10\ 1000\ 1100_2$
  - $458_{10} = 121222_3$
  - $652_{10} = 10102_5$
  
  - Once you get the result, please verify your result by converting back!
  - Don't use calculator!
-

## 2.3 Converting Fractional Numbers

---

- Fractional numbers can be approximated in all base systems, too.
- Unlike integer values, fractional numbers do **not** necessarily have **exact representations under all radices**.
  - The quantity  $\frac{1}{2}$  is exactly representable in the binary and decimal systems, but is not in the base 3 numbering system.

## 2.3 Converting Fractional Numbers

---

- Fractional decimal numbers have non-zero digits on **the right of the decimal point.**
  - Fractional values of other radix systems have nonzero digits on **the right of the *radix point*.**
- Numerals on the right of a radix point represent negative powers of the radix. For example

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$

$$0.11_2 = 1 \times 2^{-1} + 1 \times 2^{-2}$$

$$= \frac{1}{2} + \frac{1}{4}$$

$$= 0.5 + 0.25$$

$$= 0.75_{10}$$

---

## 2.3 Converting Fractional Numbers

---

- Like the integer conversions, you can use either of the following two methods:
  - The ***Subtraction*** (-) method, or
  - The ***multiplication*** (x) method.
- The subtraction method for fractions is same as the method for integers
  - Subtract *negative powers of the radix*.
- Always start with the *largest value* --- first,  $n^{-1}$ , where  $n$  is the radix.

# 2.3 Converting Fractional Numbers

- Using the subtraction method to convert the decimal  $0.8125_{10}$  to  $X_2$ .

- Our result, reading from top to bottom is:

$$0.8125_{10} = 0.1101_2$$

- Subtraction stops when the remainder becomes 0
- Of course, this method works with any base, not just binary.

$$\begin{array}{r} 0.8125 \\ - 0.5000 \\ \hline 0.3125 \\ - 0.2500 \\ \hline 0.0625 \\ - 0 \\ \hline 0.0625 \\ - 0.0625 \\ \hline 0 \end{array} = \begin{array}{l} 2^{-1} \times 1 \\ 2^{-2} \times 1 \\ 2^{-3} \times 0 \\ 2^{-4} \times 1 \end{array}$$

0.125

## 2.3 Converting Fractional Numbers

---

- Using the *multiplication* method to convert the decimal  $0.8125_{10}$  to  $X_2$ , we multiply by the radix 2.
- The first product carries into the units place.

$$\begin{array}{r} .8125 \\ \times 2 \\ \hline 1.6250 \end{array}$$

# 2.3 Converting Fractional Numbers

---

## □ Converting $0.8125_{10}$ to $X_2$ . .

- Ignoring the value in the units place at each step, continue multiplying each fractional part by the radix.

$$\begin{array}{r} .8125 \\ \times 2 \\ \hline 1.6250 \\ .6250 \\ \times 2 \\ \hline 1.2500 \\ .2500 \\ \times 2 \\ \hline 0.5000 \end{array}$$

A vertical multiplication diagram showing the conversion of 0.8125 from base 10 to base 2. The process involves multiplying the fractional part by 2 and keeping track of the integer part (the radix point) for each step. The first multiplication (0.8125 \* 2) results in 1.6250, where 1 is the integer part and 0.6250 is the new fractional part. This pattern repeats three more times, resulting in the binary fraction 0.1101, which is highlighted with a red rectangle.

# 2.3 Converting Fractional Numbers

## □ Converting $0.8125_{10}$ to $X_2$ ..

- You are finished when the product is 0, or until you have reached the desired number of binary places.
- Our result, reading from top to bottom is:

$$0.8125_{10} = 0.1101_2$$

- Multiplication stops when the fractional part becomes 0
- This method also works with any base. Just use *the target radix* as the multiplier.

A vertical multiplication diagram for converting a decimal fraction to binary. On the left, the decimal fraction  $.8125_{10}$  is written above the multiplication symbol  $\times$ , with a red vertical bar highlighting the first digit after the decimal point. To the right of the multiplication symbol is the multiplier  $2$ . The first step shows  $.8125 \times 2 = .6250$ , with the integer part  $1$  written below the line. A red arrow points down to the first digit of the result (.6). Subsequent steps show:  
Step 1:  $.6250 \times 2 = .2500$   
Step 2:  $.2500 \times 2 = .5000$   
Step 3:  $.5000 \times 2 = .0000$   
The final result is  $0.1101_2$ , where each digit corresponds to the integer part of the result from each step. The last result,  $.0000$ , is highlighted with a red box.

## 2.3 Binary and Hexadecimal Number

---

- Binary numbering (base 2) system is **the most important** radix system in computers.
  - But, it is difficult to read long binary strings
    - For example:  $11010100011011_2 = 13595_{10}$
  - For compactness, binary numbers are usually expressed as **hexadecimal (base-16)** numbers.
-

## 2.3 Converting Between Bases

- The **hexadecimal** numbering system uses the numerals 0,..,9, A,...,F
  - $12_{10} = C_{16}$
  - $26_{10} = 1A_{16}$
- It is easy to convert between base 16 and base 2, because  $16 = 2^4$ .
- Thus, to convert from binary to hexadecimal,
  - Group the binary digits into groups of **4 bits** --- a nibble.

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8

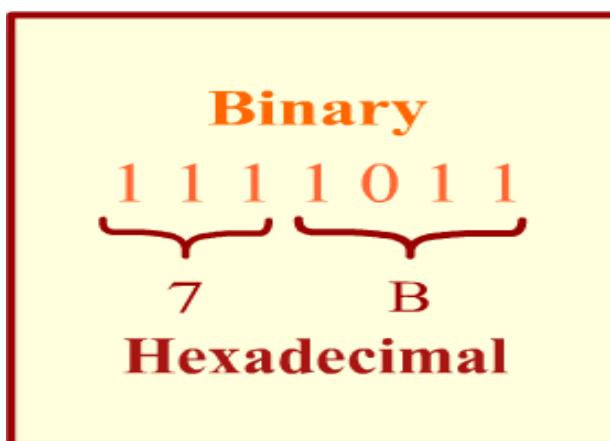
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

# Converting Binary to Hexadecimal

---

- Each hexadecimal digit corresponds to 4 binary bits.
- Example: Translate the binary integer  
**0001 0110 1010 0111 1001 0100** to hexadecimal

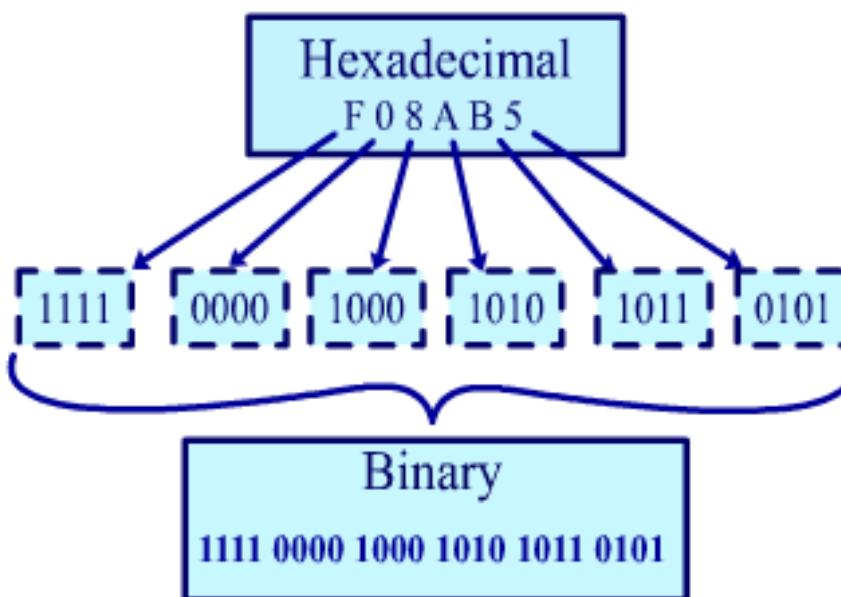
1	6	A	7	9	4
0001	0110	1010	0111	1001	0100



# Converting Hexadecimal to Binary

---

- Each hexadecimal digit can be converted to its 4-bit binary number to form the binary equivalent.



## 2.3 Converting Between Bases

---

- Using groups of hextets, the binary number  $13595_{10}$  ( $= 11010100011011_2$ ) in hexadecimal is:

*If the number of bits*

- Octal (base 8) values are derived from binary by using groups of three bits ( $8 = 2^3$ ):

**Octal was useful when a computer used six-bit words.**

## 2.3 Converting Between Bases

- Using groups of hextets, the binary number  $13595_{10}$  ( $= 11010100011011_2$ ) in hexadecimal is:

0011 0101 0001 1011  
3      5      1      B

*If the number of bits is not a multiple of 4, pad on the left with zeros!*

**351B<sub>16</sub>**

- Octal (base 8) values are derived from binary by using groups of three bits "octets" ( $8 = 2^3$ ):

011 010 100 011 011  
3      2      4      3      3

**32433<sub>8</sub>**

**Octal was useful when a computer used six-bit words.**

# Conversion between bases 2 and $2^n$

---

- Convert from base 2 to base 16

$$\underline{1} \underline{0} \underline{1} \underline{1} \underline{1} \underline{0} \underline{0} \underline{1} \underline{1} \underline{0} \underline{1} \underline{0} \underline{1} \underline{0}_2 = \quad 16$$

- Convert from base 2 to base 8

$$1 \underline{0} \underline{1} \underline{1} \underline{1} \underline{0} \underline{0} \underline{1} \underline{1} \underline{0} \underline{1} \underline{0} \underline{1} \underline{0}_2 = \quad 8$$

# Conversion between bases 2 and $2^n$

---

- Convert from base 2 to base 16

$$\underline{101110} \underline{0110} \underline{1010}_2 = 2E6A_{16}$$

- Convert from base 2 to base 8

$$101110 \underline{0110} \underline{1010}_2 = 27152_8$$

# Converting Hexadecimal to Decimal

---

- Multiply each digit by its corresponding power of 16:

$$\text{Decimal} = (d_3 \times 16^3) + (d_2 \times 16^2) + (d_1 \times 16^1) + (d_0 \times 16^0)$$

$d_i$  = hexadecimal digit at the  $i$ th position

- Examples:

- $1234_{16} = (1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$   
 $= 4660_{10}$
- $3BA4_{16} = (3 \times 16^3) + (11 \times 16^2) + (10 \times 16^1) + (4 \times 16^0) = 15268_{10}$

# Exercise

---

$58_{16} = \underline{\hspace{2cm}}_{10}$

$152_8 = \underline{\hspace{2cm}}_{10}$

$56_7 = \underline{\hspace{2cm}}_{10}$

$52_{11} = \underline{\hspace{2cm}}_{10}$

- Once you get the result, please verify your result by converting back!
  - Don't user calculator!
-

# Exercise

---

- $58_{16} = 88_{10}$
  - $152_8 = 106_{10}$
  - $56_7 = 41_{10}$
  - $52_{11} = 57_{10}$
  
  - Once you get the result, please verify your result by converting back!
  - Don't user calculator!
-

# Conversion between bases $2^m$ and $2^n$

---

- Convert from base 16 to base 8
- You can use a intermediate radix number
- For example
  - Base 16 to Base 2 (binary)
  - Base 2 (binary) to Base 8

$$\begin{aligned} A9DB3_{16} &= 1010 \ 1001 \ 1101 \ 1011 \ 0011_2 \\ &= 10 \ 101 \ 001 \ 110 \ 110 \ 110 \ 011_2 \\ &= 251663_8 \end{aligned}$$

# EXERCISES

---

$176_{10} = \underline{\hspace{2cm}}_{16}$

$55801_{10} = \underline{\hspace{2cm}}_8$

$A6_{16} = \underline{\hspace{2cm}}_{13}$

$55_8 = \underline{\hspace{2cm}}_{16}$

---

# EXERCISES

---

$$\square 176_{10} = B0_{16}$$

$$\square 55801_{10} = 154771_8$$

$$\square A6_{16} = 166_{10} = CA_{13}$$

$$\square 55_8 = 2D_{16}$$

# Conversion between bases 2 and base N

- You must familiar with the table on right
- You must familiar with the following conversion:
  - Converting from **bases 2** to **base  $2^n$**
  - Converting between **base  $2^m$**  and **base  $2^n$**
  - Converting from **bases 2** to **base 10**
  - Converting from **bases 10** (decimal) to **base 16** (hex)

hex	dec	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

# Outline

---

- Converting between different numeric-radix systems      ==> unsigned integer representation
  - **Binary addition and subtraction**
  - Two's complement representation
  - Floating-point representation
  - Characters in computer
- ==> signed  
integer  
representation

# Binary addition

---

- When the sum exceeds 1, carry a 1 over to the next-more-significant column (**addition rules**)
  - $0 + 0 = 0$  carry 0
  - $0 + 1 = 1$  carry 0
  - $1 + 0 = 1$  carry 0
  - **$1 + 1 = 0$  carry 1**

# Binary subtraction

---

## □ Subtraction rules

- $0 - 0 = 0$  borrow 0
- $0 - 1 = 1$  **borrow 1**
- $1 - 0 = 1$  borrow 0
- $1 - 1 = 0$  borrow 0

$$\begin{array}{r} {}^1 0 \\ - 1 \\ \hline \end{array} = 1$$

# Unsigned number: Addition and subtraction

---

- Exercise: Use unsigned binary to compute

- $100_{10} + 10_{10}$

$$\begin{array}{r} 0110\ 0100 \\ + 0000\ 1010 \\ \hline 0110\ 1110 \end{array}$$

- $100_{10} - 10_{10}$

$$\begin{array}{r} 0110\ 0100 \\ - 0000\ 1010 \\ \hline 0101\ 1010 \end{array}$$

- Use 8-bit unsigned numbers to calculate  $100_{10} + 100_{10} + 100_{10}$  using binary addition  
 $=(300)_{10} ==>$  will need 9-bit USigned system!  
 $==>$  Overflow; a 2-byte number!

# Unsigned number: Overflow

---

- Possible solution:
  - If data is stored in register, you should **use longer register**, which can hold more bits
  - In this case, you need a register, which has at least two bytes to hold the result

# Unsigned number: Addition & Subtraction

$$\begin{array}{r} \text{11} \\ 259 \\ + 43 \\ \hline \text{302} \end{array}$$

$$\begin{array}{r} \text{11} \\ 1000\ 000\ 11 \\ + 1010\ 11 \\ \hline \text{1001}\ 01110 \end{array}$$

$$\begin{array}{r} 259 \\ - 43 \\ \hline 216 \end{array}$$

$$\begin{array}{r} 1000\ 000\ 11 \\ - 1010\ 11 \\ \hline 110110\ 00 \end{array}$$

# Unsigned number: Subtraction

---

$$\begin{array}{r} 312 \\ - 88 \\ \hline 224 \end{array} \quad \begin{array}{r} 100111000 \\ - 1011000 \\ \hline 11100000 \end{array}$$

$$\begin{array}{r} 45 \\ 22 \\ 19 \\ 17 \\ 29 \\ \hline 132 \end{array} \quad \begin{array}{r} 111 \\ 101010 \\ 10110 \\ 10011 \\ 10001 \\ 11101 \\ \hline 10000100 \end{array}$$

---

# Unsigned number: Multiplication

$$\begin{array}{r} 10110 - 22 \\ \times 0101 - 5 \\ \hline 10110 \\ 00000 \\ \hline 0110110 - 110 \end{array}$$

$$\begin{array}{r} 887 \\ \times 65 \\ \hline 1101110111 \\ 1110000100110111 \\ \hline \rightarrow 57655 \end{array}$$

# Unsigned number: Division

$$\begin{array}{r} 14 \\ \hline 4 \overline{)59} \\ 4 \downarrow \\ \hline 19 \\ -16 \\ \hline 3 \end{array}$$

$$\begin{array}{r} 1110 \\ \hline 100 \overline{)111011} \\ -100 \downarrow \\ \hline 0110 \\ -100 \downarrow \\ \hline 0101 \\ -100 \downarrow \\ \hline 0011 \end{array}$$

$$\begin{array}{r} 11 \\ \hline 7 \overline{)77} \\ -7 \downarrow \\ \hline 7 \\ -7 \\ \hline 0 \end{array}$$

$$\begin{array}{r} 1011 \\ \hline 111 \overline{)1001101} \\ -111 \downarrow \\ \hline 01010 \\ -111 \downarrow \\ \hline 0111 \\ -111 \\ \hline 0 \end{array}$$

# Outline

---

- Converting between different numeric-radix systems
  - Binary addition and subtraction
  - **Two's complement representation**
  - Floating-point representation
  - Characters in computer
-

## 2.4 Signed Integer Representation

---

- So far, we have presented the conversions only involving ***unsigned numbers*** → all positive or 0
  - To represent signed integers, computer systems user the **high-order bit** to indicate the sign of a number.
    - The high-order bit is the **leftmost bit**, which also called the “***Most Significant Bit***” (MSB).
      - 0 → a positive number or 0;
      - 1 → a negative number or 0.
  - The remaining bits contain **the value** of the number
-

# 2.4 Signed Integer Representation

---

- In a byte, ***signed integer*** representation
  - 7 bits to represent ***the value*** of the number
  - 1 sign bit.
- There are three ways, where signed binary integers may be expressed:
  - **Signed magnitude**                          ==> signed binary rep.  
    ==> 1111 1111 = -127
  - One's complement
  - **Two's complement**                          ==> 1111 1111 = -1  
    how: MSb=1 means -ve num  
    flip & increment:  
    0000 0000+1 = 0000 0001 = (1)<sub>10</sub>

# Two's Complement Representation

## ❖ Positive numbers

✧ Signed value = Unsigned value

## ❖ Negative numbers

✧ Signed value = Unsigned value -  $2^n$

✧  $n$  = number of bits

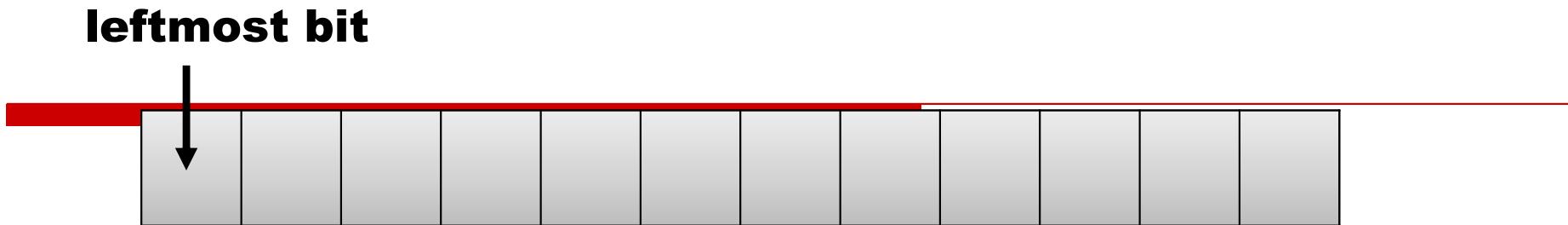
3-bit Bin Repr.	Unsigned Value	Sign-Mag Value	1C Value	2C Value
000	0	+0	+0	+0
001	1	+1	+1	+1
010	2	+2	+2	+2
011	3	+3	+3	+3
100	4	-0	-3	-4
101	5	-1	-2	-3
110	6	-2	-1	-2
111	7	-3	-0	-1

3 ways: Signed Value Representations

N=3 bit system  
Number of combinations =  $2^N = 2^3 = 8$

8-bit Binary value	Unsigned value	Signed value
00000000	0	0
00000001	1	+1
00000010	2	+2
...	...	...
01111110	126	+126
01111111	127	+127
10000000	128	-128
10000001	129	-127
...	...	...
11111110	254	-2
11111111	255	-1

# N-bit two's complement



- Positive integer
  - Set leftmost bit to **0**
  - Express magnitude in binary in the rightmost  $n-1$  bits
- Negative integer
  - Represent the magnitude (positive) as above
  - Then **negate (complement)** the result (see next slide), and add 1
- The leftmost bit is *the sign bit*
  - **0** for positive
  - **1** for negative

# Negative Integer Representation

- 2's compliment for a negative number  $-x$ 
  1. Represent the positive number  $x$  in binary
  2. *Negate* all bits
  3. *Add 1* to the result

Let n = 6 bits

Represent magnitude  $+14_{10} = 001110$

Complement each bit       $110001$

Add 1                         $\begin{array}{r} + 1 \\ \hline 110010 \end{array}$

Result  $-14_{10} = 110010_2$

Check by negating the result

Start with result  $-14_{10} = 110010$

Complement each bit       $001101$

Add 1                         $\begin{array}{r} + 1 \\ \hline 001110 \end{array}$

As expected, we get  $+14_{10} = 001110_2$

# Another Example

---

- Represent **-36** in 2's complement format

starting value	00100100 = +36
step1: reverse the bits (1's complement)	11011011
step 2: add 1 to the value from step 1	+ 1
sum = 2's complement representation	11011100 = -36

- Verification:

Sum of an integer and its 2's complement **must be zero**:

$$00100100 + 11011100 = 00000000 \text{ (8-bit sum)} \Rightarrow \text{Ignore Carry}$$

# Addition and subtraction

---

- Addition of two's complement numbers
  - Add all  $n$  bits using binary arithmetic
  - **Throw away any carry** from the leftmost bit position
  - Do this **for any sign** (whether the same or different) See Overflow rules A and C upcoming!
- For example:  $X-Y$ 
  - First, negate Y. Then, add to X
  - Thus,  $X-Y = X + (-Y)$

# Examples of addition

Let n = 6 bits

Add 5 and 6 to obtain 11

$$+5_{10} = 000101$$

$$+6_{10} = \underline{000110}$$

$$+11_{10} = 001011$$

Let n = 6 bits

$$-14 + 9 = -5$$

$$-14_{10} = 110010$$

$$+9_{10} = \underline{001001}$$

$$-5_{10} = 111011$$

Check magnitude of  $-5_{10}$

$$\text{Negate } -5_{10} = 111011$$

$$\text{Complement } 000100$$

$$\text{Add 1} \quad \begin{array}{r} + \\ 1 \end{array}$$

$$\text{Magnitude: } +5 = 000101$$

OK

$$+14 = 001110$$

$$-14 = 110010$$

$$+9 = 001001$$

$$-9 = 110111$$

Let n = 6 bits

$$-14 - 9 = -23$$

$$-14_{10} = 110010$$

$$-9_{10} = \underline{110111}$$

$$-23_{10} = 101001$$

Check magnitude of  $-23_{10}$

$$\text{Negate } -23_{10} = 101001$$

$$\text{Complement } 010110$$

$$\text{Add 1} \quad \begin{array}{r} + \\ 1 \end{array}$$

$$\text{Magnitude: } +23 = 010111$$

OK

Verification

# Wrap Up

---

- 2's complement:
  - Positive integer → Same
  - Negative integer → Complement /flip all bits and add 1
- Use two's complement to compute
  - $100_{10} + 10_{10}$
  - $-100_{10} + 10_{10}$
  - $100_{10} - 10_{10}$
  - $-100_{10} - 10_{10}$

---

Please first convert decimal to binary

# Background for EFLAGS

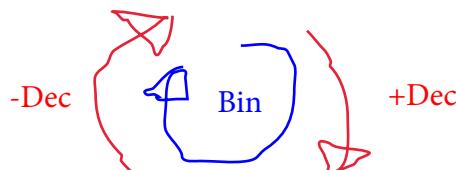
## Signed integers representation

### signed-binary

Eight-bit sign-magnitude		
Binary value	Sign-magnitude interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
:	:	:
01111101	125	125
01111110	126	126
01111111	127	127
10000000	-0	128
10000001	-1	129
10000010	-2	130
:	:	:
11111101	-125	253
11111110	-126	254
11111111	-127	255

- Two Zeros
- Range: -127 ~ +127
- FP's Significand uses this!

**-ve number:** Signed value =  $2^{n-1}$  - Unsigned value  
 Signed value = 128 - Unsigned value



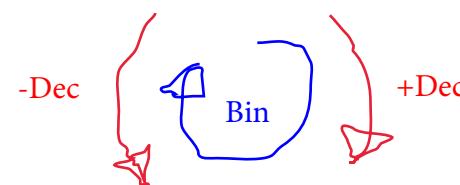
### 1's complement

Eight-bit ones' complement		
Binary value	Ones' complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
:	:	:
01111101	125	125
01111110	126	126
01111111	127	127
10000000	-127	128
10000001	-126	129
10000010	-125	130
:	:	:
11111101	-2	253
11111110	-1	254
11111111	-0	255

- Two Zeros
- Range: -127 ~ +127

$$\text{Signed value} = \text{Unsigned value} - 2^n + 1$$

$$\text{Signed value} = \text{Unsigned value} - 255$$



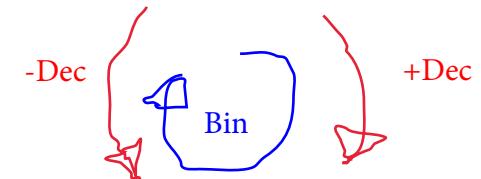
### 2's complement \*

Eight-bit two's complement		
Binary value	Two's complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
:	:	:
01111110	126	126
01111111	127	127
10000000	-128	128
10000001	-127	129
10000010	-126	130
:	:	:
11111110	-2	254
11111111	-1	255

- one Zero only
- Range: -128 ~ +127

$$\text{Signed value} = \text{Unsigned value} - 2^n$$

$$\text{Signed value} = \text{Unsigned value} - 256$$



- \* **Addition and subtraction require different behavior** depending on the sign bit in Signed Binary, whereas one's complement can ignore the sign bit and just do an end-around carry, and two's complement can ignore the sign bit and depend on the overflow behavior.
- Comparison also require inspecting the sign bit** in Signed Binary and 1's complement, whereas in two's complement, one can simply subtract the two numbers, and check if the outcome is positive or negative.

# Background for EFLAGS

## Overflow detection

- $X$ ,  $Y$  and  $Z$  are  $N$ -bit 2's-complement numbers and  $Z_{2c} = X_{2c} + Y_{2c}$
- Overflow occurs if  $X_{2c} + Y_{2c}$  exceeds the maximum value represented by  $N$ -bits.
- **If the signs of  $X$  and  $Y$  are different, don't detect overflow for  $Z_{2c} = X_{2c} + Y_{2c}$ .**
- **In case the signs of  $X$  and  $Y$  are the same, if the sign of  $X_{2c} + Y_{2c}$  is opposite, overflow detected.**
  - Case 1:  $X$ ,  $Y$  positive,  $Z$  sign bit ='1'
  - Case 2:  $X$ ,  $Y$  negative,  $Z$  sign bit ='0'
- **If  $X$ ,  $Y$  and  $Z$  have same, dont detect overflow.**

If the signs of X and Y are different,  
dont detect overflow for  $Z_{2c}=X_{2c}+Y_{2c}$

# Example

## Case-A

- $X_{2c}=(01111010)_{2c}, Y_{2c}=(10001010)_{2c},$   
 $X_{2c}+Y_{2c}=(00000100)_{2c}$  No overflow!

0	1	1	1	1	0	1	0	122	
1	0	0	0	1	0	1	0	+ -118	
<hr/>								=	
1	0	0	0	0	0	1	0	0	+4

Overflow (carry-out) 1 BUT ignore it

Good!  
No  
problem!

- Case 1: X, Y positive, Z sign bit ='1'
- Case 2: X, Y negative, Z sign bit ='0'

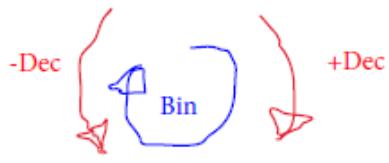
# Example

Detect overflow!

## Case-B-1

$X_{2c} = (01111010)_{2c}, Y_{2c} = (00001010)_{2c},$   
 $X_{2c} + Y_{2c} = (10000100)_{2c}$  Overflow detected

Signed value = Unsigned value -  $2^n$   
 Signed value = Unsigned value - 256



122		+		10		=		
0		0		0		1		
0		0		1		0		
<hr/>								
1		0		0		0		
1		0		1		0		
<hr/>								
132								

A negative sum of positive  
operands (or vice versa) is an  
overflow.

Ignore the sign bit and depend on  
the overflow behavior

Sign=1 negative

No carry-out, but  
sign-bit is the O.F. bit  
and dont ignore it!

BUT actual answer is -124 ==> problem.

Complement it to fix this, once this OF is  
detected! i.e.  $2^8 + (-124) = 132$

- Case 1: X, Y positive, Z sign bit ='1'
- Case 2: X, Y negative, Z sign bit ='0'

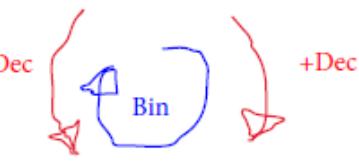
# Example

Detect overflow!

Case-B-2

□  $X_{2c}=(10011010)_{2c}, Y_{2c}=(10001010)_{2c}$ ,  
 $X_{2c}+Y_{2c}=(00100100)_{2c}$  Overflow detected

Signed value = Unsigned value -  $2^n$   
 Signed value = Unsigned value - 256

	1	0	0	1	1	0	1	0	-102
	1	0	0	0	1	0	1	0	+ -118
	<hr/>	1	0	0	1	0	1	0	=

A negative sum of  
positive operands  
(or vice versa) is  
an overflow.

Ignore the sign bit  
and depend on the  
overflow behavior

1    0    0    1    0    0    1    0    0    =  
 -220

Sign = 0

BUT actual answer is 36 ==> problem.

O.F. (carry-out) 1 BUT  
dont ignore it!

Solution:

Once this OF is detected, Do this:  
 $36 - 2^8 = -220$

- Case 1: X, Y positive, Z sign bit ='1'
- Case 2: X, Y negative, Z sign bit ='0'

# Example

Then, detect overflow!

Case-C-1

$X_{2c} = (00111010)_{2c}, Y_{2c} = (00001010)_{2c}$ ,

$X_{2c} + Y_{2c} = (01000100)_{2c}$  No overflow

Good!  
No problem!

$$\begin{array}{r}
 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 58 \\
 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & + \\
 \hline
 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 10 \\
 & & & & & & & & = \\
 & & & & & & & & 68
 \end{array}$$

Sign=0 positive

Correct math and actual answer, so no need to make correction and hence, No importance of OF even if there were to be one!

- Case 1: X, Y positive, Z sign bit ='1'
- Case 2: X, Y negative, Z sign bit ='0'

# Example

Then, detect overflow!

## Case-C-2

□  $X_{2c}=(11111010)_{2c},$

$$Y_{2c}=(10001010)_{2c},$$

$$X_{2c}+Y_{2c}=(10000100)_{2c}$$

No overflow

Good!

No problem!

1	1	1	1	1	0	1	0	-6
1	0	0	0	1	0	1	0	+
								-118
1	1	0	0	0	0	1	0	= -124

Sign = 1

Overflow (carry-out) 1 BUT ignore it

# C Programming Example

---

```
#include <stdio.h>

int main()
{
    int a = 32767;
    short b;

    printf ("size of int = %ld, size of short = %ld\n", sizeof(int), sizeof(short));

    b = (short)a;
    printf ("a = %d, b = %d\n", a, b);

    a++;
    b = (short)a;
    printf ("a = %d, b = %d\n", a, b);

    return 0;
}
```

# C Programming Example

---

```
#include <stdio.h>

int main()
{
    int a = 32767;
    short b;

    printf ("size of int = %ld, size of short = %ld\n", sizeof(int), sizeof(short));
size of int = 4, size of short = 2
    b = (short)a;
    printf ("a = %d, b = %d\n", a, b);
a = 32767, b = 32767

    a++;
    b = (short)a;
    printf ("a = %d, b = %d\n", a, b);
a = 32768, b = -32768

    return 0;
}
```

# A Production Issue (C Language)

---

```
void do_something(short argu)
{
    .....
}

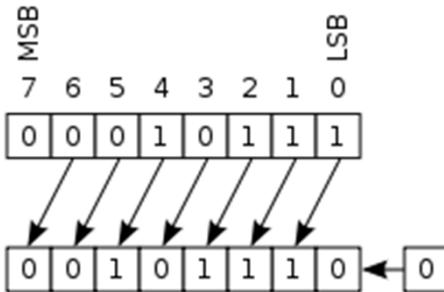
int main()
{
    int db_table_key;
    .....
    do_something(db_table_key);
    .....
}
```

## 2.4 Signed Integer Representation

---

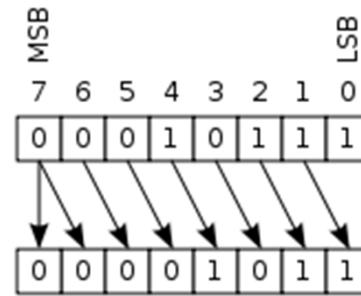
- In binary system, ***Multiplication/Division by 2*** very easily using an *arithmetic shift* operation
- A *left arithmetic shift* inserts a **0** in for the **rightmost bit** and **shifts everything else left one bit**; in effect, it multiplies by **2**
- A *right arithmetic shift* shifts everything one bit to the right, but copies the sign bit; it divides by **2**
- Let's look at the following examples...

# Bit Shifting (Arithmetic & Logical shift)



Left arithmetic shift

00010111 (decimal +23) LEFT-SHIFT  
= 00101110 (decimal +46)



Right arithmetic shift

10010111 (decimal -105) RIGHT-SHIFT  
= 11001011 (decimal -53)

- ❑ To multiply 23 by 4, simply left-shift *twice*
- ❑ To divide 105 by 4, simply right-shift *twice*

# Outline

---

- Converting between different numeric-radix systems
  - Binary addition and subtraction
  - Two's complement representation
  - **Floating-point representation**
  - Characters in computer
-

# 2.5 Floating-Point Representation

---

- The two's complement representation that we have just presented deals 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.

Work sucks.



A comedy from Mike Judge, creator of 'Beavis and Butt-head' and co-creator of 'King of the Hill'

# OfficeSpace

TWENTIETH CENTURY FOX PRESENTS A MIKE JUDGE FILM "OFFICE SPACE"

RON LIVINGSTON JENNIFER ANISTON STEPHEN ROOT GARY COLE JOHN FRIZZELL DAVID REUBEN EDWARD RAVOY TIM SCHNEIDER ASC  
R GUY RIEDEL MICHAEL ROTENBERG DANIEL RAPPAPORT PRODUCED BY NINE JUDGE www.ofspace.com

FEBRUARY 19 ONLY IN THEATRES

# 2.5 Floating-Point Representation

---

- If we are clever programmers, we can perform **floating-point calculations using any integer format**.
- This is called *floating-point emulation*, because floating point values aren't stored as such; we just create programs that make it seem as if floating-point values are being used.
- Most of today's computers are equipped with **specialized hardware** that performs floating-point arithmetic with no special programming required.

Intel 8087 in 1980 at 4-10 MHz  
for 8086

# 2.5 Floating-Point Representation

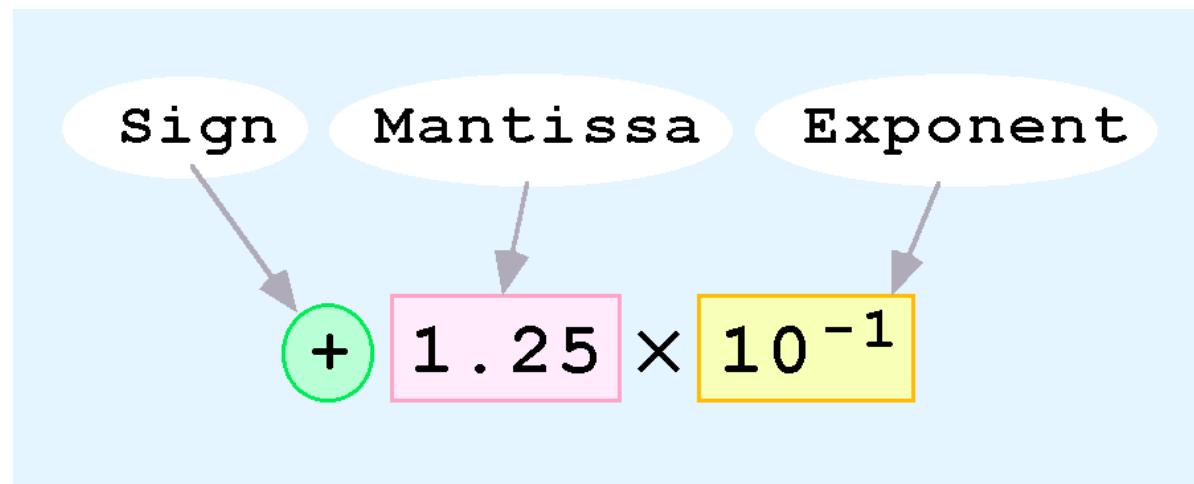
---

- 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:  
 $0.125 = 1.25 \times 10^{-1}$   
 $5,000,000 = 5.0 \times 10^6$

# 2.5 Floating-Point Representation

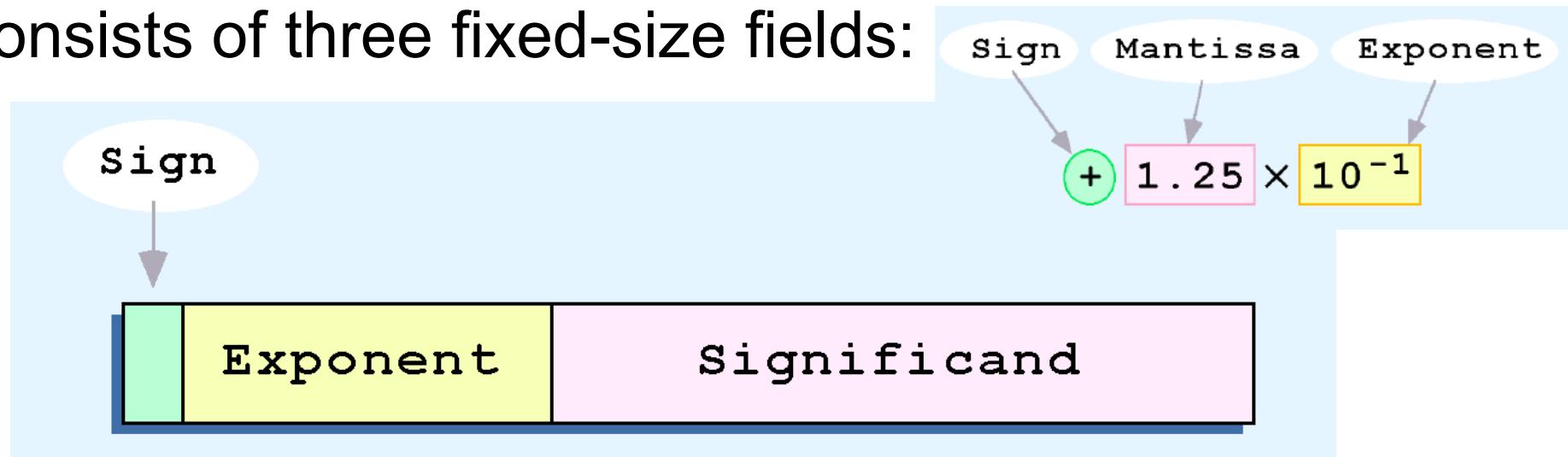
---

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



# 2.5 Floating-Point Representation

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

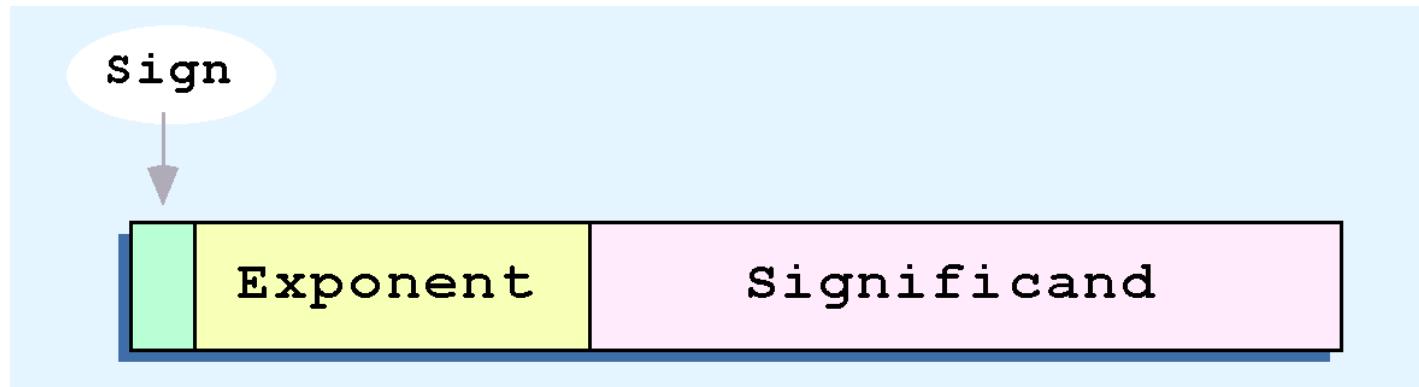


- This is the standard arrangement of these fields.

*Note: Although “significand” and “mantissa” do not technically mean the same thing, many people use these terms interchangeably. We use the term “significand” to refer to the fractional part of a floating point number.*

# 2.5 Floating-Point Representation

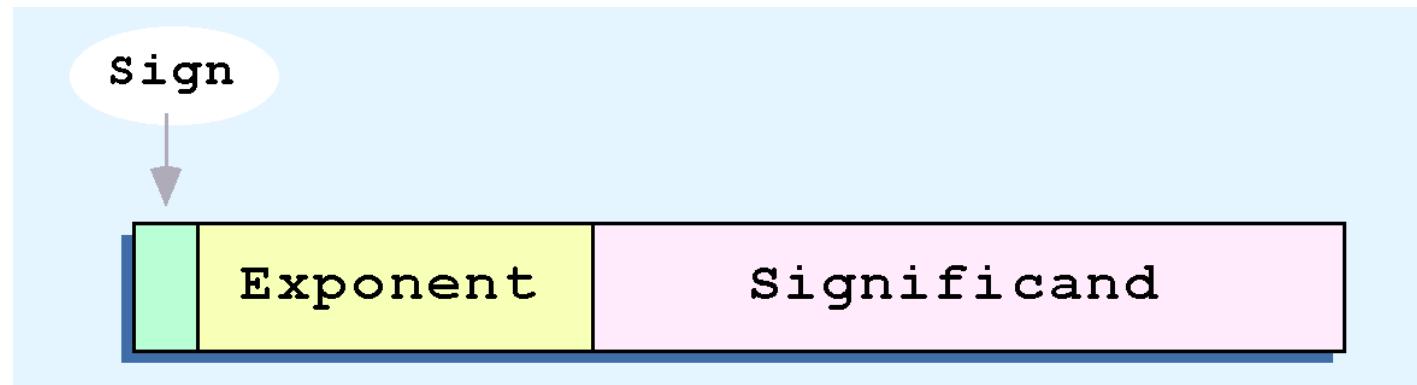
---



- The one-bit sign field is the sign of the stored value.
- The size of the exponent field determines the **range** of values that can be represented.
- The size of the significand determines the **precision** of the representation.

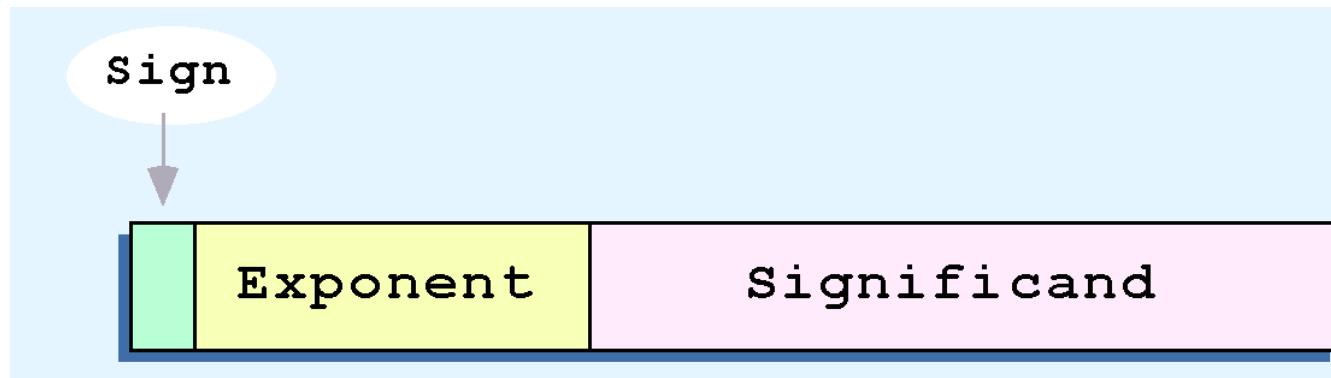
# 2.5 Floating-Point Representation

---



- We introduce 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

# 2.5 Floating-Point Representation

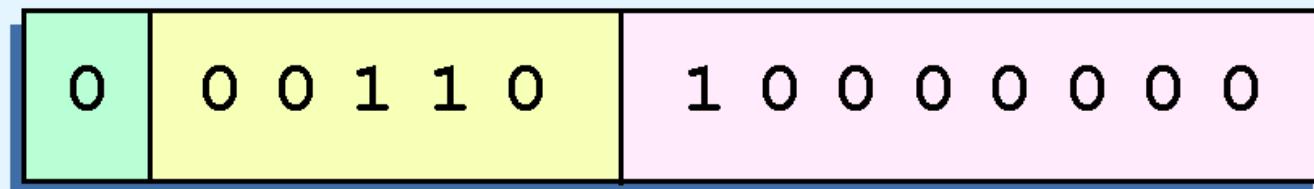


- The significand is always **preceded by an implied binary point**.
- Thus, the significand always **contains a fractional binary value**.
- The exponent indicates the **power of 2** by which the significand is **multiplied**.

# 2.5 Floating-Point Representation

---

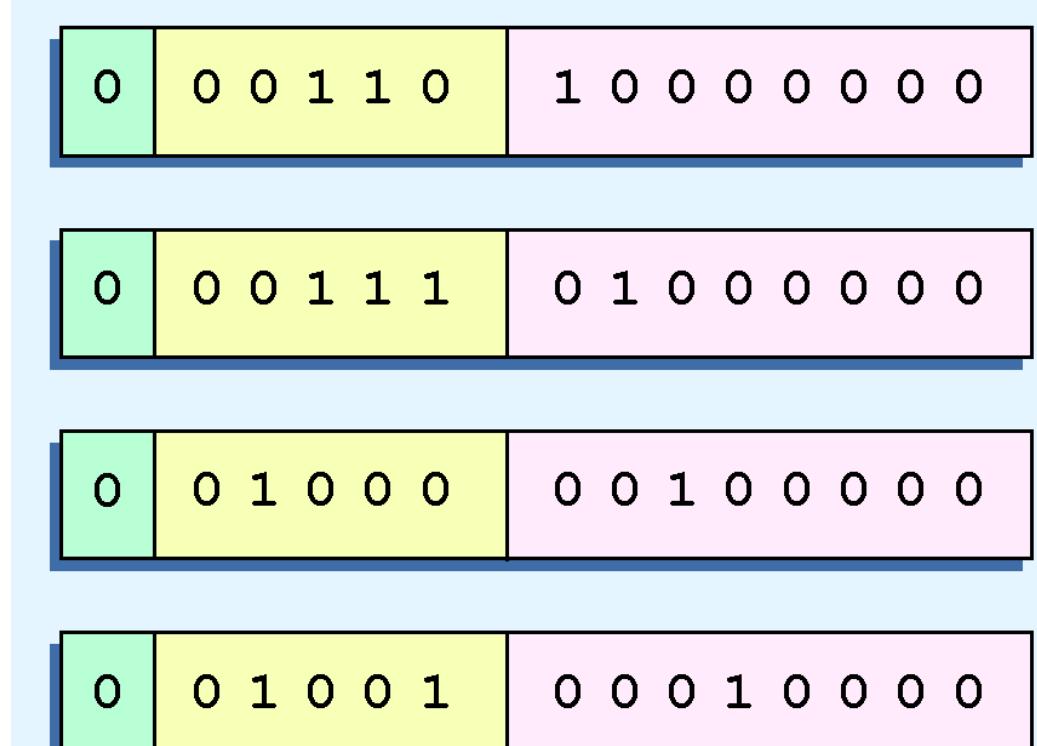
- Example:
  - Express  $32_{10}$  in the simplified 14-bit floating-point model.
- We know that  $32$  is  $2^5$ . So in (binary) scientific notation  $32 = 1.0 \times 2^5 = 0.1 \times 2^6$ 
  - In a moment, we'll explain why we prefer the second notation versus the first.
- Using this information, we put  $110$  ( $= 6_{10}$ ) in the exponent field and  $1$  in the significand as shown.



# 2.5 Floating-Point Representation

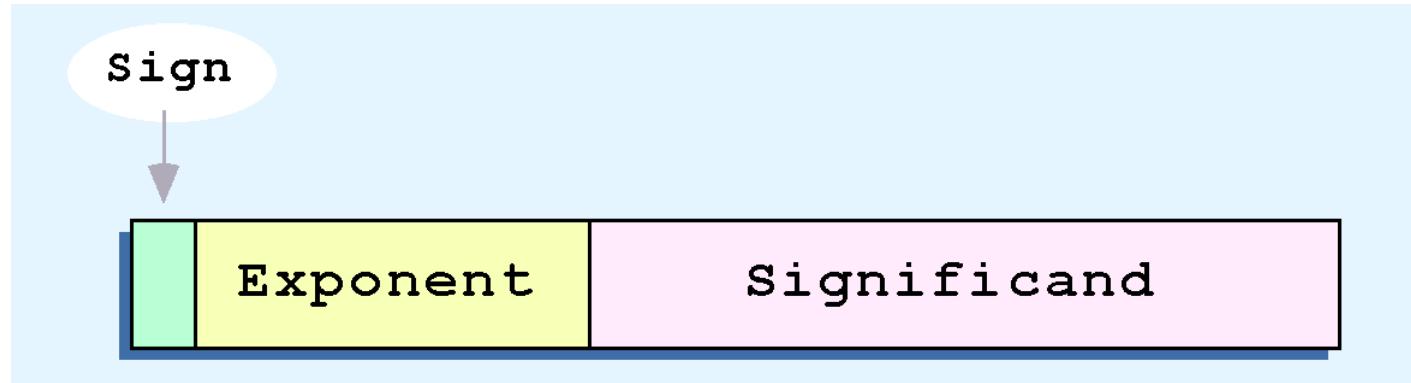
---

- The illustrations shown at the right are *all equivalent representations* for 32 using our simplified model.
- Not only do these synonymous representations **waste space**, but they can also cause **confusion**.



# 2.5 Floating-Point Representation

---



- Another problem with our system is that we have made no allowances for **negative exponents**. We have no way to express  $0.5 (=2^{-1})$ ! (Notice that there is **no sign in the exponent field**.)  
  

All of these problems can be fixed with no changes to our basic model.

# 2.5 Floating-Point Representation

---

- To resolve the problem of synonymous forms, we establish a rule **for the significand** that the **first '1' will appear after the radix point**.
- This process, called *normalization*, results in a unique pattern for each floating-point number.
  - In our simple model, all significands must have the form  $0.1xxxxxxx$
  - For example,  $4.5 = 100.1 \times 2^0 = 1.001 \times 2^2 = \mathbf{0.1001 \times 2^3}$ . The last expression is correctly normalized.

*In our simple instructional model, we use no implied bits.*

# 2.5 Floating-Point Representation

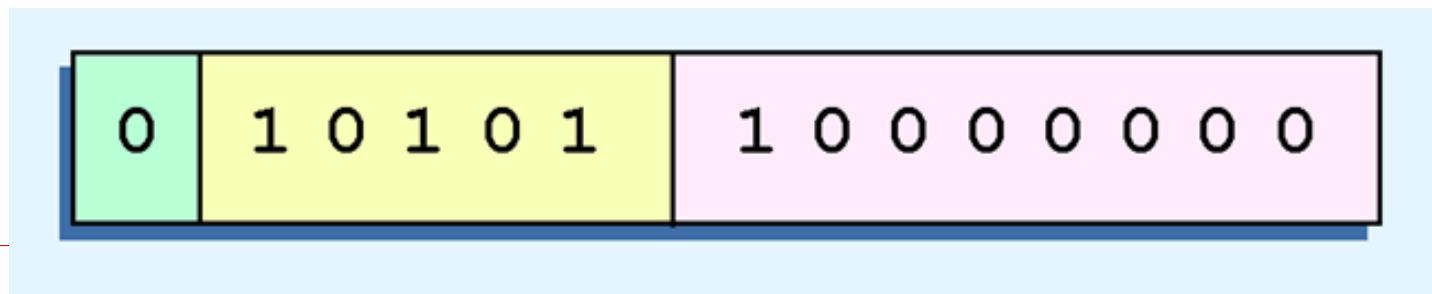
---

- To provide for negative exponents, we **will use a *biased exponent***.  $= 2^{\text{exp}-1} - 1$  where 'exp' is the number of bits of the exponent field
  - In our case, we have a 5-bit exponent.
  - $2^{5-1} - 1 = 2^4 - 1 = 15$
  - Thus will use **15 for our bias**: our exponent will use **excess 15 representation**.
- In our model, exponent **values less than 15 are negative, representing fractional numbers**.

# 2.5 Floating-Point Representation

---

- Example:
  - Express  $32_{10}$  in the revised 14-bit floating-point Simple Model.
- We know that  $32 = 1.0 \times 2^5 = 0.1 \times 2^6$ .
- To use our excess 15 biased exponent, we add 15 to 6, giving  $21_{10}$  ( $=10101_2$ ).
- So we have:



# 2.5 Floating-Point Representation

---

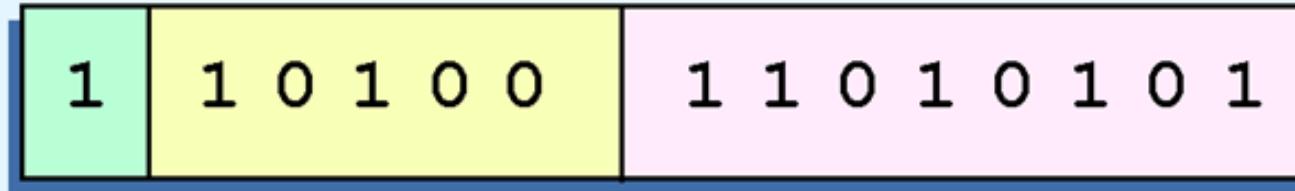
- Example:
  - Express  $0.0625_{10}$  in the revised 14-bit floating-point model.
- We know that  $0.0625$  is  $2^{-4}$ . So in (binary) scientific notation  $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$ .
- To use our excess 15 biased exponent, we add 15 to  $-3$ , giving  $12_{10}$  ( $=01100_2$ ).



# 2.5 Floating-Point Representation

---

- Example:
  - Express  $-26.625_{10}$  in the revised 14-bit floating-point model.
- We find  $26.625_{10} = 11010.101_2$ . Normalizing, we have:  
 $26.625_{10} = 0.11010101 \times 2^5$ .
- To use our excess 15 biased exponent, we add 15 to 5, giving  $20_{10}$  ( $=10100_2$ ). We also need a 1 in the sign bit.



# Floating-Point Simple model to decimal number

## Example-2

0 | 0 1 1 1 0 | 1 0 0 0 0 0 0  
FPS

$$= + 0.1 0 0 0 0 0 0 \times 2^{-1}$$

$$= (0.01)_2$$

$$= 1 \times 2^{-2}$$

$$= 0.25$$

## Example-1

0 | 1 0 1 1 0 | 1 1 0 0 1 0 0 0  
FPS

$$+ 0.1 1 0 0 1 0 0 \times 2^7$$

$$= (1 1 0 0 1 0 0)_2$$

$$= 100$$

# 2.5 Floating-Point Representation

---

- The **IEEE** has established a standard for floating-point numbers
- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent (with a bias of 127) and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent (with a bias of 1023) and a 52-bit significand.

# 2.5 Floating-Point Representation

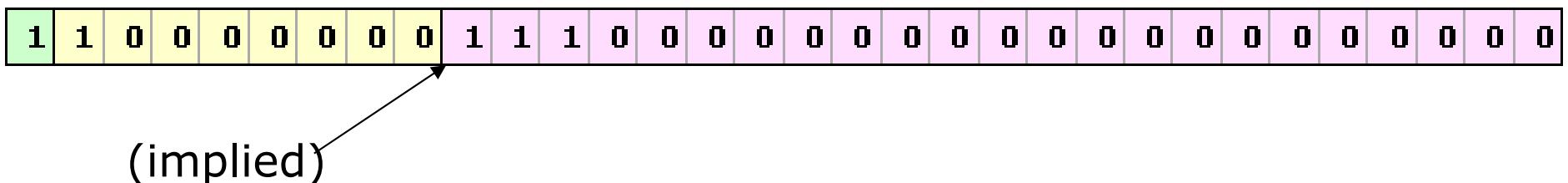
---

- In both the IEEE single-precision and double-precision floating-point standard, the significand has **an implied 1 to the LEFT of the radix point.**
  - The format for a significand using the IEEE format is: 1.xxx...
  - For example,  $4.5 = .1001 \times 2^3$  in IEEE format is  $4.5 = 1.001 \times 2^2$ . The 1 is **implied, which means it does not need to be listed in the significand** (the significand would include only 001).

# 2.5 Floating-Point Representation

---

- Example: Express -3.75 as a floating point number using IEEE single precision.
- First, let's normalize according to IEEE rules:
  - $-3.75 = -11.11_2 = -1.111 \times 2^1$
  - The bias is 127, so we add  $127 + 1 = 128$  (this is our exponent)
  - The first 1 in the significand is implied, so we have:



- Since we have an implied 1 in the significand, this equates to  $-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75$ .

# Exercise

---

- Use IEEE-754 single precision floating point standard to find binary representation of the following real number:

- 0.0625       $= 1.0 \times 2^{-4} ==> 0-0111\ 1011-0s\ (23\text{-time})$

- -26.625

$$= -1.1010101 \times 2^4 = 1-1000\ 0011-1010101\ 0s(16\text{-time})$$

# 2.5 Floating-Point Representation

---

- Using the IEEE-754 single precision floating point standard:
  - An exponent of 255 indicates a special value.
    - If the significand is zero, value is  $\pm\infty$  (MSb).
    - If the significand is nonzero, the value is NaN, “not a number,” often used to flag an error condition. An exponent of 0 indicates the zero value.
- Using the double precision standard:
  - The “special” exponent value for a double precision number is 2047, instead of the 255 used by the single precision standard.

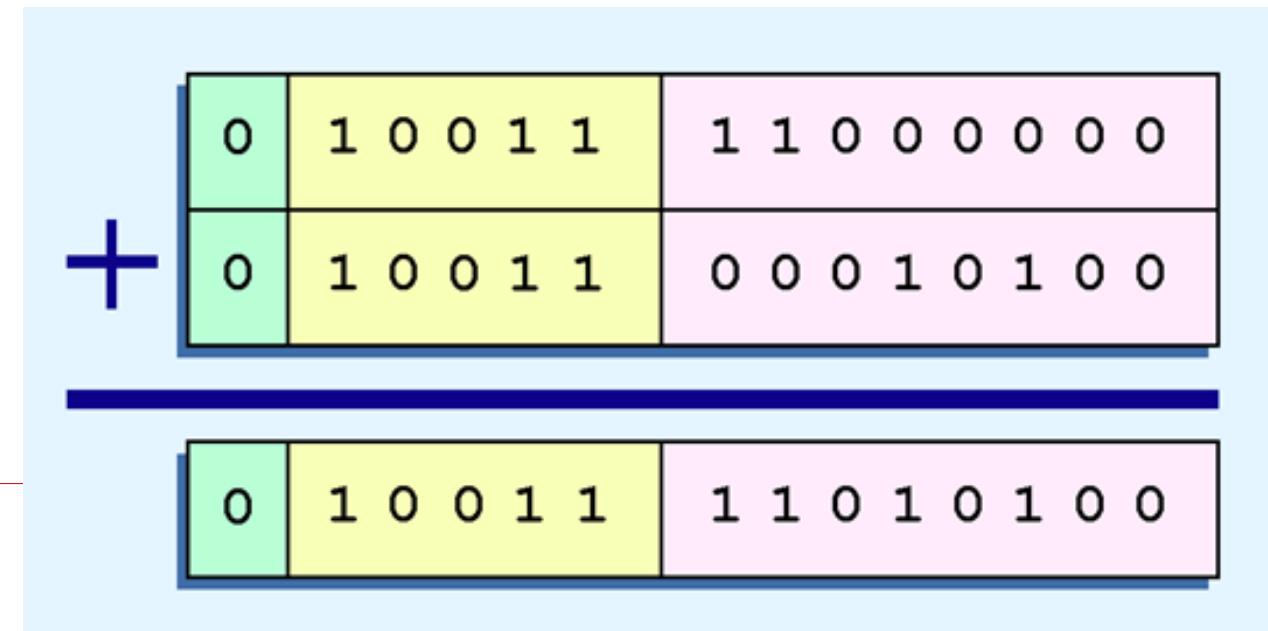
# 2.5 Floating-Point Representation

---

- Floating-point **addition** and **subtraction** are done using methods the same with how we perform calculations using pencil and paper.
- The first thing that we do is express both operands in the **same exponential power**, then add the numbers, preserving the exponent in the sum.
- If the exponent requires **adjustment**, we do so at the end of the calculation.

# 2.5 Floating-Point Representation

- Example:
  - Find the sum of  $12_{10}$  and  $1.25_{10}$  using the 14-bit “simple” floating-point model.
- We find  $12_{10} = 0.1100 \times 2^4$ . And  $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$ .
  - Thus, our sum is  $0.110101 \times 2^4$ .



## 2.5 Floating-Point Representation

---

- No matter how many bits we use in a floating-point representation, our model must be **finite**.
- The **real number** system is, of course, **infinite**, so our models can give nothing more than an **approximation** of a real value.
- At some point, every model breaks down, introducing **errors** into our calculations.
- By **using a greater number of bits** in our model, we can **reduce** these **errors**, but we can **never** totally **eliminate** them.

## 2.5 Floating-Point Representation

---

- Our job becomes one of reducing error, or at least being **aware of the possible magnitude of error** in our calculations.
- We must also be aware that **errors can compound through repetitive arithmetic operations**.
- For example, our 14-bit model **cannot** exactly represent the decimal value 128.5. In binary, it is 9 bits wide:

$$10000000.1_2 = 128.5_{10}$$

## 2.5 Floating-Point Representation

---

- When we try to express  $128.5_{10}$  in our 14-bit model, we lose the low-order bit, giving a **relative error** of:

$$\frac{128.5 - 128}{128.5} \approx 0.39\%$$

- If we had a procedure that repetitively added 0.5 to 128.5, we would have an error of nearly 2% after only four iterations.

## 2.5 Floating-Point Representation

---

- Floating-point errors can be **reduced when** we use **operands that are similar** in magnitude.
- If we were repetitively adding 0.5 to 128.5, it would have been better to iteratively add 0.5 to itself and then add 128.5 to this sum.
- In this example, the error was caused by loss of the **low-order bit**.
- Loss of the **high-order bit** is more **problematic**.

# 2.5 Floating-Point Representation

---

- Floating-point **overflow** and **underflow** can cause programs to crash.
- **Overflow** occurs when there is no room to store the high-order bits resulting from a calculation.
- **Underflow** occurs when a value is too small to store, possibly resulting in division by zero.

*Experienced programmers know that it's better for a program to crash than to have it produce incorrect, but plausible, results.*

## 2.5 Floating-Point Representation

---

- When discussing floating-point numbers, it is important to understand the terms *range*, *precision*, and *accuracy*.
- The **range** of a numeric integer format is the difference **between the largest and smallest** values that can be expressed.
- **Accuracy** refers to **how closely** a **numeric** representation **approximates** a **true** value.
- The **precision** of a number indicates how much **information** we have **about a value**

# 2.5 Floating-Point Representation

---

- Most of the time, greater precision leads to better accuracy, but this is **not** always true.
  - For example, 3.1333 is a value of pi that is accurate to two digits, but has 5 digits of precision.
- There are other problems with floating point numbers.
- Because of truncated bits, you cannot always assume that a particular floating point operation is **associative** or **distributive**.

## 2.5 Floating-Point Representation

---

- This means that we cannot assume:
$$(a + b) + c = a + (b + c) \text{ or}$$
$$a*(b + c) = ab + ac$$
- Moreover, to test a floating point value for equality to some other number, it is best to declare a “nearness to  $x$ ” epsilon value. For example, instead of checking to see if floating point  $x$  is equal to 2 as follows:

if  $x = 2$  then ...

it is better to use:

if ( $\text{abs}(x - 2) < \text{epsilon}$ ) then ...

---

(assuming we have epsilon defined correctly!)

# Outline

---

- Converting between different numeric-radix systems
  - Binary addition and subtraction
  - Two's complement representation
  - Floating-point representation
  - **Characters in computer**
-

## 2.6 Characters in Computer

---

- You might say: “*Well, I know numbers can be represented as binary, what about characters?*”
- Characters are also represented as binary
  - However, all characters uses ASCII (/ˈæski/ ass-kee), as a character-encoding scheme.
  - ASCII --- American Standard Code for Information Interchange

# ASCII Code

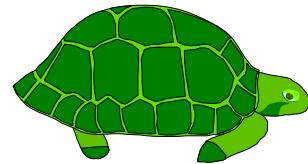
---

- It encodes 128 specified characters into 7-bit binary integers as shown by the ASCII chart.
  - The characters encoded are numbers 0 to 9, lowercase letters a to z, uppercase letters A to Z, basic punctuation symbols, control codes that originated with Teletype machines, and a space.
  - For the full ASCII table, see next page

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr		Dec	Hx	Oct	Html	Chr		Dec	Hx	Oct	Html	Chr
0	0	000	<b>NUL</b>	(null)	32	20	040	&#32;	<b>Space</b>		64	40	100	&#64;	<b>0</b>		96	60	140	&#96;	<b>`</b>
1	1	001	<b>SOH</b>	(start of heading)	33	21	041	&#33;	<b>!</b>		65	41	101	&#65;	<b>A</b>		97	61	141	&#97;	<b>a</b>
2	2	002	<b>STX</b>	(start of text)	34	22	042	&#34;	<b>"</b>		66	42	102	&#66;	<b>B</b>		98	62	142	&#98;	<b>b</b>
3	3	003	<b>ETX</b>	(end of text)	35	23	043	&#35;	<b>#</b>		67	43	103	&#67;	<b>C</b>		99	63	143	&#99;	<b>c</b>
4	4	004	<b>EOT</b>	(end of transmission)	36	24	044	&#36;	<b>\$</b>		68	44	104	&#68;	<b>D</b>		100	64	144	&#100;	<b>d</b>
5	5	005	<b>ENQ</b>	(enquiry)	37	25	045	&#37;	<b>%</b>		69	45	105	&#69;	<b>E</b>		101	65	145	&#101;	<b>e</b>
6	6	006	<b>ACK</b>	(acknowledge)	38	26	046	&#38;	<b>&amp;</b>		70	46	106	&#70;	<b>F</b>		102	66	146	&#102;	<b>f</b>
7	7	007	<b>BEL</b>	(bell)	39	27	047	&#39;	<b>'</b>		71	47	107	&#71;	<b>G</b>		103	67	147	&#103;	<b>g</b>
8	8	010	<b>BS</b>	(backspace)	40	28	050	&#40;	<b>(</b>		72	48	110	&#72;	<b>H</b>		104	68	150	&#104;	<b>h</b>
9	9	011	<b>TAB</b>	(horizontal tab)	41	29	051	&#41;	<b>)</b>		73	49	111	&#73;	<b>I</b>		105	69	151	&#105;	<b>i</b>
10	A	012	<b>LF</b>	(NL line feed, new line)	42	2A	052	&#42;	<b>*</b>		74	4A	112	&#74;	<b>J</b>		106	6A	152	&#106;	<b>j</b>
11	B	013	<b>VT</b>	(vertical tab)	43	2B	053	&#43;	<b>+</b>		75	4B	113	&#75;	<b>K</b>		107	6B	153	&#107;	<b>k</b>
12	C	014	<b>FF</b>	(NP form feed, new page)	44	2C	054	&#44;	<b>,</b>		76	4C	114	&#76;	<b>L</b>		108	6C	154	&#108;	<b>l</b>
13	D	015	<b>CR</b>	(carriage return)	45	2D	055	&#45;	<b>-</b>		77	4D	115	&#77;	<b>M</b>		109	6D	155	&#109;	<b>m</b>
14	E	016	<b>SO</b>	(shift out)	46	2E	056	&#46;	<b>.</b>		78	4E	116	&#78;	<b>N</b>		110	6E	156	&#110;	<b>n</b>
15	F	017	<b>SI</b>	(shift in)	47	2F	057	&#47;	<b>/</b>		79	4F	117	&#79;	<b>O</b>		111	6F	157	&#111;	<b>o</b>
16	10	020	<b>DLE</b>	(data link escape)	48	30	060	&#48;	<b>0</b>		80	50	120	&#80;	<b>P</b>		112	70	160	&#112;	<b>p</b>
17	11	021	<b>DC1</b>	(device control 1)	49	31	061	&#49;	<b>1</b>		81	51	121	&#81;	<b>Q</b>		113	71	161	&#113;	<b>q</b>
18	12	022	<b>DC2</b>	(device control 2)	50	32	062	&#50;	<b>2</b>		82	52	122	&#82;	<b>R</b>		114	72	162	&#114;	<b>r</b>
19	13	023	<b>DC3</b>	(device control 3)	51	33	063	&#51;	<b>3</b>		83	53	123	&#83;	<b>S</b>		115	73	163	&#115;	<b>s</b>
20	14	024	<b>DC4</b>	(device control 4)	52	34	064	&#52;	<b>4</b>		84	54	124	&#84;	<b>T</b>		116	74	164	&#116;	<b>t</b>
21	15	025	<b>NAK</b>	(negative acknowledge)	53	35	065	&#53;	<b>5</b>		85	55	125	&#85;	<b>U</b>		117	75	165	&#117;	<b>u</b>
22	16	026	<b>SYN</b>	(synchronous idle)	54	36	066	&#54;	<b>6</b>		86	56	126	&#86;	<b>V</b>		118	76	166	&#118;	<b>v</b>
23	17	027	<b>ETB</b>	(end of trans. block)	55	37	067	&#55;	<b>7</b>		87	57	127	&#87;	<b>W</b>		119	77	167	&#119;	<b>w</b>
24	18	030	<b>CAN</b>	(cancel)	56	38	070	&#56;	<b>8</b>		88	58	130	&#88;	<b>X</b>		120	78	170	&#120;	<b>x</b>
25	19	031	<b>EM</b>	(end of medium)	57	39	071	&#57;	<b>9</b>		89	59	131	&#89;	<b>Y</b>		121	79	171	&#121;	<b>y</b>
26	1A	032	<b>SUB</b>	(substitute)	58	3A	072	&#58;	<b>:</b>		90	5A	132	&#90;	<b>Z</b>		122	7A	172	&#122;	<b>z</b>
27	1B	033	<b>ESC</b>	(escape)	59	3B	073	&#59;	<b>;</b>		91	5B	133	&#91;	<b>[</b>		123	7B	173	&#123;	<b>{</b>
28	1C	034	<b>FS</b>	(file separator)	60	3C	074	&#60;	<b>&lt;</b>		92	5C	134	&#92;	<b>\</b>		124	7C	174	&#124;	<b> </b>
29	1D	035	<b>GS</b>	(group separator)	61	3D	075	&#61;	<b>=</b>		93	5D	135	&#93;	<b>]</b>		125	7D	175	&#125;	<b>}</b>
30	1E	036	<b>RS</b>	(record separator)	62	3E	076	&#62;	<b>&gt;</b>		94	5E	136	&#94;	<b>^</b>		126	7E	176	&#126;	<b>~</b>
31	1F	037	<b>US</b>	(unit separator)	63	3F	077	&#63;	<b>?</b>		95	5F	137	&#95;	<b>_</b>		127	7F	177	&#127;	<b>DEL</b>

Source: [www.LookupTables.com](http://www.LookupTables.com)

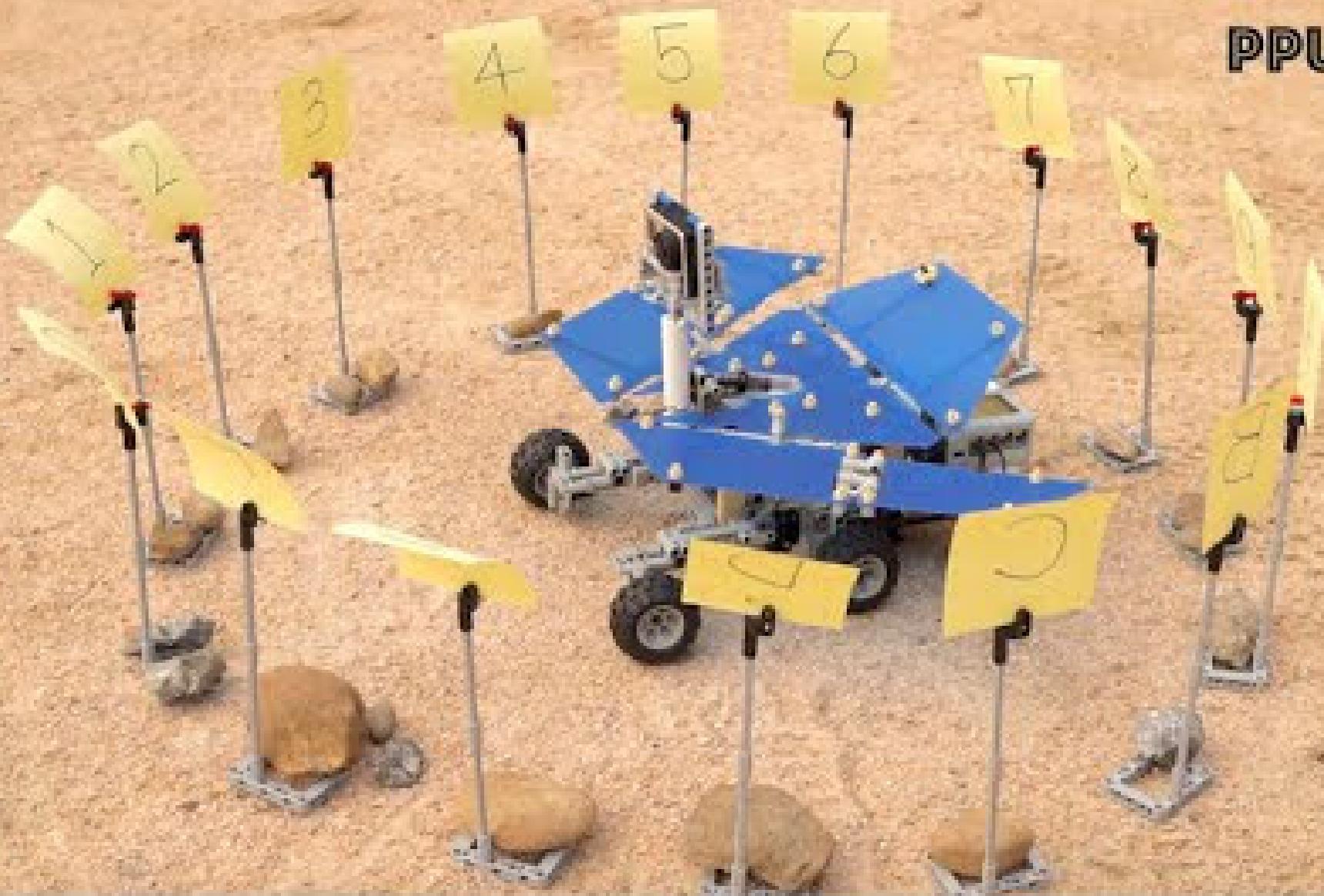
□ For example, lowercase j would become binary 1101010 (decimal 106) in ASCII.



54 68 65 20 45 6E 64

What do these hexadecimal numbers represent?

PPUNG<sup>TV</sup>



# Is ASCII Enough?

- What about other characters, which is not English characters?
  - The Unicode will be needed



א aleph	ׁ yud	ׂ ayin
ב bet	ׂ khaf	ׁ peh
ג gimmel	ׇ final haf	׆ final peh
ׁ daled	׆ lamed	ׂ tzadi
ׂ hey	ׁ mem	׆ final tzadi
ׁ vav	ׁ final mem	ׁ kuf
ׁ zayin	ׁ nun	ׁ resh
ׁ het	ׁ final nun	ׁ shin
ׁ tet	ׁ samekh	ׁ taf

অ	আ	ই	ঈ	উ	ঊ	এ	ঐ	ঔ	ঔ	ৰ
a	aa	i	ii	u	uu	e	ai	o	au	R
କ	ଖ	ଗ	ଘ	ଡ	ଚ	ଛୁ	ଜ	ଝ	ଝ	ଟଠ
k	Kh	g	G	q	c	ch	j	jh	jh	T
ଡ	ଦ	ଣ	ତ	ଥ	ଧ	ନ	ପ	ଫ	ବ	ଭମ
d	D	dw	Dw	t	D	n	p	f	b	m
ୟ	ର	ଲ	ବ	ଶ	ଷ	ସ	ହ			
y	r	l	v	sh	sh	s	h			

ASCII: 7-bit, 128 char --> C/C++ uses it for its primitive data type 'char' representation  
----- Unicode, Universal Coded Character Set, or UCS  
UCS-2: 16-bit code units, 65,536 char, insufficient, fixed-width > Java uses it for its primitive data type 'char'  
----- Unicode Transformation Format  
UTF-8: four 8-bit code units, extends ASCII, variable-width via code-points, WWW  
UTF-16: three 16-bit code units, extends UCS-2, variable-width, no ASCII, MS+Java

# Unicode

---

- **Unicode** is a computing industry standard for the consistent **encoding**, **representation**, and **handling** of text expressed in most of the world's writing systems.
- The latest version of Unicode contains a repertoire of more than 110,000 characters covering 100 scripts and multiple symbol sets.
  - As of June 2014, the most recent version is *Unicode 7.0*. The standard is maintained by the [Unicode Consortium](#).
- The most commonly used Unicode encodings are [UTF-8](#), [UTF-16](#) and the now-obsolete [UCS-2](#).
  - UTF-8 uses one byte for any ASCII character, all of which have the same code values in both UTF-8 and ASCII encoding, and up to four bytes for other characters.
  - UTF-16 extends UCS-2, using one 16-bit unit for the characters that were representable in UCS-2 and two 16-bit units ( $4 \times 8$  bit) to handle each of the additional characters.
  - UCS-2 uses a 16-bit code unit (two 8-bit bytes) for each character, but cannot encode every character in the current Unicode standard.

- [http://www.w3schools.com/charsets/ref\\_utf\\_misc\\_symbols.asp](http://www.w3schools.com/charsets/ref_utf_misc_symbols.asp)

# Chapter 2 Conclusion

---

- Computers store data in the form of bits, bytes, and words using the **binary numbering system**.
- **Hexadecimal** numbers are formed using four-bit groups called nibbles.
- Signed integers can be stored in one's complement, **two's complement**, or signed magnitude representation.
- Floating-point numbers are usually coded using the **IEEE 754 floating-point standard**.
- Floating-point operations are **not** necessarily commutative or distributive.
- **Character** data is stored using ASCII, EBCDIC, or Unicode.

---

**End of Chapter 2**

---

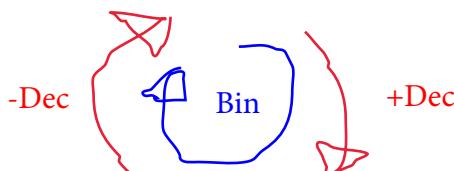
# Signed integers representation

## signed-binary

Eight-bit sign-magnitude		
Binary value	Sign-magnitude interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
:	:	:
01111101	125	125
01111110	126	126
01111111	127	127
10000000	-0	128
10000001	-1	129
10000010	-2	130
:	:	:
11111101	-125	253
11111110	-126	254
11111111	-127	255

- Two Zeros
- Range: -127 ~ +127
- FP's Significand uses this!

**-ve number:** Signed value =  $2^{n-1} - \text{Unsigned value}$   
 Signed value = 128 - Unsigned value



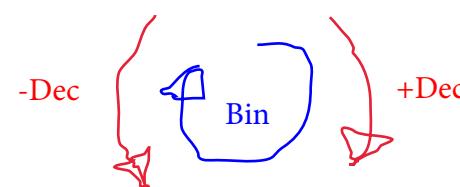
- \* - Addition and subtraction require different behavior depending on the sign bit in Signed Binary, whereas one's complement can ignore the sign bit and just do an end-around carry, and two's complement can ignore the sign bit and depend on the overflow behavior.
- Comparison also require inspecting the sign bit in Signed Binary and 1's complement, whereas in two's complement, one can simply subtract the two numbers, and check if the outcome is positive or negative.

## 1's complement

Eight-bit ones' complement		
Binary value	Ones' complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
:	:	:
01111101	125	125
01111110	126	126
01111111	127	127
10000000	-127	128
10000001	-126	129
10000010	-125	130
:	:	:
11111101	-2	253
11111110	-1	254
11111111	-0	255

- Two Zeros
- Range: -127 ~ +127

$$\begin{aligned}\text{Signed value} &= \text{Unsigned value} - 2^n + 1 \\ \text{Signed value} &= \text{Unsigned value} - 255\end{aligned}$$

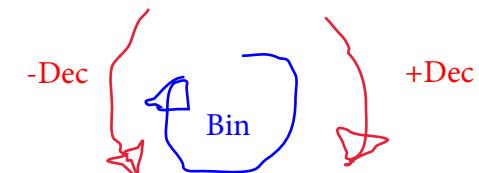


## 2's complement \*

Eight-bit two's complement		
Binary value	Two's complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
:	:	:
01111110	126	126
01111111	127	127
10000000	-128	128
10000001	-127	129
10000010	-126	130
:	:	:
11111110	-2	254
11111111	-1	255

- one Zero only
- Range: -128 ~ +127

$$\begin{aligned}\text{Signed value} &= \text{Unsigned value} - 2^n \\ \text{Signed value} &= \text{Unsigned value} - 256\end{aligned}$$



# Float-point representation

- one Zero only
- Range: -127 ~ +128

Signed Exponent value = Exponent Binary value - Bias/Offset

Signed Exponent value = Exponent Binary value - 127 (Single precision)

- Can choose any bias and thus can have any range of exponents, e.g. can eliminate -ve/+ve exponents altogether

FP models' Exponent use a Biased/excess-K/offset-Binary =  $2^{(E-1)} - 1$ , where E=number of Exponent bits FP models

use Signed-Binary for Significand ==> two Zeros

Simple (hypothetical) Model (14-bit): sign bit + 5 exponent bit + 8 Significand bits (no bit implied, 0.1xxx xxxx)

Single precision Model (32-bit): sign bit + 8 exponent bit + 23 Significand bits (one bit implied, 1.xxxxx xxxx)



Name	Common name	Base	Significand bits <sup>[b]</sup> or digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias <sup>[14]</sup>	E min	E max	Notes
binary16	Half precision	2	11	3.31	5	4.51	$2^4 - 1 = 15$	-14	+15	not basic
binary32	Single precision	2	24	7.22	8	38.23	$2^7 - 1 = 127$	-126	+127	
binary64	Double precision	2	53	15.95	11	307.95	$2^{10} - 1 = 1023$	-1022	+1023	
binary128	Quadruple precision	2	113	34.02	15	4931.77	$2^{14} - 1 = 16383$	-16382	+16383	
binary256	Octuple precision	2	237	71.34	19	78913.2	$2^{18} - 1 = 262143$	-262142	+262143	not basic
decimal32		10	7	7	7.58	96	101	-95	+96	not basic
decimal64		10	16	16	9.58	384	398	-383	+384	
decimal128		10	34	34	13.58	6144	6176	-6143	+6144	

Significand includes sign bit as MSb (left-most bit)

includes +/- inf & NaN and zero!

- "inf's" are represented with highest Exponent value for a precision, and with all zero Significand bits.
- "NaN" (error condition) is represented with highest Exponent value for a precision, and with at least one non-zero Significand bit.
- "zero" is represented with lowest Exponent value for a precision.