University of Southern California

Viterbi School of Engineering

EE352
Computer Organization and Architecture

IEEE 754 Floating Point Representation
Floating Point Arithmetic

References:

1) Textbook
2) Mark Redekopp's slide series

Shahin Nazarian

Spring 2010

Floating Point

- Programming languages support numbers with fractions (aka real in mathematics)
  - Floating point is 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 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)

Overall Sign of #

S	Exp.	fraction

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 will always 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=p/1=n)
  - 8 Exponent bits (Excess-127 representation)
  - 23 fraction (significand or mantissa) bits
  - Equiv. Decimal Range: 7 digits x $10^{\pm 38}$
- Double Precision (64-bit format)
  - 1 Sign bit (0=p/1=n)
  - 11 Exponent bits (Excess-1023 representation)
  - 52 fraction (significand or mantissa) bits
  - Equiv. Decimal Range: 16 digits x $10^{\pm 308}$

1	8	23
S	Exp.	Fraction

1	11	52
S	Exp.	Fraction

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
  - Note: Excess-N is also called biased representation

	(E')	(E)
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

Exponent Representation

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

E' (range of 8-bits shown)	E ($E = E'-127$)
11111111	Reserved
11111110	E'-127=+127
…	
10000000	E'-127=+1
01111111	E'-127=0
01111110	E'-127=-1
…	
00000001	E'-127=-126
00000000	Reserved

IEEE Exponent Special Values

E'	Fraction	Meaning
All 0's	All 0's	0
All 0's	Not all 0's (any bit = '1')	Denormalized ($0.\text{fraction} \times 2^{-126}$)
All 1's	All 0's	Infinity
All 1's	Not all 0's (any bit = '1')	NaN (Not A Number) - 0/0, $0 * \infty$, SQRT(-x)

Single-Precision Examples

1

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

$$= +1.\underbrace{011} * 2^{-1}$$

-1 +127 = 126

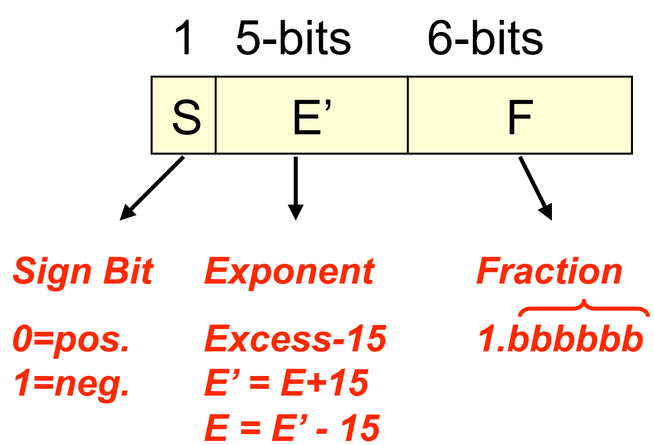
0	0111 1110	011 0000 0000 0000 0000 0000

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 number in its 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
  - 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

$+21.75 = +10101.11$

$$= +1.\underbrace{010111}_{} * 2^4$$

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

$+3.625 = +11.101$

$$= +1.\underbrace{110100}_{} * 2^1$$

1+15=16

0	10000	110100

Rounding

- Unlike integers which can represent exactly every number btn the smallest and large numbers, floating point numbers are normally approximations for a number they can't really represent. This is because there are infinite real numbers between say 0 and 1, but no more than 2^{53} can be represented exactly in double precision floating point
- IEEE754 offers several modes of rounding to let the programmer pick the desired approximation
- Rounding sounds simple, however to round accurately requires hardware to include extra bits in the calculation

Rounding Methods

- $+213.125 = 1.1010101001 * 2^7$ => Can't keep all fraction bits
- 4 Methods of Rounding (we will focus on just the first 2)

Round to Nearest	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
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 $+\infty$ (Round Up)	Round to the closest representable value greater than the number
Round toward $-\infty$ (Round Down)	Round to the closest representable value less than the number

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
  - **Guard** bits: bits immediately after LSB of fraction (in this class we will usually keep only 1 guard bit)
  - **Round** bit: bit to the right of the guard bits
  - **Sticky** bit: Logical OR of all other bits after G & R bits

$$1.010010\underbrace{10010}_{}\ \times 2^4$$

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

$$1.010010101 \times 2^4$$

GRS

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

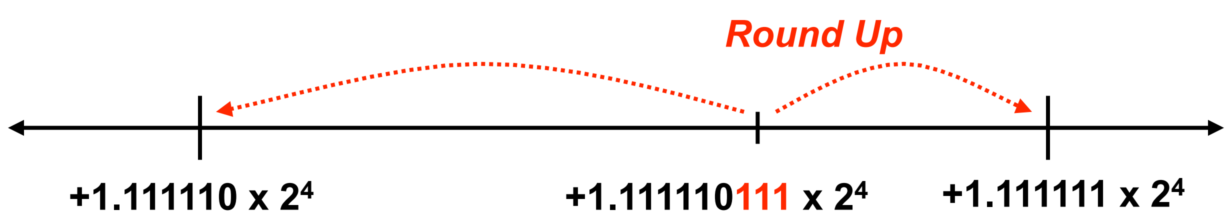
Rounding to Nearest Method

- **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**

Rounding to Nearest Method (Cont.)

- 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
1.111110111     x 2^4
          GRS
```



Precise value will be rounded to one of the representable value it lies between

In this case, round up because precise value is closer to the next higher representable values

Rounding to Nearest Method (Cont.)

- 3 Cases in binary FP:
  - G = '1' & ($R,S \neq 0,0$) =>
    - round fraction up (add 1 to fraction)
    - may require a re-normalization
  - G = '1' & ($R,S = 0,0$) =>
    - round to the closest fraction value with a '0' in the LSB
    - may require a re-normalization
  - G = '0' =>
    - leave fraction alone (add 0 to fraction)

Rounding to Nearest Method (Cont.)

GRS

$1.001100\,110 \times 2^4$

G = '1' & R,S ≠ 0,0

⇩

Round up (fraction + 1)

0	10011	001101

GRS

$1.111111\,101 \times 2^4$

G = '1' & R,S ≠ 0,0

⇩

Round up (fraction + 1)

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

Requires renormalization

0	10100	000000

GRS

$1.001101\,001 \times 2^4$

G = '0'

⇩

Leave fraction

0	10011	001101

Rounding to Nearest Method (Cont.)

- 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

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

G = '1' and R,S = '0'

⇩

Rounding options are:
1.001100 or 1.001101

In this case, round down

0	10011	001100

GRS

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

G = '1' and R,S = '0'

⇩

Rounding options are:
1.111111 or 10.000000

In this case, round up

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

Requires renormalization

0	10100	000000

GRS

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

G = '1' and R,S = '0'

⇩

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

$1.001100\textcolor{red}{001} \times 2^4$

drop G,R,S bits

0	10011	001100

GRS

$1.001101\textcolor{red}{101} \times 2^4$

drop G,R,S bits

0	10011	001101

GRS

$1.001100\textcolor{red}{111} \times 2^4$

drop G,R,S bits

0	10011	001100

FP Addition / Subtraction

- **In decimal addition:**
  - Must line up decimal point → Equal exponents

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

FP Addition / Subtraction

- **In decimal addition:**
  - Must line up decimal point — Equal exponents

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

$$\begin{array}{r} .059375 \times 10^5 \\ +\ 2.3250 \quad \times 10^5 \\ \hline \end{array}$$

Must do the same thing in binary

FP Addition/Subtraction

1. Make exponents equal by selecting number w/ smaller exponent and shifting it right
2. Convert subtraction to addition
3. If p+p or n+n
   a. Add magnitudes
   b. Sign of result, is same as operands
4. If p+n or n+p
   a. Subtract smaller magnitude from larger magnitude
   b. Sign of result is same as larger operand
5. Normalize and round

FP Addition/Subtraction 1

- **Shift the number with the smaller exponent to the right until exponents are equal (updating G,R,S bits)**

0	10010	110101

= 1.110101×2^3

\+

0	10000	010110

= 1.010110×2^1

Smaller exponent, shift right

FP Addition/Subtraction 1

- **Shift the number with the smaller exponent to the right until exponents are equal, maintaining Guard, Round, and Sticky bits**

0	10010	110101

$= 1.110101\ \underline{0}\ \underline{0}\ \underline{0} \times 2^3$

G R S

+

0	10000	010110

$= 1.010110\ \underline{0}\ \underline{0}\ \underline{0} \times 2^1$

G R S

Smaller exponent, shift right

$= 0.101011\ \underline{0}\ \underline{0}\ \underline{0} \times 2^2$ shift by 1

$= 0.010101\ \underline{1}\ \underline{0}\ \underline{0} \times 2^3$ shift by 2

Remember, shifting the fraction right is making it's value smaller, thus the exponent increases

FP Addition/Subtraction 1

- **Now add (p+p so add magnitudes)**

0	10010	110101	+	0	10000	010110

$= 1.110101\ \underline{0}\ \underline{0}\ \underline{0} \times 2^3$ $\qquad$ $= 0.010101\ \underline{1}\ \underline{0}\ \underline{0} \times 2^3$

```
            GRS
  1.110101000   x 2^3
+ 0.010101100   x 2^3
 ------------
 10.001010100   x 2^3
```

FP Addition/Subtraction 1

- **Now add**

0	10010	110101

$= 1.110101\ \underline{0}\ \underline{0}\ \underline{0} \times 2^3$

+

0	10000	010110

$= 0.010101\ \underline{1}\ \underline{0}\ \underline{0} \times 2^3$

$$
\begin{array}{rl}
 & \underline{\text{GRS}} \\
 & 1.110101\,000 \quad \times 2^3 \\
+ & 0.010101\,100 \quad \times 2^3 \\
\hline
 & 10.001010\,100 \quad \times 2^3 \\
 & 1.000101\,010 \quad \times 2^4
\end{array}
$$

Move binary point after first '1'

E' = 4 + 15

0	10011	000101

For Round-to-Nearest we look at the G,R,S bits and see that we should round down to just 1.000101

FP Addition/Subtraction 2

- **Convert subtraction to addition**

0	10000	010110

$= +1.010110 \times 2^{1}$

\-

0	01110	110101

$= 1.110101 \times 2^{-1}$

0	10000	010110

$= +1.010110 \times 2^{1}$

\+

1	01110	110101

$= -1.110101 \times 2^{-1}$

Smaller exponent, shift right

FP Addition/Subtraction 2

- **Shift the number with the smaller exponent to the right until exponents are equal**

0	10000	010110

+

1	01110	110101

$= +1.010110\ \underset{G}{\underline{0}}\ \underset{R}{\underline{0}}\ \underset{S}{\underline{0}} \times 2^{1}$

$= -\ 1.110101\ \underset{G}{\underline{0}}\ \underset{R}{\underline{0}}\ \underset{S}{\underline{0}} \times 2^{-1}$

Smaller exponent, shift right

$= -\ 0.111010\ \underset{G}{\underline{1}}\ \underset{R}{\underline{0}}\ \underset{S}{\underline{0}} \times 2^{0}$

$= -\ 0.011101\ \underset{G}{\underline{0}}\ \underset{R}{\underline{1}}\ \underset{S}{\underline{0}} \times 2^{1}$

FP Addition/Subtraction 2

- Since |A|>|B|, just subtract |A| - |B|
  - Use normal 2's complement as if binary point is not there

0	10000	010110	+	1	01110	110101

$= +1.010110\ \underset{G}{\underline{0}}\ \underset{R}{\underline{0}}\ \underset{S}{\underline{0}} \times 2^1$

$= -0.011101\ \underset{G}{\underline{0}}\ \underset{R}{\underline{1}}\ \underset{S}{\underline{0}} \times 2^1$

For subtraction, throw away the carry (for addition, keep it)f

$$\begin{array}{rl} & 1.010110000 \times 2^1 \\ - & 0.011101010 \times 2^1 \\ \hline \end{array} \quad \xrightarrow{\text{2's comp.}} \quad \begin{array}{rl} \cancel{1} & \\ & 1.010110000 \times 2^1 \\ + & 1.100010110 \times 2^1 \\ \hline & 0.111000110 \times 2^1 \end{array}$$

FP Addition/Subtraction 2

- Normalize and truncate the guard bits

0	10000	010110

+

1	01110	110101

$= 1.010110 \times 2^1$

$= .01110101 \times 2^1$

```
 ~~1~~                   GRS
       1.010110000   x 2^1
 +     1.100010110   x 2^1
 --------------------
       0.111000110   x 2^1
       1.110001100   x 2^0
 +     0.000001      x 2^0
 --------------------
       1.110010000   x 2^0
```

Move binary point after first '1'

E' = 0 + 15

0	01111	110010

For Round-to-Nearest we look at the G,R,S bits and see that since the result is halfway between the 2 round values, we pick the value with 0 in the LSB. Thus, we round up

FP Addition/Subtraction Example 3

1	10100	011010

+

0	10100	110100

$= -1.011010 \times 2^5$

$= +1.110100 \times 2^5$

Subtract smaller magnitude from larger and use sign of larger magnitude for the result

$$\begin{array}{rl} & 1.110100000 \times 2^5 \\ - & 1.011010000 \times 2^5 \\ \hline \end{array}$$

2's comp. ⇒

$$\begin{array}{rl} & \cancel{1} \\ & 1.110100000 \times 2^5 \\ + & 0.100110000 \times 2^5 \\ \hline & 0.011010000 \times 2^5 \\ & 0.011010000 \times 2^3 \end{array}$$

=

0	10010	101000

FP Multiplication / Division

Multiplication: Multiply fractions and add exponents

$$3.45 \times 10^{4} \quad * \quad 4.90 \times 10^{1}$$

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

Division: Divide fractions and subtract exponents

$$3.45 \times 10^{4} \quad \div \quad 4.90 \times 10^{1}$$

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

FP Multiplication

1. Determine sign
2. Add the exponents and subtract the Excess value (127 or 15)
3. Multiply the fractions
4. Normalize and round the resulting value

FP Multiplication

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

0	10000	010110

$= 1.010110 \times 2^1$

*

0	10011	110101

$= 1.110101 \times 2^4$

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

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

Or simply look at final exp. Value you need to represent, i.e., 5 and write the Excess-15 of 5 which is 010100

FP Multiplication

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

0	10000	010110

*

0	10011	110101

$= 1.010110 \times 2^1$

$= 1.110101 \times 2^4$

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

*

0	10011	110101

$= 1.010110 \times 2^1$

$= 1.110101 \times 2^4$

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

FP Multiplication

- **Normalize and truncate guard bits**

0	10000	010110

$*$

0	10011	110101

$= 1.010110 \times 2^1$

$= 1.110101 \times 2^4$

Exponent `10100 = 2^5`

fraction `10.011101001110`

Sign `pos. * pos. = pos.`

$1.0011101001110 \times 2^6$

$10.011101001110 \times 2^5$

GRS

1.001110101×2^6

1.001111×2^6

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

$= 1.010110 \times 2^1$

$= 2.6875$

$= 1.110101 \times 2^4$

$= 29.25$

$= 1.001111 \times 2^6$

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

$= 1.110100 \times 2^4$

/

0	10000	110000

$= 1.110000 \times 2^1$

```
  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

$= 1.110100 \times 2^4$

$= 1.110000 \times 2^1$

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

$= 1.110100 \times 2^4$

$= 1.110000 \times 2^1$

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

$= 1.110100 \times 2^4$

$= 1.110000 \times 2^1$

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

FP Division

- **Normalize and truncate guard bits**

0	10011	110100

= 1.110100×2^4

/

0	10000	110000

= 1.110000×2^1

Exponent	010010 = 2^3
fraction	1.000010011
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

$= 1.110100 \times 2^4$

$= 29$

$= 1.110000 \times 2^1$

$= 3.5$

$= 1.000010 \times 2^3$

Computed result $= 8.25$

True result $= 8.2857$

Error $= -0.0357$

Floating-Point Exceptions

- Error conditions that can be trapped (recognized by the HW) and passed to SW to deal with
  - Underflow – Result is too small to be represented as a normalized FP value (i.e. exponent is smaller than smallest possible)
  - Overflow – Result is too large to be represented
  - Inexact – Rounding has occurred
  - Invalid – Result is NaN
  - Divide-by-Zero – Just like it sounds (if not trapped, infinity is returned)

Intel FPU Exception Handling

- Control 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)

15–13	12	11 10	9 8	7	6	5	4	3	2	1	0
0 0 0	IC	RC	PC	IEM	0	PM	UM	OM	ZM	DM	IM

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

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 rounding and special values like Inf.

(-max val + max_val) + 1 ≠ -max_val + (max_val + 1)

(0) + 1 ≠ -max_val + (+inf.)

1 ≠ (inf.)