Data Representation – Floating Point

CSCI 2400 / ECE 3217: Computer Architecture

Instructor:

David Ferry

Slides adapted from Bryant & O'Hallaron's slides via Jason Fritts

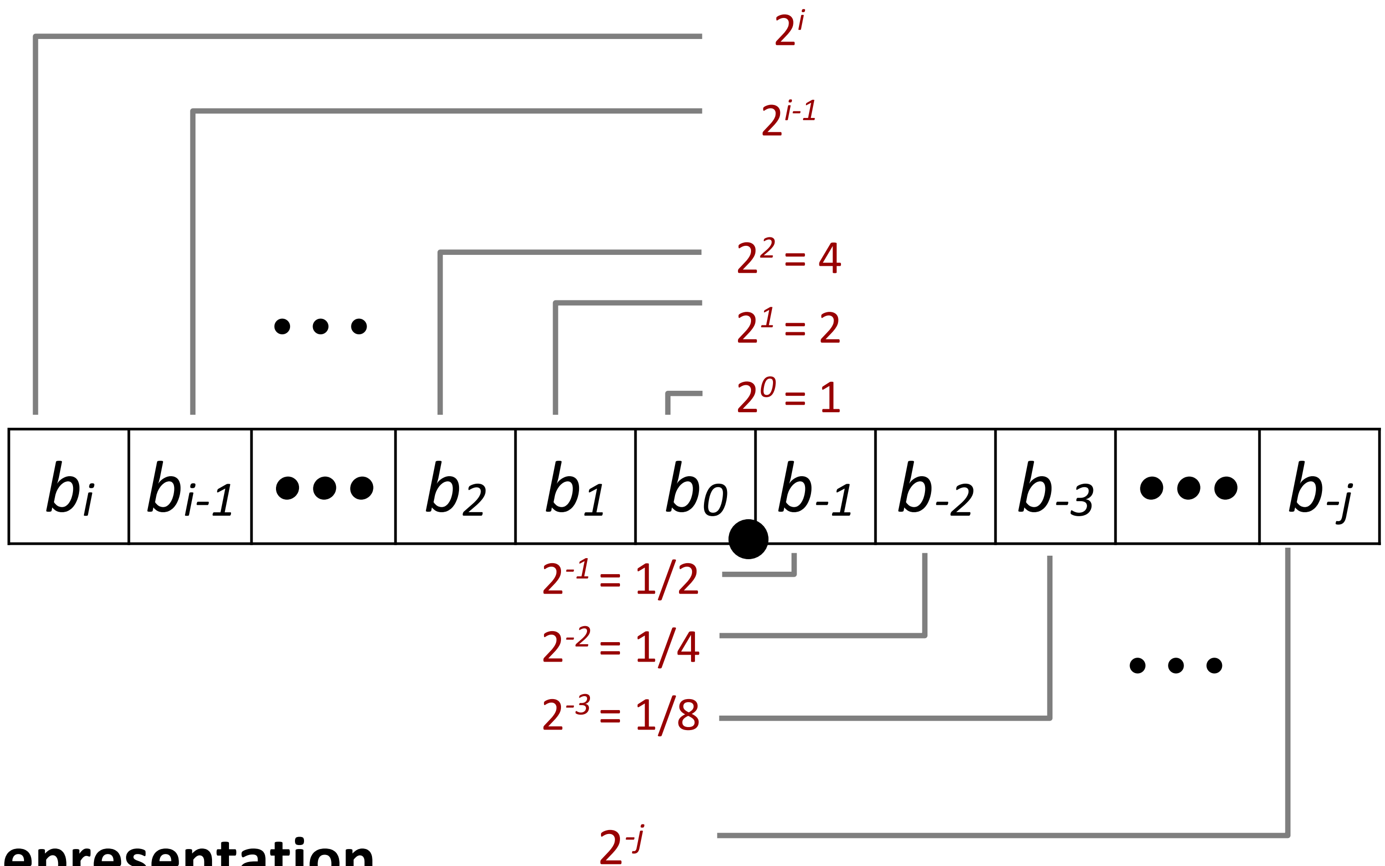
Today: Floating Point

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

Fractional binary numbers

- **What is 1011.101_2?**
- **How can we express fractions like ¼ in binary?**

Place-Value 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	$= 4 + 1 + {}^{1}/_{2} + {}^{1}/_{4} = 5\,{}^{3}/_{4}$
$2\,{}^{7}/_{8}$	10.111_2	$= 2 + {}^{1}/_{2} + {}^{1}/_{4} + {}^{1}/_{8} = 2\,{}^{7}/_{8}$
${}^{25}/_{64}$	0.011001_2	$= {}^{1}/_{4} + {}^{1}/_{8} + {}^{1}/_{64} = {}^{25}/_{64}$

- **Observations**
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
- **Limitations**
  - 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]**…$_2$

- Limited range when used with “fixed point” representations

Quick Check

- **Convert:**

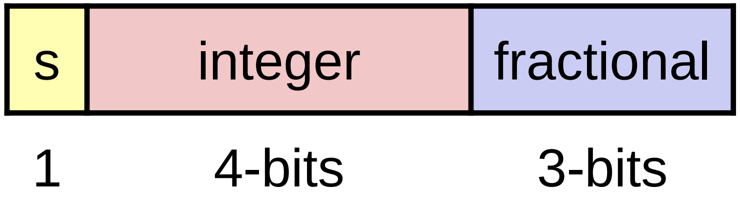
  $255\ {}^{9}\!/_{16}$ **to binary**

  10101.10101_2 **to decimal**

Quick Check

Suppose an 8-bit *fixed-point* representation with:

- **One sign bit**
- **Four integer bits**
- **Three fractional bits**



Convert:

$12\ ^{1}/_{8}$ to binary

$-6\ ^{3}/_{8}$ to binary

11010110_2 to decimal

What bit pattern(s) have the largest positive value? What is it?

What bit pattern(s) have the value closest to zero?

What bit pattern(s) have the value of zero?

Today: Floating Point

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

Floating Point Representation

- **Similar to scientific notation**

  E.g.: $1.25 \times 10^3 = 1{,}250$

  E.g.: $2.78 \times 10^{-2} = 0.0278$

- **FP is this concept but with an efficient binary format! But…**
  - Uses base 2 instead of base 10
  - Places restrictions on how certain values are represented
  - Deals with finiteness of representation

Floating Point Representation

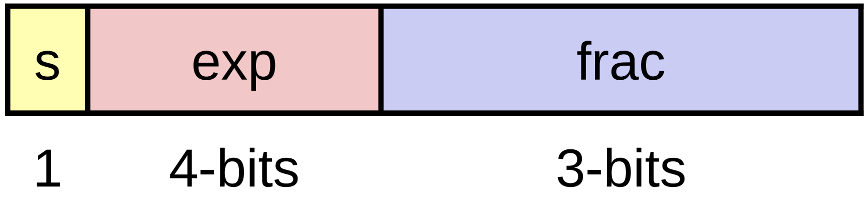
- **Numerical Form:**

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

  - **Sign bit** ***s*** determines whether number is negative or positive
  - **Significand** (mantissa) ***M*** normally a fractional value in range [1.0, 2.0)
  - **Exponent** ***E*** weights value by power of two

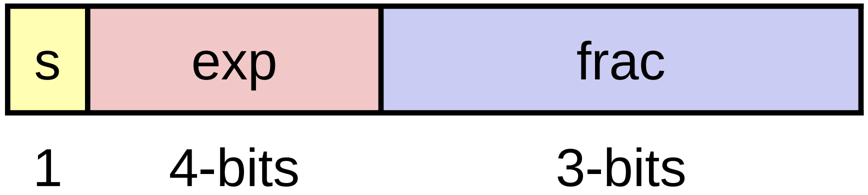
- **Encoding**
  - s is sign bit ***s***
  - **`exp`** field encodes ***E*** *(but is not equal to E)*
  - **`frac`** field encodes ***M*** *(but is not equal to M)*

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 **`(exp)`**
    - **`exp`** (not E) encoded as a 4-bit unsigned integer
    - Uses a *bias* to represent negative exponents
  - the last three bits are the fraction **`(frac)`**
    - encodes fractional part of a fractional binary number

Tiny Floating Point Example

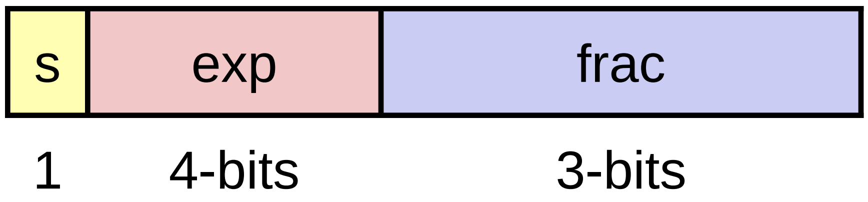


- **Fractional (Mantissa) Value**
  - Three bits- encoded left-to-right as $b_0b_1b_2$
  - Place value of b_0 is 1/2, of b_1 is 1/4, and b_2 is 1/8
  - Value usually includes an implied 1:

$$M = 1 + \frac{1}{2}b_0 + \frac{1}{4}b_1 + \frac{1}{8}b_2$$

  - If **`exp == 0000`** then the implied 1 is not used

Tiny Floating Point Example

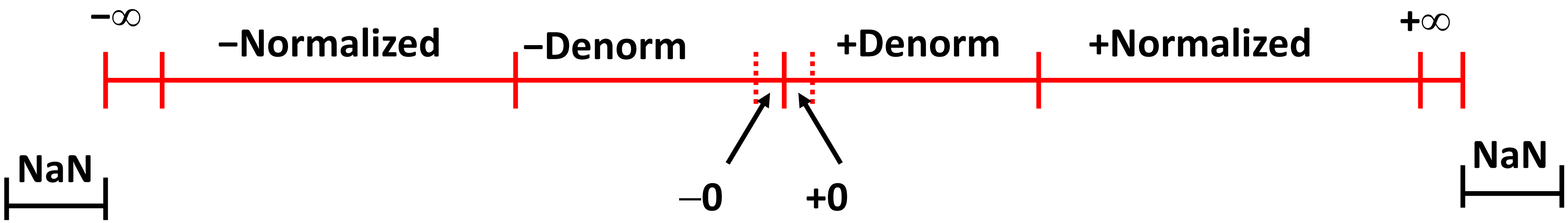


- **Exponent bias**
  - enable exponent to represent both positive and negative powers of 2
  - use half of range for positive and half for negative power
  - given k exponent bits, bias is then $2^{k-1} – 1$
- **Exponent Value**
  - is usually computed ***E = exp – bias***
  - if **`exp`** `== 0000` then ***E = 1 - bias***

Floating Point Encodings and Visualization

- **Five encodings:**
  - Two general forms: normalized, denormalized
  - Three special values: zero, infinity, NaN *(not a number)*

Name	Exponent (**exp**)	Fraction (**frac**)
zero	**exp** == 0000	**frac** == 000
denormalized	**exp** == 0000	**frac** != 000
normalized	0000 < **exp** < 1111	**frac** != 000
infinity	**exp** == 1111	**frac** == 000
NaN	**exp** == 1111	**frac** != 000



Dynamic Range (Positives)

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

norm: E = Exp – Bias

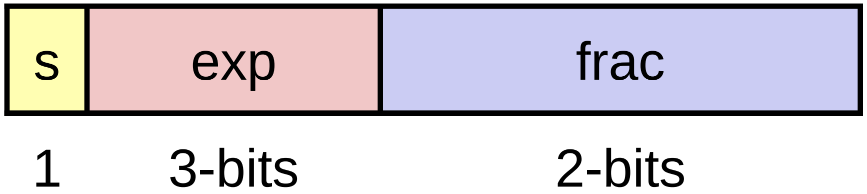
denorm: E = 1 – Bias

	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	*infinity*
	0	1111	xxx	n/a	NaN	*NaN* (not a number)

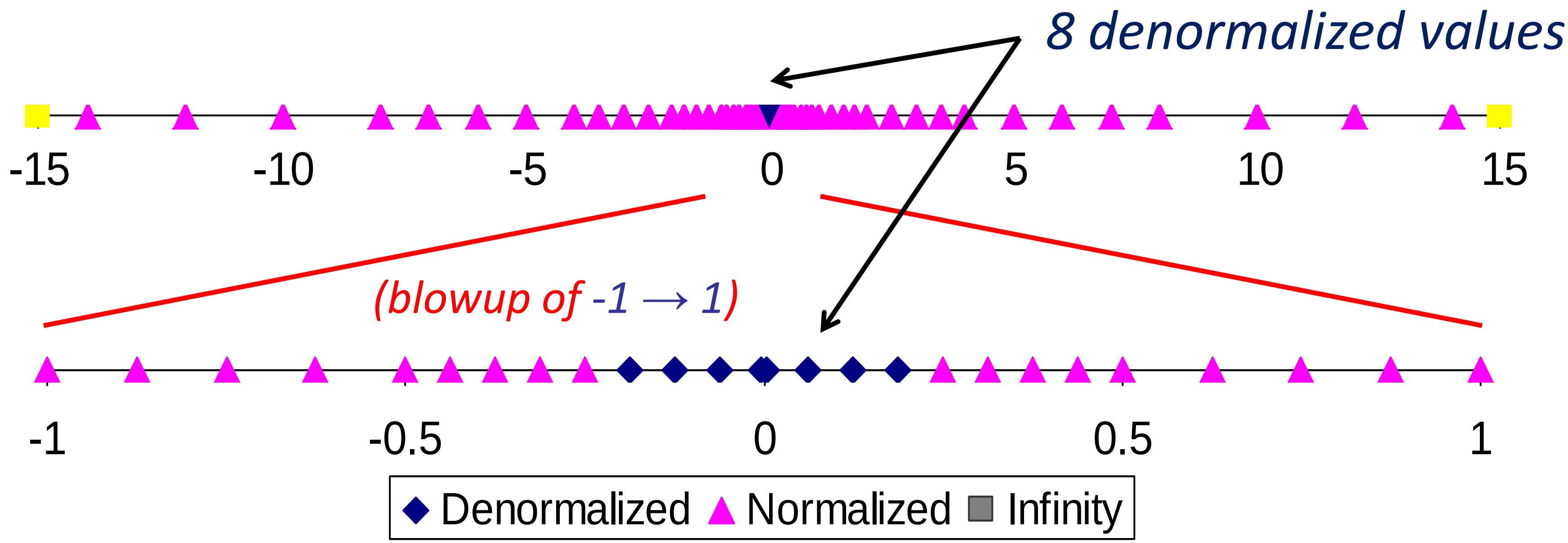
Distribution of Values

(reduced format from 8 bits to 6 bits for visualization)

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

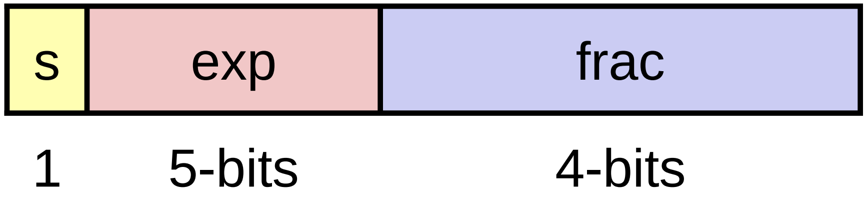


- **Notice how the distribution gets denser toward zero.**



Quick Check

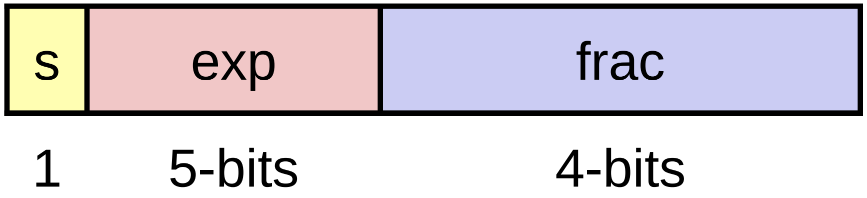
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **What is the exponent bias?**

Quick Check

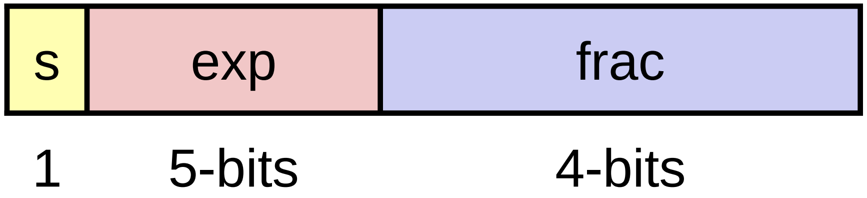
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **What is the exponent bias?** $2^{5-1} - 1 = 15$

Quick Check

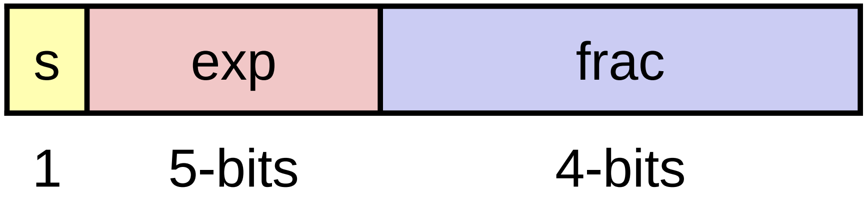
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **What is the exponent bias?** $2^{5-1} - 1 = 15$
- **How many denormalized numbers are there?**

Quick Check

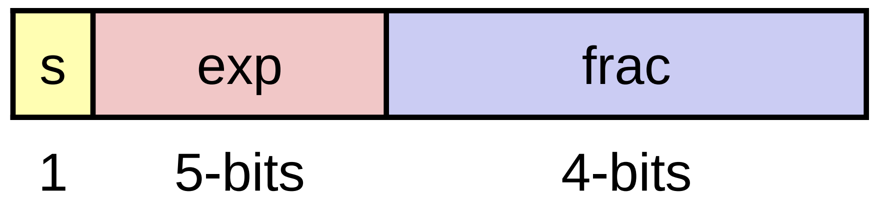
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **What is the exponent bias?** $2^{5-1} - 1 = 15$
- **How many denormalized numbers are there?**
  - Exponent = 00000, so 2^4 positive and 2^4 negative

Quick Check

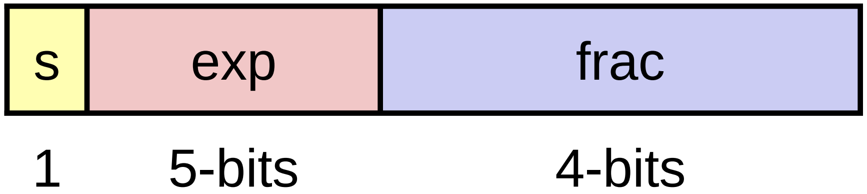
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **What is the exponent bias?** $2^{5-1} - 1 = 15$
- **How many denormalized numbers are there?**
  - Exponent = 00000, so 2^4 positive and 2^4 negative
- **What is the bit pattern of the maximum value number?**
- **What is the bit pattern of the number closest to zero?**

Quick Check

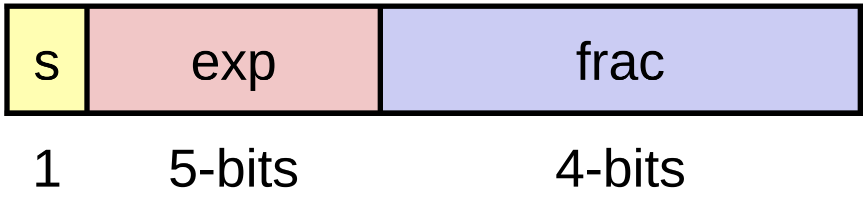
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **What is the exponent bias?** $2^{5-1} - 1 = 15$
- **How many denormalized numbers are there?**
  - Exponent = 00000, so 2^4 positive and 2^4 negative
- **What is the bit pattern of the maximum value number?**
  - Sign = 0, Exponent = 11110, frac=1111, so 0111101111
- **What is the bit pattern of the number closest to zero?**
  - Sign = ?, Exponent = 00000, frac=0001, so ?000000001

Quick Check

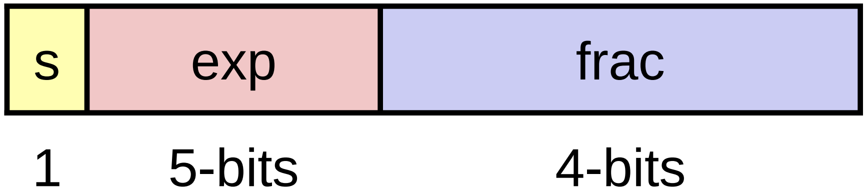
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **What is the bit pattern of the smallest positive normal number?**

Quick Check

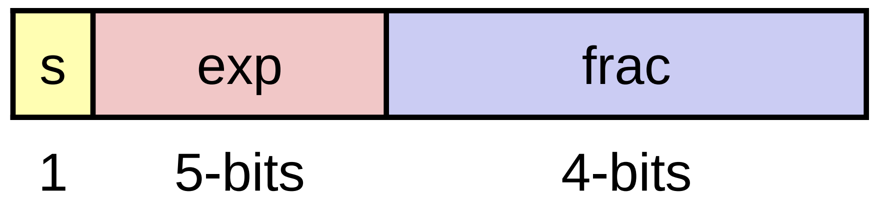
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **What is the bit pattern of the smallest positive normal number?**
  - Sign = 0, exp = 00001, frac = 0000; so 0000010000

Quick Check

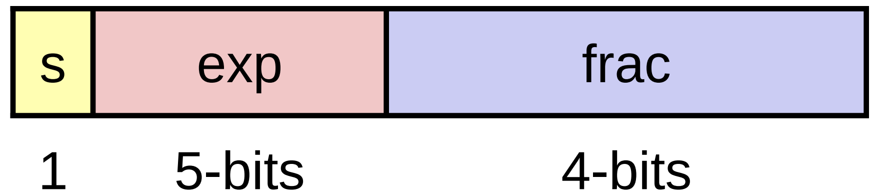
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **What is the bit pattern of the smallest positive normal number?**
  - Sign = 0, exp = 00001, frac = 0000; so 0000010000
- **What is the value of the smallest positive normal number?**

Quick Check

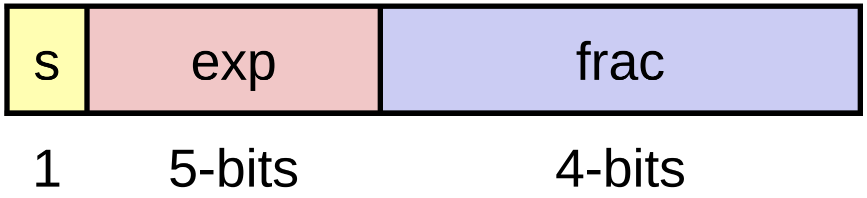
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **What is the bit pattern of the smallest positive normal number?**
  - Sign = 0, exp = 00001, frac = 0000; so 0000010000
- **What is the value of the smallest positive normal number?**
  - Value = $(-1)^s \times M \times 2^E$
  - S = 0
  - Exponent bias = 15, so E = 1 - 15 = -14
  - M = $1 + 0 \times \frac{1}{2} + 0 \times \frac{1}{4} + 0 \times \frac{1}{8} + 0 \times \frac{1}{16}$ = 1
  - Value = $(-1)^0 \times 1 \times 2^{-14}$ = ${}^1/_{2^{14}}$ = 0.00006103515

Quick Check

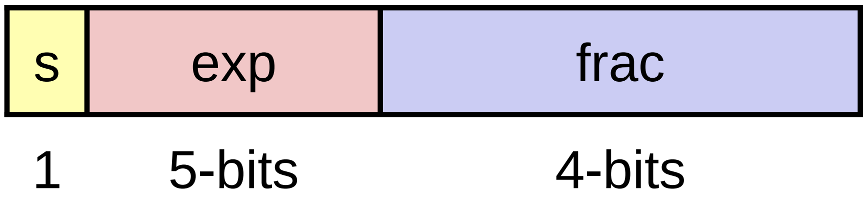
- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **Given a 32-bit floating point number, and a 32-bit integer, which can represent more discrete values?**

Quick Check

- **10-bit IEEE-like format**
  - e = 5 exponent bits
  - f = 4 fraction bits



- **Given a 32-bit floating point number, and a 32-bit integer, which can represent more discrete values?**
  - Both can represent 2^{32} values, but some bit patterns duplicate values, e.g. +0/-0, +∞/-∞, and many NaNs (exponent = 11…1, frac != 00…0)

Today: Floating Point

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

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** (mantissa) ***M*** normally a fractional value in range [1.0, 2.0)
  - **Exponent** ***E*** weights value by power of two

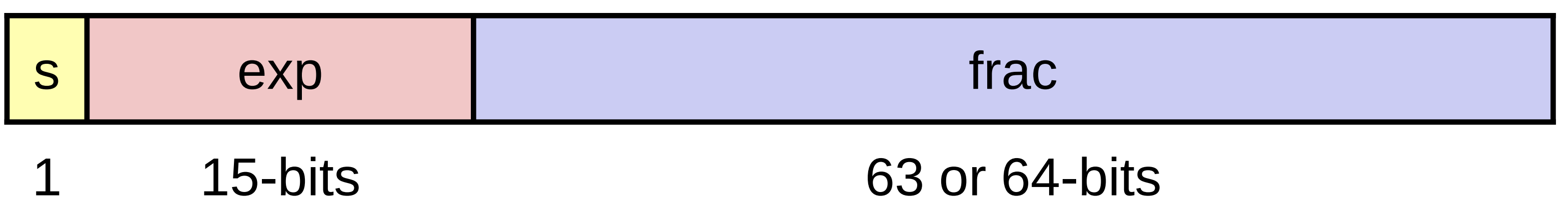
- **Encoding**
  - `s` is sign bit ***s***
  - **`exp`** field encodes ***E*** *(but is not equal to E)*
  - **`frac`** field encodes ***M*** *(but is not equal to M)*

Standard Types

- **Single precision: 32 bits (c type: `float`)**

- **Double precision: 64 bits (c type: `double`)**

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



Normalized Values

- **Condition:** *exp* **≠ 000…0 and** *exp* **≠ 111…1**

- **Exponent coded as *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*** **=** $\underline{1}.xxx\ldots x_2$
  - xxx…x: bits of *frac*

- **Decimal value of normalized FP representations:**
  - Single-precision: $Value_{10} = (-1)^s \times 1.frac \times 2^{exp-127}$
  - Double-precision: $Value_{10} = (-1)^s \times 1.frac \times 2^{exp-1023}$

Normalized Encoding Example

- **Value:** `float F = 15213.5;`
  - $15213.5_{10} = 11101101101101.1_2$

  $= 1.1101101101101_2 \times 2^{13}$

shift binary point by K bits so that only one leading 1 bit remains on the left side of the binary point (here, shifted right by 13 bits, so K = 13), then multiply by 2^K (here, 2^{13})

- **Significand**

  $M = 1.1101101101101_2$

  frac $= 11011011011010000000000_2$

- **Exponent** ***(E = Exp – Bias)***

  $E = 13$

  $Bias = 127$

  $Exp = E + Bias = 140 = 10001100_2$

s	exp	frac
0	10001100	11011011011010000000000

Denormalized Values

- **Condition:** exp = 000…0

- **Exponent value:** ***E* = 1 – *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 very close to 0.0
    - Lose precision as get smaller
    - Equispaced

Special Values

- **Special 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

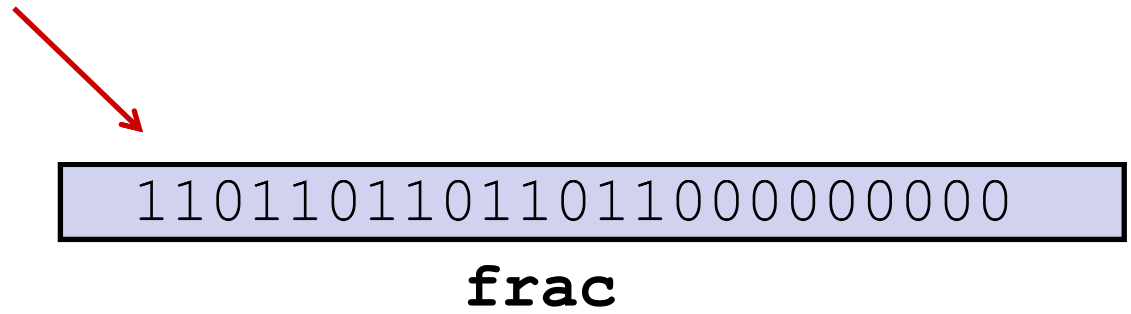
Interesting Numbers {single,double}

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}			

Precision

- **How *accurate* are floating point numbers? Consider again:**

$15213_{10} = 11101101101101.1_2$



- **Size of `frac` limits how many decimal places we can represent**
- **E.g. single precision has 23 explicit bits + 1 implicit bit**

$123456789_{10} = 111010110111100110100010101_2$

M = $1.\underline{11010110111100110100010101}_2 \times 2^{26}$

`frac`= $\underline{11010110111100110100011}_2$ **(round up)**

Loss of Precision Example

- **Value:** `float F = 123456789;`
  - $123456789_{10} = 111010110111100110100010101_2$

$= 1.11010110111100110100010101_2 \times 2^{26}$

- **Significand**

M = 1.11010110111100110100010101$_2$

frac= 11010110111100110100011$_2$

- **Reconstructed Value**

(-1)^S * M * 2^E

M = 1 + 0.5 + 0.25 + … = 1.83964955806732177734375

V = 1.83964955806732177734375 * 2^26

V = 123456792

Today: Floating Point

- Background: Fractional binary numbers
- Example and properties
- IEEE floating point standard: Definition
- **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 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

Today: Floating Point

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

Floating Point Operations: Basic Idea

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

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

- **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 = **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

Scientific Notation Multiplication

- $(2.5 \times 10^3) \times (3.0 \times 10^2)$ **= ?**
- **Compute result by pieces:**
  - Sign : *$sign_{left}$ * $sign_{right}$ = 1*1 = 1*
  - Significand : M_{left} * M_{right} = 2.5 * 3.0 = 7.5
  - Exponent : E_{left} + E_{right} = 3 + 2 = 5

Result: 7.5×10^5

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

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

Scientific Notation Addition

- $(2.5 \times 10^3) + (3.0 \times 10^2)$ **= ?**
- **Assume E_{left} is larger that E_{right}**
- **Align by decimal point:**
  - Significand :

```
  2.5
+  .3
-----
  2.8
```

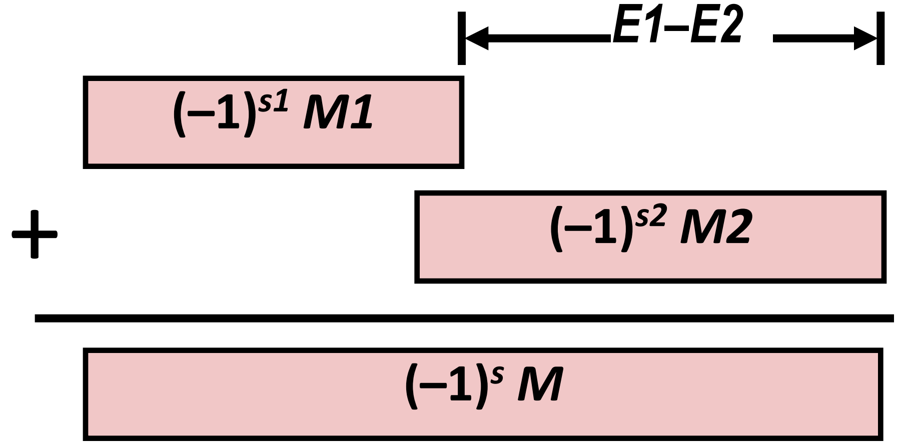
  - Exponent : $E = E_{left} = 3$

Result: 2.8×10^3

Floating Point Addition

- $(-1)^{s1}\,M1\;\,2^{E1}\;\; + \;\;(-1)^{s2}\,M2\;\,2^{E2}$
  - Assume $E1 > E2$

Get binary points lined up



- **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**
  - Commutative? ***Yes***
  - (a + b) = (b + a)
  - Associative? ***No***
    - Overflow and inexactness of rounding
    - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`

Mathematical Properties of FP Mult

- **Compare to Commutative Ring**
  - Multiplication Commutative? ***Yes***
  - Ex: `(1e20*1e-20)=(1e-20*1e20)`
  - Multiplication is Associative? ***No***
    - Possibility of overflow, inexactness of rounding
    - Ex: `(1e20*1e20)*1e-20= inf, 1e20*(1e20*1e-20)= 1e20`
  - Multiplication distributes over addition? ***No***
    - Possibility of overflow, inexactness of rounding
    - `1e20*(1e20-1e20)=0.0, 1e20*1e20 – 1e20*1e20 =NaN`

Summary

- **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