ECE232: Hardware Organization and Design

Part 4: Datapath Design – Multiplication and Floating-point representations and operations

http://www.ecs.umass.edu/ece/ece232/

Adapted from *Computer Organization and Design*, Patterson & Hennessy, UCB

MULTIPLY (unsigned)

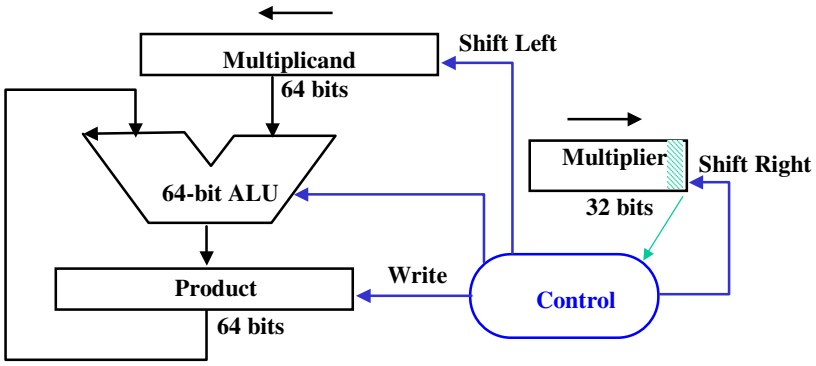
- Paper and pencil example (unsigned):

```
Multiplicand    1000
  Multiplier          1001
                      1000
                     0000
                    0000
                   1000
  Product         01001000
```

- m bits x n bits = m+n bit product
- Binary makes it easy:
  - 0 → place 0 (0 x multiplicand)
  - 1 → place a copy (1 x multiplicand)
- 3 versions of multiply hardware & algorithm:
  - successive refinement

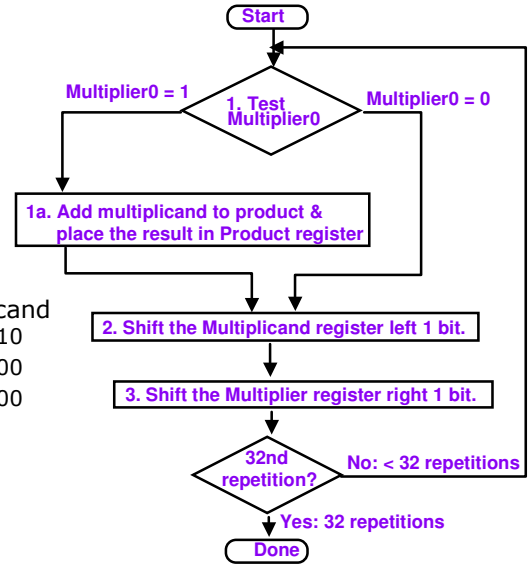
Unsigned shift-add multiplier (version 1)

- 64-bit Multiplicand reg, 64-bit ALU, 64-bit Product reg, 32-bit multiplier reg



Multiplier = datapath + control

Multiply Algorithm - Version 1



Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110		

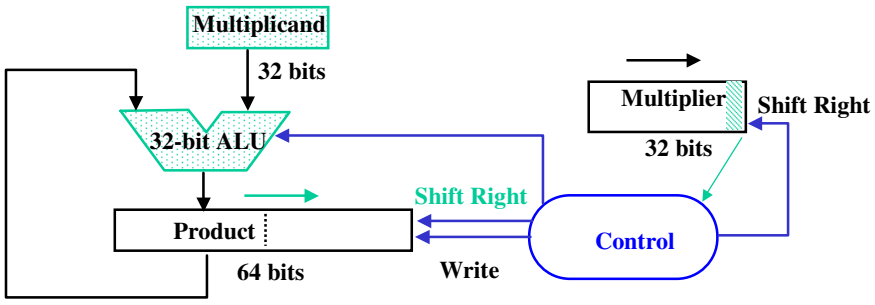
Observations on Multiply Version 1

- 1 cycle per step → 32x3 = ~ 100 cycles per multiply. However, One cycle per iteration can be saved by shifting multiplier and multiplicand in one cycle → 32x2
- 50% of the bits in multiplicand are 0 → 64-bit adder is wasted
- 0s inserted in right of multiplicand as shifted to the left → least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to the right

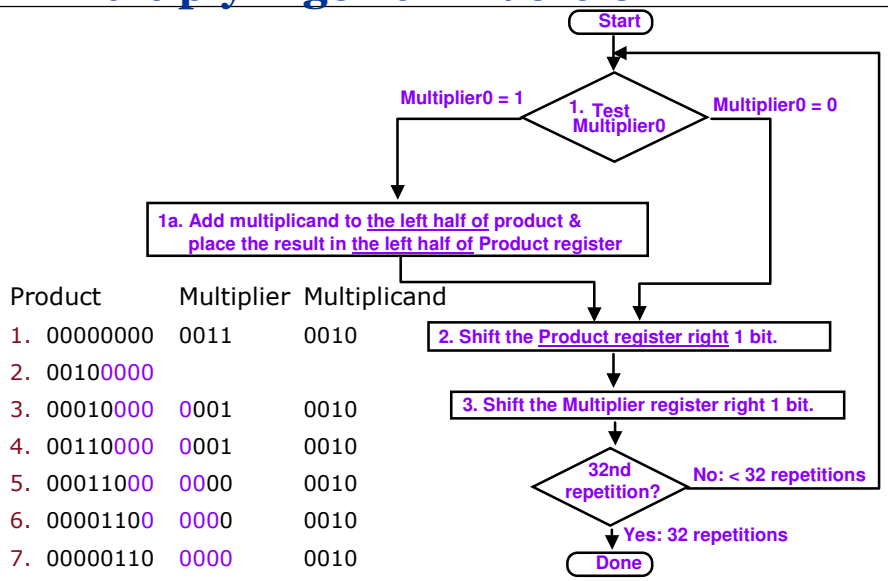
```
  1001
  1010
 ------
  0000
 1001
0000
1001
--------
10100010
```


Multiply Hardware - Version 2

- 32-bit Multiplicand reg, 32 -bit ALU, 64-bit Product reg, 32-bit Multiplier reg



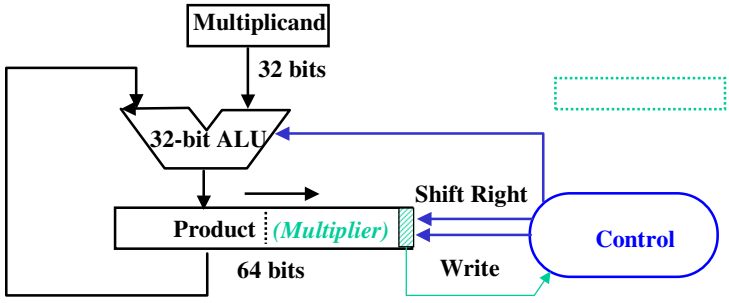
Multiply Algorithm Version 2



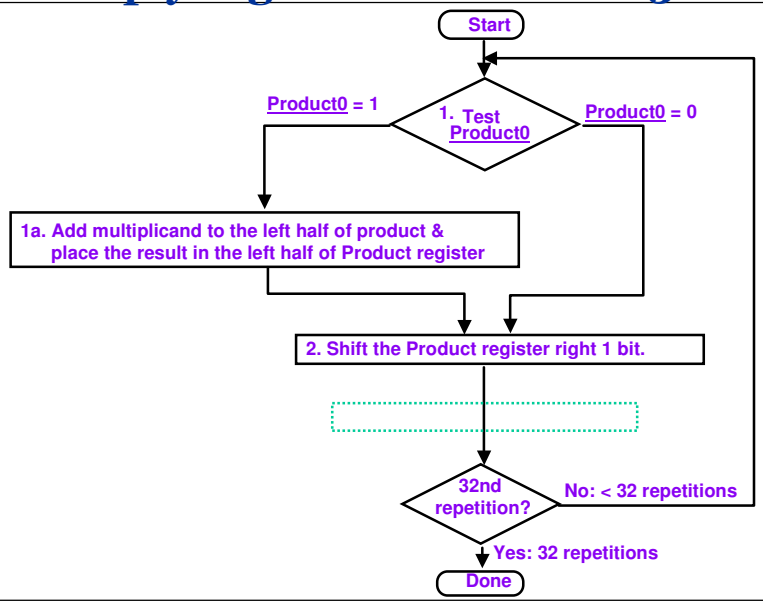
	Product	Multiplier	Multiplicand
1.	00000000	0011	0010
2.	00100000		
3.	00010000	0001	0010
4.	00110000	0001	0010
5.	00011000	0000	0010
6.	00001100	0000	0010
7.	00000110	0000	0010

Multiply Hardware - Version 3

- Product register wastes space that exactly matches size of multiplier
  → combine Multiplier register and Product register
- 32-bit Multiplicand reg, 32-bit ALU, 64-bit Product reg, (0-bit Multiplier reg)



Multiply Algorithm - Version 3



Adapted from *Computer Organization and Design*, Patterson & Hennessy, UCB, Kundu, UMass

Observations on Multiply Version 3

- 2 steps per bit because Multiplier & Product combined
- How can you make it faster?
- What about signed multiplication?
  - trivial solution: make both positive & complement product if one of operands is negative (leave out the sign bit, run for 31 steps)
  - apply definition of 2's complement
    - need to sign-extend partial products

A		1	0	1	1				-5
X	$\times$	0	0	1	1				3
$P^{(0)} = 0$		0	0	0	0				
$x_0 = 1 \Rightarrow$ Add A	$+$	1	0	1	1				
		1	0	1	1				
Shift		1	1	0	1	1			
$x_1 = 1 \Rightarrow$ Add A	$+$	1	0	1	1				
		1	0	0	0	1			
Shift		1	1	0	0	0	1		
$x_2 = 0 \Rightarrow$ Shift only		1	1	1	0	0	0	1	-15

Source: I. Koren, Computer Arithmetic Algorithms, 2nd Edition, 2002

Floating Point Numbers

- The largest 32 bit unsigned integer number is
  1111 1111 1111 1111 1111 1111 1111 1111 = 4,294,967,295
- What if we want to encode the approx. age of the earth?
  4,600,000,000 or 4.6×10^9
- or the weight in kg of one a.m.u. (atomic mass unit)
  0.00000000000000000000000000166 or 1.6×10^{-27}
- There is no way we can encode either of the above in a 32-bit integer.

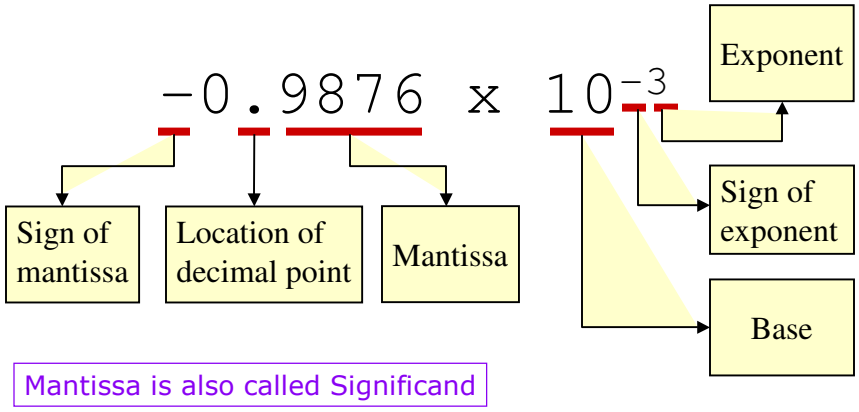
Exponential Notation

- The following are equivalent representations of 1,234

123,400.0	$\times 10^{-2}$
12,340.0	$\times 10^{-1}$
1,234.0	$\times 10^{0}$
123.4	$\times 10^{1}$
12.34	$\times 10^{2}$
1.234	$\times 10^{3}$
0.1234	$\times 10^{4}$
0.01234	$\times 10^{5}$

The representations differ in that the decimal place – the "point" - "floats" to the left or right (with the appropriate adjustment in the exponent).

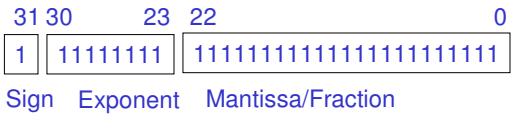
Parts of a Floating Point Number



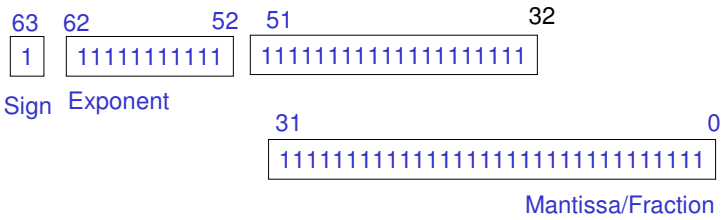
Mantissa is also called Significand

IEEE 754 Floating Point Standard

- Single Precision: 32 bits (1+8+23)

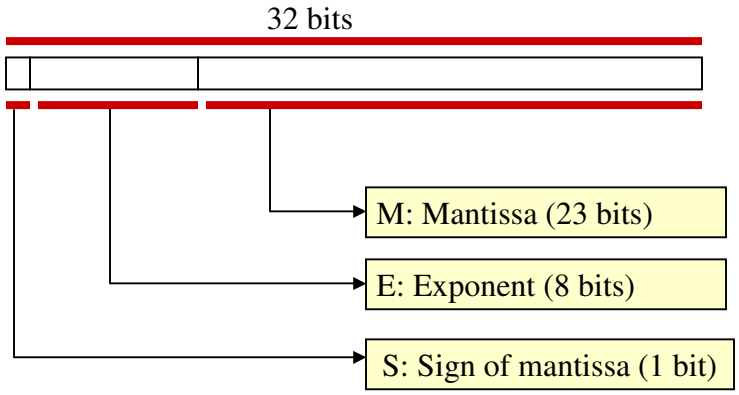


- Double Precision: 64 bits (1+11+52)



Single Precision Format

- Note that the exponent has no explicit sign bit
- Base?



Normalization

- The mantissa *M* is a *normalized* fraction
- Has an implied decimal place on left
- Has an implied (hidden) "1" on left of the decimal place
- E.g.,
  - Fraction → 10100000000000000000000
  - Represents $1.101_2 = 1.625_{10}$
- The significand=*1.f* is in the range [1, 2-ulp]
  - ulp – unit in the last position

$$F = (-1)^S 1.f * 2^{E-Bias}$$

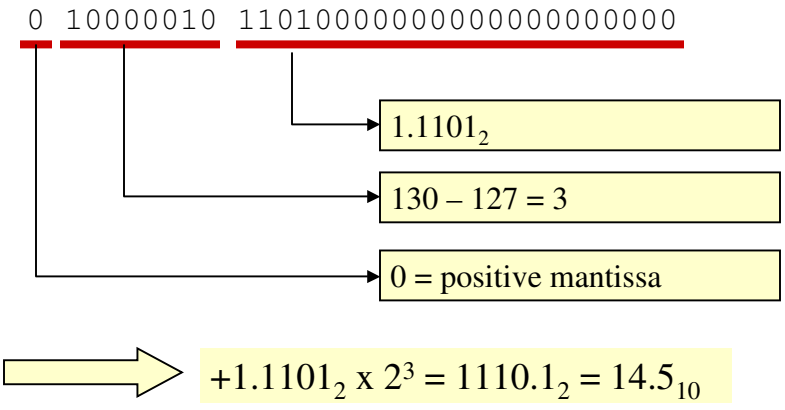
Excess Notation

- To include +ve and –ve exponents, "excess" notation is used
- Single precision: excess 127
- Double precision: excess 1023
- The value of the exponent stored is larger than the actual exponent
- E.g., excess 127, I.e., Bias=127
  - Exponent → 10000111
  - Represents 135 – 127 = 8

$$F = (-1)^S 1.f * 2^{E-127}$$

S	8 bits - biased exponent E	23 bits - unsigned fraction f

Example: Single precision



Converting to IEEE format

- Example - decimal number: -3.154×10^0
- What is the sign?
- What is the exponent?
- What is the mantissa?

Converting Mixed Numbers – Decimal to Binary

$456.78_{10} = 4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2}$

$1011.11_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$

$= 8 + 0 + 2 + 1 + 1/2 + ¼$

$= 11 + 0.5 + 0.25 = 11.75_{10}$

How to convert whole Decimal to Binary

- Successive division by 2
- $57143_{10} = 1101111100110111_2$

	1
1	**1**
3	**0**
6	**1**
13	**1**
27	**1**
55	**1**
111	**1**
223	**0**
446	**0**
892	**1**
1785	**1**
3571	**0**
7142	**1**
14285	**1**
28571	**1**
57143	

Converting fractional Decimal to Binary

Successive multiplication by 2

0	0.154	
1	0.308	0
2	0.616	0
3	1.232	1
4	0.464	0
5	0.928	0
6	1.856	1
7	1.712	1
8	1.424	1
9	0.848	0
10	1.696	1
11	1.392	1

12	0.784	0
13	1.568	1
14	1.136	1
15	0.272	0
16	0.544	0
17	1.088	1
18	0.176	0
19	0.352	0
20	0.704	0
21	1.408	1
22	0.816	0
23	1.632	1

Decimal 0.154 = .0010 0111 0110 1100 1000 101

Example Continued (-3.154)

- 3.154_{10} = 11. + binary for (.154)

0.154 = .00100111011011001000101

00100111011011001000101 = 1291845

$$\frac{1291845}{2^{23}} = 0.1539999924$$

Overall number is

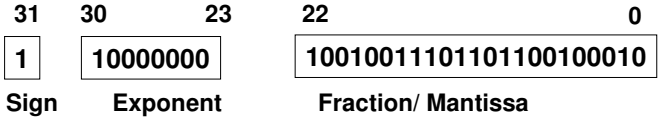
$$= 11.00100111011011001000101$$

$$= 1.10010011101101100100010 \times 2^{1}$$

31	30 ... 23	22 ... 0
1	????????	10010011101101100100010
Sign	Exponent	Fraction/ Mantissa

-3.154

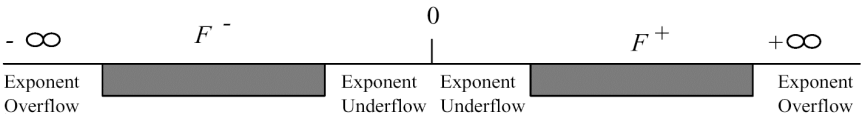
- IEEE Short Real exponents are stored as 8-bit unsigned integers with a bias of 127



Exponent	Adjusted Exponent	Binary
1	128	1000 0000
-127	0	0000 0000
+128	255	1111 1111

Floating Point Special Representations

$$F = (-1)^S 1.f * 2^{E-127} \qquad 1 \le 1.f < 2$$



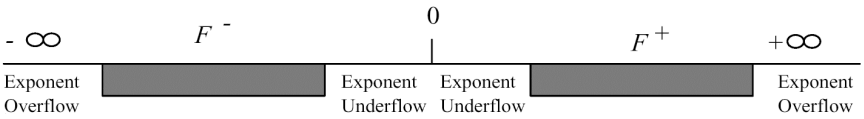
	$f = 0$	$f \neq 0$
$E = 0$	0	Denormalized
$E = 255$	$\pm\infty$	NaN

- There are *two* Zeroes, ±0, and *two* Infinities ±∞
- NaN (Not-a-Number) may have a sign and have a non-zero fraction - used for program diagnostics
- NaNs and Infinities have all 1s in the Exp field, E=255.

F+∞=∞, F/∞=**0**

Floating Point Special Representations

$$F = (-1)^S 1.f * 2^{E-127} \qquad 1 \le 1.f < 2 \qquad 1 \le E \le 254$$



Single Precision		Double Precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	nonzero	0	nonzero	± denormalized number
1-254	Anything	1-2046	Anything	± floating point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	NaN (not a number)

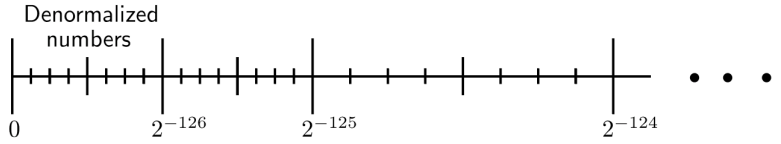
Denormalized Numbers

- −126 is the smallest exponent for a normalized number

$$F_{min}^{+} = 1.0 \cdot 2^{1-127} = 2^{-126}$$

- Denormalized numbers are the same except that exponent = −126 and significand is 0.Fraction.

$$F = (-1)^S 0.f * 2^{-126}$$



Smallest & Largest Numbers

- The smallest non-zero positive and largest non-zero negative *normalized* numbers (represented by 1 in the Exp field and 0…0 in the Fraction field) are
  - $\pm 2^{-126} \approx \pm 1.175494351 \times 10^{-38}$
- The smallest non-zero positive and largest non-zero negative *denormalized* numbers (represented by all 0s in the Exp field and 0…01 in the Fraction field) are
  - $\pm 2^{-149} \approx \pm 1.4012985 \times 10^{-45}$
- The largest finite positive and smallest finite negative numbers (represented by 254 in the Exp field and 1…1 in the Fraction field) are
  - $\pm (2)(2^{127}) \approx \pm 3.40 \times 10^{38}$

$$F_{max}^{+} = (2 - 2^{-23}) \cdot 2^{254-127} = (1 - 2^{-24}) \cdot 2^{128}$$

Single Precision Summary

Type	Exponent	Mantissa	Value
Zero	**0000 0000**	**000 0000 0000 0000 0000 0000**	**0**
One	**0111 1111**	**000 0000 0000 0000 0000 0000**	**1**
Denormalized number	**0000 0000**	**100 0000 0000 0000 0000 0000**	**5.9×10^{-39}**
Largest normalized number	**1111 1110**	**111 1111 1111 1111 1111 1111**	**3.4×10^{38}**
Smallest normalized number	**0000 0001**	**000 0000 0000 0000 0000 0000**	**1.18×10^{-38}**
Infinity	**1111 1111**	**000 0000 0000 0000 0000 0000**	**Infinity**
NaN	**1111 1111**	**010 0000 0000 0000 0000 0000**	**NaN**

Double-Precision Format

S	11 bits - biased exponent E	52 bits - unsigned fraction f

- For $1 \le E \le 2046$ -

$$F = (-1)^S \; 1.f \; 2^{E-1023}$$

	Single	Double
Word length	32 bits	64 bits
Fraction + hidden bit	$23+1$ bits	$52+1$ bits
Exponent	8 bits	11 bits
Bias	127	1023
Approximate range	$2^{128} \approx 3.8 \cdot 10^{38}$	$2^{1024} \approx 9 \cdot 10^{307}$
Smallest normalized number	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$
Approximate resolution	$2^{-23} \approx 10^{-7}$	$2^{-52} \approx 10^{-15}$

Floating-point representations and arithmetic operations:
http://www.ecs.umass.edu/ece/koren/arith/simulator/

Source: I. Koren, Computer Arithmetic Algorithms, 2nd Edition, 2002

Floating Point Operations

- Execution depends on format used for operands
- Assumption: Significands are normalized fractions in signed-magnitude representation ; exponents are biased
- Given two numbers

$$X = (-1)^{S_1} 1.f_1 2^{E_1-127}$$

$$Y = (-1)^{S_2} 1.f_2 2^{E_2-127}$$

- Calculate result of a basic arithmetic operation yielding

$$Z = (-1)^{S_3} 1.f_3 2^{E_3-127}$$

- Multiplication and division are simpler to follow than addition and subtraction

S	E	f

Adapted from *Computer Organization and Design*, Patterson & Hennessy, UCB, Kundu, UMass

Floating Point Multiplication

- Operations can be done in parallel
  1. Sign S3 positive if signs S1 and S2 are equal - negative if not
  2. When adding two exponents E1 =E1r +bias and E2 =E2r +bias: bias must be subtracted once
  3. If resulting exponent E3 is larger than Emax / smaller than Emin - overflow/underflow indication must be generated
  4. M1 and M2 are multiplied and then we must use postnormalization step

  Example: Multiply 0.5_{10} and -0.4375_{10}

  $0.5 = 1.00_2 \times 2^{-1;}$ $0.4375 = 1.11_2 \times 2^{-2}$

  Sign = ?

  Exponent=(-1+127) + (-2+127)– 127 =124 (over/underflow check)

  1.00 * 1.11 = 1.11

  Final result = $-1.11 \times 2^{-2} = -0.21875_{10}$

  Read 3.5 for details

Addition and Subtraction

- Exponents of both operands must be equal before adding or subtracting significands
- When E1=E2 - 2^{E1} can be factored out and significands M1 and M2 can be added
- Significands aligned by shifting the significand of the smaller operand |E1-E2| positions to the right, increasing its exponent, until exponents are equal
- E1≥E2 -

$$Z = ((-1)^{S_1} 1.f_1 \pm (-1)^{S_2} 1.f_2 2^{-(E_1 - E_2)}) 2^{E_1 - 127}$$

- Exponent of larger number not decreased - this will result in a significand larger than 1 - a larger significand adder required

Addition/Subtraction - postnormalization

- Addition - resultant significand **M** (sum of two aligned significands) is in range $1 \le M < 4$
- If $M>2$ - a postnormalization step - shifting significand to the right to yield **M3** and increasing exponent by one - is required (an exponent overflow may occur)
- Subtraction - Resultant significand M is in range $0 \le |M|<2$ - postnormalization step - shifting significand to left and decreasing exponent - is required if $M<1$ (an exponent underflow may occur)
- In extreme cases, the postnormalization step may require a shift left operation over all bits in significand, yielding a zero result

$$M_3 = 1.f_3$$

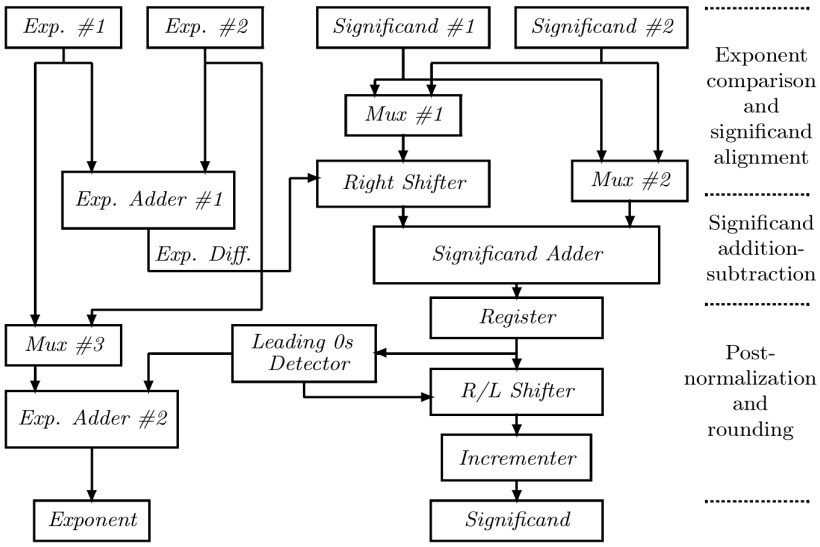
Source: I. Koren, Computer Arithmetic Algorithms, 2nd Edition, 2002

Steps in Addition/Subtraction

- Step 1: Calculate difference d of the two exponents - d=|E1 - E2|
- Step 2: Shift significand of smaller number by d positions to the right
- Step 3: Add aligned significands and set exponent of result to exponent of larger operand
- Step 4: Normalize resultant significand and adjust exponent if necessary
- Step 5: Round resultant significand and adjust exponent if necessary

Source: I. Koren, Computer Arithmetic Algorithms, 2nd Edition, 2002

Circuitry for Addition/Subtraction



Source: I. Koren, Computer Arithmetic Algorithms, 2nd Edition, 2002