CS330 - Computer Organization and Assembly Language Programming

Lecture 9

-Floating Point Representation-

Professor : Mahmut Unan – UAB CS

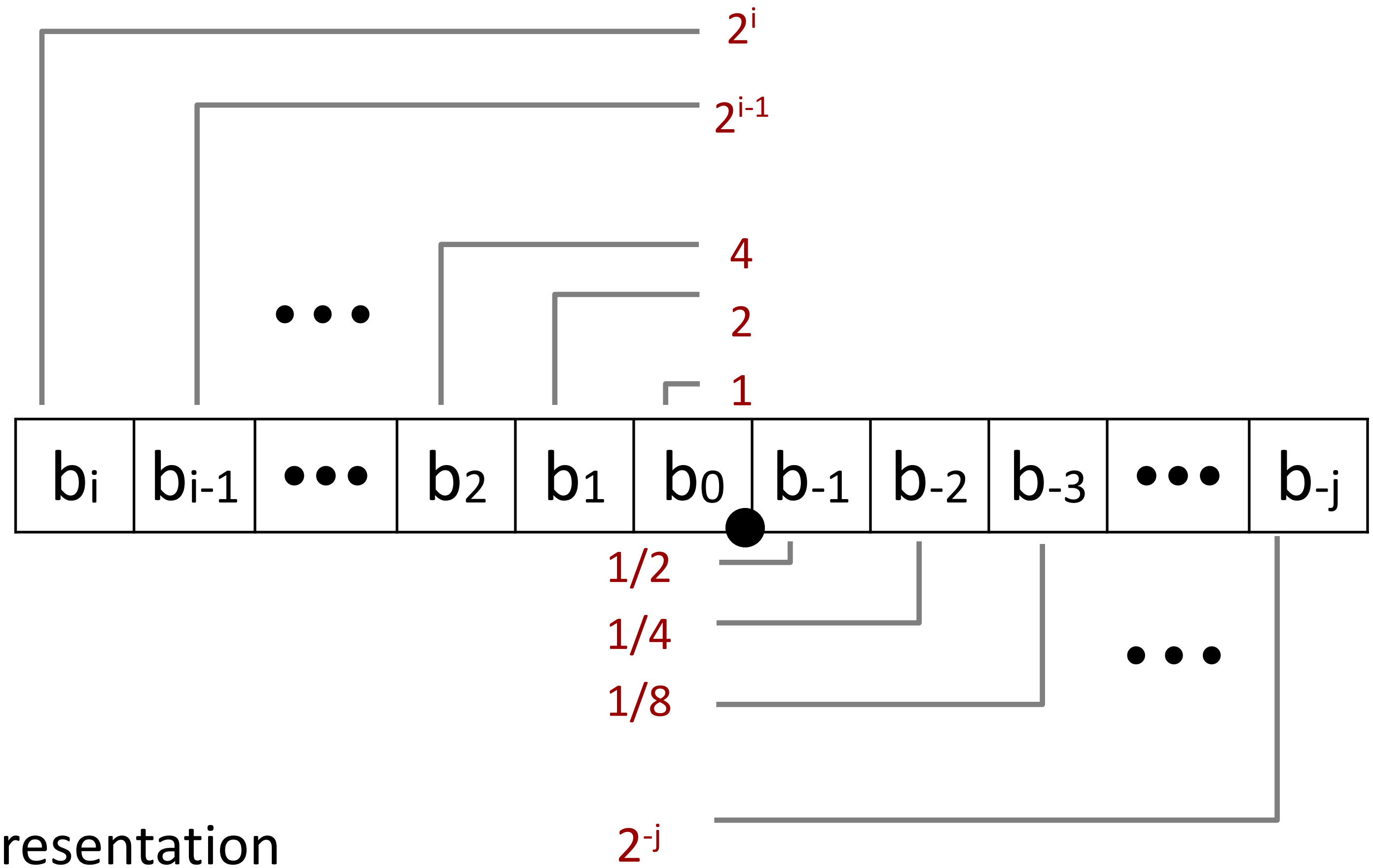
Agenda

- IEEE floating point standard
- Example and properties

Fractional binary numbers

- What is 1011.101_2?

Fractional Binary Numbers



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

Fractional Binary Numbers: Examples

- Value Representation

Value	Representation
5 3/4	$\mathtt{101.11}_2$
2 7/8	$\mathtt{10.111}_2$
1 7/16	$\mathtt{1.0111}_2$

- Observations
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

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]`…2
    - 1/5 `0.001100110011[0011]`…2
    - 1/10 `0.0001100110011[0011]`…2

- **Limitation #2**
  - Just one setting of binary point within the *w* bits
    - Limited range of numbers (very small values? very large?)

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

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)

Precision options

- Single precision: 32 bits

- Double precision: 64 bits

- The value encoded by a given bit representation can be divided into three different cases, depending on the value of **exp**.
- Case 1: Normalized Values
- Case 2: Denormalized Values
- Case 3: Special Values

Case 1: "Normalized" Values

$$\mathbf{v = (-1)^s\, M\, 2^E}$$

- *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 = $\mathbf{1.xxx…x_2}$**
  - 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"

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

Case 2: Denormalized Values

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

- **Condition: exp = 000…0**

- Exponent value: E = 1 – Bias (instead of E = 0 – Bias)
- Significand coded with implied leading 0: $M = 0.xxx\ldots x_2$
  - **`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

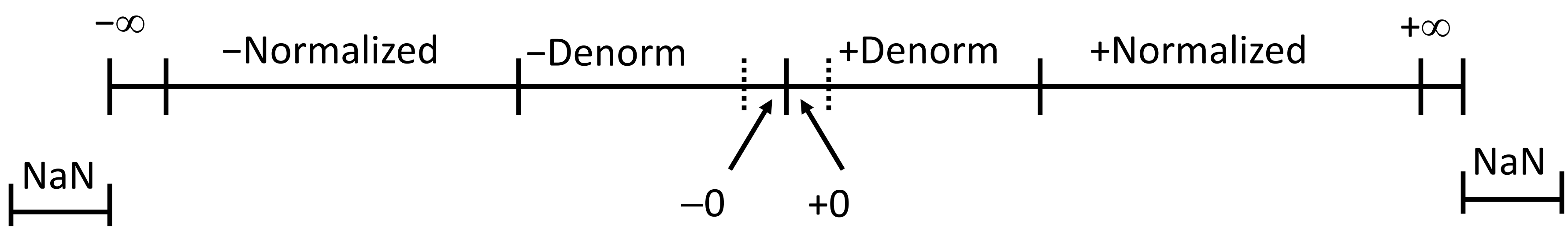
Case 3: Special Values

- Condition: **`exp`** = **`111`**…**`1`**

- Case: **`exp`** = **`111`**…**`1`**, **`frac`** = **`000`**…**`0`**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- Case: **`exp`** = **`111`**…**`1`**, **`frac`** $\neq$ **`000`**…**`0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), $\infty - \infty$, $\infty \times 0$

Visualization: Floating Point Encodings



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

Dynamic Range (Positive Only)

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

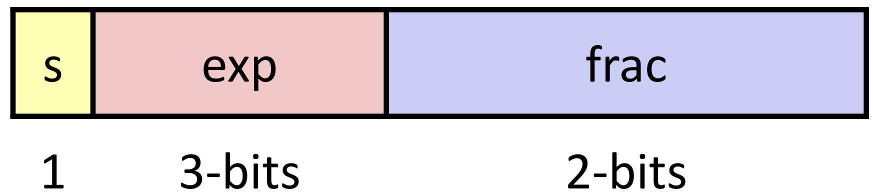
n: $E = Exp - Bias$

d: $E = 1 - Bias$

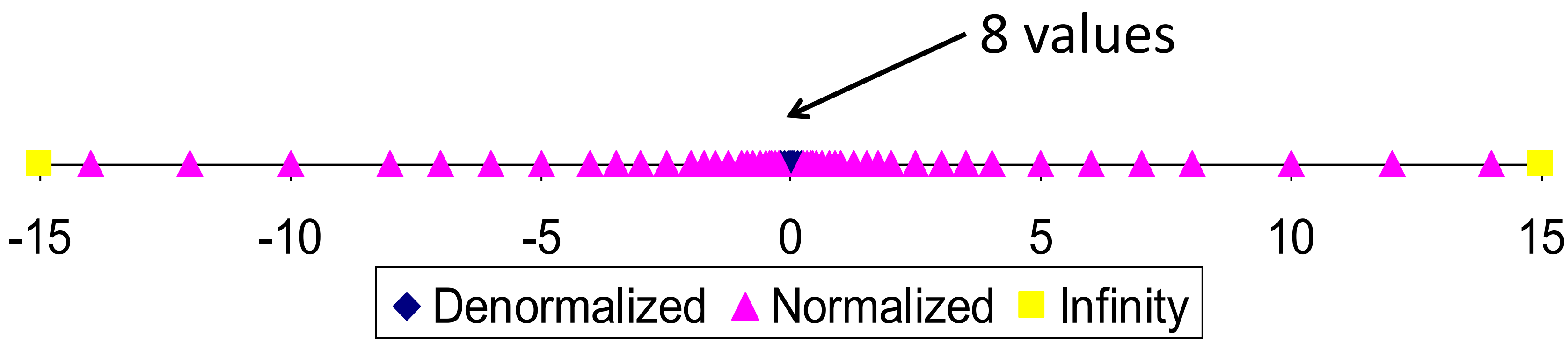
	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-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
	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	

Distribution of Values

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

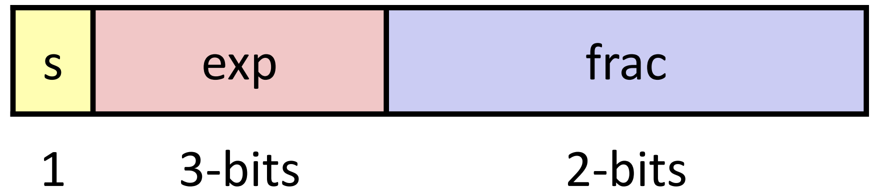


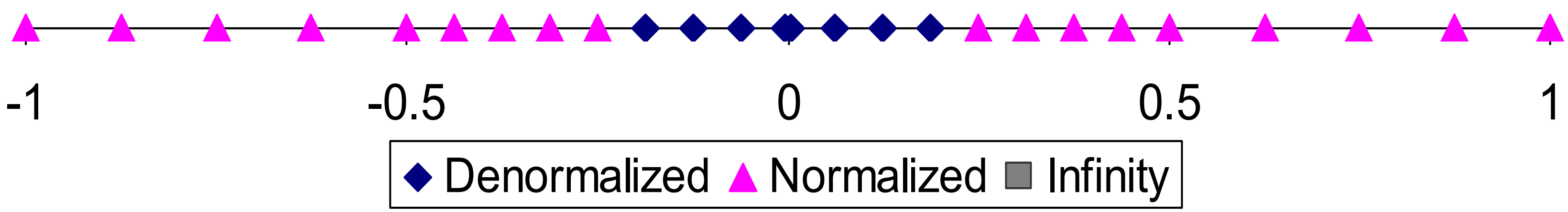
- **Notice how the distribution gets denser toward zero.**



Distribution of Values (close-up view)

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





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

Exercises

- **Convert 147.625 to IEEE 754 format**

- **Convert 0100 0011 1000 1111 0000 0000 0000 0000**

Convert 147.625 to IEEE 754 format

- Sign: 1 Bit   Exp: 8 Bits   Mantissa: 23 Bits

147: 10010011

128+16+2+1 =147

.625 = 5/8 → 101

½ + $^{1}/_{8}$

Convert 147.625 to IEEE 754 format

- Sign: 1 Bit   Exp: 8 Bits   Mantissa: 23 Bits

147: 10010011

    128+16+2+1 =147

.625 = 5/8 → 101

    ½ + $^{1}/_{8}$

10010011.101

Convert 147.625 to IEEE 754 format

- Sign: 1 Bit Exp: 8 Bits Mantissa: 23 Bits

147: 10010011

128+16+2+1 =147

.625 = 5/8 → 101

½ + $^1/_8$

10010011.101

=1.0010011101 * 2^7

Convert 147.625 to IEEE 754 format

- Sign: 1 Bit   Exp: 8 Bits   Mantissa: 23 Bits

147: 10010011

128+16+2+1 =147

.625 = 5/8 → 101

$\frac{1}{2} + {}^{1}/_{8}$

10010011.101

=1.0010011101 * 2^7

7= Exp - 127 → Exp=134

134→ 1000110 (exponent)

Convert **147.625** t

- Sign: 1 Bit   Exp: 8 Bits

147: 10010011

128+16+2+1 =147

.625 = 5/8 → 101

½ + $^{1}/_{8}$

Remember:

E = Exp – Bias

Bias;

Single precision: 127

Double precision: 1023

=1.0010011101 * 2^{7}

7= Exp - 127 → Exp=134

134→ 1000110 (exponent)

Convert 147.625 to IEEE 754 format

- Sign: 1 Bit   Exp: 8 Bits   Mantiss

147: 10010011

128+16+2+1 =147

.625 = 5/8 → 101

$\frac{1}{2} + {}^{1}/_{8}$

now, we will drop 1 and take the precision part

100

=1.0010011101 * 2^{7}

7= Exp - 127 → Exp=134

134→ 1000110 (exponent)

Convert 147.625 to IEEE 754 format

- Sign: 1 Bit   Exp: 8 Bits   Mantissa: 23 Bits

147: 10010011

128+16+2+1 =147

.625 = 5/8 → 101

½ + $^{1}/_{8}$

10010011.101

=1.0010011101 * 2^{7}

7= Exp - 127 → Exp=134

134→ 1000110 (exponent)

0010011101 → Mantissa

Convert 147.625 to IEEE 754 format

- Sign: 1 Bit Exp: 8 Bits Mantissa: 23 Bits

147: 10010011

128+16+2+1 =147

.625 = 5/8 → 101

$\frac{1}{2} + {}^{1}/_{8}$

10010011.101

=1.0010011101 * 2^7

7= Exp - 127 → Exp=134

134→ 1000110 (exponent)

0010011101 → Mantissa → it is 10 bits, we will need to add 13 zeros (Remember: Mantissa should be 23 bits for single precision)

Convert 147.625 to IEEE 754 format

- Sign: 1 Bit Exp: 8 Bits Mantissa: 23 Bits

147: 10010011

128+16+2+1 =147

.625 = 5/8 → 101

½ + $^1/_8$

10010011.101

=1.0010011101 * 2^7

7= Exp - 127 → Exp=134

134→ 1000110 (exponent)

0010011101 → Mantissa → it is 10 bits, we will need to add 13 zeros

00100111010000000000000 → Mantissa

1000110 → exp

0→ sign bit

Convert 147.625 to IEEE 754 format

- Sign: 1 Bit Exp: 8 Bits Mantissa: 23 Bits

147: 10010011

128+16+2+1 =147

.625 = 5/8 → 101

$\frac{1}{2} + {}^{1}/_{8}$

10010011.101

=1.0010011101 * 2^7

7= Exp - 127 → Exp=134

134→ 10000110 (exponent)

0010011101 → Mantissa → it is 10 bits, we will need to add 13 zeros
00100111010000000000000 → Mantissa
10000110 → exp
0→ sign bit

0 10000110 00100111010000000000000

Convert 147.625 to IEEE 754 format

- Sign: 1 Bit   Exp: 8 Bits   Mantissa: 23 Bits

147: 10010011

128+16+2+1 =147

.625 = 5/8 → 101

½ + $^{1}/_{8}$

10010011.101

=1.0010011101 * 2^{7}

7= Exp - 127 → Exp=134

134→ 10000110 (exponent)

0010011101 → Mantissa → it is 10 bits, we will need to add 13 zeros

00100111010000000000000 → Mantissa

10000110 → exp

0 → sign bit

sign | exp | Mantissa

01000011000100111010000000000000

Convert 0100 0011 1000 1111 0000 0000 0000 0000

- **0100 0011 1000 1111 0000 0000 0000 0000**

Convert 0100 0011 1000 1111 0000 0000 0000 0000

- **0100 0011 1000 1111 0000 0000 0000 0000**
- 0→sign bit → positive

Convert 0100 0011 1000 1111 0000 0000 0000 0000

- **0100 0011 1000 1111 0000 0000 0000 0000**
- 0→sign bit → positive
- 10000111→ convert to decimal → 135

E=Exp-Bias

135-127 = 8 → 2^8

Convert 0100 0011 1000 1111 0000 0000 0000 0000

- **0100 0011 1000 1111 0000 0000 0000 0000**
- 0 → sign bit → positive
- 10000111 → convert to decimal → 135

E=Exp-Bias

135-127 = 8 → 2^8

Since we always drop 1, now we can add it

1.00011110000000000000000

Convert 0100 0011 1000 1111 0000 0000 0000 0000

- **0100 0011 1000 1111 0000 0000 0000 0000**
- 0→sign bit → positive
- 10000111→ convert to decimal → 135

E=Exp-Bias

135-127 = 8 → 2^8

Since we always drop 1, now we can add it

1.00011110000000000000000

we can move the decimal points 8 times

100011110.000000000000000

Convert 0100 0011 1000 1111 0000 0000 0000 0000

- **0100 0011 1000 1111 0000 0000 0000 0000**
- 0→sign bit → positive
- 10000111→ convert to decimal → 135

E=Exp-Bias

135

convert to decimal

Since w we can add it

1.000 000

we ca points 8 times

100011110.000000000000000

Convert 0100 0011 1000 1111 0000 0000 0000 0000

- **0100 0011 1000 1111 0000 0000 0000 0000**
- 0→sign bit → positive
- 10000111→ convert to decimal → 135

E=Exp-Bias

135-127 = 8 → 2^8

Since we always drop 1, now we can add it

286 000 0 00

e th ints 8 times

100011110.000000000000000

Convert 0100 0011 1000 1111 0000 0000 0000 0000

- **0100 0011 1000 1111 0000 0000 0000 0000**
- 0 → sign bit → positive
- 10000111 → convert to decimal → 135

E=Exp-Bias

135-127 = 8 → 2^8

Since we always drop 1, now we can add it

286.00

ts 8 times

100011110.000000000000000

Floating Point Operations: Basic Idea

- $\mathbf{x} +_f \mathbf{y} = \mathrm{Round}(\mathbf{x} + \mathbf{y})$

- $\mathbf{x} \times_f \mathbf{y} = \mathrm{Round}(\mathbf{x} \times \mathbf{y})$

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

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

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 = $\mathbf{100\ldots}_2$

- **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

FP Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \times\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign s: s1 ^ s2
  - Significand M: M1 x M2
  - Exponent E: E1 + E2

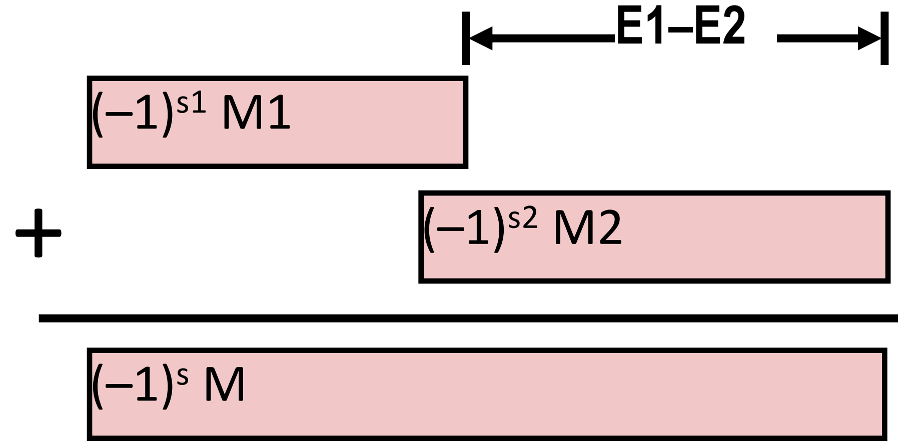
- **Fixing**
  - If $M \geq 2$, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit `frac` precision

- **Implementation**
  - Biggest chore is multiplying significands

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

Get binary points lined up



Mathematical Properties of FP Add

- Compare to those of Abelian Group
  - Closed under addition? Yes
    - But may generate infinity or NaN
  - Commutative? Yes
  - Associative? No
    - Overflow and inexactness of rounding
    - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
  - 0 is additive identity? Yes
  - Every element has additive inverse? Almost
    - Yes, except for infinities & NaNs
- Monotonicity Almost
  - $a \geq b \Rightarrow a+c \geq b+c$?
    - Except for infinities & NaNs

Mathematical Properties of FP Mult

- **Compare to Commutative Ring**
  - Closed under multiplication? Yes
    - But may generate infinity or NaN
  - Multiplication Commutative? Yes
  - Multiplication is Associative? No
    - Possibility of overflow, inexactness of rounding
    - Ex: `(1e20*1e20)*1e-20`= `inf`, `1e20*(1e20*1e-20)`= `1e20`
  - 1 is multiplicative identity? Yes
  - Multiplication distributes over addition? No
    - Possibility of overflow, inexactness of rounding
    - `1e20*(1e20-1e20)`= `0.0`, `1e20*1e20 – 1e20*1e20 =` `NaN`
- **Monotonicity**
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$? Almost
    - Except for infinities & 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`** → **`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

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

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (double)(float) d`
- `f == -(-f);`
- `1.0/2 == 1/2.0`
- `d * d >= 0.0`
- `(d+f)-d == f`

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

Table 10.3 IEEE 754 Format Parameters

Parameter	Format		
	binary32	**binary64**	**binary128**
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	–126	–1022	–16382
Approx normal number range (base 10)	$10_{-38}, 10_{+38}$	$10_{-308}, 10_{+308}$	$10_{-4932}, 10_{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2_{23}	2_{52}	2_{112}
Number of values	$1.98 \times 2_{31}$	$1.99 \times 2_{63}$	$1.99 \times 2_{128}$
Smallest positive normal number	2_{-126}	2_{-1022}	2_{-16362}
Largest positive normal number	$2_{128} - 2_{104}$	$2_{1024} - 2_{971}$	$2_{16384} - 2_{16271}$
Smallest subnormal magnitude	2_{-149}	2_{-1074}	2_{-16494}

* not including implied bit and not including sign bit

Table 10.5
Interpretation of IEEE 754 Floating-Point Numbers (page 1 of 3)

	Sign	**Biased exponent**	**Fraction**	**Value**
positive zero	0	0	0	0
negative zero	1	0	0	–0
plus infinity	0	all 1s	0	∞
Minus infinity	1	all 1s	0	–∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	$0 < e < 255$	f	$2_{e-127}(1.f)$
negative normal nonzero	1	$0 < e < 255$	f	$-2_{e-127}(1.f)$
positive subnormal	0	0	$f \neq 0$	$2_{e-126}(0.f)$
negative subnormal	1	0	$f \neq 0$	$-2_{e-126}(0.f)$

(a) binary32 format

Table 10.5
Interpretation of IEEE 754 Floating-Point Numbers (page 2 of 3)

	Sign	**Biased exponent**	**Fraction**	**Value**
positive zero	0	0	0	0
negative zero	1	0	0	–0
plus infinity	0	all 1s	0	∞
Minus infinity	1	all 1s	0	$-\infty$
quiet NaN	0 or 1	all 1s	$\neq 0$; first bit $= 1$	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$; first bit $= 0$	sNaN
positive normal nonzero	0	$0 < e < 2047$	f	$2_{e-1023}(1.f)$
negative normal nonzero	1	$0 < e < 2047$	f	$-2_{e-1023}(1.f)$
positive subnormal	0	0	$f \neq 0$	$2_{e-1022}(0.f)$
negative subnormal	1	0	$f \neq 0$	$-2_{e-1022}(0.f)$

(a) binary64 format

Creating Floating Point Number

- Steps
  - Normalize to have leading 1
  - Round to fit within fraction
  - Postnormalize to deal with effects of rounding

s	exp	frac
1	4-bits	3-bits

- 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`

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

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
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	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
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

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}			