5EC3-01: Computer Architecture

Dr. S. K. Singh, Professor, ECE,
JECRC.

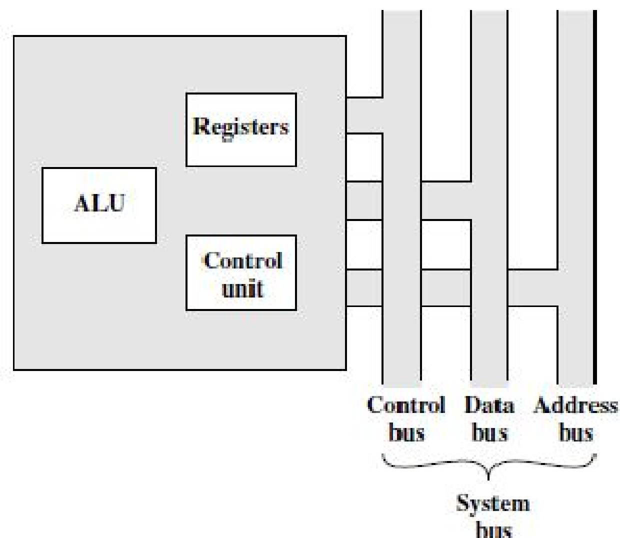
UNIT-2

PROCESSOR ORGANIZATION

To understand the organization of the processor, let us consider the requirements placed on the processor, the things that it must do:

- **Fetch instruction:** The processor reads an instruction from memory (register, cache, main memory).
- **Interpret instruction:** The instruction is decoded to determine what action is required.
- **Fetch data:** The execution of an instruction may require reading data from memory or an I/O module.
- **Process data:** The execution of an instruction may require performing some arithmetic or logical operation on data.
- **Write data:** The results of an execution may require writing data to memory or an I/O module.

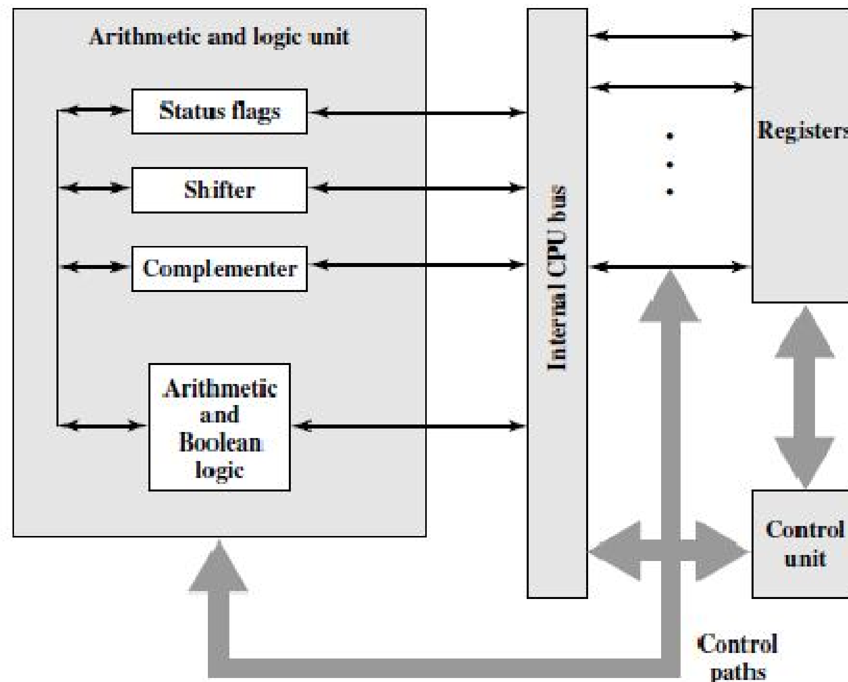
Figure shown below is the simplified view of a processor, indicating its connection to the rest of the system via the system bus.



The major components of the processor are an *arithmetic and logic unit* (ALU) and a *control unit* (CU). The ALU does the actual computation or processing of data. The control unit controls the movement of data and instructions into and out of the processor and controls the operation of the ALU. In addition, the figure shows a minimal internal memory, consisting of a set of storage locations, called *registers.*

Figure shown below is a detailed view of the processor. The data transfer and logic control paths are indicated, including an element labeled *internal processor bus.* This element is needed to transfer data between the various registers and the ALU because the ALU in fact operates only on data

in the internal processor memory. The figure also shows typical basic elements of the ALU. Note the similarity between the internal structure of the computer as a whole and the internal structure of the processor. In both cases, there is a small collection of major elements (computer: processor, I/O, memory; processor: control unit, ALU, registers) connected by data paths.



Representation of Information:

Digital Computers use Binary number system to represent all types of information inside the computers. Alphanumeric characters are represented using binary bits (i.e., 0 and 1). Digital representations are easier to design, storage is easy, accuracy and precision are greater.

There are various types of number representation techniques for digital number representation, for example: Binary number system, octal number system, decimal number system, and hexadecimal number system etc. But Binary number system is most relevant and popular for representing numbers in digital computer system.

Storing Real Number: ***(Number formats)***

These are structures as following below:

Format	Fields
Unsigned integer	Integer
Signed integer	Sign, Integer
Unsigned fixed point	Integer, Fraction
Signed fixed point	Sign, Integer, Fraction
Floating point	Sign, Exponent, Sign, Mantissa
Variable length	Sign, Size, Digits
Unsigned rational	Numerator, Denominator
Signed rational	Sign, Numerator, Denominator

Conversion between Twos Complement Binary and Decimal

128	64	32	16	8	4	2	1

(a) An eight-position twos complement value box

−128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

−128 + 2 + 1 = −125

(b) Convert binary 10000011 to decimal

−128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

−120 = −128 +8

(c) Convert decimal −120 to binary

Sign Magnitude Representation

+18	=	00010010	(sign magnitude, 8 bits)
+18	=	0000000000010010	(sign magnitude, 16 bits)
−18	=	10010010	(sign magnitude, 8 bits)
−18	=	1000000000010010	(sign magnitude, 16 bits)

This procedure will not work for twos complement negative integers. Using the same example,

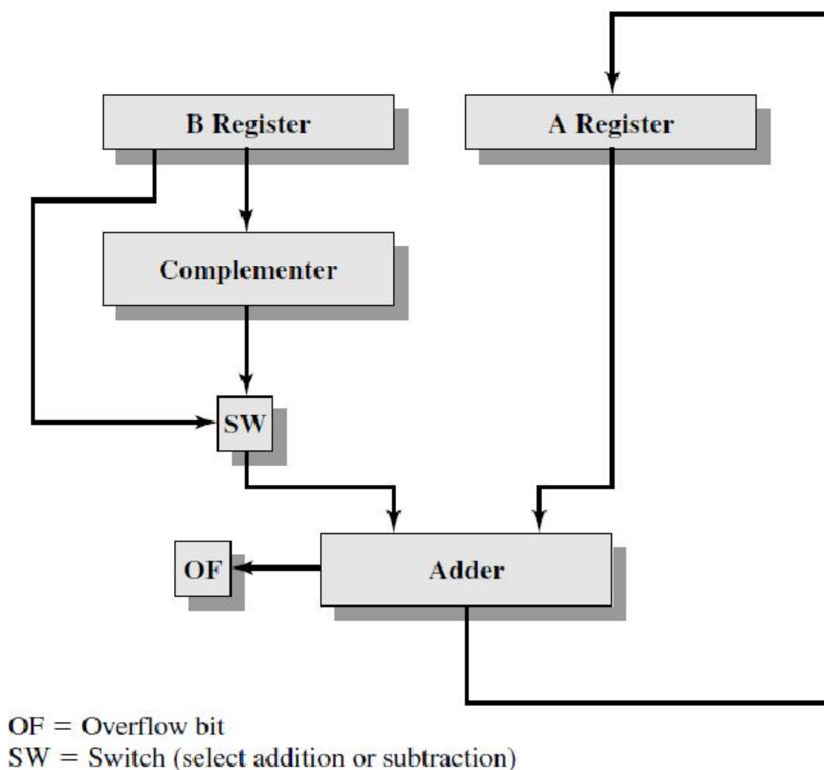
+18	=	00010010	(twos complement, 8 bits)
+18	=	0000000000010010	(twos complement, 16 bits)
−18	=	11101110	(twos complement, 8 bits)
−18	=	1111111111101110	(twos complement, 16 bits)

Instead, the rule for twos complement integers is to move the sign bit to the new leftmost position and fill in with copies of the sign bit. For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. **This is called sign extension.**

INTEGER ARITHMETIC:

Hardware for Addition and Subtraction

Figure shown below the data paths and hardware elements needed to realize addition and subtraction. The central element is a binary adder, which is accessible two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. For addition, the two numbers are presented to the adder from two registers, designated in this case as A and B registers. The result may be stored in one of these registers or in a third. The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; 1 = overflow). For subtraction, the subtrahend (B register) is passed through a twos complementer so that its twos complement is presented to the adder. Note that Figure only shows the data paths. Control signals are needed to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.



#Multiplication

To illustrates the multiplication of unsigned binary integers, as might be carried out using paper and pencil. Several important observations can be made:

1. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.

2. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0.When the multiplier is 1, the partial product is the multiplicand.

```
    1011      Multiplicand (11)
   ×1101      Multiplier (13)
   -----
    1011   )
   0000    |
  1011     >  Partial products
 1011      )
--------
10001111      Product (143)
```

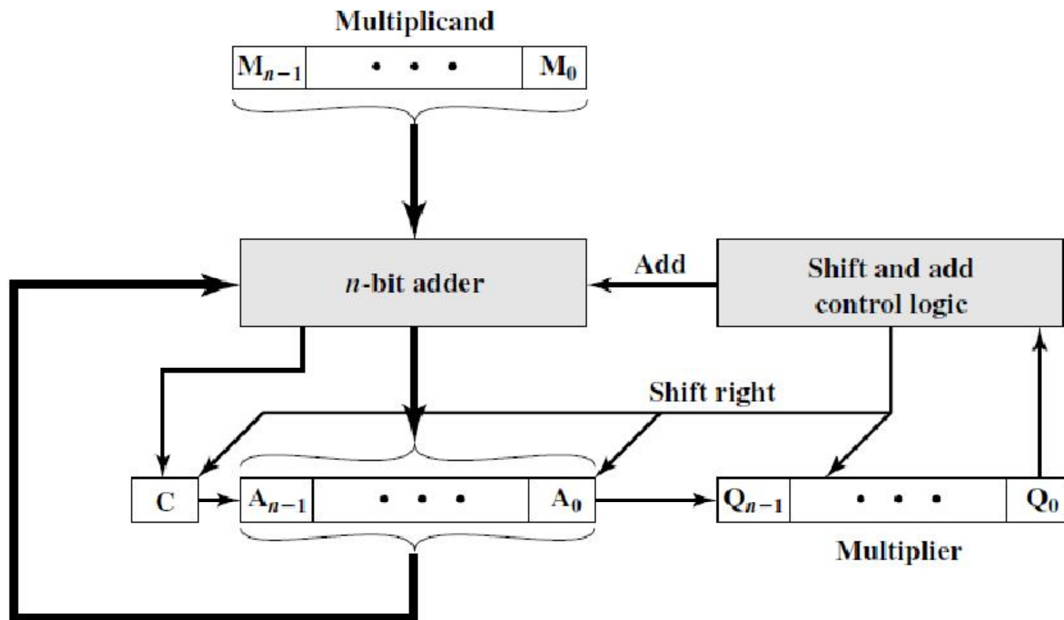
3. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.

4. *The multiplication of two n-bit binary integers results in a product of up to bits in length.* (e.g., 11 X 11 = 1001)

Compared with the pencil-and-paper approach, there are several things has do to make computerized multiplication. First, it can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed. Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

Hardware Implementation of Unsigned Binary Multiplication:

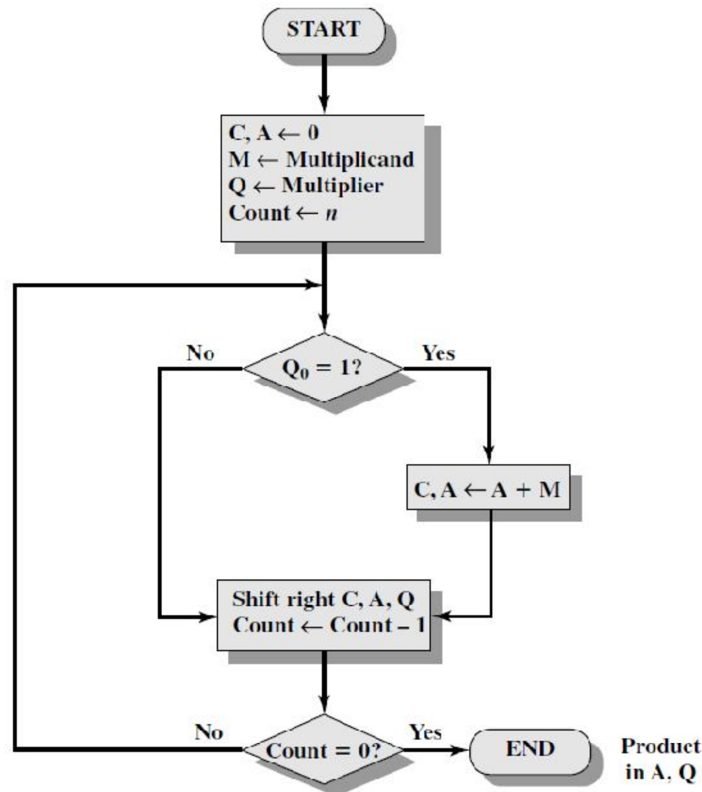
Figure shown below the multiplier and multiplicand are loaded into two registers (Q and M). A third register, the A register, is also needed and is initially set to 0. There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition. The operation of the multiplier is as follows. Control logic reads the bits of the multiplier one at a time. If Q_0 is 1, then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow. Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1}, A_0 goes into Q_{n-1}, and Q_0 is lost.



If Q_0 is 0, then no addition is per-formed, just the shift. This process is repeated for each bit of the original multiplier and an example is given below. The resulting 2n-bit product is contained in the A and Q registers.

C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

A flowchart of the operation is shown below. Note that on the second cycle, when the multiplier bit is 0, there is no add operation.



Flowchart for Unsigned Binary Multiplication

TWOS COMPLEMENT MULTIPLICATION

As addition and subtraction can be performed on numbers in twos complement notation by treating them as unsigned integers.

```
 1001
+0011
 1100
```

If these numbers are considered to be unsigned integers, then we are adding 9 (1001) plus 3 (0011) to get 12 (1100). As twos complement integers, we are adding -7 (1001) to 3 (0011) to get -4 (1100). Unfortunately, this simple scheme will not work for multiplication.

Now it can demonstrate that straightforward multiplication will not work if the multiplicand is negative. The problem is that each contribution of the negative multiplicand as a partial product must be a negative number on a 2n-bit field; the sign bits of the partial products must line up.

(a) Unsigned integers		(b) Twos complement integers	
1001	(9)	1001	(−7)
× 0011	(3)	× 0011	(3)
00001001	1001×2^0	11111001	$(-7) \times 2^0 = (-7)$
00010010	1001×2^1	11110010	$(-7) \times 2^1 = (-14)$
00011011	(27)	11101011	(−21)

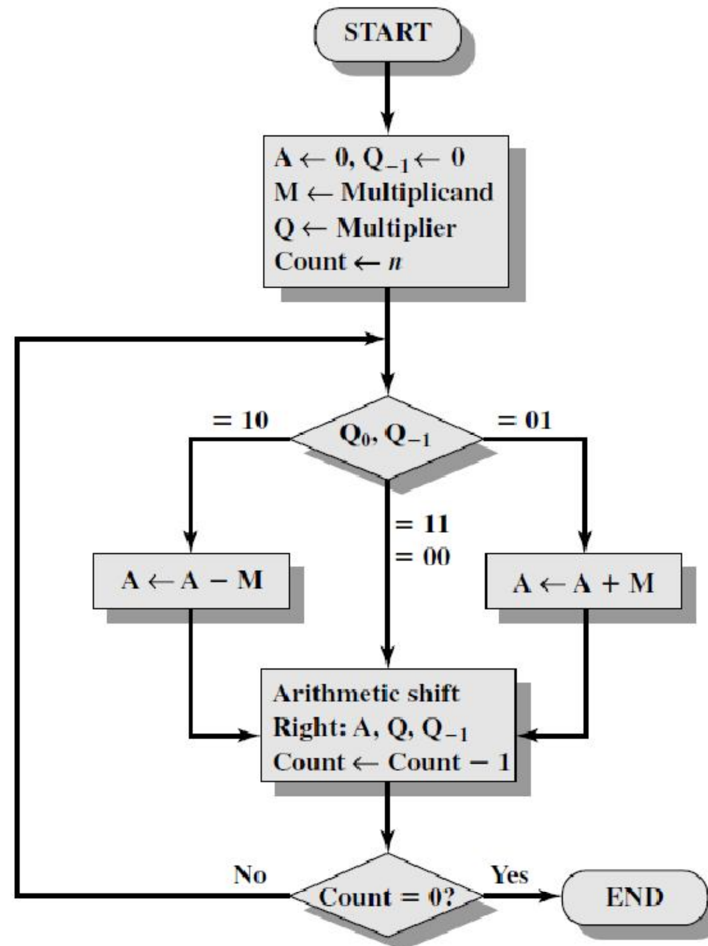
(a) Unsigned integers (b) Twos complement integers

Example shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of 9 x 3 = 27 proceeds simply. However, if 1001 is interpreted as the twos complement -7 value then each partial product must be a negative twos complement number of 2n (8) bits, as

shown in (b). Note that this is accomplished by padding out each partial product to the left with binary 1s. If the multiplier is negative, straightforward multiplication also will not work.

There are a number of ways out of this problem. One of the most common of these is **Booth's algorithm.** This algorithm also speeds up the multiplication process, relative to a more straightforward approach.

#Booth's Algorithm for Twos Complement Multiplication



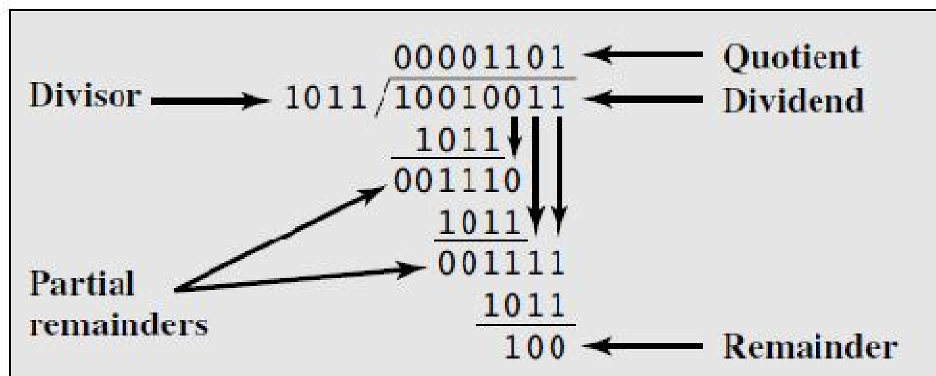
Booth's algorithm can be described as follows. The multiplier and multiplicand are placed in the Q and M registers respectively. There is also a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}. The results of the multiplication will appear in the A and Q registers. A and Q_{-1} are initialized to 0. As before, control logic scans the bits of the multiplier one at a time. Now, as each bit is examined, the bit to its right is also examined. If the two bits are the same (1–1 or 0–0), then all of the bits of the 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 whether the two bits are 0–1 or 1–0. Following the addition or subtraction, the right shift occurs. In either case, the right shift is such that the leftmost bit of A, namely A_{n-1}, not only is shifted into A_{n-2}, but also re-mains in A_{n-1}. This is required to preserve the sign of the number in A and Q. It is known as an **arithmetic shift**, because it preserves the sign bit. The sequence of events in Booth's algorithm for the multipli-cation of 7 by 3 is shown below.

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A – M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

#Division

Division is more complex than multiplication but is based on the same general principles. As the paper-and-pencil approach and the operation involves repetitive shifting and addition or subtraction.

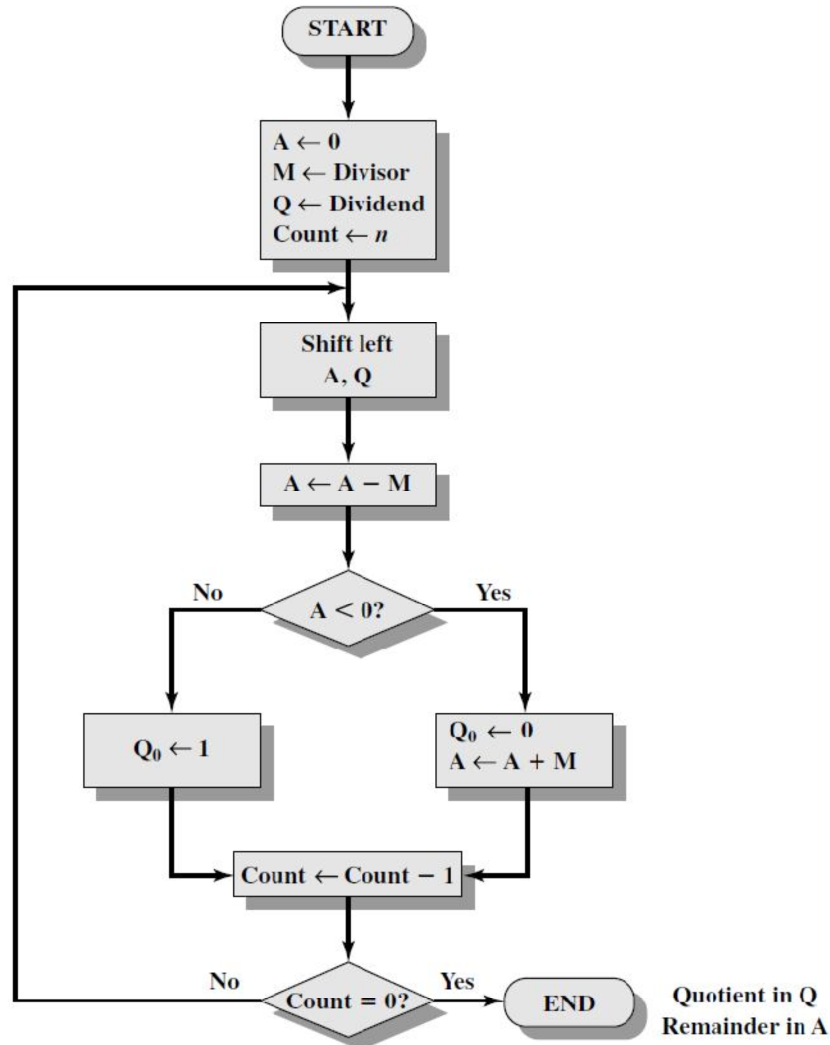
As shown the example of the long division of unsigned binary integers. First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this 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*. From this point on, the division follows a cyclic pattern.



Emxaple of Division of Unsigned Binary Integers

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. As before, 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.

Flowchart shows a machine algorithm that corresponds to the long division process. The divisor is placed in the M register, the dividend in the Q register. At each step, the A and Q registers together are shifted to the left 1 bit. M is subtracted from A to determine whether A divides the partial remainder. If it does, then Q_0 gets a 1 bit. Otherwise, Q_0 gets a 0 bit and M must be added back to A to restore the previ-ous value. The count is then decremented, and the process continues for n steps. At the end, the quotient is in the Q register and the remainder is in the A register.



Flowchart for Unsigned Binary Division

A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Example of Restoring Twos Complement Division (7/3)

There are two major approaches to store real numbers (i.e., numbers with fractional component) in modern computing.

These are:

(i) Fixed Point Notation and
(ii) Floating Point Notation.

In fixed point notation, there are a fixed number of digits after the decimal point, whereas floating point number allows for a varying number of digits after the decimal point.

Fixed-Point Representation:

This representation has fixed number of bits for integer part and for fractional part. For example, if given fixed-point representation is IIII.FFFF, then you can store minimum value is 0000.0001 and maximum value is 9999.9999. There are three parts of a fixed-point number representation: the sign field, integer field, and fractional field.

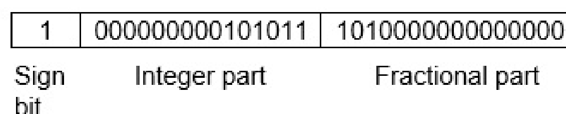
We can represent these numbers using:

- Signed representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.
- 1's complement representation: range from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for k bits.
- 2's complementation representation: range from $-(2^{(k-1)})$ to $(2^{(k-1)}-1)$, for k bits.

2's complementation representation is preferred in computer system because of unambiguous property and easier for arithmetic operations.

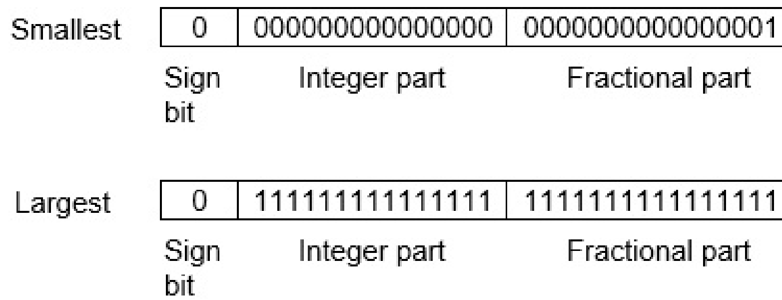
Example: Assume number is using 32-bit format which reserve 1 bit for the sign, 15 bits for the integer part and 16 bits for the fractional part.

Then, -43.625 is represented as following:



Where, 0 is used to represent +ve and 1 is used to represent -ve. 000000000101011 is 15 bit binary value for decimal 43 and 1010000000000000 is 16 bit binary value for fractional 0.625.

The advantage of using a fixed-point representation is performance and disadvantage is relatively limited range of values that they can represent. So, it is usually inadequate for numerical analysis as it does not allow enough numbers and accuracy. A number whose representation exceeds 32 bits would have to be stored inaccurately.



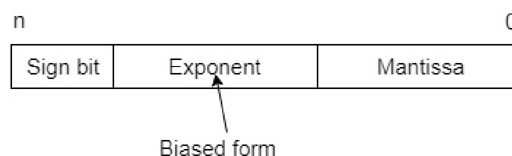
These are above smallest positive number and largest positive number which can be store in 32-bit representation as given above format. Therefore, the smallest positive number is $2^{-16} \approx$ 0.000015 approximate and the largest positive number is $(2^{15}-1)$=32768, and gap between these numbers is 2^{-16}.

Floating-Point Representation:

This representation does not reserve a specific number of bits for the integer part or the fractional part. Instead it reserves a certain number of bits for the number (called the mantissa or significand) and a certain number of bits to say where within that number the decimal place sits (called the exponent).

The floating number representation of a number has two part: the **first part** represents a **signed fixed point number called mantissa**. The **second part** of designates the **position of the decimal (or binary) point and is called the exponent**. The fixed point mantissa may be fraction or an integer. Floating -point is always interpreted to represent a number in the following form: $m \times r^e$.

Only the mantissa **m** and the exponent **e** are physically represented in the register (including their sign). A floating-point binary number is represented in a similar manner except that is uses base 2 for the exponent. A floating-point number is said to be normalized if the most significant digit of the mantissa is 1.



So, actual number is ±**S x B**$^{\pm E}$, where ± is the sign , s is the significant or mantissa, ***E*** is the exponent value, and ***Bias*** is the bias number.

Note that signed integers and exponent are represented by either sign representation, or one's complement representation, or two's complement representation.

The floating point representation is more flexible. Any non-zero number can be represented in the normalized form of $\pm(1.b_1b_2b_3 \ldots)_2 x2^n$ This is normalized form of a number x.

Example: Suppose number is using 32-bit format: the **1 bit sign bit**, **8 bits for signed exponent**, and **23 bits for the fractional part**. The leading bit 1 is not stored (as it is always 1 for a normalized number) and is referred to as a *"hidden bit*".

Then −53.5 is normalized as $-53.5 = (-110101.1)_2 = (-1.101011)x2^5$, which is represented as following below,

1	10000100	10101100000000000000000
Sign bit	Exponent part	Mantissa part

Where 10000100 is the 8-bit binary value of exponent value +5 in form of **baised representation**.

Note that with bias of 127, 8-bit exponent field is used to store true integer exponents values are **$-127 \leq n \leq 128$**.

The smallest normalized positive number that fits into 32 bits is $(1.00000000000000000000000)_2 x2^{-127} = 2^{-127} \approx 5.87x10^{-39}$, and largest normalized positive number that fits into 32 bits is $(1.11111111111111111111111)_2 x2^{128} \approx 3.40x10^{38}$. These numbers are represented as following below,

$0\ 00000000\ 00000000000000000000000_2 = 2^{-127} \approx 5.87x10^{-39}$
(smallest positive normal number)

$0\ 11111111\ 11111111111111111111111_2 = (1.11111111111111111111111)_2 x2^{128} \approx 3.40x10^{38}$
(largest positive normal number)

The precision of a floating-point format is the number of positions reserved for binary digits plus one (for the hidden bit). In the examples considered here the precision is 23+1=24.

The gap between 1 and the next normalized floating-point number is known as **machine epsilon**. the gap is $(1+2^{-23})-1=2^{-23}$for above example, but this is same as the smallest positive floating-point number because of non-uniform spacing unlike in the fixed-point scenario. Note that non-terminating binary numbers can be represented in floating point representation, e.g., 1/3 = $(0.010101 ...)_2$ cannot be a floating-point number as its binary representation is non-terminating.

IEEE Floating point Number Representation:

IEEE (Institute of Electrical and Electronics Engineers) has standardized Floating-Point Representation as following diagram.

So, actual number is $(-1)^s(1+m)x2^{(e-Bias)}$, where *s* is the sign bit, *m* is the mantissa, *e* is the exponent value, and *Bias* is the bias number. The sign bit is 0 for positive number and 1 for negative number. Exponents are represented by or two's complement representation.

According to IEEE 754 standard, the floating-point number is represented in following ways:

- Half Precision (16 bit): 1 sign bit, 5 bit exponent, and 10 bit mantissa
- Single Precision (32 bit): 1 sign bit, 8 bit exponent, and 23 bit mantissa
- Double Precision (64 bit): 1 sign bit, 11 bit exponent, and 52 bit mantissa
- Quadruple Precision (128 bit): 1 sign bit, 15 bit exponent, and 112 bit mantissa

Special Value Representation:

There are some special values depended upon different values of the exponent and mantissa in the IEEE 754 standard.

- All the exponent bits 0 with all mantissa bits 0 represents 0. If sign bit is 0, then +0, else -0.
- All the exponent bits 1 with all mantissa bits 0 represents infinity. If sign bit is 0, then $+\infty$, else $-\infty$.
- All the exponent bits 0 and mantissa bits non-zero represents denormalized number.
- All the exponent bits 1 and mantissa bits non-zero represents error.

FLOATING POINT ARITHMETIC

Arithmetic operations on floating point numbers consist of addition, subtraction, multiplication and division.

The operations are done with algorithms similar to those used on sign magnitude integers (because of the similarity of representation) -- example, only add numbers of the same sign. If the numbers are of opposite sign, must do subtraction.

 # ADDITION

Example on decimal value given in scientific notation:

$$\begin{array}{r} 3.25 \times 10^{3} \\ +\ 2.63 \times 10^{-1} \\ \hline \end{array}$$

First step: Align decimal points

Second step: Add

$$\begin{array}{r} 3.25 \qquad \times 10^{3} \\ +\ 0.000263 \times 10^{3} \\ \hline 3.250263 \times 10^{3} \end{array}$$

(presumes use of infinite precision, without regard for accuracy)

Third step: Normalize the result (already normalized!)

Example on floating point value given in binary: i.e. 100 + 0.25 = 100.25 = 1.0025 X 10^2

.25 = 0 01111101 00000000000000000000000

100 = 0 10000101 10010000000000000000000 to add these fl. pt. representations,

Step 1: Align radix points

- Shifting the mantissa LEFT by 1 bit DECREASES THE EXPONENT by 1
- Shifting the mantissa RIGHT by 1 bit INCREASES THE EXPONENT by 1

It has to shift the mantissa right, because the bits that fall off the end should come from the least significant end of the mantissa.

- Choose to shift the .25, since we want to increase it's exponent.
- Shift by 10000101 – 01111101 = 00001000 (8) places.

0 01111101 00000000000000000000000 (original value)
0 01111110 10000000000000000000000 (shifted 1 place)
(Note that hidden bit is shifted into MSB of mantissa)
0 01111111 01000000000000000000000 (shifted 2 places)
0 10000000 00100000000000000000000 (shifted 3 places)
0 10000001 00010000000000000000000 (shifted 4 places)

0 10000010 00001000000000000000000 (shifted 5 places)
0 10000011 00000100000000000000000 (shifted 6 places)
0 10000100 00000010000000000000000 (shifted 7 places)
0 10000101 00000001000000000000000 (shifted 8 places)

Step 2: Add (Include the hidden bit for the 100)

```
  0 10000101 1.10010000000000000000000  (100)
+ 0 10000101 0.00000001000000000000000  (.25)
-------------------------------------------------------------
  0 10000101 1.10010001000000000000000
```

Step 3: Normalize the result (get the "hidden bit" to be a 1), it already is for this example.

The result is 0 **10000101** 10010001000000000000000 and can be verified on https://www.exploringbinary.com/floating-point-converter/

 # SUBTRACTION

Like addition as far as alignment of radix points then the algorithm for subtraction of sign mag. numbers takes over.

#MULTIPLICATION

Example on decimal values given in scientific notation:

Algorithm: multiply mantissas
add exponents

3.0×10^1
0.5×10^2

1.50×10^3

Example in binary: Use a mantissa that is only 4 bits so that

```
   0 10000011 1110          0 10000011 1110
 x 0 10000100 1001          0 10000100 1001
 ----------------------
```

Mantissa multiplication: 1.1110
(don't forget hidden bit) x 1.1001
----
becomes 10.1110111

Add exponents: Always add true exponents (otherwise the bias gets added in twice)

Biased:
10000011 - 01111111= 00000100 (true exp is 4) and
10000100 -01111111 = 00000101 (true exp is 5)

Add true exponents 5 + 4 is 9

Re-bias exponent: 9 + 127 = 136.
Unsigned representation for 136 is 10001000.

Now put the result back together (and add sign bit).

0 10001000 10.1110111

Normalize the result:
(moving the radix point one place to the left increases the exponent by 1.)

0 10001000 10.1110111 becomes 0 10001001 1.01110111

This is the value stored (not the hidden bit!): 0 10001001 01110111

https://www.rapidtables.com/calc/math/binary-calculator.html

#DIVISION

Similar to multiplication.

True division: do unsigned division on the mantissas (don't forget the hidden bit)
subtract TRUE exponents

References:

1. Computer Organization and Architecture: Designing for Performance, 8th Edition, Authors: William Stallings Publisher: Prentice-Hall India.

--End of Unit-2---