Veronica Gonzalez

CPE 166: Advanced Logic Design

Lab Session: 02

Lab Day: Wednesday 2:00 PM

Lab 2

Binary Combinational Array Multiplier Design, Sequential Multiplier Design and Introduction to LCD

Lab Instructor: Jing Pang

Table of Contents

Objectives:

- Learn hierarchical design strategy using Verilog Hardware Description Language
- Learn array combinational multiplication algorithm
- Learn sequential shift/add multiplication algorithm
- Learn how to use LCD
- Learn how to simulate Verilog programs
- Learn how to download application program to Spartan3E starter board

Materials:

- Spartan3E Starter Board
- +5VDC Power Supply
- USB Cable
- Xilinx ISE Tool

Introduction: The purpose of this lab is to introduce the hierarchical design strategy using Verilog hardware description language to design a combinational and sequential circuits. Also, to introduce the Spartan3E board and its capabilities to be programmed by the designs made in Verilog, such as the LCD component. Knowledge of how to test our designs using testbench in Verilog and to run simulations to verify the behavior is correctly described in Verilog. We can program the Spartan3E board to do the behavior described. Learn how to make a User Constraint file to use input and output signals to the board.

Part 1: 4 By 4 Binary Combinational Array Multiplier Design

Purpose: The purpose is to learn how binary multiplication is derived using an array of logic circuits. To design a 4 bit by 4-bit combinational array we implement the design by using Half Adders, Full Adders, and AND gates. The AND gate is used to derive the logic equation for a 1-bit binary multiplier because for each of the four partial products need to be shifted left by 1-bit. The products are then added together by the half adders and full adders.

```
multiplicand  1011
multiplier  * 1101
              1011
             0000
            1011
           1011
          --------
          10001111
```

Figure 1-1: 1-Bit Binary Multiplier Example.

Inputs		Output
a	b	f
0	0	0
0	1	0
1	0	0
1	1	1
Logic Equation: f=a b		

Table 1-1: 1-Bit Binary Multiplier Truth Table.

1-1. Half Adder Design

Description: The half adder adds two 1-bit binary inputs and generates two output signals, the sum signal and the carry cout signal. The truth table is used to derive the logic equation and the equation is used to create the half adder schematic. The half adder code in Verilog is used to describe the behavior in the circuit schematic shown in Figure 1-3. The testbench in figure 1-4 is used to simulate the behavior of the half adder as shown in Figure 1-5. The simulation waveform is used against the truth table to verify that the Verilog code is describing the correct behavior of the half adder circuit.

Inputs		Outputs	
a	b	cout	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0
Logic Equations: cout = a b sum = a xor b			

Table 1-2: Half Adder Truth Table and Equations.

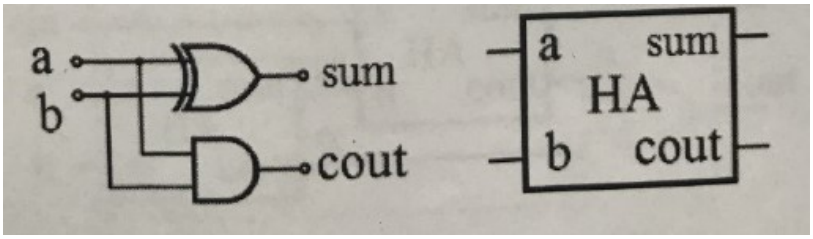

Figure 1-2: Half Adder Schematic and Symbol.

```
1  module ha(a,b,cout,sum);
2      input a,b;
3      output cout, sum;
4
5      assign cout=a&b;
6      assign sum=a^b;
7  endmodule
8
```

Figure 1-3: Half Adder Verilog Code.

```
1  module ha_tb;
2      reg a,b;
3      wire cout, sum;
4
5      ha u1(.a(a),.b(b),.cout(cout),.sum(sum));
6
7      initial begin
8      a=0; b=0;
9      #10 a=0; b=1;
10     #10 a=1; b=0;
11     #10 a=1; b=1;
12     #10 $stop;
13     end
14
15 endmodule
```

Figure 1-4: Half Adder Verilog Testbench Code.

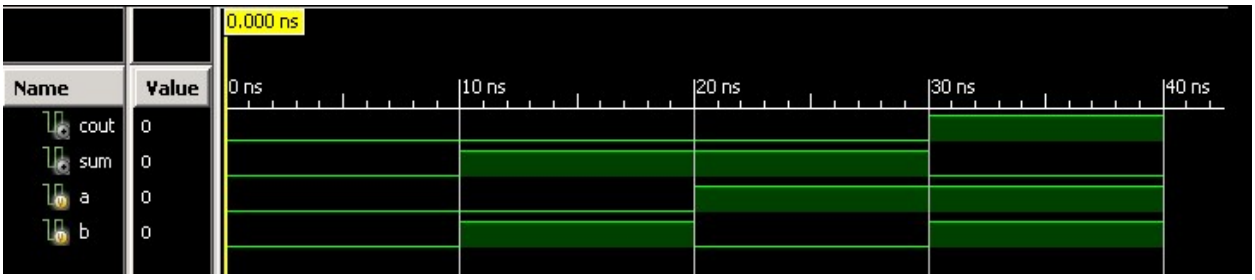

Figure 1-5: Half Adder Testbench Waveform Simulation.

1-2. Full Adder Design

Description: The full adder adds three 1-bit binary inputs a, b, and cin to generate two output signals for the sum and carry. The truth table in Table 1-3 is to derive the logic equations to create the full adder schematic. The full adder Verilog code is used to describe the behavior in

the circuit schematic shown in Figure 1-6. The Verilog testbench in figure 1-8 is used to simulate the behavior of the full adder as shown in Figure 1-9. The simulation waveform is used against the truth table to verify that the Verilog code is describing the correct behavior of the full adder circuit.

Inputs			Outputs	
a	b	cin	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1
Logic Equations: sum = a xor b xor cin cout = (a xor b) cin + a b				

Table 1-3: Full Adder Truth Table and Equations.

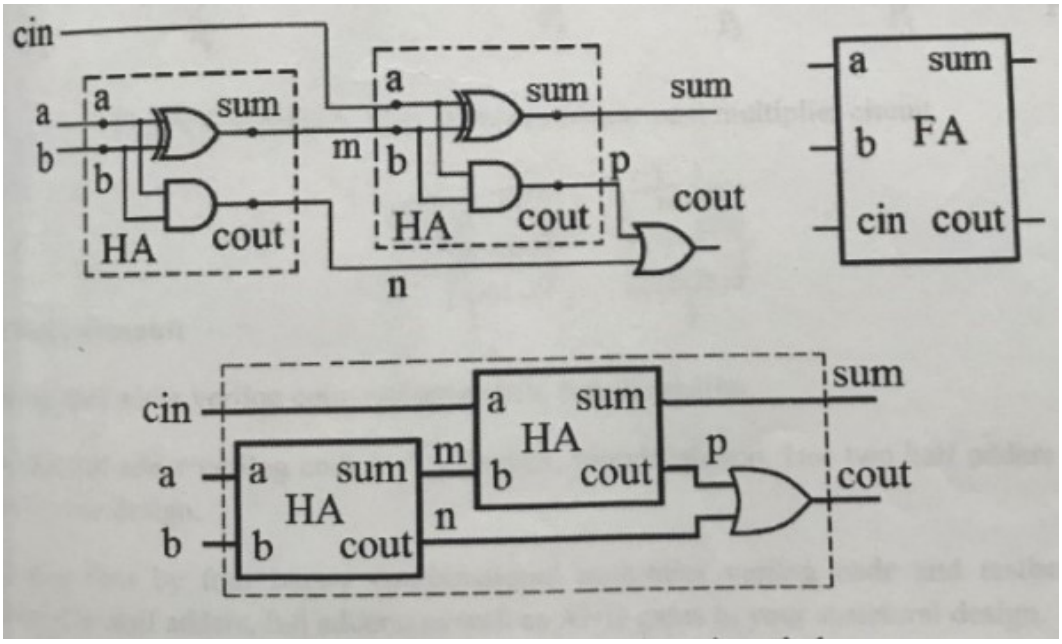

Figure 1-6: Full Adder Schematics and Symbol.

```
1  module fa(a,b,cin,cout,sum);
2     input a,b,cin;
3     output cout,sum;
4
5     wire m,n,p;
6
7     assign cout=p|n;
8
9     ha u1(.a(a),.b(b),.sum(m),.cout(n));
10    ha u2(.a(cin),.b(m),.sum(sum),.cout(p));
11
12 endmodule
```

Figure 1-7: Full Adder Verilog Code.

```
1  module fa_tb;
2     reg a,b,cin;
3     wire sum, cout;
4
5     fa u1(.a(a),.b(b),.cin(cin),.sum(sum),.cout(cout));
6
7     initial begin
8     a=0; b=0; cin=0;
9     #10 a=0; b=0; cin=1;
10    #10 a=0; b=1; cin=0;
11    #10 a=0; b=1; cin=1;
12    #10 a=1; b=0; cin=0;
13    #10 a=1; b=0; cin=1;
14    #10 a=1; b=1; cin=0;
15    #10 a=1; b=1; cin=1;
16    #10 $stop;
17    end
18
19 endmodule
20
```

Figure 1-8: Full Adder Verilog Testbench Code.

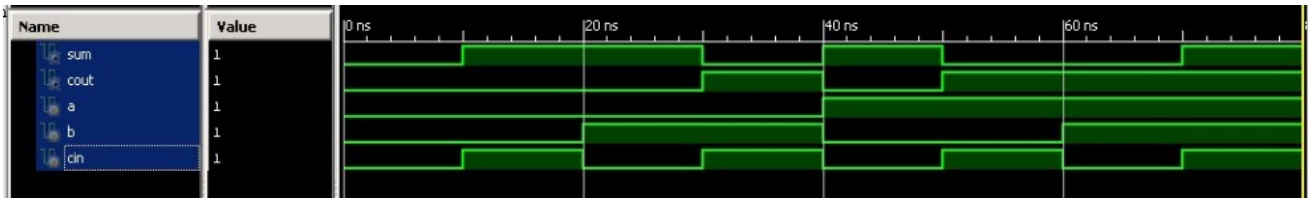

Figure 1-9: Full Adder Testbench Waveform Simulation.

1-3. 4 By 4 Binary Combinational Multiplier Design

Description: A combinational array multiplier is constructed of half adders, full adders, and AND gates to construct a complex arithmetic circuit. We refer to this as the 4 by 4 combinational multiplier. The wire names and gate names used in Verilog are as written in the Figure 1-10 circuit. To describe this design in Verilog, we use instances of the half adder and full adder that were created from figure 1-3 and figure 1-7. Then we add all the necessary AND gates by using the "assign" statements in Verilog and everything is connected by wire names written in figure 1-10. In the Verilog testbench simulation we test to see that the multiplier behaves correctly by assigning two inputs x and y, then see that the product p is correct value. From the testbench in figure 1-12 the two inputs are assigned multiplicand and operator values and we see that in the product values in the waveform in figure 1-13 is verified to be correct as one of the tests is the values from figure 1-1.

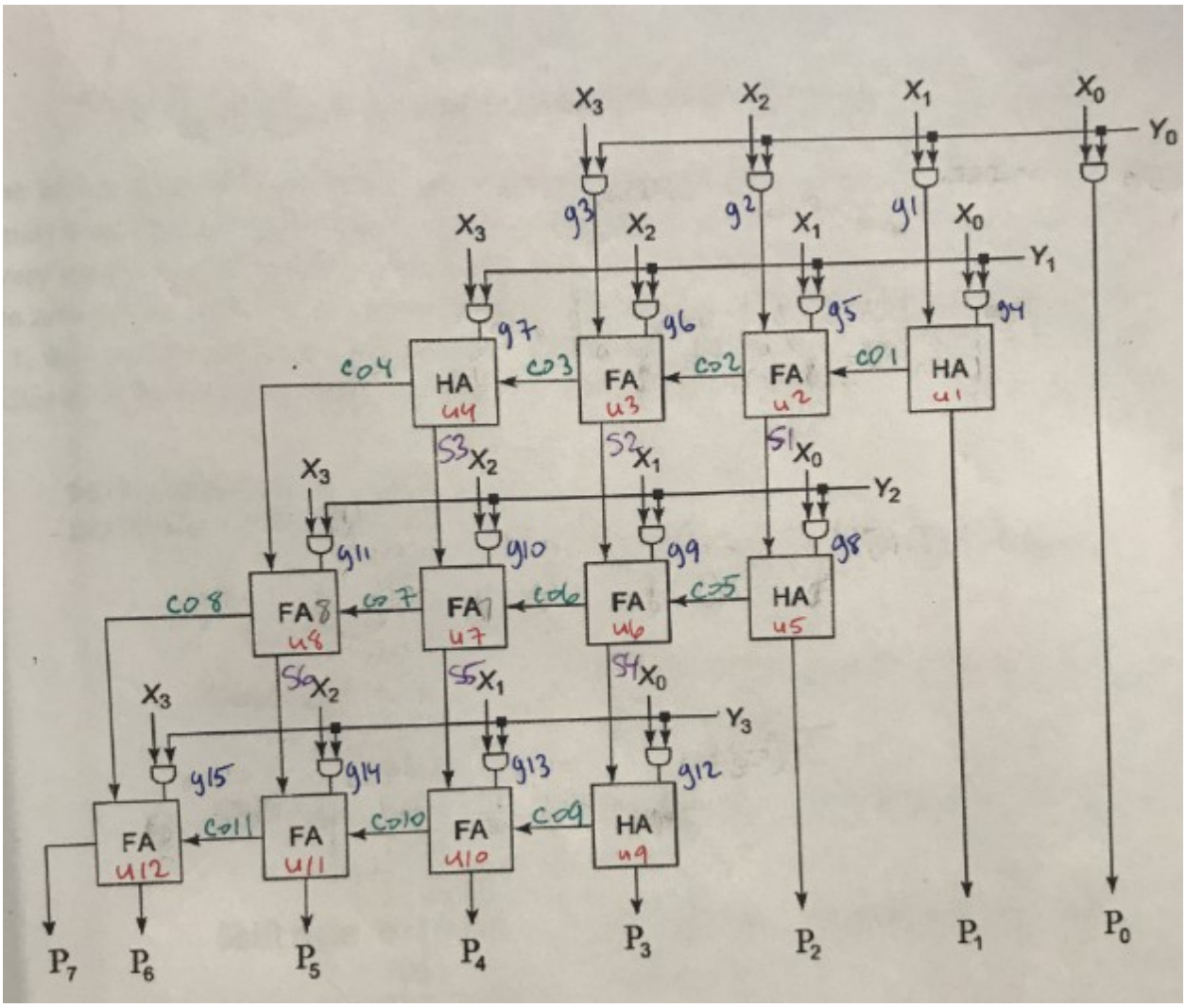

Figure 1-10: 4 By 4 Binary Combinational Multiplier Circuit.

```
module multiplexer(x,y,p);
input [3:0]x, y;
output[7:0]p;

wire g1,g2,g3,g4,g5,g6,g7,g8,g9,g10,g11,g12,g13,g14,g15;
wire co1,co2,co3,co4,co5,co6,co7,co8,co9,co10,co11;
wire s1,s2,s3,s4,s5,s6;

assign p[0]=x[0]&y[0];
assign g1= y[0] & x[1];
assign g2= y[0] & x[2];
assign g3= y[0] & x[3];

assign g4= y[1] & x[0];
assign g5= y[1] & x[1];
assign g6= y[1] & x[2];
assign g7= y[1] & x[3];

assign g8= y[2] & x[0];
assign g9= y[2] & x[1];
assign g10= y[2] & x[2];
assign g11= y[2] & x[3];

assign g12= y[3] & x[0];
assign g13= y[3] & x[1];
assign g14= y[3] & x[2];
assign g15= y[3] & x[3];

ha u1(.a(g4),.b(g1),.cout(co1),.sum(p[1]));
fa u2(.a(g5),.b(g2),.cin(co1),.cout(co2),.sum(s1));
fa u3(.a(g6),.b(g3),.cin(co2),.cout(co3),.sum(s2));
ha u4(.a(co3),.b(g7),.cout(co4),.sum(s3));
ha u5(.a(g8),.b(s1),.cout(co5),.sum(p[2]));
fa u6(.a(g9),.b(s2),.cin(co5),.cout(co6),.sum(s4));
fa u7(.a(g10),.b(s3),.cin(co6),.cout(co7),.sum(s5));
fa u8(.a(g11),.b(co4),.cin(co7),.cout(co8),.sum(s6));
ha u9(.a(g12),.b(s4),.cout(co9),.sum(p[3]));
fa u10(.a(g13),.b(s5),.cin(co9),.cout(co10),.sum(p[4]));
fa u11(.a(g14),.b(s6),.cin(co10),.cout(co11),.sum(p[5]));
fa u12(.a(g15),.b(co8),.cin(co11),.cout(p[7]),.sum(p[6]));

endmodule
```

Figure 1-11: 4 By 4 Combinational Multiplier Verilog Code.

```
1  module multiplexer_tb;
2
3     // Inputs
4     reg [3:0] x;
5     reg [3:0] y;
6
7     // Outputs
8     wire [7:0] p;
9
10    // Instantiate the Unit Under Test (UUT)
11    multiplexer uut (.x(x), .y(y), .p(p));
12
13    initial begin
14       // Initialize Inputs
15       x = 4'b0000;y = 4'b0000;
16       #10 x=4'b0001; y=4'b1101;
17       #10 x=4'b0011;
18       #10 x=4'b0100;
19       #10 x=4'b0101;
20       #10 x=4'b0110;
21       #10 x=4'b0111;
22       #10 x=4'b1011;
23       #10 $stop;
24    end
25
26 endmodule
27
28
```

Figure 1-12: 4 By 4 Combinational Multiplier Verilog Testbench Code.

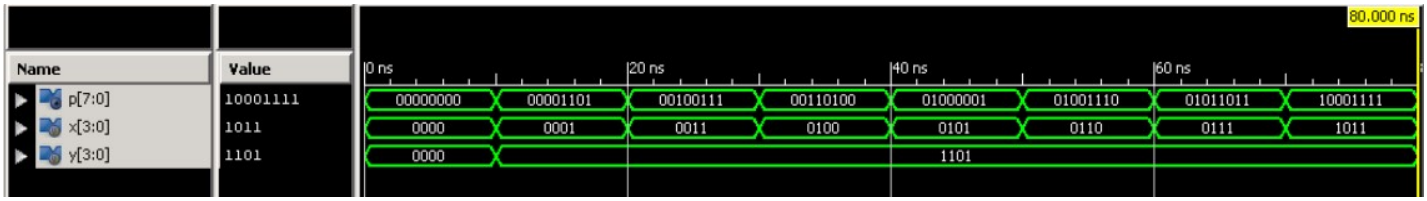

Figure 1-13: 4 By 4 Combinational Multiplier Testbench Waveform Simulation.

Part 2: 4 By 4 Binary Sequential Multiplier Design

2-1. Sequential Multiplier Circuit Design

Description: The purpose of the sequential multiplier is to learn how to do a shift and add algorithm. It is sequential because it stores data at every clock cycle, so at every clock cycle the multiplier is right shifted by one bit and the bit value determines if the multiplicand is added or a zero is added. After every addition to the accumulator the result is shifted right by one bit until all the bits of the multiplier is tested. This produces an 8-bit product. In Verilog the multiplier

design consists of an instance of 5 modules: two d flip flops, a mux, an adder, and a shift register producing the product.

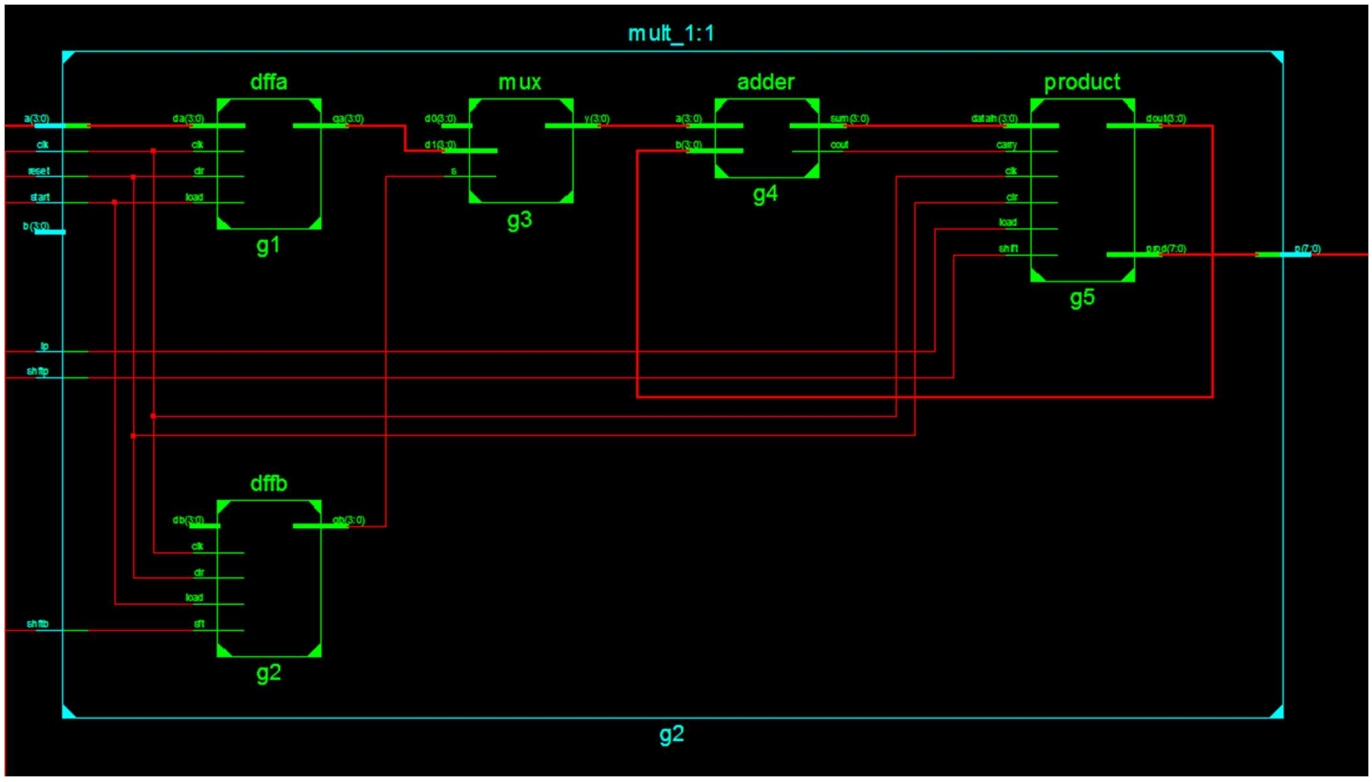

Figure 2-1: Multiplier Design Schematic.

```
1  module mult_1(a,b,reset,start,clk,lp,shftb,shftp,p);
2     input [3:0] a,b;
3     input reset, start, clk, lp, shftb, shftp;
4     output [7:0]p;
5     wire [3:0] a0,b0,y0,s1,data;
6     wire so;
7
8     //Multiplicand A
9     dffa g1(.clk(clk), .clr(reset), .load(start), .da(a), .qa(a0));
10    //multiplier B
11    dffb g2(.clk(clk), .clr(reset), .load(start), .sft(shftb), .db(b), .qb(b0));
12    //mux for 4 bit selection
13    mux g3(.s(b0[0]),.d1(a0),.d0(4'b0000),.y(y0));
14    //add
15    adder g4(.a(y0),.b(data),.cout(so),.sum(s1));
16    //product
17    product g5(.clk(clk),.clr(reset),.carry(so),.datain(s1),.load(lp),.shift(shftp),.dout(data),.prod(p));
18
19
20 endmodule
```

Figure 2-2: Multiplier Design Verilog Description.

```
module mult_1_tb;

   // Inputs
   reg [3:0] a, b;
   reg clr, load, clk, sb, lp, sp;
   // Outputs
   wire [7:0] p;

   // Instantiate the Unit Under Test (UUT)
   mult_1 uut (.a(a), .b(b), .reset(clr), .start(load), .clk(clk), .shftb(sb), .shftp(sp),.lp(lp), .p(p));

   //clock
   initial  clk = 0;

   always   #10 clk = ~ clk;

   initial begin
      // Initialize Inputs
      a = 4'b1011;
      b = 4'b1101;
      clr = 1; load = 0; sb = 0; lp = 0; sp = 0;

      #22 clr = 0; load = 1; sb = 0; lp = 0; sp = 0;
      #20 clr = 0; load = 0; sb = 0; lp = 1; sp = 0;
      #20 clr = 0; load = 0; sb = 0; lp = 0; sp = 1;
      #20 clr = 0; load = 0; sb = 1; lp = 0; sp = 0;

      #20 clr = 0; load = 0; sb = 0; lp = 1; sp = 0;
      #20 clr = 0; load = 0; sb = 0; lp = 0; sp = 1;
      #20 clr = 0; load = 0; sb = 1; lp = 0; sp = 0;

      #20 clr = 0; load = 0; sb = 0; lp = 1; sp = 0;
      #20 clr = 0; load = 0; sb = 0; lp = 0; sp = 1;
      #20 clr = 0; load = 0; sb = 1; lp = 0; sp = 0;

      #20 clr = 0; load = 0; sb = 0; lp = 1; sp = 0;
      #20 clr = 0; load = 0; sb = 0; lp = 0; sp = 1;
      #20 clr = 0; load = 0; sb = 1; lp = 0; sp = 0;

      #20 clr = 0; load = 0; sb = 0; lp = 0; sp = 0;
      // Wait 100 ns for global reset to finish
      #50  $stop;
   end

endmodule
```

Figure 2-3: Multiplier Design Verilog Testbench.

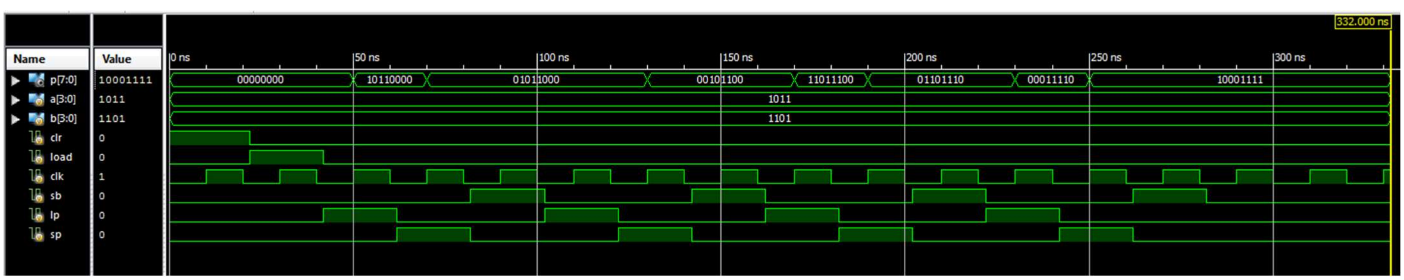

Figure 2-4: Multiplier Testbench Waveform Simulation.

2-1.1. Sequential Multiplier Design Components

D-Flip Flop for Multiplicand A

Description: The purpose of this D Flip Flop is to store the data of the 4-bit multiplicand.

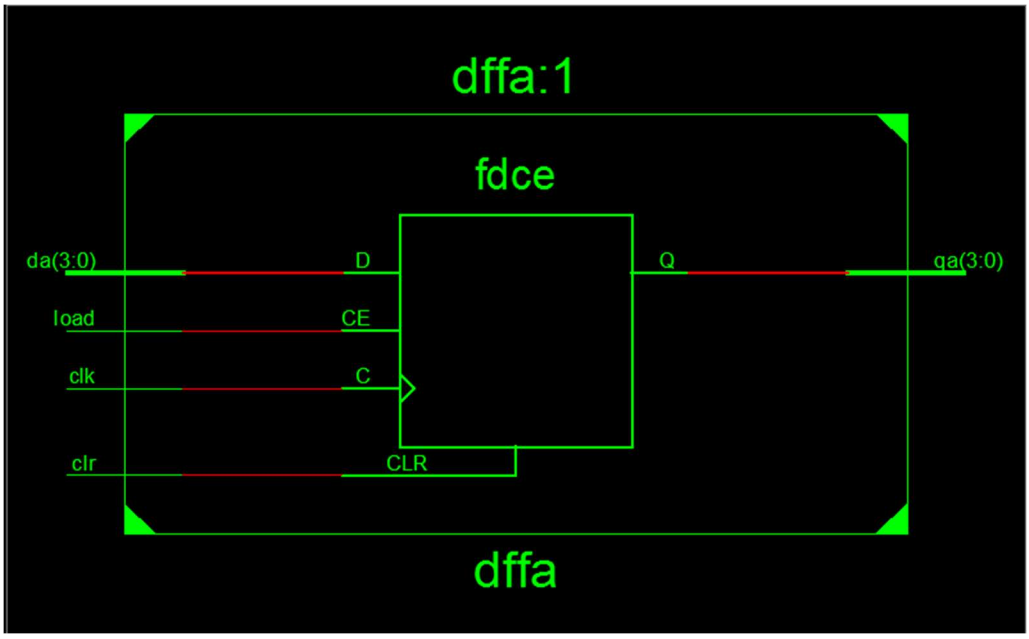

Figure 2-5: D-Flip Flop (A) Hardware Schematic

```
 1  module dffa (clk, clr, load, da, qa);
 2  input        clk, clr, load;
 3  input [3:0]  da;
 4  output [3:0] qa;
 5
 6  reg    [3:0] qa;
 7
 8  always@(posedge clr or  posedge clk)
 9  begin
10      if(clr) qa <= 0;
11      else if (load)
12         qa <= da;
13
14  end
15  endmodule
```

Figure 2-6: D-Flip Flop (A) Verilog Description.

```
1
2  module dffa_tb;
3
4     // Inputs
5     reg clk;
6     reg clr;
7     reg load;
8     reg [3:0] da;
9
10    // Outputs
11    wire [3:0] qa;
12
13    // Instantiate the Unit Under Test (UUT)
14    dffa uut (.clk(clk),.clr(clr),.load(load),.da(da),.qa(qa));
15
16    initial  clk = 0;
17
18    always   #10 clk = ~ clk;
19
20    initial
21    begin
22        clr = 1;  load = 0;
23        da  = 4'b1011;
24
25        #24  clr = 0; load = 1;
26        #20  clr = 0; load = 0;
27
28        #60 $stop;
29    end
30 endmodule
```

Figure 2-7: D-Flip Flop (A) Verilog Testbench.

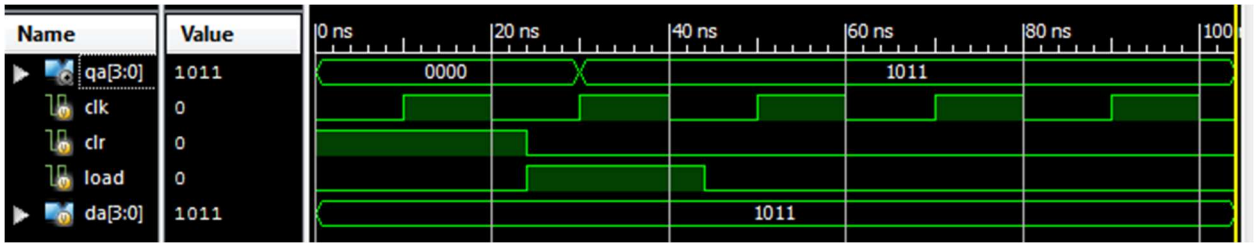

Figure 2-8: D-Flip Flop (A) Waveform Simulation.

D-Flip Flop for Multiplier B

Description: The purpose of this D Flip Flop is to store the data of 4-bit multiplier and to store its value after it is shifted. This is how to make a 4-bit shift register.

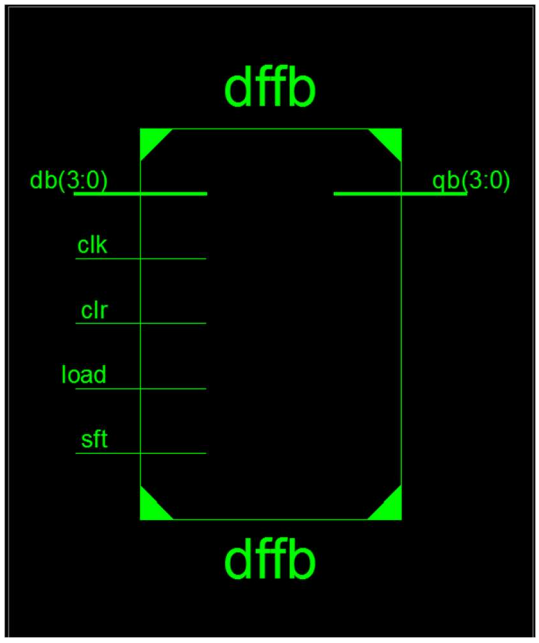

Figure 2-9: D-Flip Flop (B) Hardware Schematic Symbol.

```
 1  module dffb (clk, clr, load, sft, db, qb);
 2  input        clk, clr, load, sft;
 3  input [3:0]  db;
 4  output [3:0] qb;
 5
 6  reg    [3:0] qb;
 7
 8  always@(posedge clr or  posedge clk)
 9  begin
10      if(clr) qb <= 0;
11      else if (load) //tb load=1
12         qb <= db;
13      else if (sft)
14         qb <= { 1'b0,  qb[3:1] };
15
16         // qb[3] <= 1'b0;
17         // qb[2] <= qb[3];
18         // qb[1] <= qb[2];
19         // qb[0] <= qb[1];
20  end
21  endmodule
```

Figure 2-10: D-Flip Flop (B) Verilog Description.

```
1  `timescale 1ns/1ps
2
3  module dffb_tb;
4
5  reg clk, clr, load, sft;
6  reg [3:0] db;
7
8  wire [3:0] qb;
9
10 dffb   uut ( .clk(clk), .clr(clr), .load(load), .sft(sft),.db(db), .qb(qb));
11
12 //  dffb uut ( clk, clr, load, sft, db, qb );
13
14 initial  clk = 0;
15
16 always   #10 clk = ~ clk;
17
18 initial
19 begin
20     clr = 1;  load = 0; sft = 0;
21     db  = 4'b1101;
22
23     #24  clr = 0; load = 1; sft = 0;
24     #20  clr = 0; load = 0; sft = 1;
25
26     #60 $stop;
27 end
28
29 endmodule
```

Figure 2-11: D-Flip Flop (B) Verilog Testbench.

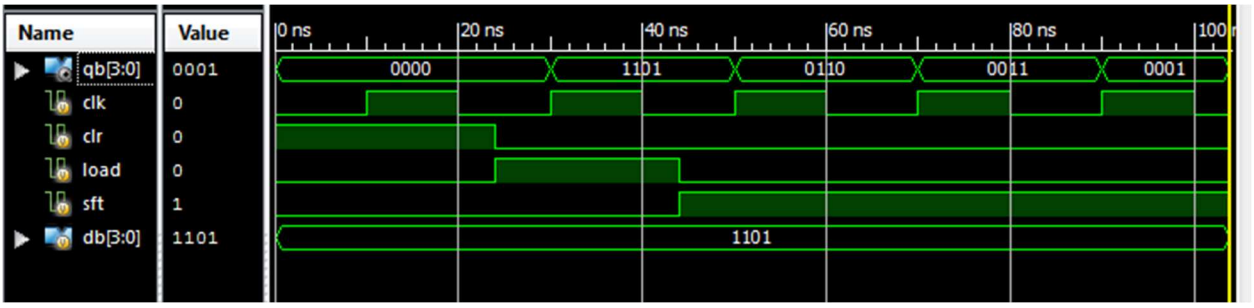

Figure 2-12: D-Flip Flop (B) Waveform Simulation.

2 to 1 Mux

Description: The purpose of the mux is to make a selection depending on the value that is being tested. If the value being tested is a 1 then the mux will select the value of the multiplicand otherwise if the value tested is a 0, then the mux will select “0000”.

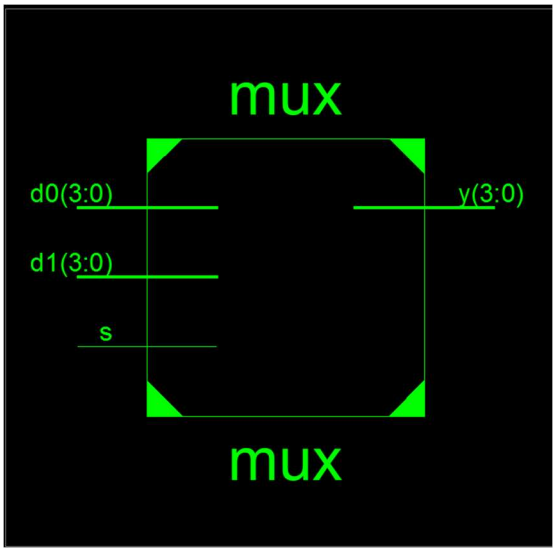

Figure 2-13: 2 to 1 Mux Schematic Symbol

```
 1  module mux ( s, d1, d0, y );
 2  input [3:0]  d1, d0;
 3  input        s;
 4  output [3:0] y;
 5
 6  reg    [3:0] y;
 7
 8  always@( s or d1 or d0 )
 9  begin
10     if (s)
11             y = d1;
12     else
13             y = d0;
14  end
15
16  endmodule
```

Figure 2-14: 2 to 1 Mux Verilog Description.

```
1  module mux_tb;
2
3     // Inputs
4     reg s;
5     reg [3:0] d1;
6     reg [3:0] d0;
7
8     // Outputs
9     wire [3:0] y;
10
11    // Instantiate the Unit Under Test (UUT)
12    mux uut (.s(s), .d1(d1), .d0(d0), .y(y));
13
14    initial begin
15       // Initialize Inputs
16       d0 = 4'b0000; d1 = 4'b1011;
17       s = 0;
18       #10;s = 1;
19       #10;s = 0;
20       #10;s = 1;
21
22       #20; $stop;
23
24    end
25 endmodule
```

Figure 2-15: 2 to 1 Mux Verilog Testbench.

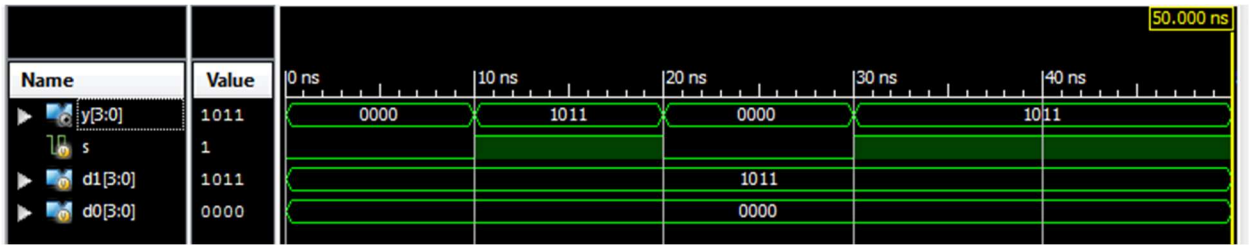

Figure 2-16: 2 to 1 Mux Waveform Simulation.

Adder

Description: The purpose of the adder is to add the value chosen by the 2 to 1 Mux and add it to the accumulator and output the result to the product shift register. This adder is an implementation of the half adder described on page 4. The only difference is that this adder is adding 4 bits instead of 1.

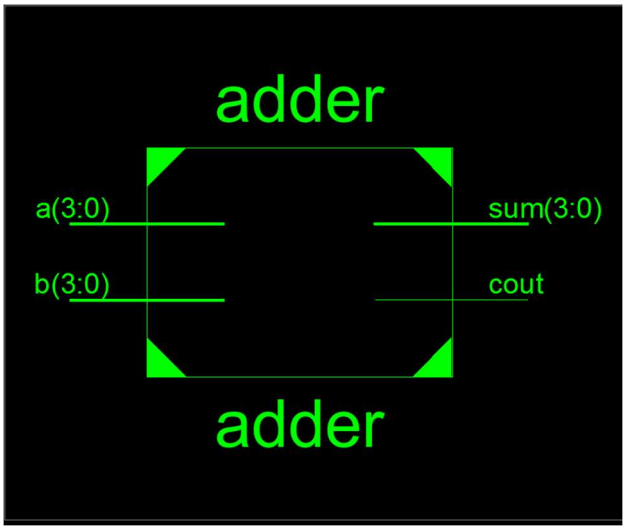

Figure 2-17: Adder Hardware Schematic Symbol.

```
1  module adder (a,b,cout,sum);
2     input [3:0]  a, b;
3     output [3:0] sum;
4     output   cout;
5
6  assign  {cout, sum} = a + b;    //behavior style
7
8  endmodule
```

Figure 2-18: Adder Verilog Description.

Product Shift Register

Description: The purpose of the product shift register is to store the first 4-bits of the partial product and shift it right every clock cycle. It is right shifted by one bit into another 4-bit shift register. The carry bit from the adder is shifted into the most significant bit of the first 4-bit shift register. The result is the multiplication result by multiplying Multiplicand (A) and Multiplier (B).

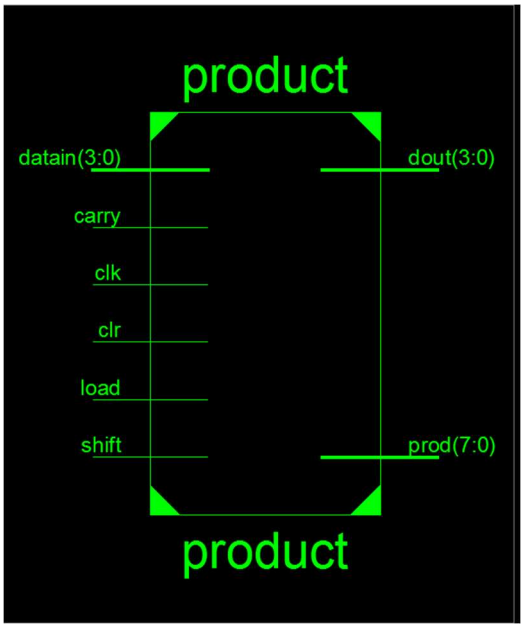

Figure 2-19: Product Schematic Symbol.

```
1  module product(clk,clr,carry,datain,load,shift,dout,prod);
2
3  input clk,clr,carry,load,shift;
4  input [3:0]datain;
5  output [3:0]dout;
6  output [7:0]prod;
7  reg [3:0]dout;
8  reg [7:0]prod;
9
10
11 always@(posedge clr or  posedge clk)
12 begin
13
14          if(clr)begin
15             prod <= 8'b00000000;
16             dout<= 4'b0000;
17          end
18          else if (load)begin
19             prod[7:4]<=datain;
20          end
21          else if (shift)
22          begin
23             //prod[7:0]<={carry,prod[7:1]};
24             prod[7] <= carry;
25             prod[6:3] <= datain;
26             prod[2:0] <= prod[3:1];
27             dout <= { carry, datain[3:1] };
28          end
29
30 end
31 endmodule
```

Figure 2-20: Product Shift Register Verilog Description.

2-2. Finite State Machine Design

Description: The purpose of the finite state machine is to control the multiplier logic circuit. The Finite State Machine makes a selection and outputs the signal to the Multiplier Design.

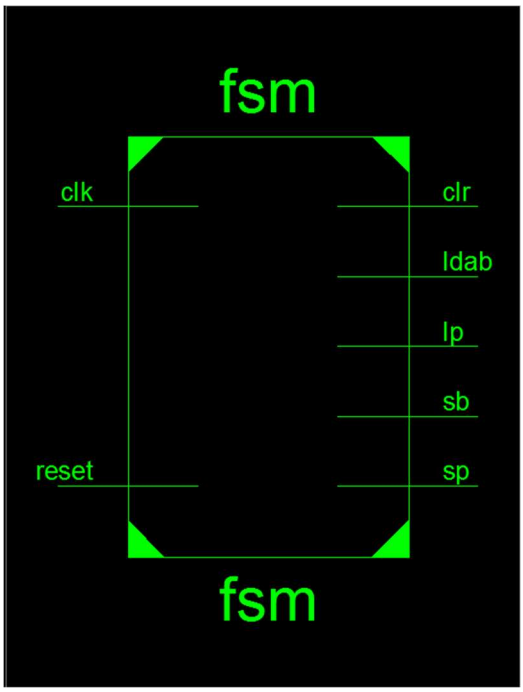

Figure 2-21: Finite State Machine Schematic.

```
1  module fsm( clk, reset, clr, ldab, sb, lp, sp );
2  input reset,clk;
3  output clr, ldab, sb, lp, sp;
4
5  parameter s_clr = 3'b000, s_ld_ab = 3'b001, s_ldp =3'b010, s_shftp =3'b011, s_shftb =3'b100;
6  reg [2:0]   cs, ns;
7  reg clr,  ldab, sb, lp, sp;
8
9  always@(posedge clk or posedge reset)
10 begin
11       if(reset)
12          cs <= s_clr;      //first state to clear all registers
13       else
14          cs <=  ns;
15 end
16
17  // next state and output block
18  always@(cs)
19  begin
20      case(cs)
21        s_clr:   begin
22                   ns  =  s_ld_ab;
23                   clr = 1;  ldab=0; sb=0; lp=0; sp=0;
24                 end
25
26       s_ld_ab:  begin
27                    ns = s_ldp;
28                   clr = 0;  ldab=1; sb=0; lp=0; sp=0;
29                 end
30         s_ldp:  begin
31                   ns = s_shftp;
32                   clr =0; ldab = 0; sb =0; lp =1; sp =0;
33                 end
34
35       s_shftp:  begin
36                   ns = s_shftb;
37                   clr = 0; ldab = 0; sb = 0; lp = 0; sp = 1;
38                 end
39
40       s_shftb:  begin
41                  ns = s_ldp;
42                  clr = 0; ldab = 0; sb = 1; lp = 0; sp = 0;
43                 end
44
45
46        default:  begin
47                    ns = s_clr;
48                    clr = 1;  ldab=0; sb=0; lp=0; sp=0;
49                  end
50      endcase
51 end
52 endmodule
```

Figure 2-22: FSM Verilog Description Code.

```
1  `timescale 1ns / 1ps
2
3  module fsm_tb;
4
5     // Inputs
6     reg clk;
7     reg reset;
8
9     // Outputs
10    wire clr;
11    wire ldab;
12    wire sb;
13    wire lp;
14    wire sp;
15
16    // Instantiate the Unit Under Test (UUT)
17    fsm uut (.clk(clk), .reset(reset), .clr(clr), .ldab(ldab),
18          .sb(sb), .lp(lp), .sp(sp));
19
20    initial  clk = 0;
21
22    always   #10 clk = ~ clk;
23
24    initial begin
25       // Initialize Inputs
26       reset = 1;
27
28       #20 reset =0;
29       #300 $stop;
30
31    end
32
33 endmodule
```

Figure 2-23: FSM Verilog Testbench Code.

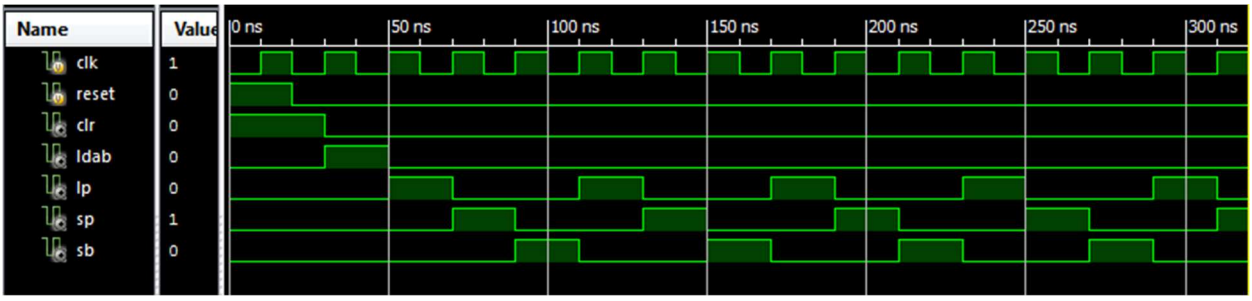

Figure 2-24: FSM Testbench Waveform Simulation.

2-3. Top Level Design

Description: The top-level design is the top module of the hierarchical design and it consists of an instance of the Multiplier and an instance of the finite state machine.

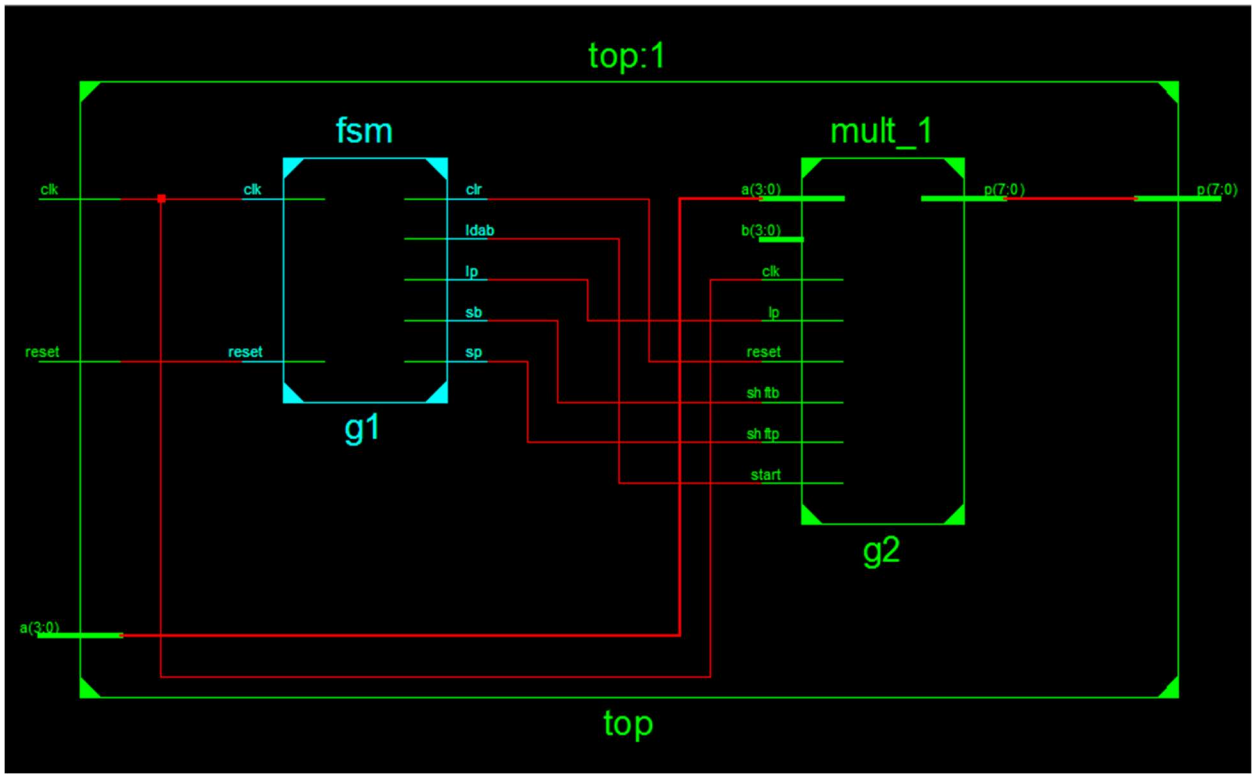

Figure 2-25: Top Level Schematic

```
1  module top(clk, reset, a, p);
2  input clk, reset;
3  input  [3:0]   a;
4  output [7:0]   p;      //8-bit final product output
5  wire  [3:0]    b;
6  wire  clr,load,sb,lp,sp;              //internal wires
7  assign   b = 4'b1101;
8
9     fsm g1(.clk(clk), .reset(reset), .clr(clr), .ldab(load), .sb(sb), .lp(lp), .sp(sp));
10    mult_1 g2(.a(a),.b(b),.reset(clr),.start(load),.clk(clk),.lp(lp),.shftb(sb),.shftp(sp),.p(p));
11
12 endmodule
```

Figure 2-26: Top Level Verilog Description Code.

```
1  `timescale 1ns / 1ps
2
3  module top_tb;
4
5     // Inputs
6     reg clk;
7     reg reset;
8     reg [3:0] a;
9
10    // Outputs
11    wire [7:0] p;
12
13    // Instantiate the Unit Under Test (UUT)
14    top uut (
15       .clk(clk),
16       .reset(reset),
17       .a(a),
18       .p(p)
19    );
20
21    initial  clk = 0;
22
23    always   #10 clk = ~ clk;
24
25    initial begin
26       // Initialize Inputs
27       reset = 1;
28       a = 4'b1011;
29       #20 reset=0;
30       // Wait 100 ns for global reset to finish
31       #300 $stop;
32
33    end
34
35 endmodule
```

Figure 2-27: Top Level Verilog Testbench Code.

Figure 2-28: Top Level Testbench Waveform Simulation

Part 3: Display “CPE 166” on LCD

Description: The purpose is to learn the basic LCD capabilities of the Spartan3E FPGA board. The LCD supports 2-line x 16-character display. It has three control signals, including LCD_RW, LCD_RS, and LCD_E as shown in figure 3-1. The LCD also has three types of internal memory storage including Display Data RAM, Character Generation ROM, and Character Generation RAM. To display data the DD RAM stores the character code for each character. The codes used for each character is shown in figure 3-2.

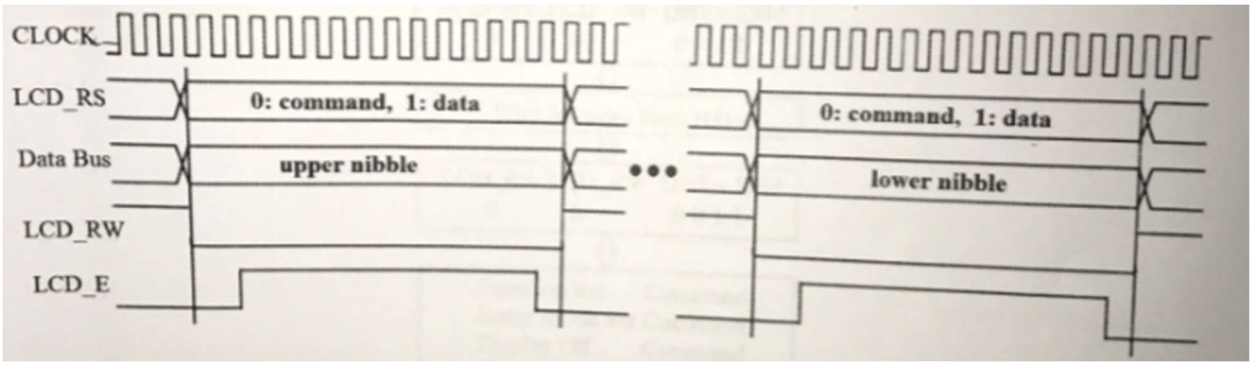

Figure 3-1: LCD interface signals.

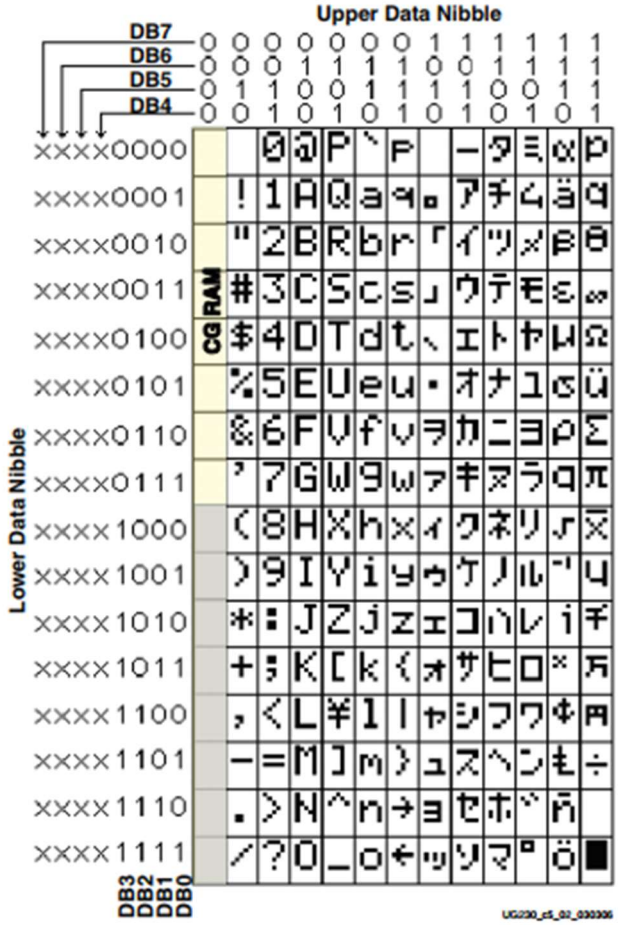

Figure 3-2: LCD Character Set from the Xilinx User Guide.

```
module mylcd(clk, sf_ce0, lcd_rs, lcd_rw, lcd_e, lcd_4, lcd_5, lcd_6, lcd_7);
  parameter             k = 18;
  input                   clk;        // synthesis attribute PERIOD clk "50 MHz"
  reg   [k+8-1:0]       count;
  output              sf_ce0;     // high for full LCD access
  reg                   sf_ce0;

  output              lcd_e, lcd_rs, lcd_rw, lcd_7, lcd_6, lcd_5, lcd_4;
  reg                   lcd_e, lcd_rs, lcd_rw, lcd_7, lcd_6, lcd_5, lcd_4;

  reg                   lcd_busy;
  reg                   lcd_stb;
  reg       [5:0]     lcd_code;
  reg       [6:0]     lcd_stuff;

  always @ (posedge clk) begin
    count      <= count + 1;
    lcd_busy <= 1'b1;
    sf_ce0     <= 1;                 // StrataFlash Chip Enable
    case (count[k+7:k+2])
       0: lcd_code <= 6'h03;        // power-on initialization
       1: lcd_code <= 6'h03;
       2: lcd_code <= 6'h03;
       3: lcd_code <= 6'h02;
       4: lcd_code <= 6'h02;        // function set
       5: lcd_code <= 6'h08;
       6: lcd_code <= 6'h00;        // entry mode set
       7: lcd_code <= 6'h06;
       8: lcd_code <= 6'h00;        // display on/off control
       9: lcd_code <= 6'h0C;
      10: lcd_code <= 6'h00;        // display clear
      11: lcd_code <= 6'h01;
      12: lcd_code <= 6'h24;        // C
      13: lcd_code <= 6'h23;
      14: lcd_code <= 6'h25;        // P
      15: lcd_code <= 6'h20;
      16: lcd_code <= 6'h24;        // E
      17: lcd_code <= 6'h25;
      18: lcd_code <= 6'h22;        //
      19: lcd_code <= 6'h20;
      20: lcd_code <= 6'h23;        // 1
      21: lcd_code <= 6'h21;
      22: lcd_code <= 6'h23;        // 6
      23: lcd_code <= 6'h26;
      24: lcd_code <= 6'h23;        // 6
      25: lcd_code <= 6'h26;
      default: lcd_code <= 6'h10;
    endcase
    lcd_stb <= ^count[k+1:k+0] & ~lcd_rw & lcd_busy;
    lcd_stuff <= {lcd_stb,lcd_code};
    {lcd_e,lcd_rs,lcd_rw,lcd_7,lcd_6,lcd_5,lcd_4} <= lcd_stuff;
  end
endmodule
```

Figure 3-3: LCD Display Verilog Description Code.

```
1
2  NET "sf_ce0" LOC = "D16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
3  NET "clk" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
4
5
6  NET "lcd_e" LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
7  NET "lcd_rs" LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
8  NET "lcd_rw" LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
9
10 NET "lcd_4" LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
11 NET "lcd_5" LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
12 NET "lcd_6" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
13 NET "lcd_7" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW = SLOW ;
```

Figure 3-4: LCD UCF file Code.

Figure 3-5: LCD Display on FPGA Board.

Conclusion: In this lab we learned how to design hardware using hierarchical design in combinational and sequential circuits. This includes having a knowledge of the basic syntax rules used in Verilog and how to design modules to describe the behavior of a piece of hardware. We are then able to use the instances of different modules (pieces of hardware) to describe an overall hardware that contains the many components, this is what we call the top module of the hierarchical design. We are able to test the many components using the testbench in Verilog and simulate the results to test the hardware that it will produce the correct logic

behavior. We are also introduced to the Spartan3E board which we can program our designs to the board and verify that the result is what we intended in the testbench. Knowledge of the LCD component of the Spartan3E is learned and how to implement the design to display the different characters using Figure 3-2.