# Floating Point

CENG331 - Computer Organization

**Instructor:**

Murat Manguoğlu (Section 1)

Adapted from slides of the textbook: http://csapp.cs.cmu.edu/

# Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

# Who Cares About FP numbers?

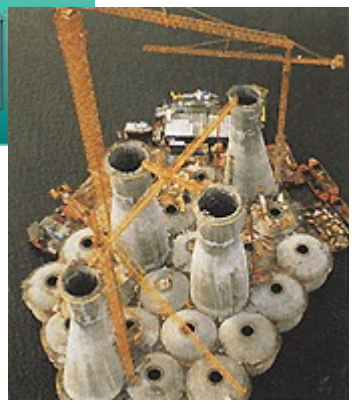
- Important for scientific code
  - But for everyday consumer use?
    - “My bank balance is out by 0.0002¢!” ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

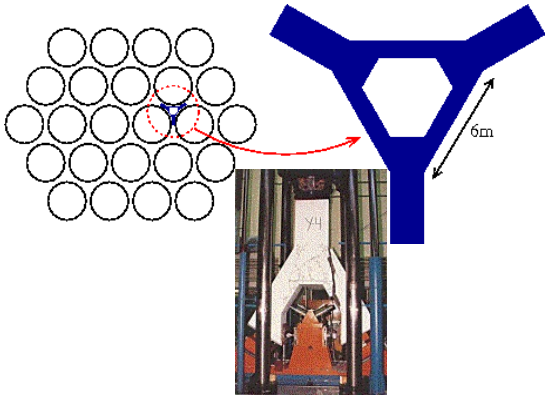
# Ariane 5 explosion

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after lift-off. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. A board of inquiry investigated the causes of the explosion and in two weeks issued a report. It turned out that the cause of the failure was a software error in the inertial reference system. Specifically **a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,768, the largest integer storeable in a 16 bit signed integer**, and thus the conversion failed.

source: http://www.ima.umn.edu/~arnold/disasters/

# The sinking of the Sleipner A offshore platform






The Sleipner A platform produces oil and gas in the North Sea and is supported on the seabed at a water depth of 82 m. It is a Condeep type platform with a concrete gravity base structure consisting of 24 cells and with a total base area of 16 000 m2. Four cells are elongated to shafts supporting the platform deck. The first concrete base structure for Sleipner A sprang a leak and sank under a controlled ballasting operation during preparation for deck mating in Gandsfjorden outside Stavanger, Norway on 23 August 1991.

The post accident investigation traced the error to **inaccurate finite element approximation of the linear elastic model** of the tricell (using the popular finite element program NASTRAN). The shear stresses were underestimated by 47%, leading to insufficient design. In particular, certain concrete walls were not thick enough. More careful finite element analysis, made after the accident, predicted that failure would occur with this design at a depth of 62m, which matches well with the actual occurrence at 65m.

source: http://www.ima.umn.edu/~arnold/disasters/

# Patriot missile failure

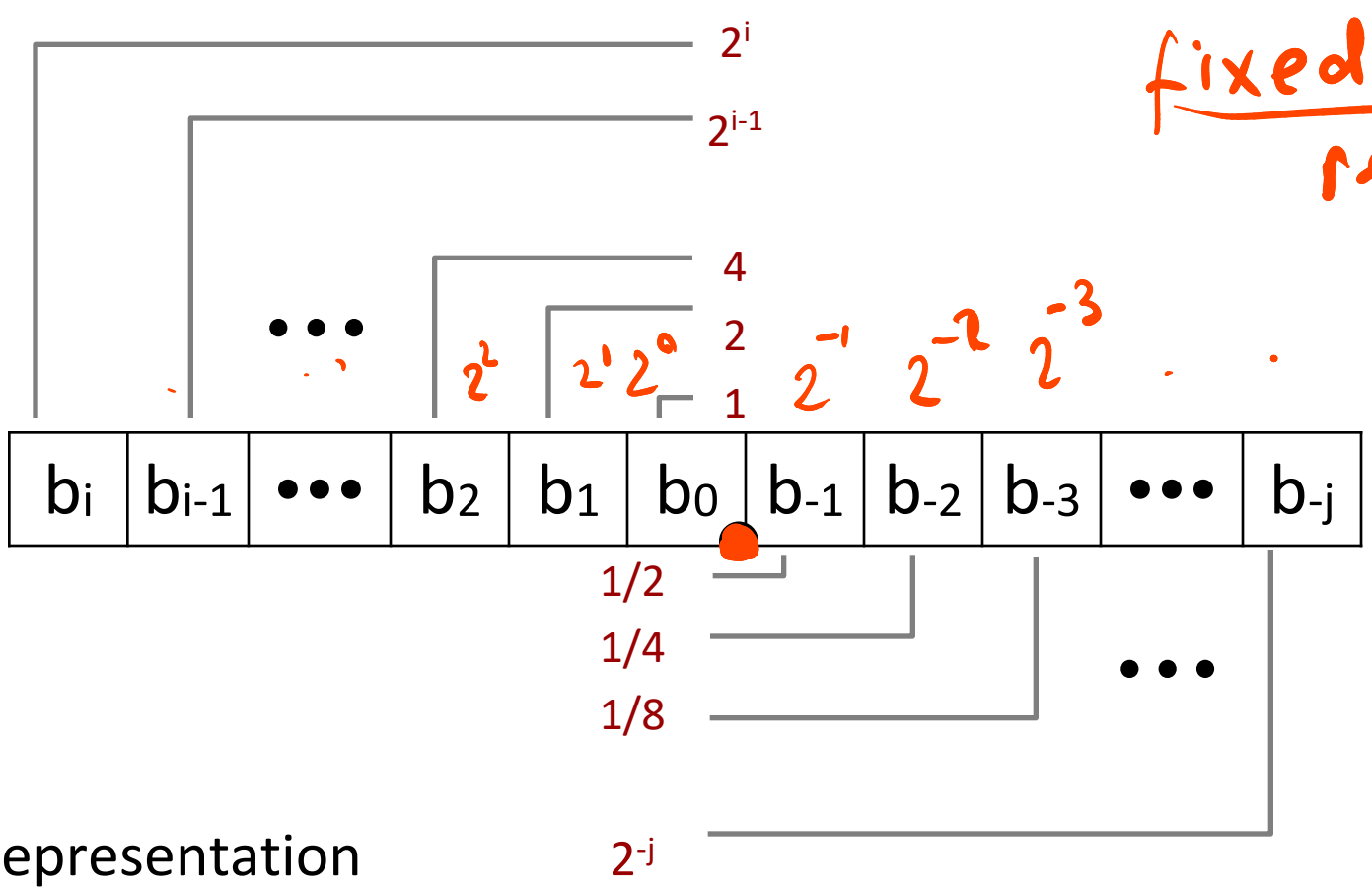
On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soldiers. A report of the General Accounting office, GAO/IMTEC-92-26, entitled Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia reported on the cause of the failure. It turns out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors. Specifically**, the time in tenths of second as measured by the system's internal clock was multiplied by 1/10 to produce the time in seconds. This calculation was performed using a 24 bit fixed point register. In particular, the value 1/10, which has a non-terminating binary expansion, was chopped at 24 bits after the radix point. The small chopping error, when multiplied by the large number giving the time in tenths of a second, lead to a significant error.** Indeed, the Patriot battery had been up around 100 hours, and an easy calculation shows that the resulting time error due to the magnified chopping error was about 0.34 seconds. (The number 1/10 equals 1/24+1/25+1/28+1/29+1/212+1/213+.... In other words, the binary expansion of 1/10 is 0.0001100110011001100110011001100.... Now the 24 bit register in the Patriot stored instead 0.00011001100110011001100 introducing an error of 0.0000000000000000000000011001100... binary, or about 0.000000095 decimal. Multiplying by the number of tenths of a second in 100 hours gives 0.000000095×100×60×60×10=0.34.) A Scud travels at about 1,676 meters per second, and so travels more than half a kilometer in this time. This was far enough that the incoming Scud was outside the "range gate" that the Patriot tracked. Ironically, the fact that the bad time calculation had been improved in some parts of the code, but not all, contributed to the problem, since it meant that the inaccuracies did not cancel.

source: http://www.ima.umn.edu/~arnold/disasters/

# Fractional binary numbers

- What is $1011.101_2$?

# Fractional Binary Numbers




fixed point repr.

- 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
  - 5 3/4 — $101.11_2$
  - 2 7/8 — $10.111_2$
  - 1 7/16 — $1.0111_2$

- Observations
  - Divide by 2 by shifting right (unsigned)
  - 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 $1.0 - \varepsilon$

# Representable Numbers

- Limitation #1
  - 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  $0.0001100110011[0011]\ldots_2$

- Limitation #2
  - Just one setting of binary point within the *w* bits
    - Limited range of numbers (very small values?  very large?)

# Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

# IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - 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
    - There are ways to make floating point operations fast in the hardware
    - Numerical computations are what computers (and the most powerful computers ) are really doing

# Floating Point Representation

- Numerical Form:

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

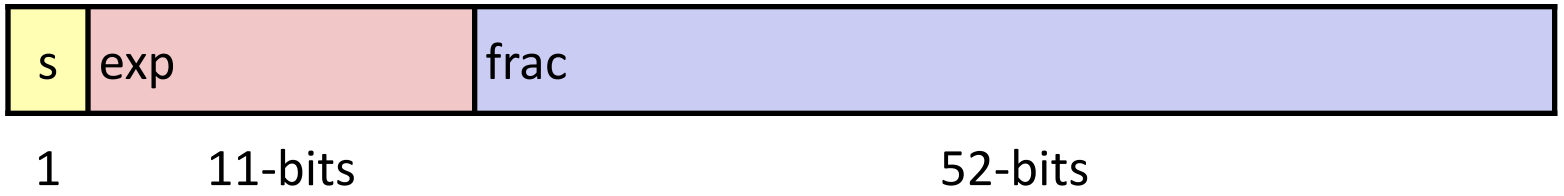
  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two

- 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)

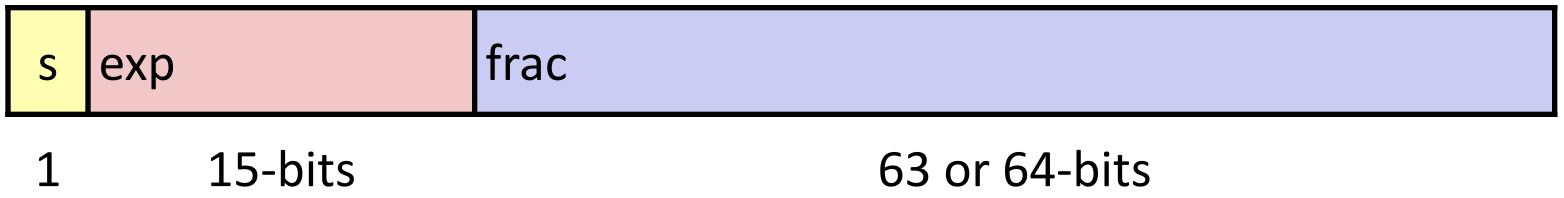
# Precision options

- Single precision: 32 bits


- Double precision: 64 bits




- Extended precision: 80 bits (Intel only)



# "Normalized" Values

$v = (-1)^s M 2^E$

- When: exp ≠ 000…0 and exp ≠ 111…1
- Exponent coded as a biased value: E = Exp – Bias
  - Exp: unsigned value of exp field
  - 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 = $1.xxx…x_2$
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

# Normalized Encoding Example

$v = (-1)^s M 2^E$
$E = Exp - Bias$

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

- **Significand**

  $M = 1.1101101101101_2$

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

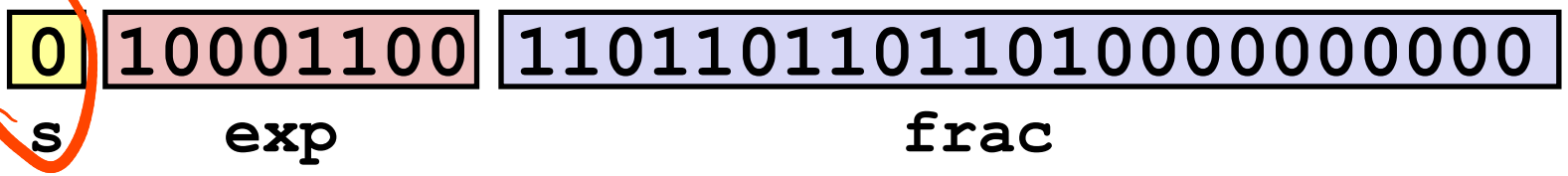
- **Exponent**

  $E = 13$

  $Bias = 127$

  $Exp = 140 = 10001100_2$

- **Result:**

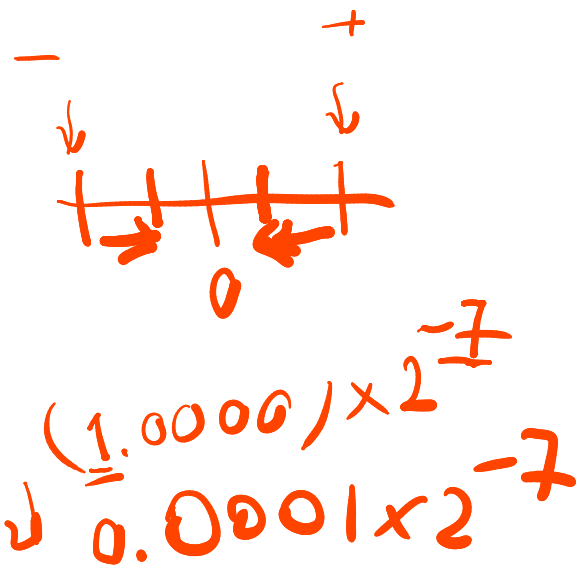


# Denormalized Values

$v = (-1)^s M 2^E$
$E = 1 - Bias$

- Condition: exp = 000…0

- Exponent value: E = 1 – Bias (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 closest to 0.0
    - Equispaced



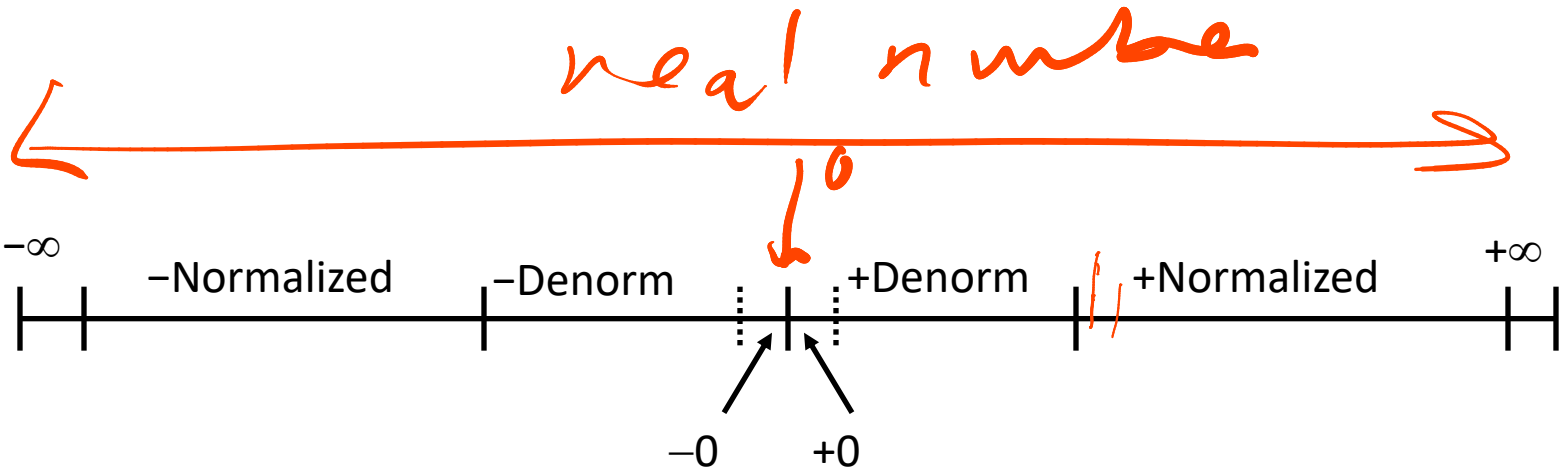
# Special Values

- 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 = +∞, 1.0/−0.0 = −∞

- Case: **`exp`** = **`111…1`**, **`frac`** ≠ **`000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

# Visualization: Floating Point Encodings

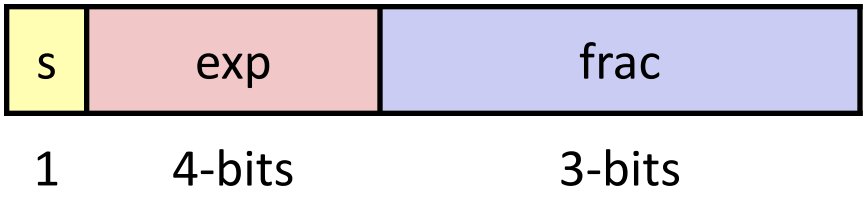



NaN

# Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example
- Rounding, addition, multiplication
- Floating point in C
- Summary

# 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

# Dynamic Range (Positive Only)

$v = (-1)^s M 2^E$

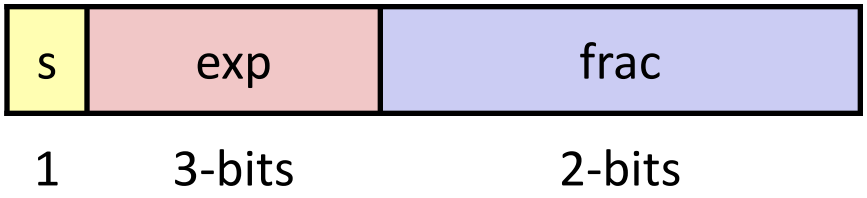
n: E = Exp – Bias

d: E = 1 – Bias

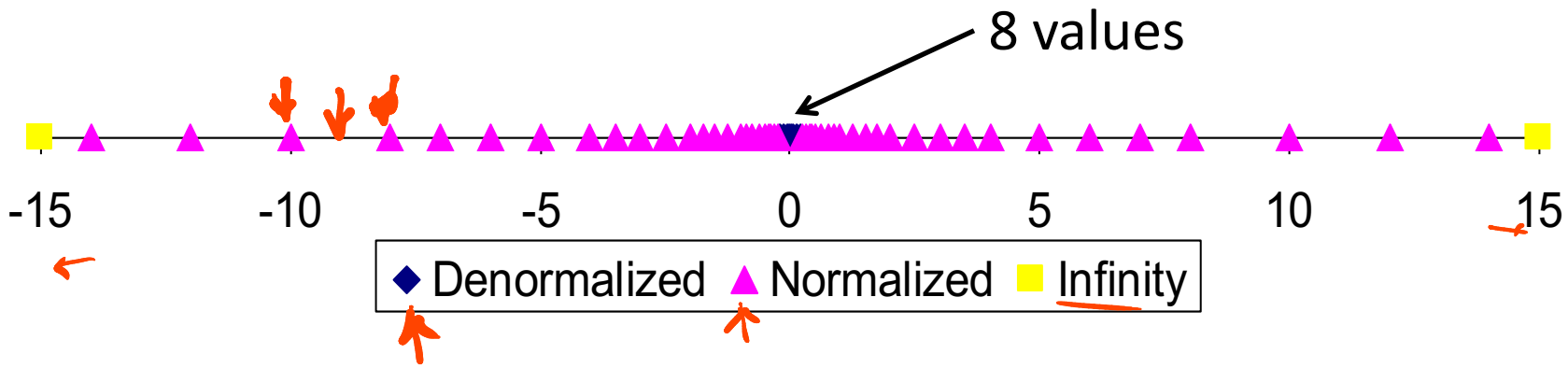
| | s | exp | frac | E | Value | |
|---|---|---|---|---|---|---|
| Denormalized numbers | 0 | 0000 | 000 | -6 | 0 | |
| | 0 | 0000 | 001 | -6 | 1/8*1/64 = 1/512 | closest to zero |
| | 0 | 0000 | 010 | -6 | 2/8*1/64 = 2/512 | |
| | … | | | | | |
| | 0 | 0000 | 110 | -6 | 6/8*1/64 = 6/512 | |
| | 0 | 0000 | 111 | -6 | 7/8*1/64 = 7/512 | largest denorm |
| | 0 | 0001 | 000 | -6 | 8/8*1/64 = 8/512 | smallest norm |
| | 0 | 0001 | 001 | -6 | 9/8*1/64 = 9/512 | |
| | … | | | | | |
| | 0 | 0110 | 110 | -1 | 14/8*1/2 = 14/16 | |
| | 0 | 0110 | 111 | -1 | 15/8*1/2 = 15/16 | closest to 1 below |
| Normalized numbers | 0 | 0111 | 000 | 0 | 8/8*1 = 1 | |
| | 0 | 0111 | 001 | 0 | 9/8*1 = 9/8 | closest to 1 above |
| | 0 | 0111 | 010 | 0 | 10/8*1 = 10/8 | |
| | … | | | | | |
| | 0 | 1110 | 110 | 7 | 14/8*128 = 224 | |
| | 0 | 1110 | 111 | 7 | 15/8*128 = 240 | largest norm |
| | 0 | 1111 | 000 | n/a | inf | |

# Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$

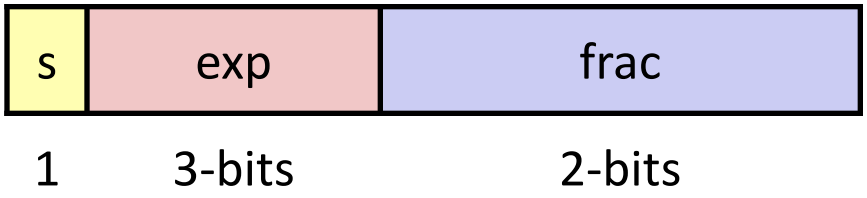


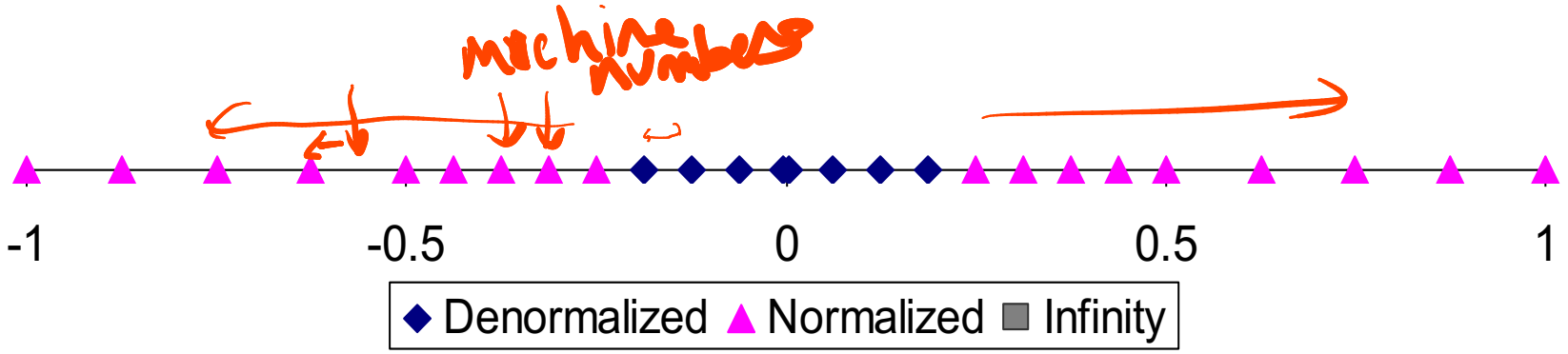

- Notice how the distribution gets denser toward zero.



# Distribution of Values (close-up view)

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



# Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

# 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`**

# Accuracy and Precision

- Accuracy*: absolute or relative error of an approximate quantity
- Precision*: the accuracy with which the basic arithmetic operations are performed
  - all fraction bits are significant
  - Single: approx $2^{-23}$
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx$ 6 decimal digits of precision
  - Double: approx $2^{-52}$
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx$ 16 decimal digits of precision

* Accuracy and Numerical Stability of Algorithms, Nicholas J. Higham

# Floating Point Arithmetic

**Patterson Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3rd Edition**

# Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - $1.0015 \times 10^{2}$
- 4. Round and renormalize if necessary
  - $1.002 \times 10^{2}$

# Floating-Point Addition

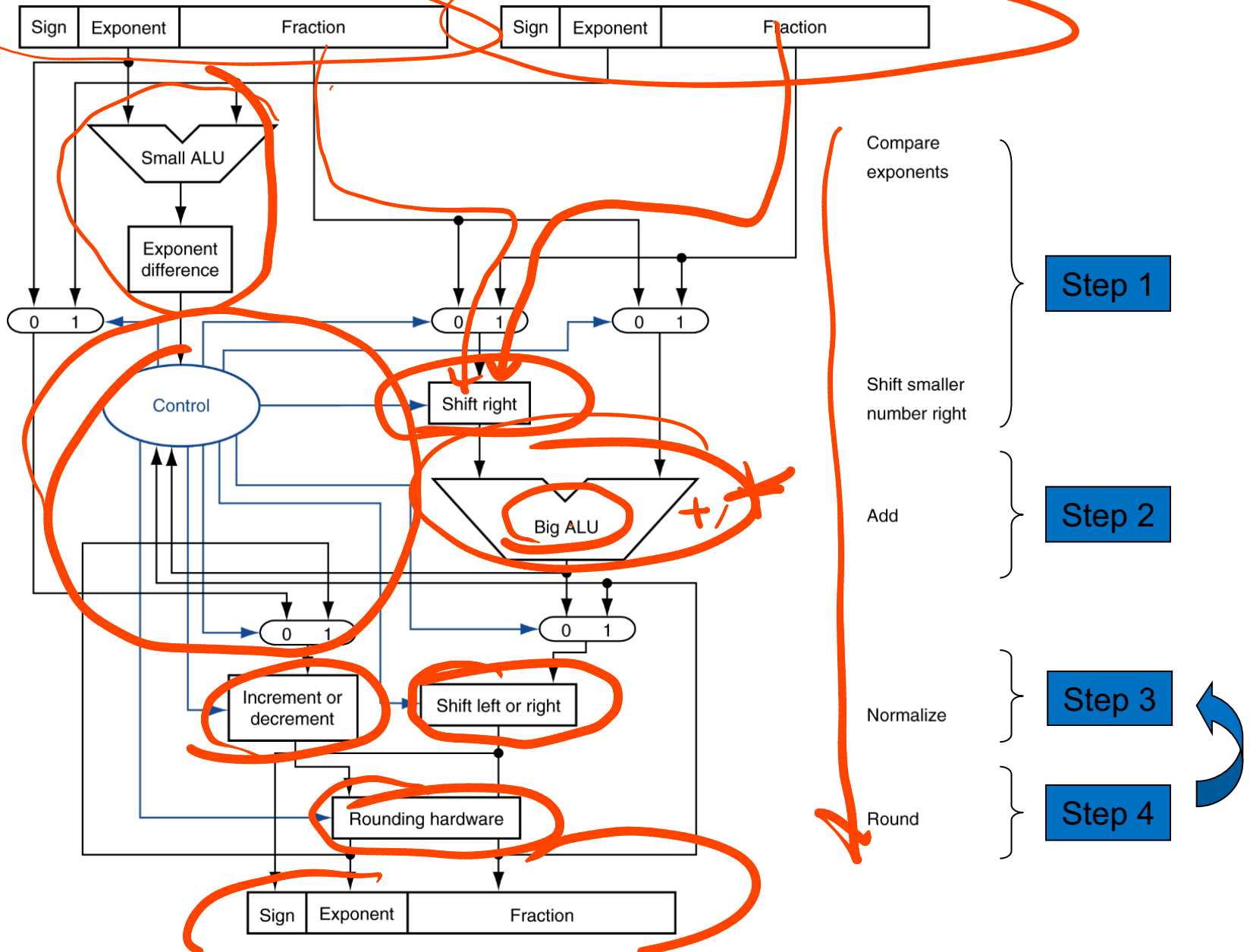
FPU

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

# FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

# FP Adder Hardware



# Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} * 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - $1.0212 \times 10^{6}$
- 4. Round and renormalize if necessary
  - $1.021 \times 10^{6}$
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

# Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ ($0.5 \times -0.4375$)
- 1. Add exponents
  - Unbiased: $-1 + -2 = -3$
  - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve ✕ –ve $\Rightarrow$ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

# FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be **pipelined**

$t_{FP}^{*} > t_{FP}^{+}$

# Real-World Pipelines: Car Washes

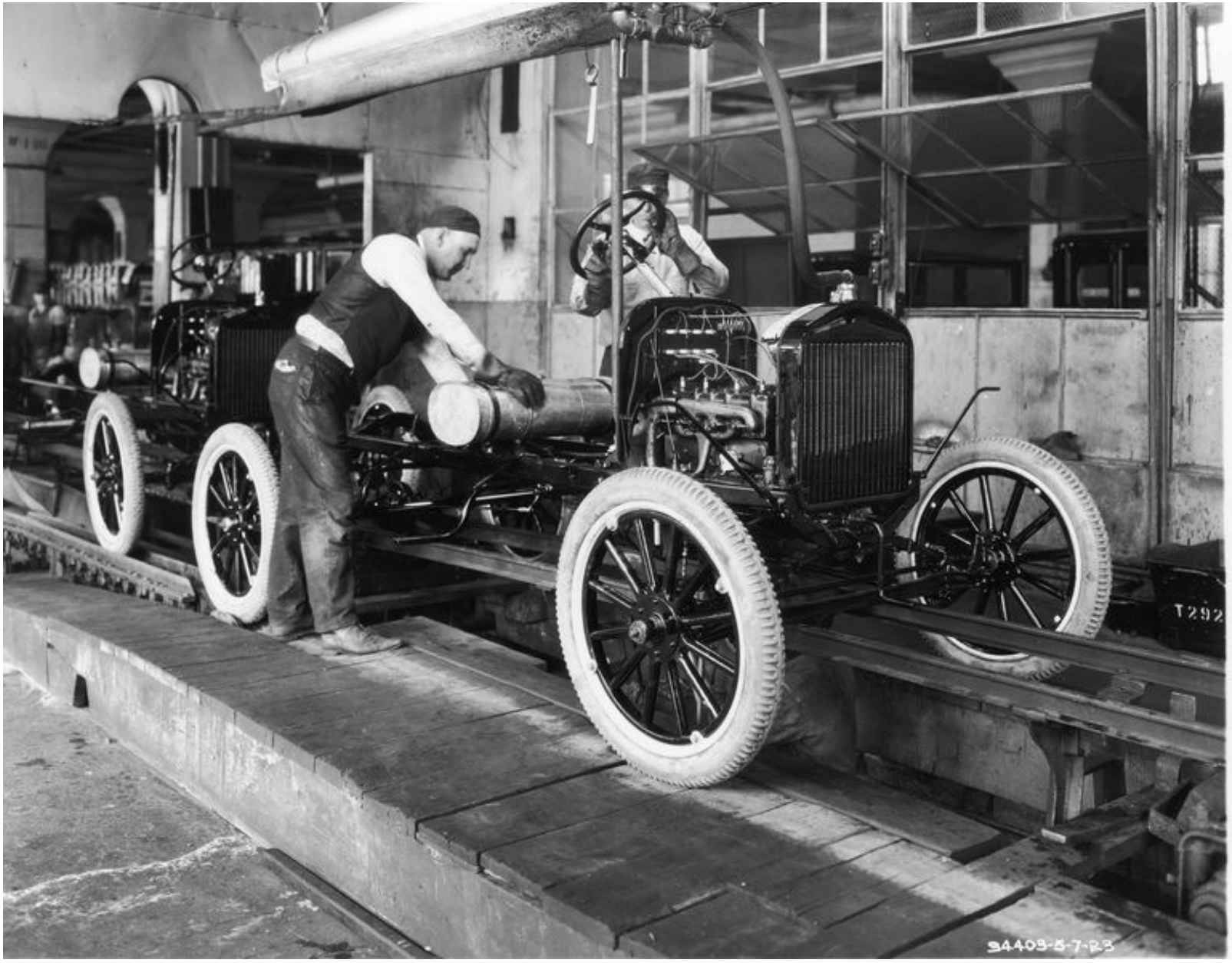
## Sequential


## Parallel

## Pipelined

- Idea
  - Divide process into independent stages
  - Move objects through stages in sequence
  - At any given times, multiple objects being processed

# Car production: Ford assembly line

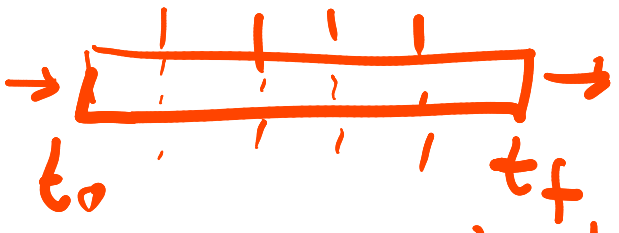



A Ford assembly line with a worker attaching a gas tank. (circa 1923). (Photo by Fotosearch/Getty Images)

# Pipelining history

- Industrial production: Ford motor company (early 1900s)
  - Workers worked only on a part of the car in the pipeline
  - Leading to more efficient utilization of the resources (workers)
    - Later it was discovered that working on the same part causes workers to make mistaked (solution: rotate the workers)
- Computing
  - IBM7030 (1959) had instruction pipeline
  - Later it appears almost everywhere
    - Instruction, FP arith., Data Comm. , etc.

# Pipelining (in general)



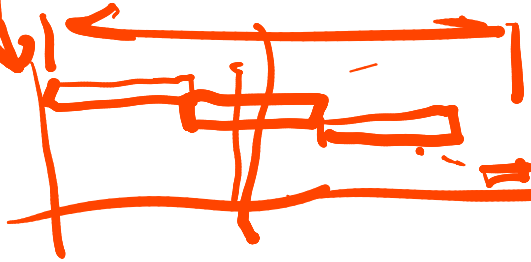

assume

k - subtasks of equal length

assume we have n of the tasks

$T = t_f - t_0$

Seq. op. cost: $T_{seq} = nT$

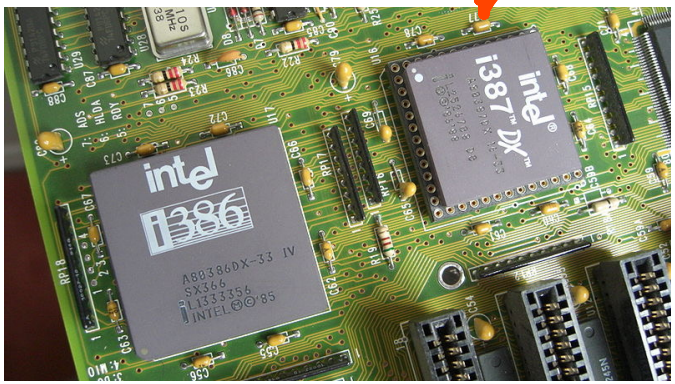


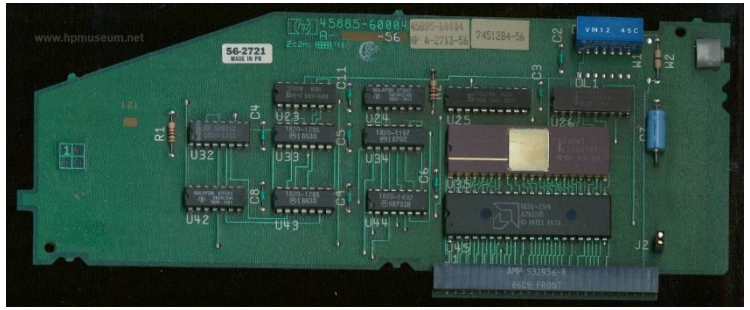
Pipelining cost $T_p$, each subtask $\frac{T}{k}$


$$T_p = \frac{(k-1)T}{k} + \frac{nT}{k}$$

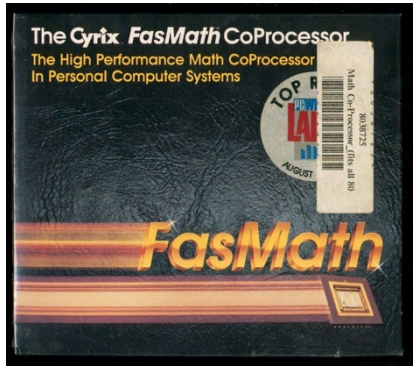
# x86 FP Architecture

- Originally based on 8087 FP coprocessor (x87)
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …







# x86 FP Architecture

- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Not easy to generate and optimize code
  - Result: poor FP performance
  - With XMM registers in X86-64 this is no longer the case

# Streaming SIMD Extension 2 (SSE2)

- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

| Machine | Underflow $\lambda$ | Overflow $\Lambda$ |
|---|---|---|
| DEC PDP-11, VAX, F and D formats | $2^{-128} \approx 2.9 \times 10^{-39}$ | $2^{127} \approx 1.7 \times 10^{38}$ |
| DEC PDP-10; Honeywell 600, 6000; Univac 110x single; IBM 709X, 704X | $2^{-129} \approx 1.5 \times 10^{-39}$ | $2^{127} \approx 1.7 \times 10^{38}$ |
| Burroughs 6X00 single | $8^{-51} \approx 8.8 \times 10^{-47}$ | $8^{76} \approx 4.3 \times 10^{68}$ |
| H-P 3000 | $2^{-256} \approx 8.6 \times 10^{-78}$ | $2^{256} \approx 1.2 \times 10^{77}$ |
| IBM 360, 370; Amdahl1; DG Eclipse M/600; ... | $16^{-65} \approx 5.4 \times 10^{-79}$ | $16^{63} \approx 7.2 \times 10^{75}$ |
| Most handheld calculators | $10^{-99}$ | $10^{100}$ |
| CDC 6X00, 7X00, Cyber | $2^{-976} \approx 1.5 \times 10^{-294}$ | $2^{1070} \approx 1.3 \times 10^{322}$ |
| DEC VAX G format; UNIVAC, 110X double | $2^{-1024} \approx 5.6 \times 10^{-309}$ | $2^{1023} \approx 9 \times 10^{307}$ |
| HP 85 | $10^{-499}$ | $10^{500}$ |
| Cray I | $\approx 2^{-8192} \approx 9.2 \times 10^{-2467}$ | $\approx 2^{8192} \approx 1.1 \times 10^{2466}$ |
| DEC VAX H format | $2^{-16384} \approx 8.4 \times 10^{-4933}$ | $2^{16383} \approx 5.9 \times 10^{4931}$ |
| Burroughs 6X00 double | $8^{-32755} \approx 1.9 \times 10^{-29581}$ | $8^{32780} \approx 1.9 \times 10^{29603}$ |
| Proposed IEEE Standard: INTEL i8087; Motorola 6839 | | |
| single | $2^{-126} \approx 1.2 \times 10^{-38}$ | $2^{127} \approx 1.7 \times 10^{38}$ |
| double | $2^{-1022} \approx 2.2 \times 10^{-308}$ | $2^{1023} \approx 9 \times 10^{307}$ |
| double-extended | $\leq 2^{-16382} \approx 3.4 \times 10^{-4932}$ | $\geq 2^{16383} \approx 5.9 \times 10^{4931}$ |

Table 1: Floating-Point Over/Underflow Thresholds

Source: https://people.eecs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf

# Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- X86 floating point
- Floating point in C
- Summary

# 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

# Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither
**d** nor **f** is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (double)(float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0  ⇒  ((d*2) < 0.0)`
- `d > f  ⇒  -f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

# Alternatives to IEEE floating point standart

- Tapered Floating Point
  - Morris, Robert. "Tapered floating point: A new floating-point representation." *IEEE Transactions on Computers* 100.12 (1971): 1578-1579
  - Gustafson, John L. *The End of Error: Unum Computing*. CRC Press, 2017.
  - Gustafson, John L., and Isaac T. Yonemoto. "Beating floating point at its own game: Posit arithmetic." *Supercomputing Frontiers and Innovations* 4.2 (2017): 71-86.

- Symmetric Level  Index
  - Clenshaw, Charles W., and Frank WJ Olver. "Beyond floating point." *Journal of the ACM (JACM)* 31.2 (1984): 319-328.

# Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form M x $2^E$
- One 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 & serious numerical applications programmers

*Thank you!*