

```
#####
#           cs315 Week 9
#
#   -> Floating point representation & addition & subtraction
#
#####
```

Floating-point numbers  
\* used to represent real values  
\* have finite precision

--> single-precision - 32 bits (1 word)  
\* 1 bit sign  
\* 8 bit exponent  
\* 23 bit magnitude

--> double-precision - 64 bits (2 words)  
\* 1 bit sign  
\* 11 bit exponent  
\* 52 bit magnitude

~ doubles require two registers or two memory locations  
~ doubles MUST start in an even numbered register  
~ doubles occupy both the even register and the following odd register

Example:  
li.d \$f4, 3.14 stores a double in both \$f4 and \$f5

registers may be written with the 'concatenation' bar to denote they are part of a double

Example:  
li.d \$f4, 3.14 # \$f4\f5 <-> 3.14

Floating-point standard (FPS) will be discussed more in lecture. See appendix F in MIPS book for floating-point instructions  
--> NOTE: appendix F may have typos

FPS consist of 3 parts:

1) sign  
2) exponent  
3) magnitude  
--> sign, exponent, and magnitude (which includes hidden bit)

FPS format (this is the format we use to represent floating-point numbers):

!-- sign --!-- magnitude --!-- (hidden bit) --!-- fraction--!  
 ^ ^ ^ ^  
 1 bit unsigned binary 1 bit unsigned binary  
 (8 bits) (7 bits)

We should note that:

!-- sign --!-- (hidden bit) --!-- fraction --!<<<<< sign magnitude (9 bits)  
 ^  
 1 bit

Definition of mantissa:

Mantissa = (hidden bit) + fraction <<<<< 8 bits mantissa in FPS representation (or simply class representation of floating-point numbers)  
 ^ ^  
 1 bit 7 bits

Sign bit:

- \* is only 1 bit
- \* 0 for positive, 1 for negative (similar to sign magnitude number system)

Exponent:

- \* 8 bits for 32 bit IEEE 754 standard
- \* 8 bits for the FPS we will use in class
- \* bias 127
  - > bias exponent = actual exponent + 127
  - > actual exponent = bias exponent - 127

Example:

if FPS exponent (bias exponent): 131, then actual exponent is: 4

- \* bias is used to allow negative exponents

Example:

if FPS exponent (bias exponent): 125, then actual exponent is: -2

Magnitude:

- \* 23 bits for 32 bit IEEE 754 standard
- \* 7 bits for the FPS we will use in class (which includes hidden bit)
- \* shortened to 7 for brevity, but the process used for these 7 is the same processed used for all 23
- \* hidden bit will always be a 1
- \* not represented in hardware ('hidden')
  - > less circuitry, therefore less material cost and less energy use
  - > we represent numbers in normal format:
    - i.e. 1.001
    - ^

this is hidden bit (always 1), we show it in parenthesis

- \* used in the arithmetic we will be doing
- \* show in very clear parenthesis

Normalizing:

- \* normal form: 1.\_\_\_\_\_
- Example:
- 1.23456  
3.14151

#####
#####

\* Precision

- > precision will be lost when using FPS
  - unavoidable trade-off for using a finite number of bits
- > lost when converting, lost when shifting values, etc.
  - converting -7.4 to 8-bit binary then back to decimal demonstrates loss of precision
  - >>> -7.325 is the closest we get to -7.4

\* FPS addition of same sign

- 1) convert both values to binary
- 2) normalize both values
- 3) match lower exponent to higher exponent (raise exponent n, right-shift n bits)
- 4) add magnitudes, keep overflow carry bit
- 5) normalize result if necessary
- 6) set result sign to same sign as operands

\* FPS addition of different signs

- 1) convert both values to binary
- 2) normalize both values
- 3) match lower exponent to higher exponent

- 4) convert sign to 10th bit (+ = 0, - = 1, is used to handle signed overflow)
  - 5) convert both values to 2's complement
  - 6) add magnitudes (as 2's complement)
  - 7) convert result to sign-magnitude
  - 8) convert 10th bit to sign
  - 9) normalize if necessary

#####
Example 1:

$$\begin{array}{r}
 & 3.125 \\
 + & 2.500 \\
 \hline
 & 5.625 \text{ (expected)}
 \end{array}$$

3 => 11

0.125 |  
0.250 | 0  
0.500 | 0  
0.000 | 1

001 => .625

$$3.125 \Rightarrow 11.001 \Rightarrow 1.1001 * 2^4$$

^  
normalize

$0 - 10000000$  (1)  $1001000 \leq 3.125$   
 sign exponent ^ fraction  
 ^  
 hidden bit (we show it in)

2 -> 10

$$\begin{array}{r} 0.500 \\ 0.000 \end{array} \begin{array}{l} | \\ 1 \end{array} \begin{array}{r} 0.500 \\ 0.000 \end{array}$$

$2.500 \Rightarrow 10.1 \Rightarrow 1.01 * 2^1$   
  ^  
    normalize

$0 - 10000000$  (1) 01000000 <= 2.500  
 sign exponent ^ fraction  
 ^  
 hidden bit (we show it in)

Essentially:

$$1.1001 * 2^1 + 1.01 * 2^1 = 2^1 * (1.1001 + 1.01)$$

exponents are the same, so there is no need to match exponents.

+ 3.125 => 0 - 100000000 (1) 1001000  
+ 2.500 => 0 - 100000000 (1) 0100000

0 - 100000001 (1) 0110100 => 1.0110100 \* 2^2 = 101.10100 \* 2^0 = 101.10100 => + 5.625 (correct result)

sign extend to 10 bits when two numbers are the same sign (because magnitude is will get larger when we add two numbers with the same sign):  
 --> so more bits might be needed to represent the result of addition

```

    00 (1) 1001000
+ 00 (1) 0100000
-----
    0 1 (0) 1101000 -> (1) 0110100 <<< normalize by shift right 1 time which is essentially divide by 2 <<< thus we need to compensate division
    ^                                by 2, with adding exponent by 1
    sign of the result is positive
#####

```

Example 2:

```

3.125
-
2.500
-----
0.625 (expected)

3.125 => 0 - 10000000 (1) 1001000
-
2.500 => 0 - 10000000 (1) 0100000
-----

```

sign extend to 10 bits when two numbers are different sign (we could sign extend it to 9 bits because we are subtracting two numbers and magnitude is getting smaller, but lets be consistent):

Essentially:

```

1.1001 * 2^1 - 1.01 * 2^1 = 2^1 * (1.1001 - 1.01)
    ^
    exponents are the same, so there is no need to match exponents

```

```

00 (1) 1001000 rewrite the number      -> 00 (1) 1001000
- 00 (1) 0100000 find additive inverse   -> 11 (0) 1100000
-----
          |   |
          00 (0) 0101000 -> (1) 0100000 <<< normalize by shift left 2 times which is essentially multiply by 2^2
          ^                                , thus we need to compensate multiply 2^2 by subtracting -2 from exponent
          ^
          11 (0) 1011111
+
1           sign of the result is positive which was expected (however, it is not in normal form)
-----
11 (0) 1100000

```

```

0 - 01111110 (1) 0100000 => 1.0100000 * 2^-1 = 0.1010000 => + 0.625 (correct result)
    ^
    ^

```

1 (or biased exponent of: 128) - 2 = -1 <<< biased exponent of 126

#####
Example 3:

```

(-3.125)
+
2.500
-----
-0.625 (expected)

(-3.125) => 1 - 10000000 (1) 1001000
+
2.500     => 0 - 10000000 (1) 0100000
-----

```

sign extend to 10 bits when two numbers are different sign (we could sign extend it to 9 bits because we are subtracting two numbers and magnitude is getting smaller, but lets be consistent):

Essentially:

$$-1.1001 * 2^1 + 1.01 * 2^1 = 2^1 * (-1.1001 + 1.01)$$

^  
exponents are the same, so there is no need to match exponents

```
|-----|  
|  
| 1 (1) 1001000  find additive inverse  -> 11 (0) 0111000  
| + 0 (1) 0100000  rewrite the number   -> 00 (1) 0100000  
|-----|  
|  
| sign           11 (1) 1011000 -> number is negative here, convert it back to sign magnitude (|- sign |- magnitude -|)  
|  
| sign of the result is negative  
}  
|----> 10 (1) 1001000 <-> sign magnitude number  
|    11 (0) 0110111  
+  
-----  
11 (0) 0111000
```

11 (1) 1011000 --> convert result back to sign magnitude (positive) --> 00 (0) 0101000 -> (1) 0100000 <<< normalized by shift left 2 times, which is  
essentially multiply by 2, thus we need to  
compensate multiplication by 2 with subtracting  
exponent with 2

1 - 01111110 (1) 0100000 => 1.0100000 \* 2^-1 = 0.1010000 = - 0.625 (correct result)  
^  
^  
^

1 (or biased exponent of: 128) - 2 = -1 --> 01111110 <<< biased exponent of 126

```
#####
Example 4:
```

27 => 11011

```
0.1251  
0.2501 0  
0.5001 0  
0.0001 1  
-----  
001 => .625
```

27.125 => 11011.001 => 1.1011001 \* 2^4 (biased exponent = 4 + 127 = 131 <<< 10000011)  
^ ^  
normalize fraction

7 => 111

```
0.5001  
0.0001 1  
-----  
1 => 0.500
```

7.5 => 111.1 => 1.111 \* 2^2 (biased exponent = 2 + 127 = 129 <<< 10000001)  
^ ^  
normalize ^  
^  
fraction

```
27.125  FPS = 0 - 10000011 (1) 1011001
+ 7.50   FPS = 0 - 10000001 (1) 1110000
-----
```

34.625 (expected)

```
0 - 10000011 (1) 1011001    rewrite FPS      --> 0 - 10000011 (1) 1011001
+ 0 - 10000001 (1) 1110000    match exponents --> 0 - 10000011 (0) 0111100
----- |           |           |
|           |           0 - 10000100 (1) 0001010 => 1.0001010 * 2^5
|           |           = 100010.10 * 2^0
|           |           = 100010.10
|           |           = +34.5 (correct result)
10000001 => 129 + 2 (to match) <----|
^
```

smaller exponent, we need to match it with larger exponent. We matched the exponent, but we also need to compensate for adding exponent with 2 thus, shift right 2 times (divide by 2)

add two positive numbers, without exponents

```
00 (1) 1011001
+ 00 (0) 0111100
-----
```

01 (0) 0010101 --> (1) 0001010

^

sign of result is positive

```
#####
Example 5:
```

exponents are matched <-|

```
|           |
|           |
29.877  FPS = 0 - 10000011 (1) 1101111
- 23.62   FPS = 0 - 10000011 (1) 0111100
-----
```

6.257 (expected)

```
-----|           |
|           |
|           0 - 10000011 (1) 1101111
| - 0 - 10000011 (1) 0111100
-----|           |
|           0 - 10000001 (1) 1001100 => 1.1001100 * 2^2
|           = 110.01100 * 2^0
|--> 131 (biased)          = 110.01100
so true exponent:          = +6.375 (correct result)
* 131 - 127 = 4
```

----- add a positive numbers with additive inverse of second number { $a - b = a + (-b)$ }, without exponents

```
|           |
| -> (1) 1101111    rewrite the number (convert from sign magnitude to 2's complement)      --> 00 (1) 1101111
+ (1) 0111100    calculate additive inverse                                         --> 11 (0) 1000100
-----
```

```
00 (0) 0110011 --> (1) 1001100
^           ^
```

sign of result is positive

we shifted left 2 times to  
normalize the result, so we  
need to compensate it by  
subtracting exponent with 2  
thus, result exponent:  
 $131 - 2 = 129$

^

which is: 10000001

```
#####
#####
```

Example 6:

```
(- 63.874)  FPS = 1 - 10000100 (1) 1111111
- (152.69)  FPS = 0 - 10000110 (1) 0011000
-----
-216.564 (expected)
```

```
remember, to match exponent, we <-----|
match smaller one with larger one      |
and don't forget shifting to          |
compensate for adjusting exponent     |
|
1 - 10000100 (1) 1111111      match exponents --> 1 - 10000110 (0) 0111111
- 0 - 10000110 (1) 0011000      rewrite FPS      --> 0 - 10000110 (1) 0011000
-----
```

-> then find additive inverse of both numbers, add them up two number:

```
(0) 0111111 -->    11 (1) 1000001
+ (1) 0011000 --> + 11 (0) 1101000
-----
(1) 1010111      11 (0) 0101001 --> -215
^
sign of the result should be negative
```

now find the unsigned representation of the result:

```
11 (0) 0101001 <- 2's complement
00 (1) 1010111 <- unsigned binary <<< normal form already, so we don't do anything with it
```

```
1 - 10000110 (1) 1010111 => - 1.1010111 * 2^7 = - 11010111 * 2^0 = - 11010111 = -215 (correct result)
```

```
#####
#####
```

Example 7:

```
5153.475    FPS = 0 - 10001011 (1) 0100001
- 49.875     FPS = 0 - 10000100 (1) 1000111
-----
5103.6 (expected)
```

```
0 - 10001011 (1) 0100001    rewrite FPS      --> 0 - 10001011 (1) 0100001
- 0 - 10000100 (1) 1000111    match exponents --> 0 - 10001011 (0) 0000001
-----
```

--> find additive inverse of second number

```
(1) 0100001 -->    00 (1) 0100001
- (0) 0000001 --> + 11 (1) 1111111
-----
00 (1) 0100000 --> (1) 0100000, therefore:-> 0 - 10001011 (1) 0100000
^
```

sign of the result should be positive, also number is already in normal form, so we don't do anything with it

```
0 - 10001011 (1) 0100000 => + 1.0100000 * 2^12 = 1010000000000 * 2^0 = + 1010000000000 = +5120 (correct result)
```

```
#####
#####
```