18-600 Foundations of Computer Systems

Lecture 4:
"Floating Point"

- Required Reading Assignment:
  - **Chapter 2 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron**
- Assignments for This Week:
  - Lab 1

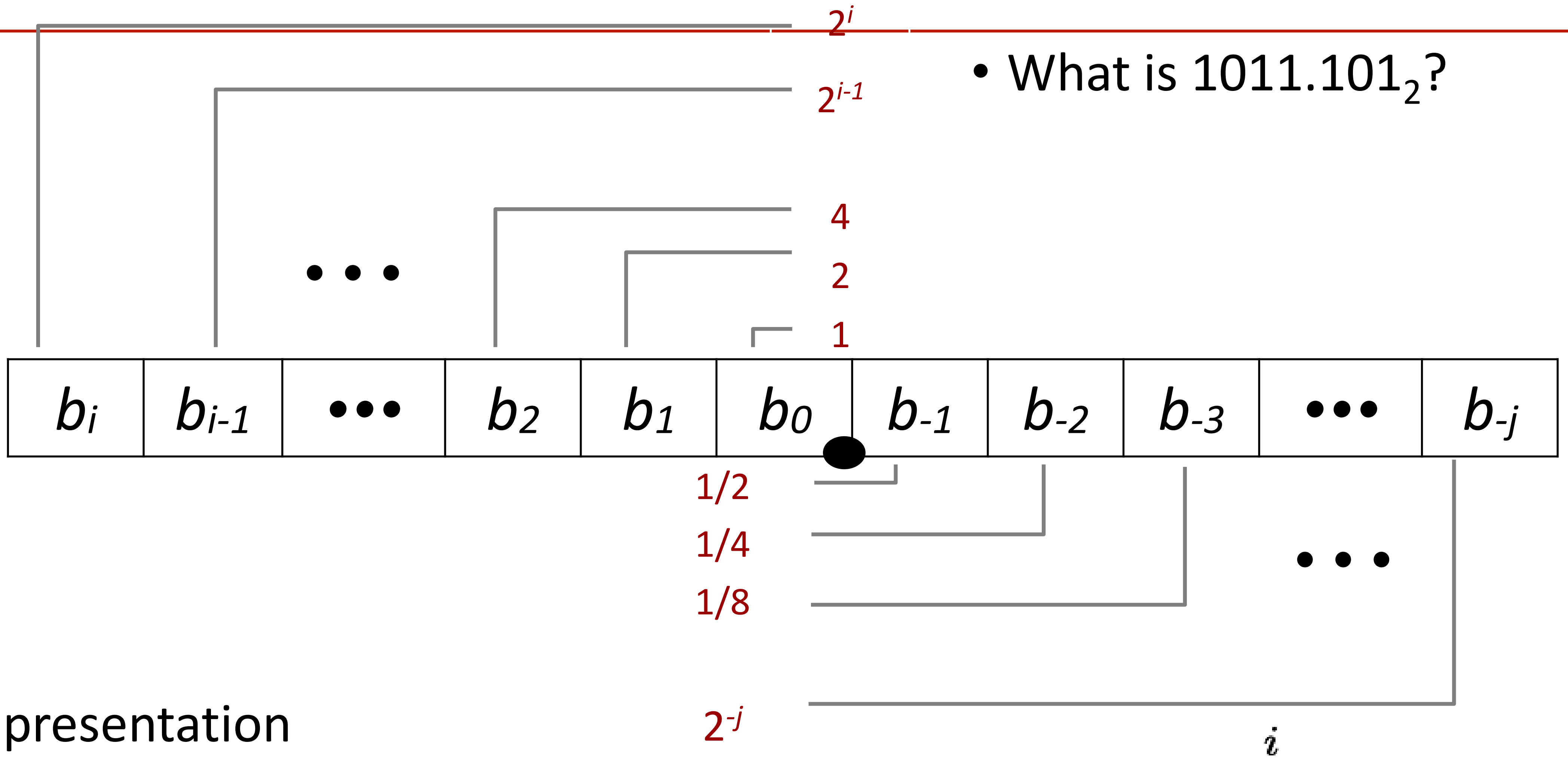
Electrical & Computer ENGINEERING

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Creating floating point number
- Summary of floating point number
- Quick introduction of assembly language

Fractional Binary Numbers

- What is 1011.101_2?



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \times 2^k$$

Fractional Binary Numbers: Examples

- **Value** **Representation**

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)
  - 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 $1.0 - \varepsilon$

Floating Point Representation

- Numerical Form:

$$(-1)^s \boldsymbol{M} \; 2^{\boldsymbol{E}}$$

  - **Sign bit *s*** determines whether number is negative or positive
  - **Significand *M*** (mantissa) 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

Precision options

- 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

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
  - "Fixed precision" not one-size-fits-all
    - More to the left (fewer digits to the left of it, but more to the right)? Smaller magnitude.
    - More to the right (more digits to the left of it, but fewer to the right)? Less precision.

Today: Floating Point

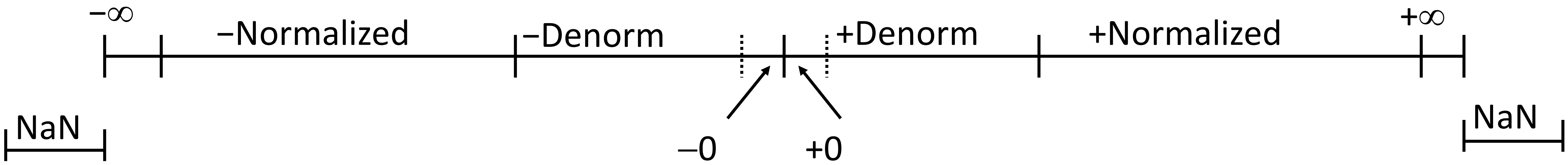
- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Creating floating point number
- Summary of floating point number
- Quick introduction of assembly language

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

Visualization: IEEE-like Floating Point Ranges

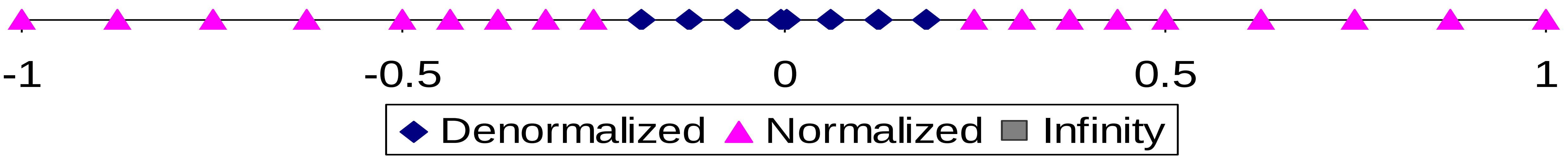


Notice:

- Two different range types (Normalized, Denormalized)
- Special values [Not-A-Number(NaN), infinity)
- Weirdness (+/- Zero)

Visualization: IEEE-like Distribution of Values

- Example IEEE-like format
- Distribution is dense near zero for greatest precision
  - Distribution is uniform and close nearest to zero in denormalized space
  - Distribution grows from there in normalized space
- The normalized range allows precision to be increasingly traded for magnitude as moving away from zero toward extremes



Encoding Exponent: Normalized Values

- Need to encode positive and negative exponents
  - As before, don't want sign bit, as it makes the number line discontinuous and breaks math
  - Don't want 2s compliment, because we want a smooth transition to denormalized numbers (We'll see this shortly)

- Subtract "half" of range from value to provide a negative range and put zero near center. The value subtracted is called the *bias*.
  - The range can't be exactly half: There are an odd number of numbers once 0 is considered
  - Since dividing by 2 integer-style rounds down (truncates), the bias will be half minus 1
  - I.e., The bias is 2^{k-1}-1, where *k* is the number of exponent bits.
  - Subtracting when interpreting implies adding when encoding.

- *Examples:*
  - Single precision (8-bit exponent): Bias = (2^8 -1)=127
  - Double precision (11-bit exponent): Bias = (2^{11} -1)= 1023

Encoding Significand: "Normalized" Values

- Significand encoded in unsigned binary
  - "Negative sign" is encoded as a separate, leading flag

- Encoded with implied leading 1: ***M*** = $1.xxx...x_2$
  - We know there is a leading 0 in the significand
    - 0 is the special case of an all 0 bit pattern (to keep int and float 0s comparable)
  - So, why store it. Just assume it is there and put it back upon decode.
  - Get extra leading bit for "free

- xxx…x: bits of frac field encode number [1.0, 2.0)
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)

Encoding Normalized Numbers

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

- $(-1)^s$
  - If negative, set s=1, so $(-1)^s = -1$, making the number negative
  - If negative, set s=0, so $(-1)^0 = 1$, making the number non-negative
- ***M***
  - Encode number in Base-2 scientific notation, shifting point until leading digit is a 1
  - Forget the 1, we know it is there. Store as many of the high-order bits as possible in the allocated number of bits, drop the rest. They are low-order, anyway.
- $2^{\boldsymbol{E}}$
  - Figure out the exponent from the scientific notation
  - Figure out the bias, based upon the number of bits allocated to the exponent
    - The bias is $2^{k-1}-1$, where *k* is the number of exponent bits.
  - Add the bias to the exponent.
  - Store the biased value in the space provided for the exponent
  - Changing exponent provides "normalization"

Normalized Encoding Example

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

$E = Exp - Bias$

- Value: `float F = 18600.0;`
  - $18600_{10} = 100100010101000_2 = 1.00100010101_2 \times 2^{14}$
- Significand
  - *M* = $1.\underline{00100010101}_2$
  - **frac** = $\underline{00100010101}000000000000_2$ **(23 bits)**
- Exponent
  - *E* = 14 (Unbiased exponent)
  - *Bias* = 127
  - *Exp* = 141 = 10001101_2 (BIASED exponent)
- Result:

s	exp	frac
0	10001101	00100010101000000000000

Denormalized Values

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

- Condition: exp = 000…0
  - The exponent is no longer changing
  - **`exp`** = 000…0, **`frac`** ≠ 000…0
    - Numbers closest to 0.0
    - Fixed exponent makes numbers equispaced – no normalization
- Exponent value: $E = 1 - \text{Bias}$ (instead of $E = 0 - Bias$)
  - Encoding numbers smaller than normalized range
  - Bias is fixed at one smaller than what it was.
- Significand coded with implied leading 0: $M = 0.xxx\ldots x_2$
  - Can't shift it to find a 1. If there was a leading one, value would be in normalized range.
  - **`xxx`**…**`x`**: bits of **`frac`**
- Zero Value: **`exp`** = 000…0, **`frac`** = 000…0
  - Note distinct values: +0 and –0 (why?)

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

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Creating floating point number
- Summary of floating point number
- Quick introduction of assembly language

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)

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

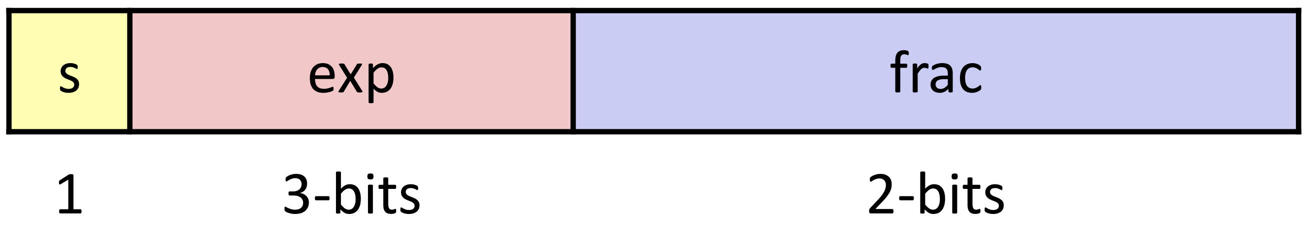
n: E = Exp – Bias

d: E = 1 – Bias

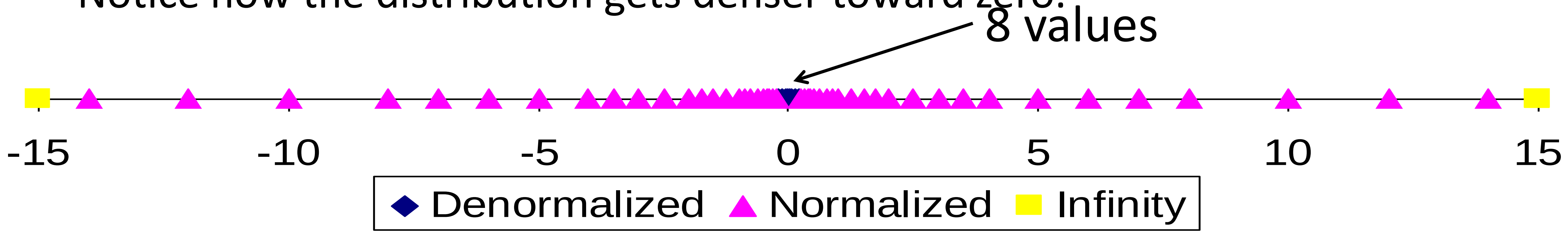
	s exp frac	E	Value	
	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	closest to zero
Denormalized numbers	0 0000 010	-6	2/8*1/64 = 2/512	
	…			
	0 0000 110	-6	6/8*1/64 = 6/512	
	0 0000 111	-6	7/8*1/64 = 7/512	largest denorm
	0 0001 000	-6	8/8*1/64 = 8/512	smallest norm
	0 0001 001	-6	9/8*1/64 = 9/512	
	…			
	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	

A Second Look: Distribution of Values

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



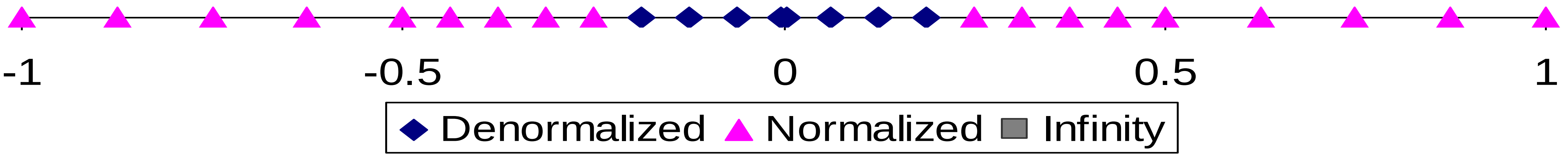
- Notice how the distribution gets denser toward zero.



A Second Look: Value Distribution (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



Why Bias, Not 2s Complement for Exponent?

- It makes for nice addition and subtraction of exponents, which is good for multiplication and division, right?

	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	

Notice smooth transition across exponents. Values change by 1/16 as mantissa Increments, within and across exponent ranges.

Why not 2s Complement for Mantissa?

- It worked for us nice before, right?
  - We can't directly add or subtract them, anyway
  - We need to adjust for exponent
  - Little-to-no gain
  - Added cost to complement, etc.

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

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Creating floating point number
- Summary of floating point number
- Quick introduction of assembly language

Floating Point Operations: Basic Idea

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

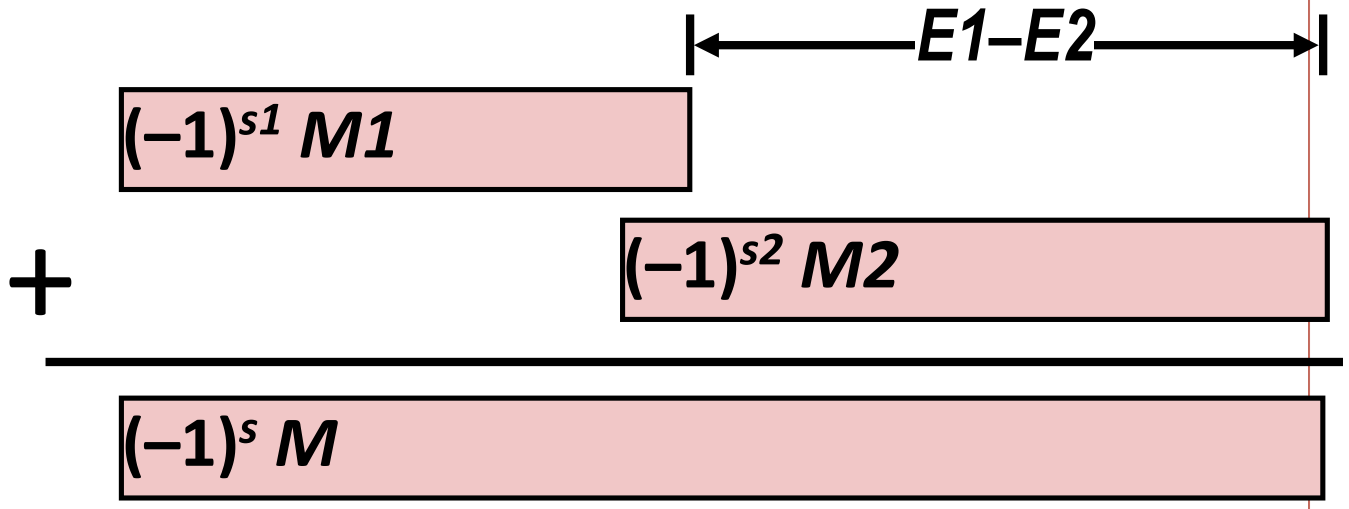
FP Multiplication

- $(-1)^{s1}$ $\boldsymbol{M1}$ $2^{\boldsymbol{E1}}$ x $(-1)^{\boldsymbol{s2}}$ $\boldsymbol{M2}$ $2^{\boldsymbol{E2}}$
- Exact Result: $(-1)^{s}$ $\boldsymbol{M}$ $2^{\boldsymbol{E}}$
    - 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

Floating Point Addition

- $(-1)^{s1}\ \boldsymbol{M1}\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ \boldsymbol{M2}\ \ 2^{E2}$
  - Assume $E1 > E2$
- Exact Result: $(-1)^{s}\ \boldsymbol{M}\ \ 2^{E}$
  - 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



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
  - $a \geq b \Rightarrow a+c \geq b+c$? ***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
    - `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

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Creating floating point number
- Summary of floating point number
- Quick introduction of assembly language

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);`
- `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
- Creating floating point number
- Summary of floating point number
- Quick introduction of assembly language

Creating Floating Point Number

s	exp	frac
1	4-bits	3-bits

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

- 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

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Creating floating point number
- Summary of floating point number
- Quick introduction of assembly language

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

Today: Floating Point

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

18-600 Foundations of Computer Systems

Lecture 5:
"Machine Programs I: (Basics)"

September 11, 2017

Next Time …

Electrical & Computer ENGINEERING