Floating Point

(with contributions from Dr. Bin Ren, William & Mary Computer Science)

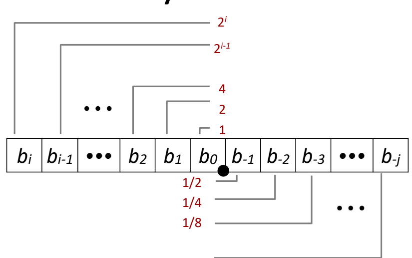
1

Floating Point

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

2

Fractional Binary Numbers



- **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

- **What is 1011.101_2?**
  - 1/2 + 1/8 = 5/8 11 5/8 or 11.625
- **What is 123.45 in binary?**
  - 123 = 1111011
  - to get the .45, use repeated multiplication by 2
    - if product < 1, bit is 0
    - if product >= 1, bit is 1 and subtract 1 from product

.45 * 2 =	
.9 * 2 =	0
(1.8 – 1) * 2 =	1
(1.6 – 1) * 2 =	1
(1.2 – 1) * 2 =	1
.4	0

  - 1111011.01110

4

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)
    - compare 101.11_2 with 10.111_2
  - Multiply by 2 by shifting left
    - compare 101.11_2 with 1011.1_2
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - notation sometimes seen: $1.0 - \varepsilon$

5

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?)

6

Representable Numbers

- **Numbers 0.111…11 base 2 represent numbers just below 1**
  - 0.111111 base 2 = 63/64
- **Only finite-length encodings**
  - 1/3 and 5/7 cannot be represented exactly
- **Fractional binary notation can only represent numbers that can be written x * 2^y (i.e. 63/64 = 63*2^{-6})**
  - Otherwise, approximated
  - Increasing accuracy = lengthening the binary representation but still have finite space

7

Practice

- **Fractional value of the following binary values:**
  - .01 =
  - .010 =
  - 1.00110 =
  - 11.001101 =
- **123.45 base 10**
  - Binary value =
  - FYI also equals:
    - 1.2345 x 10^2 in normalized form
    - 12345 x 10^{-2} using significand/mantissa/coefficient and exponent

8

Floating Point

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

9

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
    - Intel-based PCs
    - Apple
    - Unix/Linux
- **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

10

Floating Point Representation

- **Numerical Form:**

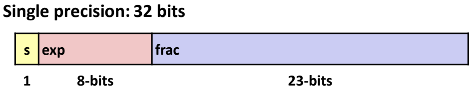
  $(-1)^s\, M\, 2^E$

  - **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)

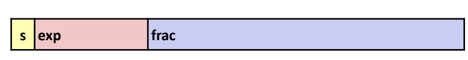
11

Precision options

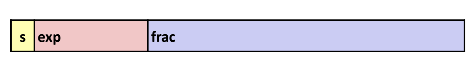
- **Single precision: 32 bits**



- **Double precision: 64 bits**



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



12

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…x₂**
  - 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"

13

Bias Notes

- **Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder.**
  - To solve this problem the exponent is biased to put it within an unsigned range suitable for comparison.
  - By arranging the fields so that the sign bit is in the most significant bit position, the biased exponent in the middle, then the mantissa in the least significant bits, the resulting value will be ordered properly, whether it's interpreted as a floating point or integer value. This allows high speed comparisons of floating point numbers using fixed point hardware.
- **When interpreting the floating-point number, the bias is subtracted to retrieve the actual exponent.**

14

Significand Notes

- **Represents the fraction, or precision bits of the number.**
- **It is composed of an implicit (i.e., hidden) leading bit and the fraction bits.**
- **In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in *normalized* form.**
  - This basically puts the radix point after the first non-zero digit
  - Nice optimization available in base two, since the only possible non-zero digit is 1.
  - Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly.
  - As a result, the mantissa/significand has effectively 24 bits of resolution, by way of 23 fraction bits.

15

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* = $1.\underline{1101101101101}_2$
  - **frac** = $\underline{1101101101101}0000000000_2$

- **Exponent**
  - *E* = 13
  - *Bias* = 127
  - *Exp* = 140 = 10001100_2

- **Result:**

s	exp	frac
0	10001100	11011011011010000000000

16

Normalized Encoding Example 2

- **Value: π, rounded to 24 bits of precision**
  - sign: 0

- **Significand**
  - *s* = 11.00100100001111110110 11 (including hidden bit)
  - *M* = $1.10010010000111111011011_2$ (x 2^1)
  - **frac** = $10010010000111111011011_2$

- **Exponent**
  - *E* = 1
  - *Bias* = 127
  - *Exp* = 128 = 10000000_2

- **Result:**

s	exp	frac
0	10000000	10010010000111111011011

17

Normalized Encoding Practice

- **Value: -π**
  - same as before; only sign bit changes

s	exp	frac
1	10000000	10010010000111111011011

- **Value: -78 3/8 (-78.375)**

`1001110.011 = 1.001110011 x 2⁶   6 + 127 = 133 or 10000101`

s	exp	frac
1	10000101	00111001100000000000000

- **Value: 63 11/32 (127.34375)**

`111111.01011 = 1.1111101011 x 2⁵   5 + 127 = 132 or 10000100`

s	exp	frac
0	10000100	11111010110000000000000

- **Value: -1/64 (-0.015625)**

`0.000001 = 1.0 x 2⁻⁶   -6 + 127 = 121 or 01111001`

s	exp	frac
1	01111001	00000000000000000000000

18

Denormalized Values

- Also called denormal or subnormal numbers
- Values that are very close to zero
  - Fill the "underflow" gap around zero
  - Gradual underflow = numeric values are spaced evenly near 0.0
- Any number with magnitude smaller than the smallest normal number
  - When the exponent field is all zeros
  - E = 1-bias
  - Significand M = f without implied leading 1
  - h = 0 (hidden bit)
- Representation of numeric value 0
  - -0.0 and +0.0 are considered different in some ways and the same in others

Denormalized Values

- In a normal floating point value, there are no leading zeros in the significand, instead leading zeros are moved to the exponent.
- e.g., 0.0123 would be written as $1.23 * 10^{-2}$
- Denormalized numbers are numbers where this representation would result in an exponent that is too small (the exponent usually having a limited range). Such numbers are represented using leading zeros in the significand.

Denormalized Values

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

- **Condition:** exp = 000…0
- **Exponent value:** $E = 1 - \text{Bias}$ (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

Denormalized Decoding Example

- What is the decimal value of the following?

s	exp	frac
0	00000000	00000010000111000000000

  - sign: 0 (positive)
  - we know it's a denormalized value because exp is all 0s
- Exponent
  $E = 1 - 127 = -126$ (same for all denormalized numbers)
- Significand
  frac = $00000010000111000000000_2$
  M = $0.00000010000111000000000_2$
- Result:
  - $0.00000010000111000000000 \times 2^{-126} = 1.0000111 \times 2^{-133}$
  - $10000111 \times 2^{-140} = 135 \times 2^{-140} = 9.69 \times 10^{-41}$

Denormalized Encoding Example

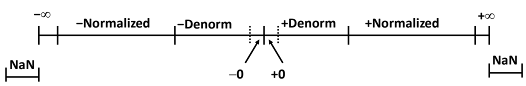
- Value: $-25\ 15/32 \times 2^{-132}$
  - sign: 1
  - power of 2 indicates denormalized number (< -126)
- Exponent
  $Bias$ = 127
  Exp = 00000000_2 (same for all denormalized numbers)
  E = 1 – 127 = – 126
- Significand
  s = $11001.01111 \times 2^{-132}$ (move 6 places left to match -126)
  M = $0.01100101111000000000000_2$ ($\times 2^{-126}$)
  frac = $01100101111000000000000_2$
- Result:

s	exp	frac
1	00000000	01100101111000000000000

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



25

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}			
Largest Denormalized	**00…00**	**11…11**	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
▪ Single ≈ 1.18×10^{-38}			
▪ Double ≈ 2.2×10^{-308}			
Smallest Pos. Normalized	**00…01**	**00…00**	$1.0 \times 2^{-\{126,1022\}}$
▪ Just larger than largest denormalized			
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}			

26

Floating Point

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

27

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

28

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
    - Increment exponent as shift right

Value	*Binary*	*Fraction*	*Exponent*
128	10000000	1.0000000	7
13	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

29

Dynamic Range (Positive Only)

$v = (-1)^s M 2^E$
n: E = Exp – Bias
d: E = 1 – Bias

	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	

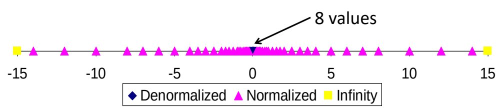
30

Distribution of Values

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

s	exp	frac
1	3-bits	2-bits

- **Notice how the distribution gets denser toward zero.**

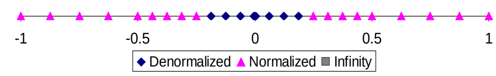


31

Distribution of Values (close-up view)

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

s	exp	frac
1	3-bits	2-bits



32

Special Properties of the IEEE Encoding

- **FP 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
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

33

Floating Point

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

34

Floating Point Operations: Basic Idea

- `x +f y = Round(x + y)`
- `x ×f y = Round(x × 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`

35

Rounding

- **Rounding Modes (illustrate 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

36

Closer Look at Round-To-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

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

Rounding Binary Numbers

- **Binary Fractional Numbers**
  - "Even" when least significant bit is `0`
  - "Half way" when bits to right of rounding position = $100..._2$
- **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

Rounding

`1.BBGRXXX`

- **Guard bit: LSB of result**
- **Round bit: 1st bit removed**
- **Sticky bit: OR of remaining bits**

- **Round up conditions**
  - Round = 1, Sticky = 1 → > 0.5
  - Guard = 1, Round = 1, Sticky = 0 → Round to even

Value	*Fraction*	*GRS*	*Incr?*	*Rounded*
128	1.0000000	000	N	1.000
13	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
142	1.0001110	011	Y	1.001
63	1.1111100	111	Y	10.000

Postnormalize

- **Issue**
  - Rounding may have caused overflow
  - Handle by shifting right once & incrementing exponent

Value	*Rounded*	*Exp*	*Adjusted*	*Result*
128	1.000	7		128
13	1.101	3		13
17	1.000	4		16
19	1.010	4		20
142	1.001	7		144
63	10.000	5	1.000/6	64

FP 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

FP Multiplication Example

- **What is the product of the following?**

s	exp	frac
1	10011100	11000000000000000000000
1	11110000	01100000000000000000000

- **Sign**
  - 1^1 = 0
- **Exponent**
  - *E1* = 156 - 127= 29 *E2* = 240 - 127= 113
  - *E* = 29 + 113 + 1 = 143 + 127 = 270 (1 0000 1110 - overflow)
- **Significand**
  - `frac` = $00110100000000000000000_2$
- **Result:**

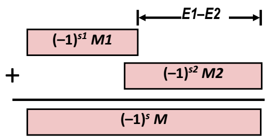
0	00001110	00110100000000000000000

```
     1.110
   x 1.011
     1 110
    11 100
 + 1110 000
  10.011 010
= 1.001101 x 2^1
```

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

Get binary points lined up



43

FP Addition Example

- **What is the sum of the following?**

s	exp	frac
0	01111101	00000000000000000011000
0	10000001	11000000000000000000000

- **Sign**
  - sign with larger exp = 0
- **Exponent**
  - $E1 = 125 - 127 = -2$ $\quad E2 = 129 - 127 = 2$
  - E = 1000 0001 ($E2$, or 2 + 127 = 129)

```
        .01 0000 0000 0000 0000 0011 000
    + 111 .00 0000 0000 0000 0000 0000 000
      111 .01 0000 0000 0000 0000 0011 000
    =  1.1101 0000 0000 0000 0000 0011  x 2^2
    =  1.1101 0000 0000 0000 0000 010   x 2^2
```

- **Significand**
  - `frac` = **1101 0000 0000 0000 0000 010**$_2$ (after round to even)
- **Result:**

s	exp	frac
0	10000001	11010000000000000000010

44

Floating Point

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

45

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
  - `int` → `float`
    - Cannot overflow; will round according to rounding mode
  - `int/float` → `double`
    - Exact conversion, as long as `int` has ≤ 53 bit word size
  - `float/double` → `int`
    - Truncates fractional part; like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - `double` → `float`
    - Can overflow (range smaller); may be rounded (precision smaller)

46

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

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


47

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

48