Gabriel Emerson
ELEC 4200 – Lab 6
09/30/21

Lab 6: Behavioral Modeling, Timing Constraints, Architectural Wizard, IP Catalog, Counters

Goals:

The goal for this lab is to become familiar with Behavioral modeling with Timing Constraints, using the Architectural Wizard (in the IP Catalog) and counters. This Lab revolves around the fact that you will sometimes need a different clock cycle to get the program working exactly how you like. We also developed a few Testbenches to test and verify the functionality of the module. This is the first lab we have used the IP Catalog with the Architectural Wizard.

Design Process:

Task 1 is a call back to an earlier Lab where we created Multiplexers. In this task we are to create a 4-1 Mux using only if/else-if statements. We can do this easily be reverting to our old lab and finding what we did then. The output should be verified using a created test bench and also verified visually by uploading it to the board. The output should match that of the truth table shown in Figure 1.

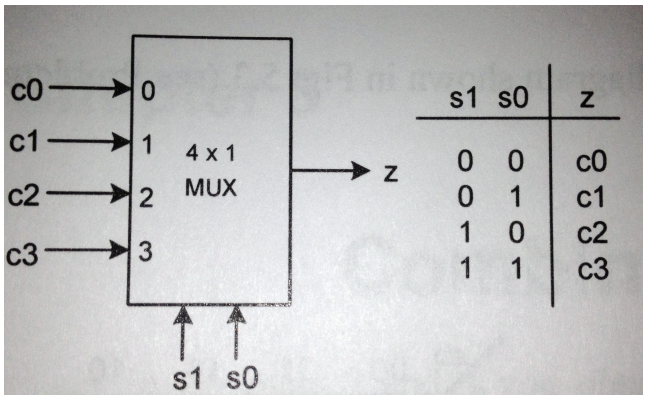

Figure 1

Task 2 was also the same as an earlier lab task, in which we created a gray code generator. This time we will only use a case statement. There is also an enable bit that, when enabled, will turn on the generator, but with this bit off nothing will show at the output. We can also review the Project Summary Report to verify this module. We will also upload this to the board to visually verify the functionality of the program. The Gray code generator should follow the output of the truth table shown in Figure 2.

Natural-binary code				Gray code			
B3	B2	B1	B0	G3	G2	G1	G0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Figure 2

In task 3 we are going to design a one-second pulse generator. This is done by using the Architectural Wizard, located in the IP Catalog. The design we will use will take the clock given to us on the board, split it one time, and then split it again to get the desired output of a 1Hz clock. The instructions of how to all of this is found in the Lab writeup. Then we will output the signal to an LED and verify the timing of the clock blinking.

In task 4 we also use the clock splitting method but instead of just sending out the pulse, we use it in the modification of a previous task of the binary to bcd converter. After setting this up we can just upload it to the board to verify the functionality of the program.

Task 5 begins to get more complicated by using the IP catalog to create a counter in the Vivado software. We then use it with the bcd converter to show the values of the current counter spot. We then also make it count at a specific time using the clock splitting method to get exactly 1 Hz in the output. Then we will simply upload the program to the board to verify the numbers counting up at exactly 1 Hz.

Task 6 is slightly different and creates a carry look ahead adder using gate-level modeling (similar to how we did in a previous lab). We will instead create this by using parameter statements shown in the Lab writeup. This module should specifically use instance parameter value assignment statement method. We will then use a testbench to verify our methods of the parameter statements.

Task 7 is the counter part of Task 6. Instead of using the parameter statements, we will use the defparam statements that are discussed in the lab writeup. The difference of the parameter and defparam statements, is the defparam is commonly used in RTL designs due to its ability to override the default values. If we wanted to simply test our idea circuit, it would be easier and more efficient to just use the parameter statements.

Detailed Design:

Task 1 was designing the 4 to 1 mux like we have done before in the Lab. This is a pretty simple task especially when using behavioral modeling. The code for implementing this Mux is shown in Figure 3. We also had to design a testbench for testing our Mux which can be seen in Figure 4.

```
module Lab6_1(in, sel, out);
    input [3:0] in;
    input [1:0] sel;
    output reg out;
    always @ (in or sel)
        begin
            if(sel == 2'b00) out = in[0];
            else if(sel == 2'b01) out = in[1];
            else if(sel == 2'b10) out = in[2];
            else out = in[3];
    end
endmodule
```

Figure 3

```
module Lab6_1Test();
    reg [3:0] in;
    reg [1:0]sel;
    wire out;
    Lab6_1 UUT(.in(in), .sel(sel), .out(out));
    initial
       begin
        in = 0;
        #5 sel = 2'b00;
        #5 in = 1;
        #5 sel = 2'b01;
        #5 in = 7;
        #5 sel = 2'b10;
        #5 in = 0;
        #5 sel = 2'b11;
        #5 in = 15;
        #10 $finish;
    end
endmodule
```

Figure 4

Task 2 was also a call back to a task we have done in a previous lab. The design of the gray code generator using case statements are simply creating a case for each input and what the output should be. We also have to make sure to check the input bit that enables the gray code generator. The code for this task is shown in Figure 5.

```
module Lab6_2(bcd,enin,enout,out);
    input enin;
    input [3:0] bcd;
    output reg enout;
    output reg [3:0] out;

    always @ (enin or bcd)
        if (enin == 1'b1)
            begin
            enout = 1'b0;
            case(bcd)
                4'b0000: out = 4'b0000; //00
                4'b0001: out = 4'b0001; //01
                4'b0010: out = 4'b0011; //02
                4'b0011: out = 4'b0010; //03
                4'b0100: out = 4'b0110; //04
                4'b0101: out = 4'b0111; //05
                4'b0110: out = 4'b0101; //06
                4'b0111: out = 4'b0100; //07
                4'b1000: out = 4'b1100; //08
                4'b1001: out = 4'b1101; //09
                default: begin
                     out = 4'b1111;
                     enout = 1'b1;
                     end
            endcase
        end
        else
             begin out = 4'b1111; enout = 1'b1;
        end
endmodule
```

Figure 5

Task 3 was the beginning of splitting the clock using the Architectural Wizard and another module given to us in the lab writeup. We are to simply take the clock pulse on the board and convert it to a 1 Hz pulse, then output it to an LED to verify. This is done by using the Wizard to split the 100MHz in half to 50MHz, then we use the code given to split the output again down to 1 Hz. The code for this task is shown in Figure 6.

```
module Lab6_3(input D, input clk, input rst, output reg Q, output locked);
    wire clk5;
    wire clk_out;
    always @ (posedge clk or posedge rst)
    begin
        if (rst == 1)
            Q <= 1'b0;
        else
        //begin
            //if (D)
                Q = clk_out;
            //else
                //Q = 0;
        //end
    end
    clk_wiz_0 UUT1 ( .clk_in1(clk), .clk_out1(clk5), .locked(locked));
    divider DUT1 (.clock_in(clk5), .clock_out(clk_out));

endmodule
```

Figure 6

Task 4 creates a bcd converter and outputs it at the pulse split down to 1 Hz. We can use the clock splitting method used to get back down to the desired frequency, then use the code from our old bcd converter and modify them to get the desired program. I was not quite able to get this task working correctly, but the code that was still in progress is shown in Figure 7.

```
module Lab2_4to2(v,z,m,seg,set0,clk);
    input [3:0]v;
    output reg [3:0] m;
    output reg z;
    output reg [6:0] seg;
    output [7:0] set0;
    input clk;
    wire clk_out;
    assign set0 = 8'b11111100;

    clk_wiz_0 UUT1 ( .clk_in1(clk), .clk_out1(clk5), .locked(locked));
    divider DUT1 (.clock_in(clk5), .clock_out(clk_out));

    always @(v)
    begin
        if(v < 4'b1010)
            {z,m} = {1'b0,v};
        else
        begin
            z=1'b1;
            m=v-4'b1010;
        end
        case (m) //case statement
            0 : seg = 7'b0000001;
            1 : seg = 7'b1001111;
            2 : seg = 7'b0010010;
            3 : seg = 7'b0000110;
            4 : seg = 7'b1001100;
            5 : seg = 7'b0100100;
            6 : seg = 7'b0100000;
            7 : seg = 7'b0001111;
            8 : seg = 7'b0000000;
            9 : seg = 7'b0000100;
            //switch off 7 segment character if bcd digit != decimal number.
            default : seg = 7'b1111111;
        endcase
    end
endmodule
```

Figure 7

Task 5 was bringing together multiple different aspects of what we have worked on so far in the lab. In this task we use a Counter from the IP Catalog, and also our clock splitting method, with our bcd converter, to get a counter that counts up on the bcd display at exactly 1 Hz. This was a difficult task that I was not able to get completely done. However, the code that was still in progress is shown in Figure 8.

```
module Lab6_5(clk, ce, reset, count);
    input clk, ce, reset;
    output [7:0]count;
    wire [3:0] MSB,LSB;
    wire threshold1, threshold2;
    wire counter1_rst, counter2_rst;

    dec_binary_counter C1(.CLK(clk), .CE(ce), .SCLR(counter1_rst), .THRESH0(threshold1), .Q(LSB));
    dec_binary_counter C2(.CLK(clk), .CE(threshold1), .SCLR(counter2_rst), .THRESH0(threshold2), .Q(MSB));

    assign counter1_rst = reset | threshold1;
    assign counter2_rst = reset | threshold2;
    assign count = {MSB, LSB};
endmodule


module dff(D, clk, rst, Q);
    input D, clk, rst;
    output reg Q;

    always @ (posedge clk or posedge rst)
    begin
        if (rst == 1)
            Q <= 1'b0;
        else
            Q <= D;
    end
endmodule


module clk_divider(clk, reset, clk_out);...
```

Figure 8

Task 6 was going back to the design on the carry look ahead adder but now we will use the parameter statements of Verilog. These make the design of our gate-level adder much easier to read and modify. We also were not supposed to upload this to the board, but instead just design and test using our own testbench. The code used for this task is shown in Figure 9, and the testbench used to test this is shown in Figure 10.

```
module Lab6_6(a, b, cin, s, p, g);
    input a,b,cin;
    output s,p,g;
    wire w1;
    parameter AND_DELAY = 2, OR_DELAY = 2, XOR_DELAY = 2;

    xor #(XOR_DELAY) (w1,a,b);
    xor #(XOR_DELAY) (s,w1,cin);
    or #(OR_DELAY) (p,a,b);
    and #(AND_DELAY) (g,a,b);
endmodule


module cla_logic(cin, p, g, cout);
    input cin;
    input [3:0]p,g;
    output [3:0]cout;
    parameter AND_DELAY = 2, OR_DELAY = 2;
    wire [3:0]c;

    and #(AND_DELAY) (c[0],p[0],cin);
    or #(OR_DELAY) (cout[0],g[0],c[0]);
    and #(AND_DELAY) (c[1],p[1],c[0]);
    or #(OR_DELAY) (cout[1],g[1],c[1]);
    and #(AND_DELAY) (c[2],p[2],c[1]);
    or #(OR_DELAY) (cout[2],g[2],c[2]);
    and #(AND_DELAY) (c[3],p[3],c[2]);
    or #(OR_DELAY) (cout[3],g[3],c[3]);
endmodule


module Lab6_1_1(a, b, cin, s, cout);
    input [3:0]a, b;
    input cin;
    output [3:0]s;
    output cout;
    wire [3:0]p, g;
    wire [3:0]cla_out;

    Lab6_6 #(3,3,4) FA0(a[0], b[0], cin, s[0], p[0], g[0]);
    Lab6_6 #(3,3,4) FA1(a[1], b[1], cla_out[0], s[1], p[1], g[1]);
    Lab6_6 #(3,3,4) FA2(a[2], b[2], cla_out[1], s[2], p[2], g[2]);
    Lab6_6 #(3,3,4) FA3(a[3], b[3], cla_out[2], s[3], p[3], g[3]);

    cla_logic #(3,3) CLA(cin, p, g, cla_out);
    buf(cout, cla_out[3]);
endmodule
```

Figure 9

```
module Lab6_6Test();
    reg [3:0]a_in;
    reg [3:0]b_in;
    reg cin;
    wire [3:0]s_out;
    wire cout;
    integer i,k;

    Lab6_1_1 DUT(.a(a_in), .b(b_in), .cin(cin), .s(s_out), .cout(cout));

    initial
    begin
        cin = 0;
        a_in = 0;
        b_in = 0;
        for(i = 0; i < 2; i=i+1)
        begin
            if(i == 1)
            cin = 1;
            for(k = 0; k < 15; k=k+1)
            begin
                #10 a_in = k; b_in = k+1;
            end
        end
    end
endmodule
```

Figure 10

Task 7 was very similar to task 6, but instead of using the parameter statements, we used the override method of defparam statements. There are very little changes for this task but the difference of the two is how the parameters are assigned. This method is often used in the design of RTL circuits. The code and testbench for this task are shown in Figures 11 and 12 respectively.

```
module Lab6_7(a, b, cin, s, p, g);
    input a,b,cin;
    output s,p,g;
    wire w1;
    parameter AND_DELAY = 2, OR_DELAY = 2, XOR_DELAY = 2;

    xor #(XOR_DELAY) (w1,a,b);
    xor #(XOR_DELAY) (s,w1,cin);
    or #(OR_DELAY) (p,a,b);
    and #(AND_DELAY) (g,a,b);
endmodule


module cla_logic(cin, p, g, cout);
    input cin;
    input [3:0]p,g;
    output [3:0]cout;
    parameter AND_DELAY = 2, OR_DELAY = 2;
    wire [3:0]c;

    and #(AND_DELAY) (c[0],p[0],cin);
    or #(OR_DELAY) (cout[0],g[0],c[0]);
    and #(AND_DELAY) (c[1],p[1],c[0]);
    or #(OR_DELAY) (cout[1],g[1],c[1]);
    and #(AND_DELAY) (c[2],p[2],c[1]);
    or #(OR_DELAY) (cout[2],g[2],c[2]);
    and #(AND_DELAY) (c[3],p[3],c[2]);
    or #(OR_DELAY) (cout[3],g[3],c[3]);
endmodule


module Lab6_7_1(a, b, cin, s, cout);
    input [3:0]a, b;
    input cin;
    output [3:0]s;
    output cout;
    wire [3:0]p, g;
    wire [3:0]cla_out;

    Lab6_7 FA0(a[0], b[0], cin, s[0], p[0], g[0]);
    Lab6_7 FA1(a[1], b[1], cla_out[0], s[1], p[1], g[1]);
    Lab6_7 FA2(a[2], b[2], cla_out[1], s[2], p[2], g[2]);
    Lab6_7 FA3(a[3], b[3], cla_out[2], s[3], p[3], g[3]);
    cla_logic CLA(cin, p, g, cla_out);
    buf(cout, cla_out[3]);
endmodule
```

Figure 11

```
module Lab6_7Test();
    reg [3:0]a_in;
    reg [3:0]b_in;
    reg cin;
    wire [3:0]s_out;
    wire cout;
    integer i,k;

    defparam DUT.FA0.AND_DELAY=3, DUT.FA0.OR_DELAY=3, DUT.FA0.XOR_DELAY=4;
    defparam DUT.FA1.AND_DELAY=3, DUT.FA1.OR_DELAY=3, DUT.FA1.XOR_DELAY=4;
    defparam DUT.FA2.AND_DELAY=3, DUT.FA2.OR_DELAY=3, DUT.FA2.XOR_DELAY=4;
    defparam DUT.FA3.AND_DELAY=3, DUT.FA3.OR_DELAY=3, DUT.FA3.XOR_DELAY=4;
    defparam DUT.CLA.AND_DELAY=3, DUT.CLA.OR_DELAY=3;

    Lab6_7_1 DUT(.a(a_in), .b(b_in), .cin(cin), .s(s_out), .cout(cout));

    initial
    begin
        cin = 0;
        a_in = 0;
        b_in = 0;
        for(i = 0; i < 2; i=i+1)
        begin
            if(i == 1)
            cin = 1;
            for(k = 0; k < 15; k=k+1)
            begin
                #10 a_in = k; b_in = k+1;
            end
        end
    end
endmodule
```

Figure 12

Verification:

Tasks 1,6, and 7, are all verified using testbenches designed in the lab. Tasks 6 & 7 are only tested in this way, however tasks 1-5 are all uploaded to the board to be verified visually. For tasks 1 and 2 we can simply view the output on the board and compare to the truth tables of the circuits being drawn. Then tasks 3-5 are all having to view them on the board.

Tasks 4 and 5 were not completed in this lab due to time constraints. It took me awhile to get the clock splitter down, but after figuring it out the rest of the lab did not seem to be that difficult.

The output simulations for tasks 1,6, and 7, are all shown in the Figures below. Task 1 is Figure 13, Task 6 in Figure 14, and Task 7 in Figure 15.

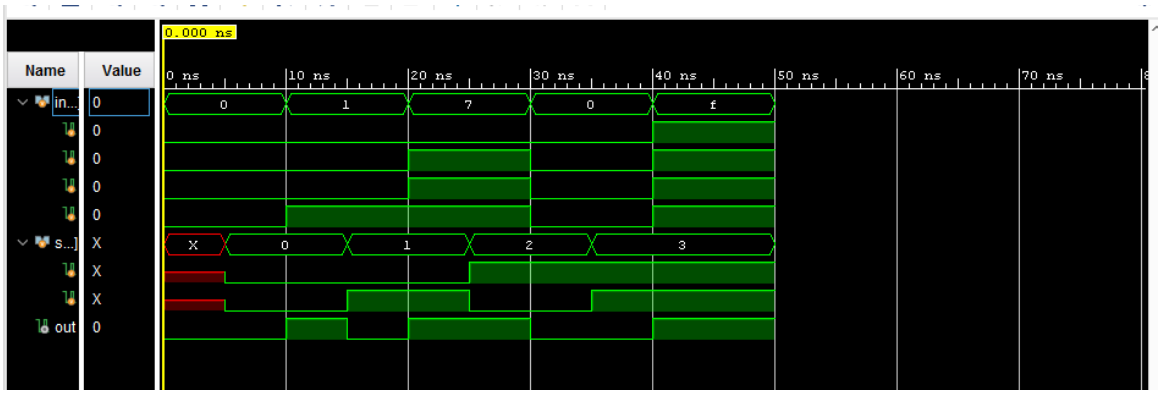

Figure 13

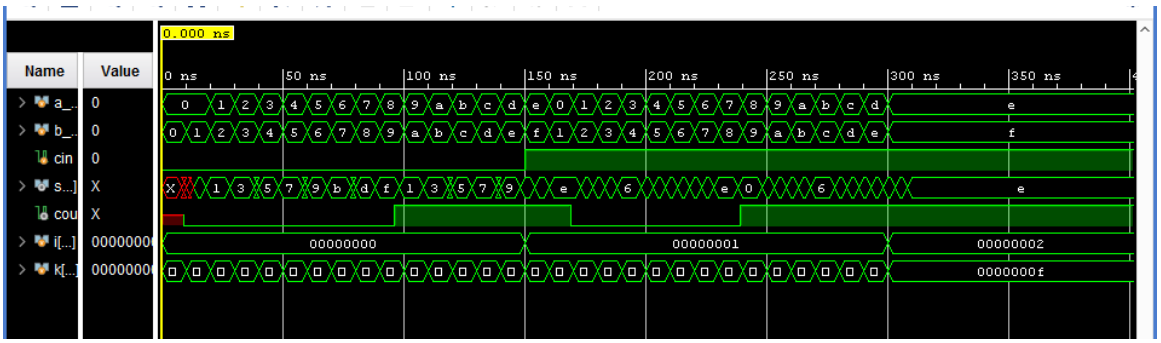

Figure 14

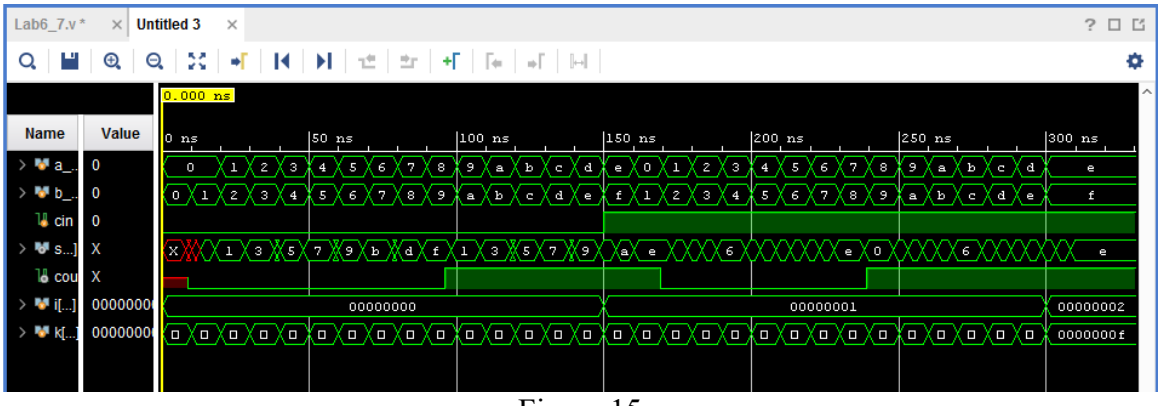

Figure 15

Conclusion:

What I learned in this lab is how to use the IP Catalog for Counter, and the Architectural Wizard. This came in handy when trying to split the clock for most of these tasks. Using the Wizard will surely come in handy when doing anything that requires a clock cycle that isn't at the standard 100 MHz. I will most likely need it for the final project coming up real soon. I had a problem when I got to task 3 and was not sure of how to get the clock splitter working. However, after working on this for some time, I was able to finally get the idea of it down. This means since I got caught up in tasks, I was unable to complete tasks 4 and 5, but got close on task 4. I do understand the general idea of task 5, but was unable to really get into it. Next time I will be able to get through it faster since I now understand the workings of the Wizard, and splitting the clock pulse.