VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

REPORT
Logic Design with HDL

Topic:
MATRIX MULTIPLICATION

Instructor: Mr. Nguyễn Thành Lộc
Group No: 6
Class: CC02
Completion Day May 29, 2025
Group members: Lê Lương Trung Hiếu (2452330)
Trần Nguyễn Nguyên Khoa (2452563)
Hoàng Trọng Huy Minh (2452743)
Lê Hoàng Phúc (2452990)
Cao Thiên (2453192)

Contents

1 Chapter 1: Abstract and Introduction

Introduction:

- Efficient hardware implementation of matrix multiplication is a critical task in many computational domains, including signal processing, computer vision, and neural networks. To support signed arithmetic in these applications, high-performance signed multipliers are essential. This project explores and compares two popular signed multiplier architectures—Baugh-Wooley and Booth multipliers—through the implementation of a 4×4 matrix multiplier in Verilog HDL.
- Signed multiplication is particularly important in matrix operations where negative values are common. Two of the most prominent signed multiplication algorithms for hardware implementation are the **Baugh-Wooley multiplier and the Booth multiplier**. **The Baugh-Wooley algorithm** is optimized for **two's complement arithmetic** and offers a regular partial product structure, which simplifies logic design and timing closure.
- In contrast, **the Booth-based architecture** employs a sequential Booth multiplication algorithm, which processes the multiplier in 4-bit chunks.For 8-bit operands, this approach requires three clock cycles per multiplication. The control FSM includes a COMPUTE state to account for this latency, followed by an OUTPUT state where the sum of the four products is assigned. While Booth's recoding technique can reduce the number of partial products and may lead to area savings, this implementation trades off throughput by serializing each multiplication step and avoiding pipelining.
- This project presents a comparative HDL implementation of matrix multiplication using both Baugh-Wooley and Booth multipliers, written in Verilog. A 4×4 matrix multiplication module is developed, where each matrix element is an 8-bit signed number. The design incorporates four parallel multipliers to calculate each matrix element's dot product, increasing throughput. The core multiplication logic is modular, allowing for the integration of either Baugh-Wooley or Booth implementations.
- **And in this report**, using simulation and waveform analysis, we would analyze and test not only the correctness but also effectiveness of the theory and the code created based on it.

2 Chapter 2: Backgrounds and applications

Background

- Matrix multiplication is a foundational operation in digital signal processing, computer vision, robotics, cryptography, and machine learning. In these domains, data is frequently represented in matrix form, and processing often involves extensive matrix computations. Hardware acceleration of such computations is critical to meet real-time performance requirements, especially when operating on large datasets or performing complex tasks.
- Signed multipliers are a key component in these operations, especially when dealing with two's complement representation—commonly used in digital systems to handle negative values. Among the various multiplier architectures, **Baugh-Wooley** and **Booth** multipliers are notable for their suitability in signed arithmetic. Each architecture offers different trade-offs in terms of speed, area, power consumption, and design complexity.
- The **Baugh-Wooley multiplier** leverages the regularity of two's complement arithmetic to simplify the generation of partial products and improve timing performance. Its highly parallel structure makes it well-suited for high-throughput applications.
- The **Booth multiplier**, on the other hand, reduces the number of partial products through recoding, potentially leading to reduced hardware complexity. However, its sequential nature and dependency on control logic (e.g., FSM) can limit throughput if not properly pipelined.
- In Verilog-based digital system design, understanding the strengths and weaknesses of these architectures is essential for choosing the most appropriate multiplier for a given application.

Applications

Efficient signed matrix multiplication has wide-ranging applications, such as:

- **Digital Signal Processing (DSP):** Filters, transforms (e.g., DFT/FFT), and correlation algorithms rely heavily on fast matrix operations.
- **Machine Learning and Neural Networks:** Matrix multiplication forms the core of operations in dense layers, convolutional layers, and attention mechanisms. Signed multipliers are crucial when using quantized signed data.
- **Image and Video Processing:** Operations such as convolution, edge detection, and color space transformations use signed matrix multiplications extensively.
- **Control Systems and Robotics:** State estimation, sensor fusion, and motion planning often involve signed matrix arithmetic.

- **Embedded Systems and ASIC/FPGA Accelerators:** For real-time, low-power applications, implementing matrix multiplication in hardware using efficient signed multipliers is vital.

By exploring and comparing the Baugh-Wooley and Booth multipliers in the context of a 4×4 matrix multiplier, this project addresses both the theoretical and practical aspects of high-performance arithmetic circuit design. The results can inform design decisions in a wide range of applications that depend on signed, high-throughput, and resource-efficient matrix multiplications.

3 Chapter 3: Design

3.1 Ideal Design – Matrix Multiplication System

The system is designed to compute the product of two 4×4 signed integer matrices, A and B, using Verilog HDL. Each matrix signal has a size of 8 bits, in which MSB will be the sign bit and the remaining 7 bits will be the magnitude). The result matrix C will contain 17-bit signed numbers, is generated using a custom-designed multiply-accumulate process controlled by an FSM.

The system operates in **four main stages:**

3.1.1 Input Interface and Data Storage

- The input signal `din` port. receives 32 signed 8-bit integers sequentially, where:
  - The first 16 values are stored into matrix A.
  - The next 16 values are stored into matrix B.
- The signal `wrt_en` is used to trigger data storage at each clock cycle. Two register arrays (`A[0:15]` and `B[0:15])` are used to store the matrices.

3.1.2 Control Unit (FSM Controller)

- The process is managed by a finite state machine (FSM) with four states:
  - `IDLE`: Waiting for data input.
  - `LOAD_A`: Loads matrix A.
  - `LOAD_B`: Loads matrix B.
  - `OUTPUT`: Performs matrix multiplication and outputs results.
- State transitions are triggered by `wrt_en` and internal counters (`load_count`, `output_count`).

3.1.3 Multiply-Accumulate Core

- the system uses four parallel instances of a custom 8×8 signed multiplier module, `baugh_wooley_multiplier`, based on the Baugh–Wooley algorithm. These modules compute the partial products:
- For each result element `C[i][j]`, the system calculates:

$$C[i][j] = \sum_{k=0}^{3} A[i][k] \times B[k][j]$$

- The partial results are then summed to form the final output.

3.1.4 Output Handling

- Once the computation begins, the system outputs one result per clock cycle via the 17-bit `dout` port. After 16 results, the `done` signal is raised to indicate that the full 4×4 matrix multiplication is complete.

3.1.5 Design Highlights

- **Custom multiplier**: The Baugh–Wooley multiplier is implemented manually to support signed multiplication efficiently in hardware.
- **Parallelism**: Four multipliers run concurrently to increase throughput.
- **Simplicity**: The design does not yet apply pipelining, prioritizing clarity and functional correctness.
- **Expandability**: Can be upgraded to pipelined architecture or reused in larger matrix multipliers.

3.2 Block diagram

The matrix multiplier system consists of the following functional blocks:

- **Input Register:** Receives 8-bit signed data via the `din` port and stores it in matrix A or B based on `wrt_en` control.
- **Matrix A Storage:** Holds 16 elements of matrix A, each 8-bit signed.
- **Matrix B Storage:** Holds the next 16 elements for matrix B.
- **FSM Controller:** A finite state machine that manages the stages: loading A, loading B, computing, and outputting.
- **MAC Core (Multiply and Accumulate):** Contains four parallel custom Baugh–Wooley multipliers to compute `A[i][k] × B[k][j]` and accumulate the result.
- **Output Register:** Stores and outputs each 17-bit result `C[i][j]` sequentially via the `dout` port.

Input/Output signals:

- `clk, rst`: Clock and synchronous reset signals.
- `din [7:0]`: 8-bit signed input data.
- `wrt_en`: Write enable signal for loading data.
- `dout [16:0]`: 17-bit signed result output.
- `done`: Indicates completion of the computation.

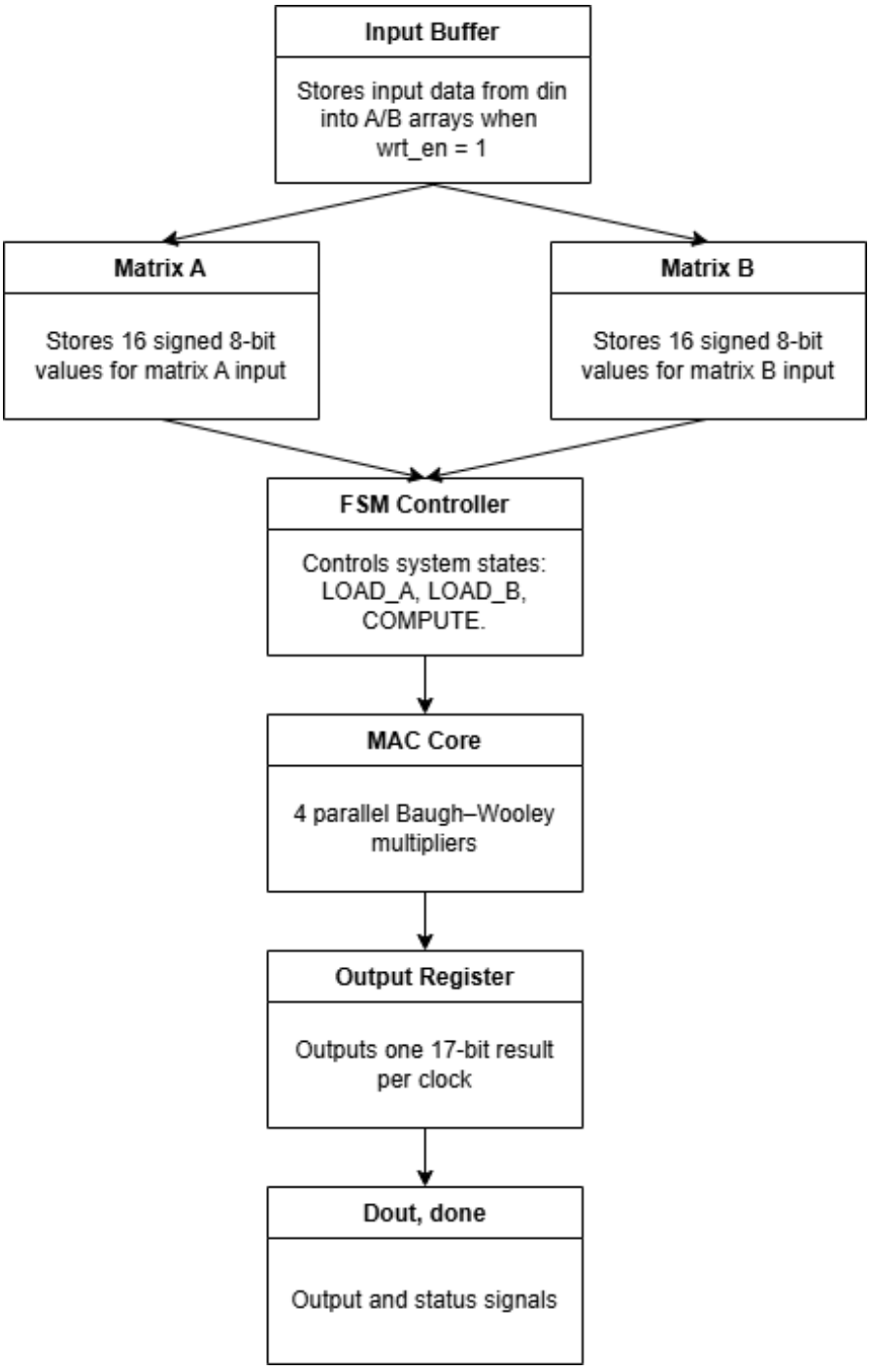

Figure 1: The Matrix Multiplier System Blocks

The Booth multiplier system consists of the following functional blocks:

- **Input Registers:** Receive two signed 8-bit inputs — `M` (multiplicand) and `R` (multiplier).
- **Register Setup:** The multiplicand is sign-extended to form `A`; the multiplier remains in `R`.
- **Booth Encoding Logic:** Analyzes the multiplier bits using the Booth algorithm to generate encoding patterns that reduce the number of partial products.
- **Control Signal Generator:** Produces control signals such as `PnM`, `M_Sel`, and `En` to guide partial product selection and operations.
- **Partial Product Generator:** Generates multiple shifted versions of the multiplicand: `Mx1`, `Mx2`, `Mx4`, and `Mx8`.
- **Booth Operations:** Includes B-op and C-op blocks that select and align partial products based on the Booth encoding and control logic.
- **Adder Tree Stages:** Multi-level adder structure that computes intermediate sums using carry-in and high-bit propagation: `T = Hi + B + Ci_B`, followed by `S = T + C + Ci_C`.
- **Guard Logic and Shift Bit:** Ensures correct bit-width alignment and overflow handling during shifting.
- **Shift and Accumulate:** Performs iterative shifting and accumulation with rounding/guard logic.
- **Final Result:** The final 16-bit signed product `P` is produced after all accumulation steps.

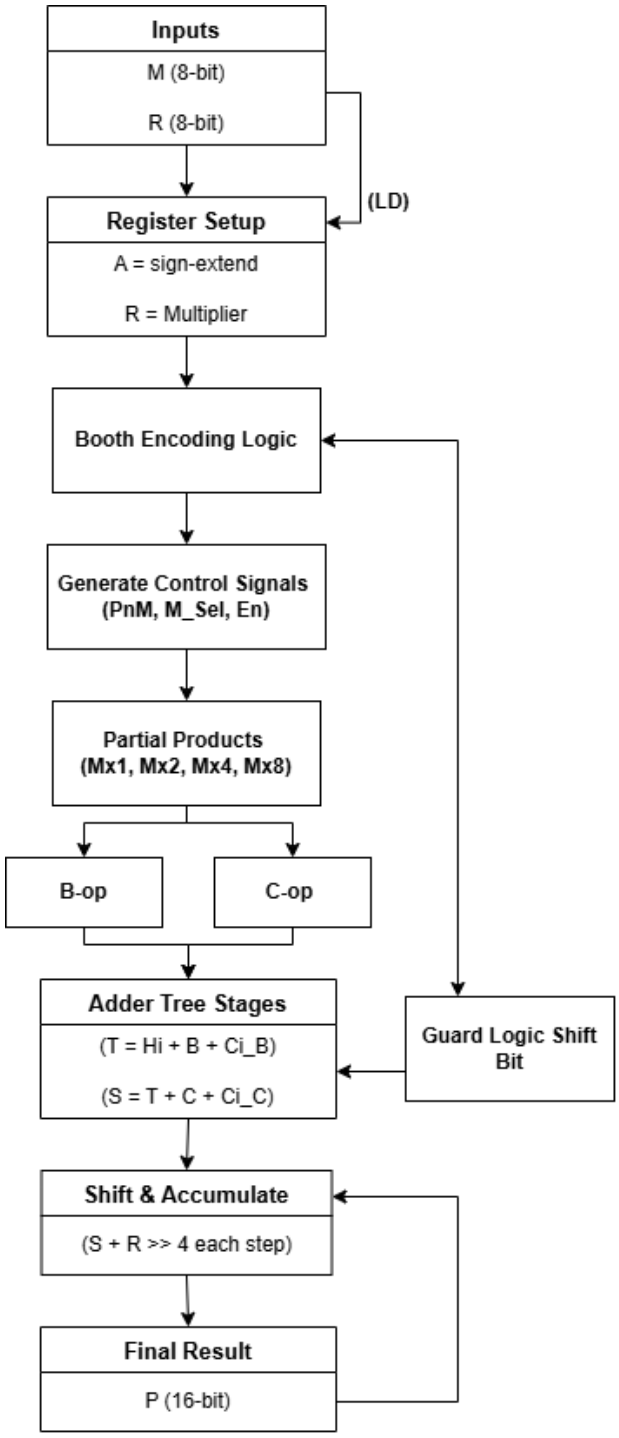

Figure 2: Booth Multiplier Functional Blocks

3.3 Flow chart

To coordinate the operation of the matrix multiplier system, a finite state machine (FSM) is implemented. The FSM consists of four main states:

- **IDLE:** Waits for the `wrt_en` signal to begin data loading.
- **LOAD_A:** Receives and stores the first 16 input values into Matrix A.
- **LOAD_B:** Continues receiving the next 16 values for Matrix B.
- **COMPUTE:** Performs the matrix multiplication using four parallel multipliers and outputs one result per clock cycle.

The FSM transitions between these states based on internal counters and input control signals. Once all 16 results are computed, the `done` signal is asserted, and the FSM returns to the IDLE state.

Figure 3 shows the overall control flow of the FSM.

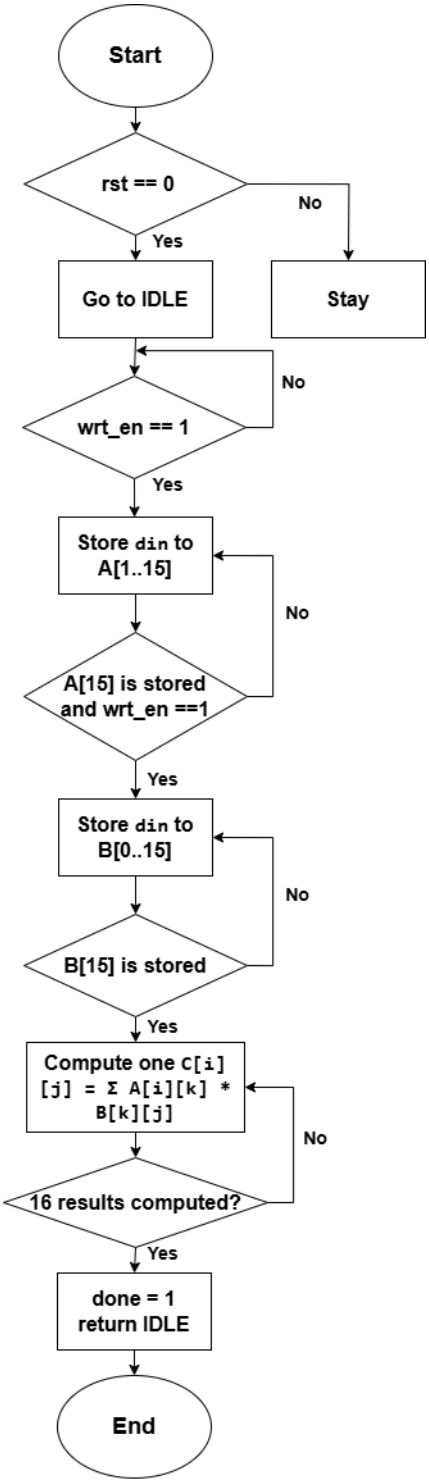

Figure 3: FSM flowchart for matrix multiplier control logic

3.4 Algorithms

Algorithm 1: FSM Controller for 4x4 Matrix Multiplication

Input: 32 signed 8-bit values via `din`, controlled by `wrt_en`
Output: 16 signed 17-bit values via `dout`

Initialize: state ← IDLE, load_count ← 0, output_count ← 0
while *true* **do**
  if *rst == 1* **then**
    Reset all states and counters
  end
  else
    if *state == IDLE* ***and*** *wrt_en* **then**
      Store din into A[0], load_count ← 1, state ← LOAD_A
    if *state == LOAD_A* ***and*** *wrt_en* **then**
      Store din into A[load_count], increment load_count
      if *load_count == 16* **then**
        state ← LOAD_B
      end
    if *state == LOAD_B* ***and*** *wrt_en* **then**
      Store din into B[load_count - 16], increment load_count
      if *load_count == 32* **then**
        output_count ← 0, state ← COMPUTE
      end
    if *state == COMPUTE* **then**
      Compute $C[i][j] = \sum A[i][k] \times B[k][j]$
      Output to `dout`, increment output_count
      if *output_count == 16* **then**
        Set `done` ← 1, state ← IDLE
      end
  end
end

4 Chapter 4: Implement

4.1 Module: Baugh-Wooley multiplier

```
‘timescale 1ns / 1ps

module Matrix_multi(
    input wire clk, rst, wrt_en,
    input wire signed [7:0] din,
    output reg signed [16:0] dout
);
    reg signed [7:0] A [0:15];
    reg signed [7:0] B [0:15];

    reg [2:0] state;
    reg [5:0] load_count;
    reg [3:0] output_count;

    localparam IDLE = 3’d0,
               LOAD_A = 3’d1,
               LOAD_B = 3’d2,
               OUTPUT = 3’d3;

    wire [1:0] row = output_count[3:2];
    wire [1:0] col = output_count[1:0];

    wire signed [15:0] p0, p1, p2, p3;
    wire signed [16:0] sum;

    baugh_wooley_multiplier bw0(.a(A[{row,2’b00}]), .b(B[{2’b00,col}]), .p(p0));
    baugh_wooley_multiplier bw1(.a(A[{row,2’b01}]), .b(B[{2’b01,col}]), .p(p1));
    baugh_wooley_multiplier bw2(.a(A[{row,2’b10}]), .b(B[{2’b10,col}]), .p(p2));
    baugh_wooley_multiplier bw3(.a(A[{row,2’b11}]), .b(B[{2’b11,col}]), .p(p3));

    assign sum = p0 + p1 + p2 + p3;

    always @(posedge clk) begin
        if (rst) begin
            state <= IDLE;
            load_count <= 0;
            output_count <= 0;
            dout <= 0;
        end else begin
            case (state)
                IDLE:
                    if (wrt_en) begin
                        A[0] <= din;
                        load_count <= 1;
                        state <= LOAD_A;
                    end
                LOAD_A:
                    if (wrt_en) begin
                        A[load_count] <= din;
```

```
                    load_count <= load_count + 1;
                    if (load_count == 15)
                        state <= LOAD_B;
                end
            LOAD_B:
                if (wrt_en) begin
                    B[load_count - 16] <= din;
                    load_count <= load_count + 1;
                    if (load_count == 31) begin
                        state <= OUTPUT;
                        output_count <= 0;
                    end
                end
            OUTPUT:
                begin
                    dout <= sum;
                    output_count <= output_count + 1;
                    if (output_count == 15)
                        state <= IDLE;
                end
        endcase
    end
  end
endmodule
```

Listing 1: Matrix_multi Module with Baugh-Wooley Multipliers

```
module baugh_wooley_multiplier(
    input wire signed [7:0] a,
    input wire signed [7:0] b,
    output reg signed [15:0] p
);
    integer i, j;
    reg signed [15:0] acc;
    reg bit;

    always @* begin
        acc = 16'sd0;

        for (i = 0; i <= 6; i = i + 1) begin
            for (j = 0; j <= 6; j = j + 1) begin
                bit = a[i] & b[j];
                acc = acc + (bit << (i + j));
            end
        end

        for (j = 0; j <= 6; j = j + 1) begin
            bit = a[7] & b[j];
            acc = acc - (bit << (7 + j));
        end

        for (i = 0; i <= 6; i = i + 1) begin
            bit = a[i] & b[7];
            acc = acc - (bit << (i + 7));
```

```
        end

        bit = a[7] & b[7];
        acc = acc + (bit << 14);

        p = acc;
    end
endmodule
```

Listing 2: Baugh-Wooley Multiplier Module

```
`timescale 1ns / 1ps
module tb_Matrix_multi_scaled_time;

    // Inputs
    reg clk, rst, wrt_en;
    reg signed [7:0] din;

    // Outputs
    wire signed [16:0] dout;

    // Internal signals for monitoring
    wire [2:0] state_monitor;
    wire [3:0] output_count_monitor;

    // Instantiate the Unit Under Test (UUT)
    Matrix_multi uut (
        .clk(clk),
        .rst(rst),
        .wrt_en(wrt_en),
        .din(din),
        .dout(dout)
    );

    // Expose internal signals for monitoring
    assign state_monitor = uut.state;
    assign output_count_monitor = uut.output_count;

    // Clock generation: 2ns period (for faster simulation)
    always #1 clk = ~clk;

    // Matrix and result storage
    reg signed [7:0] matrix_A [0:15];
    reg signed [7:0] matrix_B [0:15];
    integer i;

    initial begin
        // Initialize signals
        clk = 0;
        rst = 1;
        wrt_en = 0;
        din = 0;

        // Brief reset (2ns)
```

```
        #2 rst = 0;

        // Initialize matrix_A
        matrix_A[0] = 8'd3; matrix_A[1] = 8'd7; matrix_A[2] = 8'd12; matrix_A[3] = 8'd2;
        matrix_A[4] = 8'd1; matrix_A[5] = 8'd4; matrix_A[6] = 8'd6; matrix_A[7] = 8'd8;
        matrix_A[8] = 8'd9; matrix_A[9] = 8'd5; matrix_A[10] = 8'd10; matrix_A[11] =
            8'd11;
        matrix_A[12] = 8'd7; matrix_A[13] = 8'd2; matrix_A[14] = 8'd4; matrix_A[15] =
            8'd8;

        // Initialize matrix_B
        matrix_B[0] = 8'd5; matrix_B[1] = 8'd1; matrix_B[2] = 8'd3; matrix_B[3] = 8'd8;
        matrix_B[4] = 8'd2; matrix_B[5] = 8'd7; matrix_B[6] = 8'd4; matrix_B[7] = 8'd6;
        matrix_B[8] = 8'd9; matrix_B[9] = 8'd5; matrix_B[10] = 8'd2; matrix_B[11] = 8'd1;
        matrix_B[12] = 8'd3; matrix_B[13] = 8'd6; matrix_B[14] = 8'd1; matrix_B[15] =
            8'd7;

        // --- Write matrix A ---
        #2 wrt_en = 1;
        for (i = 0; i < 16; i = i + 1) begin
            din = matrix_A[i];
            #2;
        end
        wrt_en = 0;

        // --- Write matrix B ---
        wrt_en = 1;
        for (i = 0; i < 16; i = i + 1) begin
            din = matrix_B[i];
            #2;
        end
        wrt_en = 0;

        // Wait for OUTPUT state
        wait(state_monitor == 3);

        // Display outputs
        for (i = 0; i < 16; i = i + 1) begin
            @(posedge clk);
            $display("Output␣%0d:␣%0d", i, dout);
        end

        #10 $finish;
    end

endmodule
```

Listing 3: Testbench for Matrix_multi (Scaled Time)

4.2 Booth multiplier

```
module Matrix_multi(
```

```
    input wire clk, rst, wrt_en,
    input wire signed [7:0] din,
    output reg signed [16:0] dout
);
    reg signed [7:0] A [0:15];
    reg signed [7:0] B [0:15];

    reg [2:0] state;
    reg [5:0] load_count;
    reg [3:0] output_count;
    reg [1:0] compute_count; // Counter for 3-cycle multiplication latency

    localparam IDLE = 3'd0,
               LOAD_A = 3'd1,
               LOAD_B = 3'd2,
               COMPUTE = 3'd3,
               OUTPUT = 3'd4;

    wire [1:0] row = output_count[3:2];
    wire [1:0] col = output_count[1:0];

    wire signed [15:0] p0, p1, p2, p3;
    wire valid0, valid1, valid2, valid3;
    reg ld; // Load signal for Booth multipliers
    wire signed [16:0] sum;

    // Instantiate four Booth multipliers
    Booth_Multiplier_4xA #(
        .N(8)
    ) bw0 (
        .Rst(rst),
        .Clk(clk),
        .Ld(ld),
        .M(A[{row,2'b00}]),
        .R(B[{2'b00,col}]),
        .Valid(valid0),
        .P(p0)
    );
    Booth_Multiplier_4xA #(
        .N(8)
    ) bw1 (
        .Rst(rst),
        .Clk(clk),
        .Ld(ld),
        .M(A[{row,2'b01}]),
        .R(B[{2'b01,col}]),
        .Valid(valid1),
        .P(p1)
    );
    Booth_Multiplier_4xA #(
        .N(8)
    ) bw2 (
        .Rst(rst),
```

```
        .Clk(clk),
        .Ld(ld),
        .M(A[{row,2'b10}]),
        .R(B[{2'b10,col}]),
        .Valid(valid2),
        .P(p2)
    );
    Booth_Multiplier_4xA #(
        .N(8)
    ) bw3 (
        .Rst(rst),
        .Clk(clk),
        .Ld(ld),
        .M(A[{row,2'b11}]),
        .R(B[{2'b11,col}]),
        .Valid(valid3),
        .P(p3)
    );

    assign sum = p0 + p1 + p2 + p3;

    always @(posedge clk) begin
        if (rst) begin
            state <= IDLE;
            load_count <= 0;
            output_count <= 0;
            compute_count <= 0;
            ld <= 0;
            dout <= 0;
        end else begin
            case (state)
                IDLE: begin
                    if (wrt_en) begin
                        A[0] <= din;
                        load_count <= 1;
                        state <= LOAD_A;
                    end
                end
                LOAD_A: begin
                    if (wrt_en) begin
                        A[load_count] <= din;
                        load_count <= load_count + 1;
                        if (load_count == 15) state <= LOAD_B;
                    end
                end
                LOAD_B: begin
                    if (wrt_en) begin
                        B[load_count-16] <= din;
                        load_count <= load_count + 1;
                        if (load_count == 31) begin
                            state <= COMPUTE;
                            output_count <= 0;
                            compute_count <= 0;
```

```
                        ld <= 1; // Start first multiplication
                    end
                end
            end
            COMPUTE: begin
                ld <= 0; // Pulse ld for one cycle
                compute_count <= compute_count + 1;
                if (compute_count == 2) begin // 3 cycles total (0,1,2)
                    state <= OUTPUT;
                    compute_count <= 0;
                end
            end
            OUTPUT: begin
                dout <= sum;
                output_count <= output_count + 1;
                ld <= 1; // Start next multiplication
                if (output_count == 15) begin
                    state <= IDLE;
                    ld <= 0;
                end else begin
                    state <= COMPUTE; // Return to COMPUTE for next element
                end
            end
        endcase
    end
  end
endmodule
```

Listing 4: `Matrix_multi` Module Implementation for Booth module

```
module Booth_Multiplier_4xA #(
    parameter N = 8
)(
    input Rst,
    input Clk,
    input Ld,
    input [(N - 1):0] M,
    input [(N - 1):0] R,
    output reg Valid,
    output reg [((2*N) - 1):0] P
);

    localparam pNumCycles = ((N + 1)/4);

    reg [4:0] Cntr;
    reg [4:0] Booth;
    reg Guard;
    reg [(N + 3):0] A;
    wire [(N + 3):0] Mx8, Mx4, Mx2, Mx1;
    reg PnM_B, M_Sel_B, En_B;
    reg PnM_C, M_Sel_C, En_C;
    wire [(N + 3):0] Hi;
    reg [(N + 3):0] B, C;
    reg Ci_B, Ci_C;
```

```
25      wire [(N + 3):0] T, S;
26      reg [((2*N) + 3):0] Prod;
27
28      always @(posedge Clk)
29      begin
30          if(Rst)
31              Cntr <= #1 0;
32          else if(Ld)
33              Cntr <= #1 pNumCycles;
34          else if(|Cntr)
35              Cntr <= #1 (Cntr - 1);
36      end
37
38      always @(posedge Clk)
39      begin
40          if(Rst)
41              A <= #1 0;
42          else if(Ld)
43              A <= #1 {{4{M[(N - 1)]}}, M};
44      end
45
46      assign Mx8 = {A, 3'b0};
47      assign Mx4 = {A, 2'b0};
48      assign Mx2 = {A, 1'b0};
49      assign Mx1 = A;
50
51      always @(*) Booth <= {Prod[3:0], Guard};
52      assign Hi = Prod[((2*N) + 3):N];
53
54      // Control logic for operand B
55      always @(*)
56      begin
57          case(Booth)
58              5'b00000, 5'b00001, 5'b00010, 5'b00011, 5'b00100,
59              5'b11011, 5'b11100, 5'b11101, 5'b11110, 5'b11111 :
60                  {PnM_B, M_Sel_B, En_B} <= 3'b000;
61              5'b00101, 5'b00110, 5'b00111, 5'b01000, 5'b01001, 5'b01010 :
62                  {PnM_B, M_Sel_B, En_B} <= 3'b001;
63              5'b01101, 5'b01110, 5'b01111 :
64                  {PnM_B, M_Sel_B, En_B} <= 3'b011;
65              5'b10000, 5'b10001, 5'b10010 :
66                  {PnM_B, M_Sel_B, En_B} <= 3'b111;
67              default :
68                  {PnM_B, M_Sel_B, En_B} <= 3'b101;
69          endcase
70      end
71
72      // Control logic for operand C
73      always @(*)
74      begin
75          case(Booth)
76              5'b00000, 5'b01111, 5'b10111, 5'b11000, 5'b11111 :
77                  {PnM_C, M_Sel_C, En_C} <= 3'b000;
```

```
78              5'b00001, 5'b00010, 5'b01001, 5'b01010,
79              5'b10001, 5'b10010, 5'b11001, 5'b11010 :
80                  {PnM_C, M_Sel_C, En_C} <= 3'b001;
81              5'b00011, 5'b00100, 5'b01011, 5'b01100 :
82                  {PnM_C, M_Sel_C, En_C} <= 3'b011;
83              5'b00101, 5'b00110, 5'b01101, 5'b01110,
84              5'b10101, 5'b10110, 5'b11101, 5'b11110 :
85                  {PnM_C, M_Sel_C, En_C} <= 3'b101;
86              default :
87                  {PnM_C, M_Sel_C, En_C} <= 3'b111;
88          endcase
89      end
90
91      // Operand B mux logic
92      always @(*)
93      begin
94          case({PnM_B, M_Sel_B, En_B})
95              3'b001: {Ci_B, B} <= {1'b0, Mx4};
96              3'b011: {Ci_B, B} <= {1'b0, Mx8};
97              3'b101: {Ci_B, B} <= {1'b1, ~Mx4};
98              3'b111: {Ci_B, B} <= {1'b1, ~Mx8};
99              default: {Ci_B, B} <= 0;
100         endcase
101     end
102
103     // Operand C mux logic
104     always @(*)
105     begin
106         case({PnM_C, M_Sel_C, En_C})
107             3'b001: {Ci_C, C} <= {1'b0, Mx1};
108             3'b011: {Ci_C, C} <= {1'b0, Mx2};
109             3'b101: {Ci_C, C} <= {1'b1, ~Mx1};
110             3'b111: {Ci_C, C} <= {1'b1, ~Mx2};
111             default: {Ci_C, C} <= 0;
112         endcase
113     end
114
115     assign T = Hi + B + Ci_B;
116     assign S = T + C + Ci_C;
117
118     always @(posedge Clk)
119     begin
120         if(Rst)
121             Prod <= #1 0;
122         else if(Ld)
123             Prod <= #1 R;
124         else if(|Cntr)
125             Prod <= #1 {{4{S[(N + 3)]}}, S, Prod[(N - 1):4]};
126     end
127
128     always @(posedge Clk)
129     begin
130         if(Rst)
```

```
            Guard <= #1 0;
        else if(Ld)
            Guard <= #1 0;
        else if(|Cntr)
            Guard <= #1 Prod[3];
    end

    always @(posedge Clk)
    begin
        if(Rst)
            P <= #1 0;
        else if(Cntr == 1)
            P <= #1 {S, Prod[(N - 1):4]};
    end

    always @(posedge Clk)
    begin
        if(Rst)
            Valid <= #1 0;
        else
            Valid <= #1 (Cntr == 1);
    end
endmodule
```

Listing 5: Booth Multiplier 4xA

```
`timescale 1ns / 1ps

module tb_Matrix_multi_scaled_time;

    // Inputs
    reg clk, rst, wrt_en;
    reg signed [7:0] din;

    // Outputs
    wire signed [16:0] dout;

    // Internal signals for monitoring
    wire [2:0] state_monitor;
    wire [3:0] output_count_monitor;

    // Instantiate the Unit Under Test (UUT)
    Matrix_multi uut (
        .clk(clk),
        .rst(rst),
        .wrt_en(wrt_en),
        .din(din),
        .dout(dout)
    );

    // Expose internal signals for monitoring
    assign state_monitor = uut.state;
    assign output_count_monitor = uut.output_count;

```

```
    // Clock generation: 2ns period (for faster simulation)
    always #1 clk = ~clk;

    // Matrix and result storage
    reg signed [7:0] matrix_A [0:15];
    reg signed [7:0] matrix_B [0:15];
    integer i;

    initial begin
        // Initialize signals
        clk = 0;
        rst = 1;
        wrt_en = 0;
        din = 0;

        // Brief reset (2ns)
        #2 rst = 0;

        // Initialize matrix_A with 1- and 2-digit decimal numbers
        matrix_A[0] = 8'd3; matrix_A[1] = 8'd7; matrix_A[2] = 8'd12; matrix_A[3] = 8'd2;
        matrix_A[4] = 8'd1; matrix_A[5] = 8'd4; matrix_A[6] = 8'd6; matrix_A[7] = 8'd8;
        matrix_A[8] = 8'd9; matrix_A[9] = 8'd5; matrix_A[10] = 8'd10; matrix_A[11] =
            8'd11;
        matrix_A[12] = 8'd7; matrix_A[13] = 8'd2; matrix_A[14] = 8'd4; matrix_A[15] =
            8'd8;

        // Initialize matrix_B with 1- and 2-digit decimal numbers
        matrix_B[0] = 8'd5; matrix_B[1] = 8'd1; matrix_B[2] = 8'd3; matrix_B[3] = 8'd8;
        matrix_B[4] = 8'd2; matrix_B[5] = 8'd7; matrix_B[6] = 8'd4; matrix_B[7] = 8'd6;
        matrix_B[8] = 8'd9; matrix_B[9] = 8'd5; matrix_B[10] = 8'd2; matrix_B[11] = 8'd1;
        matrix_B[12] = 8'd3; matrix_B[13] = 8'd6; matrix_B[14] = 8'd1; matrix_B[15] =
            8'd7;

        // --- Write matrix A ---
        #2 wrt_en = 1;
        for (i = 0; i < 16; i = i + 1) begin
            din = matrix_A[i];
            #2;
        end
        wrt_en = 0;

        // --- Write matrix B ---
        wrt_en = 1;
        for (i = 0; i < 16; i = i + 1) begin
            din = matrix_B[i];
            #2;
        end
        wrt_en = 0;

        // Wait for OUTPUT state
        wait(state_monitor == 3);

        // Print each output
```

```
79          for (i = 0; i < 16; i = i + 1) begin
80              @(posedge clk);
81              $display("Output␣%0d:␣%0d", i, dout);
82          end
83
84          #200 $finish;
85      end
86
87  endmodule
```

Listing 6: Testbench for Booth module

Code overview

This Verilog module performs matrix multiplication of two 4×4 matrices, A and B, using Booth multipliers. The result is a matrix $C = A \times B$, with each element $C[i][j]$ computed one at a time.

Inputs and Outputs

- **Inputs:**
  - `clk`: Clock signal.
  - `rst`: Reset signal.
  - `wrt_en`: Write enable for loading data.
  - `din` (signed 8-bit): Input data stream for matrices A and B.
- **Output:**
  - `dout` (signed 17-bit): Resulting element $C[i][j]$.

Internal Registers and Signals

- **Matrices:**
  - `A[0:15]`, `B[0:15]`: Flattened row-major storage of 4×4 matrices.
- **FSM State Register:**
  - `state` (3-bit): Tracks module states: IDLE, LOAD_A, LOAD_B, COMPUTE, OUTPUT.
- **Counters:**
  - `load_count` (6-bit): Counts loaded elements.
  - `output_count` (4-bit): Indexes output matrix elements.
  - `compute_count` (2-bit): Handles 3-cycle multiplier latency.

- **Indexing:**
  - `row` = `output_count[3:2]`, `col` = `output_count[1:0]`
- **Multiplier Outputs:**
  - `p0-p3` (signed 16-bit): Partial products.
  - `valid0-valid3`: Validity flags for multiplier outputs.
- **Sum:**

$$\texttt{sum} = p0 + p1 + p2 + p3 \quad \text{(17-bit)}$$

Finite State Machine (FSM)

1. **IDLE**: Waits for `wrt_en`, transitions to LOAD_A.
2. **LOAD_A**: Loads 16 elements of A.
3. **LOAD_B**: Loads 16 elements of B.
4. **COMPUTE**: Pulses `ld`, waits 3 cycles.
5. **OUTPUT**: Assigns `sum` to `dout`, increments `output_count`.

Booth Multiplier Module: Booth_Multiplier_4xA

A sequential Booth multiplier module for two 8-bit signed inputs producing a 16-bit signed product over 3 cycles.

Parameters and I/O

- **Parameter:** $N = 8$
- **Inputs:**
  - `Rst, Clk, Ld`
  - `M, R` (8-bit signed): Multiplicand and multiplier
- **Outputs:**
  - `Valid`: High when product is ready.
  - `P`: 16-bit signed product.

Operation

- On `Ld`:
  - `Cntr` set to 3
  - `A` loaded with M
  - `Prod` loaded with R
  - `Guard` set to 0
- Each cycle:
  - `Booth` = { `Prod[3:0], Guard` }
  - Decoded to determine operation using shifted A
  - Operand B and C selected or disabled
- Adder Tree:
  - $T =$ `Hi + B + Ci_B`
  - $S = T + C +$ `Ci_C`
- Update:
  - `Prod` shifts right by 4, `S` inserted at top
  - `Guard` updated from `Prod[3]`
- Completion:
  - When `Cntr == 1`, `P` set to product, `Valid` is asserted

4.3 Comparison

Booth Multiplier

- Reference from `Booth_Multiplier_4xA[1]`, a sequential multiplier that processes the multiplier in 4-bit chunks.
- For an 8-bit multiplier ($N = 8$), requires $\lceil (N+1)/4 \rceil = 3$ cycles.
- FSM includes a `COMPUTE` state for 3-cycle latency, followed by `OUTPUT` to assign the sum.
- Four instances compute $A[i][k] \times B[k][j]$ for $k = 0$ to 3, and their outputs are summed as:

$$\texttt{sum} = p_0 + p_1 + p_2 + p_3$$

- No pipelining; each output element waits for previous multiplication to complete.

Baugh-Wooley Multiplier

- Uses the Baugh-Wooley technique for matrix multiplication.
- Product computed in one clock cycle after inputs stabilize.
- FSM lacks a `COMPUTE` state; the `OUTPUT` state directly assigns the result.
- Four multiplier instances compute partial products, with immediate summation.

Latency and Throughput

Booth Multiplier

- **Per Output Latency:** 4 clock cycles per element:
  - 1 cycle in `OUTPUT` to assign `ld`.
  - 3 cycles in `COMPUTE` to generate valid output.
- **Total Time for 16 Outputs:**

  2 (reset)+16 (LOAD_A)+16 (LOAD_B)+3 (initial COMPUTE)+16×4 (per output) = 101 cycles
- **Total Time (2 ns clock):** 101×2 ns $= 202$ ns
- **Throughput:** 1 output every 4 cycles.

Baugh-Wooley Multiplier

- **Per Output Latency:** 1 clock cycle per element (fully combinational multiplication).
- **Total Time for 16 Outputs:**

  $$2 \text{ (reset)} + 16 \text{ (LOAD\_A)} + 16 \text{ (LOAD\_B)} + 16 \text{ (OUTPUT)} = 50 \text{ cycles}$$
- **Total Time (2 ns clock):** 50×2 ns $= 100$ ns
- **Throughput:** 1 output every cycle.

Resource Usage

Booth Multiplier

- Each `Booth_Multiplier_4xA` is sequential and uses registers and a pipelined adder tree.
- Estimated LUTs per 8-bit multiplier: $\sim$ 50–100
- Total for 4 multipliers: $\sim$ 200–400 LUTs

- Flip-flops per multiplier: $\sim$ 20–50, total $\sim$ 80–200
- Lower combinational delay may support faster clock (e.g., < 2 ns), but limited by sequential latency.

Baugh-Wooley Multiplier

- Each `baugh_wooley_multiplier` is combinational, with full partial product arrays and correction terms.
- Estimated LUTs per 8-bit multiplier: $\sim$ 200–300
- Total for 4 multipliers: $\sim$ 800–1200 LUTs
- Minimal flip-flops used ($\sim$ 0–20 total)
- Higher combinational delay may necessitate slower clock (e.g., 5–10 ns depending on synthesis).

5 Chapter 5: Results

5.1 Experiment setups

The matrix multiplier design was verified using a custom testbench written in Verilog and simulated using Vivado Simulator. The experimental environment and conditions are summarized as follows:

- **Simulation Tool:** Vivado Simulator
- **Testbench:** `tb_Matrix_multi`
- **Clock Period:** 2 ns (500 MHz)
- **Reset Duration:** 0–4 ns (2 clock cycles)
- **Input Signal:** `din` (8-bit signed), loaded sequentially
- **Write Control:** `wrt_en` signal enabled during LOAD_A and LOAD_B
- **Data Count:** 16 values for Matrix A and 16 values for Matrix B
- **Output Signal:** `dout` (17-bit signed), 1 element per cycle during COMPUTE
- **FSM States:** IDLE → LOAD_A → LOAD_B → COMPUTE → IDLE

The goal of the simulation is to verify the functional correctness of data loading, multiplication using four parallel cores, and the sequential output of 16 computed values. Waveform snapshots were captured for each FSM stage and are presented in the following section.

5.2 Waveform

5.2.1 Waveform for Baugh-Wooley method

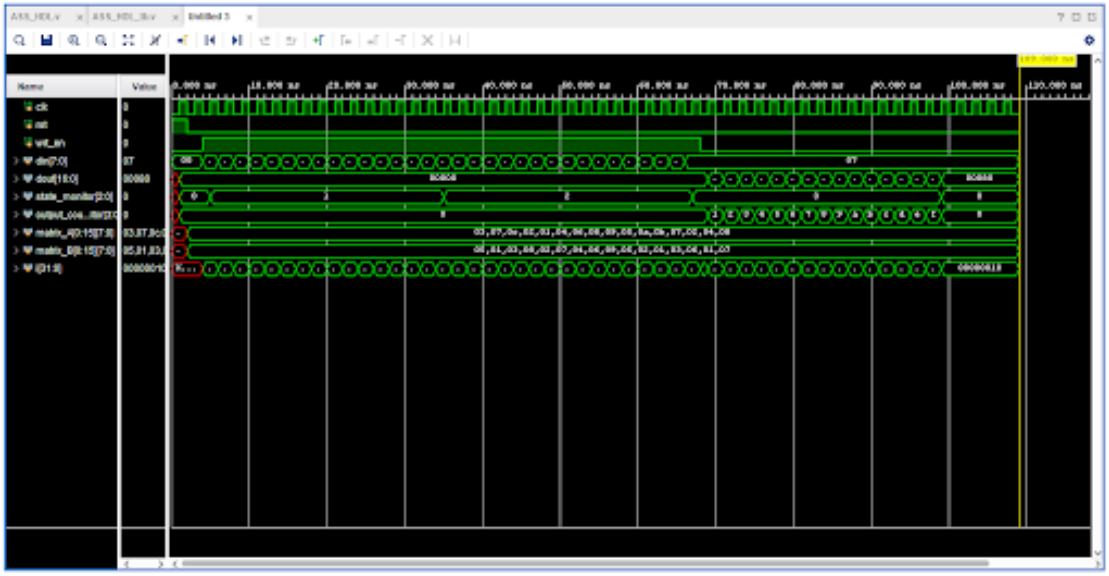

Figure 4: Waveform

5.2.2 Explanation:

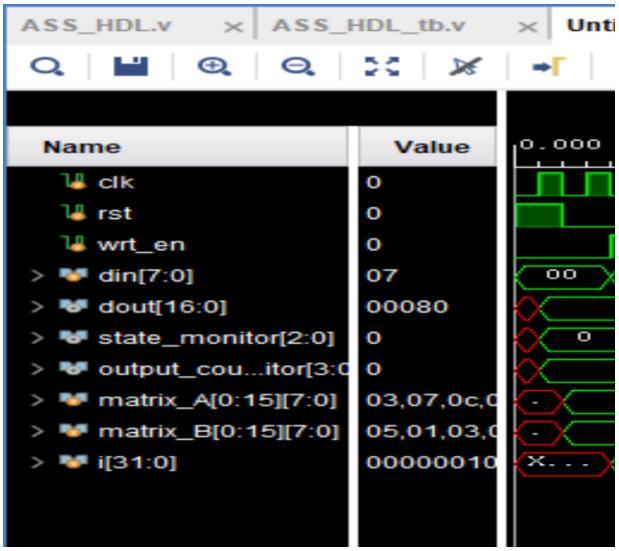

Figure 5: STATE IDLE

- The system remains in the `IDLE` state and waits for the `wrt_en` signal to be asserted.
- Once `wrt_en` is high, the first value from `din` is stored in `A[0]`.
- The internal counter `load_count` is initialized to 1 to track the next input index.
- The `done` signal is also reset to 0 to indicate that the system is ready for a new computation cycle.
- After this initialization step, which takes exactly one clock cycle (2 ns), the FSM transitions to the `LOAD_A` state.

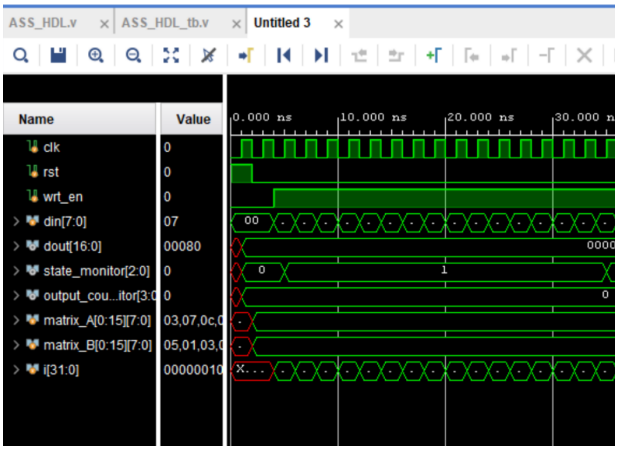

Figure 6: STATE LOAD_A

- The system waits for the `wrt_en` signal to be asserted.
- Once active, input values from the `din` port are transferred sequentially into matrix `A`, starting from `A[1]` and continuing to `A[15]`.
- (Note: `A[0]` was already loaded during the IDLE state.)
- The entire LOAD_A stage spans from **5 ns to 35 ns**, which corresponds to 15 clock cycles at 2 ns per cycle.
- After successfully loading all 16 elements into matrix A, the FSM transitions to the `LOAD_B` state.

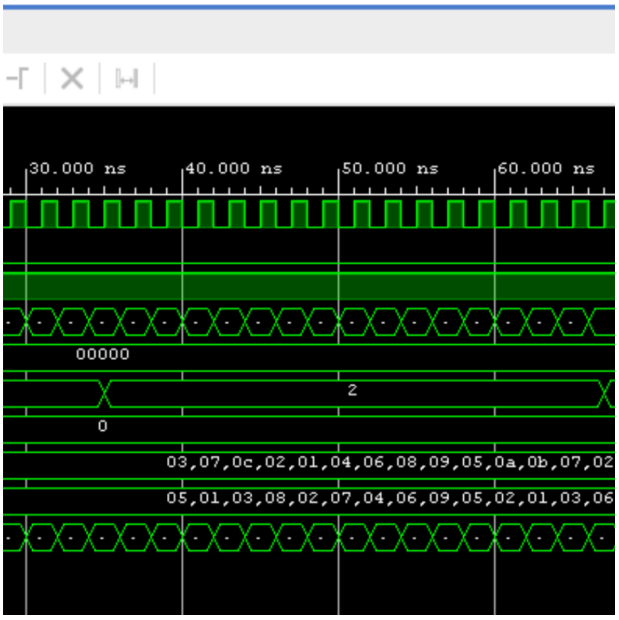

Figure 7: STATE LOAD_B

- The system waits for the `wrt_en` signal to be enabled.
- Once active, the values from the input port `din` are sequentially transferred into matrix `B`, starting from `B[0]` up to `B[15]`.
- The process is controlled by an internal counter that ensures exactly 16 values are stored.
- The entire LOAD_B stage occurs from **35 ns to 67 ns**, which corresponds to 16 clock cycles with a 2 ns period per cycle.
- After completing the loading of matrix B, the FSM automatically transitions to the COMPUTE state.

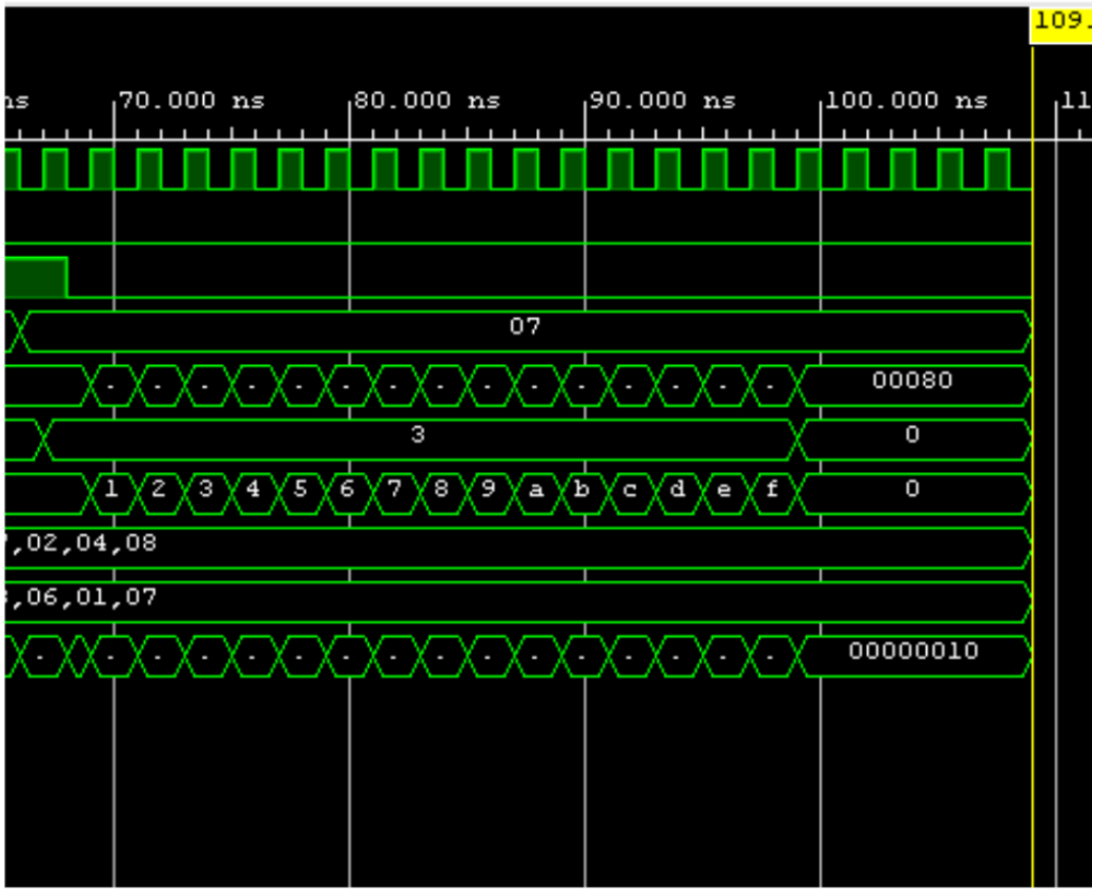

Figure 8: STATE COMPUTE & OUTPUT

- During the COMPUTE state, the system calculates one result element at position `C[i][j]` in the output matrix.
- Each element is computed using four partial products, obtained from the `baugh_wooley_multiplier` module instances.
- These four multipliers operate in parallel to compute:

$$\texttt{sum} = p_0 + p_1 + p_2 + p_3$$

  where each p_k is a signed 16-bit product.
- The result of this computation is assigned to the output signal `dout`.
- The full COMPUTE phase occurs from **67 ns to 99 ns**, producing 16 output values sequentially, one per clock cycle.
- After completing all outputs, the FSM transitions back to the `IDLE` state and remains there for 10 ns before simulation ends.

5.2.3 Waveform for Booth multiplier method

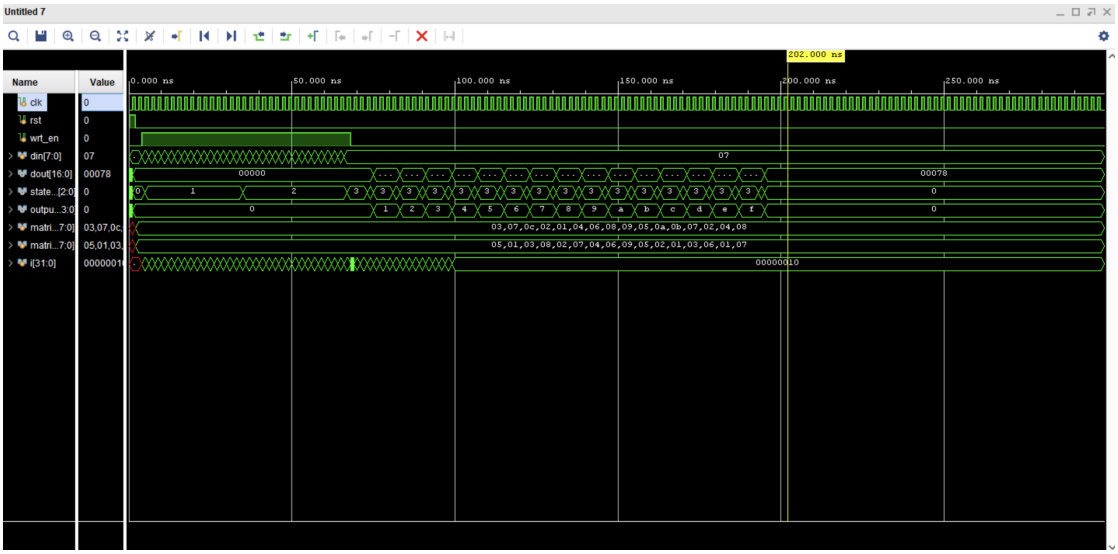

Figure 9: Waveform

5.2.4 Explanation:

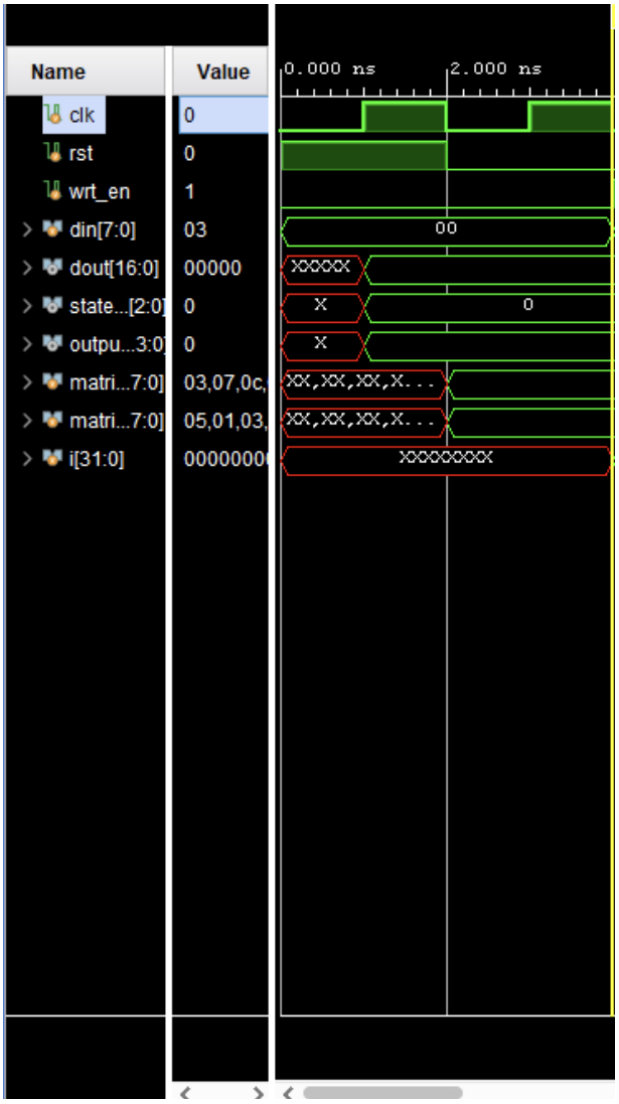

Figure 10: STATE IDLE

- 2 cycles of Reset (reset and wait for wrt_en) from 0-4 ns.

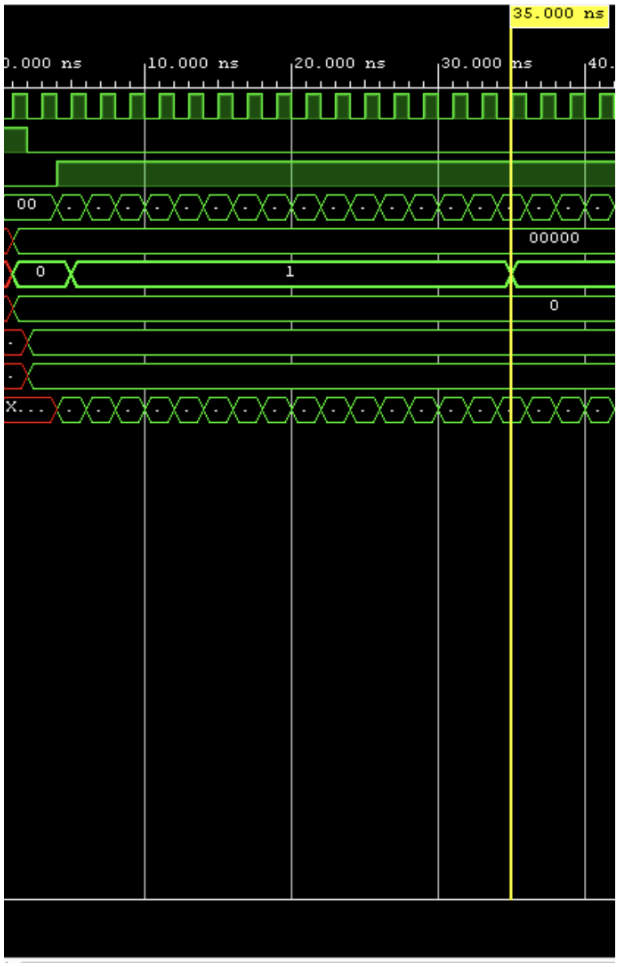

Figure 11: LOAD A

- wrt_en turns on, wait for state LOAD A to 5ns.
- LOAD A for 16 clock cycles = 32ns
- All processes are from 4ns -> 35ns.

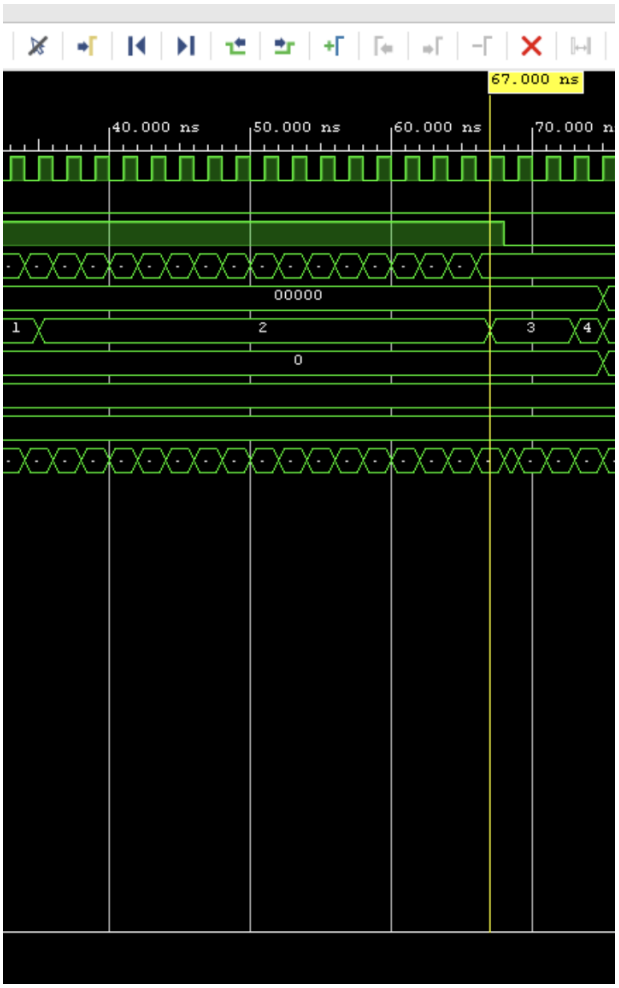

Figure 12: LOAD B

- LOAD B need 16 clock cycles to take in 16 elements from the matrix. needs 32 ns. The process takes place from 35 ns -> 67 ns.

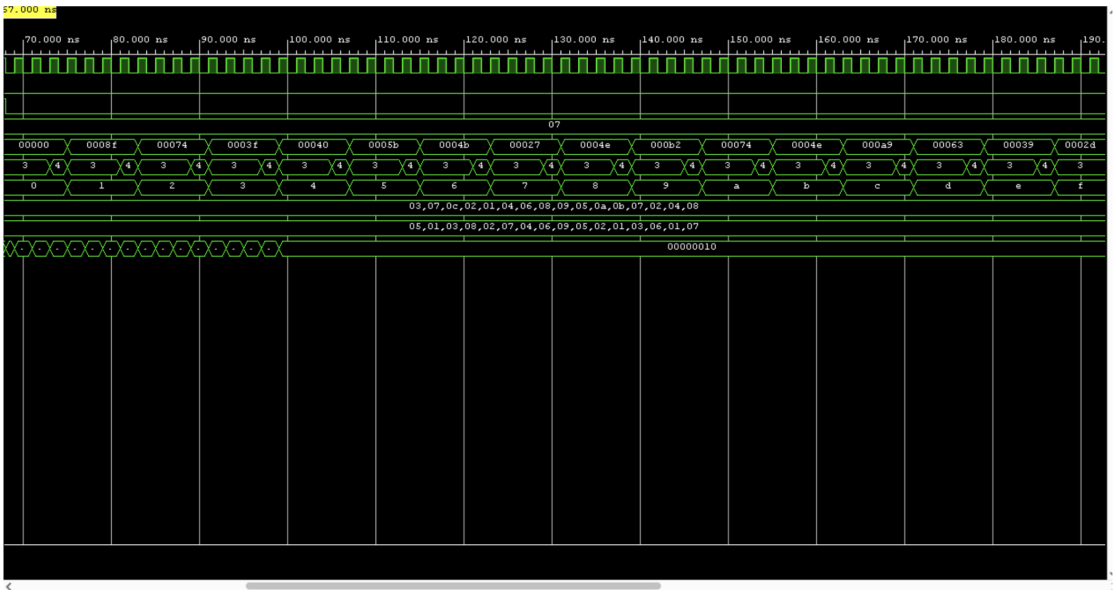

Figure 13: COMPUTE & OUTPUT

- COMPUTE state is to handle the 3-cycle latency of the Booth multipliers.
- The ld signal is pulsed for one cycle to start the four multiplications, and a counter (compute_count) waits for 3 cycles before transitioning to OUTPUT.
- The sum of the four partial products is assigned to dout, and ld is pulsed again for the next element's multiplications, maintaining one output per cycle after the initial latency.
- All processes need 3 (initial COMPUTE)+164 (OUTPUT + COMPUTE per element) = 67clock cycles $\Longrightarrow$ 134ns.
- The process takes place from 67 ns -> 201 ns with the IDLE state at the end of the process.

6 Conclusion

- The Booth design uses fewer LUTs and adds flip-flops, making it more resource-efficient for FPGAs with limited combinational resources but requiring more sequential elements. Moreover, the Booth design may consume less power in dynamic scenarios, while the Baugh-Wooley design might be more power-efficient in static conditions.
- In terms of timing constraints, Booth design can operate at a higher clock frequency, but the overall execution time, as observed in the waveform, is longer due to latency. The Baugh-Wooley design is better for applications where latency is critical and clock frequency can be adjusted.
- In Booth Multiplier, the sequential nature makes it scalable to larger bit widths with manageable resource growth, as the number of cycles increases linearly with N/4, while the Baugh-Wooley technique, with its combinational nature, leads to an exponential growth with bit width. This makes the Baugh-Wooley technique less suitable for large matrices due to the delay and area constraints.

7 Chapter 7: References

References

[1] *Book.*
https://github.com/MorrisMA/Booth_Multipliers/blob/master/Src/Booth_Multiplier_4xA.ucf

[2] *Book: VLSI Design 2011 by Gerald E. Sobelman.*
http://dce.hust.edu.vn/wpcontent/uploads/2017/05/MultiplierBaughWooley.pdf