PES UNIVERSITY, Bangalore
(Established under Karnataka Act No.16 of 2013)

A REPORT

on

KARATASUBA MULTIPLIER

Submitted by

LIKHITHA D N
SRN: PES1UG22EC139

BACHELOR OF TECHNOLOGY

in

ELECTRONICS AND COMMUNICATION ENGINEERING

Academic Year: 2023-24

ABSTRACT

The Karatsuba algorithm is a method for fast multiplication, particularly useful for large numbers. Unlike traditional multiplication methods, which require a lot of time as the numbers grow larger, the Karatsuba algorithm breaks the multiplication into smaller, more manageable pieces. This approach reduces the overall number of calculations, making it faster and more efficient. This report focuses on implementing and optimizing the Karatsuba multiplier using hardware description languages, specifically System Verilog, to create a more efficient multiplication process on FPGA platforms.

In this research, we explore various techniques to improve the performance of the Karatsuba multiplier, such as pipelining and recursion. These methods allow the multiplication process to be completed more quickly and efficiently, even with very large numbers. The implementation was tested on an FPGA, and the results showed significant improvements in speed and resource usage compared to traditional methods. This makes the optimized Karatsuba multiplier a powerful tool for applications that require fast and efficient large-number multiplications, such as in digital signal processing and cryptography.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 IMPORTANCE OF EFFICIENT MULTIPLICATION

Multiplication is a key operation in computing, especially in digital signal processing, cryptography, and numerical analysis. As the size of the numbers increases, the time and resources needed for multiplication also rise. Traditional methods like long multiplication become inefficient for large numbers, leading to slower processing and higher power consumption. This is particularly problematic in modern systems that require high performance and real-time processing.

1.2 OVERVIEW OF THE KARATSUBA ALGORITHM

The Karatsuba algorithm, introduced by Anatolii Karatsuba in 1962, addresses the inefficiencies of traditional multiplication methods. It uses a divide-and-conquer strategy to reduce the number of multiplications needed to find the product of two large numbers. Instead of multiplying the entire numbers directly, the algorithm breaks them into smaller parts, performs three multiplications on these parts, and then combines the results with some additions and subtractions. This method reduces the overall time complexity from $O(n^2)$ in traditional methods to approximately $O(n^{1.585})$, making it much faster for large numbers.

1.3 OBJECTIVES OF THE RESEARCH

This research aims to implement and improve the Karatsuba multiplication algorithm in hardware using SystemVerilog, focusing on high performance on FPGA platforms. The main goals are to use pipelining and recursion to make the multiplier faster and more efficient. By designing and testing these improvements, the research will show how the Karatsuba algorithm can be a practical solution for fast, large-number multiplication in real-world applications. The results will help understand how effective the Karatsuba algorithm is when used in hardware for different computing tasks.

CHAPTER 2

THEORETICAL BACKGROUND

2.1 THE KARATSUBA ALGORITHM

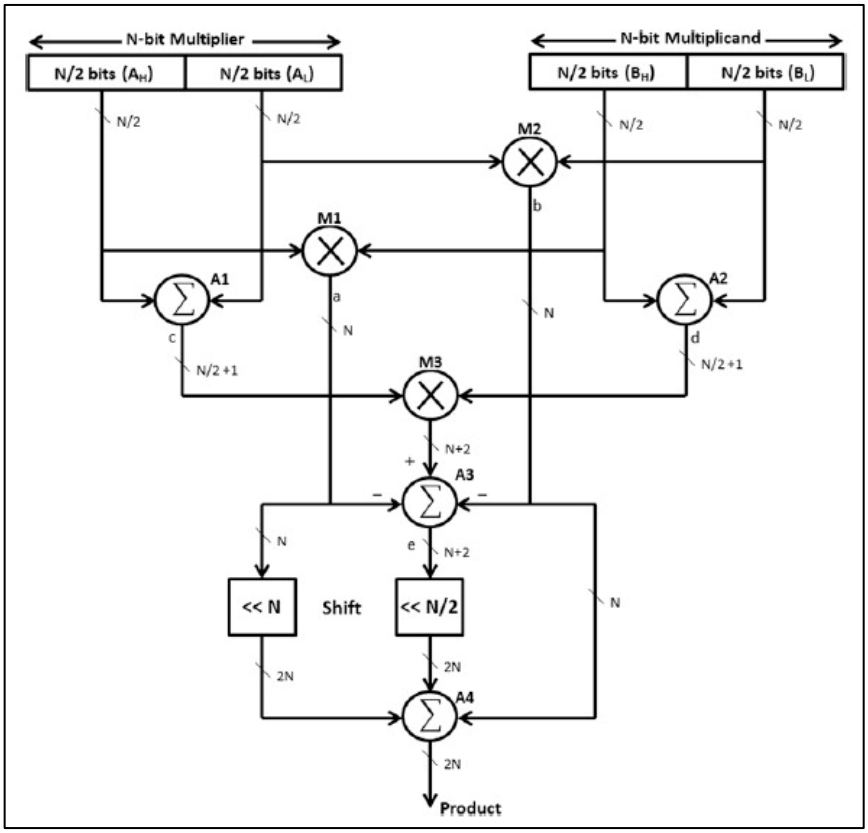
The Karatsuba algorithm is a multiplication technique designed to improve the efficiency of multiplying large numbers. Traditional multiplication methods, such as the grade-school or long multiplication method, involve multiplying each digit of the first number by every digit of the second number, which results in a time complexity of $O(n^2)$. However, as the size of the numbers increases, this approach becomes increasingly inefficient.

The Karatsuba algorithm, introduced by Anatolii Karatsuba in 1962, utilizes a divide-and-conquer strategy. The algorithm breaks each number into two smaller parts and computes three multiplications instead of four, as would be required in the traditional method. The smaller products are then combined using additions and subtractions to form the final result. This reduction in the number of multiplications lowers the overall time complexity, making the algorithm much faster for large inputs.

2.2 COMPLEXITY ANALYSIS

The efficiency of the Karatsuba algorithm is best understood through its complexity analysis. Traditional multiplication has a time complexity of $O(n^2)$, where nnn is the number of digits in the numbers being multiplied. The Karatsuba algorithm reduces this complexity by reducing the number of multiplications needed.

The algorithm works by splitting each number into two halves, performing three multiplications, and then combining the results. The time complexity of the Karatsuba algorithm is approximately $O(n^{1.585})$. This reduction in complexity becomes increasingly significant as the size of the input numbers grows, making Karatsuba particularly advantageous for very large numbers.



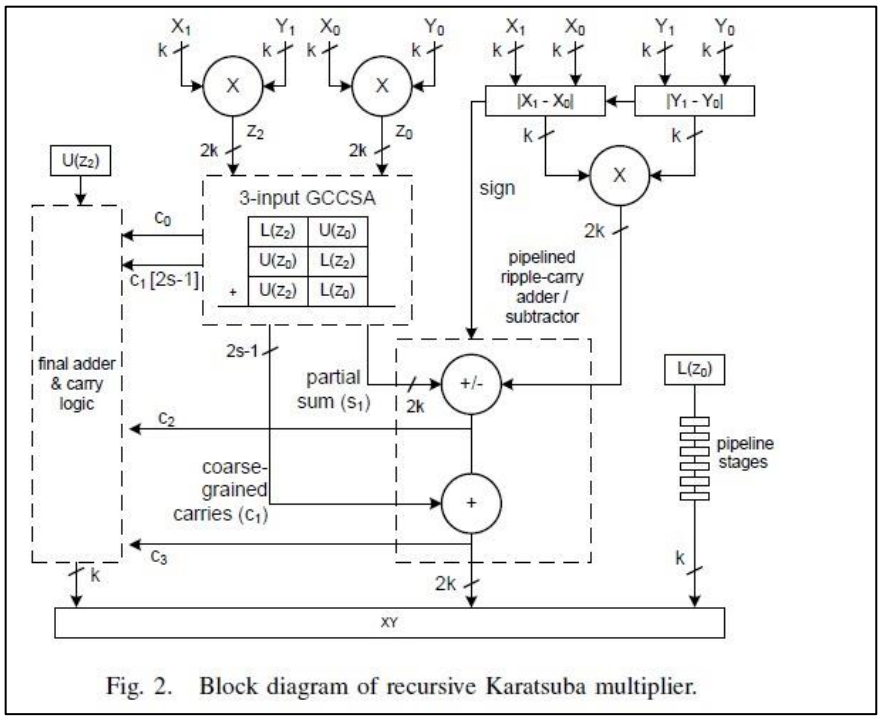


Fig. 2. Block diagram of recursive Karatsuba multiplier.

CHAPTER 3

SYSTEMVERILOG IMPLEMENTATION

3.1 BASIC DESIGN

The basic design of the Karatsuba multiplier in SystemVerilog starts with a straightforward implementation of the algorithm. The input numbers are split into high and low parts, and the necessary multiplications are performed on these parts. The results are then combined to produce the final product.

```
//Verilog
module karatsuba(
input logic [15:0] p, //3 digits
input logic [15:0] q, //3 digits
output logic [31:0] result //6 digits
);
logic [15:0] a,b;
logic [31:0] r0,r1;
logic [24:0] middle_term;

assign a=p[15:8]*q[15:8];
assign b=p[7:0]*q[7:0];
assign r1={a,16'b0000000000000000};
assign r0=b;
assign middle_term={(p[15:8]+p[7:0])*(q[15:8]+q[7:0])-a-b,8'b00000000};
assign result=r1+middle_term+r0;
endmodule
```

3.2 PIPELINED DESIGN USING BASE MULTIPLIER

In this approach, the Karatsuba algorithm is implemented by using multiple base multipliers rather than relying on recursion. A base multiplier is a simple, fixed-size multiplier that performs the multiplication of smaller sections of the input numbers. The Karatsuba algorithm takes advantage of these base multipliers to handle larger inputs by splitting the numbers into smaller parts and using the base multipliers to compute partial products.

The advantage of using base multipliers is that it simplifies the hardware design and makes it easier to optimize for specific sizes, which can lead to better performance on FPGA platforms.

```
module karatsuba1 #(parameter WIDTH = 32) (
    input  logic [WIDTH-1:0] x,
    input  logic [WIDTH-1:0] y,
    output logic [(2*WIDTH)-1:0] product,
    input  logic clk,   // Clock input for pipelining
    input  logic rst_n  // Asynchronous reset
);
    localparam HALF_WIDTH = WIDTH / 2;
    // Split inputs into high and low parts
    logic [HALF_WIDTH-1:0] x_high, x_low, y_high, y_low;
    logic [WIDTH-1:0] z0, z1, z2;
    logic [(2*WIDTH)-1:0] z0_ext, z1_ext, z2_ext;
    logic [WIDTH:0] tmp1, tmp2;
    // Pipeline registers for z0
    logic [WIDTH-1:0] z0_reg;
    // Reset
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            z0_reg <= '0;
        end else begin
            z0_reg <= z0;
        end
    end
    assign x_low  = x[HALF_WIDTH-1:0];
    assign x_high = x[WIDTH-1:HALF_WIDTH];
    assign y_low  = y[HALF_WIDTH-1:0];
    assign y_high = y[WIDTH-1:HALF_WIDTH];
```

```
    // Compute partial products
    base_multiplier #(HALF_WIDTH) u_base_mult1 (.a(x_low), .b(y_low), .product(z0));
    base_multiplier #(HALF_WIDTH) u_base_mult2 (.a(x_high), .b(y_high), .product(z2));
    base_multiplier #(WIDTH) u_base_mult3 (.a({x_high, x_low}), .b({y_high, y_low}), .product(z1));
    // Extend partial products to match final width
    assign z0_ext = {{(WIDTH){1'b0}}, z0_reg};
    assign z2_ext = {z2, {(WIDTH){1'b0}}};
    assign tmp1 = z0_reg + z2;
    assign tmp2 = z1 - tmp1;
    assign z1_ext = {tmp2, {(HALF_WIDTH){1'b0}}};
    // Final addition to get the product
    assign product = z2_ext + z1_ext + z0_ext;
endmodule
```

3.3 PIPELINED DESIGN USING RECURSION

To improve performance, the Karatsuba multiplier can be implemented using pipelining. Pipelining allows different stages of the multiplication process to operate simultaneously, increasing the throughput and making the design more efficient for large inputs.

In a pipelined design, the multiplications and additions are divided into separate stages, with registers placed between them to store intermediate results. This enables the hardware to process multiple sets of inputs concurrently.

```
//Verilog
module karatsubasync #(parameter WIDTH = 32) (
    input  logic [WIDTH-1:0] x,
    input  logic [WIDTH-1:0] y,
    input  logic clk,
    input  logic rst_n,
    output logic [(2*WIDTH)-1:0] product
);
    localparam HALF_WIDTH = WIDTH / 2;
    logic [HALF_WIDTH-1:0] x_high, x_low, y_high, y_low;
    logic [WIDTH-1:0] z0, z1, z2;
    logic [(2*WIDTH)-1:0] z0_ext, z1_ext, z2_ext;
    logic [WIDTH-1:0] sum_cg_csa, z1_sub;
    logic [WIDTH-1:0] z0_reg,q1;
    always_ff @(posedge clk or posedge rst_n) begin
        if (!rst_n) begin
            z0_reg <= '0;
        end else begin
            z0_reg <= z0;
        end
    end
    assign q1=z0_reg;
    assign x_low  = x[HALF_WIDTH-1:0];
    assign x_high = x[WIDTH-1:HALF_WIDTH];
    assign y_low  = y[HALF_WIDTH-1:0];
    assign y_high = y[WIDTH-1:HALF_WIDTH];
    base_multiplier #(HALF_WIDTH) u_base_mult1 (.a(x_low), .b(y_low), .product(z0));
    base_multiplier #(HALF_WIDTH) u_base_mult2 (.a(x_high), .b(y_high), .product(z2));
    base_multiplier #(WIDTH) u_base_mult3 (.a(x_high-x_low), .b(y_high-y_low), .product(z1));
    assign z0_ext = {{(WIDTH){1'b0}}, z0_reg};
    assign z2_ext = {z2, {(WIDTH){1'b0}}};
    cg_csa2 #(WIDTH) u_cg_csa (
        .a(q1),
        .b(z2),
        .sum(sum_cg_csa)
    );
```

```
    pipelined_ripple_carry_adder_subtractor #(WIDTH) u_subtractor (
        .a(z1),
        .b(sum_cg_csa),
        .subtract(1'b1),
        .result(z1_sub)
    );
    assign z1_ext = {z1_sub, {(HALF_WIDTH){1'b0}}};
    final_adder #(2*WIDTH) u_final_adder (
        .a(z2_ext),
        .b(z1_ext),
        .c(z0_ext),
        .sum(product),
        .carry()
    );
endmodule

//Base multiplier
module base_multiplier #(parameter WIDTH = 16) (
    input logic [WIDTH-1:0] a,
    input logic [WIDTH-1:0] b,
    output logic [(2*WIDTH)-1:0] product
);
    assign product = a * b;
endmodule

//coarse grained carry-save adder
module cg_csa2 #(parameter WIDTH = 32) (
    input logic [WIDTH-1:0] a,
    input logic [WIDTH-1:0] b,
    output logic [WIDTH-1:0] sum
);
    assign sum = a + b ;
endmodule

//ripple carry adder subtractor
module pipelined_ripple_carry_adder_subtractor #(parameter WIDTH = 32) (
    input logic [WIDTH-1:0] a,
    input logic [WIDTH-1:0] b,
    input logic subtract, // 0 for add, 1 for subtract
    output logic [WIDTH-1:0] result
);
    assign result = subtract ? (a - b) : (a + b);
endmodule

//final adder
module final_adder #(parameter WIDTH = 64) (
    input logic [WIDTH-1:0] a,
    input logic [WIDTH-1:0] b,
    input logic [WIDTH-1:0] c,
    output logic [WIDTH-1:0] sum,
    output logic [WIDTH-1:0] carry
);
    assign {carry, sum} = a + b + c;
endmodule
```

CHAPTER 4

EXPERIMENTAL SETUP

4.1 TESTBENCH

To validate the functionality and performance of the Karatsuba multiplier design using base multipliers, a comprehensive testbench is created in SystemVerilog. The testbench simulates various input scenarios, ensuring that the multiplier produces correct results across different cases, including edge cases like multiplying by zero or one, and large inputs that stress the design.

Test Bench for Basic Design:

```
module karatsuba_tb();
logic [15:0]p;
logic [15:0]q;
logic [31:0]result;

karatsuba uut(.p(p),.q(q),.result(result));
initial begin
p = 16'b0000010011010010; //1234
q = 16'b0001011001101110; // 5678
//(1234x5678=708562)
#10 p = 16'b0000000001111011;//123
q = 16'b0000010110110000;//1456
//(123x1456=179800)
#10;  $finish;
end
endmodule
```

Test Bench for Pipelined Design Using Base Multiplier:

```
module tb_karatsuba1;
    parameter WIDTH = 32;
    logic [WIDTH-1:0] x, y;
    logic [(2*WIDTH)-1:0] product;
    logic clk;
    logic rst_n;
    karatsuba1 #(WIDTH) dut (
        .x(x),
        .y(y),
        .product(product),
        .clk(clk),
        .rst_n(rst_n)
    );
    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
    // Testbench logic
    initial begin
        rst_n = 0;
        #10 rst_n = 1;
        x = 1234;
        y = 5678; // Expected product: 7006652
        #100;
        x = 246;
        y = 568; // Expected product: 139728
        #100;
        x = 123;
        y = 1456; // Expected product: 179088
        #100;
        x = 689;
        y = 730; // Expected product: 502970
        #100;
        $display("x = %0d, y = %0d, product = %0d", x, y, product);
        $finish;
    end
endmodule
```

Test Bench for Pipelined Design Using Recursion:

```
module karatsubasync_tb;
    parameter WIDTH = 32;
    logic [WIDTH-1:0] x, y;
    logic [(2*WIDTH)-1:0] product;
    logic clk;
    logic rst_n;
    karatsubasync #(WIDTH) dut (
        .x(x),
        .y(y),
        .product(product),
        .clk(clk),
        .rst_n(rst_n)
    );
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
    initial begin
        rst_n = 0;
        #10 rst_n = 1;
        x = 1234;
        y = 5678;// Expected product: 7006652
        #100;
        x = 246;
        y = 568; // Expected product: 139728
        #100;
        x = 123;
        y = 1456; // Expected product: 179088
        #100;
        x = 689;
        y = 730; // Expected product: 502970
        #100;
        $display("x = %0d, y = %0d, product = %0d", x, y, product);
        $finish;
    end
endmodule
```

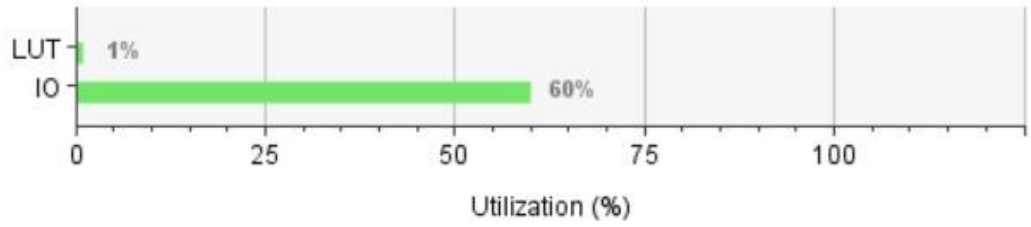
4.2 LUTs (Look-Up Tables)

Look-Up Tables (LUTs) are a fundamental resource in FPGA design, used to implement logic functions. In this experiment, the efficiency of the Karatsuba multiplier is assessed by analyzing the number of LUTs it consumes on the FPGA. The design's LUT usage is an important metric, as it reflects the hardware resources required and influences the overall performance and scalability of the multiplier.

LUT for Basic Design:

Name	Slice LUTs (20800)	Bonded IOB (106)
karatsuba	308	64

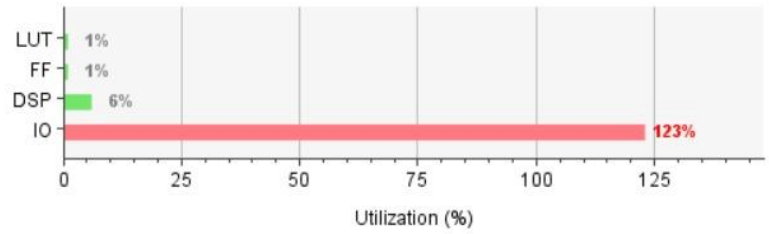
Resource	Utilization	Available	Utilization %
LUT	308	20800	1.48
IO	64	106	60.38



LUT for Pipelined Design Using Base Multiplier:

Name	Slice LUTs (20800)	Slice Registers (41600)	DSPs (90)	Bonded IOB (106)	BUFGCTRL (32)
karatsuba1	91	1	5	130	1
u_base_mult1 (base_...	25	0	1	0	0
u_base_mult2 (base_...	50	0	1	0	0
u_base_mult3 (base_...	15	0	3	0	0

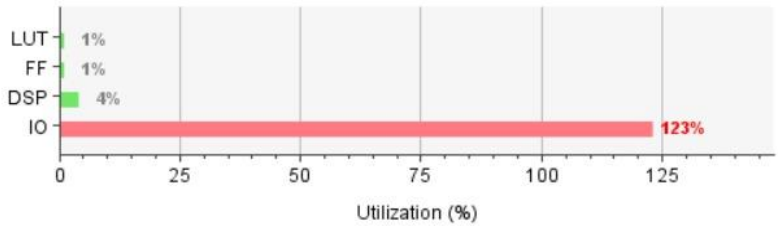
Resource	Utilization	Available	Utilization %
LUT	91	20800	0.44
FF	1	41600	0.00
DSP	5	90	5.56
IO	130	106	122.64



LUT for Pipelined Design Using Recursion:

Name	Slice LUTs (20800)	Slice Registers (41600)	DSPs (90)	Bonded IOB (106)	BUFGCTRL (32)
karatsubasync	74	1	4	130	1
u_base_mult1 (base_...	25	0	1	0	0
u_base_mult2 (base_...	0	0	1	0	0
u_cg_csa (cg_csa2)	0	0	1	0	0
u_final_adder (final_a...	32	0	0	0	0

Resource	Utilization	Available	Utilization %
LUT	74	20800	0.36
FF	1	41600	0.00
DSP	4	90	4.44
IO	130	106	122.64

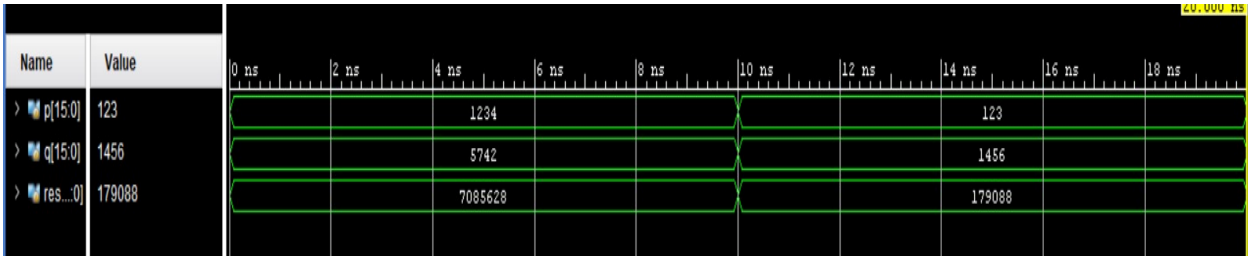


CHAPTER 5

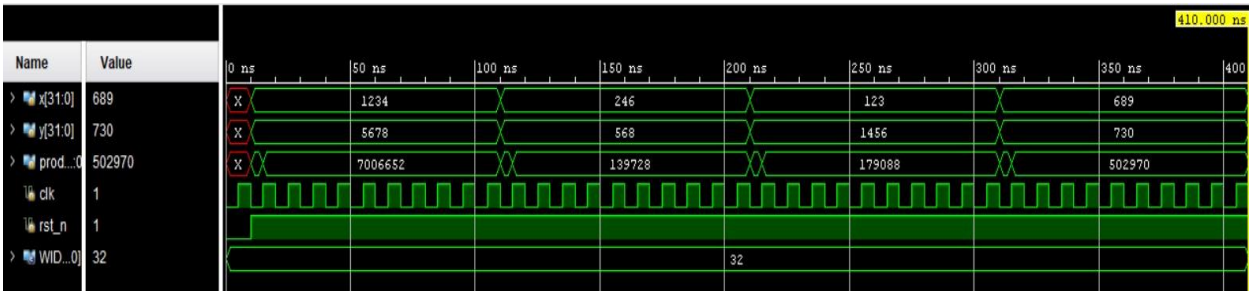
RESULTS

5.1 SIMULATION

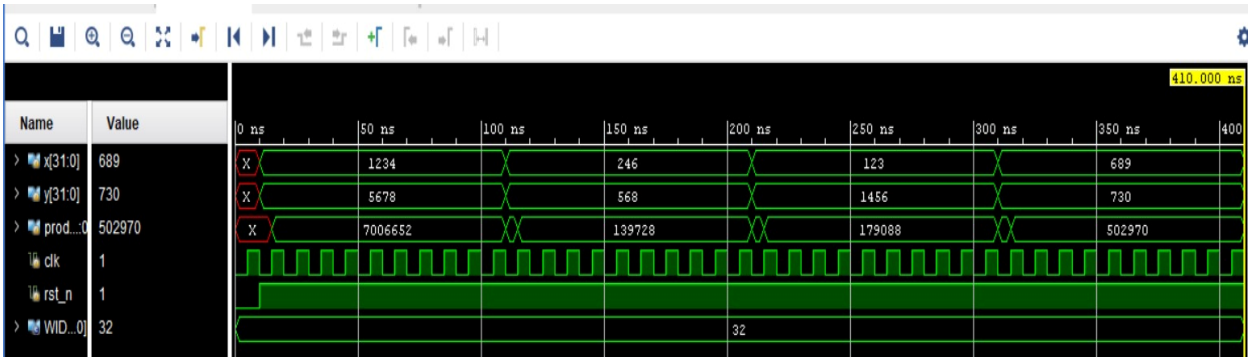
Simulation for Basic Design:



Simulation for Pipelined Design Using Base Multiplier:

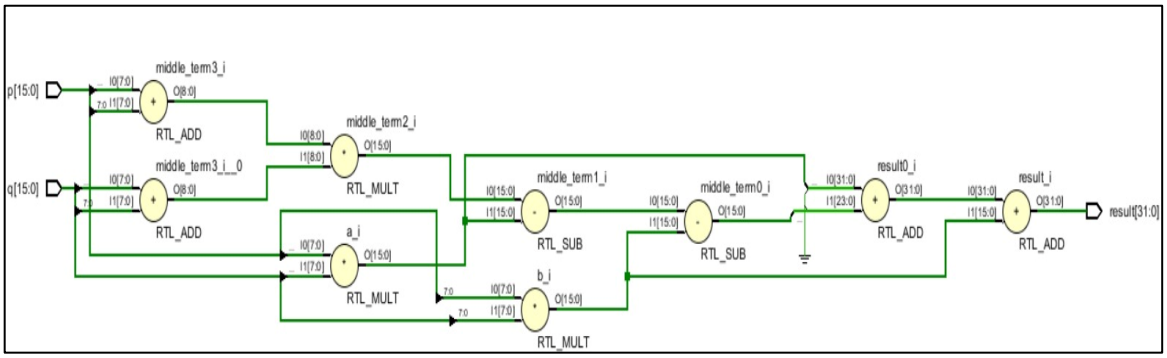


Simulation for Pipelined Design Using Recursion:

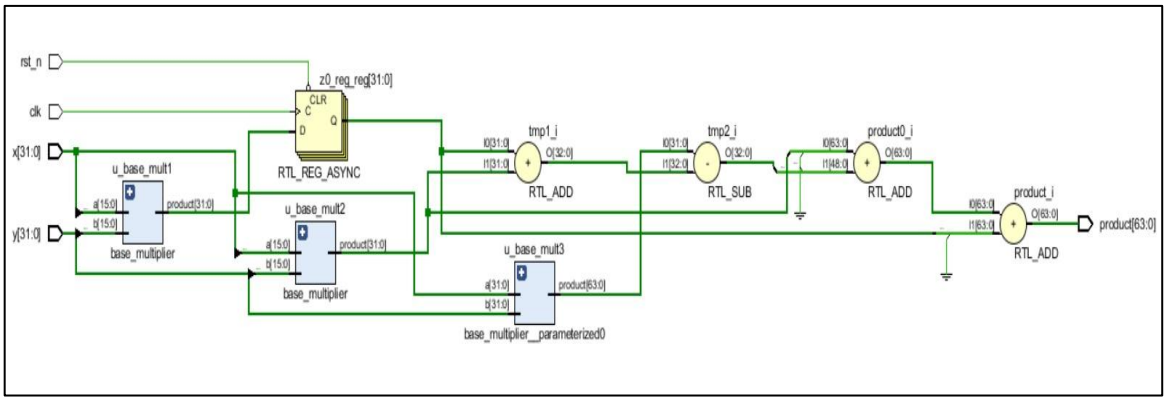


5.2 ELABORATED DESIGN

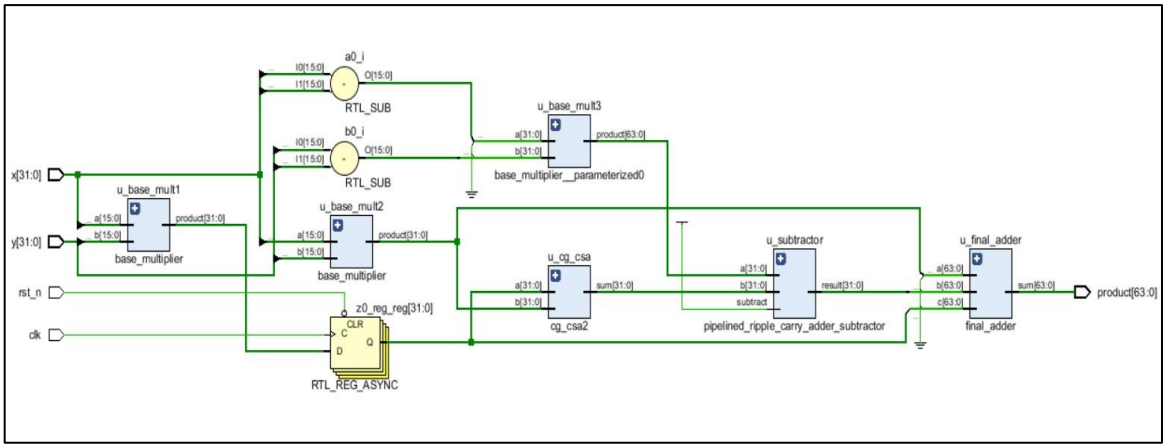
Elaborated Design for Basic Design:



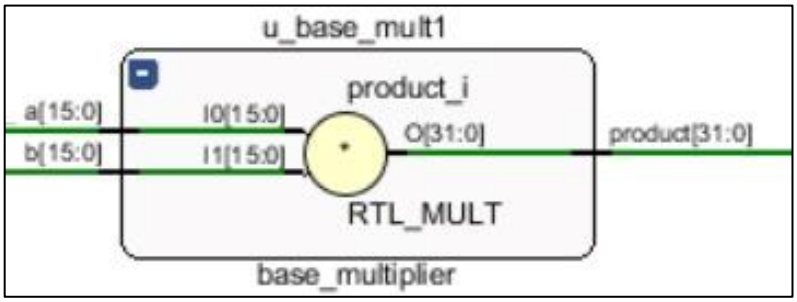
Elaborated Design for Pipelined Design Using Base Multiplier:



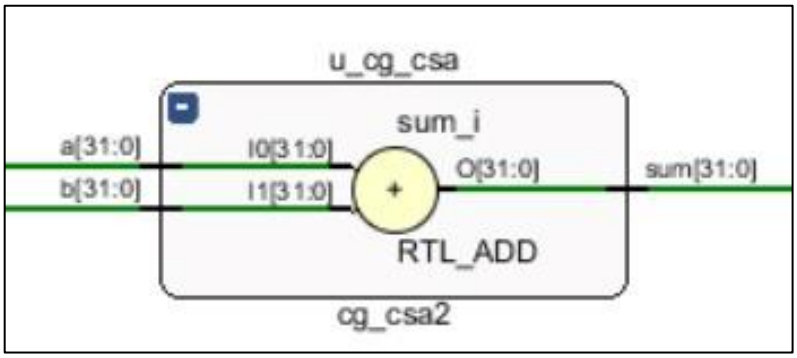
Elaborated Design for Pipelined Design Using Recursion:



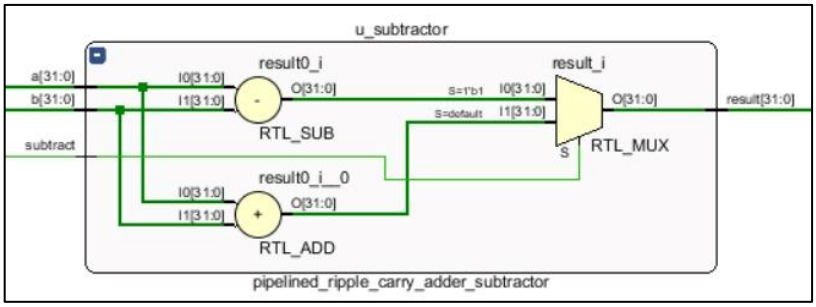
Elaborated Design for Base Multiplier:



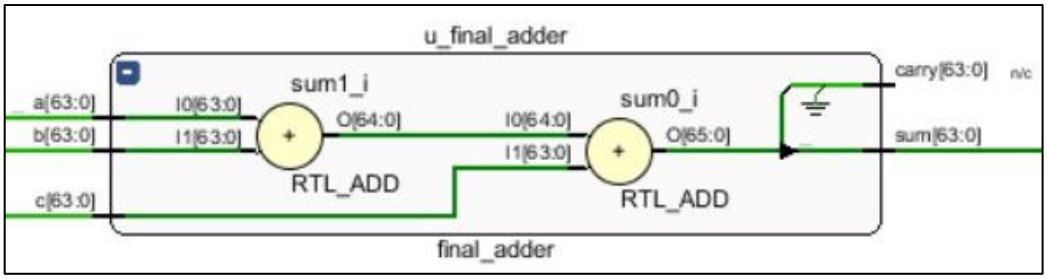
Elaborated Design for Coarse Grained Carry-Save Adder:



Elaborated Pipelined Design for Ripple Carry Adder Subtractor:



Elaborated Design for Final Adder:



CHAPTER 6

CONCLUSION

In conclusion, the Karatsuba multiplier offers a significant improvement in multiplication efficiency, especially for large numbers, by reducing the number of necessary multiplications compared to traditional methods. By implementing this algorithm in hardware using SystemVerilog, and optimizing it with techniques such as pipelining and base multipliers, we have demonstrated its potential for high-performance applications. The hardware-based Karatsuba multiplier can handle complex computations more quickly and with fewer resources, making it a valuable approach for tasks that require large-integer multiplication, such as cryptography and digital signal processing.

The experimental results, including successful simulations and efficient LUT utilization, show that the Karatsuba multiplier design is both effective and scalable. The use of base multipliers instead of recursion simplified the design, making it easier to implement on FPGA platforms while maintaining high performance. This study highlights the practical benefits of the Karatsuba algorithm in hardware, suggesting that it can be a powerful tool for improving the speed and efficiency of various computational tasks in real-world applications.

REFERENCES

[1]. Chu, P. P. *FPGA Prototyping by Verilog Examples: Xilinx Spartan-3 Version*. Wiley-Interscience, 2008.

[2]. Fang, X., et al. "A High-Performance Karatsuba Multiplier Architecture Based on Operand Splitting." *IEEE Transactions on Computers* 56.10 (2007): 1302-1310.

[3]. Zhu, Y., et al. "Efficient Montgomery Multiplication Using Karatsuba Algorithm on Reconfigurable Hardware." *Journal of Parallel and Distributed Computing* 71.10 (2011): 1383-1392.

[4]. Jain, N., et al. "A Karatsuba-Based Montgomery Multiplication Algorithm." *IEEE Access* 9 (2021): 12035-12042.

[5]. Rathore, V., et al. "FlexKA: A Flexible Karatsuba Multiplier Hardware Architecture for Variable-Sized Large Integers." *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023.

[6]. Bultmann, R., et al. "A Karatsuba-Based Montgomery Multiplication Algorithm for Large Integers." *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, 2007.