* Booth's multiplication
Flow + Practice *

Lecture – 11

Booth’s Multiplication Algorithm

- It employs both addition and subtraction.
- It treats positive and negative operands uniformly.
- No special actions are required for negative numbers.
- It provides faster execution.

Booth's Multiplication Algorithm

```
        0010two
x       0110two
       ---------
+       0000 shift (0 in multiplier)
+      0010  add   (1 in multiplier)
+     0010   add   (1 in multiplier)
+    0000    shift (0 in multiplier)
    -----
    00001100two
```

Booth observed that an ALU that could add or subtract could get the same result in more than one way. For example, since

$$6_{ten} = -2_{ten} + 8_{ten}$$

or

$$0110_{two} = -0010_{two} + 1000_{two}$$

we could replace a string of 1s in the multiplier with an initial subtract when we first see a 1 and then later add when we see the bit *after* the last 1. For example,

```
        0010two
x       0110two
       ---------
+       0000   shift (0 in multiplier)
-      0010    sub (first 1 in multiplier)
+     0000     shift (middle of string of 1s)
+    0010      add (prior step had last 1)
    -----
    00001100two
```

Booth's Multiplication Algorithm

Current bit	Bit to the right	Explanation	Example
1	0	Beginning of a run of 1s	$0000111\mathbf{10}00_{two}$
1	1	Middle of a run of 1s	$00001\mathbf{11}1000_{two}$
0	1	End of a run of 1s	$000\mathbf{01}111000_{two}$
0	0	Middle of a run of 0s	$0\mathbf{00}01111000_{two}$

- **Step – 1:**
  Two adjacent bits $x_i x_{i-1}$ are examined in each step.
  If $x_i x_{i-1}$ =01, then Y is added to the current partial product P_i.
  If $x_i x_{i-1}$ =10, then Y is subtracted from P_i.
  If $x_i x_{i-1}$ =00 or 11, then neither addition or subtraction is performed.

- **Step – 2:**
  Shift the product register right 1 bit.

Booth’s Multiplication Algorithm

- It effectively skips over runs of 1s and runs of 0s that it encounters in *X*.
- It reduces the average number of add-subtract steps and allows faster multipliers to be designed.
- It involves more complex circuitry.

Example

Step	Multiplicand (M)	Multiplier (Q)	Q_{-1}	Action	A (Accumulator)	Q (Multiplier)	Q_{-1}
Initial	01100 (12)	00111 (7)	0	-	00000	00111	0
1	01100	00111	0	Check Q_0Q_{-1} = 10	10100 (A = A - M)	00111	0
				Arithmetic Shift Right	11010	00011	1
2	01100	00011	1	Check Q_0Q_{-1} = 11	- (No action)	-	-
				Arithmetic Shift Right	11101	00001	1
3	01100	00001	1	Check Q_0Q_{-1} = 11	- (No action)	-	-
				Arithmetic Shift Right	11110	10000	1
4	01100	10000	1	Check Q_0Q_{-1} = 01	01010 (A = A + M)	10000	1
				Arithmetic Shift Right	00101	01000	0

01100_2 X 00111_2

Example

Step	Multiplicand (M)	Multiplier (Q)	Q_{-1}	Action	A (Accumulator)	Q (Multiplier)	Q_{-1}
Initial	0010	1101	0	-	0000	1101	0
1	0010	1101	0	$Q_0Q_{-1} = 10 \rightarrow A = A - M$	1110 (A - M)	1101	0
				Arithmetic Shift Right	1111	0110	1
2	0010	0110	1	$Q_0Q_{-1} = 01 \rightarrow A = A + M$	0001 (A + M)	0110	1
				Arithmetic Shift Right	0000	1011	0
3	0010	1011	0	$Q_0Q_{-1} = 10 \rightarrow A = A - M$	1110 (A - M)	1011	0
				Arithmetic Shift Right	1111	0101	1
4	0010	0101	1	$Q_0Q_{-1} = 11 \rightarrow$ No action	-	-	-
				Arithmetic Shift Right	1111	1010	1

2_{10} X $-3_{10} = 0010_2$ X 1101_2

Validity of Booth's Multiplication Algorithm

- Let X is a positive integer and contains a subsequence X^* consisting of a run of k 1s flanked by two 0s.

$$X^* = x_i x_{i-1} x_{i-2} \cdots x_{i-k+1} x_{i-k} x_{i-k-1}$$
$$= 0\,1\,1 \ldots 1\,1\,0$$

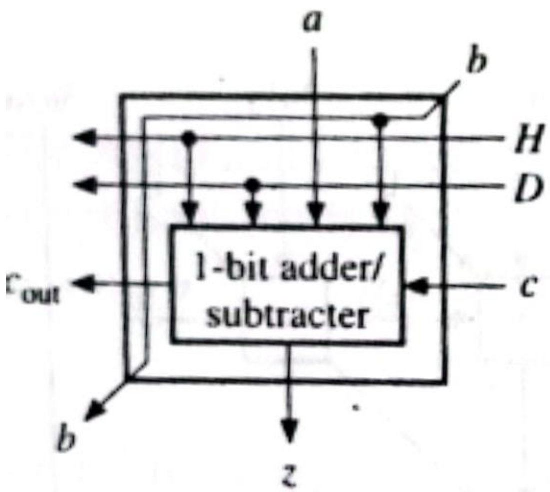
- In normal multiplication the contribution of X^* to $P = X \times Y$ is $\sum_{j=i-k}^{i-1} 2^j Y$
- In booth's multiplication, $x_i x_{i-1}$ =01 which contributes $2^i Y$ *to* P.
- When $x_{i-k} x_{i-k-1}$ = 10 the contribution is $-2^{i-k} Y$ *to* P.
- So the net contribution is

$$2^i Y - 2^{i-k} Y = 2^{i-k} Y (2^k - 1) Y$$
$$= 2^{i-k} \sum_{m=0}^{k-1} 2^m Y$$
$$= \sum_{m=0}^{k-1} 2^{m+i-k} Y$$

- If j=m+i-k then we get $\sum_{j=i-k}^{i-1} 2^j Y$

Array Implementation of the Booth Multiplication Algorithm

- It requires a multifunction cell capable of addition, subtraction and no operation (skip).



H	D	Function
0	X	$z = a$ (no operation)
1	0	$c_{out}z = a$ **plus** b **plus** c (add)
1	1	$c_{out}z = a - b - c$ (subtract)

- The functions of B are defined by $z = a \oplus bH \oplus cH$ and $C_{out} = (a \oplus D)(b+c) + bc$

Array Implementation of the Booth Multiplication Algorithm

$z = a \oplus bH \oplus cH$ and $C_{out} = (a \oplus D)(b+c) + bc$

- When HD = 10 the equations reduce to full adder equations.

$$z = a \oplus b \oplus c$$

$$c_{out} = ab + ac + bc$$

- When HD = 11 the equations reduce to full subtracter equations.

$$z = a \oplus b \oplus c$$

$$c_{out} = \bar{a}b + \bar{a}c + bc$$

- When H = 0 then z = a and carry plays no role in the final result.
- A n $\times$ n bit multiplier is constructed from $n^2 + n(n-1)/2$ cells.

Array Implementation of the Booth Multiplication Algorithm

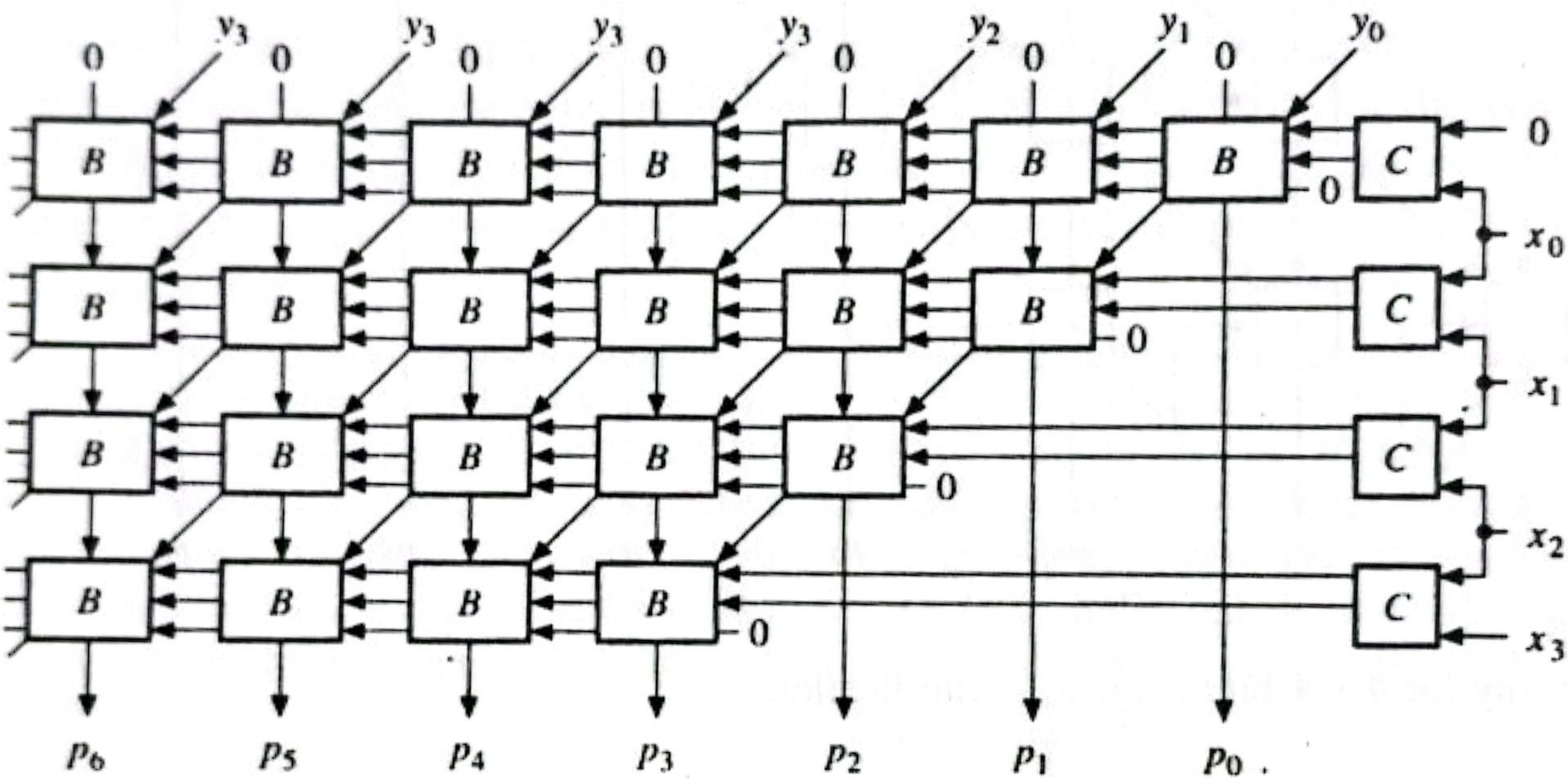
- C cell generates control input H and D required by the B cells depending on the combination of $x_i x_{i-1}$.

$$H = x_i \oplus x_{i-1}$$

$$D = x_i \bar{x}_{i-1}$$

x_i	x_{i-1}	H	D
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	0

Array Implementation of the Booth Multiplication Algorithm



Floating-Point Addition

- $9.999 \times 10^{1} + 1.610 \times 10^{-1}$. Assume that we can store four decimal digits of the significand and two decimal digits of the exponent.

Step 1: Align the decimal point of the number that has the smaller exponent.

$$1.610_{ten} \times 10^{-1} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}$$

After shifting, the number is

Step 2: Add the significands.

$$\begin{array}{rr} & 9.999_{ten} \\ + & 0.016_{ten} \\ \hline & 10.015_{ten} \end{array}$$

The sum is $10.015_{ten} \times 10^{1}$.

Floating-Point Addition

Step 3: Normalize the result.

$$10.015_{ten} \times 10^1 = 1.0015_{ten} \times 10^2$$

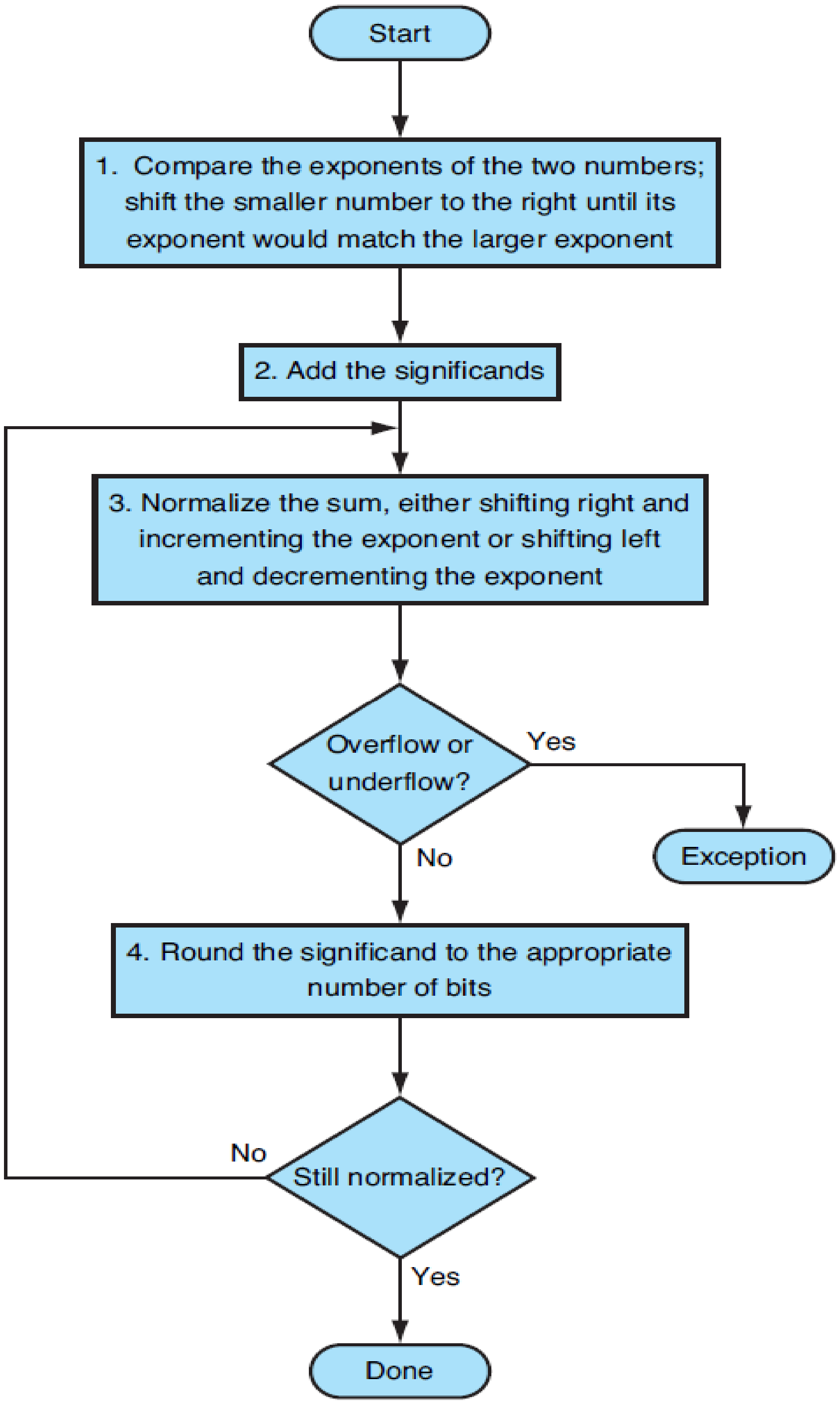
Check for underflow and overflow.

Step 4: Round the result.

Truncate the number if the digit to the right of the desired point is between 0 and 4. Add 1 to the digit if the number to the right is between 5 and 9.

$$1.002_{ten} \times 10^2$$

Floating-Point Addition



Example of Floating-Point Addition

Add the numbers 0.5 and -0.4375 in binary.

$$
\begin{aligned}
0.5_{\text{ten}} &= 1/2_{\text{ten}} && = 1/2^1{}_{\text{ten}} \\
&= 0.1_{\text{two}} && = 0.1_{\text{two}} \times 2^0 && = 1.000_{\text{two}} \times 2^{-1} \\
-0.4375_{\text{ten}} &= -7/16_{\text{ten}} && = -7/2^4{}_{\text{ten}} \\
&= -0.0111_{\text{two}} && = -0.0111_{\text{two}} \times 2^0 && = -1.110_{\text{two}} \times 2^{-2}
\end{aligned}
$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{\text{two}} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{\text{two}} \times 2^{-2} = -0.111_{\text{two}} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{\text{two}} \times 2^{-1} + (-0.111_{\text{two}} \times 2^{-1}) = 0.001_{\text{two}} \times 2^{-1}$$

Step 3. Normalize the sum, checking for overflow or underflow:

$$
\begin{aligned}
0.001_{\text{two}} \times 2^{-1} = 0.010_{\text{two}} \times 2^{-2} &= 0.100_{\text{two}} \times 2^{-3} \\
&= 1.000_{\text{two}} \times 2^{-4}
\end{aligned}
$$

Example of Floating-Point Addition

Step 4. Round the sum:

$$1.000_{\text{two}} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$\begin{aligned}1.000_{\text{two}} \times 2^{-4} &= 0.0001000_{\text{two}} = 0.0001_{\text{two}} \\ &= 1/2^4{}_{\text{ten}} \qquad = 1/16_{\text{ten}} \qquad = 0.0625_{\text{ten}}\end{aligned}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.

Floating-Point Multiplication

- $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$. Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1: Calculate the exponent of the product by simply adding the exponents of the operands together.

$$\text{New exponent} = 10 + (-5) = 5$$

Using the biased notation,

New exponent = (10+127)+(-5+127) – 127

= 137 + 122 – 127

=132 = 5 + 127

Step 2: Multiply significands.

$$\begin{array}{r} 1.110_{ten} \\ \times\ 9.200_{ten} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{ten} \end{array}$$

$10.212000_{ten} = 10.212 \times 10^5$

Floating-Point Multiplication

Step 3: Normalize the product and check for underflow and overflow.

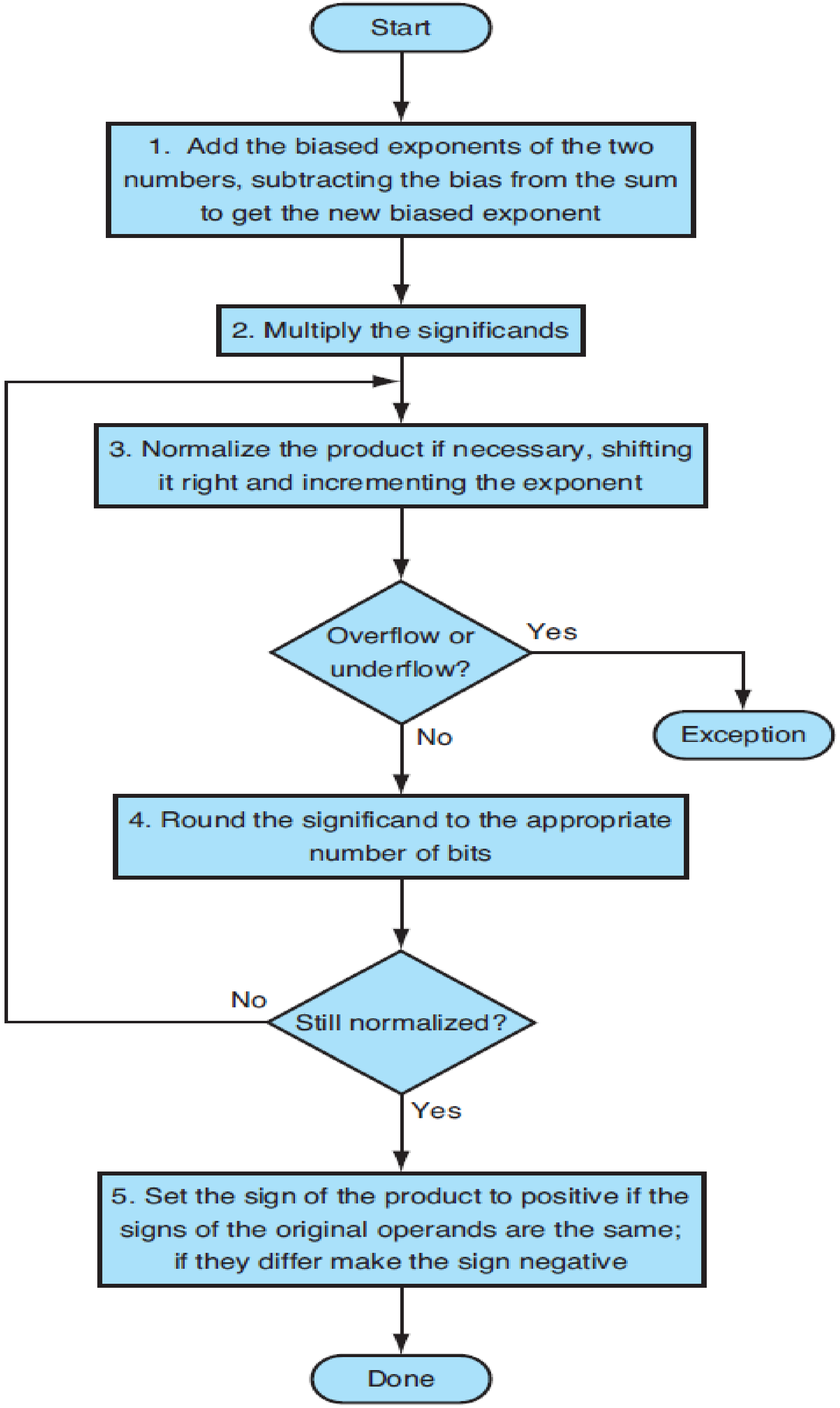
$$10.212_{ten} \times 10^5 = 1.0212_{ten} \times 10^6$$

Step 4: The number 1.0212×10^6 is rounded to 1.021×10^6.

Step 5: Set the sign the of the product. The sign of the product depends on the sign of the original operands. If they are both the same, the sign is positive. Otherwise it is negative.

So, the product is $+ 1.021 \times 10^6$

Floating-Point Multiplication



Example of Floating-Point Addition

Multiply the numbers 0.5 and -0.4375 in binary.

$$
\begin{aligned}
0.5_{ten} &= 1/2_{ten} &&= 1/2^1{}_{ten} \\
&= 0.1_{two} &&= 0.1_{two} \times 2^0 &&= 1.000_{two} \times 2^{-1} \\
-0.4375_{ten} &= -7/16_{ten} &&= -7/2^4{}_{ten} \\
&= -0.0111_{two} &&= -0.0111_{two} \times 2^0 &&= -1.110_{two} \times 2^{-2}
\end{aligned}
$$

Step 1: -1 + (-2) = -3

Using biased representation,

(-1+127) + (-2+127) – 127 = -3 + 127 = 124

Step 2:

$$
\begin{array}{r}
1.000_{two} \\
\times \quad 1.110_{two} \\
\hline
0000 \\
1000 \\
1000 \\
1000 \\
\hline
1110000_{two}
\end{array}
$$

The product is $1.11000 \times 2^{-3} = 1.110 \times 2^{-3}$

Example of Floating-Point Multiplication

Step 3: The product is normalized and there is no overflow and underflow.

Step 4: It is already rounded.

Step 5: The signs of the original operands are different. So the sign of the product is negative. So the Final product is

$$- 1.110 \times 2^{-3}$$

$$= - 0.21875$$