PRACTICAL NO.: 01

Objective: To implement the binary multiplication of two binary numbers.

Theory:

Binary multiplication is one of the four binary arithmetic. The other three fundamental operations are addition, subtraction and division. In the case of a binary operation, we deal with only two digits, i.e 0 and 1. The operation performed while finding the binary product issimilar to the conventional multiplication method. The four major steps in binary digit multiplication are:

- 0 × 0 = 0
- 0 × 1 = 0
- 1 × 0 = 0
- 1 × 1 = 1

Program Code:

```
#include <stdio.h>

int binaryproduct(int binary1, int binary2);

int main() {
    long binary1, binary2, multiply = 0;
    int digit, factor = 1;

    printf("Enter the first binary number: ");
    scanf("%ld", &binary1);

    printf("Enter the second binary number: ");
    scanf("%ld", &binary2);

    while (binary2 != 0) {
        digit = binary2 % 10;
        if (digit == 1) {
            binary1 = binary1 * factor;
            multiply = binaryproduct(binary1, multiply);
        } else {
            binary1 = binary1 * factor;
        }
        binary2 = binary2 / 10;
        factor = 10;
    }

    printf("Product of two binary numbers: %ld", multiply);
    return 0;
}
```

```c
int binaryproduct(int binary1, int binary2) {
    int i = 0, remainder = 0, sum[20];
    int binaryprod = 0;

    while (binary1 != 0 || binary2 != 0) {
        sum[i] = (binary1 % 10 + binary2 % 10 + remainder) % 2;
        remainder = (binary1 % 10 + binary2 % 10 + remainder) / 2;
        binary1 = binary1 / 10;
        binary2 = binary2 / 10;
        i++;
    }

    if (remainder != 0) {
        sum[i] = remainder;
        i--;
    }

    while (i >= 0) {
        binaryprod = binaryprod * 10 + sum[i];
        i--;
    }

    return binaryprod;
}
```

Output:

PRACTICAL NO.: 02

Objective: To convert Hexadecimal number to Decimal number in C programming.

Theory:

Hexadecimal Number System is a base 16 number system that consists of 16 digits ranging from 0-9 and letters A-F where A is equivalent to 10, B is equivalent to 11 up to F which is equivalent to 15 in base ten system. The place value of Hexadecimal number goes up in factorsof sixteen.

- A hexadecimal no. can be denoted using 16 as a subscript or capital letter H to the right of the number. For example, 98B can be written as $98B_{16}$ or 98BH.

Converting hexadecimal numbers to decimal numbers.

To convert hexadecimal no. to base 10 equivalent we proceed as follows:First, write

the place values starting from right hand side.

- If a digit is a letter such as 'A' write its decimal equivalent
- Multiply each hexadecimal number with its corresponding place value and then add the products.
- The following example illustrate how to convert hexadecimal number to decimal number

Convert the hexadecimal number 111_{16} to decimal number. $1*16^2+1*16^1+1*16^0$

256+16+1

273

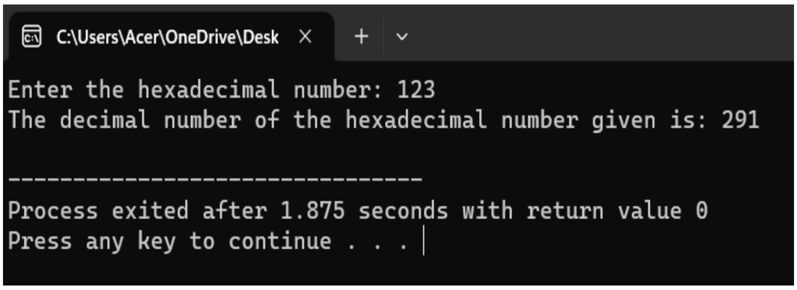
Therefore $111_{16} = 273_{10}$

Program Code:

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

int hexDigitToDecimal(char digit) {
   if (digit >= '0' && digit <= '9')
      return digit - '0';
   else if (digit >= 'A' && digit <= 'F')
      return digit - 'A' + 10;
   else if (digit >= 'a' && digit <= 'f')
      return digit - 'a' + 10;
   else
      return -1;  // Invalid hex digit
}
}
int hexToDecimal(char* hex) {
```

```c
    int length = strlen(hex);
    int decimal = 0;

    for (int i = 0; i < length; i++) {
        int digitValue = hexDigitToDecimal(hex[i]);
        if (digitValue == -1) {
            printf("Invalid Hexadecimal digit '%c' at position %d\n", hex[i], i);
            return -1;
        }
        decimal += digitValue * pow(16, length - i - 1);
    }
    return decimal;
}

int main() {
    char hex[100];
    int decimal;

    printf("Enter the hexadecimal number: ");
    scanf("%s", hex);

    decimal = hexToDecimal(hex);
    if (decimal != -1)
        printf("The decimal number of the hexadecimal number given is: %d\n", decimal);

    return 0;
}
```

Output:



PRACTICAL NO.: 03

Objective: Write a program to add two binary numbers in C programming.

Theory:

A binary number system is one of the four types of number system. In computer applications, where binary numbers are represented by only two symbols or digits, i.e. 0 (zero) and 1(one). The binary numbers here are expressed in the base-2 numeral system. For example, $(101)_2$ is a binary number. Each digit in this system is said to be a bit.

Binary Arithmetic Operation:

In mathematics, the four basic arithmetic operations applied on numbers are addition, subtraction, multiplication & division. In computers, the same operations are performed inside the central processing unit by the arithmetic and logic unit (ALU). However, ALU cannot perform binary subtraction directly. It performs binary subtraction using multiplication and division process, ALU uses a method called shifting before adding the bits.

Binary Addition

The five possible additions in binary are

$0+0=0$

$0+1_2=1_2$

$1_2+0=1_2$

$1_1+1_1=10_2$(read as 0, carry 1)

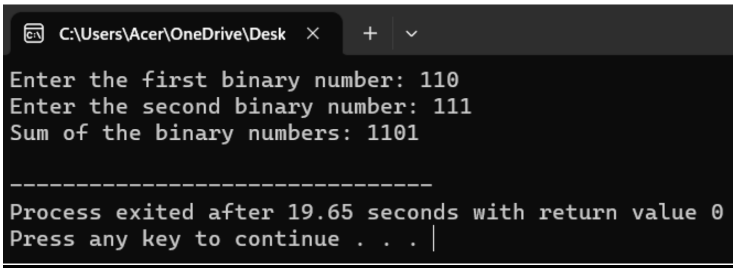
$1_2+1_2+1_2=11_2$(read as 1, carry 1)

Program Code:

```
#include <stdio.h>
#include <math.h>

// Function to add two binary numbers
int addBinary(int a, int b) {
   int carry = 0, result = 0, i = 0;

   while (a || b) {
      int bit_a = a % 10;
      int bit_b = b % 10;
      int sum = bit_a + bit_b + carry;
      carry = sum / 2;
      sum %= 2;
```

```
        result += sum * pow(10, i);
        a /= 10;
        b /= 10;
        i++;
    }

    if (carry) {
        result += carry * pow(10, i);
    }

    return result;
}

int main() {
    int binary1, binary2;

    printf("Enter the first binary number: ");
    scanf("%d", &binary1);

    printf("Enter the second binary number: ");
    scanf("%d", &binary2);

    int sum = addBinary(binary1, binary2);
    printf("Sum of the binary numbers: %d\n", sum);

    return 0;
}
```

Output:



PRACTICAL NO.: 04

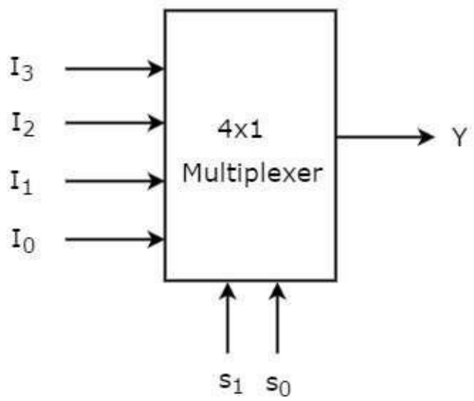
Objective: #Multiplexer

Theory:

Multiplexer is a combinational circuit that has maximum of 2^n data inputs, 'n' selection lines and single output line. One of these data inputs will be connected to the output based on the values of selection lines.

Since there are 'n' selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination will select only one data input. Multiplexer is also called as Mux.

Eg. The block diagram of 4x1 Multiplexer is shown in the following figure.



Program Code:

Filename: mux_example.py

```
from myhdl import block, always_comb, instance, Signal, delay, StopSimulation

@block
def mux_2to1(S, A, B, Y):
    @always_comb
    def logic():
        if S == 0:
            Y.next = A
        else:
            Y.next = B

    return logic
# Instantiate signals for inputs, select, and output
S = Signal(bool(0))
```

```
A = Signal(bool(0))
B = Signal(bool(0))
Y = Signal(bool(0))

# Instantiate the 2-to-1 multiplexer module
mux_inst = mux_2to1(S, A, B, Y)

# Simulation process
@block
def testbench():
    @instance
    def stimulus():
        for s in range(2):
            S.next = s
            A.next = 0
            B.next = 1
            yield delay(10)
            print(f"S = {int(S)}, A = {int(A)}, B = {int(B)}, Y = {int(Y)}")
        raise StopSimulation

    return stimulus

# Simulate the testbench
tb = testbench()
tb.run_sim()
```

Output:

```
[Running] python -u
"c:\Users\Acer\OneDrive\Desktop\Aakash_Shrestha\mux_example.py"
S = 0, A = 0, B = 1, Y = 0
S = 1, A = 0, B = 1, Y = 0

[Done] exited with code=0 in 0.325 seconds
```

PRACTICAL NO.: 05

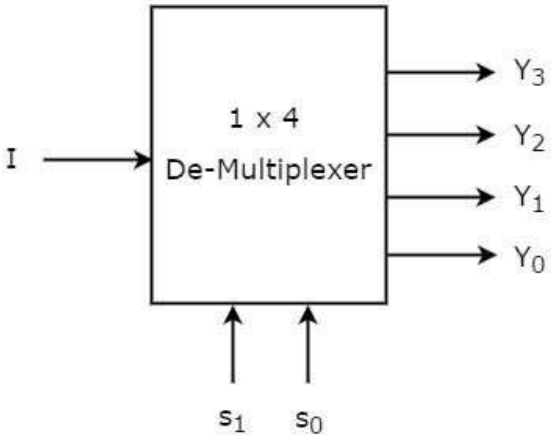
Objective: #Demultiplexer

Theory:

De-Multiplexer is a combinational circuit that performs the reverse operation of Multiplexer. It has single input, 'n' selection lines and maximum of 2^n outputs. The input will be connected to one of these outputs based on the values of selection lines.

Since there are 'n' selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination can select only one output. De-Multiplexer is also called as De-Mux.

Eg. The block diagram of 1x4 De-Multiplexer is shown in the following figure.



Program Code:

filename : demux_example.py

```
from myhdl import block, always_comb, instance, Signal, delay, StopSimulation

@block
def demux_1to2(S, X, Y, Z):
    @always_comb
    def logic():
        if S == 0:
            Y.next = X
            Z.next = 0
        else:
            Y.next = 0
            Z.next = X

    return logic

# Instantiate signals for select, input, and outputs
S = Signal(bool(0))
X = Signal(bool(0))
```

```
Y = Signal(bool(0))
Z = Signal(bool(0))

# Instantiate the 1-to-2 demultiplexer module
demux_inst = demux_1to2(S, X, Y, Z)

# Simulation process
@block
def testbench():
    @instance
    def stimulus():
        for s in range(2):
            S.next = s
            X.next = 1

            yield delay(10)
            print(f"S = {int(S)}, X = {int(X)}, Y = {int(Y)}, Z = {int(Z)}")

        raise StopSimulation

    return stimulus

# Simulate the testbench
tb = testbench()
tb.run_sim()
```

Output:

```
[Running] python -u
"c:\Users\Acer\OneDrive\Desktop\Aakash_Shrestha\demux_example.py"
S = 0, X = 1, Y = 0, Z = 0
S = 1, X = 1, Y = 0, Z = 0

[Done] exited with code=0 in 0.495 seconds
```

PRACTICAL NO.: 06

Objective: #ALU Simulation

Theory:

An arithmetic logic unit (ALU) is a key component of a computer's central processor unit. The ALU performs all arithmetic and logic operations that must be performed on instruction words. The ALU is split into two parts in some microprocessor architectures: the AU and the LU.

It can execute all arithmetic and logic operations, including Boolean comparisons, such assubtraction, addition, and shifting (XOR, OR, AND, and NOT operations).

Program Code:

ALU Simulation: programming code forfile name : alu_example.py

```
from myhdl import block, always_comb, instance, Signal, intbv, delay, StopSimulation

# ALU operations
ALU_OP_ADD = 0
ALU_OP_SUB = 1
ALU_OP_AND = 2
ALU_OP_OR = 3
@block
def alu(ALUOp, A, B, Result):
    @always_comb
    def logic():
        if ALUOp == ALU_OP_ADD:
            Result.next = A + B
        elif ALUOp == ALU_OP_SUB:
            Result.next = A - B
elif ALUOp == ALU_OP_AND:
            Result.next = A & B
        elif ALUOp == ALU_OP_OR:
            Result.next = A | B

    return logic
# Instantiate signals for ALU operation, inputs, and result
ALUOp = Signal(intbv(0, min=0, max=4))
A = Signal(intbv(0, min=0, max=256))
B = Signal(intbv(0, min=0, max=256))
```

```
Result = Signal(intbv(0, min=0, max=512))

# Instantiate the ALU module
alu_inst = alu(ALUOp, A, B, Result)
# Simulation process
@block
def testbench():
    @instance
    def stimulus():
        for op in range(4):
            for a in range(256):
                for b in range(256):
                    ALUOp.next = op
                    A.next = intbv(a)[8:]  # Ensure value is within bounds
                    B.next = intbv(b)[8:]  # Ensure value is within bounds
                    yield delay(10)
                    print(f"ALUOp = {ALUOp}, A = {A}, B = {B}, Result = {Result}")

        raise StopSimulation

    return stimulus

# Simulate the testbench
tb = testbench()
tb.run_sim()

from myhdl import block, always_comb, intbv ,instance, Signal, delay, StopSimulation
# Define floating-point representation
EXP_BITS = 4
FRAC_BITS = 8
@block
def float_add(A_sign, A_exp, A_frac, B_sign, B_exp, B_frac, Result_sign, Result_exp, Result_frac):
    @always_comb
    def logic():
        # Perform floating-point addition
        if A_exp > B_exp:
            Result_exp.next = A_exp
            Result_frac.next = A_frac + (B_frac >> (A_exp - B_exp))
        else:
            Result_exp.next = B_exp
            Result_frac.next = B_frac + (A_frac >> (B_exp - A_exp))

        Result_sign.next = A_sign  # Assume A is positive
    return logic
# Instantiate signals for inputs and result
```

```
A_sign = Signal(bool(0))
A_exp = Signal(intbv(0, min=0, max=2**EXP_BITS))
A_frac = Signal(intbv(0, min=0, max=2**FRAC_BITS))
B_sign = Signal(bool(0))
B_exp = Signal(intbv(0, min=0, max=2**EXP_BITS))
B_frac = Signal(intbv(0, min=0, max=2**FRAC_BITS))
Result_sign = Signal(bool(0))
Result_exp = Signal(intbv(0, min=0, max=2**EXP_BITS))
Result_frac = Signal(intbv(0, min=0, max=2**FRAC_BITS))

# Instantiate the floating-point addition module
float_add_inst = float_add(A_sign, A_exp, A_frac, B_sign, B_exp, B_frac, Result_sign, Result_exp,
Result_frac)

# Simulation process
@block
def testbench():
    @instance
    def stimulus():
        for a_exp in range(1, 2**EXP_BITS):
            for b_exp in range(1, 2**EXP_BITS):
                for a_frac in range(2**FRAC_BITS):
                    for b_frac in range(2**FRAC_BITS):
                        A_sign.next = 0
                        A_exp.next = a_exp
                        A_frac.next = a_frac
                        B_sign.next = 0
                        B_exp.next = b_exp
                        B_frac.next = b_frac
                        yield delay(10)
                        print(f"A = {A_sign} {A_exp} {A_frac}, B = {B_sign} {B_exp} {B_frac}, Result = {Result_sign}
{Result_exp} {Result_frac}")
        raise StopSimulation

    return stimulus

# Simulate the testbench
tb = testbench()
tb.run_sim()
```

Output:

```
Enter the number of processes: 3

Enter the process ID: 1
Enter the burst time: 6
Enter the priority: 1

Enter the process ID: 2
Enter the burst time: 10
Enter the priority: 2

Enter the process ID: 3
Enter the burst time: 5
Enter the priority: 0

Process ID      Burst Time      Priority        Waiting Time    Turnaround Time
3               5               0               0               5
1               6               1               5               11
2               10              2               11              21

Average Waiting Time = 5.333333
Average Turnaround Time = 12.333333
```

```
A = 0, B = 0, Cin = 0, Sum = 0, Cout =
A = 1, B = 0, Cin = 0, Sum = 0, Cout =
A = 0, B = 1, Cin = 0, Sum = 0, Cout =
A = 1, B = 1, Cin = 0, Sum = 0, Cout =
A = 0, B = 0, Cin = 1, Sum = 0, Cout =
A = 1, B = 0, Cin = 1, Sum = 0, Cout =
A = 0, B = 1, Cin = 1, Sum = 0, Cout =
A = 1, B = 1, Cin = 1, Sum = 0, Cout =
```

```
[Running] python -u
"c:\Users\Acer\OneDrive\Desktop\Aakash_Sh
restha\alu_example.py"
ALUOp = 0, A = 00, B = 00, Result = 000
ALUOp = 0, A = 00, B = 01, Result = 000
ALUOp = 0, A = 00, B = 02, Result = 000
ALUOp = 0, A = 00, B = 03, Result = 000
ALUOp = 0, A = 00, B = 04, Result = 000
ALUOp = 0, A = 00, B = 05, Result = 000
ALUOp = 0, A = 00, B = 06, Result = 000
ALUOp = 0, A = 00, B = 07, Result = 000
ALUOp = 0, A = 00, B = 08, Result = 000
ALUOp = 0, A = 00, B = 09, Result = 000
ALUOp = 0, A = 00, B = 0a, Result = 000
ALUOp = 0, A = 00, B = 0b, Result = 000
ALUOp = 0, A = 00, B = 0c, Result = 000
ALUOp = 0, A = 00, B = 0d, Result = 000
ALUOp = 0, A = 00, B = 0e, Result = 000
ALUOp = 0, A = 00, B = 0f, Result = 000
ALUOp = 0, A = 00, B = 10, Result = 000
ALUOp = 0, A = 00, B = 11, Result = 000
ALUOp = 0, A = 00, B = 12, Result = 000
ALUOp = 0, A = 00, B = 13, Result = 000
ALUOp = 0, A = 00, B = 14, Result = 000
ALUOp = 0, A = 00, B = 15, Result = 000
ALUOp = 0, A = 00, B = 16, Result = 000
ALUOp = 0, A = 00, B = 17, Result = 000
ALUOp = 0, A = 00, B = 18, Result = 000
ALUOp = 0, A = 00, B = 19, Result = 000
ALUOp = 0, A = 00, B = 1a, Result = 000
ALUOp = 0, A = 00, B = 1b, Result = 000
ALUOp = 0, A = 00, B = 1c, Result = 000
ALUOp = 0, A = 00, B = 1d, Result = 000
ALUOp = 0, A = 00, B = 1e, Result = 000
ALUOp = 0, A = 00, B = 1f, Result = 000
ALUOp = 0, A = 00, B = 20, Result = 000
ALUOp = 0, A = 00, B = 21, Result = 000
ALUOp = 0, A = 00, B = 22, Result = 000
```

```
[Running] python -u
"c:\Users\Acer\OneDrive\Desktop\Aakash_Shrestha\float
_add_example.py"
A = False 1 00, B = False 1 00, Result = False 0 00
A = False 1 00, B = False 1 01, Result = False 0 00
A = False 1 00, B = False 1 02, Result = False 0 00
A = False 1 00, B = False 1 03, Result = False 0 00
A = False 1 00, B = False 1 04, Result = False 0 00
A = False 1 00, B = False 1 05, Result = False 0 00
A = False 1 00, B = False 1 06, Result = False 0 00
A = False 1 00, B = False 1 07, Result = False 0 00
A = False 1 00, B = False 1 08, Result = False 0 00
A = False 1 00, B = False 1 09, Result = False 0 00
A = False 1 00, B = False 1 0a, Result = False 0 00
A = False 1 00, B = False 1 0b, Result = False 0 00
A = False 1 00, B = False 1 0c, Result = False 0 00
A = False 1 00, B = False 1 0d, Result = False 0 00
A = False 1 00, B = False 1 0e, Result = False 0 00
A = False 1 00, B = False 1 0f, Result = False 0 00
A = False 1 00, B = False 1 10, Result = False 0 00
A = False 1 00, B = False 1 11, Result = False 0 00
A = False 1 00, B = False 1 12, Result = False 0 00
A = False 1 00, B = False 1 13, Result = False 0 00
A = False 1 00, B = False 1 14, Result = False 0 00
A = False 1 00, B = False 1 15, Result = False 0 00
A = False 1 00, B = False 1 16, Result = False 0 00
A = False 1 00, B = False 1 17, Result = False 0 00
A = False 1 00, B = False 1 18, Result = False 0 00
A = False 1 00, B = False 1 19, Result = False 0 00
A = False 1 00, B = False 1 1a, Result = False 0 00
A = False 1 00, B = False 1 1b, Result = False 0 00
A = False 1 00, B = False 1 1c, Result = False 0 00
A = False 1 00, B = False 1 1d, Result = False 0 00
A = False 1 00, B = False 1 1e, Result = False 0 00
A = False 1 00, B = False 1 1f, Result = False 0 00
A = False 1 00, B = False 1 20, Result = False 0 00
A = False 1 00, B = False 1 21, Result = False 0 00
A = False 1 00, B = False 1 22, Result = False 0 00
```

PRACTICAL NO.: 07

Objective: #Booth Multiplication

Theory:

Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement representation in efficient way, i.e., less number of additions/subtractions required. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{(k+1)}$to 2^{m}.

Hardware Implementation of Booths Algorithm – The hardware implementation of the booth algorithm requires the register configuration shown in the figure below.

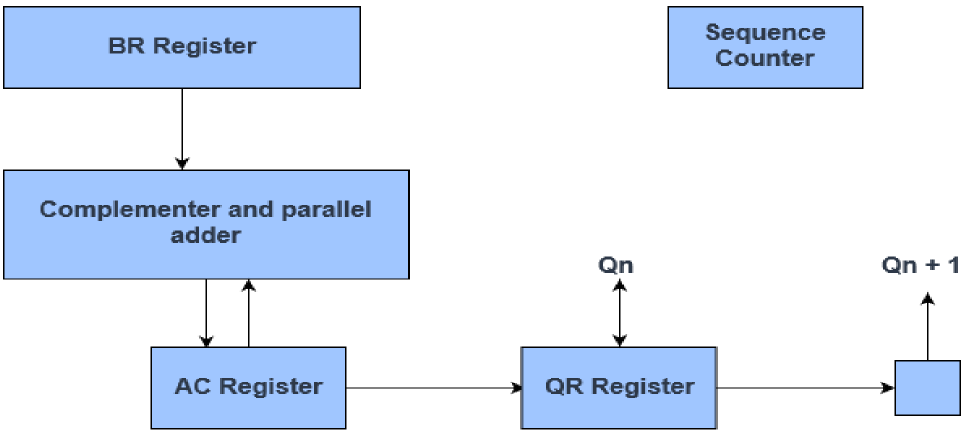


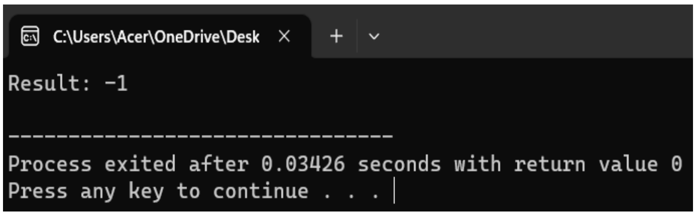
Fig. Hardware for booth algorithm

Program Code:

```
#include <stdio.h>

int boothMultiplication(int multiplicand, int multiplier) {
    int A = 0;              // Accumulator
    int Q = multiplier;     // Multiplier
    int M = multiplicand;   // Multiplicand
    int Qn_1 = 0;           // Extension bit
    int i, bit;

    for (i = 0; i < 4; i++) {  // Assuming 4-bit binary representation
        bit = Q & 1;          // Get the least significant bit of Q
        if (bit != Qn_1) {
            A = A + (bit ? M : -M);
```

```
}
        Qn_1 = bit;
        Q = (Q >> 1) | (Qn_1 << 3);  // Right shift Q and insert Qn_1 at MSB
        A >>= 1;                    // Right shift A
    }

    return A;
}

int main() {
    int multiplicand = -4;    // Binary: 1100
    int multiplier = -3;      // Binary: 1101

    int result = boothMultiplication(multiplicand, multiplier);
    printf("Result: %d\n", result);

    return 0;
}
```

Output:



Program Code:

```
#include <stdio.h>

int boothMultiplication(int multiplicand, int multiplier) {
    int A = 0;              // Accumulator
    int Q = multiplier;     // Multiplier
    int M = multiplicand;   // Multiplicand
    int Qn_1 = 0;           // Extension bit
    int i, bit;

    for (i = 0; i < 8; i++) {  // Using 8-bit representation
        bit = Q & 1;          // Get the least significant bit of Q
        if (bit != Qn_1) {
```

```
            A = A + (bit ? M : -M);
        }

 return A;
 }

 int main() {
    int multiplicand = 0b00101111;  // Binary: 00101111 (decimal: 47)
    int multiplier = 0b11110111;   // Binary: 11110111 (decimal: -9)

    int result = boothMultiplication(multiplicand, multiplier);
    printf("Result: %d\n", result);

    return 0;
}
```

Output:

```
C:\Users\Acer\OneDrive\Desk
Result: 1

--------------------------------
Process exited after 0.02767 seconds with return value 0
Press any key to continue . . .
```