# Lesson 2: Binary Systems

Jiun-Long Huang
Department of Computer Science
National Chiao Tung University

# Outline

- Unsigned Integers
- Signed Integers
- Floating-Point Numbers

# Unsigned Integers

# Binary Numbers

- Bit
  - Can be 0 or 1
- Why binary (base 2) numbers?
  - Why decimal (base 10) numbers?

# Binary Numbers (cont'd)

- Logic gate
  - ON (1) and OFF (0)
- Binary (Base 2)→Octal (Base 8) →Hexadecimal (Base 16)
- Octal
  - 0 1 2 3 4 5 6 7
- Hexadecimal
  - 0 1 2 3 4 5 6 7 8 9 A B C D E F

# Binary Numbers (cont'd)

- Hour/Minute/Second can be seen as base 60
  - HH:MM:SS
- 02:03:04→7384 seconds

  $2*60^2+3*60^1+4*60^0$

  $=7200+180+4$

  $=7384$

# Binary Numbers (cont'd)

- 128 seconds→00:02:08

  128/60=2…8

  2/60=0…2

- 3800 seconds→01:03:20

  3800/60=63…20

  63/60=1…3

# Binary Numbers (cont'd)

- Decimal: 19
  - Binary: 10011
  - Octal: 23
  - Hexadecimal:13
- Transform a number in base N to the equivalent one in base 10

$(19)_{10}=1*10^1+9*10^0$

$(10011)_2=1*2^4+1*2^1+1*2^0$

$=16+2+1$

$(23)_8=2*8^1+3*8^0$

$=16+3$

$(13)_{16}=1*16^1+3*16^0$

$=16+3$

# Binary Numbers (cont'd)

- Transform a number in base 10 to the equivalent one in base N
  - $(1000)_{10} \rightarrow (??)_{16}$

    $1000/16=62...8$

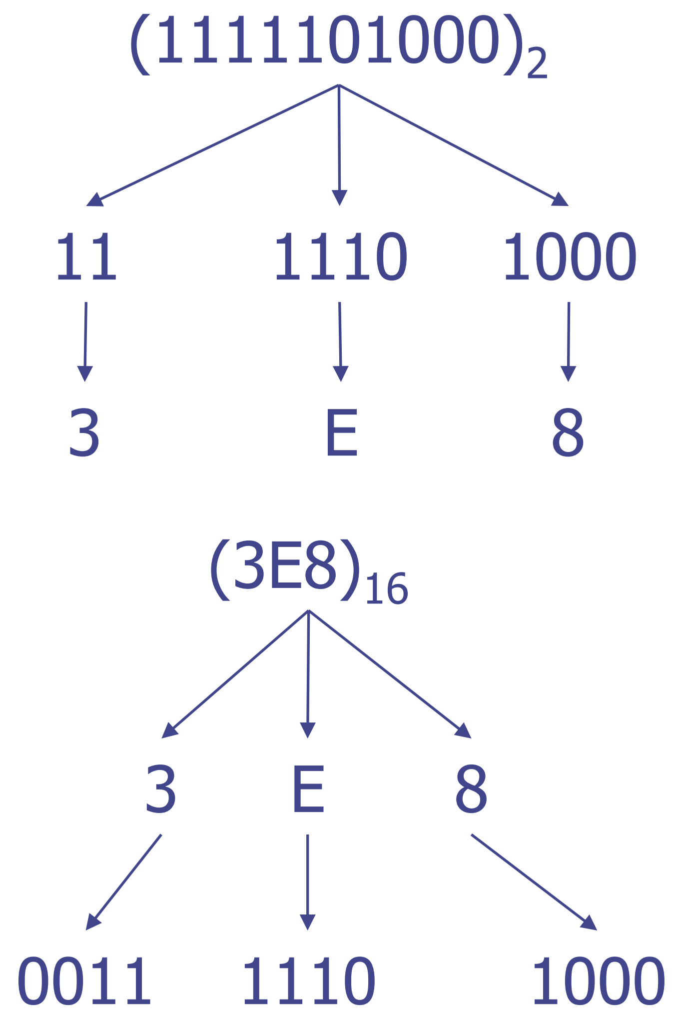
    $62/16=3...(14)_{10}...E$

    $3/16=0...(3)_{10}...3$

    $(1000)_{10}=(3E8)_{16}$

# Binary Numbers (cont'd)

- Transformation between binary and hexadecimal numbers
  - A hexadecimal digit is equivalent to four binary digits
- How about octal numbers?



# Addition and Multiplication in Binary Numbers

- $(1011)_2+(0110)_2=$
  - $(10001)_2$
- $(1011)_2*(0110)_2=$
  - $(1000010)_2$

```
111

 1011
+0110
-----
10001
```

```
    1011
   *0110
--------
    0000
   1011
  1011
+0000
--------
  100010
```

# Overflow

- Overflow
  - Reason: Computers only use limited number of bits to represent numbers
  - The value is out of the range
- Consider 4-bit integers
  - $15+1 \rightarrow (1111)_2+(0001)_2=(0000)_2=0$
  - $0-1 \rightarrow (0000)_2-(0001)_2 \rightarrow (1111)_2 \rightarrow 15$

# Ranges of Unsigned Integers

- 8-bit unsigned integers
  - $(11111111)_2 \sim (00000000)_2$
  - $2^8-1 \sim 0 \rightarrow 255 \sim 0$
- 16-bit unsigned integers
  - $2^{16}-1 \sim 0 \rightarrow 65535 \sim 0$
- 32-bit unsigned integers
  - $2^{32}-1 \sim 0$
- 64-bit unsigned integers
  - $2^{64}-1 \sim 0$

# Signed Integers

# Signed Numbers

- Four-best known representations for signed numbers
  - Signed and magnitude
  - One's complement
  - Two's complement
  - Offset binary (also known as Excess-K)

*https://en.wikipedia.org/wiki/Signed_number_representations*

# Signed and Magnitude

- Consider 4-bit integers
- Using a sign bit to represent positive and negative numbers
  - +5→0101
  - -5→1101

```
   5+2

  0101
 +0010
------
  0111

5+(-2)

  0101
 +1010
------
  0011
```

# Signed and Magnitude

- 0→0000(+0) or 1000(-0)
  - 0 has two representations
- The procedures of adding a positive number and adding a negative number are different
  - Complicated circuits
- Range: +7~+0,-0~-7

# Two's Complement

- 2's complement
  - +5=0101
  - -5=1011
  - The leftmost bit is 1 <-> negative number
  - Range:
    - 7~-8

+5

0101

1<->0

1010

plus 1

1011

-5

1000

0111

1000

# Two's Complement

- There is exactly one 0
- The procedures of adding positive and negative numbers are the same
  - Simple circuits

1101→is a negative number

↓ minus 1

1100

↓ 0<->1

0011→3

```
   5-2=       2-5=
5+(-2)     2+(-5)

  0101       0010
 +1110      +1011
------     ------
  0011       1101
```

Thus, the decimal value of 1101 is -3

# Overflow

- Overflow
  - Computers only use limited number of bits to represent numbers
  - The value is out of the range
- E.g.,
  - $7+1 \rightarrow (0111)_2+(0001)_2=(1000)_2=-8$
  - $-8-1 \rightarrow -8+(-1) \rightarrow$
  - $(1000)_2+(1111)_2=(0111)_2=7$

# Ranges of Signed Integers

- 8-bit integers
  - $(01111111)_2 \sim (10000000)_2$
  - $2^7-1 \sim -2^7 \rightarrow 127 \sim -128$
- 16-bit integers
  - $2^{15}-1 \sim -2^{15} \rightarrow 32767 \sim -32768$
- 32-bit integers
  - $2^{31}-1 \sim -2^{31}$
- 64-bit integers
  - $2^{63}-1 \sim -2^{63}$

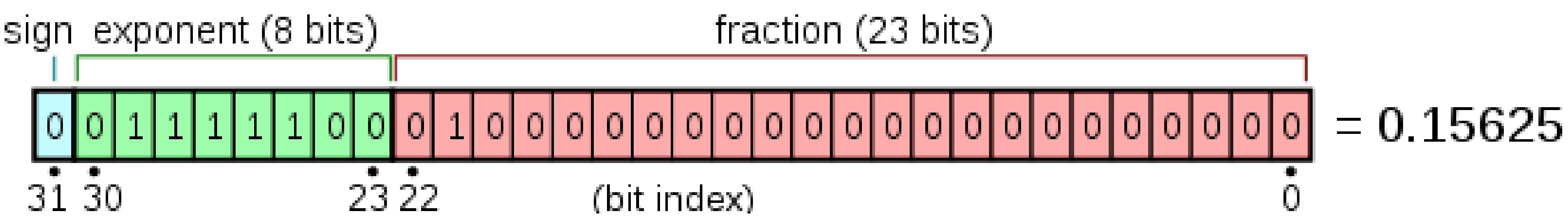
# Floating-Point Numbers

# IEEE 754-2008: Single-Precision (32 Bits) Floating-Point Numbers

- The IEEE 754 standard specifies a binary32 (single-precision floating-point numbers) as having:
  - Sign bit: 1 bit
  - Exponent width: 8 bits
  - Significand precision: 24 bits (23 explicitly stored)
    - Also known as fraction and mantissa

# IEEE 754-2008: Double-Precision (64 Bits) Floating-Point Numbers

- The number has value $v = s \times m \times 2^{e}$, where
  - $s = +1$ (positive numbers) when the sign bit is 0
  - $s = -1$ (negative numbers) when the sign bit is 1
  - $e = \text{Exp} - 127$ (excess-127 representation)
  - $m = 1.\text{fraction}$ in binary (normalized)




*https://en.wikipedia.org/wiki/Single-precision_floating-point_format*

# IEEE 754-2008: Single-Precision (32 Bits) Floating-Point Numbers

- The sign bit is zero
- The exponent is $-3$
- The significand is 1.01 (in binary), which is 1.25 in decimal.

  $(1.01)_2=1*2^0+0*2^{-1}+1*2^{-2}$

- The represented number is therefore $+1.25 \times 2^{-3}$, which is +0.15625.

# IEEE 754-2008: Single-Precision (32 Bits) Floating-Point Numbers

- Show IEEE 754 form of -5
- $(-5)_{10}=(-1.25*2^2)_{10}=(-1.01*2^2)_2$
- s bit: 1
- Exp=127+2=129

$2^{-1}=0.5$ $2^{-2}=0.25$

$\underline{1}\ \underline{10000001}\ \underline{01000000000000000000000}$

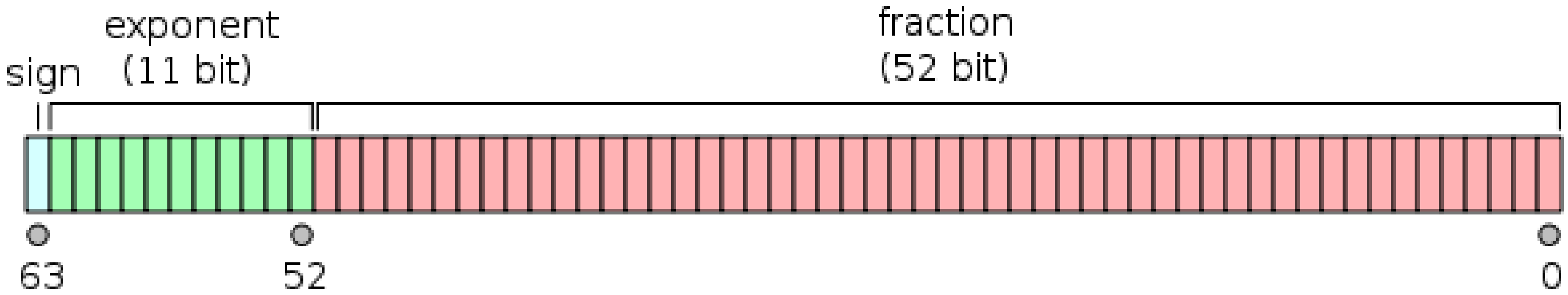
# IEEE 754-2008: Double-Precision (64 Bits) Floating-Point Numbers

- The IEEE 754 standard specifies a binary64 as having:
  - Sign bit: 1 bit
  - Exponent: 11 bits
  - Significand precision: 53 bits (52 explicitly stored)

*https://en.wikipedia.org/wiki/Double-precision_floating-point_format*

# IEEE 754-2008: Double-Precision (64 Bits) Floating-Point Numbers

- The number has value $v = s \times 2^e \times m$, where
  - $s = +1$ (positive numbers) when the sign bit is 0
  - $s = -1$ (negative numbers) when the sign bit is 1
  - $e = \text{Exp} - 1023$ (excess-1023 representation)
  - $m = 1.$fraction in binary (normalized)



# Round-off Error

- Since single/double precision floating-point numbers use finite bits, overflow may occur.
  - The significand is of finite bits, and thus, round-error may occur.
- Round-off error (the consequence of using finite precision floating point numbers on computers) is also called truncation error.

*https://en.wikipedia.org/wiki/Round-off_error*

- Consider storing the following number by single precision floating-point number

$(+1.\underline{00000000000000000000001}1 * 2^{0})_2$

1 01111111 00000000000000000000001

# Underflow

- Arithmetic underflow occurs when the true result of a floating point operation is smaller in magnitude (that is, closer to zero) than the smallest value representable as a normal floating point number in the target datatype.
  - E.g., $2^{-20000}$ in double-precision floating number representation

*https://en.wikipedia.org/wiki/Arithmetic_underflow*

# Discussion

- Is it a good idea to use 32-bit floating-point number to replace 32-bit signed integer?
  - Floating-point number calculation is much slower
  - Floating-point number may cause truncation error
    - It is a critical problem in some domains such as finance