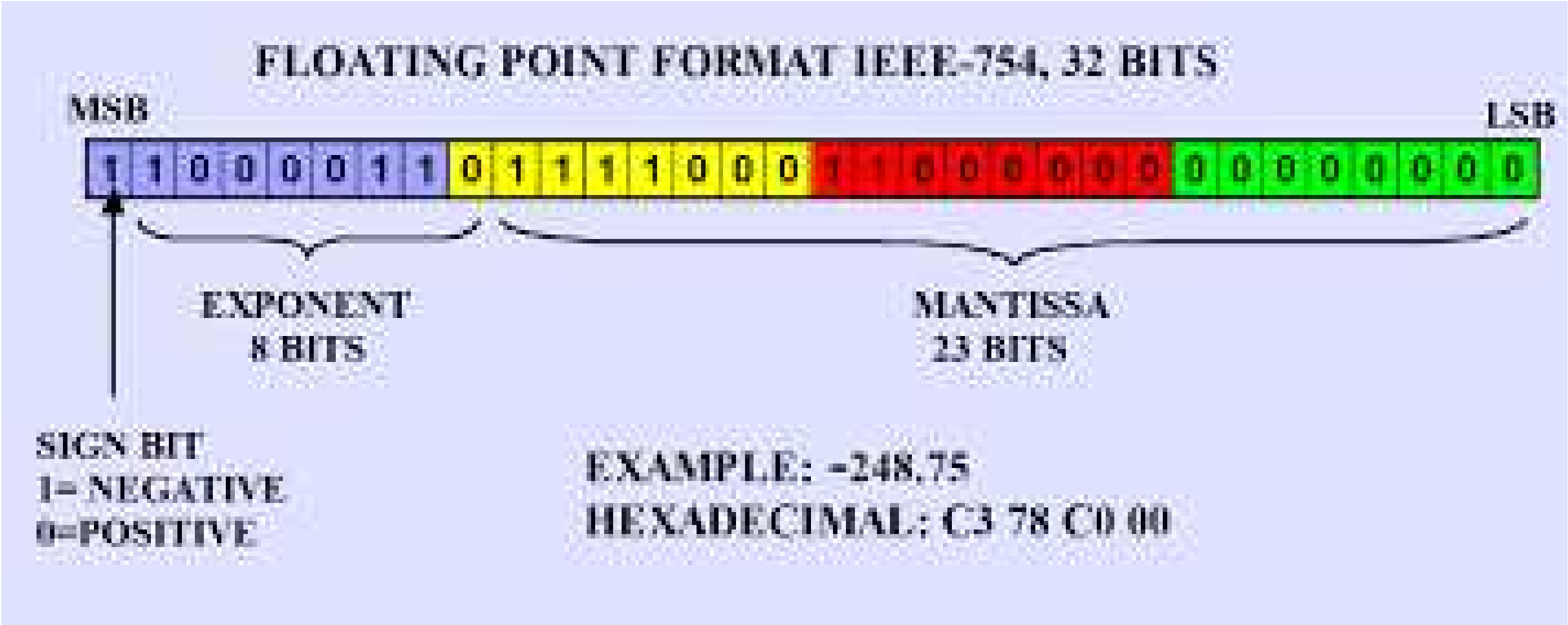
CS429: Computer Organization and Architecture

Floating Point

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: February 5, 2020 at 15:08



- IEEE Floating Point Standard
- Rounding
- Floating point operations
- Mathematical properties

Floating Point Puzzles

```
int x = ...;
float f = ...;
double d = ...;
```

For each of the following, either:

- argue that it is true for all argument values, or
- explain why it is not true.

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
```

IEEE Floating Point Standard

IEEE Standard 754

- Established in 1985 as a uniform standard for floating point arithmetic
- It is supported by all major CPUs.
- Before 1985 there were many idiosyncratic formats.

Driven by Numerical Concerns

- Nice standards for rounding, overflow, underflow
- Hard to make go fast: numerical analysts predominated over hardware types in defining the standard
- Now all (add, subtract, multiply) operations are fast except divide.

Fractional Binary Numbers: Examples

The binary number $b_i b_{i-1} b_2 b_1 \ldots b_0 . b_{-1} b_{-2} b_{-3} \ldots b_{-j}$ represents a particular (positive) sum. Each digit is multiplied by a power of two according to the following chart:

Bit:	b_i	b_{i-1}	$\ldots$	b_2	b_1	b_0	.	b_{-1}	b_{-2}	b_{-3}	$\ldots$	b_{-j}
Weight:	2^i	2^{i-1}	$\ldots$	4	2	1	.	1/2	1/4	1/8	$\ldots$	2^{-j}

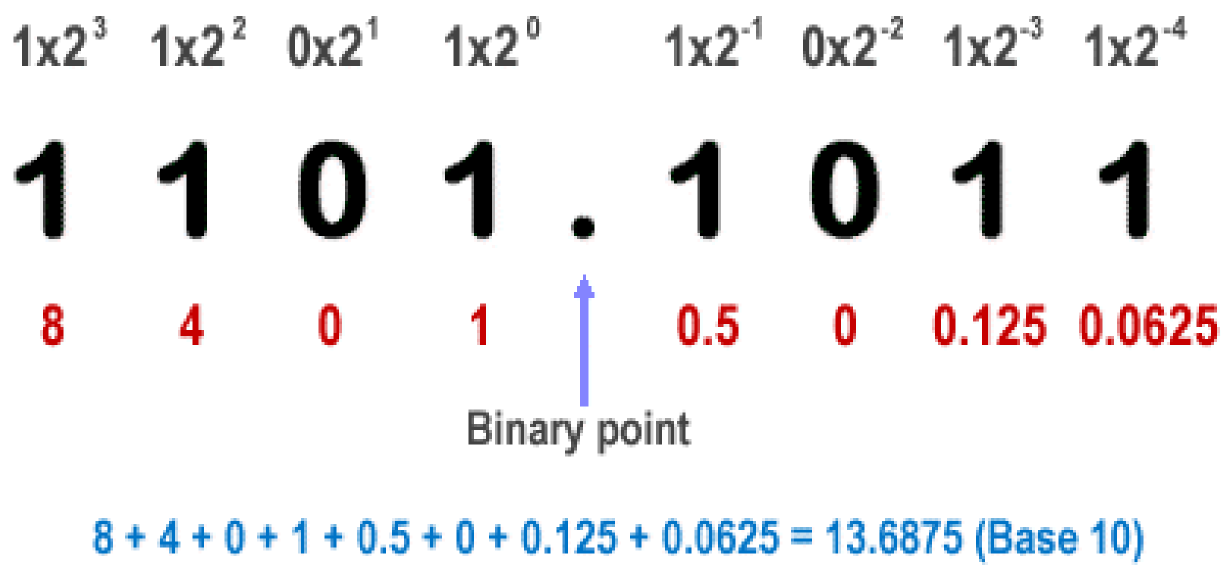
Representation:

- Bits to the right of the *binary point* represent fractional powers of 2.
- This represents the rational number:

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

The sign is treated separately.

Fractional Binary Numbers: Example



Fractional Binary Numbers: Examples

Value	**Representation**
$5 + 3/4$	101.11_2
$2 + 7/8$	10.111_2
$63/64$	0.111111_2

Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of the form $0.11111\ldots_2$ are just below 1.0
  - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i \rightarrow 1.0$
  - We use the notation $1.0 - \epsilon$.

Representable Numbers

Limitation

- You can only represent numbers of the form $y + x/2^i$.
- Other fractions (rationals) have repeating bit representations
- Irrationals have infinite, non-repeating representations

Value	**Representation**
1/3	$0.0101010101[01]_2$
1/5	$0.001100110011[0011]_2$
1/10	$0.0001100110011[0011]_2$

Aside: Converting Decimal Fractions to Binary

If you want to convert a decimal fraction to binary, it's easy if you use a simple iterative procedure.

1. Start with the decimal fraction (> 1) and multiply by 2.
2. Stop if the result is 0 (terminated binary) or a result you've seen before (repeating binary).
3. Record the whole number part of the result.
4. Repeat from step 1 with the fractional part of the result.

$$
\begin{aligned}
0.375 * 2 &= 0.75 \\
0.75 * 2 &= 1.5 \\
0.5 * 2 &= 1.0 \\
0.0 &
\end{aligned}
$$

The result (following the binary point) is the series of whole numbers components of the answers read from the top, i.e., 0.011.

Aside: Converting Decimal Fractions to Binary (2)

Let's try another one, 0.1 or $1/10$

$$
\begin{aligned}
0.1 * 2 &= 0.2 \\
0.2 * 2 &= 0.4 \\
0.4 * 2 &= 0.8 \\
0.8 * 2 &= 1.6 \\
0.6 * 2 &= 1.2 \\
0.2 * 2 &= 0.4
\end{aligned}
$$

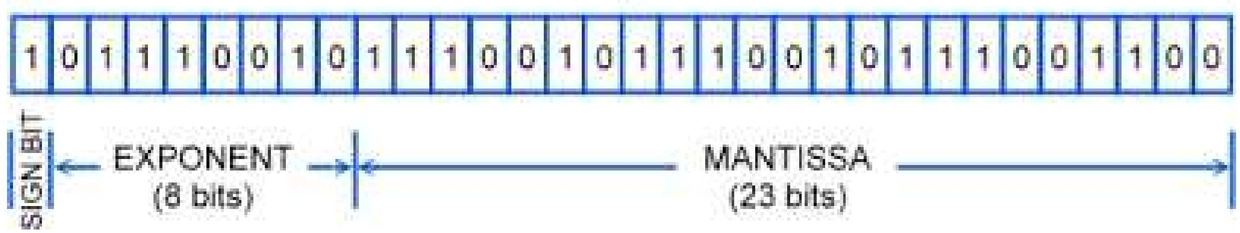
We could continue, but we see that it's going to repeat forever (since 0.2 repeats our multiplicand from the second line). Reading the integer parts from the top gives 0[0011], since we'll repeat the last 4 bits forever.

Floating Point Representation

Numerical Form

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

- Sign bit s determines whether number is negative or positive.
- Significand M is normally a fractional value in the range $[1.0 \ldots 2.0)$
- Exponent E weights value by power of two.



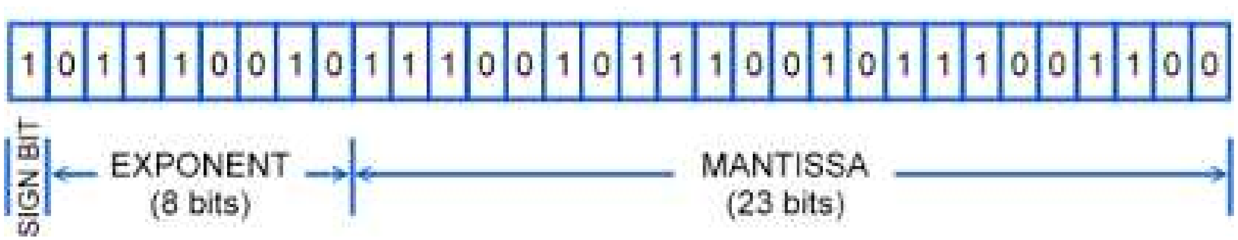
Floats (32-bit floating point numbers)

Encoding

s	exp	frac

- The most significant bit is the sign bit.
- The `exp` field encodes E.
- The `frac` field encodes M.

Float format:



Floating Point Precisions

Encoding

s	exp	frac

- The most significant bit is the sign bit.
- The `exp` field encodes E.
- The `frac` field encodes M.

Sizes

- Single precision: 8 exp bits, 23 frac bits, for 32 bits total
- Double precision: 11 exp bits, 52 frac bits, for 64 bits total
- Extended precision: 15 exp bits, 63 frac bits (only Intel-compatible machines)

Normalized Numeric Values

Condition: exp $\neq 000\ldots0$ and exp $\neq 111\ldots1$

Exponent is coded as a biased value

$E = Exp - Bias$

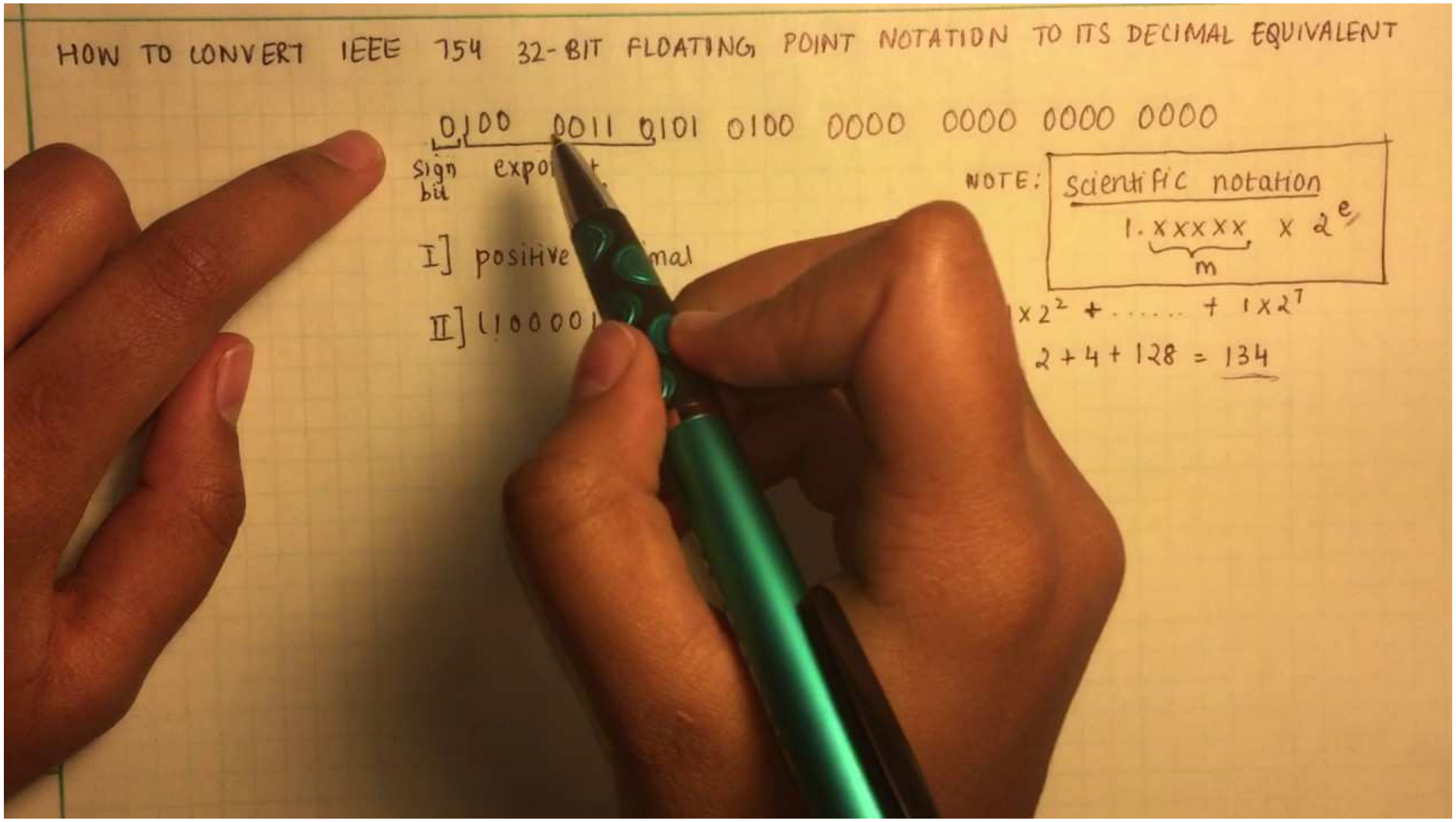
- Exp: unsigned value denoted by `exp`.
- $Bias$: Bias value
  - In general: $Bias = 2^{e-1} - 1$, where e is the number of exponent bits
  - Single precision: 127 (Exp: $1\ldots254$, $E: -126\ldots127$)
  - Double precision: 1023 (Exp: $1\ldots2046$, $E: -1022\ldots1023$)

Significand coded with implied leading 1

$M = 1.xxx\ldots x_2$

- $xxx\ldots x$: bits of `frac`
- Minimum when $000\ldots0$ ($M = 1.0$)
- Maximum when $111\ldots1$ ($M = 2.0 - \epsilon$)
- We get the extra leading bit "for free."

Converting Between Float and Decimal



Normalized Encoding Example

Value:

`float F = 15213.0;`

$$15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$$

Significand

M $= 1.1101101101101_2$

`frac` $= 11011011011010000000000$

Exponent

E $= 13$

Bias $= 127$

Exp $= 140 = 10001100$

Normalized Encoding Example

Floating Point Representation

Hex: 466DB400

Binary: 0100 0110 0110 1101 1011 0100 0000 0000

140: 100 0110 0

15213: 1110 1101 1011 01

Normalized Example

Given the bit string 0x40500000, what floating point number does it represent?

Normalized Example

Given the bit string 0x40500000, what floating point number does it represent?

Writing this as a bit string gives us:

$$0\ 10000000\ 10100000000000000000000$$

We see that this is a positive, normalized number.

$$\text{exp} = 128 - 127 = 1$$

So, this number is:

$$1.101_2 \times 2^1 = 11.01_2 = 3.25_{10}$$

Denormalized Values

Condition: `exp = 000...0`

Value

- Exponent values: E = -Bias + 1 Why this value?
  - Floats: -126; Doubles: -1022
- Significand value: $\mathrm{M} = 0.xxx\ldots x_2$, where $xxx\ldots x$ are the bits of frac.

Cases

- $\exp = 000\ldots0$ and $\mathrm{frac} = 000\ldots0$
  - represents values of 0
  - notice that we have distinct +0 and -0
- $\exp = 000\ldots0$ and $\mathrm{frac} \neq 000\ldots0$
  - These are numbers very close to 0.0
  - Lose precision as they get smaller
  - Experience "gradual underflow"

Denormalized Example

Given the bit string 0x80600000, what floating point number does it represent?

Denormalized Example

Given the bit string 0x80600000, what floating point number does it represent?

Writing this as a bit string gives us:

1 00000000 11000000000000000000000

We see that this is a negative, denormalized number with value:

$$-0.11_2 \times 2^{-126} = -1.1_2 \times 2^{-127}$$

Why That Exponent

The exponent (*it's not a bias*) for denormalized floats is -126. Why that number?

The smallest positive *normalized* float is $1.0_2 \times 2^{-126}$. Where did I get that number? All positive normalized floats are greater or equal.

The largest positive *denormalized* float is $0.11111111111111111111111_2 \times 2^{-126}$. Why? All positive denorms are between this number and 0.

Note that the smallest norm and the largest denorm are incredibly close together. How close? Thus, the normalized range flows naturally into the denormalized range *because of this choice of exponent for denorms*.

Condition: `exp = 111...1`

Cases

- exp $= 111 \ldots 1$ and frac $= 000 \ldots 0$
  - Represents value of infinity (∞)
  - Result returned for operations that overflow
  - Sign indicates positive or negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- exp $= 111 \ldots 1$ and frac $\neq 000 \ldots 0$
  - Not-a-Number (NaN)
  - Represents the case when no numeric value can be determined
  - E.g., sqrt(-1), $\infty - \infty$

How many 32-bit NaN's are there?

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 with a bias of 7.
- The last three bits are the frac.

This has the general form of the IEEE Format

- Has both normalized and denormalized values.
- Has representations of 0, NaN, infinity.

7	6 3	2 0
s	exp	frac

Values Related to the Exponent

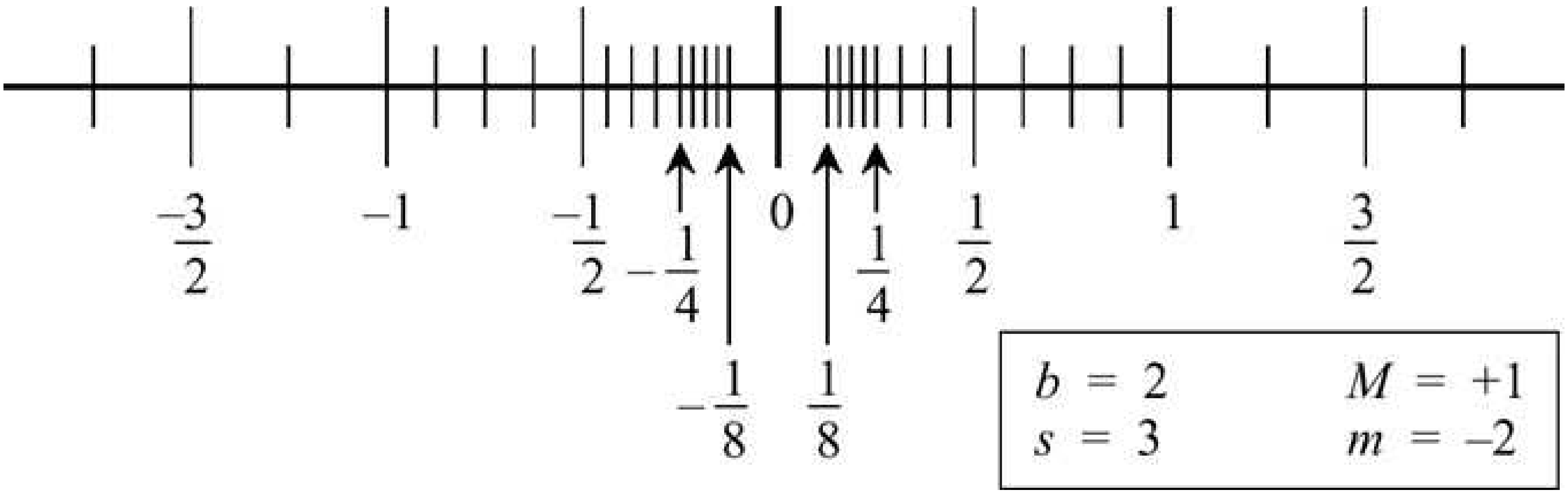
Exp	exp	E	2^E	comment
0	0000	-6	1/64	(denorms)
1	0001	-6	1/64	
2	0010	-5	1/32	
3	0011	-4	1/16	
4	0100	-3	1/8	
5	0101	-2	1/4	
6	0110	-1	1/2	
7	0111	0	1	
8	1000	+1	2	
9	1001	+2	4	
10	1010	+3	8	
11	1011	+4	16	
12	1100	+5	32	
13	1101	+6	64	
14	1110	+7	128	
15	1111	n/a		(inf, NaN)

Dynamic Range

	s	exp	frac	E	Value	
	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 \times 1/64 = 1/512$	closest to zero
Denormalized numbers	0	0000	010	-6	$2/8 \times 1/64 = 2/512$	
				...		
	0	0000	110	-6	$6/8 \times 1/64 = 6/512$	
	0	0000	111	-6	$7/8 \times 1/64 = 7/512$	largest denorm
	0	0001	000	-6	$8/8 \times 1/64 = 8/512$	smallest norm
	0	0001	001	-6	$9/8 \times 1/64 = 9/512$	
				...		
	0	0110	110	-1	$14/8 \times 1/2 = 14/16$	
Normalized numbers	0	0110	111	-1	$15/8 \times 1/2 = 15/16$	closest to 1 below
	0	0111	000	0	$8/8 \times 1 = 1$	
	0	0111	001	0	$9/8 \times 1 = 9/8$	closest to 1 above
	0	0111	010	0	$10/8 \times 1 = 10/8$	
				...		
	0	1110	110	7	$14/8 \times 128 = 224$	
	0	1110	111	7	$15/8 \times 128 = 240$	largest norm
	0	1111	000	n/a	∞	

Simple Float System

Notice that the floating point numbers are not distributed evenly on the number line.



Suppose M is the largest possible exponent, m is the smallest, $\frac{1}{8}$ is the smallest positive number representable, and $\frac{7}{4}$ the largest positive number representable. What is the format?

Interesting FP Numbers

Description	exp	frac	Numeric value
Zero	$00\ldots00$	$00\ldots00$	0.0
Smallest Pos. Denorm	$00\ldots00$	$00\ldots01$	$2^{\{-23,-52\}} \times 2^{\{-126,-1022\}}$

- Single $\approx 1.4 \times 10^{-45}$
- Double $\approx 4.9 \times 10^{-324}$

Largest Denorm.	$00\ldots00$	$11\ldots11$	$(1.0-\epsilon) \times 2^{\{-126,-1022\}}$

- Single $\approx 1.18 \times 10^{-38}$
- Double $\approx 2.2 \times 10^{-308}$

Smallest Pos. Norm.	$00\ldots01$	$00\ldots00$	$1.0 \times 2^{\{-126,-1022\}}$

- Just larger than the largest denomalized.

One	$01\ldots11$	$00\ldots00$	1.0
Largest Norm.	$11\ldots10$	$11\ldots11$	$(2.0-\epsilon) \times 2^{\{127,1023\}}$

- Single $\approx 3.4 \times 10^{38}$
- Double $\approx 1.8 \times 10^{308}$

Special Properties of Encoding

FP Zero is the Same as Integer Zero: All bits are 0.

Can (Almost) Use Unsigned Integer Comparison

- Must first compare sign bits.
- Must consider $-0 = 0$.
- NaNs are problematic:
  - Will be greater than any other values.
  - What should the comparison yield?
- Otherwise, it's OK.
  - Denorm vs. normalized works.
  - Normalized vs. infinity works.

Floating Point Operations

Conceptual View

- First compute the exact result.
- Make it fit into the desired precision.
  - Possibly overflows if exponent is too large.
  - Possibly round to fit into frac.

Rounding Modes (illustrated with $ rounding)

	$1.40	**$1.60**	**$1.50**	**$2.50**	**-$1.50**
Toward Zero	$1	$1	$1	$2	-$1
Round down ($-\infty$)	$1	$1	$1	$2	-$2
Round up ($+\infty$)	$2	$2	$2	$3	-$1
Nearest even (default)	$1	$2	$2	$2	-$2

1. Round down: rounded result is close to but no greater than true result.
2. Round up: rounded result is close to but no less than true result.

Closer Look at Round to Even

Default Rounding Mode

- Hard to get any other kind without dropping into assembly.
- All others are statistically biased; the sum of a large set of values will consistently be under- or over-estimated.

Applying to Other Decimal Places / Bit Positions

When exactly halfway between two possible values, round so that the least significant digit is even.

E.g., round to the 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 the right of rounding position $= 10[0]_2$.

Examples

E.g., Round to nearest 1/4 (2 bits to right of binary point).

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00	$(< 1/2$: down)	2
2 3/16	10.00110_2	10.01	$(> 1/2$: up)	2 1/4
2 7/8	10.11100_2	11.00	(1/2: up)	3
2 5/8	10.10100_2	10.10	(1/2: down)	2 1/2

Round to Even

When rounding to even, *first check that the value to round is actually exactly halfway between two values.* Then, consider the two possible choices and choose the one with a 0 in the final position.

Example: round to nearest 1/4 using round to even:

1.10	1 1/2
1.1010000	1 5/8
1.11	1 3/4

1.11	1 3/4
1.1110000	1 7/8
10.00	2.0

FP Multiplication

Operands: $(-1)^{S_1} \times M_1 \times 2^{E_1}, (-1)^{S_2} \times M_2 \times 2^{E_2}$

Exact Result: $(-1)^S \times M \times 2^E$

- Sign S: S_1 xor S_2
- Significand M: $M_1 \times M_2$
- Exponent E: $E_1 + E_2$

Fixing

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

Implementation

Biggest chore is multiplying significands.

Multiplication Examples

Decimal Example

$$
\begin{aligned}
& (-3.4 \times 10^2)(5.2 \times 10^4) & \\
= & -(3.4 \times 5.2)(10^2 \times 10^4) & \\
= & -17.68 \times 10^6 & \\
= & -1.768 \times 10^7 & \text{adjust exponent} \\
= & -1.77 \times 10^7 & \text{round}
\end{aligned}
$$

Binary Example

$$
\begin{aligned}
& (-1.01 \times 2^2)(1.1 \times 2^4) & \\
= & -(1.01 \times 1.1)(2^2 \times 2^4) & \\
= & -1.111 \times 2^6 & \\
= & -10.0 \times 2^6 & \text{round to even} \\
= & -1.0 \times 2^7 & \text{adjust exponent}
\end{aligned}
$$

Multiplication Warning

Binary Example

$$
\begin{aligned}
& (-1.01 \times 2^2)(1.1 \times 2^4) \\
= & -(1.01 \times 1.1)(2^2 \times 2^4) \\
= & -1.111 \times 2^6 \\
= & -10.0 \times 2^6 && \text{round to even} \\
= & -1.0 \times 2^7 && \text{adjust exponent}
\end{aligned}
$$

Be careful if you try to do this in the floating point format, rather than in scientific notation. Since the exponents are biased in FP format, adding them would give you:

```
(2 + bias) + (4 + bias) = 6 + 2*bias
```

To adjust you have to subtract the bias.

FP Addition

Operands: $(-1)^{S_1} \times M_1 \times 2^{E_1}, (-1)^{S_2} \times M_2 \times 2^{E_2}$
Assume $E_1 > E_2$

Exact Result: $(-1)^{S} \times M \times 2^{E}$

- Sign S, Significand M; result of signed align and add.
- Exponent E: E_1

Fixing

- If $M \geq 2$, shift M right, increment E
- If $M < 1$, shift M left k positions, decrement E by k
- if E is out of range, overflow
- Round M to fit frac precision

If you try to do this in the FP form, recall that both exponents are biased.

Addition Examples

Decimal Example

$$
\begin{aligned}
& (-3.4 \times 10^2) + (5.2 \times 10^4) & \\
= & (-3.4 \times 10^2) + (520.0 \times 10^2) & \text{align exponents} \\
= & (-3.4 + 520.0) \times 10^2 & \\
= & 516.6 \times 10^2 & \\
= & 5.166 \times 10^4 & \text{fix exponent} \\
= & 5.17 \times 10^4 & \text{round}
\end{aligned}
$$

Binary Example

$$
\begin{aligned}
& (-1.01 \times 2^2) + (1.1 \times 2^4) & \\
= & (-1.01 \times 2^2) + (110.0 \times 2^2) & \text{align exponents} \\
= & (-1.01 + 110.0) \times 2^2 & \\
= & 100.11 \times 2^2 & \\
= & 1.0011 \times 2^4 & \text{fix exponent} \\
= & 1.01 \times 2^4 & \text{round}
\end{aligned}
$$

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, because of overflow and inexactness of rounding.
- 0 is additive identity? Yes.
- Every element has additive inverse? Almost, except for infinities and NaNs.

Monotonicity

- $a \geq b \Longrightarrow a + c \geq b + c$? Almost, except for infinities and NaNs.

Mathematical Properties of FP Mult

Compare to those of Commutative Ring

- Closed under multiplication? Yes, but may generate infinity or NaN.
- Multiplication Commutative? Yes.
- Multiplication is Associative? No, because of possible overflow and inexactness of rounding.
- 1 is multiplicative identity? Yes.
- Multiplication distributes over addition? No, because of possible overflow and inexactness of rounding.

Monotonicity

- $a \geq b \ \& \ c \geq 0 \Longrightarrow a \times c \geq b \times c$? Almost, except for infinities and NaNs.

Floating Point in C

C guarantees two levels

- float: single precision
- double: double precision

Conversions

- Casting among int, float, and double changes numeric values
- Double or float to int:
  - truncates fractional part
  - like rounding toward zero
  - not defined when out of range: generally saturates to TMin or TMax
- int to double: exact conversion as long as int has $\leq$ 53-bit word size
- int to float: will round according to rounding mode.

Answers to FP Puzzles

```c
int x = ...;
float f = ...;
double d = ...;
```

Assume neither d nor f is NaN.

`x == (int)(float) x`		No: 24 bit significand
`x == (int)(double) x`		Yes: 53 bit significand
`f == (float)(double) f`		Yes: increases precision
`d == (float) d`		No: loses precision
`f == -(-f)`		Yes: just change sign bit
`2/3 == 2/3.0`		No: $2/3 == 0$
`d < 0.0`	$\rightarrow$ `((d*2) < 0.0)`	Yes
`d > f`	$\rightarrow$ `-f > -d`	Yes
`d*d >= 0.0`		Yes
`(d+f)-d == f`		No: not associative

Ariane 5

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing $7 billion.

The destroyed rocket and its cargo were valued at $500 million.

Ariane 5

The cause of the failure was a software error in the inertial reference system.

Specifically a 64-bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16-bit signed integer.

The number was larger than 32,767, the largest integer storeable in a 16-bit signed integer, and thus the conversion failed.

Summary

IEEE Floating Point has Clear Mathematical Properties

- Represents numbers of the form $\pm M \times 2^E$.
- Can reason about operations independent of implementation: as if computed with perfect precision and then rounded.
- Not the same as real arithmetic.
  - Violates associativity and distributivity.
  - Makes life difficult for compilers and serious numerical application programmers.