EE 357 Unit 3

IEEE 754 Floating Point Representation

Floating Point Arithmetic

Floating Point

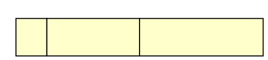
- Used to represent very small numbers (fractions) and very large numbers
  - Avogadro's Number: $+6.0247 * 10^{23}$
  - Planck's Constant: $+6.6254 * 10^{-27}$
  - Note: 32 or 64-bit integers can't represent this range
- Floating Point representation is used in HLL's like C by declaring variables as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

Floating Point Representation

- Similar to ______________________
  -
- Floating Point representation uses the following form
  -
  - 3 Fields: ________, ____________, ____________________



USC Viterbi School of Engineering

Normalized FP Numbers

- Decimal Example
- In binary the only significant digit is ________
- Thus normalized FP format is:
- FP numbers will always be normalized before being _________________________
  - Note:

USC Viterbi School of Engineering

IEEE Floating Point Formats

- Single Precision (32-bit format)
  - 1 Sign bit
  - ___ Exponent bits using ______________ representation
  - ___ Fraction bits
  - Equiv. Decimal Range: 7 digits x $10^{\pm 38}$
- Double Precision (64-bit format)
  - 1 Sign bit
  - ___ Exponent bits using ______________ representation
  - ___ Fraction bits
  - Equiv. Decimal Range: 16 digits x $10^{\pm 308}$

S	Exp.	fraction

S	Exp.	fraction

USC Viterbi School of Engineering

Exponent Representation

- Exponent includes its own sign (+/-)
- Rather than using 2's comp. system, Single-Precision uses Excess-127 while Double-Precision uses Excess-1023
  - 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.		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

USC Viterbi School of Engineering

Exponent Representation

- FP formats
  ____
  ____
  ____
  ____
- Thus, for single-precision the range of exponents is
  ____

E' (range of 8-bits shown)	E (E = E'-127)

IEEE Exponent Special Values

E'	Fraction	Meaning

Single-Precision Examples

1

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

2

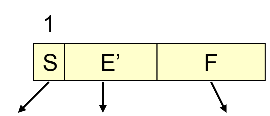
+0.6875 = +0.1011

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 _____________. How does a 32-bit float allow us to represent such a greater range?
  -
- Double Precision (64-bits) Equivalent Decimal Range:
  - 16 significant decimal digits * $10^{\pm 308}$

IEEE Shortened Format

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



Examples

1. 1 | 10100 | 101101

2. +21.75 = +10101.11

3. 1 | 01101 | 100000

4. +3.625 = +11.101

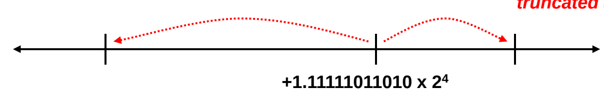
Truncation & Rounding

- May have more bits than fraction can store due to arithmetic operations, etc.
- Need to truncate these bits by rounding the number to a value that can be represented with the given number of fraction bits (Assume 6-bits)

```
   0.00001011010  x 2^4
+  1.11010000000  x 2^4
   1.11111011010  x 2^4
```

fraction can only be 6-bits

Extra bits that need to be truncated (rounded off)



Rounding Methods

- 4 Methods of Rounding
  - We will focus on just the first 2 methods

Round to ________	Similar to rounding you learned in grade school.
Round to ________	Round the representable value closest to but not greater in magnitude than the precise value. Equivalent to ____________________
Round toward ________	Round to the closest representable value
Round toward ________	Round to the closest representable value

Rounding Implementation

- It is possible to have a large number of bits after the fraction
- To do the rounding though we can keep only a subset of the extra bits after the fraction
  1. ________ bits: bits immediately after LSB of fraction ______________________________
  2. ________:
  3. ________:

1.01001010010 x 2⁴

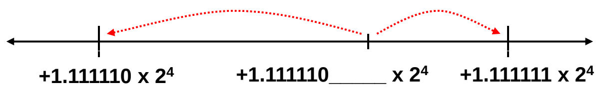
Rounding to Nearest Method

- Same idea as rounding in decimal
  - .51 and up, round up,
  - .49 and down, round down,
  - .50 exactly we round up in decimal
    - In this method we treat it differently…If precise value is exactly half way between 2 representable values, round towards the number with 0 in the LSB

Round to Nearest Method

- Round to the closest representable value
  - If precise value is exactly half way between 2 representable value, round towards the number with 0 in the LSB

```
1.11111011010   x 2^4
1.111110___     x 2^4
        GRS
```



Rounding to Nearest Method

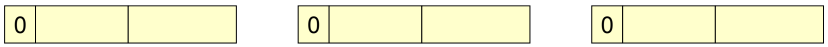
- 3 Cases in binary FP:
  - ________________ => Greater than half way
    - Round fraction up (add 1 to fraction)
    - [may require renormalization]
  - ________________=> Exactly half way
    - Round to the closest fraction value with a '0' in the LSB
    - [may require a re-normalization]
  - ________________=> Less than half way
    - Leave fraction alone (add 0 to fraction)

Round to Nearest

GRS: 1.001100110 x 2^4

GRS: 1.111111101 x 2^4

GRS: 1.001101001 x 2^4



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

GRS: $1.001100\mathbf{100} \times 2^4$ GRS: $1.111111\mathbf{100} \times 2^4$ GRS: $1.001101\mathbf{100} \times 2^4$

0		

0		

0		

Round to 0 (Chopping)

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

GRS: $1.001100\mathbf{001} \times 2^4$ → drop G,R,S bits ⇩

0	10011	001100

GRS: $1.001101\mathbf{101} \times 2^4$ → drop G,R,S bits ⇩

0	10011	001101

GRS: $1.001100\mathbf{111} \times 2^4$ → drop G,R,S bits ⇩

0	10011	001100

FP Addition / Subtraction

- In decimal addition:

$$\begin{array}{r} 5.9375 \times 10^3 \\ +\ 2.3250 \times 10^5 \\ \hline \end{array}$$

FP Addition/Subtraction

1. Make exponents equal by selecting number w/ _________ exponent and shifting it ________
2. Convert subtraction to addition
3. If p+p or n+n
   a. ___________ magnitudes
   b. Sign of result = _________________
4. If p+n or n+p
   a. ____
   b. Sign of result = _______________________
5. Normalize and round

USC Viterbi School of Engineering

FP Addition/Subtraction

- Remember to update G,R,S when shifting to make exponents equal

A = | 0 | 10010 | 110101 | + B = | 0 | 10000 | 010110 |

= 1.110101×2^3 = 1.010110×2^1

USC Viterbi School of Engineering

FP Addition/Subtraction

- Since |A|>|B|, ________________
  - –

| 0 | 10000 | 010110 | + | 1 | 01110 | 110101 |

= +1.010110 0 0 0 x 2^1 (G R S)

= - 1.110101 0 0 0 x 2^{-1} ← Smaller exponent, shift right

= - 0.011101 0 1 0 x 2^1 (G R S)

1.010110000 x 2^1

\- ________ x 2^1 ⇒

⇐

| | | |

USC Viterbi School of Engineering

FP Addition/Subtraction Example 3

| 1 | 10100 | 011010 | + | 0 | 10100 | 110100 |

= -1.011010×2^5 = $+1.110100 \times 2^5$

= | 0 | 10010 | 101000 |

USC Viterbi School of Engineering

FP Multiplication / Division

Multiplication: Multiply fractions and add exponents

$3.45 \times 10^4 * 4.90 \times 10^1$

$= (3.45 * 4.90) \times 10^{(4+1)}$

Division: Divide fractions and subtract exponents

$3.45 \times 10^4 * 4.90 \times 10^1$

$= (3.45 / 4.90) \times 10^{(4-1)}$

FP Multiplication

1.

2.

3.

4.

FP Multiplication

- Add the exponents and subtract the Excess value (IEEE=127, shortened IEEE=15)

0	10000	010110

= 1.010110 x 2^1

*

0	10011	110101

= 1.110101 x 2^4

```
   10000 = 2^1
 + 10011 = 2^4
 ---------
  100011
 -001111
 ---------
  010100 = 2^5
```

This result is Excess-30, so subtract 15 to get Excess-15

FP Multiplication

- Multiply fractions
  - keep extra guard bits (extra LSB's)

0	10000	010110

= 1.010110 x 2^1

*

0	10011	110101

= 1.110101 x 2^4

Exponent 10100 = 2^5

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

Make sure to move the binary point

FP Multiplication

- Determine sign

0	10000	010110

= 1.010110 x 2^1

*

0	10011	110101

= 1.110101 x 2^4

Exponent 10100 = 2^5

fraction 10.011101001110

Sign pos. * pos. = pos.

FP Multiplication

- Normalize and truncate guard bits

0	10000	010110	*	0	10011	110101	

= 1.010110 x 2^1 = 1.110101 x 2^4

Exponent `10100 = 2^5`
fraction `10.011101001110`
Sign `pos. * pos. = pos.`

10.011101001110 x 2^5

GRS

1.001110101 x 2^6

1.001111 x 2^6

0	10101	001111

For Round-to-Nearest we look at the G,R,S bits see that we should round up by adding 1 to the LSB.

FP Multiplication

- Analyze results

0	10000	010110	*	0	10011	110101	=	0	10101	001111

= 1.010110 x 2^1 = 1.110101 x 2^4 = 1.001111 x 2^6

= 2.6875 = 29.25 Computed result = 79

True result = 78.609375

Error = +0.390625

FP Division

1. Determine the sign
2. Subtract the exponents and add the Excess value (127 or 15)
3. Divide the fractions
4. Normalize and round the resulting value

FP Division

- Subtract the exponents and add the Excess value (IEEE=127, shortened IEEE=15)

0	10011	110100	/	0	10000	110000

= 1.110100 x 2^4 = 1.110000 x 2^1

```
  10011 = 2^4
- 10000 = 2^1
-------
 000011
+001111
-------
 010010 = 2^3
```

This result is Excess-0, so add 15 to get Excess-15

FP Division

- Divide fractions (align binary point by moving it to the right of the divisor)

0	10011	110100	/	0	10000	110000

= 1.110100×2^4 = 1.110000×2^1

Exponent 010010 = 2^3

1.11 ⟌ 1.1101000000 = 111 ⟌ 111.01000000

FP Division

- Divide fractions
  - take it out to guard, round
  - If there is a remainder, set sticky bit.

0	10011	110100	/	0	10000	110000

= 1.110100×2^4 = 1.110000×2^1

Exponent 010010 = 2^3

```
            GRS
      001.000010011
111 ⟌ 111.010000000
    - 111
        0.01000
      - 0.00111
        0.00001000
      - 0.00000111
        0.00000001
```

If any remainder after Round-Bit, simply set the Sticky bit.

FP Division

- Determine sign

0	10011	110100	/	0	10000	110000

= 1.110100×2^4 = 1.110000×2^1

Exponent 010010 = 2^3
fraction 1.000010011
Sign pos. / pos. = pos.

FP Division

- Normalize and truncate guard bits

0	10011	110100	/	0	10000	110000

= 1.110100×2^4 = 1.110000×2^1

Exponent 010010 = 2^3
fraction 1.00001001
Sign pos. / pos. = pos.

1.000010011×2^3 *Luckily, it is already in normal form*

1.000010011×2^3 ***For Round-to-Nearest we look at the G,R,S bits see that we should round down***

= 1.000010×2^3

0	10010	000010

FP Division

- Analyze results

0	10011	110100	/	0	10000	110000	=	0	10010	000010

= 1.110100×2^4

= 29

= 1.110000×2^1

= 3.5

= 1.000010×2^3

Computed result = 8.25

True result = 8.2857

Error = -0.0357

USC Viterbi School of Engineering

Floating-Point Exceptions

- Error conditions that can be trapped (recognized by the HW) and passed to SW to deal with
  1. ________ – Result is
  2. _________ – Result is
  3. Inexact –
  4. Invalid – Result is
     - Can be a
  5. Divide-by-Zero – Just like it sounds
     - If not trapped,

© Mark Redekopp, All rights reserved

USC Viterbi School of Engineering

Intel FPU Exception Handling

- Status word
  - RC = Rounding Control
    - 00 (nearest), 01 (down), 10 (up), 11 (truncate)
  - PC = Precision Control
  - PM = Precision Mask
  - UM/OM = Underflow / Overflow Mask
  - ZM / DM = Div/0 / Denormalized Mask
  - IM = Invalid Mask (NaN)

15	12	11 10	9 8	7		5	4	3	2	1	0
0 0 0	IC	RC	PC	IEM	0	PM	UM	OM	ZM	DM	IM

© Mark Redekopp, All rights reserved

USC Viterbi School of Engineering

Intel FPU Exception Handling

- Status word
  - P = Precision event occurred
  - U = Underflow occurred
  - O = Overflow occurred
  - Z = Divide by zero occurred
  - D = Denormalized number occurred
  - I = Invalid number occurred

15 12 11 10 9 8 6	5	4	3	2	1	0
Other bits indicating status	P	U	O	Z	D	I

© Mark Redekopp, All rights reserved

Warning

- FP addition/subtraction is NOT associative
  - Because of rounding / inability to precisely represent fractions, $(a+b)+c \neq a+(b+c)$

(small + LARGE) – LARGE ≠ small + (LARGE – LARGE)

Why? Because of ____________________