The Hardware/Software Interface

CSE351 Spring2013

Floating-Point Numbers

Roadmap

- Data & addressing
- **Integers & floats**
- Machine code & C
- x86 assembly programming
- Procedures & stacks
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

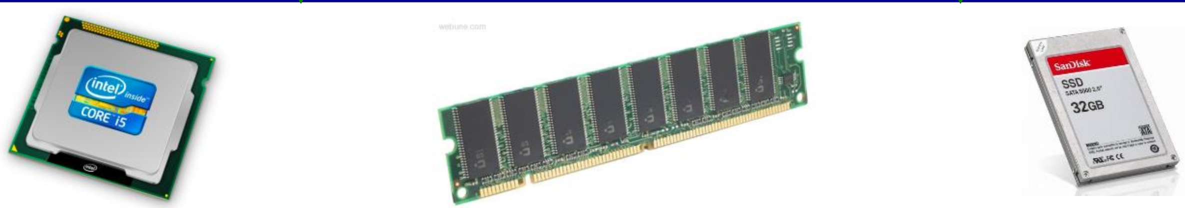
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8

Mac

Computer system:



Today's Topics

- Background: fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

Fractional Binary Numbers

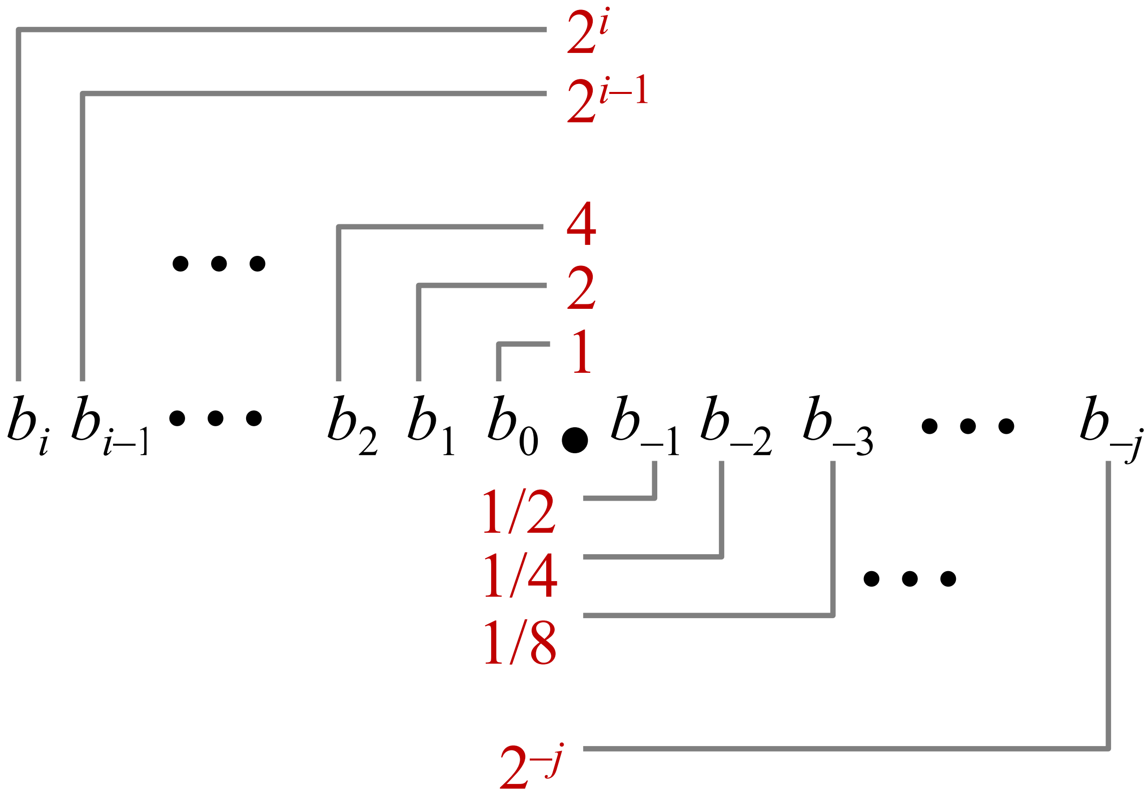
- What is 1011.101_2?

Fractional Binary Numbers

- **What is 1011.101_2?**

- **How do we interpret fractional *decimal* numbers?**
  - e.g. 107.95_{10}
  - Can we interpret fractional binary numbers in an analogous way?

Fractional Binary Numbers



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

Fractional Binary Numbers: Examples

- **Value** **Representation**
  - 5 and 3/4 `101.11`$_2$
  - 2 and 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 **`0.111111`**…$_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Shorthand notation for all **`1`** bits to the right of binary point: 1.0 – ⍰

Representable Values

- **Limitations of fractional binary numbers:**
  - Can only exactly represent numbers that can be written as $x * 2^y$
  - 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]`…$_2$

Fixed Point Representation

- **We might try representing fractional binary numbers by picking a fixed place for an implied binary point**
  - "fixed point binary numbers"
- **Let's do that, using 8-bit fixed point numbers as an example**
  - #1: the binary point is between bits 2 and 3
    $b_7\ b_6\ b_5\ b_4\ b_3\ [.]\ b_2\ b_1\ b_0$
  - #2: the binary point is between bits 4 and 5
    $b_7\ b_6\ b_5\ [.]\ b_4\ b_3\ b_2\ b_1\ b_0$
- **The position of the binary point affects the *range* and *precision* of the representation**
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers

Fixed Point Pros and Cons

- **Pros**
  - It's simple. The same hardware that does integer arithmetic can do fixed point arithmetic
    - In fact, the programmer can use ints with an implicit fixed point
    - ints are just fixed point numbers with the binary point to the right of b_0
- **Cons**
  - There is no good way to pick where the fixed point should be
    - Sometimes you need range, sometimes you need precision – the more you have of one, the less of the other.

IEEE Floating Point

- **Analogous to scientific notation**
  - Not 12000000 but 1.2 x 10^7; not 0.0000012 but 1.2 x 10^{-6}
    - (write in C code as: 1.2e7; 1.2e-6)
- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs today
- **Driven by numerical concerns**
  - Standards for handling rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

Floating Point Representation

- **Numerical form:**

$$V_{10} = (-1)^{s} * M * 2^{E}$$

  - 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 a (possibly negative) power of two

- **Representation in memory:**
  - 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**

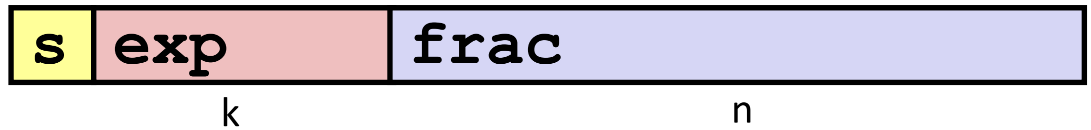
s	exp	frac
1	k=8	n=23

- **Double precision: 64 bits**

s	exp	frac
1	k=11	n=52

Normalization and Special Values

$V = (-1)^{\mathbf{s}} * \mathbf{M} * 2^{\mathbf{E}}$



- **"Normalized" means the mantissa M has the form 1.xxxxx**
  - 0.011 x 2^5 and 1.1 x 2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it

- **How do we represent 0.0? Or special / undefined values like 1.0/0.0?**

Normalization and Special Values

$V = (-1)^{s} * M * 2^{E}$

s	exp	frac
	k	n

- **"Normalized" means the mantissa M has the form 1.xxxxx**
  - 0.011×2^5 and 1.1×2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it

- **Special values:**
  - The bit pattern 00...0 represents zero
  - If **`exp`** == 11...1 and **`frac`** == 00...0, it represents ?
    - e.g. 1.0/0.0 = –1.0/–0.0 = +∞, 1.0/–0.0 = –1.0/0.0 = –∞
  - If **`exp`** == 11...1 and **`frac`** != 00...0, it represents NaN: "Not a Number"
    - Results from operations with undefined result, e.g. sqrt(–1), $\infty - \infty$, $\infty * 0$

How do we do operations?

- **Unlike the representation for integers, the representation for floating-point numbers is not *exact***

Floating Point Operations: Basic Idea

$V = (-1)^{s} * M * 2^{E}$

s	exp	frac
	k	n

- $\mathtt{x} +_{\mathtt{f}} \mathtt{y} = \mathit{Round}(\mathtt{x} + \mathtt{y})$

- $\mathtt{x} *_{\mathtt{f}} \mathtt{y} = \mathit{Round}(\mathtt{x} * \mathtt{y})$

- **Basic idea for floating point operations:**
  - First, compute the exact result
  - Then, *round* the result to make it fit into desired precision:
    - Possibly overflow if exponent too large
    - Possibly drop least-significant bits of significand to fit into **`frac`**

Rounding modes

- **Possible rounding modes (illustrate with dollar rounding):**

	\$1.40	**\$1.60**	**\$1.50**	**\$2.50**	**–\$1.50**
Round-toward-zero	\$1	\$1	\$1	\$2	–\$1
Round-down (-∞)	\$1	\$1	\$1	\$2	–\$2
Round-up (+∞)	\$2	\$2	\$2	\$3	–\$1
Round-to-nearest	\$1	\$2	??	??	??
Round-to-even	\$1	\$2	\$2	\$2	–\$2

- **What could happen if we're repeatedly rounding the results of our operations?**
  - If we always round in the same direction, we could introduce a statistical bias into our set of values!
- **Round-to-even avoids this bias by rounding up about half the time, and rounding down about half the time**
  - Default rounding mode for IEEE floating-point

Mathematical Properties of FP Operations

- **If overflow of the exponent occurs, result will be ∞ or -∞**
- **Floats with value ∞, -∞, and NaN can be used in operations**
  - Result is usually still ∞, -∞, or NaN; sometimes intuitive, sometimes not

- **Floating point operations are not always associative or distributive, due to rounding!**
  - (3.14 + 1e10) - 1e10 != 3.14 + (1e10 - 1e10)
  - 1e20 * (1e20 - 1e20) != (1e20 * 1e20) - (1e20 * 1e20)

Floating Point in C

- **C offers two levels of precision**

  `float` single precision (32-bit)

  `double` double precision (64-bit)

- **Default rounding mode is round-to-even**
- **`#include <math.h>` to get `INFINITY` and `NAN` constants**
- **Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results**
  - Just avoid them!

Floating Point in C

- **Conversions between data types:**
  - Casting between `int`, `float`, and `double` changes the bit representation!!
  - `int` → `float`
    - May be rounded; overflow not possible
  - `int` → `double` or `float` → `double`
    - Exact conversion, as long as int has ≤ 53-bit word size
  - `double` or `float` → `int`
    - Truncates fractional part (rounded toward zero)
    - Not defined when out of range or NaN: generally sets to Tmin

Summary

- **As with integers, floats suffer from the fixed number of bits available to represent them**
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”
- **Mathematically equivalent ways of writing an expression may compute different results**
  - Violates associativity/distributivity
- **Never test floating point values for equality!**

Additional details

- Exponent bias
- Denormalized values – to get finer precision near zero
- Tiny floating point example
- Distribution of representable values
- Floating point multiplication & addition
- Rounding

Normalized Values

$V = (-1)^s * M * 2^E$

s	exp	frac
	k	n

- **Condition:** `exp` $\neq$ `000…0` and `exp` $\neq$ `111…1`
- **Exponent coded as *biased* value: *E* = *exp - Bias***
  - **`exp`** is an *unsigned* value ranging from 1 to 2^k-2 (k == # bits in **`exp`**)
  - *Bias* = $2^{k-1} - 1$
    - Single precision: 127 (so *exp*: 1…254, *E*: -126…127)
    - Double precision: 1023 (so *exp*: 1…2046, *E*: -1022…1023)
  - These enable negative values for E, for representing very small values
- **Significand coded with implied leading 1: *M* = `1.xxx…x`$_2$**
  - **`xxx…x`**: the n bits of **`frac`**
  - Minimum when `000…0` ($M = 1.0$)
  - Maximum when `111…1` ($M = 2.0 - \varepsilon$)
  - Get extra leading bit for "free"

Normalized Encoding Example

$V = (-1)^{s} * M * 2^{E}$

s	exp	frac
	k	n

- **Value:** `float f = 12345.0;`
  - $12345_{10} = 11000000111001_2$
    $= 1.1000000111001_2 \times 2^{13}$ (normalized form)

- **Significand:**

  M = $1.\underline{1000000111001}_2$

  `frac`= $\underline{1000000111001}0000000000_2$

- **Exponent:** E = exp - Bias, so exp = E + Bias

  E = 13

  $Bias$ = 127

  `exp` = 140 = 10001100_2

- **Result:**

0	10001100	10000001110010000000000
s	exp	frac

Denormalized Values

- **Condition: `exp` = 000…0**

- **Exponent value: *E* = exp – *Bias* + 1 (instead of E = exp – 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 value 0
    - 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

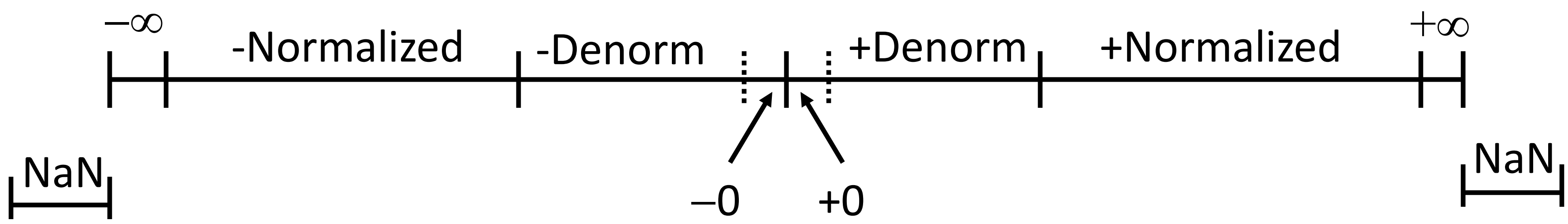
Special Values

- **Condition:** `exp` = `111…1`

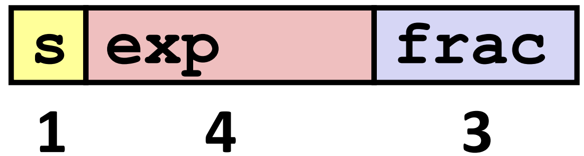
- **Case:** `exp` = `111…1`, `frac` = `000…0`
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = –1.0/–0.0 = +∞, 1.0/–0.0 = –1.0/0.0 = –∞

- **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), $\infty - \infty$, $\infty * 0$

Visualization: Floating Point Encodings



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`

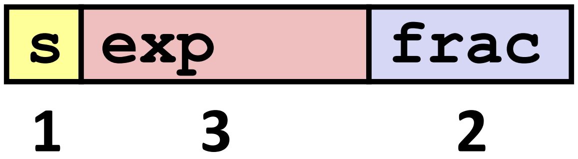
- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

Dynamic Range (Positive Only)

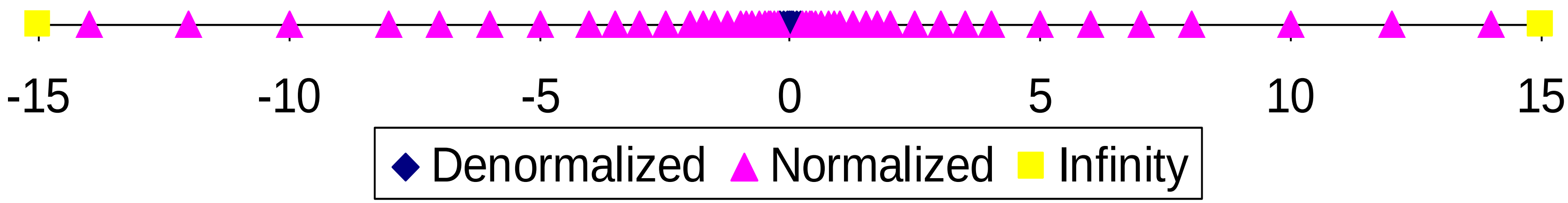
	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001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

Distribution of Values

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

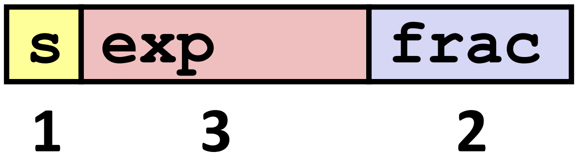


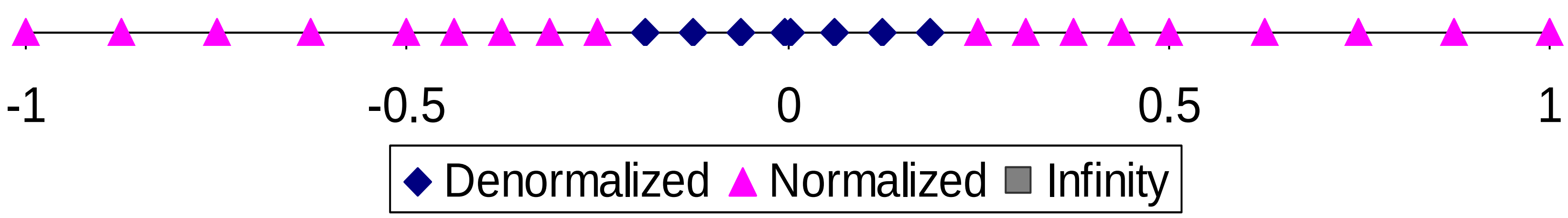
- **Notice how the distribution gets denser toward zero.**



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	$2^{-\{23,52\}} * 2^{-\{126,1022\}}$
▪ Single $\approx 1.4 * 10^{-45}$			
▪ Double $\approx 4.9 * 10^{-324}$			
■ Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) * 2^{-\{126,1022\}}$
▪ Single $\approx 1.18 * 10^{-38}$			
▪ Double $\approx 2.2 * 10^{-308}$			
■ Smallest Pos. Norm.	00…01	00…00	$1.0 * 2^{-\{126,1022\}}$
▪ Just larger than largest denormalized			
■ One	01…11	00…00	1.0
■ Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) * 2^{\{127,1023\}}$
▪ Single $\approx 3.4 * 10^{38}$			
▪ Double $\approx 1.8 * 10^{308}$			

Special Properties of Encoding

- **Floating point zero (0^+) exactly the same bits as integer zero**
  - All bits = 0

- **Can (Almost) Use Unsigned Integer Comparison**
  - Must first compare sign bits
  - Must consider $0^- = 0^+ = 0$
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Floating Point Multiplication

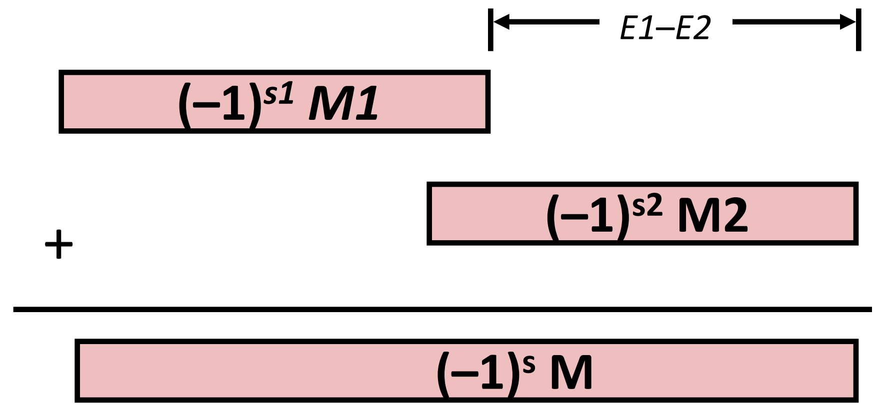
$$(-1)^{s1}\ M1\ \ 2^{E1} \quad * \quad (-1)^{s2}\ M2\ \ 2^{E2}$$

- **Exact Result: $(-1)^{s}\ M\ \ 2^{E}$**
  - Sign s: s1 ^ s2 // xor of s1 and s2
  - Significand M: M1 * 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

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

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**
  - "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

Floating Point and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for ( i=0; i<10; i++ ) {
    f2 += 1.0/10.0;
  }

  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf("f1 = %10.8f\n", f1);
  printf("f2 = %10.8f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf ("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

Memory Referencing Bug

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)   ->     3.14
fun(1)   ->     3.14
fun(2)   ->     3.1399998664856
fun(3)   ->     2.00000061035156
fun(4)   ->     3.14, then segmentation fault
```

Explanation:

Saved State	4
d7 … d4	3
d3 … d0	2
a[1]	1
a[0]	0

Location accessed by `fun(i)`

Representing 3.14 as a Double FP Number

- **1073741824 = 0100 0000 0000 0000 0000 0000 0000 0000**
- **3.14 = 11.0010 0011 1101 0111 0000 1010 000…**
- **$(-1)^s\ M\ 2^E$**
  - S = 0 encoded as 0
  - M = 1.1001 0001 1110 1011 1000 0101 000…. (leading 1 left out)
  - E = 1 encoded as 1024 (with bias)

s	exp (11)	frac (first 20 bits)
0	**100 0000 0000**	**1001 0001 1110 1011 1000**

frac (the other 32 bits)
0101 0000 …

Memory Referencing Bug (Revisited)

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  ->    3.14
fun(1)  ->    3.14
fun(2)  ->    3.1399998664856
fun(3)  ->    2.00000061035156
fun(4)  ->    3.14, then segmentation fault
```

Saved State		4
d7 ... d4	0100 0000 0000 1001 0001 1110 1011 1000	3
d3 ... d0	0101 0000 ...	2
a[1]		1
a[0]		0

Location accessed by `fun(i)`

Memory Referencing Bug (Revisited)

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  ->     3.14
fun(1)  ->     3.14
fun(2)  ->     3.1399998664856
fun(3)  ->     2.00000061035156
fun(4)  ->     3.14, then segmentation fault
```

Saved State		4
d7 … d4	0100 0000 0000 1001 0001 1110 1011 1000	3
d3 … d0	0100 0000 0000 0000 0000 0000 0000 0000	2
a[1]		1
a[0]		0

Location accessed by fun(i)

Memory Referencing Bug (Revisited)

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)   ->     3.14
fun(1)   ->     3.14
fun(2)   ->     3.1399998664856
fun(3)   ->     2.00000061035156
fun(4)   ->     3.14, then segmentation fault
```

Saved State		4
d7 … d4	0100 0000 0000 0000 0000 0000 0000 0000	3
d3 … d0	0101 0000 …	2
a[1]		1
a[0]		0

Location accessed by fun(i)