SRM Institute of Science and Technology
College of Engineering and Technology

Electronics and Communication Engineering

MINI PROJECT REPORT

ODD Semester: 2024-2025

Lab code & Sub Name : 21CSS201T & Computer Organization and Architecture

Year & Semester : II & III

Project Title : Performance analysis of booth alternate, bit pair recoding, booth, shift add multiplier for 8-bit input

Team Members:
1. A. Avinash Sastry (RA2311004010007)
2. Emmanuel Joe (RA2311004010014)
3. Adithya (RA23110040100016)
4. Ashwin Krishnan (RA2311004010017)

Particulars	**Max. Marks**	**Marks Obtained** **Name: A. Avinash Sastry** **Register No: RA2311004010007**
Review 1 and 2	05	
Demo verification &viva	03	
Project Report	02	
Total	**10**	

Date :

Staff Name :

Signature :

Performance analysis of booth alternate, bit pair recoding, booth, shift add multiplier for 8-bit input

OBJECTIVE:

The objective of this code is to multiply two 8 bit-binary numbers and verifying the result, and also comparing the various performance standards (like device utilization summary, time taken, cell usage and many more) and finding out which was the ideal multiplier.

ABSTRACT:

Scientific calculators and computers took the world by storm during the late 1970s and 1990s, and a big part as to how these devices helped human life was reducing time taken for doing tedious tasks and performing simple arithmetic and logical operations. Multiplying two numbers is basic mathematics taught to primary school students, but multiplying large numbers like 3 digit or even 4-digit numbers is tedious and time consuming, which is we calculators and computers were created to perform these tasks for us. Through this project, we will be showing how a computer multiplies these two numbers by accepting these numbers in binary format, the various methods it uses and which is the most suitable method of them all.

INTRODUCTION:

The project, Performance Analysis of Booth multiplier, Booth Alternate multiplier, Bit Pair Recoding multiplier and Shift-Add Multipliers for 8-bit Inputs, focuses on evaluating the efficiency of various multiplication algorithms for digital circuits[1]. Multiplication is a fundamental operation in many computing tasks, and efficient multiplier designs are crucial for improving the performance of processors, especially in areas like signal processing and embedded systems [2,3]. In this project, four different multiplication techniques are analyzed:

- Booth Algorithm
- Bit-pair Recoding Algorithm
- Shift Add multiplier
- Booth Alternate multiplier

HARDWARE/SOFTWARE REQUIREMENTS:

- Programming languages used: Verilog
- Software used: QuestaSim

CONCEPTS/WORKING PRINCIPLE

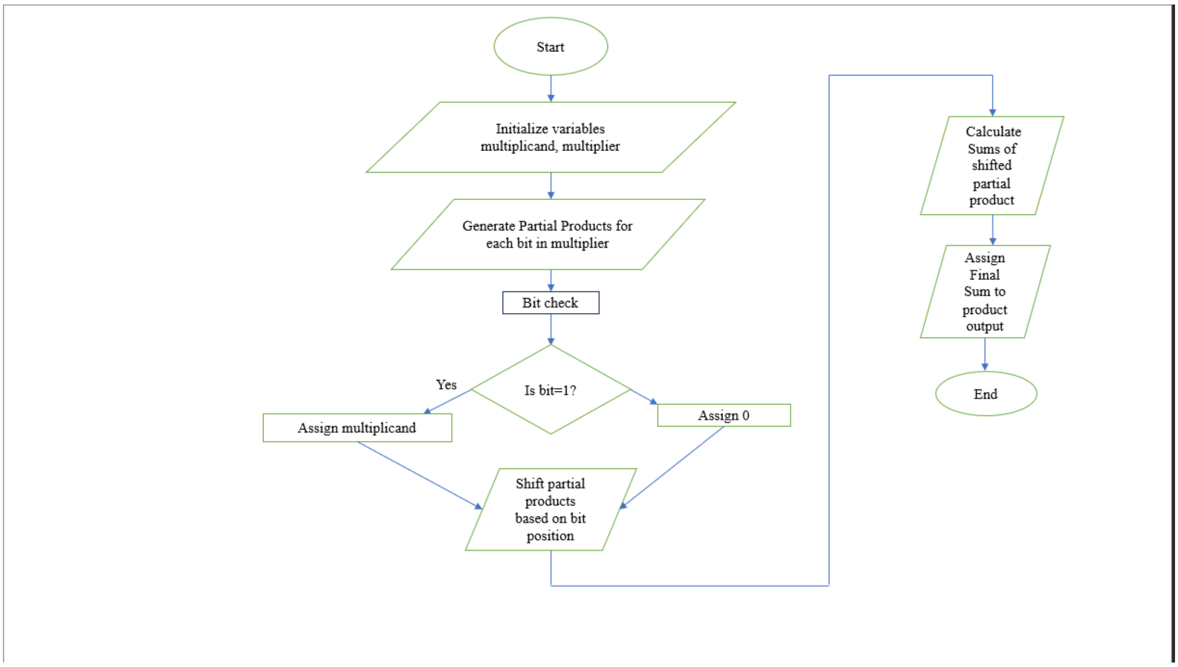


Fig 1. flowchart representing the working of shift add multiplier

The given figure explains how shift add multiplication works for n bit numbers[2-3]From initialization of the multiplier and multiplicand to the various intermediate steps involved while performing this multiplier.

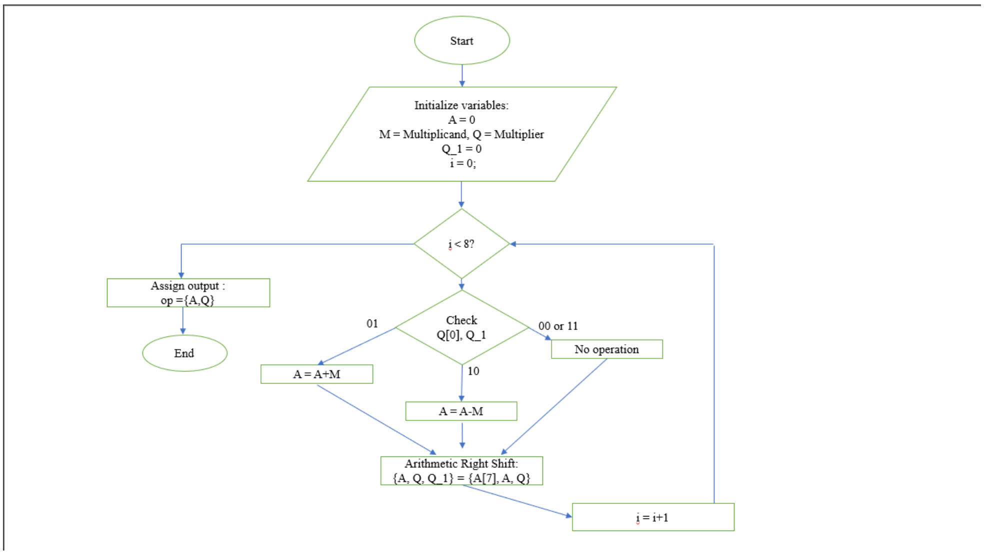
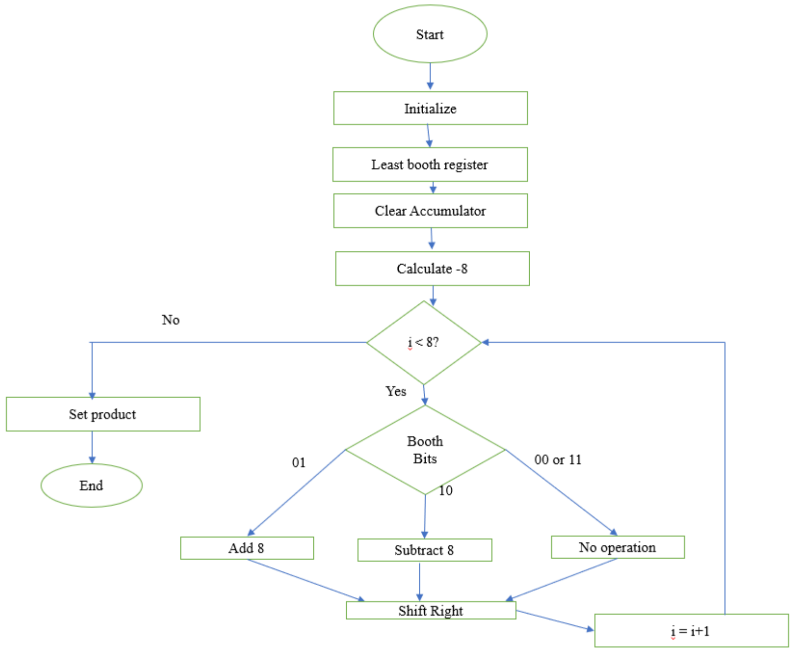


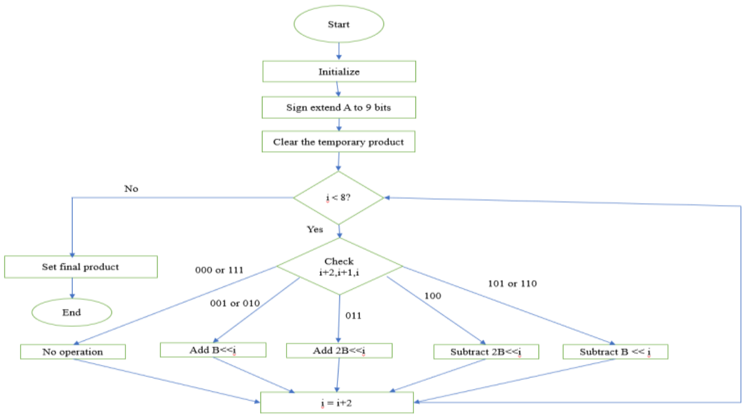
Fig 2. flowchart representing the working of booth multiplier

The given flowchart shows a step-by-step explanation of how the booth multiplier would work for an 8-bit number, as in the first decision making step, we can see the statement "i<8?"[1-3]. The rest of the flowchart shows the various intermediate steps involved while performing booth multiplier.



<u>Fig 3. flowchart representing the working of booth alternate multiplier</u>

The given flowchart shows a step-by-step explanation of how the booth alternate multiplier would work for an 8-bit number. It can noticed that in booth alternate multiplier there are more data entries to be provided than decision making steps which would in theory result in longer compilation and execution time.



<u>Fig 4. flowchart representing the working of shift add multiplier</u>

The given flowchart shows a step-by-step explanation of how the bit pair recoding multiplier would work for an 8-bit number, as in the first instruction box it can be seen the command stating "Sign extend A to 9 bits". This line means that on top of the 8-bit number being inputted it must extend the number to 9 bits for ease calculation and grouping of numbers into 3 bits.[1-3]

APPROACH/METHODOLOGY/PROGRAMS:

1) **Shift Add Multiplier Code:**

```
module shiftadd(
          input [7:0] multiplicand,   // 8-bit multiplicand
   input [7:0] multiplier,     // 8-bit multiplier
   output [15:0] product       // 16-bit product
          );


   wire [15:0] partial_products [7:0]; // Array to store partial products
   wire [15:0] shifted_products [7:0]; // Array to store shifted products
   wire [15:0] sum [6:0];             // Array to store intermediate sums

   // Generate partial products
   genvar i;
   generate
      for (i = 0; i < 8; i = i + 1) begin : gen_partial_products
         assign partial_products[i] = multiplier[i] ? {8'b0, multiplicand} : 16'b0;
      end
   endgenerate

   // Generate shifted partial products
   generate
      for (i = 0; i < 8; i = i + 1) begin : gen_shifted_products
         assign shifted_products[i] = partial_products[i] << i;
      end
   endgenerate

   // Sum the shifted partial products
   assign sum[0] = shifted_products[0] + shifted_products[1];
   assign sum[1] = sum[0] + shifted_products[2];
   assign sum[2] = sum[1] + shifted_products[3];
   assign sum[3] = sum[2] + shifted_products[4];
   assign sum[4] = sum[3] + shifted_products[5];
   assign sum[5] = sum[4] + shifted_products[6];
   assign sum[6] = sum[5] + shifted_products[7];

   // Assign the final product
   assign product = sum[6];

endmodule
```

2) Booth Algorithm Verilog Code:

```
module booth_multiplier(op, multiplicand, multiplier);
   output [15:0] op;
   input [7:0] multiplicand, multiplier;
```

```
    reg [7:0] A, M, Q;
    reg Q_1;
    integer i;

    always @(*) begin
        // Initialize variables
        A = 8'b0;
        M = multiplicand;
        Q = multiplier;
        Q_1 = 1'b0;

        // Perform Booth's algorithm for all 8 bits
        for (i = 0; i < 8; i = i + 1) begin
            case ({Q[0], Q_1})
                2'b01: A = A + M;  // Add multiplicand
                2'b10: A = A - M;  // Subtract multiplicand
            endcase
            // Arithmetic right shift
            {A, Q, Q_1} = {A[7], A, Q};
        end
    end

    assign op = {A, Q};
endmodule
```

3) <u>Booth Alternate Multiplier Verilog Code:</u>

```
module booth_alternate_multiplier (
    input [7:0] A, // Multiplier
    input [7:0] B, // Multiplicand
    output reg [15:0] Product // 16-bit Product
);

    reg [15:0] Acc; // Accumulator
    reg [7:0] NegB; // Negative of multiplicand (-B)
    reg [8:0] Booth; // Booth register (9 bits for A and extra bit for Booth encoding)
    integer i;

    // Booth's algorithm for alternate multiplier
    always @(*) begin
        // Initialize Booth register with multiplier A and extra bit (0 at the LSB)
        Booth = {A, 1'b0};
        // Initialize accumulator (product) to 0
        Acc = 16'd0;
        // Calculate negative of B (for subtraction when needed)
        NegB = -B;

        // Perform Booth's algorithm
        for (i = 0; i < 8; i = i + 1) begin
```

```
        // Check the last two bits of Booth register (Booth[i] and Booth[i-1])
        case (Booth[1:0])
          2'b01: begin
            // Add multiplicand B shifted to the correct position (i-th position)
            Acc = Acc + (B << i);
          end
          2'b10: begin
            // Subtract multiplicand B shifted to the correct position (i-th position)
            Acc = Acc + (NegB << i);
          end
          default: begin
            // No operation needed for 00 or 11
          end
        endcase

        // Perform arithmetic right shift on Booth register
        Booth = {Booth[8], Booth[8:1]}; // Shift with sign extension
      end

      // Assign final product to output
      Product = Acc;
    end

endmodule
```

4) Bit Pair Recoding Multiplier Verilog Code

```
module bit_pair_recoding_multiplier(
    input [7:0] A, // Multiplier
    input [7:0] B, // Multiplicand
    output reg [15:0] Product // 16-bit Product
);

    reg signed [8:0] tempA; // Signed extended A (9-bit)
    reg signed [15:0] tempProduct; // Temp product to accumulate partial sums
    integer i;

    always @(*) begin
      tempA = {A[7], A}; // Sign extend the multiplier A to 9 bits
      tempProduct = 16'd0; // Initialize product to zero

      // Perform bit pair recoding and Booth's algorithm
      for (i = 0; i < 8; i = i + 2) begin
        case ({tempA[i+2], tempA[i+1], tempA[i]}) // Select bit pairs
          3'b000, 3'b111: begin
            // No addition required for 0 (000) and -0 (111)
          end
          3'b001, 3'b010: begin
            tempProduct = tempProduct + (B << i); // Add B shifted left by i positions
```

```
            end
            3'b011: begin
              tempProduct = tempProduct + ((B << i) << 1); // Add 2B shifted left by i
positions
            end
            3'b100: begin
              tempProduct = tempProduct - ((B << i) << 1); // Subtract 2B shifted left by i
positions
            end
            3'b101, 3'b110: begin
              tempProduct = tempProduct - (B << i); // Subtract B shifted left by i positions
            end
          endcase
        end

        Product = tempProduct; // Assign final product
      end

endmodule
```

OUTPUT:

The outputs of the programs are given below:

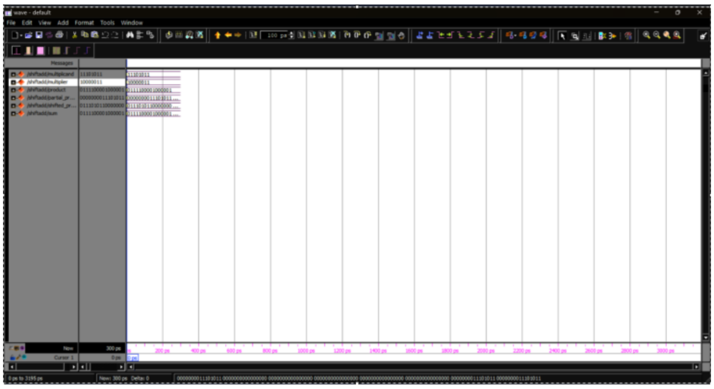


Fig 5. shows the output of shift add multiplier

Consider the multiplicand(A) = 11101011(235 in decimal) and multiplier = 10000011(131 in decimal) and it can be seen the output achieved as 0111100001000001(30,785 in decimal which is the required output in 16 bits(8+8)[1].The other three outputs are the partial products, shifted products and intermediate sum. These three outputs are the intermediate steps that take place before it can get the product of the two numbers.

Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slices:	34	960	3%	
Number of 4 input LUTs:	64	1920	3%	
Number of bonded IOBs:	32	66	48%	

Fig6 shows the device utilization summary of the shift add multiplier

The number of slices utilized are 3%. A slice is a collection of logic elements showing the amount of available logic resources used. This means that only 3%(34 out of a total of 960) of the total available logic slides on the FPGA device are being used. LUTs utilization is also 3% and this means that only 3% of the look up tables are used for combinational logic and this is good as the lower the LUTs, the less complex the circuit is. IOBs are Input/Output Blocks used to interface with external devices[1-4].

Total Compute time:34.460ns (25.001ns logic, 9.459ns route)
CPU: 1.49 / 1.65 s | Elapsed: 1.00 / 1.00 s

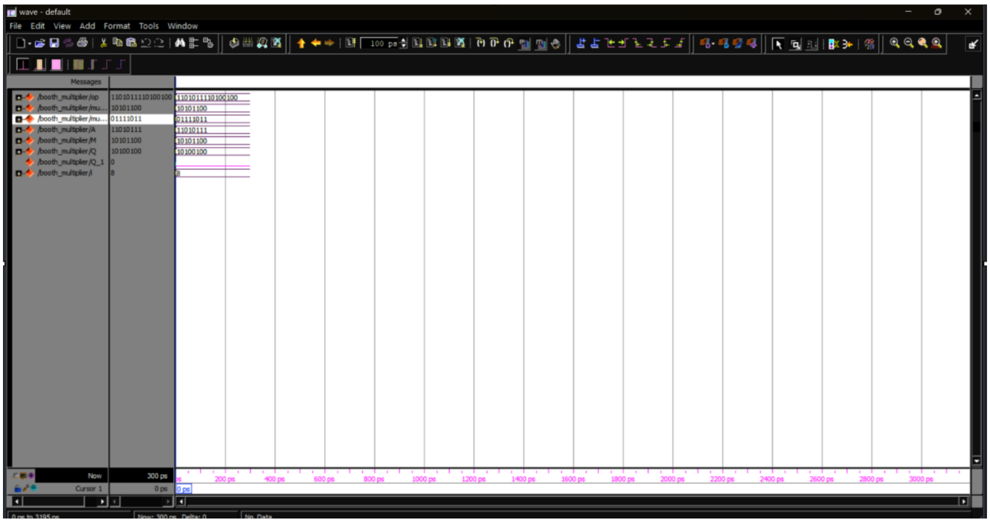


Fig7 shows the output of booth multiplier

Consider the multiplicand(A) = 10101100(-84 in decimal) and multiplier = 01111011(123 in decimal) and it can be seen that the output achieved as 1101011110100100 (-10332 in decimal which is the required output in 16 bits(8+8)[1].The other three outputs are the partial products, shifted products and intermediate sum. These three outputs are the intermediate steps that take place before it can get the product of the two numbers.

Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slices:	71	960	7%	
Number of 4 input LUTs:	131	1920	6%	
Number of bonded IOBs:	32	66	48%	

Fig8 shows the device utilization summary of the booth multiplier

The number of slices utilized are 7%. A slice is a collection of logic elements showing the amount of available logic resources used. This means that only 7%(71 out of a total of 960) of the total available logic slides on the FPGA device are being used. LUTs utilization is 6% and this means that only 6% of the look up tables are used for combinational logic and this is good as the lower the LUTs, the less complex the circuit is. IOBs are Input/Output Blocks used to interface with external devices[1-4].

Total Compute Time: 49.329ns (31.233ns logic, 18.096ns route)
CPU: 1.90 / 2.13 s | Elapsed: 1.00 / 2.00 s

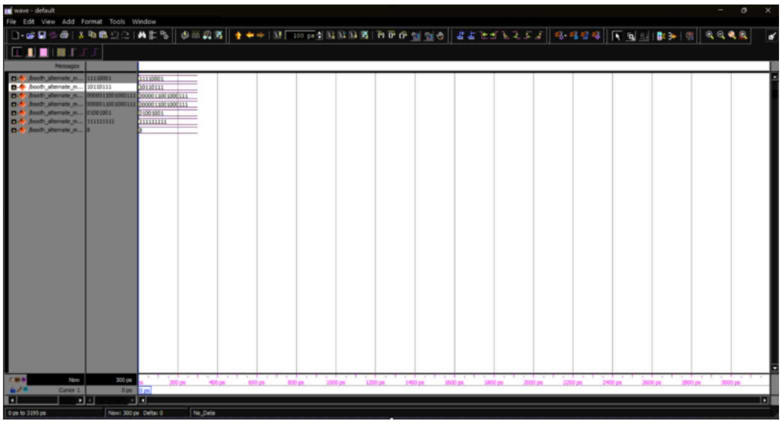


Fig 9. shows the device utilization summary of the booth alternate multiplier

Consider the multiplicand(A) = 11110001(-15 in decimal) and multiplier = 10110111 (-73 in decimal) and it can be seen that the output achieved as 0000010001000111 (1095 in decimal which is the required output in 16 bits(8+8)[1].The other three outputs are the partial products, shifted products and intermediate sum. These three outputs are the intermediate steps that take place before it can get the product of the two numbers.

Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slices:	138	960	14%	
Number of 4 input LUTs:	229	1920	11%	
Number of bonded IOBs:	32	66	48%	

Fig 10. shows the device utilization summary of the booth alternate multiplier

The number of slices utilized are 3%. A slice is a collection of logic elements showing the amount of available logic resources used. This means that only 14%(138 out of a total of 960) of the total available logic slides on the FPGA device are being used. LUTs utilization is 11% and this means that only 11% of the look up tables are used for combinational logic and this is good as the lower the LUTs, the less complex the circuit is. IOBs are Input/Output Blocks used to interface with external devices[1-4].

Total Compute time: 51.674ns (31.714ns logic, 19.960ns route)
CPU: 2.15 / 2.35 s | Elapsed: 3.00 / 3.00 s

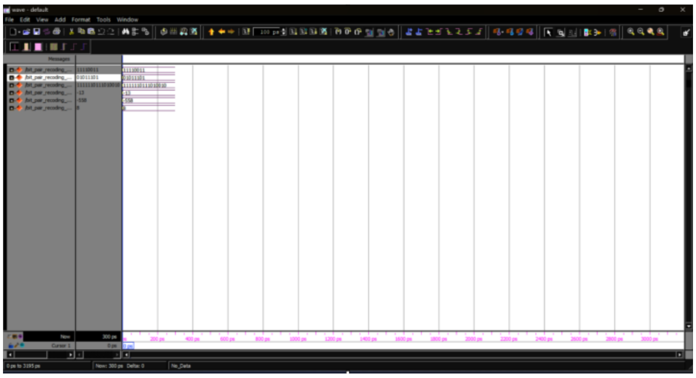


Fig 11. shows the output of bit pair recoding multiplier

Consider the multiplicand(A) = 11110011(-13 in decimal) and multiplier = 01011101 (93 in decimal) and can see the output achieved as 1111101101000111 (1209 in decimal which is the required output in 16 bits(8+8)[1].The other three outputs are the partial products, shifted products and intermediate sum. These three outputs are the intermediate steps that take place before can get the product of the two numbers.

Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slices:	104	960	10%	
Number of 4 input LUTs:	164	1920	8%	
Number of bonded IOBs:	32	66	48%	

Fig 12. shows the device utilization summary of the shift add multiplier

The number of slices utilized are 10%. A slice is a collection of logic elements showing the amount of available logic resources used. This means that only 10%(104 out of a total of 960) of the total available logic slides on the FPGA device are being used. LUTs utilization is 8% and this means that only 8% of the look up tables are used for combinational logic and this is good as the lower the LUTs, the less complex the circuit is. IOBs are Input/Output Blocks used to interface with external devices[1-4].

Total Compute Time: 35.371ns (22.906ns logic, 12.465ns route)
CPU: 2.69 / 2.87 s | Elapsed: 3.00 / 3.00 s

RESULT:

Table 1: Comparative Performance Analysis of the different multiplier algorithms

Algorithm	Total Compute Time	CPU time	Elapsed Time
Shift Add Multiplier	34.460 ns (25.001ns logic, 9.459 ns route)	1.49 / 1.65s	1.00 / 1.00s
Booth Multiplier	49.329 ns (31.233 ns logic, 18.096 ns route)	1.90 / 2.13s	1.00 / 2.00s
Booth Alternate	51.674 ns (31.714 ns logic, 19.960 ns route)	2.15 / 2.35s	3.00 / 3.00s
Bit-Pair Recoding	35.371 ns (22.906 ns logic, 12.465 ns route)	2.69 / 2.87s	3.00 / 3.00s

From the table it can be seen that shift add multiplier is the fastest algorithm while booth alternate multiplier is the slowest algorithm. The reasoning for this can be the large route time taken for booth alternate multiplier to compute (almost a 10ns jump compared to shift add multiplier) [1]. Another fast multiplier was bit-pair recoding multiplier, its logic time was faster than shift add multipliers but the increased code length, i.e. the route caused it to be slower than shift add multiplier [1-2].

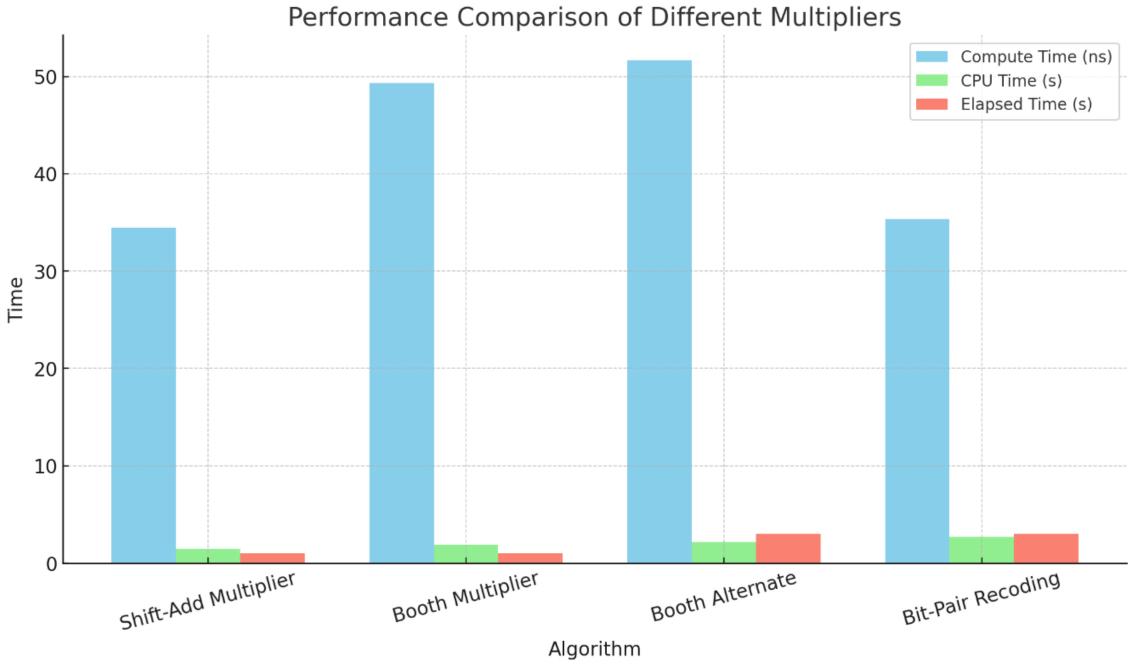


Fig 13. Graphical representation of Table 1

From the graph there is a clearer understanding of how long each multiplier took to compute. It also shows the comparison of the CPU rendering time and elapsed time taken. From the graph it is clear that both the booth multipliers are slow, but are still in use due to the ability to multiply signed (negative) numbers.

CONCLUSIONS:

This project provides a comprehensive performance analysis of four multipliers—Booth, Booth Alternate, Bit Pair Recoding, and Shift-Add—using 8-bit inputs. Each technique demonstrates unique advantages in terms of speed, resource utilization. Through detailed examination of factors such as total compute time and device utilization, the Shift-Add and Bit Pair Recoding multipliers emerge as faster alternatives, whereas Booth-based algorithms show strength in handling signed operations efficiently.

In practical applications, the choice of a multiplier depends heavily on the trade-offs between computation speed, hardware cost, and ease of implementation. For systems requiring minimal logic and faster execution times, the Shift-Add and Bit Pair Recoding methods are preferable. However, for applications involving more complex arithmetic, such as signal processing, Booth's algorithms provide reliable solutions.

In conclusion, this analysis offers valuable insights into selecting an appropriate multiplier design based on specific performance requirements, aiding in the development of optimized digital circuits.

REFERENCES:

[1] A. Ghalyan and V. Kadyan, “Performance Analysis and Verification of Multipliers,” International Journal of Computer (IJC), vol. 13, pp. 93–102, 2014.

[2] R. Hussin, A.Yeon Md .Shakaff, N.Idris, Z.Sauli, Rizalafande C.Ismail, Afzan Kamarudin ,”An Efficient Modified Booth Multiplier Architecture”,IEEE International Conference on Electronics Design ,Dec. 2008.

[3] P.kumar G.Parate, Prafulla S. Patil, Dr (Mrs) S. Subbaraman ,“ASIC Implementation of 4 Bit Multipliers”,IEEE First International Conference on Emerging Trends in Engineering and Technology, pp. 408-413,2008.

[4] S R. Vaidya, D. R. Dandekar, “Performance Comparison of Multipliers for Power-Speed Trade-off in VLSI Design”, Recent Advances in Networking VLSI and Signal Processing, pp.262-265, June 2011.