**National Institute of Technology Silchar**
**NITS**

**Department of Computer Science and Engineering**
**July,2020**

**CS-205**
**Computer Architecture and Organization**

---

# Project Topic:

**Simulating and Study of Restoring Division Algorithm for signed and unsigned integers**

---

**Submitted By:**

- Tanveer Ahmed 18-1-5-022
- Pranjit Gautam 18-1-5-023
- Nirmita Biswas 18-1-5-024
- Manditjyoti Borah 18-1-5-025
- Mayur Raj Bharati 18-1-5-026

**Under The Supervision of:**

Dr. Malaya Dutta Borah

COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY SILCHAR

# CERTIFICATE

It is certified that the work contained in this thesis entitled Simulating and Study of Restoring Division Algorithm for signed and unsigned integers submitted by **Tanveer Ahmed(18- 1-5-022), Pranjit Gautam (18-1-5-023),Nirmita Biswas (18-1-5-024), Manditjyoti Borah (18-1-5-025) ,Mayur Raj Bharati (18-1-5-026)** for the B.Tech. End Semester Project for fourth semester. Examination July, 2020 is absolutely based on their own work carried out under my supervision.

Place : NIT Silchar
Date: 12/07/2020

Dr. Malaya Dutta Borah
Computer Science And Engineering
National Institute of Technology Silchar

COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY SILCHAR

# DECLARATION

Mini Project Title: **Simulating and Study of Restoring Division Algorithm for signed and unsigned integers.**Degree for which the Mini Project is submitted: **Bachelor of Tech- nology.**

We declare that the presented thesis represents largely our own ideas and work in our own words. Where others ideas or words have been included, we have adequately cited and listed in the reference materials. The thesis has been prepared without resorting to plagiarism.We have adhered to all principles of academic honesty and integrity.No falsified or fabricated data have been presented in the thesis. We understand that any violation of the above will cause for disciplinary action by the Institute, including revoking the conferred degree, if conferred, and can also evoke penal action from the sources which have not been properly cited or from whom proper permission has not been taken.

- Tanveer Ahmed 18-1-5-022
- Pranjit Gautam 18-1-5-023
- Nirmita Biswas 18-1-5-024
- Manditjyoti Borah 18-1-5-025
- Mayur Raj Bharati 18-1-5-026

COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY SILCHAR

---

## ABSTRACT

**Computer Architecture** is concerned with balancing the performance, efficiency, cost, and reliability of a computer system. It refers to those attributes of a system that have a direct impact on the logical execution of a program such as the instruction set, I/O mechanisms etc. The study of **Computer Organization** helps optimize performance-based products. It also helps plan the selection of a processor for a particular project.

A basic computer consists of three important components- Processor, Memory and Input/Output Devices. Further the processor consists of the Arithmetic and Logic Unit (ALU) and the Control Unit (CU). The ALU performs various mathematical and logical operations of the operands of the decoded instructions. The ALU makes use of various algorithms to perform various arithmetic as well as logical operations.

There are number of binary division algorithm like Digit Recurrence Algorithm restoring, non-restoring and SRT Division (Sweeney, Robertson, and Tocher), Multiplicative Algorithm, Approximation Algorithms, CORDIC Algorithm and Continued Product Algorithm. Among them, the **Restoring Division Algorithm** is of great importance. This algorithm makes the division of signed as well as unsigned integers in a fast and efficient manner using only a few processor registers.

This project report details a study of the Restoring Division Algorithm for Signed integers. Our project is solely not only about the study of the algorithm and the background theory required for the study such as Representation of Signed integers and SB element but also implements the working of the algorithm in a simulator and simulates the step by step approach using a python program. We will present the study of various concepts, along with various numerical examples as well as we will design the circuit required for the algorithm and simulate the circuit using a Verilog program.

# Contents

**1 Introduction** **2**

1.1 Restoring Division . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**2 Background Theory** **4**

2.1 Signed and Unsigned Integers . . . . . . . . . . . . . . . . . . . . . . . 4

2.1.1 Representation of Binary Numbers: . . . . . . . . . . . . . . . . 4

2.1.2 Unsigned Numbers: . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1.3 Representation of Unsigned Binary Numbers: . . . . . . . . . . . 5

2.1.4 Signed Numbers: . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1.5 Representation of Signed Binary Numbers: . . . . . . . . . . . . 5

2.2 SB Circuit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**3 Restoring Division Algorithm:** **7**

**4 Examples Demonstrated for the Restoring Division Algorithm** **10**

**5 Circuit Diagram** **13**

**6 Appendix** **14**

**7 Restoring Division Algorithm Implementation** **17**

**8 Website Designed for Performing Restoring Division Algorithm on Signed Integers** **23**

**9 Conclusion** **25**

**10 References:** **25**

**COMPUTER SCIENCE AND ENGINEERING**
**NATIONAL INSTITUTE OF TECHNOLOGY SILCHAR**

# 1 Introduction

The processor is the main substance of a computer system that executes instructions that make up a computer program. The Arithmetic and Logic Unit (ALU) is an integral part of the processor.

The ALU is the mathematical brain of the computer. It is a combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers. It is a fundamental building block of many types of computing circuits. It performs various arithmetic and logical operations on the operands of the decoded instructions. It uses various adder circuits to do addition, subtraction and other arithmetic operations.

Various algorithms are used to perform different logical as well as arithmetic operations in the ALU. For the division operation, one can use the **Restoring Division** algorithm as it is simple and is lower in complexity than division by convergence. This algorithm can be used for both signed as well as unsigned integers. This algorithm makes use of three registers, namely Accumulator (A), Register Q to store the dividend and Register M to store the divisor. After the division process is completed, A contains the Remainder and Q contains the Quotient. In this algorithm, if a certain step is unsuccessful, we restore the value in Accumulator. Hence this algorithm is known as Restoring division algorithm.

## 1.1 Restoring Division

We assume that both the dividend and divisor are positive and hence the quotient and the remainder are positive or zero.
Figure 1 shows the circuit which implements the division algorithm. In each step of the algorithm, the divisor is shifted one position to the right, and the quotient is shifted one position to the left.

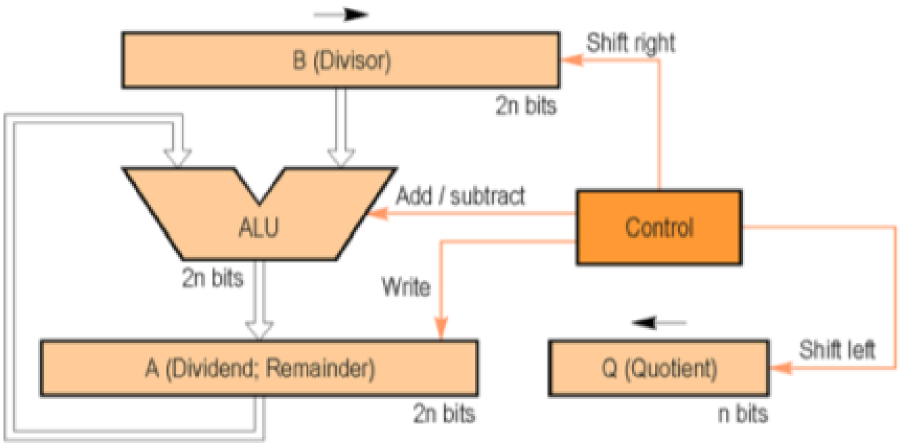


Figura 1: First Version of the Divider Circuit

Figure 3.21 shows the steps of the first version of the division algorithm. Initially, the dividend is loaded into the right half of the $2n$ -bit $A$ register, and the divisor is loaded into the left half of the $2n-$ bit $B$ register. The $n$ -bit quotient register

$(Q)$ is set to 0, and the counter $N$ is set to $n+1$. In order to determine whether the divisor is smaller than the partial remainder, the divisor register $(B)$ is subtracted from the remainder register $(A)$. If the result is negative, the next step is to restore the previous value by adding the divisor back to the remainder, generating a 0 in the $Q_0$ position of the quotient register. This is the reason why this method is called restoring dinision. If the result is positive, a 1 is generated in the $Q_0$ position of the quotient register. In the next step, the divisor is shifted to the right, aligning the divisor with the dividend for the next iteration.

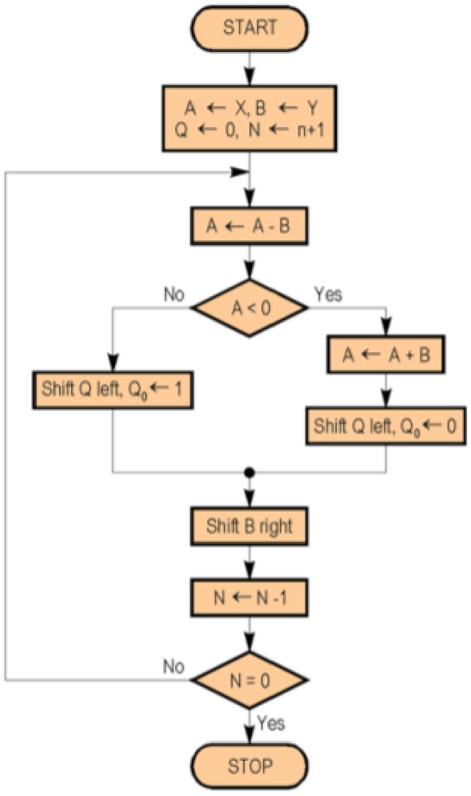


Figura 2: The first version of the restoring division algorithm.

# 2 Background Theory

## 2.1 Signed and Unsigned Integers

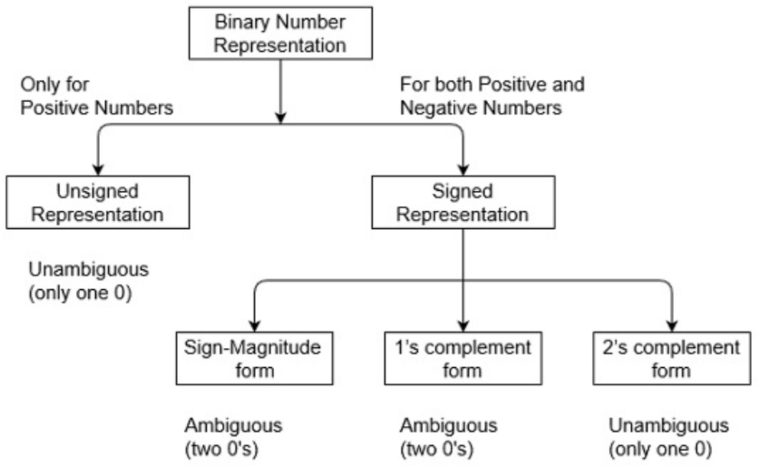
Variables such as integers can be represent in two ways, i.e., signed and unsigned. Signed numbers use sign flag or can be distinguish between negative values and positive values. Whereas unsigned numbers stored only positive numbers but not negative numbers.

Number representation techniques like: Binary, Octal, Decimal and Hexadecimal number representation techniques can represent numbers in both signed and unsigned ways. Binary Number System is one the type of Number Representation techniques. It is most popular and used in digital systems. Binary system is used for representing binary quantities which can be represented by any device that has only two operating states or possible conditions. For example, a switch has only two states: open or close.

In the Binary System, there are only two symbols or possible digit values, i.e., 0 and 1. Represented by any device that only 2 operating states or possible conditions. Binary numbers are indicated by the addition of either an 0b prefix or an 2 suffix.

### 2.1.1 Representation of Binary Numbers:

Binary numbers can be represented in signed and unsigned way. Unsigned binary numbers do not have sign bit, whereas signed binary numbers uses signed bit as well or these can be distinguishable between positive and negative numbers. A signed binary is a specific data type of a signed variable.



### 2.1.2 Unsigned Numbers:

Unsigned numbers don't have any sign, these can contain only magnitude of the number. So, representation of unsigned binary numbers are all positive numbers only. For example, representation of positive decimal numbers are positive by default. We always assume that there is a positive sign symbol in front of every number.

### 2.1.3 Representation of Unsigned Binary Numbers:

Since there is no sign bit in this unsigned binary number, so N bit binary number represent its magnitude only. Zero (0) is also unsigned number. This representation has only one zero (0), which is always positive. Every number in unsigned number representation has only one unique binary equivalent form, so this is unambiguous representation technique. The range of unsigned binary number is from 0 to $(2^n - 1)$

**Example-1**: To represent decimal number 92 in unsigned binary number. We simply convert it into Binary number, it contains only magnitude of the given number.
$= (92)_{10}$
$= (1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0)_{10}$
$= (1011100)_2$
It's 7 bit binary magnitude of the decimal number 92 .

### 2.1.4 Signed Numbers:

Signed numbers contain sign flag. this representation distinguish positive and negative numbers. This technique contains both sign bit and magnitude of a number. For example. in representation of negative decimal numbers, we need to put negative symbol in front of given decimal number.

### 2.1.5 Representation of Signed Binary Numbers:

There are three types of representations for signed binary numbers. Because of extra signed bit. binary number zero has two representation. either positive (0) or negative (1). so ambiguous representation. But 2 's complementation representation is unambiguous representation because of there is no double representation of number 0 . These are: Sign-Magnitude form, 1 's complement form, and 2 's complement form which are explained as following below,

Practically one always deals with signed integers. The best way to represent a negative integer is the two's-complement representation.

In two's-complement, negative numbers are represented by the bit pattern which is one greater than the one's-complement of the positive number. This representation is considered the best as this representation gives a unique representation of 0.

Now whenever we observe that the most significant bit (MSB) of a number is 0, we can directly say that the number is positive and can get the number simply by converting it to its decimal form. But when we observe that the MSB of a number is 1, we have to understand that the number is negative and is stored in its two's-complement form.

Accordingly we find the two's-complement of that number again (as two's-complement of a two's-complement gives the actual number back) to get the actual number back.

There is an easier way to get the two's-complement of a number as follows:
a. Starting from the LSB, find the first '1'.
b. Invert all the bits to the left of that '1'.

Example: If a number is 1001, it means the number is negative and the actual number is the two's-complement of 1001, i.e.,

1001 = 0111 which is -7 in decimal

**2.1.5(a) Sign-Magnitude form:** For $n$ bit binary number, 1 bit is reserved for sign symbol, If the value of sign bit is 0 . then the given number will be positive. else if the value of sign bit is 1, then the given number will be negative. Remsining (n-1) bits represent magnitude of the number. Since magnitude of number zero (0) is always 0, so there osn be two representation of number zero (0). positive (+0) and negative (-0). which depends on value of sign bit. Hence these representations are ambiguous generally because of two representation of number zero (0). Generally sign bit is a most significant bit (MSB) of representation.

For example. range of 6 bit Sign-Magnitude form binary number is from $(2^5 - 1)$ to $(2^5 - 1)$ which is equal from minimum value -31 (i.e., 1 11111) to maximum value +31 (i.e., 011111 ). And zero (0) has two representation, -0 (i.e., 100000 ) and +0 (i.e., 0.00000 ).

**2.1.5(b) 1's complement form:** Since, 1 's complement of a number is obtained by inverting each bit of given number. So, we represent positive numbers in binary form and negative numbers in 1 's complement form. There is extra bit for sign representation, If value of sign bit is 0 , then number is positive and you can directly represent it in simple binary form, but if value of sign bit 1 , then number is negative and you have to take 1 "s complement of given binary number. You can get negative number by 1 "s complement of a positive number and positive number by using 1 's complement of a negative number. Therefore, in this representation, zero (0) ean have two representation, thats why 1 's complement form is also ambiguous form.

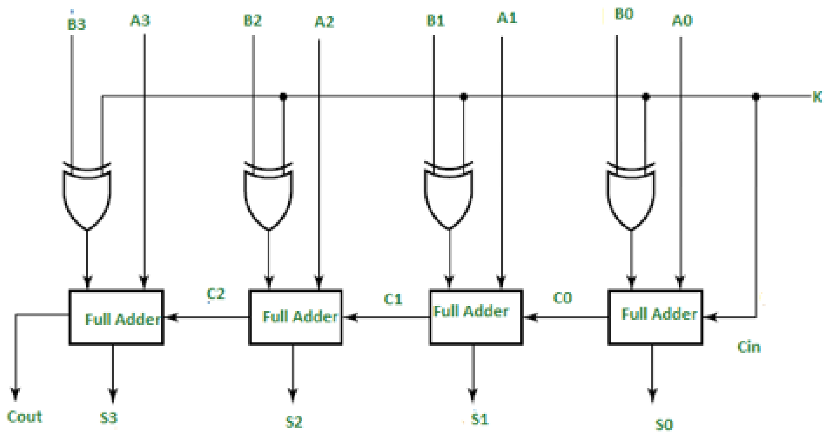
For example, range of 6 bit 1 's complement form binary number is from $(2^5 - 1)$ to $(2^5 - 1)$ which is equal from minimum value -31 (i.e. 100000 ) to maximum value +31 (i.e., 011111 ). And zero (0) has two representation, -0 (i.e., 111111 ) and +0 (i.e., 000000 ).

**2.1.5(c) 2 's complement form:** Since. 2 : s complement of a number is obtained by inverting each bit of given number plus 1 to least significant bit (LSB). So, we represent positive numbers in binary form and negative numbers in 2 's complement form. There is extra bit for sign representation. If value of sign bit is 0. then number is positive and you can directly represent it in simple binary form, but if value of sign bit 1 . then number is negative and you have to take 2 's complement of given binary number. You can get negative number by 2 's complement of a positive number and positive number by directly using simple binary representation. If value of most significant bit (MSB) is 1, then take 2 s complement from, else not. Therefore, in this representation, zero (0) has only one (unique) representation which is always positive. The range of 2 's complement form is from $\left(2^{(n-1)}\right)$ to $\left(2^{(n-1)}-1\right)$.

For example. range of 6 bit 2 's complement form binary number is from $(2^5)$ to $(2^5-1)$ which is equal from minimum value -32 (i.e., 100000 ) to maximum value +31 (i.e., 011111 ). And zero (0) has two representation, -0 (i.e. 111111 ) and +0 (i.e. 000000 )

## 2.2 SB Circuit

The SB circuit or the adder/subtractor circuit is a circuit that is capable of adding or subtracting numbers. It does adding or subtracting depending on a control signal. When the control signal is 0, it behaves as an adder circuit and when the control signal is 1, it behaves as a subtractor circuit. Given below is a circuit which performs 4-bit addition/subtraction of two 4-bit numbers:



As we can see, when the control signal K is 0, the input to the XOR gates becomes 0 and input bits B's, so the outcome of the XOR gates becomes B's only and hence it performs addition of the bits. However, if the control signal K is 1, the input to the XOR gates becomes 1 and the input bits B's, this however inverts the bits B's thereby providing with a two's-complement to the adder circuit, hence performing subtraction of the bits.

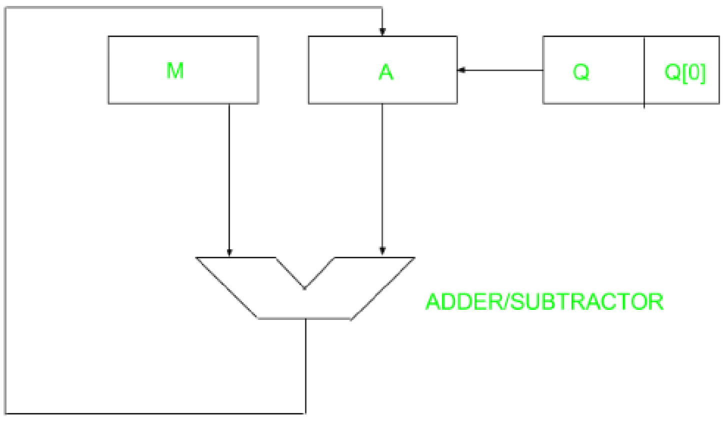
| Input 1 | Input 2 | Output |
|---|---|---|
| A | 0 | A |
| A | 1 | $A^C$ |

Figura 3: Truth Table of XOR Gate.

# 3 Restoring Division Algorithm:

In this algorithm, the dividend is store in the register Q and the divisor is stored in the register M. Further one more register, Accumulator (A) is also used in this algorithm. The initial contents of A is the signed extension of Q. For example if the number of bits in dividend is 4 and dividend is positive then $A$ will be $(0000)_2$ and if dividend is negative, A will be $(1111)_2$ . Upon completion of this algorithm, Q will get the quotient and A will get the remainder.

A division algorithm provides a quotient and a remainder when we divide two number. They are generally of two type slow algorithm and fast algorithm. Slow division algorithm are restoring, non-restoring, non-performing restoring, SRT algorithm and under fast comes Newton–Raphson and Goldschmidt.



Here, register Q contain quotient and register A contain remainder. Here, n-bit dividend is loaded in Q and divisor is loaded in M. Value of Register is initially kept 0 and this is the register whose value is restored during iteration due to which it is named Restoring.

Steps used to carry out the algorithm:

1. The number of steps required is equal to the number of bits in the dividend.

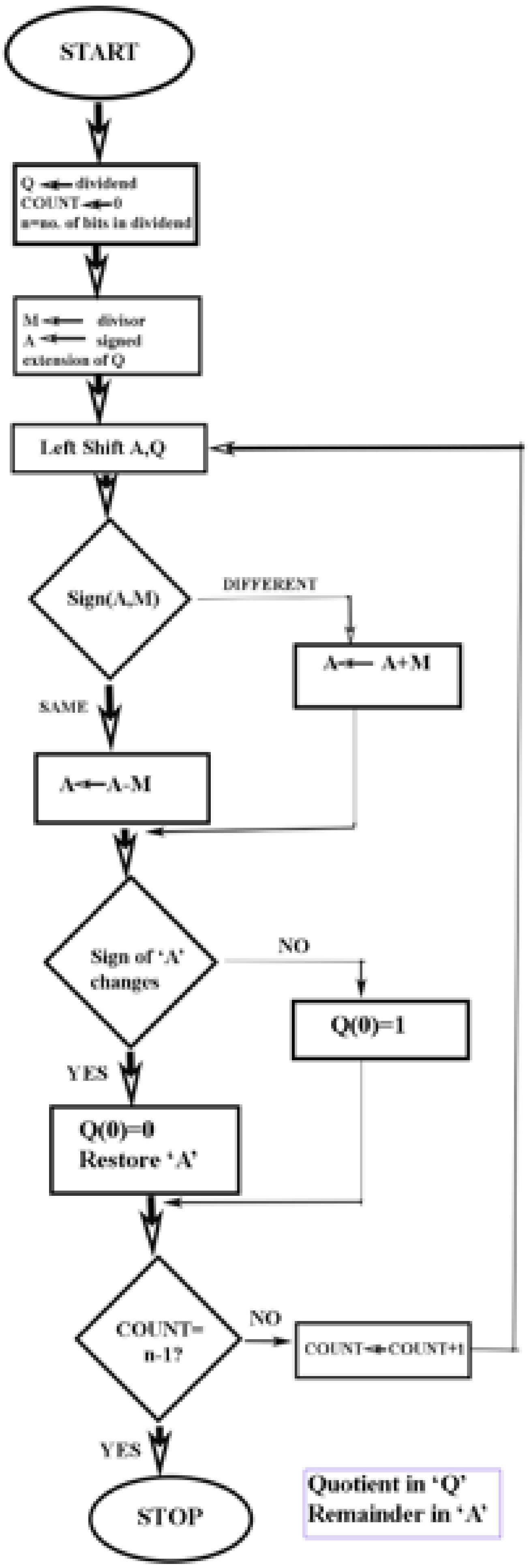
2. At each step, left shift $A$, Q by one position.

3. If sign of $A$ and $M$ is the same, then subtract $M$ from $A$. If however sign of $A$ and M is different, add M to A.

4. After the operation, if the sign of A remains the same or A becomes 0, then the step is said to be successful and quotient bit 1 will be added and restoration is not required.

If however, sign of A changes, then the step is said to be unsuccessful and the quotient bit 0 will be added and restoration will be done. since restoration is done when the step is unsuccessful, hence this method is called Restoring Division.

5. Repeat the steps 2 to 4 for every bit of the dividend.

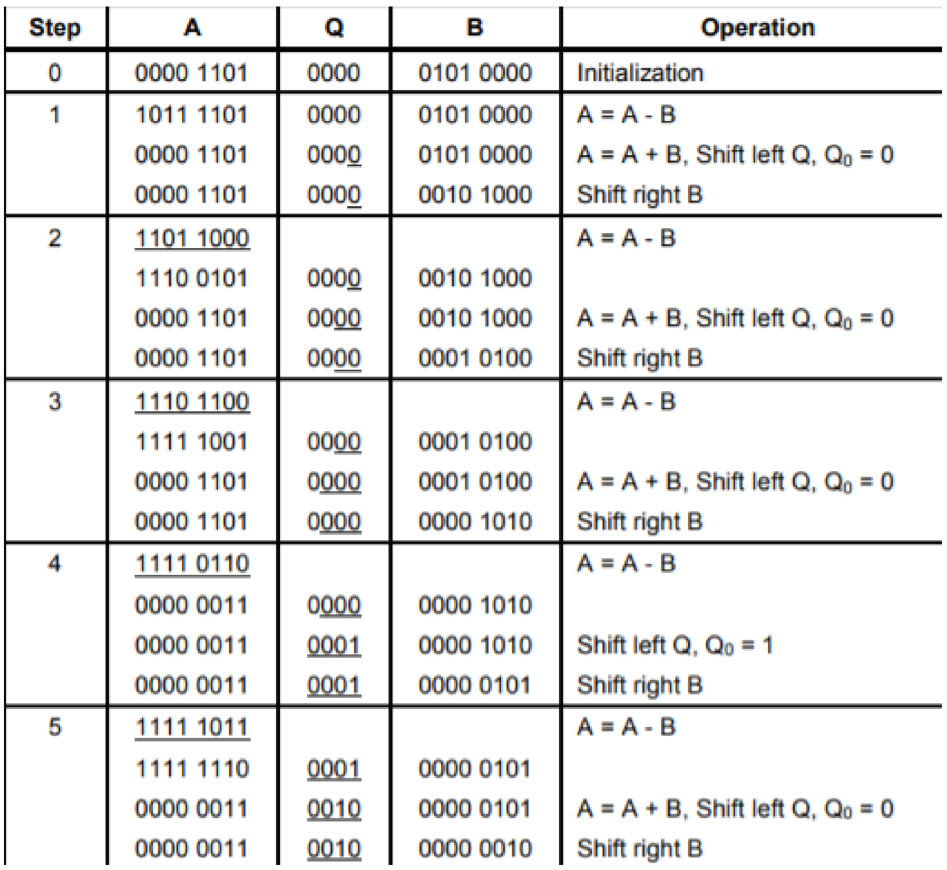

START
Q ⟵ dividend
COUNT ⟵ 0
n=no. of bits in dividend
M ⟵ divisor
A ⟵ signed extension of Q
Left Shift A,Q
Sign(A,M)
DIFFERENT
A ⟵ A+M
SAME
A ⟵ A-M
Sign of 'A' changes
NO
Q(0)=1
YES
Q(0)=0
Restore 'A'
COUNT= n-1?
NO
COUNT ⟵ COUNT+1
YES
STOP
Quotient in 'Q'
Remainder in 'A'

## 4 Examples Demonstrated for the Restoring Division Algorithm

**Example 1:**
Using the first version of the restoring division algorithm, we demonstrate the algorithm by dividing the 4-bit numbers $X = 13$ and $Y = 5\,(1101_2 \div 0101_2)$

**Implementation:** Table 1 shows the contents of the registers in each step of the operation, finally obtaining a quotient of 2 and a remainder of 3.

Table 1 Division example using the first version of the algorithm.



| Step | A | Q | B | Operation |
|---|---|---|---|---|
| 0 | 0000 1101 | 0000 | 0101 0000 | Initialization |
| 1 | 1011 1101 | 0000 | 0101 0000 | A = A - B |
| | 0000 1101 | 0000 | 0101 0000 | A = A + B, Shift left Q, $Q_0$ = 0 |
| | 0000 1101 | 0000 | 0010 1000 | Shift right B |
| 2 | 1101 1000 | | | A = A - B |
| | 1110 0101 | 0000 | 0010 1000 | |
| | 0000 1101 | 0000 | 0010 1000 | A = A + B, Shift left Q, $Q_0$ = 0 |
| | 0000 1101 | 0000 | 0001 0100 | Shift right B |
| 3 | 1110 1100 | | | A = A - B |
| | 1111 1001 | 0000 | 0001 0100 | |
| | 0000 1101 | 0000 | 0001 0100 | A = A + B, Shift left Q, $Q_0$ = 0 |
| | 0000 1101 | 0000 | 0000 1010 | Shift right B |
| 4 | 1111 0110 | | | A = A - B |
| | 0000 0011 | 0000 | 0000 1010 | |
| | 0000 0011 | 0001 | 0000 1010 | Shift left Q, $Q_0$ = 1 |
| | 0000 0011 | 0001 | 0000 0101 | Shift right B |
| 5 | 1111 1011 | | | A = A - B |
| | 1111 1110 | 0001 | 0000 0101 | |
| | 0000 0011 | 0010 | 0000 0101 | A = A + B, Shift left Q, $Q_0$ = 0 |
| | 0000 0011 | 0010 | 0000 0010 | Shift right B |

Shifting the partial remainder to the left instead of shifting the divisor to the right produces the same alignment and simplifies the hardware necessary for the ALU and the divisor register. Both the divisor register and the ALU could have the size reduced to half ($n$ bits instead of $2n$ bits).

The second improvement comes from the fact that the first step of the algorithm cannot generate a digit of 1 in the quotient, because, in this case, the quotient would be too large for its register. By switching the order of the operations to shift and then to subtract, one iteration of the algorithm can be removed.

Another observation is that the size of the A register could be reduced to half, and the A and Q registers could be combined, shifting the bits of the quotient into the A register instead of shifting in zeros as in the preceding algorithm. The A and Q registers are shifted left together. Figure 3 shows the final version of the restoring division algorithm.

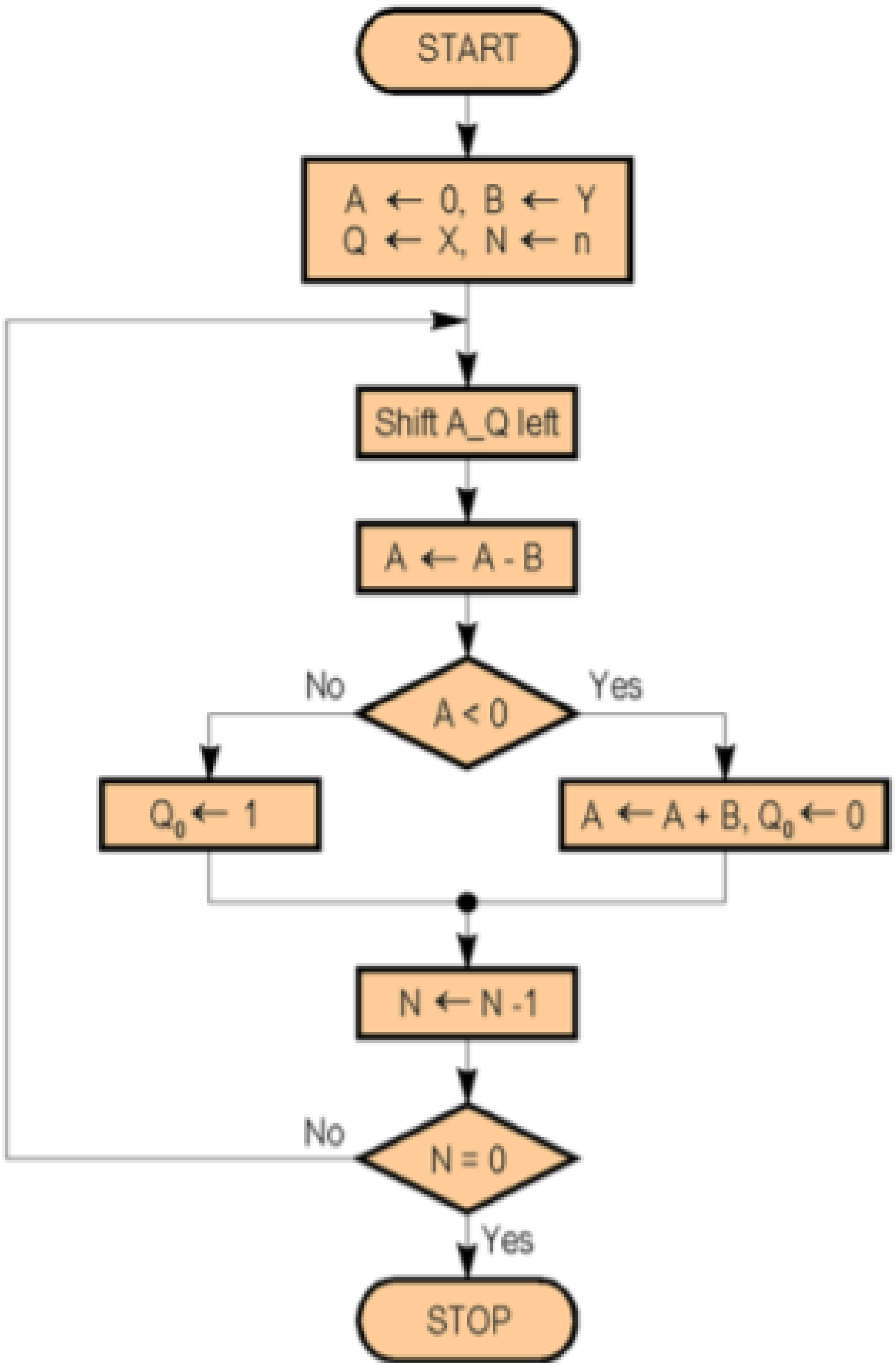


Figura 4: Final version of the restoring division algorithm.

The final version of the restoring divider circuit is shown in Figure 4

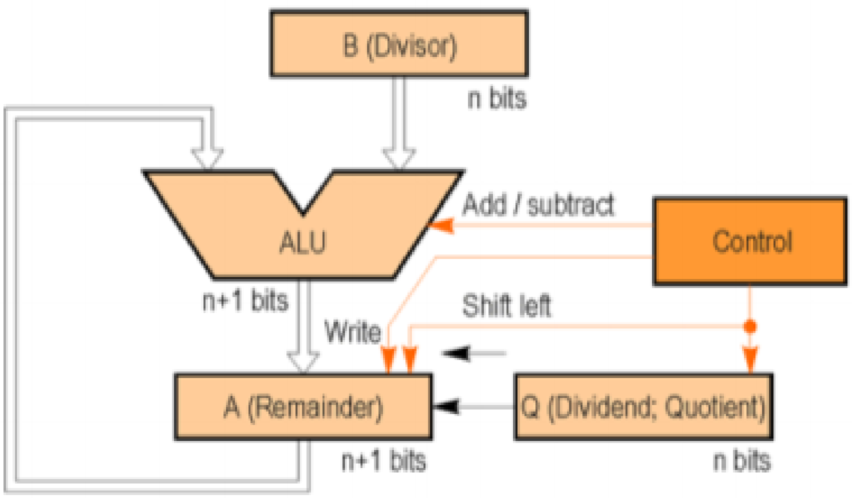


Figura 5: Final version of the restoring divider circuit.

**Example 2:**
Using the final version of the algorithm, we divide X = 13 by Y = 5

Table 2 shows the contents of the registers in each step of the operation. The remainder is formed in the A register, and the quotient in the Q register.

Table 2. Division example using the final version of the restoring division algorithm.

| Step | A | Q | B | Operation |
|---|---|---|---|---|
| 0 | 0 0000 | 1101 | 0101 | Initialization |
| 1 | 0 0001<br>1 1011<br>1 1100<br>0 0101<br>0 0001 | 1010<br><br><br><br>1010 | <br><br><br><br>0101 | Shift left A_Q<br>A = A - B<br><br>A = A + B, $Q_0$ = 0 |
| 2 | 0 0011<br>1 1011<br>1 1110<br>0 0101<br>0 0011 | 0100<br><br><br><br>0100 | <br><br><br><br>0101 | Shift left A_Q<br>A = A - B<br><br>A = A + B, $Q_0$ = 0 |
| 3 | 0 0110<br>1 1011<br>0 0001 | 1000<br><br>1001 | <br><br>0101 | Shift left A_Q<br>A = A - B<br>$Q_0$ = 1 |
| 4 | 0 0011<br>1 1011<br>1 1110<br>0 0101<br>0 0011 | 0010<br><br><br><br>0010 | <br><br><br><br>0101 | Shift left A_Q<br>A = A - B<br><br>A = A + B, $Q_0$ = 0 |

**Example 3:**
We try to implement the Restoring Division Algorithm on (-7)/(-4).

$$7 = 0111 \quad 4 = 0100$$
$$-7 = 1001 \quad -4 = 1100$$

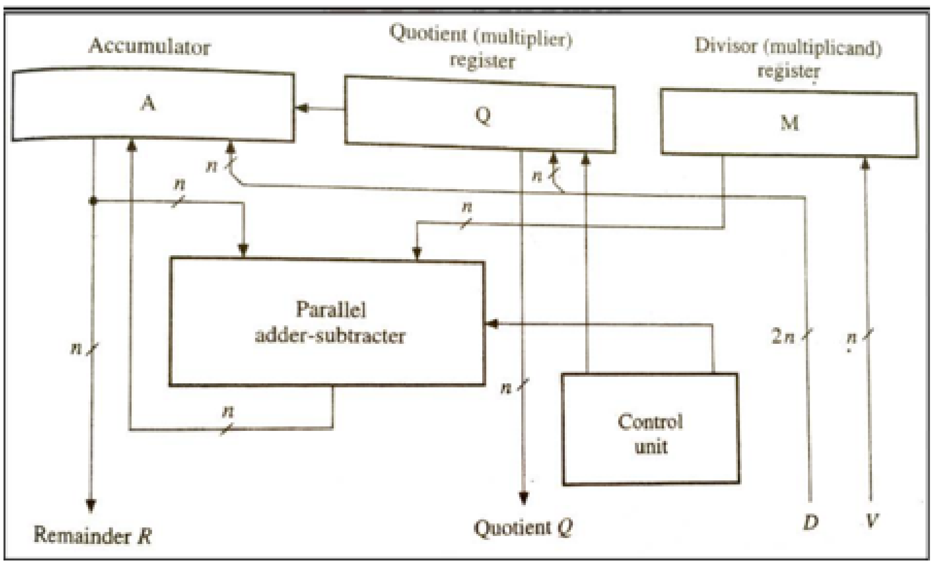
Here dividend is -7 which is negative. Hence initially value of A will be $(1111)_2$.

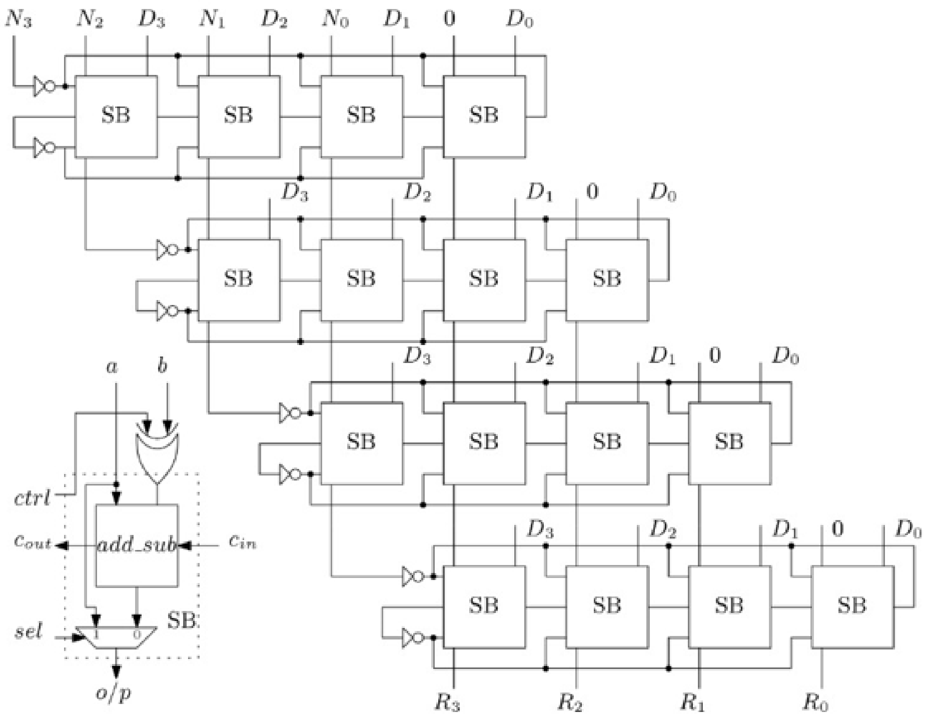
| | **Accumulator A** | **Dividend Q** | **Divisor M** |
|---|---|---|---|
| Initial Values | 1111 | 1001 | 1100 |
| Step 1: | | | |
| Left-Shift A,Q | 1111 | 001_ | |
| Sign (A,M) same: A-M | +0100 | | |
| Sign A changes: Unsuccessful | 0011 | 0010 | |
| Restore | 1111 | 0010 | |
| Step 2: | | | |
| Left-Shift A,Q | 1110 | 010 | |
| Sign (A,M) same: A-M | +0100 | | |
| Sign A changes: Unsuccessful | 0010 | 0100 | |
| Restore | 1110 | 0100 | |
| Step 3: | | | |
| Left-Shift A,Q | 1100 | 100_ | |
| Sign (A,M) same: A-M | +0100 | | |
| Dividend(A,Q) = 0: Successful | 0000 | 1001 | |
| Step 4: | | | |
| Left-Shift A,Q | 0001 | 001_ | |
| Sign (A,M) different: A+M | +1100 | | |
| Sign A changes: Unsuccessful | 1101 | 0010 | |
| Restore | 0001 | 0010 | |

Hence, remainder is given by A as 1 and quotient is given by Q as 2 which is true because:

$$-4 \times 2 + 1 = -7$$

## 5 Circuit Diagram

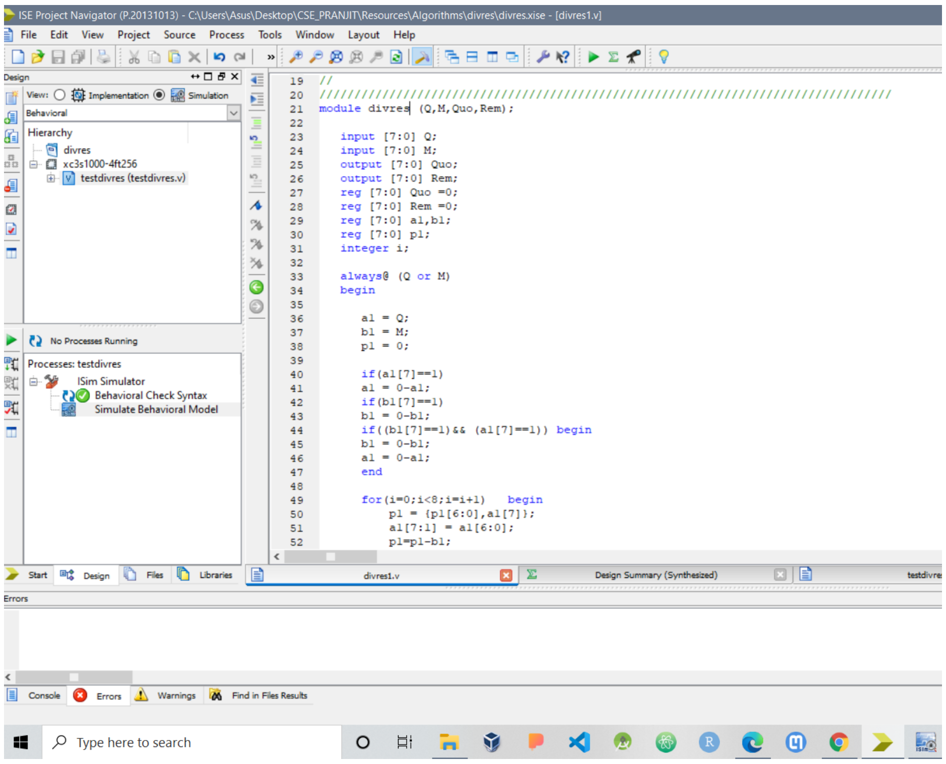

Accumulator
A
Quotient (multiplier) register
Q
Divisor (multiplicand) register
M
Parallel adder-subtracter
Control unit
n
2n
Remainder R
Quotient Q
D
V



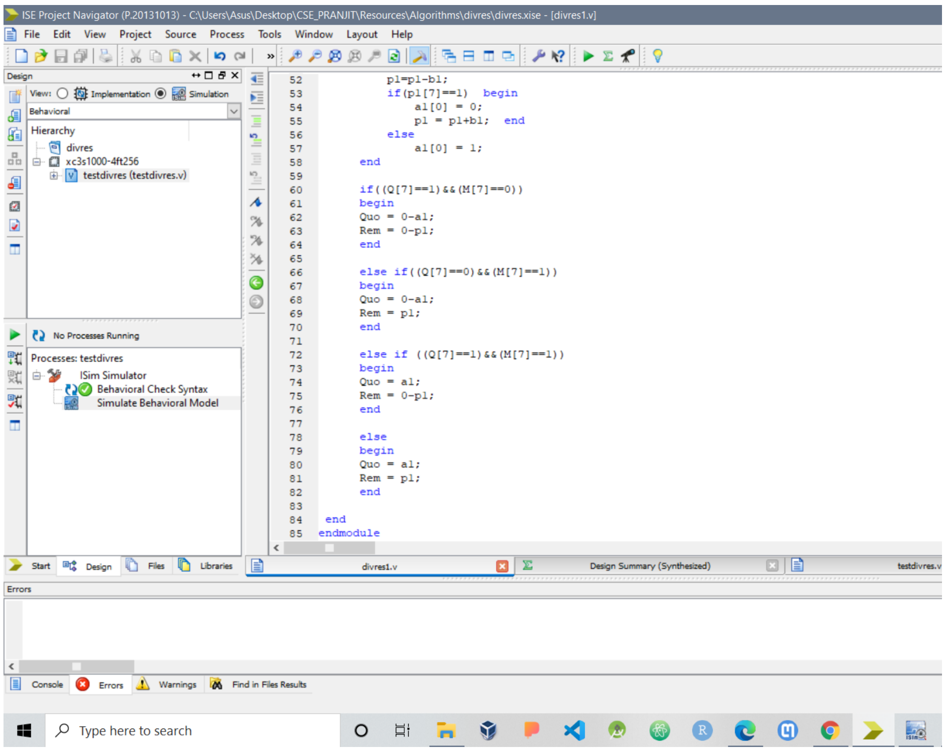
The circuit diagram is verified and simulated using a Verilog program which is attached at the Appendix and the results are simulated in Xilinx PlanAhead 14.7 of which is shown below:

## 6 Appendix

**Verilog Program Designed for Simulating the Restoring Division Algorithm in ISE Project Navigator**

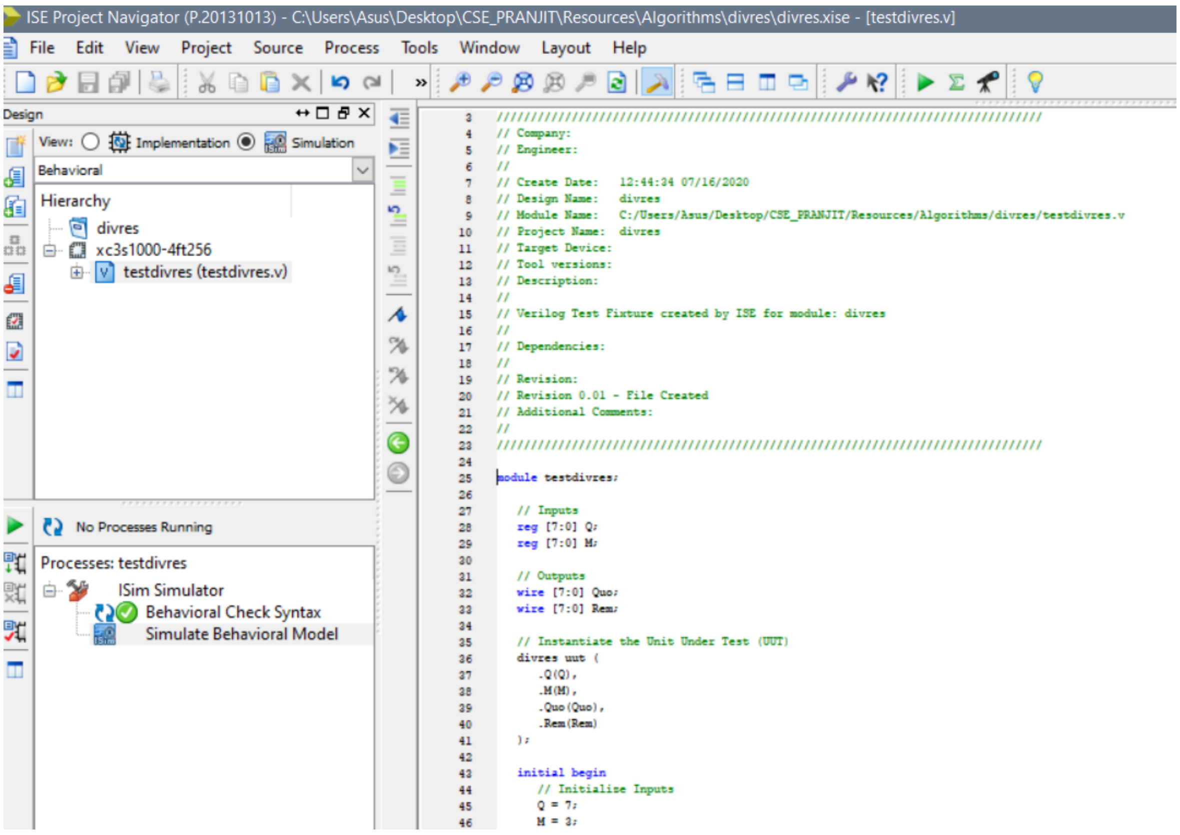

ISE Project Navigator (P.20131013) - C:\Users\Asus\Desktop\CSE_PRANJIT\Resources\Algorithms\divres\divres.xise - [divres1.v]

```
//
////////////////////////////////////////////////////////////////////////////////
module divres (Q,M,Quo,Rem);

   input [7:0] Q;
   input [7:0] M;
   output [7:0] Quo;
   output [7:0] Rem;
   reg [7:0] Quo =0;
   reg [7:0] Rem =0;
   reg [7:0] al,bl;
   reg [7:0] pl;
   integer i;

   always@ (Q or M)
   begin

      al = Q;
      bl = M;
      pl = 0;

      if(al[7]==1)
      al = 0-al;
      if(bl[7]==1)
      bl = 0-bl;
      if((bl[7]==1)&& (al[7]==1)) begin
      bl = 0-bl;
      al = 0-al;
      end

      for(i=0;i<8;i=i+1)   begin
          pl = {pl[6:0],al[7]};
          al[7:1] = al[6:0];
          pl=pl-bl;
```

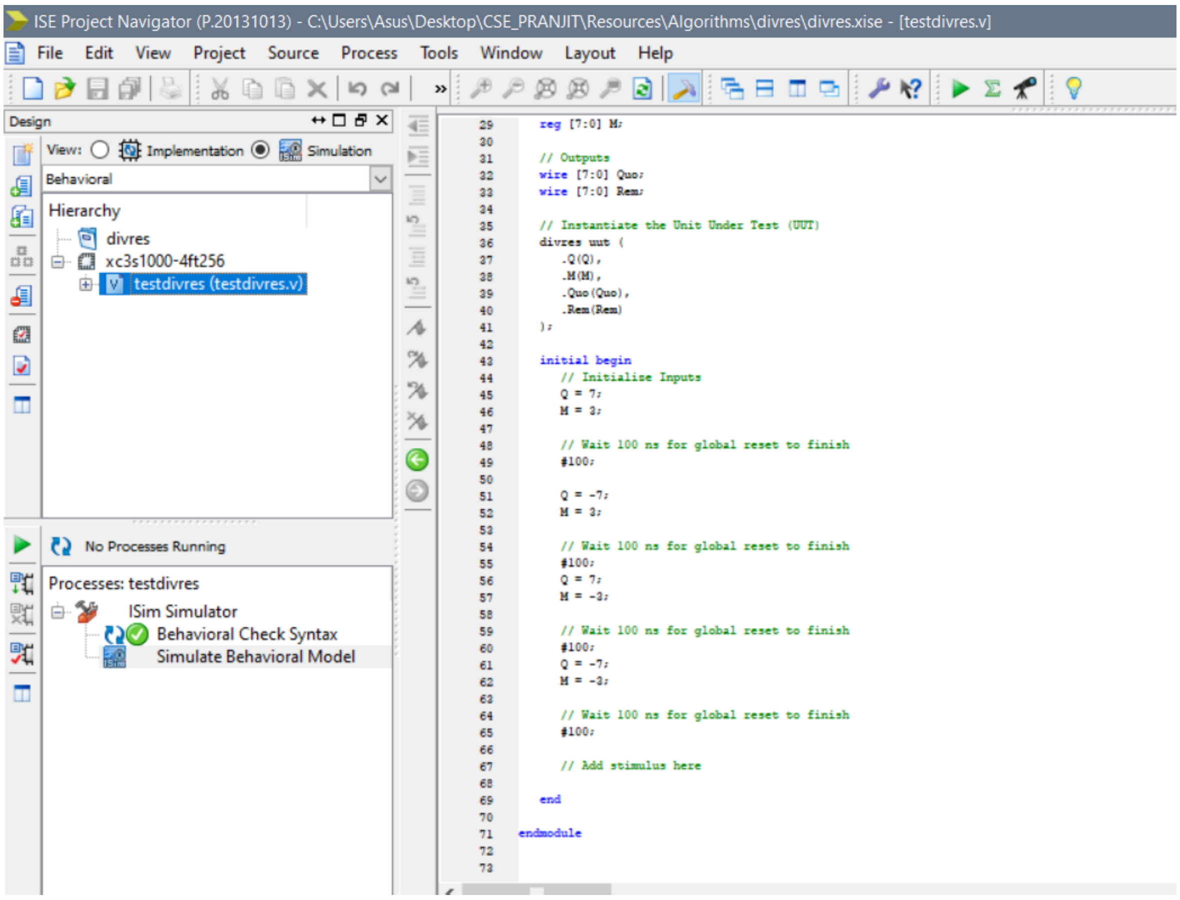

ISE Project Navigator (P.20131013) - C:\Users\Asus\Desktop\CSE_PRANJIT\Resources\Algorithms\divres\divres.xise - [divres1.v]
File Edit View Project Source Process Tools Window Layout Help
Design
View: Implementation Simulation
Behavioral
Hierarchy
divres
xc3s1000-4ft256
testdivres (testdivres.v)
No Processes Running
Processes: testdivres
ISim Simulator
Behavioral Check Syntax
Simulate Behavioral Model
Start Design Files Libraries
divres1.v
Design Summary (Synthesized)
testdivres.v
Errors
Console Errors Warnings Find in Files Results
Type here to search

```
                pl=pl-bl;
                if(pl[7]==1)  begin
                    al[0] = 0;
                    pl = pl+bl;  end
                else
                    al[0] = 1;
            end

            if((Q[7]==1)&&(M[7]==0))
            begin
            Quo = 0-al;
            Rem = 0-pl;
            end

            else if((Q[7]==0)&&(M[7]==1))
            begin
            Quo = 0-al;
            Rem = pl;
            end

            else if ((Q[7]==1)&&(M[7]==1))
            begin
            Quo = al;
            Rem = 0-pl;
            end

            else
            begin
            Quo = al;
            Rem = pl;
            end

     end
endmodule
```

The input given as dividend +/- 7 and divisor +/- 3


ISE Project Navigator (P.20131013) - C:\Users\Asus\Desktop\CSE_PRANJIT\Resources\Algorithms\divres\divres.xise - [testdivres.v]
File Edit View Project Source Process Tools Window Layout Help
Design
View: Implementation Simulation
Behavioral
Hierarchy
divres
xc3s1000-4ft256
testdivres (testdivres.v)
No Processes Running
Processes: testdivres
ISim Simulator
Behavioral Check Syntax
Simulate Behavioral Model

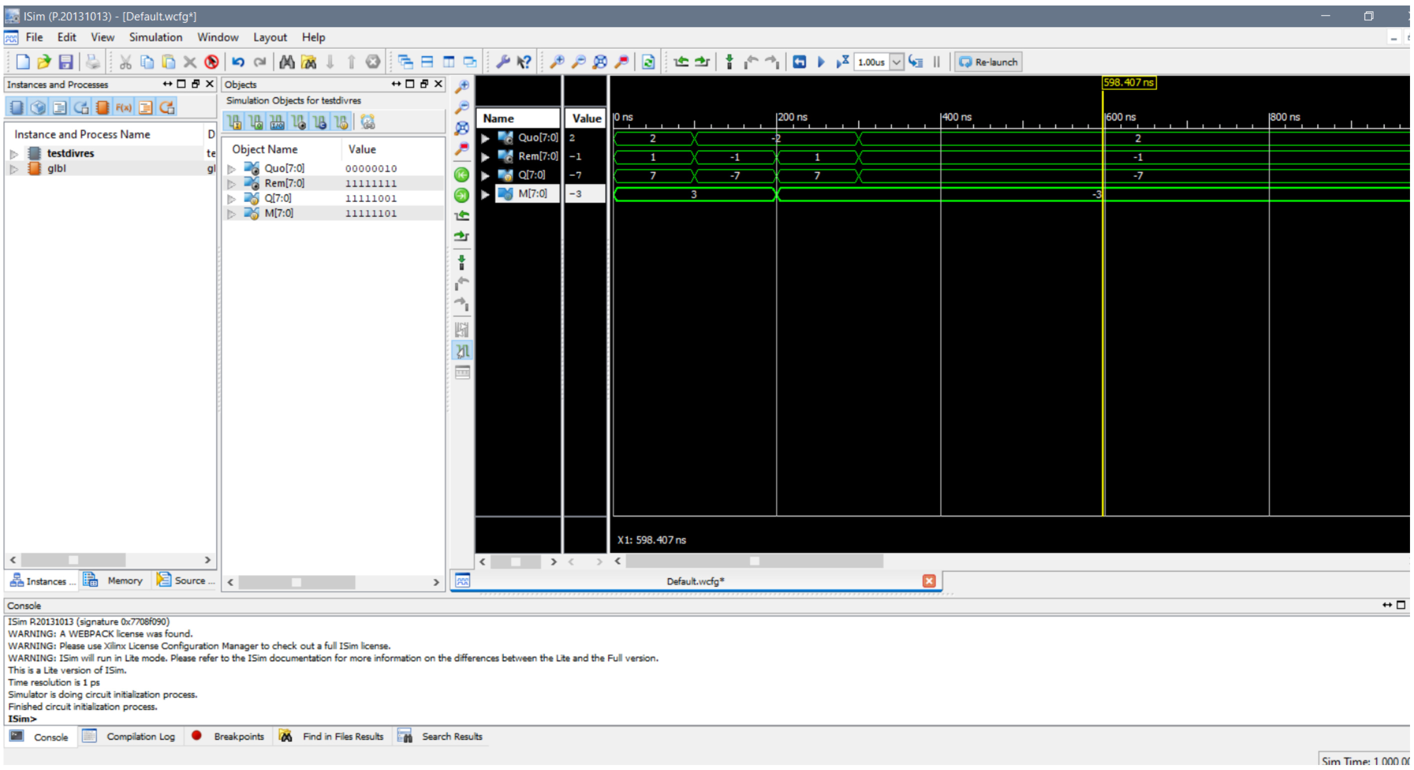
```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   12:44:34 07/16/2020
// Design Name:   divres
// Module Name:   C:/Users/Asus/Desktop/CSE_PRANJIT/Resources/Algorithms/divres/testdivres.v
// Project Name:  divres
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: divres
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module testdivres;

	// Inputs
	reg [7:0] Q;
	reg [7:0] M;

	// Outputs
	wire [7:0] Quo;
	wire [7:0] Rem;

	// Instantiate the Unit Under Test (UUT)
	divres uut (
		.Q(Q),
		.M(M),
		.Quo(Quo),
		.Rem(Rem)
	);

	initial begin
		// Initialize Inputs
		Q = 7;
		M = 3;
```



ISE Project Navigator (P.20131013) - C:\Users\Asus\Desktop\CSE_PRANJIT\Resources\Algorithms\divres\divres.xise - [testdivres.v]

```
    reg [7:0] M;

    // Outputs
    wire [7:0] Quo;
    wire [7:0] Rem;

    // Instantiate the Unit Under Test (UUT)
    divres uut (
        .Q(Q),
        .M(M),
        .Quo(Quo),
        .Rem(Rem)
    );

    initial begin
        // Initialize Inputs
        Q = 7;
        M = 3;

        // Wait 100 ns for global reset to finish
        #100;

        Q = -7;
        M = 3;

        // Wait 100 ns for global reset to finish
        #100;
        Q = 7;
        M = -3;

        // Wait 100 ns for global reset to finish
        #100;
        Q = -7;
        M = -3;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here

    end

endmodule
```

**Output : Simulated Output obtained in ISim**



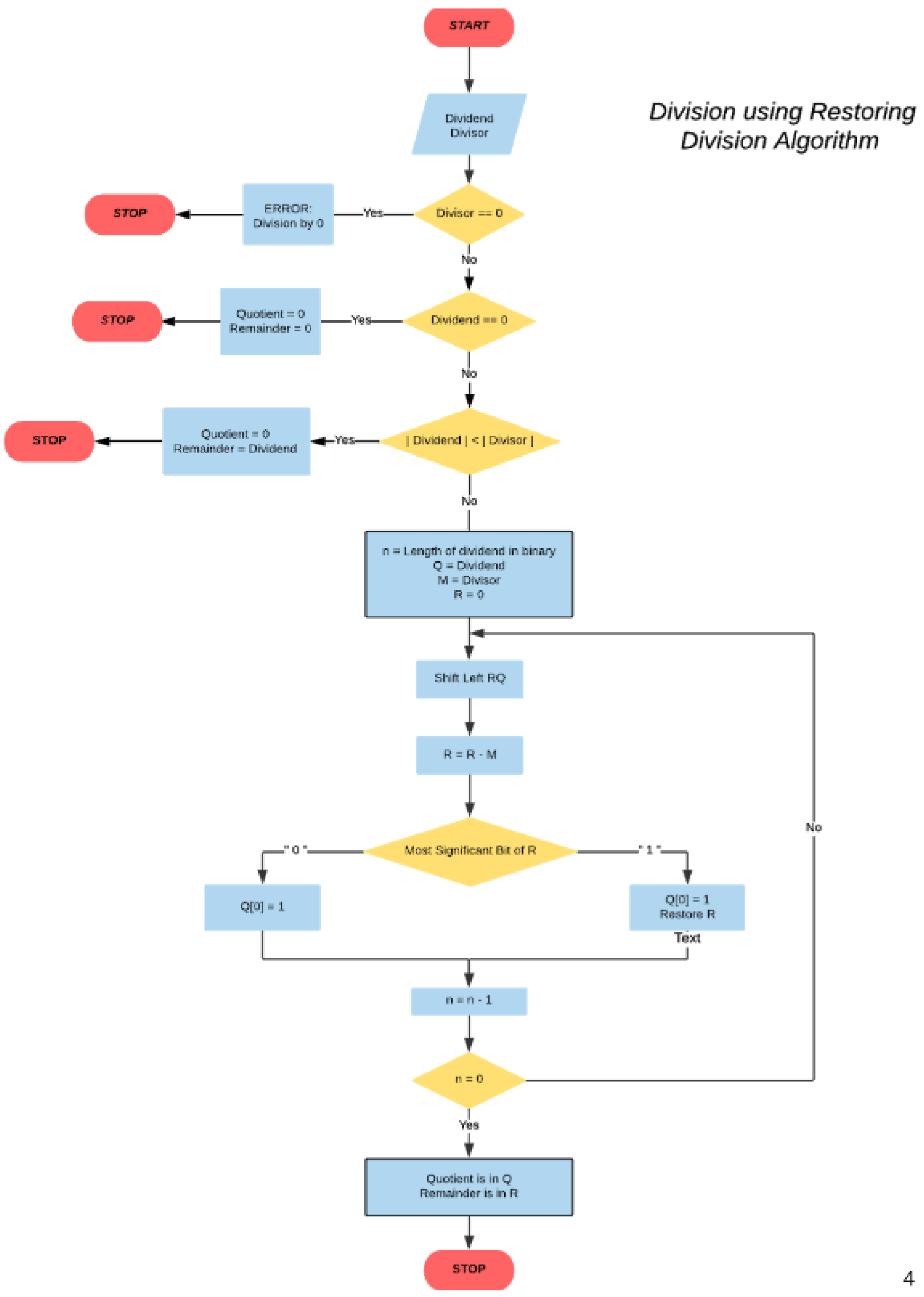
ISim (P.20131013) - [Default.wcfg*]

| Object Name | Value |
|---|---|
| Quo[7:0] | 00000010 |
| Rem[7:0] | 11111111 |
| Q[7:0] | 11111001 |
| M[7:0] | 11111101 |

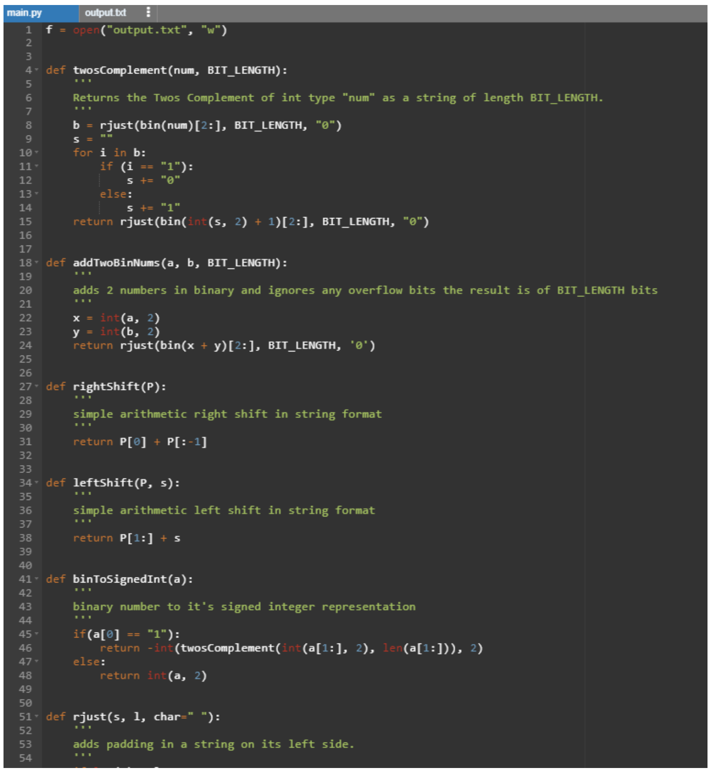
```
ISim P.20131013 (signature 0x7708f090)
WARNING: A WEBPACK license was found.
WARNING: Please use Xilinx License Configuration Manager to check out a full ISim license.
WARNING: ISim will run in Lite mode. Please refer to the ISim documentation for more information on the differences between the Lite and the Full version.
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
ISim>
```

# 7 Restoring Division Algorithm Implementation

**1.Restoring Division Algorithm Implemented for Signed Integers**
**Implemented Restoring Division Algorithm Flow Chart**

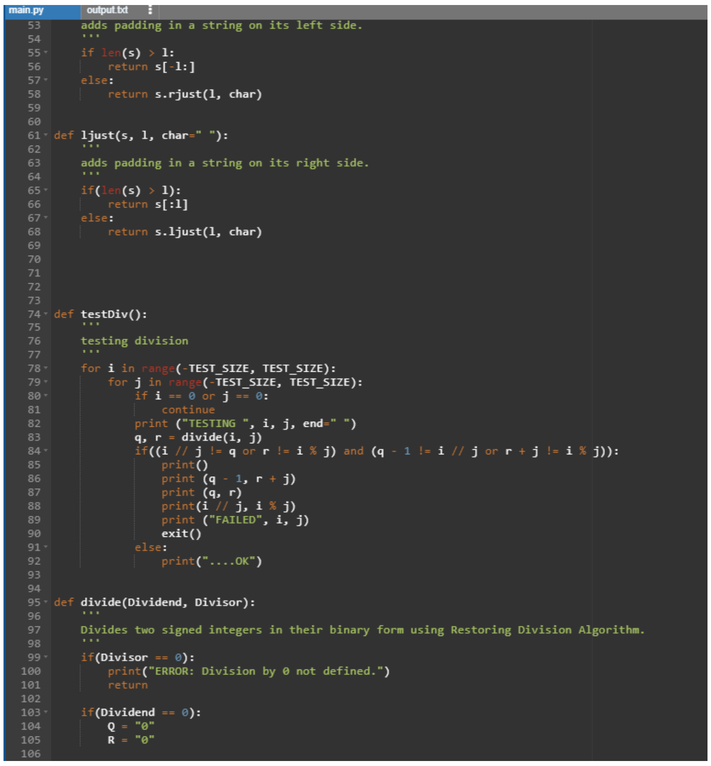


Python program developed for Restoring Division Algorithm on Signed Integers



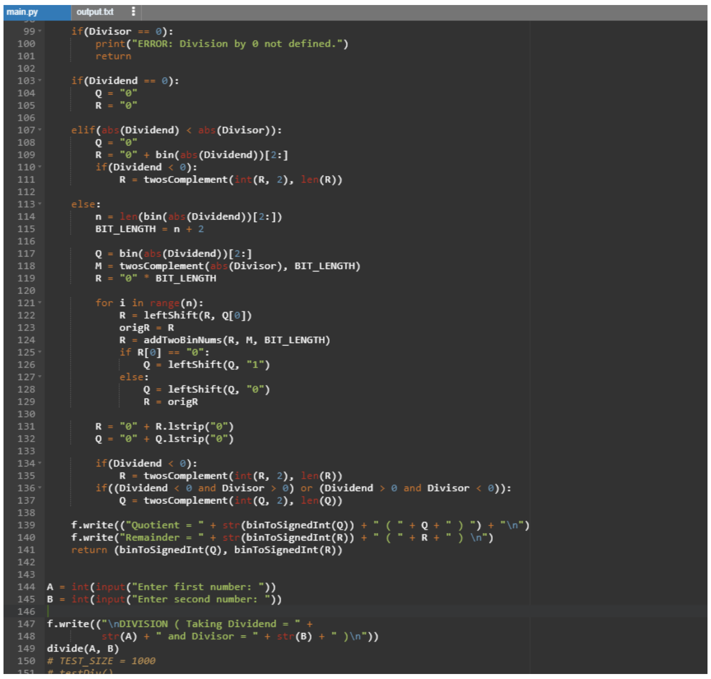
main.py output.txt

```python
f = open("output.txt", "w")


def twosComplement(num, BIT_LENGTH):
    '''
    Returns the Twos Complement of int type "num" as a string of length BIT_LENGTH.
    '''
    b = rjust(bin(num)[2:], BIT_LENGTH, "0")
    s = ""
    for i in b:
        if (i == "1"):
            s += "0"
        else:
            s += "1"
    return rjust(bin(int(s, 2) + 1)[2:], BIT_LENGTH, "0")


def addTwoBinNums(a, b, BIT_LENGTH):
    '''
    adds 2 numbers in binary and ignores any overflow bits the result is of BIT_LENGTH bits
    '''
    x = int(a, 2)
    y = int(b, 2)
    return rjust(bin(x + y)[2:], BIT_LENGTH, '0')


def rightShift(P):
    '''
    simple arithmetic right shift in string format
    '''
    return P[0] + P[:-1]


def leftShift(P, s):
    '''
    simple arithmetic left shift in string format
    '''
    return P[1:] + s


def binToSignedInt(a):
    '''
    binary number to it's signed integer representation
    '''
    if(a[0] == "1"):
        return -int(twosComplement(int(a[1:], 2), len(a[1:])), 2)
    else:
        return int(a, 2)


def rjust(s, l, char=" "):
    '''
    adds padding in a string on its left side.
    '''
```



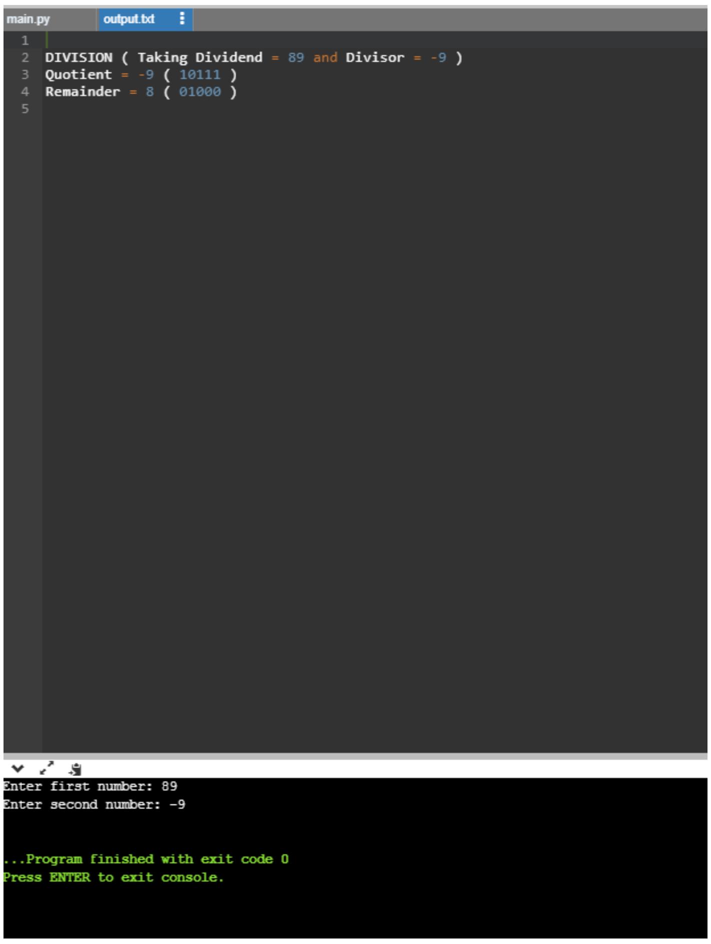
main.py output.txt

```python
    adds padding in a string on its left side.
    '''
    if len(s) > l:
        return s[-l:]
    else:
        return s.rjust(l, char)


def ljust(s, l, char=" "):
    '''
    adds padding in a string on its right side.
    '''
    if(len(s) > l):
        return s[:l]
    else:
        return s.ljust(l, char)





def testDiv():
    '''
    testing division
    '''
    for i in range(-TEST_SIZE, TEST_SIZE):
        for j in range(-TEST_SIZE, TEST_SIZE):
            if i == 0 or j == 0:
                continue
            print ("TESTING ", i, j, end=" ")
            q, r = divide(i, j)
            if((i // j != q or r != i % j) and (q - 1 != i // j or r + j != i % j)):
                print()
                print (q - 1, r + j)
                print (q, r)
                print(i // j, i % j)
                print ("FAILED", i, j)
                exit()
            else:
                print("....OK")


def divide(Dividend, Divisor):
    '''
    Divides two signed integers in their binary form using Restoring Division Algorithm.
    '''
    if(Divisor == 0):
        print("ERROR: Division by 0 not defined.")
        return

    if(Dividend == 0):
        Q = "0"
        R = "0"

```



```python
    if(Divisor == 0):
        print("ERROR: Division by 0 not defined.")
        return

    if(Dividend == 0):
        Q = "0"
        R = "0"

    elif(abs(Dividend) < abs(Divisor)):
        Q = "0"
        R = "0" + bin(abs(Dividend))[2:]
        if(Dividend < 0):
            R = twosComplement(int(R, 2), len(R))

    else:
        n = len(bin(abs(Dividend))[2:])
        BIT_LENGTH = n + 2

        Q = bin(abs(Dividend))[2:]
        M = twosComplement(abs(Divisor), BIT_LENGTH)
        R = "0" * BIT_LENGTH

        for i in range(n):
            R = leftShift(R, Q[0])
            origR = R
            R = addTwoBinNums(R, M, BIT_LENGTH)
            if R[0] == "0":
                Q = leftShift(Q, "1")
            else:
                Q = leftShift(Q, "0")
                R = origR

        R = "0" + R.lstrip("0")
        Q = "0" + Q.lstrip("0")

        if(Dividend < 0):
            R = twosComplement(int(R, 2), len(R))
        if((Dividend < 0 and Divisor > 0) or (Dividend > 0 and Divisor < 0)):
            Q = twosComplement(int(Q, 2), len(Q))

    f.write(("Quotient = " + str(binToSignedInt(Q)) + " ( " + Q + " ) ") + "\n")
    f.write("Remainder = " + str(binToSignedInt(R)) + " ( " + R + " ) \n")
    return (binToSignedInt(Q), binToSignedInt(R))


A = int(input("Enter first number: "))
B = int(input("Enter second number: "))

f.write(("\nDIVISION ( Taking Dividend = " +
        str(A) + " and Divisor = " + str(B) + " )\n"))
divide(A, B)
# TEST_SIZE = 1000
# testDiv()
```

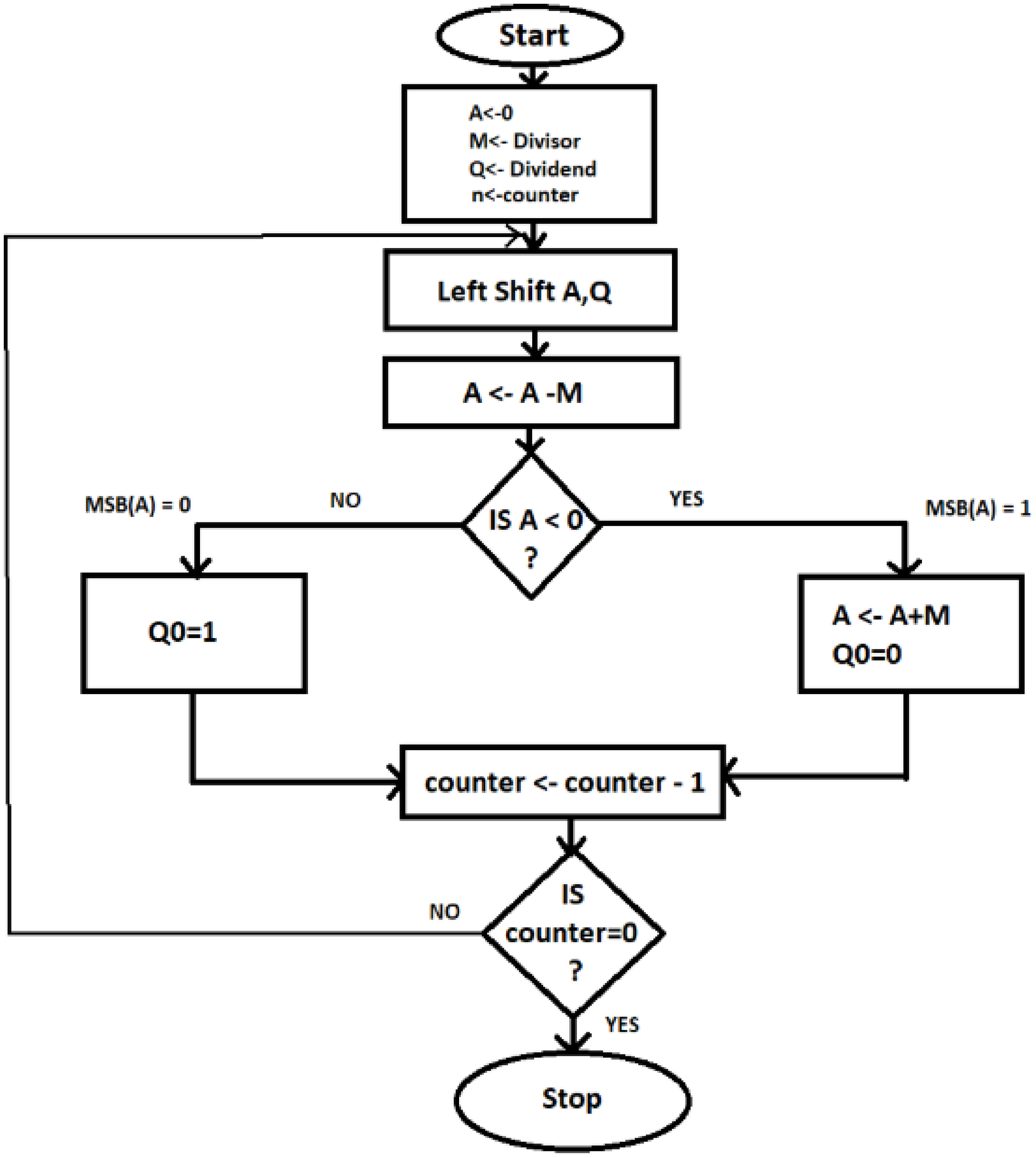
## Output Obtained



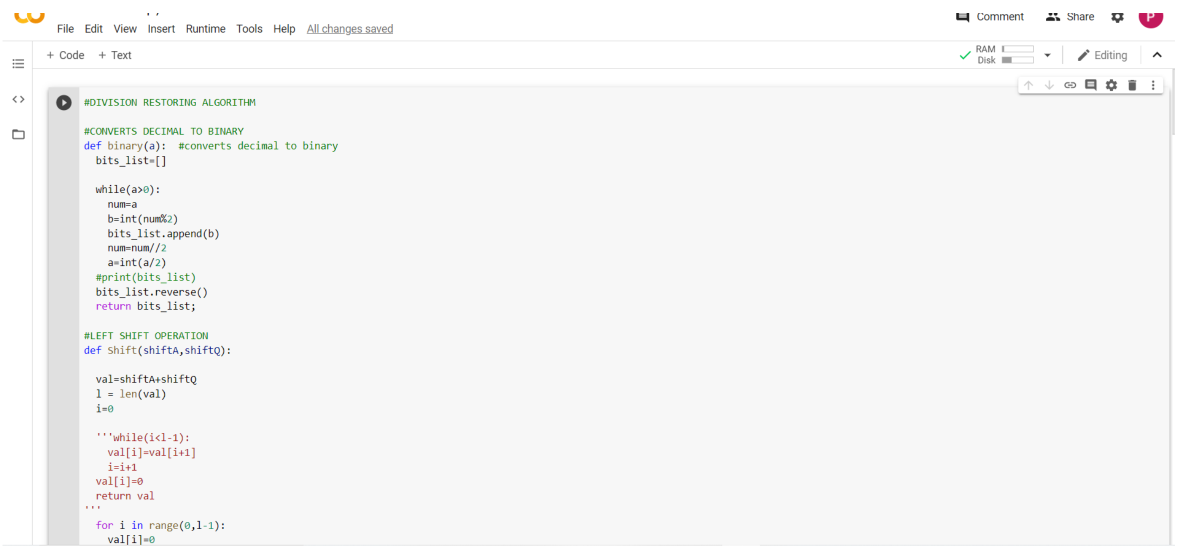
```

DIVISION ( Taking Dividend = 89 and Divisor = -9 )
Quotient = -9 ( 10111 )
Remainder = 8 ( 01000 )

```

```
Enter first number: 89
Enter second number: -9


...Program finished with exit code 0
Press ENTER to exit console.
```

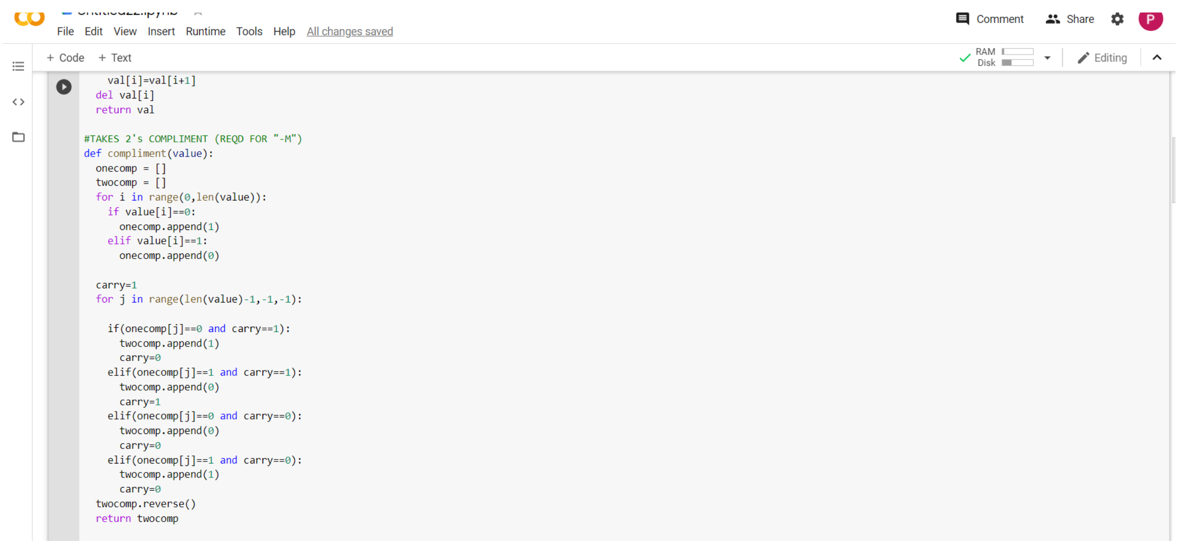
**Restoring Division Algorithm Implementation on Unsigned Integers using the flow chart**



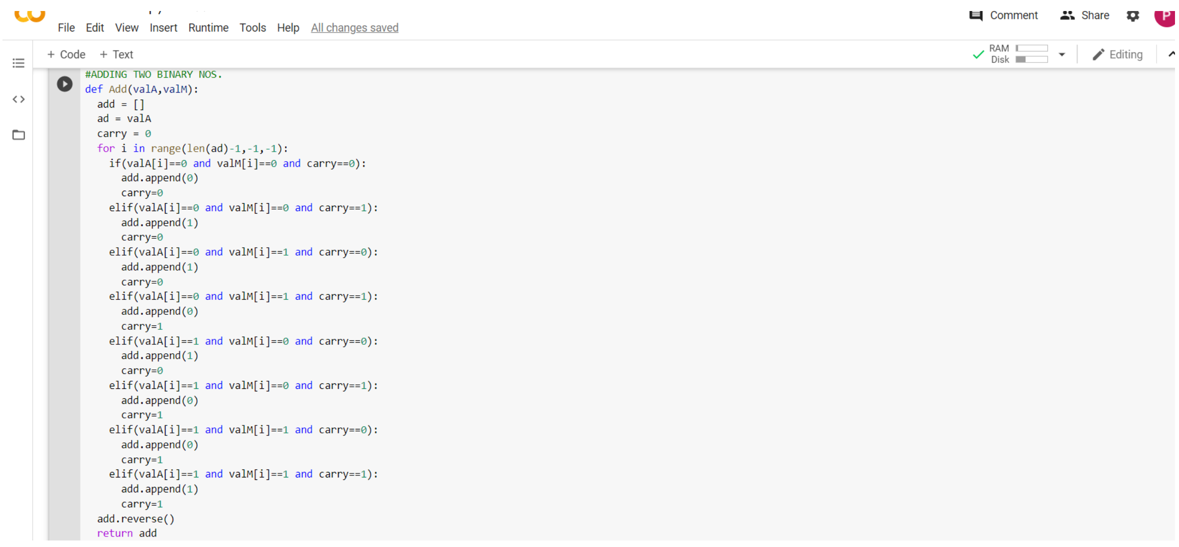
## Python Program Designed for Implementation of Restoring Division Algorithm on Unsigned Integers



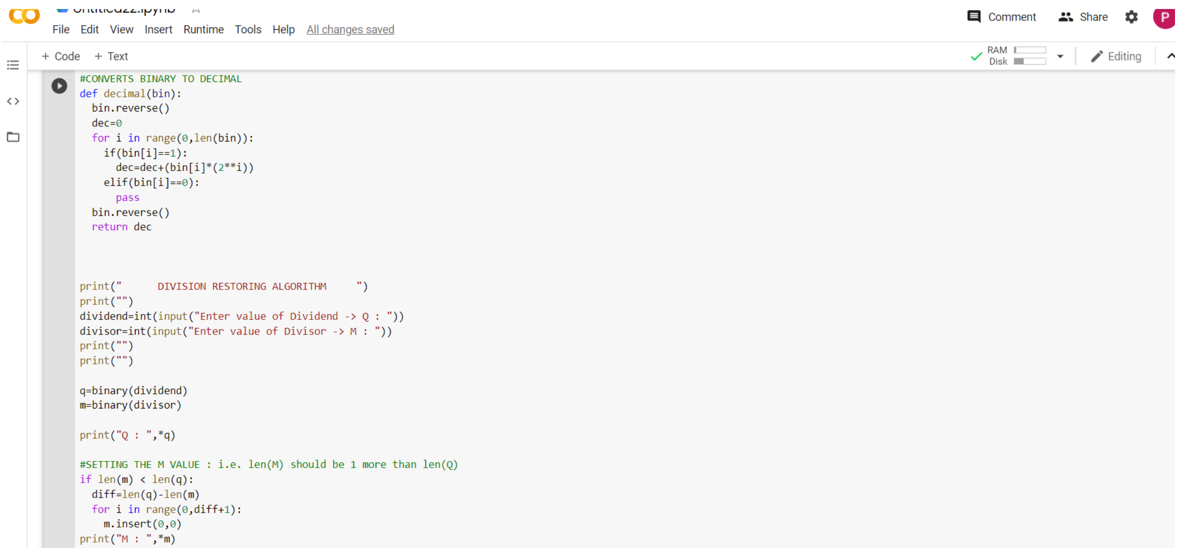
```
#DIVISION RESTORING ALGORITHM

#CONVERTS DECIMAL TO BINARY
def binary(a):  #converts decimal to binary
  bits_list=[]

  while(a>0):
    num=a
    b=int(num%2)
    bits_list.append(b)
    num=num//2
    a=int(a/2)
  #print(bits_list)
  bits_list.reverse()
  return bits_list;

#LEFT SHIFT OPERATION
def Shift(shiftA,shiftQ):

  val=shiftA+shiftQ
  l = len(val)
  i=0

  '''while(i<l-1):
    val[i]=val[i+1]
    i=i+1
  val[i]=0
  return val
'''
  for i in range(0,l-1):
    val[i]=0
```



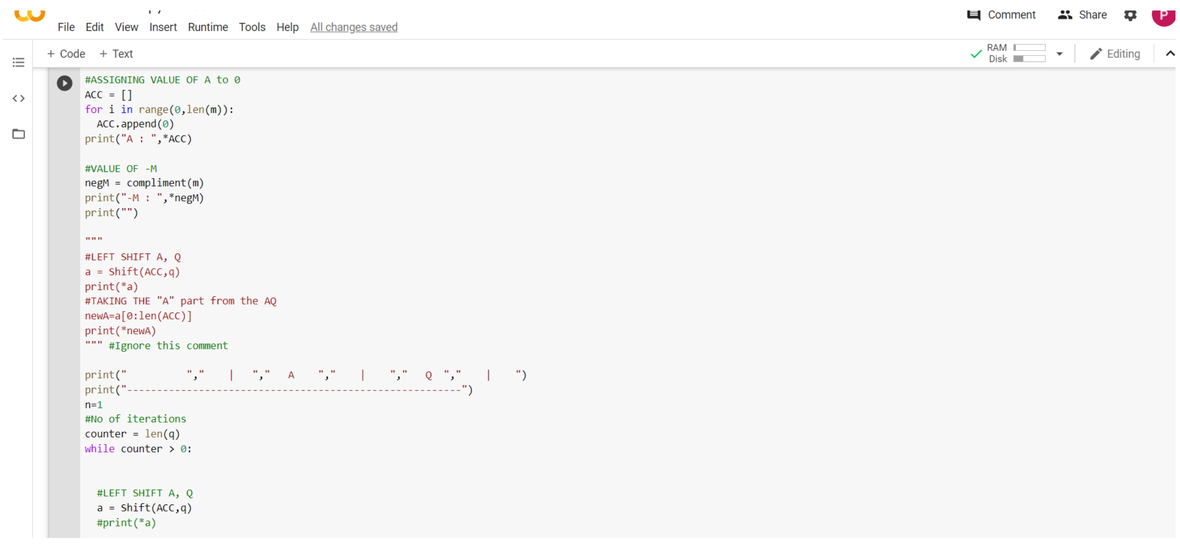
```
    val[i]=val[i+1]
  del val[i]
  return val

#TAKES 2's COMPLIMENT (REQD FOR "-M")
def compliment(value):
  onecomp = []
  twocomp = []
  for i in range(0,len(value)):
    if value[i]==0:
      onecomp.append(1)
    elif value[i]==1:
      onecomp.append(0)

  carry=1
  for j in range(len(value)-1,-1,-1):

    if(onecomp[j]==0 and carry==1):
      twocomp.append(1)
      carry=0
    elif(onecomp[j]==1 and carry==1):
      twocomp.append(0)
      carry=1
    elif(onecomp[j]==0 and carry==0):
      twocomp.append(0)
      carry=0
    elif(onecomp[j]==1 and carry==0):
      twocomp.append(1)
      carry=0
  twocomp.reverse()
  return twocomp
```



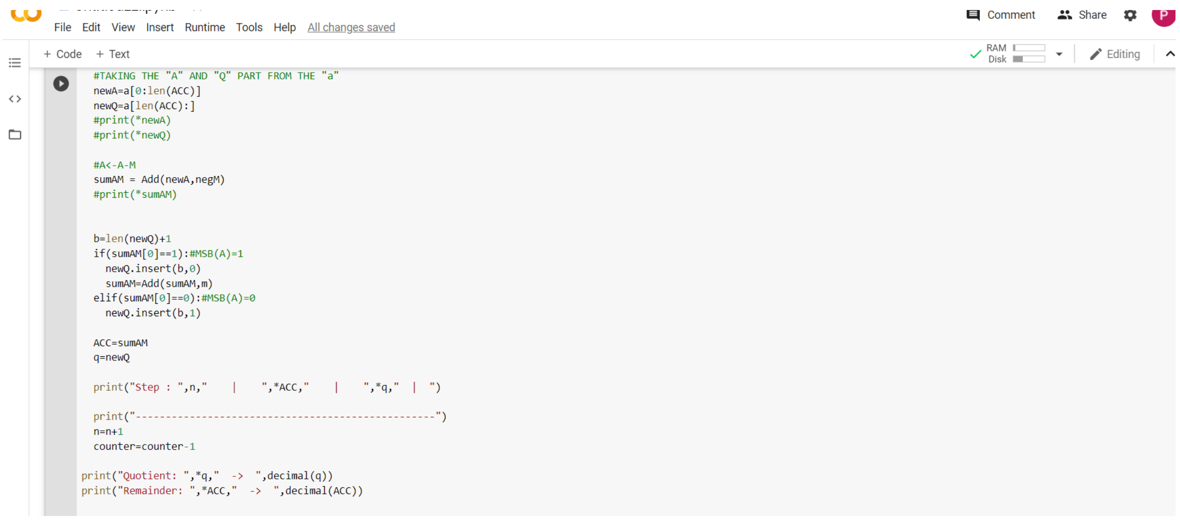
```
#ADDING TWO BINARY NOS.
def Add(valA,valM):
  add = []
  ad = valA
  carry = 0
  for i in range(len(ad)-1,-1,-1):
    if(valA[i]==0 and valM[i]==0 and carry==0):
      add.append(0)
      carry=0
    elif(valA[i]==0 and valM[i]==0 and carry==1):
      add.append(1)
      carry=0
    elif(valA[i]==0 and valM[i]==1 and carry==0):
      add.append(1)
      carry=0
    elif(valA[i]==0 and valM[i]==1 and carry==1):
      add.append(0)
      carry=1
    elif(valA[i]==1 and valM[i]==0 and carry==0):
      add.append(1)
      carry=0
    elif(valA[i]==1 and valM[i]==0 and carry==1):
      add.append(0)
      carry=1
    elif(valA[i]==1 and valM[i]==1 and carry==0):
      add.append(0)
      carry=1
    elif(valA[i]==1 and valM[i]==1 and carry==1):
      add.append(1)
      carry=1
  add.reverse()
  return add
```



```
#CONVERTS BINARY TO DECIMAL
def decimal(bin):
  bin.reverse()
  dec=0
  for i in range(0,len(bin)):
    if(bin[i]==1):
      dec=dec+(bin[i]*(2**i))
    elif(bin[i]==0):
      pass
  bin.reverse()
  return dec



print("      DIVISION RESTORING ALGORITHM     ")
print("")
dividend=int(input("Enter value of Dividend -> Q : "))
divisor=int(input("Enter value of Divisor -> M : "))
print("")
print("")

q=binary(dividend)
m=binary(divisor)

print("Q : ",*q)

#SETTING THE M VALUE : i.e. len(M) should be 1 more than len(Q)
if len(m) < len(q):
  diff=len(q)-len(m)
  for i in range(0,diff+1):
    m.insert(0,0)
print("M : ",*m)
```

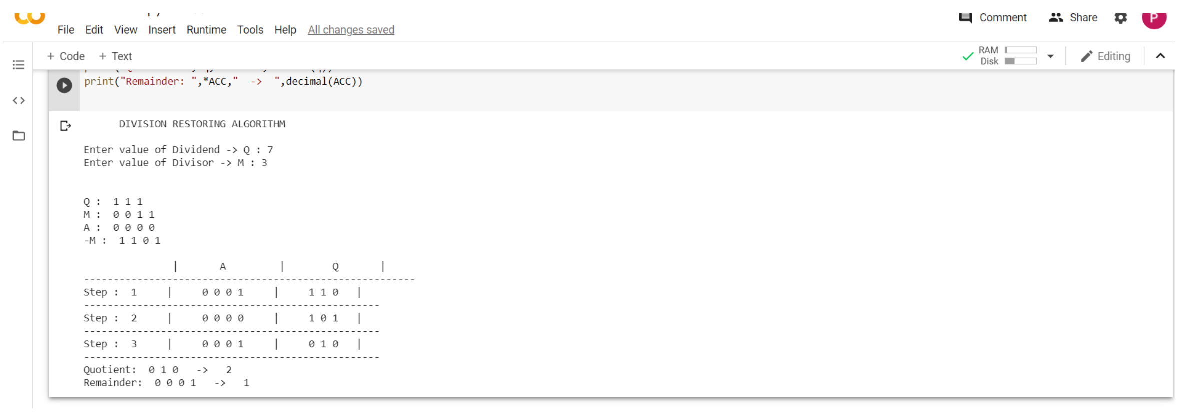


```
#ASSIGNING VALUE OF A to 0
ACC = []
for i in range(0,len(m)):
  ACC.append(0)
print("A : ",*ACC)

#VALUE OF -M
negM = compliment(m)
print("-M : ",*negM)
print("")

"""
#LEFT SHIFT A, Q
a = Shift(ACC,q)
print(*a)
#TAKING THE "A" part from the AQ
newA=a[0:len(ACC)]
print(*newA)
""" #Ignore this comment

print("          ","   |  ","  A   ","   |   ","  Q ","   |   ")
print("--------------------------------------------------------")
n=1
#No of iterations
counter = len(q)
while counter > 0:


  #LEFT SHIFT A, Q
  a = Shift(ACC,q)
  #print(*a)
```



```
  #TAKING THE "A" AND "Q" PART FROM THE "a"
  newA=a[0:len(ACC)]
  newQ=a[len(ACC):]
  #print(*newA)
  #print(*newQ)

  #A<-A-M
  sumAM = Add(newA,negM)
  #print(*sumAM)


  b=len(newQ)+1
  if(sumAM[0]==1):#MSB(A)=1
    newQ.insert(b,0)
    sumAM=Add(sumAM,m)
  elif(sumAM[0]==0):#MSB(A)=0
    newQ.insert(b,1)

  ACC=sumAM
  q=newQ

  print("Step : ",n,"    |    ",*ACC,"    |    ",*q," |  ")

  print("--------------------------------------------------")
  n=n+1
  counter=counter-1

print("Quotient: ",*q,"  ->  ",decimal(q))
print("Remainder: ",*ACC,"  ->  ",decimal(ACC))
```

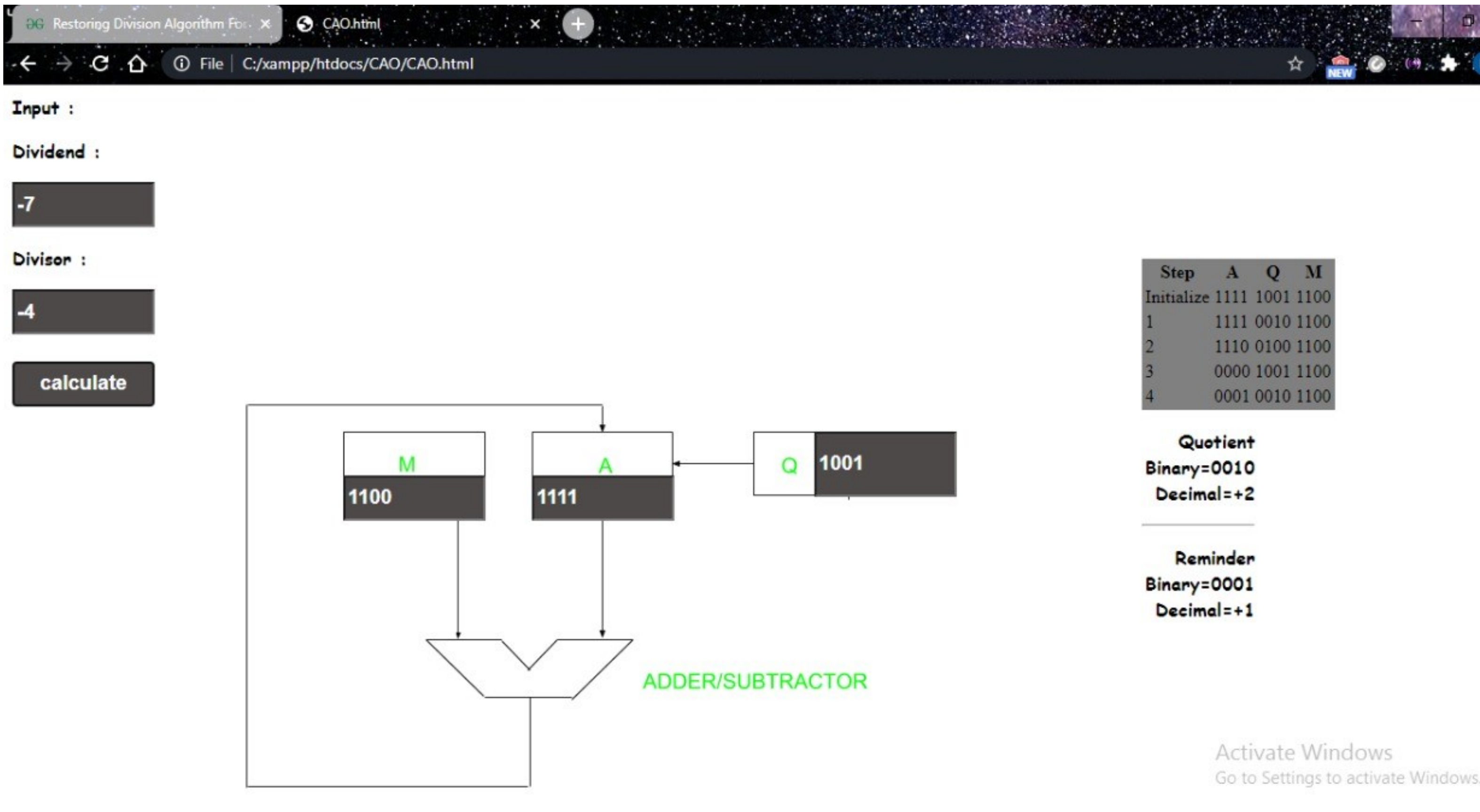
## Output Generated



```
print("Remainder: ",*ACC,"  ->  ",decimal(ACC))
```

```
      DIVISION RESTORING ALGORITHM

Enter value of Dividend -> Q : 7
Enter value of Divisor -> M : 3


Q :  1 1 1
M :  0 0 1 1
A :  0 0 0 0
-M :  1 1 0 1

               |       A         |        Q        |
-------------------------------------------------------
Step :  1     |     0 0 0 1     |     1 1 0   |
-------------------------------------------------
Step :  2     |     0 0 0 0     |     1 0 1   |
-------------------------------------------------
Step :  3     |     0 0 0 1     |     0 1 0   |
-------------------------------------------------
Quotient:  0 1 0   ->   2
Remainder:  0 0 0 1   ->   1
```

## 8 Website Designed for Performing Restoring Division Algorithm on Signed Integers


Restoring Division Algorithm For
CAO.html
File
C:/xampp/htdocs/CAO/CAO.html
calculate
M
A
Q
ADDER/SUBTRACTOR
Activate Windows
Go to Settings to activate Windows

Input :

Dividend :

-7

Divisor :

-4

M: 1100

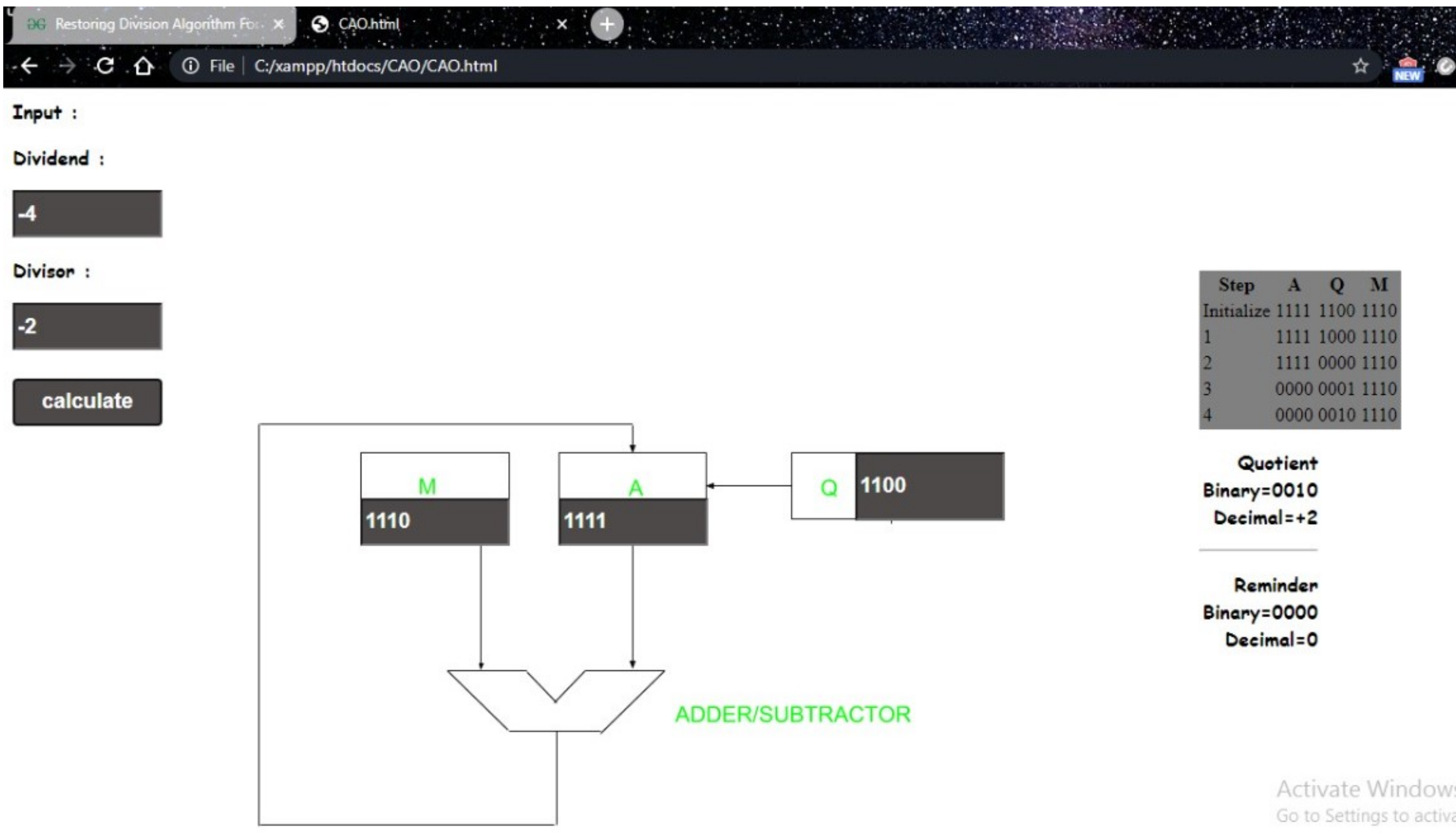
A: 1111

Q: 1001

| Step | A | Q | M |
|---|---|---|---|
| Initialize | 1111 | 1001 | 1100 |
| 1 | 1111 | 0010 | 1100 |
| 2 | 1110 | 0100 | 1100 |
| 3 | 0000 | 1001 | 1100 |
| 4 | 0001 | 0010 | 1100 |

Quotient
Binary=0010
Decimal=+2

Reminder
Binary=0001
Decimal=+1



Input :

Dividend :

-4

Divisor :

-2

calculate

M 1110

A 1111

Q 1100

| Step | A | Q | M |
|---|---|---|---|
| Initialize | 1111 | 1100 | 1110 |
| 1 | 1111 | 1000 | 1110 |
| 2 | 1111 | 0000 | 1110 |
| 3 | 0000 | 0001 | 1110 |
| 4 | 0000 | 0010 | 1110 |

Quotient
Binary=0010
Decimal=+2

Reminder
Binary=0000
Decimal=0

## 9 Conclusion

Our project describes the Restoring Division Algorithm for Signed Integers along with various numerical examples as well as visual simulations. Though there are many division algorithms available like Digit Recurrence Algorithm, SRT Division. CORDIC Algorithm, but the Restoring Division Algorithm is considered in most processors because of its simplicity and low complexity. The circuit diagram is simple as compared to other division algorithms' circuit diagrams which was simulated using a Verilog program in our project.

## 10 References:

1. Integer Representation and Division Algorithms–Bharat Accharya Lecture Notes

2. Computer System Architecture – Morris M. Mano

3. VHDL Implementation of Restoring Division Algorithm – Sukhmeet Kaur, Suman, Manpreet Singh Manna, Rajeev Agarwal.

4. Division Algorithms – Wikipedia

5. Restoring Division Algorithm – GeeksforGeeks