Experiment vi: ALU

Objective:

1. To implement ALU using VHDL

Theory:

The Arithmetic and Logical unite is the fundamental component in a computing system like a computer. It is basically the actual data processing element within the central processing unit (CPU) in a computing system. It performs all the arithmetic and logical operations and forms the backbone of modern computer technology.

Architecture:

```
--design for alu
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.NUMERIC_STD.all;
entity ALU is
 generic ( constant N: natural:=1); -- number of shifted or rotated bits
 Port (
 A, B: in STD_LOGIC_VECTOR (7 downto 0); -- 2 inputs 8-bit
 ALU_Sel: in STD_LOGIC_VECTOR (3 downto 0); -- 1 input 4-bit for selecting function
 ALU_Out: out STD_LOGIC_VECTOR (7 downto 0); -- 1 output 8-bit
 Carryout: out std_logic; -- Carryout flag
 );
end ALU;
architecture Behavioral of ALU is
signal ALU_Result: std_logic_vector (7 downto 0);
signal tmp: std_logic_vector (8 downto 0);
begin
 process(A,B,ALU_Sel)
 begin
 case(ALU_Sel) is
 when "0000" => -- Addition
 ALU_Result <= A + B ;
 when "0001" => -- Subtraction
 ALU_Result <= A - B ;
 when "0010" => -- Multiplication
 ALU_Result<=std_logic_vector(to_unsigned((to_integer(unsigned(A)) *
to_integer(unsigned(B))),8)) ;
 when "0011" => -- Division
 ALU_Result <= std_logic_vector(to_unsigned(to_integer(unsigned(A)) /
to_integer(unsigned(B)),8)) ;
 when "0100" => -- Logical shift left
 ALU_Result <= std_logic_vector(unsigned(A) sll N);
 when "0101" => -- Logical shift right
 ALU_Result <= std_logic_vector(unsigned(A) srl N);
 when "0110" => -- Rotate left
 ALU_Result <= std_logic_vector(unsigned(A) rol N);
```

```
 when "0111" => -- Rotate right
 ALU_Result <= std_logic_vector(unsigned(A) ror N);
 when "1000" => -- Logical and
 ALU_Result <= A and B;
 when "1001" => -- Logical or
 ALU_Result <= A or B;
 when "1010" => -- Logical xor
 ALU_Result <= A xor B;
 when "1011" => -- Logical nor
 ALU_Result <= A nor B;
 when "1100" => -- Logical nand
 ALU_Result <= A nand B;
 when "1101" => -- Logical xnor
 ALU_Result <= A xnor B;
 when "1110" => -- Greater comparison
 if(A>B) then
 ALU_Result <= x"01" ;
 else
 ALU_Result <= x"00" ;
 end if;when "1111" => -- Equal comparison
 if(A=B) then
 ALU_Result <= x"01" ;
 else
 ALU_Result <= x"00" ;
 end if;
 when others => ALU_Result <= A + B ;
 end case;
end process;
ALU_Out <= ALU_Result; -- ALU out
tmp <= ('0' & A) + ('0' & B);
Carryout <= tmp(8); -- Carryout flag
end Behavioral;
```

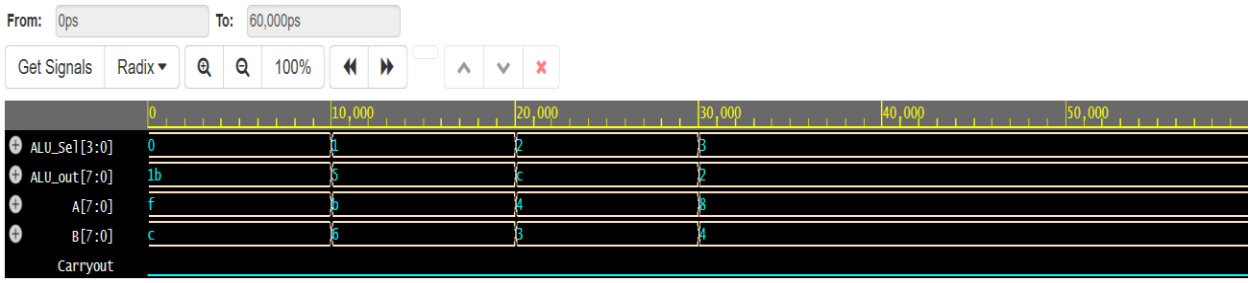
Source code:

```
 -- Testbench for alu adder
library IEEE;
use IEEE.std_logic_1164.all;

entity testbench is
  -- empty
end testbench;

architecture tb of testbench is

  -- DUT component
  component ALU is
 Port (
 A, B: in STD_LOGIC_VECTOR (7 downto 0); -- 2 inputs 8-bit
```

```
 ALU_Sel: in STD_LOGIC_VECTOR (3 downto 0); -- 1 input 4-bit for selecting function
 ALU_Out: out STD_LOGIC_VECTOR (7 downto 0); -- 1 output 8-bit
 Carryout: out std_logic; -- Carryout flag
 );

  end component;

  signal A, B: std_logic_vector(7 downto 0);
  signal ALU_Sel: std_logic_vector(3 downto 0);
  signal ALU_out: std_logic_vector(7 downto 0);
  signal Carryout: std_logic;

begin

  -- Connect DUT
  DUT: ALU
    port map (
      A => A,
      B => B,
      ALU_Sel => ALU_Sel,
      ALU_out => ALU_out,
      Carryout => Carryout
    );

  -- Test process
  process
  begin
    -- Test cases
    A <= "00001111";
    B <= "00001100";
    ALU_Sel<= "0000";
    wait for 10 ns;  -- Expected output: S_out = "0000", Cout_out = '0'


    A <= "00001011";
    B <= "00000110";
    ALU_Sel <= "0001";
    wait for 10 ns;  -- Expected output: S_out = "0010", Cout_out = '0'

    A <= "00000100";
    B <= "00000011";
    ALU_Sel <= "0010";
    wait for 10 ns;
     A <= "00001000";
    B <= "00000100";
    ALU_Sel <= "0011";
    wait for 10 ns;

    -- End of simulation
    wait for 20 ns;
```

```
 assert false report "End of simulation" severity failure;
 end process;

end tb;
```

Output:



Conclusion:

In conclusion, designing an ALU in VHDL enables efficient implementation of arithmetic and logic operations in digital systems. VHDL's flexibility and modularity allow for easy testing, simulation, and refinement, ensuring reliable performance. It is a powerful tool for creating high-performance, cost-effective hardware designs.

Experiment vii: 3-Segment Pipeline

Objective:

1. To implement 3-Segment Pipeline using VHDL

Theory:

The Arithmetic and Logical unite is the fundamental component in a computing system like a computer. It is basically the actual data processing element within the central processing unit (CPU) in a computing system. It performs all the arithmetic and logical operations and forms the backbone of modern computer technology.

Architecture:

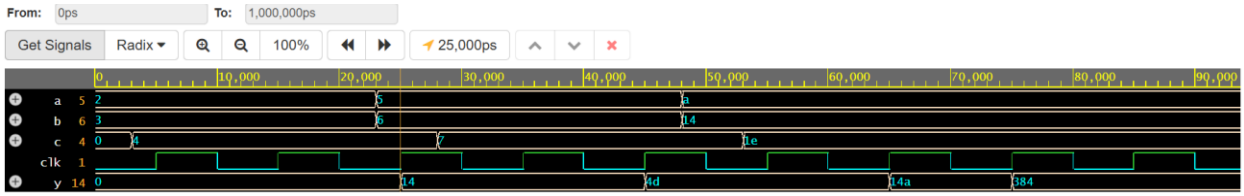
```
--design for pipeline
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pipeline is
    Port (
        a : in integer;
        b : in integer;
        c : in integer;
        clk : in STD_LOGIC;
        y : out integer
    );
end pipeline;

architecture Behavioral of pipeline is
    signal r1, r2, r3, r4, r5 : integer := 0;
begin
    y <= r5;

    process(clk)
    begin
        if rising_edge(clk) then
            -- Pipeline stage 1
            r1 <= a;
            r2 <= b;

            -- Pipeline stage 2 (registered)
            r4 <= r1 + r2;
            r3 <= c;

            -- Pipeline stage 3 (registered)
            r5 <= r4 * r3;
        end if;
    end process;
end Behavioral;
```

Source code:

```
--testbench for pipeline
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pipeline_tb is
end pipeline_tb;

architecture Behavioral of pipeline_tb is
    component pipeline
        Port (
            a : in integer;
            b : in integer;
            c : in integer;
            clk : in STD_LOGIC;
            y : out integer
        );
    end component;

    signal a, b, c : integer := 0;
    signal y : integer;
    signal clk : STD_LOGIC := '0';

    constant clk_period : time := 10 ns;

begin
    uut: pipeline port map (
        a => a,
        b => b,
        c => c,
        clk => clk,
        y => y
    );

    -- Clock generation
    clk_process: process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
```

```
        -- Test case 1
        a <= 2;
        b <= 3;
        wait for 3 ns;
        c <= 4;
        wait for clk_period*2;

        -- Test case 2
        a <= 5;
        b <= 6;
        wait for 5 ns;
        c <= 7;
        wait for clk_period*2;

        -- Test case 3
        a <= 10;
        b <= 20;
        wait for 5 ns;
        c <= 30;
        wait for clk_period*2;

        -- End simulation
        wait;
    end process;
end Behavioral;
```

Output:



Conclusion:

In conclusion, implementing a 3-segment pipeline in VHDL enhances system performance by allowing parallel processing of data in stages. This approach increases throughput and reduces latency by efficiently organizing tasks across different pipeline segments. VHDL provides the necessary tools to model, simulate, and optimize the pipeline, ensuring smooth operation and scalability in complex digital systems.

Experiment 4: Booth's Multiplication Algorithm

Objective:

1. To implement Booth's Multiplication Algorithm.

Theory:

Hardware Implementation

It needs same hardware as that of addition and subtraction of signed-magnitude. In addition, it needs two more registers Q and SC.

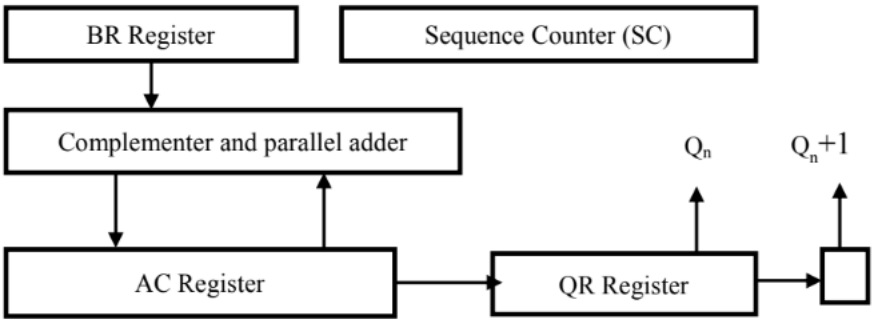


Fig: hardware for booths algorithm

Algorithm:

1. Initially, BR←Multiplicand and QR←Multiplier.
2. AC←0, Q_{n+1}←0 and SC←n (no of bits in multiplier).
3. Inspect Q_n, Qn+1,
   i. If (Q_nQ_{n+1}=10), first 1 in a string of 1' s has been encountered, subtraction required. i.e., **AC←AC+$\overline{BR}$+1**
   ii. If (Q_nQ_{n+1}=01), first 0 in a string of 0' s has been encountered, addition required. **i.e., AC←AC+BR**
   iii. If (Q_nQ_{n+1}=00 or 11(equal)), do nothing
4. Shift right the partial product and the multiplier (including bit Q_{n+1}). This Arithmetic Shift Right (ashr) operation shifts AC and QR to the right and leaves the sign bit in AC unchanged.
5. SC←SC-1. If (SC=0) stop the process else continue and go to step 3.
6. Result in AC and QR

Flowchart:

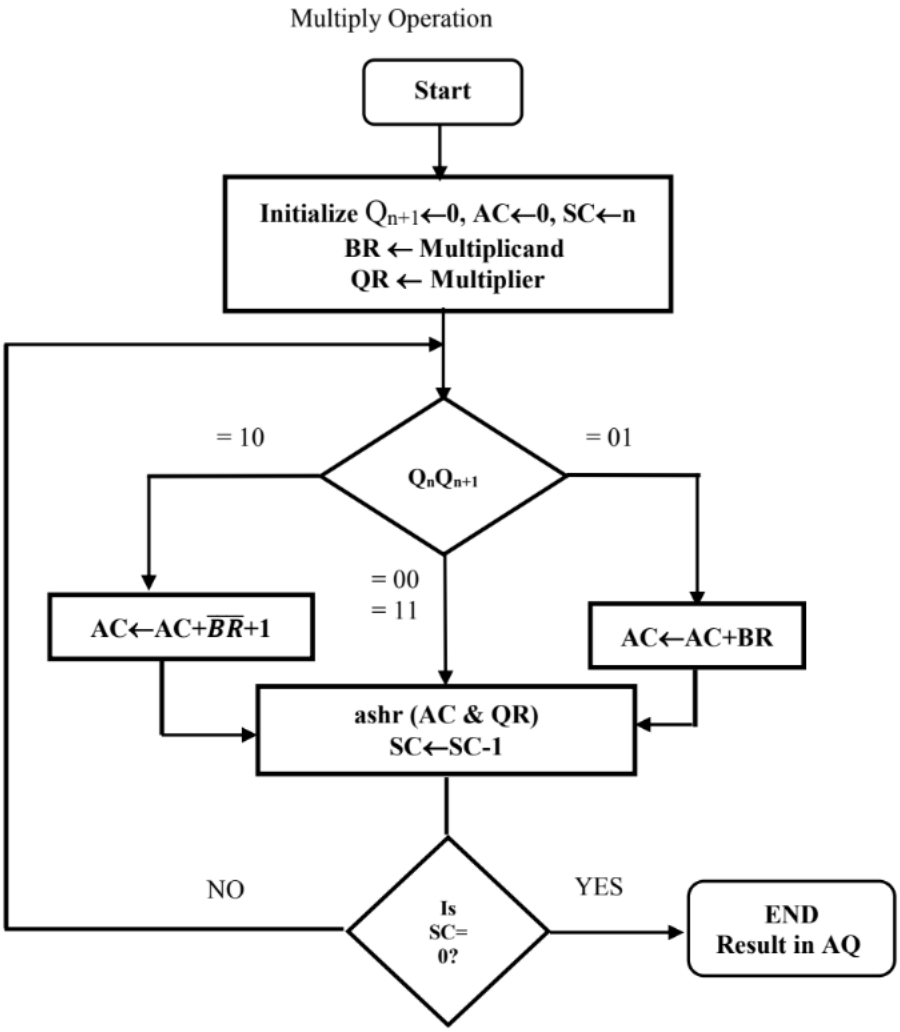


Fig: Booths Algorithm for multiplication of Signed-2's Complement numbers

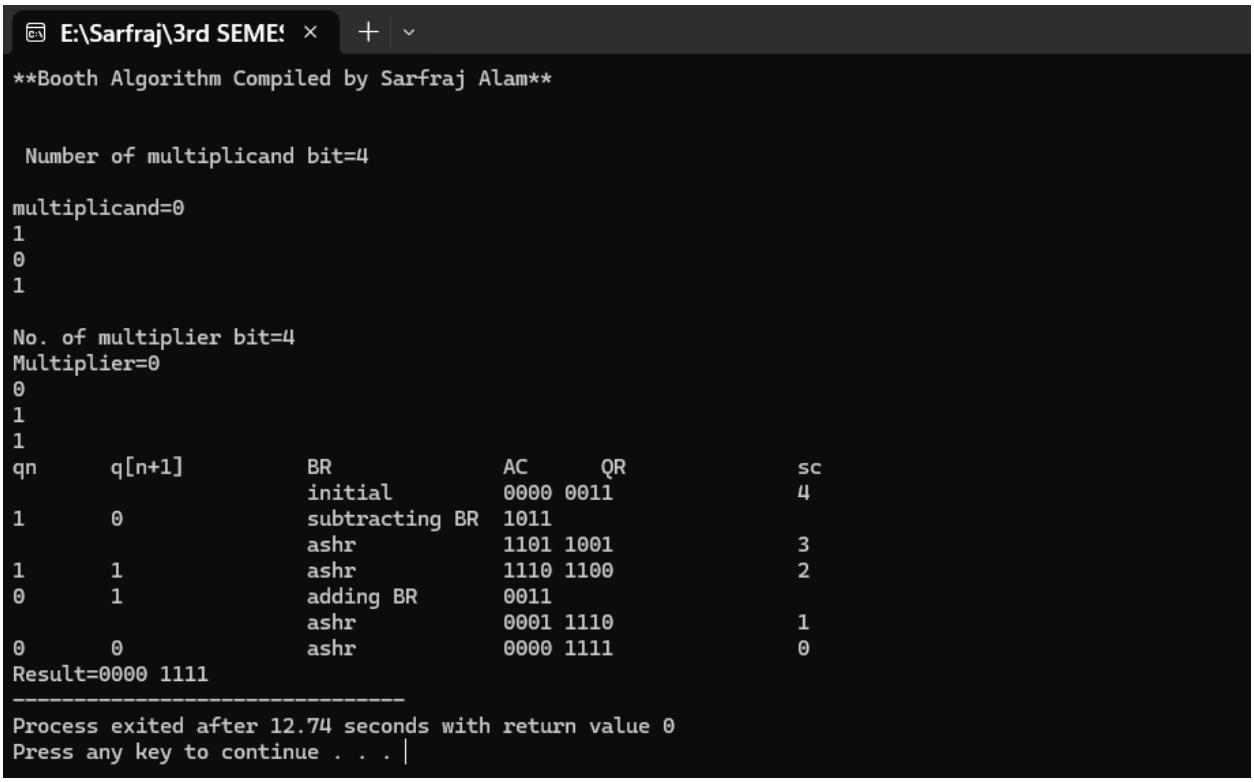
Source code

```
#include<iostream>
using namespace std;
void add(int a[], int x[], int q);
void complement(int a[], int n) {
  int i;
  int x[8] = { NULL };
  x[0] = 1;
  for (i = 0; i < n; i++) {
    a[i] = (a[i] + 1) % 2;
  }
  add(a, x, n);
}
```

```
void add(int ac[], int x[], int q) {
  int i, c = 0;
  for (i = 0; i < q; i++) {
    ac[i] = ac[i] + x[i] + c;
    if (ac[i] > 1) {
      ac[i] = ac[i] % 2;
      c = 1;
    }else
    c = 0;
  }
}

void ashr(int ac[], int qr[], int &qn, int q) {
  int temp, i;
  temp = ac[0];
  qn = qr[0];
  cout << "\t\tashr\t\t";
  for (i = 0; i < q - 1; i++) {
    ac[i] = ac[i + 1];
    qr[i] = qr[i + 1];
  }
  qr[q - 1] = temp;
}

void display(int ac[], int qr[], int qrn) {
  int i;
  for (i = qrn - 1; i >= 0; i--)
    cout << ac[i];
  cout << " ";
  for (i = qrn - 1; i >= 0; i--)
    cout << qr[i];
}

int main(int argc, char **argv) {
  int mt[10], br[10], qr[10], sc, ac[10] = { 0 };
  int brn, qrn, i, qn, temp;
  cout<<"**Booth Algorithm Compiled by Sarfraj Alam**\n"<<endl;
  cout << "\n Number of multiplicand bit=";
  cin >> brn;
  cout << "\nmultiplicand=";

  for (i = brn - 1; i >= 0; i--)
      cin >> br[i]; //multiplicand
    for (i = brn - 1; i >= 0; i--)
      mt[i] = br[i];
    complement(mt, brn);
    cout << "\nNo. of multiplier bit=";
    cin >> qrn;
    sc = qrn;
    cout << "Multiplier=";
```

```
  for (i = qrn - 1; i >= 0; i--)
    cin >> qr[i];
  qn = 0;
  temp = 0;
  cout << "qn\tq[n+1]\t\tBR\t\tAC\tQR\t\tsc\n";
  cout << "\t\t\tinitial\t\t";

  display(ac, qr, qrn);
  cout << "\t\t" << sc << "\n";

  while (sc != 0) {
    cout << qr[0] << "\t" << qn;
    if ((qn + qr[0]) == 1) {
      if (temp == 0) {
        add(ac, mt, qrn);
        cout << "\t\tsubtracting BR\t";
        for (i = qrn - 1; i >= 0; i--)
          cout << ac[i];
        temp = 1;
      }
      else if (temp == 1) {
        add(ac, br, qrn);
        cout << "\t\tadding BR\t";
        for (i = qrn - 1; i >= 0; i--)
          cout << ac[i];
        temp = 0;
      }
      cout << "\n\t";
      ashr(ac, qr, qn, qrn);
    }
    else if (qn - qr[0] == 0)
      ashr(ac, qr, qn, qrn);
    display(ac, qr, qrn);
    cout << "\t";
    sc--;
    cout << "\t" << sc << "\n";
  }

cout << "Result=";
display(ac, qr, qrn);}
```

Output:



Conclusion:

Booth's Algorithm is an efficient way to multiply signed binary numbers using shifts and additions. It handles both positive and negative inputs with ease and mimics how real hardware performs multiplication

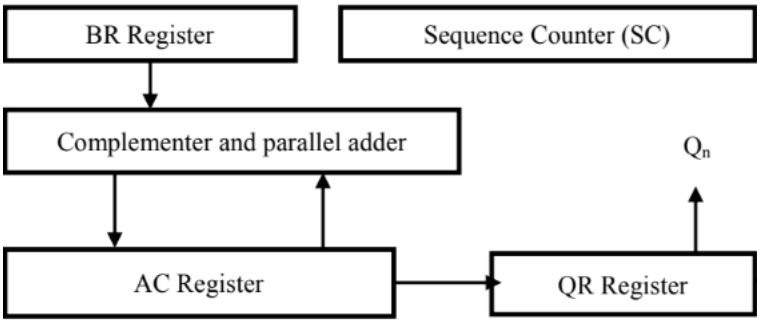
Experiment 5: Restoring Division Algorithm

Objective:

1. To implement Restoring Division Algorithm.

Theory:

Hardware Implementation



Algorithm:

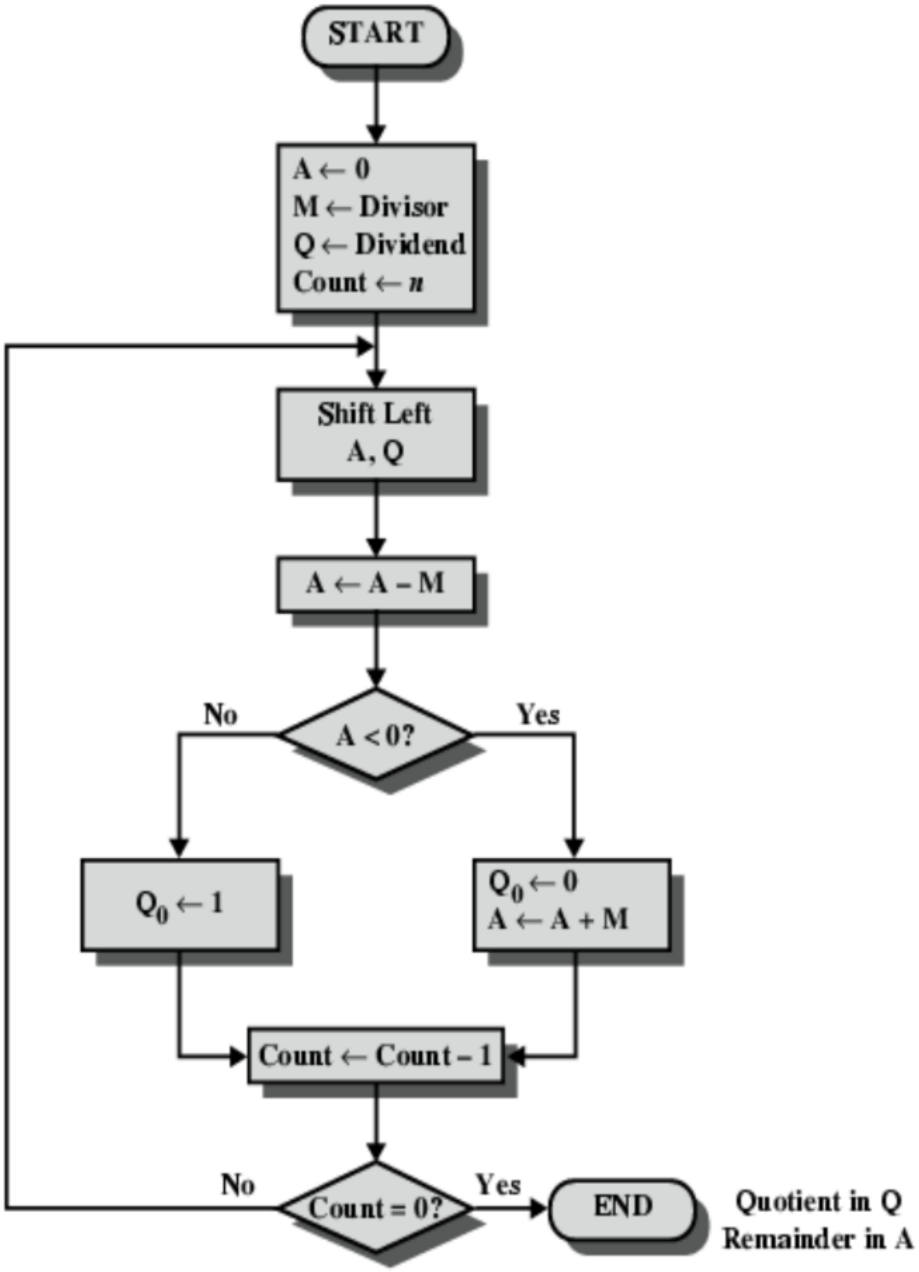
Step 1: Initialize A, Q and M registers to zero, dividend and divisor respectively and counter to n where n is the number of bits in the dividend.

Step 2: Shift A, Q left one binary position.

Step 3: Subtract M from A placing answer back in A. If sign of A is 1, set Q to zero and add M back to A (restore A). If sign of A is 0, set Q to 1.

Step 4: Decrease counter; if counter > 0, repeat process from step 2 else stop the process. The final remainder will be in A and quotient will be in Q.

Flowchart:

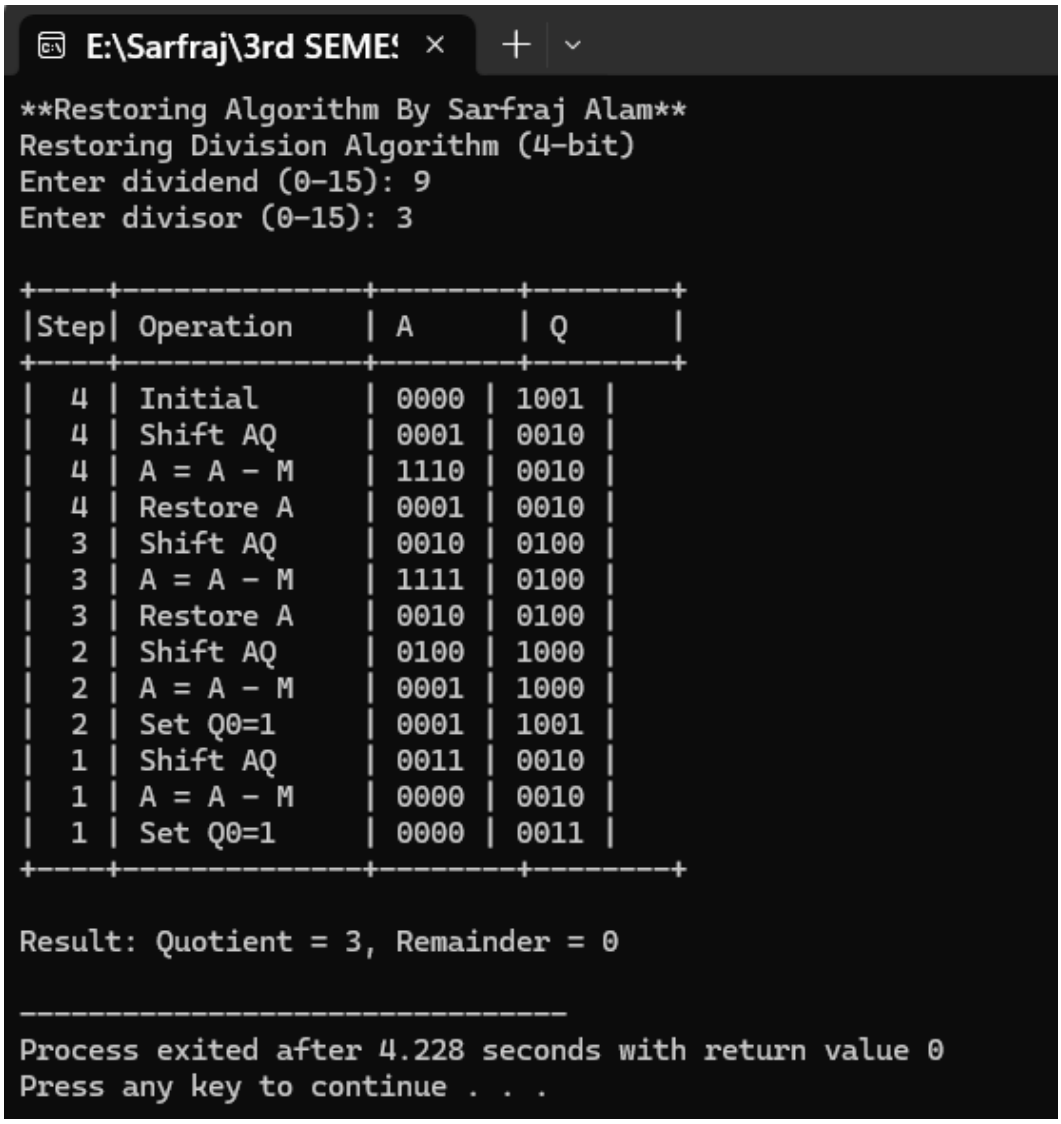

START
A ← 0
M ← Divisor
Q ← Dividend
Count ← n
Shift Left
A, Q
A ← A – M
No
A < 0?
Yes
Q0 ← 1
Q0 ← 0
A ← A + M
Count ← Count – 1
No
Count = 0?
Yes
END
Quotient in Q
Remainder in A

Source code:

```
#include <stdio.h>
void print_binary(int num, int bits) {
    int i;
    for (i = bits - 1; i >= 0; i--) {
        printf("%d", (num >> i) & 1);
    }
}
void print_step(int step, const char* op, int A, int Q, int bits) {
    printf("| %2d | %-12s | ", step, op);
    print_binary(A, bits);
    printf(" | ");
    print_binary(Q, bits);
    printf(" |\n");
}
int main() {
    int dividend, divisor;
    int bits = 4;
    int A, Q, M;
    int step;
    printf("**Restoring Algorithm By Sarfraj Alam**\n");
    printf("Restoring Division Algorithm (4-bit)\n");
    printf("Enter dividend (0-15): ");
    scanf("%d", &dividend);
    printf("Enter divisor (0-15): ");
    scanf("%d", &divisor);
    A = 0;
    Q = dividend;
    M = divisor;
    step = bits;
    printf("\n+----+--------------+--------+--------+\n");
    printf("|Step| Operation    | A      | Q      |\n");
    printf("+----+--------------+--------+--------+\n");
    print_step(step, "Initial", A, Q, bits);

    for (step = bits; step > 0; step--) {
        // Shift AQ left
        A = (A << 1) | ((Q >> (bits-1)) & 1);
        Q <<= 1;
        print_step(step, "Shift AQ", A, Q, bits);

        // Subtract M from A
        A -= M;
        print_step(step, "A = A - M", A, Q, bits);

        if (A < 0) {
            A += M; // Restore
            print_step(step, "Restore A", A, Q, bits);
```

```
        } else {
            Q |= 1; // Set LSB
            print_step(step, "Set Q0=1", A, Q, bits);
        }
    }

    printf("+----+--------------+--------+--------+\n");
    printf("\nResult: Quotient = %d, Remainder = %d\n", Q & 0xF, A & 0xF); // Mask to 4 bits
    return 0;
}
```

Output:



Conclusion:

The Restoring Division Algorithm provides a systematic method to divide binary numbers using shifting and subtraction. It is simple to implement in C and effectively handles unsigned binary division, giving accurate quotient and remainder.