# Inf2C - Computer Systems
# Lectures 2-3
# Data Representation

Vijay Nagarajan

School of Informatics

University of Edinburgh

# Last lecture

- Course overview
  - – Piazza: up & running. Use it!
  - – Labs: drop-in. Start in week 2
  - – Tutorials: start in week 3
- Moore's law
- Types of computer systems
- Computer components
- Computer system stack

# Lecture 2: Data Representation

- The way in which data is represented in computer hardware affects
  - complexity of circuits
  - cost
  - speed
  - reliability
- Must consider how to design hardware for
  - Storing data: memory
  - Manipulating data: processing (e.g., adders, multipliers)

# Lecture outline

- The bit – atomic unit of data
- Representing numbers
  - Integers
  - Floating point
- Representing text

# The bit

- Information represented as sequences of symbols
  - Humans use letters, numerals, punctuation, whitespace
  - Computers use just 0s and 1s – ***bits***
- ***Bit*** – an acronym for Binary digiT
- Advantages: easy to do computation, very reliable, simple circuits
- Disadvantages: little information per bit → must use many bits. 256 ≡ 1 0000 0000, 'A' ≡ 0100 0001

# Natural numbers representation

- Non-negative (unsigned) integers are very simple to represent in binary

| | Most significant bit | | | Least significant bit |
|---|---|---|---|---|
| | MSB | | | LSB |
| **Bit position** | n-1 | n-2 | 1 | 0 |
| **Binary:** | ☐ | ☐ •••• | ☐ | ☐ |
| **Decimal:** | $*2^{n-1}+$ | $*2^{n-2}+\cdots$ | $*2^{1}+$ | $*2^{0}$ |

# Basic operations

- Addition, subtraction with unsigned binary numbers is easy:

13

$$\begin{array}{r} 1\,1\,1\,1\;\; \\ 01101 \\ +01011 \\ \hline 11000 \end{array} \qquad\qquad \begin{array}{r} 0\,0\,1\,0\;\; \\ 01101 \\ -01011 \\ \hline 00010 \end{array}$$

11

24

2

# Fixed bit-length arithmetic

- Hardware cannot handle infinitely long bit sequences
- We end up with a few fixed-size data types
  - Byte: always 8 bits
  - Word: the typical unit of data on which a processor operates (32 or 64 bits most common today)
- Overflow happens when a result does not fit
  - Numbers wrap-around when they become too large
  - Arithmetic is modulo $2^N$, where N=number of bits

# What about negative numbers?

- Sign-magnitude representation:
  - Use 1st bit (MSB) as the sign
  - 0 → positive, 1 → negative,

    0010 ≡ 2   1010 ≡ -2
- Complicates addition and subtraction
  - The actual operation depends on the sign
- Has positive and negative zero
  - 0000 ≡ 0   1000 ≡ -0

Better way: 2's complement representation

# Two's complement: the intuition

- Want: X + (-X) = 0
- Insight: don't need the full sum to be 0
  - Only the bits within a computer's fixed width need to be 0 (exploit overflows!)
- Approach:
  - Represent the negation of X as $2^N$-X
  - Then: X + (-X) = X + ($2^N$-X) = $2^N$
    - Recall: largest number represented with N bits: $2^N$-1
    - Note that N lowest bits of the sum are all 0!

# Two's complement: example

Given:

- 3-bit fixed width (N=3)
- X = 2 (decimal) → 0 1 0 (binary)

$2^N = 8$ (dec) → 1 0 0 0 (bin)

$-X = 2^N - X = 8 - 2 = 6$ (dec) → 1 1 0 (bin)

Check:

$X + (-X) = 0\,1\,0 + 1\,1\,0 = 1\,\boxed{0\,0\,0}$ ✓

**3-bit fixed width**

# Efficiently computing 2's complement

EASY!

"Flip the bits and add 1"

Example:

X = 0 1 0 (bin) → 2 (dec)

Flip the bits: 1 0 1

Add 1: 1 1 0 (bin) → -X

# The roots of the idea

**John von Neumann** (died: 1957)

- Co-inventor of the stored program concept
- Proposed 2's complement idea in a 1945 paper
- Also came up with cellular automata, numerical weather forecasting, concept of global warming
- Outside of computing: linear programming, quantum logic, policy of mutually assured destruction, and more!

# 2's complement details

- The MSB is the sign
- A – B = A + (2's complement of B)
- Arithmetic operations do not depend on the operands' signs
- Range is asymmetric: $-2^{n-1}$ to $2^{n-1}-1$
- There are two kinds of overflows:
  - – Positive overflow produces a negative number
  - – Negative underflow produces a positive number

# Converting between data types

- Converting a 2's complement number from a smaller to a larger representation is done by **sign extension**

Example: from byte to short (16 bits):

2 = 0 0 0 0 0 0 1 0 ⇒ ? ? ? ? ? ? ? ? 0 0 0 0 0 0 1 0

-2 = 1 1 1 1 1 1 1 0 ⇒ ? ? ? ? ? ? ? ? 1 1 1 1 1 1 1 0

2 = 00000010 ⇒ 0000000000000010

-2 = 11111110 ⇒ 1111111111111110

**(byte)** **(short)** **(byte)** **(short)**

# Shifting

- Shifting: move the bits of a data type left or right
  - Data bits falling off the edge are lost
- For left shifts, 0s fill in the empty bit places
- For right shifts, two options:
  - Fill with 0 (logical shift): for non-numerical data
  - Fill with MSB (arithmetic shift): for 2's complement numbers
- Shift left by *n* is equivalent to multiplying by $2^n$
- Shift right by *n* is equivalent to dividing by $2^n$ and rounding towards $-\infty$
- Example:

  6 = 0 0 0 0 0 1 1 0 << 2 → 0 0 0 1 1 0 0 0 = 24

  6 = 0 0 0 0 0 1 1 0 >> 2 → 0 0 0 0 0 0 0 1 = 1

  -6 = 1 1 1 1 1 0 1 0 >> 2 → 1 1 1 1 1 1 1 0 = -2

# Hexadecimal notation

- Binary numbers (and other binary-encoded information) are too long and tedious for us (humans) to use
- Solution: use a more compact encoding
  - Hexadecimal (base 16) is most common
- Hex digits: 0-9 and A-F
  – $A=10_{dec}$, B=11, …, F=15
- Conversion to/from binary is very easy:

  Every 4 bits correspond to 1 hex digit

$$\underbrace{1\;1\;1\;1}_{F(15)}\;\underbrace{1\;0\;0\;0}_{8} \quad = 0xF8$$

Hex is just a convenience for humans
Computers use the binary form

# Real numbers - floating point

- Java's `float` **(32 bits)**
  `double` **(64 bits)**

- Binary representation:
  – example 0.75 in base 10 $\Rightarrow$ 0.11 in base 2

  $(2^{-1} + 2^{-2} = 0.5 + 0.25 = 0.75)$

# Real numbers - floating point

- Java's `float` **(32 bits)**
  `double` **(64 bits)**

- Binary representation:
  – example 0.75 in base 10 $\Rightarrow$ 0.11 in base 2

$$(2^{-1} + 2^{-2} = 0.5 + 0.25 = 0.75)$$

- Normalized form: one non-zero digit before the point

**Mantissa (aka significand)** **exponent**

$$0.11 \Rightarrow 1.1\text{x}2^{-1}$$

**Always 1 in binary**

# Why normalize?

Three reasons:

1. Simplifies machine representation
(don't need to represent the fraction separator)

2. Simplifies comparisons

– Which one is bigger: 0.0000101 or 0.000001 ?

$1.01 \times 2^{-5}$ vs $1.0 \times 2^{-6}$

3. Is more compact for very small/large numbers

– E.g., $0.0000000000000001 = 1.0 \times 2^{-16}$

or can be made more compact (by rounding fraction)

# Floating point conversion example #1

Convert the number 25 to floating point with normalization

1) 25 in base 10 $\Rightarrow$ 11001 in base 2

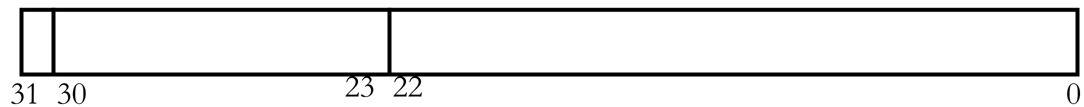
2) 11001 to normalized floating point $\Rightarrow$ $1.1001 \times 2^4$

Understand that:

- The number is normalized
- 1.1001 is mantissa (aka significand)
- 4 is exponent
- sign is "+" (implicit here)

# IEEE 754 Floating Point standard

- Need a standard to represent and compute with fixed-width floats
- 32 bit representation:




**sign (s)** **exponent (exp)** **mantissa (m)**

$$(-1)^s \text{ x } (1.m) \text{ x } 2^{exp-127}$$ **Bias**

**“1.” is implicit**

e.g.,

$(0.75)_{10} \rightarrow (0.11)_2 \rightarrow (1.1x2^{-1})_2$

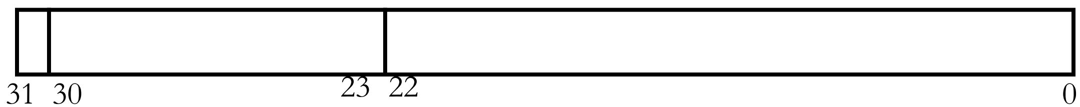
$\rightarrow$ s = 0, m = 1, exp = 126

$\rightarrow$ 0 01111110 10000000000000000000000

Note: representation does NOT use 2’s complement

# IEEE 754 Floating Point standard

- Need a standard to represent and compute with fixed-width floats
- 32 bit representation:




**sign (s)** **exponent (exp)** **mantissa (m)**

$$(-1)^s \times (1.m) \times 2^{exp-127}$$

**Bias**

**"1." is implicit**

e.g.,

$(0.75)_{10} \rightarrow (0.11)_2 \rightarrow (1.1\text{x}2^{-1})_2$

$\rightarrow$ s = 0, m = 1, exp = 126

$\rightarrow$ 0 01111110 10000000000000000000000

- 64 bit representation:
  - exponent = 11 bits; mantissa= 52 bits

# IEEE 754 Floating Point standard

- Why bias?
  – Avoids the complexity of +/- exponents
  – Simplifies relative ordering of FP numbers

- Note: processors usually have specialized floating point units to perform FP arithmetic

# IEEE 754 floating point conversion #2

Example: Convert 23.5 (decimal) to IEEE 754 floating point

Start: 23 in base 10 $\Rightarrow$ 10111 in base 2

# IEEE 754 floating point conversion #2

## Example: Convert 23.5 (decimal) to IEEE 754 floating point

Start: 23 in base 10 ⇒ 10111 in base 2

1) 23.5 in base 10 ⇒ 10111.1 in base 2

2) 10111.1 to normalized floating point ⇒ $1.01111 \times 2^4$

3) S = 0
   M = 01111 is mantissa (remember: "1." is implicit)
   Exp = 4+127 = 131 in base 10 ⇒ 1000 0011 in base 2

| 31 | 30 … 23 | 22 … 0 |
|---|---|---|
| 0 | 1 0 0 0 0 0 1 1 | 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| **sign (s)** | **exponent (exp)** | **mantissa (m)** |

**Pad with 0s**

# IEEE 754: Special Values

| Exponent | Mantissa | Meaning |
|---|---|---|
| 0 | 0 | 0 |
| 1-254 | Anything | Floating point number |
| 255 (0xFF) | 0 | Infinity (signed) |
| 255 (0xFF) | Non-zero | Not-a-number (NaN) |

**32-bit representation**

# Representing characters

- Characters need to be encoded in binary too
- Operations on characters have simpler requirements than on numbers, so the encoding choice is not crucial
- Most common representation is ASCII
  - – Each character is held in a byte
  - – E.g. '0' is 0x30, 'A' is 0x41, 'a' is 0x61
- Java uses Unicode which can encode characters from all languages
  - – 16 bits per character

# Representing strings

- Words, sentences, etc. are just strings of characters
- How is the end of a string identified?
  - No common standard exists. Different programming languages use different encodings
  - In C: a special character, encoded as 0x00
    - Also called NULL character
  - In Java: string length is kept with the string itself
    - string is an object and *length* is one of its member variables

# Summary

- Computers use binary representation
- Signed numbers: sign-magnitude vs 2's complement
- Floating point
- Characters and strings