**Batch: A1**

**Roll No.: 16010123012**

**Experiment / assignment / tutorial No. 3**

**TITLE :** To study and implement Restoring method of division

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

**Expected OUTCOME of Experiment: (Mention CO /CO’s attained here)**

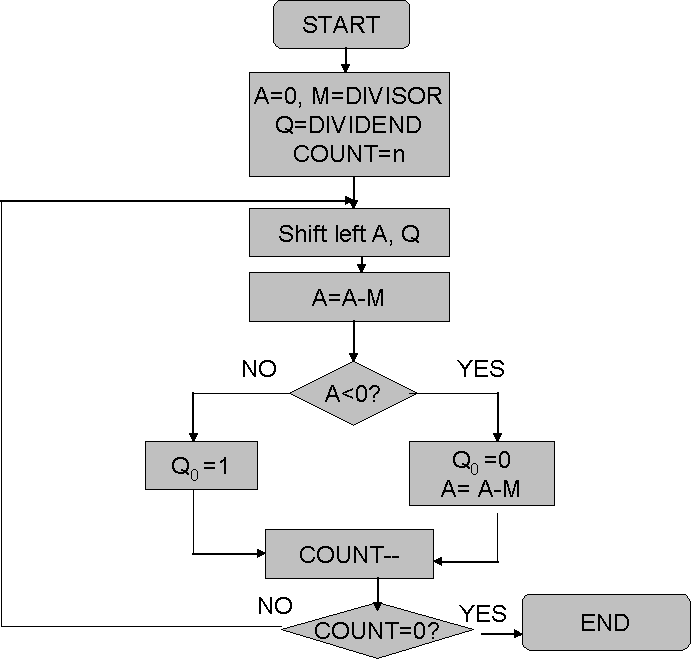
**Books/ Journals/ Websites referred:**

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.

**Pre Lab/ Prior Concepts:**

The Restoring algorithm works with any combination of positive and negative numbers

**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. Q0=0 & store present A
9. Else Q0=0 & restore previous A
10. Decrement count.
11. If count=0 go to 11
12. Else go to 3
13. STOP

**Code for Restoring division in C:**

#include <stdio.h>

#define BITS 4

void print\_binary(int number, int bit\_width) {

for (int i = bit\_width - 1; i >= 0; i--) {

printf("%d", (number >> i) & 1);

}

}

void restoring\_division(int dividend, int divisor, int \*quotient, int \*remainder) {

int accumulator = 0;

int divisor\_value = divisor;

int quotient\_value = dividend;

int count = BITS;

int limit = (1 << BITS) - 1;

printf("\n");

while (count > 0) {

accumulator = ((accumulator << 1) | (quotient\_value >> (BITS - 1))) & limit;

quotient\_value = (quotient\_value << 1) & limit;

printf("Accumulator after shift: ");

print\_binary(accumulator, BITS);

printf("\n");

accumulator -= divisor\_value;

if (accumulator < 0) {

accumulator += divisor\_value;

quotient\_value = quotient\_value & ~1;

} else {

quotient\_value = quotient\_value | 1;

}

count -= 1;

}

\*quotient = quotient\_value;

\*remainder = accumulator;

}

int main() {

int dividend, divisor, quotient, remainder;

printf("Enter dividend : ");

scanf("%d", &dividend);

printf("Enter divisor : ");

scanf("%d", &divisor);

restoring\_division(dividend, divisor, &quotient, &remainder);

printf("\nVaues - Decimal, Binary\n");

printf("Dividend : %d, ", dividend);

print\_binary(dividend, BITS);

printf("\nDivisor : %d, ", divisor);

print\_binary(divisor, BITS);

printf("\nQuotient : %d, ", quotient);

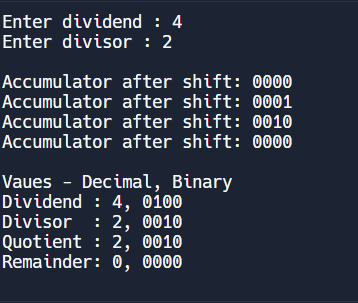
print\_binary(quotient, BITS);

printf("\nRemainder: %d, ", remainder);

print\_binary(remainder, BITS);

return 0;

}

**Output:-**  


**Example:- (Handwritten solved problems needs to be uploaded)**

**Conclusion –**

In this experiment we learned about Restoring method of division and how to

Divide numbers using it. We also verified it using a code that divides numbers

using Restoring method of division.

**Post Lab Descriptive Questions**

1. **What are the advantages of restoring division over non restoring division?**

1.Enhanced Error Recovery: Restoring division boasts superior error recovery capabilities. If an error or an incorrect quotient estimate occurs during the division process, restoring division can more effectively rectify the error and resume the division operation accurately. This advantage stems from the fact that restoring division continually adjusts the remainder towards the correct value.

2. Streamlined Hardware Implementation: Restoring division tends to offer a simpler hardware implementation due to its iterative nature. The hardware design for restoring division is generally more straightforward, making it an attractive option for situations where minimizing hardware complexity is a priority.

3. Versatility Across Radices: Restoring division is adaptable to division in various radix systems, including binary, decimal, and others, with relatively minor modifications. This adaptability makes it a versatile choice for different number representations.

4. Incremental Quotient Computation: Restoring division calculates the quotient incrementally, meaning that at each step, it adds a partial quotient bit to the final quotient. This feature can be advantageous in applications where obtaining the quotient progressively is beneficial, rather than waiting for the entire division to finish.

5. Easier to Understand and Implement: Restoring division is easier to grasp and implement. The algorithm is conceptually straightforward because it directly mirrors the long division process used in decimal arithmetic.

**Date: 12/08/24**