

## Data Representation

- Refs: [O'H&Bryant, sect 2.1,2.4], [Null&Lobur, sect 2.5–2.7], [Tanenbaum, appendix B], related web links
- bit, byte and word
- characters and strings, ASCII
- big-endian, little-endian
- representing real numbers using floating point numbers
  - binary fractions
  - formats (IEEE)
  - pitfalls
- Admin notices:
  - tute problem: wed12-14 underfull; propose to move to wed09-11

## Characters and Strings

- text can be stored in memory by using a number to represent every character
  - ASCII (American Standard Code for Information Interchange): 7 bits, 128 characters
  - EBCDIC: Extended Binary-Coded Decimal Interchange Code (8 bits)
  - UCS/UNICODE: attempts to extend ASCII to other languages (65,536 characters)
- a string is a sequence of characters, usually terminated with a byte value of 0
- example: 43 4F 4D 50 32 33 30 30 0<sub>16</sub>

## Bit, Byte and Word

- basic unit of memory is the bit (Binary digIT)
- one bit is too small to be generally useful, so bits are grouped together:
  - byte: 8 bits (e.g. 1010 1010 or 1000 0001)
  - word: 1, 2, 4 or 8 bytes (usually 4 or 8)
  - word length depends on processor, operating system, etc. (e.g. 8 bytes for Intel's IA-64 or AMD x64-86 architectures)
- data and machine instructions are stored and accessed in words (in memory)
- 1 Kilobyte (KB) = 1024 bytes = 2<sup>10</sup> bytes
- 1 Megabyte (MB) = 1024 KB = 1,048,576 bytes
- 1 Gigabyte (GB) = 1024 MB
- typical modern computers have millions of bytes of memory (e.g. PC with 512 Megabytes, servers 4-16 Gigabytes)
- CD-ROMs store around 650 Megabytes (MB)
- DVDs store around 4.7 Gigabytes (GB), up to 17 GB

## Big-endian versus Little-endian Integers

- small (how small?) integers are sometimes stored in one byte
- most integers are stored in (longer) words; how are they arranged in memory?  
Example: 126540713<sub>10</sub> stored in 4 bytes: 00000111 10001010 11011011 10101001<sub>2</sub>

Address	big-endian	little-endian
FF <sub>16</sub>	10101001	00000111
FE <sub>16</sub>	11011011	10001010
FD <sub>16</sub>	10001010	11011011
FC <sub>16</sub>	00000111	10101001
- big-endian stores the most significant byte (MSB) at the lowest address in the word, little-endian in the highest (origin of term endian)
- similar to use of me@cs.anu.edu.au OR me@au.edu.anu.cs (e.g. JANET network in the UK)
- little endian: (Intel) more natural and consistent way to pick up 1, 2, 4, or longer byte integers (making multi-precision arithmetic easier)
- big endian: (SPARC, IBM) MSB first gives easy testing of +ve/-ve. Numbers stored in order they are printed, making binary to decimal conversion easier
- potential confusion in communication with differing endianess!

## Representing Real Numbers: Fixed and Floating Point

- fixed-point numbers:
  - a fixed number of bits for the integer component and a fixed number of bits for the fractional component
  - needs a large number of bits to represent a useful range
  - e.g. mass of electron is  $9 \times 10^{-28}$  grams, mass of the sun is  $2 \times 10^{33}$ , a range of  $10^{60}$ !

need 200 bits to cover this range.  $\Rightarrow$  200 bits of precision with solar masses (wasteful), but only a couple of bits with atomic masses (inaccurate)
- floating point numbers:
  - similar to scientific notation:  $x = f \times 10^e$  e.g.  $3.141 = 0.3141 \times 10^1$
  - general form  $x = -1^s \times m \times 2^e$ 
    - ◆ 1 bit for sign ( $s$ : 0 - positive, 1 - negative)
    - ◆ precision is determined by the number of bits used for the mantissa ( $m$ )
    - ◆ range is determined by the number of bits used for the exponent ( $e$ )
  - floating point numbers are usually normalised, e.g.  $x = -1^s \times 1.m \times 2^e$

## Binary Fractions

- same approach for the representation of a number in binary or in decimal
- a “binary point” delimits the whole and fractional parts
- fractional binary numbers can be translated into decimal in the following ways:

Base 2 (binary)	Base 10 (decimal)
0.001	0.125
0.01	0.25
0.1	0.5
1.0	1.0
10.0	2.0
0.00011001100110011...	$\frac{1}{10}$

- the contribution of each digit underpins both representations

## Fractions: Binary to Decimal

- conversion of a binary representation of a whole number to a decimal representation by adding the contribution of each digit in the same way
- the contribution of each digit is based on the power associated with each digit. e.g:

$2^3$	$2^2$	$2^1$	$2^0$	.	$2^{-1}$	$2^{-2}$
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	0	1	1	.	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
$1 \times 2^3$	$0 \times 2^2$	$1 \times 2^1$	$1 \times 2^0$	.	$0 \times 2^{-1}$	$1 \times 2^{-2}$
8	0	2	1	.	0	.25

Thus,

$$1011.01_2 = 8_{10} + 2_{10} + 1_{10} + 0.25_{10} \\ = 11.25_{10}$$

i.e. 1011.01 (binary encoding) is equivalent to 11.25 (decimal encoding)

## Fractions: Decimal to Binary

- recall that the conversion of a whole number in a larger base to a smaller base,  $k$ , occurs by repeatedly dividing (the whole part of) the number by  $k$ 
  - the remainders at each stage (in reversed order) forms the answer
- mirroring this, the conversion of the fractional part occurs by repeatedly *multiplying* (the *fractional part* of) the number by  $k$ 
  - the digit of *the whole part* of the number at each stage (in *forward order*) forms the answer
- e.g. converting  $0.6875_{10}$  to base 2 is achieved by:

$$0.6875 \times 2 = 1.375 \\ 0.375 \times 2 = 0.75 \\ 0.75 \times 2 = 1.5 \\ 0.5 \times 2 = 1.0$$

Thus,  $0.6875_{10} = 0.1011_2$

- Q: what is  $0.15625_{10}$  in base 2?
- is this process guaranteed to terminate?

