SUBJECT NAME: Computer Organization & Architecture

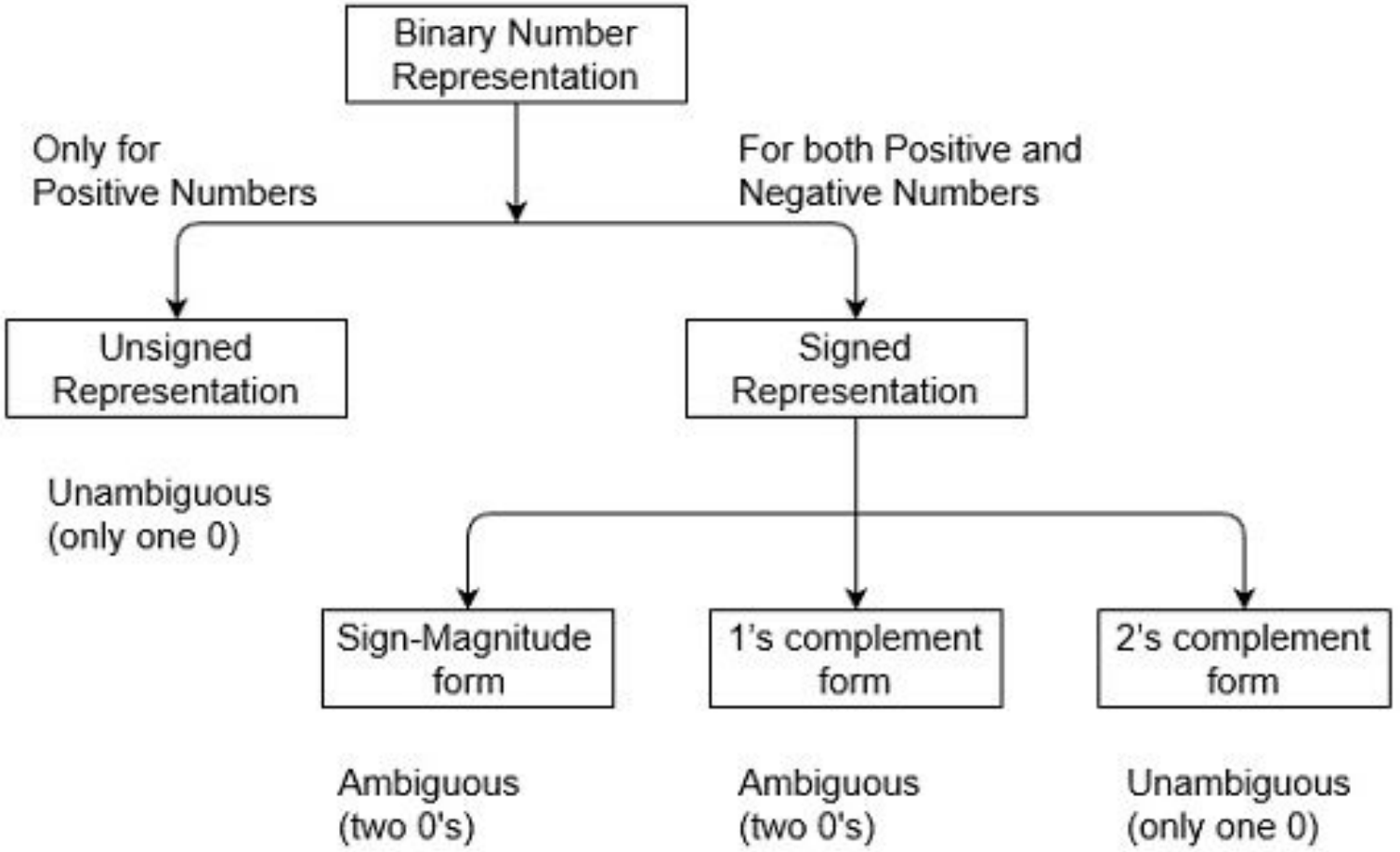
SUBJECT CODE: 203105253

UNIT 2

DATA REPRESENTATION:
Signed number representation, fixed and floating point representations, Character representation.
Computer arithmetic -integer addition and Subtraction, ripple carry adder, carry look-ahead adder, etc. multiplication shift-and add, Booth multiplier, carry save multiplier, etc.
Division restoring and non-restoring techniques, floating point arithmetic

Signed number representation

Binary numbers can be represented in signed and unsigned way. Unsigned binary numbers do not have sign bit, whereas signed binary numbers uses signed bit as well or these can be distinguishable between positive and negative numbers. A signed binary is a specific data type of a signed variable.


Binary Number Representation
Only for Positive Numbers
For both Positive and Negative Numbers
Unsigned Representation
Signed Representation
Unambiguous (only one 0)
Sign-Magnitude form
1's complement form
2's complement form
Ambiguous (two 0's)
Ambiguous (two 0's)
Unambiguous (only one 0)

Signed Magnitude Representation

- The left most bit (Sign bit) in the binary number represents sign of the number

Figure shows the Sign Magnitude format for 8 bit signed Number

The Most Significant bit (MSB) represents sign of the number.

If MSB is 1, Number is Negative

If MSB is 0, Number is Positive

- Example

- +6 = 0000 0110
- -14 = 1000 1110
- +24 = 0001 1000
- -64 = 1100 0000

- For unsigned 8 bit binary number the decimal range is 0 to 255
- For signed magnitude 8 bit binary numbers the largest magnitude is reduced from 255 to 127.

- Max positive num

0	1	1	1	1	1	1	1

=+127

- Max Negative num

1	1	1	1	1	1	1	1

=-127

There are several problems in fact:

- Addition and subtraction operations require:
  - Considering both signs and the magnitudes of each number;
- There are two representations of 0:

$$+0_{10} = 00000000$$
$$-0_{10} = 10000000 \quad \text{(sign magnitude)}$$

  - We need to test for two cases representing zero;
  - This operation is frequently used in computers...
- Because of these drawbacks sign-magnitude representation is rarely used.

Like the sign magnitude system:

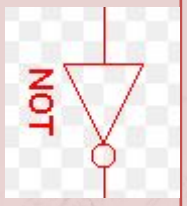
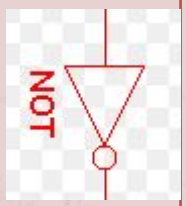
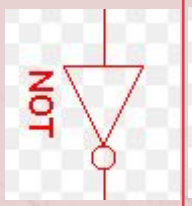
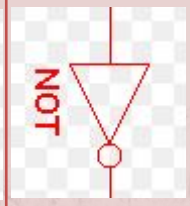
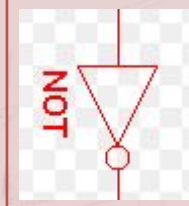
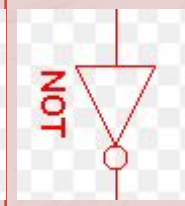
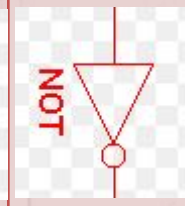
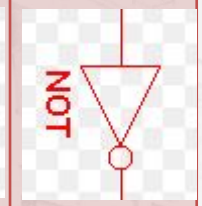
- uses the most significant bit as a sign bit;
- Easy to test whether an integer is positive or negative

However, it differs from the use of the sign-magnitude representation:

- in the way that the other bits are interpreted.

1’s Complement Representation

- The 1’s complement of a binary number is the number that results when we change all 1’s to zero and zero to ones.
- 1’s complement of (11010100)2

1	1	0	1	0	1	0	0
							
0	0	1	0	1	0	1	1

2's Complement Representation

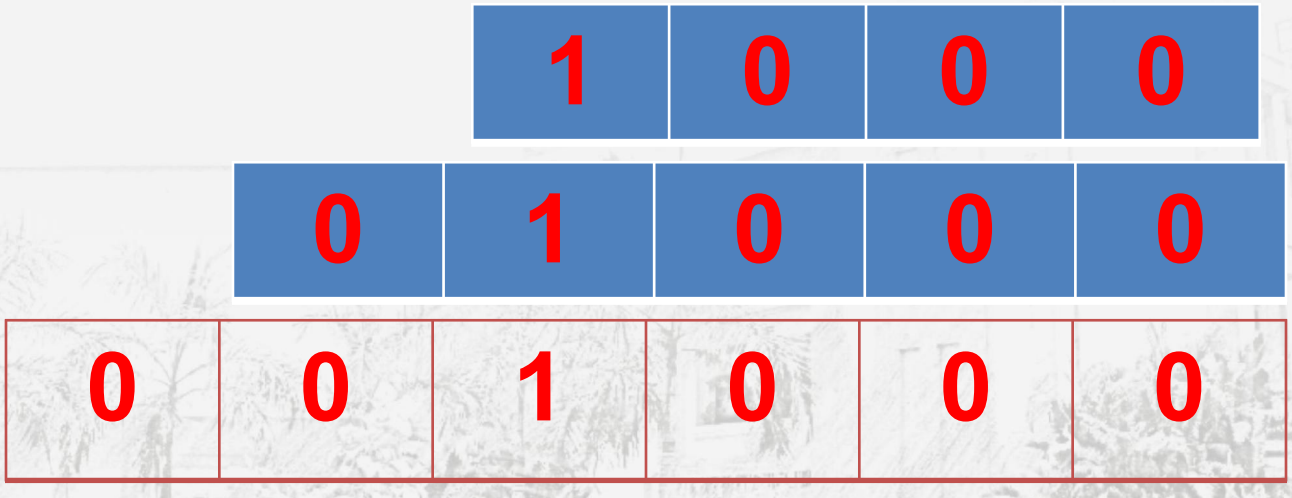
- The 2's complement is the binary number that results when we add 1 to the 1's complement
- 2's complement = 1's complement +1
- 2's complement form is used to represent negative numbers
- Eg 2's complement (11000100)

- 1 1 0 0 0 1 0 0 (number)
- 1 1 (Carry)
- 0 0 1 1 1 0 1 1 (1's Complement)
- \+ 1 (Add 1)

- **0 0 1 1 1 1 0 0**

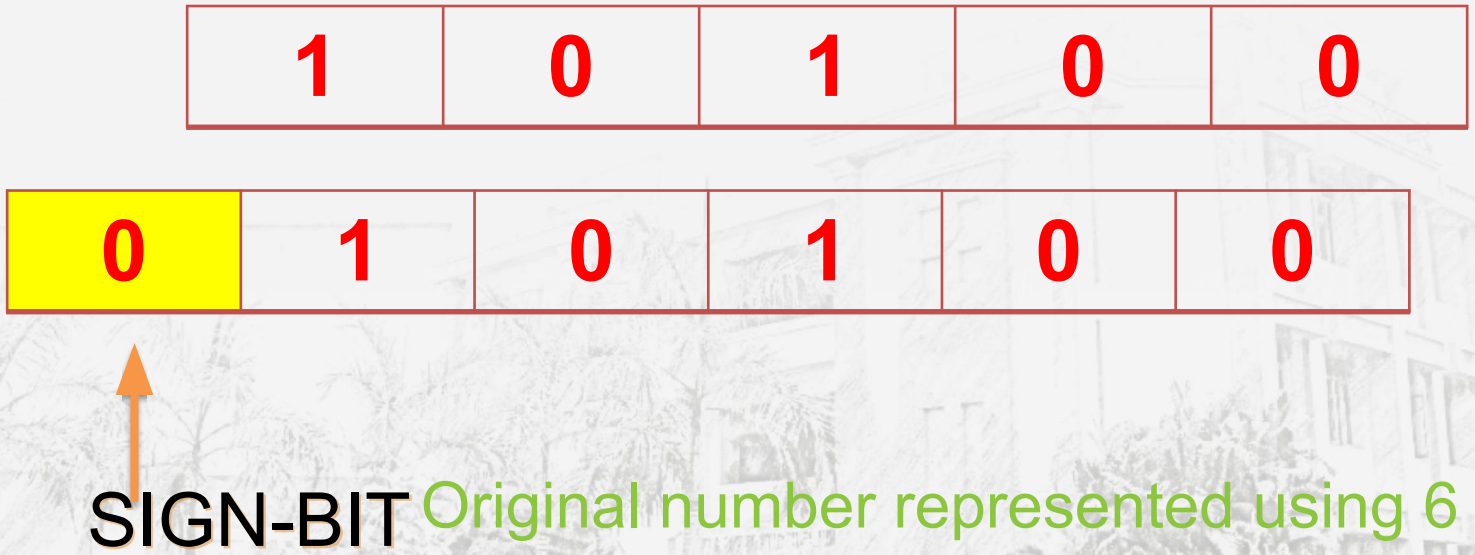
Sign Extension

- When we add two binary numbers the result may extend by 1 bit i.e the resulted binary number may need one more bit if there is a carry after addition of Most significant bit.
- To represent such result in correct format we have to allocate the additional bit for representing the magnitude of the numbers

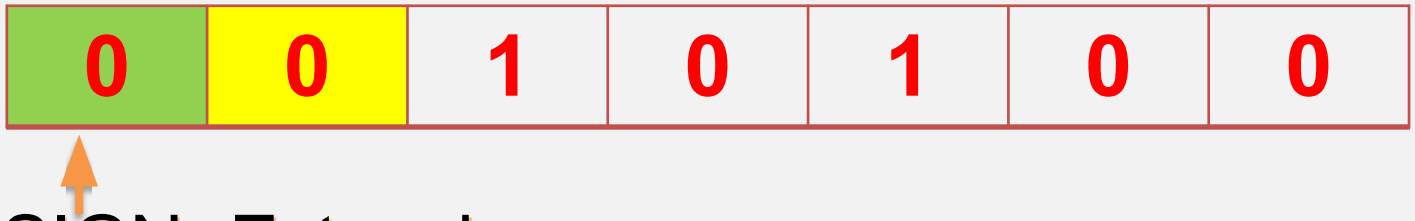


- In case of negative binary number can extend the bit by appending bit 1 to the left of the **Most Significant bit** of the Number.

- Eg. +20
-



represented using 7 bit



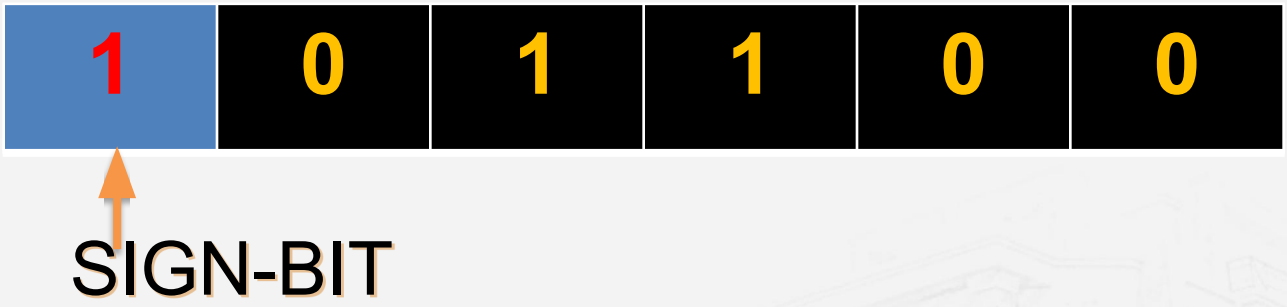
SIGN- Extension

represented using 8 bit

SIGN- Extension

NEGATIVE NUMBER

- EG -20
- 6 BITS IN 2'S COMPLEMENT



7 BITS REPRESENATION

8 BITS REPRESENATION

FLOATING POINT REPRESENTATION

- To accommodate very large integer and very fractions, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds.
- In this case the binary point is said to float and the numbers are called floating point numbers
- Has three fields
- sign
- significant digit
- exponent

- E.g
- **111101.1000110**
- normalized form

EXPONENT

- **1.111011000110 X 2^5**

Sign

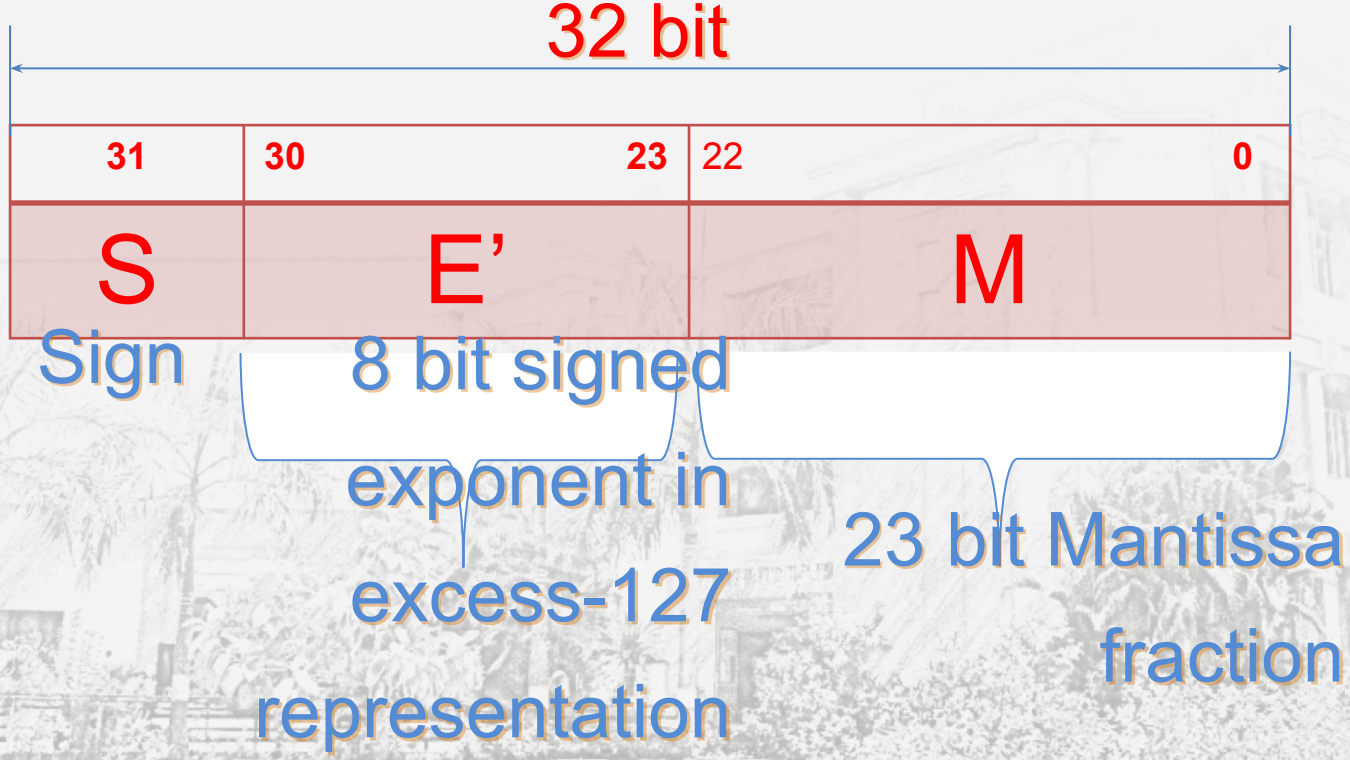
Significant digit

Scaling factor

- We can say that
- Sign = 0
- Mantissa= 11101100110
- Exponent =5

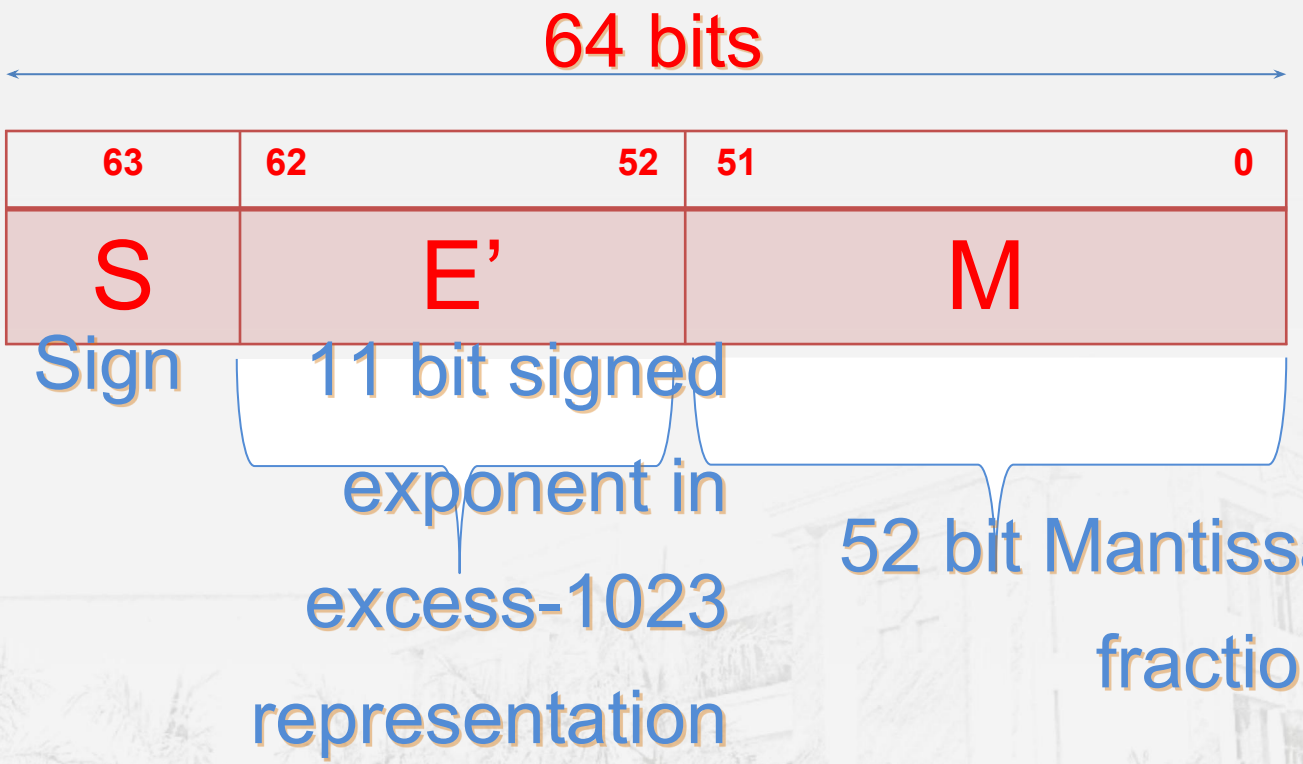
IEEE STANDARD FOR FLOATING POINT NUMBER

- The stds for representing floating point numbers in 32 bits and 64 bits have been developed by
- INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS(IEEE)
- Referred to as IEEE 754 standards



- **Sign of number**
- 0 signifies +ve number
- 1 signifies –ve number
- Instead of the signed exponent E the value actually stored in the exponent field is **E'=E + bias**
- **For 32 bit bias is 127**
- **Hence E'= E+127**
- For 32 bit representation the actual exponent E is in the range -126 <=E<=127

Double Precision representation



- In double precision format value actually stored in the exponent field is given as
- $E' = E + 1023$

- Bia value is 1023 and hence it is also called excess-1023 format
- Range of E' for 64 bit representation is
- $0<E'<2047$ and
- $-1022<= E<= 1023$

Example

- 1259.125 Represent in single & Double Precision Format.
- Step 1 Convert given number into Binary
- Integer Part

	Q	R	
16	1259	11	B
16	78	14	E
16	4	4	4

LSD

MSD

$$(1259)_{10}=(4EB)_{16}$$

4				E				B				HEX
0	1	0	0	1	1	1	0	1	0	1	1	Binary

- FRACTION PART

Fraction		Base	Product
0.125	X	2	0.25
0.25	X	2	0.50
0.50	X	2	1.00

OUTPUT
0
0
1

$$(0.125)_{10}=(0.001)_2$$

- Binary Number
- 10011101011 + 0.001= 10011101011.001

- STEP 2: Normalize the number

- 10011101011.001
- = $1.0011101011001 \times 2^{10}$

- STEP 3 Single Precision Representation (IEEE 754 32 bit format)
- For a given number S=0, E =10 and M = 00111101011001
- Bias=127 so E'= E+127 = 10+127 = 137_{10}
- E'= 10001001

S	EXPONENT								MANTISSA																			
0	1	0	0	0	1	0	0	1	0	0	1	1	1	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0

- STEP 4 Double Precision Representation (IEEE 754 64 bit format)
- For a given number S=0, E =10 and M = 0011101011001
- Bias=1023 so E'= E+1023 = 10+1023 = 1033_{10}
- E'= 10000001001

SIGN	EXPONENT									
0	1	0	0	0	0	0	0	1	0	0	1

MANTISSA													
0	0	1	1	1	0	1	1	0	0	1	---	0	0

CHAR REPRESENTATION

- Character data is composed of letters, symbols, and numerals that are not used in calculations. • Examples of character data include your name, address, and hair colour.
- Character data is commonly referred to as “text.”
- Digital devices employ several types of codes to represent character data, including ASCII, Unicode, and their variants. • ASCII (American Standard Code for Information Interchange, pronounced “ASK ee”) requires seven bits for each character. • The ASCII
- Extended ASCII is a superset of ASCII that uses eight bits for each character. • For example, Extended ASCII represents the uppercase letter A as 01000001. • Using eight bits instead of seven bits allows Extended ASCII to provide codes for 256 characters.

- Unicode (pronounced “YOU ni code”) uses sixteen bits and provides codes or 65,000 characters. • This is a bonus for representing the alphabets of multiple languages.

• UTF-8(**8**-bit Unicode Transformation Format) is a variable-length coding scheme that uses seven bits for common ASCII characters but uses sixteen-bit Unicode as necessary.

Addition

Addition proceeds as if the two numbers were unsigned integers:

```
 1001  = −7
+0101  =  5
 1110  = −2
```

(a) (−7) + (+5)

```
 1100  = −4
+0100  =  4
10000  =  0
```

(b) (−4) + (+4)

- Carry bit beyond (indicated by shading) is ignored;

```
 0011  = 3
+0100  = 4
 0111  = 7
```

(c) (+3) + (+4)

```
 1100  = −4
+1111  = −1
11011  = −5
```

(d) (−4) + (−1)

But what happens with this case?

```
  0101 = 5
+0100 = 4
  1001 = Overflow
```

(e) (+5) + (+4)

```
  1001 = −7
+1010 = −6
 10011 = Overflow
```

(f) (−7) + (−6)

- Two numbers of the same sign produce a different sign…

- **Overflow:**
  - Result is larger that what can be stored with the word;
    - Solution: Increase word size;
  - When overflow occurs:
    - ALU must signal this fact so that no attempt is made to use the result.

- Take the twos complement of the subtrahend **(S)** and add it to the minuend **(M)**.
  - ***M* + (−*S*)**
  - *I.e.:* subtraction is achieved using addition;

```
        0010 =  2
       +1001 = −7
        1011 = −5

(a) M = 2 = 0010
    S = 7 = 0111
   −S =     1001
```

```
        0101 =  5
       +1110 = −2
       10011 =  3

(b) M = 5 = 0101
    S = 2 = 0010
   −S =     1110
```

- Carry bit beyond (indicated by shading) is ignored

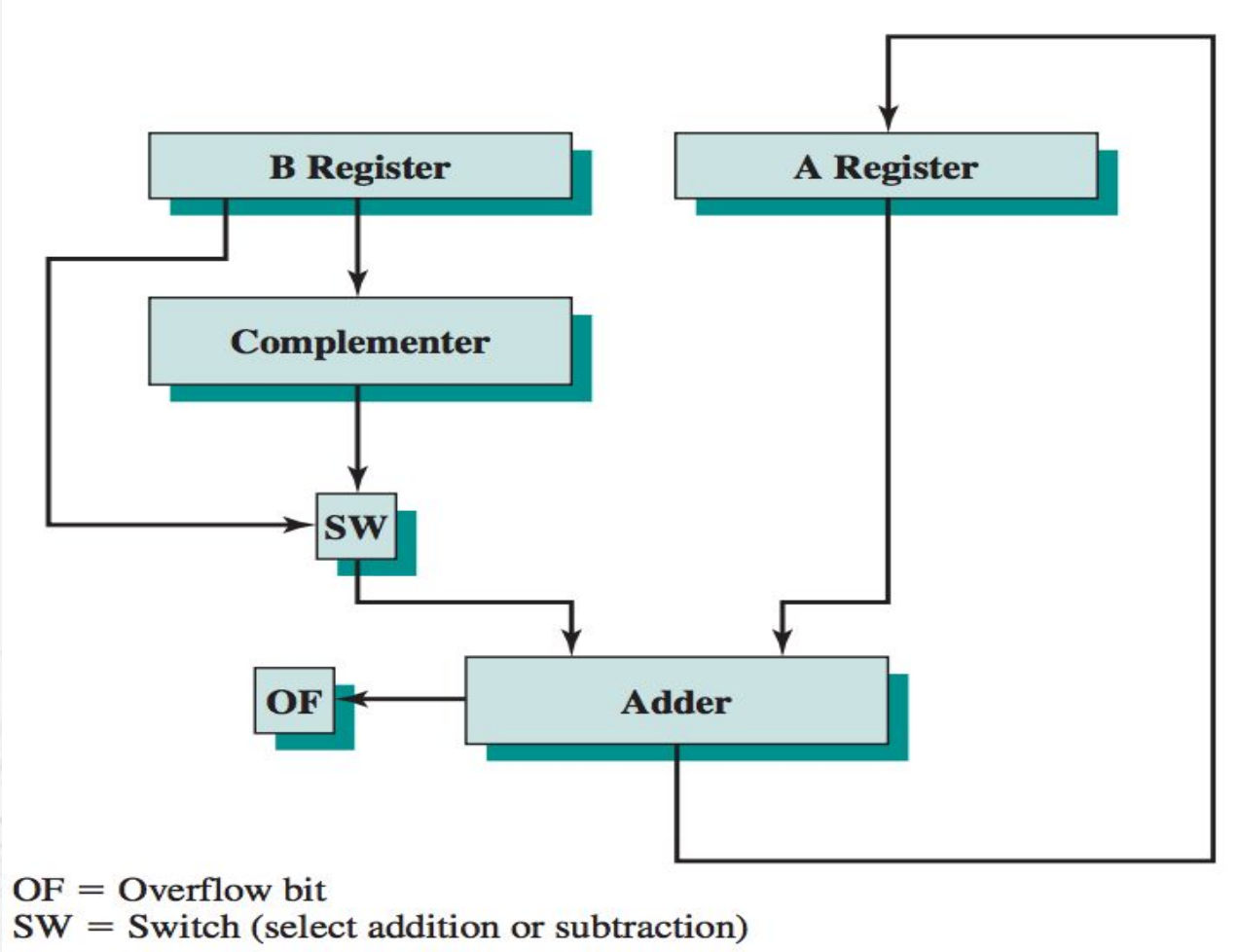
Subtraction is achieved using addition: *M* + (−*S*)

```
      0111 =  7
     +0111 =  7
      1110 =  Overflow

(e)  M =   7 =  0111
     S =  −7 =  1001
    −S =        0111
```

```
      1010 = −6
     +1100 = −4
     10110 =  Overflow

(f)  M = −6 =  1010
     S =  4 =  0100
    −S =       1100
```

- Carry bit beyond (indicated by shading) is ignored;
- **Overflow:** two numbers of the same sign produce a different sign;

Hardware Block Diagram for Adder



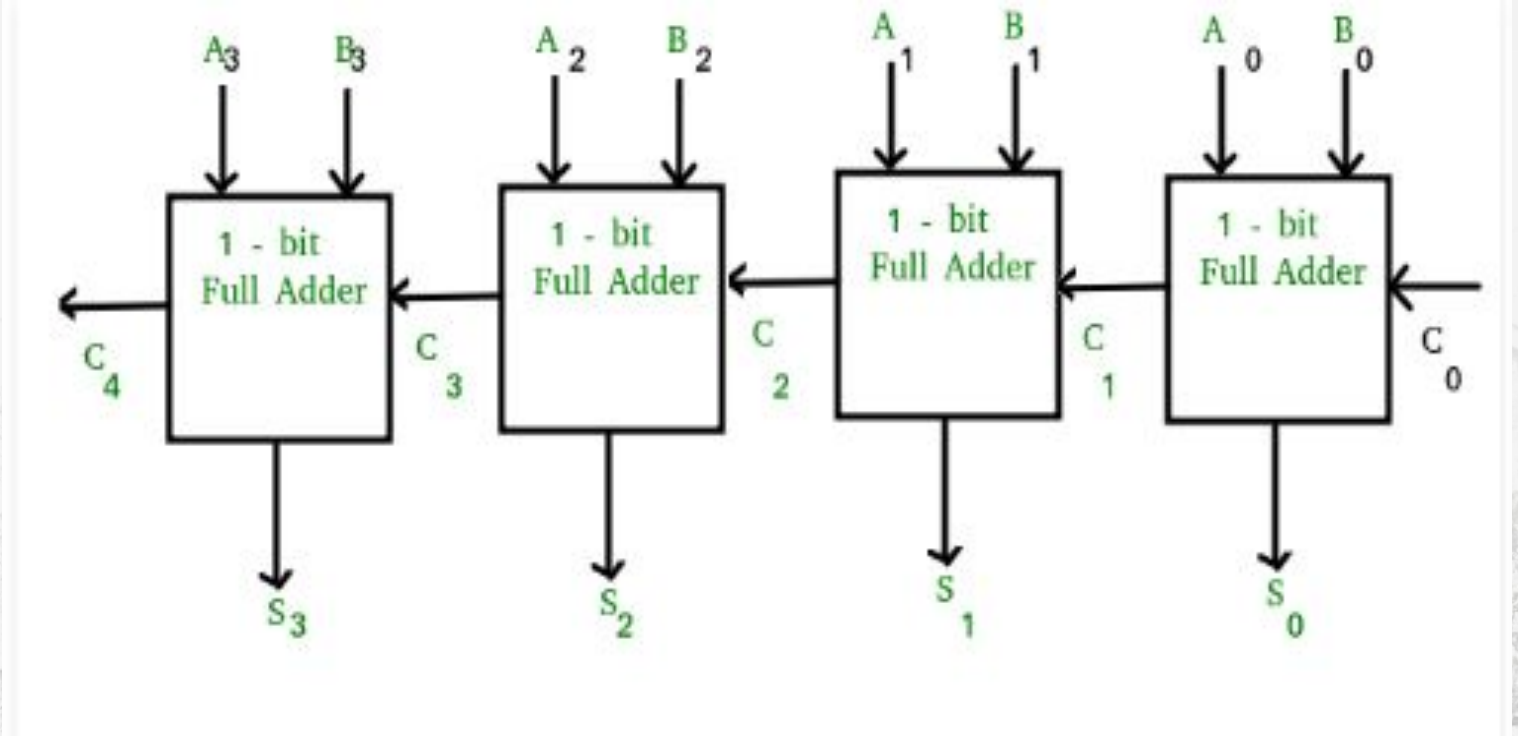
Block diagram of hardware for addition and subtraction

From the previous figure:

- Central element is a binary adder:
  - Presented with two inputs;
  - Produces a sum and an overflow indication;
- Adder treats the two numbers as unsigned integers
- For the **addition** operation:
  - The two numbers are presented in two registers;
  - Result may be stored in one of these registers or in a third;
- For the **subtraction** operation:
  - Subtrahend (B register) is passed through a twos complementor;
- Overflow indication is stored in a 1-bit overflow flag.

Ripple carry adder

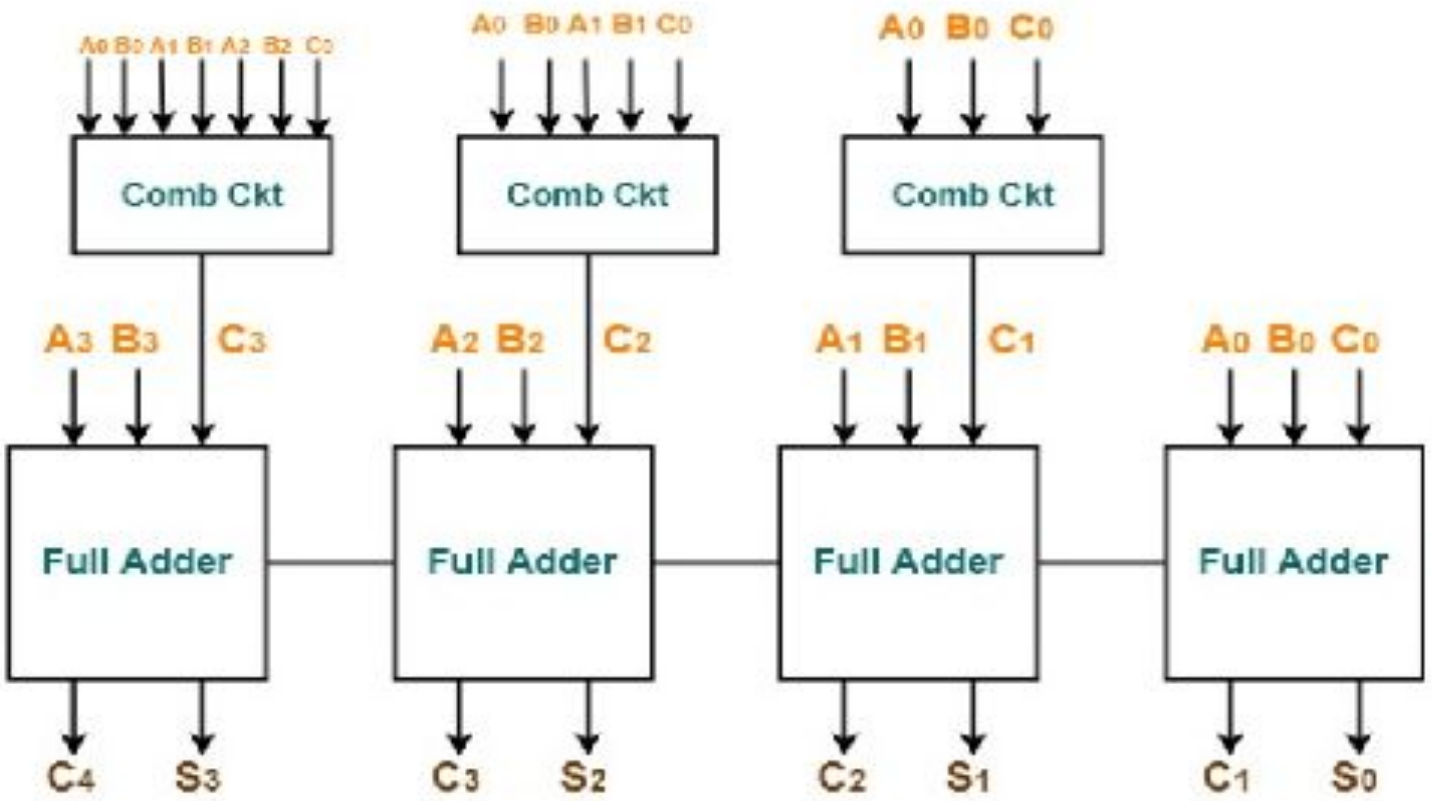
- A ripple carry adder is a logic circuit in which the carry-out of each full adder is the carry in of the succeeding next most significant full adder. It is called a ripple carry adder because each carry bit gets rippled into the next stage.



- In ripple carry adders, for each adder block, the two bits that are to be added are available instantly. However, each adder block waits for the carry to arrive from its previous block. So, it is not possible to generate the sum and carry of any block until the input carry is known. The Ith block waits for the i-1th block to produce its carry. So there will be a considerable time delay which is carry propagation delay.
- Consider the above 4-bit ripple carry adder. The sum S_{4} is produced by the corresponding full adder as soon as the input signals are applied to it. But the carry input C_{4} is not available on its final steady state value until carry C_{3} is available at its steady state value. Similarly C_{3} depends on C_{2} and C_{2} on C_{1}. Therefore, though the carry must propagate to all the stages in order that output S_{3} and carry C_{4} settle their final steady-state value

- The propagation time is equal to the propagation delay of each adder block, multiplied by the number of adder blocks in the circuit. For example, if each full adder stage has a propagation delay of 20 nanoseconds, then S_{3} will reach its final correct value after 60 (20 × 3) nanoseconds. The situation gets worse, if we extend the number of stages for adding more number of bits.

carry look-ahead adder



4-bit-Carry-Look-ahead-Adder-Logic-Diagram

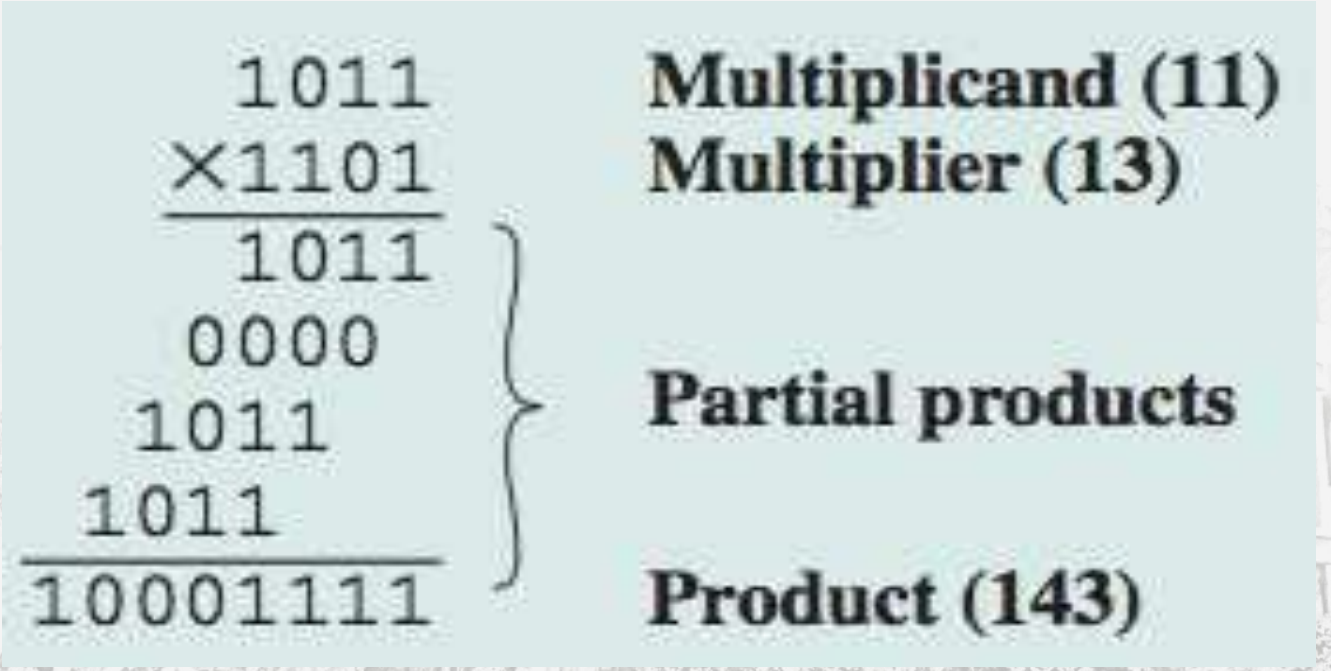
- In this adder, the carry input at any stage of the adder is independent of the carry bits generated at the independent stages. Here the output of any stage is dependent only on the bits which are added in the previous stages and the carry input provided at the beginning stage. Hence, the circuit at any stage does not have to wait for the generation of carry-bit from the previous stage and carry bit can be evaluated at any instant of time.

- **Advantages of Carry Look-ahead Adder**
- In this adder, the propagation delay is reduced. The carry output at any stage is dependent only on the initial carry bit of the beginning stage. Using this adder it is possible to calculate the intermediate results. This adder is the fastest adder used for computation.
- **Applications**
- High-speed Carry Look-ahead Adders are used as implemented as IC's. Hence, it is easy to embed the adder in circuits.

Multiplication

Multiplication is a complex operation:

- Compared with addition and subtraction;
- Again lets consider multiplying for the following cases:
  - Two unsigned numbers;
  - Two signed (twos complement) numbers;



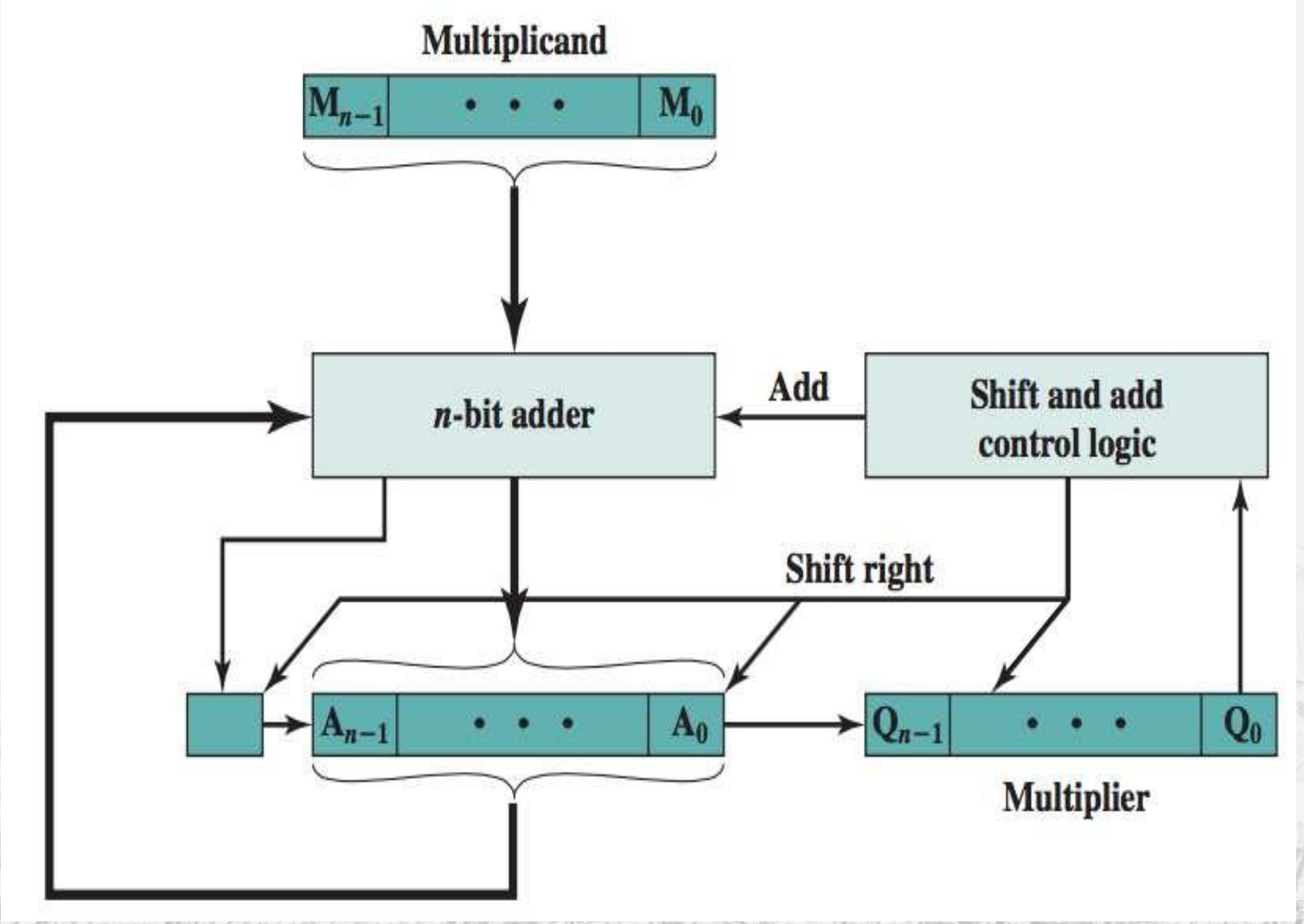
Unsigned multiplication

- Several important observations:

- 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;
- 3 Total product is produced by summing the partial products:
  - each successive partial product is shifted one position to the left relative

- 4 Multiplication of two *n*-bit binary integers produces up to 2*n*

SHIFT-ADD MULTIPLIER



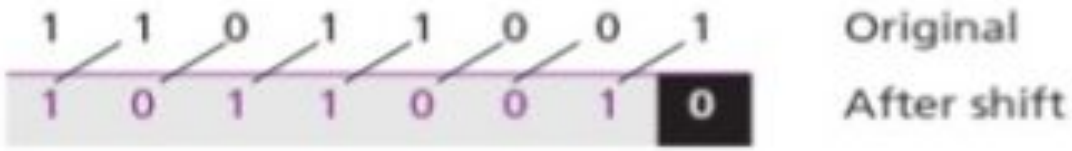
Hardware Implementation

- Control logic then 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:
    - Then no addition is performed, just the shift;
    - Process is repeated for each bit of the original multiplier;
    - Resulting 2n-bit product is contained in the A and Q registers

Arithmetic Shift

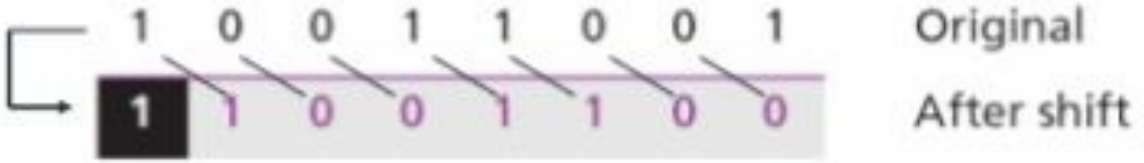
- Arithmetic shift left
- Multiplies a signed binary number by 2

The solution is shown below. The leftmost bit is lost and a 0 is inserted as the rightmost bit.



- Arithmetic shift Right

The solution is shown below. The leftmost bit is retained and also copied to its right neighbor bit.



MULTIPLICATION ALGORITHMS

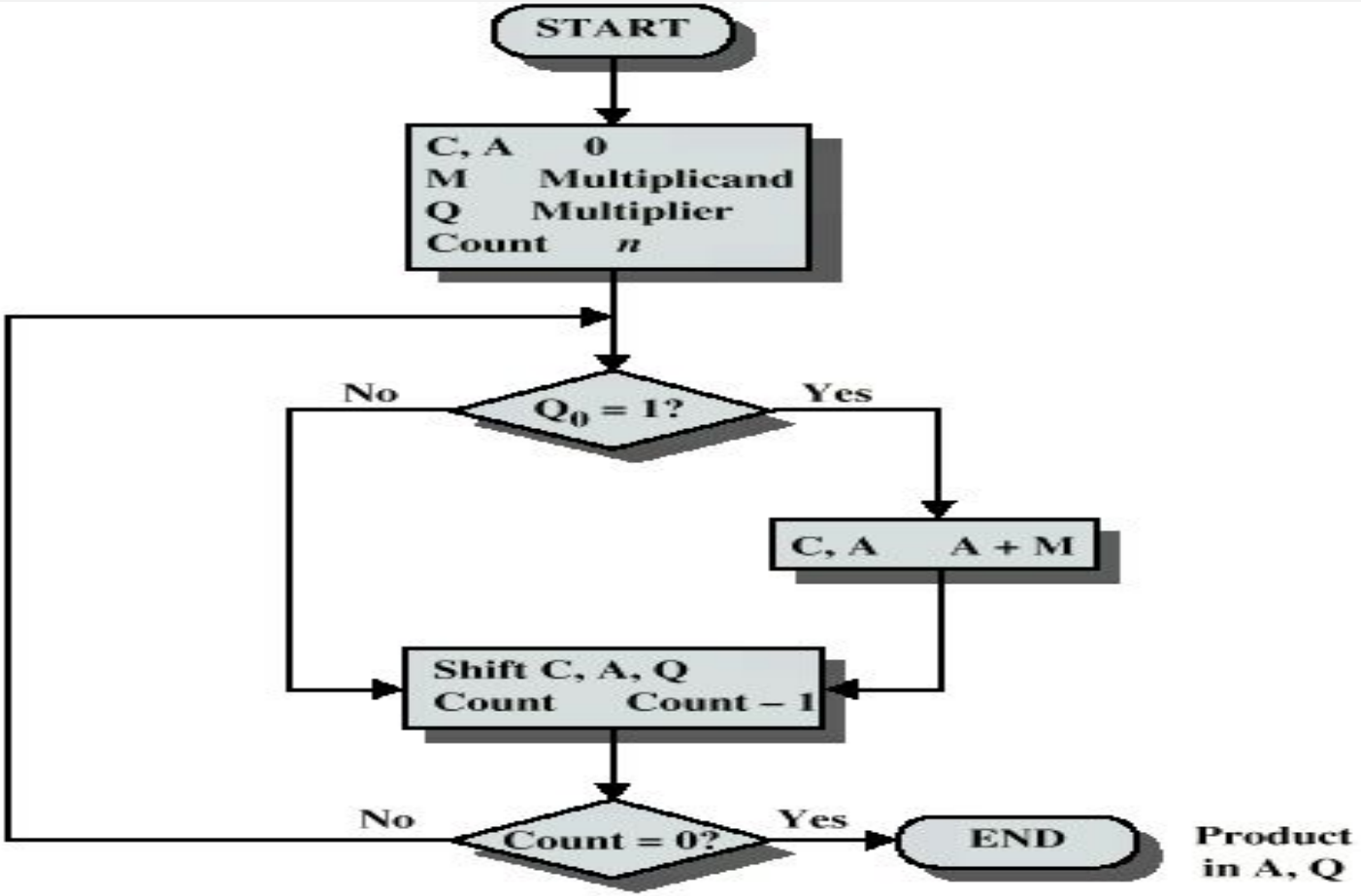
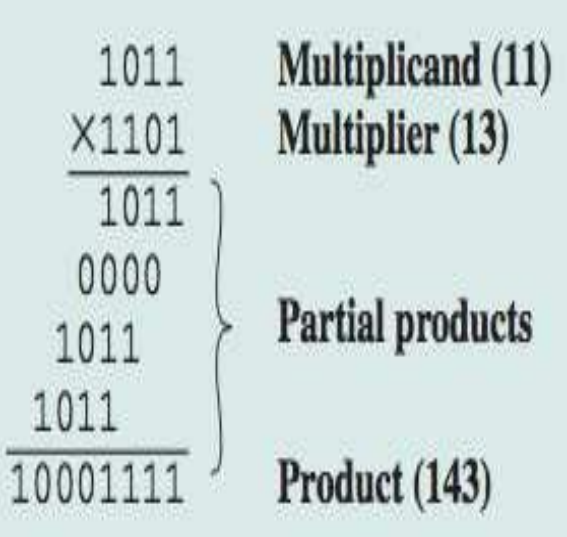


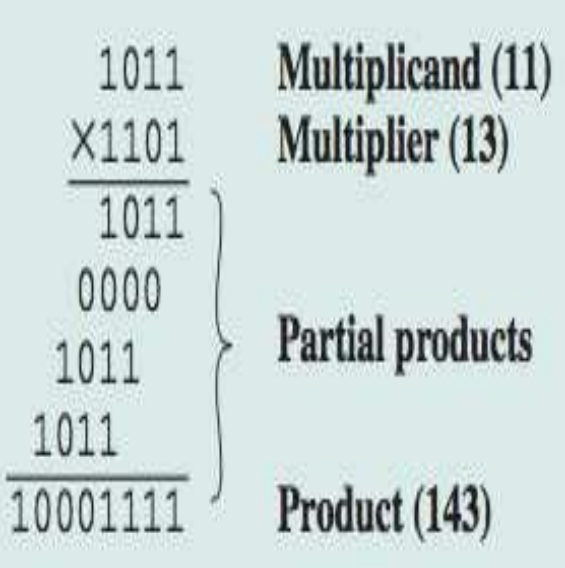
Figure 8.9 Flowchart for Unsigned Binary Multiplication

Example



C	A	Q	M	
0	0000	1101	1011	Initial Values

Example



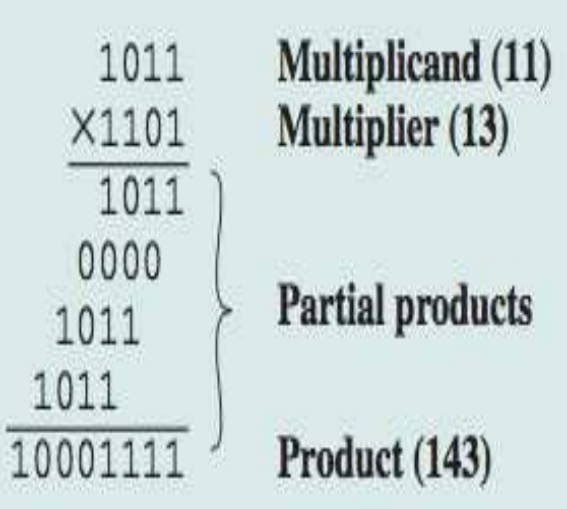
C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add

Example

```
   1011      Multiplicand (11)
  X1101      Multiplier (13)
  ------
   1011
  0000
 1011        Partial products
1011
--------
10001111     Product (143)
```

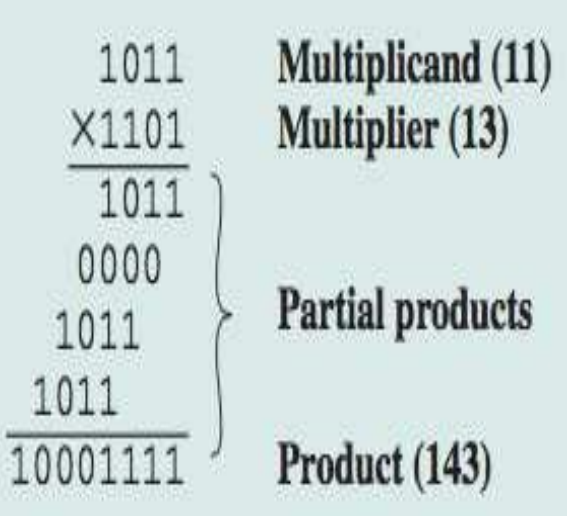
C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right

Example



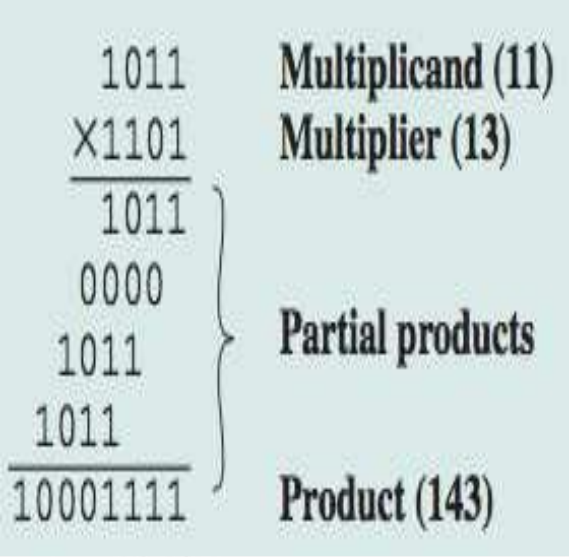
C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right
0	0010	1111	1011	Shift

Example



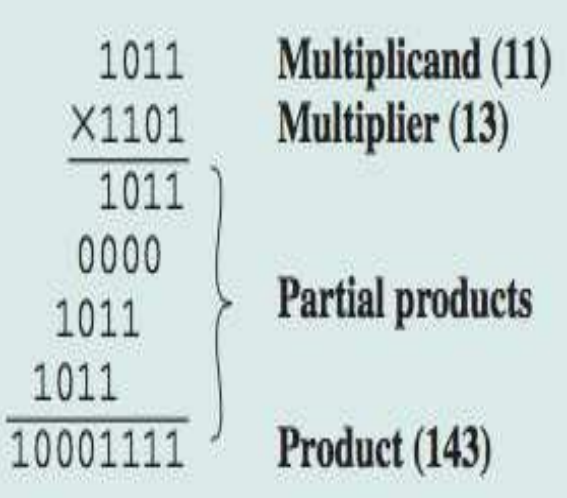
C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right
0	0010	1111	1011	Shift
0	1101	1111	1011	Add

Example



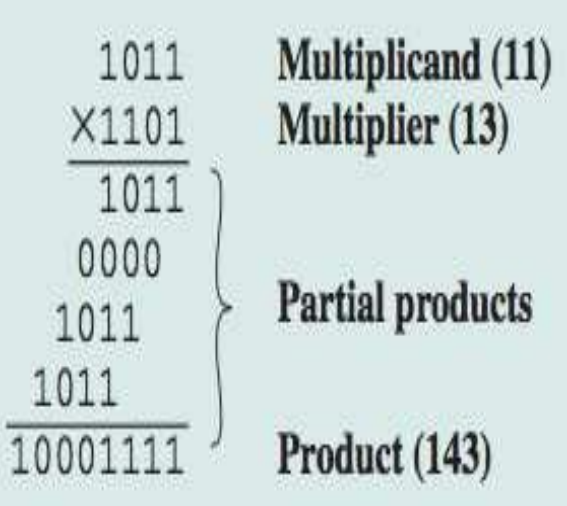
C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	0110	1111	1011	Shift

Example



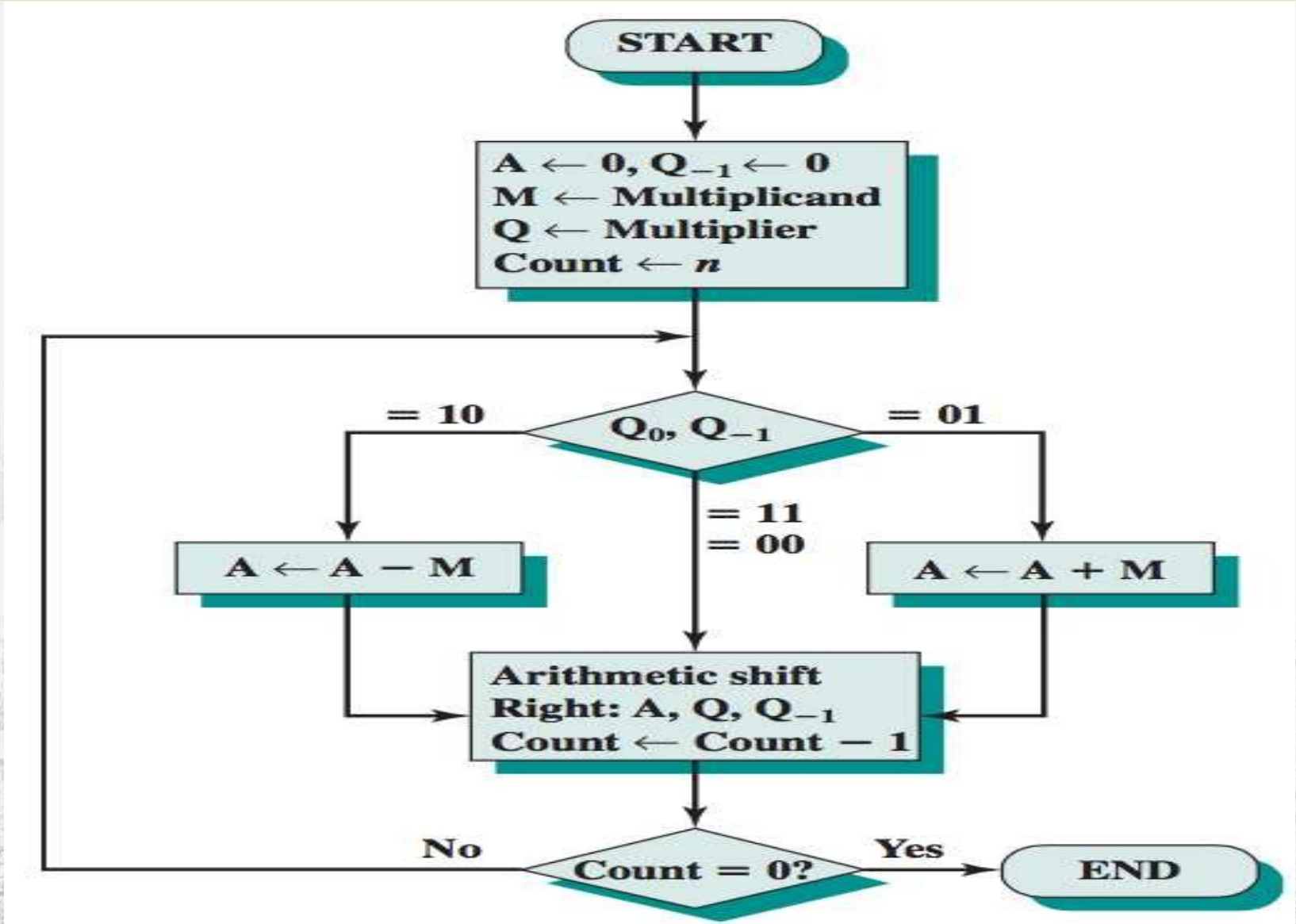
C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	0110	1111	1011	Shift
1	0001	1111	1011	Add

Example



C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	0110	1111	1011	Shift
1	0001	1111	1011	Add
0	1000	1111	1011	Shift

BOOTH ALGORITHM (for Sign Bit Number)



BOOTH ALGORITHM

- **STEP1 : Load A=0,** Q_{-1} =0

  M= Multiplicand

  Q= Multiplier

  SC=0

STEP 2

Check the status of $Q_0\ Q_{-1}$

If $Q_0\ Q_{-1}$ =10 --> A=A-M

$Q_0\ Q_{-1}$ =01 → A=A+M

STEP 3

Arithmetic Shift Right: A, Q, Q_{-1}

STEP 4 Decrement SC if not zero , Repeat Step 2 to 4

STEP 5 STOP

EG both positive (5X4)

- Multiplicand (M)=0101 (5), Multiplier (Q)= 0100 (4)

SC				A				Q					OPERATION
				A3	A2	A1	A0	Q3	Q2	Q1	Q0	Q-1	
1	**0**	**0**		**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**Initial**

EG both positive (5X4)

- Multiplicand (M)=0101 (5), Multiplier (Q)= 0100 (4)

SC				A				Q					OPERATION
				A3	A2	A1	A0	Q3	Q2	Q1	Q0	Q-1	
1	0	0		0	0	0	0	0	1	0	0	0	Initial
0	**1**	**1**		**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**Arith. Shift R**

EG both positive (5X4)

- Multiplicand (M)=0101 (5), Multiplier (Q)= 0100 (4)

SC				A				Q					OPERATION
				A3	A2	A1	A0	Q3	Q2	Q1	Q0	Q-1	
1	0	0		0	0	0	0	0	1	0	0	0	Initial
0	1	1		0	0	0	0	0	0	1	0	0	Arith. Shift R
0	**1**	**0**		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**Arith. Shift R**

EG both positive (5X4)

- Multiplicand (M)=0101 (5), Multiplier (Q)= 0100 (4)

SC				A				Q					OPERATION
				A3	A2	A1	A0	Q3	Q2	Q1	Q0	Q-1	
1	0	0		0	0	0	0	0	1	0	0	0	Initial
0	1	1		0	0	0	0	0	0	1	0	0	Arith. Shift R
0	1	0		0	0	0	0	0	0	0	1	0	Arith. Shift R
				0	**0**	**0**	**0**	**A**					**Q0 Q-1=10 A<-A-M**
			+	**1**	**0**	**1**	**1**	**2's complement of M**					
				1	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	

EG both positive (5X4)

- Multiplicand (M)=0101 (5), Multiplier (Q)= 0100 (4)

SC				A				Q					OPERATION
				A3	A2	A1	A0	Q3	Q2	Q 1	Q 0	Q-1	
1	0	0		0	0	0	0	0	1	0	0	0	Initial
0	1	1		0	0	0	0	0	0	1	0	0	Arith. Shift R
0	1	0		0	0	0	0	0	0	0	1	0	Arith. Shift R
				0	0	0	0	A					Q0 Q-1=10 A<-A-M
			+	1	0	1	1	2's complement of M					
				1	0	1	1	0	0	0	1	0	
0	**0**	**1**		**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**A.S.R**

EG both positive (5X4)

SC				A				Q					OPERATION
				A3	A2	A1	A0	Q3	Q2	Q1	Q0	Q-1	
1	0	0		0	0	0	0	0	1	0	0	0	Initial
0	1	1		0	0	0	0	0	0	1	0	0	Arith. Shift R
0	1	0		0	0	0	0	0	0	0	1	0	Arith. Shift R
				0	0	0	0	A					Q0 Q-1=10 A<-A-M
			+	1	0	1	1	2's complement of M					
				1	0	1	1	0	0	0	1	0	
0	0	1		1	1	0	1	1	0	0	0	1	A.S.R
				1	**1**	**0**	**1**	**A**					**Q0 Q-1=01 A<-A+M**
			+	**0**	**1**	**0**	**1**	**M**					
				0	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	

EG

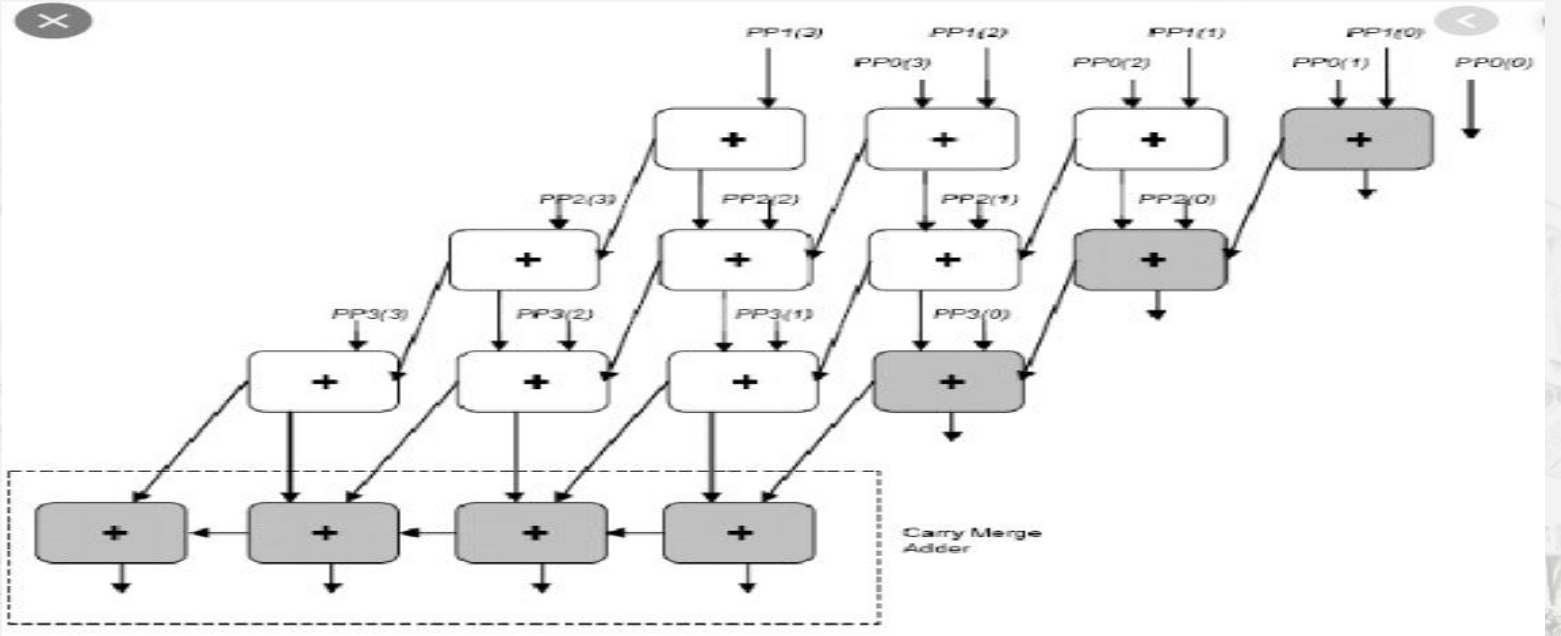
SC				A				Q					OPERATION
				A3	A2	A1	A0	Q3	Q2	Q1	Q0	Q-1	
1	0	0		0	0	0	0	0	1	0	0	0	Initial
0	1	1		0	0	0	0	0	0	1	0	0	Arith. Shift R
0	1	0		0	0	0	0	0	0	0	1	0	Arith. Shift R
				0	0	0	0	A					Q0 Q-1=10 A<-A-M
			+	1	0	1	1	2's complement of M					
				1	0	1	1	0	0	0	1	0	
0	0	1		1	1	0	1	1	0	0	0	1	A.S.R
				1	1	0	1	A					Q0 Q-1=01 A<-A+M
			+	0	1	0	1	M					
				0	0	1	0	1	0	0	0	1	
0	**0**	**0**		**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**A.S.R**

EG

SC				A				Q					OPERATION
				A3	A2	A1	A0	Q3	Q2	Q1	Q0	Q-1	
1	0	0		0	0	0	0	0	1	0	0	0	Initial
0	1	1		0	0	0	0	0	0	1	0	0	Arith. Shift R
0	1	0		0	0	0	0	0	0	0	1	0	Arith. Shift R
				0	0	0	0	A					Q0 Q-1=10 A<-A-M
			+	1	0	1	1	2's complement of M					
				1	0	1	1	0	0	0	1	0	
0	0	1		1	1	0	1	1	0	0	0	1	A.S.R
				1	1	0	1	A					Q0 Q-1=01 A<-A+M
			+	0	1	0	1	M					
				0	0	1	0	1	0	0	0	1	
0	0	0		0	0	0	1	0	1	0	0	0	A.S.R
RESULT				**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	

CARRY SAVE MULTIPLIER

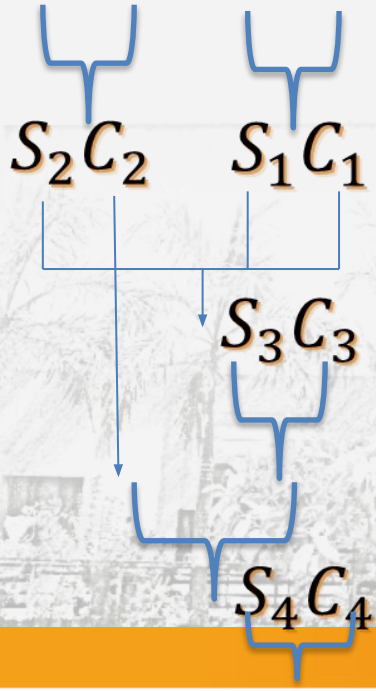
- The carry-save array multiplier uses an array of carry-save adders for the accumulation of partial product. It uses a carry-propagate adder for the generation of the final product. This reduces the critical path delay of the multiplier since the carry-save adders pass the carry to the next level of adders rather than the adjacent ones.



EXAMPLE OF CARRY SAVE MULTIPLIER

				1	**0**	**1**	**1**	
			X	1	1	0	1	
				1	0	1	1	A
			0	0	0	0	X	B
		1	0	1	1	x	X	C
	1	0	1	1	x	x	X	D

C D B A



EXAMPLE OF CARRY SAVE MULTIPLIER

STEP 1

		1	0	1	1	A
+	0	0	0	0	x	B
	0	1	0	1	1	S1
	0	0	0	0	x	C1

STEP 2

	1	0	1	1	x	x	C
1	0	1	1	x	x	x	D
1	1	1	0	1	x	x	S2
0	0	1	0	0	x	x	C2

STEP 3

		0	1	0	1	1	S1
		0	0	0	0	x	C1
1	1	1	0	1	x	x	S2
1	1	1	1	1	1	1	S3
0	0	0	0	0	0	x	C3

STEP 4

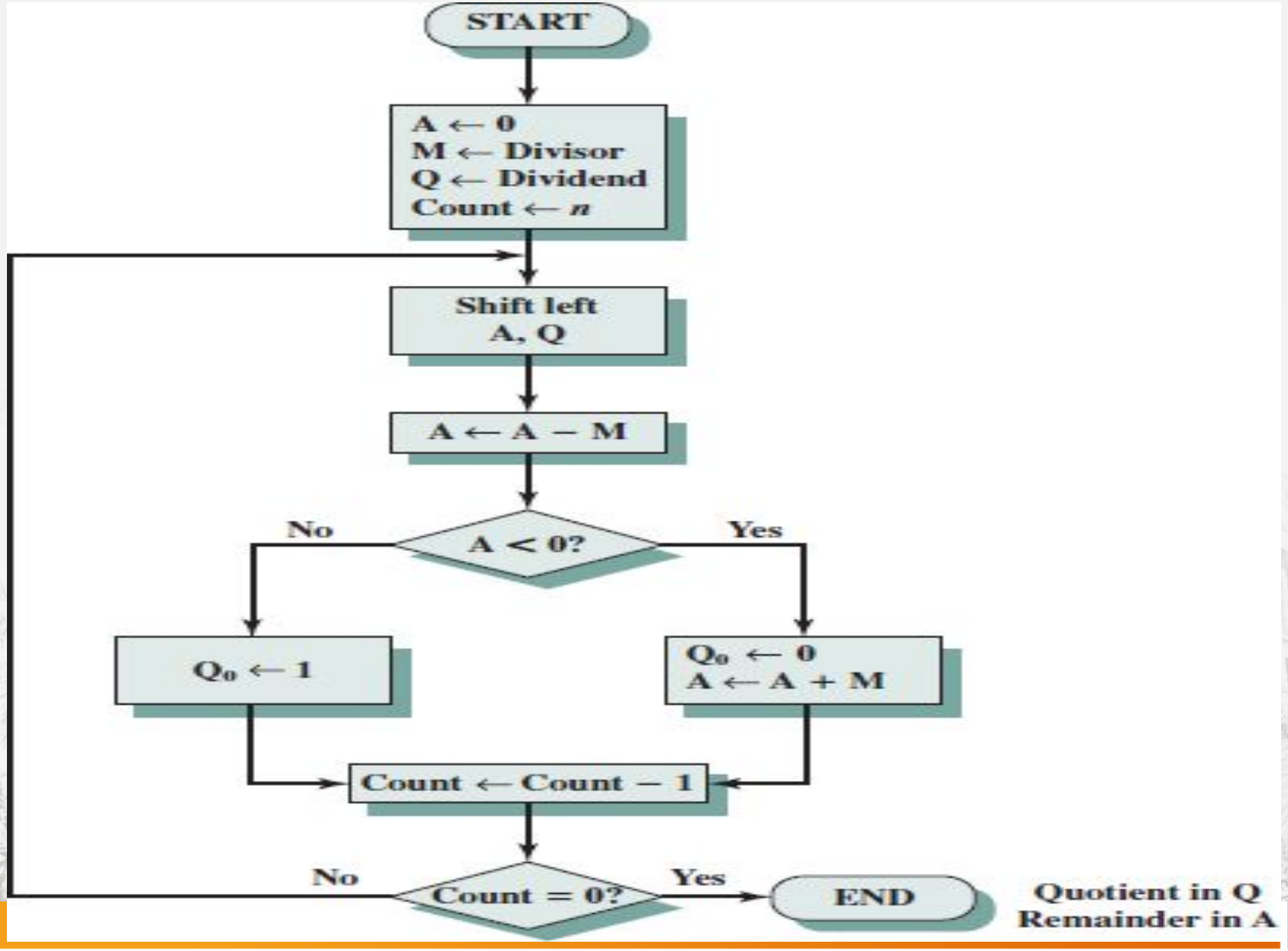
1	1	1	1	1	1	1	S3
0	0	0	0	0	0	X	C3
0	0	1	0	0	x	X	C2
1	1	0	1	1	1	1	S4
0	1	0	0	0	0	x	C4

EXAMPLE OF CARRY SAVE MULTIPLIER

STEP 5								
	1	1	0	1	1	1	1	S4
	0	1	0	0	0	x	x	C4
1	0	0	0	1	1	1	1	

RESULT

RESTORING DIVISION ALGORITHM



- STEP 1: Shift A and Q left one binary position.
- STEP 2: Subtract divisor (M) from A and place answer back in A

  (A=A-M)

STEP 3: If Sign Bit of A is 1 Set Q0 to 0 and ADD Divisor back to A. otherwise Set Q0 to 1.

STEP 4: Decrement Count By 1 and Repeat steps 1,2 & 3 N times.

Example

- Dividend=1010 (Q)
- Divisor= 0011 (M) 2's complement 1 1 1 0 1

	A Register					Q Register				SC
Initially	0	0	0	0	0	1	0	1	0	1 0 0 (4)
Shift left A, Q	0	0	0	0	1	0	1	0		
Subtract M	1	1	1	0	1					
Set Q0	1	1	1	1	0					0 1 1 (3)
Restore (A+B	0	0	0	1	1	0	1	0	0	
	0	0	0	0	1					

First cycle

Example

- Dividend=1010 (Q)
- Divisor= 0011 (M) 2's complement 1 1 1 0 1

	A Register					Q Register				SC
Aft 1st cycle	0	0	0	0	1	0	1	0	0	
Shift left A, Q	0	0	0	1	0	1	0	0		
Subtract M	1	1	1	0	1					
Set Q0	1	1	1	1	1					0 1 0 (2)
Restore (A+B	0	0	0	1	1					
	0	0	0	1	0	1	0	0	0	

second cycle

Example

- Dividend=1010 (Q)
- Divisor= 0011 (M) 2's complement 1 1 1 0 1

	A Register					Q Register				SC
Aft 2nd cycle	0	0	0	1	0	1	0	0	0	
Shift left A, Q	0	0	1	0	1	0	0	0	0	
Subtract M	1	1	1	0	1					
Set Q0	0	0	0	1	0					0 0 1 (1)
	0	0	0	1	0	0	0	0	1	

third cycle

Example

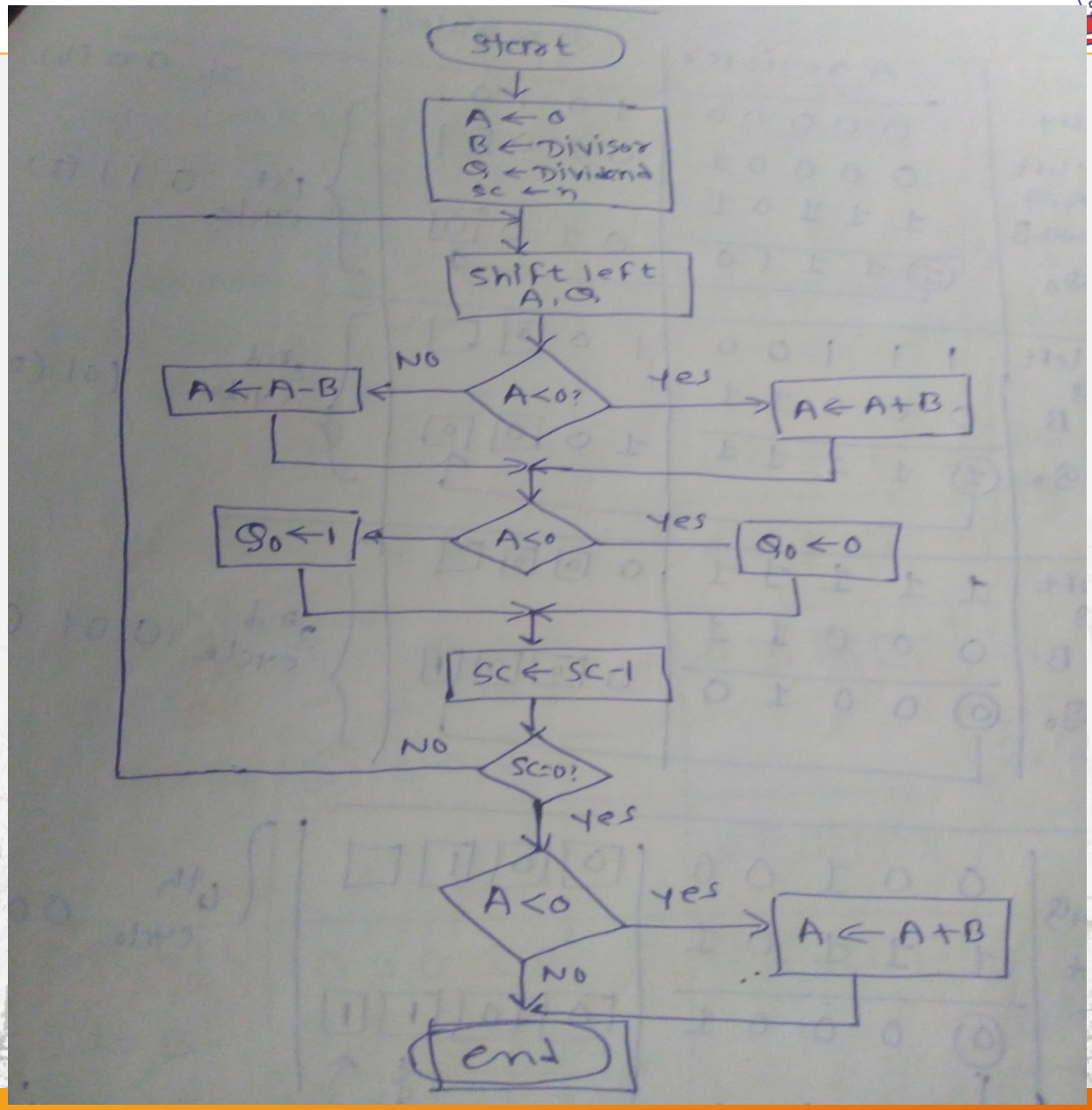
- Dividend=1010 (Q)
- Divisor= 0011 (M) 2's complement 1 1 1 0 1

	A Register					Q Register				SC
Aft 3rd cycle	0	0	0	1	0	0	0	0	1	
Shift left A, Q	0	0	1	0	0	0	0	1	0	
Subtract M	1	1	1	0	1					
Set Q0	0	0	0	0	1					0 0 0 (1)
ANS	0	0	0	0	1	0	0	1	1	

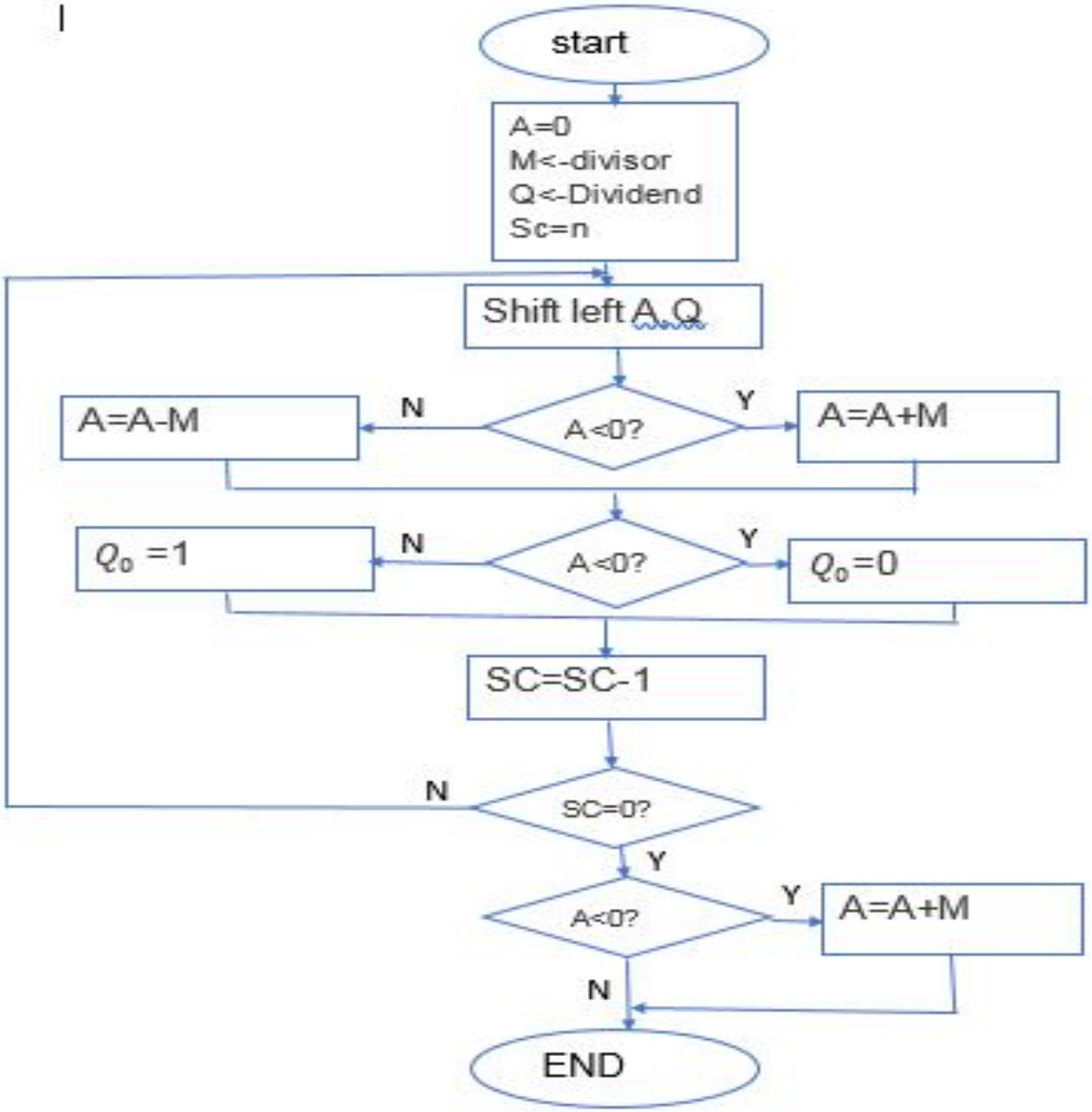
4th cycle

REMAINDER QUOTIENT

NON RESTORING DIVISION



NON RESTORING DIVISION



- STEP 1:
- If the sign of A is 0, shift A and Q left. And subtract Divisor (M) from A. otherwise Shift A & Q Left and ADD Divisor to A.
- If Sign of A is 0, Set Q0 =1 other wise set Q0=0
- Step 2 Repeat Steps 1 and 2 n Times.
- Step 3 if the Sign of A is 1, ADD M to A.

EXAMPLE

- Dividend (Q)= 1010
- Divisor (M)= 0011

	A Register					Q Register				SC
Initially	0	0	0	0	0	1	0	1	0	100 (4)
Shift left A,Q	0	0	0	0	1	0	1	0		
Subtract M	1	1	1	0	1					011(3)
SET Q0	1	1	1	1	0	0	1	0	0	

1st cycle

EXAMPLE

- Dividend (Q)= 1010
- Divisor (M)= 0011

	A Register					Q Register				SC
Aft 1st cycle	1	1	1	1	0	0	1	0	0	
Shift left A,Q	1	1	1	0	0	1	0	0	0	
ADD M	0	0	0	1	1					101(2)
SET Q0	1	1	1	1	1	1	0	0	0	

2nd cycle

EXAMPLE

- Dividend (Q)= 1010
- Divisor (M)= 0011

	A Register					Q Register				SC
Aft 2nd cycle	1	1	1	1	1	1	0	0	0	
Shift left A,Q	1	1	1	1	1	0	0	0	0	
ADD M	0	0	0	1	1					001(1)
SET Q0	0	0	0	1	0	0	0	0	1	

3rd cycle

EXAMPLE

- Dividend (Q)= 1010
- Divisor (M)= 0011

	A Register					Q Register				SC
Aft 3rd cycle	0	0	0	1	0	0	0	0	1	
Shift left A,Q	0	0	1	0	0	0	0	1	0	
Subtract M	1	1	1	0	1					001(1)
SET Q0	0	0	0	0	1	0	0	1	1	

4th cycle

Result	0	0	0	0	1	0	0	1	1

REMAINDER

QUOTIENT

FLOATING POINT ARTHMETIC

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E}$ $X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E}$ } $X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

FLOATING POINT ARTHMETIC EXAMPLE

- $X = 0.5 \times 10^{2}$
- $Y = 0.8 \times 10^{3}$
- $X + Y = (0.5 \times 10^{2-3} + 0.8) \times 10^{3} = 0.85 \times 10^{3} = 850$
- $X - Y = (0.5 \times 10^{2-3} - 0.8) \times 10^{3} = -0.75 \times 10^{3} = -750$
- $X \times Y = (0.5 \times 0.8) \times 10^{2+3} = 0.4 \times 10^{5}$
- $X \div Y = (0.5 \div 0.8) \times 10^{2-3} = 0.625 \times 10^{-1}$

- $X = 0.5 \times 10^5$
- $Y = 0.8 \times 10^3$

- $X + Y = (0.8 \times 10^{3-5} + 0.5) \times 10^5 = 0.508 \times 10^5$

- $X - Y = (0.8 \times 10^{3-5} - 0.5) \times 10^5 = -0.492 \times 10^5$

- $X \times Y = (0.5 \times 0.8) \times 10^{5+3} = 0.4 \times 10^8$

- $X \div Y = (0.5 \div 0.8) \times 10^{5-3} = 0.625 \times 10^2$