Floating Point

CSC 235 - Computer Organization

References

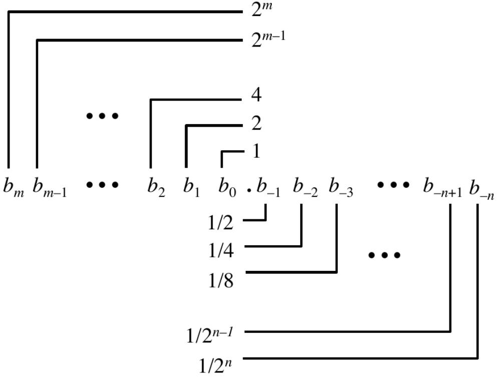
- Slides adapted from CMU

Outline

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

Fractional Binary Numbers

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



Fractional Binary Number Examples

Value	Representation
$23/4$	$101.11 = 4 + 1 + 1/2 + 1/4$
$23/8$	$10.111 = 2 + 1/2 + 1/4 + 1/8$
$23/16$	$1.0111 = 1 + 1/4 + 1/8 + 1/16$

- Observations
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.1111\ldots_2$ are just below 1.0

Representable Numbers

- Limitation 1
  - Can only exactly represent numbers of the form $\frac{x}{2^k}$
    - Other rational numbers have repeating bit representations
  - Example
    - $1/3 = 0.01010101[01]\ldots_2$
- Limitation 2
  - Just one setting of binary point within the w bits
    - limited range of numbers

IEEE Floating Point

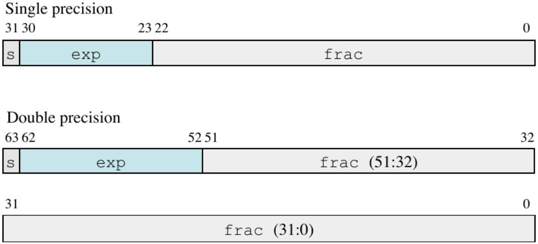
- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
  - Supported by all major CPUs
- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Difficult to make fast in hardware

Floating Point Representation

- Numerical Form: $(-1)^s \cdot M \cdot 2^E$
  - **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:
  - most significant bit 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
  - **exp** field is 8 bits
  - **frac** field is 23 bits
- Double precision: 64 bits
  - **exp** field is 11 bits
  - **frac** field is 52 bits



Floating Point Numbers

- Three different “kinds” of floating point numbers based on the **exp** field:
  - normalized: exp bits are not all ones and not all zeros
  - denormalized: exp bits are all zero
  - special: exp bits are all one

Normalized Values

- 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
- Significand coded with implied leading 1: $M = 1.xx \ldots x_2$
  - $xxx \ldots x$: bits of frac field
  - minimum when $frac = 000 \ldots 0$ ($M = 1.0$)
  - maximum when $frac = 111 \ldots 1$ ($M = 2.0 - \epsilon$)
  - get extra leading bit for "free"

Normalized Encoding Example

- Value: `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$
- Significand
  - $M = 1.1101101101101$
  - $frac = 11011011011010000000000$
- Exponent
  - $E = 13$
  - $bias = 127$
  - $exp = 140 = 10001100_2$

Denormalized Values

- Exponent value: $E = 1 - bias$ (instead of $exp - bias$)
- Significand coded with implied leading 0: $M = 0.xxx \ldots x_2$
  - $xxx \ldots x$: bits of frac
- Cases
  - $exp = 000 \ldots 0, frac = 000 \ldots 0$
    - represents zero value
    - Note distinct values: $+0$ and -0
  - $exp = 000 \ldots 0, frac \neq 000 \ldots 0$
    - numbers closest to 0.0
    - equally spaced

Special Values

- Case: $exp = 111\ldots1, frac = 000\ldots0$
  - represents value ∞ (infinity)
  - operation that overflows
  - both positive and negative
  - examples: 1.0/0.0 = -1.0/-0.0 = $+\infty$, 1.0/-0.0 = $-\infty$
- Case: $exp = 111\ldots1, frac \neq 000\ldots0$
  - Not-a-Number (NaN)
  - represents case when no numeric value can be determined
  - examples: sqrt(-1), $\infty = \infty$, $\infty \times 0$

C `float` Decoding Example 1

- `float` value = `0xC0A00000`
- binary: 1100 0000 1010 0000 0000 0000 0000 0000
- $E = exp - bias = 129 - 127 = 2_{10}$
- $s = 1$ negative number
- $M = 1.01000000000000000000000 = 1 + 1/4 = 1.25_{10}$
- $v = (-1)^s \cdot M \cdot 2^E = (-1)^1 \cdot 1.25 \cdot 2^2 = -5_{10}$

C `float` Decoding Example 2

- `float` value = `0x001C0000`
- binary: 0000 0000 0001 1100 0000 0000 0000 0000
- $E = exp - bias = 1 - 127 = -126_{10}$
- $s = 0$ positive number
- $M = 0.00111000000000000000000 = 1/8 + 1/16 + 1/32 = 7 \cdot 2^{-5}$
- $v = (-1)^s \cdot M \cdot 2^E = (-1)^0 \cdot 7 \cdot 2^{-5} \cdot 2^{-126} = 7 \cdot 2^{-131}$

Tiny Floating Point Example

- 8-bit floating point representation
  - the sign bit is 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

Dynamic Range ($s = 0$)

	s	exp	frac	E	value
	0	0000	000	-6	0
closest to zero	0	0000	001	-6	1/512
largest denorm	0	0000	111	-6	7/512
smallest norm	0	0001	000	-6	8/512
closest to 1 below	0	0110	111	-1	15/16
	0	0111	000	0	1
closest to 1 above	0	0111	001	0	9/8
largest norm	0	1110	111	7	240
	0	1111	000	-	`inf`

Special Properties of the IEEE Encoding

- Floating point zero same as integer zero
- Can (almost) use unsigned integer comparison
  - must first compare sign bits
  - must consider -0 $=$ 0
  - NaNs are problematic
  - Otherwise OK
    - Denormalized vs. normalized
    - Normalized vs. infinity

Floating Point Operations: Basic Idea

- $x +_f y = round(x + y)$
- $x \times_f y = round(x \times y)$
- Basic idea
  - first compute exact result
  - make it fit into the desired precision
    - possibly overflow if exponent is too large
    - possibly round to fit into *frac*

Rounding

- Rounding modes (illustrate with rounding USD)

	\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
towards zero	\$1	\$1	\$1	\$2	-\$1
round down ($-\infty$)	\$1	\$1	\$1	\$2	-\$2
round up (∞)	\$2	\$2	\$2	\$3	-\$1
nearest even	\$1	\$2	\$2	\$2	-\$2

- Nearest even rounds to the nearest, but if half-way in-between then round to nearest even

Closer Look at Round-To-Even

- Default Rounding Mode
  - Difficult 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
    - Example round to the nearest hundredth: $7.8950000 = 7.90$ (halfway – round up)
    - Example round to the nearest hundredth: $7.8850000 = 7.88$ (halfway – round down)

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 the nearest $1/4$ (2 bits right of binary point)

value	binary	rounded	action	rounded value
$2\frac{3}{32}$	10.00011	10.00	down	2
$2\frac{3}{16}$	10.00110	10.01	up	$2\frac{1}{4}$
$2\frac{7}{8}$	10.11100	11.00	up	3
$2\frac{5}{8}$	10.10100	10.10	down	$2\frac{1}{2}$

Rounding

- Terminology
  - guard bit: least significant bit of result
  - round bit: the first bit removed
  - sticky bit: OR of remaining bits
- Round up conditions
  - round $= 1$, sticky $= 1 \rightarrow > 0.5$
  - guard $= 1$, round $= 1$, sticky $= 0 \rightarrow$ round to even

Rounding Example

- Round to three bits after the binary point

fraction	GRS	Incr?	Rounded
1.0000000	000	N	1.000
1.1010000	100	N	1.101
1.0001000	010	N	1.000
1.0011000	110	Y	1.010
1.0001010	011	Y	1.001
1.1111100	111	Y	10.000

Floating Point Multiplication

- $(-1)^{s1} \cdot M1 \cdot 2^{E1} \times (-1)^{s2} \cdot M2 \cdot 2^{E2}$
- Exact result: $(-1)^{s} \cdot M \cdot 2^{E}$
  - 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

Floating Point Addition

- $(-1)^{s1} \cdot M1 \cdot 2^{E1} + (-1)^{s2} \cdot M2 \cdot 2^{E2}$, Assume $E1 > E2$
- Exact result: $(-1)^{s} \cdot M \cdot 2^{E}$
  - sign s, significand M
    - result of signed align and add, that is align at binary point
  - 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
  - If E out of range, overflow
  - Round M to fit *frac* precision

Properties of Floating Point Addition

- Compare to those of Abelian Group
  - Closed under addition, but may generate infinity or NaN
  - Commutative
  - Not associative
  - 0 is additive identity
  - Almost every element has an additive inverse, except for infinities and NaNs
- Monotonicity
  - $a \geq b \rightarrow a + c \geq b + c$ except for infinities and NaNs

Properties of Floating Point Multiplication

- Compare to Commutative Ring
  - Closed under multiplication, but may generate infinity or NaN
  - Commutative
  - **Not** associative: possibility of overflow, inexactness of rounding
  - 1 is multiplicative identity
  - Multiplication does **not** distribute over addition
- Monotonicity
  - $a \geq b \wedge c \geq 0 \rightarrow a * c \geq b * c$ except for infinities and NaNs

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` to `int`
    - truncates fractional part (like rounding to zero)
    - not defined when out of range or NaN
  - `int` to `double`
    - exact conversion, as long as `int` has $\leq$ 53 bit word size
  - `int` to `float`
    - will round according to rounding mode

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^E$
- 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 and distributivity
  - makes life difficult for compilers and serious numerical applications programmers