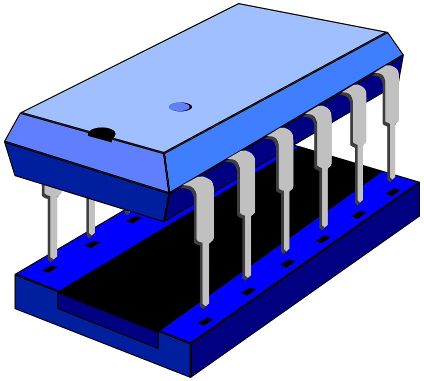

FLOATING POINT NUMBERS

ing point standard

Bernhard Kainz (with thanks to **N. Dulay** and **E. Edwards**)

b.kainz@imperial.ac.uk

IEEE floating point standard

- IEEE: **institute of electrical and electronic engineers (USA)**
- **Comprehensive standard** for binary floating point arithmetic
- Widely adopted independent of architecture
- Standard defines:
  - **Format** of binary floating point numbers, i.e. how the fields are stored in memory
  - **Semantics** of arithmetic operations
  - Rules for **error conditions**

Single precision format (32-bit)

Sign S	Exponent E	Significand F
1 bit	8 bits	23 bits

- Coefficient is ca he IEEE standard
- Value represented is $\pm \mathbf{1}.\boldsymbol{F} \times \mathbf{2}$

- The **normal bit** (the 1.) is omit e significand field ➔ a **hidden bit**
- Single precision yields **24 bits** (approx. **7 decimal digits** of precision)
- **Normalised ranges** in decimal are approximately:

$$-\mathbf{10^{38}}\ \textbf{to}\ -\mathbf{10^{-38}}, \quad \mathbf{0}, \quad \mathbf{10^{38}}\ \textbf{to}\ \mathbf{10^{-38}}$$

Exponent field

- In the IEEE standard, exponents are stored as **excess values**, not as 2's complement

- Example: **In 8**

-1	**0000**
…	
0	**111**
1	**1000 0000**
…	…
128	**1111 1111**

- Allows non-negative floating point numbers to be compared using simple integer comparisons

Double precision format (64-bit)

Sign S	Exponent E	Significand F
1 bit	11 bits	52 bits

- Value represent
- Double precision yields **53 bit** **6 decimal digits** of precision)
- **Normalised ranges** in decimal are approximately:

$$\mathbf{-10^{308}\ \text{to}\ -10^{-308},\quad 0,\quad 10^{308}\ \text{to}\ 10^{-308}}$$

- Single precision generally reserved for when memory is scarce or for debugging numerical calculations since rounding errors show up more quickly

Example: conversion **to** IEEE format

What is 42.6875 in IEEE single precision format?

1. Convert to **binary number**: $42.6875 = 101010.1011$
2. **Normalise**: 1.010101011×2^5
3. **Significand field**: $0101\,0101\,1000\,0000\,0000\,000$
4. **Exponent field** is $(5 + 127 = 132)$: $1000\,0100$

Sign S	Exponent E	Significand F
0	1000 0100	0101 0101 1000 0000 0000 000

Hex: 422A C000

Example: conversion **from** IEEE format

What is the IEEE single precision value represented by **BEC0 0000** in decimal?

Sign S	Exponent E	Significand F
1	0	0.0000 0000 000

1. **Exponent field:** $1 = 125$
2. True **binary exponent:** $7 = -2$
3. **Significand field + hidden bit:**
   $1.1000\ 0000\ 0000\ 0000\ 0000\ 000$
4. So **unsigned value** is $1.1 \times 2^{-2} = 0.011$ (binary)
   $= 0.25 + 0.125 = 0.375$ (decimal)
5. Adding **sign bit** gives finally $\mathbf{-0.375}$

Example: addition

Carry out the addition $\mathbf{42.6875} + \mathbf{0.375}$ in IEEE single precision arithmetic

Number	Sign	Significand
42.6875	0	01 1000 0000 0000 0000 000
0.375	0	00 0000 0000 0000 0000 000

- To add these numbers, expon e the same ➔ make the smaller exponent equal to the larger by shifting significand accordingly
- **Note:** must restore **hidden bit** when carrying out floating point operations

Example: addition (cont.)

- **Significand** of larger no.: $1.0101\ 0101\ 1000\ 0000\ 0000\ 000$
- **Significand** of smaller no.: $1.1000\ 0000\ 0000\ 0000\ 0000\ 000$

- Exponents differ by ($1000\ 0100\ - 0111\ 1101 = 7$) so shift binary point of smaller no.

- **Significand** of smaller no.: 0.000 [illegible] $0000\ 0000\ 000$
- **Significand** of larger no.: $1.0101\ 0101\ 1000\ 0000\ 0000\ 000$
- **Significand** of **sum**: $1.0101\ 1000\ 1000\ 0000\ 0000\ 000$

- So **sum** is $1.0101\ 1000\ 1 \times 2^5 = 10\ 1011.0001 = 43.0625$

Sign S	Exponent E	Significand F
0	1000 0100	0101 1000 1000 0000 0000 000

Special values

- IEEE formats can encode five kinds of values: **zero**, **normalised numbers**, **denormalised numbers**, **infinity** and **not-a-number (NaNs)**

- Single precision

IEEE value	S field			True exponent
± 0	0 or 1	0	0 (all zeros)	
$\pm$ denormalised no.	0 or 1	0	Any non-zero bit pattern	-126
$\pm$normalised no.	0 or 1	$1 \dots 254$	Any bit pattern	$-126 \dots 127$
$\pm\infty$	0 or 1	255	0 (all zeros)	
Not-a-number	0 or 1	255	Any non-zero bit pattern	

Denormalised numbers

- An **all zero exponent** is used to represent both **zero** and **denormalised numbers**
- An **all one exponent** is used to represent **infinities** and **not-a-numbers**

- Means **range fo** **is reduced**, for single precision the exponent r $26 \ldots 127$ rather than $-127 \ldots 128$

- **Denormalised numbers** represent values between the underflow limits and zero, i.e. for single precision we have $\pm\, 0.F \times 2^{-126}$
- Allows a more **gradual shift to zero** – useful in some numerical applications

Infinities and NaNs

- Infinities represent values **exceeding the overflow limits** and for divisions of non-zero quantities by zero
- You can do basic 'arithmetic' with them, e.g.:

$$\infty \qquad = \infty$$

- NaNs represent which have **no (real) mathematical interpret**

$$\frac{0}{0}, \quad +\infty + -\infty, \quad 0 \times \infty, \quad \text{square root of a negative number}$$

- Operations resulting in NaNs can either yield a NaN result (**quiet NaN**) or an exception (**signalling NaN**)

Special Operations

Operation	Result
N ÷ ± Infinity	0
± Infinity × [illegible]	[illegible]ity
± non-z[illegible]	[illegible]ity
Infinity + Infinity	[illegible]
± 0 ÷ ± 0	[illegible]
Infinity - Infinity	*NaN*
± Infinity ÷ ± Infinity	*NaN*
± Infinity × 0	*NaN*

☺ N ☺

Floating Point Precision

- C code:

```
#include <stdio.h>
int main() {

  float a, b, c;

  float EPSILON =

  a = 1.345f;  b = 1.123f;

  c = a + b;

  if (c == 2.468)
    printf ("They are equal.\n");
  else
    printf ("\nThey are not equal! The value of c is %.10f or %f\n",c,c);

 // With some tolerance

 if (((2.468 - EPSILON) < c) && (c  < (2.468 + EPSILON)))
   printf ("\n%.10f is equal to 2.468 with tolerance\n\n", c);
}
```

Run-time

Finding Machine Epsilon

- Pseudo-code

```
Set machineEps = 1.0;

Loop

    machineEps = machineE

Until ((1 + machineEps/2.0) != 1)

Print machineEps
```

Finding Machine Epsilon

- C code

```
#include <stdio.h>

int main( int argc, char **argv )
{
  float machEp

  do {
    machEps /= 2.0f;
    // If next epsilon yields 1, then break, because current
    // epsilon is the machine epsilon.
  }
  while ((float)(1.0 + (machEps/2.0f)) != 1.0);

  printf( "\nCalculated Machine epsilon: %G\n\n", machEps );
  return 0;
}
```

Finding Machine Epsilon

- In Java

```
public class machEps
{
  private static void calculateMachineEpsilonFloat() {
        float machE

        do {
           machEps /= 2.0f;
        } while ((float)(1.0 + (machEps

        System.out.println( "Calculated machine epsilon: " + machEps );
  }

  public static void main (String args[])
  {
        calculateMachineEpsilonFloat ();
  }
}
```

Run-time

Special Operations

- Example

```
#include <stdio.h>

int main (int argc, char **argv)
{
  float a =

  float b = a * -100;

  float c = b/a;

  int d = 2 * 10 + 3;

  printf ("\nValue of a = %f\n\n", a);
  printf ("\nValue of b = %f\n\n", b);
  printf ("\nValue of c = %f\n\n", c);
  printf ("\nValue of d = %d\n\n", d);
}
```

Run-time