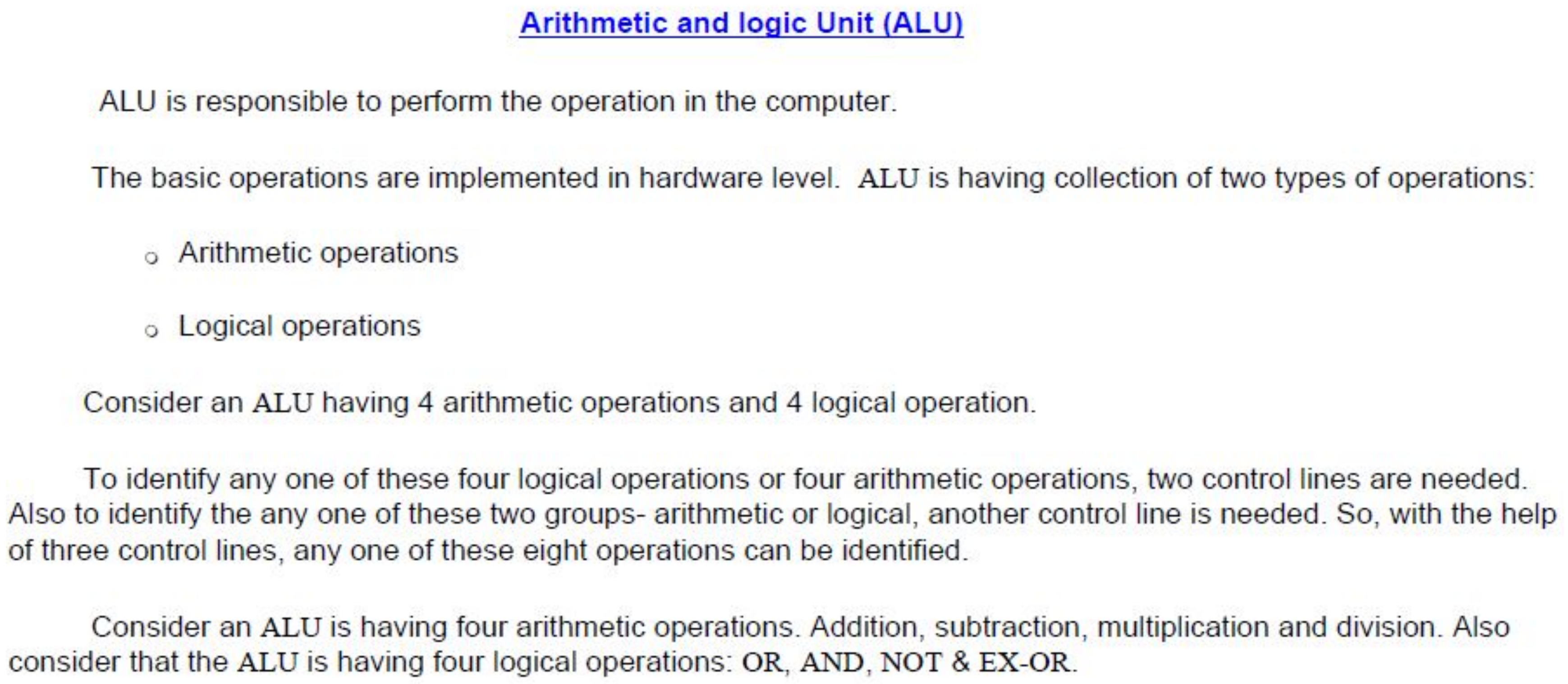
Computer Arithmetic

LECTURE:2

Arithmetic and logic Unit (ALU)



Arithmetic and logic Unit (ALU)

ALU is responsible to perform the operation in the computer.

The basic operations are implemented in hardware level. ALU is having collection of two types of operations:

- Arithmetic operations
- Logical operations

Consider an ALU having 4 arithmetic operations and 4 logical operation.

To identify any one of these four logical operations or four arithmetic operations, two control lines are needed. Also to identify the any one of these two groups- arithmetic or logical, another control line is needed. So, with the help of three control lines, any one of these eight operations can be identified.

Consider an ALU is having four arithmetic operations. Addition, subtraction, multiplication and division. Also consider that the ALU is having four logical operations: OR, AND, NOT & EX-OR.

Arithmetic and logic Unit (ALU)

We need three control lines to identify any one of these operations. The input combination of these control lines are shown below:

Control line C_2 is used to identify the group: logical or arithmetic, ie

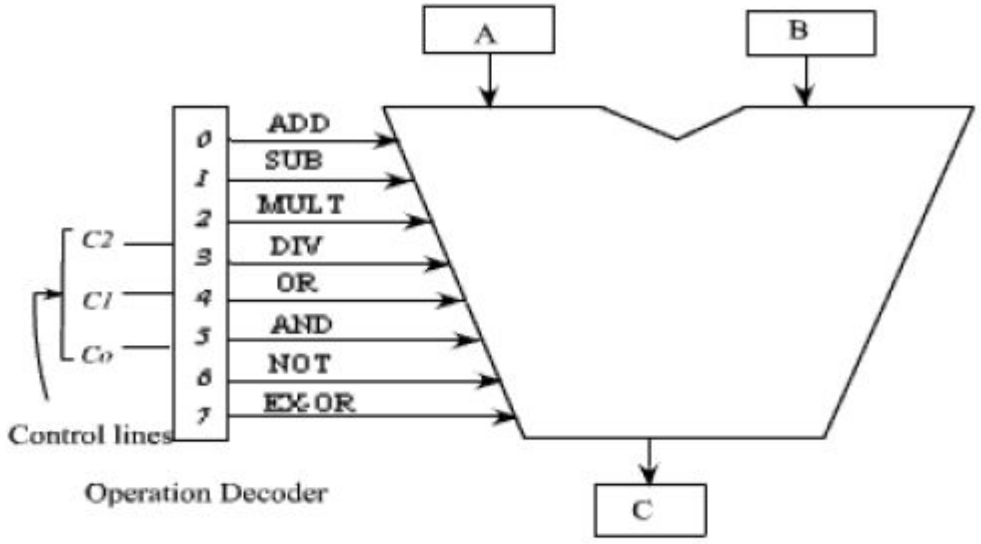
$C_2 = 0$: arithmetic operation $C_2 = 1$: logical operation.

Control lines C_0 and C_1 are used to identify any one of the four operations in a group. One possible combination is given here.

C_1	C_0	Arithmetic $C_2 = 0$	Logical $C_2 = 1$
0	0	Addition	OR
0	1	Subtraction	AND
1	0	Multiplication	NOT
1	1	Division	EX-OR

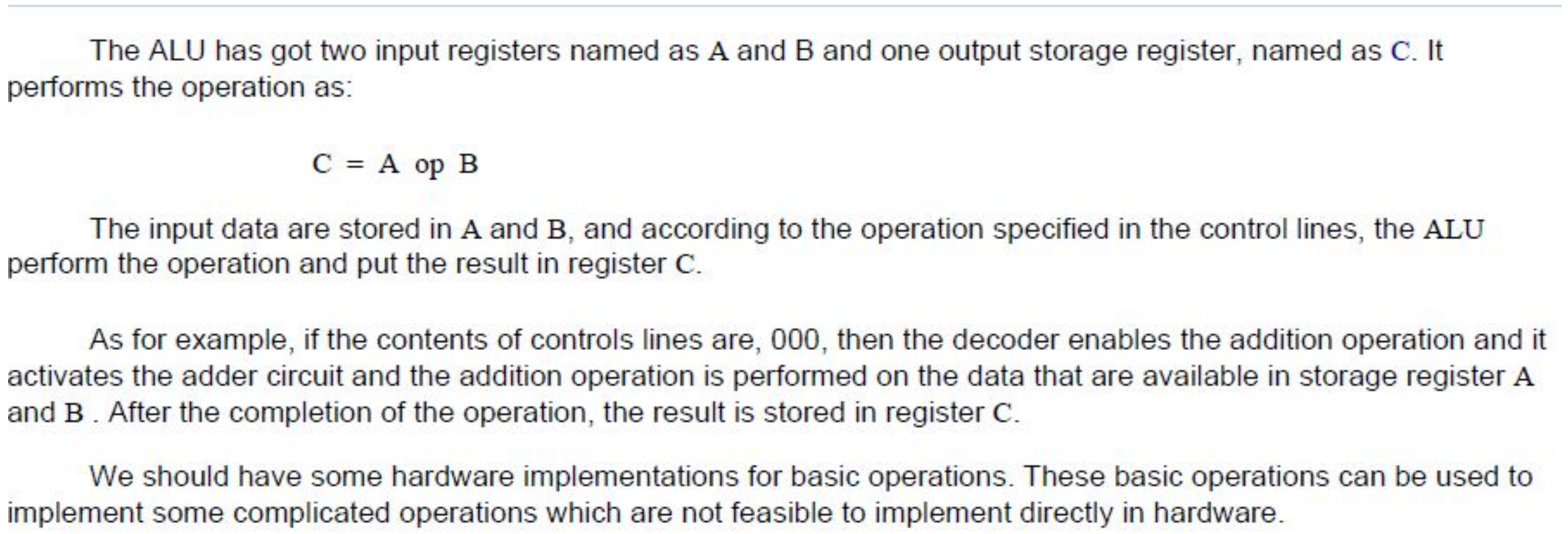
Arithmetic and logic Unit (ALU)

A 3×8 decode is used to decode the instruction. The block diagram of the ALU is shown in the figure.



Block Diagram of the ALU

Arithmetic and logic Unit (ALU)



The ALU has got two input registers named as A and B and one output storage register, named as C. It performs the operation as:

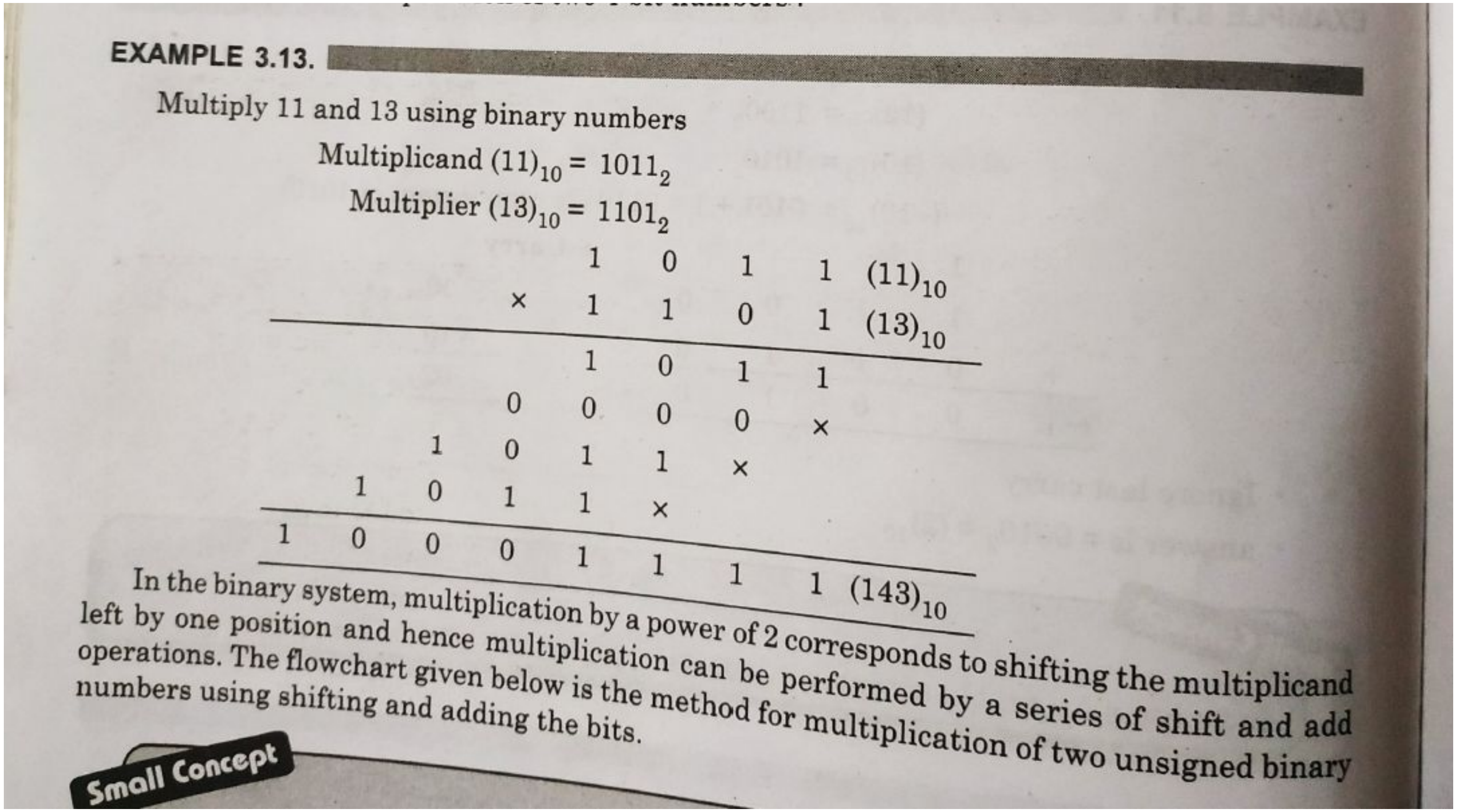
$$C = A \text{ op } B$$

The input data are stored in A and B, and according to the operation specified in the control lines, the ALU perform the operation and put the result in register C.

As for example, if the contents of controls lines are, 000, then the decoder enables the addition operation and it activates the adder circuit and the addition operation is performed on the data that are available in storage register A and B . After the completion of the operation, the result is stored in register C.

We should have some hardware implementations for basic operations. These basic operations can be used to implement some complicated operations which are not feasible to implement directly in hardware.

Unsigned Multiplication



EXAMPLE 3.13.

Multiply 11 and 13 using binary numbers

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

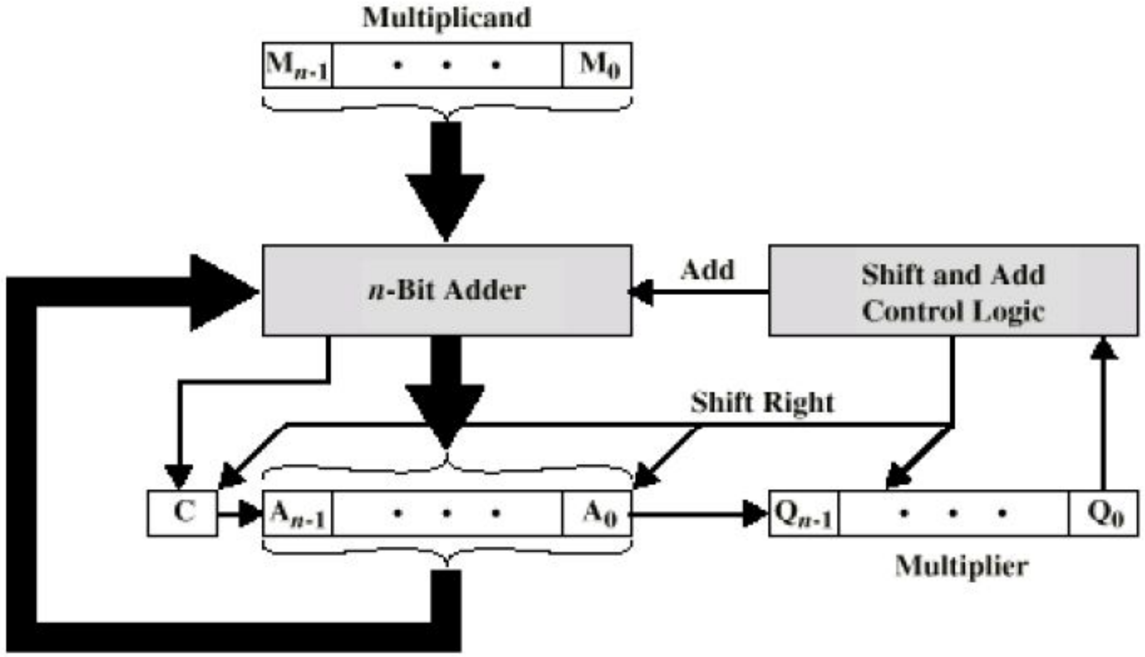
Multiplier $(13)_{10} = 1101_2$

$$\begin{array}{cccccccccc}
 & & & & 1 & 0 & 1 & 1 & (11)_{10} \\
 & & & \times & 1 & 1 & 0 & 1 & (13)_{10} \\
\hline
 & & & & 1 & 0 & 1 & 1 & \\
 & & & 0 & 0 & 0 & 0 & \times & \\
 & & 1 & 0 & 1 & 1 & \times & & \\
 & 1 & 0 & 1 & 1 & \times & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & (143)_{10} \\
\hline
\end{array}$$

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.

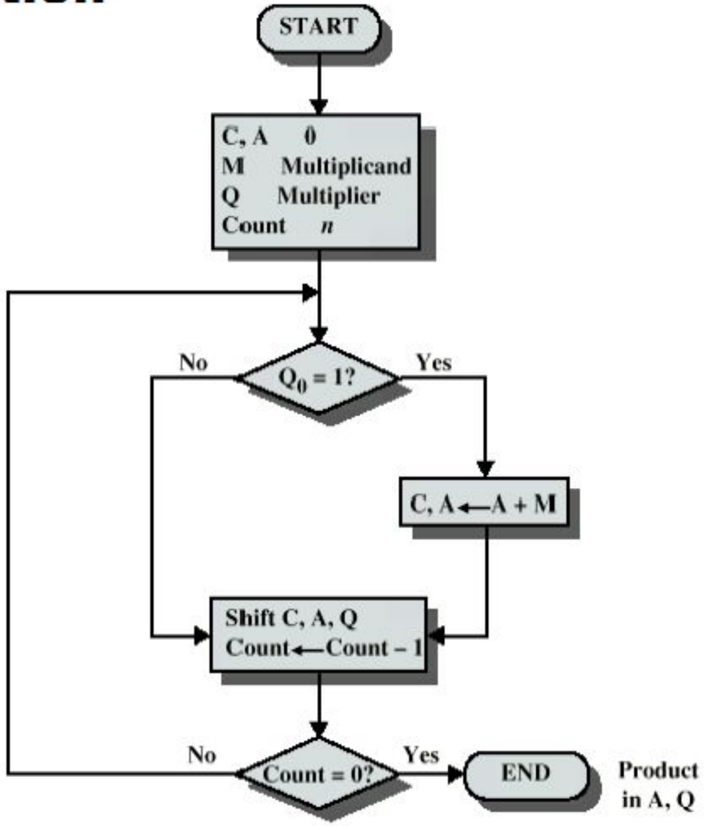
Small Concept

Unsigned Binary Multiplication

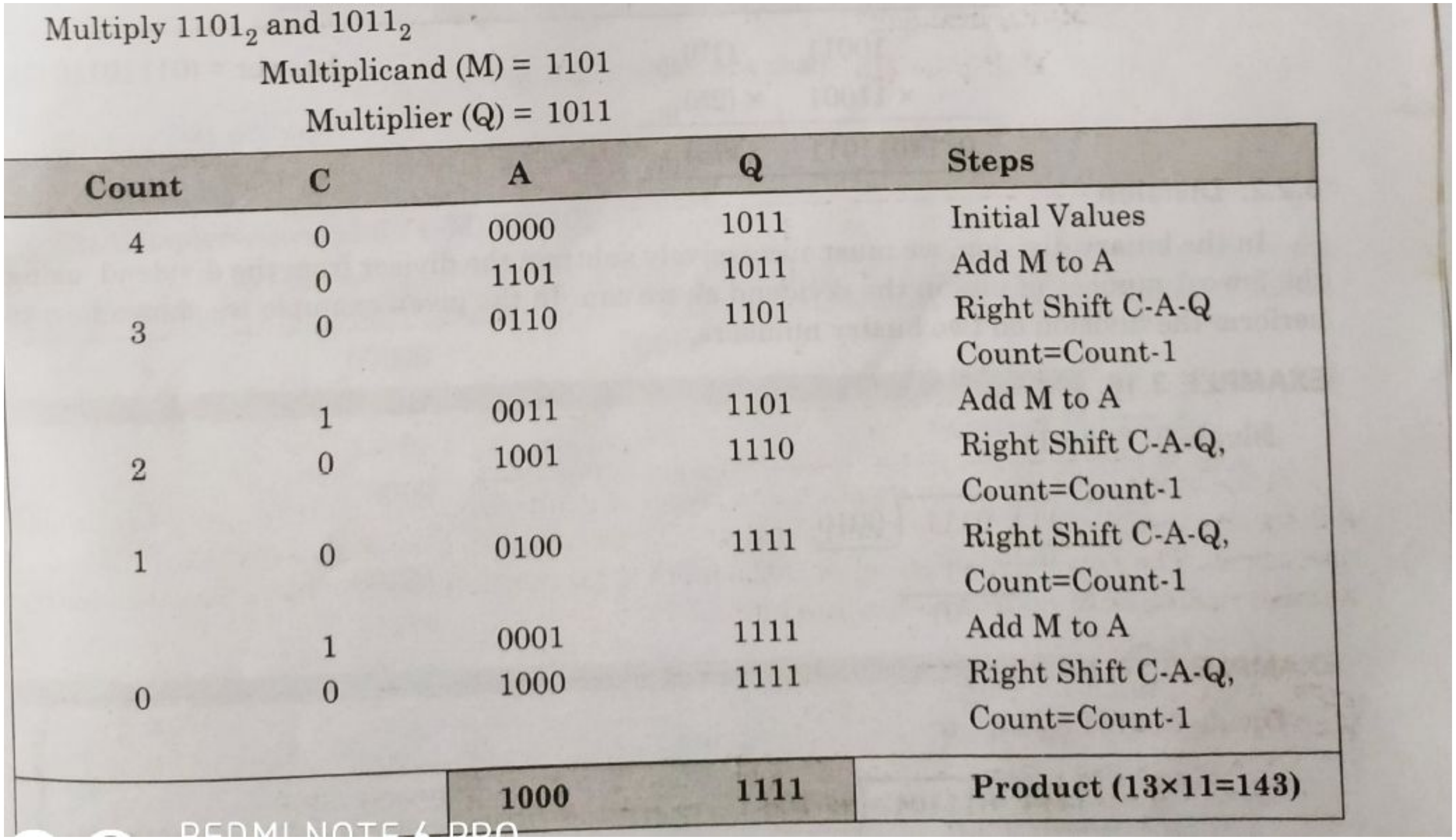


Block Diagram

Flowchart for Unsigned Binary Multiplication



Example



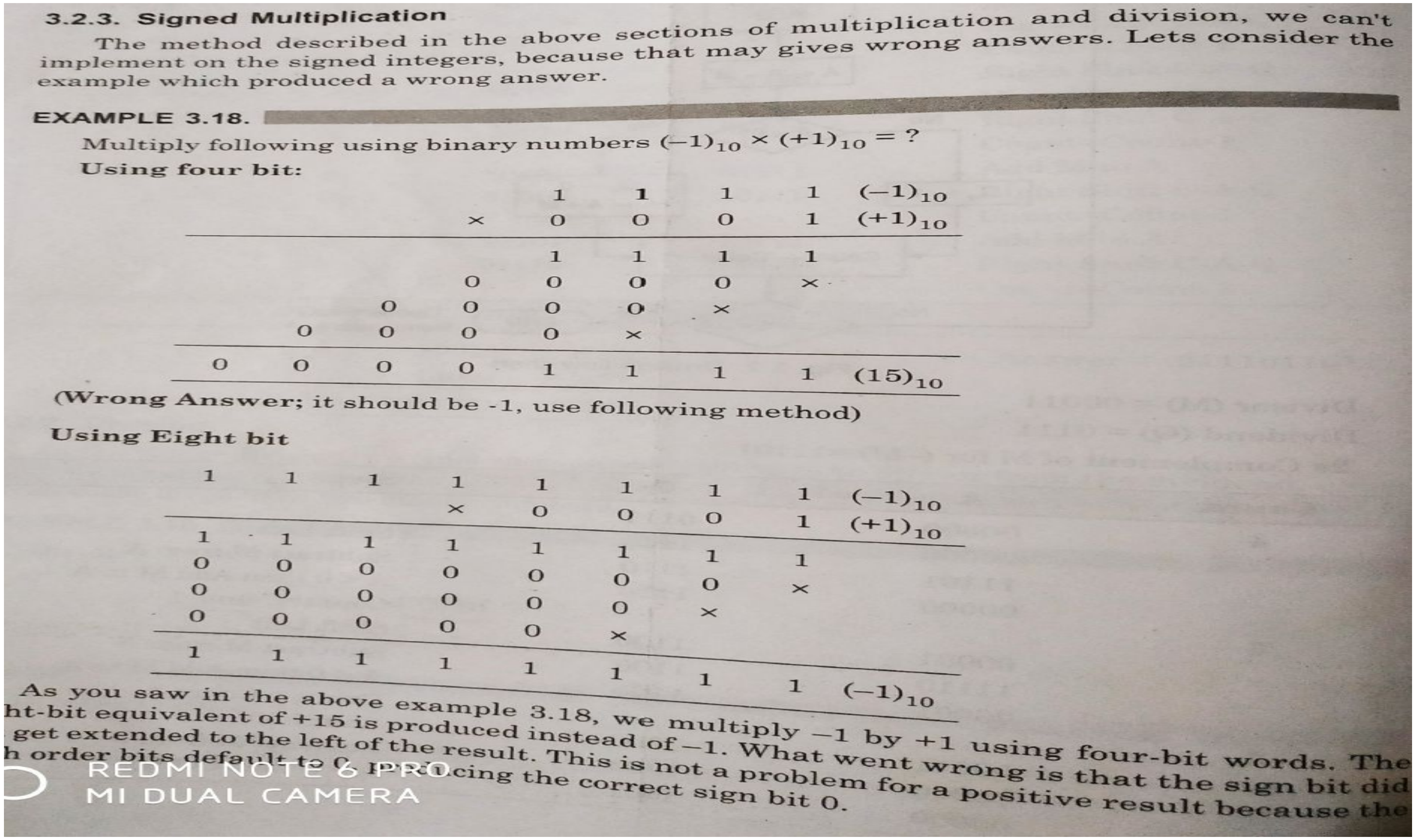
Multiply 1101_2 and 1011_2

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)**

Sign Multiplication



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
----------------------------------------
```

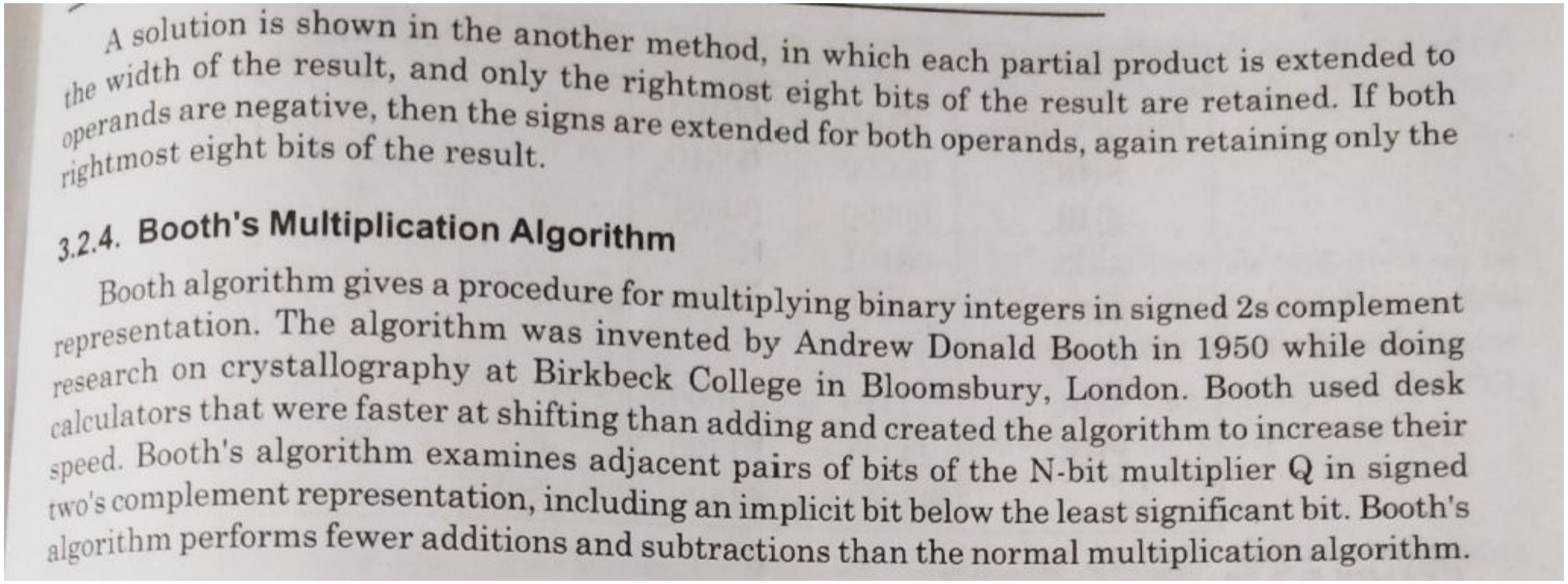
(Wrong Answer; it should be -1, use following method)

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 ht-bit equivalent of +15 is produced instead of −1. What went wrong is that the sign bit did get extended to the left of the result. This is not a problem for a positive result because the h order bits default to 0, producing the correct sign bit 0.

Booth's Algorithm

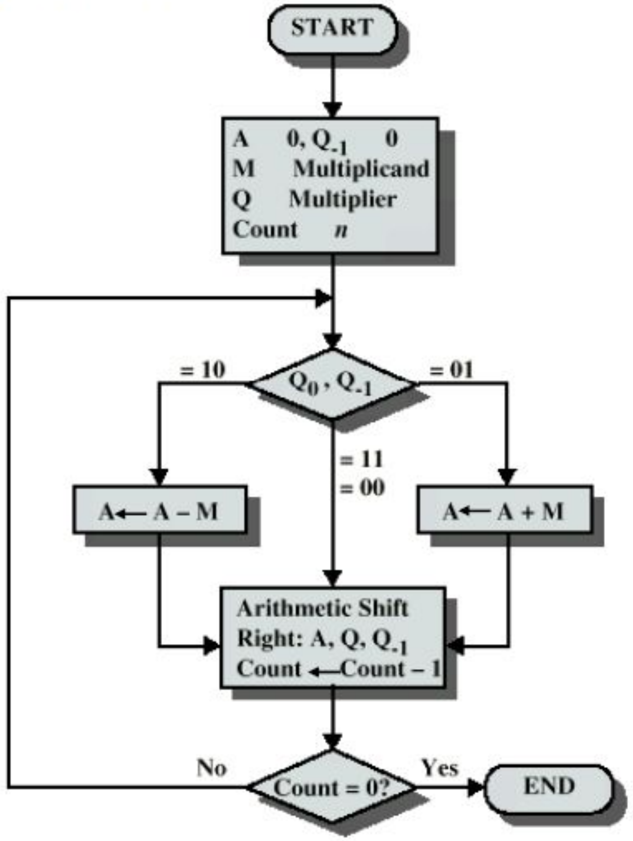


A solution is shown in the another method, in which each partial product is extended to the width of the result, and only the rightmost eight bits of the result are retained. If both operands are negative, then the signs are extended for both operands, again retaining only the rightmost eight bits of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

Booth's Algorithm



EXAMPLE 3.19.

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

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

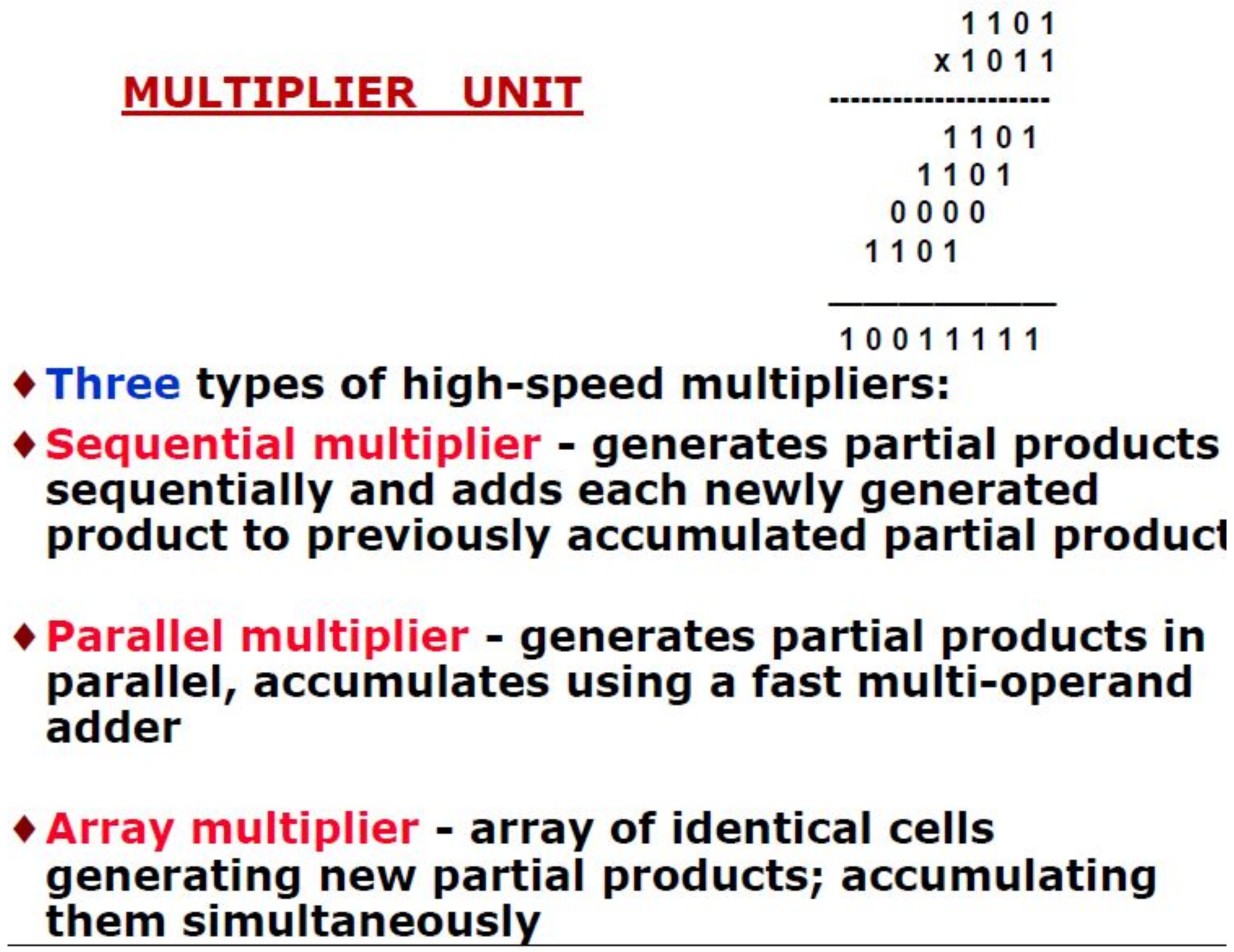
$$Q = 4 = 00100,$$

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

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

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)	

Multiplier unit

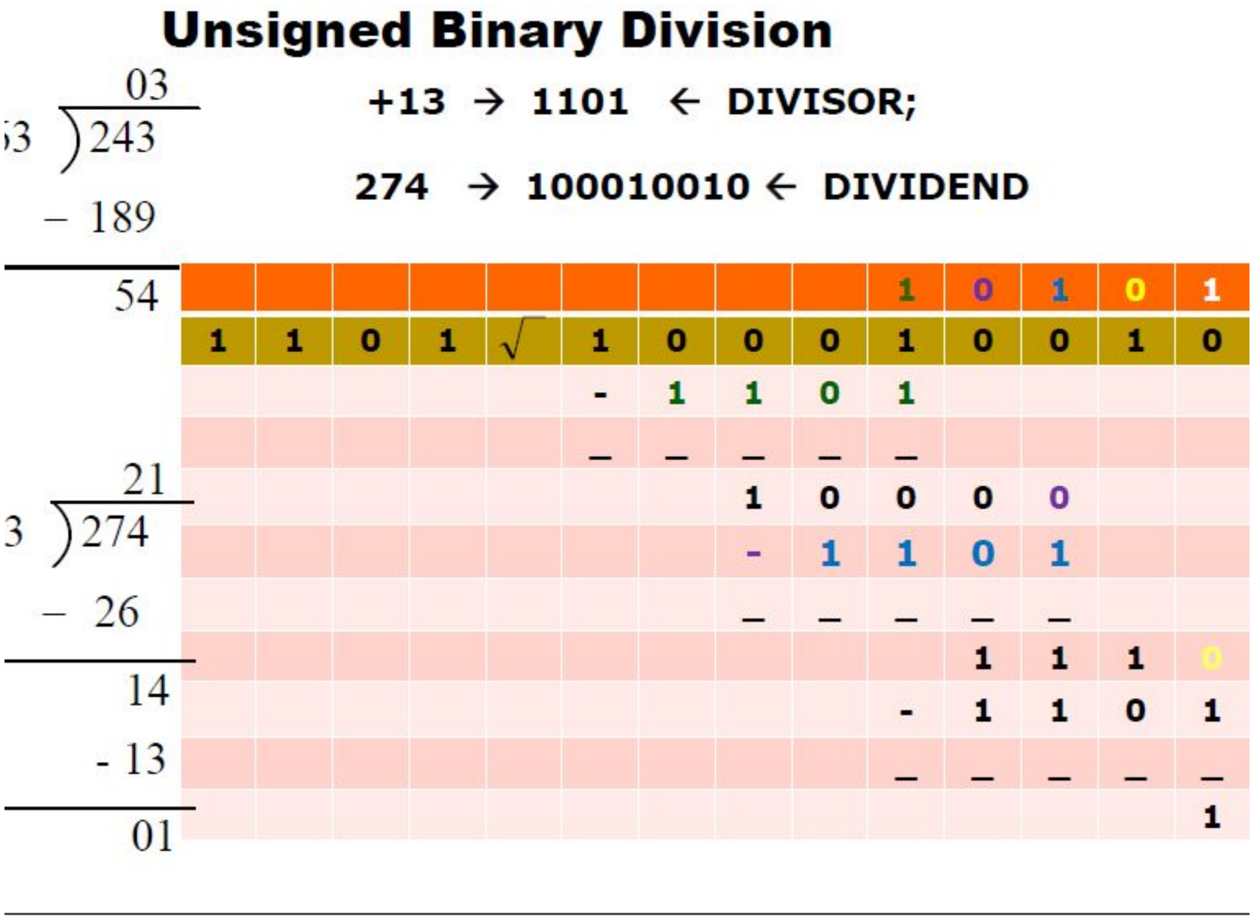


MULTIPLIER UNIT

```
       1 1 0 1
     x 1 0 1 1
 --------------------
      1 1 0 1
     1 1 0 1
    0 0 0 0
   1 1 0 1
 ____________
 1 0 0 1 1 1 1 1
```

- **Three types of high-speed multipliers:**
- **Sequential multiplier - generates partial products sequentially and adds each newly generated product to previously accumulated partial product**
- **Parallel multiplier - generates partial products in parallel, accumulates using a fast multi-operand adder**
- **Array multiplier - array of identical cells generating new partial products; accumulating them simultaneously**

Unsigned Binary Division



Unsigned Binary Division

+13 → 1101 ← DIVISOR;

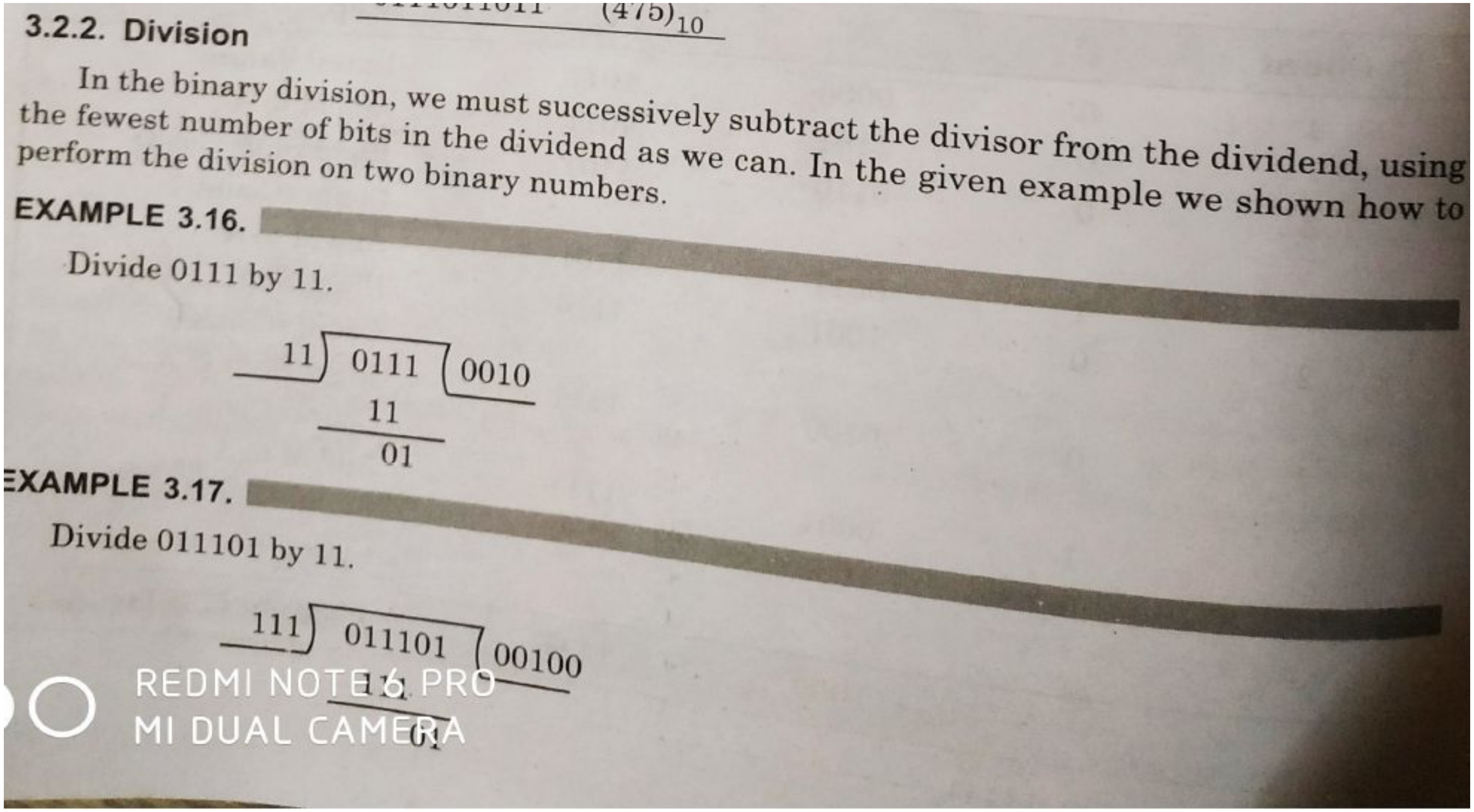
274 → 100010010 ← DIVIDEND

```
    03
3 )243
 − 189
    54

    21
3 )274
 −  26
    14
  - 13
    01
```

									1	0	1	0	1
1	1	0	1	√	1	0	0	0	1	0	0	1	0
					-	1	1	0	1				
					_	_	_	_	_				
							1	0	0	0	0		
							-	1	1	0	1		
							_	_	_	_	_		
										1	1	1	0
									-	1	1	0	1
									_	_	_	_	_
													1

Division



$(475)_{10}$

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.

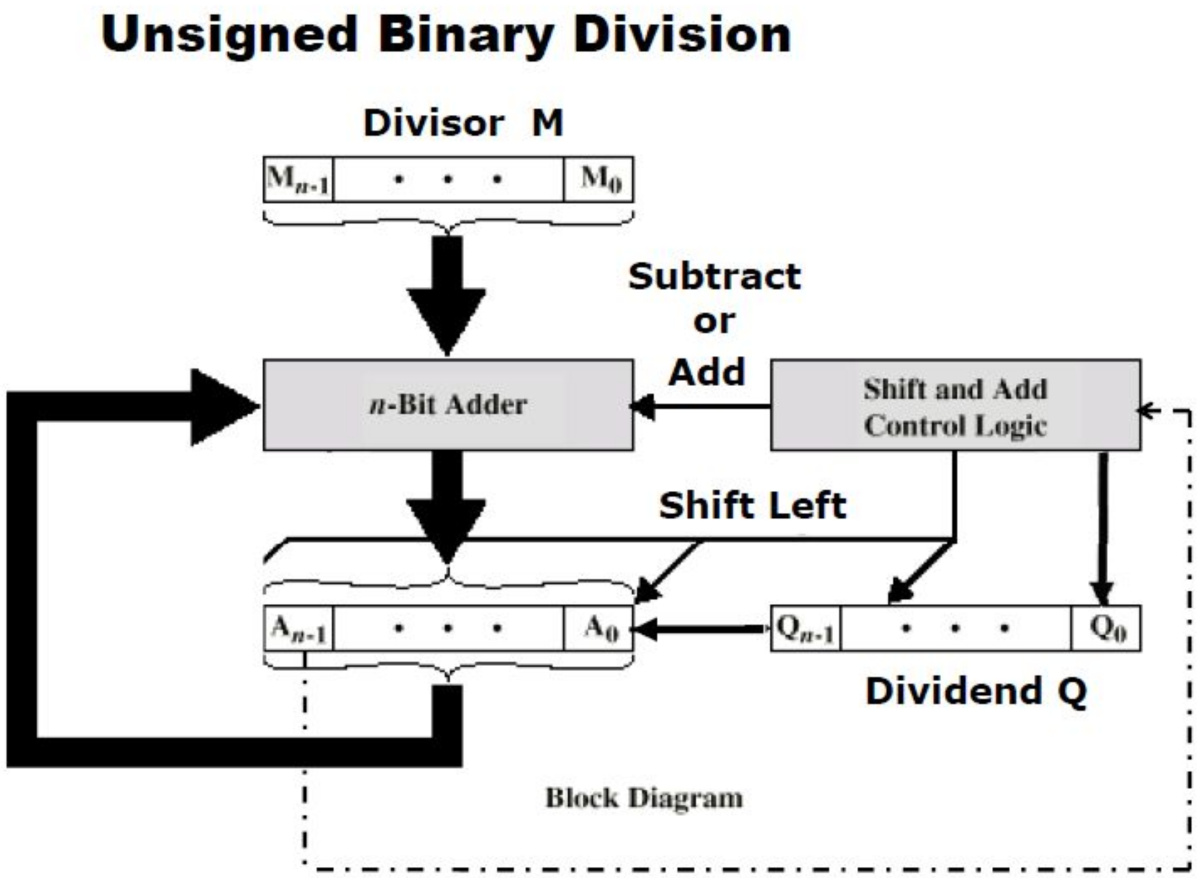
```
11 ) 0111 ( 0010
     11
     01
```

EXAMPLE 3.17.

Divide 011101 by 11.

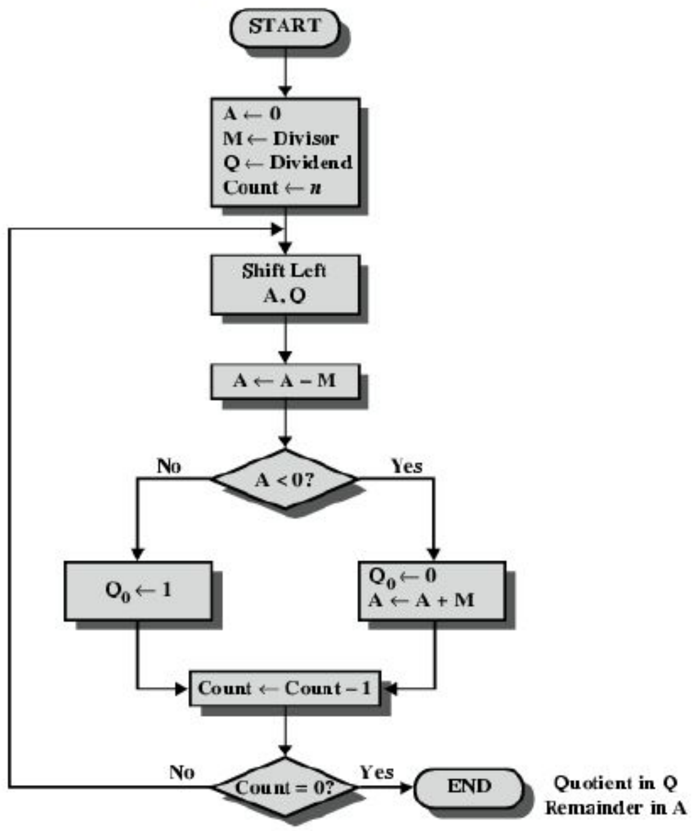
```
111 ) 011101 ( 00100
      11
      01
```

Unsigned Binary Division



Unsigned Binary Division

Flowchart for Unsigned Binary Division



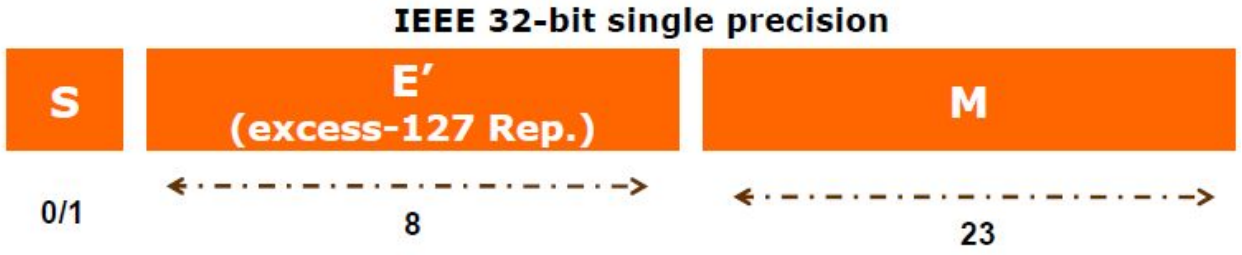
Divisor (M) = 00011

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

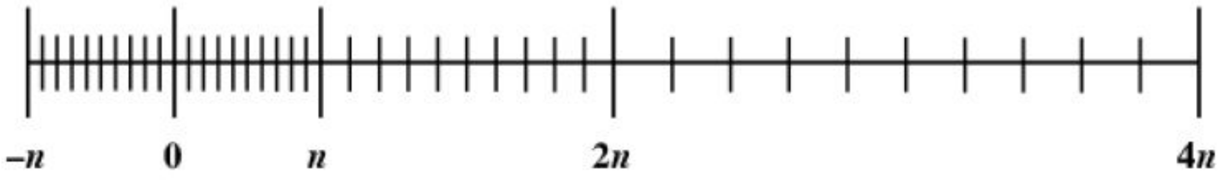
Floating Point Numbers



IEEE 64-bit single precision

|E| = 11 |M| = 52

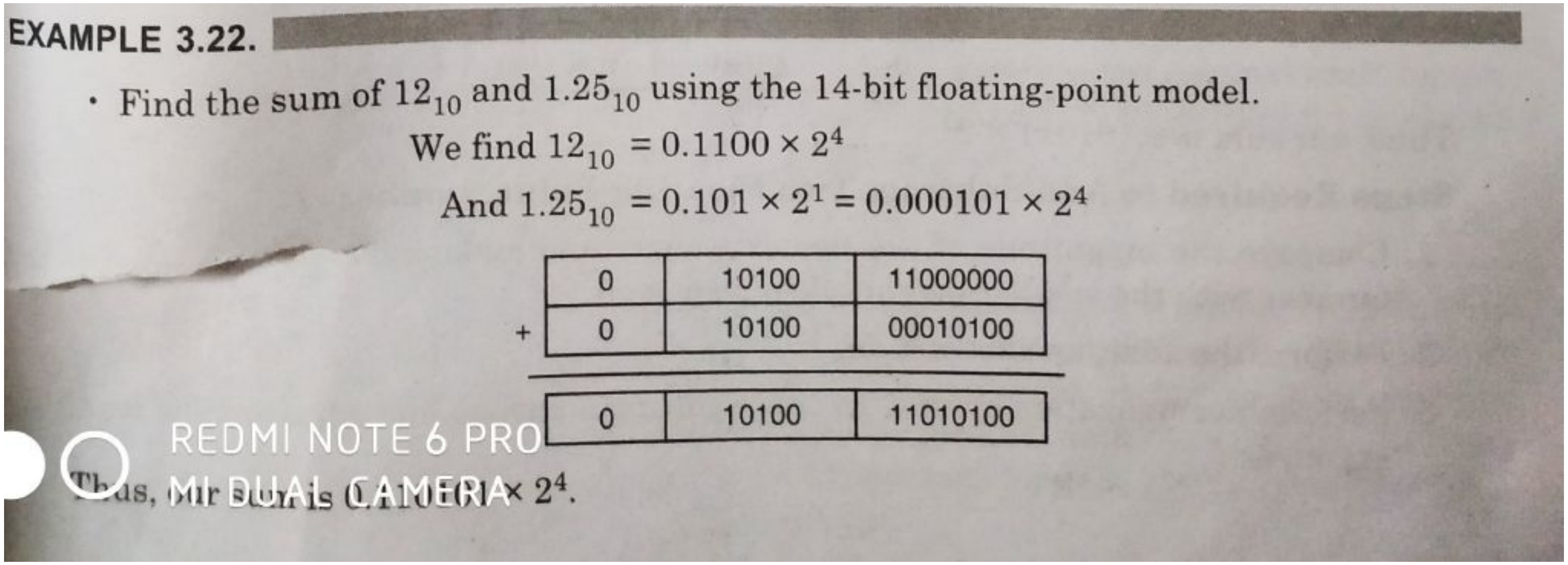
Density of Floating Point Numbers



Floating-Point Formats of Three Machines

	IBM/370	DEC/VAX	Cyber 70
Word length (double)	32 (64) bits	32 (64) bits	60 bits
Significand+{hidden bit}	24 (56) bits	$23+1$ $(55+1)$ bits	48 bits
Exponent	7 bits	8 bits	11 bits
Bias	64	128	1024
Base	16	2	2
Range of M	$\frac{1}{16} \le M < 1$	$\frac{1}{2} \le M < 1$	$1 \le M < 2$
Representation of M	Signed-magnitude	Signed-magnitude	One's complement
Approximate range	$16^{63} \approx 7 \cdot 10^{75}$	$2^{127} \approx 1.9 \cdot 10^{38}$	$2^{1023} \approx 10^{307}$
Approximate resolution	$2^{-24} \approx 10^{-7}\,(10^{-17})$	$2^{-24} \approx 10^{-7}\,(10^{-17})$	$2^{-48} \approx 10^{-14}$

Floating point sum



EXAMPLE 3.22.

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

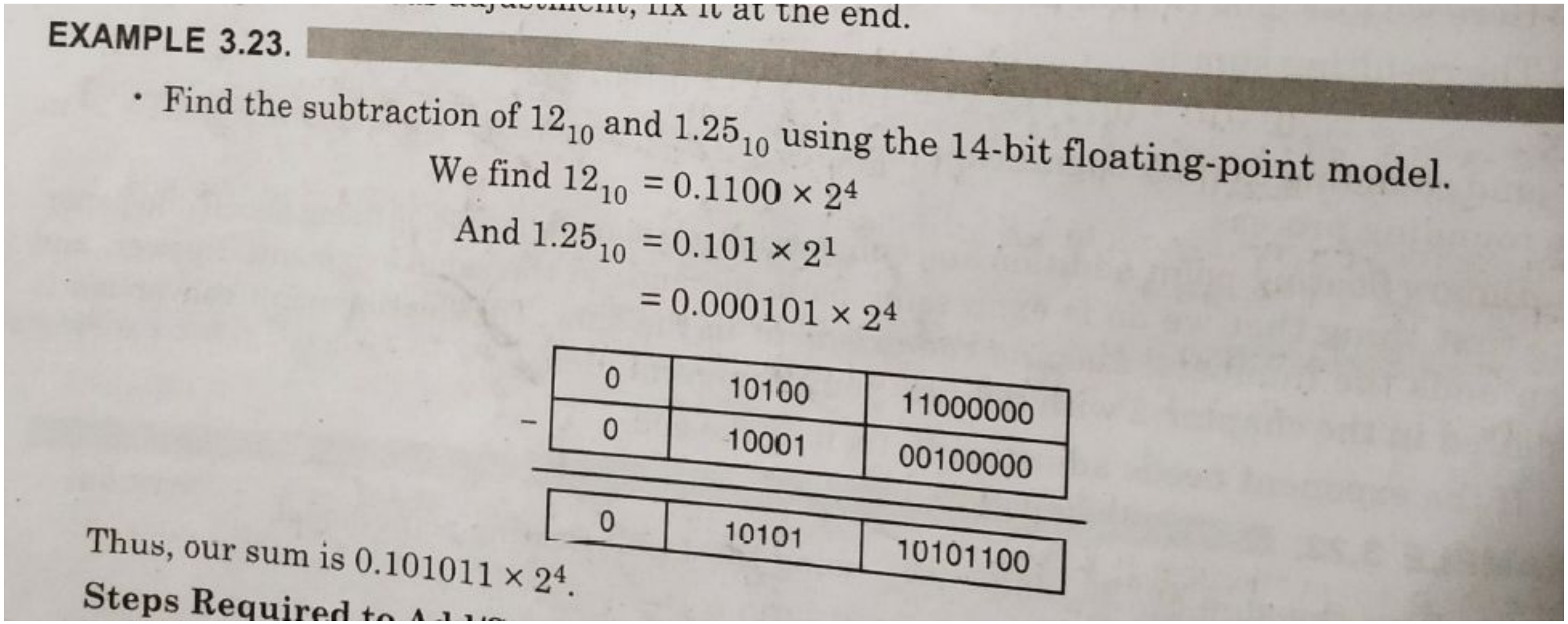
We find $12_{10} = 0.1100 \times 2^4$

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

	0	10100	11000000
+	0	10100	00010100
	0	10100	11010100

Thus, our sum is 0.110101×2^4.

Floating point subtraction



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

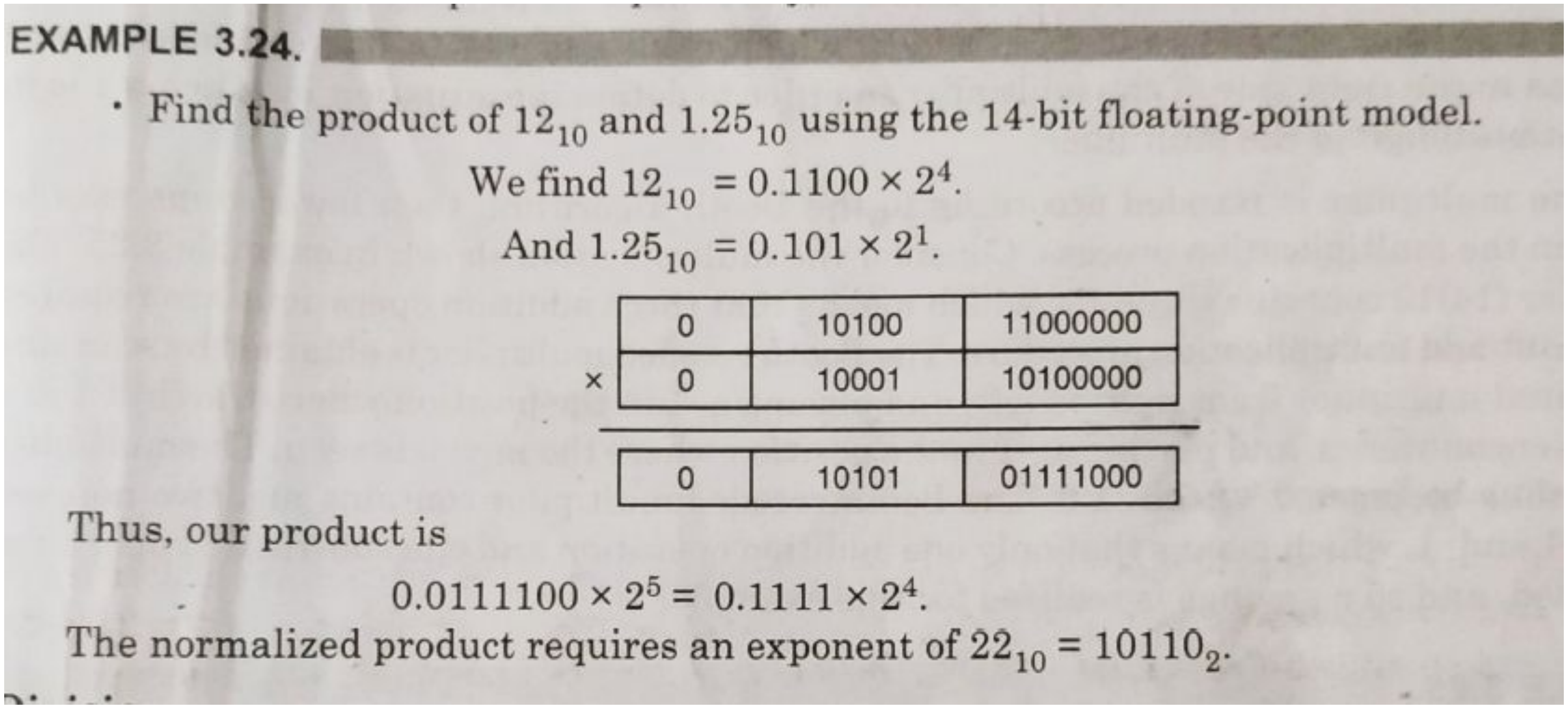
$= 0.000101 \times 2^4$

	0	10100	11000000
−	0	10001	00100000
	0	10101	10101100

Thus, our sum is 0.101011×2^4.

Steps Required to Add/S

Floating point multiplication



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

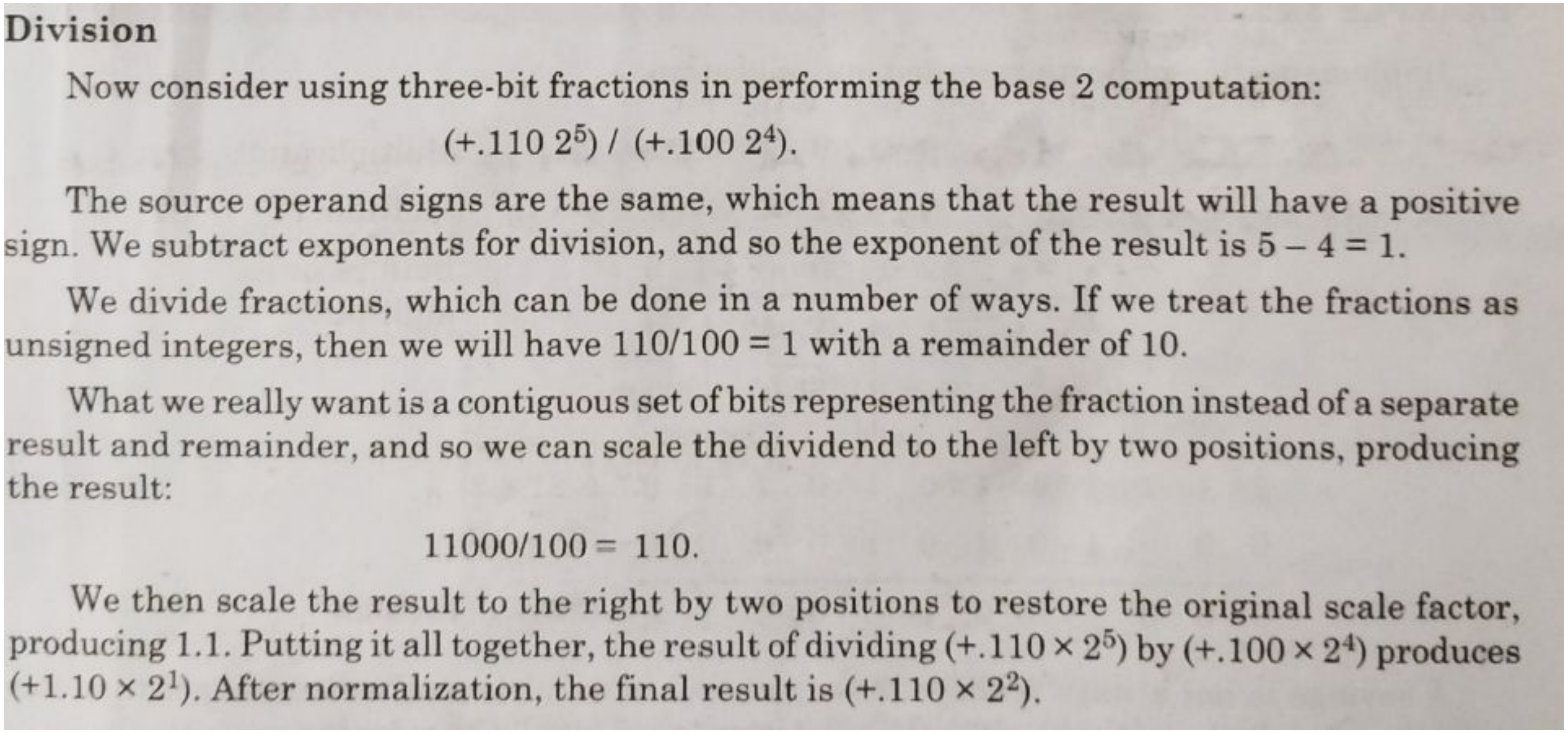
	0	10100	11000000
×	0	10001	10100000
	0	10101	01111000

Thus, our product is

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

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

Floating point Division



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 $(+.110 \times 2^2)$.

??????...Thank You..