EE2016: Microprocessers Lab - Assignment 3 Report

Aadarsh Ramachandiran (EE23B001), Aditya Kartik (EE23B003), Rohita Datta (EE23B065)

August 31, 2024

Abstract

This is a detailed report on the Verilog assignment on the simulation and implementation of a 4x4-bit Wallace Multiplier on an FPGA board, programming of an LCD to output a set of characters on its display using the FPGA and later, attempting to display the output of the multiplier on the LCD display. The report contains the details of the solution including code, observation and comments on the programming and debugging experience.

1 Introduction

This assignment involves (i) simulating a Wallace Multiplier for two 4-bit inputs using Xilinx Vivado (ii) implementing the Wallace Multiplier design using switches and LEDs on the FPGA board (iii) writing verilog code to output a set of characters on an LCD and (iv) extending the multiplier to output the resultant product on the LCD connected to the FPGA board.

1.1 Wallace Multiplier

A Wallace Multiplier is based on the idea of a Wallace tree. It is a hardware implementation of an n-bit binary multiplier. By multiplying numbers using partial products and reducing the number of partial products to be summed through a series of reductions using adders, it makes the multiplication more efficient. Further details are explained in 2.1

1.2 Interfacing of an LCD with edgeA7 FPGA Board

An LCD is interfaced with the FPGA chip as shown in **Figure 1**. The figure shows the I/O pins involved. In particular, the enable, register select (RS) and data pins need to be used. Register select (RS) setting helps to switch from command register (RS=0) to data register (RS=1). The command register stores instructions (such as clear the display,move the cursor to a certain location etc.) given to the LCD.
(Note: An R/W line is additionally available and one may use it, in general, to specify read (from) or write (to) LCD. In our case, we will restrict to write and assume a default setting (of 0) for write.)

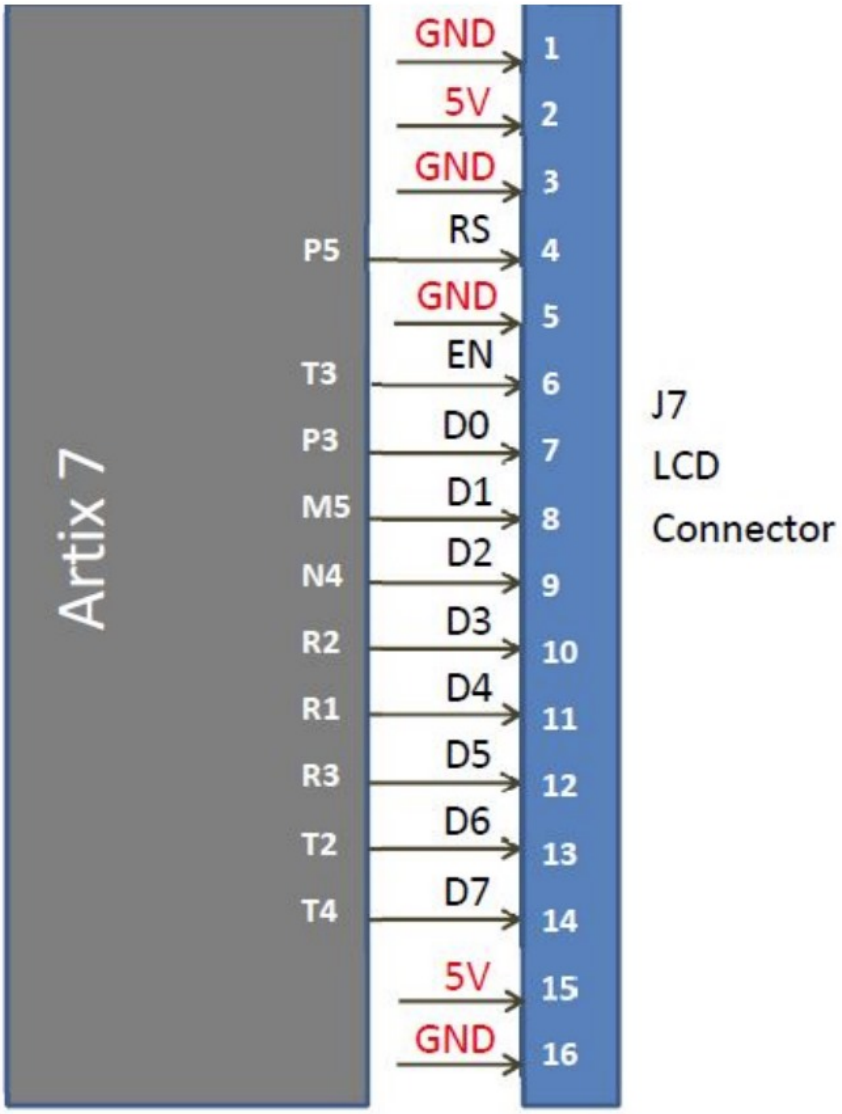

Figure 1: Interfacing the LCD

The LCD is an accessory to the FPGA board and will be used for displaying alphanumeric characters and other symbols using the codes given below. One must take care while handling the LCD board because one of the most common causes of LCD panel failure is physical damage. LCD panels are made up of several layers, including a glass substrate, polarizers, and liquid crystal material. These components are delicate and can be easily damaged by impact, pressure, or bending.

2 Design

2.1 Wallace Multiplier

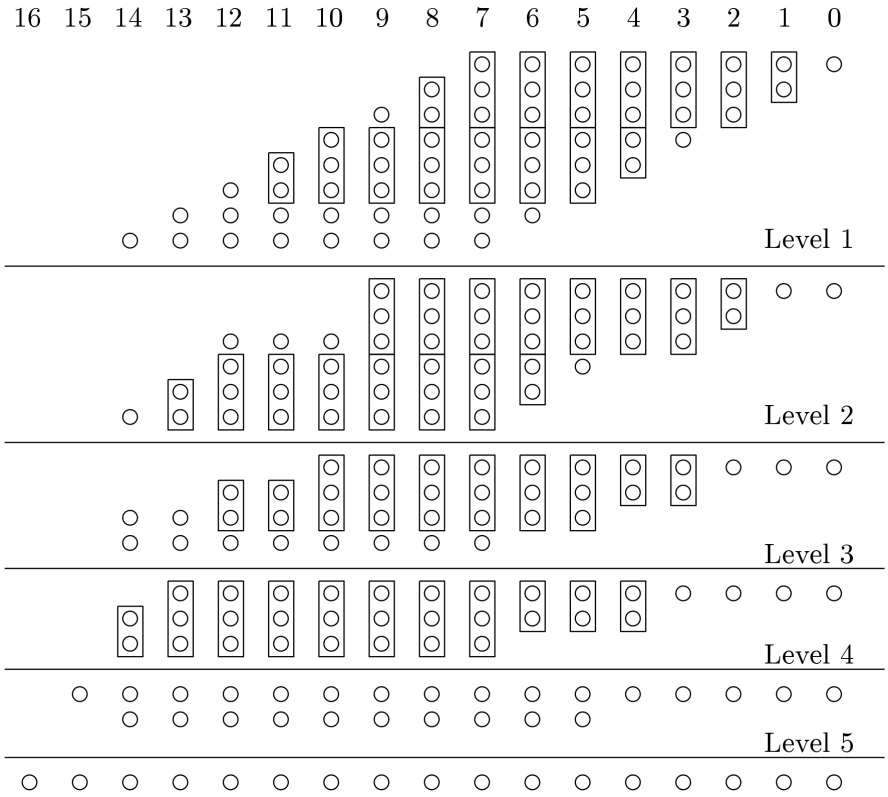

Figure 2: Four layer Wallace reduction of an 8 × 8 product matrix (source)

2.1.1 Description

Figure 2 shows the core idea behind the working of a Wallace multiplier. The multiplier attempts to reduce the number of adders in each level by assigning weights to each term and summing them to get to the next level with fewer terms. Each term $a_m \times b_n$ is assigned a weight of 2^{m+n}, where a_m is the $(m+1)$th digit in vector **a** and b_n is the $(n+1)$th digit in vector **b**. For example, when multiplying a= 101 and b = 1, the partial product a_2b_0 gets the weight 2^{2+0}, which is 4.
The reduction happens in this manner:

1. Any three wires with the same weights are input into a full adder. The result will be an output wire of the same weight and an output wire with a higher weight for each set of three input wires.
2. If there are two wires of the same weight left, input them into a half adder.
3. If there is just one wire left, connect it to the next layer.

For the case of multiplication of two 4-bit numbers $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$ the method goes as follows:

- **Reduction layer 1:**
  - Pass the only weight-1 wire through (a_0b_0), output: 1 weight-1 wire.
  - Add a half adder for weight 2 (a_1b_0 and a_0b_1), outputs: 1 weight-2 wire, 1 weight-4 wire.
  - Add a full adder for weight 4 (a_0b_2, a_1b_1, a_2b_0), outputs: 1 weight-4 wire, 1 weight-8 wire.
  - Add a full adder for weight 8 (a_0b_3, a_1b_2, a_2b_1), and pass the remaining wire (a_3b_0) through, outputs: 2 weight-8 wires, 1 weight-16 wire.

  - Add a full adder for weight 16 (a_1b_3, a_2b_2, a_3b_1), outputs: 1 weight-16 wire, 1 weight-32 wire.
  - Add a half adder for weight 32 (a_2b_3, a_3b_2), outputs: 1 weight-32 wire, 1 weight-64 wire.
  - Pass the only weight-64 wire (a_3b_3) through, output: 1 weight-64 wire.

- **Wires at the output of reduction layer 1:**
  - Weight 1 – 1
  - Weight 2 – 1
  - Weight 4 – 2
  - Weight 8 – 3
  - Weight 16 – 2
  - Weight 32 – 2
  - Weight 64 – 2

- **Reduction layer 2:**
  - Add a full adder for weight 8, and half adders for weights 4, 16, 32, 64.

- **Outputs:**
  - Weight 1 – 1
  - Weight 2 – 1
  - Weight 4 – 1
  - Weight 8 – 2
  - Weight 16 – 2
  - Weight 32 – 2
  - Weight 64 – 2
  - Weight 128 – 1

- Group the wires into a pair of integers and use an adder to add them.

The partial product generator generates partial products using a simple two-input AND gate that is fed to the Wallace tree adder. Multiple half and full adders are used that does additions in multiple levels and also considers carry generated by a previous level adder. The last level of the Wallace tree adder can be implemented using a ripple carry adder. (Alternatively, a set of full-adders also work) In the next section, the code for the implementation of a Wallace Multiplier for two 4-bit numbers is shown.

2.1.2 Code

Wallace Multiplier module:

```
// This code describes a module to perform "Wallace" multiplication of
//two 4-bit numbers
// The Wallace multiplier is a circuit that does multiplication of two
//integers faster by producing partial products and then adding them up
//with layers of appropriate adders.

module unsigned_mult(m, a, b);
input [3:0] a, b;
output [7:0] m;

wire [3:0] p0, p1, p2, p3;
wire s5, s4, s3, s2, s1, s0;
wire c4, c3, c2, c1, c0;
```

```
wire k1, l1, k2, l2;

// assign partial products
assign p0 = a & {4{b[0]}}; // {4{x}} for 1'bx input results in 4'bxxxx output
assign p1 = a & {4{b[1]}};
assign p2 = a & {4{b[2]}};
assign p3 = a & {4{b[3]}};

assign m[0] = p0[0];

half_adder ha1(s0, c0, p0[1], p1[0]);
assign m[1] = s0;

full_adder fa1(s1, c1, p0[2], p1[1], p2[0]);

half_adder ha3(k1, l1, p2[1], p3[0]);
full_adder fa2(s2, c2, p0[3], p1[2], k1);

half_adder ha4(k2, l2, p2[2], p3[0]);
full_adder fa3(s3, c3, k2, l1, p1[3]);

full_adder fa4(s4, c4, p2[3], p3[2], l2);

ripple_carry_adder rca(m[6:2], m[7], {p3[3], s4, s3, s2, s1}, {c4, c3, c2, c1, c0});
// {} denotes concatenation
endmodule

// We use the half-adder and full-adder and ripple carry adder modules
//created in Assignment 1, here.
// The ripple carry adder is modified to handle 5-bit inputs.

// This code describes a dataflow model of a half adder logic circuit
module half_adder(s, c, a, b);
input a;
input b;
output s;
output c;
assign s = a ^ b;
assign c = a & b;
endmodule

// This code describes a dataflow model of a full adder logic circuit
module full_adder(s, c, x, y, z);
input x, y, z;
output s, c;
wire s1, c1, c2;

assign s1 = x ^ y;
assign c1 = x & y;
assign s = s1 ^ z;
assign c2 = s1 & z;
assign c = c1 | c2;
endmodule

// This code describes a model for 5-bit ripple carry adder using x5 full adders
// (carry input port is removed since it is not necessary)
```

```
module ripple_carry_adder(o, cout, i1, i2);
input [4:0] i1, i2;
wire cin;
wire c1, c2, c3, c4;
output[4:0] o;
output cout;

assign cin = 1'b0; // in our case

full_adder fa1(o[0], c1, i1[0], i2[0], cin);
full_adder fa2(o[1], c2, i1[1], i2[1], c1);
full_adder fa3( o[2], c3, i1[2], i2[2], c2);
full_adder fa4(o[3], c4, i1[3], i2[3], c3);
full_adder fa5(o[4], cout, i1[4], i2[4], c4);
```

endmodule

We also write a testbench to simulate the circuit and verify its veracity using Vivado.
Wallace Multiplier testbench:

```
`timescale 1ns/1ps
module tb_wallace_multiplier();
reg [3:0] a, b;
wire [7:0] m;
unsigned_mult um(m, a, b);
initial begin
    $dumpfile("test_wallace_multiplier.vcd");
    $dumpvars(1, tb_wallace_multiplier);
    #10
    a = 4'd0;
    b = 4'd0;
    #10
    a = 4'd3;
    b = 4'd2;
    #10
    a = 4'd12;
    b = 4'd13;
    #10 $finish;
end
initial $monitor($time, "a=%b, b=%b, m=%b", a, b, m);
endmodule
```

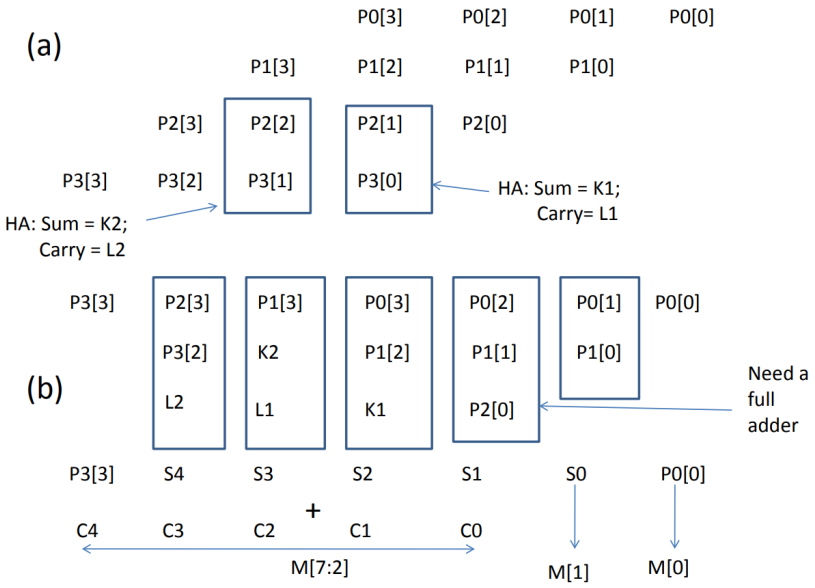

Figure 3: Visual representation of the multiplication method demonstrated in the code

2.2 Displaying characters on an LCD

2.2.1 Description

We desire to display a set of characters (”1 2 3 4 5 6 7 8 9 A B C D E F G”) on the liquid crystal display (LCD) as shown in **Figure 4**. We achieve this by interfacing the LCD with an FPGA and programming it using Verilog in a module called lcd. We input a clock signal which dictates the sequential actions of the module. The outputs lcd_e, lcd_rs and data are for enabling the lcd display at clock edge triggers, selecting appropriate register and the data to be sent to the lcd. The data is of 8-bit size and can either consist of a command for some action in the LCD or an ASCII character to be displayed whose inputs are given in hexadecimal format (8-bits). The program interprets any given data as a command or display output based on the corresponding register select value. 0 denotes command register and 1 denotes data register. We use five common commands which are for function control, switching on the display, incrementing the cursor, clearing the display and for choosing the second line. This seemed sufficient for our purposes. Other common commands can be found in **Figure 5**.

Figure 4: The desired output of characters in LCD display

Hex Code	Command to LCD Instruction Register
01	Clear display screen
02	Return home
04	Decrement cursor (shift cursor to left)
06	Increment cursor (shift cursor to right)
05	Shift display right
07	Shift display left
08	Display off, cursor off
0A	Display off, cursor on
0C	Display on, cursor off
0E	Display on, cursor blinking
0F	Display on, cursor blinking
10	Shift cursor position to left
14	Shift cursor position to right
18	Shift the entire display to the left
1C	Shift the entire display to the right
80	Force cursor to beginning (1st line)
C0	Force cursor to beginning (2nd line)
38	2 lines and 5×7 matrix

Figure 5: Common commands to LCD instruction register

We make use of a count variable and a switch-case statement block based on the count to facilitate the sequential inputs to be given. The inputs will be the five commands on entry and then the ASCII values for the characters to be displayed, in order. The default statement in the switch case is the command to force cursor back to first line. A clock divider module made in the previous assignment is added here to reduce the frequency to allow readability of the output. The verilog code for the LCD module can be seen below.

2.2.2 Code

LCD module:

```
// This code is to display a set of characters in 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 the two lines
assign command [1] = 8'h0C; // Keep display on, cursor and blink off
assign command [2] = 8'h06; // Increment cursor, no shift
assign command [3] = 8'h01; // Clear display
```

```
assign command [4] = 8'hC0; // Choose second line

clk_divider cd(out_Clk, in_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
  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
  7: begin lcd_rs = 1; data = 8'h32; end
  8: begin lcd_rs = 1; data = 8'h33; end
  9: begin lcd_rs = 1; data = 8'h34; end
  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
  20: begin lcd_rs = 1; data = 8'h46; end
  21: begin lcd_rs = 1; data = 8'h47; end

  default: begin lcd_rs = 0; data = 8'h80; end

  endcase
end
endmodule

// Code for clock divider module used (taken from Assignment 2)
// This code demonstrates a clock divider module
// to reduce clock frequency to observe results in an FPGA board
module clk_divider(outClk, inClk);
input inClk;
output reg outClk;
//reg clockCount;
reg [25:0] clockCount;

always@(posedge inClk)
begin
// if (clockCount >= 1'd1)
if (clockCount >= 26'd50000000)
    begin
        // clockCount = 1'd0;
        clockCount = 26'd0;
        outClk = ~ outClk;
    end
clockCount <= clockCount + 1;
```

end
endmodule

2.3 Displaying the Product of a Multiplier on an LCD

2.3.1 Description

We have implemented a fully functional four-bit multiplier and programmed an LCD to display characters. We also attempt to combine both to display the product of two four-bit inputs from an FPGA on the LCD screen written in a format as shown.

$$Product\,is\,=$$

$$XXXX$$

where the X's denote a placeholder for the resultant product. We follow a similar method for the display as previously. However, we additionally have to convert the binary product output to a decimal representation encoded as ASCII. For this, we attempt to first convert the binary output to a binary coded decimal (BCD) and then display each set of four bits in the BCD form in its ASCII representation. The conversion from binary to BCD is implemented using a behavioural module that uses the double-dabble algorithm. The key idea is to initialize two registers, one to store the binary number and the other to store the BCD, shift left 1-bit at a time from the binary register to the BCD one while correcting the BCD values by adding three each time the value per four-bit group in the BCD register becomes greater than or equal to five. The idea is inspired from an observation that a binary number in an invalid BCD format becomes valid after an addition of six to the number. Given our way of left-shifting (which can be interpreted as $\times 2$) we add three and we check for numbers greater than five since after being doubled numbers greater than five become invalid and addition of three occurs twice which results in addition of six thereby correcting the BCD values, wherever necessary.
We also attempt to detect changes in the input which would result in clearing of existing outputs and the display of the new output. This can be done with a simple if condition in the always block preceding the switch-case statement. The code implementation for this module can be seen below.

2.3.2 Code

LCD bonus module:

```
// This code is to display the product of a multiplier in an lcd display
module lcd(in_Clk, a, b, lcd_rs, lcd_e, data);
input in_Clk;
input [3:0] a, b;
output reg [7:0] data;
output reg lcd_rs;
output lcd_e;

reg [3:0] a_old = 4'b0000, b_old = 4'b0000;
wire [7:0] command [0:5];
reg [31:0] count = 0;
wire out_Clk;
wire [7:0] m;
wire [11:0] bcd;
assign command [0] = 8'h38; // Control signal to display on the two lines
assign command [1] = 8'h0C; // Keep display on, cursor and blink off
assign command [2] = 8'h06; // Increment cursor, no shift
assign command [3] = 8'h01; // Clear display
assign command [4] = 8'h80; // choose first line
assign command [5] = 8'hC0; // Choose second line
```

```
clk_divider cd(out_Clk, in_Clk);
assign lcd_e = out_Clk;

unsigned_mult um(m, a, b);
// convert m to bcd
bin2bcd b1(bcd, m);

always@(posedge lcd_e) begin
  if (a != a_old || b != b_old) begin
    count <= 0;
    a_old <= a;
    b_old <= b;
  end
  if (count <= 21) begin
    count  <= count + 1;
    end
  case(count)
  // Entry command
  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
  // Display "Product is ="
  6: begin lcd_rs = 1; data = 8'h50; end // P
  7: begin lcd_rs = 1; data = 8'h72; end // r
  8: begin lcd_rs = 1; data = 8'h6F; end // o
  9: begin lcd_rs = 1; data = 8'h64; end // d
  10: begin lcd_rs = 1; data = 8'h75; end // u
  11: begin lcd_rs = 1; data = 8'h63; end // c
  12: begin lcd_rs = 1; data = 8'h74; end // t
  13: begin lcd_rs = 1; data = 8'h20; end // space
  14: begin lcd_rs = 1; data = 8'h69; end // i
  15: begin lcd_rs = 1; data = 8'h73; end // s
  16: begin lcd_rs = 1; data = 8'h20; end // space
  17: begin lcd_rs = 1; data = 8'h3D; end // equal to

  18: begin lcd_rs = 0; data = command[5]; end // command for next line

  // Display the product
  19: begin lcd_rs = 1; data = bcd[11:8] + 8'h30; end
  20: begin lcd_rs = 1; data = bcd[7:4] + 8'h30; end
  21: begin lcd_rs = 1; data = bcd[3:0] + 8'h30; end

  default: begin lcd_rs = 0; data = 8'h80; end
  endcase
end
endmodule

// Code for clock divider module used (taken from Assignment 2)
// This code demonstrates a clock divider module
// to reduce clock frequency to observe results in an FPGA board
module clk_divider(outClk, inClk);
input inClk;
output reg outClk = 0;
```

```
//reg clockCount;
reg [25:0] clockCount;

always@(posedge inClk)
begin
// if (clockCount >= 1'd1)
if (clockCount >= 26'd50000000)
    begin
        // clockCount = 1'd0;
        clockCount = 26'd0;
        outClk = ~ outClk;
    end
clockCount <= clockCount + 1;

end
endmodule

// We need to convert binary to bcd for display in lcd
module bin2bcd( // double-dabble algorithm - behavioural model
   output reg [11:0] bcd,
   input [7:0] bin
   );
integer i;
always @(bin) begin
    bcd=0;
    // Iterate once for each bit in input number
    for (i=0;i<8;i=i+1) begin
        // If any BCD digit is >= 5, add three
        if (bcd[3:0] >= 5) bcd[3:0] = bcd[3:0] + 3;
            if (bcd[7:4] >= 5) bcd[7:4] = bcd[7:4] + 3;
            if (bcd[11:8] >= 5) bcd[11:8] = bcd[11:8] + 3;
            // Shift one bit, and shift in the MSB from input
            bcd = {bcd[10:0],bin[7-i]};
    end
end
endmodule
```

3 Implementation

Once the Verilog code for all the circuits are done, we proceed to simulate the modules in Xilinx and realize the results on an FPGA board (and LCD display). We create a new project in Vivado and add the required Verilog files (related modules and testbench) through the "Add sources" option.
Note: It must be ensured that the testbench is declared as "top-level" in the hierarchy of files.

3.1 Wallace Multiplier

3.1.1 Simulating in Xilinx

After adding the verilog files, we simulate them. Vivado shows the outputs and pops open the waveform veiwer to display the waveforms of various signals. The waveforms for the Wallace Multiplier can be found in 6.

3.1.2 FPGA Realization

Once the outputs are analyzed and the circuit is verified to be correct, we proceed to implement it on the hardware. We write a Xilinx Design Constraints (.xdc) file to configure the required ports to be

used in the FPGA board. (switches for the inputs and LEDs for displaying the output)

Code for the.xdc file to implement the Wallace Multiplier

```
# This describes the constraints file used for implementation of the Wallace
Multiplier on the FPGA board
# 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] }];

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

We connect the FPGA board to the computer and turn it on before generating the bitstream by clicking "Generate Bitstream" in the Xilinx menu. Then, we program the device by clicking "Program device". The circuit will be successfully implemented in the FPGA board.

3.2 Displaying a set of characters on the LCD

We use the Verilog LCD module along with an appropriate .xdc constraints file to display hexadecimal characters on the LCD. The constraints file dictates the ports to be used for the command and data pins.

Code for the .xdc file for displaying characters on the LCD

```
# Design constraints file for lcd display module
# Data pins
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]}];
# Command pins
set_property -dict { PACKAGE_PIN T3 IOSTANDARD LVCMOS33 } [get_ports {lcd_e}];
set_property -dict { PACKAGE_PIN P5 IOSTANDARD LVCMOS33 } [get_ports {lcd_rs}];
set_property -dict { PACKAGE_PIN N11 IOSTANDARD LVCMOS33 } [get_ports { in_Clk }];
```

3.3 Displaying the product on the LCD

We show the output of the Wallace multiplier on the LCD for various pairs of 4-bit numbers (which are input using the sliding switches on the FPGA) by printing "Product is =" on the first line and

the actual product in decimal format on the second line of the LCD. The following code describes the constraints file which determine the ports to be used. As before, we declare ports for the data pins, command pins and additionally, sliding switches for the input entries.

Code for the .xdc file to display the Wallace Multiplier's output on LCD

```
# Design constraints file for lcd display module
# Data pins
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]}];
# Number pins
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] }];
# Command pins
set_property -dict { PACKAGE_PIN T3 IOSTANDARD LVCMOS33 } [get_ports {lcd_e}];
set_property -dict { PACKAGE_PIN P5 IOSTANDARD LVCMOS33 } [get_ports {lcd_rs}];
set_property -dict { PACKAGE_PIN N11 IOSTANDARD LVCMOS33 } [get_ports { in_Clk }];
```

4 Observation

4.1 Wallace Multiplier

We use the sliding switch in the FPGA board for the two four-bit inputs and the output (product) will be displayed via the LEDs. The range of possible outputs is [0,255] hence, we can comfortably display the desirable output using 8 LEDs representing each bit of the product.

4.2 LCD

The required characters were displayed on the LCD screen by connecting it to the computer and programming the hardware using the Verilog code. Following that, the product for inputs given via switches was displayed on the LCD. Upon changing the inputs using the switches, the LCD was reset to initial state and printed out the new result. The speed at which the output was being printed could be controlled by varying the clock divider code.

Note: One concerning issue we observed during the experiment is that Xilinx does not really throw an error for some bugs in the code. The simulation and hardware implementation will be done successfully, only that the outputs would be wrong. Careful analysis of the code must be done to verify its correctness.

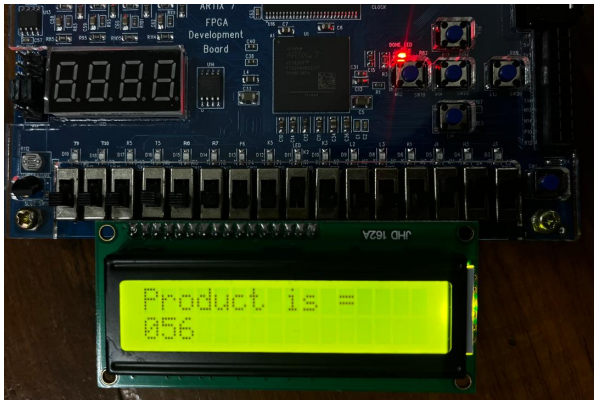

Figure 6: The product of the multiplication was displayed on the LCD as shown

5 Conclusion

The experiment successfully demonstrated the design and simulation of a Wallace Multiplier, which was simulated and implemented on an FPGA board using LEDs and switches. Further, programming an LCD to display a simple set of characters was also demonstrated. This was then used to extend the multiplier by displaying its product on the LCD.

6 Appendices

The verilog codes used in this assignment are available here
The waveforms for various circuits can be seen below:

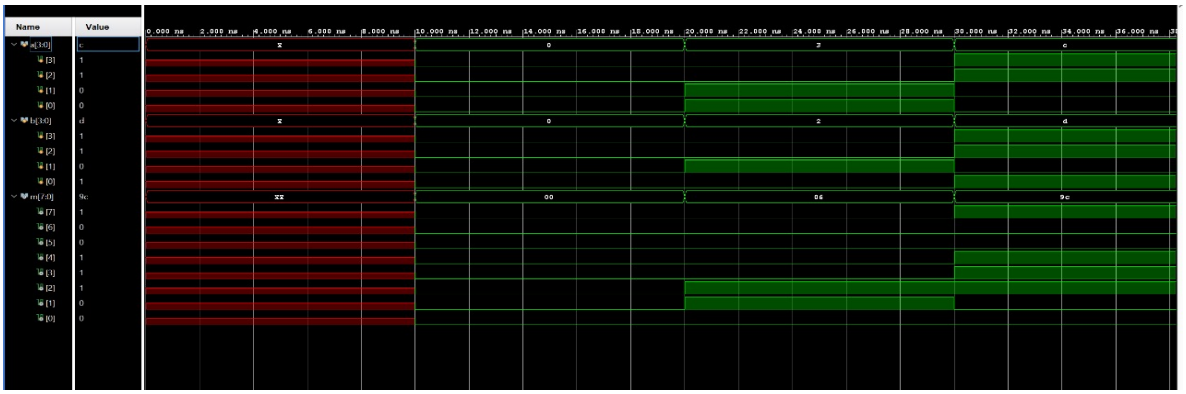

Figure 7: Wallace Multiplier Waveform

The 4 bit inputs, a and b, and their product can be seen in the above waveform. The input values fed from the test bench and the product can be seen here in hexadecimal form (initially in indeterminate form). In the settings, we can change it from hexadecimal to decimal or binary as per our convenience.

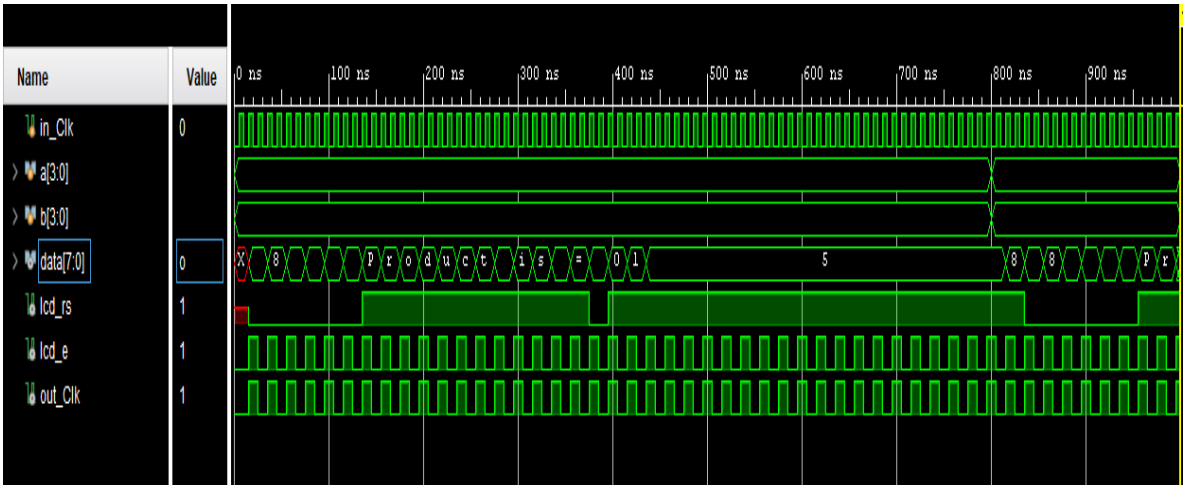

Figure 8: LCD Bonus Waveform

The LCD output (data) of printing "Product is =" followed the the output can be realised from this. Note that, the control commands are specific to the LCD hardware and, hence, will be displayed as something else in the simulation. We neglect this for our purposes.