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 - memories
  - Manipulating data – e.g. adders, multipliers
    - How would an algorithm for adding Roman numbers look like?

Lecture outline

- The bit – atomic unit of data
- Representing numbers
- Representing text

The bit

- Information represented as sequences of symbols
  - In text, symbols are letters, numerals, punctuation, whitespace
  - With computers, we use just 0s and 1s, ***bits***
- ***Bit*** is an acronym for Binary digiT
- Disadvantages: little information per bit, must use many of them. 512 ≡ 1 0000 0000, 'A' ≡ 0100 0001
- Advantages: easy to do computation, very reliable, simple circuits

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 binary numbers is easy:

13

1 1 1 1	0 0 1 0
01101	01101
+01011	−01011
11000	00010

11

24

2

Fixed bit-length arithmetic

- Hardware cannot handle infinite long bit sequences
- We end up with a few fixed sized data types
  – Byte: always 8 bits
  – Word: the 'natural' unit of access, usually 32 bits
- Overflow happens when a result does not fit
  – Numbers wrap-around when they become too large
  – Comp. arithmetic is modulo 2^n, n=number of bits

What about negative numbers?

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

    0110 ≡ 6 1110 ≡ -6
- Complicates addition and subtraction
  - The actual operation depends on the sign
- There is a better way

Two's complement representation

- If doing mod 2k arithmetic on numbers 0…2k-1, treat numbers k…2k-1 as -k…-1
- To find the value of a binary number, consider the MSB as having negative weighting:

	n-1	n-2		1	0
Binary:	☐	☐	••••	☐	☐

Decimal: $- \ast 2^{n-1} + \ast 2^{n-2} + \cdots \ast 2^{1} + \ast 2^{0}$

- Arithmetic operations do not depend on the operands' signs
- 0110 ≡ 6 1110 ≡ -2

2's complement quirks

- The MSB is the sign
- 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
- To negate a number

  Invert all bits ($0 \leftrightarrow 1$) and add 1, at the LSB

  $-(-2^{n-1})$ overflows!
- A-B = A + 2's complement of B

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 = 11111110 ⇒ 1111111111111110   2 = 00000010 ⇒ 0000000000000010

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

Shifting

- Shifting: move the bits of a data type left or right
  - Data bits falling off the edge are lost
- 0s fill up the empty bit places for left shifts
- For right shifts, two options:
  - Fill with 0: for non-numerical data (or positive integers)
  - Fill with the MSB: 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 -∞
- Example

  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 data) are too long and tedious for us to use
- Hexadecimal (base 16) is very commonly used in computer programming
- Hex digits: 0-9 and A-F
  - A=10, 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, computers use the binary form

Real numbers - floating point

- Java's **`float` (32 bits)**
  `double` (64 bits)
- IEEE 754:
  - example 0.75 in base 10 $\Rightarrow$ 0.11 in base 2

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

  - Normalized:

    mantissa **exponent**

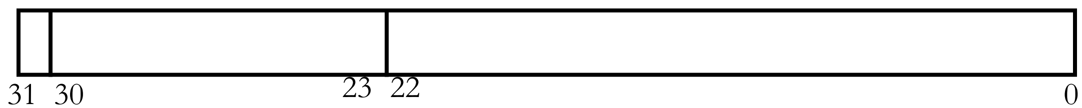
    $0.11 \Rightarrow 1.1\text{x}2^{-1}$

    implict (always 1)

  - example: 25 in base 10 $\Rightarrow$ 11001 in base 2 $\Rightarrow$ $1.1001\text{x}2^4$

Floating Point

- 32 bit:



sign (s) **exponent (exp)** **mantissa or significand (sig)**

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

e.g.,

$(0.75)_{10} \rightarrow (0.11)_2 \rightarrow (1.1\text{x}2^{-1})_2 \rightarrow$ 0 01111110 10000000000000000000000

- 64 bit:
  – exponent = 11 bits; significand = 52 bits
- Note: processors usually have specialized floating point units and extra fp registers to perform fp arithmetic

Representing characters, strings

- 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 many (all?) languages
  - 16 bits per character required
- Words, sentences, etc. are just strings of characters
  - A special character, encoded as 0x00, shows where the string ends (in C)
  - Or the string length is kept with the string itself (in Java)

Summary

- Computers use binary representation
- 2's complement
- Floating point
- Characters and strings