

# 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

## 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 =  $2^{10}$  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:  $126540713_{10}$  stored in 4 bytes:  $00000111\ 10001010\ 11011011\ 10101001_2$

| Address   | big-endian | little-endian |
|-----------|------------|---------------|
| $FF_{16}$ | 10101001   | 00000111      |
| $FE_{16}$ | 11011011   | 10001010      |
| $FD_{16}$ | 10001010   | 11011011      |
| $FC_{16}$ | 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 endianness!



# 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,

$$\begin{aligned}
 1011.01_2 &= 8_{10} + 2_{10} + 1_{10} + 0.25_{10} \\
 &= 11.25_{10}
 \end{aligned}$$

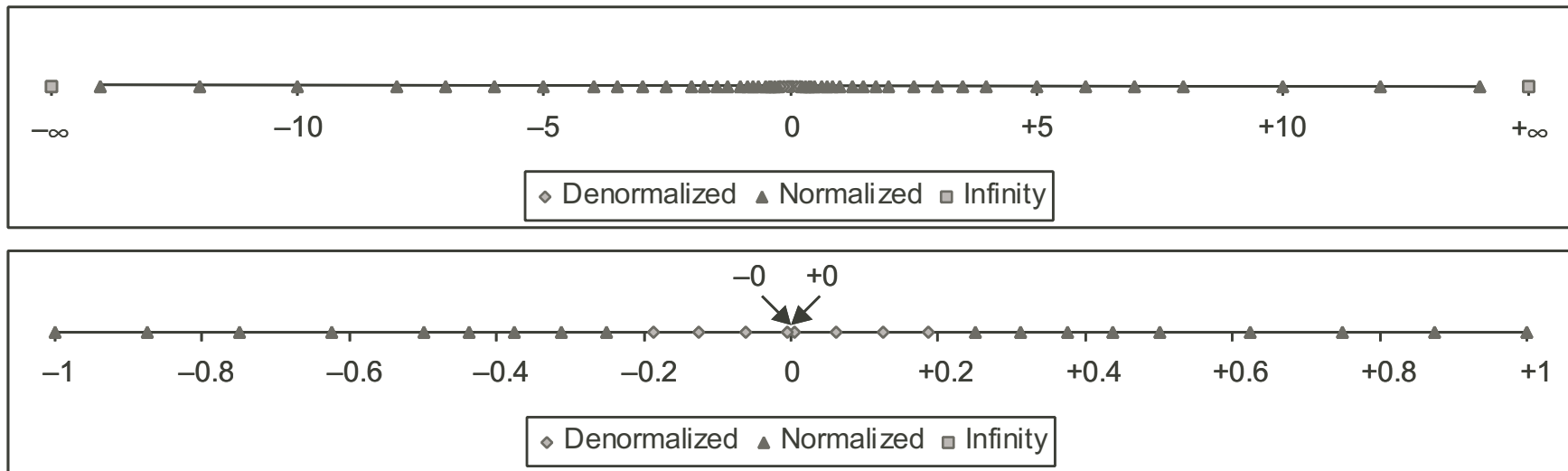
i.e. 1011.01 (binary encoding) is equivalent to 11.25 (decimal encoding)





## Floating Point Pitfalls: 'REALs aint Reals'

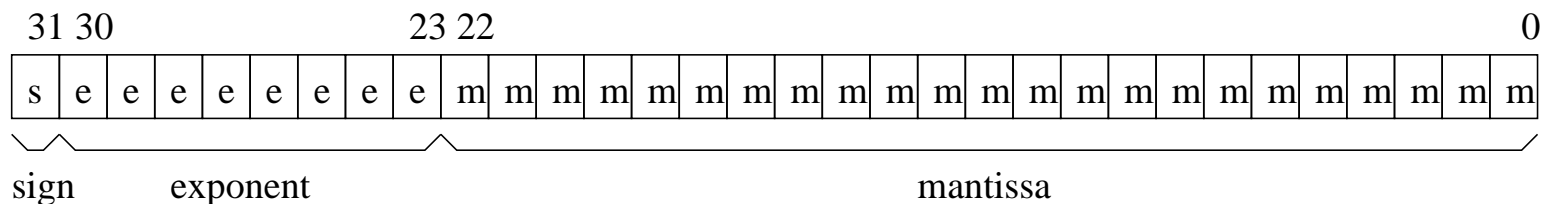
- from [O'H&Bryant, sect 2.4.3, fig 2.22]: representable values in an 8-bit f.p. format



- not all numbers can be represented:
  - overflow: numbers bigger than  $\text{maximum\_mantissa} \times 2^{\text{maximum\_exponent}}$
  - underflow: numbers smaller than  $\text{minimum\_mantissa} \times 2^{\text{minimum\_exponent}}$
- rounding errors can & do accumulate to become significant
  - e.g. Patriot Missile System

## IEEE 754 Single Precision - Details

- mantissa ( $m$ ) is represented in 23 bits
  - since a normalized form is used, the leading '1.' is not represented explicitly
  - thus we have a precision of  $2^{-24} \approx 10^{-7}$
- the exponent ( $e$ ) is stored in 8 bits, with 127 added to the actual value  
(thus  $e$  is an unsigned number, representing values from  $-127$  to 128)
- thus, the number represented is  $-1^s \times 1.m \times 2^{e-127}$



- what *tradeoffs* are involved here?
- zero is represented by a special bit pattern (of all 0's) – why is this important?
- e.g.  $1.0 = -1^0 \times 1.00\dots0 \times 2^{127-127}$  has  $s = 0_2$ ,  $m = 000\dots0_2$ ,  $e = 0111\ 1111_2$ ;  
binary: 0011111110000...;hex: 3F800000  
-2.5 =?