CS 356 Unit 3

IEEE 754 Floating Point Representation

Floating Point

- Used to represent very small numbers (fractions) and very large numbers
  - Avogadro's Number: $+6.0247 * 10^{23}$
  - Planck's Constant: $+6.6254 * 10^{-27}$
  - Note: 32 or 64-bit integers can't represent this range
- Floating Point representation is used in HLL's like C by declaring variables as **`float`** or **`double`**

Fixed Point

- Unsigned and 2's complement fall under a category of representations called "Fixed Point"
- The radix point is assumed to be in a fixed location for all numbers [Note: we could represent fractions by implicitly assuming the binary point is at the left…A variable just stores bits…you can assume the binary point is anywhere you like]
  - Integers: 10011101. (binary point to right of LSB)
    - For 32-bits, unsigned range is 0 to ~4 billion
  - Fractions: .10011101 (binary point to left of MSB)
    - Range [0 to 1)
- **Main point**: By fixing the radix point, we limit the range of numbers that can be represented
  - Floating point allows the radix point to be in a different location for each value

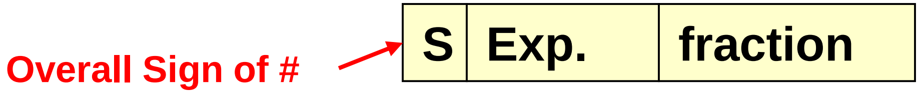
Bit storage

Fixed point Rep.

Floating Point Representation

CS:APP 2.4.2

- Similar to scientific notation used with decimal numbers
  - $\pm$D.DDD * 10 $^{\pm exp}$
- Floating Point representation uses the following form
  - $\pm$b.bbbb * 2$^{\pm exp}$
  - 3 Fields: sign, exponent, fraction (also called mantissa or significand)



Normalized FP Numbers

- Decimal Example
  - $+0.754*10^{15}$ is not correct scientific notation
  - Must have exactly one significant digit before decimal point: $+7.54*10^{14}$
- In binary the only significant digit is '1'
- Thus normalized FP format is:

$\pm 1.bbbbbb * 2^{\pm exp}$

- FP numbers will **always be normalized** before being stored in memory or a reg.
  - The *1.* is actually not stored but assumed since we always will store normalized numbers
  - If HW calculates a result of $0.001101*2^5$ it must normalize to $1.101000*2^2$ before storing

IEEE Floating Point Formats

- Single Precision (32-bit format)
  - 1 Sign bit (0=pos/1=neg)
  - 8 Exponent bits
    - Excess-127 representation
    - More on next slides
  - 23 fraction (significand or mantissa) bits
  - Equiv. Decimal Range:
    - 7 digits x $10^{\pm 38}$

1	8	23
S	Exp.	Fraction

- Double Precision (64-bit format)
  - 1 Sign bit (0=pos/1=neg)
  - 11 Exponent bits
    - Excess-1023 representation
    - More on next slides
  - 52 fraction (significand or mantissa) bits
  - Equiv. Decimal Range:
    - 16 digits x $10^{\pm 308}$

1	11	52
S	Exp.	Fraction

Exponent Representation

- Exponent needs its own sign (+/-)
- Rather than using 2's comp. system we use Excess-N representation
  - Single-Precision uses Excess-127
  - Double-Precision uses Excess-1023
  - w-bit exponent => Excess-$2^{(w-1)}$-1
  - This representation allows FP numbers to be easily compared
- Let E' = stored exponent code and E = true exponent value
- For single-precision: E' = E + 127
  - 2^1 => E = 1, E' = 128_{10} = 10000000_2
- For double-precision: E' = E + 1023
  - 2^{-2} => E = -2, E' = 1021_{10} = 01111111101_2

2's comp.	E' (stored Exp.)	Excess-127
-1	1111 1111	+128
-2	1111 1110	+127
-128	1000 0000	1
+127	0111 1111	0
+126	0111 1110	-1
+1	0000 0001	-126
0	0000 0000	-127

Comparison of 2's comp. & Excess-N

Q: Why don't we use Excess-N more to represent negative #'s

Comparison & The Format

- Why put the exponent field before the fraction?
  - Q: Which FP number is bigger:
    $0.9999*2^2$ or $1.0000*2^1$
  - A: We should look at the exponent first to compare FP values and only look at the fraction if the exponents are equal
- By placing the exponent field first we can compare entire FP values as single bit strings (i.e. as if they were unsigned)

0	**10000010**	**0000001000**

0	**10000001**	**1110000000**

0100000100000001000

0100000011110000000

< > = ???

Exponent Representation

- FP formats reserved the exponent values of all 1's and all 0's for special purposes
- Thus, for single-precision the range of exponents is -126 to + 127

E' (range of 8-bits shown)	E (=E'-127) and special values
255 = 11111111	Reserved
254 = 11111110	E'-127=+127
…	
128 = 10000000	E'-127=+1
127 = 01111111	E'-127=0
126 = 01111110	E'-127=-1
…	
1 = 00000001	E'-127=-126
0 = 00000000	Reserved

IEEE Exponent Special Values

Exp. Field	Fraction Field	Meaning
000…00	0000...0000	±0
	Non-Zero	Denormalized **(±0.bbbbbb * 2^{-126})**
111…11	0000...0000	± infinity
	Non-Zero	NaN (Not A Number) - 0/0, 0*∞,SQRT(-x)

Single-Precision Examples

CS:APP 2.4.3

1

$2^7=128$ $2^1=2$

1	**1000 0010**	**110 0110 0000 0000 0000 0000**

130-127=3

$$-1.1100110 * 2^3$$
$$= -1110.011 * 2^0$$
$$= -14.375$$

2

+0.6875 = +0.1011

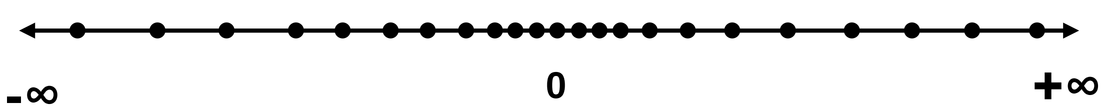
$$= +1.011 * 2^{-1}$$

-1 +127 = 126

0	0111 1110	011 0000 0000 0000 0000 0000

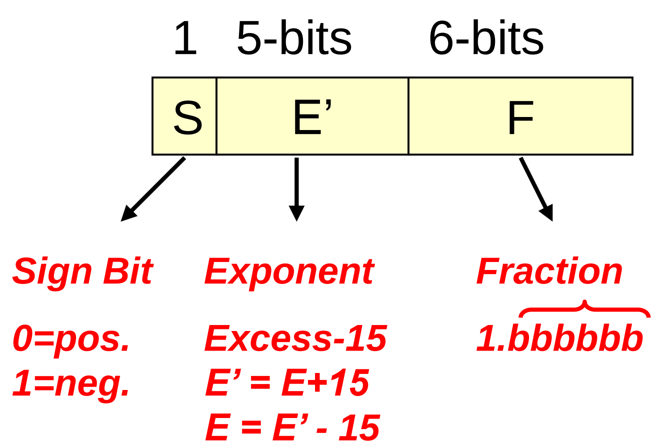
Floating Point vs. Fixed Point

- Single Precision (32-bits) Equivalent Decimal Range:
  - 7 significant decimal digits * $10^{\pm 38}$
  - Compare that to 32-bit signed integer where we can represent ±2 billion. How does a 32-bit float allow us to represent such a greater range?
  - FP allows for range but sacrifices precision (can't represent all numbers in its range)
- Double Precision (64-bits) Equivalent Decimal Range:
  - 16 significant decimal digits * $10^{\pm 308}$



12-bit "IEEE Short" Format

- 12-bit format defined just for this class (doesn't really exist)
  - 1 Sign Bit
  - 5 Exponent bits (using Excess-15)
    - Same reserved codes
  - 6 Fraction (significand) bits



Examples

1

1	10100	101101

20-15=5

$-1.101101 * 2^5$

$= -110110.1 * 2^0$

$= -110110.1 = -54.5$

2

$+21.75 = +10101.11$

$= +1.010111 * 2^4$

4+15=19

0	10011	010111

3

1	01101	100000

13-15=-2

$-1.100000 * 2^{-2}$

$= -0.011 * 2^0$

$= -0.011 = -0.375$

4

$+3.625 = +11.101$

$= +1.110100 * 2^1$

1+15=16

0	10000	110100

ROUNDING

The Need To Round

CS:APP 2.4.4

- Integer to FP
  - +725 = 1011010101 = $1.011010101 * 2^9$
    - If we only have 6 fraction bits, we can't keep all fraction bits
- FP ADD / SUB

```
  5.9375 x 10^1           .00059375 x 10^5
+ 2.3256 x 10^5   ==>   + 2.3256      x 10^5
```

- FP MUL / DIV

```
          1.010110
*         1.110101
------------------
10.011101001110
```

```
         1.010110
       * 1.110101
     ------------
          1010110
        1010110--
      1010110----
     1010110-----
  + 1010110------
  ---------------
   10.011101001110
```

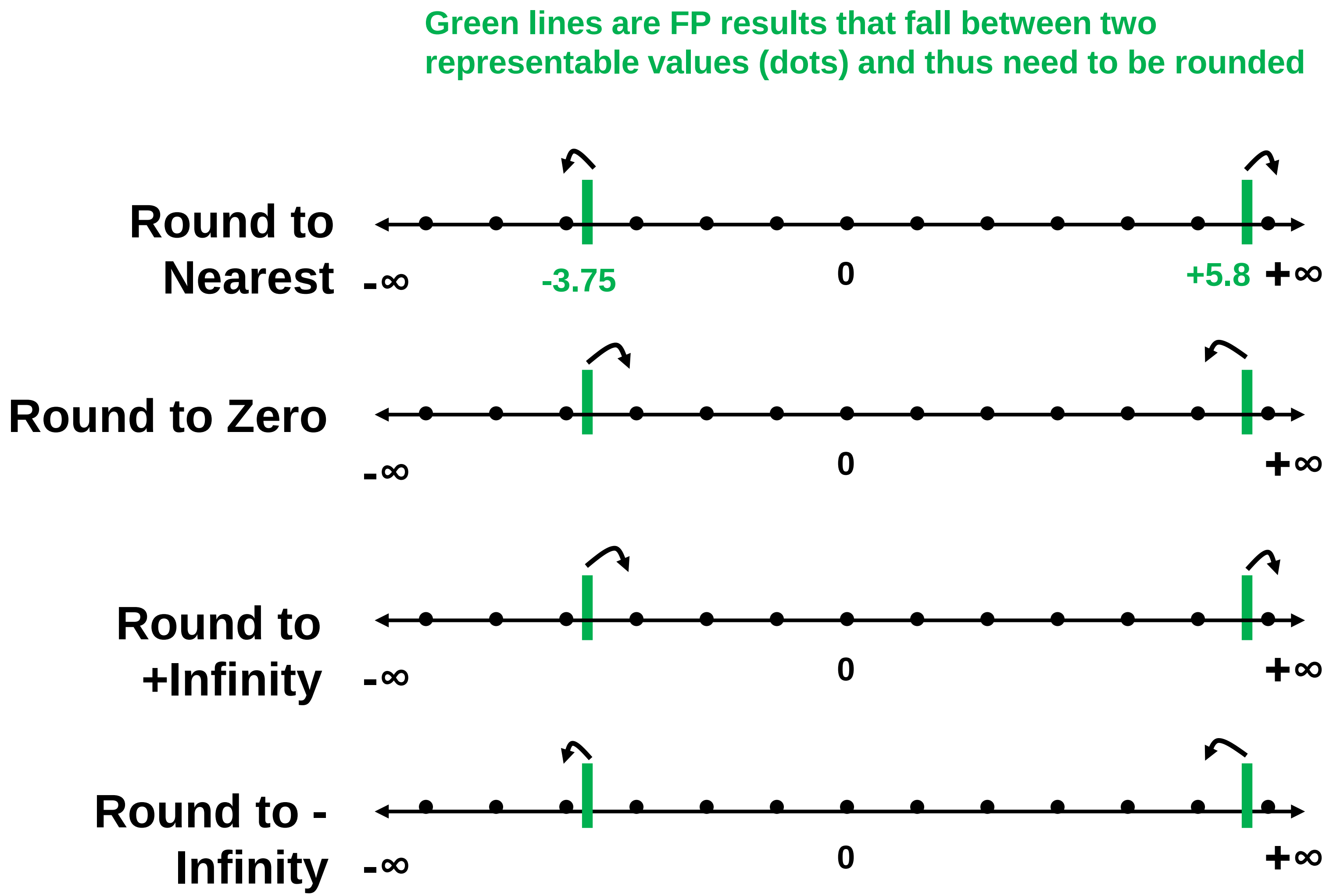
Make sure to move the binary point

Rounding Methods

- 4 Methods of Rounding (you are only responsible for the first 2)

Round to Nearest (Round to Even)	Normal rounding you learned in grade school. Round to the nearest representable number. If exactly halfway between, round to representable value w/ 0 in LSB (**i.e. nearest even fraction**).
Round towards 0 (Chopping)	Round the representable value closest to but not greater in magnitude than the precise value. Equivalent to just dropping the extra bits.
Round toward +∞ (Round Up)	Round to the closest representable value greater than the number
Round toward -∞ (Round Down)	Round to the closest representable value less than the number

Number Line View Of Rounding Methods



Rounding to Nearest Method

- Same idea as rounding in decimal
- Examples: Round 1.23xx to the nearest 1/100th
  - 1.2351 to 1.2399 => round up to 1.24
  - 1.2301 to 1.2349 => round down to 1.23
  - 1.2350 => Rounding options 1.23 or 1.24
    - Choose the option with an even digit in the LS place (i.e. 1.24)
  - 1.2450 => Rounding options 1.24 or 1.25
    - Choose the option with an even digit in the LS place (i.e. 1.24)
- Which option has the even digit is essentially a 50-50 probability of leading to rounding up vs. rounding down
  - Attempt to reduce bias in a sequence of operations

Rounding in Binary

- What does "exactly" half way correspond to in binary
  (i.e. 0.5 dec. = ??)
- Hardware will keep some additional bits beyond what can be stored to help with rounding
  - Referred to as the Guard bit(s), Round bit, and Sticky bit (GRS)
- Thus, if the additional bits are:
  - 10…0 = Exactly half way
  - 0x…x = Less than half way (round down)
  - Anything else = More than half way (round up)

0.5 = 0. 1 0 0

Bits that fit in FRAC field

$1.0100101 01 \times 2^4$ → `1.010010101 x 2^4`

GRS

Additional bits: 101

Round to Nearest

$1.001100\mathbf{110} \times 2^4$

Additional bits: 110

⇩

Round up (fraction + 1)

0	10011	001101

$1.111111\mathbf{101} \times 2^4$

Additional bits: 101

⇩

Round up (fraction + 1)

$$\begin{array}{rl} & 1.111111 \times 2^4 \\ + & 0.000001 \times 2^4 \\ \hline & 10.000000 \times 2^4 \\ & 1.000000 \times 2^5 \end{array}$$

Requires renormalization

0	10100	000000

$1.001101\mathbf{001} \times 2^4$

Additional bits: 001

⇩

Leave fraction

0	10011	001101

Round to Nearest

- In all these cases, the numbers are halfway between the 2 possible round values
- Thus, we round to the value w/ 0 in the LSB

$1.001100\textbf{100} \times 2^4$

Additional bits: 100

⇩

Rounding options are: 1.001100 or 1.001101

In this case, round down

0	10011	001100

$1.111111\textbf{100} \times 2^4$

Additional bits: 100

⇩

Rounding options are: 1.111111 or 10.000000

In this case, round up

$$
\begin{array}{rl}
 & 1.111111 \times 2^4 \\
+ & 0.000001 \times 2^4 \\
\hline
 & 10.000000 \times 2^4 \\
 & 1.000000 \times 2^5
\end{array}
$$

Requires renormalization

0	10100	000000

$1.001101\textbf{100} \times 2^4$

Additional bits: 100

⇩

Rounding options are: 1.001101 or 1.001110

In this case, round up

0	10011	001110

Round to 0 (Chopping)

- Simply drop the G,R,S bits and take fraction as is

GRS

`1.001100001 x 2^4` (GRS = **001**)

drop G,R,S bits

0	10011	001100

GRS

`1.001101101 x 2^4` (GRS = **101**)

drop G,R,S bits

0	10011	001101

GRS

`1.001100111 x 2^4` (GRS = **111**)

drop G,R,S bits

0	10011	001100

MAJOR IMPLICATIONS FOR PROGRAMMERS

FP Addition/Subtraction

CS:APP 2.4.5

- FP addition/subtraction is NOT associative
  - Because of rounding and use of infinity
    $(a+b)+c \neq a+(b+c)$
  - Add similar, small magnitude numbers before larger magnitude numbers
- Example of rounding

$$(0.0001 + 98475) - 98474 \neq 0.0001 + (98475-98474)$$
$$98475-98474 \neq 0.0001 + 1$$
$$1 \neq 1.0001$$

- Example of infinity

$$1 \quad + \quad 1.11...1*2^{127} \quad - \quad 1.11...1*2^{127}$$

Floating point MUL/DIV

- Also not associative
- Doesn't distribute over addition
  - a*(b+c) ≠ a*b + a*c
  - Example†:
    - (big1 * big2) / (big3 * big4) => Overflow on first mul.
    - 1/big3 * 1/big4 * big1 * big2 => Underflow on first mul.
    - (big1 / big3) * (big2 / big4) => Better
- Note: Take care even with integer mul/div
  - F = (9/5)*C + 32
  - Should be F = (9*C)/5 + 32

†https://www.soa.org/News-and-Publications/Newsletters/Compact/2014/may/Losing-My-Precision--Tips-For-Handling-Tricky-Floating-Point-Arithmetic.aspx

FP Comparison

- Beware of equality (==) check or even less- or greater-than
- Generally don't use FP as loop counters
- Common approach to replace equality check
  - Check if difference of two values is within some small epsilon
  - Many questions are raised by this…(what epsilon, what about sign, transitive equality)?

```
float x = 0.2 + 0.3; // 0.5?
float y = 0.15 + 0.35; // 0.5?
if(x == y) printf("Equal\n");
```

Will "Equal" be printed?

```
double t;
int cnt=0;
for(t=0.0; t < 1.0; t += 0.1)
{
  printf("%d\n", cnt++);
}
```

What values of 'cnt' will be printed?

```
bool simple_within(
  float a, float b, float eps)
{
  return fabs(a-b) < eps;
}
```

FP & Compiler Optimizations

- Suppose we want to compute:

```
x = a + b + c;
y = b + c + d;
```

- Can the compiler optimize this as:

```
temp = b + c;
x = a + temp;
y = temp + d;
```

Floating point values in C

CS:APP 2.4.6

- Two types: `float` and `double`
  - IEEE floating point when supported
  - Rounds to even
- No standard way to change rounding
- No standard way to get special values

Casting and C

Cast	Overflow Possible?	Rounding Possible?	Notes
int to float	No	Yes	
int to double	No	No	
float to double	No	No	
double to float	Yes	Yes	
float/double to int	Yes	Yes	Round to 0 is used to truncate fractional values (i.e. 1.9 => 1) If overflow, use MAX-NEG int.

FURTHER INQUIRY

Rounding Implementation

- There may be a large number of bits after the fraction
- To implement any of the methods we can keep only a subset of the extra bits after the fraction [hardware is finite]
  - **G**uard bits: bits immediately after LSB of fraction (many HW implementations keep up to 16 additional guard bits)
    - ****Lookup online the usage & importance of these guard bits****
  - **R**ound bit: bit to the right of the guard bits
  - **S**ticky bit: Logical OR of all other bits after Guard & R bits

1.01001010010×2^4

Logical OR (output is '1' if any input is '1', '0' otherwise

1.010010101×2^4

GRS

We can perform rounding to a 6-bit fraction using just these 3 bits.

More

- Some links
  - https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
  - http://floating-point-gui.de/