MODULE - II

DATA REPRESENTATION AND COMPUTER ARITHMETIC

Data representation:

Numbers - represented by a string of bits, called a binary number.

Text - can also be represented by a string of bits, called character codes.

Number representation:

Decimal to binary conversion:

- Divide successively by 2 and write the remainders in reverse.

E.g. $(25)_{10}$.

$$
\begin{array}{r|l}
2 & 25 \\
\hline
2 & 12 - 1 \\
\hline
2 & 6 - 0 \\
\hline
2 & 3 - 0 \\
\hline
 & 1 - 1
\end{array}
$$

$(25)_{10} = (11001)_2$.

- Another method:

...	2^4	2^3	2^2	2^1	2^0
	16	8	4	2	1
	1	1	0	0	1

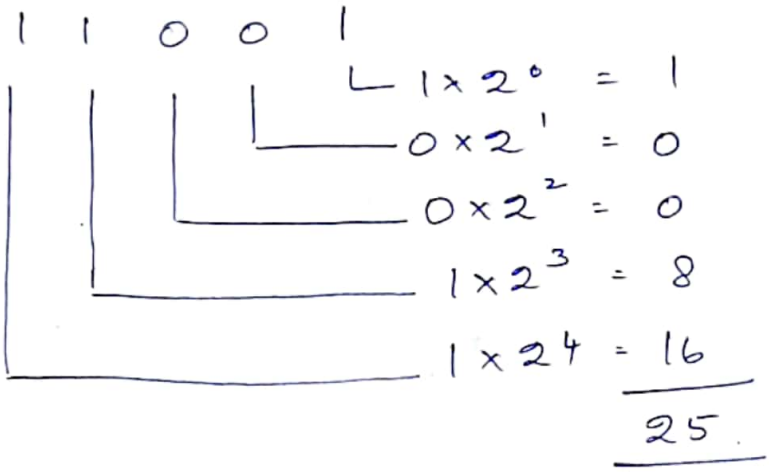
$\therefore (25)_{10} = (11001)_2$.

Put a one under the numbers than can be summed to get the decimal number.

Binary to decimal conversion:

- Multiply by powers of 2. and add.

E.g.: $(11001)_2$



Numbers can be signed or unsigned.

Representation of signed numbers:

3 methods:

- Sign-and-magnitude
- 1's complement
- 2's complement

In all 3 methods, the leftmost bit is 0 for positive numbers and 1 for negative numbers.

Positive numbers have the same representation in all systems, but negative numbers have different representations.

E.g: +5 → 0101

-5 → Sign-magnitude: 1101
1's complement: 1010
2's complement: 1011

1's complement → complement / reverse each bit of the positive number.

2's complement → add 1 to the 1's complement

2's complement method yields the most efficient way to carry out addition and subtraction and is the representation that is most often used in computers.

Addition and subtraction of signed numbers. (using 2's complement).

Rules:

1. To add 2 numbers, add their n-bit representations, ignoring the carry-out sign from the MSB. The sum will be the algebraically correct value as long as the answer is in the range -2^{n-1} through $+2^{n-1}-1$.

2. To subtract 2 numbers, X and Y, that is, to perform X-Y, form the 2's complement of Y and add to X as in rule 1.

When the result of addition / subtraction, does not fall within -2^{n-1} and $+2^{n-1}-1$, we say that an arithmetic overflow has occurred

A simple way to detect an overflow — when the operands have the same sign and the result has the opposite sign.

Examples:

1. $(+2)_{10} + (+3)_{10}$

```
  0 0 1 0
+ 0 0 1 1
---------
  0 1 0 1   → (+5)10
---------
```

2. $(+4)_{10} + (-6)_{10}$.

```
  0 1 0 0
+ 1 0 1 0
---------
  1 1 1 0   → (-2)10.
---------
```

2's comp of 6:

```
  0 1 1 0
  1 0 0 1   - 1's comp.
+       1
---------
  1 0 1 0   - 2's comp.
```

3. $(-5)_{10} + (-2)_{10}$

```
  1 0 1 1
+ 1 1 1 0
---------
  1 0 0 1   → (-7)10.
---------
```

2's comp. of 1001:

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

4. $(+7)_{10} + (-3)_{10}$.

```
  0 1 1 1
+ 1 1 0 1
---------
  0 1 0 0   → (4)10.
---------
```

5. $(-3)_{10} - (-7)_{10} \Rightarrow (-3)_{10} + (7)_{10}$.

```
  1 1 0 1
+ 0 1 1 1
---------
  0 1 0 0   → (4)10.
---------
```

(carry generated is discarded)

6. $(+2)_{10} - (+4)_{10}$ $\Rightarrow (+2)_{10} + (-4)_{10}$.

$$\begin{array}{r} 0010 \\ +\,1100 \\ \hline 1110 \end{array} \rightarrow (-2)_{10}.$$

7. $(+6)_{10} - (+3)_{10} \Rightarrow (+6)_{10} + (-3)_{10}$.

$$\begin{array}{r} 0110 \\ +\,1101 \\ \hline 0011 \end{array} \rightarrow (+3)_{10}.$$

8. $(-7)_{10} - (-5)_{10} \Rightarrow (-7)_{10} + (5)_{10}$.

$$\begin{array}{r} 1001 \\ +\,0101 \\ \hline 1110 \end{array} \rightarrow (-2)_{10}.$$

9. $(-7)_{10} - (+1)_{10} \Rightarrow (-7)_{10} + (-1)_{10}$.

$$\begin{array}{r} 1001 \\ +\,1111 \\ \hline 1000 \end{array} \rightarrow (-8)_{10}.$$

10. $(+2)_{10} - (-3)_{10} \Rightarrow (+2)_{10} + (+3)_{10}$

$$\begin{array}{r} 0010 \\ 0011 \\ \hline 0101 \end{array} \rightarrow (+5)_{10}.$$

Appendix

Binary, signed-integer representations

B	Sign and mag.	1's comp.	2's comp
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

2's comp is preferred since +0 and -0 are both represented by 0000

Addition / Subtraction in 1's complement form:

1. $(+2)_{10} + (+3)_{10}$.

$$\begin{array}{r} 0010 \\ 0011 \\ \hline 0101 \end{array} \rightarrow (+5)_{10}$$

2. $(+4)_{10} + (-6)_{10}$.

$$\begin{array}{r} 0100 \\ 1001 \\ \hline 1101 \end{array} \rightarrow (-2)_{10}.$$

3. $(-5)_{10} + (-2)_{10}$

```
  1010
+ 1101
------
1] 0111
     +1
------
  1000  → (-7)10.
------
```

(carry generated must be added)

4. $(-7)_{10} - (+1)_{10} \Rightarrow (-7)_{10} + (-1)_{10}$.

```
  1000
+ 1110
------
1] 0110
     + 1
------
  0111
```

→ Overflow. ⇒ Result exceeds −7

5. $(-2)_{10} + (-3)_{10}$.

```
  1101
+ 1100
------
1] 1001
     +1
------
  1010 → (-5)10.
```

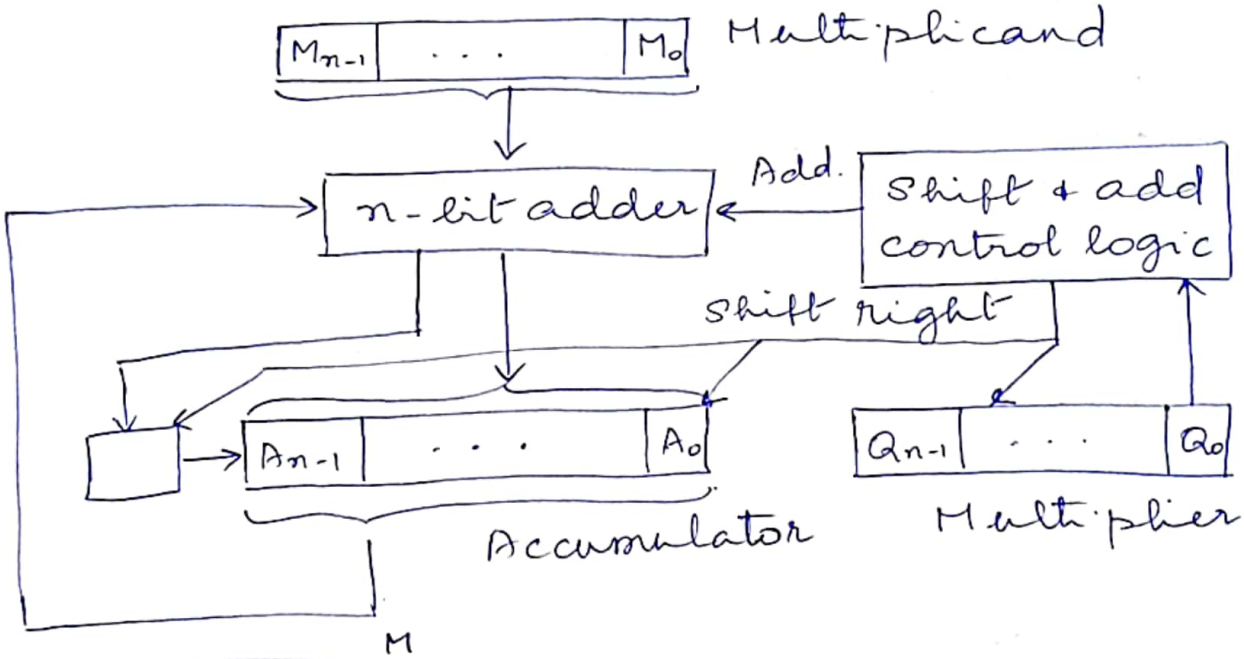
(carry generated must be added).

MULTIPLICATION:

Manual multiplication:

```
    1 1 0 1     (13) - Multiplicand
    1 0 1 1     (11) - Multiplier.
  -----------
    1 1 0 1
  1 1 0 1
0 0 0 0                 } Partial products
1 1 0 1
-----------------
1 0 0 0 1 1 1 1   (143) - Product.
-----------------
```

Sequential circuit binary multiplier:



	M		
	1 1 0 1		} Initial configuration
0	0 0 0 0	1 0 1 1	
C	A	Q	
0	1 1 0 1	1 0 1 1	} First cycle — Add + shift
0	0 1 1 0	1 1 0 1	
1	0 0 1 1	1 1 0 1	} Second cycle — Add + shift
0	1 0 0 1	1 1 1 0	
0	1 0 0 1	1 1 1 0	} Third cycle — No add + shift
0	0 1 0 0	1 1 1 1	

| 1101 | M.

C	A	Q.	
0	0100	1111	← End of third cycle.
1	0001	1111	} Fourth cycle: Add + Shift
0	1000	1111	}

Product.

Using the sequential hardware, shown in the previous page, it is clear that a MULTIPLY instruction takes much more time to execute than an ADD instruction.

∴ Techniques have been proposed to speed up multiplication.

Signed-operand multiplication:

$(-13)_{10} \times (+11)_{10}$.

```
          1 0 0 1 1   (-13)
          0 1 0 1 1   (+11)
10 10
1 1 1 1 1 1 0 0 1 1
1 1 1 1 1 0 0 1 1
0 0 0 0 0 0 0 0
1 1 1 0 0 1 1
0 0 0 0 0 0
-----------------
1 1 0 1 1 1 0 0 0 1  → (-143)10
-----------------
```

Extend the sign bit value of the multiplicand to the left as far as the product will extend.

If the multiplier is negative, reverse the signs of the multiplicand* and the multiplier to get the right answer.

Booth algorithm can handle a negative multiplier and can be used to reduce computations to an extent too.

* That is, take the 2's complement of the multiplicand (becomes -ve) and the multiplier (becomes +ve).

BOOTH ALGORITHM:

Recode the multiplier. - As the multiplier is scanned from right to left, when moving from 0 to 1, perform -1 times the shifted multiplicand and when moving from 1 to 0, perform 1 times the shifted multiplicand.

Multiplier Bit i	Bit i-1	Version of multiplicand
0	0	0 × M
0	1	+1 × M
1	0	-1 × M
1	1	0 × M

Booth recoding of a multiplier.

Multiplier ↙

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 ↘

Recoded multiplier

Example 1:

$(-13)_{10} \times (11)_{10}$ by Booth's algorithm:

1 0 0 1 1

+1 -1 +1 0 -1

0 0 0 0 0 0 1 1 0 1

0 0 0 0 0 0 0 0 0 0

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 → $(-143)_{10}$.

Recoding $(11)_{10}$.

0 1 0 1 1 0

↓

+1 -1 +1 0 -1

2's complement of

10011 → 01101

Example 2:

$(+13)_{10} \times (-6)_{10}$

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

$\hookrightarrow (-78)_{10}$

2's comp. of 6

```
0 0 1 1 0
1 1 0 0 1
    + 1
-----------
1 1 0 1 0  → -6
```

Recoded multiplier

0 -1 +1 -1 0

2's comp. of 13

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

Categories of Booth recoded multipliers:

Worst case (Alternate 1's and 0's)

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

⇓

+1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1

Ordinary multiplier:

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

⇓

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

Good multiplier (Blocks of 1's).

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

⇓

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

Attractive features of Booth algorithm:

1. Handles +ve and -ve numbers uniformly.
2. Achieves efficiency in the no. of additions when the multiplier has blocks of 1's.

On an average, the speed of doing multiplication with the Booth algorithm is the same as with the normal algorithm.

Example 3:

$(+45)_{10} \times (+30)_{10}$.

$(+45)_{10}$: 0101101 $(+30)_{10}$: 0011110.

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

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 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 → $(1350)_{10}$.

Recoded multiplier: 0 +1 0 0 0 -1 0

2's comp of 0101101 is 1010011

Example 4:

$(010111)_2 \times (110110)_2$.

Recoded multiplier: 0 -1 +1 0 -1 0

2's comp. of 010111 : 101001

0 1 0 1 1 1

0 -1 +1 0 -1 0

0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 0 1 0 0 1

0 0 0 0 0 0 0 0 0 0

0 0 0 1 0 1 1 1

1 1 1 0 1 0 0 1

0 0 0 0 0 0 0

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

Verification :

$(010111)_2 \rightarrow 16+4+2+1 = (23)_{10}$

$(110110)_2 \rightarrow -(001010) = -(8+2) = (-10)_{10}$

Product should be $(-230)_{10}$

Product obtained :

$(111100011010)_2 = -(000011100110)$

$= (-230)_{10}$.

Example 5 :

$(110011)_2 \times (101100)_2$.

Recoded multiplier : -1 +10 -100

2's complement of 110011 : 001101

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

Verification :

$(110011)_2 = -(001101) = (-13)_{10}$

$(101100)_2 = -(010100) = (-20)_{10}$.

Product should be $(260)_{10}$

Product obtained :

$(000100000100)_2 = (260)_{10}$.

FAST MULTIPLICATION: MODIFIED BOOTH / BIT-PAIR RECODING

Halves the maximum number of summands. Bits of the multiplier are grouped in 3's and recoded according to the following table.

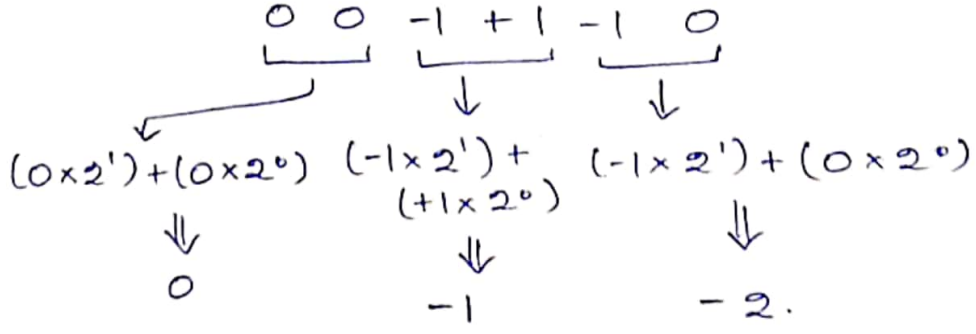
Multiplier bits			Multiplicand selected at position i.
i+1	i	i-1	
0	0	0	0 × M
0	0	1	+1 × M
0	1	0	+1 × M.
0	1	1	+2 × M
1	0	0	-2 × M.
1	0	1	-1 × M.
1	1	0	-1 × M.
1	1	1	0 × M.

To understand how the last column is derived, consider the following example:

Multiplier: 1 1 0 1 0 0.

First derive the Booth recoded multiplier: 0 -1 +1 -1 0

Group the above in pairs. Since we have an odd number of bits, add a zero after before the MSB.



After performing bit-pair recoding on the multiplier, proceed to multiply the multiplicand with the same.

Note:

$-2 \times$ Multiplicand = $2 \times$ 2's complement of multiplicand.

Further, multiplying any number by 2 results in the number followed by a 0 in the binary form.

For example: $2 \times 3 = 6$

$11 \quad 110$

$2 \times 5 = 10$

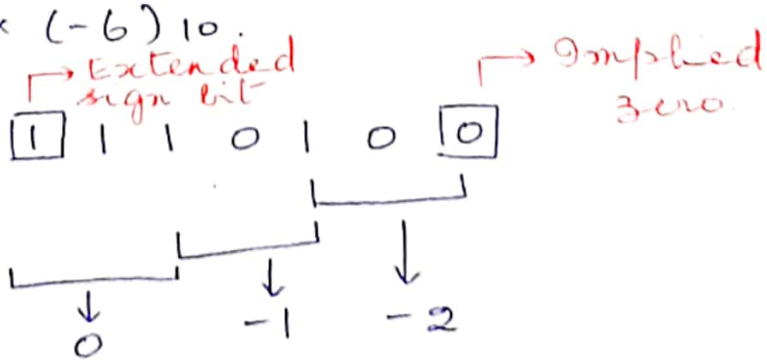
$101 \quad 1010$

∴ When the multiplicand is multiplied by 2, the result will be the multiplicand followed by 0.

When the multiplicand is multiplied by -2, the result will be the 2's complement of the multiplicand followed by 0.

Example 1: $(+13)_{10} \times (-6)_{10}$.

Bit pair recoded multiplier:



∴ The bit-pair recoded multiplier is 0 -1 -2.

Multiplicand: 01101

2's comp. of multiplicand: 10011

```
                    0   1   1   0   1
                    0   -1  -2
                ---------------------
            1   1   1   1   1   0   0   1   1   0
            1   1   1   1   1   0   0   1   1
            0   0   0   0   0   0
        ---------------------------------------
    X   1   1   1   0   1   1   0   0   1   0
```

} Shift by 2 places each time.

carry discarded.

Product: 1111011001 0 $\Rightarrow -(0001001110)_2$.

$= (-78)_{10}$.

Example 2:

$(010111)_2 \times (110110)_2 \Rightarrow (23)_{10} \times (-10)_{10}$.

Bit-pair recoding of the multiplier:

```
    1   1   0   1   1   0   0
                        |___|
                    |_______|   -2
            |_______|   +2
        |___|   -1
```

2's complement of $(010111)_2 = (101001)_2$.

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

Product: 1111000110 10 $\Rightarrow -(000011100110)_2$

$= (-230)_{10}$.

Example 3:

$(001111)_2 \times (001111)_2 \Rightarrow (15)_{10} \times (15)_{10}$.

Bit-pair recoding of the multiplier:

0 0 1 1 1 1 0

+1 0 -1

∴ The bit-pair recoded multiplier is +10-1.

2's complement of $(001111)_2 = (110001)_2$.

```
                0. 0 0 1 1 1 1
                     +1  0  -1
                  ---------------
1 1 1 1 1 1 1 1 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1
-----------------------------------
0 0 0 0 1 1 1 0 0 0 0 1
-----------------------------------
```

Product: $(000011100001)_2 = (225)_{10}$.

Example 4:

$(110101)_2 \times (011011)_2$.

Bit-pair recoding of the multiplier:

0 1 1 0 1 1 0

+2 -1 -1

Bit-pair recoded multiplier: +2-1-1.

2's complement of $(110101)_2 = (001011)_2$.

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

Verification :

$(110101)_2 = -(001011) = (-11)_{10}$

$(011011)_2 = (27)_{10}$.

The product should be $(-297)_{10}$.

Product obtained :

$(111011010111)_2 = -(000100101001)_2$.

$= (-297)_{10}$.

Hardware Implementation of Booth's Algorithm

1. $(13)_{10} \times (-11)_{10}$.

$(+13)_{10} \rightarrow 01101$ $\quad$ $(+11)_{10} \rightarrow 01011$

$(-13)_{10} \rightarrow 10011$ $\quad$ $(-11)_{10} \rightarrow 10101$

	M	A	Q	Q_{-1}	
	01101	00000	10101	0	
A-M		10011	10101	0	First cycle.
Shift		11001	11010	1	
A+M		00110	11010	1	Second cycle.
Shift		00011	01101	0	
A+M					
A-M		10110	01101	0	Third cycle.
Shift		11011	00110	1	
A+M		01000	00110	1	Fourth cycle.
Shift		00100	00011	0	
A-M		10111	00011	0	Fifth cycle.
Shift		11011	10001	1	

Product.

$(1101110001)_2 = -(0010001111)_2 = (-143)_{10}$

2. $(-3)_{10} \times (7)_{10}$.

$(+3)_{10} \rightarrow 0011$ $\quad$ $(+7)_{10} \rightarrow 0111$

$(-3)_{10} \rightarrow 1101$

	A	Q	Q-1	M	
	0000	0111	0	1101	
A-M	0011	0111	0		} First cycle.
Shift	0001	1011	1		
No add	0001	1011	1		} Second cycle.
Shift	0000	1101	1		
No add.	0000	1101	1		} Third cycle
Shift	0000	0110	1		
A+M	1101	0110	1		} Fourth cycle.
Shift	1110	1011	0		

Product.

$(11101011)_{2} \rightarrow -(00010101)_{2} = (-21)_{10}$.

Hardware implementation of modified Booth's algorithm:

1. $(110101)_2 \times (011011)_2$

2's comp. of M
001011

	A	Q	Q_{-1}	M	
	000000	011011	0	110101	
			-1		
A-M	001011	011011	0		} First cycle.
Shift twice	000010	110110	1		
			-1		
A-M	001101	110110	1		} Second cycle
Shift twice	000011	011101	1		
			+2		
A+2M	101101	011101	1		} Third cycle.
Shift twice	111011	010111	0		

Product.

Verification: $(110101)_2 = -(001011)_2 = (-11)_{10}$

$(011011)_2 = (27)_{10}$

Expected product $= (-297)_{10}$

Product obtained: $(111011010111)_2$

$= -(000100101001)_2$

$= (-297)_{10}$.

Note:

No. of cycle = No. of bits in multiplier / 2.

2. $(13)_{10} \times (-11)_{10}$

$(13)_{10} = (01101)_2$.

$(-11)_{10} = -(01011)_2 = (10101)_2$.

For modified Booth's algorithm, ensure that the multiplier and multiplicand are represented with an even no. of bits

$\therefore (13)_{10} \rightarrow (001101)_2$ 2's comp: 110011

$(-11)_{10} \rightarrow (110101)_2$.

	A	Q	Q_{-1}	M
	000000	110101	0	001101
			+1	
A+M	001101	110101	0	} First cycle.
Shift	000011	011101	0	
			+1	
A+M	010000	011101	0	} Second cycle.
Shift	000100	001110	0	
			-1	
A-M	110111	000111	0	} Third cycle.
Shift	111101	110001	1	

Product.

Product: $(111101110001)_2$

$= -(000010001111)_2$

$= (-143)_{10}$

RESTORING DIVISION - FOR UNSIGNED numbers.

Long hand division example:

```
               10101
       ___________
  1101)100010010
         1101↓↓
        -----
          10000
           1101↓↓
         --------
             1110
             1101
            -----
                1
```

A circuit that implements this longhand division operates as follows:

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

Algorithm for Restoring Division.

Do the following n times: n = no. of bits in the dividend

1. Shift A and Q one position to the left.
2. Subtract M from A, and place the result in A
3. If sign (MSB) of A is 1, set Q_0 to 0 and add M back to A (restore A). Otherwise, set Q_0 to 1.

Initially,

A contains 0's. (n+1 zeros)

Q contains the dividend (n-bits long)

M contains the divisor.

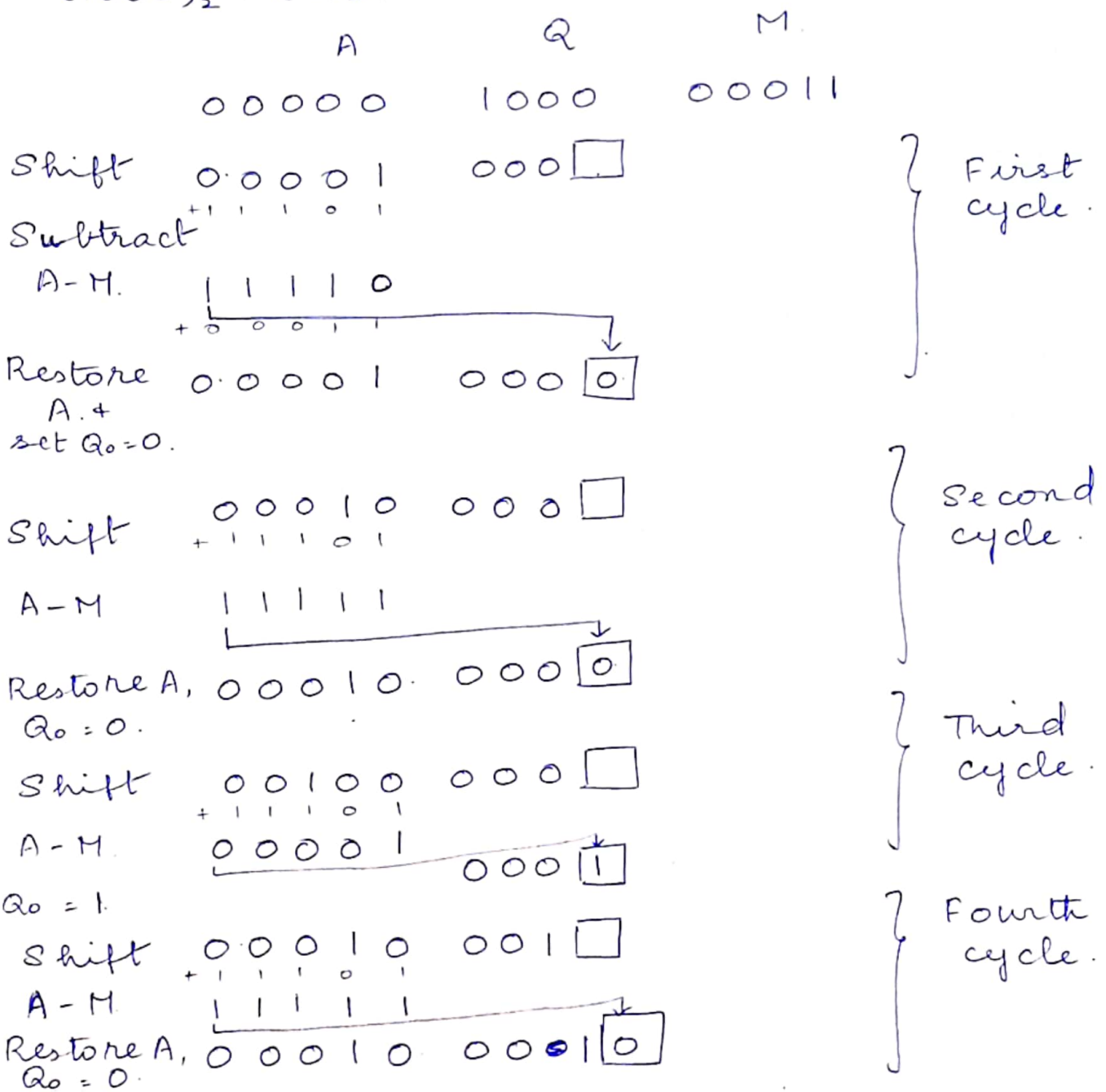
At n cycles,

A contains the remained

Q contains the quotient.

Example 1:

$(1000)_2 \div (11)_2$.



∴ Quotient = 10
Remainder = 10.

Example 2:

$(1010)_2 \div (0011)_2$

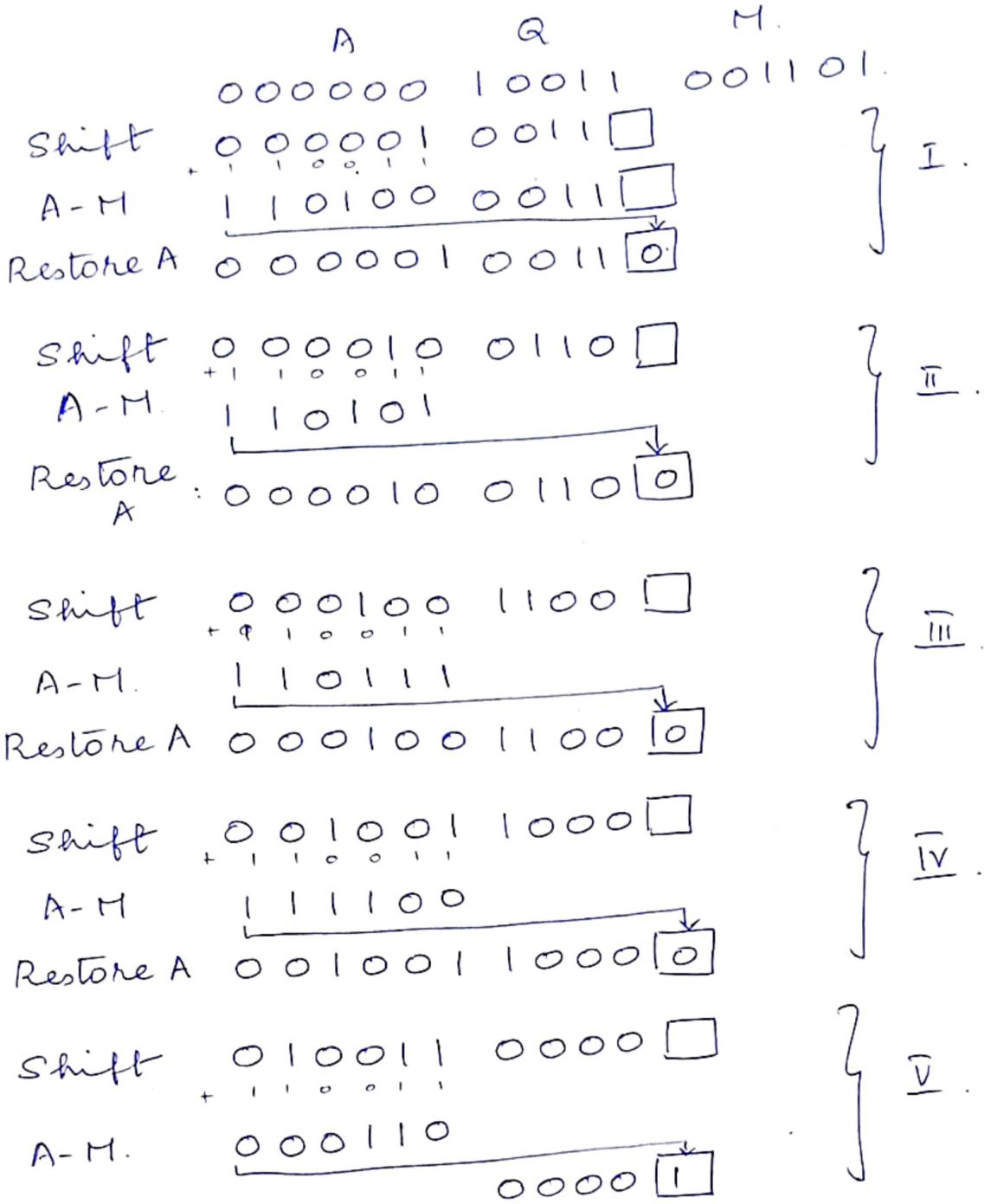
n = no. of bits in dividend = 4.
∴ A should contain 5 bits.

	A	Q	M	
	00000	1010	00011	
Shift	00001	010 □		First cycle.
A − M	+ 11101			
	11110			
Restore A, $Q_0 = 0$.	00001	010 [0]		
Shift	00010	100 □		Second cycle.
A − M	+ 11101			
	11111			
Restore A, $Q_0 = 0$.	00010	100 [0]		
Shift	00101	000 □		Third cycle.
A − M	+ 11101			
	00010			
$Q_0 = 1$.		000 [1]		
Shift	00100	001 □		
A − M	+ 11101			
	00001			
$Q_0 = 1$		001 [1]		

∴ Quotient = 0011.
Remainder = 0001.

Example 3:

$(10011)_2 \div (1101)_2$.



∴ Quotient = 0001

Remainder = 0110.

NON-RESTORING DIVISION:

The division algorithm can be made more efficient / less complex by avoiding the "restoring" step. To do that, consider what actually happens in the restoring division algorithm, specifically within the accumulator.

① Shift A, Q ⇒ 2A
to the left

Subtract M ⇒ 2A - M.

If A is positive, we stop here.

If A is negative:

② Add M to A to restore A ⇒ A + M.

Shift left ⇒ 2(A+M).

Subtract M ⇒ 2(A+M) - M = 2A + M.

∴ If A is positive, shift left (2A) and then subtract M. If A is negative, shift left and then add M.

The non-restoring division algorithm is as follows:

Step 1: Repeat the following N times:

(a) If the sign of A is 0, shift A and Q one position to the left and subtract M from A, else, shift A and Q to the left and add M to A.

(b) If the sign of A is 0, set Q_0 to 1, if not, set Q_0 to 0.

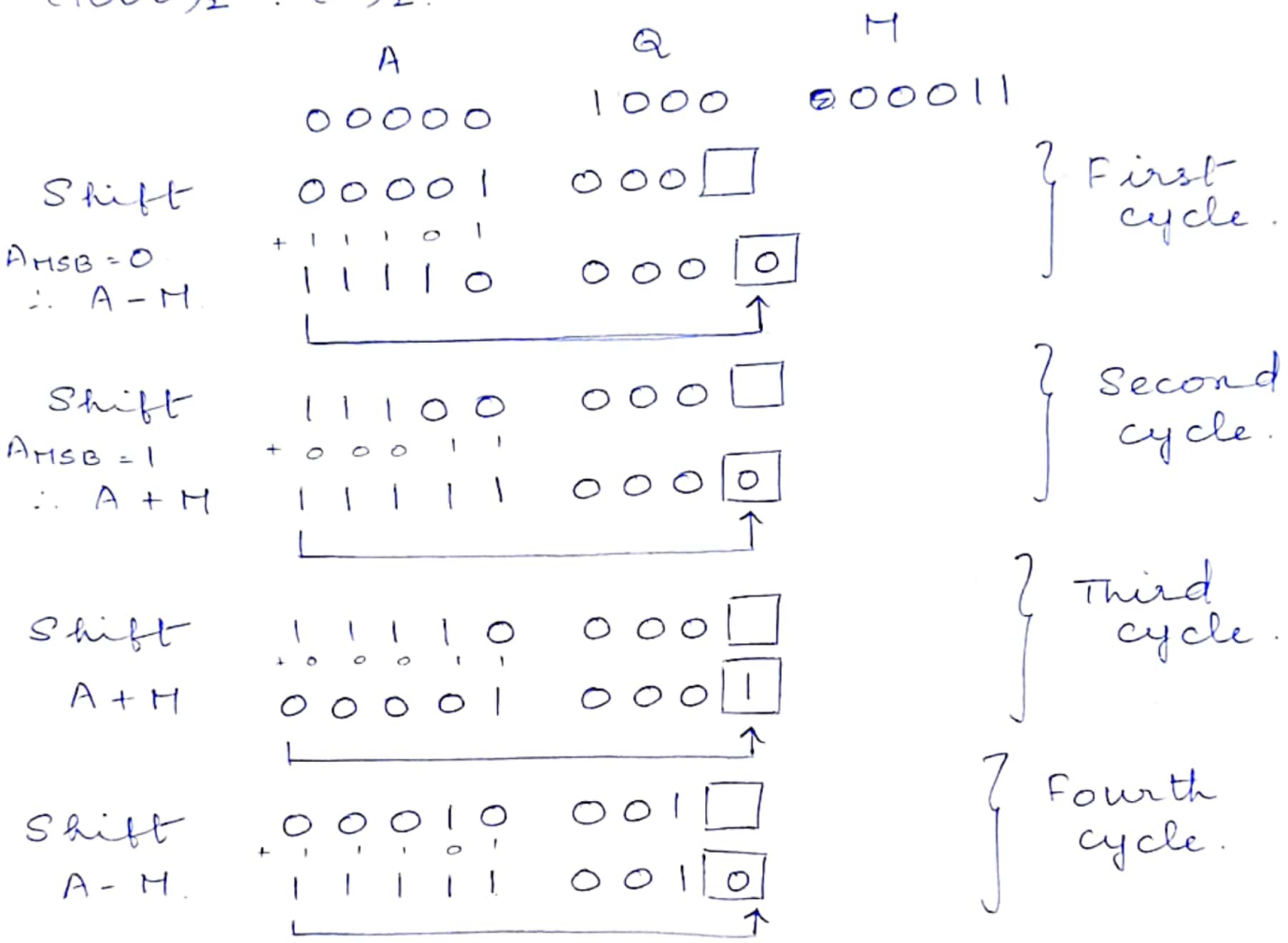
Step 2: If the sign of A is 1, add M to A.

(Step 2 is performed to leave the proper remainder in A, after n cycles).

Advantage: Restore operation not required and only one add / subtract operation per cycle.

Example 1:

$(1000)_2 \div (11)_2$.



Quotient = 0010.

To find the remainder, since AMSB = 1, add M. to it:

A : 1 1 1 1 1

M : 0 0 0 1 1

Remainder : 0 0 0 1 0

Example 2:

$(10011)_2 \div (1101)_2$.

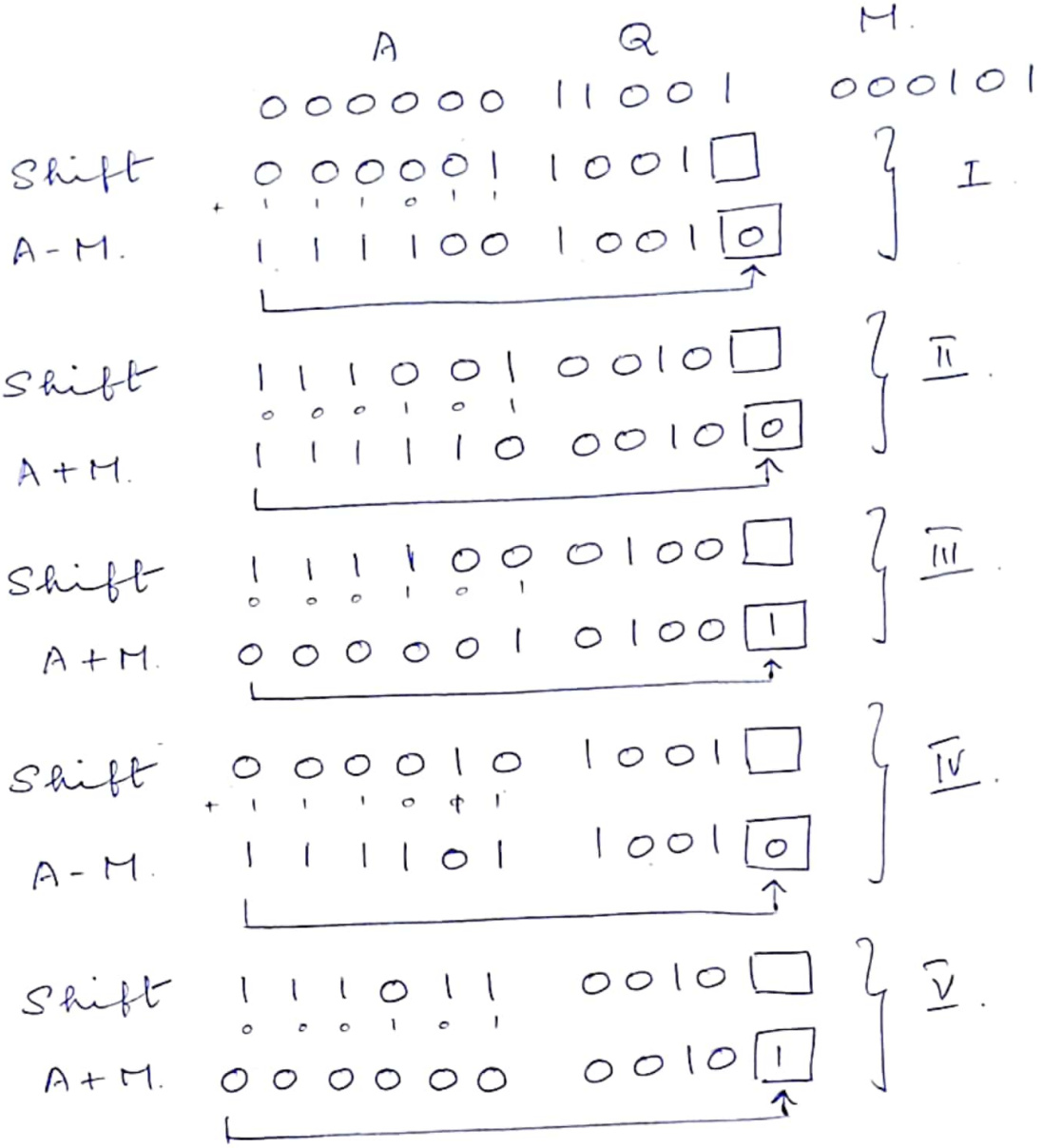
M : 001101.
2's comp : 110011.

	A	Q	M
	000000	10011	1101
Shift	000001	0011 □	} I.
A − M	110100	0011 0	
Shift	101000	0110 □	} II
A + M	110101	0110 0	
Shift	101010	1100 □	} III.
A + M.	110111	1100 0	
Shift	101111	1000 □	} IV.
A + M	111100	1000 0	
Shift	111001	0000 □	} V.
A + M	000110	0000 1	

∴ Quotient = 0001
Remainder = 0110. ($\because$ A MSB = 0).

Example 3:

$(25)_{10} \div (5)_{10} \Rightarrow (11001)_2 \div (101)_2$.



Quotient $= (101)_2$.

Remainder $= 0$.

FLOATING POINT NUMBERS AND OPERATIONS

To accomodate both very large integers and very small 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 such a case, the binary point is said to float and the numbers are called floating point numbers.

Decimal scientific notation

Examples: 6.0247×10^{23}, 6.6254×10^{-27}, etc.

↳ 5 significant digits

Scale factors

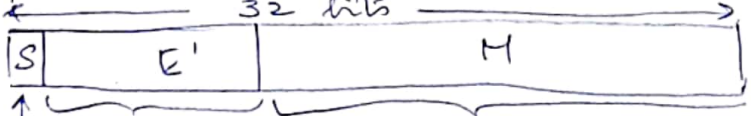
When decimal point is placed to the right of the first non-zero significant digit, the number is said to be NORMALIZED.

Floating point representation → one in which a number is represented by its sign, a string of significant digits, called the MANTISSA, and an EXPONENT to an implied base for the scale factor.

IEEE Standard Floating-Point Formats:

- This standard describes both the representation and the way in which the 4 arithmetic operations are to be performed.

Single precision representation (32 bits)



Sign bit
0: positive
1: negative

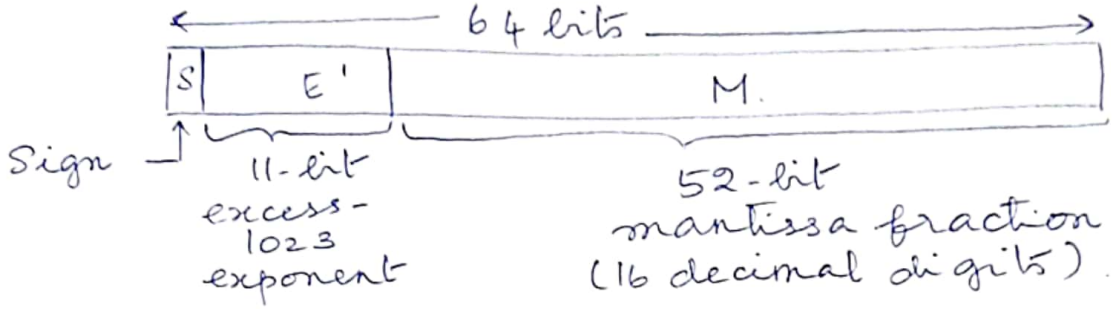
8-bit signed exponent in excess-127 representation

23-bit mantissa fraction (7 digital decimal number)

Excess-127 format:
$E' = E + 127$

⇒ Value represented:
$\pm M \times 2^{E'-127}$

Double precision representation:



Value represented: $\pm 1.M \times 2^{E'-1023}$.

Excess-1023 format: $E' = E + 1023$.

E → signed exponent

E' → unsigned integer. (Biased Exponent).

On the case of single precision, E will range between −126 and +127, whereas E' will range between 0 and 255. The end values 0 and 255 are used to represent special values. For normal values, E' ranges between 1 and 124.

Single precision format allows a scale factor in the range 2^{-126} to 2^{+127}, which is approximately $10^{\pm 38}$ in decimal form.

On the case of double precision, E' ranges between 1 and 2046 for normal values and 0 and 2047 are used for special values. E lies between −1022 and +1023, which is approximately $10^{\pm 308}$ in decimal form.

Special values:

When E' = 0 and M = 0, ⇒ 0 is represented.

When E' = 255 (or 2047) and M = 0 ⇒ ∞ is represented.

When E' = 0 and M ≠ 0 ⇒ denormal numbers
↳ $\pm 0.M \times 2^{-126}$.
They are smaller than the smallest normal number.

No implied 1 to the left of the binary point.

The purpose of introducing denormal numbers is allow for gradual underflow and hence deal with very small numbers.

When $E' = 255$ (or 2047) and $M \neq 0 \Rightarrow$ Not a Number (NaN)

↓

Result of performing an invalid operation like $0/0$ or $\sqrt{-1}$.

Problems:

1. Represent $(1259.125)_{10}$ in single and double precision format.

Soln:

① Conversion from decimal to binary

2	1259	
2	629	-1
2	314	-1
2	157	-0
2	78	-1
2	39	-0
2	19	-1
2	9	-1
2	4	-1
2	2	-0
	1	-0

$0.125 \times 2 = 0.25$ → 0

$0.25 \times 2 = 0.50$ → 0

$0.5 \times 2 = 1.00$ → 1

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

$(1259)_{10} = (10011101011)_2$.

$(1259.125)_{10} = (10011101011.001)_2$.

② Normalize the number:

$1.00111010111001 \times 2^{10}$

③ Single precision format:

$S = 0 \quad E = 10 \quad M = 00111010111001$

$E' = E + 127 = (137)_{10}$.

$E' = (10001001)_2$.

2 | 137
2 | 68 - 1
2 | 34 - 0
2 | 17 - 0
2 | 8 - 1
2 | 4 - 0
2 | 2 - 0
1 - 0.

The number in single precision format is

1	8 bits	23 bits
0	10001001	00111010111001000000000

④ Double precision format:

$S = 0 \quad E = 10 \quad M = 00111010111001$

$E' = E + 1023 = (1033)_{10}$.

$E' = (10000001001)_2$.

2 | 1033
2 | 516 - 1
2 | 258 - 0
2 | 129 - 0
2 | 64 - 1
2 | 32 - 0
2 | 16 - 0
2 | 8 - 0
2 | 4 - 0
2 | 2 - 0
1 - 0.

The number in double precision format is

1	11 bits	52 bits
0	10000001001	00111010111001000...0

2. Represent $(-307.1875)_{10}$ in single and double precision formats.

Soln:

① Binary conversion:

$(307)_{10} = (100110011)_2$

2 | 307	
2 | 153 - 1	
2 | 76 - 1	
2 | 38 - 0	
2 | 19 - 0	
2 | 9 - 1	
2 | 4 - 1	
2 | 2 - 0	
1 - 0	

$0.1875 \times 2 = 0.3750$ → 0

$0.375 \times 2 = 0.750$ → 0

$0.75 \times 2 = 1.5$ → 1

$0.5 \times 2 = 1.0$ → 1

$(0.1875)_{10} = (0.0011)_2$.

$-(307.1875)_{10} = -(100110011.0011)_2$

② Normalize the number.

-1.00110011×2^8

③ Single precision:

$S = 1 \quad E = 8 \quad M = 00110011$.

$E' = E + 127 = (135)_{10}$.

$E' = (10000111)_2$.

2 | 135	
2 | 67 - 1	
2 | 33 - 1	
2 | 16 - 1	
2 | 8 - 0	
2 | 4 - 0	
2 | 2 - 0	
1 - 0	

The number in single precision form is:

1 10000111 00110011001100000000000

8 bits 23 bits

④ Double precision format:

S = 1 E = 8 M = 00110011001 1

$E' = E + 1023 = (1031)_{10}$

$E' = (10000000111)_2$.

2 | 1031
2 | 515 - 1
2 | 257 - 1
2 | 128 - 1
2 | 64 - 0
2 | 32 - 0
2 | 16 - 0
2 | 8 - 0
2 | 4 - 0
2 | 2 - 0
1 - 0

The number in double precision format is

1 | 10000000111 | 00110011001100000....0

11 bits | 52 bits

3. Give the IEEE 754 binary representation of $-(0.75)_{10}$ in single and double precision.

Soln:

① Binary conversion:

$0.75 \times 2 = 1.50$ → 1

$0.5 \times 2 = 1.00$ → 1

$-(0.75)_{10} = -(0.11)_2$.

② Normalize the number:

-1.1×2^{-1}

③ Single precision format:

S = 1 E = -1 M = 1

$E' = E + 127 = (126)_{10} = (01111110)_2$.

The number in single precision format is

1 | 01111110 | 10000000000...0

8 bits | 23 bits

④ Double precision format:

S = 1 E = -1 $E' = E + 1023 = (1022)_{10} = (01111111110)_2$

The number in double precision format is

1 | 01111111110 | 100000...00 → 52 bits

Floating point operations:

Addition and subtraction:

Steps / Rules:

1. Given 2 numbers, choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result to the larger exponent.
3. Perform addition / subtraction on the mantissa and determine the sign of the result.
4. Normalize the result, if necessary.

Multiplication Rules:

1. Add the exponents and subtract 127.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the result, if required.

Division Rules:

1. Subtract the exponents and add 127.
2. Divide the mantissas and determine the sign of the result.
3. Normalize the result, if required.

Note:

For multiplication and division, 127 is subtracted / added to the exponent, assuming excess-127 representation. If excess-1023 is to be used, add / subtract 1023 instead.

Problems:

1. Add $(3.75)_{10}$ and $(5.125)_{10}$ in binary form and represent in single precision form.

Soln:

① Binary conversion:

$(3.75)_{10}$

$(3)_{10} \rightarrow (11)_2$.

$(0.75)_{10}$

$0.75 \times 2 = 1.50$ → 1

$0.50 \times 2 = 1.00$ → 1

$\therefore (3.75)_{10} = (11.11)_2$.

$(5.125)_{10}$

$(5)_{10} \rightarrow (101)_2$.

$(0.125)_{10}$.

$0.125 \times 2 = 0.25$ → 0

$0.25 \times 2 = 0.5$ → 0

$0.5 \times 2 = 1.0$ → 1

$(5.125)_{10} = (101.001)_2$.

② Normalize:

$11.11 \rightarrow 1.111 \times 2^1$

$101.001 \rightarrow 1.01001 \times 2^2$.

③ Shift the mantissa of the number with the smaller exponent.

$1.111 \times 2^1 \rightarrow 0.1111 \times 2^2$.

④ Add the mantissas:

$$\begin{array}{r} 1.01001 \times 2^2 \\ 0.11110 \times 2^2 \\ \hline 10.00111 \times 2^2 \end{array}$$

⑤ Normalize the result:

1.000111×2^3

⑥ Single precision format: $S = 0$, $E = 3$, $E' = 130$

M = 000111

0 10000010 00011100000...0

8 bits 23 bits

2. Add $(-3.75)_{10}$ and $(5.125)_{10}$ in binary form.

Soln:

① Binary conversion:

$(-3.75)_{10}$.

$0.75 \times 2 = 1.50$ → 1

$0.50 \times 2 = 1.00$ → 1

$(-3.75)_{10} \rightarrow (-11.11)_2$.

$(5.125)_{10}$

$0.125 \times 2 = 0.25$ → 0

$0.25 \times 2 = 0.5$ → 0

$0.5 \times 2 = 1.0$ → 1

$(5.125)_{10} = (101.001)_2$.

② Normalize:

$-11.11 \rightarrow -1.111 \times 2^1$

$101.001 \rightarrow 1.01001 \times 2^2$.

③ Shift the mantissa of the number with the smaller exponent.

$-1.111 \times 2^1 \rightarrow -0.1111 \times 2^2$

④ Add the mantissas:

$-0.1111 \times 2^2 \rightarrow 1.0001 \times 2^2$ (2's comp. form)

$$\begin{array}{r} 1.00010 \times 2^2 \\ +\ 1.01001 \times 2^2 \\ \hline 1\,\big]\ 0.01011 \times 2^2 \end{array}$$

⑤ Normalize the result:

$0.01011 \times 2^2 = 1.011 \times 2^0$.

Single precision form: $E' = 0 + 127 = 127$. $S = 0$.

$$0\ \underbrace{01111111}_{8\text{ bits}}\ \underbrace{01100000\ldots0}_{23\text{ bits}}$$

3. Add $(3.75)_{10}$ and $(-5.125)_{10}$ in binary form.

Soln:

① Binary conversion:

$(3.75)_{10} \rightarrow (11.11)_2$

$(-5.125)_{10} \rightarrow -(101.001)_2$.

② Normalize.

$11.11 \rightarrow 1.111 \times 2^1$

$-101.001 \rightarrow -1.01001 \times 2^2$.

③ Adjust the mantissa of the number with the smaller exponent:

$1.111 \times 2^1 \rightarrow 0.1111 \times 2^2$.

④ Add the mantissas:

$-1.01001 \times 2^2 \longrightarrow 0.10111 \times 2^2$.

$$\begin{array}{r} 0.11110 \times 2^2 \\ +\ 0.10111 \times 2^2 \\ \hline 1.10101 \times 2^2 \end{array}$$

This is already normalized ⇒ It's a negative number.

∴ Take the 2's complement.

-0.01011×2^2

⑤ Normalize:

-1.011×2^0

Single precision format: $S = 1$, $E = 0$, $E' = 127$

$$1 \quad \underbrace{01111111}_{8 \text{ bits}} \quad \underbrace{011000\ldots0}_{23 \text{ bits}}$$

4. Multiply $(4.5)_{10}$ and $(2.5)_{10}$ in binary form.

Soln:

① Binary conversion:

$0.5 \times 2 = 1.00$ → 1

$(0.5)_{10} \rightarrow (0.1)_2$.

$\therefore (4.5)_{10} = (100.1)_2$

$(2.5)_{10} = (10.1)_2$.

② Normalize:

$100.1 \rightarrow 1.001 \times 2^2$.

$10.1 \rightarrow 1.01 \times 2^1$

③ Add the exponents. (9+ dealing with biased exponents, then add them and subtract 127).

$\therefore 1 + 2 = 3 = E$.

④ Multiply the mantissas.

$$\begin{array}{r} 1.001 \\ \times 1.01 \\ \hline 1001 \\ 0000 \\ 1001 \\ \hline 1.01101 \end{array}$$

Result: 1.01101×2^3 (From steps 3 + 4).