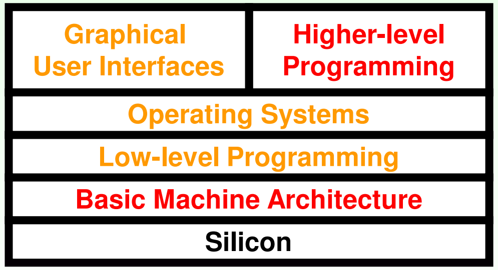
CSCU9V4
Systems

Systems Lecture 5

Floating Point Numbers



Fractional numbers

- How can we represent numbers with fractional parts (like 1/2, 0.1, 1.00005) when we have a discrete representation?
- There are two main methods
  - Store decimal digits to some specified number of decimal places (fixed for the variable)
    - 0000000000000000000000000000000000.00000000000 00000000000000009
    - 2000000000000000000000000000000000.00000000000 00000000000000000
    - Called **fixed point numbers**
  - Store most significant bits, and remember (separately) the magnitude of the value currently being held. A Java example:
    - float x ;
      // x can hold number of metres to the moon,
      // or diameter of a nerve fibre in metres
    - Called **floating point numbers** – the decimal point moves...
- The second method uses normalised numbers, and is more common

$9x10^{-28}$

$2x10^{+33}$

Normalised numbers

- A normalised number is one where the decimal point has been moved, to be after (usually) the first digit. Separately, we remember how far we had to move the decimal point. We call this *floating point notation*
    - 123,000 = + 1.230×10^{5}
      -0.0001234 = - 1.234×10^{-4}
      +31.423 $\cong$ + 3.142×10^{1}
- Sometimes called Scientific Notation (scientists use it…)
    - electron mass is 9×10^{-28} grams
    - Mass of sun is $2 \times 10^{+33}$ grams
- So, the representation of a real number contains three parts, each a bit pattern:
    - The overall sign (+ or -)
    - The mantissa (1230, 1234 or 3142) (9, 2 above)
    - The exponent (5 or -4 or 1) (-28, +33 above)
        - For reasons that do not concern us, the exponent is stored in excess notation.

Normalised Fraction (Tanenbaum)

Slide not used

exponent

mantissa *or* **fraction**

Example 1: Exponentiation to the base 2

2^{-1} 2^{-2} 2^{-3} 2^{-4} 2^{-5} 2^{-6} 2^{-7} 2^{-8} 2^{-9} 2^{-10} 2^{-11} 2^{-12} 2^{-13} 2^{-14} 2^{-15} 2^{-16}

Unnormalized: $\underbrace{0}_{\text{Sign} +}\ \underbrace{1010100}_{\text{Excess 64 exponent is } 84-64=20} . \underbrace{0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1}_{\text{Fraction is } 1\times2^{-12}+1\times2^{-13}+1\times2^{-15}+1\times2^{-16}}$ $= 2^{20}(1\times2^{-12}+1\times2^{-13}+1\times2^{-15}+1\times2^{-16}) = 432$

To normalize, shift the fraction left 11 bits and subtract 11 from the exponent.

Normalized: $\underbrace{0}_{\text{Sign} +}\ \underbrace{1001001}_{\text{Excess 64 exponent is } 73-64=9} . \underbrace{1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0}_{\text{Fraction is } 1\times2^{-1}+1\times2^{-2}+1\times2^{-4}+1\times2^{-5}}$ $= 2^{9}(1\times2^{-1}+1\times2^{-2}+1\times2^{-4}+1\times2^{-5}) = 432$

Example 2: Exponentiation to the base 16

16^{-1} 16^{-2} 16^{-3} 16^{-4}

Unnormalized: $\underbrace{0}_{\text{Sign} +}\ \underbrace{1000101}_{\text{Excess 64 exponent is } 69-64=5} . \underbrace{0000\ \ 0000\ \ 0001\ \ 1011}_{\text{Fraction is } 1\times16^{-3}+\text{B}\times16^{-4}}$ $= 16^{5}(1\times16^{-3}+\text{B}\times16^{-4}) = 432$

To normalize, shift the fraction left 2 hexadecimal digits, and subtract 2 from the exponent.

Normalized: $\underbrace{0}_{\text{Sign} +}\ \underbrace{1000011}_{\text{Excess 64 exponent is}} . \underbrace{0001\ \ 1011\ \ 0000\ \ 0000}_{\text{Fraction is } 1\times16^{-1}+\text{B}\times16^{-2}}$ $= 16^{3}(1\times16^{-1}+\text{B}\times16^{-2}) = 432$

Floating Point

- Notice that the mantissa is always held to a given *precision*
  - so at the right-hand end it is zero-filled or truncated
- eg 0.23000×10^2

  range

  precision

- Generally to represent a real number 32, 64, or even 128 bits would be used. The more bits the greater the precision

- If the exponent is too big to be stored we get *floating-point overflow.*

- This (should) always cause an error that the programmer must take notice of (unlike integer overflow).

Floating Point in Binary

- Let's use 8 bits of storage as an example…

Bit positions

87654321

01101011

range

Sign bit

Exponent

Mantissa

precision

In this example we follow standard mathematical practice and assume a normalised number where a 1 always precedes the decimal place, i.e. leading 1. is assumed. This is **similar to** normalised IEEE numbers which also assume a one *followed* by the decimal place that is, the 1. is assumed.

Sign = 0 is positive; Exponent = 2 (excess 4); Mantissa = 1.1011

1. is assumed

Number = 110.11 which is **6.75** in decimal

4

2

0.5

0.25

Effects of Floating-point Representation

- Only a restricted set of numbers are expressible - from the last slide
  - the length of the binary fraction to 4,
  - and the exponent to 3, (excess 4)
  - with 1 bit for a sign bit (8 bits in all).
- Then the largest number we could express would be: 1.1111×2^3
- and the next largest would be 1.1110×2^3, etc.
- Why excess 4, and not 2's complement?
  - Value of 0 is interpreted as -4, except when mantissa is also 0
  - Then number (all 0's) is interpreted as 0.
- No continuum with floating point numbers
  - With real numbers, another real number always fits between two real numbers, no matter how close they are
  - Precision is limited with floating point numbers
- This is called rounding
- This affects accuracy and causes underflow.

Floating point and Real numbers

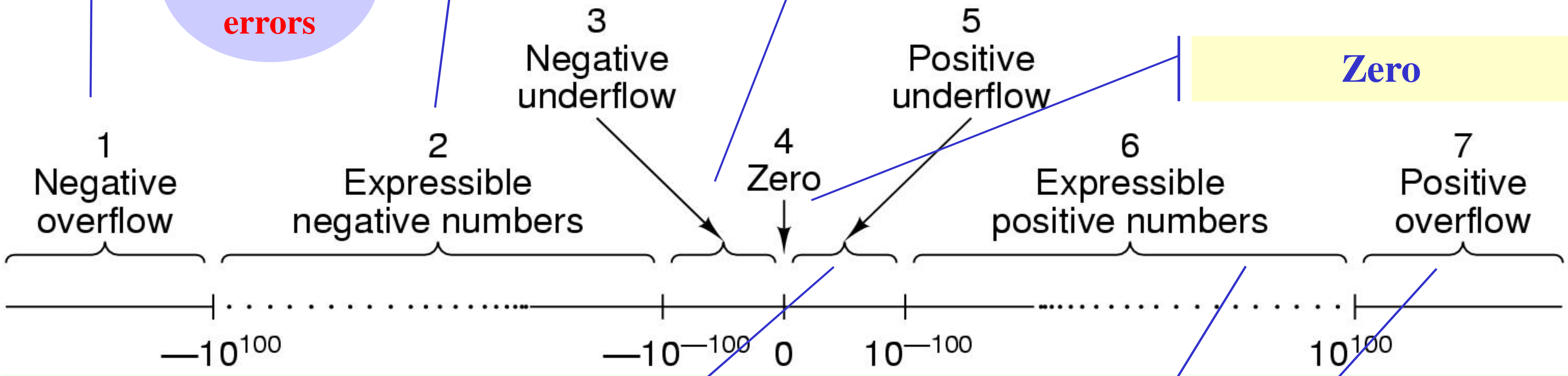
Large negative numbers less than -0.999×10^{99}

Negative numbers between -0.999×10^{99} and -0.100×10^{-99}

Note rounding errors

Small negative numbers with magnitudes less than 0.100×10^{-99}

Zero



Small positive numbers with magnitudes less than 0.100×10^{-99}

Positive numbers between 1.000×10^{-99} and 0.999×10^{99}

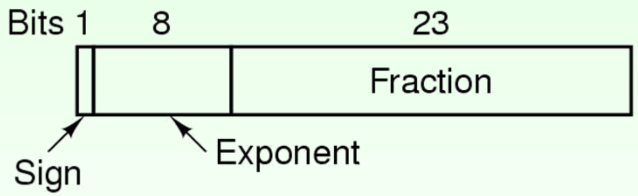
Large positive numbers greater than 0.999×10^{99}

- consider an example with
  - 3 decimal digit fraction
  - 2 decimal digit expon.

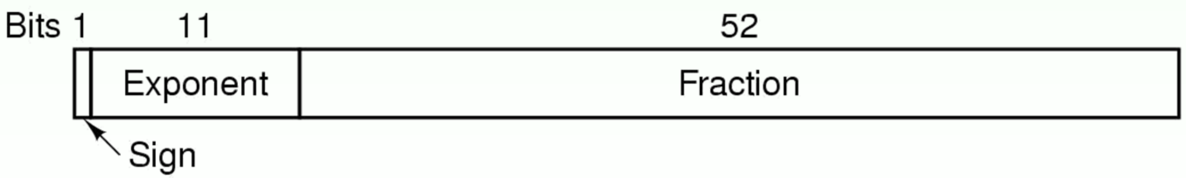
IEEE Floating-Point Standard 754

- There are 3 formats defined:
  - 32 bit (single precision), 64 bit (double precision), 80 bit (ext. precision)
  - 80 bit is only used inside floating point units (do not consider this further)

32-bit	64-bit
1 sign bit	1 sign bit
8 exponent bits (excess 127)	11 exponent bits (excess 1023)
23 fraction bits	52 fraction bits



(a)



(b)

IEEE Floating-Point Standard 754

- The binary fraction
  - is normalised so the leftmost bit is always one (exponent adjusted)
  - the leading bit is omitted and simply assumed to be present
  - The dot is omitted and assumed to be present
  - If all 23 or 52 bits are 0, the fraction has the numerical value of 1.0
  - If all the bits are 1, the fraction is numerically slightly less than 2

- The exponent uses
  - 'excess 127' representation (32-bit case), 'excess 1023' (64-bit case).
    - Allows coding of negative exponents
    - That is: to represent N, we use the binary representation of 127 + N (or 1023 + N).
  - In addition, the bit patterns all 0's and all 1's (255, 2047) are not used and have special meanings.
  - In the 32-bit case we can have exponents of 2^{-126} (00000001) to 2^{+127} (11111110).

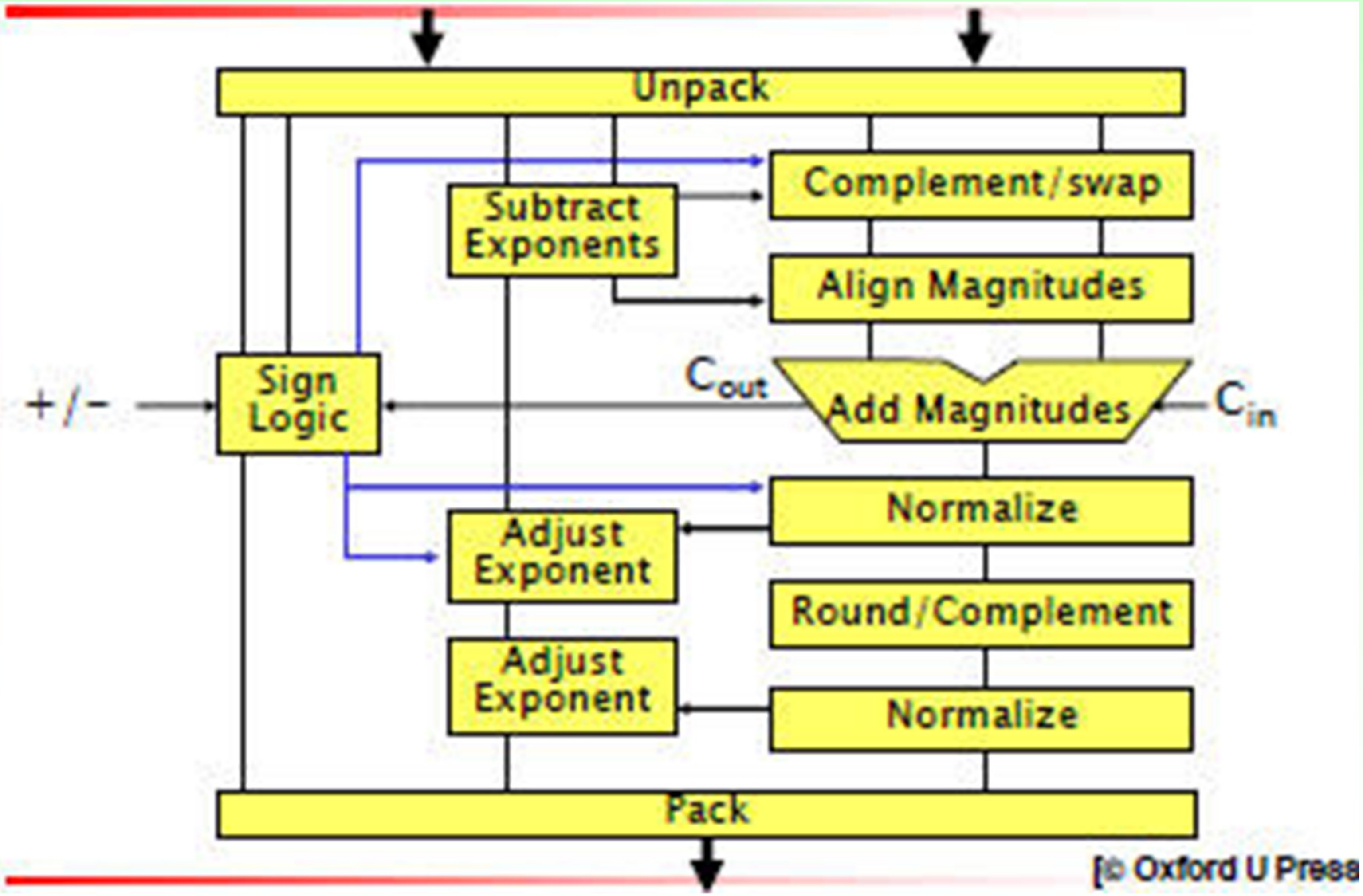
Numerical Types in IEEE 754

- A problem with floating point numbers is how to deal with underflow, overflow, and un-initialised numbers
- In addition to normalised numbers, IEEE 754 introduces 4 other numerical types
- Denormalised (handle underflow):
  - exponent is 0, assumed bit is not 1 but 0,
  - The smallest normalised number is 1.0×2^{-126} (1 as exponent, 0 as fraction)
  - The biggest denormalised number is $0.9999999 \times 2^{-127}$ (0 as exponent, all 1 as fraction), 23 significant bits (24 for normalised)

Normalized	±	0 < Exp < Max	Any bit pattern
Denormalized	±	0	Any nonzero bit pattern
Zero	±	0	0
Infinity	±	1 1 1…1	0
Not a number	±	1 1 1…1	Any nonzero bit pattern

Sign bit

Floating point adder



Notes on Using Floating-point Numbers

- Be very wary of using *equality* in Boolean conditions.
  - intuitive expressions are frequently wrong, e.g.

    0.05 == 1.0/20.0 is FALSE!

- Instead of using equality in tests,
  - use **<=** or **>=** as appropriate.

- Watch out for (i.e. test for) overflow (result too large to be represented), and underflow (result too near zero to be represented).
  - e.g. if **x** is small, take care with expressions like **N/x**.

- Values are always approximations. Multiplying **x** by a number greater than 1 will inherently magnify any error in **x**.

Floating point numbers in Java

- The two floating-point types in Java are `float` and `double`

Type	Size	Largest	Smallest	Precision
float	32 bits	$\pm 3.4*10^{38}$	$\pm 1.4*10^{-45}$	6-7 digits
double	64 bits	$\pm 1.79*10^{308}$	$\pm 4.94*10^{-324}$	14-15 digits

Floating point numbers in Java (and C)

- Java provides error-handling for floating point numbers in its `java.lang.Float` and `java.lang.Double` classes

- Some of the facilities provided:
  - positive infinity
    - test whether the resulting number was too large
  - negative infinity
  - not-a-number
    - the result of division by zero

- C DOES NOT! (beyond 'NaN')

End of Lecture