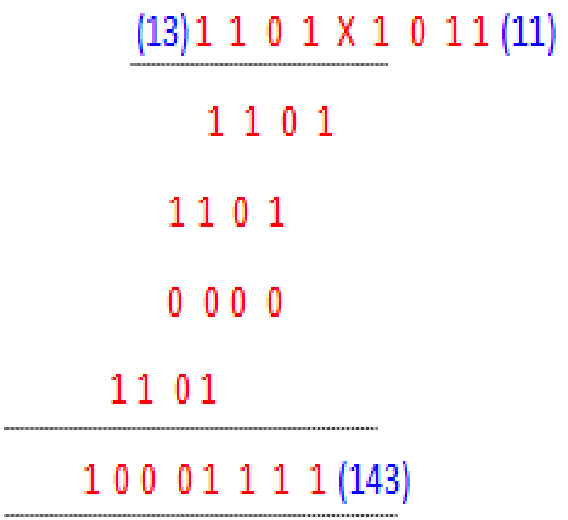
MULTIPLICATION AND DIVISION

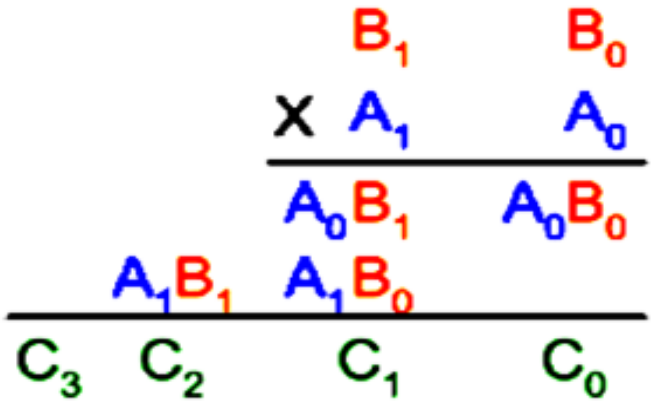
Binary Multiplier

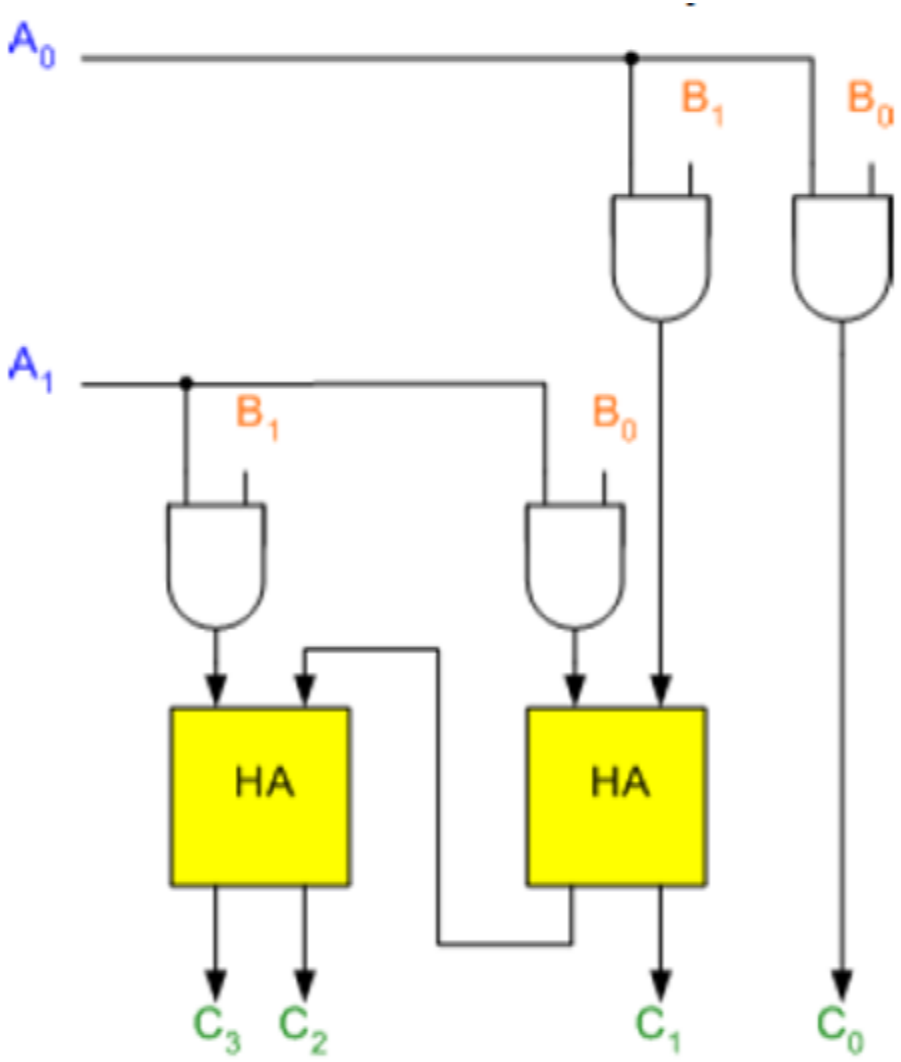
- Multiplication of binary numbers is performed in the same way as with decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit.
- The result of each such multiplication forms a partial product. Successive partial products are shifted one bit to the left.
- The product is obtained by adding these shifted partial products.
- Example 1: Consider multiplication of two numbers, say A and B (2 bits each),
- C = A x B.

- The first partial product is formed by multiplying the B1B0 by A0. The multiplication of two bits such as A0 and B0 produces a 1 if both bits are 1; otherwise it produces a 0 like an AND operation. So the partial products can be implemented with AND gates.
- The second partial product is formed by multiplying the B1B0 by A1 and is shifted one position to the left.
- The two partial products are added with two half adders (HA). Usually there are more bits in the partial products and



Binary Multiplier





The least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate as shown in the Figure.

Shift-and-Add Multiplier

- Shift-and-add multiplication is similar to the multiplication performed by paper and pencil.
- This method adds the multiplicand X to itself Y times, where Y denotes the multiplier.
- To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.

Shift-and-Add Multiplier

- As an example, consider the multiplication of two unsigned 4-bit numbers,
- 8 (1000) and 9 (1001).

```
Multiplicand                1000 x
Multiplier                  1001
                            1000
                           0000
                          0000
                         1000
Product                  1001000
```

Shift-and-Add Multiplier

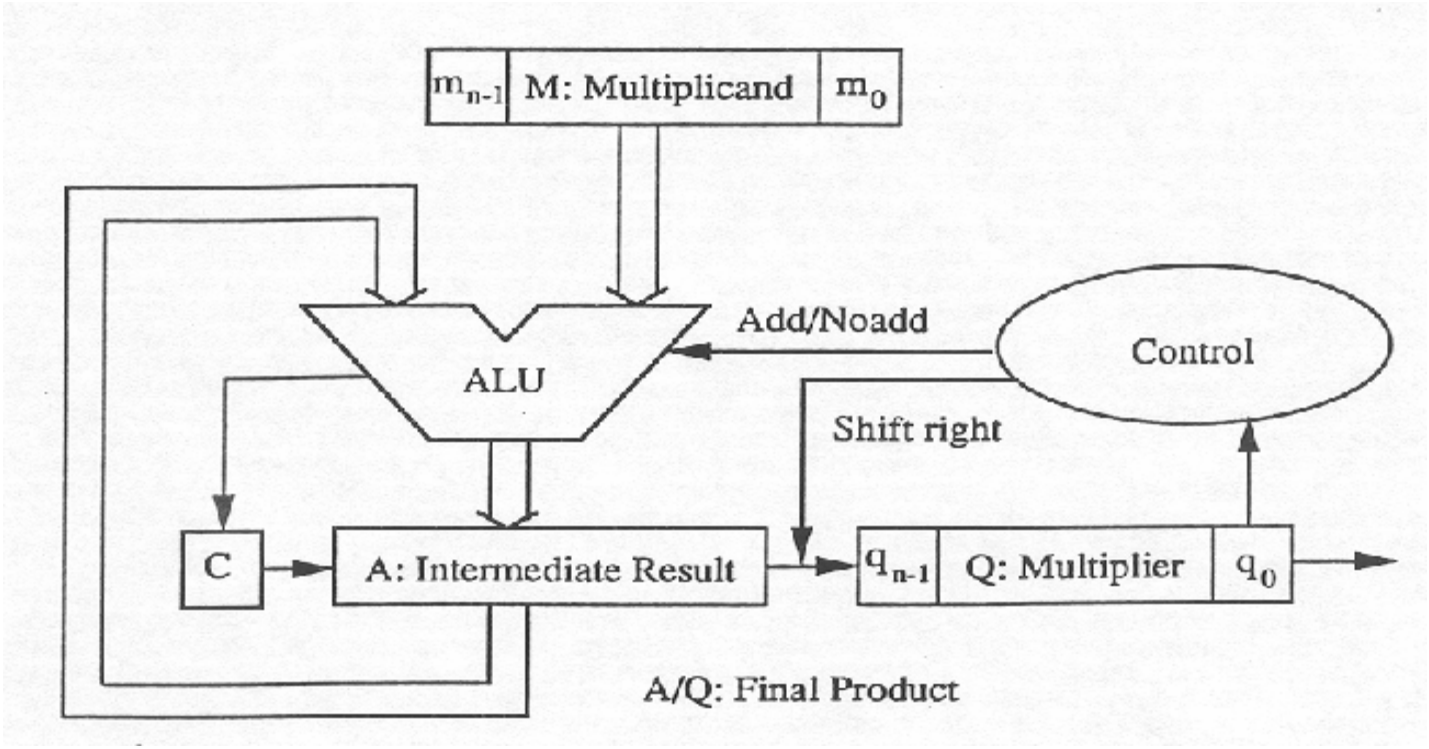
- In the case of binary multiplication, since the digits are 0 and 1, each step of the multiplication is simple.
- If the multiplier digit is 1, a copy of the multiplicand (1 ×multiplicand) is placed in the proper positions;
- If the multiplier digit is 0, a number of 0 digits (0 × multiplicand) are placed in the proper positions.
- Consider the multiplication of positive numbers. The first version of the multiplier circuit, which implements the shift-and-add multiplication method for two n-bit

MULTIPLICATION OF POSITIVE BINARY NUMBERS

1. Multiplication of two fixed point binary number in signed magnitude representation is done with process of successive shift and add operation.
2. In the multiplication process we are considering successive bits of the multiplier, least significant bit first.
3. If the multiplier bit is 1, the multiplicand is copied down else 0's are copied down.
4. The numbers copied down in successive lines are shifted one position to the left from the previous number.
5. Finally numbers are added and their sum form the product.
6. The sign of the product is determined from the sign of the multiplicand and multiplier. If they are alike, sign of the product is positive else negative.
7. **M bits x N bits= (M+N) bit product**

```
  10111 (Multiplicand)
x 10011 (Multiplier)
-------
  10111
  10111
 00000
00000
10111
---------
011011010   (Product)
---------
```

HARDWARE IMPLEMENTATION OF MULTIPLICATION



Limitations:

1. The ALU is twice as wide as necessary.
2. The multiplicand register takes twice as many bits as needed .
3. The product register won't need 2n bits till the last step – Being filled.

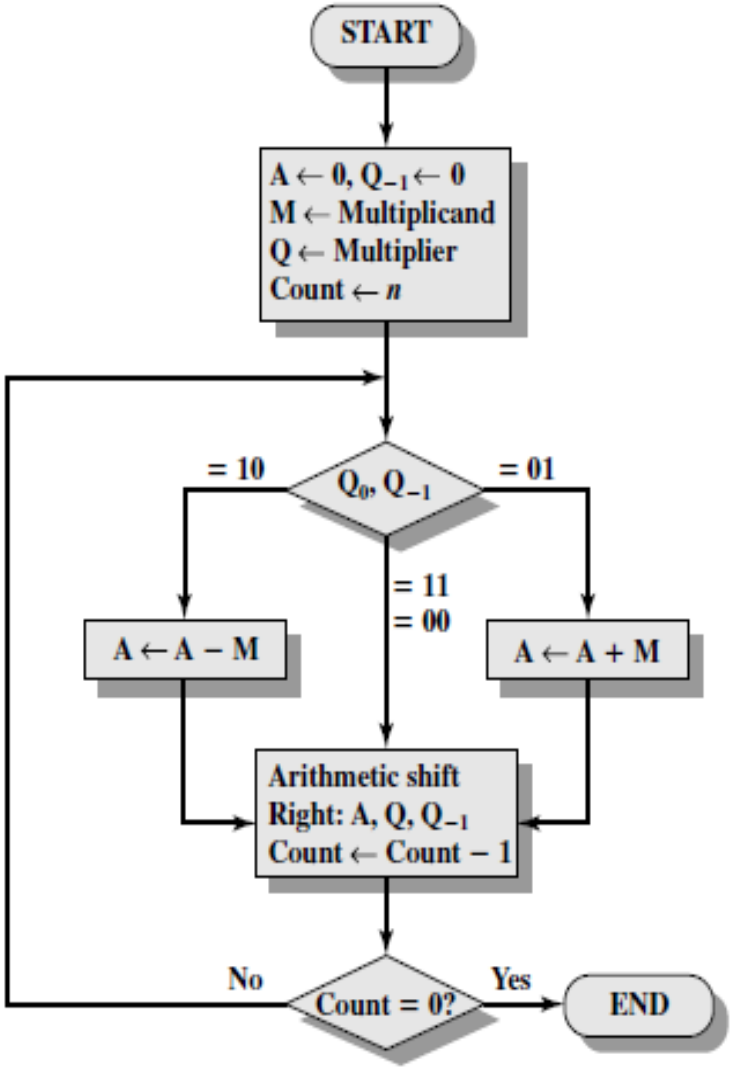
4. The multiplier register is being emptied during the process

BOOTH'S MULTIPLICATION

1. Multiply 7 and 3.
M=0111; Q=0011; -M=1001

C	ACC	Q	Q_{n-1}	INITIAL VALUE
4	0000	0011	0	Initial
	1001	0011	0	A□A-M
3	1100	1001	1	ASR ACC, Q, Q_{n-1}
2	1110	0100	1	ASR ACC, Q, Q_{n-1}
	0101	0100	1	A□A+M
1	0010	1010	0	ASR ACC, Q, Q_{n-1}
0	0001	0101	0	ASR ACC, Q, Q_{n-1}

Product=ACC.Q=00010101=10101=21

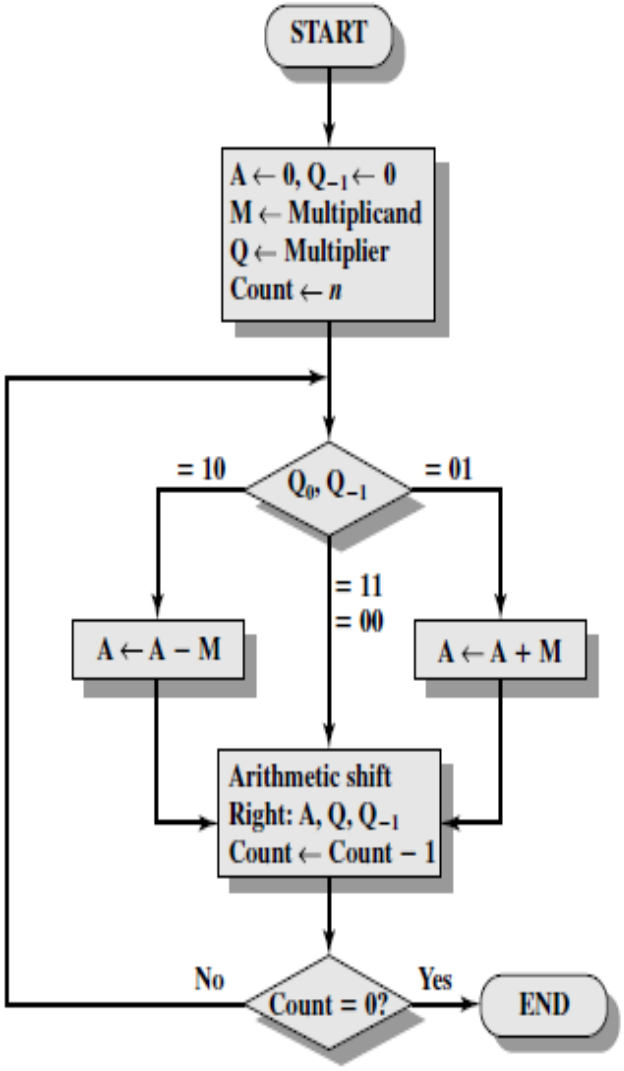


BOOTH'S MULTIPLICATION

2. 15x 14

M=01111 (15) Q=01110(14) -M=10001

C	ACC	Q	Q_{n-1}	INITIAL VALUE
5	00000	01110	0	Initial
4	00000	00111	0	ASR ACC, Q, Q_{n-1}
	10001	00111	0	A□A-M
3	11000	10011	1	ASR ACC, Q, Q_{n-1}
2	11100	01001	1	ASR ACC, Q, Q_{n-1}
1	11110	00100	1	ASR ACC, Q, Q_{n-1}
	01101	00100	1	A□A+M
0	00110	10010	0	ASR ACC, Q, Q_{n-1}



ANS: 11010010

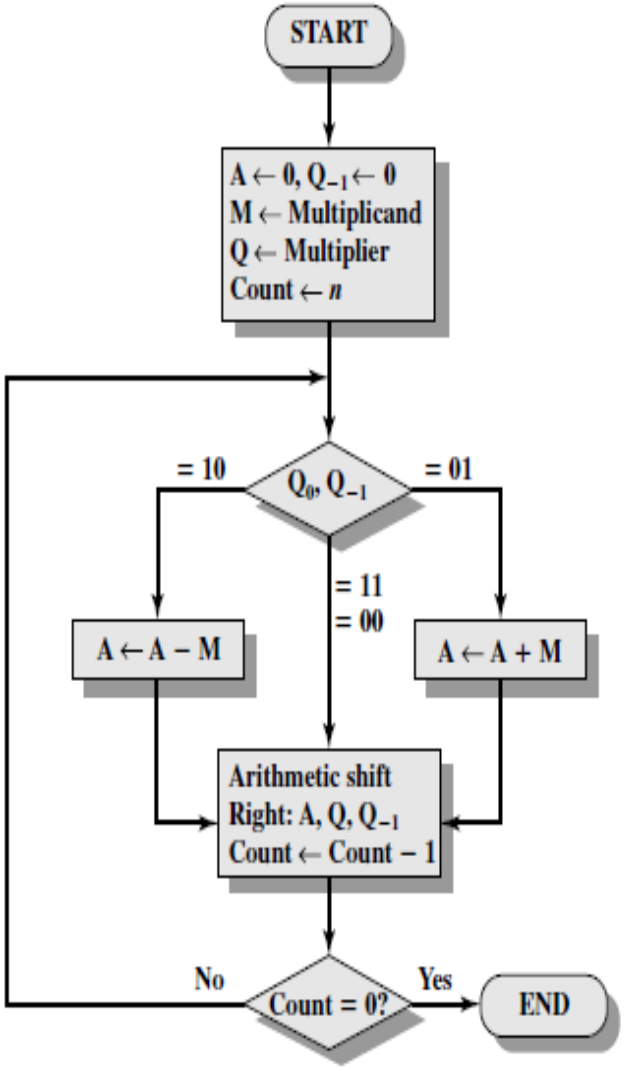
BOOTH'S MULTIPLICATION

3. -5x -4

M=1011 (-5) Q=1100(-4) -M=0101

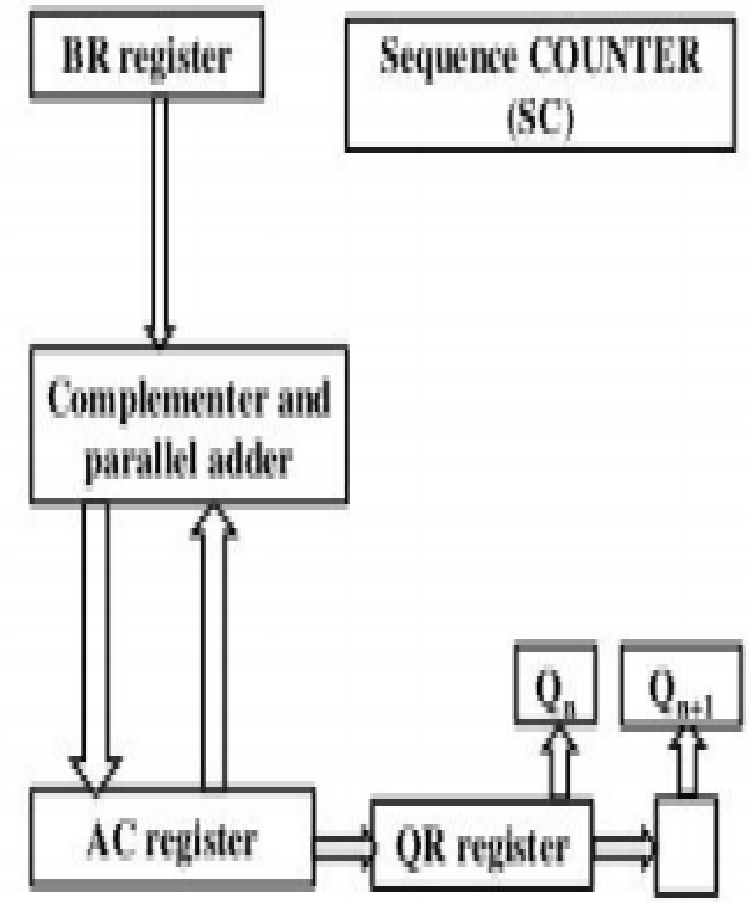
C	ACC	Q	Q_{n-1}	INITIAL VALUE
4	0000	1100	0	Initial
3	0000	0110	0	ASR ACC, Q, Q_{n-1}
2	0000	0011	0	ASR ACC, Q, Q_{n-1}
	0101	0011	0	A A-M
1	0010	1001	1	ASR ACC, Q, Q_{n-1}
0	0001	0100	1	ASR ACC, Q, Q_{n-1}

ANS: 10100



BOOTH'S ALGORITHM

1. Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement representation in efficient way, i.e., less number of additions/ subtractions required.
2. Sign bits are not separated from rest of the registers.
3. Qn holds LSB of the multiplier in register QR.
4. Flip flop Q n+1 is appended to facilitate double bit inspection of the multiplier.



FAST MULTIPLICATION

1. BOOTH RECODING OF MULTIPLERS
2. BIT PAIR RECODING OF MULTIPLIERS
3. CARRY SAVE ADDITION OF SUMMANDS

The BOOTH RECODED MULTIPLIER

- Booth multiplication reduces the number of additions for intermediate results, but can sometimes make it worse as we will see.
- Booth multiplier recoding table

Multiplier		Version of multiplicand selected by bit
Bit *i*	Bit *i*-1	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

FIND THE RECODED MULTIPLER

Multiplier		Version of multiplicand selected by bit
Bit *i*	Bit *i*-1	
0	0	0 × M
0	1	+1 × M
1	0	–1 × M
1	1	0 × M

1. 30_{10}

Binary value: 11110

Add sign bit: 011110

Add augmented 0: 0111100

Recoded multiplier:

0		1		1		1		1		0		0
	+1		0		0		0		-1		0	

Recoded multiplier= +1 0 0 0 -1 0

2. 100_{10}

Binary value: 1 1 0 0 1 0 0

Add sign bit: 01100100

Add augmented 0: 011001000

011001000

0		1		1		0		0		1		0		0		0
	+1		0		-1		0		+1		-1		0		0	

Recoded multiplier= +1 0 -1 0 +1 -1 0 0

FIND THE RECODED MULTIPLER

3. -11
Binary value: 1011
Two's complement:0101
Insert sign bit: 1 0101
Add augmented 0: 101010

101010

1		0		1		0		1		0
	-1		+1		-1		+1		-1	

Multiplier		Version of multiplicand selected by bit
Bit *i*	Bit *i*-1	
0	0	0 × M
0	1	+1 × M
1	0	–1 × M
1	1	0 × M

Recoded multiplier= -1 +1 -1 +1 -1

0 1 1 0 1 x 0 1 0 1 1

-1 +1 -1 +1 -1

1	1	1	1	1	1	0	0	1	1
0	0	0	0	0	1	1	0	1	
1	1	1	1	0	0	1	1		
0	0	0	1	1	0	1			
1	1	0	0	1	1				
1	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**1**

2's complement of -143= 1101110001

Steps to perform Booth's bit pair recoding

1. Make the number of bits in multiplier and multiplicand equal.
2. Find twos complement of negative number
3. Insert sign bit.
4. Add augmented zero
5. Find bit pair recoding of multiplier
6. Extend the sign bits of partial product1 to make it as 2n bits.

6.1 If the recoded multiplier is +1, then place the multiplicand as such in partial product

6.2 If the recoded multiplier is -1, then take two's complement of multiplicand and place it as partial product.

6.3 If the recoded multiplier is 0, then place 0's in partial product.

Perform multiplication of 13 x 11 using Booth's bit pair recoding

13: 01101

11:01011

Augment 0: 010110

0		1		0		1		1		0
	+1		-1		+1		0		-1	

Recoded multiplier= +1 -1 +1 0 -1

Recoded multiplier= +1 -1 +1 0 -1

0 1 1 0 1 x 0 1 0 1 1

+1 -1 +1 0 -1

1	1	1	1	1	1	0	0	1	1
0	0	0	0	0	0	0	0	0	
0	0	0	0	1	1	0	1		
1	1	1	0	0	1	1			
0	0	1	1	0	1				
0	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**

10001111=+143

FAST MULTIPLICATION: BOOTHS MODIFIED BIT PAIR RECODING

1. This bit-pair recoding technique halves the maximum number of summands. It is derived from the Booth algorithm.
2. Group the Booth-recoded multiplier bits in pairs, and observe the following: The pair (+1 -1) is equivalent to the pair (0 +1).
3. That is, instead of adding —1 times the multiplicand M at shift position i to + 1 x M at position i + 1, the same result is obtained by adding +1 x M at position I Other examples are: (+1 0) is equivalent to (0 +2) (-1

i+1	i	i-1	At position i of multiplicand
0	0	0	0
0	0	1	+1
0	1	0	+1
0	1	1	+2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Steps to perform Booth's bit pair recoding

1. Make the number of bits in multiplier and multiplicand equal.
2. Find twos complement of negative number
3. Insert sign bit.
4. Find bit pair recoding of multiplier
5. Extend the sign bits of partial product1 to make it as 2n bits.

6.1 If the recoded multiplier is +2, then multiply the multiplicand by 10.

6.2 If the recoded multiplier is -2, then take 2's complement of multiplicand and multiply it by 10

6.3 If the recoded multiplier is -1, then take 2'c complement of the multiplicand

6.4 If the recoded multiplier is +1, then multiply the multiplicand by 1.

Perform multiplication of -11 and 27 using modified Booth's bit pair recoding.

Binary value of 11: 1011

Binary value of 27: 011011

Twos complement of -11:110101

Modified bit pair recoding:

Add implied 0: 0110110

0	1	1	0	1	1	0
	+2		-1		-1	

Recoded multiplier: +2 -1 -1

1 1 0 1 0 1 x 011011

+2 -1 -1

0	0	0	0	0	0	0	0	1	0	1	1
0	0	0	0	0	0	1	0	1	1		
1	1	1	0	1	0	1	0				
1	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**1**

Twos complement of -297= 111011010111

Perform multiplication of 13and -6 using modified Booth's bit pair recoding.

Binary value of 13: 01101

Binary value of 6: 110

Twos complement -6:11010

Modified bit pair recoding:

Add implied 0: 110100

1	1	1	0	1	0	0
	0		-1		-2	

If a three bit pairing cannot be done, then extend the sign bit at MSB.

Recoded multiplier: 0 -1 -2

01101x11010

0 -1 -2

1	1	1	1	1	0	0	1	1	0
1	1	1	1	0	0	1	1		
0	0	0	0	0	0				
1	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**0**

Two complement of -78=1110110010

Carry-Save Addition of Summands

- A **carry-save adder** is a type of digital adder, used to efficiently compute the sum of three or more binary numbers.
- A carry-save adder (CSA), or 3-2 adder, is a very fast and cheap adder that does not propagate carry bits.
- A Carry Save Adder is generally used in binary multiplier, since a binary multiplier involves addition of more than two binary numbers after multiplication.
- It can be used to speed up addition of the several summands required in multiplication
- It differs from other digital adders in that it outputs two (or more) numbers, and the answer of the original summation can be achieved by adding these outputs together.
- A big adder implemented using this technique will usually be much faster than conventional addition of those numbers.

Fast Multiplication

- Bit pair recoding reduces summands by a factor of 2
- Summands are reduced by carry save addition
- Final product can be generated by using carry look ahead adder

Carry-Save Addition of Summands

- **Disadvantage** of the **Ripple Carry Adder -** Each full adder has to wait for its carry-in from its previous stage full adder. This increase propagation time. This causes a delay and makes ripple carry adder extremely slow. RCAr is very slow when adding many bits.
- **Advantage of the Carry Look ahead Adder -** This is an improved version of the Ripple Carry Adder. Fast parallel adder. It generates the carry-in of each full adder simultaneously without causing any delay. So, CLAr is faster (because of reduced propagation delay) than RCAr.
- **Disadvantage the Carry Look-ahead Adder** - It is costlier as it reduces the propagation delay by more complex hardware. It gets more complicated as the number of bits increases.

Operation: Carry-Save Addition of Summands

- Consider the addition of many summands, We can:

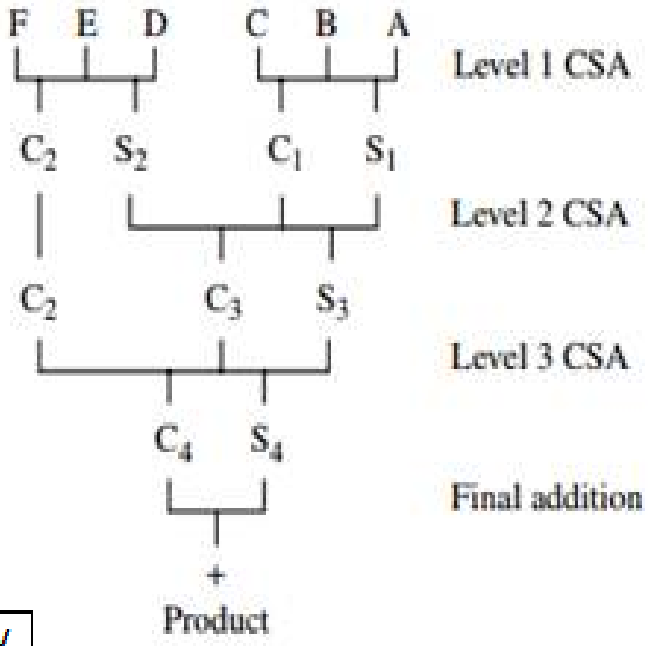
- Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
- Group all of the S and C vectors into threes, and perform carry-save addition of them, generating a further set of S and C vectors in one more full-adder delay
- Continue with this process until there are only two vectors remaining
- They can be added in a Ripple Carry Adder (RPA) or Carry Look-ahead Adder (CLA) to produce the desired product

Multiply 101101 x 111111 by carry save addition of summands

Group the multiplier into subgroups of 3 bits.

101101 x 111111

				1	0	1	1	0	1
			1	0	1	1	0	1	X
		1	0	1	1	0	1	X	X
S1		**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**
C1	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**x**



									1	0	1	1	0	1	x	x	X
								1	0	1	1	0	1	X	X	X	X
							1	0	1	1	0	1	X	X	X	X	X
					S2		**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**X**	**X**	**X**
					C2	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**X**	**x**	**X**	**x**

Add S1, C1, S2

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

Add S3, C3 and C2

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

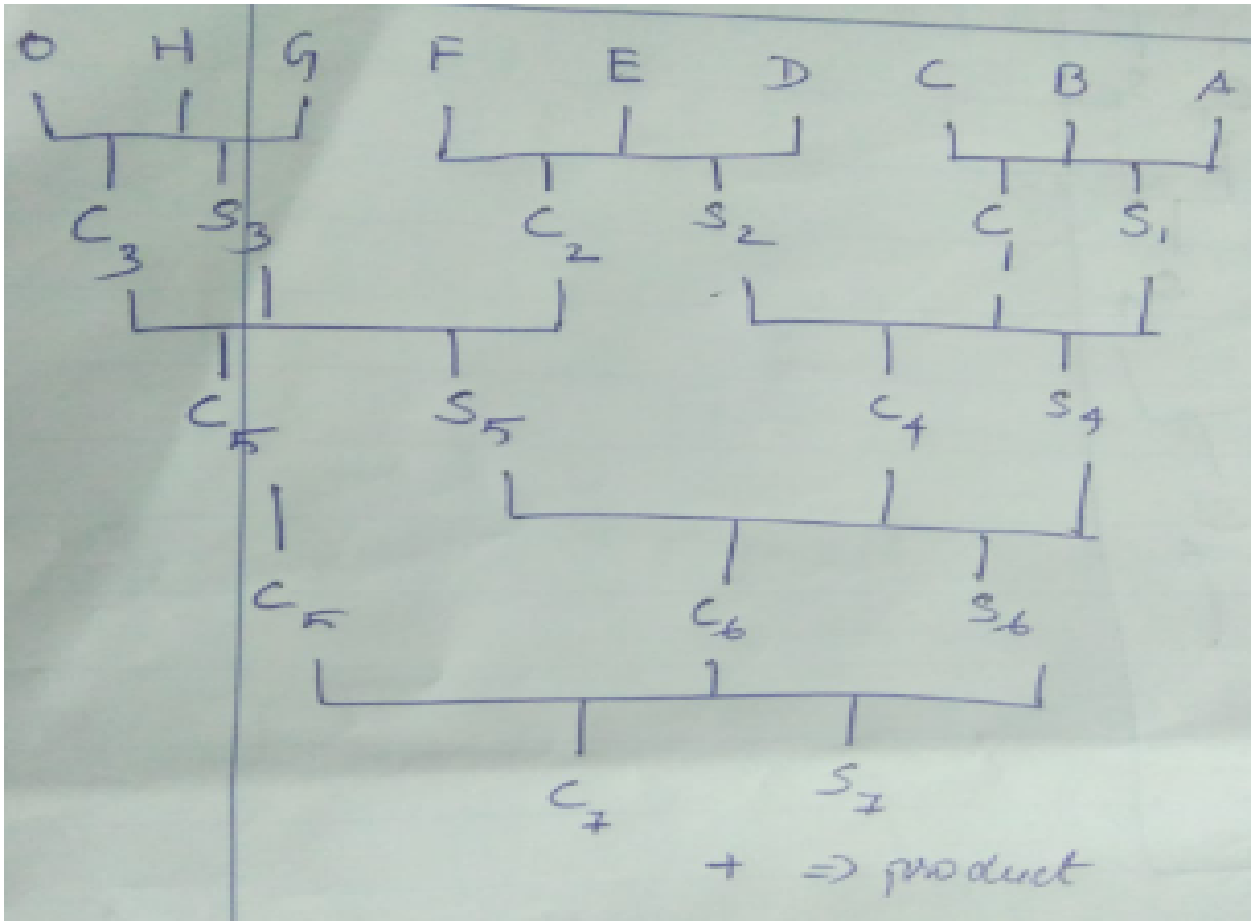
Add S4, C4

S4						0	1	0	1	1	1	0	1	0	0	1	1
C4					0	0	1	0	1	0	1	0	0	0	0	0	X
					0	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**

Product= 101100010011

CARRY SAVE ADDITION OF HIGHER NUMBERS

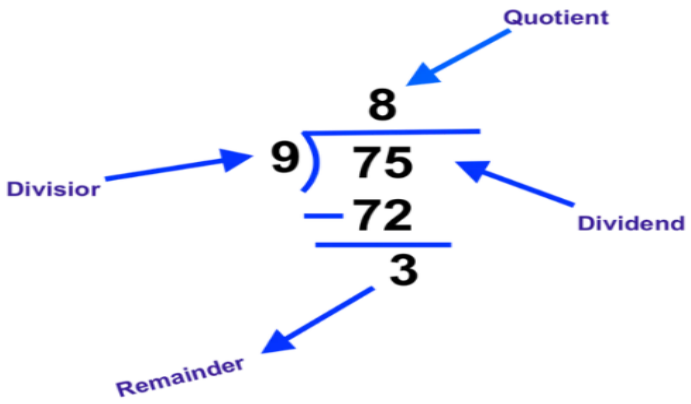
Grouping of 8 digit number.



Integer Division (Unsigned numbers)

Longhand Division operates as follows:

- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction



Restoring Division

- Similar to multiplication circuit
- An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation.
- Register A is set to 0
- After the division operation is complete, the n-bit quotient is in register Q and the remainder is in register A.
- The required subtractions are facilitated by using 2's complement arithmetic.
- The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.

Divide 11/3 using restoring division.

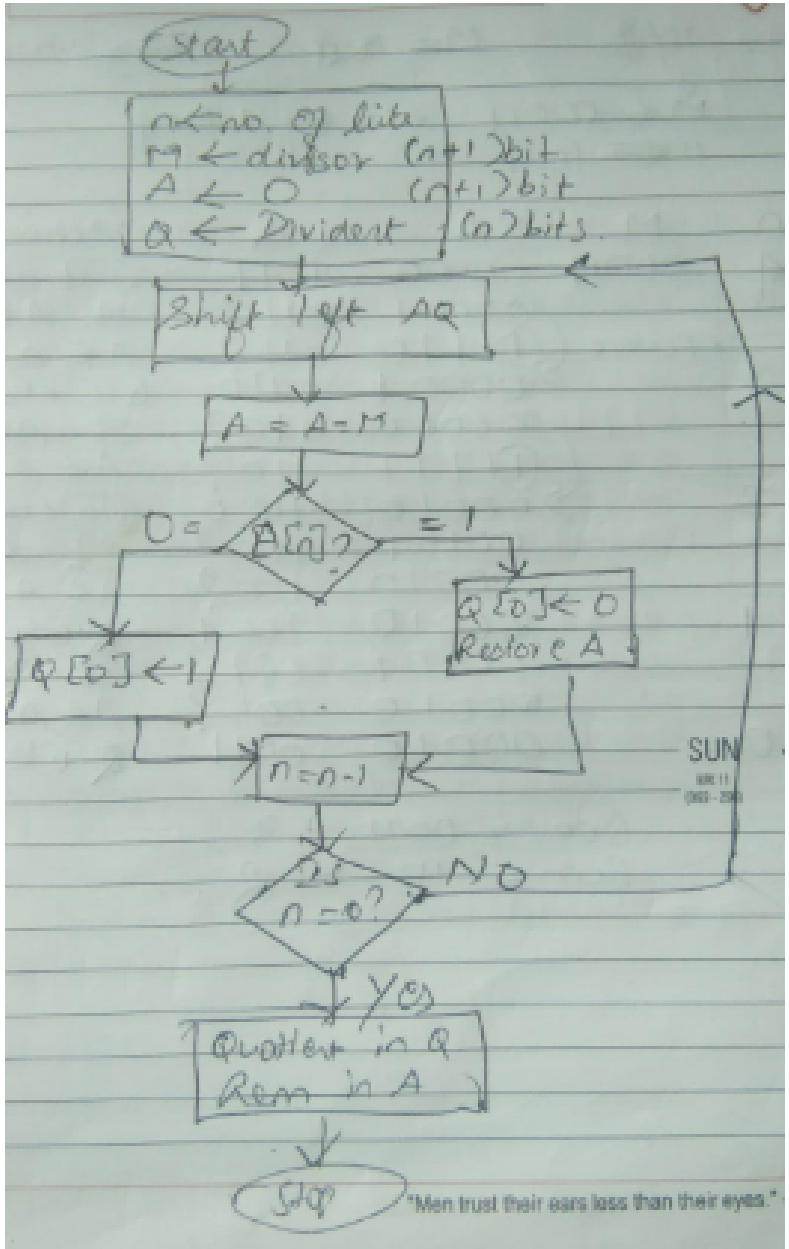
M=00011 Q=1011

-M=11101

N	A	Q	Action
4	00000	1011	Initialize
	00001	011?	SL AQ
	11110	011?	A←A-M
	11110	0110	Q0=0
3	00001	0110	Restore A
	00010	110?	SL AQ
	11111	110?	A←A-M
	11111	1100	Q0=0
2	00010	1100	Restore A
	00101	100?	SL AQ
	00010	100?	A←A-M
1	00010	1001	Q0=1
	00101	001?	SL AQ
	00010	001?	A←A-M
0	00010	0011	Q0=1

Quotient (Q)=0011

Remainder (A)=10

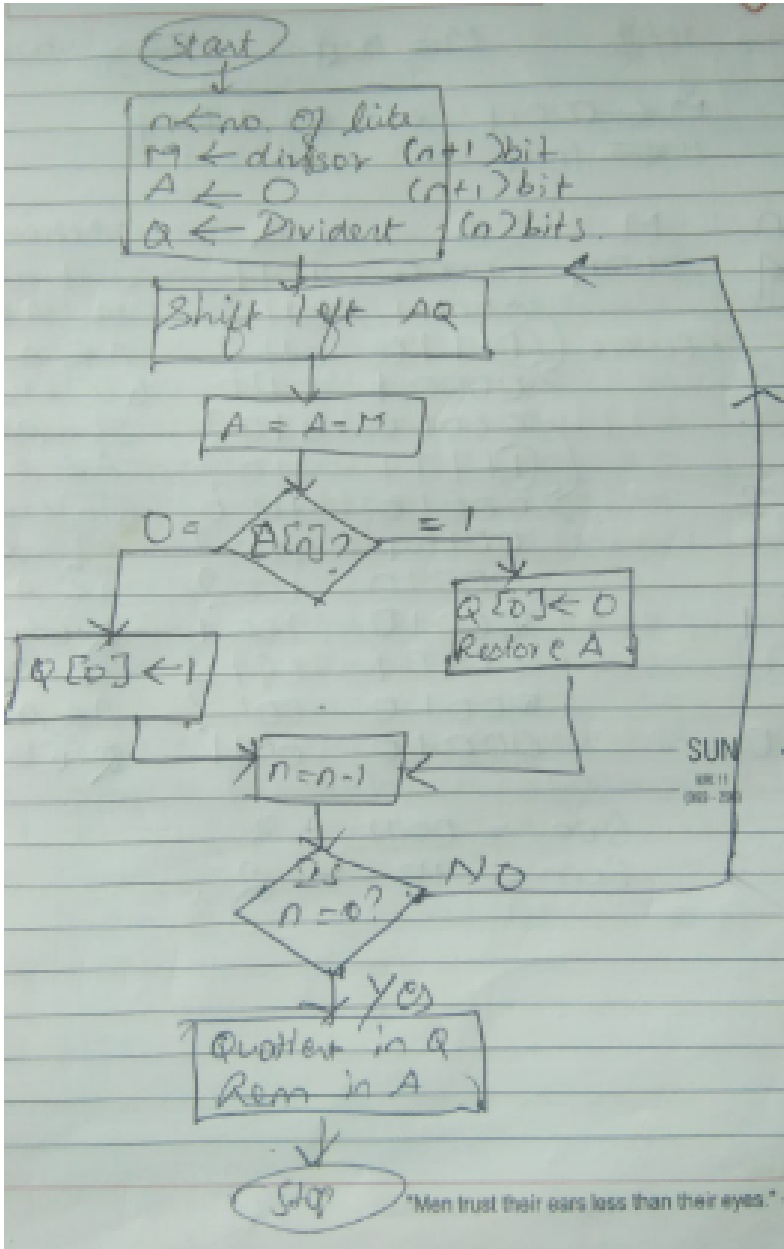


Divide 27/5 using restoring division.

M=000101 -M=111011
Q=11011

N	A	Q	Action
5	000000	11011	Initial
	000001	1011?	SL AQ
	111011	1011?	A←A-M
	111011	10110	Q0=0
4	000001	10110	Restore A
	000011	0110?	SL AQ
	111110	0110?	A←A-M
	111110	01100	Q0=0
3	000011	01100	Restore A
	000110	1100?	SL AQ
	000001	1100?	A←A-M
2	000001	11001	Q0=1
	000011	1001?	SL AQ
	111110	1001?	A←A-M
	111110	10010	Q0=10
1	000011	10010	Restore A
	000111	0010?	SL AQ
	000001	0010?	A←A-M
0	000010	00101	Q0=1

Quotient (Q)=101
Remainder (A)=10



Non-Restoring Division

- Initially Dividend is loaded into register Q,
  and n-bit Divisor is loaded into register M
- Let M' is 2's complement of M
- Set Register A to 0
- Set count to n
- SHL AQ denotes shift left AQ by one position leaving Q_0 blank.
- Similarly, a square symbol in Q_0 position denote, it is to be calculated later

Non-Restoring Division

Restoring division can be improved using non-restoring algorithm

The effect of restoring algorithm actually is:
If A is positive, we shift it left and subtract M, that is compute $2A-M$
If A is negative, we restore it $(A+M)$, shift it left, and subtract M, that is, $2(A+M)-M = 2A+M$.
Set q_0 to 1 or 0 appropriately.

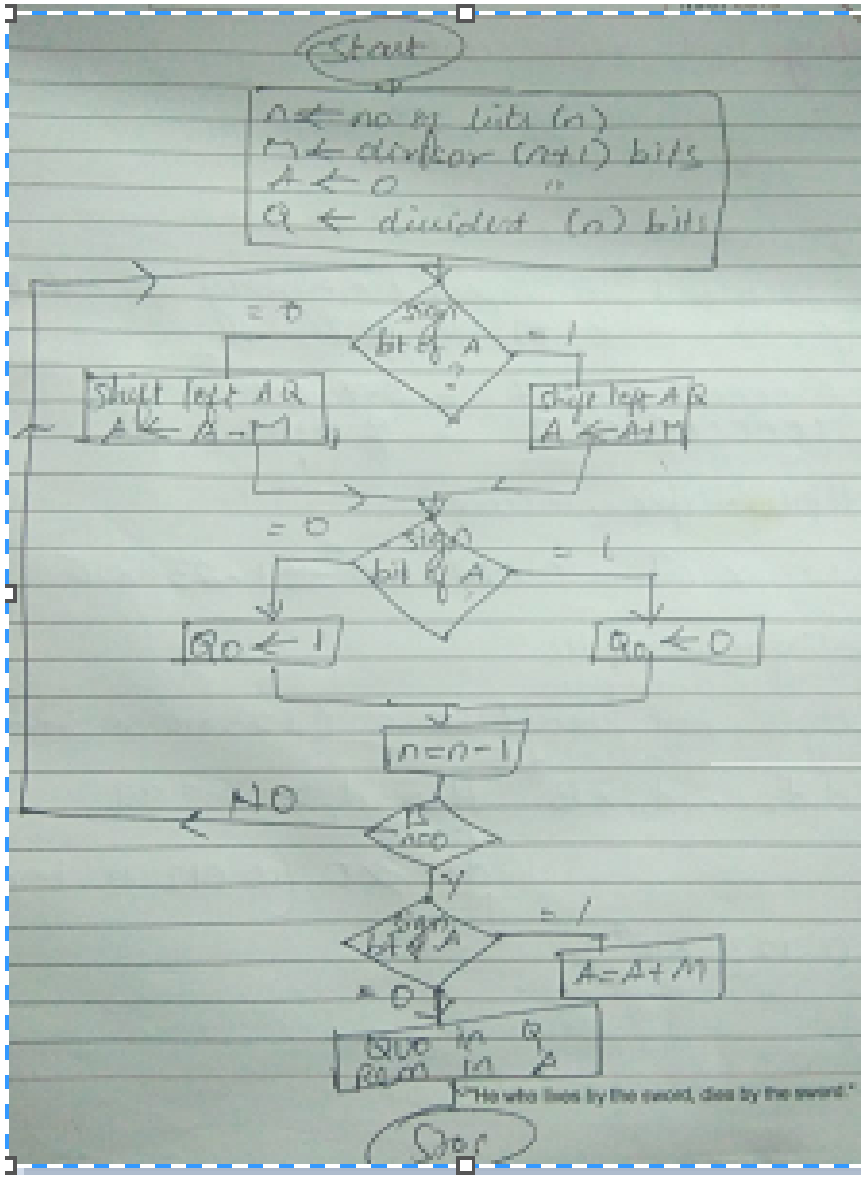
Non-restoring algorithm is:
Set A to 0.
Repeat n times:
  If the sign of A is positive:
    Shift A and Q left and subtract M. Set q0 to 1.
  Else if the sign of A is negative:
    Shift A and Q left and add M. Set q0 to 0.
If the sign of A is 1, add A to M.

Perform non restoring division on 11/3.

M=00011 Q=1011 -M=11101

N	A	Q	Action
4	00000	1011	Initial
	00001	011?	SL AQ
	11110	011?	A←A-M
3	11110	0110	Q0=0
	11100	110?	SL AQ
	11111	110?	A←A+M
2	11111	1100	Q0=0
	11111	100?	SL AQ
	00010	100?	A←A+M
1	00010	1001	Q0=1
	00101	001?	SL AQ
	00010	001?	A←A-M
0	00010	0011	Q0=1

Sign bit of A=0. Hence
Quotient (Q)= 11
Remainder (A)=10

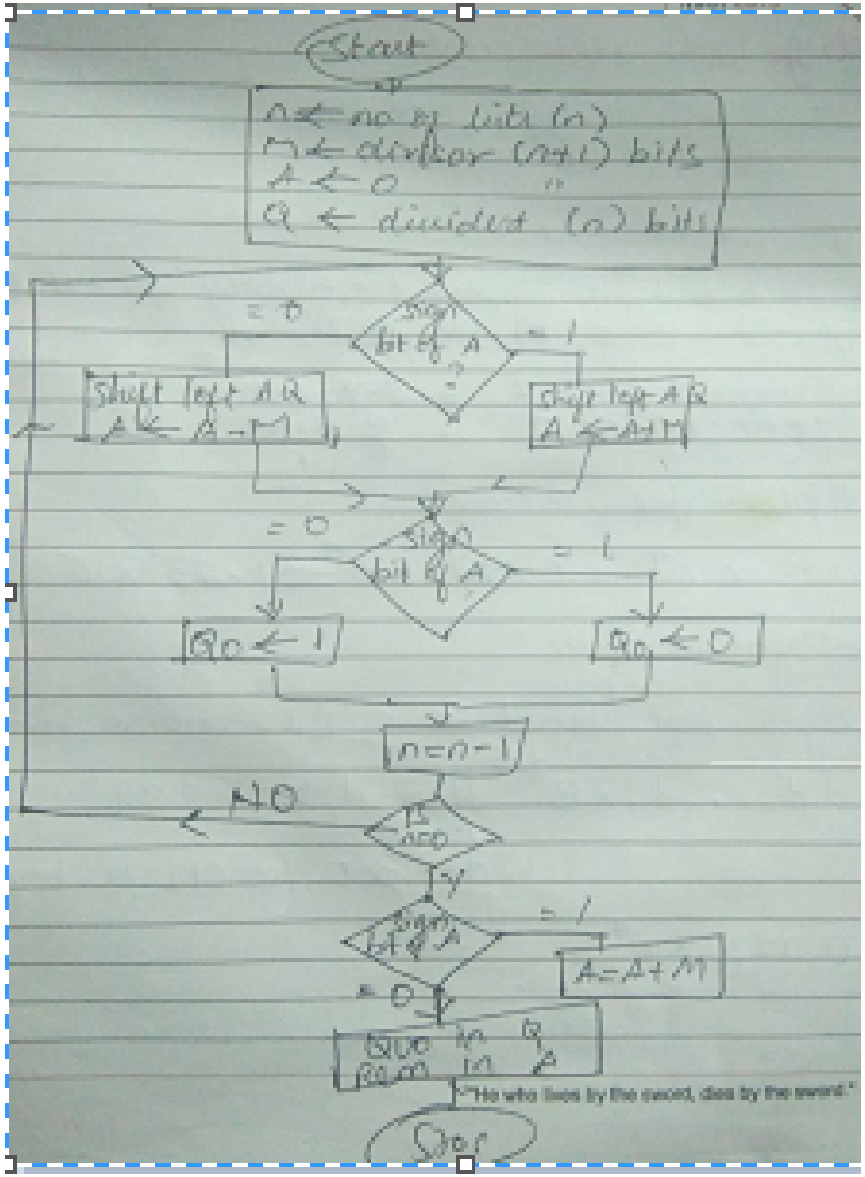


Find 30/9 by non restoring division.

M=001001 Q=11110 -M=110111

N	A	Q	Action
5	000000	11110	Initial
	000001	1110?	LS AQ
	111000	1110?	A←A-M
4	111000	11100	Q0=0
	110001	1100?	LS AQ
	111010	1100?	A←A+M
3	111010	11000	Q0=0
	110101	1000?	LS AQ
	111110	1000?	A←A+M
2	111110	10000	Q0=0
	111101	0000?	LS AQ
	000110	0000?	A←A+M
1	000110	00001	Q0=1
	001100	0001?	LS AQ
	000011	0001?	A←A-M
0	000011	00011	Q0=0

Sign bit of A=0. Hence
Quotient (Q)= 11
Remainder (A)=11



Method	Advantages	Limitation
Restoring division	Full width comparison is required to find the new quotient digit.	Slower; time consuming because of restoration.
Non restoring division	Test subtraction is not required. The sign bit determines whether addition/ subtraction is required.	An extra bit must be maintained in the partial remainder to keep track of the sign.

Floating Point Numbers and Operations

Floating Point Number Representation (FPR)

$$\pm \text{Significant} \times \text{Base}^{\pm \text{Exponent}}$$

Example: $+10.55 \times 10^{55}$

Need for FPR

- The number $0.00000000007 = 0.7 \times 10^{-10}$

The number $70000000000 = 7 \times 10^{10}$

Through FPR both smaller and larger numbers can be represented in consise way (less no. of bits).

- Also multiple representations are possible for the same number

Eg : $0.123 \times 10^4 = 0.0123 \times 10^5 = 123 \times 10^3$

NORMALIZATION OF NUMBERS

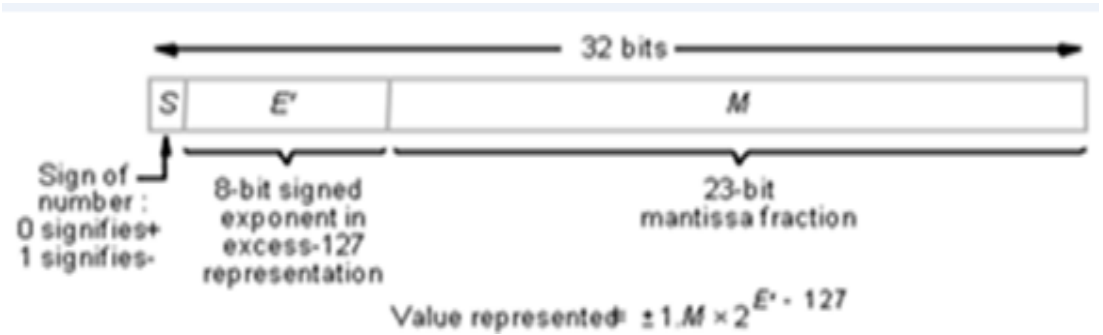
1. A number is normalized when it is written in scientific notation with one non-zero decimal digit before the decimal point.
2. A normalized number provides more accuracy than corresponding **de-normalized** number.

Rules for normalization:

1. Make the integer part of the number to be zero. (1.23×10^6 is invalid).
2. Express as $0.d_1d_2d_3 \ldots d_n \times B \text{ ^ } \pm E$, then $d_1 >= 1$ and $d_i >= 0$ for all other values. (Eg: 0.123×10^4)

TECHNIQUES: IEEE REPRESENTATION (IEEE 754)

1. Single precision (32 bits)



1 Sign bit of mantissa. (0=Positive, 1-negative)

8 bits -Biased exponent 127.

127 is added with exponent to represent numbers between the range -127 to +128 .

-127+127=0 ; 128+127=255

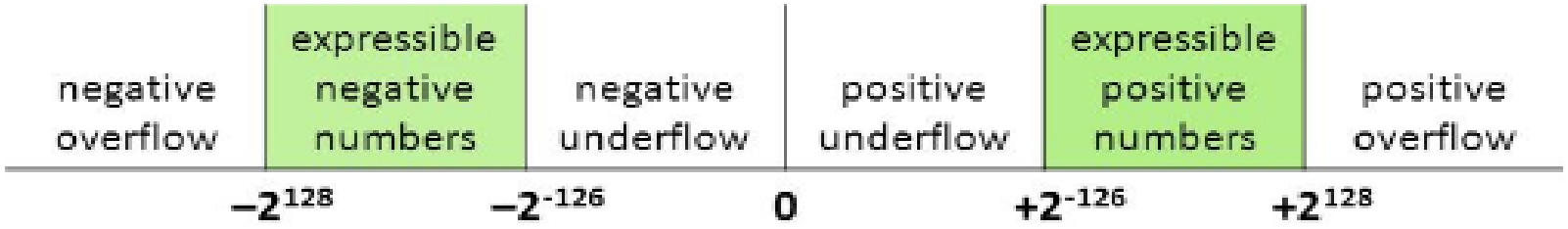
All numbers will be now shifted between 0 and 255. This shifting is done to fit the numbers in the range.

23 bits-truncated significant bit

The bits after the decimal is always 1 (normalization rules). So no need to represent explicitly. 23+1=24 bits are accommodated within 23 bits. Hence termed as truncated significant bits.

2. Double Precision Format (64 bits)

- **1 Sign bit of mantissa. (0=Positive, 1-negative)**
- **11 bits -Biased exponent 1023.**
- **52 bits-truncated significant bit**



Overflow and underflow conditions

1. Express 85.125 in single and double precision format.

Binary value of 85: 1010101

Binary value of 0.125: 0.001

85.125= 1010101.001

$= 1.010101001 \times 2^6$

Sign bit=0

Single Precision Format:

Biased exponent= 127 +6=133=10000101

Normalized mantissa=010101001

Sign || exponent || mantissa

0 10000101 010101001

Double Precision Format:

Biased exponent= 1023 +6=133=1029=10000000101

Normalized mantissa=010101001

Sign || exponent || mantissa

0 10000000101 010101001

Floating Point Numbers and Operations

Floating Point Arithmetic Addition/Subtraction

Steps to add two floating point numbers:

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

Example: Add 1.1100×2^4 and 1.1000×2^2

1. Alignment: 1.1000×2^2 has to be aligned to 0.0110×2^4 1.1100
2. Addition: Add the numbers to get 10.0010×2^4 0.0110
3. Normalization: Find normalized result 10.0010

0.1000×2^6 (assuming 4-bits are allowed after decimal point)

Floating Point Numbers and Operations

Steps to subtract two floating point numbers:

1. Compare the magnitudes of the exponents and make suitable alignment to the number with the smaller magnitude of exponent.
2. Takes twos complement of negative number and then add.
3. Perform normalization by shifting the resulting mantissa and adjusting the resulting exponent

Add 0.5 + (-0.4375).

$0.5 = 1.000 \times 2^{-1}$

$-0.4375 = -0.0111 = -1.11 \times 2^{-2}$

Rewrite smaller no.

$$1.000 \times 2^{-1}$$

$$-0.1110 \times 2^{-1}$$

Twos complement of -0.1110×2^{-1} is 1.001×2^{-1}

Add

$$1.000 \times 2^{-1}$$

$$-1.001 \times 2^{-1}$$

SUM: 0.001×2^{-1}

Multiply 1.110×10^{10} by 9.200×10^{-5}.

Steps:

1. Add the exponents
2. Multiply the significant digits
3. Normalize and round off the product

1. Add the exponents: 10-5=5
2. Multiply: 1.110 x 9.200=10.210000
3. Normalize:

10.210000×10^{5}

After normalization: 1.021×10^{6}

TRUNCATION AND GUARD BITS

1. Guard bits- bits that improve accuracy
2. Truncation- removing guard bits

Methods of Truncation:

1. **Chopping:** remove guard bits
2. **Von Neumann rounding:** If the guard bits are 0, simply drop. If guard bits are 1, then LSB of the retained bit is 1.

Eg: round off to 2 bits: 1.110

a) 1.110= 1.11

b) 1.101=1.11

c) 1.111=1.11

3. **Rounding:** If the MSB of guard bit is 1, then add another 1 irrespective of 0's or 1's in truncated bits

Eg: Truncate to 2 bits

1.1000001=1.11