Computer Arithmetic

Integer Representation: (Fixed-point representation)
An eight bit word can be represented the numbers from zero to 255 including
00000000 = 0
00000001 = 1
- - - - - - -
11111111 = 255
In general if an n-bit sequence of binary digits $a_{n-1}, a_{n-2} \ldots\ldots a_1, a_0$; is interpreted as unsigned integer A.

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

Sign magnitude representation
There are several alternative convention used to represent negative as well as positive integers, all of which involves treating the most significant (left most) bit in the word as sign bit. If the sign bit is 0, the number is +ve and if the sign bit is 1, the number is –ve. In n bit word the right most n-1 bit hold the magnitude of integer.
For an example,
+18 = 00010010
- 18 = 10010010 (sign magnitude)
The general case can be expressed as follows:

$A = \sum_{i=0}^{n-2} 2^i a_i$ if $a_{n-1} = 0$

$A = -\sum_{i=0}^{n-2} 2^i a_i$ if $a_{n-1} = 1$

There are several drawbacks to sign-magnitude representation. One is that addition or subtraction requires consideration of both signs of number and their relative magnitude to carry out the required operation. Another drawback is that there are two representation of zero. For an example:
$+0_{10}$ = 00000000
-0_{10} = 10000000 this is inconvenient.

2's complement representation:
Like sign magnitude representation, 2's complement representation uses the most significant bit as sign bit making it easy to test whether the integer is negative or positive. It differs from the use of sign magnitude representation in the way that the other bits are interpreted. For negation, take the Boolean complement (1's complement) of each bit of corresponding positive number, and then add one to the resulting bit pattern viewed as unsigned integer. Consider n bit integer A in 2's complement representation. If A is +ve then the sign bit a_{n-1} is zero. The remaining bits represent the magnitude of the number.

$A = \sum_{i=0}^{n-2} 2^i a_i$ for $A \geq 0$

The number zero is identified as +ve and therefore has zero sign bit and magnitude of all 0's. We can see that the range of +ve integer that may be represented is from 0 (all the magnitude bits are zero) through $2^{n-1} - 1$ (all of the magnitude bits are 1).
Now for –ve number integer A, the sign bit a_{n-1} is 1. The range of –ve integer that can be represented is from -1 to -2^{n-1}.

2's complement, $A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i \, a_i$

Defines the twos complement of representation of both positive and negative number.
For an example:
+18 = 00010010
1's complement = 11101101
2's complement = 11101110 = -18

Addition and Subtraction Using One's Complement

One of the main advantages of One's Complement is in the addition and subtraction of two binary numbers. In mathematics, subtraction can be implemented in a variety of different ways as A – B, is the same as saying A + (-B) or -B + A etc. Therefore, the complication of subtracting two binary numbers can be performed by simply using addition

When the two numbers to be added are both positive, the sum A + B, they can be added together by means of the direct sum (including the number and bit sign), because when single bits are added together, "0 + 0", "0 + 1", or "1 + 0" results in a sum of "0" or "1". This is because when the two bits we want to be added together are odd ("0" + "1" or "1 + 0"), the result is "1". Likewise, when the two bits to be added together are even ("0 + 0" or "1 + 1") the result is "0" until you get to "1 + 1" then the sum is equal to "0" plus a carry "1".

Truth Table			
Input		Output	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Overflow rule: If two numbers are added and they are both positive or both negative, then overflow occurs if and only of the result have the opposite sign.

Subtraction of Two Binary Numbers

Example: An 8-bit digital system is required to subtract the following two numbers 115 and 27 from each other using one's complement. So, in decimal this would be: 115 – 27 = 88.

115_{10} in binary is: 01110011_2

27_{10} in binary is: 00011011_2

Now we need to find the complement of the second binary number, (00011011) while leaving the first number (01110011) unchanged. So, by changing all the 1's to 0's and 0's to 1's, the one's complement of 00011011 is therefore equal to 11100100.

Adding the first number and the complement of the second number gives:

```
          01110011
        + 11100100
          --------
Overflow → 1 01010111
```

The 8-bit result from above is: 01010111 (the overflow "1" cancels out because it exceeds the memory capacity) and to convert it back from a one's complement answer to the real answer we now have to add "1" to the one's complement result, therefore:

```
  01010111
+         1
  --------
  01011000
```

Multiplication Algorithm

The multiplier and multiplicand bits are loaded into two registers Q and M. A third register A is initially set to zero. C is the 1-bit register which holds the carry bit resulting from addition. Now, the control logic reads the bits of the multiplier one at a time. If Q_0 is 1, the multiplicand is added to the register A and is stored back in register A with C bit used for carry. Then all the bits of CAQ are shifted to the right 1 bit so that C bit goes to A_{n-1}, A_0goes to $Q_{n-1}$and $Q_0$is lost. If Q_0 is 0, no addition is performed just do the shift. The process is repeated for each bit of the original multiplier. The resulting 2n bit product is contained in the QA register.

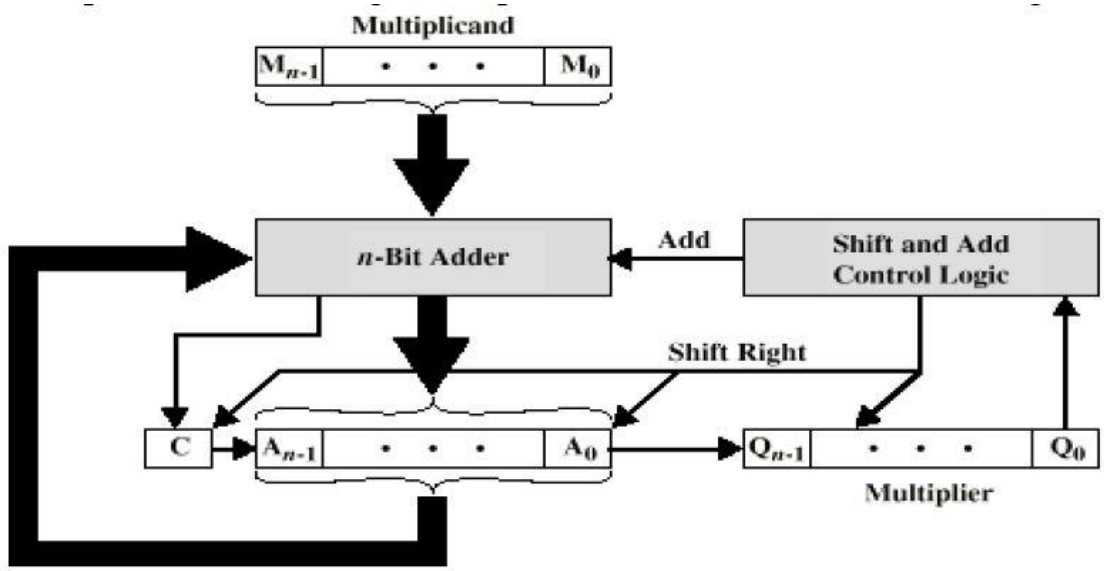


Fig: Block diagram of multiplication

There are three types of operation for multiplication.

- It should be determined whether a multiplier bit is 1 or 0 so that it can designate the partial product. If the multiplier bit is 0, the partial product is zero; if the multiplier bit is 1, the multiplicand is partial product.
- It should shift partial product.
- It should add partial product.

Unsigned Binary Multiplication

```
      1011 Multiplicand 11
    X 1101 Multiplier 13
      1011
     0000        Partial Product
    1011
+ 1011
10001111     Product (143)
```

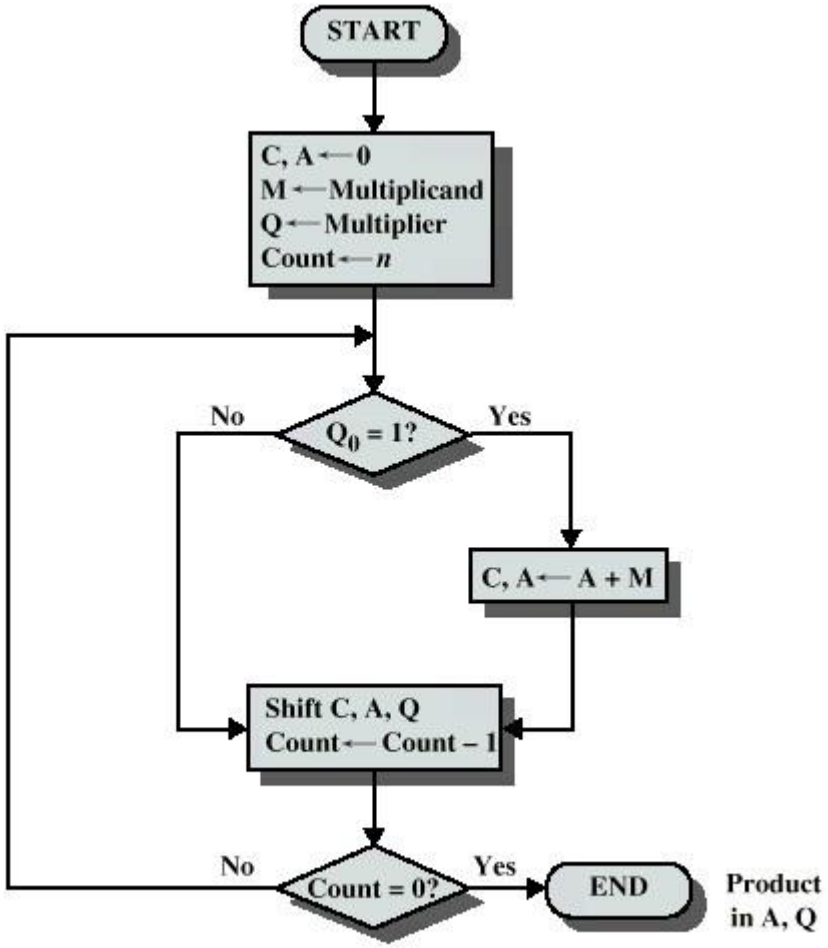


Fig: Flowchart of Unsigned Binary Multiplication

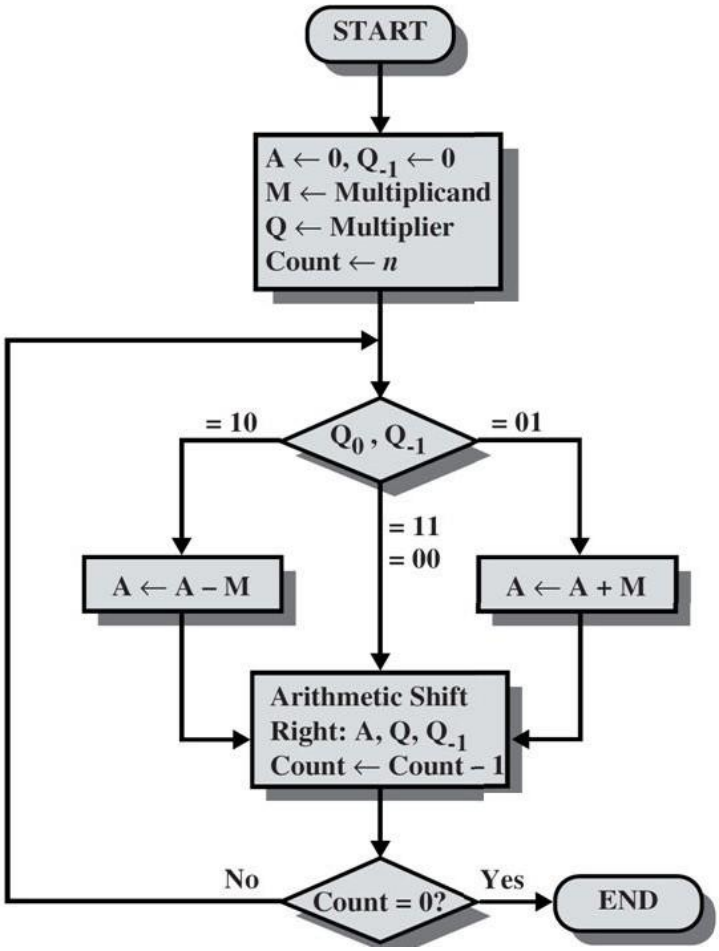
Example: Multiply 15 x 11 using unsigned binary method

C	A	Q	M	Count	Remarks
0	0000	1011	1111	4	Initialization
0 0	1111 0111	1011 1101	- -	- 3	Add (A ← A + M) Logical Right Shift C, A, Q
1 0	0110 1011	1101 0110	- -	- 2	Add (A ← A + M) Logical Right Shift C, A, Q
0	0101	1011	-	1	Logical Right Shift C, A, Q
1 0	0100 1010	1011 0101	- -	- 0	Add (A ← A + M) Logical Right Shift C, A, Q

Result = 1010 0101 = $2^7 + 2^5 + 2^2 + 2^0 = 165$

Signed Multiplication (Booth Algorithm) – 2's Complement Multiplication

Multiplier and multiplicand are placed in Q and M register respectively. There is also one bit register placed logically to the right of the least significant bit Q_0 of the Q register and designated as Q_{-1}. The result of multiplication will appear in A and Q resister. A and Q_{-1} are initialized to zero if two bits (Q0 and Q_{-1}) are the same (11 or 00) then all the bits of A, Q and Q_{-1} registers are shifted to the right 1 bit. If the two bits differ then the multiplicand is added to or subtracted from the A register depending on weather the two bits are 01 or 10. Following the addition or subtraction the arithmetic right shift occurs. When count reaches to zero, result resides into AQ in the form of signed integer $[-2^{n-1} \times a_{n-1} + 2^{n-2} \times + \cdots \ldots \ldots \ldots \ldots . 2^1 \times a_1 + 2^0 \times a_0]$.



Example: Multiply 9 x -3 = -27 using Booth Algorithm
+3 = 00011, -3 = 11101 (2's complement of +3)

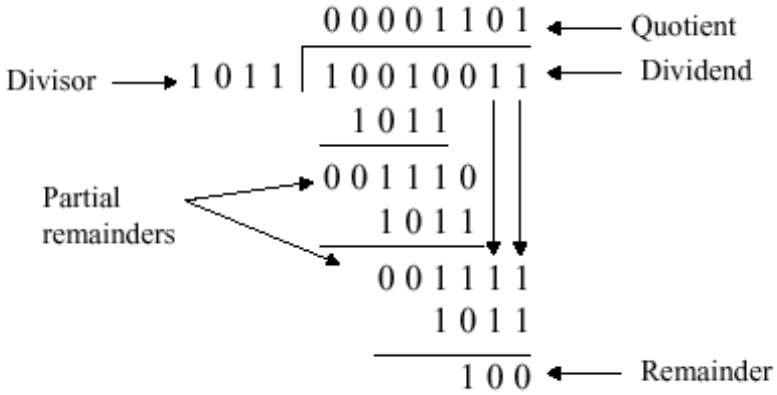
A	Q	Q_{-1}	Add (M)	Sub ($\overline{M}$ +1)	Count	Remarks
00000	1110**1**	**0**	01001	10111	5	Initialization
10111 11011	11101 1111**0**	0 **1**	- -	- -	- 4	Sub (A ← A - M) as Q_0Q_{-1} = 10 Arithmetic Shift Right A, Q, Q_{-1}
00100 00010	11110 0111**1**	1 **0**	- -	- -	- 3	Add (A ← A + M) as Q_0Q_{-1} = 01 Arithmetic Shift Right A, Q, Q_{-1}
11001 11100	01111 1011**1**	0 **1**	- -	- -	- 2	Sub (A ← A - M) as Q_0Q_{-1} = 10 Arithmetic Shift Right A, Q, Q_{-1}
11110	0101**1**	**1**	-	-	1	Arithmetic Shift Right A, Q, Q_{-1} as Q_0Q_{-1} = 11
11111	00101	1	-	-	0	Arithmetic Shift Right A, Q, Q_{-1} as Q_0Q_{-1} = 11

Result in AQ = 11111 00101 = $-2^9+2^8+2^7+2^6+2^5+2^2+2^0$ = -512+256+128+64+32+4+1 = -27

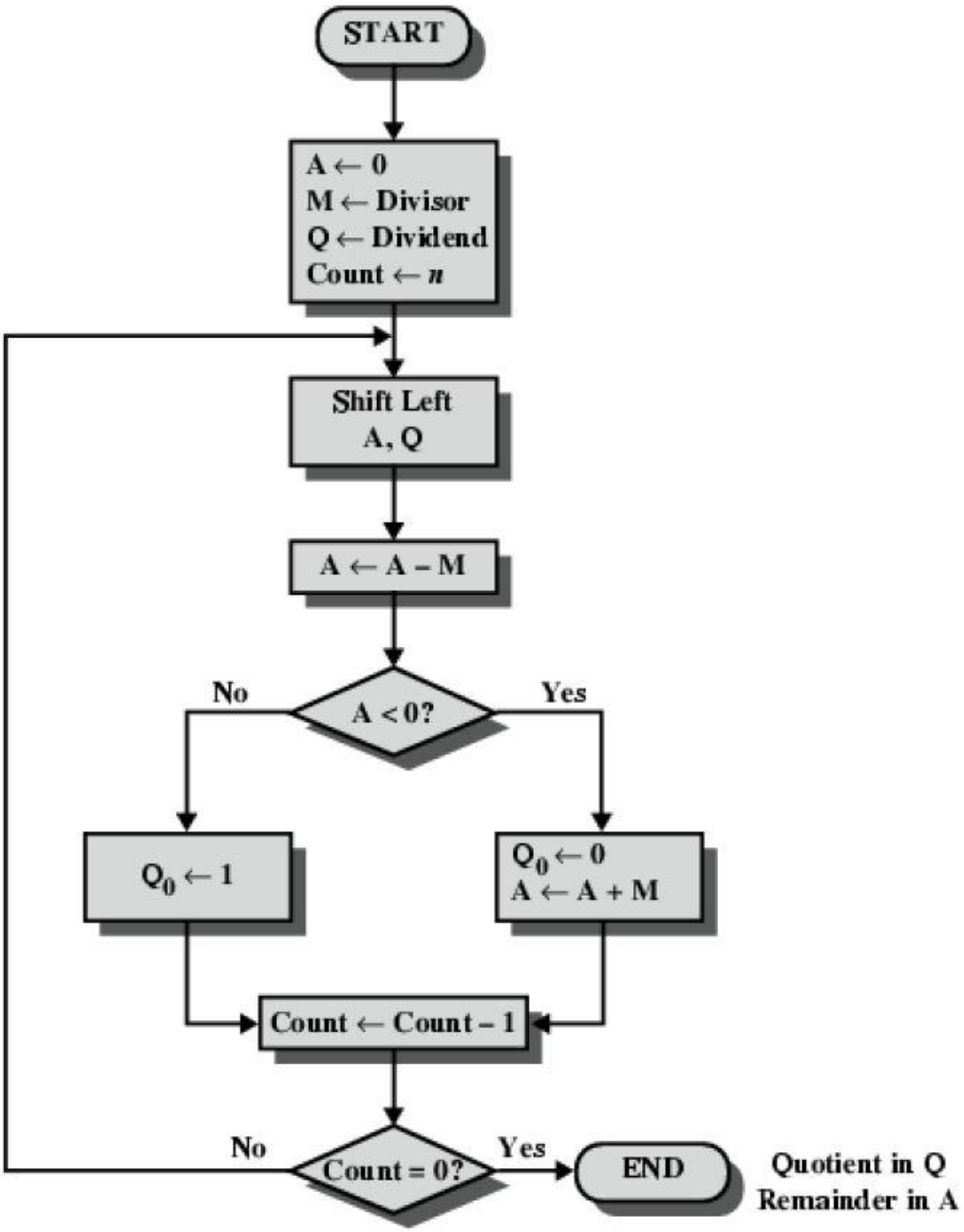
Division Algorithm

Division is somewhat more than multiplication but is based on the same general principles. The operation involves repetitive shifting and addition or subtraction.

First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a *partial remainder*. The division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. The divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.



Restoring Division (Unsigned Binary Division)



Algorithm:

Step 1: Initialize A, Q and M registers to zero, dividend and divisor respectively and counter to n where n is the number of bits in the dividend.

Step 2: Shift A, Q left one binary position.

Step 3: Subtract M from A placing answer back in A. If sign of A is 1, set Q0 to zero and add M back to A (restore A). If sign of A is 0, set Q0 to 1.

Step 4: Decrease counter; if counter > 0, repeat process from step 2 else stop the process. The final remainder will be in A and quotient will be in Q.

Example: Divide 15 (1111) by 4 (0100)

A	Q	M	$\overline{M}+1$	Count	Remarks
00000	1111	00100	11100	4	Initialization
00001	111□	-	-	-	Shift Left A, Q
11101	111□	-	-	-	Sub (A ← A – M)
00001	1110	-	-	3	Q_0 ← 0, Add (A← A + M)
00011	110□	-	-	-	Shift Left A, Q
11111	110□	-	-	-	Sub (A ← A – M)
00011	1100	-	-	2	Q_0 ← 0, Add (A← A + M)
00111	100□	-	-	-	Shift Left A, Q
00011	100□	-	-	-	Sub (A ← A – M)
00011	1001	-	-	1	Set Q_0 ← 1
00111	001□	-	-	-	Shift Left A, Q
00011	001□	-	-	-	Sub (A ← A – M)
00011	0011	-	-	0	Set Q_0 ← 1

Quotient in Q = 0011 = 3
Remainder in A = 00011 = 3

Floating Point Representation

The floating point representation of the number has two parts. The first part represents a signed fixed point numbers called mantissa or significand. The second part designates the position of the decimal (or binary) point and is called exponent. For example, the decimal no + 6132.789 is represented in floating point with fraction and exponent as follows.

Fraction	Exponent
+0.6132789	+04

This representation is equivalent to the scientific notation $+0.6132789 \times 10^{+4}$

The floating point is always interpreted to represent a number in the following form $\pm M \times R^{\pm E}$. Only the mantissa M and the exponent E are physically represented in the register (including their sign). The radix R and the radix point position of the mantissa are always assumed.

A floating point binary no is represented in similar manner except that it uses base 2 for the exponent. For example, the binary no +1001.11 is represented with 8 bit fraction and 0 bit exponent as follows.

0.1001110×2^{100}

Fraction	Exponent
01001110	000100

The fraction has zero in the leftmost position to denote positive. The floating point number is equivalent to $M \times 2^E = +(0.1001110)_2 \times 2^{+4}$

Floating Point Arithmetic

The basic operations for floating point arithmetic are

Floating point number	*Arithmetic Operations*
$X = X_s \times B^{XE}$	$X + Y = (X_s \times B^{XE-YE} + Y_s) \times B^{YE}$
$Y = Y_s \times B^{YE}$	$X - Y = (X_s \times B^{XE-YE} - Y_s) \times B^{YE}$
	$X * Y = (X_s \times Y_s) \times B^{XE+YE}$
	$X / Y = (X_s / Y_s) \times B^{XE-YE}$

There are four basic operations for floating point arithmetic. For addition and subtraction, it is necessary to ensure that both operands have the same exponent values. This may require shifting the radix point on one of the operands to achieve alignment. Multiplication and division are straighter forward.
A floating point operation may produce one of these conditions:

- Exponent Overflow: A positive exponent exceeds the maximum possible exponent value.
- Exponent Underflow: A negative exponent which is less than the minimum possible value.
- Significand Overflow: The addition of two significands of the same sign may carry in a carry out of the most significant bit.
- Significand underflow: In the process of aligning significands, digits may flow off the right end of the significand.

BCD Adder

BCD stand for binary coded decimal. Suppose, we have two 4-bit numbers A and B. The value of A and B can varies from 0(0000 in binary) to 9(1001 in binary) because we are considering decimal numbers.

The output will varies from 0 to 18, if we are not considering the carry from the previous sum. But if we are considering the carry, then the maximum value of output will be 19 (i.e. 9+9+1 = 19).
When we are simply adding A and B, then we get the binary sum. Here, to get the output in BCD form, we will use BCD Adder.

Example 1

Input
A = 0111 and B = 1000
Output
Y = 1 0101
Explanation: We are adding A(=7) and B(=8).
The value of binary sum will be 1111(=15).
But, the BCD sum will be 1 0101, where 1 is 0001 in binary and 5 is 0101 in binary.

Example 2

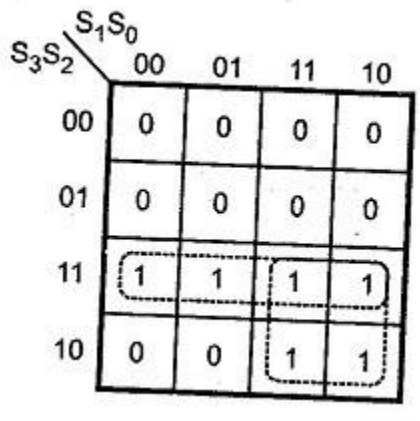
Input
A = 0101 and B = 1001
Output
Y = 1 0100
Explanation: We are adding A(=5) and B(=9).
The value of binary sum will be 1110(=14).
But, the BCD sum will be 1 0001, where 1 is 0001 in binary and 5 is 0001 in binary.

Note – If sum of two numbers is less than or equal to 9, then the value of BCD sum and binary sum will be same otherwise they will differ by 6(0110 in binary).

Now, let's move to the table and find out the logic when we are going to add "0110".

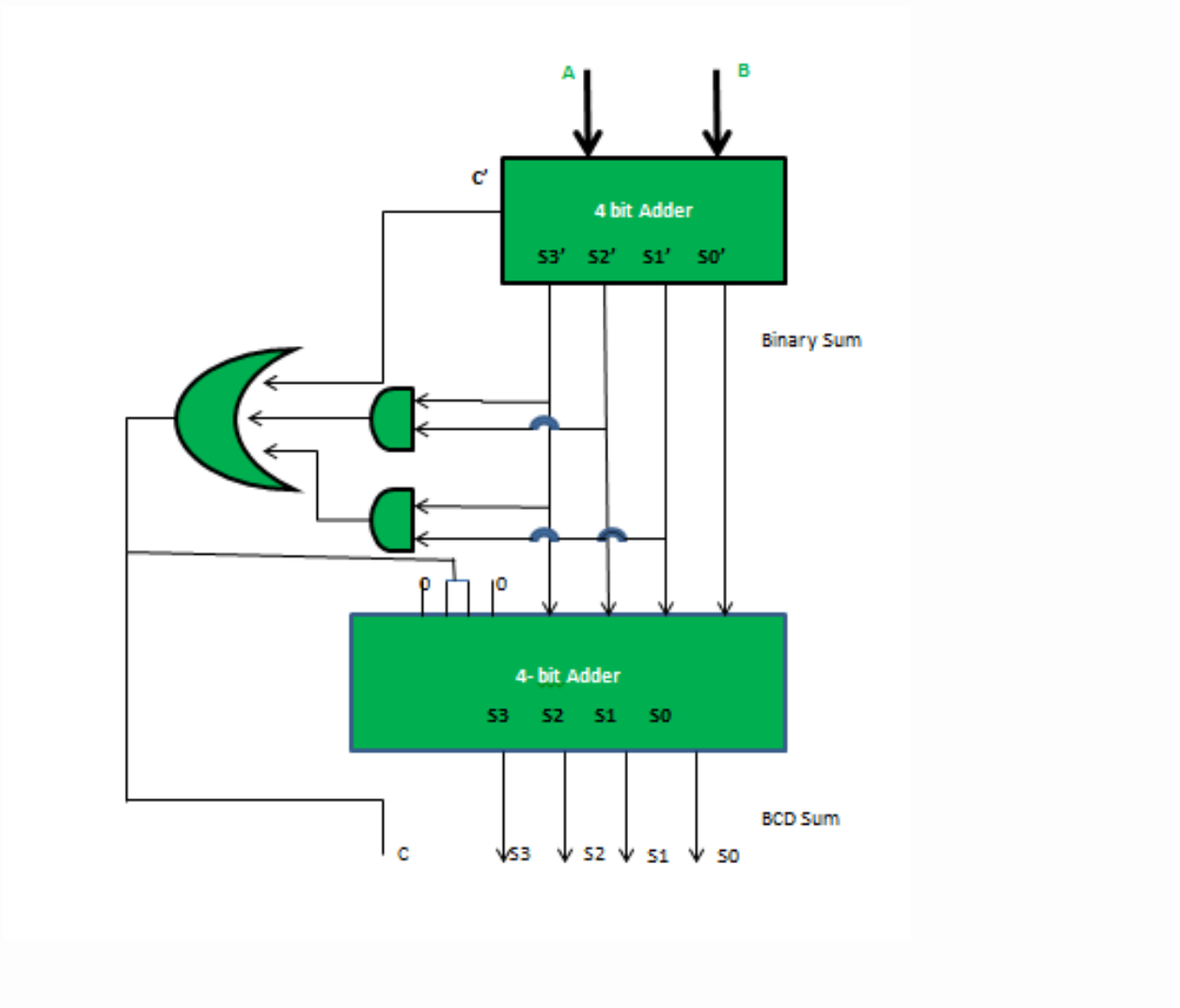
Inputs				Output
S_3	S_2	S_1	S_0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

We are adding "0110"(=6) only after sum exceeds 1001(=9).



From K-Map $Y = S_3S_2 + S_3S_1$

Hardware Implementation



Arithmetic Pipeline

Arithmetic Pipelines are mostly used in high-speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.

To understand the concepts of arithmetic pipeline in a more convenient way, let us consider an example of a pipeline unit for floating-point addition and subtraction.

The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers defined as:

$X = A * 2^a = 0.9504 * 10^3$

$Y = B * 2^b = 0.8200 * 10^2$

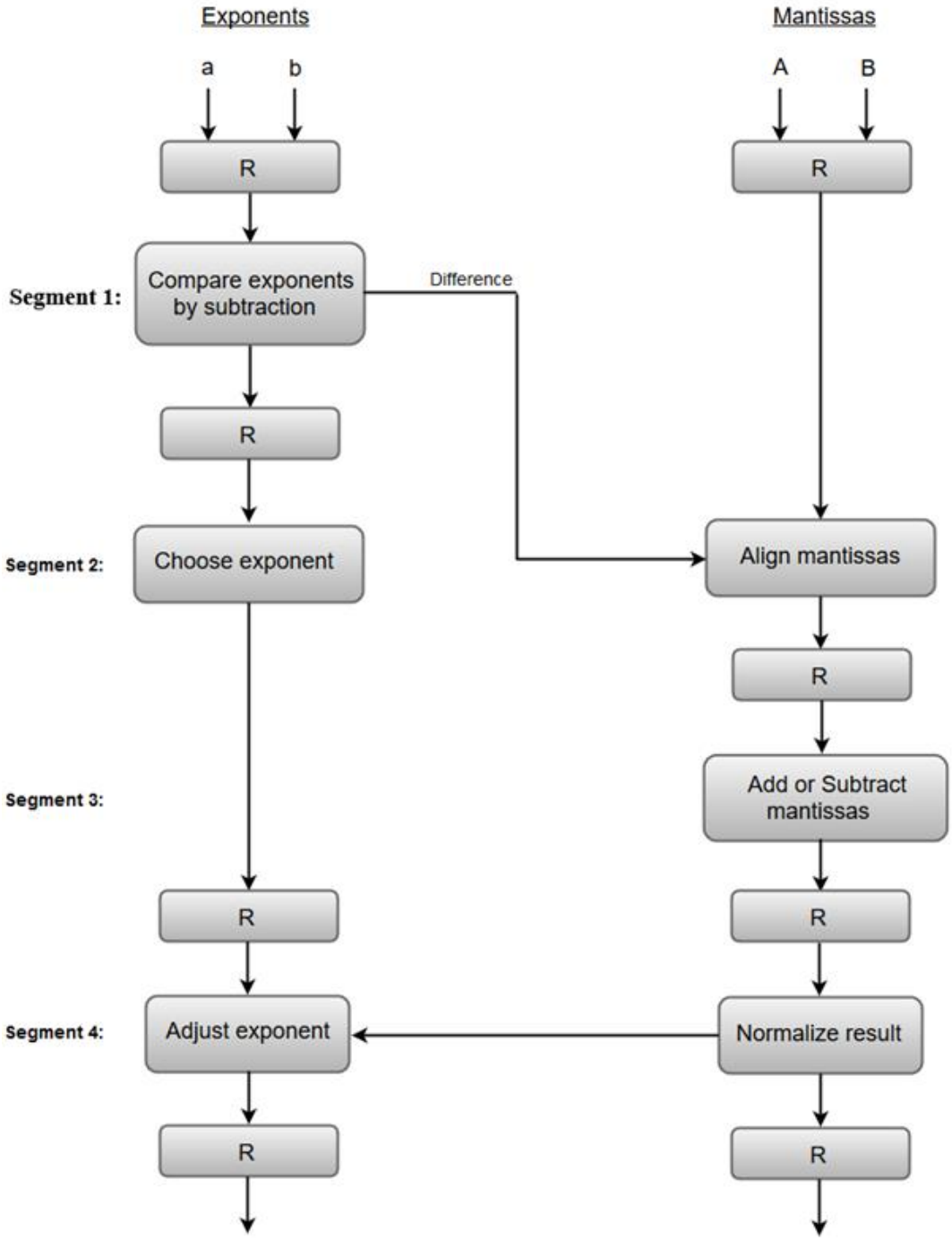
Where A and B are two fractions that represent the mantissa and a and b are the exponents.

The combined operation of floating-point addition and subtraction is divided into four segments. Each segment contains the corresponding suboperation to be performed in the given pipeline. The suboperations that are shown in the four segments are:

1. Compare the exponents by subtraction.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

The following block diagram represents the suboperations performed in each segment of the pipeline.

Pipeline organization for floating point addition and subtraction:



1. Compare exponents by subtraction

   The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result.

   The difference of the exponents, i.e., **3** - **2** = **1** determines how many times the mantissa associated with the smaller exponent must be shifted to the right.

2. Align the mantissas

   The mantissa associated with the smaller exponent is shifted according to the difference of exponents determined in segment one.

   $X = 0.9504 * 10^3$

   $Y = 0.08200 * 10^3$

3. Add mantissas

   The two mantissas are added in segment three.

   $Z = X + Y = 1.0324 * 10^3$

4. Normalize the result

   After normalization, the result is written as:

   $Z = 0.1324 * 10^4$