# **FR. Conceicao Rodrigues College of Engineering Department of Computer Engineering**

**8. Write a program to implement Restoring/Non Restoring Algorithm for Division.**

# **Course, Subject & Experiment Details**

| **Academic Year** | **2023-24** | **Estimated Time** | **Experiment No. 8– 02 Hours** |
| --- | --- | --- | --- |
| **Course & Semester** | **S.E. (Computers) – Sem. III** | **Subject Name** | **Digital Logic & Computer Organization and Architecture** |
| **Chapter No.** | **2** | **Chapter Title** | **Data Representation and Arithmetic algorithms** |
| **Experiment Type** | **Software** | **Subject Code** | **CSC304** |

**Rubrics**

| **Roll No** | Date of Performance  04/ 09/ 2023 | Timeline (2) | Practical Skill & Applied  Knowledge  **(4)** | Output  (4) | Total (10) |
| --- | --- | --- | --- | --- | --- |
| 9914 | Date of Submission:  09/ 09/ 2023 |  |  |  |  |

# **Aim & Objective of Experiment**

* + Understanding behaviour of Division algorithm for unsigned numbers
  + Implementing Restoring / Non-restoring Division algorithms.

# **Problem Statement**

Write a C/ Java / Python program to implement Restoring / Non restoring algorithm for Division.

# **Brief Theoretical Description**

Division operation implements as follows: it position the divisor appropriately with respect to the dividend and performs a subtraction If the reminder is Zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed. On the other hand, if the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

**Restoring Division**

An n-bit positive divisor is loaded into register M and an n-bit positive dividend is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n-bit quotients in register Q and the remainder is in register A. the required subtraction are facilitated by using 2’s complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.

**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 is the result is negative. Consider the sequence of operation that takes place after the subtraction operation in the preceding algorithm. If A is positive, we shift left and subtract M, that is, we perform 2A-M. If A is negative, we restore it by performing A+M, and then we shift it left and subtract M. This is equivalent to performing 2A+M. the q0 bit is appropriately set 0 or 1 after the correct operation has been performed.

## **Algorithm:**

**Restoring Division**

Do the following n times:

1. Shift A and Q left one binary position.
2. Subtract M from A, and place the answer back in A.
3. If the sign of A is 1, set q0 to 0 add M back to A (that is restore A); otherwise, set q0 to 1.

## **Non-restoring Division**

**Step 1:** do the following n times:

* 1. 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.
  2. Now, if the sign of A is 0, set q0 to 1; otherwise, set q0 to 0.

**Step 2:** if the sign of A is 1, add M to A

1. ***Attach the program***

**Restoring Division Code:**

#include <stdlib.h>

#include <stdio.h>

*// Function to perform binary addition of two arrays.*

void add(int *accumulator*[], int *divisorBin*[], int *n*)

{

int i = 0, temp = 0, sum = 0;

for (i = 0; i < *n*; i++)

{

sum = 0;

sum = *accumulator*[i] + *divisorBin*[i] + temp;

*// Determine the result of binary addition and handle carry.*

if (sum == 0)

{

*accumulator*[i] = 0;

temp = 0;

}

else if (sum == 2)

{

*accumulator*[i] = 0;

temp = 1;

}

else if (sum == 1)

{

*accumulator*[i] = 1;

temp = 0;

}

else if (sum == 3)

{

*accumulator*[i] = 1;

temp = 1;

}

}

}

int main()

{

int x, y;

*// Define arrays to hold accumulator, quotient, and divisor.*

int accumulator[100] = {0};

int quotient[100], divisor[100];

*// Prompt the user to enter the dividend and divisor in decimal.*

printf("Enter the Dividend in decimal :");

scanf("%d", &x);

printf("Enter the Divisor in decimal:");

scanf("%d", &y);

int i = 0;

*// Convert decimal inputs to binary and store them in arrays.*

while (x > 0 || y > 0)

{

if (x > 0)

{

quotient[i] = x % 2; *// Store binary digit of quotient.*

x = x / 2;

}

else

{

quotient[i] = 0; *// Fill with zero if dividend is exhausted.*

}

if (y > 0)

{

divisor[i] = y % 2; *// Store binary digit of divisor.*

y = y / 2;

}

else

{

divisor[i] = 0; *// Fill with zero if divisor is exhausted.*

}

i++;

}

int n = i;

int divisor2Com[50];

*// Calculate the two's complement of the divisor.*

for (i = 0; i < n; i++)

{

if (divisor[i] == 0)

{

divisor2Com[i] = 1; *// Flip bits for two's complement.*

}

else

{

divisor2Com[i] = 0;

}

}

divisor2Com[n] = 1;

*// Invert the divisor2Com array.*

for (i = 0; i <= n; i++)

{

if (divisor2Com[i] == 0)

{

divisor2Com[i] = 1;

i = n + 2;

}

else

{

divisor2Com[i] = 0;

}

}

int l;

divisor[n] = 0;

int k = n;

int n1 = n + n - 1;

int j, mi = n - 1;

*// Perform division using the restoring algorithm.*

for (i = n; i != 0; i--)

{

for (j = n; j > 0; j--)

{

accumulator[j] = accumulator[j - 1]; *// Shift accumulator and quotient.*

}

accumulator[0] = quotient[n - 1]; *// Assign MSB of quotient to accumulator.*

for (j = n - 1; j > 0; j--)

{

quotient[j] = quotient[j - 1]; *// Shift quotient.*

}

*// Perform binary addition and check the sign of the accumulator.*

add(accumulator, divisor2Com, n + 1);

if (accumulator[n] == 1)

{

quotient[0] = 0;

add(accumulator, divisor, n + 1);

}

else

{

quotient[0] = 1;

}

}

*// Print the quotient and remainder.*

printf("\nQuotient : ");

for (l = n - 1; l >= 0; l--)

{

printf("%d", quotient[l]); *// Output binary quotient.*

}

printf("\nRemainder : ");

for (l = n; l >= 0; l--)

{

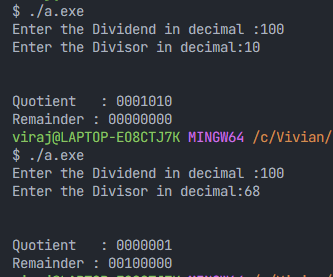
printf("%d", accumulator[l]); *// Output binary remainder.*

}

return 0;

}

**OUTPUT:**

****

**Non Restoring Division Code:**

#include <stdio.h>

#include <stdlib.h>

*// Define an array to store the binary representation of the dividend and divisor.*

int accumulator[100] = {0};

int quotient[100], divisor[100], l;

*// Function to perform binary addition of accumulator and divisorBin.*

void add(int *accumulator*[], int *divisorBin*[], int *n*)

{

int i = 0, temp = 0, sum = 0;

for (i = 0; i < *n*; i++)

{

sum = 0;

sum = *accumulator*[i] + *divisorBin*[i] + temp;

*// Handle different cases of binary addition.*

if (sum == 0)

{

*accumulator*[i] = 0;

temp = 0;

}

else if (sum == 2)

{

*accumulator*[i] = 0;

temp = 1;

}

else if (sum == 1)

{

*accumulator*[i] = 1;

temp = 0;

}

else if (sum == 3)

{

*accumulator*[i] = 1;

temp = 1;

}

}

}

int main()

{

int x, y;

*// Prompt the user to enter the dividend and divisor in decimal.*

printf("Enter the Dividend in decimal :");

scanf("%d", &x);

printf("Enter the Divisor in decimal:");

scanf("%d", &y);

int i = 0;

*// Convert decimal inputs to binary and store them in arrays.*

while (x > 0 || y > 0)

{

if (x > 0)

{

quotient[i] = x % 2;

x = x / 2;

}

else

{

quotient[i] = 0;

}

if (y > 0)

{

divisor[i] = y % 2;

y = y / 2;

}

else

{

divisor[i] = 0;

}

i++;

}

int n = i;

*// Compute the two's complement of the divisor.*

int divisor2Com[50];

printf("\n");

for (i = 0; i < n; i++)

{

if (divisor[i] == 0)

{

divisor2Com[i] = 1;

}

else

{

divisor2Com[i] = 0;

}

}

divisor2Com[n] = 1;

*// Invert the bits of the two's complement to get the negative value.*

for (i = 0; i <= n; i++)

{

if (divisor2Com[i] == 0)

{

divisor2Com[i] = 1;

i = n + 2;

}

else

{

divisor2Com[i] = 0;

}

}

divisor[n] = 0;

int j;

*// Perform binary division using the accumulator, quotient, and divisor.*

for (i = n; i != 0; i--)

{

if (accumulator[n] == 0)

{

for (j = n; j > 0; j--)

{

accumulator[j] = accumulator[j - 1];

}

accumulator[0] = quotient[n - 1];

for (j = n - 1; j > 0; j--)

{

quotient[j] = quotient[j - 1];

}

add(accumulator, divisor2Com, n + 1);

}

else

{

for (j = n; j > 0; j--)

{

accumulator[j] = accumulator[j - 1];

}

accumulator[0] = quotient[n - 1];

for (j = n - 1; j > 0; j--)

{

quotient[j] = quotient[j - 1];

}

add(accumulator, divisor, n + 1);

}

if (accumulator[n] == 1)

{

quotient[0] = 0;

}

else

{

quotient[0] = 1;

}

}

*// Handle the case of a non-zero remainder.*

if (accumulator[n] == 1)

{

add(accumulator, divisor, n + 1);

}

*// Print the quotient and remainder.*

printf("\nQuotient : ");

for (l = n - 1; l >= 0; l--)

{

printf("%d", quotient[l]);

}

printf("\nRemainder : ");

for (l = n; l >= 0; l--)

{

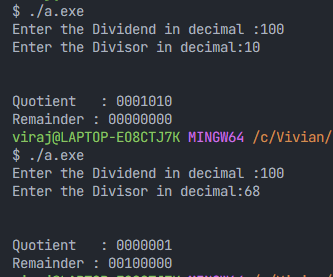
printf("%d", accumulator[l]);

}

return 0;

}

**OUTPUT:**

****

# **Conclusion:**

We were able to implement the restoring and non-restoring division algorithms for unsigned integers using C. The results showed that the non-restoring division algorithm is faster and simpler than the restoring division algorithm, as it avoids the need for restoring the value of register A after each subtraction.