7. Floating Point Arithmetic

EECS 370 – Introduction to Computer Organization – Winter 2015

Robert Dick, Andrew Lukefahr, and Satish Narayanasamy

EECS Department
University of Michigan in Ann Arbor, USA

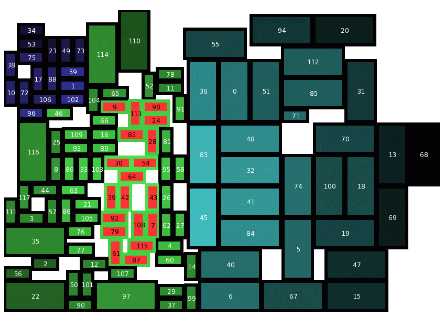
Robert Dick

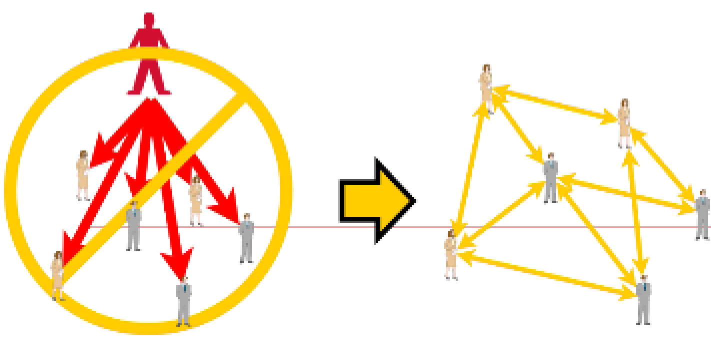
- Born in Upstate New York
  - Father blew up ships and gives investment advice
  - Mother was music teacher
- Bachelor’s degree from Clarkson University
- Ph.D. from Princeton University
- Taught at Tsinghua University, Northwestern University
- Came to Michigan Jan. 2009
- Recently co-founded a wearable electronics company

  http://stryd.com/

Robert Dick

3/11





- Embedded system design
  - Censorship and surveillance resistant communication
  - Distributed sensing systems
  - Data compression
  - Integrated circuit design automation
  - Wearable electronics

Robert Dick Contact Info

- To reach me
  - Robert Dick
  - 2417-E EECS
  - Cellphone: 847-530-1824
  - dickrp@umich.edu

Why floating point

- Have to represent them somehow
- Rational numbers
  - Ok, but can be cumbersome to work with
  - Falls apart for sqrt(2) and other irrational numbers
- Fixed point
  - Do everything in thousandths (or millions, etc.)
  - Not always easy to pick the right units
  - Different scaling factors for different stages of computation
- Scientific notation: this is good!
  - Exponential notation allows HUGE dynamic range
  - Constant (approximately) relative precision across the whole range

Lots of ways to do floating point

- Decimal: 2.99792458×10^8
- Hexadecimal: $1.1de784a \times 16^7$
- Binary: $1.0001110111100111100001001010 \times 2^{28}$
- Wilder alternatives
  - Arbitrary precision arithmetic
    - Software support for arbitrary number of digits (or bits)
    - Powerful, but almost always slow
  - Represent numbers by their logarithms
    - Used for centuries in slide rules
    - Makes multiplication and division really fast and easy
    - But addition and subtraction become quite painful

Floating point before IEEE-754 standard

- Late 1970s formats
  - About two dozen different, incompatible floating point number formats
  - Decimal, binary, octal, hexadecimal all in use
  - Precisions from about 4 to about 17 decimal digits
  - Ranges from about 10^{19} to 10^{322}
- Sloppy arithmetic
  - Last few bits were often wrong, and in different ways
  - Overflow sometimes detected, sometimes ignored
  - Arbitrary, almost random rounding modes
    - Truncate, round up, round to nearest
  - Addition and multiplication not necessarily commutative
    - Small differences due to roundoff errors

IEEE floating point

- Standard set by IEEE
  - John Palmer at Intel took the lead in 1976 for a good standard
  - First working implementation: Intel 8087 floating point coprocessor, 1980
  - Full formal adoption: 1985
  - Updated in 2008
- Rigorous specification for high accuracy computation
  - Made every bit count
  - Dependable accuracy even in the lowest bits
  - Predictable, reasonable behavior for exceptional conditions
    - (divide by zero, overflow, etc.)

IEEE Floating point format (single precision)

- Sign bit: (0 is positive, 1 is negative)
- Significand: (also called the *mantissa*; stores the 23 most significant bits after the decimal point)
- Exponent: used biased base 127 encoding
  - Add 127 to the value of the exponent to encode:
  - -127 → 00000000   1 → 10000000
  - -126 → 00000001   2 → 10000001
  - …   …
  - 0 → 01111111   128 → 11111111
- How do you represent zero ? Special convention:
  - Exponent: -127 (all zeroes), Significand 0 (all zeroes), Sign + or -

Floating Point Representation

$$10.625_{10} \Rightarrow 1010.101_{2}$$

$$1.010101 \times 2^{3}$$

This must be a 1!
So don't store it.

+/-	Mantissa (1010101)	Exponent (3)
1 bit	23 bits	8 bits

Floating Point Representation

$10.625_{10} \Rightarrow 1010.101_2$

1.010101×2^3

This must be a 1!
So don't store it.

+/-	Mantissa (1010101)	Exponent (3)
1 bit	23 bits	8 bits

$10.625_{10} = 0 \quad 10000010 \quad 01010100000000000000000$

Class Problem

- What is the value (in decimal) of the following IEEE 754 floating point encoded number?

1	**10000101**	**01011001000000000000000**

Floating point multiplication

- Add exponents (don't forget to account for the bias)
- Multiply significands (don't forget the implicit **1** bits)
- Renormalize if necessary
- Compute sign bit (simple exclusive-or)

Floating point multiply

$10.625_{10} = 1010.101_2$ ⇒ 0 | 10000010 | 01010100000000000000000

+ ×

$10_{10} = 1010_2$ ⇒ 0 | 10000010 | 01000000000000000000000

-127

0	10000010	01010100000000000000000
	+	×
0	10000010	01000000000000000000000
	-127	
0	10000101	10101001000000000000000

$$
\begin{array}{r}
1\,0\,1\,0\,1\,0\,1 \\
\times \qquad 1\,0\,1 \\
\hline
1\,0\,1\,0\,1\,0\,1 \\
1\,0\,1\,0\,1\,0\,1\,0\,0 \\
\hline
1\,1\,0\,1\,0\,1\,0\,0\,1
\end{array}
$$

1101010.01_2

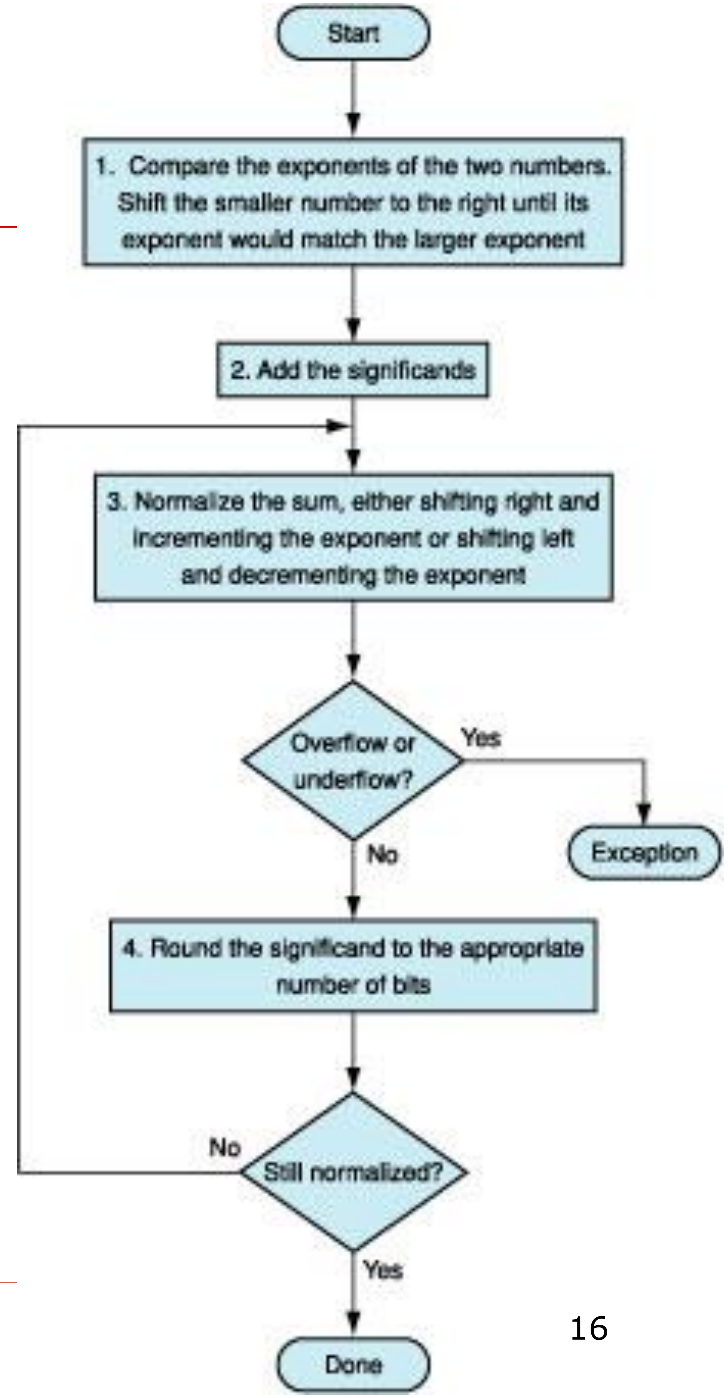
$= 106.25_{10}$

Floating point addition

- More complicated than floating point multiplication!
- If exponents are unequal, must shift the significand of the smaller number to the right to align the corresponding place values
- Once numbers are aligned, simple addition (could be subtraction, if one of the numbers is negative)
- Renormalize (which could be messy if the numbers had opposite signs; for example, consider addition of +1.5000 and – 1.4999)
- Added complication: rounding to the correct number of bits to store could denormalize the number, and require one more step

Floating point Addition

1. Shift smaller exponent right to match larger.
2. Add significands
3. Normalize and update exponent
4. Check for "out of range"



Class Problem

Show how to add the following 2 numbers using IEEE floating point addition: 100.125 + 13.75

More precision and range

- We have described IEEE-754 binary32 floating point format, commonly known as "single precision" ("float" in C/C++)
  - 24 bits precision; equivalent to about 7 decimal digits
  - $3.4 * 10^{38}$ maximum value
  - Good enough for most but not all calculations
- IEEE-754 also defines a larger binary64 format, "double precision" ("double" in C/C++)
  - 53 bits precision, equivalent to about 16 decimal digits
  - $1.8 * 10^{308}$ maximum value
  - Most accurate physical values currently known only to about 47 bits precision, about 14 decimal digits

Extreme floating point

- binary128, “quad precision”; recent addition to standard:
  - 113 bits precision, about 34 decimal digits
  - $1.2*10^{4932}$ maximum value
  - Very rarely used, but some computations require extreme accuracy to limit cumulative roundoff error
- Another recent addition was binary16, “half precision”
  - 11 bits precision, about 3.3 decimal digits
  - 65504 maximum value
  - Used in graphics processors for accurate rendering of scenes with a large dynamic range in lighting levels.
  - Minimizes storage per pixel

More or less precision and range

Table 3.5—Binary interchange format parameters

Parameter	binary16	binary32	binary64	binary128	binary{k} ($k \geq 128$)
k, storage width in bits	16	32	64	128	multiple of 32
p, precision in bits	11	24	53	113	$k - \text{round}(4 \times \log2(k)) + 13$
$emax$, maximum exponent e	15	127	1023	16383	$2^{(k-p-1)} - 1$
Encoding parameters					
$bias$, $E - e$	15	127	1023	16383	$emax$
sign bit	1	1	1	1	1
w, exponent field width in bits	5	8	11	15	$\text{round}(4 \times \log2(k)) - 13$
t, trailing significand field width in bits	10	23	52	112	$k - w - 1$
k, storage width in bits	16	32	64	128	$1 + w + t$

The function round( ) in Table 3.5 rounds to the nearest integer.

For example, binary256 would have $p = 237$ and $emax = 262143$.