Floating Point

Professor Hugh C. Lauer

CS-2011, Machine Organization and Assembly Language

Today: Floating Point

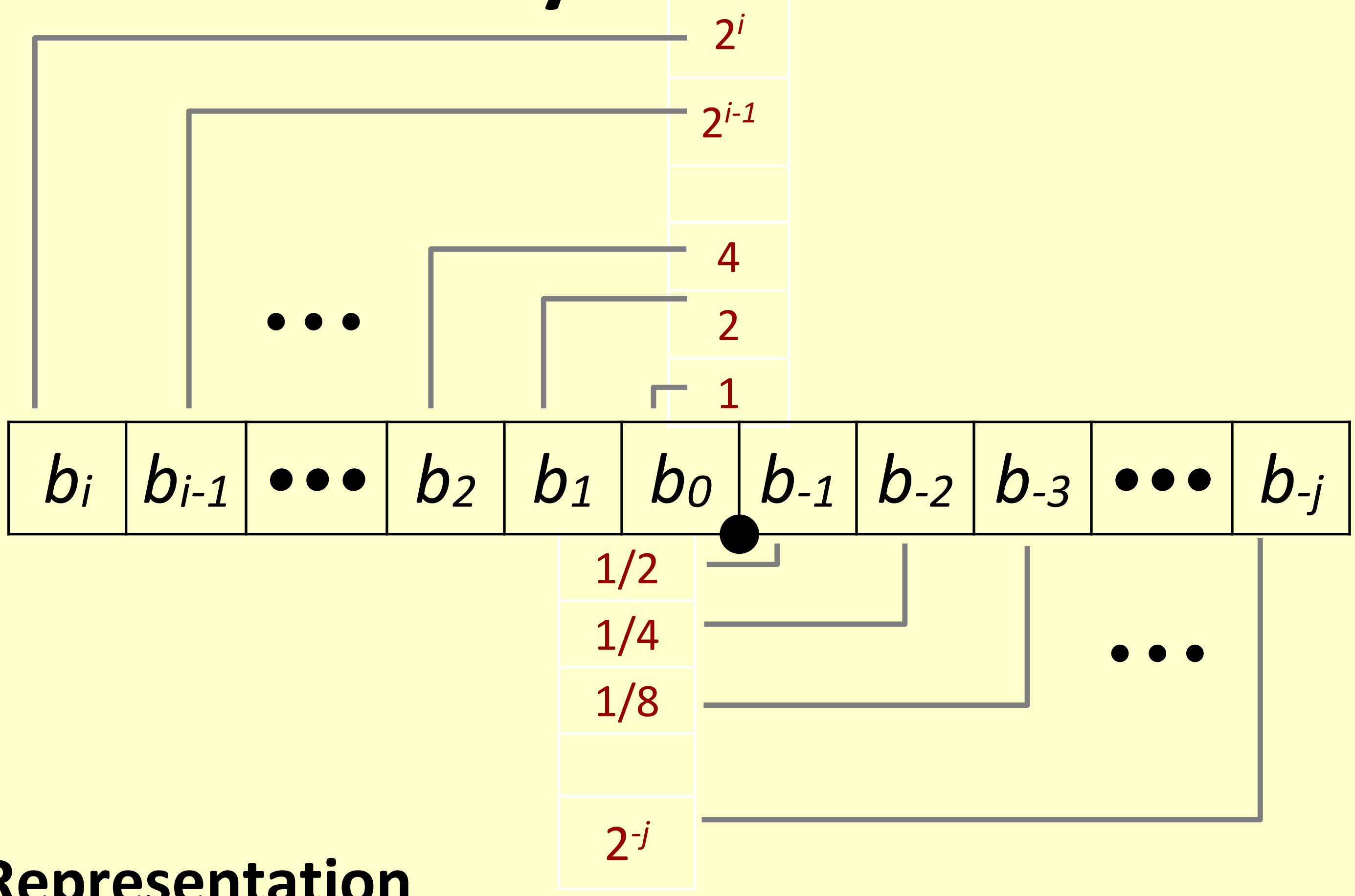
Reading Assignment: §2.4

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

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	101.11_2
2 7/8	010.111_2
1 7/16	001.0111_2

- **Observations**
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - 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 $0.0101010101[01]\ldots_2$
    - 1/5 $0.001100110011[0011]\ldots_2$
    - 1/10 $0.0001100110011[0011]\ldots_2$
- **Limitation #2**
  - Many, many bits needed for very large or small numbers with fixed binary point
    - Planck's constant:– 6.626068×10^{-34} erg sec
    - Avogadro's number:– 6.022×10^{23} particles per mole

Today: Floating Point

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

Floating point

- **A way to *approximate* real numbers in computers**
  - And also most rational numbers

- **Examples:–**
  - 3.14159265358979323846 — π
  - 2.99792458×10^8 m/s — *c*, the velocity of light
  - $6.62606885 \times 10^{-27}$ erg sec — *h*, Planck's constant

- **In C (and most other programming languages):–**
  - 3.14159265358979323846
  - 2.99792458e8
  - 6.62606885e-27

IEEE floating point

- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
    - Same equations gave different answers on different machines!
  - Now supported by all major processors

- **Driven by numerical concerns**
  - Nice standards for rounding, overflow, underflow
  - Difficult to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

Floating point representation

- **Numerical Form:**

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

  - **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

Precisions

- **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	63 or 64-bits

Normalized values

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

- **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"

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

Denormalized values

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

- **Condition: exp = 000…0**

- **Exponent value: $E = -Bias + 1$ (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 very close to 0.0
    - Lose precision as get smaller
    - Equispaced

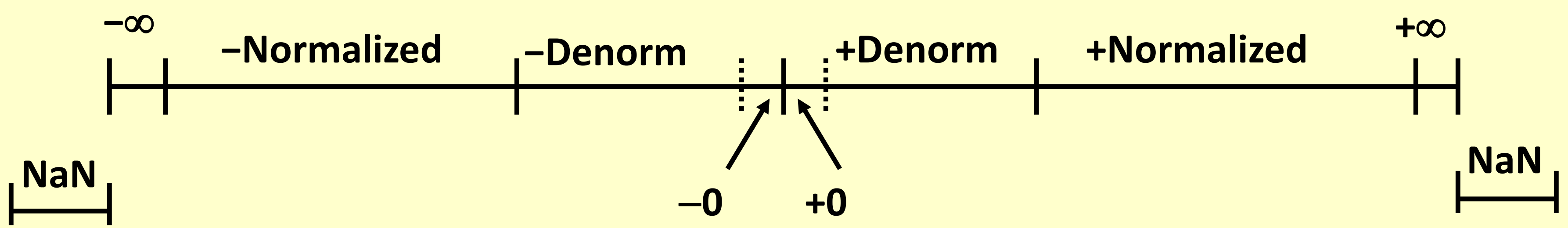
Special values

- **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` ≠ 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



Today: Floating Point

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

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

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

n: E = Exp – Bias

d: E = 1 – Bias

Dynamic range (positive only)

	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	

Dynamic range (positive only)

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

n: E = Exp – Bias

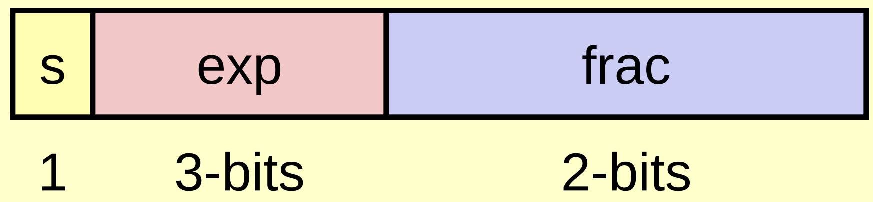
d: E = 1 – Bias

	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	
	…					
					4/8*128 = 224	
					5/8*128 = 240	largest norm
	0	1111	000	n/a	inf	

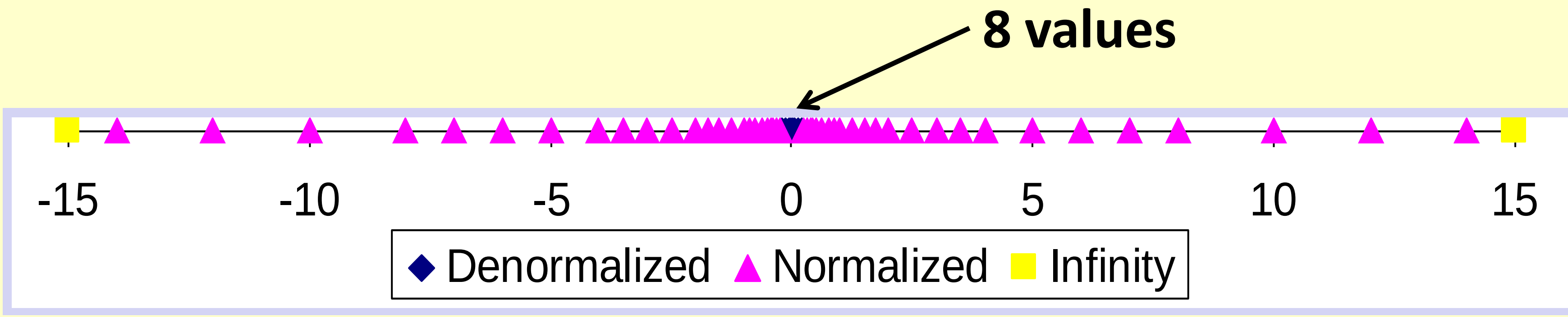
Note: the value *1* has exponent = bias and significand = all zeros

Distribution of values

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

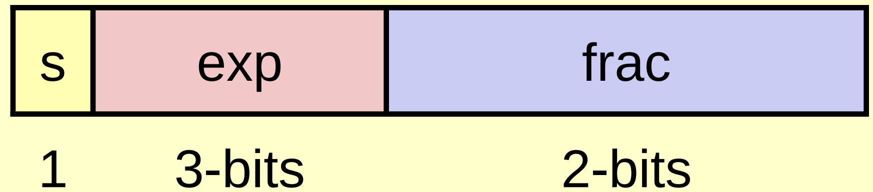


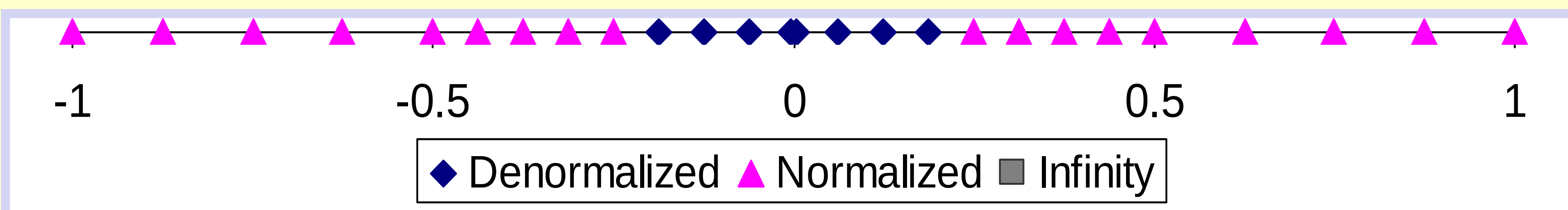
- **Notice how the distribution gets denser toward zero.**



Distribution of values (close-up view)

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





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}			

Special properties of 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

Today: Floating Point

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

Floating point operations: Basic idea

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

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

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

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

- **What are the advantages of the modes?**

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

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)

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

FP multiplication

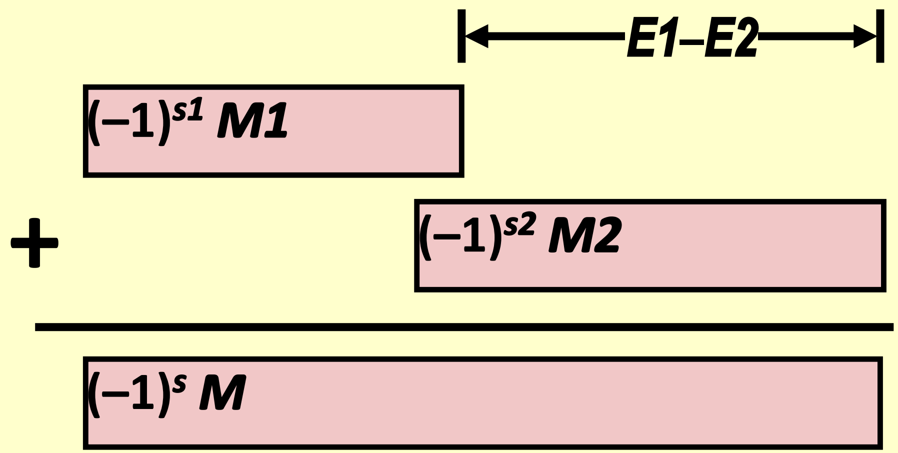
- $(-1)^{s1}\, M1\;\, 2^{E1}\quad \times \quad (-1)^{s2}\, M2\;\, 2^{E2}$
- **Exact Result:** $(-1)^{s}\, M\;\, 2^{E}$
  - Sign *s*: $s1$ ^ $s2$
  - Significand *M*: $M1 \times M2$
  - Exponent *E*: $E1 + E2$

- **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

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

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***
    - Except for infinities & NaNs
- **Monotonicity**
  - $a \geq b \Rightarrow a+c \geq b+c$? ***Almost***
    - 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

- **Monotonicity**
  - $a \geq b \ \& \ c \geq 0 \ \Rightarrow a * c \geq b * c$? **Almost**
    - Except for infinities & NaNs

Today: Floating Point

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

Floating point in C

- ***C* guarantees two levels**
  - **`float`** single precision
  - **`double`** double precision
- **Conversions/casting**
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representations
  - **`double`/`float`** → **`int`**
    - Truncate fractional part — i.e., rounding toward zero
    - Not defined when out-of-range, **`NaN`**, etc.; — generally set to *TMin*
  - **`int`** → **`double`**
    - Exact conversion for numbers that fit into ≤ 53 bits
  - **`int`** → **`float`**
    - 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 == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0     ⇒  ((d*2) < 0.0)`
- `d > f       ⇒  -f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

Today: Floating Point

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

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

Questions?

More Slides

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

Sticky bit: OR of remaining bits

Round bit: 1st bit removed

- **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

Questions?