Course Name:	**Computer Organization and Architecture Laboratory**	**Semester:**	**III**
Date of Performance:	**14/08/25**	**Batch No:**	**B1**
Faculty Name:	**Prof. Sheetal Pereira**	**Roll No:**	**16010124080**
Faculty Sign & Date:		**Grade/Marks:**	**25**

Experiment No: 02

Title: Restoring method of division

Aim and Objective of the Experiment:

Aim: The basis of algorithm is based on paper and pencil approach and the operation involve repetitive shifting with addition and subtraction. So the main aim is to depict the usual process in the form of an algorithm.

Objective: To implement the restoring division algorithm using shift, subtract/add operations to simulate binary division.

COs to be achieved:

CO1: Describe and define the structure of a computer with buses structure and detail working of the ALU

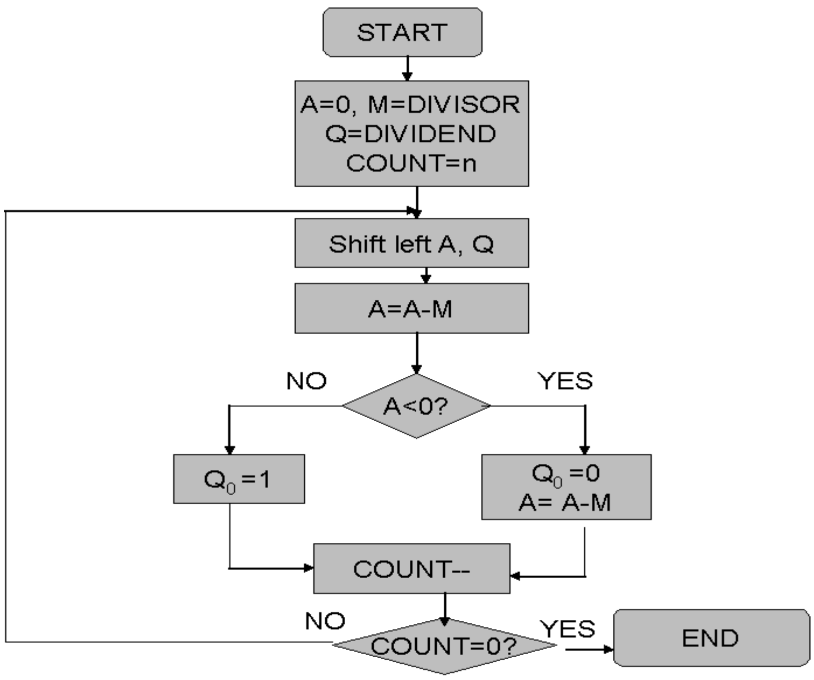
Books/Journals/Websites referred:

1. Carl Hamacher, Zvonko Vranesic and Safwat Zaky, "Computer Organization", Fifth Edition, TataMcGraw-Hill.
2. William Stallings, "Computer Organization and Architecture: Designing for Performance", Eighth Edition, Pearson.
3. Dr. M. Usha, T. S. Srikanth, "Computer System Architecture and Organization", First Edition, Wiley-India.

Theory: The **restoring division algorithm** is a binary division technique that mimics long division used in decimal arithmetic. It operates on the principle of repeated subtraction and left-shifting. The dividend and divisor are stored in binary registers, and at each iteration, the contents of the accumulator and the quotient registers are left-shifted. The divisor is subtracted from the accumulator, and if the result is negative, the previous value of the accumulator is restored (hence the name "restoring"). This process is repeated for a number of steps equal to the bit length of the operands. The final values in the **quotient** and **accumulator** represent the result of the division and remainder, respectively.

It is primarily used in hardware-based division circuits due to its simple control logic but is slower than the **non-restoring** method due to the additional restoring step.

Flowchart for Restoring of Division:



Design Steps:

1. Start
2. Initialize A=0, M=Divisor, Q=Dividend and count=n (no of bits)
3. Left shift A, Q
4. If MSB of A and M are same

5. Then A=A-M
6. Else A=A+M
7. If MSB of previous A and present A are same
8. Q_0=0 & store present A
9. Else Q_0=0 & restore previous A
10. Decrement count.
11. If count=0 go to 11
12. Else go to 3
13. STOP

Example: - (Handwritten solved problem needs to be uploaded):-

Q M
7 ÷ 3

M : 0011 Q : 0111
-M : 1101

A	Q	count
0000	0111	4
0000	111□	
1101	1110	
0000	1110	3
0001	110□	
1110	1100	
0001	1100	2
0011	100□	
0000	1001	1
0001	001□	
1110	0010	0

R = 0001 Q = 0010

0000
1101

1101
0011
000

0001
1101
111
+0011
0001

0011
+ 1101
0000

0001
+ 1101
~~0000~~ 1
1110

1100

Q M
7 ÷ 3

M : 0011
-M : 1101
Q : 0111

A	Q	count
0000	0111	4
0000	111$\boxed{\ }$	
1101	1110	
0000	1110	3
0001	110$\boxed{\ }$	
1110	1100	
0001	1100	2
0011	100$\boxed{\ }$	
0000	1001	1
0001	001$\boxed{\ }$	
1110	0010	0

R = 0001 Q = 0010

0000
1101

1101
0011
0000

0001
1101
1110
+0011
0001

0011
+ 1101
0000

0001
+ 1101
~~0000~~ ~~1101~~
1110

1100

CODE:-

```python
def compute_bits(num1,num2):

    a = max(num1,num2)

    # Calculate minimum bits to represent max number plus one bit for sign
if needed

    # For simplicity just add some bits depending on range:

    if a <= 7:

        length = 4

    elif a <= 15:

        length = 5
```

```
    elif a <= 21:
        length = 6
    elif a <= 28:
        length = 7
    elif a <= 35:
        length = 8
    elif a <= 41:
        length = 9
    elif a <= 49:
        length = 10
    else:
        print("NOT ENOUGH TO COMPUTE YOUR RESULT")
        length = None
    return length


def to_binary(num: int, bit_length: int) -> list:
    binary_num = [0] * bit_length
    i = bit_length - 1

    while num > 0 and i >= 0:
        binary_num[i] = num % 2
        num = num // 2
        i -= 1
```

```
    return binary_num



def binary_addition(b1, b2):
    carry = 0
    final = [0] * len(b1)

    for i in range(len(b1) - 1, -1, -1):
        total = b1[i] + b2[i] + carry
        final[i] = total % 2
        carry = total // 2

    return final



def to_negative(binary):
    inverted = [1 - bit for bit in binary]
    negative = binary_addition(inverted, ([0]*(len(binary)-1)) + [1])
    if len(negative) > len(binary):
        negative = negative[1:]
    return negative
```

```python
def to_set_initials():
    question = input("Enter division (format m/q): ")
    nums = question.split('/')

    Dividend, Divisor = int(nums[0]), int(nums[1])
    bits_required = compute_bits(Dividend, Divisor)
    if bits_required is None:
        exit()

    M = to_binary(Divisor, bits_required)
    M_neg = to_negative(M)
    Q = to_binary(Dividend, bits_required)
    A = [0] * bits_required
    n = bits_required

    # print(f"M: {M}   M_neg: {M_neg}   Q: {Q}   A: {A}")

    return (M, M_neg, Q, A, n)



def to_restore_divide(M, M_neg, Q, A, n):
    while n != 0:
        # Left shift A and Q combined by 1 bit
        A = A[1:] + [Q[0]]
```

```
        Q = Q[1:]

        A = binary_addition(A, M_neg)

        if A[0] == 1:
            Q = Q + [0]
            A = binary_addition(A, M)
        else:
            Q = Q + [1]

        n -= 1

    return (A, Q)



def to_decimal(binary_list):
    decimal = 0
    length = len(binary_list)
    for i, bit in enumerate(binary_list):
        decimal += bit * (2 ** (length - i - 1))
    return decimal

my_values = to_set_initials()
```

```
remainder, quotient = to_restore_divide(*my_values)

print(f"The answer is : Remainder {to_decimal(remainder)}\nQuotient
{to_decimal(quotient)}")
```

Post Lab Subjective/Objective type Questions:

1. **Explain the role of the accumulator (A) in the restoring division algorithm and how it affects the quotient generation.**
   - **A stores the partial remainder** during division.
   - Each step: shift A and quotient left, subtract divisor from A.
   - If A ≥ 0 after subtraction, set quotient bit to 1.
   - If A < 0, restore A (add divisor back) and set quotient bit to 0.
   - So, A helps decide each quotient bit by showing if divisor fits or not.

2. **What are the advantages of restoring division over non restoring division?**
   - Simpler and easier to understand and implement.
   - Always restores remainder when subtraction fails, ensuring correctness.
   - Quotient bits are directly set by checking remainder sign.
   - Less complex hardware compared to non-restoring division.

Conclusion:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\KJSCE\Desktop\cc> & C:/Users/KJSCE/AppData/Local/Programs/Python/Python312/python.exe c:/Users/KJSCE/Desktop/cc/restoring_div.py
Enter division (format m/q): 10/6
The answer is : Remainder 4
Quotient 1
PS C:\Users\KJSCE\Desktop\cc> & C:/Users/KJSCE/AppData/Local/Programs/Python/Python312/python.exe c:/Users/KJSCE/Desktop/cc/restoring_div.py
Enter division (format m/q): 15/4
The answer is : Remainder 3
Quotient 3
PS C:\Users\KJSCE\Desktop\cc> & C:/Users/KJSCE/AppData/Local/Programs/Python/Python312/python.exe c:/Users/KJSCE/Desktop/cc/restoring_div.py
Enter division (format m/q): 8/4
The answer is : Remainder 0
Quotient 2
PS C:\Users\KJSCE\Desktop\cc>
```

Signature of faculty in-charge with Date: