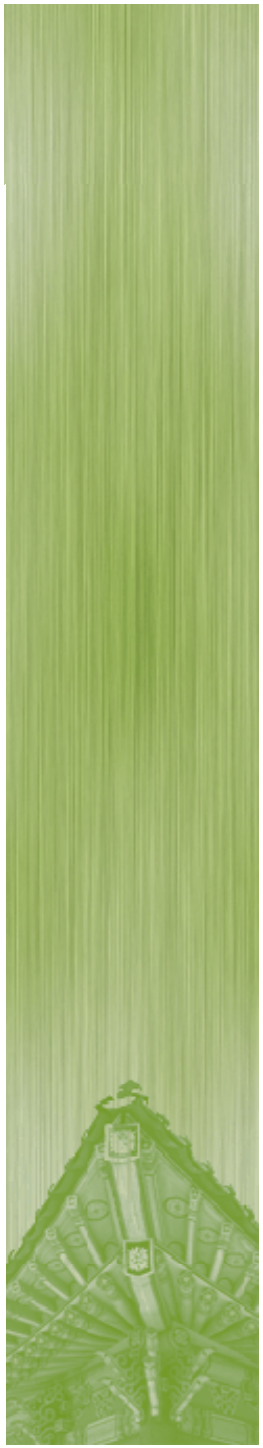


# Representing and Manipulating Floating Points

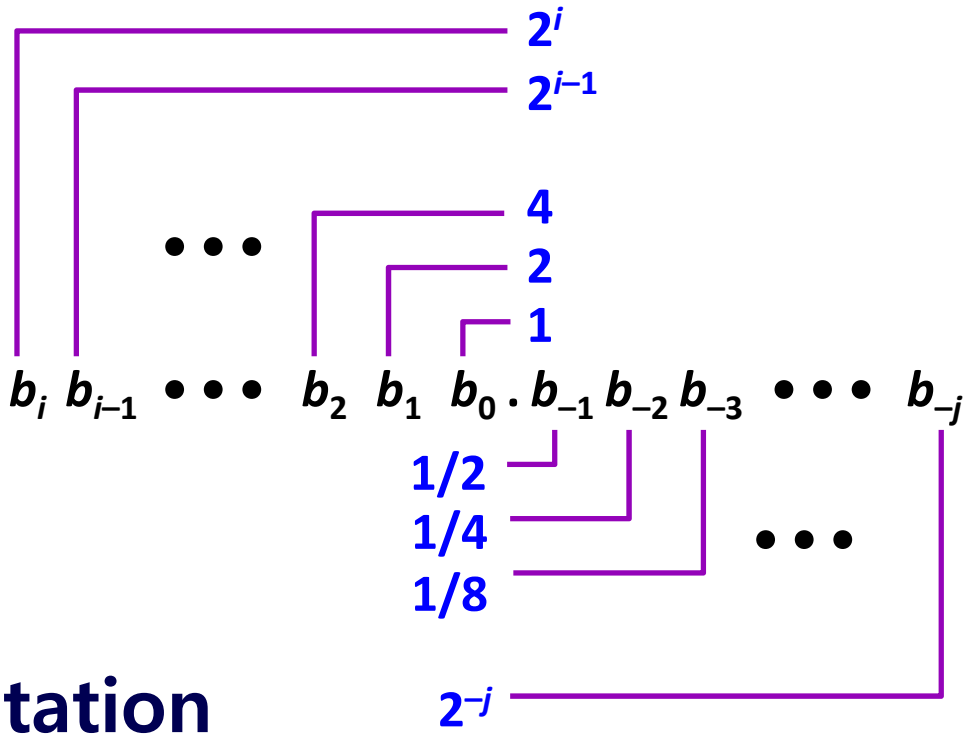
Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

SKK UNIVERSITY

# The Problem

- **How to represent fractional values with finite number of bits?**
  - 0.1
  - 0.612
  - 3.14159265358979323846264338327950288…

# Fractional Binary Numbers (1)




- **Representation**
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \cdot 2^k$

# Fractional Binary Numbers (2)

- **Examples:**

| Value | Representation |
|---|---|
| 5-3/4 | $101.11_2$ |
| 2-7/8 | $10.111_2$ |
| 63/64 | $0.111111_2$ |

- **Observations**
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $\mathbf{0.111111\ldots_2}$ just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

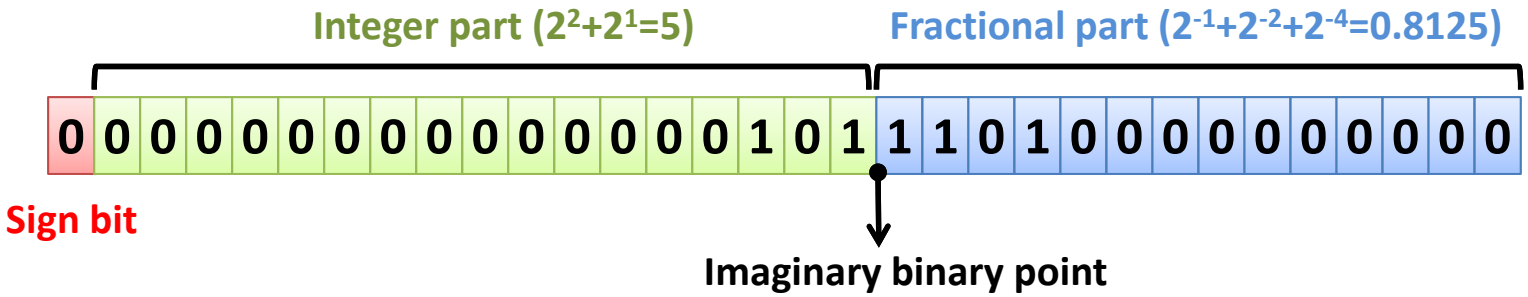
# Fractional Binary Numbers (3)

- **Representable numbers**
  - Can only exactly represent numbers of the form $x/2^k$
  - Other numbers have repeating bit representations

| Value | Representation |
|---|---|
| 1/3 | $0.0101010101[01]\ldots_2$ |
| 1/5 | $0.001100110011[0011]\ldots_2$ |
| 1/10 | $0.0001100110011[0011]\ldots_2$ |

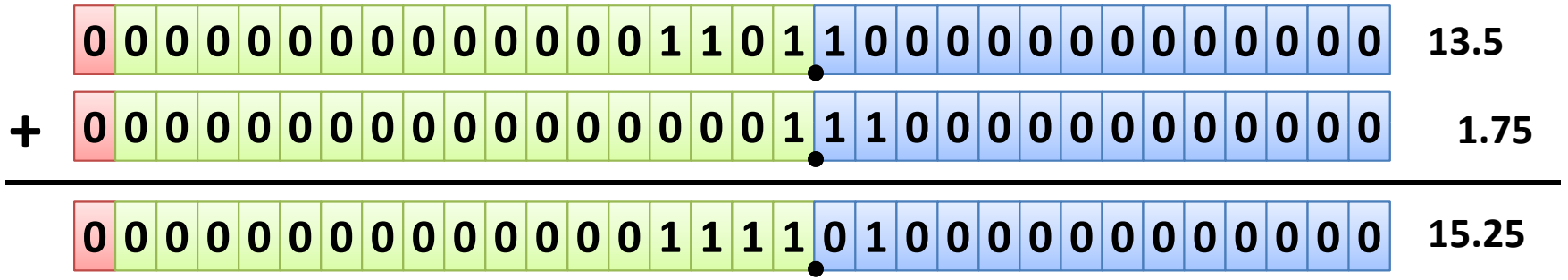
# Fixed-Point Representation (1)

- ***p.q* Fixed-point representation**
  - Use the rightmost ***q*** bits of an integer as representing a fraction
  - Example: 17.14 fixed-point representation
    - 1 bit for sign bit
    - 17 bits for the integer part
    - 14 bits for the fractional part
    - An integer ***x*** represents the real number ***x*** / $2^{14}$
    - Maximum value: ($2^{31}$ - 1) / $2^{14}$ ≒ 131071.999



# Fixed-Point Representation (2)

- **Properties**
  - Convert ***n*** to fixed point: ***n*** * ***f***
  - Add ***x*** and ***y*** : ***x*** + ***y***




  - Subtract ***y*** from ***x*** : ***x*** - ***y***
  - Add ***x*** and ***n*** : ***x*** + ***n*** * ***f***
  - Multiply ***x*** by ***n*** : ***x*** * ***n***
  - Divide ***x*** by ***n*** : ***x*** / ***n***

***x***, ***y***: fixed-point number
***n***: integer
***f*** = 1 << ***q***

# Fixed-Point Representation (3)

- **Pros**
  - Simple
  - Can use integer arithmetic to manipulate
  - No floating-point hardware needed
  - Used in many low-cost embedded processors or DSPs (digital signal processors)

- **Cons**
  - Cannot represent wide ranges of numbers
    - 1 Light-Year = 9,460,730,472,580.8 km
    - The radius of a hydrogen atom: 0.000000000025 m

# Representing Floating Points

- **IEEE standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs
  - William Kahan, a primary architect of IEEE 754, won the Turing Award in 1989.
  - Driven by numerical concerns
    - Nice standards for rounding, overflow, underflow
    - Hard to make go fast
    - Numerical analysts predominated over hardware types in defining standard.

# FP Representation

- **Numerical form: $-1^s \times M \times 2^E$**
  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0)
  - Exponent E weights value by power of two
- **Encoding**


  - MSB is sign bit
  - **exp** field encodes E (Exponent)
  - **frac** field encodes M (Mantissa)

# FP Precisions

- **Encoding**


  - MSB is sign bit
  - **exp** field encodes E (Exponent)
  - **frac** field encodes M (Mantissa)
- **Sizes**
  - Single precision: 8 **exp** bits, 23 **frac** bits (32bits total)
  - Double precision: 11 **exp** bits, 52 **frac** bits (64bits total)
  - Extended precision: 15 **exp** bits, 63 **frac** bits
    - Only found in Intel-compatible machines
    - Stored in 80 bits (1 bit wasted)

# Normalized Values (1)

- **Condition: exp ≠ 000…0 and exp ≠ 111…1**
- **Exponent coded as biased value**
  - ***E = Exp – Bias***
  - ***Exp*** : unsigned value denoted by **exp**
  - ***Bias*** : Bias value
    - Single precision: 127 (***Exp*** : 1..254, ***E*** : -126..127)
    - Double precision: 1023 (***Exp*** : 1..2046, ***E*** : -1022..1023)
- **Significand coded with implied leading 1**
  - ***M* = 1.xxx…$x_2$**
    - Minimum when 000…0 (***M*** = 1.0)
    - Maximum when 111…1 (***M*** = 2.0 - ε)
  - Get extra leading bit for "free"

# Normalized Values (2)

- **Value:** float f = 2003.0;

  $2003_{10}$ = $11111010011_2$ = $1.1111010011_2 \times 2^{10}$

- **Significand**

  *M* = $1.\underline{1111010011}_2$

  frac = $\underline{1111010011}0000000000000_2$

- **Exponent**

  *E* = 10

  *Exp* = *E + Bias* = 10 + 127 = 137 = $10001001_2$

| Floating Point Representation: | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Hex: | 4 | 4 | F | A | 6 | 0 | 0 | 0 |
| Binary: | 0100 | 0100 | 1111 | 1010 | 0110 | 0000 | 0000 | 0000 |
| 137: | 100 | 0100 | 1 | | | | | |
| 2003: | | | 1111 | 1010 | 0110 | | | |

# Denormalized Values

- **Condition: exp = 000…0**
- **Value**
  - Exponent value ***E* = 1 – *Bias***
  - Significand value **M = 0.xxx…$x_2$** (no implied leading 1)
- **Cases**
  - **exp = 000…0, frac = 000…0**
    - Represents value 0
    - Note that have distinct values +0 and -0
  - **exp = 000…0, frac ≠ 000…0**
    - Numbers very close to 0.0
    - "Gradual underflow": possible numeric values are spaced evenly near 0.0

# Special Values

- **Condition: exp = 111…1**
- **Cases**
  - **exp = 111…1, frac = 000…0**
    - Represents value ∞ (infinity)
    - Operation that overflows
    - Both positive and negative
    - e.g. 1.0/0.0 = -1.0/-0.0 = +∞, 1.0/-0.0 = -∞
  - **exp = 111…1, frac ≠ 000…0**
    - Not-a-Number (NaN)
    - Represents case when no numeric value can be determined
    - e.g., sqrt(-1), ∞ - ∞, ∞ * 0, …

# Tiny FP Example (1)

- **8-bit floating point representation**
  - The sign bit is in the most significant bit
  - The next four bits are the **exp**, with a bias of 7
  - The last three bits are the **frac**

- **Same general form as IEEE format**
  - Normalized, denormalized
  - Representation of 0, NaN, infinity

# Tiny FP Example (2)

- Values related to the exponent (*Bias* = 7)

| Description | Exp | exp | E = Exp - Bias | $2^E$ |
|---|---|---|---|---|
| Denormalized | 0 | 0000 | -6 | 1/64 |
| Normalized | 1 | 0001 | -6 | 1/64 |
| | 2 | 0010 | -5 | 1/32 |
| | 3 | 0011 | -4 | 1/16 |
| | 4 | 0100 | -3 | 1/8 |
| | 5 | 0101 | -2 | 1/4 |
| | 6 | 0110 | -1 | 1/2 |
| | 7 | 0111 | 0 | 1 |
| | 8 | 1000 | 1 | 2 |
| | 9 | 1001 | 2 | 4 |
| | 10 | 1010 | 3 | 8 |
| | 11 | 1011 | 4 | 16 |
| | 12 | 1100 | 5 | 32 |
| | 13 | 1101 | 6 | 64 |
| | 14 | 1110 | 7 | 128 |
| inf, NaN | 15 | 1111 | - | - |

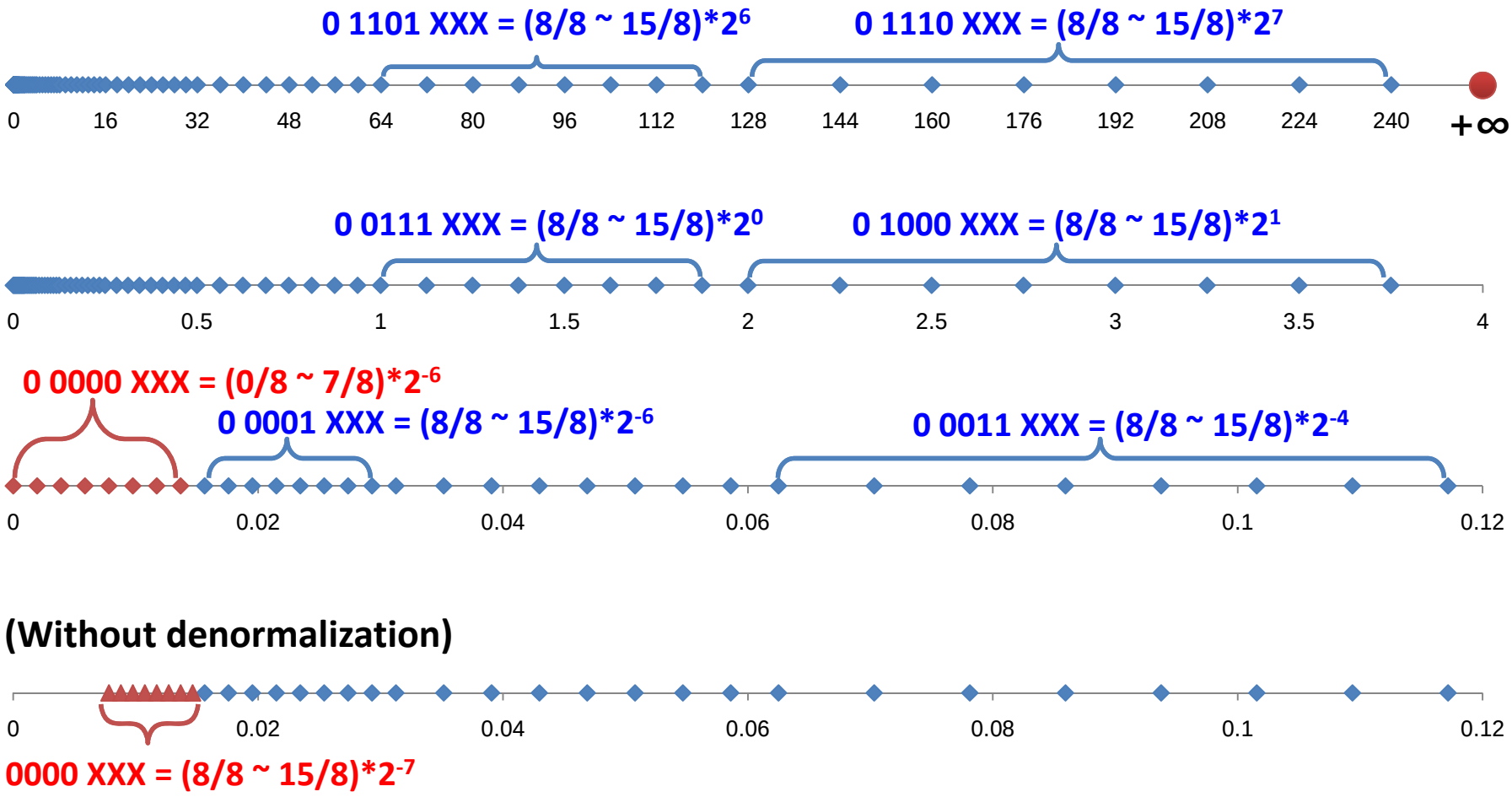
# Tiny FP Example (3)

- Dynamic range

| Description | Bit representation | e | E | f | M | V |
|---|---|---|---|---|---|---|
| Zero | 0 0000 000 | 0 | -6 | 0 | 0 | 0 |
| Smallest pos. | 0 0000 001 | 0 | -6 | 1/8 | 1/8 | 1/512 |
| | 0 0000 010 | 0 | -6 | 2/8 | 2/8 | 2/512 |
| | 0 0000 011 | 0 | -6 | 3/8 | 3/8 | 3/512 |
| | 0 0000 110 | 0 | -6 | 6/8 | 6/8 | 6/512 |
| Largest denorm. | 0 0000 111 | 0 | -6 | 7/8 | 7/8 | 7/512 |
| Smallest norm. | 0 0001 000 | 1 | -6 | 0 | 8/8 | 8/512 |
| | 0 0001 001 | 1 | -6 | 1/8 | 9/8 | 9/512 |
| | 0 0110 110 | 6 | -1 | 6/8 | 14/8 | 14/16 |
| | 0 0110 111 | 6 | -1 | 7/8 | 15/8 | 15/16 |
| One | 0 0111 000 | 7 | 0 | 0 | 8/8 | 1 |
| | 0 0111 001 | 7 | 0 | 1/8 | 9/8 | 9/8 |
| | 0 0111 010 | 7 | 0 | 2/8 | 10/8 | 10/8 |
| | 0 1110 110 | 14 | 7 | 6/8 | 14/8 | 224 |
| Largest norm. | 0 1110 111 | 14 | 7 | 7/8 | 15/8 | 240 |
| Infinity | 0 1111 000 | - | - | - | - | +∞ |

# Tiny FP Example (4)

- Encoded values (nonnegative numbers only)



# Interesting Numbers

| Description | exp | frac | Numeric Value |
|---|---|---|---|
| Zero | 000 … 00 | 000 … 00 | 0.0 |
| Smallest Positive denormalized | 000 … 00 | 000 … 01 | Single: $2^{-23}$ X $2^{-126}$ ≈ 1.4 X $10^{-45}$<br>Double: $2^{-52}$ X $2^{-1022}$ ≈ 4.9 X $10^{-324}$ |
| Largest Denormalized | 000 … 00 | 111 … 11 | Single: (1.0 – ε) X $2^{-126}$ ≈ 1.18 X $10^{-38}$<br>Double: (1.0 – ε) X $2^{-1022}$ ≈ 2.2 X $10^{-308}$ |
| Smallest Positive Normalized | 000 … 01 | 000 … 00 | Single: 1.0 X $2^{-126}$, Double: 1.0 X $2^{-1022}$<br>(Just larger than largest denormalized) |
| One | 011 … 11 | 000 … 00 | 1.0 |
| Largest Normalized | 111 … 10 | 111 … 11 | Single: (2.0 – ε) X $2^{127}$ ≈ 3.4 X $10^{38}$<br>Double: (2.0 – ε) X $2^{1023}$ ≈ 1.8 X $10^{308}$ |

# Special Properties

- **FP zero same as integer zero**
  - All bits = 0
- **Can (almost) use unsigned integer comparison**
  - Must first compare sign bits
  - Must consider -0 = 0
  - NaNs problematic
    - Will be greater than any other values
  - Otherwise OK
    - Denormalized vs. normalized
    - Normalized vs. Infinity

# Floating Point in C (1)

- **C guarantees two levels**
  - **float** (single precision) vs. **double** (double precision)
- **Conversions**
  - **double** or **float** → **int**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN
      - » Generally sets to TMin
  - **int** → **double**
    - Exact conversion, as long as **int** has ≤ 53 bit word size
  - **int** → **float**
    - Will round according to rounding mode

# Floating Point in C (2)

- **Example 1:**

```
#include <stdio.h>

int main ()  {
        int n = 123456789;
        int nf, ng;
        float f;
        double g;

        f = (float) n;
        g = (double) n;
        nf = (int) f;
        ng = (int) g;
        printf ("nf=%d ng=%d\n", nf, ng);
}
```

# Floating Point in C (3)

- **Example 2:**

```
#include <stdio.h>

int main ()  {
        double  d;

        d = 1.0 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
                + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;

        printf ("d = %.20f\n", d);
}
```

# Floating Point in C (4)

- **Example 3:**

```
#include <stdio.h>

int main ()  {
        float f1 = (3.14 + 1e20) – 1e20;
        float f2 = 3.14 + (1e20 – 1e20);

        printf ("f1 = %f, f2 = %f\n", f1, f2);
}
```

# Ariane 5

- **Ariane 5 tragedy (June 4, 1996)**
  - Exploded 37 seconds after liftoff
  - Satellites worth $500 million
- **Why?**
  - Computed horizontal velocity as floating point number
  - Converted to 16-bit integer
    - Careful analysis of **Ariane 4** trajectory proved 16-bit is enough
  - Reused a module from 10-year-old s/w
    - Overflowed for **Ariane 5**
    - No precise specification for the S/W

# Summary

- **IEEE floating point has clear mathematical properties**
  - Represents numbers of form $M \times 2^E$
  - Can reason about operations independent of implementation
    - As if computed with perfect precision and then rounded
  - Not the same as real arithmetic
    - Violates associativity/distributivity
    - Makes life difficult for compilers and serious numerical applications programmers