EE2016 - Experiment 3

Nishanth Senthil Kumar(EE23B049)
Arjun Krishnaswamy (EE23B009)
Gopalkrishna Ramanujam (EE23B029)

14 August 2024

1 Aim of Experiment

- Designing a 4 bit unsigned number multiplier (Wallace Multiplier), simulating it, and showing the result in the FPGA board.
- Displaying a set of characters on the LCD screen, and printing the result of the Wallace multiplier on the LCD display.

2 Wallace Multiplier

2.1 Overview

- A Wallace multiplier is a hardware implementation of a binary multiplier, a digital circuit that multiplies two integers. It uses a selection of full and half adders (the Wallace tree or Wallace reduction) to sum partial products in stages until two numbers are left.
- Our task was to design it for two 4 bit unsigned numbers and compute the output. The maximum output would be 225, so we need the output to be able to handle upto 8 bits.
- To implement this, we have to use 8 Full adders and 4 Half Adders.

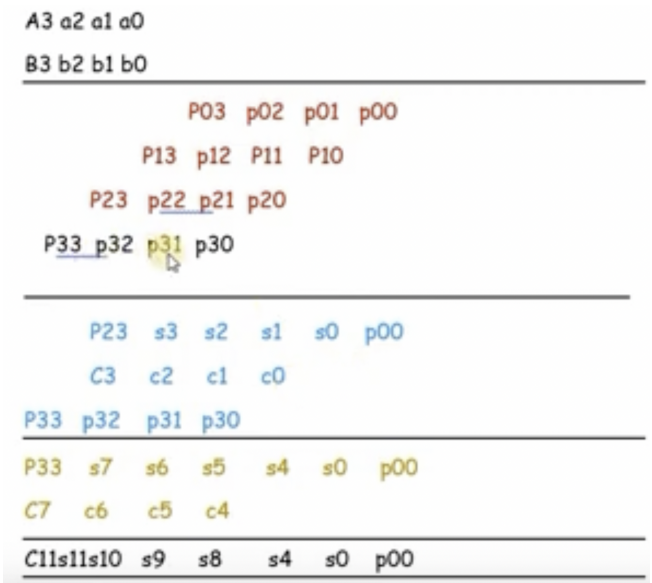

Figure 1: A schematic of the 4 bit multiplier

2.2 Implementation of the Wallace Multiplier on Verilog

```
module fulladder(
    input a,b,c,
    output sum,carry

);
    assign sum=(a^b)^(c);
    assign carry=(a&b)| c&(a^b);

endmodule

module half_adder(
    input a,
    input b,
    output sum,
    output carry
);

    assign sum = a ^ b;
    assign carry = a & b;

endmodule


module unsigned_mult( M, a, b );

//two 4 bit inputs and one 8 bit output

input [3:0]a,b;
output [7:0]M;

//a register to store all the product terms

reg p[3:0][3:0];

wire s5,s4,s3,s2,s1,s0,s6,s7,s8,s9,s10,s11;
wire c5,c4,c3,c2,c1,c0,c6,c7,c8,c9,c10,c11;

//integers used in the for loops
integer i,j;

//generating all the product terms and storing it in p for easy access
always @(a or b)
    begin
        for(i=0;i<=3;i++)
        begin
            for(j=0;j<=3;j++)
            begin
                p[i][j]=a[j]&b[i];
            end
        end
    end


    //calling the full adder and half adder modules, and assigning the product values
```

```
    assign M[0]=p[0][0];

    half_adder h1(.a(p[0][1]),.b(p[1][0]),.sum(s0),.carry(c0));

    assign M[1]=s0;

    fulladder f1(.a(p[0][2]),.b(p[1][1]),.c(p[2][0]),.sum(s1),.carry(c1));
    fulladder f2(.a(p[0][3]),.b(p[1][2]),.c(p[2][1]),.sum(s2),.carry(c2));
    half_adder h2(.a(p[1][3]),.b(p[2][2]),.sum(s3),.carry(c3));

    half_adder h3(.a(s1),.b(c0),.sum(s4),.carry(c4));
    fulladder f3(.a(s2),.b(c1),.c(p[3][0]),.sum(s5),.carry(c5));
    fulladder f4(.a(s3),.b(c2),.c(p[3][1]),.sum(s6),.carry(c6));
    fulladder f5(.a(p[2][3]),.b(c3),.c(p[3][2]),.sum(s7),.carry(c7));

    assign M[2]=s4;

    half_adder h4(.a(s5),.b(c4),.sum(s8),.carry(c8));
    fulladder f6(.a(s6),.b(c5),.c(c8),.sum(s9),.carry(c9));
    fulladder f7(.a(s7),.b(c6),.c(c9),.sum(s10),.carry(c10));
    fulladder f8(.a(p[3][3]),.b(c7),.c(c10),.sum(s11),.carry(c11));

    assign M[3]=s8;
    assign M[4]=s9;
    assign M[5]=s10;
    assign M[6]=s11;
    assign M[7]=c11;



endmodule
```

- As stated earlier, 8 Full Adders and 4 Half adders are being used to construct the Wallace Multiplier. The full adder and the half adder modules are declared at the top of the code.
- The main module "unsigned_mult" contains 2 4-bit inputs a and b, and one 8-bit output M.
- A 2-D register P has been declared to store all the individual product terms (like p0,p1 and so on) and a for loop is used to generate the product terms and store it in this 2-D register.
- The,as per the Wallace tree sequence, full and half adders have been instantiated and the product bits have been assigned the output.
- Afer writing this code, it was tested locally on our systems before using it on Vivado.

2.3 Test Bench for Wallace Multiplier

```
module test_bench;
    reg [3:0]a,b;
    wire [7:0]M;

    unsigned_mult Dut(.a(a),.b(b),.M(M));
    initial begin
        #0
        a=4'b000;
        b=4'b1111;

        #10
```

```
        a=4'b1000;
        b=4'b1111;

        #10
        a=4'b0010;
        b=4'b1111;

        #10
        a=4'b0100;
        b=4'b1111;

        #10
        a=4'b0100;
        b=4'b1011;

        #10
        a=4'b1111;
        b=4'b1111;

        #10
        a=4'b0001;
        b=4'b0001;


    end
    initial $monitor("%b%b%b%b%b%b%b%b",M[7],M[6],M[5],M[4],M[3],M[2],M[1],M[0]);


endmodule
```

- This test bench was used first to locally test the Wallace multiplier module, and later on, it was used on Vivado to generate the input and output Waveform.
- Monitor function was used to print the values of the output wire onto the terminal.

2.4 Output Waveform

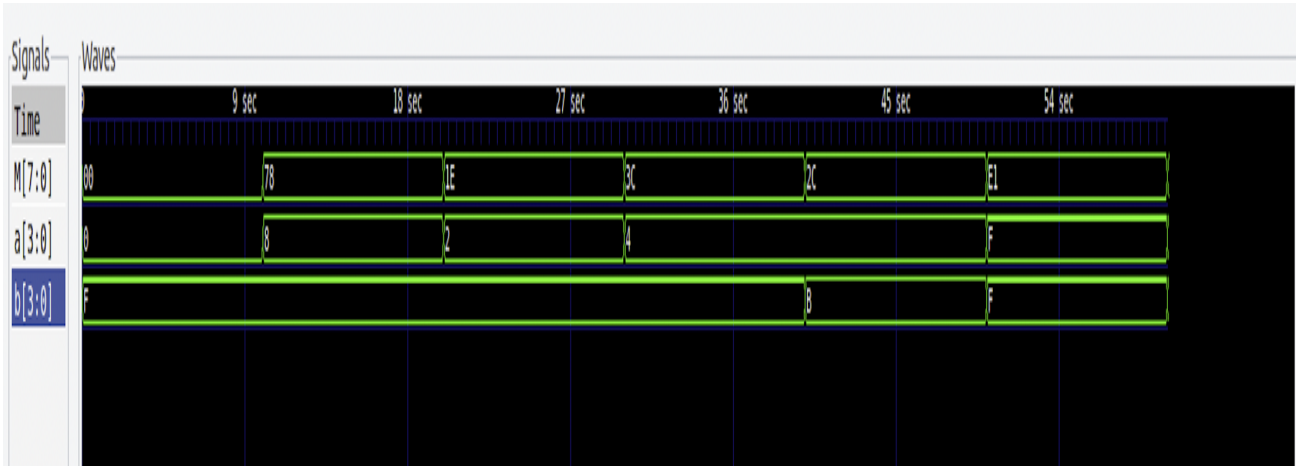

Figure 2: Output Waveform

2.5 Implementing it on the FPGA

- After checking that the module worked using our test bench, We set out to implement in on the FGPA.
- We displayed the 8 bit output on the LEDs and gave two 4-bit inputs a and b through the FPGA switches.

3 LCD module

- An LCD screen is an electronic display module that uses liquid crystal to produce a visible image.
- The 16×2 translates a display of 16 characters per line in 2 such lines. In this LCD, each character is displayed in a 5×7 pixel matrix.
- The FPGA has 16 inbuilt pins to accommodate the LCD module, and it can be directly connected.
- A 16X2 LCD has two registers, namely, command and data. The register select is used to switch from one register to other. RS=0 for the command register, whereas RS=1 for the data register.
- Command Register: The command register stores the command instructions given to the LCD. A command is an instruction given to an LCD to do a predefined task. Examples like:
  - initializing it
  - clearing its screen
  - setting the cursor position
  - controlling display etc.
- Processing for commands happens in the command register.
- Data Register: The data register stores the data to be displayed on the LCD. The data is the ASCII value of the character to be displayed on the LCD. When we send data to LCD, it goes to the data register and is processed there. When RS=1, the data register is selected.

3.1 Writing onto LCD module

- We make use of some LCD RS and LCD E to display characters on the lCD
- Initially, LCD RS is set to zero and we set up our display before we start displaying characters. This includes commands such as clearing display, choosing a line and incrementing cursor
- After our LCD is ready to display characters, we start displaying a string of characters (one every clock cycle) using their ASCII hexadecimal values.
- In this experiment we displayed numbers from 1 to 9 and letters A to F
- This idea of representing characters was used further to represent the product of the Wallace multiplier on the display

3.2 Displaying on the LCD using Verilog

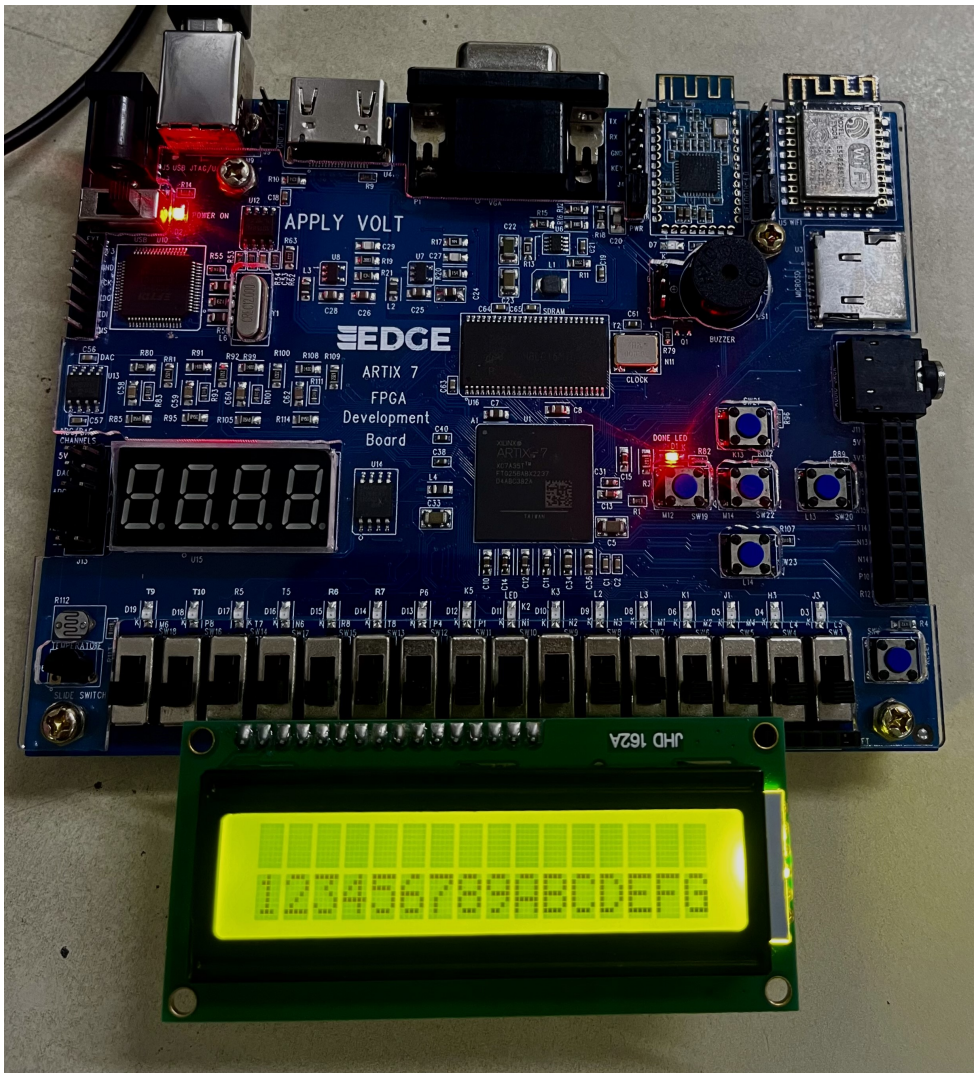

Figure 3: LCD

4 Wallace multiplier on LCD

- The task was to print the two numbers to be multiplied and their product on the LCD
- The LCD module code was modified to print - "Product of XX and YY = ZZ".
- It is to be noted that an additional command to move the cursor to the next line was required to display this output
- The Wallace multiplier module was initialised in the top module (LCD module) and the 8 bit output was converted into it's hexadecimal ASCII representation
- The above process was done with the help of an additional module called "Binary-to-decimal". The algorithm used was repeated division by 10 and remainder with 10 to get the different digits of the number and finally each digit was converted to it's ASCII hexadecimal representation by adding 8'h30 to it.
- The binary to decimal conversion was also done to the two input 4 bit numbers so that they could be displayed.

4.1 Verilog code for displaying wallace multiplier on LCD

- Note: The full_adder, half_adder and the unsigned_mult modules referenced in the top module are the same as the one written under the "Wallace Multiplier" section.

```
module clk_divider (in_Clk,out_Clk);
input in_Clk;
output out_Clk;
reg out_Clk = 0;
reg [27:0]clockCount=0;
always @(posedge in_Clk)

    begin
        clockCount<=clockCount+1;
        if (clockCount == 1000000)
        begin
            clockCount<= 0;
            out_Clk <= ~out_Clk;
        end

    end

endmodule
module binary_to_decimal(
    input [7:0] binary_in,
    output [7:0] digit1,
    output [7:0] digit2,
    output [7:0] digit3
);
    wire [9:0] bin_in;
    assign bin_in = binary_in;
    reg [7:0] temp1, temp2;
    reg [3:0] a, b, c;

    // Calculate decimal digits
    always @(binary_in) begin
        temp1 = bin_in / 10;
        temp2 = bin_in / 100;

        a = bin_in % 10;
        b = temp1 % 10;
        c = temp2 % 10;
    end

    // Convert digits to ASCII
    assign digit1 = a + 8'h30;
    assign digit2 = b + 8'h30;
    assign digit3 = c + 8'h30;

endmodule

module lcd(in_Clk, lcd_rs, lcd_e, data,a,b);
input in_Clk;
output reg [7:0] data;
input [3:0] a,b;
output reg lcd_rs;
reg [7:0]temp;
output lcd_e;
```

```
wire [7:0] M;
wire [7:0] digit1,digit2,digit3,a1,a2,b1,b2,a3,b3;

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'h80; // choose the second line

clk_divider c0 (in_Clk, out_Clk);
assign lcd_e = out_Clk;

reg [3:0] a_prev, b_prev;
reg [3:0] checker=0;

unsigned_mult  hi212(M,a,b,in_Clk);
binary_to_decimal hi21(M,digit1, digit2, digit3);
binary_to_decimal he122(a,a1,a2,a3);
binary_to_decimal he123(b,b1,b2,b3);

always@(posedge lcd_e)
begin
    count = count +1;
\\this piece of code makes the display refresh when the inputs a or b change

if(checker==0)
begin
 a_prev=a;
 b_prev=b;
 checker=1;
end

else
begin

if(a_prev!=a || b_prev!=b)
begin

begin
    count=0;
end

     a_prev=a;
     b_prev=b;
end
end

    case(count)
    1: begin lcd_rs = 0; data = command[0]; end
    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'h50; end // P
    7: begin lcd_rs = 1; data = 8'h52; end // R
    8: begin lcd_rs = 1; data = 8'h4f; end // O
    9: begin lcd_rs = 1; data = 8'h44; end // D
    10: begin lcd_rs = 1; data = 8'h55; end // U
    11: begin lcd_rs = 1; data = 8'h43; end // C
    12: begin lcd_rs = 1; data = 8'h54; end // T
    13: begin lcd_rs = 1; data = 8'h20; end // space
    14: begin lcd_rs = 1; data = 8'h4F; end // O
    15: begin lcd_rs = 1; data = 8'h46; end // F
    16: begin lcd_rs = 1; data = 8'h20; end // space
    17: begin lcd_rs = 1; data = a2; end // :
    18: begin lcd_rs= 1;data=a1;end
    19: begin lcd_rs = 1; data = 8'h26; end // space
    20: begin lcd_rs = 1; data = b2; end // :
    21: begin lcd_rs= 1;data=b1;end
    22: begin lcd_rs=0;data=8'hc0;end
    23: begin lcd_rs = 1; data = 8'h3D; end // space
    24: begin lcd_rs = 1; data = digit3; end // 1s place
    25: begin lcd_rs = 1; data = digit2; end // 10s place
    26: begin lcd_rs = 1; data = digit1; end // 100s place

  //returns cursor to original position
  default: begin lcd_rs = 0; data = 8'h80; end

    endcase
end
endmodule
```

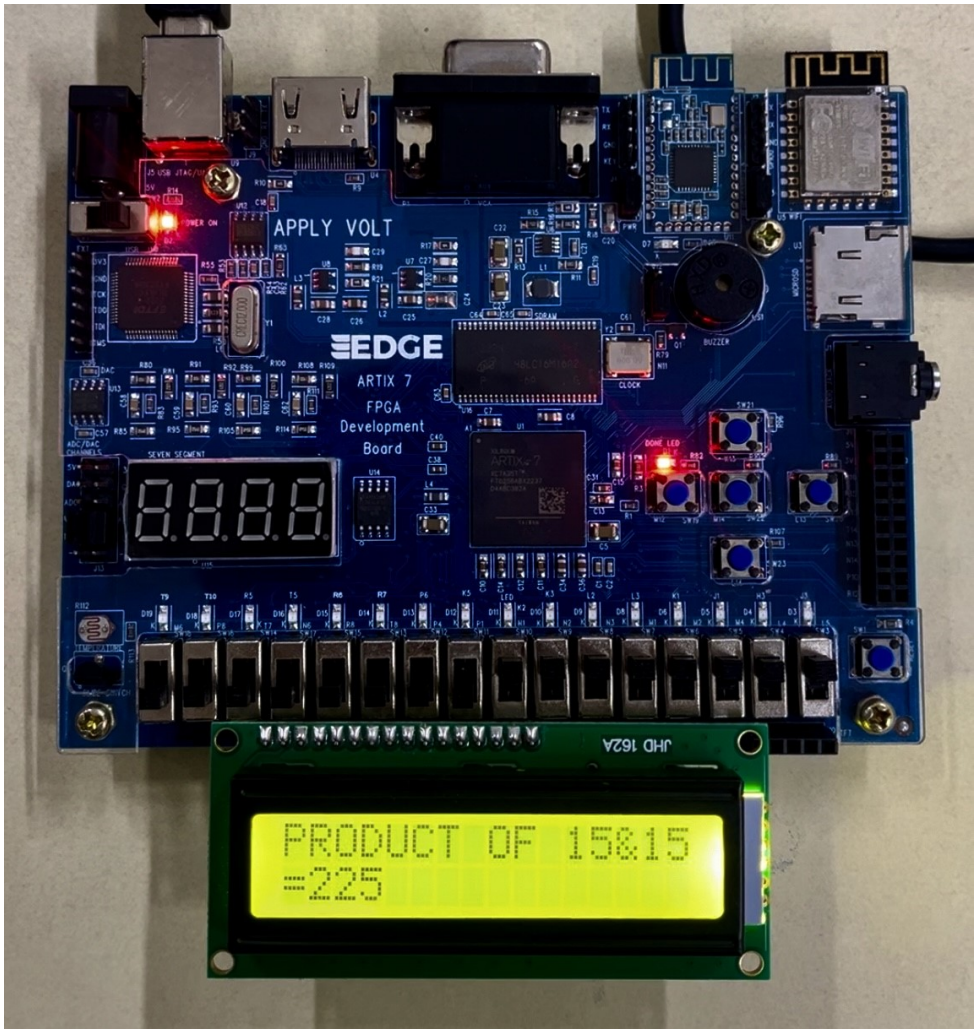

Figure 4: Wallace Multiplier product on LCD

4.2 XDC file

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


# Switches
set_property -dict { PACKAGE_PIN L5    IOSTANDARD LVCMOS33 } [get_ports { a[0] }];
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] }];

# 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}];
```

5 Programming and Debugging experience

- We faced quite a few issues while trying to use the LCD display.
- We did not have a clear understanding of what a "top module" actually meant in the beginning of the LCD part of the experiment, we ran our code with "unsigned_mult" as the top module for a while before realising that was the error.
- This killed a lot of out time, as the top modules input and output bits have to be configured in the xdc file of the FPGA, we had done this for the LCD module, but it kept throwing up the error that some inputs and outputs were not assigned a port.
- After fixing this and successfully generated the bitstream, the LCD module did not display anything. This was later discovered to be the fault of the LCD screen, which did not work. We replaced the screen and the LCD screen displayed content.
- The "count" variable never entered the case statement after it crossed the value "26" which resulted in LCD's inability to change the output displayed when we changed our inputs. The product 'M' changed on the display only when we regenerated the bitstream.
- Our goal was to make use of changes in inputs 'a' and 'b' to trigger reset of the 'count' variable so that the LCD was ready to display the new product obtained.
- In the process of debugging, we realised that a top module cannot have more than 1 "always@" blocks.
- To make the LCD screen refresh whenever the input changes, we kept trying to make the count reset into 0 inside another always block in the LCD module, however this kept giving the error message that the lcd module was no longer recognised as the top module.
- We tried writing always@(a or b), but for some reason, the screen would freeze, possibly due to entering of some race conditions.
- To make sure the 'count' variable reset every time we changed the input, we had to create two new variables 'a prev' and 'b prev' to and compared these with 'a' and 'b' and reset count to 0 whenever the comparison gave a false output.
- To ensure 'a prev' and 'b prev' has some value stored initially to perform the comparison, we had to use a 'checker' variable which was initialised to 0. After this, 'a prev' and 'b prev' were given values 'a' and 'b' respectively and the 'checker' value was updated to 1. Once this was performed, we could directly compare the previous and current values of our inputs and ensure that the LCD displayed the product each time.
- The checker effectively acts like how a boolean type is used in C.
- Before we implemented this logic, we had initially tried making 'count' equal 0 whenever it exceeded 26 but this resulted in a never ending loop in which the product was displayed and cleared again and again even when we didn't change the inputs.
- Thus, the 'a prev','b prev' and checker variables helped us solve this issue.
- The XDC file which was already filled(the one we used) initally had all the LCD ports in reverse, so we had to change the order of that.

- There is one flaw in this code, the "count" variable which is used to display the LCD values is a register of size 32 bits, and it only resets when a or b change. In case you leave it without changing, it will keep counting till it overflows, and when it overflows in Verilog, it resets to 0. As a result, if you don't change inputs for a long enough time, the display will refresh once the register overflows. This however, wont pose any functionality problems apart from refreshing. This can be avoided if required by an if statement, setting the register value to 27 after it is near overflow, or make it stop counting when its past 26.

```
if(count==32'hFFFFFFFF)
begin
    count=27;
end
```

  This snippet of code can be added to the always block if required. We set it to 27 to ensure it does not refresh the screen when it is not required.

- To conclude, if we wrote our code with a little foresight and tested it with testbenches (simulating it on Vivado) instead of manually debugging it, we could have potentially saved a lot of time during the development phase.

6 References

- Note : Some of them are links, hover over them.
- Moodle reference material for Experiment 3.
- Clock divider module from Experiment 2.
- Discussion on Always block.
- Overflow in Verilog.
- A link on LCDs and its workings.
- Wallace Tree Multiplier
- For loops in Verilog