MULTIPLICATION OF POSITIVE NUMBERS:

* The usual algorithm for multiplying integers is shown in fig

```
          1  1  0  1      (13) Multiplicand M
        x 1  0  1  1      (11) Multiplier Q
    ---------------
          1  1  0  1      (PP0)
       1  1  0  1         (PP1)
    0  0  0  0            (PP2)
 1  1  0  1               (PP3)
 ---------------
 1  0  0  0  1  1  1  1   (143) Product P
```

* This method can be applied only for unsigned numbers and to positive signed numbers.

* The product of 2·n-digit numbers can be accomodated in 2n digits, so the product of two 4-bit numbers in above eg fits into 8 bits.

* In the binary system, multiplication of the multiplicand by one bit of the multiplier is easy.

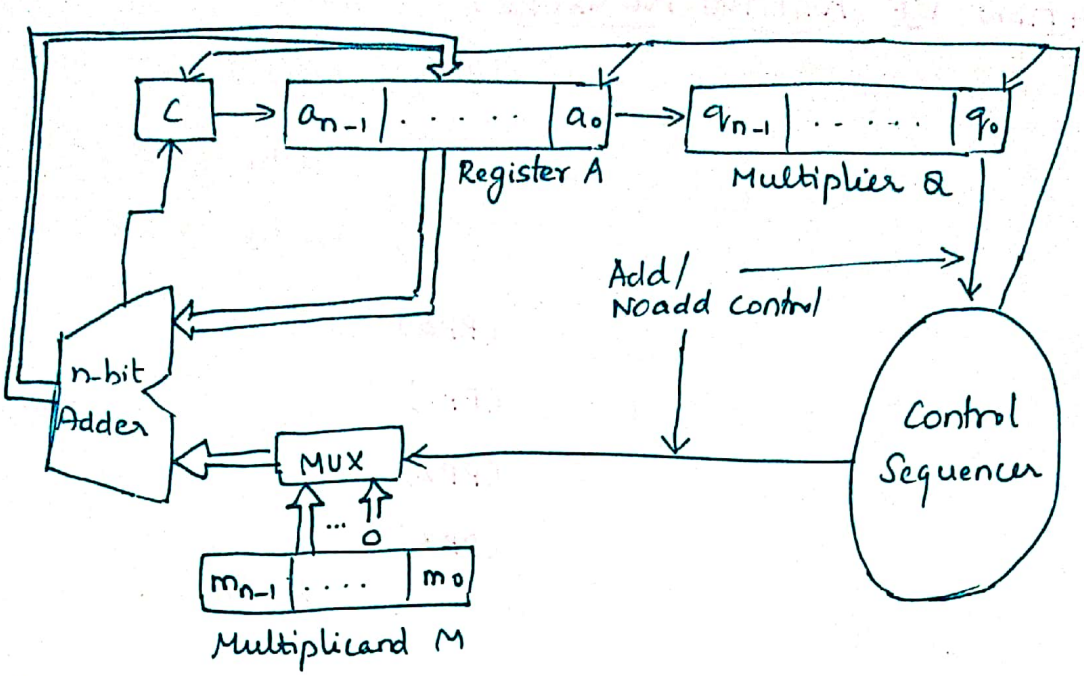
* If the multiplier bit is 1, the multiplicand is entered in the appropriate position to be added to partial product.

* If the multiplier bit is 0, then 0's are entered.

Block Diagram:

* The simplest way to perform multiplication is to use the adder circuitry in the ALU for a number of sequential steps.

* The figure shows the block diagram (ie) hardware arrangement for sequential multiplication. It performs multiplication by using single n-bit adder n times to implement the spatial addition.



* Registers A and Q Combined hold PP_i while multiplier bit q_i generates the signal Add/Noadd. This signal controls the addition of the Multiplicand M to PP_i to generate $PP_{(i+1)}$.

* The product is computed in n cycles. The partial product grows in length by one bit per cycle. The carry out from the adder is stored in flip flop C.

* At the start, the multiplier is loaded into register Q, the multiplicand in register M and C and A are cleared to 0.

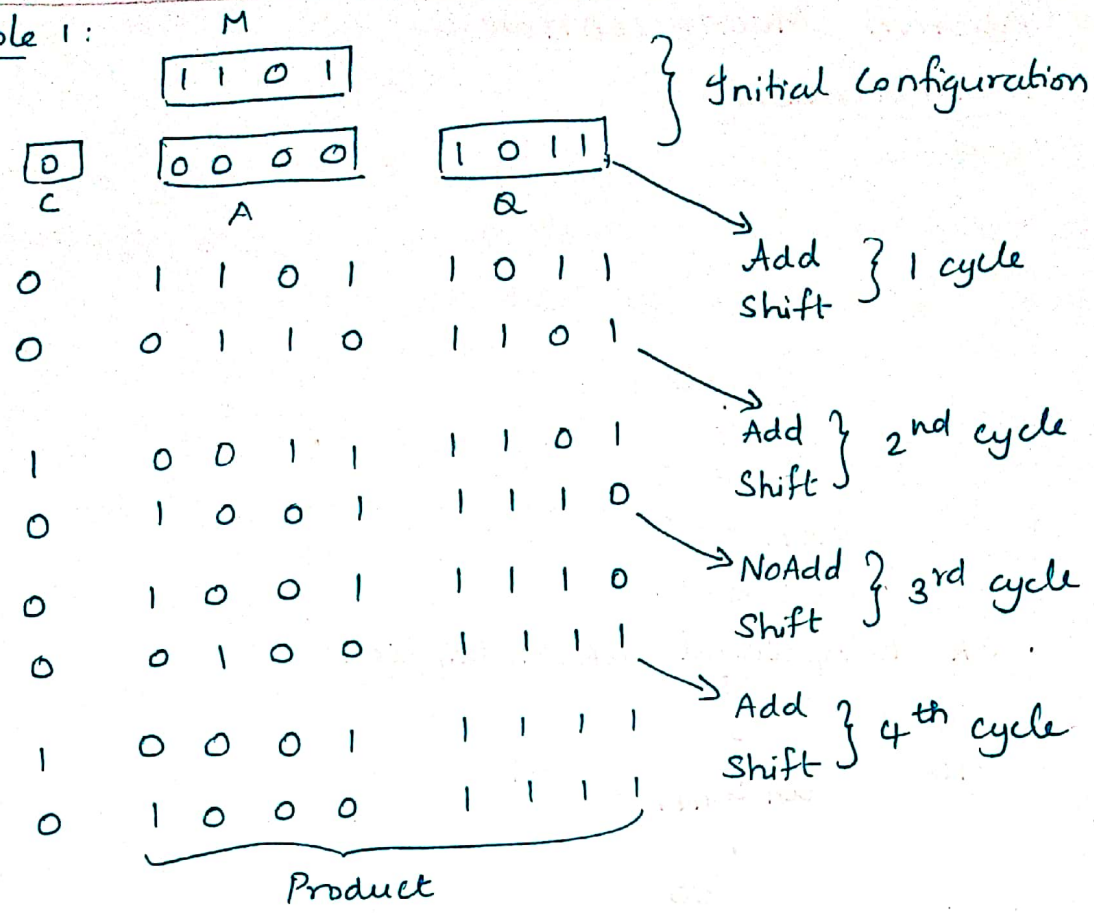
* At the end of each cycle, C, A and Q are shifted right one bit position to allow for growth of partial product as the multiplier is shifted out of Q register.

* Because of this shifting, the multiplier bit q_i appears at the LSB position of Q to generate Add/NoAdd signal.

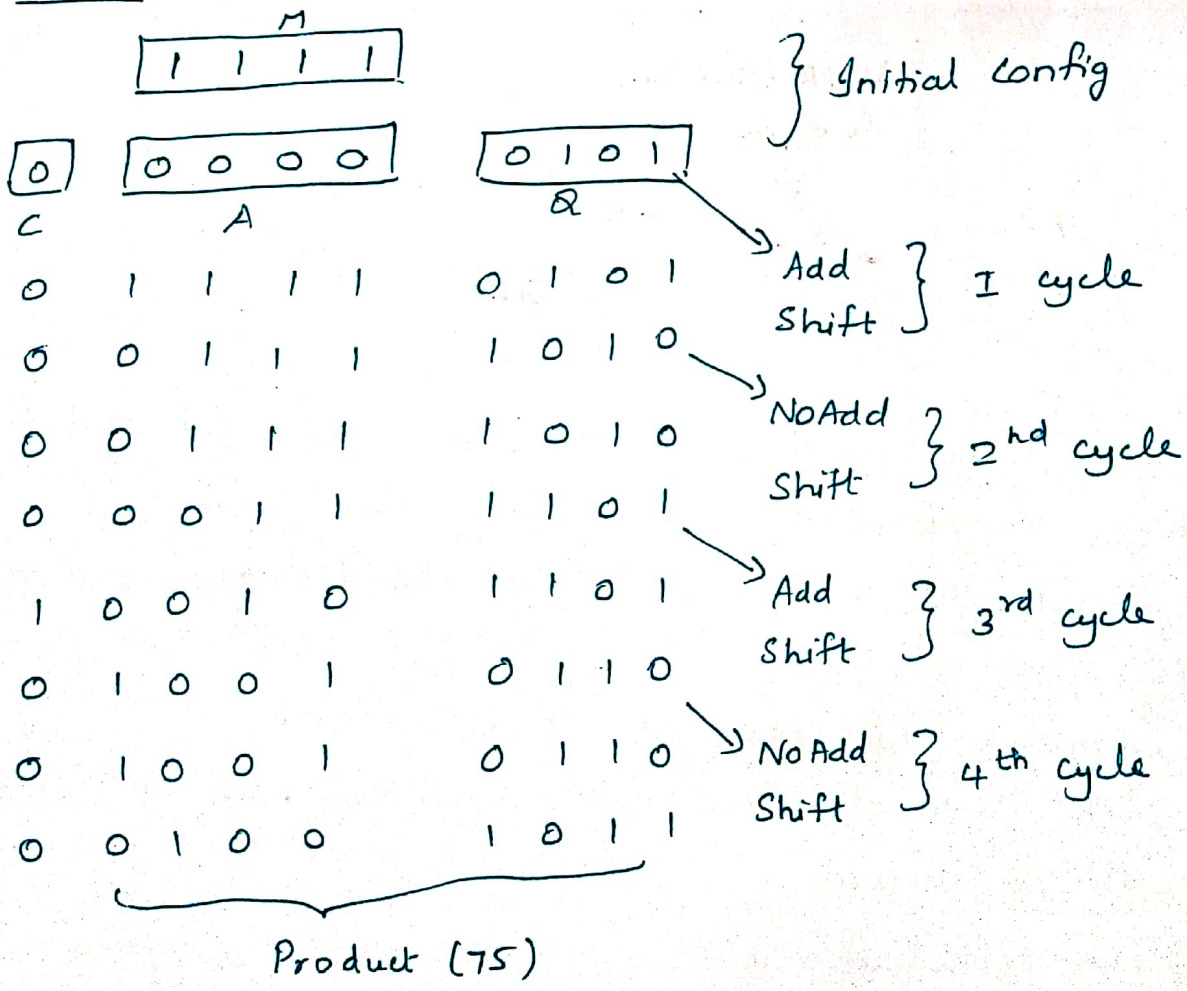
* The carry out from the adder is leftmost bit of $PP_{(i+1)}$ and it must be held in C flip flop to be shifted right with the contents of A and Q.

* After n cycles, high-order half of the product is held in register A, and low order half is in register Q.

Example 1:



Example 2: 15 x 5 . 15 ⇒ 1111 5 ⇒ 0101

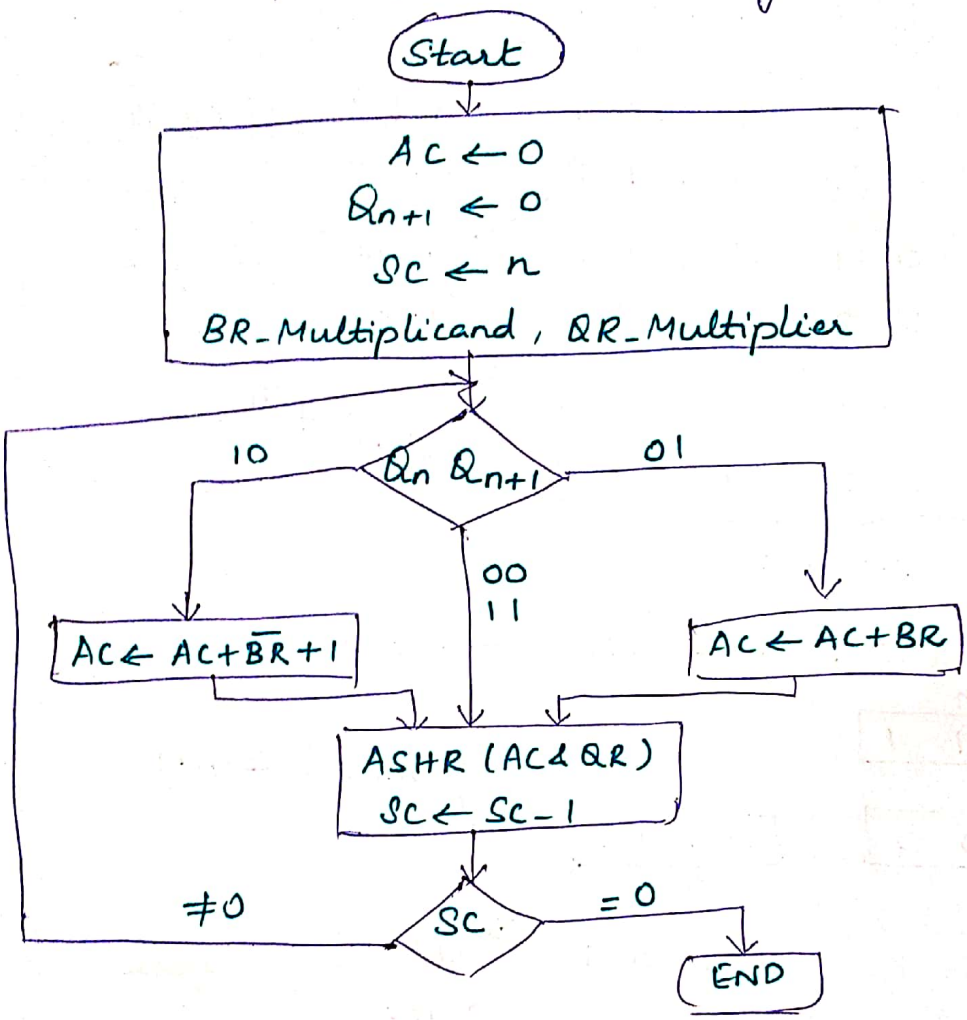


SIGNED OPERAND MULTIPLICATION:

BOOTH'S MULTIPLICATION:

* The booth algorithm is a multiplication algorithm that enables to multiply two signed binary in 2's complement.

* Pictorial representation of Booth Alg:



Steps:

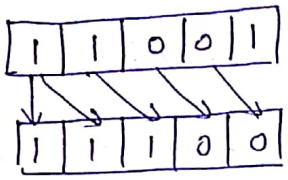
1. Set the Multiplicand and Multiplier bits as M and Q.
2. Initially, set AC and Q_{n+1} registers value to 0.
3. SC represents the no. of multiplier bits (Q) and is a Sequence Counter that is decremented after each cycle.
4. Qn represents the last bit of Q.
5. On each cycle of the booth algorithm, Qn and Q_{n+1} bits will be checked.
   (i) When 2 bits Qn and Q_{n+1} are 00 or 11, perform arithmetic shift right (ashr) Operation to partial product AC

(ii) If bits in Q_n and Q_{n+1} is 01, the multiplicand bits are to be added to AC. Then, perform Arithmetic shift right operation to AC and QR bits.

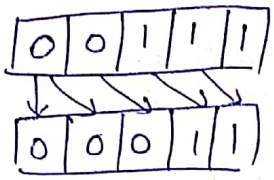
(iii) If bits in Q_n and Q_{n+1} is 10, the multiplicand bits are to be subtracted from AC. Then, perform ashr operation to AC and QR bits.

Ashr: → shift bits right and MSB is to be filled with value of previous MSB.

eg 1.



eg 2



6. The operation continues n times where n is no. of bits in multiplier.

7. Results of multiplication will be in AC and QR registers.

Example 1: Multiply 7 and 3 by using Booth's multiplication Alg.

Q_n	Q_{n+1}	M = (0111)	AC	QR	Q_{n+1}	SC
1	0	Initial	0 0 0 0	0 0 1 1	0	4
		Subtract M from AC (M'+1) = 1001	1 0 0 1	0 0 1 1	0	4
		Perform ashr	1 1 0 0	1 0 0 1	1	3
1	1	Perform ashr	1 1 1 0	0 1 0 0	1	2
0	1	Add M to AC	0 0 0 1	0 1 0 0	1	2
		Perform ashr	0 0 1 0	1 0 1 0	0	1
0	0	Perform ashr	0 0 0 1	0 1 0 1	0	0
			Product			

7 × 3 = 21. Binary representation of 21 is 10101.

∴ 21 is represented as 00010101.

Example 2: Multiply 23 and -9 by Booth's Alg.

$(010111)_2$

$+9 = (01001)$

$-9 = (110111)_2$

Q_n	Q_{n+1}	$M = 010111$ $M'+1$ $= 101001$	AC	Q	Q_{n+1}	SC
		Initially.	000000	110111	0	6
			101001			
1	0	Subtract M from AC	101001	110111	0	6
		Ashr	110100	111011	1	5
1	1	Ashr	111010	011101	1	4
1	1	Ashr	111101	001110	1	3
			010111			
0	1	Add M to AC	010100	001110	1	3
		Ashr	001010	000111	0	2
			101001			
1	0	Subtract M from AC	110011	000111	0	2
		Ashr	111001	100011	1	1
1	1	Ashr	111100	110001	1	0

$Q_{n+1} = 1$, it means output is negative.

$23 \times -9 = -207$

$+207 \Rightarrow (0000\,1100\,1111)$

$-207 \Rightarrow (1111\,0011\,0001)$

$(+207) \Rightarrow (000011001111)_2$

$(-207) \Rightarrow (1111\,0011.0001)_2$

2|207
2|103 - 1
2|51 - 1
2|25 - 1
2|12 - 1
2|6 - 0
2|3 - 0
1 - 1

2|207
2|103 - 1
2|51 - 1
2|25 - 1
2|12 - 1
2|6 - 0
2|3 - 0
1 - 1

BOOTH ALGORITHM - BIT PAIR RECODING
FAST MULTIPLICATION:

* The Booth algorithm generates a 2n-bit product and treats both positive and negative 2's complement n-bit operands uniformly.

* Consider a multiplication operation, in which multiplier is positive and has single block of 1's, for eg, 0011110. To derive the product, we could add 4 appropriately shifted versions of the multiplicand.

* However, the number of operations can be reduced with bit pair recoding of the multipliers.

* In the booth scheme, -1 times the shifted multiplicand is selected when moving from 1 to 0, and +1 times shifted multiplicand is selected when moving from 0 to 1, as the multiplier is scanned from left to right.

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit i-1	
0	0	0 x M
0	1	+1 x M
1	0	-1 x M
1	1	0 x M

Booth multiplier recoding table

* An example for recoding:

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

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

Example: 45 × 30. The normal multiplication and Booth multiplication is illustrated.

```
                    0 1 0 1 1 0 1
                    0 0 1 1 1 1 0
                  ---------------
                  0 0 0 0 0 0 0
                1 0 1 0 1 1 0 1 -
              1 0 1 0 1 1 0 1 - -
            0 1 0 1 1 0 1 - - -
          0 1 0 1 1 0 1 - - - -
        0 0 0 0 0 0 0 - - - - -
      0 0 0 0 0 0 0 - - - - - -
    ---------------------------
  0 0 0 1 0 1 0 1 0 0 0 1 1 0
```

Recoding:

```
0  0  1  1  1  1  0   (Multiplier)
0 +1  0  0  0 -1  0   (Recoded multiplier)
```

```
                    0 1 0 1 1 0 1
                    0 +1 0 0 0 -1 0
                  ---------------
                  0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 1 0 0 1 1        ← 2's complement of multiplicand
1 1 1 1 1 1 1 0 1 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 1 0 1
0 0 0 0 0 0 0 0
-----------------------
0 0 0 1 0 1 0 1 0 0 0 1 1 0
```

* The Booth algorithm can also be used directly for negative multipliers.

eg. $+13 \times (-6)$

$2|13$ — $2|6$ — 1
$2|6$ — $2|3$ — 0
$2|3$ — 1 — 1
1 — 1

$+13 = (01101)_2$

$+6 = (00110)_2$

$(-6) = (11010)_2$

$$\begin{array}{rr} & 01101 \ (+13) \\ \times & 11010 \ (-6) \end{array}$$

Recoding of multiplier.

1 1 0 1 0

0 -1 +1 -1 0

$$\begin{array}{rr} & 01101 \\ & 0\ -1\ +1\ -1\ 0 \\ \hline 0\ 0\ 0\ 0\ 0 & 0\ 0\ 0\ 0\ 0 \\ 1\ 1\ 1\ 1 & 1\ 0\ 0\ 1\ 1 \\ 0\ 0\ 0\ 0 & 1\ 1\ 0\ 1 \\ 1\ 1\ 1\ 0 & 0\ 1\ 1 \\ 0\ 0\ 0\ 0 & 0\ 0 \\ \hline 1\ 1\ 1\ 0 & 1\ 1\ 0\ 0\ 1\ 0 \end{array} \quad (-78)$$

$2|78$ — $2|39$ — 0
$2|19$ — 1
$2|9$ — 1
$2|4$ — 1
$2|2$ — 0
1 — 0

$+78 = 0001001110$

$-78 = 1110110010$

Bit-pair Recoding:

* A technique called bit-pair recoding halves the maximum number of summands. It is directly derived from Booth algorithm.

Multiplier bit-pair		Multiplier bit on right	Multiplicand Selected as position i
i+1	i	i-1	
0	0	0	0 × M
0	0	1	+1 × M
0	1	0	+1 × M
0	1	1	+2 × M
1	0	0	−2 × M
1	0	1	−1 × M
1	1	0	−1 × M
1	1	1	0 × M

eg.

1 1 1 0 1 0 [0] → Implied bit

0 0 −1 +1 −1 0

↓ ↓ ↓

0 −1 −2

(+13) × (−6) · (0 1 1 0 1) × (1 1 0 1 0)

⇓

0 1 1 0 1

0 −1 +1 −1 0

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	1	1	-
0	0	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

⇓

0 1 1 0 1

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

01101 x −2.

↳ Multiply with +2 then take 2's comp

01101 × 10 (+2)

00000

01101

011010

⇓ 2's complement.

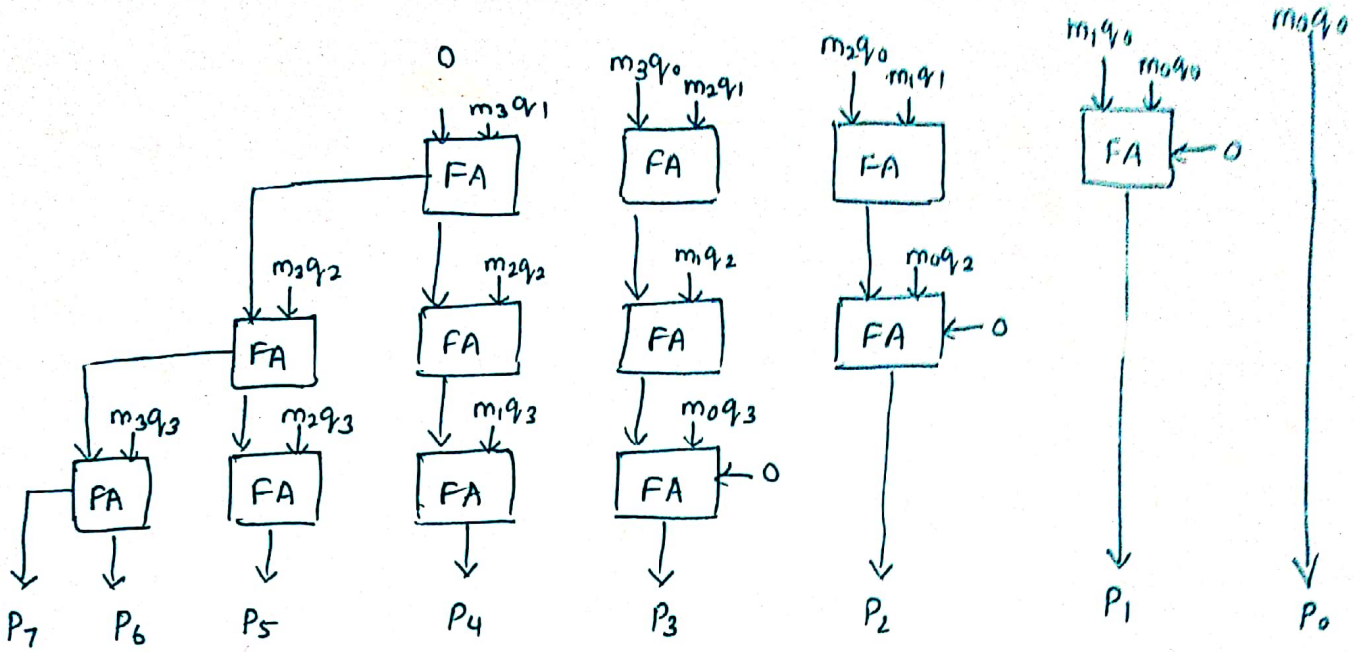
100110

CARRY SAVE ADDITION OF SUMMANDS:

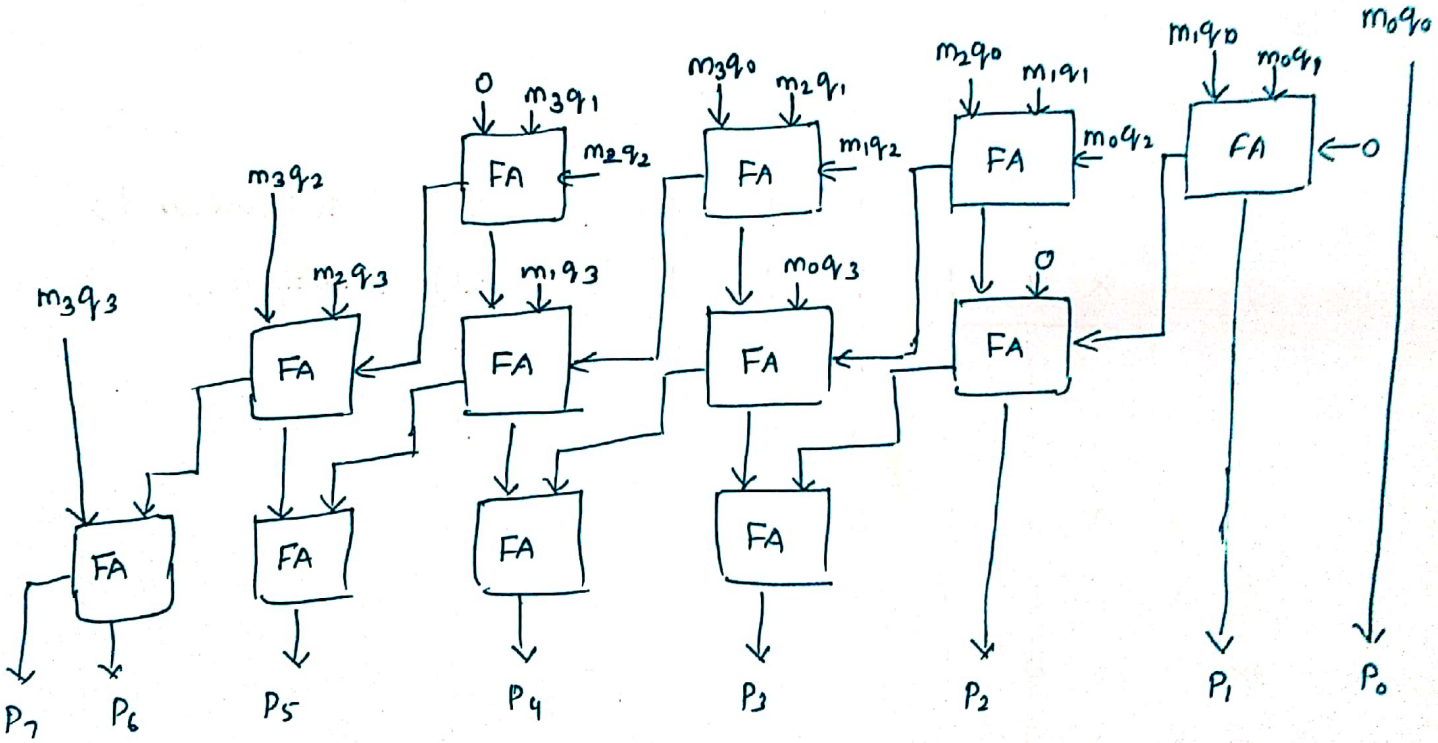
* Multiplication requires the addition of several summands. A technique called carry Save Addition (CSA) speeds up the addition process.

* Consider the array for 4x4 multiplication. The first figure shows the structure of general array.

* Instead of letting the carries ripple along the rows, they can be saved and introduced into the next row, at the correct weighted positions as shown in the second figure.



Ripple Carry Array



Carry Save Array

* Delay through the carry save array is somewhat less than delay through ripple carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.

* A more significant reduction in delay can be achieved as follows. The summands can be grouped in threes and carry-save addition can be performed on each of these groups in parallel to generate set of S and C vectors in one Full Adder delay.

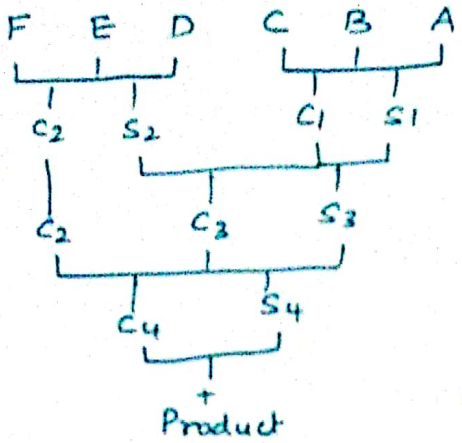
* Then, gene group all of the S and C vectors into threes, and carry save addition are performed and further set of S and C vectors can be generated.

* The process has to be repeated until only two vectors are remaining. They can then be added in ripple-carry or carry look ahead adder to produce the desired product.

* Consider an example of adding six shifted versions of the multiplicand to multiply two 6-bit unsigned numbers where all six bits of the multiplier are equal to 1.

```
                1 0 1 1 0 1   (45) (Multiplicand)
              x 1 1 1 1 1 1   (63) (Multiplier)
            -----------------
                1 0 1 1 0 1     A
              1 0 1 1 0 1       B
            1 0 1 1 0 1         C
          1 0 1 1 0 1           D
        1 0 1 1 0 1             E
      1 0 1 1 0 1               F
    -----------------------------
    1 0 1 1 0 0 0 1 0 0 1 1   (2835) (Product)
```

* The six summands A, B, ... F are added by carry save addition. Three levels of carry save addition need to be performed.



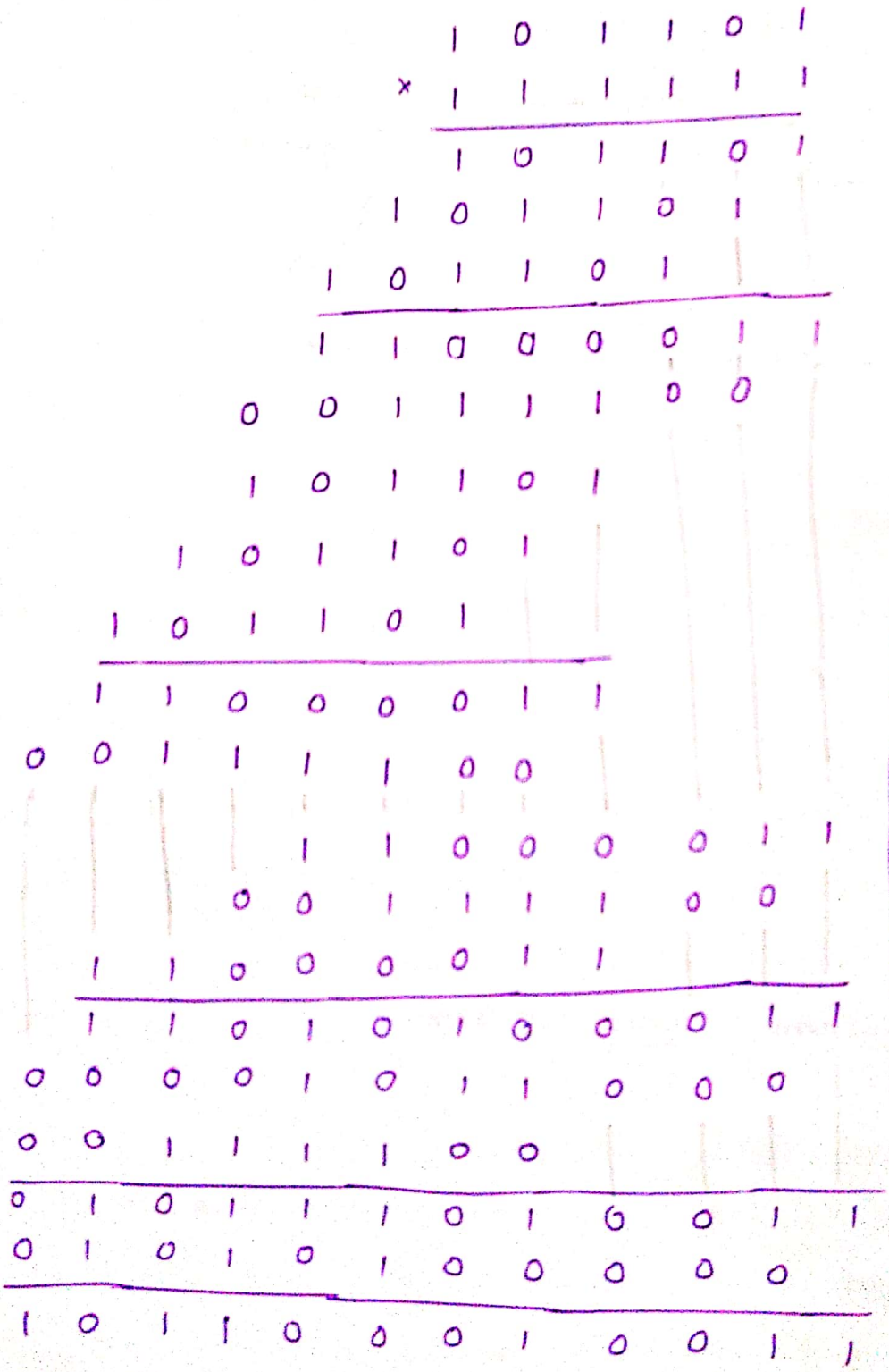
Level 1 CSA

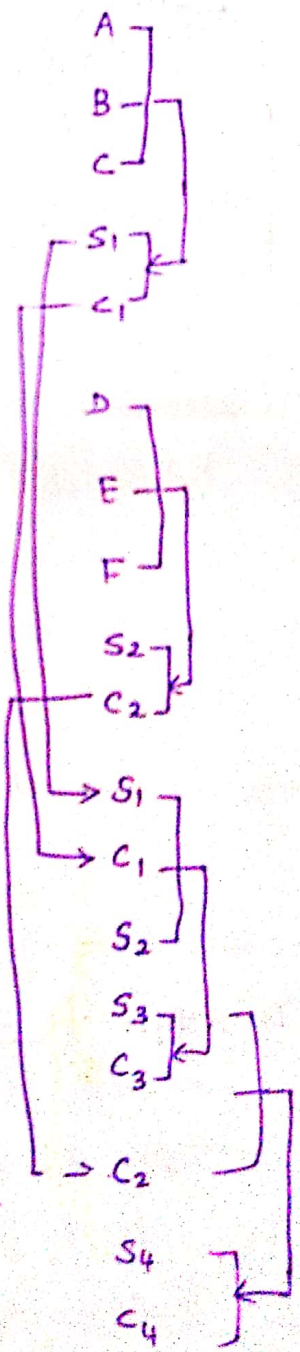
Level 2 CSA

Level 3 CSA

Final Addition

⇒ This represents the Schematic representation of carry save addition operations.

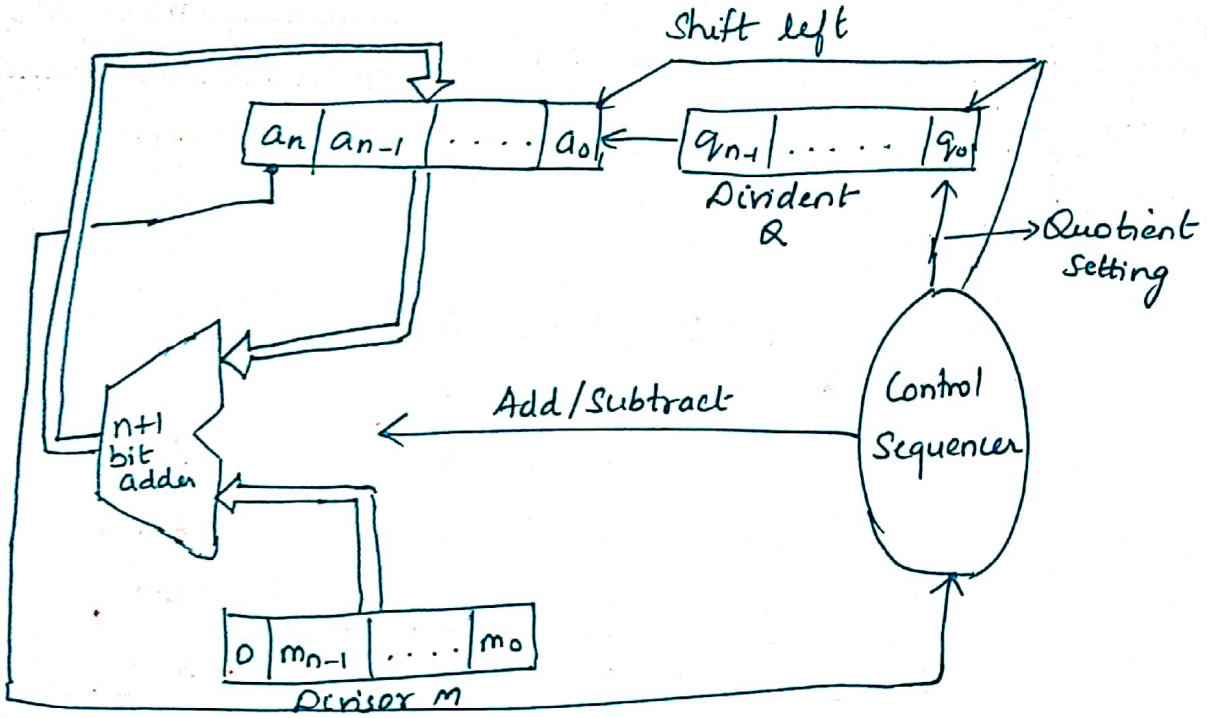




INTEGER DIVISION:

* Two methods - Restoring Division and Non-restoring division.

Restoring Division



* The figure shows the logic circuit arrangement that implements restoring division.

* An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at start. Register A is set to 0.

* The extra bit position at the left end of both A and M accomodates the sign bit during subtractions.

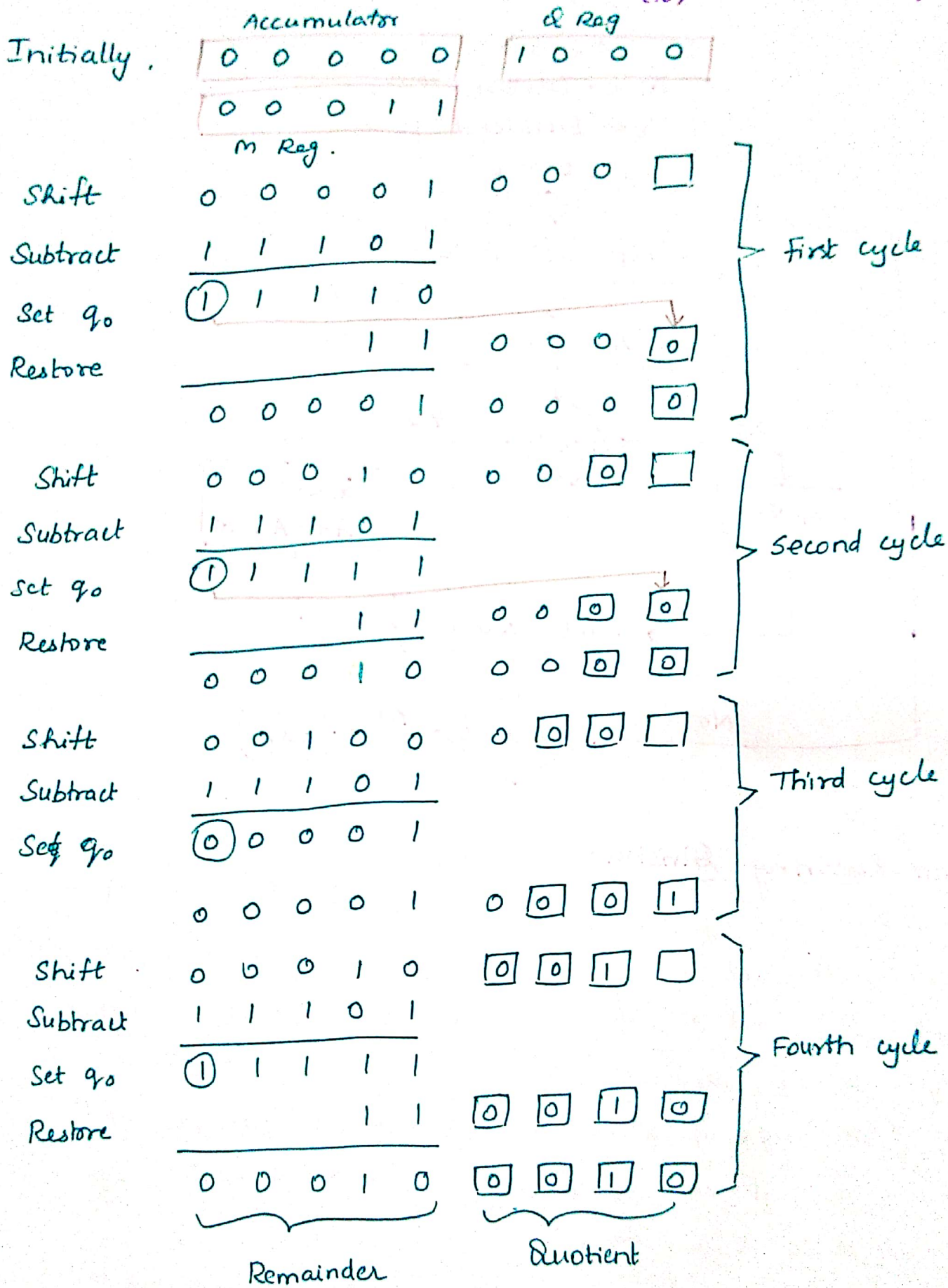
* At the end, the n-bit quotient is in register Q and the remainder is in register A.

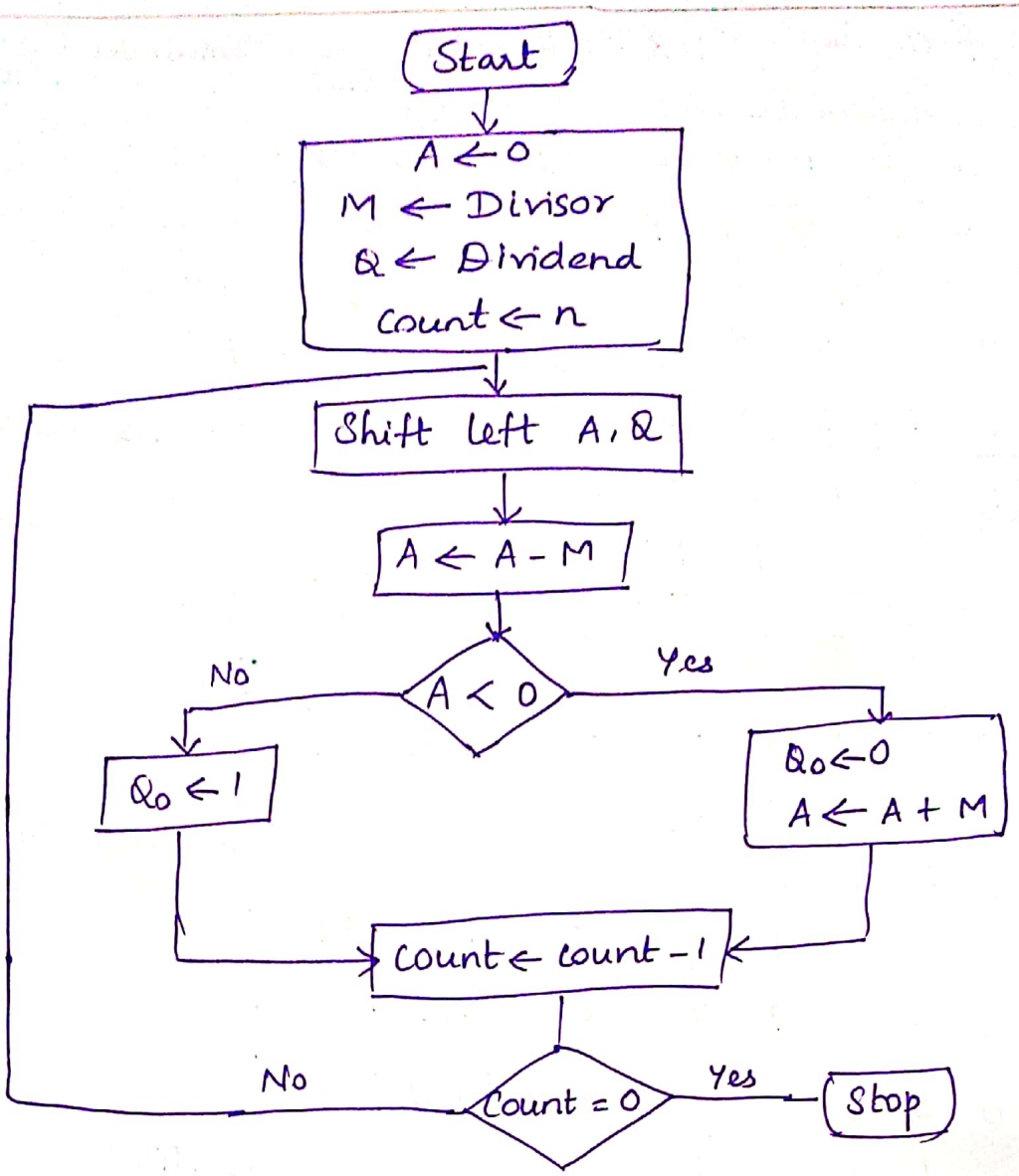
Algorithm to perform restoring division:

Do the following n times:

1. Shift A and Q left one binary position.
2. Subtract M from A, place the answer back in A.
3. If the sign of A is 1, set q_0 to 0 and add M back to A (ie. restore A); otherwise, set q_0 to 1.

Eg 1: $1000 \div 11$ (ie) $8 \div 3$ Quotient = 2 (10) Remainder = 2 (10)





Non-Restoring Division:

* The restoring division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction. Subtraction is said to be unsuccessful if the result is negative.

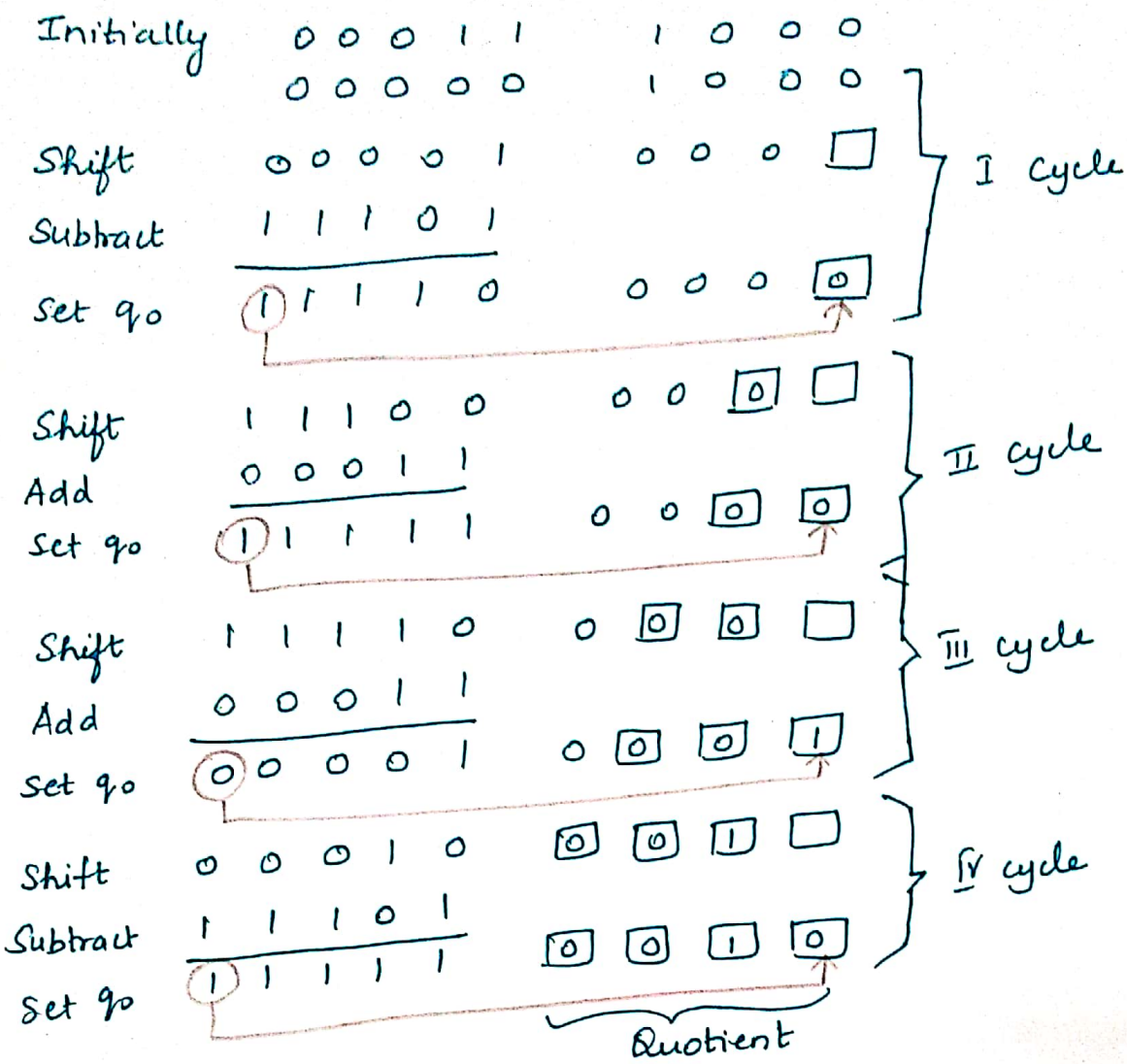
Step 1: Do the following n times

(i) If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.

(ii) Now, if sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

Step 2: If the sign of A is 1, add M to A.

Eg: 8 ÷ 3



Step 2 : If sign of A is 1, add M to A