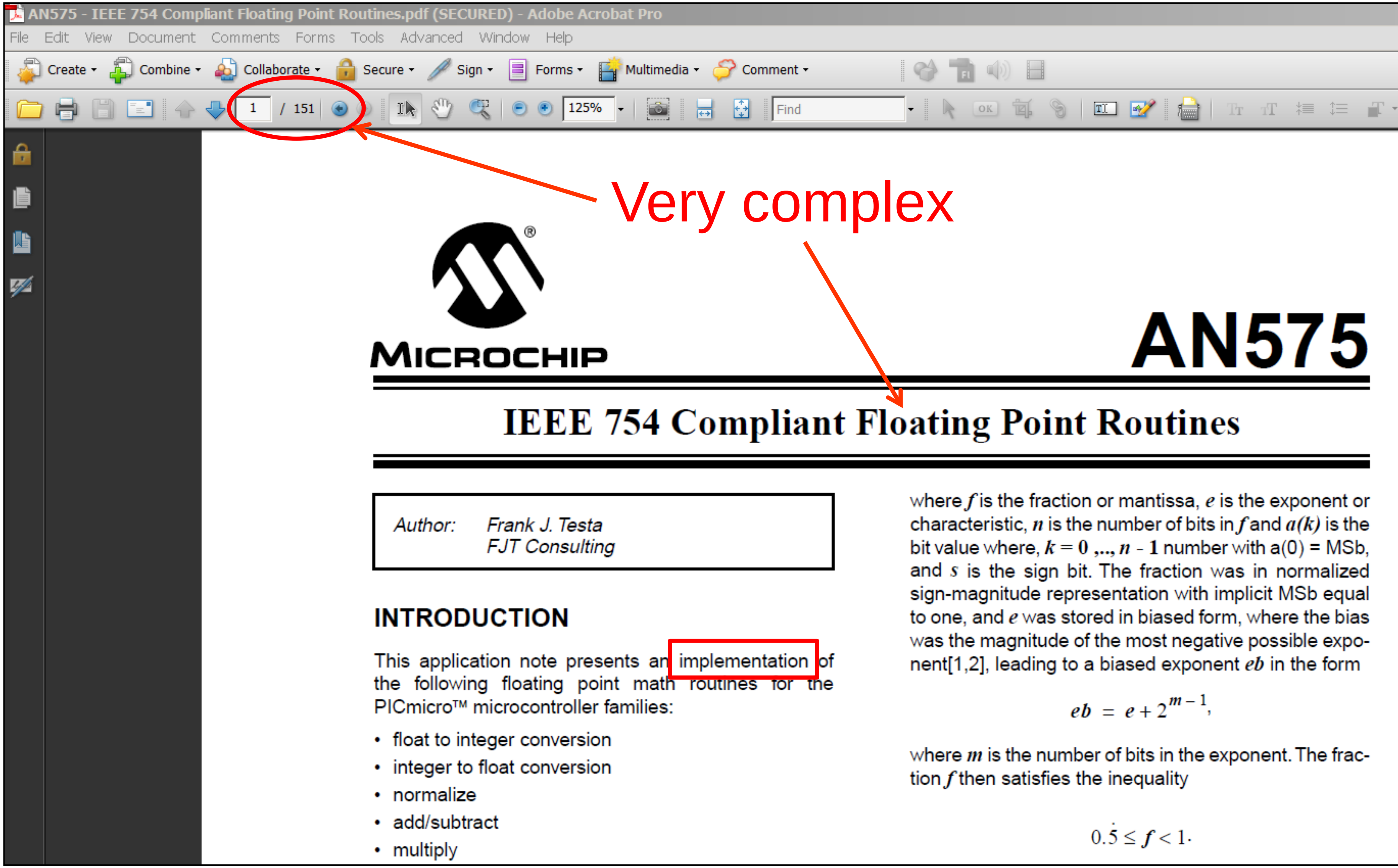
EEC 417/517
Embedded Systems
Cleveland State University

Lab 11
Floating-Point Arithmetic

Dan Simon
Rick Rarick

Spring 2018

Floating-Point Arithmetic



IEEE 754 Compliant Floating Point Routines

Author: Frank J. Testa
FJT Consulting

INTRODUCTION

This application note presents an implementation of the following floating point math routines for the PICmicro™ microcontroller families:

- float to integer conversion
- integer to float conversion
- normalize
- add/subtract
- multiply

where f is the fraction or mantissa, e is the exponent or characteristic, n is the number of bits in f and $a(k)$ is the bit value where, $k = 0 ,..., n - 1$ number with a(0) = MSb, and s is the sign bit. The fraction was in normalized sign-magnitude representation with implicit MSb equal to one, and e was stored in biased form, where the bias was the magnitude of the most negative possible exponent[1,2], leading to a biased exponent eb in the form

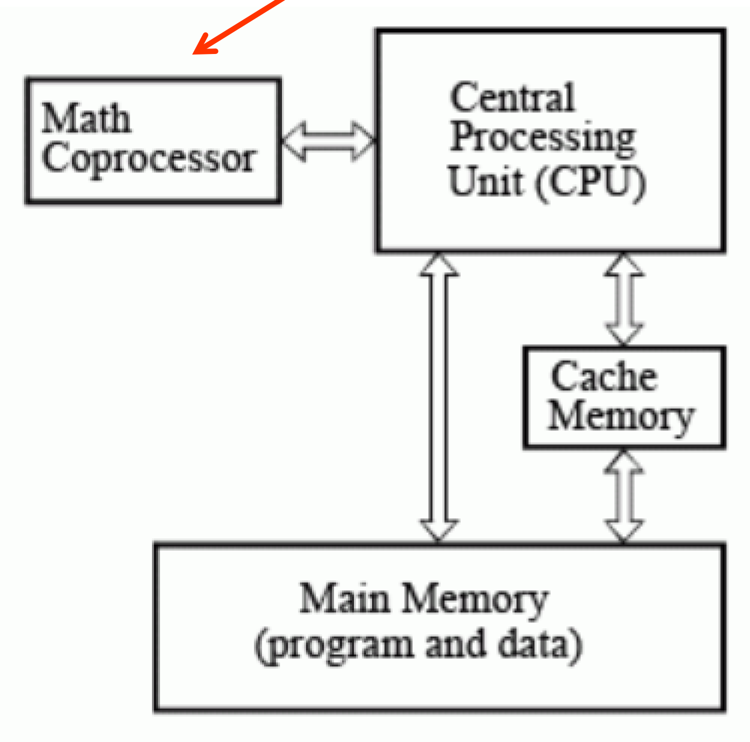
$$eb = e + 2^{m-1},$$

where m is the number of bits in the exponent. The fraction f then satisfies the inequality

$$0.\dot{5} \le f < 1.$$

Floating-Point Arithmetic

Originally implemented
in a Math Coprocessor
(or Floating-Point Unit)



Now integrated into CPU

Floating-Point Arithmetic

1. Computing devices store numbers as binary symbols (digits) in memory registers: 1001 0101

2. The meaning of the symbols depends entirely on the **interpretation** assigned to the symbols.

3. The most common interpretations (or formats) are the **fixed-point** and **floating-point** formats.

4. Many other formats exist and are used in various applications.

Floating-Point Representation

1. The algebraic form of a number x in scientific notation is

$$x = m \times 10^e$$

m is called the **mantissa** (or significand), and ***e*** is called the **exponent**.

2. If $1 \le m < 10$, then $x = m \times 10^e$ is said to be in **normalized scientific notation**.

3. This means that ***m*** lies in the interval

$$1.00000\ldots \le m < 9.99999\ldots \longrightarrow \boxed{\begin{array}{l} 1.356 \times 10^2 \\ 4.0093 \times 10^5 \\ 9.99 \times 10^{-3} \\ \xcancel{0.539 \times 10^3} \end{array}}$$

so the first digit of m cannot be '0'.

4. Exception: $x = 0$ if and only if $m = 0$

Floating-Point Representation

1. The algebraic form of a Base-2 number in floating-point is

$$x = f \times 2^e$$

   where $\boldsymbol{f}$ is a called the **fraction** (or significand or mantissa) and $\boldsymbol{e}$ is called the **exponent**.

2. If $0.1_2 \le f < 1$, then $x = f \times 2^e$ is said to be in Base-2 **normalized floating-point notation**. Note: in Base-10, this means that

$$0.5 \le f < 1 \quad (0.1_2 = 2^{-1} = 0.5)$$

3. In normalized form, f lies in the interval

$$\left(0.10000\ldots \le f < 0.11111\ldots\right)_2$$

   so the first digit after the binary point must be '1'.

4. Exception: $x = 0$ if and only if $f = 0$

Floating-Point Representation

Any number can be represented in normalized scientific notation by allowing the decimal point to "float":

$$8104.75 = 8.10475 \times 10^{3}$$

$$0.00147 = 1.47 \times 10^{-3}$$

Similarly, any number can be represented in normalized Base-2 notation by allowing the binary point to "float":

$$1101.01_2 = 0.110101_2 \times 2^{4}$$

$$0.0010111_2 = 0.10111_2 \times 2^{-2}$$

Floating-Point Representation

1. In normalized floating-point notation, $0.1_2 \le f < 1$, but this can also be expressed in decimal representation as $0.5 \le f < 1$.

   Example: If $f = 0.1011_2$, then

$$f = 1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3} + 1\times 2^{-4}$$
$$= \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = 0.5 + 0.125 + 0.0625 = 0.6875$$

2. Note that in the algebraic representation $x = f \times 2^e$ of a Base-2 floating-point number, we have temporarily ignored the sign of the number.

3. The sign can be easily included :

positive: $s = 0$
negative: $s = 1$

$$x = (-1)^s f \times 2^e$$

Floating-Point Representation

1. How can the algebraic form $x = (-1)^s f \times 2^e$ be represented in computing machines?

2. In other words, how can $x = (-1)^s f \times 2^e$ be stored as 0's and 1's in memory registers?

3. We need a standard format so that programs written under the standard will run correctly on various machines.

4. The standard format (since about 1990) for representing Base-2 floating-point numbers in computing devices is the IEEE Standard for Floating-Point Arithmetic (IEEE 754).

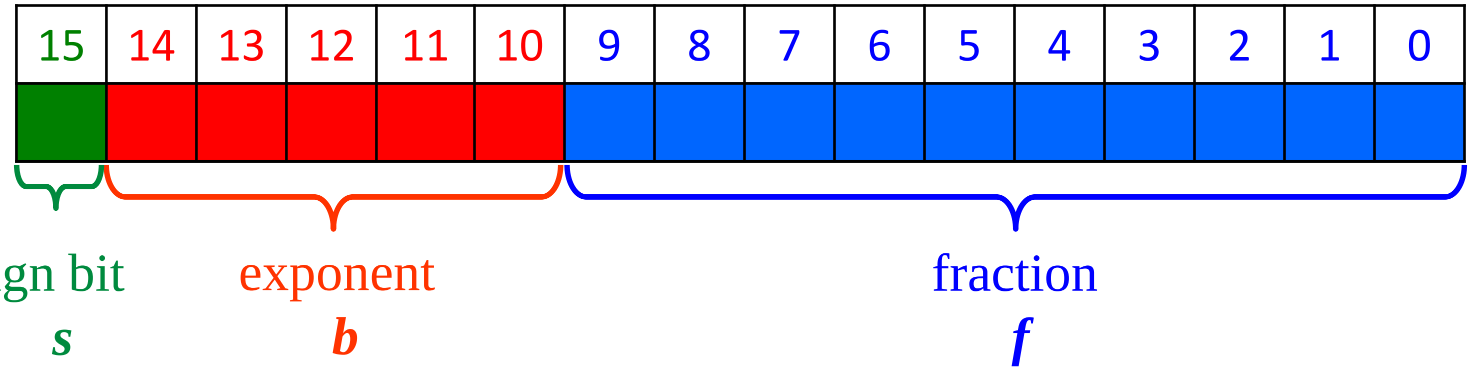
Floating-Point Representation

1. The IEEE 754 standard includes standards for 16, 32, 64, and 128-bit machines.

2. Example: 16-bit IEEE 754

1 is omitted because it occurs in all floating point representations

$$x = (-1)^s \left(1 + f\right) \times 2^e$$



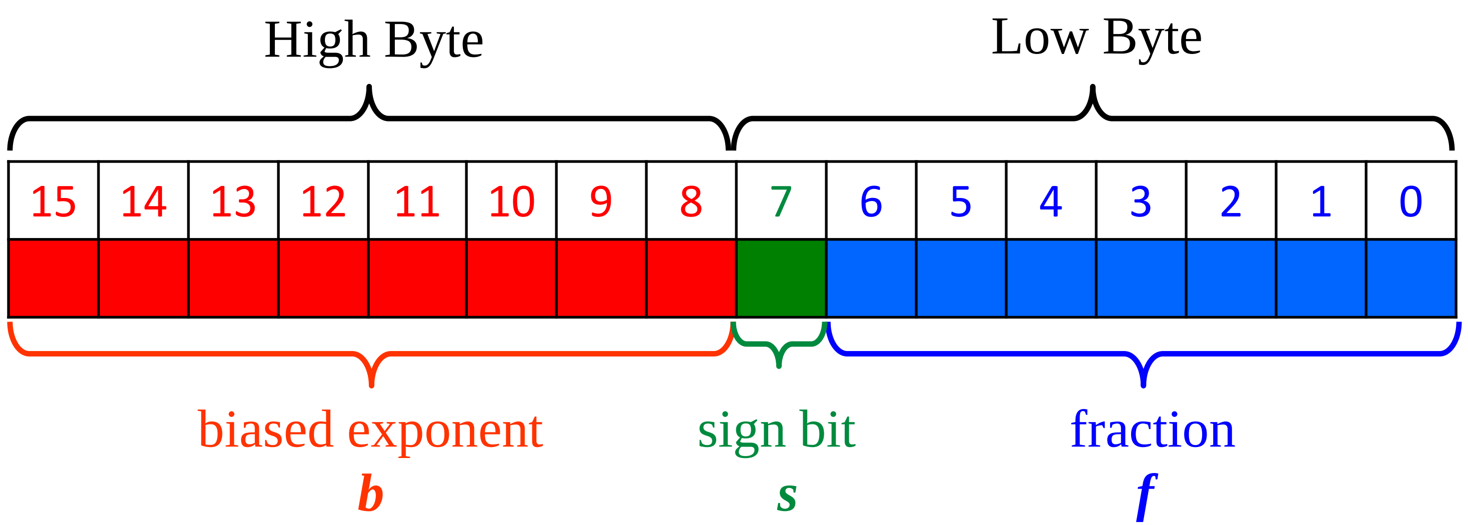
3. The exponent ***b*** is called a **biased** or **offset** exponent and is defined below.

Floating-Point Representation

We will look at a **modified** 16-bit IEEE 754 format which is defined in Microchip's Application Note 575.

Algebraic representation: $x = (-1)^s f \times 2^e$

Modified Floating-Point Representation



where $b = e + 128$

Modified Floating-Point

1. An 8-bit exponent register stores unsigned integers in the range,

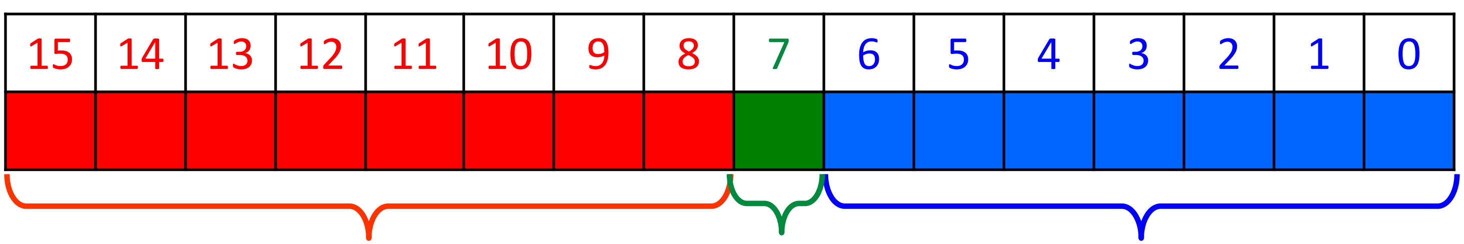
$$b \in [0,\ 255].$$

2. But we want the range of exponents to include positive and negative values:

$$e \in [-128,\ 127].$$

3. So we **interpret** the exponent to be **biased** by 128.

$$x = (-1)^s\, f \times 2^e$$



biased exponent
$\boldsymbol{b} = \boldsymbol{e} + 128$

sign bit
$\boldsymbol{s}$

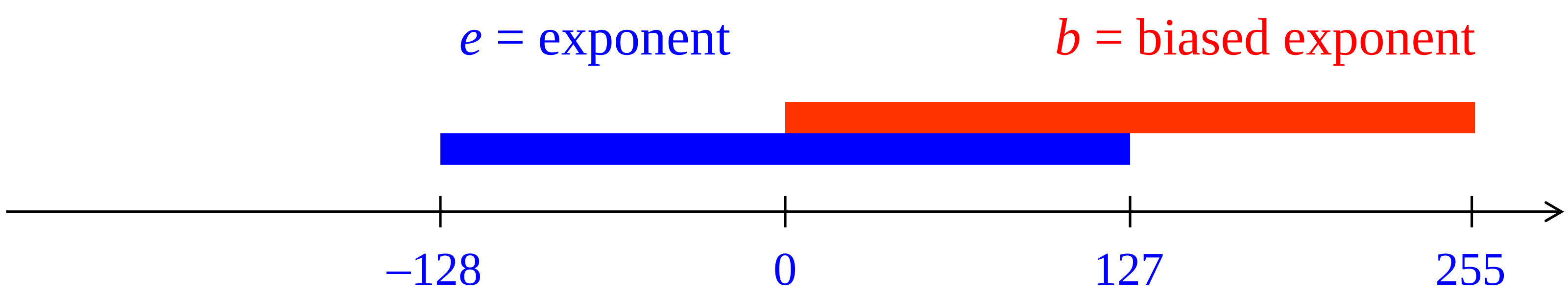
fraction
$\boldsymbol{f}$

Biased Exponent

	Biased Exponent	Exponent
Binary	**b = e + 128**	**e = b - 128**
00000000	0	-128
00000001	1	-127
00000010	2	-126
01111111	127	-1
10000000	128	0
10000001	129	1
11111101	253	125
11111110	254	126
11111111	255	127

Floating-Point Biased Exponent

$$\boxed{x = (-1)^s\, f \times 2^e, \qquad -128 \le e \le 127}$$



$$\boxed{\text{Define } b = e + 128}$$

$$\boxed{x = (-1)^s\, f \times 2^{b-128}, \qquad 0 \le b \le 255}$$

Modified Floating-Point

algebraic representation

$$\boxed{x = (-1)^s\, f \times 2^e}$$

$$-128 \le e \le 127$$

computer floating-point representation

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	s	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	s	f_1	f_2	f_3	f_4	f_5	f_6	f_7

b = biased exponent

$$\boldsymbol{b} = \boldsymbol{e} + 128$$

$$0 \le b \le 255$$

$$b_i \in \{0, 1\}$$

s = sign bit

$$s \in \{0, 1\}$$

f = fraction

$$0.1_2 \le f < 1$$

$$(\text{ if } x \ne 0 \ )$$

$$f_i \in \{0, 1\}$$

Modified Floating-Point

Example: decimal $\rightarrow$ computer floating-point $\rightarrow$ hex

$$x = -11.125$$

$$x = -11.125 = (-1)^1(1011.001_2) = (-1)^1(0.1011001_2)\times 2^4 = (-1)^s f \times 2^e$$

$$\boxed{\begin{aligned} &e = 4 \\ &b = e + 128 = 132 = 1000\ \ 0100_2 \\ &s = 1 \\ &f = 0.101\ 1001_2 \end{aligned}}$$

$$x \rightarrow \underbrace{1000\ \ 0100}_{b}\ \underset{\uparrow s}{1}\underbrace{101\ \ 1001}_{f} = \text{0x84D9}$$

Modified Floating-Point

Example: hex $\rightarrow$ computer floating-point $\rightarrow$ decimal

$$x \rightarrow 0x8358 = \underbrace{1000\ 0011}_{b}\ \underset{\uparrow s}{0}\ \underbrace{101\ 1000}_{f}$$

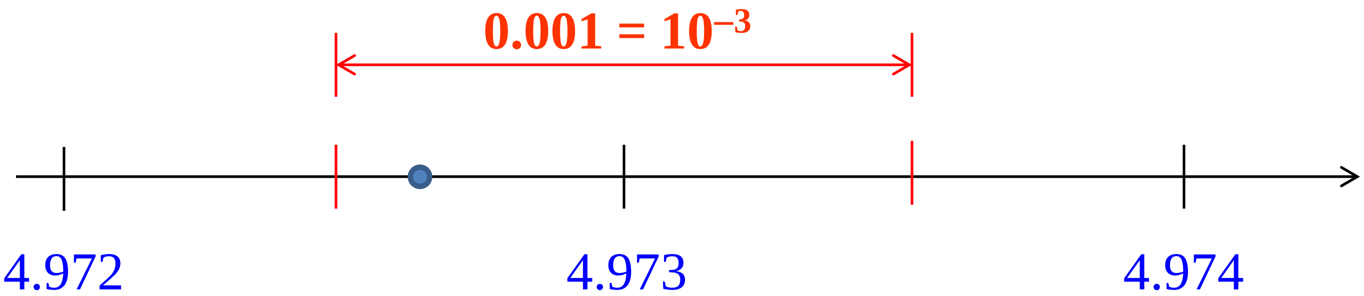
$$\boxed{\begin{aligned} &b = 1000\ \ 0011_2 = 130 \\ &e = b - 128 = 2 \\ &s = 0 \\ &f = 0.1011_2 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = 0.6875 \end{aligned}}$$

$$x = (-1)^s f \times 2^e = (-1)^0 (0.6875)(2^2) = 2.75$$

Note that the symbol f is **overloaded**: $f = 0.1011000_2 = 101\ 1000$

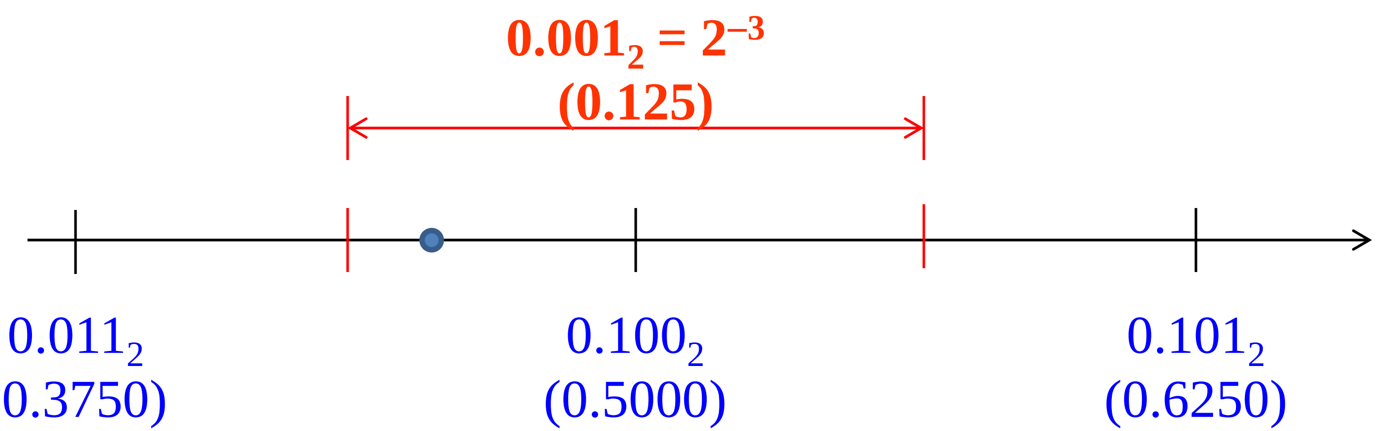
Floating-Point Precision (Round-off Error)

Decimal rounding



$$\text{Precision} = \pm(1/2)\left(10^{-3}\right) = \pm 0.0005$$
$$= \pm(1/2)\left(10^{-\text{number of decimal places}}\right)$$

Binary rounding



$$\text{Precision} = \pm(1/2)\left(2^{-3}\right) = \pm 0.0625$$
$$= \pm(1/2)\left(2^{-\text{number of binary places}}\right)$$

Floating-Point Precision

The representation of a floating point number is **not** unique:

$$0.5 = 0.1_2 \times 2^0 \quad \Rightarrow \quad \text{Normalized floating-point}$$

$$0.5 = 0.0000001_2 \times 2^6 \quad \Rightarrow \quad \text{Non-normalized floating-point}$$

Example: Convert the above two representations of 0.5 to Modified Floating-Point :

Floating-Point Precision

Normalized: $x = 0.1_2 \times 2^0 = (-1)^s f \times 2^e$

$s = 0,\;\; f = 0.1_2 = 2^{-1} \quad \text{and} \quad e = 0 \;\Rightarrow\; b = e + 128 = 128$

$x \rightarrow \underbrace{1000\ 0000}_{b}\ \underset{\uparrow s}{0}\underbrace{100\ 0000}_{f} = \text{0x8040}$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	s	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

b f (7-bits)

$$\boxed{x = f \times 2^e = (2^{-1} \pm 2^{-8}) \times 2^0 = 0.5 \pm 0.0039}$$

Floating-Point Precision

Non-normalized: $x = (-1)^s f \times 2^e = 0.0000001_2 \times 2^6$

$s = 0, \quad f = 0.0000001_2 = 2^{-7}$ and $e = 6 \;\Rightarrow\; b = e + 128 = 134$

$$x \to \underbrace{1000\ 0110}_{b}\ \underset{\uparrow s}{0}\underbrace{000\ 0001}_{f} = \text{0x8601}$$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	s	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1

b *f*

$$\boxed{x = f \times 2^e = (2^{-7} \pm 2^{-8}) \times 2^6 = 0.5 \pm 0.25}$$

Floating-Point Precision

1. $0.5 \rightarrow$ 0x8040 $= 0.5 \pm 0.0039$ (normalized)

2. $0.5 \rightarrow$ 0x8601 $= 0.5 \pm 0.25$ (non-normalized)

3. Normalized numbers ($f_1 = 1$ = first digit to right of binary point) are the most precise floating-point numbers.

4. Therefore, the normalized representation (0x8040 in the above example) is always used, and the representation is then unique.

5. Normalization:

   a) Rotate the fraction register one bit to the left until $f_1 = 1$
   b) Subtract one from the exponent with each rotation

Floating Point Range

Range for normalized floating point:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	s	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

p = smallest positive number

$\boxed{x = (-1)^s\, f \times 2^e}$

$b = 0$, $e = -128$, $f = 0.1_2 = 0.5$

$p = 0.5 \times 2^{-128} = 1.47 \times 10^{-39}$ (smaller for non-normalized, $f = 0.000\,0001_2$)

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	s	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}
1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1

P = largest positive number

$b = 255$, $e = 127$, $f = 0.111\,1111_2 = 0.9921875$

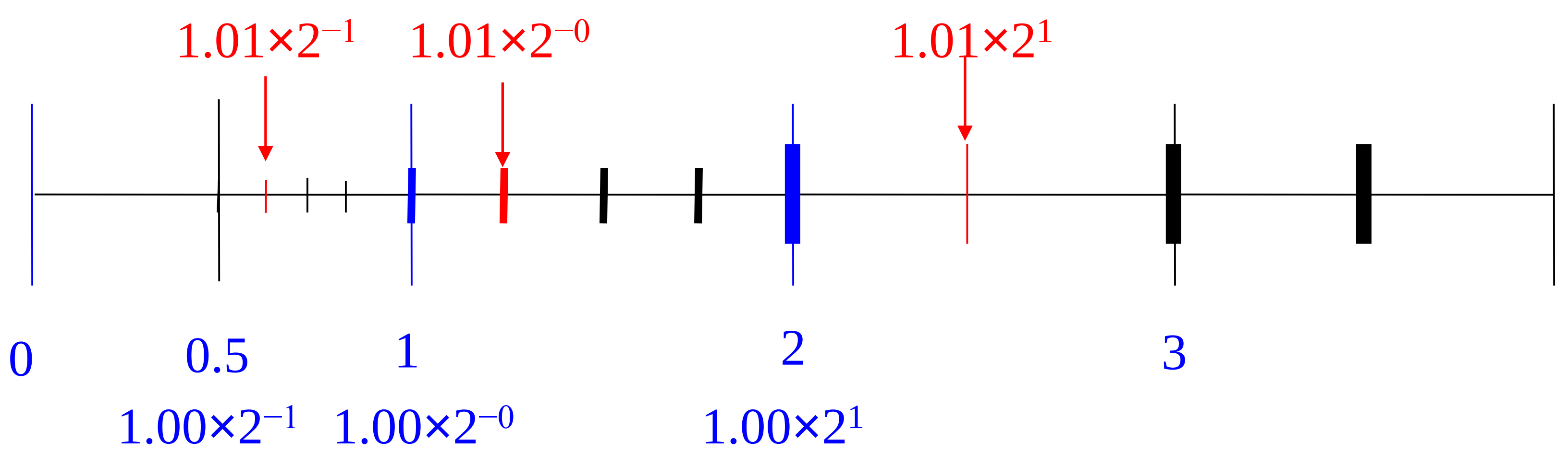
$P = 0.9921875 \times 2^{127} = 1.69 \times 10^{38}$

Distribution of Floating Point Numbers on the Real Number Line

- **3 bit mantissa**
- **exponent {-1,0,1}**

e = -1	e = 0	e = 1
1.00 X 2^(-1) = 1/2	1.00 X 2^0 = 1	1.00 X 2^1 = 2
1.01 X 2^(-1) = 5/8	1.01 X 2^0 = 5/4	1.01 X 2^1 = 5/2
1.10 X 2^(-1) = 3/4	1.10 X 2^0 = 3/2	1.10 X 2^1= 3
1.11 X 2^(-1) = 7/8	1.11 X 2^0 = 7/4	1.11 X 2^1 = 7/2

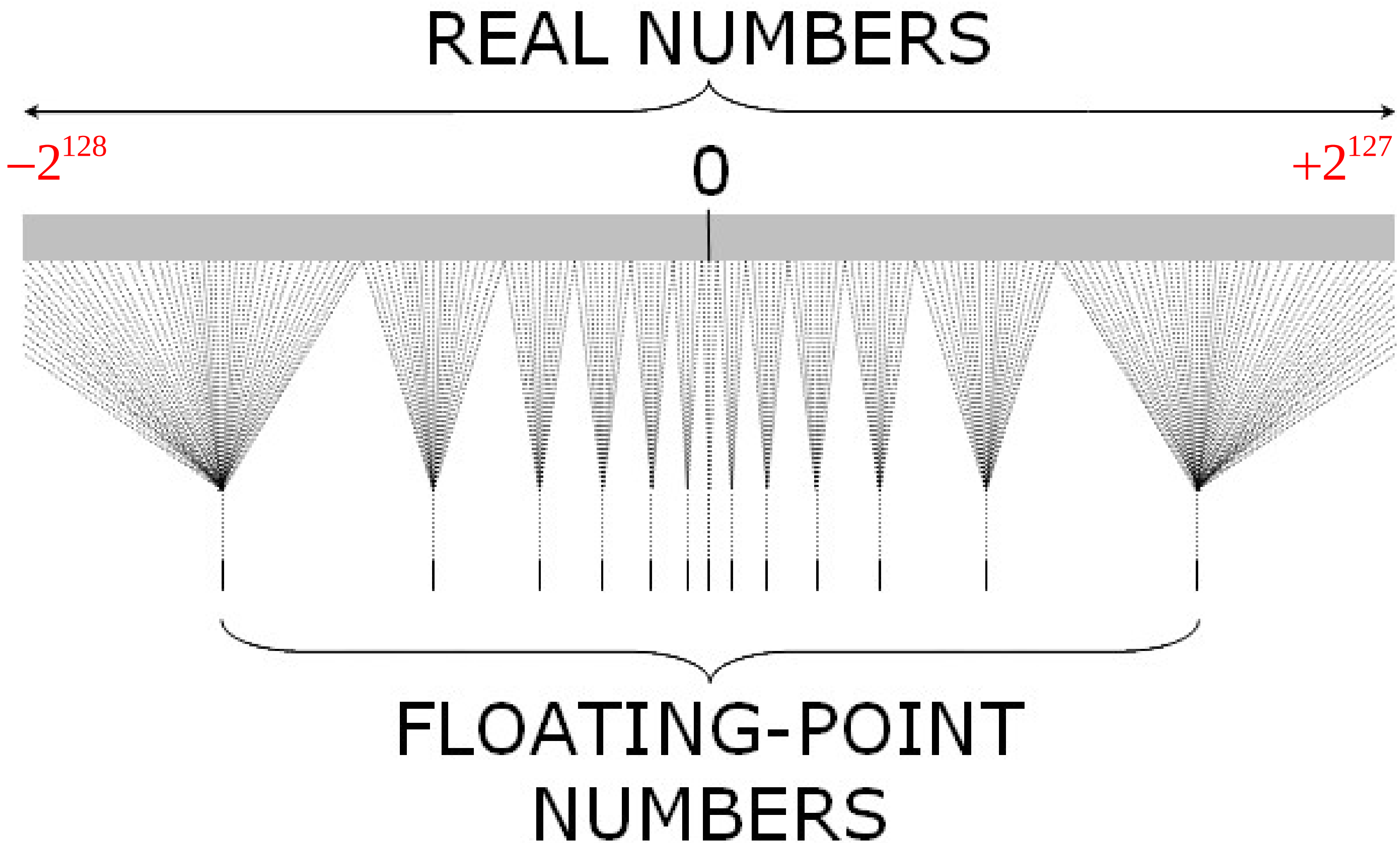
Non-uniformly distributed, round-off error varies over range.



round-off error small

round-off error large

Floating-Point Distribution



Non-uniformly distributed, round-off error varies over range.

Converting a Decimal Integer to a Binary Integer

The Division Algorithm can be used to convert
a decimal integer to a binary integer.

Division Algorithm: Dividend = Quotient $\times$ Divisor + Remainder
with Remainder < Divisor

Example: Convert 1379 to a binary integer.

We write 1379 in its Base-2 expansion starting with the LSB on the left:

$$1379 = \underbrace{a_0}_{\text{LSB}} 2^0 + a_1 2^1 + a_2 2^2 + a_3 2^3 + \cdots \qquad \text{where } a_i \in \{0, 1\}$$

We can determine the coefficients by repeatedly applying the Division Algorithm.

Converting a Decimal Integer to a Binary Integer

$$1379 = a_0 2^0 + a_1 2^1 + a_2 2^2 + a_3 2^3 + \cdots \qquad \text{where } a_i \in \{0, 1\}$$

1. **Divide both sides by two:**

Dividend = Quotient × Divisor + Remainder

$$1379 = 689 \times 2 + 1$$

$$1379/2 = 689 + 1/2$$

$$689 + 1/2 = a_0 \frac{2^0}{2} + a_1 \frac{2^1}{2} + a_2 \frac{2^2}{2} + a_3 \frac{2^3}{2} + \cdots$$

$$= \underbrace{a_0/2}_{1/2} + \underbrace{\left(a_1 2^0 + a_2 2^1 + a_3 2^2 + \cdots\right)}_{\text{integer}}$$

Since $a_0 \in \{0, 1\}$ and $\left(a_1 2^0 + a_2 2^1 + a_3 2^2 + \cdots\right)$ is an integer, we must have $a_0 = 1$. Subtracting 1/2 from both sides gives

$$689 = a_1 2^0 + a_2 2^1 + a_3 2^2 + \cdots$$

Converting a Decimal Integer to a Binary Integer

$$689 = a_1 2^0 + a_2 2^1 + a_3 2^2 + \cdots$$

2. **Divide both sides by two:**

$$689/2 = 344 + 1/2 = a_1/2 + \left(a_2 2^0 + a_3 2^1 + a_4 2^2 + \cdots\right)$$

So, $a_1 = 1$ and $344 = a_2 2^0 + a_3 2^1 + a_4 2^2 + \cdots$

3. **Divide both sides by two**:

$$344/2 = 172 + 0/2 = a_2/2 + \left(a_3 2^0 + a_4 2^1 + a_5 2^2 + \cdots\right)$$

So, $a_2 = 0$ and $172 = a_3 2^0 + a_4 2^1 + a_5 2^2 + \cdots$

Converting a Decimal Integer to a Binary Integer

4. **Divide both sides by two:**

$$172/2 = 86 + 0/2 = a_3/2 + \left(a_4 2^0 + a_5 2^1 + a_6 2^2 + \cdots\right)$$

So, $a_3 = 0$ and $86 = a_4 2^0 + a_5 2^1 + a_6 2^2 + \cdots$

5. **And so on until the quotient is zero.**

Converting a Decimal Integer to a Binary Integer

The algorithm can be tabularized:
Dividend = Quotient × Divisor + Remainder

Dividend	Quotient = Dividend ÷ 2	Remainder	Symbol
1379	689	1	a_0
689	344	1	a_1
344	172	0	a_2
172	86	0	a_3
86	43	0	a_4
43	21	1	a_5
21	10	1	a_6
10	5	0	a_7
5	2	1	a_8
2	1	0	a_9
1	0	1	a_{10}

LSB

MSB

$$a_{10} \cdots a_2 a_1 a_0$$

$$1379 = 101\ 0110\ 0011_2$$

Converting a Decimal Fraction to a Binary Fraction

Example: Convert 0.14159 to a binary fraction.

$$0.14159 = f_1 2^{-1} + f_2 2^{-2} + f_3 2^{-3} + f_4 2^{-4} + \cdots$$

1. **Multiply both sides by two**:

$$0.28318 = f_1 + \left(f_2 2^{-1} + f_3 2^{-2} + f_4 2^{-3} + \cdots \right)$$

Since $f_1 \in \{0, 1\}$ and $0 \le \left(f_2 2^{-1} + f_3 2^{-2} + f_4 2^{-3} + \cdots \right) < 1,$

we must have $f_1 = 0$ (the integer part of 0.28318)

So, $0.28318 = f_2 2^{-1} + f_3 2^{-2} + f_4 2^{-3} + \cdots$

2. **Multiply both sides by two**:

$$0.56636 = f_2 + \left(f_3 2^{-1} + f_4 2^{-2} + \cdots \right)$$

So, $f_2 = 0$ and $0.56636 = f_3 2^{-1} + f_4 2^{-2} + \cdots$

Converting a Decimal Fraction to a Binary Fraction

3. Multiply both sides by two:

$$1.13272 = f_3 + \left(f_4 2^{-1} + f_5 2^{-2} + \cdots \right)$$

So, $f_3 = 1$ and subtracting 1 from both sides,

$$0.13272 = f_4 2^{-1} + f_5 2^{-2} + \cdots$$

4. Multiply both sides by two:

$$0.26544 = f_4 + \left(f_5 2^{-1} + \cdots \right) \quad \text{so} \quad f_4 = 0$$

So the first four binary digits of the Base-2 expansion of 0.14159 is

$$\boxed{0.14159 \approx 0.0010_2 = \frac{1}{8} = 0.125}$$

The process can be continued to any desired accuracy.

Converting a Decimal Fraction to a Binary Fraction

Example: 0.14159

Fraction	Fraction × 2	Integer Part
0.14159	0.28318	$f_1 = 0$
0.28318	0.56636	$f_2 = 0$
0.56636	1.13272	$f_3 = 1$
0.13272	0.26544	$f_4 = 0$
0.26544	0.53088	$f_5 = 0$
0.53088	1.06176	$f_6 = 1$
0.06176	0.12352	$f_7 = 0$
0.12352	0.24704	$f_8 = 0$

So, the 7-bit approximation is $0.14159 \approx 0.0010010_2 = 0.140625$

We don't have to round up since $f_8 = 0$.

Example: decimal $\rightarrow$ computer floating-point $\rightarrow$ hex

To convert $\pi \approx 3.14159$ to Modified Floating-Point , convert the integer and fractional parts separately:

$$3 = 11_2, \text{ and from the previous table } 0.14159 \approx 0.0010010_2$$

So,

$$\pi \approx 3.14159 \approx 11.0010010_2 = 0.11001001_2 \times 2^2 \qquad \text{(Normalized)}$$

Since we only use 7 bits for the fractional part, we drop the eighth bit and round the seventh bit up:

$$\pi \approx (-1)^s f \times 2^e = (-1)^0 \times 0.1100101_2 \times 2^2$$

$$s = 0, \qquad f = 0.1100101_2, \qquad b = e + 128 = 130$$

Example (cont) : decimal $\rightarrow$ computer floating-point $\rightarrow$ hex

$$\pi \rightarrow \underbrace{1000\ 0010}_{b\,=\,130}\ \overset{s}{0}\underbrace{110\ 0101}_{f\,=\,0.7890625} = 0x8265$$

Floating-point answer:

$$\pi \approx f \times 2^{e}$$
$$= 0.7890625 \times 2^{2}$$
$$= 3.15625$$

Error = (3.15625 – π) / π = 0.47%

Floating Point Addition

$$C = A + B = \left(f_A \times 2^{e_A} \right) + \left(f_B \times 2^{e_B} \right)$$

1. If $e_A = e_B$, then $C = \left(f_A + f_B \right) \times 2^{e_A}$

2. If $e_A < e_B$, then increase e_A until it is equal to e_B, rotating the f_A register one bit right (`rrf`) for each increment of e_A .

3. If $e_B < e_A$, then increase e_B until it is equal to e_A, rotating the f_B register one bit right (`rrf`) for each increment of e_B .

Computer Floating Point Addition

Example: $C = A + B =$ 0x82E0 + 0x82C0 $\boxed{C = (f_A + f_B) \times 2^{e_A}}$

$b_C = b_A = b_B =$ 0x82 = 130 (biased exponent)

E0 = $1\underbrace{110\ 0000}_{}$ C0 = $1\underbrace{100\ 0000}_{}$ (implied binary point after sign bit)

$f_A =$ 0x60 = 0.75 $f_B =$ 0x40 = 0.50

Add fractional parts (ignore the sign bits):

$$\overline{f}_C = f_A + f_B = 0.110\ 0000_2 + 0.100\ 0000_2 = 1.010\ 0000_2 = 0.101\ 0000_2 \times 2^1$$

$$C = (f_A + f_B) \times 2^{e_A} = 0.1010000_2 \times 2^1 \times 2^{e_A} = 0.1010000_2 \times 2^{e_A+1},$$

so we must add 1 to the biased exponent: $f_C = 0.101\ 0000_2$, $b_C =$ 0x82 + 1 = 0x83

Prepend the sign bit to f_C: 1101 0000 = 0xD0 **Final answer:** $C =$ 0x83D0

Floating Point Addition Algorithm

$$C = \left(f_A + f_B \right) \times 2^{e_C}, \text{ where } e_C = e_A = e_B$$

Remove sign bits s_A and s_B from the low bytes in computer format.

If $s_A = 0$ and $s_B = 1$ then
    if $f_A > f_B$
        $f_C = f_A - f_B$ and $s_C = 0$ **positive number > negative number**
    else
        $f_C = f_B - f_A$ and $s_C = 1$ **negative number > positive number**
    Prepend the sign bit to f_C
end if

Floating Point Subtraction

```
;**********************************************************
FloatSub(fA, fB )                 ; This routine calculates
                                  ; A – B

     fB = (fB  xor  1000 0000)    ; -B: Complement sign bit

     call FloatAdd(fA, fB)        ; This routine calculates
                                  ; A + (-B)

;**********************************************************
```

Reminder:

A	**B**	**A xor B**
0	0	0
0	1	1
1	0	1
1	1	0

Floating Point Multiplication

$$C = A \times B = \left(f_A \times 2^{e_A} \right) \times \left(f_B \times 2^{e_B} \right) = \left(f_A \times f_A \right) \times 2^{e_A + e_B} = f_C \times 2^{e_C}$$

Mantissa: $f_C = f_A \times f_B$

Exponent: $e_C = e_A + e_B$

Sign bit: $s_C = s_A$ xor s_B

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

$$\begin{aligned} f_A \times f_B &= (0.a_0 \ldots a_6) \times (0.b_0 \ldots b_6) \\ &= 0.c_0 \ldots c_{13} \end{aligned}$$

(Use integer multiply routine)

Round the result to 7 MSBs $\rightarrow f_C = 0.c_0 \ldots c_6$

Floating Point Multiplication

$e_C = e_A + e_B$ (unbiased exponent)

But remember the computer floating point number contains biased exponents b_A and b_B

Assume 8-bit exponents:

$$(b_C - 128) = (b_A - 128) + (b_B - 128)$$
$$b_C = b_A + b_B - 128$$

Enhancement: Check for overflow

Floating Point Division

$$C = A / B = \left(f_A \times 2^{e_A} \right) / \left(f_B \times 2^{e_B} \right) = \left(f_A \,/\, f_A \right) \times 2^{e_A - e_B} = f_C \times 2^{e_C}$$

Mantissa: $f_C = f_A / f_B$

Exponent: $e_C = e_A - e_B$ (convert to biased exponent subtraction)

Sign bit: $s_C = s_A$ xor s_B

Use an integer divide routine that gives a 16-bit result:

$$\begin{aligned} f_A / f_B &= (0.a_0 \ldots a_6) / (0.b_0 \ldots b_6) = (a_0 \ldots a_6) / (b_0 \ldots b_6) \\ &= (c_0 \ldots c_7).(c_8 \ldots c_{15}) \end{aligned}$$

Shift division result so c_8 is the MSB equal to 1 ($c_0 \ldots c_7 = 0$), incrementing e_C each right shift, decrementing e_C each left shift .

End of Lab 11