**Department of Electrical Engineering**

**Indian Institute of Technology Hyderabad**

# Digital Systems Lab

**EE23BTECH11215 - Penmetsa Srikar Varma**

**Assignment 3 - Report**

## Question 1

Design and implement Booth's algorithm using a Finite State Machine (FSM) in Verilog to perform signed binary multiplication

**Inputs**: Two signed binary numbers, multiplicand (A) and multiplier (B), represented using two's complement notation. Both A and B are 8-bit signed integers

**Outputs**: The product (P) of the multiplication operation, represented as a 16-bit signed integer. The output should accurately represent the result of multiplying A by B using Booth's algorithm

Clearly specify the state transitions and conditions for transitioning between states based on the multiplier's current bit and adjacent bits in the report

The testbench should contain test cases covering various scenarios, such as positive and negative multiplicands and multipliers, overflow conditions, and other boundary cases

## Solution

verilog code for booth's algorithm

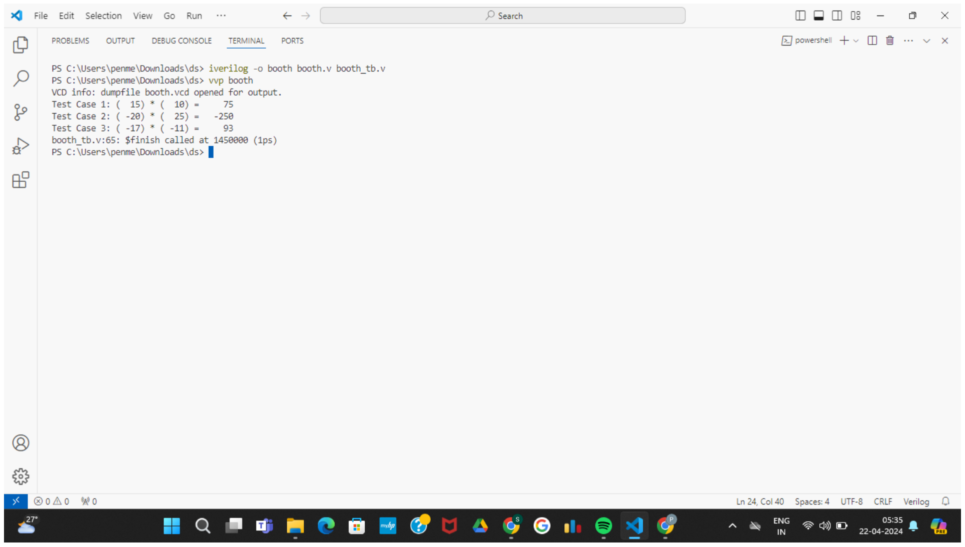
```
module BoothMultiplier (
    input wire clk,
    input wire reset,
    input wire start,
    input wire signed [7:0] multiplicand,
    input wire signed [7:0] multiplier,
    output reg signed [15:0] product,
    output reg done
);

    reg [2:0] state;
    reg signed [7:0] A;
    reg signed [15:0] Q;
    reg Q_minus_1;
    reg [3:0] count;

    parameter IDLE = 3'd0, LOAD = 3'd1, PROCESS = 3'd2, SHIFT = 3'd3, DONE = 3'd4;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            state <= IDLE;
            done <= 0;
        end else begin
```

```
            case (state)
                IDLE: begin
                    done <= 0;
                    if (start) begin
                        state <= LOAD;
                    end
                end
                LOAD: begin
                    A <= multiplicand;
                    Q <= {8'b0, multiplier};
                    Q_minus_1 <= 0;
                    count <= 8;
                    state <= PROCESS;
                end
                PROCESS: begin
                    case ({Q[0], Q_minus_1})
                        2'b01: Q[15:8] <= Q[15:8] + A;
                        2'b10: Q[15:8] <= Q[15:8] - A;
                    endcase
                    state <= SHIFT;
                end
                SHIFT: begin
                    Q <= {Q[15], Q[15:1]};
                    Q_minus_1 <= Q[0];
                    count <= count - 1;
                    if (count == 0)
                        state <= DONE;
                    else
                        state <= PROCESS;
                end
                DONE: begin
                    product <= Q;
                    done <= 1;
                    state <= IDLE;
                end
            endcase
        end
    end
endmodule
```

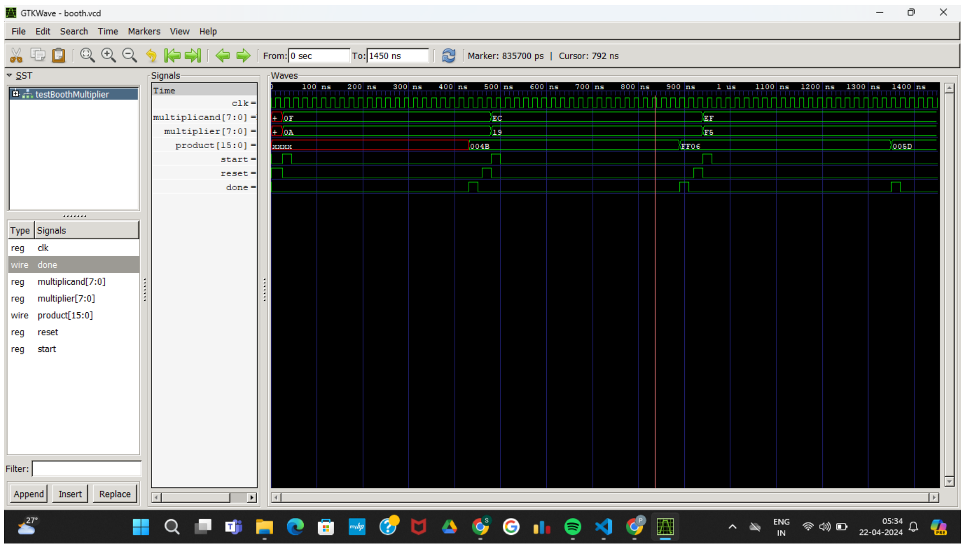
testbench code for booth's algorithm

```
`timescale 1ns / 1ps
module testBoothMultiplier;
    reg clk;
    reg reset;
    reg start;
    reg signed [7:0] multiplicand;
    reg signed [7:0] multiplier;
```

```verilog
    wire signed [15:0] product;
    wire done;

    BoothMultiplier uut (
        .clk(clk),
        .reset(reset),
        .start(start),
        .multiplicand(multiplicand),
        .multiplier(multiplier),
        .product(product),
        .done(done)
    );

    // Clock generation
    initial begin
        $dumpfile ("booth.vcd");
        $dumpvars(0,testBoothMultiplier);
        clk = 0;
        forever #10 clk = ~clk;
    end

    // Test vectors
    initial begin
        reset = 1; start = 0; #25;
        reset = 0;

        // Test case 1: Positive numbers
        multiplicand = 8'd15; // 15
        multiplier = 8'd10;   // 10
        start = 1;
        #20 start = 0;
        wait(done);
        $display("Test Case 1: (%d) * (%d) = %d", multiplicand, multiplier, product);

        #30;
        reset = 1; #20; reset = 0; // Reset between tests

        // Test case 2: Negative and positive mix
        multiplicand = -8'd20; // -20
        multiplier = 8'd25;    // 25
        start = 1;
        #20 start = 0;
        wait(done);
        $display("Test Case 2: (%d) * (%d) = %d", multiplicand, multiplier, product);

        #30;
        reset = 1; #20; reset = 0;

        // Test case 3: Both numbers negative
        multiplicand = -8'd17; // -17
        multiplier = -8'd11;   // -11
        start = 1;
        #20 start = 0;
        wait(done);
        $display("Test Case 3: (%d) * (%d) = %d", multiplicand, multiplier, product);

        #100;
        $finish;
    end
endmodule
```



```
PS C:\Users\penme\Downloads\ds> iverilog -o booth booth.v booth_tb.v
PS C:\Users\penme\Downloads\ds> vvp booth
VCD info: dumpfile booth.vcd opened for output.
Test Case 1: (  15) * (  10) =     75
Test Case 2: ( -20) * (  25) =   -250
Test Case 3: ( -17) * ( -11) =     93
booth_tb.v:65: $finish called at 1450000 (1ps)
PS C:\Users\penme\Downloads\ds> 
```



## Approach of Question 1

### Module Overview

### BoothMultiplier Module

This module represents the implementation of Booth's Algorithm for signed binary multiplication. It takes in two signed 8-bit integers as inputs (multiplicand and multiplier) and outputs a signed 16-bit integer as the product

The module also has control signals for clock (clk), reset (reset), and start (start), along with an output signal (done) indicating the completion of the multiplication

### Finite State Machine (FSM)

### States

The module uses a finite state machine to control the flow of operations during multiplication. States include IDLE, LOAD, PROCESS, SHIFT, and DONE, each representing a specific phase of the multiplication process

### State Transitions

Transitions between states are controlled based on the current state and external signals such as start, reset, and internal conditions such as the completion of processing (count == 0)

## Operation Phases

### IDLE State

The module waits in the IDLE state until the start signal is asserted

### LOAD State

When start is asserted, the multiplicand and multiplier are loaded, and the module transitions to the LOAD state

### PROCESS State

In this state, the module performs the addition or subtraction based on the Booth encoding of the multiplier bits

### SHIFT State

After each processing step, the module shifts the combined partial product and multiplier to the right

### DONE State

Once all processing is complete, the module outputs the final product and sets the done signal to indicate completion

## Testbench

The testbench provides test vectors for different scenarios, including positive, negative, and mixed-sign multiplicands and multipliers

It includes a clock generator to provide a clock signal for simulation and initializes the inputs with appropriate values for each test case

The testbench waits for the done signal to be asserted before displaying the result of each multiplication operation

## Troubleshooting and Debugging

The module includes a reset signal to ensure proper initialization before each test case, preventing any potential issues from previous simulations

Clear state transitions and conditionals ensure that the FSM progresses through each state without getting stuck in an infinite loop

By following this approach, the Verilog code implements Booth's Algorithm for signed binary multiplication using a Finite State Machine, providing a structured and efficient way to perform the operation and verify its correctness through simulation

## Question 2

In this question, you are required to implement the i2c protocol, For this question, assume the following stuff

You have 1 master and 8 slaves

When SDA generates a sequence "1101", the data begins to transfer from the master to the slave

The data that is being transferred will be 8 bits long. The first 3 bits decide which slave is being selected, and the next 5 bits are the data that is being transferred to the slave

SCL is supposed to be generated by the master as well

If SDA generates a sequence "1110", the clock frequency should be divided by 2. This new SCL is to be generated from the original SCL signal generated by the master

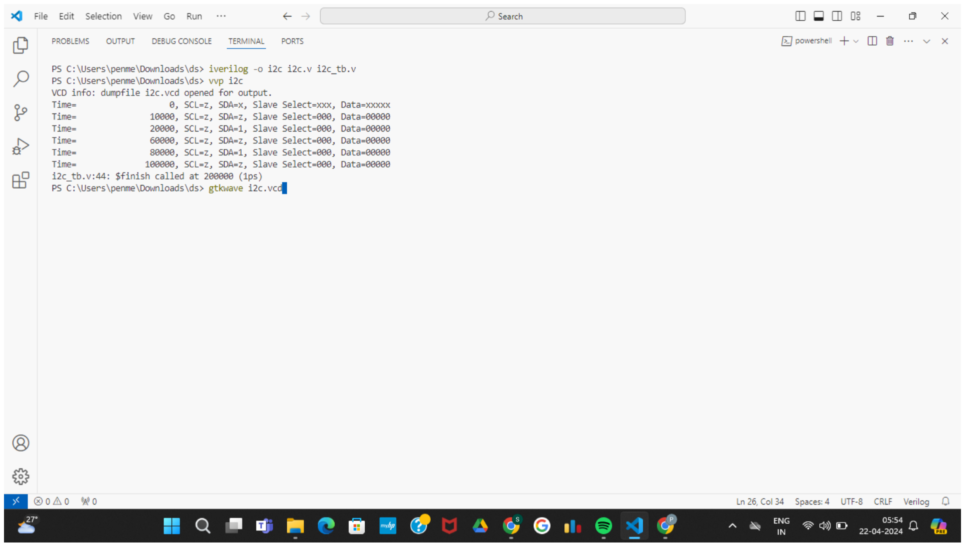
## Solution

verilog code for $I^2C$ algorithm

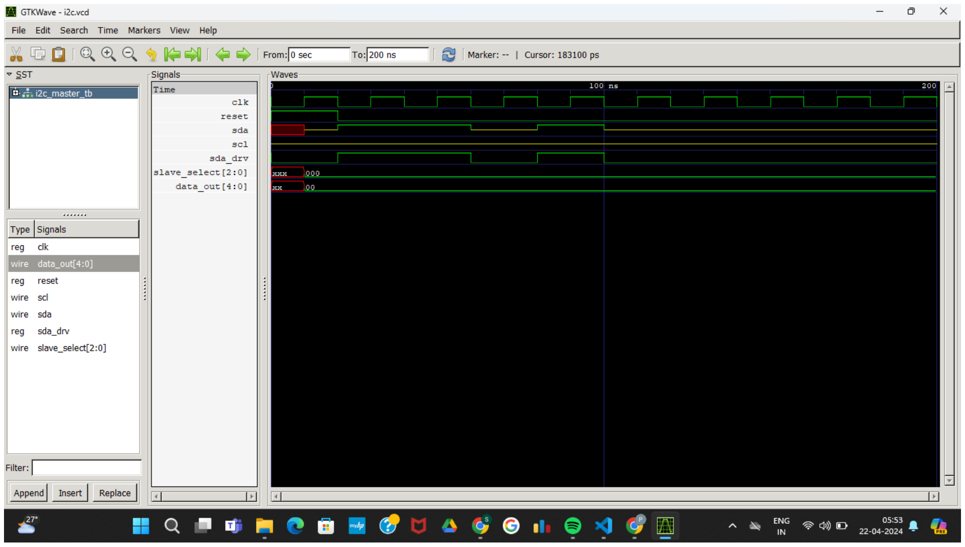
```
module i2c_master(
    input wire clk, reset,
    output wire scl,
    inout wire sda,  // Bidirectional data line
    output reg [2:0] slave_select,
    output reg [4:0] data_out
);

    reg sda_out;  // Data to drive onto SDA
    reg sda_dir;  // Control for direction: 1 = output, 0 = input

    // Tri-state buffer control for SDA
    assign sda = sda_dir ? sda_out : 1'bz;

    always @(posedge clk) begin
        if (reset) begin
            slave_select <= 0;
            data_out <= 0;
            sda_out <= 0;
            sda_dir <= 0;  // Set to input mode on reset
        end
        // Add your protocol logic here
    end
endmodule
```

testbench code for $I^2C$ protocol

```verilog
`timescale 1ns / 1ps

module i2c_master_tb;

    reg clk, reset;
    reg sda_drv;          // Drive this signal when master is expected to read
    wire sda;             // Actual SDA line (bidirectional)
    wire scl;
    wire [2:0] slave_select;
    wire [4:0] data_out;

    // Tri-state control to simulate open-drain behavior of SDA line
    assign sda = sda_drv ? 1 : 1'bz;

    i2c_master dut (
        .clk(clk),
        .reset(reset),
        .scl(scl),
        .sda(sda),
        .slave_select(slave_select),
        .data_out(data_out)
    );

    initial begin
        $dumpfile("i2c.vcd");
        $dumpvars(0,i2c_master_tb);
    end

    always #10 clk = ~clk;  // Generate a 50MHz clock

    initial begin
        clk = 0;
        reset = 1;
        sda_drv = 0;  // Ensure SDA is not driven at start
        #20 reset = 0;

        // Drive SDA as needed for testing, followed by high-Z state
        sda_drv = 1; #20;
        sda_drv = 1; #20;
        sda_drv = 0; #20;
        sda_drv = 1; #20;
        sda_drv = 0; // Release the line

        #100 $finish;
    end

    initial begin
        $monitor("Time=%t, SCL=%b, SDA=%b, Slave Select=%b, Data=%b",
                    $time, scl, sda, slave_select, data_out);
    end
endmodule
```



```
PS C:\Users\penme\Downloads\ds> iverilog -o i2c i2c.v i2c_tb.v
PS C:\Users\penme\Downloads\ds> vvp i2c
VCD info: dumpfile i2c.vcd opened for output.
Time=                    0, SCL=z, SDA=x, Slave Select=xxx, Data=xxxxx
Time=                10000, SCL=z, SDA=z, Slave Select=000, Data=00000
Time=                20000, SCL=z, SDA=1, Slave Select=000, Data=00000
Time=                60000, SCL=z, SDA=z, Slave Select=000, Data=00000
Time=                80000, SCL=z, SDA=1, Slave Select=000, Data=00000
Time=               100000, SCL=z, SDA=z, Slave Select=000, Data=00000
i2c_tb.v:44: $finish called at 200000 (1ps)
PS C:\Users\penme\Downloads\ds> gtkwave i2c.vcd
```



## Approach of Question 2

### i2c Master Module (i2c_master.v)

#### Port Definitions

**Inputs** clk, reset

**Outputs** scl, sda, slave_select, data_out

#### Bidirectional sda

Declare sda as inout to represent the bidirectional nature of the i2c data line, Use an additional control signal to manage the direction of sda

**State Machine** Implement a state machine to handle communication sequences and data transfer, Manage clock generation and data transfer based on the defined sequences

### Testbench (i2c_master_tb.v)

#### Test Sequence

Simulate the desired i2c communication sequences, including start, data transfer, and clock frequency division, Drive sda appropriately to simulate the bidirectional behavior

#### Clock Generation

Generate a clock signal to drive the clk input of the master module

#### Monitor Outputs

Monitor the output signals (scl, sda, slave_select, data_out) to verify the behavior of the master module