EE2016 Experiment 3

Group 3 - EE23B039, EE23B033, EE23B027

1 Wallace Multiplier

1.1 Verilog code

```
//circuit_half_adder

module half_adder(I1, I2, S, C);
        input I1, I2;
        output S, C;

        xor(S, I1, I2);
        and(C, I1, I2);

endmodule

//circuit full_adder

module full_adder(a, b, cin, s, cout);

        input a;
        input b;
        input cin;
        output cout;
        output s;
        wire s1, c1, c2;

        xor(s1, a, b);
        and(c1, s1, cin);
        and(c2, a, b);
        xor(s, cin, s1);
        or(cout, c1, c2);

endmodule

//circuit 4-bit ripple adder

module FFA(A, B, sum, cout);

  input [3:0] A, B;
  output reg [3:0] sum;
  output reg cout;
  reg cin;
  integer i;

  always @* begin
    cin = 0;
    for (i = 0; i < 4; i = i + 1) begin
      {cout, sum[i]} = A[i] + B[i] + cin;
      cin = cout;
    end
  end
endmodule

//circuit for 4 bit wallace multiplier
```

```
module unsigned_mult( m, a, b );

        input [3:0]a,b;
        output [7:0]m ;
        wire [3:0]p0, p1, p2, p3,k3, k4, s5;
        wire k1, l1, k2, l2, s0, c0, s1, c1, s2, c2, s3, c3, s4, c4, c5, s6, c6;
        assign p0 = a & {4{b[0]}};
        assign p1 = a & {4{b[1]}};
        assign p2 = a & {4{b[2]}};
        assign p3 = a & {4{b[3]}};

        half_adder ha1(p2[1], p3[0], k1, l1);
        half_adder ha2(p2[2], p3[1], k2, l2);
        half_adder ha3(p0[1], p1[0], s0, c0);
        full_adder fa1(p0[2], p1[1], p2[0], s1, c1);
        full_adder fa2(p0[3], p1[2], k1, s2, c2);
        full_adder fa3(p1[3], k2, l1, s3, c3);
        full_adder fa4(p2[3], p3[2], l2, s4, c4);

        assign k3[0] =s1;
        assign k3[1] =s2;
        assign k3[2] =s3;
        assign k3[3] =s4;

        assign k4[0] =c0;
        assign k4[1] =c1;
        assign k4[2] =c2;
        assign k4[3] =c3;

        FFA ffa1(k3, k4, s5, c5);
        full_adder fa5(p3[3], c4, c5, s6, c6);


        assign m[0] = p0[0];
        assign m[1] = s0;
        assign m[2] = s5[0];
        assign m[3] = s5[1];
        assign m[4] = s5[2];
        assign m[5] = s5[3];
        assign m[6] = s6;
        assign m[7] = c6;


endmodule

module top_module(
    input clk,                  // Clock input
    output [7:0] led            // 8 LEDs
);

    reg [3:0] a;
    reg [3:0] b;
    wire [7:0] m;

    unsigned_mult multiplier (
        .a(a),
        .b(b),
        .m(m)
    );

    // Hardcoding input values
    initial begin
        a = 4'b1010; // 10 in decimal
        b = 4'b1101; // 13 in decimal
```

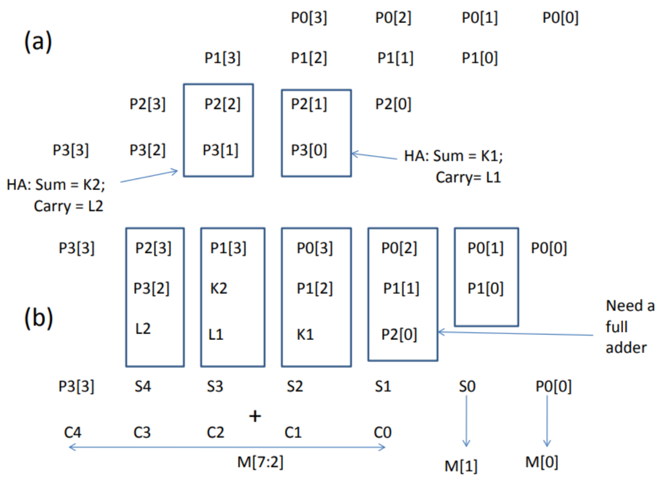

Figure 1: The Wallace Multiplier

Figure 1: Wallace Multiplier

```
    end

    assign led = m;

endmodule
```

Code for wallace multiplier was written basing the above architecture which was given in the hand-out. Same number of half adders and full adders were used where ever mentioned and at the last step where P3[3] S4 S3 S2 S1 had to be added with C4 C3 C2 C1 C0, a 4 bit ripple carry adder was used to first add S4 S3 S2 S1 with C3 C2 C1 C0 and their sum is assigned to M[5] M[4] M[3] M[2] and carry is taken forward to be added with P3[3] and C4 using a full adder again. The sum generated here is M[6] and the carry is M[7].

1.2 Testbench for simulation

```
`timescale 1ns/1ps

module test_wallace;
    reg [3:0] a, b;
    wire [7:0] m;

    // Instantiate the Wallace multiplier module
    unsigned_mult DUT (
        .m(m),
        .a(a),
        .b(b)
    );

    initial begin
        // Initializing values
        a = 4'b0000; b = 4'b0000;
        #10;
        $display("a = %b, b = %b, m = %b", a, b, m);

        a = 4'b1010; b = 4'b1101;
        #10;
        $display("a = %b, b = %b, m = %b", a, b, m);

        a = 4'b1111; b = 4'b0000;
        #10;
        $display("a = %b, b = %b, m = %b", a, b, m);

        a = 4'b0011; b = 4'b1010;
        #10;
        $display("a = %b, b = %b, m = %b", a, b, m);

        a = 4'b0101; b = 4'b1100;
        #10;
        $display("a = %b, b = %b, m = %b", a, b, m);

        a = 4'b1111; b = 4'b1111;
        #10;
        $display("a = %b, b = %b, m = %b", a, b, m);

        // End of simulation
        $finish;
    end

endmodule
```

1.3 Constraints file

```
# Clock signal
set_property -dict { PACKAGE_PIN N11    IOSTANDARD LVCMOS33 } [get_ports { clk }];

# LEDs
set_property -dict { PACKAGE_PIN J3    IOSTANDARD LVCMOS33 } [get_ports { led[0] }]; # LSB
set_property -dict { PACKAGE_PIN H3    IOSTANDARD LVCMOS33 } [get_ports { led[1] }];
set_property -dict { PACKAGE_PIN J1    IOSTANDARD LVCMOS33 } [get_ports { led[2] }];
set_property -dict { PACKAGE_PIN K1    IOSTANDARD LVCMOS33 } [get_ports { led[3] }];
set_property -dict { PACKAGE_PIN L3    IOSTANDARD LVCMOS33 } [get_ports { led[4] }];
set_property -dict { PACKAGE_PIN L2    IOSTANDARD LVCMOS33 } [get_ports { led[5] }];
set_property -dict { PACKAGE_PIN K3    IOSTANDARD LVCMOS33 } [get_ports { led[6] }];
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { led[7] }];
```

2 LCD Display

2.1 Verilog Code

```
//circuit for lcd_display

module lcd(in_Clk, lcd_rs, lcd_e, data);
        input in_Clk;
        output reg [7:0] data;
        output reg lcd_rs;
        output lcd_e;
        wire [7:0] command [0:4];
        reg [31:0] count=0;
        wire out_Clk;


        assign command [0] = 8'h38; // control signal to display on two lines
        assign command [1] = 8'h0C;
        // keep display on but cursor off
        assign command [2] = 8'h06; // increment the cursor
        assign command [3] = 8'h01; // clear the display
        assign command [4] = 8'hC0; // choose the second line
        clk_divider c0 (in_Clk, 1 ,out_Clk);
        assign lcd_e = out_Clk;
        always@(posedge lcd_e) begin
                count <= count+1
        case(count)
                1: begin lcd_rs = 0; data = command[0]; end // fill in suitably
                2: begin lcd_rs = 0; data = command[1]; end
                3: begin lcd_rs = 0; data = command[2]; end
                4: begin lcd_rs = 0; data = command[3]; end
                5: begin lcd_rs = 0; data = command[4]; end
                6: begin lcd_rs = 1; data = 8'h31; end // fill in hex corresp to ASCII for 1
                7: begin lcd_rs = 1; data = 8'h32; end // 2
                8: begin lcd_rs = 1; data = 8'h33; end // 3
                9: begin lcd_rs = 1; data = 8'h34; end // 4
                10: begin lcd_rs = 1; data = 8'h35; end
                11: begin lcd_rs = 1; data = 8'h36; end
                12: begin lcd_rs = 1; data = 8'h37; end
                13: begin lcd_rs = 1; data = 8'h38; end
                14: begin lcd_rs = 1; data = 8'h39; end
                15: begin lcd_rs = 1; data = 8'h41; end
                16: begin lcd_rs = 1; data = 8'h42; end
                17: begin lcd_rs = 1; data = 8'h43; end
                18: begin lcd_rs = 1; data = 8'h44; end
                19: begin lcd_rs = 1; data = 8'h45; end // E
                20: begin lcd_rs = 1; data = 8'h46; end // F
                21: begin lcd_rs = 1; data = 8'h47; end // hex corresp to ASCII for G
                default: begin lcd_rs = 0; data = 8'h00; end
        // fill in hex value to return cursor to initial position
        endcase
        end
endmodule


module clk_divider(inClk,reset,outClk);
        input inClk;
        input reset;
        output outClk;
        reg outClk;
        reg[32:0] clockCount;
        always @( posedge inClk)
        begin
        if (reset == 1'b0)
        begin
        clockCount <= 33'b0;
        outClk <= 1'b0;
        end

        else
        begin
        if (clockCount == 33'd25000000)
        begin
        clockCount <= 33'b0;
        outClk <= ~outClk;
        end
        else
        begin
        clockCount <= clockCount + 33'd1;
        end
        end

        end
endmodule
```

2.2 Constraints File

```
# Clock signal
set_property -dict { PACKAGE_PIN N11    IOSTANDARD LVCMOS33 } [get_ports { in_Clk }];

# 2x16 LCD
set_property -dict { PACKAGE_PIN P3 IOSTANDARD LVCMOS33 } [get_ports {data[7]}];
set_property -dict { PACKAGE_PIN M5 IOSTANDARD LVCMOS33 } [get_ports {data[6]}];
set_property -dict { PACKAGE_PIN N4 IOSTANDARD LVCMOS33 } [get_ports {data[5]}];
set_property -dict { PACKAGE_PIN R2 IOSTANDARD LVCMOS33 } [get_ports {data[4]}];
set_property -dict { PACKAGE_PIN R1 IOSTANDARD LVCMOS33 } [get_ports {data[3]}];
set_property -dict { PACKAGE_PIN R3 IOSTANDARD LVCMOS33 } [get_ports {data[2]}];
set_property -dict { PACKAGE_PIN T2 IOSTANDARD LVCMOS33 } [get_ports {data[1]}];
set_property -dict { PACKAGE_PIN T4 IOSTANDARD LVCMOS33 } [get_ports {data[0]}];
set_property -dict { PACKAGE_PIN T3 IOSTANDARD LVCMOS33 } [get_ports {lcd_e}];
set_property -dict { PACKAGE_PIN P5 IOSTANDARD LVCMOS33 } [get_ports {lcd_rs}];
```

3 Wallace Multiplier on LCD Display

```
module half_adder(I1, I2, S, C);
input I1, I2;
output S, C;

xor(S, I1, I2);
and(C, I1, I2);

endmodule

//circuit full_adder

module full_adder(a, b, cin, s, cout);

input a;
input b;
input cin;
output cout;
output s;
wire s1, c1, c2;

xor(s1, a, b);
and(c1, s1, cin);
and(c2, a, b);
xor(s, cin, s1);
or(cout, c1, c2);

endmodule

//circuit 4-bit ripple adder

module FFA(A, B, sum, cout);

  input [3:0] A, B;
  output reg [3:0] sum;
  output reg cout;
  reg cin;
  integer i;

  always @* begin
    cin = 0;
    for (i = 0; i < 4; i = i + 1) begin
      {cout, sum[i]} = A[i] + B[i] + cin;
      cin = cout;
    end
  end
endmodule

//circuit for 4 bit wallace multiplier

module unsigned_mult( m, a, b );

input [3:0]a,b;
output [7:0]m ;
wire [3:0]p0, p1, p2, p3,k3, k4, s5;
wire k1, l1, k2, l2, s0, c0, s1, c1, s2, c2, s3, c3, s4, c4, c5, s6, c6;
assign p0 = a & {4{b[0]}};
    assign p1 = a & {4{b[1]}};
    assign p2 = a & {4{b[2]}};
    assign p3 = a & {4{b[3]}};

half_adder ha1(p2[1], p3[0], k1, l1);
half_adder ha2(p2[2], p3[1], k2, l2);
half_adder ha3(p0[1], p1[0], s0, c0);
full_adder fa1(p0[2], p1[1], p2[0], s1, c1);
full_adder fa2(p0[3], p1[2], k1, s2, c2);
full_adder fa3(p1[3], k2, l1, s3, c3);
full_adder fa4(p2[3], p3[2], l2, s4, c4);

assign k3[0] =s1;
assign k3[1] =s2;
assign k3[2] =s3;
assign k3[3] =s4;

assign k4[0] =c0;
assign k4[1] =c1;
assign k4[2] =c2;
assign k4[3] =c3;

FFA ffa1(k3, k4, s5, c5);
full_adder fa5(p3[3], c4, c5, s6, c6);


assign m[0] = p0[0];
assign m[1] = s0;
assign m[2] = s5[0];
assign m[3] = s5[1];
assign m[4] = s5[2];
assign m[5] = s5[3];
assign m[6] = s6;
assign m[7] = c6;


endmodule


// Binary to BCD conversion module
module bin_to_bcd(
    input [7:0] binary,
    output reg [3:0] hundreds,
    output reg [3:0] tens,
    output reg [3:0] ones
);
    integer i;
    always @(binary) begin
        // Initialize BCD digits
        hundreds = 4'd0;
        tens = 4'd0;
        ones = 4'd0;

        // Binary to BCD conversion algorithm
        for (i = 7; i >= 0; i = i - 1) begin
            if (hundreds >= 5)
                hundreds = hundreds + 4'd3;
            if (tens >= 5)
                tens = tens + 4'd3;
            if (ones >= 5)
                ones = ones + 4'd3;
            hundreds = hundreds << 1;
            hundreds[0] = tens[3];
            tens = tens << 1;
            tens[0] = ones[3];
            ones = ones << 1;
            ones[0] = binary[i];
        end
    end
endmodule


module lcd_display(clk, rst, lcd_rs, lcd_e, lcd_data, a, b, product);
    input clk, rst;
    input [3:0] a, b;               // 4-bit inputs
    input [7:0] product;
    output reg lcd_rs;
    output  lcd_e;
    output reg [7:0] lcd_data;
    wire [3:0] hundreds, tens, ones;

    reg [31:0] count = 0;
    wire [7:0] command [0:5];
    wire out_Clk;
    bin_to_bcd(product, hundreds, tens, ones);

    assign command[0] = 8'h38; // Two-line mode
    assign command[1] = 8'h0C; // Display on, cursor off
    assign command[2] = 8'h06; // Increment cursor
    assign command[3] = 8'h01; // Clear display
    assign command[4] = 8'h80; // First line
    assign command[5] = 8'hC0; // Second line

    // Clock divider for generating LCD enable signal
    clk_divider c0(clk, 1, out_Clk);
    assign lcd_e = out_Clk;
    if (rst)begin
    always @(posedge lcd_e)begin

            count <= count + 1;

        case(count)
            1: begin lcd_rs = 0; lcd_data = command[0]; end
            2: begin lcd_rs = 0; lcd_data = command[1]; end
            3: begin lcd_rs = 0; lcd_data = command[2]; end
            4: begin lcd_rs = 0; lcd_data = command[3]; end
            5: begin lcd_rs = 0; lcd_data = command[4]; end
            6: begin lcd_rs = 1; lcd_data = 8'h50; end
            7: begin lcd_rs = 1; lcd_data = 8'h72; end
            8: begin lcd_rs = 1; lcd_data = 8'h6F; end
            9: begin lcd_rs = 1; lcd_data = 8'h64; end
            10: begin lcd_rs = 1; lcd_data = 8'h75; end
            11: begin lcd_rs = 1; lcd_data = 8'h63; end
            12: begin lcd_rs = 1; lcd_data = 8'h74; end
            13: begin lcd_rs = 1; lcd_data = 8'h20; end
            14: begin lcd_rs = 1; lcd_data = 8'h69; end
            15: begin lcd_rs = 1; lcd_data = 8'h73; end
            16: begin lcd_rs = 0; lcd_data = command[5]; end
            17: begin lcd_rs = 1; lcd_data = 8'h30 + hundreds; end
            18: begin lcd_rs = 1; lcd_data = 8'h30 + tens; end
            19: begin lcd_rs = 1; lcd_data = 8'h30 + ones; end
            default: begin lcd_rs = 0; lcd_data = command[5]; count = 0; end
        endcase
    end
    end
    else
         begin lcd_rs = 0; lcd_data = command[3]; end

endmodule

// Clock Divider Module
module clk_divider(inClk, reset, outClk);
    input inClk;
    input reset;
    output reg outClk;
    reg [32:0] clockCount;

    always @(posedge inClk) begin
        if (!reset) begin
            clockCount <= 33'b0;
            outClk <= 1'b0;
        end else if (clockCount == 33'd25000000) begin
            clockCount <= 33'b0;
            outClk <= ~outClk;
        end else begin
            clockCount <= clockCount + 33'd1;
        end
    end
endmodule

// Top Module
module top_module(
    input clk,
    input [3:0] a, b, // 4-bit switches for inputs
    output lcd_rs, lcd_e,
    output [7:0] lcd_data
);
    wire [7:0] product;


    // Instantiate the Wallace multiplier
     unsigned_mult multiplier (
        .a(a),
        .b(b),
        .m(product)
    );

    // Instantiate the LCD display module
    lcd_display display (
        .clk(clk),
        .rst(rst),
        .lcd_rs(lcd_rs),
        .lcd_e(lcd_e),
        .lcd_data(lcd_data),
        .a(a),
        .b(b),
        .product(product)
    );
endmodule
```

3.1 Constraints File

```
# Clock signal
set_property -dict { PACKAGE_PIN N11    IOSTANDARD LVCMOS33 } [get_ports { clk }];

# 2x16 LCD
set_property -dict { PACKAGE_PIN P3 IOSTANDARD LVCMOS33 } [get_ports {lcd_data[7]}];
set_property -dict { PACKAGE_PIN M5 IOSTANDARD LVCMOS33 } [get_ports {lcd_data[6]}];
set_property -dict { PACKAGE_PIN N4 IOSTANDARD LVCMOS33 } [get_ports {lcd_data[5]}];
set_property -dict { PACKAGE_PIN R2 IOSTANDARD LVCMOS33 } [get_ports {lcd_data[4]}];
set_property -dict { PACKAGE_PIN R1 IOSTANDARD LVCMOS33 } [get_ports {lcd_data[3]}];
set_property -dict { PACKAGE_PIN R3 IOSTANDARD LVCMOS33 } [get_ports {lcd_data[2]}];
set_property -dict { PACKAGE_PIN T2 IOSTANDARD LVCMOS33 } [get_ports {lcd_data[1]}];
set_property -dict { PACKAGE_PIN T4 IOSTANDARD LVCMOS33 } [get_ports {lcd_data[0]}];
set_property -dict { PACKAGE_PIN T3 IOSTANDARD LVCMOS33 } [get_ports {lcd_e}];
set_property -dict { PACKAGE_PIN P5 IOSTANDARD LVCMOS33 } [get_ports {lcd_rs}];

# Switches
set_property -dict { PACKAGE_PIN L5    IOSTANDARD LVCMOS33 } [get_ports { a[0] }];#LSB
set_property -dict { PACKAGE_PIN L4    IOSTANDARD LVCMOS33 } [get_ports { a[1] }];
set_property -dict { PACKAGE_PIN M4    IOSTANDARD LVCMOS33 } [get_ports { a[2] }];
set_property -dict { PACKAGE_PIN M2    IOSTANDARD LVCMOS33 } [get_ports { a[3] }];
set_property -dict { PACKAGE_PIN M1    IOSTANDARD LVCMOS33 } [get_ports { b[0] }];
set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports { b[1] }];
set_property -dict { PACKAGE_PIN N2    IOSTANDARD LVCMOS33 } [get_ports { b[2] }];
set_property -dict { PACKAGE_PIN N1    IOSTANDARD LVCMOS33 } [get_ports { b[3] }];
set_property -dict { PACKAGE_PIN M6    IOSTANDARD LVCMOS33 } [get_ports { rst }];#MSB
```