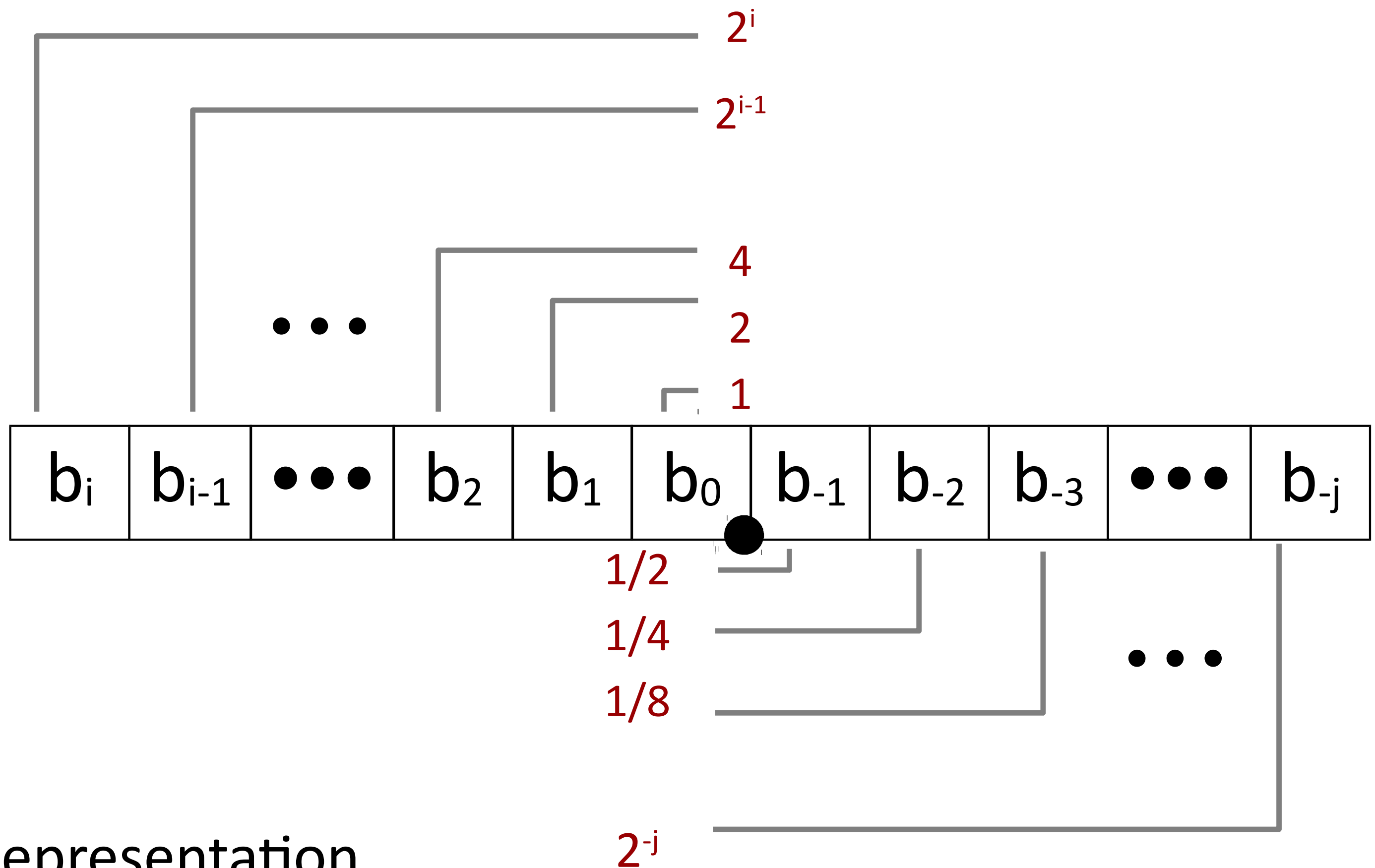
Floating Point Numbers

Lecture 1 - 2015
Mads Chr. Olesen

Credits to Alexandre David (AAU),
Randy Bryant & Dave O'Hallaron (CMU)

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \times 2^k$$

Fractional Binary Numbers: Examples

- Value Representation

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2
63/64	1.0111_2

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

Fixed point numbers available in some DSP.

PP2.45

Representable Numbers

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

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

IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats, emphasis on performance and ease of implementation, not precision
  - Supported by all major CPUs

- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

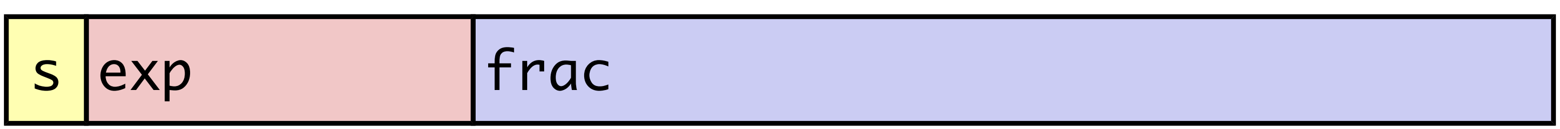
Floating Point Representation

- Numerical Form:

$$(-1)^s * M * 2^E$$

  - 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 `s` is sign bit s
  - `exp` field encodes E (but is not equal to E)
  - `frac` field encodes M (but is not equal to M)

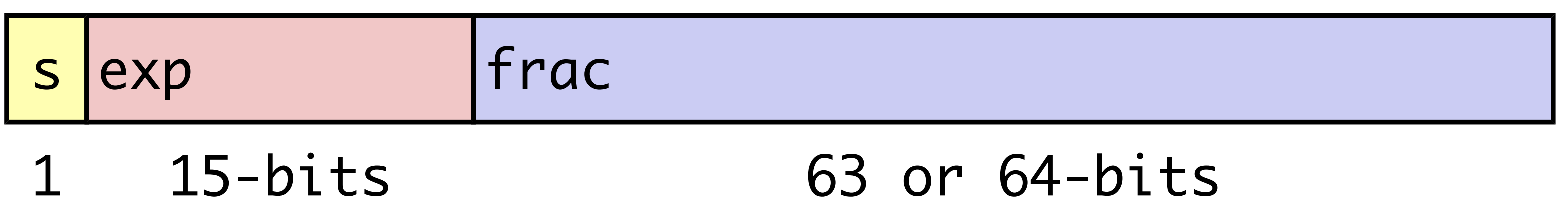


Precisions

- Single precision: 32 bits

- Double precision: 64 bits

- Extended precision: 80 bits (Intel only)



Normalized Values

- Condition: exp ≠ 000…0 and exp ≠ 111…1

- Exponent coded as biased value: E = Exp – Bias
  - Exp: unsigned value `exp`
  - Bias = 2^{k-1} - 1, where k is number of exponent bits
    - 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 = $1.xxx\ldots x_2$
  - xxx…x: bits of `frac`
  - Minimum when `000…0` (M = 1.0)
  - Maximum when `111…1` (M = 2.0 – ε)
  - Get extra leading bit for "free"

PP2.49

Normalized Encoding Example

- **Value:** `Float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**

  $M = 1.\underline{1101101101101}_2$

  `frac` $= \underline{1101101101101}0000000000_2$

- **Exponent**

  $E = 13$

  $Bias = 127$

  $Exp = 140 = 10001100_2$

- **Result:**

0	10001100	11011011011010000000000
s	exp	frac

Denormalized Values

- Condition: `exp = 000…0`

- Exponent value: E = –Bias + 1 (instead of E = 0 – Bias)
- Significand coded with implied leading 0: $M = 0.xxx...x_2$
  - `xxx…x`: bits of `frac`
- Cases
  - `exp` = `000…0`, `frac` = `000…0`
    - Represents zero value
    - Note distinct values: +0 and –0 (why?)
  - `exp` = `000…0`, `frac` ≠ `000…0`
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced

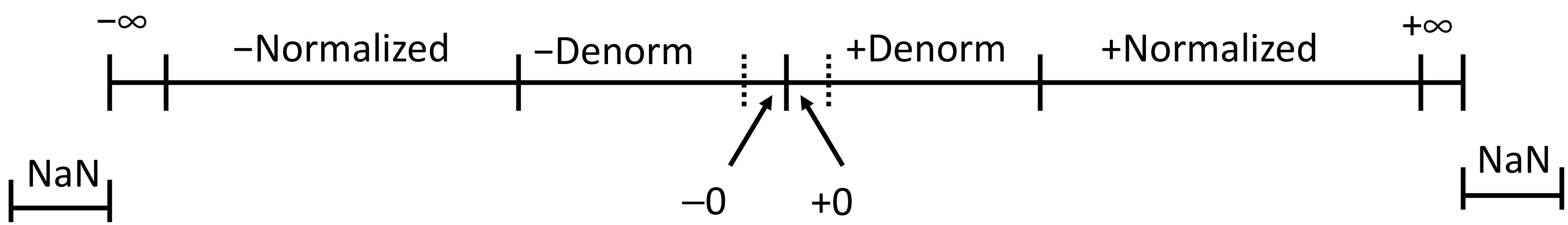
Special Values

- Condition: `exp` = `111…1`

- Case: `exp` = `111…1`, `frac` = `000…0`
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- Case: `exp` = `111…1`, `frac` $\neq$ `000…0`
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), $\infty - \infty$, $\infty \times 0$

Visualization: Floating Point Encodings



Summary

1. Normalized

s	$\neq 0$ & $\neq 255$	f

2. Denormalized

s	0	0	0	0	0	0	0	0	f

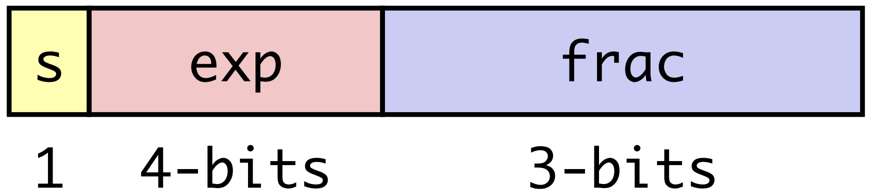
3a. Infinity

s	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3b. NaN

s	1	1	1	1	1	1	1	1	$\neq 0$

(Tiny Floating Point Example)

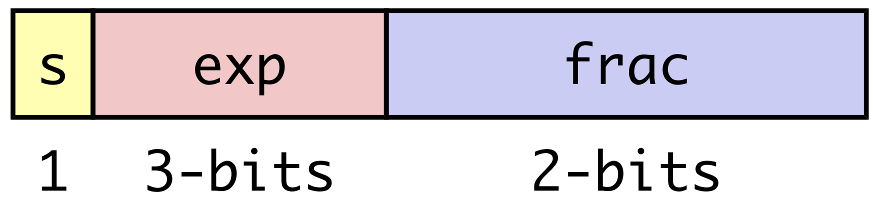


- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the `frac`

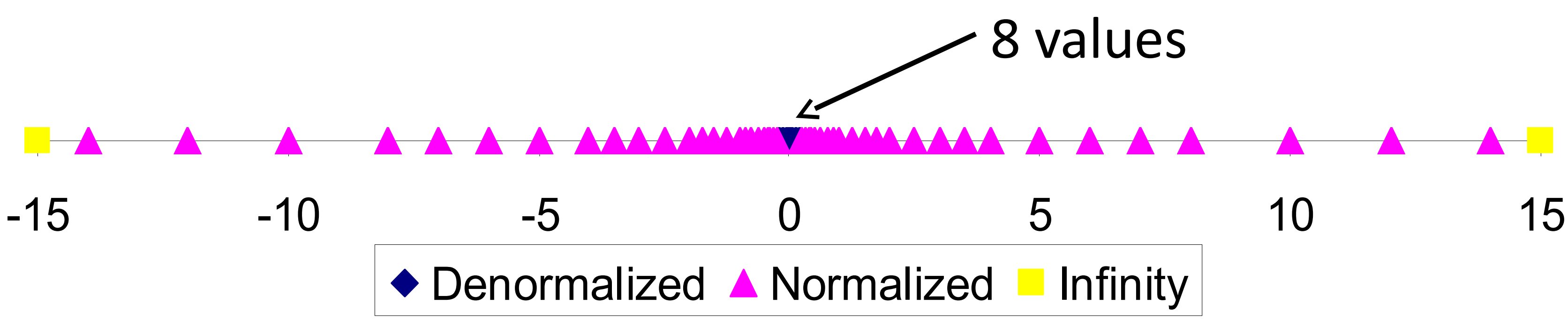
- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 23-1-1 = 3

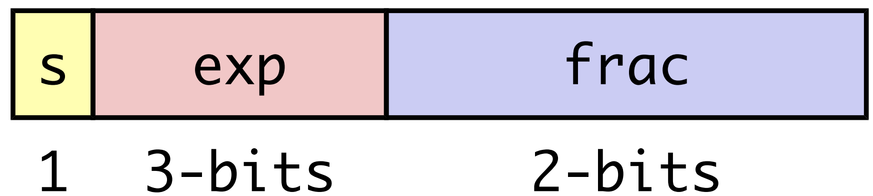


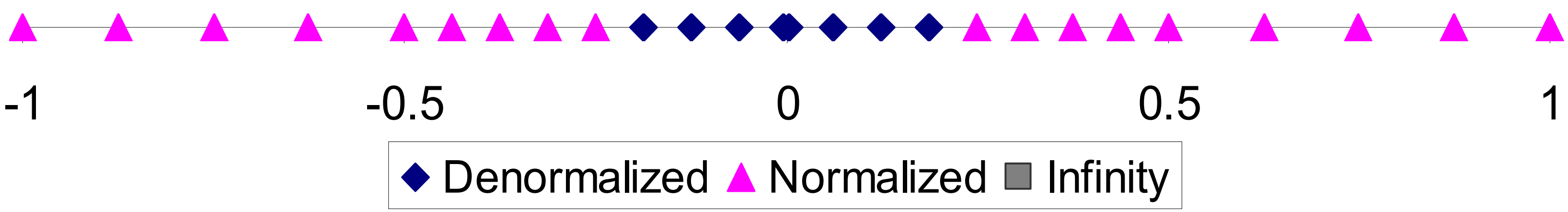
- Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3





Special Properties of Encoding

- 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
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Floating Point Operations: Basic Idea

- $x +_f y = \text{Round}(x + y)$

- $x \times_f y = \text{Round}(x \times y)$

- Basic idea
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`

Rounding

- Rounding Modes (illustrate with $ rounding)

	$1.40	$1.60	$1.50	$2.50	–$1.50
Towards zero	$1	$1	$1	$2	–$1
Round down (−∞)	$1	$1	$1	$2	–$2
Round up (+∞)	$2	$2	$2	$3	–$1
Nearest Even (default)	$1	$2	$2	$2	–$2

- What are the advantages of the modes?

Closer Look at Round-To-Even

- Default Rounding Mode
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated

- Applying to Other Decimal Places / Bit Positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

PP2.50

Rounding Binary Numbers

- Binary Fractional Numbers
  - "Even" such that least significant bit becomes `0`
  - "Half way" when bits to right of rounding position = `100`…$_2$

- Examples
  - Round to nearest 1/4 (2 bits right of binary point)

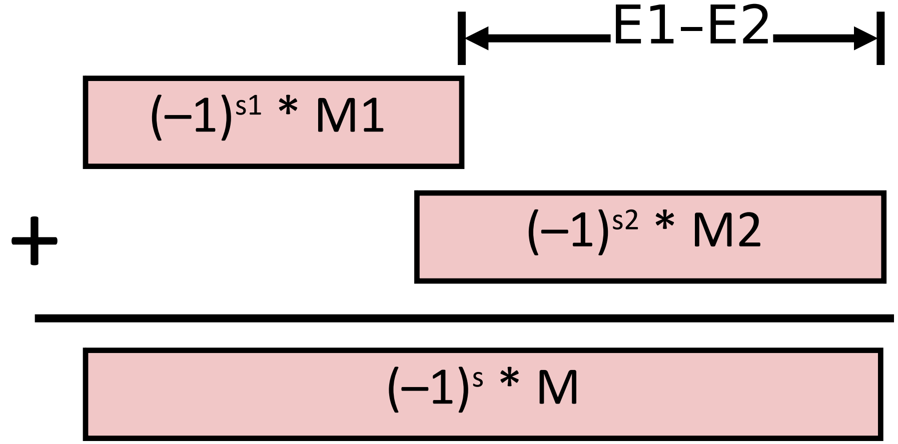
Value	Binary	Rounded	Action	Rounded Value
2 3/32	$10.00\mathbf{011}_2$	10.00_2	(<1/2—down)	2
2 3/16	$10.00\mathbf{110}_2$	10.01_2	(>1/2—up)	2 1/4
2 7/8	$10.11\mathbf{100}_2$	11.00_2	(1/2—up)	3
2 5/8	$10.10\mathbf{100}_2$	10.10_2	(1/2—down)	2 1/2

FP Multiplication

- $(-1)^{s1} * M1 * 2^{E1}$ x $(-1)^{s2} * M2 * 2^{E2}$
- Exact Result: $(-1)^{s} * M * 2^{E}$
  - Sign s: s1 ^ s2
  - Significand M: M1 * M2
  - Exponent E: E1 + E2
- Fixing
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit `frac` precision
- Implementation
  - Biggest chore is multiplying significands

Floating Point Addition

- $(-1)^{s1} * M1 * 2^{E1} \ + \ (-1)^{s2} * M2 * 2^{E2}$
  - Assume E1 > E2



- Exact Result: $(-1)^{s} * M * 2^{E}$
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: E1

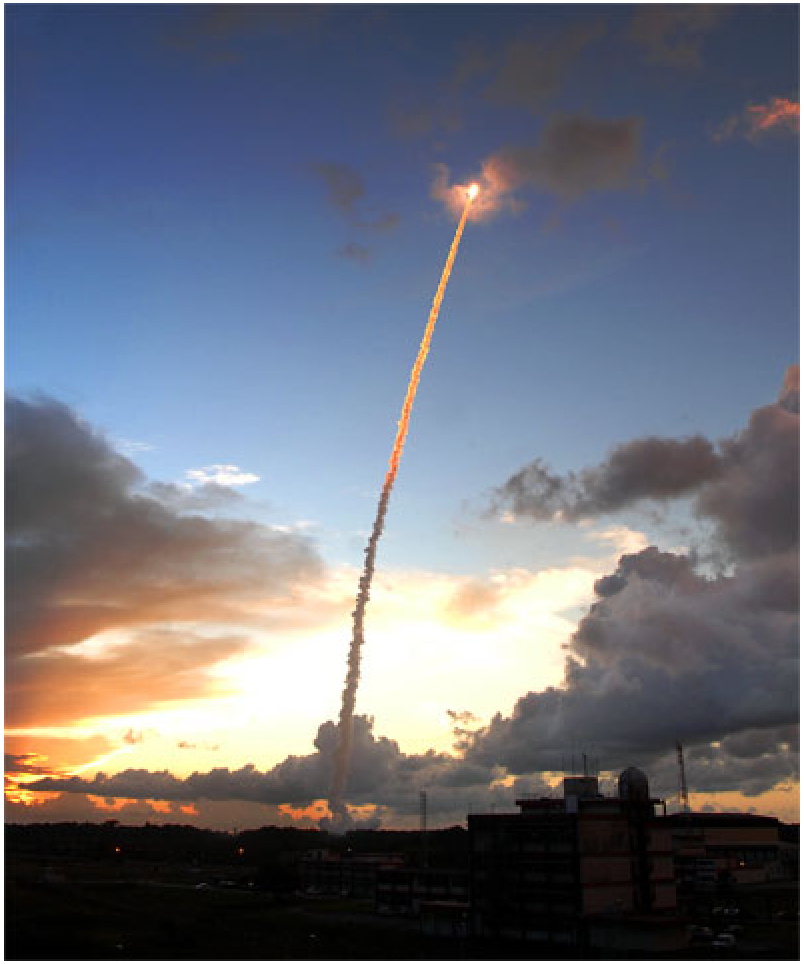
- Fixing
  - If M ≥ 2, shift M right, increment E
  - if M < 1, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit `frac` precision

(Floating Point in C)

- C Guarantees Two Levels
  - `float`   single precision
  - `double`   double precision
- Conversions/Casting
  - Casting between `int`, `float`, and `double` changes bit representation
  - `double`/`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

Ariane 5

- Convert 64-bit floating point to 16-bit signed integer
- Overflow occurred.
- $500 million firework



Patriot Missile

- First Gulf War
- Time kept as 1/10 second, as an integer
  To find time in seconds, multiply by 1/10
- 1/10 second = 0.0001100110011001100110011001100....
  (small rounding error)
- After 100 hours: total rounding error of 0.34secs => missed Scud missile by more than 0.5 km => 28 dead