CS 33

Data Representation (Part 2)

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective." 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.

Today: Floating Point

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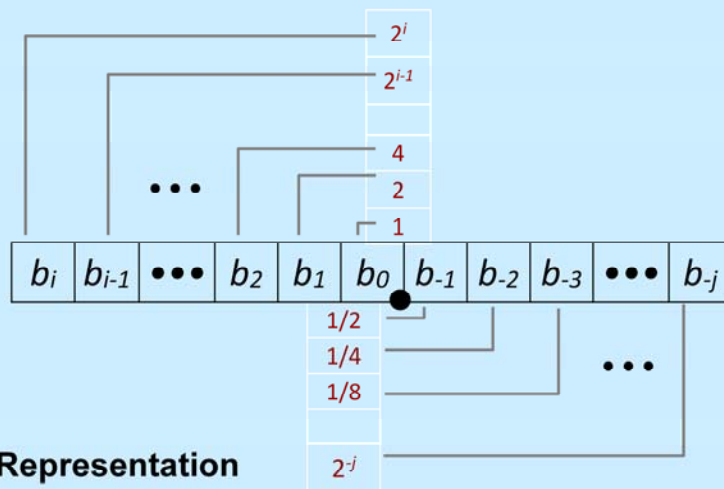- Rounding, addition, multiplication
- Floating point in C
- Summary

Supplied by CMU.

Fractional binary numbers

- What is 1011.101_2?

Supplied by CMU.

Supplied by CMU.

Fractional Binary Numbers: 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 (unsigned)
  - multiply by 2 by shifting left
  - numbers of form $0.111111..._2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + ... + 1/2^i + ...$ ➡ 1.0
    - use notation $1.0 - \varepsilon$

Supplied by CMU.

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 decimal point within the *w* bits
    - » limited range of numbers (very small values? very large?)

Supplied by CMU.

Supplied by CMU.

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

Supplied by CMU.

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).
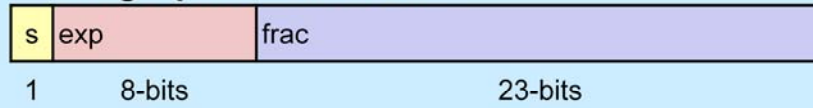 - exponent E weights value by power of two
- **Encoding**
  - MSB s is sign bit s
  - exp field encodes E (but is not equal to E)
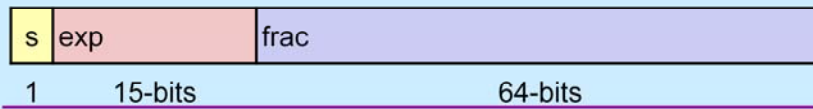 - frac field encodes M (but is not equal to M)

s	exp	frac

Supplied by CMU.

Precision options

- **Single precision: 32 bits**

s	exp	frac
1	8-bits	23-bits

- **Double precision: 64 bits**

s	exp	frac
1	11-bits	52-bits

- **Extended precision: 80 bits (Intel only)**

s	exp	frac
1	15-bits	64-bits

Supplied by CMU.

"Normalized" Values

- When: 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: $M = 1.xxx...x_2$
  - xxx...x: bits of frac
  - minimum when frac=000...0 ($M = 1.0$)
  - maximum when frac=111...1 ($M = 2.0 - \varepsilon$)
  - get extra leading bit for "free"

Supplied by CMU.

Normalized Encoding Example

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

- **Significand**

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

  $frac = \qquad\quad \underline{1101101101101}0000000000_2$

- **Exponent**

  $E \quad = \qquad 13$

  $bias \;= \qquad 127$

  $exp \;\;= \qquad 140 \quad = \qquad 10001100_2$

- **Result:**

0	10001100	11011011011010000000000
s	exp	frac
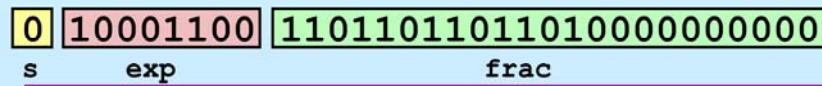
Supplied by CMU.

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 closest to 0.0
    - » equispaced

Supplied by CMU.

Special Values

- Condition: $\texttt{exp} = 111...1$

- Case: $\texttt{exp} = 111...1$, $\texttt{frac} = 000...0$
  - 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: $\texttt{exp} = 111...1$, $\texttt{frac} \neq 000...0$
  - not-a-number (NaN)
  - represents case when no numeric value can be determined
  - e.g., sqrt(−1), $\infty - \infty$, $\infty \times 0$

Supplied by CMU.
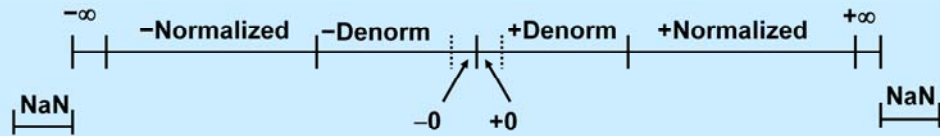
Supplied by CMU.

Supplied by CMU.

Tiny Floating Point Example

s	exp	frac
1	4-bits	3-bits

- **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

Supplied by CMU.
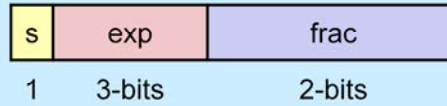
Dynamic Range (Positive Only)

	s	exp	frac	E	Value	
	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
Denormalized	0	0000	010	-6	2/8*1/64 = 2/512	
numbers	...					
	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	0	0111	000	0	8/8*1 = 1	
numbers	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	

Supplied by CMU.
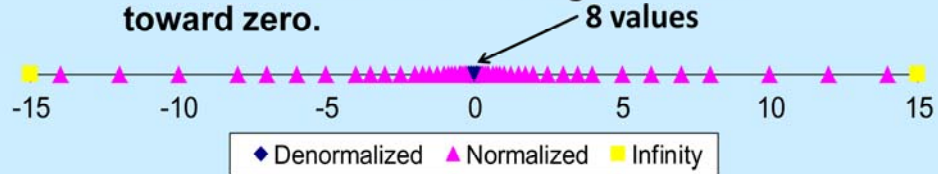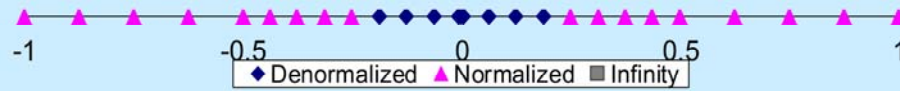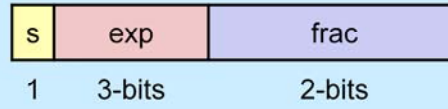
Supplied by CMU.

Supplied by CMU.

Supplied by CMU.

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`

Supplied by CMU.

Rounding

- **Rounding modes (illustrated 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

Supplied by CMU.

Closer Look at Round-To-Nearest-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)

IV–24

Supplied by CMU.

Rounding Binary Numbers

- **Binary fractional numbers**
  - "even" when least significant bit is 0
  - "half way" when bits to right of rounding position = $100\ldots_2$

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

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

Supplied by CMU.

FP Multiplication

- $(-1)^{s1} M1 \ 2^{E1} \quad \times \quad (-1)^{s2} M2 \ 2^{E2}$
- Exact result: $(-1)^s M \ 2^E$
  - sign s: $s1$ ^ $s2$
  - significand M: $M1 \times M2$
  - exponent E: $E1 + E2$

- Fixing
  - if $M \geq 2$, shift M right, increment E
  - if E out of range, overflow
  - round M to fit `frac` precision
- Implementation
  - biggest chore is multiplying significands

Supplied by CMU.

Floating Point Addition
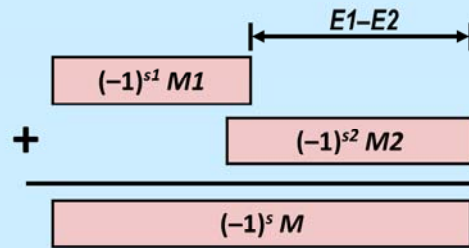
- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
  - assume $E1 > E2$

- **Exact result:** $(-1)^s\ M\ \ 2^E$
  - sign s, significand M:
    - » result of signed align & add
  - exponent E: $E1$

- **Fixing**
  - if $M \geq 2$, 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

Supplied by CMU.

Supplied by CMU.

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
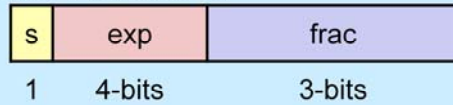
Supplied by CMU.

Supplied by CMU.

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

Supplied by CMU.

Creating Floating Point Number

s	exp	frac
1	4-bits	3-bits

- **Steps**
  - normalize to have leading 1
  - round to fit within fraction
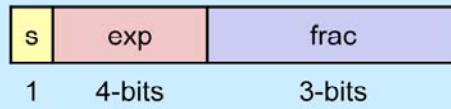 - postnormalize to deal with effects of rounding

- **Case study**
  - convert 8-bit unsigned numbers to tiny floating point format
  example numbers

128	10000000
15	00001101
33	00010001
35	00010011
138	10001010
63	00111111

Supplied by CMU.

Normalize

s	exp	frac
1	4-bits	3-bits

- **Requirement**
  - set binary point so that numbers of form 1.xxxxx
  - adjust all to have leading one
    - » decrement exponent as shift left

Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
15	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Supplied by CMU.

Supplied by CMU.

Postnormalize

- **Issue**
  - rounding may have caused overflow
  - handle by shifting right once & incrementing exponent

Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	$1.000*2^6$	64

Supplied by CMU.

Float is not Rational …

- **Floating addition**
  - commutative: $a +^f b = b +^f a$
    - » yes!
  - associative: $a +^f (b +^f c) = (a +^f b) +^f c$
    - » no!
      - $2 +^f (1e10 +^f -1e10) = 2$
      - $(2 +^f 1e10) +^f -1e10 = 0$

Float is not Rational …

- **Multiplication**
  - commutative: $a *^f b = b *^f a$
    - » yes!
  - associative: $a *^f (b *^f c) = (a *^f b) *^f c$
    - » no!
      - $1e20 *^f (1e20 *^f 1e-20) = 1e20$
      - $(1e20 *^f 1e20) *^f 1e-20 = +\infty$

Float is not Rational …

- **More …**
  - multiplication distributes over addition:
    $a *^f (b +^f c) = (a *^f b) +^f (a *^f c)$
    - » no!
    - » $1e20 *^f (1e20 +^f -1e20) = 0$
    - » $(1e20 *^f 1e20) +^f (1e20 *^f -1e20) = NaN$
  - loss of significance:
    x=y+1
    z=2/(x-y)
    z==2?
    - » not necessarily!
      - consider y = 1e20