Floating Point Numbers

Kai Shen

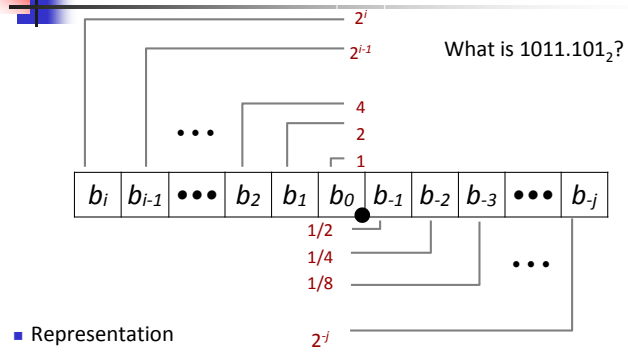
1

Floating Point

- Mathematical background: fractional binary numbers
- Representation on computers: IEEE floating point standard
- Rounding, addition, multiplication

2

Fractional Binary Numbers



What is 1011.101_2?

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

3

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**
  - Each bit has half the weight of the immediately adjacent bit on the left
  - Divide by 2 by shifting right; multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$

4

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	$0.0001100110011[0011]\ldots_2$

5

From Math to Computers

- Representation of fractional binary numbers on computers?
  - Think about the limited number of bits we have
  - Certain numbers bits before the binary point and certain numbers after → Fixed Point
    - limited range
  - Floating Point: numbers of bits before/after binary point may change (or floating binary point)
    - higher range, but may compromise precision (unlike integers, there is no absolute precision anyway)

6

IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point representation
    - 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

7

Floating Point Representation

- Numerical form:

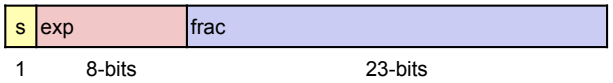
$$(-1)^s\ M\ 2^E$$

  - **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)

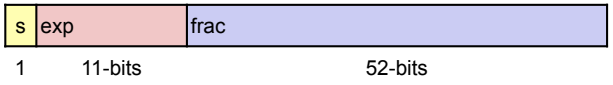
8

Precisions

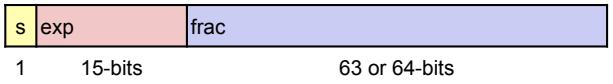
- Single precision: 32 bits



- Double precision: 64 bits



- Extended precision: 80 bits (Intel only)



9

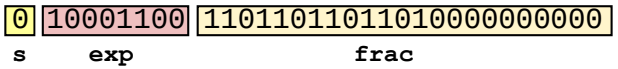
Normalized Values

- 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"

10

Normalized Encoding Example

- **`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:



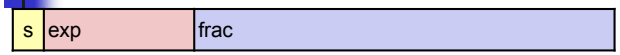
11

Denormalized Values

- Condition: exp = 000…0
- Significand coded with implied leading 0: ***M*** = $0.xxx\ldots x_2$
  - Smooth transition to 0
- Exponent value: ***E*** = 1–***Bias*** (instead of ***E*** = 0 – ***Bias***)
  - Same effective ***E*** as in the case of exp = 00…01
  - Smooth transition between largest value of exp = 00…00 and smallest value of exp = 00…01
- Special case: **`exp`** = 000…0, **`frac`** = 000…0
  - Represents zero value (distinct +0 and –0)

12

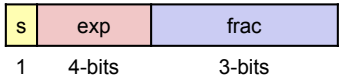
Special Values



- 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

13

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

14

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 numbers	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	

15

Interesting Numbers

{single,double}

Description	*exp*	*frac*	*Numeric Value*
Zero	00…00	00…00	0.0
Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
Single ≈ 1.4×10^{-45}			
Double ≈ 4.9×10^{-324}			
One	01…11	00…00	1.0
Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
Single ≈ 3.4×10^{38}			
Double ≈ 1.8×10^{308}			

16

Special Properties of Encoding

- Floating point 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
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

17

Floating Point Operations: Basic Idea

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

18

Rounding

- Rounding modes

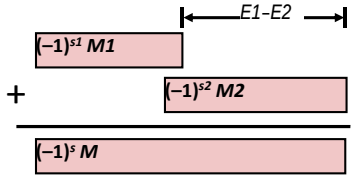
	1.40	1.60	1.50	2.50	–1.50
Towards zero	1	1	1	2	–1
Round down (−∞)	1	1	1	2	–2
Round up (+∞)	2	2	2	3	–1
Nearest even (default)	1	2	2	2	–2

- Nearest even
  - Round to nearest acceptable point
  - When exactly halfway between two adjacent points, round so that least significant digit is even
- Why?
  - Statistically unbiased, even digit is easier to represent

19

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

20

Properties of FP Add

- Commutative?
- Associative?
  - Overflow and inexactness of rounding
  - `(1e20 + -1e20) + 3.14  = 3.14`
  - `1e20 + (-1e20 + 3.14) = ??`
- Monotonicity: `a ≥ b ⇒ a+c ≥ b+c?`
  - Except for infinities & NaNs

21

Floating Point 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* x *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

22

Mathematical Properties of FP Mult

- Multiplication is commutative?
- Multiplication is associative?
  - Possibility of overflow, inexactness of rounding
- Monotonicity: `a ≥ b & c ≥ 0 ⇒ a*c ≥ b*c?`
  - Except for infinities & NaNs

23

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

24

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

25