
System I

Information Representation

Haifeng Liu

Zhejiang University

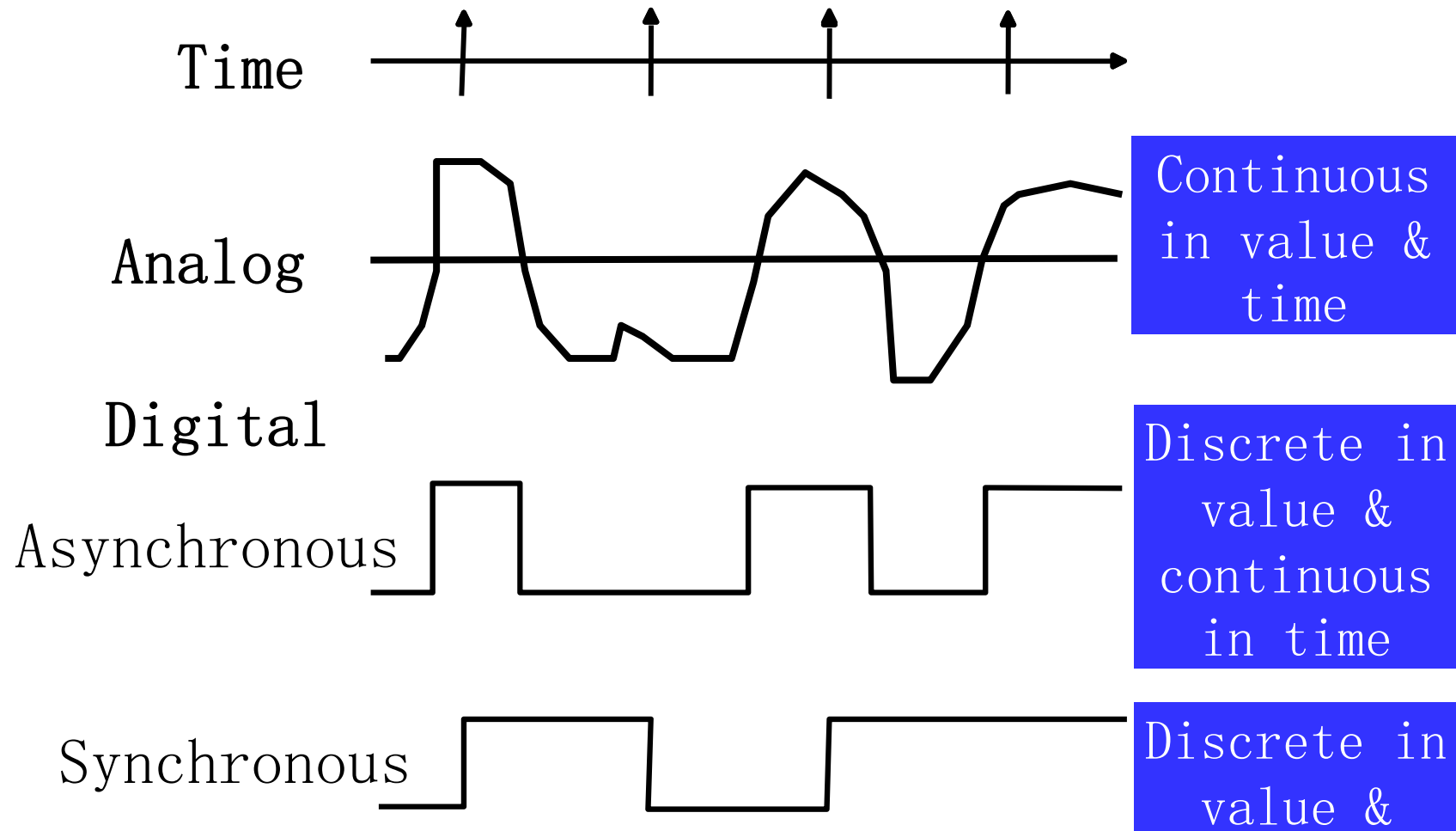
Overview

- **Binary number representation**
- Representation of numeric data
- Representation of non-numeric data
- Data width and storage

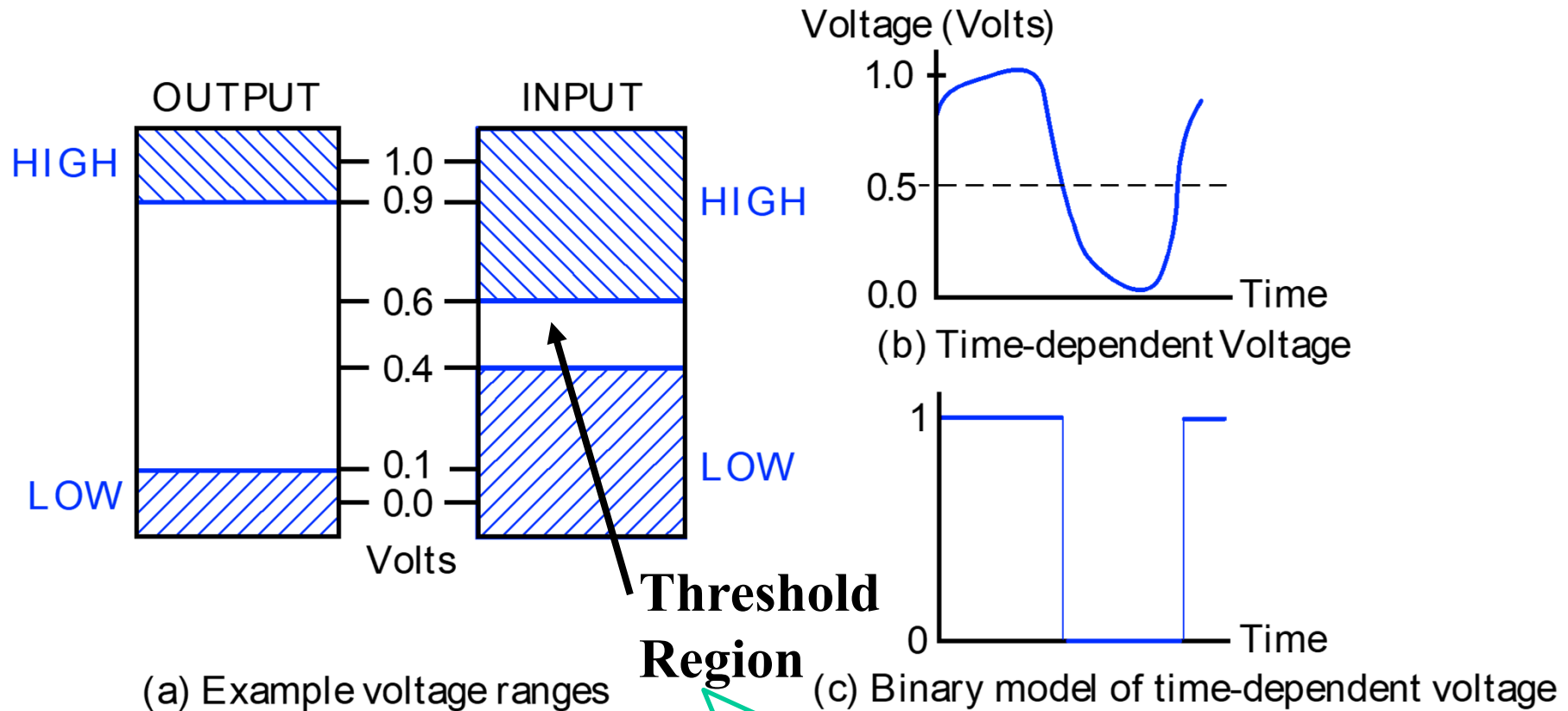
INFORMATION REPRESENTATION - Signals

- **Information variables represented by physical quantities.**
- **For digital systems, the variables take on discrete values.**
- **Two level, or binary values are the most prevalent values in digital systems.**
- **Binary values are represented abstractly by:**
 - **digits 0 and 1**
 - **words (symbols) False (F) and True (T)**
 - **words (symbols) Low (L) and High (H)**
 - **and words On and Off.**
- **Binary values are represented by values or ranges of values of physical quantities**

Signal Examples Over Time



Signal Example – Physical Quantity: Voltage



Why binary is most prevalent in digital systems?

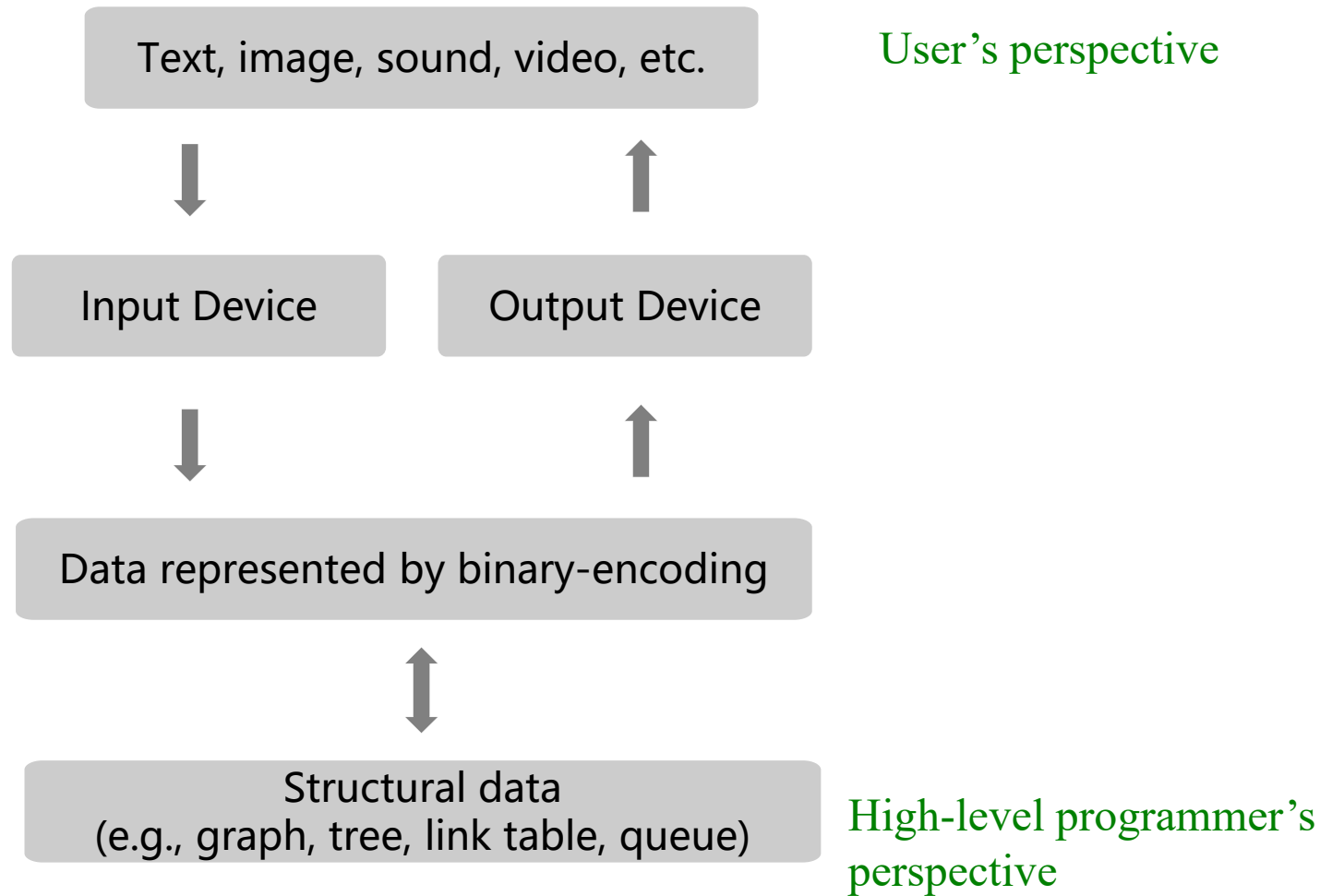
Binary Values: Other Physical Quantities

- **Other physical quantities that can be used to represent 0 and 1**
 - **CPU** Voltage
 - **Disk** Magnetic Field Direction
 - **CD** Surface Pits/Light
 - **DRAM** Electrical Charge

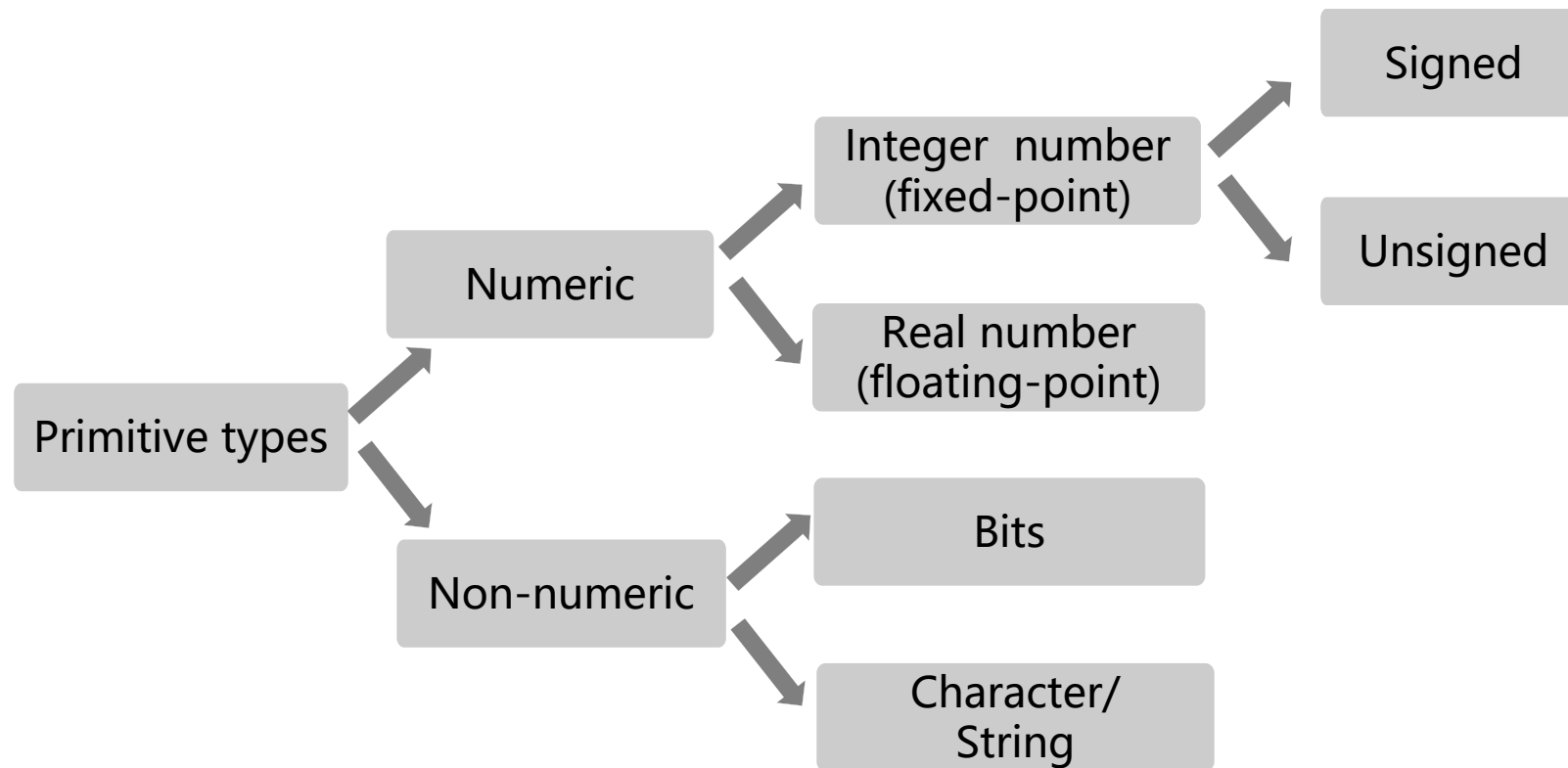
External information and internal data

- **Continuous vs. discrete**
- **Advanced types vs. primitive types**

External Information



Internal Data



Low-level programmer's/
hardware system designer's
perspective

Carry Counting System

- Positive radix, positional number systems
- A number with *radix* r is represented by a string of digits:

$A_{n-1}A_{n-2} \dots A_1A_0 \cdot A_{-1}A_{-2} \dots A_{-m+1}A_{-m}$
in which $0 \leq A_i < r$ and \cdot is the *radix point*.

- The string of digits represents the **power series**:

$$\begin{aligned} (\text{Number})_r = & \left(\sum_{i=0}^{n-1} A_i \cdot r^i \right) + \left(\sum_{j=-m}^{-1} A_j \cdot r^j \right) \\ & \text{(Integer Portion)} + \text{(Fraction Portion)} \end{aligned}$$

About the Decimal Number

123.456

$$1*10^2+2*10^1+3*10^0+4*10^{-1}+5*10^{-2}+6*10^{-3}$$

What About the Binary Number?

$$(100101.01)_2$$

$$1*2^5+0*2^4+0*2^3+1*2^2+0*2^1+1*2^0+0*2^{-1}+1*2^{-2}$$

Different Radixes

Name	Radix	Digits
Binary	2	0,1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Different Radixes

Decimal (Base 10)	Binary (Base 2)	Octal (Base 8)	Hexadecimal (Base 16)
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10

Converting R-base to Decimal

- To convert to decimal, use decimal arithmetic to form Σ (digit \times respective power of R).
- Convert $(100101.01)_2$ to N_{10} :
 - $1*2^5+0*2^4+0*2^3+1*2^2+0*2^1+1*2^0+0*2^{-1}+1*2^{-2}=(37.25)_{10}$
- Convert $3A.CH$ to N_{10} :
 - $3*16^1+10*16^0+12*16^{-1}=(58.75)_{10}$

Converting Decimal to R-base

- **To convert from one base to another:**
 - **Convert the integer part**
 - **Convert the fraction part**
 - **Join the two results together**


Convert the Integral Part

- Repeatedly divide the number by the new radix and save the remainders. The digits for the new radix are the remainders in *reverse order* of their computation.
 - If the new radix is > 10 , then convert all remainders > 10 to digits A, B, ...

Example: Convert 135_{10} To Base 2

$$(135)_{10} = (1000\ 0111)_2$$

2		1	3	5	
2		6	7	1
2		3	3	1
2		1	6	1
		2	8	0
		2	4	0
		2	2	0
		2	1	0
		2	0	1



Convert the Fractional Part

- Repeatedly multiply the fraction by the new radix and save the integer digits that result. The digits for the new radix are the integer digits in *order* of their computation.
 - If the new radix is > 10 , then convert all remainders > 10 to digits A, B, ...

Example: Convert 0.6875_{10} To Base 2

$$(0.6875)_{10} = (0.1011)_2$$

$$\begin{array}{l} \underline{2 \times 0.6875 \dots\dots\dots} = 1.375 \\ \underline{2 \times 0.375 \dots\dots\dots} = 0.75 \\ \underline{2 \times 0.75 \dots\dots\dots} = 1.5 \\ 2 \times 0.5 \dots\dots\dots = 1 \end{array}$$



Convert 135.6875_{10} To Base 2

- Convert 135 to Base 2

$$(135)_{10} = (10000111)_2$$

- Convert 0.6875 to Base 2:

$$(0.6875)_{10} = (0.1011)_2$$

- Join the results together

$$(135.6875)_{10} = (10000111.1011)_2$$

Additional Issue - Fractional Part

- Note that in this conversion, the fractional part can become 0 as a result of the repeated multiplications.
- In general, it may take many bits to get this to happen or it may never happen.
- Example Problem: Convert 0.65_{10} to N_2
 - $0.65 = 0.1010011001001 \dots$
 - The fractional part begins repeating every 4 steps yielding repeating 1001 forever!
- Solution: Specify number of bits to right of radix point and round or truncate to this number.

Why Do Repeated Division and Multiplication Work?

- Divide the integer portion of the power series by radix r . The remainder of this division is A_0 , represented by the term A_0/r .
- Discard the remainder and repeat, obtaining remainders A_1, \dots
- Multiply the fractional portion of the power series by radix r . The integer part of the product is A_{-1} .
- Discard the integer part and repeat, obtaining integer parts A_{-2}, \dots
- This demonstrates the algorithm for any radix $r > 1$.

Octal (Hexadecimal) to Binary and Back

- **Octal (Hexadecimal) to Binary:**
 - **Restate the octal (hexadecimal) as three (four) binary digits starting at the radix point and going both ways.**
- **Binary to Octal (Hexadecimal):**
 - **Group the binary digits into three (four) bit groups starting at the radix point and going both ways, padding with zeros as needed in the fractional part.**
 - **Convert each group of three bits to an octal (hexadecimal) digit.**

Hexadecimal to Binary and Back

(0)H = 0000

(1)H = 0001

(2)H = 0010

(3)H = 0011

(4)H = 0100

(5)H = 0101

(6)H = 0110

(7)H = 0111

(8)H = 1000

(9)H = 1001

(A)H = 1010

(B)H = 1011

(C)H = 1100

(D)H = 1101

(E)H = 1110

(F)H = 1111

Some Examples

$$(67.731)_8 = (110\ 111\ .111\ 011\ 001)_2$$

$$(312.64)_8 = (011\ 001\ 010\ .\ 110\ 1)_2$$

$$(11\ 111\ 101\ .\ 010\ 011\ 11)_2 = (375.236)_8$$

$$(10\ 110.11)_2 = (26.6)_8$$

$$(2B.5E)_{16} = (0010\ 1011.0101\ 1110)_2$$

$$(21A.5)_{16} = (0010\ 0001\ 1010\ .\ 0101)_2$$

$$(1001101.01101)_2 = (0100\ 1101.0110\ 1000)_2 = (4D.68)_{16}$$

$$(110\ 0101.101)_2 = (65.A)_{16}$$

Octal to Hexadecimal via Binary

- Convert octal to binary.
- Use groups of four bits and convert as above to hexadecimal digits.
- Example: Octal to Binary to Hexadecimal

6 3 5 . 1 7 7₈

➡ 110 | 011 | 101 . 001 | 111 | 111₂

➡ 1 | 1001 | 1101 . 0011 | 1111 | 1(000)₂

➡ 1 9 D . 3 F 8₁₆

Why do these conversions work?

Simple BASE CONVERSION(From Decimal to Binary) - Positive Powers of 2

- Useful for Base Conversion

Exponent	Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Exponent	Value
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536
17	131,072
18	262,144
19	524,288
20	1,048,576
21	2,097,152

Special Powers of 2

- 2^{10} (1024) is Kilo, denoted "K"
- 2^{20} (1,048,576) is Mega, denoted "M"
- 2^{30} (1,073, 741,824)is Giga, denoted "G"
- 2^{40} (1,099,511,627,776) is Tera, denoted "T"

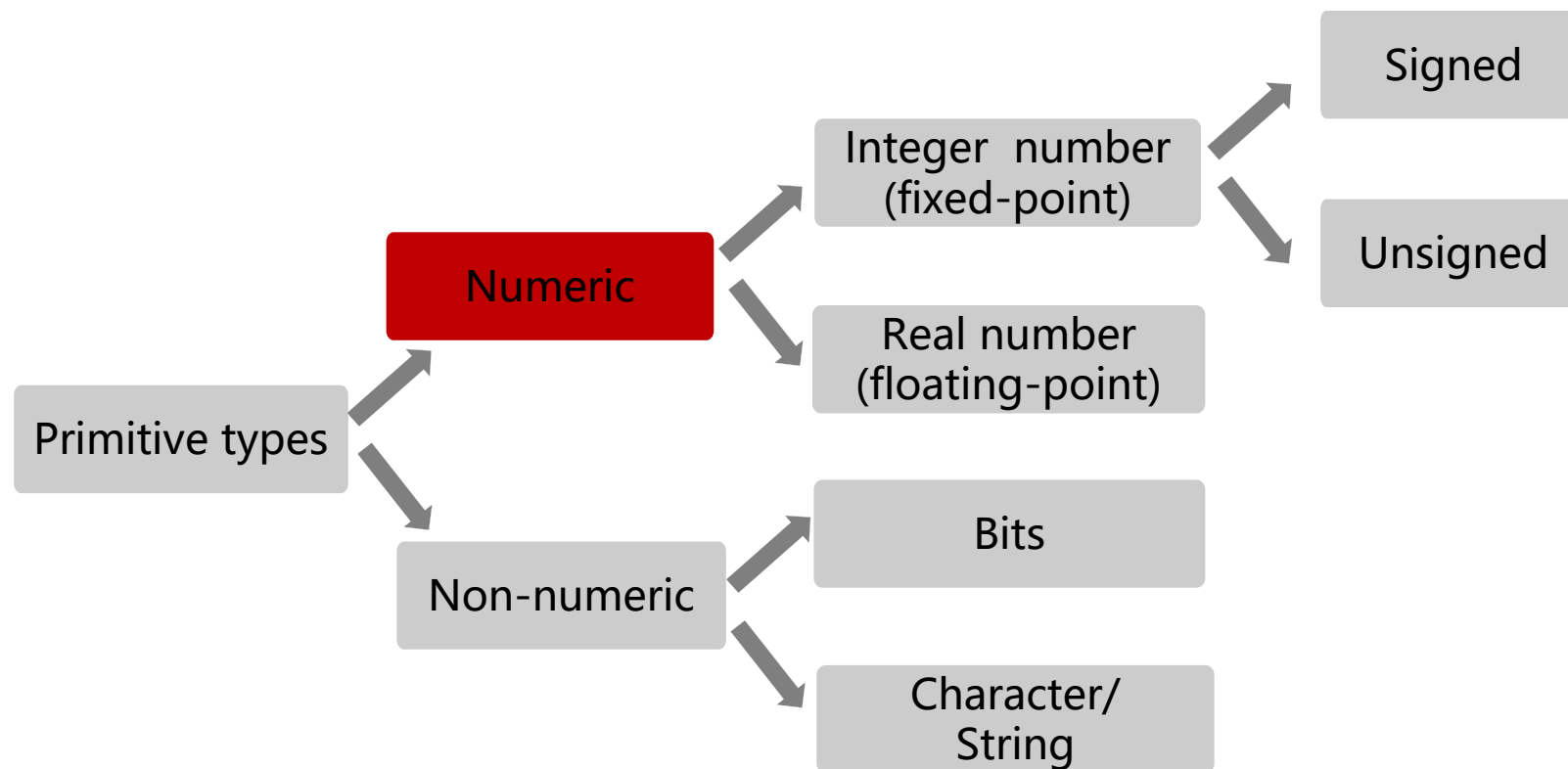
2^X vs. 10^Y Bytes Ambiguity

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

Overview

- Binary number representation
- **Representation of numeric data**
- Representation of non-numeric data
- Data width and storage

Numeric Data



Low-level programmer's/
hardware system designer's
perspective

Elements to Represent Numerical Data

- Carry counting system
- Number representation
 - Fixed-point vs. floating-point
- Encoding method

Representation of Numeric Data

- Fixed-point number representation
- Representation of integers
- Floating-point number representation
- Binary codes for decimal digits

Representation of Numeric Data

- Fixed-point number representation
- Representation of integers
- Floating-point number representation
- Binary codes for decimal digits

Fixed-point Number Representation

- The number represented by the computer's internal code is called **machine number**, and the corresponding value is called **true value**.
- After encoding X_T with n-bit binary numbers, the machine number X can be expressed as:

$$X = X_{n-1}X_{n-2} \cdots X_1X_0$$

- Unsigned integer
 - $0 \leq X \leq 2^n - 1$
- Signed integer
 - Depend on the encoding method

Encoding Methods

- sign-and-magnitude
- Complements
- Inverse code
- Frameshift code

Signed and Unsigned Numbers Possible Representations

- Sign Magnitude One's Complement Two's Complement

000 = +0

001 = +1

010 = +2

011 = +3

100 = -0

101 = -1

110 = -2

111 = -3

000 = +0

001 = +1

010 = +2

011 = +3

100 = -3

101 = -2

110 = -1

111 = -0

000 = +0

001 = +1

010 = +2

011 = +3

100 = -4

101 = -3

110 = -2

111 = -1

- Which one is best? Why?

- Issues: number of zeros, ease of operations

sign-and-magnitude (原码)

- A positive number and the corresponding negative number share the same value part
 - The conversion between the original code and true value is straightforward, i.e., just alter the sign bit
- **The coding rules are as follows:**
 - $X_T > 0, X_{n-1} = 0, X_i = X_i' (0 \leq i \leq n-2)$
 - $X_T < 0, X_{n-1} = 1, X_i = X_i' (0 \leq i \leq n-2)$

There are two ways to represent 0!

Complements

- Two complements:
 - Diminished Radix Complement of N
 - $(r - 1)$'s complement for radix r
 - 1's complement for radix 2
 - Defined as $(r^n - 1) - N$
 - Radix Complement
 - r 's complement for radix r
 - 2's complement in binary
 - Defined as $r^n - N$

Binary 1's Complement(反码)

- For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits):

$$(r^n - 1) = 256 - 1 = 255_{10} \text{ or } 11111111_2$$

- The 1's complement of 01110011_2 is then:

$$\begin{array}{r} 11111111 \\ - 01110011 \\ \hline 10001100 \end{array}$$

- Since the $2^n - 1$ factor consists of all 1's and since $1 - 0 = 1$ and $1 - 1 = 0$, the one's complement is obtained by complementing each individual bit (bitwise NOT).

Binary 2's Complement(补码)

- For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits), we have:

$$(r^n) = 256_{10} \text{ or } 100000000_2$$

- The 2's complement of 01110011 is then:

$$\begin{array}{r} 100000000 \\ - 01110011 \\ \hline 10001101 \end{array}$$

- Note the result is the 1's complement plus 1, a fact that can be used in designing hardware

Alternate 2's Complement Method

- Given: an n -bit binary number, beginning at the least significant bit and proceeding upward:
 - Copy all least significant 0's
 - Copy the first 1
 - Complement all bits thereafter.

- 2's Complement Example:

10010100

- Copy underlined bits:

100

- and complement bits to the left:

01101100

More common: use of 2's complement

---- negatives have one additional number

$$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000)_2 = (0)_{10}$$

$$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 = (1)_{10}$$

.....

.....

$$(0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101)_2 = (2,\ 147,\ 483,\ 645)_{10}$$

$$(0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110)_2 = (2,\ 147,\ 483,\ 646)_{10}$$

$$(0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111)_2 = (2,\ 147,\ 483,\ 647)_{10}$$

$$(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000)_2 = (-2,\ 147,\ 483,\ 648)_{10}$$

$$(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 = (-2,\ 147,\ 483,\ 647)_{10}$$

$$(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010)_2 = (-2,\ 147,\ 483,\ 646)_{10}$$

.....

.....

$$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101)_2 = (-3)_{10}$$

$$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110)_2 = (-2)_{10}$$

$$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111)_2 = (-1)_{10}$$

Two's Biased notation

■ Negating Biased notation number:

invert all bits and add 1 with end

Defining : Assume: $X = \pm x_{(n-1)}x_{(n-2)}x_{(n-3)}x_{(n-4)}\dots x_0$

$$[X]_b = 2^n + X \quad -2^n \leq X \leq 2^n$$

$$[0]_b = 10000\dots(2^n)$$

$X = +1011$ $[X]_b = 11011$

sign bit "1" Positive

$X = -1011$ $[X]_b = 00101$

sign bit "0" Negative

2's Biased notation VS 2's complement

- Only reverse sign bit

e.g.

$X = +1011$ $[X]_c = 01011$ $[X]_b = 11011$

$X = -1011$ $[X]_c = 10101$ $[X]_b = 00101$

biase

$$\text{IEEE 754: } [X]_b = 2^n + X$$

Frameshift Code (移码)

- Be used to represent the **exponent** part of a floating-point number
- A floating-point number is represented by two fixed-point numbers with a sign bit
 - A fixed-point decimal is used to represent the significand of the floating-point number
 - A fixed-point integer is used to represent the order of power of two of the floating-point number



- Will discuss it later

sign extension

- Expansion
 - e.g. 8 bit numbers to 64/32 bit numbers
- Required for operations with registers(32/64 bits) and immediate operands (8 bits)
- Sign extension
 - Take the lower 8 bits as they are
 - Copy the highest bit to the remaining 24/56 bits
 - 0000 0010 → 2
0000 0000 0000 0000 0000 0000 0000 0010
 - 1111 1110 → -2
1111 1111 1111 1111 1111 1111 1111 1110

Sign Extension Example

```
short int x = 15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
 - Unsigned: zeros added
 - Signed: sign extension
 - Both yield expected result

- Truncating (e.g., unsigned to unsigned short)
 - Unsigned/signed: bits are truncated
 - Result reinterpreted
 - Unsigned: mod operation
 - Signed: similar to mod
 - For small numbers yields expected behavior

Negation: Complement & Increment

- Claim: Following Holds for 2's Complement

$$\sim x + 1 == -x$$

- Complement

- Observation: $\sim x + x == 1111\dots111 == -1$

x								
	1	0	0	1	1	1	0	1
+								
~x								
	0	1	1	0	0	0	1	0
	<hr/>							
-1								
	1	1	1	1	1	1	1	1

- Complete Proof?

Complement & Increment Examples

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011
y	-15213	C4 93	11000100 10010011

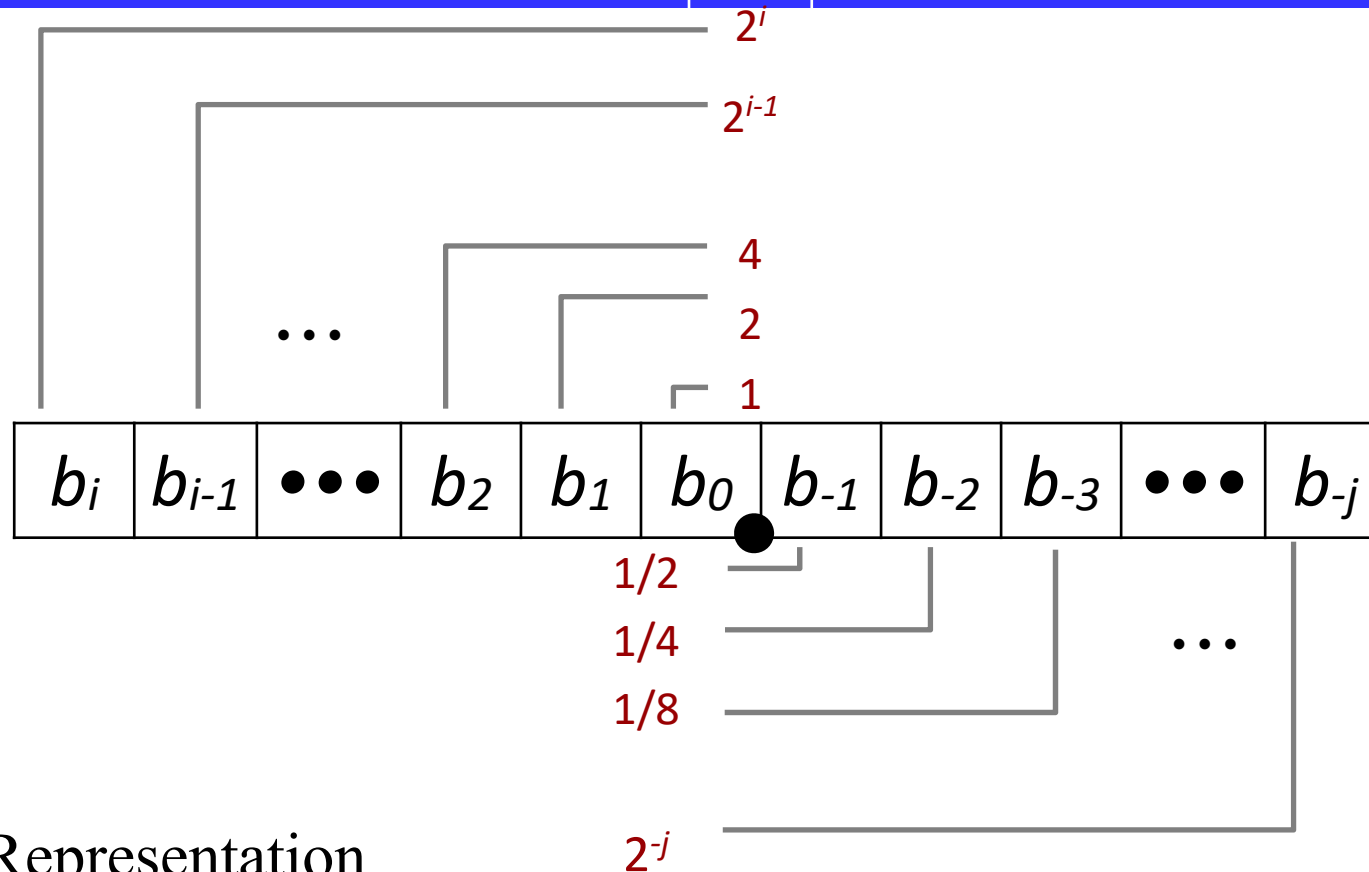
x = 0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

Representation of Numeric Data

- Fixed-point number representation
- Representation of integers
- Floating-point number representation
- Binary codes for decimal digits

Reflect on Fractional Binary Numbers



■ Representation

- Bits to right of “binary point” represent fractional powers of 2

- Represents rational number:
$$\sum_{k=-j}^i b_k \times 2^k$$

Limitation of Fractional Binary Numbers

- Approximate representation
 - Can only exactly represent numbers of the form $x/2^k$
 - Other rational numbers have repeating bit representations
 - Value Representation
 - 1/3 0.0101010101[01]...₂
 - 1/5 0.001100110011[0011]...₂
 - 1/10 0.0001100110011[0011]...₂
- Implementation issues
 - Limited bit width
 - How to determine the binary point?

Floating point numbers

- Reasoning
 - Larger number range than integer range
 - Fractions
 - Numbers like e (2.71828) and π (3.14159265....)
- Representation for non-integral numbers
 - Including very small and very large numbers
- Like scientific notation
 - -2.34×10^{56} ← normalized
 - $+0.002 \times 10^{-4}$ ← not normalized
 - $+987.02 \times 10^9$ ← not normalized

Floating-point Number Representation

- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Floating Point

- In binary
 - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types float and double in C
- Representation
 - Sign
 - Fraction
 - Exponent
 - More bits for fraction: more accuracy
 - More bits for exponent: increases the range

IEEE Floating Point

- IEEE Standard 754
 - Designed by William Kahan since 1976
 - Established in 1985 as uniform standard for floating point arithmetic
 - Before that, many idiosyncratic formats
 - Supported by all major CPUs

- Driven by numerical concerns
 - Nice standards for rounding, overflow, underflow
 - Hard to make fast in hardware
 - Numerical analysts predominated over hardware designers in defining standard



Floating Point Representation

- Numerical Form:

$$(-1)^s M 2^E$$

- **Sign bit** s determines whether number is negative or positive
- **Significand** M normally a fractional value in range $[1.0, 2.0)$.
- **Exponent** E weights value by power of two



$$x = (-1)^s \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 \Rightarrow non-negative, 1 \Rightarrow negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
 - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
 - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
 - Ensures exponent is unsigned
 - Single: Bias = 127; Double: Bias = 1203

Precisions

- Single precision: 32 bits



- Double precision: 64 bits



- Extended precision: 80 bits (Intel only)



- Represent -0.75

- ## Single precision

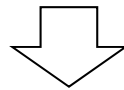
1 bit 8 bits 23 bits

- 1bit 11 bits 20 bits

[illegible]

Converting Binary to Decimal Floating Point

31	30 23	22 0
1	1000 0001	010 0000 0000 0000 0000 0000
1 bit	8 bits	23 bits



$$\begin{aligned} (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0 \end{aligned}$$

Single-Precision Range

- **Exponents 00000000 and 11111111 reserved**
- **Smallest value**
 - Exponent: 00000001
 \Rightarrow actual exponent = $1 - 127 = -126$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Largest value**
 - exponent: 11111110
 \Rightarrow actual exponent = $254 - 127 = +127$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- **Exponents 0000...00 and 1111...11 reserved**
- **Smallest value**
 - Exponent: 000000000001
 \Rightarrow actual exponent = $1 - 1023 = -1022$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value**
 - Exponent: 111111111110
 \Rightarrow actual exponent = $2046 - 1023 = +1023$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
 - all fraction bits are significant
 - Single: approx 2^{-23}
 - Equivalent to $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 7$ decimal digits of precision
 - Double: approx 2^{-52}
 - Equivalent to $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Limitations

- Overflow:

The number is too big to be represented

- Underflow:

The number is too small to be represented

Denormal Numbers

- Exponent = 000...0 \Rightarrow hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{1-\text{Bias}}$$

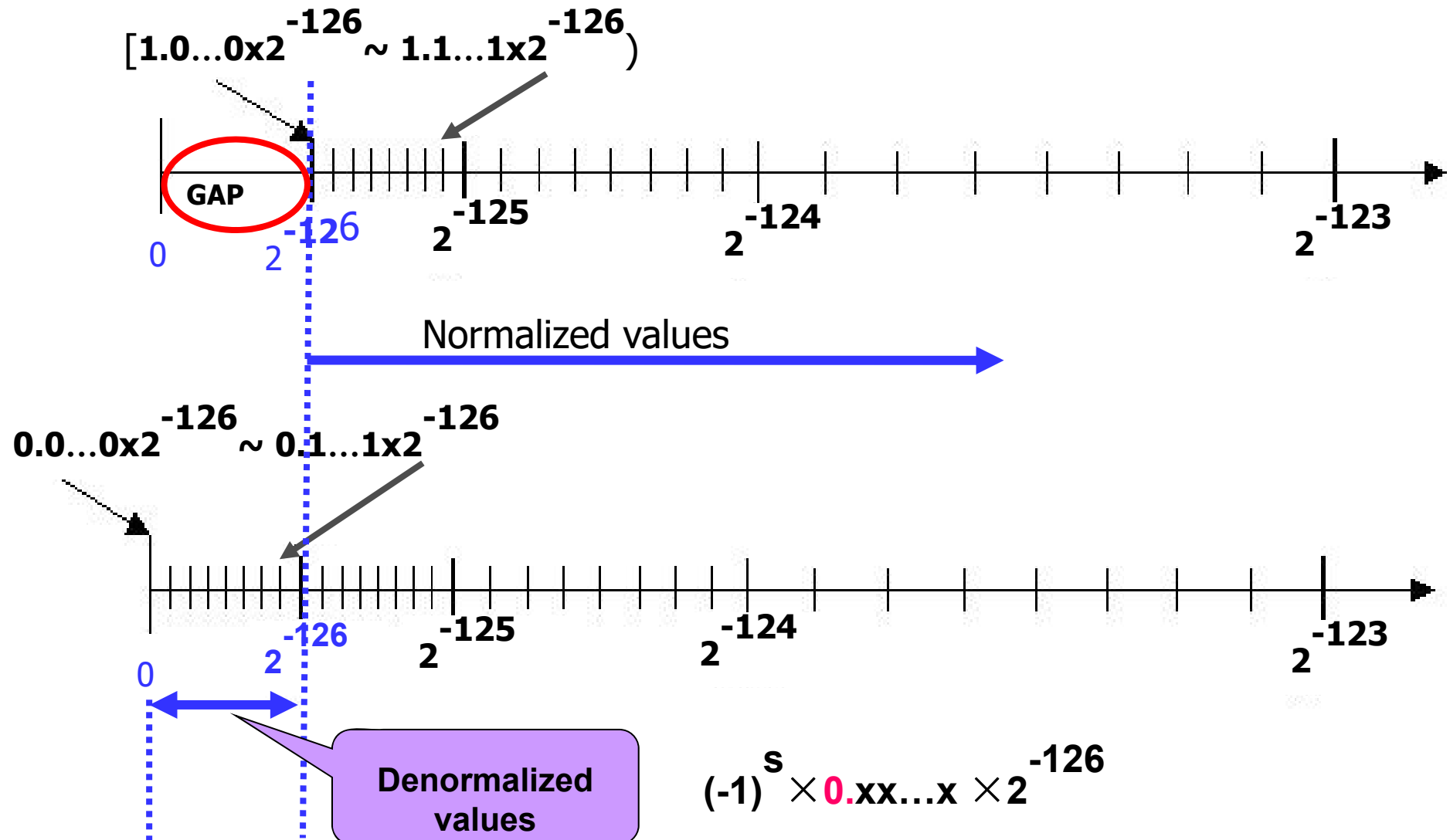
- Smaller than normal numbers
 - allow for gradual underflow, with diminishing precision
- Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!



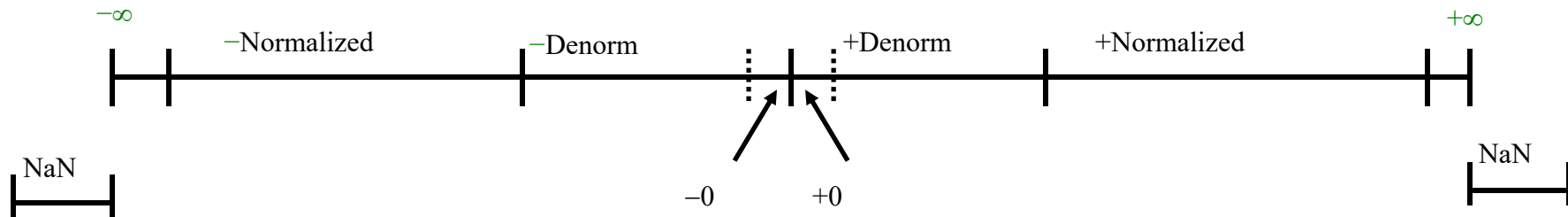
Normalized vs. Denormalized Values



Infinites and NaNs

- Exponent = 111...1, Fraction = 000...0
 - \pm Infinity
 - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction \neq 000...0
 - Not-a-Number (NaN)
 - Indicates illegal or undefined result
 - e.g., 0.0 / 0.0
 - Can be used in subsequent calculations

Visualization: Floating Point Encodings

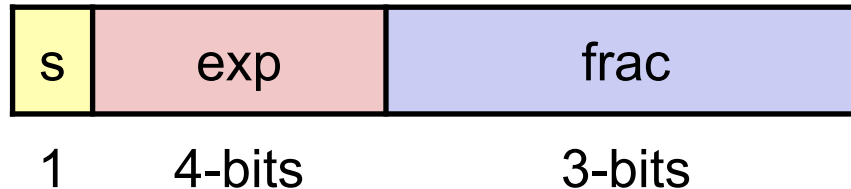


IEEE 754 standard

- $00\dots 00_{\text{two}}$ represents 0;
- instead of interrupting on a divide by 0, software can set the result to a bit pattern representing $+\infty$ or $-\infty$;
- *NaN*: Not a Number.
- $\pm\infty$ - represented by $f=0, E=255, S=0,1$ - must obey all mathematical conventions: $F+\infty=\infty, F/\infty=0$
- **denormalized numbers** - represented by $E=0$ - values smaller than smallest normalized number - lowering probability of exponent underflow

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	denormalized number
1-254	Anything	1-2046	Anything	Floating-point number
255	0	2047	0	infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Tiny Floating Point Example



- 8-bit Floating Point Representation
 - the sign bit is in the most significant bit
 - the next four bits are the exponent, with a bias of 7
 - the last three bits are the **frac**
- Same general form as IEEE Format
 - normalized, denormalized
 - representation of 0, NaN, infinity

Special Properties of Encoding

- FP Zero Same as Integer Zero
 - All bits = 0
- Can (Almost) Use Unsigned Integer Comparison
 - Must first compare sign bits
 - Must consider $-0 = 0$
 - NaNs problematic
 - Will be greater than any other values
 - What should comparison yield?
 - Otherwise OK
 - Denorm vs. normalized
 - Normalized vs. infinity

Floating Point Operations: Basic Idea

- $\mathbf{x} +_{\mathbf{f}} \mathbf{y} = \mathbf{Round}(\mathbf{x} + \mathbf{y})$
- $\mathbf{x} \times_{\mathbf{f}} \mathbf{y} = \mathbf{Round}(\mathbf{x} \times \mathbf{y})$
- Basic idea
 - First **compute exact result**
 - Make it fit into desired precision
 - Possibly overflow if exponent too large
 - Possibly **round to fit into frac**

Rounding

- Rounding Modes (illustrate with \$ rounding)

■		\$1.40	\$1.60	\$1.50	\$2.50	−\$1.50
•	Towards zero	\$1	\$1	\$1	\$2	−\$1
•	Round down ($-\infty$)	\$1	\$1	\$1	\$2	−\$2
•	Round up ($+\infty$)	\$2	\$2	\$2	\$3	−\$1
•	Nearest Even (default)	\$1	\$2	\$2	\$2	−\$2

- What are the advantages of the modes?

Closer Look at Round-To-Even

- Default Rounding Mode
 - Hard to get any other kind without dropping into assembly
 - All others are statistically biased
 - Sum of set of positive numbers will consistently be over- or under- estimated
- Applying to Other Decimal Places / Bit Positions
 - When exactly halfway between two possible values
 - Round so that least significant digit is even
 - E.g., round to nearest hundredth

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

Rounding Binary Numbers

- Binary Fractional Numbers

- “Even” when least significant bit is 0
- “Half way” when bits to right of rounding position = 100...₂

- Examples

- Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011 ₂	10.00 ₂	(<1/2—down)	2
2 3/16	10.00110 ₂	10.01 ₂	(>1/2—up)	2 1/4
2 7/8	10.11100 ₂	11.00 ₂	(1/2—up)	3
2 5/8	10.10100 ₂	10.10 ₂	(1/2—down)	2 1/2

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^E$
- One can reason about operations independent of implementation
 - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
 - Violates associativity/distributivity
 - Makes life difficult for compilers & serious numerical applications programmers
 - Some famous disasters caused by numerical errors

Disasters Attributable to Bad Numerics

■ The Patriot Missile Failure

- On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people.
- It turns out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors.



■ The Explosion of the Ariane 5

- On June 4, 1996, an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing \$7 billion. The destroyed rocket and its cargo were valued at \$500 million.
- It turned out that the cause of the failure was a software error in the inertial reference system.



Representation of Numeric Data

- Fixed-point number representation
- Representation of integers
- Floating-point number representation
- **Binary codes for decimal digits**

Binary Codes for Decimal Digits

- The computer needs to process decimal data, so it must be represented by binary numbers or **Binary Coded Decimal (BCD)** codes.
- **Weighted BCD Code:** The weighted BCD code means that the four binary digits of the decimal digits have a certain right. The common used one is **8421 code**.
- **No weight BCD Code:** Commonly used coding schemes are **Gray code** and **Excess3 code**.

DECIMAL CODES

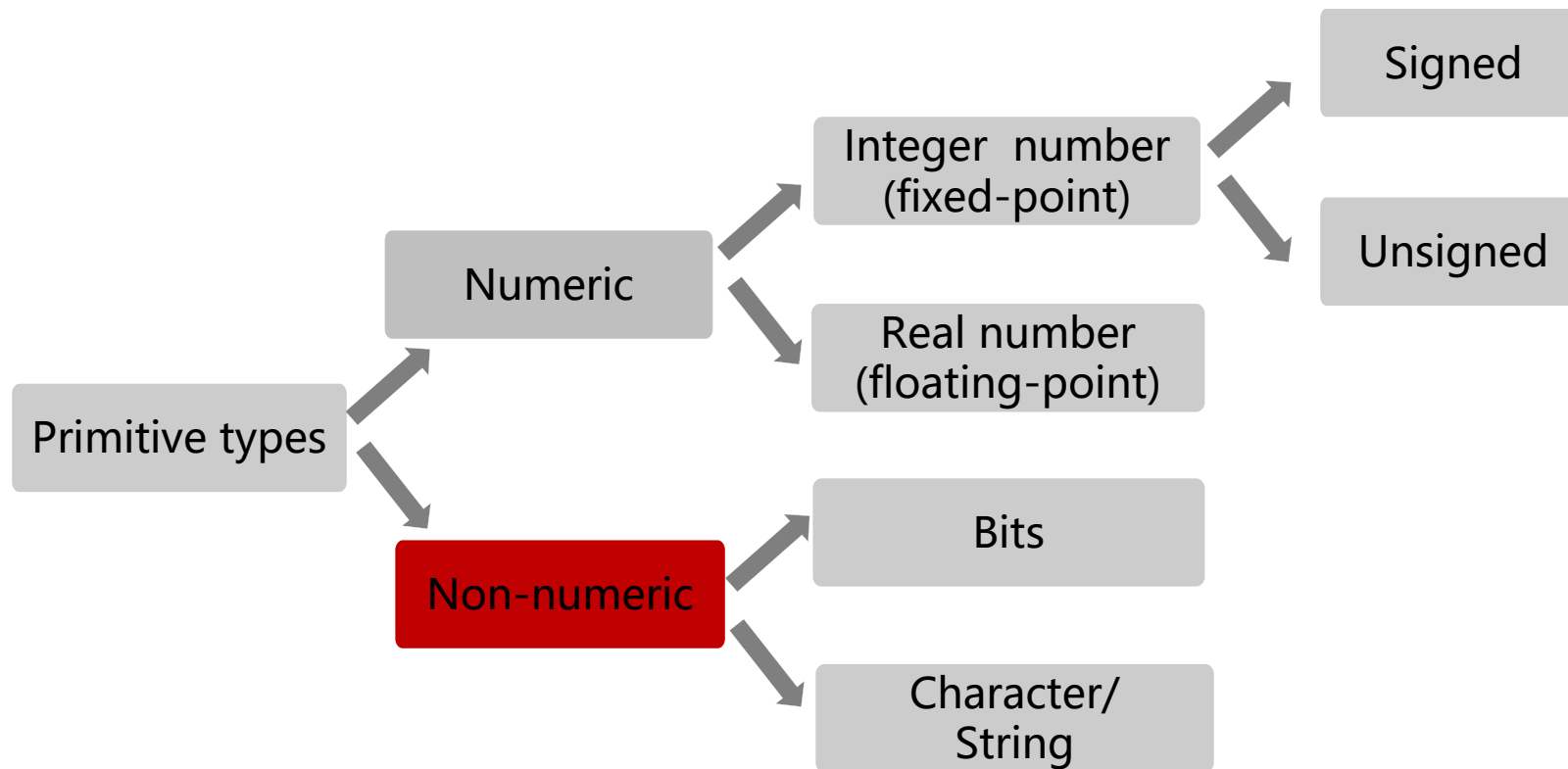
- There are over 8,000 ways that you can chose 10 elements from the 16 binary numbers of 4 bits. A few are useful:

Decimal	8,4,2,1	Excess3	8,4,-2,-1	Gray
0	0000	0011	0000	0000
1	0001	0100	0111	0100
2	0010	0101	0110	0101
3	0011	0110	0101	0111
4	0100	0111	0100	0110
5	0101	1000	1011	0010
6	0110	1001	1010	0011
7	0111	1010	1001	0001
8	1000	1011	1000	1001
9	1001	1100	1111	1000

Overview

- Binary number representation
- Representation of numeric data
- **Representation of non-numeric data**
- Data width and storage

Non-numeric Data



Low-level programmer's/
hardware system designer's
perspective

Representation of logical values

- Under normal circumstances, each word or other addressable unit can be regarded as a whole data unit.
- However, sometimes it is necessary to treat an n -bit data as consisting of n 1-bit data, each is 0 or 1.
- Therefore, n -bit binary numbers can represent n logical values.

Alphanumeric Codes

Applications require the handling of data consisting not only numbers but also **letters, **symbols** and other media.**

But computer can only handle binary data, thus encoded these symbols to binary code.

7-Bits ASCII Code

- American Standard Code for Information Interchange. It uses 7-bits to represent:
- 94 Graphic printing characters.
 - 10 digits:
 - 26 uppercase letter
 - 26 lowercase letter
 - 32 special printable characters
- 34 Non-printing characters
 - For format (e.g. BS = Backspace, CR = carriage return)
 - For text (e.g. STX and ETX)
 - For communication (e.g. ACK)

7 BIT ASCII CODE TABLE

<div> <div>b6b5b4</div> <div>B3b2b1b0</div> </div>				000	001	010	011	100	101	110	111
0	0	0	0	NUL	DLE	SP	0	@	P	,	p
0	0	0	1	SOM	DC	!	1	A	Q	a	q
0	0	1	0	STX	DC	“	2	B	R	b	r
0	0	1	1	ETX	DC	#	3	C	S	c	s
0	1	0	0	EOT	DC	\$	4	D	T	d	t
0	1	0	1	ENQ	NAA	%	5	E	U	e	u
0	1	1	0	ACA	SYN	&	6	F	V	f	v
0	1	1	1	BEL	ETB	,	7	G	W	g	w
1	0	0	0	BS	CAN	(8	H	X	h	x
1	0	0	1	HT	EM)	9	I	Y	i	y
1	0	1	0	LF	SUB	*	:	J	Z	j	z
1	0	1	1	VT	ESC	+	;	A	[k	
1	1	0	0	FF	FS	,	<	L	\	l	
1	1	0	1	CR	GS	—	=	M]	m	
1	1	1	0	SO	RS	.	>	N		n	~
1	1	1	1	SI	US	/	?	O	←	o	DEL

ASCII Properties

- Digits 0 to 9 span Hexadecimal values 30_{16} to 39_{16}
- Upper case A-Z span 41_{16} to $5A_{16}$
- Lower case a-z span 61_{16}
 - Lower to upper case translation (and vice versa) occurs by flipping bit 6
- Delete (DEL) is all bits set
 - A carryover from when punched paper tape was used to store messages
 - Punching all holes in a row erased a mistake

UNICODE

- **UNICODE extends ASCII to 16 bits (65,536) universal characters codes**
 - **For encoding characters in world languages**
 - **Available in many modern applications**
 - **2 byte (16-bit) code words**
 - **See Reading Supplement – Unicode on the Companion Website**
<http://www.prenhall.com/mano>

Representation of Chinese Characters

- The Chinese character system includes the following Chinese character codes: input code, internal code, font code.
- **Input Code:** Chinese characters are encoded using the corresponding keys.
- **Internal Code:** The internal code of Chinese characters consists of several rows and several columns. The row number is called the area code, and the column number is called the tag number.
- **Font Code**

Overview

- Binary number representation
- Representation of numeric data
- Representation of non-numeric data
- **Data width and storage**

Data Width and Unit

- **Bit:** The smallest unit that makes up a binary number
- **Byte:** 8bits
- **Machine has “word size”**
 - Nominal size of integer-valued data
 - Including addresses
 - Most current machines use 32 bits (4 bytes) words
 - Limits addresses to 4GB
 - Becoming too small for memory-intensive applications
 - High-end systems use 64 bits (8 bytes) words
 - Potential address space $\approx 1.8 \times 10^{19}$ bytes
 - x86-64 machines support 48-bit addresses: 256 Terabytes
 - Machines support multiple data formats
 - Fractions or multiples of word size
 - Always integral number of bytes

Data Representations

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

Unit of Main Memory Capacity

K:1KB= 2^{10} byte=1024byte

M:1MB= 2^{20} byte=1 048 576byte

G:1GB= 2^{30} byte=1 073 741 824byte

T:1TB= 2^{40} byte=1 099 511 627 776byte

P:1PB= 2^{50} byte=1 125 899 906 842 624byte

E:1EB= 2^{60} byte=1 152 921 504 606 846 976byte

Z:1ZB= 2^{70} byte=1 180 591 620 717 411 303 424 byte

Y:1YB= 2^{80} byte=1 208 925 819 614 629 174 706 176 byte

Unit of Frequency and Line Width

b/s:bps

kb/s:1kb/s= 10^3 b/s=1000bps

Mb/s:1Mb/s= 10^6 b/s=1000kbps

Gb/s:1Gb/s= 10^9 b/s=1000Mbps

Tb/s:1Tb/s= 10^{12} b/s=1000Gbps

Data Storage and Arrangement Order

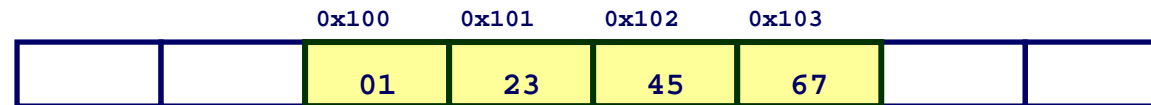
- **Least Significant Bit vs. Least Significant Byte**
- **Most Significant Bit vs. Most Significant Byte**

- How should bytes within a multi-byte word be ordered in memory?
- Conventions
 - Big Endian: Sun, PPC Mac, Internet
 - Least significant byte has highest address
 - Little Endian: x86
 - Least significant byte has lowest address

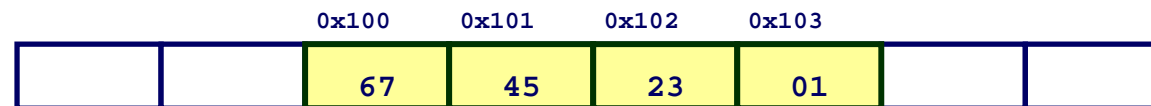
Byte Ordering Example

- Example
 - Variable x has 4-byte representation 0x01234567
 - Address given by &x is 0x100

Big Endian



Little Endian



Summary

- **Introduction to computer systems**
- **Binary number representation**
- **Arithmetic operations**
- **Representation of numeric data**
- **Representation of non-numeric data**
- **Data width and storage**

END