PROJECT REPORT

ON

BOOTH'S ALGORITHM

Submitted

By

VAKKANTI LAKSHMI REVANTH KUMAR – U202405204810354

PATHAN VASEEMA KHAN – U202405204816359

BATCH – PGDVD 18

DEPARTMENT OF PG DIPLOMA IN VLSI DESIGN AND VERIFICATION

CRANES VARSITY

BANGALORE, KARNATAKA, INDIA - 560001

OCTOBER - 2024

TABLE OF CONTENTS

ABSTRACT

Booth's multiplier has revolutionized digital multiplication by providing a faster and more area-efficient solution. The algorithm's effectiveness stems from its ability to recode the multiplier into a signed-digit representation, allowing for the elimination of redundant partial products. This results in a significant reduction in computational complexity, making Booth's multiplier an attractive choice for resource-constrained applications. Furthermore, its scalability enables seamless integration into various digital systems, from low-power embedded devices to high-performance computing architectures. The multiplier's simplicity also facilitates easier implementation and verification, reducing design time and increasing reliability. With its widespread adoption, Booth's multiplier has become a cornerstone of modern digital design, empowering developers to create innovative solutions that demand high-speed multiplication. Its impact extends across diverse fields, including cryptography, digital signal processing, and artificial intelligence, where efficient multiplication is crucial. As technology continues to evolve, Booth's multiplier remains an indispensable tool, driving advancements in digital systems and application.

INTRODUCTION

Booth's algorithm is a notable method in computer arithmetic for efficiently performing binary multiplication, particularly with signed integers. Introduced by Andrew Booth in 1951, the algorithm leverages a series of arithmetic shifts and conditional additions or subtractions to minimize the number of operations required to compute the product of two numbers. Its efficiency and simplicity make it especially valuable in hardware implementations like CPUs and digital signal processors (DSPs), where it saves processing time and circuit complexity. Booth's algorithm is well-regarded for its ability to handle both positive and negative numbers seamlessly, which is essential for many applications in computing and digital electronics.

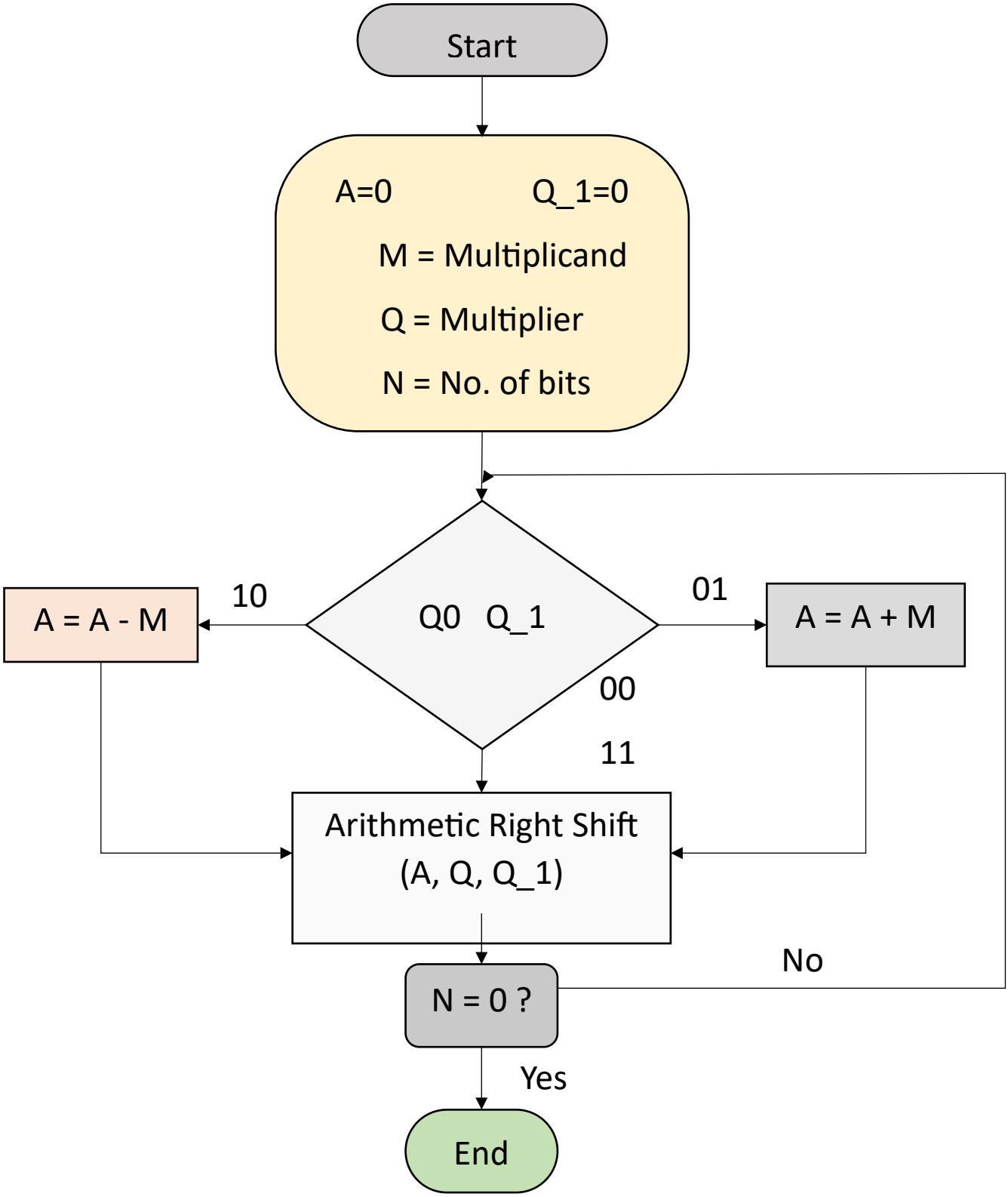
The core idea behind Booth's algorithm is to reduce the number of addition and subtraction operations required during multiplication by encoding sequences of identical bits in the multiplier. In traditional binary multiplication, each bit of the multiplier requires either adding the multiplicand to a partial product or performing no operation. However, Booth's algorithm groups consecutive 1's in the multiplier, allowing it to represent them in fewer steps. Specifically, it uses pairs of bits from the multiplier at a time to determine the operation: 00 means no operation and a shift; 01 means adding the multiplicand to the accumulator; 10 means subtracting the multiplicand; and 11 requires no operation and a shift as well. This system helps minimize the additions and subtractions, speeding up the multiplication process, particularly when the multiplier has long sequences of 1's or 0's.

The process of Booth's algorithm can be broken down into several steps. First, the multiplicand and multiplier are represented in signed binary form (usually in 2's complement format for hardware compatibility). Three main registers are typically used: the accumulator (A), where the intermediate product accumulates; the multiplier register (Q), which holds the bits of the multiplier; and a one-bit flag (Q-1) to keep track of the previous bit of the multiplier. The algorithm then proceeds by examining the pair of the least significant bit of Q and the Q-1 bit. Based on this pair, Booth's algorithm either adds, subtracts, or shifts the contents of A and Q together. Each iteration, the combined A and Q registers undergo an arithmetic right shift, propagating the sign bit to maintain the correct signed value in A. This continues until all bits of the multiplier have been processed, resulting in the desired product in the combined A and Q registers.

OBJECTIVE

Booth's Algorithm is to efficiently multiply two signed binary numbers by minimizing the number of addition and subtraction operations required. This approach optimizes performance, especially in digital circuits and computer arithmetic, where processing speed and resource usage are critical. By analysing patterns in the multiplier bits, Booth's Algorithm determines whether to add, subtract, or skip operations based on the combination of the current and previous bits. This technique leverages arithmetic right shifts and conditional operations to simplify the multiplication process, making it faster and more efficient compared to straightforward binary multiplication. As a result, Booth's Algorithm is particularly beneficial in hardware implementations, where reduced computational steps lead to lower power consumption, faster execution, and optimized circuit design. This efficiency makes it an essential algorithm in the field of digital signal processing, computer graphics, and other applications requiring rapid and accurate signed binary multiplication.

FLOW CHART


Start
A=0
Q_1=0
M = Multiplicand
Q = Multiplier
N = No. of bits
Q0 Q_1
10
01
00
11
A = A - M
A = A + M
Arithmetic Right Shift
(A, Q, Q_1)
N = 0 ?
No
Yes
End

FLOW CHART EXPLANATION

The flowchart illustrates **Booth's Algorithm** for binary multiplication, specifically for multiplying two signed binary numbers. Booth's Algorithm is effective for handling both positive and negative multipliers and multiplicands. Here's a breakdown of the flowchart:

1. **Initialization:**
    - **A = 0**: Accumulator initialized to 0.
    - **Q**: Represents the multiplier.
    - **Q_1**: Extra bit initialized to 0, used to keep track of the last bit of the multiplier.
    - **M**: Represents the multiplicand.
    - **N**: Number of bits in the multiplier.
2. **Decision Making Based on Q0 and Q_1:**
    - The algorithm checks the last bit of the multiplier (Q0) and the extra bit (Q_1) to determine the next steps:
        - **10 (Q0 = 0, Q_1 = 1)**: Perform A=A−M.
        - **01 (Q0 = 1, Q_1 = 0)**: Perform A=A+M.
        - **00 or 11**: No addition or subtraction; proceed to the next step.
3. **Arithmetic Right Shift (A, Q, Q_1):**
    - After the addition or subtraction (if any), perform an arithmetic right shift on the combined A, Q, Q−1. This means shifting the bits to the right to make room for the next operation.
    - This step decreases the bit count N by 1.
4. **Loop until N = 0:**
    - If N≠0, repeat the process starting from the decision step.
    - If N=0, the algorithm ends, and the result of the multiplication is stored in the combined A and Q registers.
5. **End of Algorithm:**
    - Once N=0, the process terminates, and the product of the multiplication is represented by the combined values in A and Q.

EXAMPLE

=> -3 * 5 = -15 (4 – bit)

M = -3 = 1 1 0 1 Q = 0 1 0 1

-M = - (-3) = 3 = 0 0 1 1

A	Q (Q3 Q2 Q1 Q0)	Q_1	Operation
0 0 0 0	0 1 0 1	0	Initial
0 0 1 1 0 0 0 1	0 1 0 1 1 0 1 0	0 1	A = A – M Right Shift
1 1 1 0 1 1 1 1	1 0 1 0 0 1 0 1	1 0	A = A + M Right Shift
0 0 1 0 0 0 0 1	0 1 0 1 0 0 1 0	0 1	A = A – M Right Shift
1 1 1 0 1 1 1 1	0 0 1 0 0 0 0 1	1 0	A = A + M Right Shift

The result is in 2's Complimentary form

If you want original number, again you can do 2's compliment

1 1 1 1 0 0 0 1

0 0 0 0 1 1 1 0

+ 1

0 0 0 0 1 1 1 1

To illustrate with an example, consider multiplying -3 by 5 using Booth's algorithm. In binary, -3 is represented as 1101 (using 4-bit 2's complement), and 5 as 0101. Starting with A and Q-1 initialized to 0, Booth's algorithm performs a series of shifts and conditional additions or subtractions based on the bits of the multiplier. By the end of the algorithm, the combined contents of A and Q yield the final product, which in this case would be -15, demonstrating Booth's algorithm's ability to handle signed numbers efficiently.

RESULTS

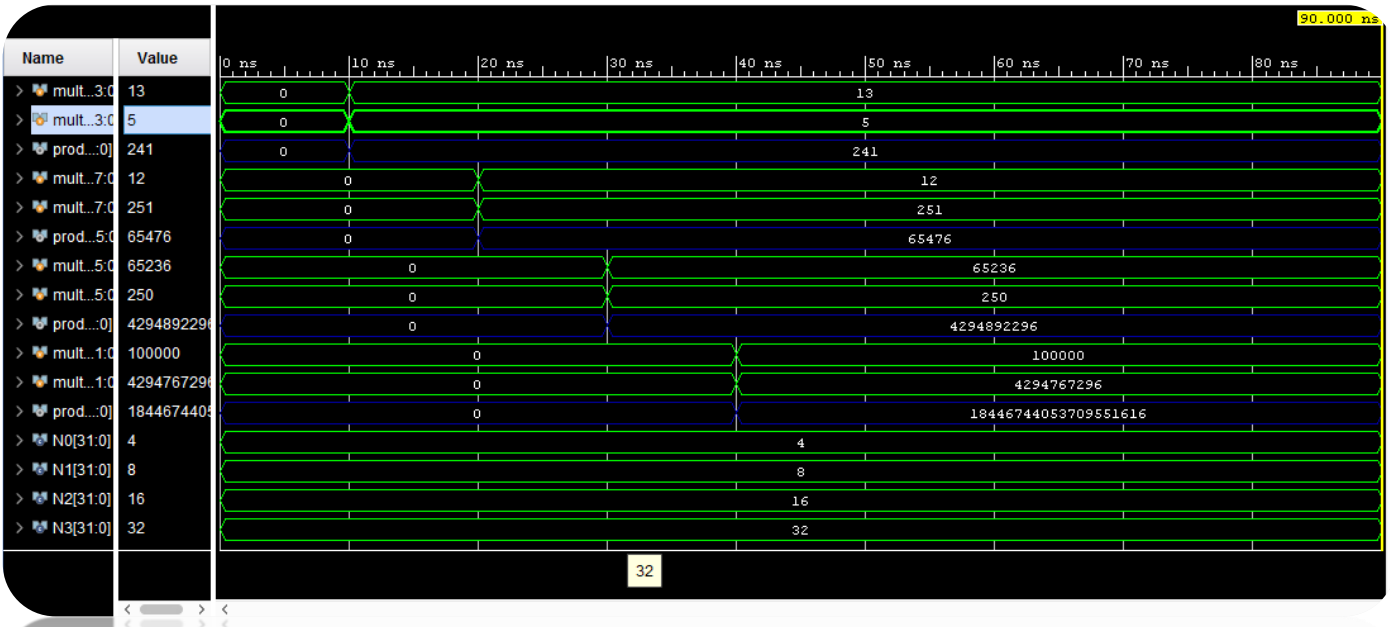


Fig: Output Wave Form for 4,8,16,32 - Bits

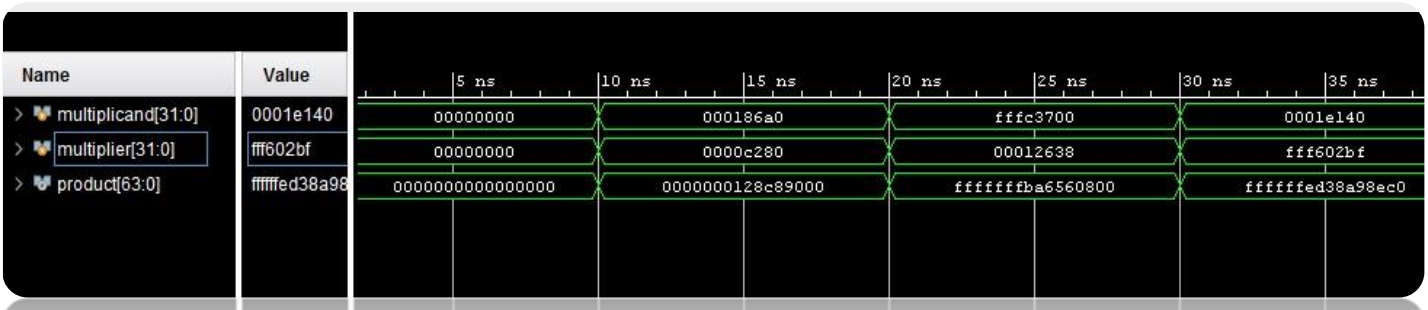


Fig: Output Wave Form for 32 - Bits

ADVANTAGES AND APPLICATIONS

ADVANTAGES:

- Efficient handling of signed numbers
- Faster Multiplication
- Less Complexity
- Flexible in scaling
- Simplicity in implementation

APPLICATIONS:

- Digital signal processing
- Microcontrollers and embedded systems
- Computer graphics
- Image processing
- Telecommunications
- Control systems

CONCLUSION

Booth's Algorithm is a highly efficient approach for binary multiplication, especially advantageous when working with signed numbers. The algorithm reduces the total number of addition and subtraction operations needed by using a systematic examination of the multiplier's bits. By grouping sequences of identical bits, it identifies patterns that allow certain operations to be skipped entirely. This results in fewer computational steps, which translates to faster processing times and greater efficiency—essential qualities in digital arithmetic. This optimization not only improves speed but also reduces power consumption, making it particularly useful in processor and digital circuit design. In applications where high-speed arithmetic is crucial, such as graphics processing, embedded systems, and real-time computation, Booth's Algorithm stands out as a valuable tool, balancing accuracy with performance. Its ability to handle signed numbers without additional complexity further solidifies its importance in the design and functioning of modern digital systems.

REFERENCES

[1] Chengdong Liang, Lijuan Su, Jinzhao Wu and Juxia Xiong, "An innovative Booth algorithm," 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), Xi'an, China, 2016, pp. 1711-1715, doi: 10.1109/IMCEC.2016.7867510. keywords: {Algorithm design and analysis;Encoding;Delays;Software algorithms;Encryption;Field programmable gate arrays;Complexity theory;Large-number Multiplier;Modified Booth algorithm;RSA;FPGA},

[2] A. S. Tariq, R. Amin, M. N. I. Mondal and M. A. Hossain, "Faster implementation of Booth's algorithm using FPGA," 2016 2nd International Conference on Electrical, Computer & Telecommunication Engineering (ICECTE), Rajshahi, Bangladesh, 2016, pp. 1-4, doi: 10.1109/ICECTE.2016.7879580. keywords: {Field programmable gate arrays;Algorithm design and analysis;Computers;Central Processing Unit;Hardware design languages;Communications technology;Computer architecture;FPGA;Booth's algorithm;signed binary multiplication;speed-up factor;time efficieny;CPU},

[3] Manjunath, V. Harikiran, K. Manikanta, S. Sivanantham and K. Sivasankaran, "Design and implementation of 16×16 modified booth multiplier," 2015 Online International Conference on Green Engineering and Technologies (IC-GET), Coimbatore, India, 2015, pp. 1-5, doi: 10.1109/GET.2015.7453817. keywords: {Adders;Signal processing algorithms;Digital signal processing;Delays;Multimedia communication;Generators;Modified Booth encoder;Booth decode;Wallace tree},

[4] P. E. Madrid, B. Millar and E. E. Swartzlander, "Modified Booth algorithm for high radix multiplication," Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors, Cambridge, MA, USA, 1992, pp. 118-121, doi: 10.1109/ICCD.1992.276194. keywords: {Drives;Random access memory;Ferroelectric films;Nonvolatile memory;Arithmetic;Peak to average power ratio},

[5] B. Jeevan, P. Samskruthi, P. Sahithi and K. Sivani, "Implementation of parallel multiplier based on Booth computing method using FPGA," 2022 International Conference on Advances in Computing, Communication and Applied Informatics (ACCAI), Chennai, India, 2022, pp. 1-8, doi: 10.1109/ACCAI53970.2022.9752479. keywords: {Computer architecture;Very large scale integration;Encoding;Software;Delays;Hardware design languages;Informatics;Booth encoding;partial product reduction tree;FPGA;digital VLSI},

[6] https://vlsiverify.com/verilog/verilog-codes/booth-multiplier/

APPENDIX

```
// N-bit booths_Algorithm

module booth_multiplier_Nbit #(parameter N = 8)(
    input signed [N-1:0] multiplicand,
    input signed [N-1:0] multiplier,
    output reg signed [2*N-1:0] product
);
    reg [N-1:0] A, Q, M;
    reg Q_1;
    integer i;

    always @(*) begin
        A = {N{1'b0}};
        Q = multiplier;
        M = multiplicand;
        Q_1 = 1'b0;
        product = {2*N{1'b0}};

        for (i = 0; i < N; i = i + 1) begin
            case ({Q[0], Q_1})
                2'b01: A = A + M;
                2'b10: A = A - M;
                default: ;
            endcase

            {A, Q, Q_1} = {A[N-1], A, Q};
        end

        product = {A, Q};
    end
endmodule
```

```
// Test Bench
module booth_multiplier_Nbit_tb;
    parameter N0 = 4;
    parameter N1 = 8;
    parameter N2 = 16;
    parameter N3 = 32;
    reg signed [N0-1:0] multiplicand4;
    reg signed [N0-1:0] multiplier4;
    wire signed [2*N0-1:0] product4;

    reg signed [N1-1:0] multiplicand8;
    reg signed [N1-1:0] multiplier8;
    wire signed [2*N1-1:0] product8;

    reg signed [N2-1:0] multiplicand16;
    reg signed [N2-1:0] multiplier16;
    wire signed [2*N2-1:0] product16;

    reg signed [N3-1:0] multiplicand32;
    reg signed [N3-1:0] multiplier32;
    wire signed [2*N3-1:0] product32;

    booth_multiplier_Nbit #(.N(N0)) uut4 (
        .multiplicand(multiplicand4),
        .multiplier(multiplier4),
        .product(product4)
    );

    booth_multiplier_Nbit #(.N(N1)) uut8 (
        .multiplicand(multiplicand8),
        .multiplier(multiplier8),
        .product(product8)  );
```

```
    booth_multiplier_Nbit #(.N(N2)) uut16 (
        .multiplicand(multiplicand16),
        .multiplier(multiplier16),
        .product(product16)
    );

    booth_multiplier_Nbit #(.N(N3)) uut32 (
        .multiplicand(multiplicand32),
        .multiplier(multiplier32),
        .product(product32)
    );

    initial begin

            multiplicand4 = 4'd0; multiplier4 = 4'd0;
            multiplicand8 = 8'd0; multiplier8 = 8'd0;
            multiplicand16 = 16'd0; multiplier16 = 16'd0;
            multiplicand32 = 32'd0; multiplier32 = 32'd0;

        #10 multiplicand4 = -4'd3;
            multiplier4 = 4'd5;
            $display("4-bit Multiplicand: %d, Multiplier: %d, Product: %d", multiplicand4,
multiplier4, product4);

        #10 multiplicand8 = 8'd12;
            multiplier8 = -8'd5;
            $display("8-bit Multiplicand: %d, Multiplier: %d, Product: %d", multiplicand8,
multiplier8, product8);

        #10 multiplicand16 = -16'd300;
            multiplier16 = 16'd250;
            $display("16-bit Multiplicand: %d, Multiplier: %d, Product: %d", multiplicand16,
multiplier16, product16);
```

```
        #10 multiplicand32 = 32'd100000;
            multiplier32 = -32'd200000;
            $display("32-bit Multiplicand: %d, Multiplier: %d, Product: %d", multiplicand32,
multiplier32, product32);


        #50 $stop;
    end
endmodule
```