Experiment No. 9
Implement Non-restoring algorithm using c-programming
Name: Shravani Sandeep Raut
Roll Number: 48
Date of Performance:
Date of Submission:

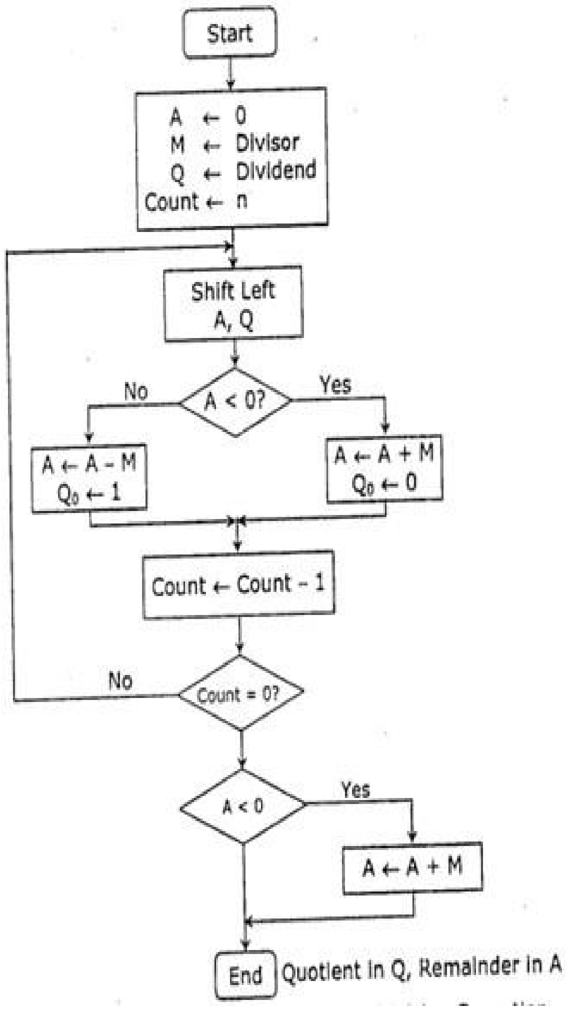
Aim - To implement Non-Restoring division algorithm using c-programming.

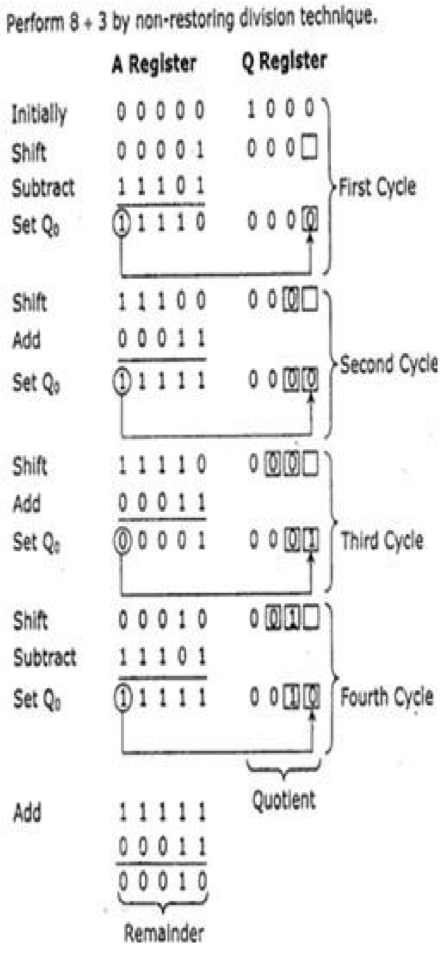
Objective -

1. To understand the working of Non-Restoring division algorithm.
2. To understand how to implement Non-Restoring division algorithm using c programming.

Theory:

In each cycle content of the register, A is first shifted and then the divisor is added or subtracted with the content of register A depending upon the sign of A. In this, there is no need of restoring, but if the remainder is negative then there is a need of restoring the remainder. This is the faster algorithm of division.




Perform 8 ÷ 3 by non-restoring division technique.

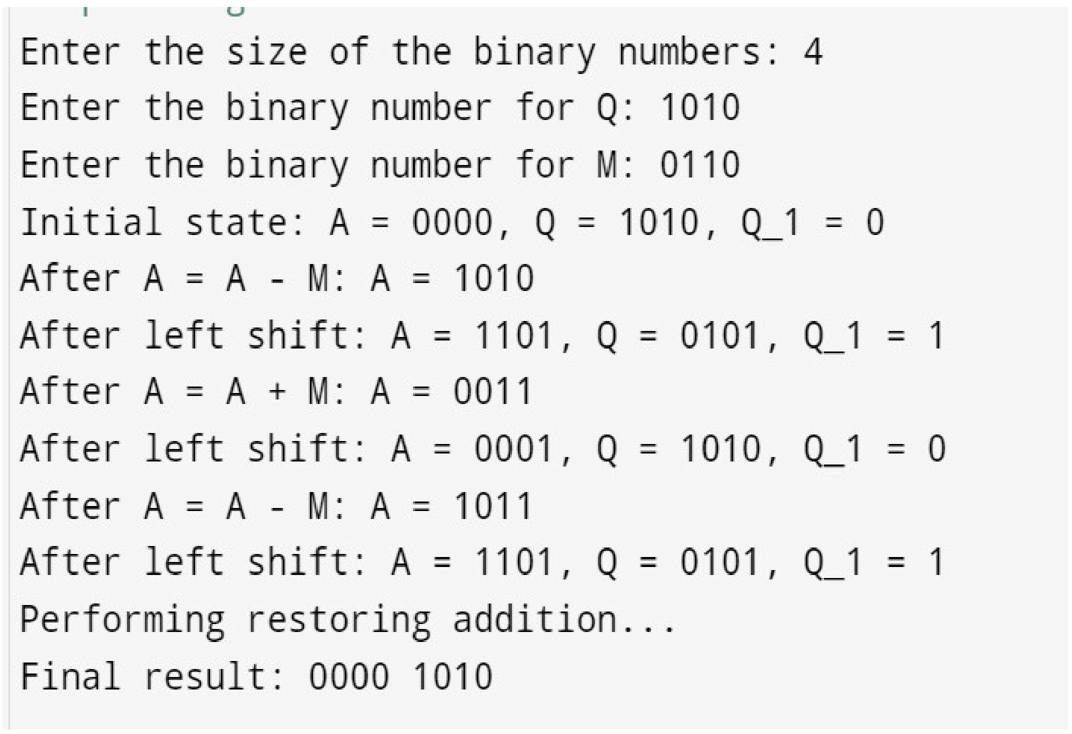
	A Register	Q Register	
Initially	0 0 0 0 0	1 0 0 0	First Cycle
Shift	0 0 0 0 1	0 0 0 $\boxed{}$	
Subtract	1 1 1 0 1		
Set Q_0	1 1 1 1 0	0 0 0 $\boxed{0}$	
Shift	1 1 1 0 0	0 0 $\boxed{0}$ $\boxed{}$	Second Cycle
Add	0 0 0 1 1		
Set Q_0	1 1 1 1 1	0 0 $\boxed{0}$ $\boxed{0}$	
Shift	1 1 1 1 0	0 $\boxed{0}$ $\boxed{0}$ $\boxed{}$	Third Cycle
Add	0 0 0 1 1		
Set Q_0	0 0 0 0 1	0 0 $\boxed{0}$ $\boxed{1}$	
Shift	0 0 0 1 0	0 $\boxed{0}$ $\boxed{1}$ $\boxed{}$	Fourth Cycle
Subtract	1 1 1 0 1		
Set Q_0	1 1 1 1 1	0 0 $\boxed{1}$ $\boxed{0}$	
		Quotient	
Add	1 1 1 1 1		
	0 0 0 1 1		
	0 0 0 1 0		
	Remainder		

Program –

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Function to perform left shift on the accumulator and Q register
void leftShift(int *accumulator, int *Q, int *Q_1, int size) {
    int i;
    // Shift the accumulator and Q register
    for (i = size - 1; i > 0; i--) {
        accumulator[i] = accumulator[i - 1];
        Q[i] = Q[i - 1];
    }
    // Shift the sign bit of accumulator to maintain its sign
    accumulator[0] = accumulator[1];
    // Q[0] gets the value of Q_1
    Q[0] = *Q_1;
}

// Function to perform addition on the accumulator
void add(int *accumulator, int *M, int size) {
    int carry = 0;
    int i;
    for (i = size - 1; i >= 0; i--) {
        int sum = accumulator[i] + M[i] + carry;
        accumulator[i] = sum % 2;
        carry = sum / 2;
    }
}

// Function to perform subtraction on the accumulator
void subtract(int *accumulator, int *M, int size) {
    int borrow = 0;
    int i;
    for (i = size - 1; i >= 0; i--) {
        int diff = accumulator[i] - M[i] - borrow;
        if (diff < 0) {
            diff += 2;
            borrow = 1;
        } else {
            borrow = 0;
        }
        accumulator[i] = diff;
    }
}

// Function to take two's complement of M (negate M)
void twosComplement(int *M, int size) {
    int i;
    // Flip all bits
    for (i = 0; i < size; i++) {
        M[i] = 1 - M[i];
    }
    // Add 1 to the result
    int carry = 1;
    for (i = size - 1; i >= 0 && carry; i--) {
        int sum = M[i] + carry;
        M[i] = sum % 2;
        carry = sum / 2;
    }
}

// Function to implement the non-restoring Booth's algorithm
void boothsAlgorithm(int *Q, int *M, int size) {
    int accumulator[size];
    int Q_1 = 0;
    int negM[size];
    int i;

    // Initialize accumulator to 0 and negM to -M (two's complement of M)
```

```
memset(accumulator, 0, sizeof(accumulator));
memcpy(negM, M, sizeof(int) * size);
twosComplement(negM, size);

printf("Initial state: A = ");
for (i = 0; i < size; i++) printf("%d", accumulator[i]);
printf(", Q = ");
for (i = 0; i < size; i++) printf("%d", Q[i]);
printf(", Q_1 = %d\n", Q_1);

for (i = 0; i < size; i++) {
   if (Q[0] == 0 && Q_1 == 1) {
      // A = A + M
      add(accumulator, M, size);
      printf("After A = A + M: A = ");
      for (int j = 0; j < size; j++) printf("%d", accumulator[j]);
      printf("\n");
   } else if (Q[0] == 1 && Q_1 == 0) {
      // A = A - M
      subtract(accumulator, M, size);
      printf("After A = A - M: A = ");
      for (int j = 0; j < size; j++) printf("%d", accumulator[j]);
      printf("\n");
   }

   // Perform left shift
   leftShift(accumulator, Q, &Q_1, size);
   printf("After left shift: A = ");
   for (int j = 0; j < size; j++) printf("%d", accumulator[j]);
   printf(", Q = ");
   for (int j = 0; j < size; j++) printf("%d", Q[j]);
   printf(", Q_1 = %d\n", Q_1);
}

// If the final result is negative (sign bit is 1), perform restoring addition
if (accumulator[0] == 1) {
        printf("Performing restoring addition...\n");
        add(accumulator, M, size);
    }

    // Display the final result
    printf("Final result: ");
    for (i = 0; i < size; i++) {
        printf("%d", accumulator[i]);
    }
    printf(" ");
    for (i = 0; i < size; i++) {
        printf("%d", Q[i]);
    }
    printf("\n");
}

void binaryStringToArray(char *binaryString, int *array, int size) {
    for (int i = 0; i < size; i++) {
        array[i] = binaryString[i] - '0';
    }
}

int main() {
    int size;
    printf("Enter the size of the binary numbers: ");
    scanf("%d", &size);

    char Q_str[size + 1], M_str[size + 1];
    int Q[size], M[size];

    // Input binary strings
    printf("Enter the binary number for Q: ");
    scanf("%s", Q_str);
    printf("Enter the binary number for M: ");
    scanf("%s", M_str);
    // Convert input strings to binary arrays
    binaryStringToArray(Q_str, Q, size);
    binaryStringToArray(M_str, M, size);

    // Run Booth's algorithm
    boothsAlgorithm(Q, M, size);

    return 0;
}
```

Output:



```
Enter the size of the binary numbers: 4
Enter the binary number for Q: 1010
Enter the binary number for M: 0110
Initial state: A = 0000, Q = 1010, Q_1 = 0
After A = A - M: A = 1010
After left shift: A = 1101, Q = 0101, Q_1 = 1
After A = A + M: A = 0011
After left shift: A = 0001, Q = 1010, Q_1 = 0
After A = A - M: A = 1011
After left shift: A = 1101, Q = 0101, Q_1 = 1
Performing restoring addition...
Final result: 0000 1010
```

Conclusion -

The Non-Restoring Division algorithm effectively divides two binary numbers using a series of shifts and conditional subtractions, avoiding restoring steps. Through its implementation in C programming, we gain insights into handling binary arithmetic, bitwise operations, and efficient division methods, enhancing our understanding of low-level computational techniques.