CHAPTER 3

Computer Arithmetic

Learning objectives

- Introduction
- Fixed Point Addition and Subtraction
- Multiplication and Division
- Floating Point Arithmetic
- High Performance Arithmetic

INTRODUCTION

In the second chapter we explained different methods to represent the numbers in a digital computer, but how arithmetic operations will be perform on them. Because without the arithmetic operations the data representation is only the way to store data in the memory. We need high computation on the data and its processing.

In this chapter we cover four basic arithmetic operations: addition, subtraction, multiplication, and division. We begin by describing how these four operations can be performed on fixed point numbers, and continue with a description of how these four operations can be performed on floating point numbers.

3.1. FIXED POINT ADDITION AND SUBTRACTION

The addition and subtraction is the basic arithmetic operations, here we cover addition and subtraction of both signed and unsigned fixed point numbers in detail. Since the two's complement representation of integers is almost universal in today's computers, we will focus primarily on two's complement operations. We will briefly cover operations on 1's complement and BCD numbers, which have a foundational significance for other areas of computing, such as networking (for 1's complement addition) and hand-held calculators (for BCD arithmetic.)

3.1.1. Binary Addition

Binary addition is performed in the same manner as decimal addition. The complete table for binary addition is as follows:

Table 3.1: Binary Addition

A	B	A+B	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

'Carry over' are performed in the same manner as in decimal arithmetic. Since 1 is the largest digit in the binary system, any sum greater than 1 requires that a digit be carried.

EXAMPLE 3.1.

Decimal	Binary	Decimal	Binary
5	101	3 ¼	11.01
+6	+110	+5 ¾	+101.11
11	1011	9	1001.00

EXAMPLE 3.2.

Add 12 and 10 using binary numbers

```
  1                    ← Carry
    1  1  0  0          12
+   1  0  1  0        + 10
-------------        ------
  1  0  1  1  0         22
-------------        ------
```

EXAMPLE 3.3.

Add 22 and 15 using binary numbers

```
    1  1  1             ← Carry
    1  0  1  1  0          22
+   0  1  1  1  1        + 15
-------------------     ------
  1  0  0  1  0  1         37
-------------------     ------
```

EXAMPLE 3.4.

Add 1101.10_2 and 1100.01_2 using binary numbers

```
      1
    1 1 0 1 . 1 0
+   1 1 0 0 . 0 1
-----------------
1   1 0 0 1 . 1 1
```

←Carry

3.1.2. Binary Subtraction

Subtraction is the inverse operation of addition. To subtract, it is necessary to establish a procedure for subtracting a larger from a smaller digit. The only case in which this occurs with binary numbers is when 1 is subtracted from 0. It is necessary to borrow 1 from the next column to the left. This is the binary subtraction table.

Table 3.2: Binary Subtraction

A	B	A-B
0	0	0
1	0	1
0	1	1
1	1	0

With a borrow of 1

EXAMPLE 3.5.

Decimal	Binary
9	1001
-5	-101
4	100

Decimal	Binary
16	10000
-3	-11
13	1111

EXAMPLE 3.6.

Subtract 10 from 12 using binary numbers

```
      1
  1 1 0 0
- 1 0 1 0
---------
  0 0 1 0
```

←Borrow

```
  12
- 10
----
  02
----
```

EXAMPLE 3.7.

Subtract 15 from 22 using binary numbers

```
        1 1 1
  1 0 1 1 0
- 0 1 1 1 1
-----------
  0 0 1 1 1
```

← Borrow

```
  22
- 15
----
  07
----
```

EXAMPLE 3.8.

Subtract 1100.01_2 from 1101.10_2 using binary numbers

← Borrow

	1	1	0	1	.	1	0
−	1	1	0	0	.	0	1
	0	0	0	1	.	0	1

3.1.3. Binary Subtraction 1s Complement

The digital computer can perform subtraction using addition method and this can be done using 1s complement. The following steps are involved to perform subtraction using 1s complement:

Step-1: convert both numbers in the binary form.

Step-2: convert negative number into 1s complement

Step-3: add both numbers

Step-4: add last carry with remaining digits

EXAMPLE 3.9.

Subtract 10 from 12 using 1s complement

$(12)_{10} = 1100_2$

$(10)_{10} = 1010_2$

$(-10)_{10} = 0101$ (1s complement of 1010)

		1			← Carry	
		1	1	0	0	12
+		0	1	0	1	− 10
	1	0	0	0	1	02
	add that			+	1	
		0	0	1	0	

Small Concept

In 1's complement binary subtraction, you must add last carry with remaining digits.

EXAMPLE 3.10.

Subtract 15 from 22 using 1s complement

$(22)_{10} = 10110_2$

$(15)_{10} = 01111_2$

$(-15)_{10} = 10000_2$ (1s complement of 01111)

						← Carry
	1	0	1	1	0	22
+	1	0	0	0	0	− 15
1	0	0	1	1	0	07
add that				+	1	
	0	0	1	1	1	

3.1.4. Binary Subtraction 2s Complement

The digital computer can perform subtraction using addition method and this can be done also using 2s complement. The following steps are involved to perform subtraction using 2s complement:

Step-1: convert both numbers in the binary form.

Step-2: convert negative number into 2s complement

Step-3: add both numbers

Step-4: ignore last carry

EXAMPLE 3.11.

Subtract 10 from 12 using 2s complement

$(12)_{10} = 1100_2$

$(10)_{10} = 1010_2$

$(-10)_{10} = 0101 + 1 = 0110$ (2s complement of 1010)

					←Carry
	1				
	1	1	0	0	12
+	0	1	1	0	− 10
←1	0	0	1	0	02

- Ignore last carry
- answer is = $0010_2 = (2)_{10}$

Small Concept

In 2's complement binary subtraction, you must ignore (remove) last carry.

EXAMPLE 3.12.

Subtract 15 from 22 using 2s complement

$(22)_{10} = 10110_2$

$(15)_{10} = 01111_2$

$(-15)_{10} = 10000 + 1 = 10001_2$ (2s complement of 1010)

		←Carry				
		1	0	1	1	0
+		1	0	0	0	1
←1		0	0	1	1	1

22
− 15
07

- Ignore last carry
- answer is = $00111_2 = (7)_{10}$

3.2. MULTIPLICATION AND DIVISION

Multiplication and division of fixed point numbers can be accomplished with addition, subtraction, and shift operations. The sections that follow describe methods for performing multiplication and division of fixed point numbers in both unsigned and signed forms using these basic operations. We will first cover unsigned multiplication and division, and then we will cover signed multiplication and division.

3.2.1. Multiplication

Multiplication can be performed by repeated addition of the multiplicand to itself multiplier number of times.

Consider the multiplication of two 4-bit numbers :

EXAMPLE 3.13.

Multiply 11 and 13 using binary numbers

Multiplicand $(11)_{10} = 1011_2$

Multiplier $(13)_{10} = 1101_2$

		1	0	1	1	$(11)_{10}$
	×	1	1	0	1	$(13)_{10}$
		1	0	1	1	
	0	0	0	0	×	
1	0	1	1	×		
1	0	1	1	×		
1	0	0	0	1	1	1 1 $(143)_{10}$

In the binary system, multiplication by a power of 2 corresponds to shifting the multiplicand left by one position and hence multiplication can be performed by a series of shift and add operations. The flowchart given below is the method for multiplication of two unsigned binary numbers using shifting and adding the bits.

Small Concept

Computer perform all arithmatic operations using addition, internally.

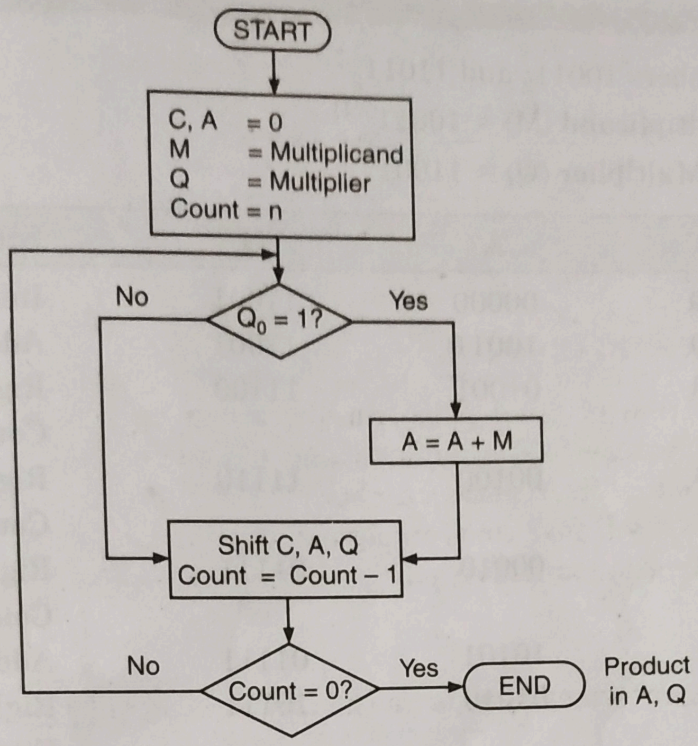

Fig. 3.1. Flowchart for Unsigned Binary Multiplication

Consider the multiplication of two 4-bit numbers:

EXAMPLE 3.14.

Multiply 1101_2 and 1011_2

Multiplicand (M) = 1101

Multiplier (Q) = 1011

Count	C	A	Q	Steps
4	0	0000	1011	Initial Values
	0	1101	1011	Add M to A
3	0	0110	1101	Right Shift C-A-Q Count=Count-1
	1	0011	1101	Add M to A
2	0	1001	1110	Right Shift C-A-Q, Count=Count-1
1	0	0100	1111	Right Shift C-A-Q, Count=Count-1
	1	0001	1111	Add M to A
0	0	1000	1111	Right Shift C-A-Q, Count=Count-1
		1000	**1111**	**Product (13×11=143)**

Answer = (1000 1111)$_2$

EXAMPLE 3.15.

Multiply 5 bit numbers 10011_2 and 11011_2

Multiplicand (M) = 10011

Multiplier (Q) = 11001

Count	C	A	Q	Steps
		00000	11001	Initial Values
5	0	10011	11001	Add M to A
	0	01001	11100	Right Shift C-A-Q
4	0	00100	11110	Count=Count-1
				Right Shift C-A-Q
3	0	00010	01111	Count=Count-1
				Right Shift C-A-Q
2	0	10101	01111	Count=Count-1
	0	01010	10111	Add M to A
1	0			Right Shift C-A-Q
				Count=Count-1
	0	11101	10111	Add M to A
0	0	01110	11011	Right Shift C-A-Q
				Count=Count-1
	01110	11011	Product = 475	

$$\begin{array}{rl} 10011 & (19)_{10} \\ \times\ 11001 & \times\ (25)_{10} \\ \hline 0111011011 & (475)_{10} \end{array}$$

Answer = $(0111011011)_2$

3.2.2. Division

In the binary division, we must successively subtract the divisor from the dividend, using the fewest number of bits in the dividend as we can. In the given example we shown how to perform the division on two binary numbers.

EXAMPLE 3.16.

Divide 0111 by 11.

$$\begin{array}{r|l} 11 & 0111 \\ & 11 \\ \hline & 01 \end{array} \quad 0010$$

EXAMPLE 3.17.

Divide 011101 by 11.

$$\begin{array}{r|l} 111 & 011101 \\ & 11 \\ \hline & 01 \end{array} \quad 00100$$

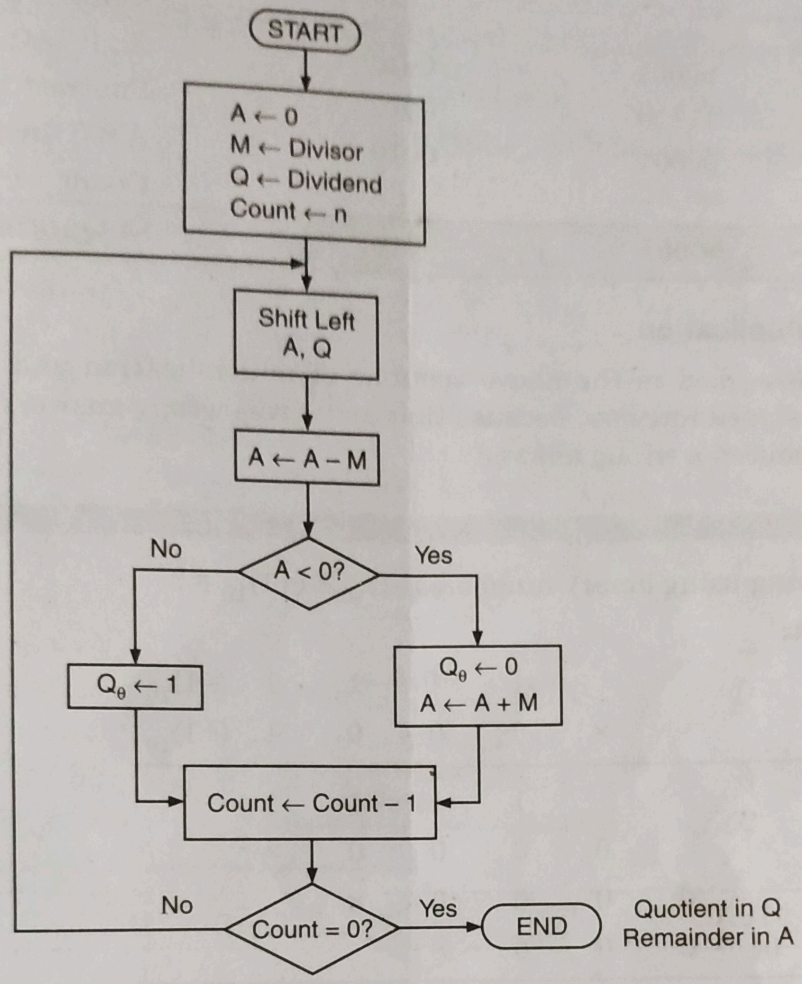

Fig. 3.2. Division flow chart

Divisor (M) = 00011

Dividend (Q) = 0111

2s Complement of M for (–M) =11101

Count	A	Q	Steps
4	00000	0111	
	00000	1110	Shift Left
	11101	1110	Subtract M from A
	00000	1110	A < 0 then Add M to A
3			Count=Count-1
	00001	1100	Shift Left
	11110	1100	Subtract M from A
	00001	1100	A < 0 then Add M to A
2			Count = Count-1
	00011	1000	Shift Left
	00000	1000	Subtract M from A
	00000	1001	A > 0 then Set Q_0=1

1		0010	Count = Count-1
	00001	0010	Shift Left
	11110	0010	Subtract M from A
	00001		A < 0 then Add M to A
0			Count = Count-1
Reminder ←	**00001**	**0010**	**→ Quotient**

3.2.3. Signed Multiplication

The method described in the above sections of multiplication and division, we can't implement on the signed integers, because that may gives wrong answers. Lets consider the example which produced a wrong answer.

EXAMPLE 3.18.

Multiply following using binary numbers $(-1)_{10} \times (+1)_{10} = ?$

Using four bit:

				1	1	1	1	$(-1)_{10}$
			×	0	0	0	1	$(+1)_{10}$
				1	1	1	1	
			0	0	0	0	×	
		0	0	0	0	×		
	0	0	0	0	×			
0	0	0	0	1	1	1	1	$(15)_{10}$

(Wrong Answer; it should be -1, use following method)

Using Eight bit

1	1	1	1	1	1	1	1	$(-1)_{10}$
			×	0	0	0	1	$(+1)_{10}$
1	1	1	1	1	1	1	1	
0	0	0	0	0	0	0	×	
0	0	0	0	0	0	×		
0	0	0	0	0	×			
1	1	1	1	1	1	1	1	$(-1)_{10}$

As you saw in the above example 3.18, we multiply –1 by +1 using four-bit words. The eight-bit equivalent of +15 is produced instead of –1. What went wrong is that the sign bit did not get extended to the left of the result. This is not a problem for a positive result because the high order bits default to 0, producing the correct sign bit 0.

A solution is shown in the another method, in which each partial product is extended to the width of the result, and only the rightmost eight bits of the result are retained. If both operands are negative, then the signs are extended for both operands, again retaining only the rightmost eight bits of the result.

3.2.4. Booth's Multiplication Algorithm

Booth algorithm gives a procedure for multiplying binary integers in signed 2s complement representation. The algorithm was invented by Andrew Donald Booth in 1950 while doing research on crystallography at Birkbeck College in Bloomsbury, London. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth's algorithm examines adjacent pairs of bits of the N-bit multiplier Q in signed two's complement representation, including an implicit bit below the least significant bit. Booth's algorithm performs fewer additions and subtractions than the normal multiplication algorithm.

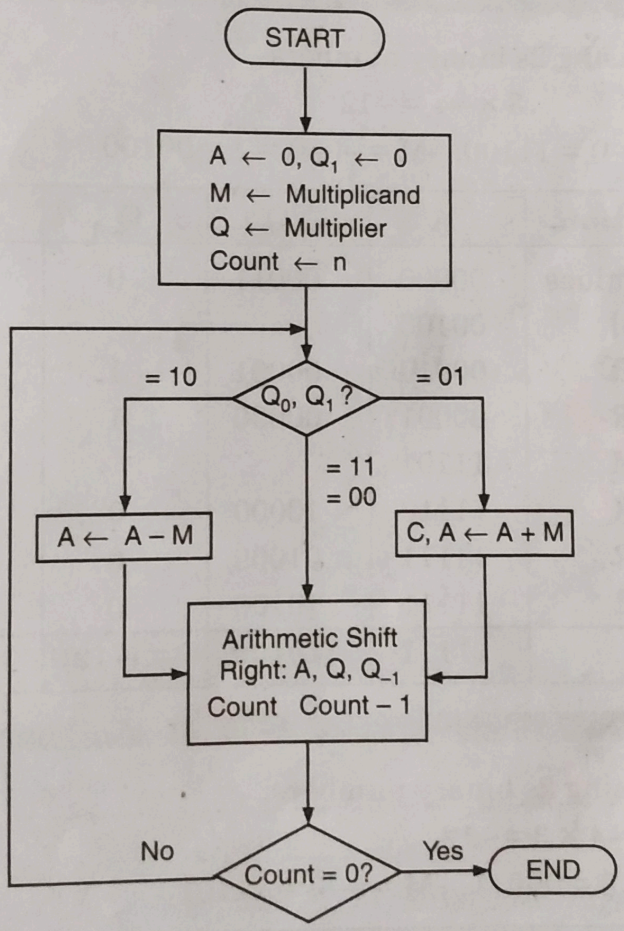

Fig. 3.3. Flow chart of Booth's multiplying

EXAMPLE 3.19.

Multiply (+4) × (–5) using 2s binary numbers.

$$4 \times -5 = -20$$

$$Q = 4 = 00100,$$

$$M = (-5) = 11011,$$

$$-M = -(-5) = 5 = 00101$$

Operation	A	Q	Q_{-1}	Count
Initial values	00000	00100	0	5
SHR	00000	00010	0	4
SHR	00000	00001	0	3
A–M	00101			
SHR	00010	10000	1	2
A+M	11101			
SHR	11110	11000	0	1
SHR	11111	01100	0	0
	11111	01100	= (–20)	

M = 11011

EXAMPLE 3.20.

Multiply (+3) × (–4) using 2s binary numbers.

$$3 \times -4 = -12$$

Q = 3 = 00011, M = (–4) = 11100, –M = –(–4) = 4 = 00100

Operation	A	Q	Q_{-1}	Count
Initial values	00000	00011	0	5
A–M	00100			
SHR	00010	00001	1	4
SHR	00001	00000	1	3
A+M	11101			
SHR	11110	10000	0	2
SHR	11111	01000	0	1
SHR	11111	10100	0	0
	11111	**10100**	**= (–12)**	

EXAMPLE 3.21.

Multiply (+4) × (–3) using 2s binary numbers.

$$-4 \times 3 = -12$$

Q = (–4) = 11100, M = 3 = 00011, –M = (–3) = 11101

Operation	A	Q	Q_{-1}	Count
Initial values	00000	11100	0	5
SHR	00000	01110	0	4
SHR	00000	00111	0	3
A–M	11101			
SHR	11110	10011	1	2
SHR	11111	01001	1	1
SHR	11111	10100	0	0
	11111	**10100**	**= (–12)**	

Signed division is more difficult. We will not explore the methods here, but as a general technique, we can convert the operands into their positive forms, perform the division, and then convert the result into its true signed form as a final step.

3.3. FLOATING POINT ARITHMETIC

All the arithmetic operations such as addition, subtraction, multiplication and division on floating point numbers can be carried out using the fixed point arithmetic operations described in the previous sections, with attention given to maintaining aspects of the floating point representation. In this section, we explore floating point arithmetic in base 2 and base 10, keeping the requirements of the floating point representation in mind.

3.3.1. Floating Point Addition

Floating point addition is differs from integer addition in floating point addition, exponents must be handled as well as the magnitudes of the operands. As in general (in decimal number system base 10) addition using scientific notation, the exponents of the operands must be made equal for addition. The fractions are then added as appropriate, and the result is normalized. This process of adjusting the fractional part, and also rounding the result can lead to a loss of precision in the result. Consider the unsigned floating point addition:

$$0.101 \times 2^3 + 0.111 \times 2^4$$

In the above addition the fractions have three significant digits.

First you have to adjust the smaller exponent to be equal to the larger exponent, and adjust the fraction accordingly. Thus

$$0.101 \times 2^3 = 0.010 \times 2^4$$

Here we loss 0.001×23 of precision in the process.

The resulting sum is

$$(0.010 + 0.111) \times 2^4 = 1.001 \times 2^4 = 0.1001 \times 2^5$$

and rounding to three significant digits, 0.100×2^5, and we have lost another 0.001×2^4 in the rounding process.

Binary floating-point addition and subtraction are done the same as using pencil and paper. The first thing that we do is expressing both operands in the same exponential power, and then adds the numbers, keeping the exponent in the sum. The floating point conversion is explored in the chapter-2 with floating point representation.

If the exponent needs adjustment, fix it at the end.

EXAMPLE 3.22.

- Find the sum of 12_{10} and 1.25_{10} using the 14-bit floating-point model.

We find $12_{10} = 0.1100 \times 2^4$

And $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$

	0	10100	11000000
+	0	10100	00010100
	0	10100	11010100

Thus, the sum is 0.1101×2^4.

3.3.2. Floating Point Subtraction

Floating point subtraction is same as the floating point addition. As in general (in decimal number system base 10) subtraction using scientific notation, the exponents of the operands must be made equal for subtraction same as addition. The fractions are then subtracted as appropriate, and the result is normalized. This process of adjusting the fractional part and also rounding the result can lead to a loss of precision in the result. Consider the unsigned floating point subtraction:

$$0.111 \times 2^4 - 0.101 \times 2^3$$

In the above subtraction the fractions have three significant digits.

First you have to adjust the smaller exponent to be equal to the larger exponent, and adjust the fraction accordingly. Thus

$$0.101 \times 2^3 = 0.010 \times 2^4,$$

Here we loss 0.001×2^3 of precision in the process.

The resulting subtraction is

$$(0.111 - 0.010) \times 2^4 = 0.101 \times 2^4$$

and rounding to three significant digits, 0.101×2^4.

Binary floating-point subtractions are done the same as using pencil and paper. The first thing that we do is expressing both operands in the same exponential power, and then subtract the numbers, keeping the exponent same. The floating point conversion is explored in the chapter-2 with floating point representation.

If the exponent needs adjustment, fix it at the end.

EXAMPLE 3.23.

- Find the subtraction of 12_{10} and 1.25_{10} using the 14-bit floating-point model.

We find $12_{10} = 0.1100 \times 2^4$

And $1.25_{10} = 0.101 \times 2^1$

$= 0.000101 \times 2^4$

	0	10100	11000000
−	0	10001	00100000
	0	10101	10101100

Thus, our sum is 0.101011×2^4.

Steps Required to Add/Subtract Two Floating-Point Numbers :

1. Compare the magnitude of the two exponents and make suitable alignment to the number with the smaller magnitude of exponent.
2. Perform the addition/subtraction.
3. Perform normalization by shifting the resulting mantissa and adjusting the resulting exponent.

3.3.3. Floating Point Multiplication and Division

Floating point multiplication and division are performed in a manner similar to floating point addition and subtraction, except that the sign, exponent, and fraction of the result can be computed separately. If the operands have the same sign, then the sign of the result is positive. Unlike signs produce a negative result. The exponent of the result before normalization is obtained by adding the exponents of the source operands for multiplication, or by subtracting the divisor exponent from the dividend exponent for division. The fractions are multiplied or divided according to the operation, followed by normalization. Multiply the two operands and add exponents. If the exponent requires adjustment, fix it at the end.

EXAMPLE 3.24.

Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model.

We find $12_{10} = 0.1100 \times 2^4$.

And $1.25_{10} = 0.101 \times 2^1$.

	0	10100	11000000
×	0	10001	10100000
	0	10101	01111000

Thus, our product is

$$0.0111100 \times 2^5 = 0.1111 \times 2^4.$$

The normalized product requires an exponent of $22_{10} = 10110_2$.

Division

Now consider using three-bit fractions in performing the base 2 computation:

$$(+.110\ 2^5) / (+.100\ 2^4).$$

The source operand signs are the same, which means that the result will have a positive sign. We subtract exponents for division, and so the exponent of the result is $5 - 4 = 1$.

We divide fractions, which can be done in a number of ways. If we treat the fractions as unsigned integers, then we will have 110/100 = 1 with a remainder of 10.

What we really want is a contiguous set of bits representing the fraction instead of a separate result and remainder, and so we can scale the dividend to the left by two positions, producing the result:

$$11000/100 = 110.$$

We then scale the result to the right by two positions to restore the original scale factor, producing 1.1. Putting it all together, the result of dividing $(+.110 \times 2^5)$ by $(+.100 \times 2^4)$ produces $(+1.10 \times 2^1)$. After normalization, the final result is $(+.110 \times 2^2)$.

3.4. HIGH PERFORMANCE ARITHMETIC

Various methods are exists for speeding the process of arithmetic operations. Here in this section we emphasis to speed up the process of multiplication. Two most popular methods are described in the sections below. The first approach, which gains the performance by skipping over blocks of 1's and eliminates addition steps. Next is a parallel multiplier, in which a cross product among all pairs of multiplier and multiplicand bits is formed. The result of the cross product is summed by rows to produce the final product.