ELEC-4200

Digital System Design

FROM: Jacob Howard

TO: Prof. Ujjwal Guin

DUE DATE: 2/25/21

Lab 6

Introduction

The goal of this lab was to familiarize ourselves with the Architectural Wizard to configure clocking resources, use the IP Catalog tool to configure and use counters and memories, and design counters using the IP Catalog.

Task 1

In Task 1, we were asked to write a behavioral model to design a 1-bitwide4-to-1 mux using the if-else-if statement and develop a testbench to verify the design. The simulation can be seen in *Figure 1*, the code can be seen in *Code 1* and the simulation code can be seen in *Testbench 1* below. *Most designs were verified to work by TA*

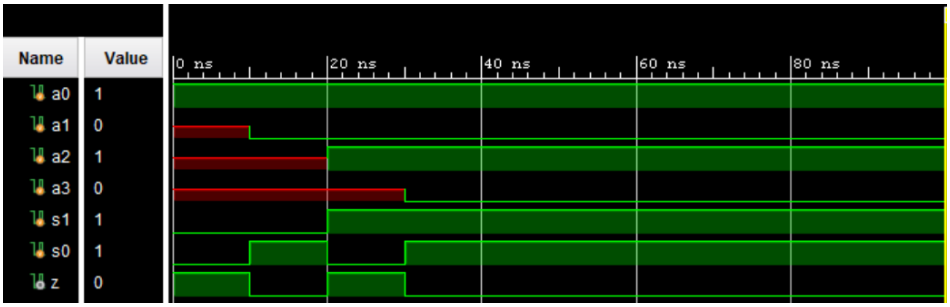

Figure 1

```
module Task1(
    input a0,
    input a1,
    input a2,
    input a3,
    input s0,
    input s1,
    output reg z
    );

    //4-1 mux using if else statements
    always @ (a0 or a1 or a2 or a3 or s0 or s1)

    begin
    if (s1 == 0 & s0 == 0)
    z = a0;
    else if ((s1 == 0) & (s0 == 1))
    z = a1;
    else if ((s1 == 1) & (s0 == 0))
    z = a2;
    else if ((s1 ==1) & (s0 == 1))
    z = a3;
    end

endmodule
```

Code 1

```
initial begin
    #100 $finish; //runs simulation 100 times
    end
initial begin
s0 = 0;
s1 = 0;
a0 = 1;
#10
s1 = 0;
s0 = 1;
a1 = 0;
#10
s1 = 1;
s0 = 0;
a2 = 1;
#10
s1 = 1;
s0 = 1;
a3 = 0;
end
endmodule
```

Testbench 1

Task 2

In Task 2, we were asked to design a gray code generator using the case statement. This was fairly simple using case statements. The code can be seen in *Code 2*.

```
always @ (a or enable)
   begin
   enableLED = 0;
   if (enable == 1)
   begin
   enableLED = 0;
   case(a)
      4'b0000 : begin z = 4'b0000; end
      4'b0001 : begin z = 4'b0001; end
      4'b0010 : begin z = 4'b0011; end
      4'b0011 : begin z = 4'b0010; end
      4'b0100 : begin z = 4'b0110; end
      4'b0101 : begin z = 4'b1110; end
      4'b0110 : begin z = 4'b1010; end
      4'b0111 : begin z = 4'b1011; end
      4'b1000 : begin z = 4'b1011; end
      4'b1001 : begin z = 4'b1001; end
      default : begin z = 4'b1111; end
   endcase
   end
   else
   z = 4'b1111;
   enableLED = 1;
   end
endmodule
```

Code 2

Task 3

In Task 3, we were asked to design a one-second pulse generator using the clocking wizard to generate a 5 MHz clock. This was our first time using the clocking wizard, so it was confusing at first to set it up correctly. I did figure out the code and it can be seen in *Code 3* below.

```
    clk_wiz_0 inst(.clk_in1(clk_in),
.clk_out1(clk_out), .locked(lock));
.enable(enable), .Q(Q));

    integer k = 0;

    always @ (posedge clk_out or posedge reset)
        begin
            if(reset)
                Q=0;
            else
             if (enable)
              begin
              if (k ==2500000)
              begin
               Q <= ~Q;
               k = 0;
               end
               else
               k = k +1;
               end
        end
endmodule
```

Code 2

Task 4

In Task, we were asked to modify the design of task 2 in lab 2 to display the 4-bit binary inputs converted to BCD values on two 7-segment displays. The display would show the same number on 2 7-segment displays by using the clock to refresh each display. The design can be seen in *Code 4* below.

```
clk_wiz_0 inst(.clk_in1(clk_in), .clk_out1(clk_out),
.locked(lock));

   integer k = 0;

   always @ (posedge clk_out or posedge reset)
      begin
        if(reset)
          Q=0;
          else
            if (enable)
              begin
               if (k ==10000)
              begin
            Q <= ~Q;
            k = 0;
             end
            else
           k = k +1;
         end
      end

   always @(Q)
   begin
      if (Q)
      begin
         AN = 8'b11111110;
         case(m_out)
            0   :seg0=7'b0000001;
            1   :seg0=7'b1001111;
            2   :seg0=7'b0010010;
            3   :seg0=7'b0000110;
            4   :seg0=7'b1001100;
            5   :seg0=7'b0100100;
            6   :seg0=7'b0100000;
            7   :seg0=7'b0001111;
            8   :seg0=7'b0000000;
            9   :seg0=7'b0000100;
            default: seg0=7'bx;
         endcase
      end
   else
```

```
    begin
      AN = 8'b11111101;
      case(z)
        0   :seg0=7'b0000001;
        1   :seg0=7'b1001111;
        2   :seg0=7'b0010010;
        3   :seg0=7'b0000110;
        4   :seg0=7'b1001100;
        5   :seg0=7'b0100100;
        6   :seg0=7'b0100000;
        7   :seg0=7'b0001111;
        8   :seg0=7'b0000000;
        9   :seg0=7'b0000100;
        default: seg0=7'bx;
      endcase
    end
```

Code 4

Task 5

In Task 5, we were asked to use the IP Catalog to generate a simple 4-bit counter core that counts up from 0 to 9. I was never able to figure out a correct design for task 5 and therefore did not test code in the lab.

Task 6

In task 6, we were asked to design a carry-look-ahead adder similar to that you designed in Lab 2 but using gate-level modeling. This was a very tedious task and I wish we did not have to design gate-level modeling. I do believe I got the code working and my design can be seen below in *Code 6.*

```
nor #1 (g2, a[0], b[0]);
   nand #1 (g3, a[0], b[0]);
   and #3 (g1, ~g2, g3);
   and #3 (g4, ~cin, g3);
   nor #1 (g5, g4, g2); // g5 is the cin for the second bit adder
   xor #4 (s[0], g1, cin);

   nor #1 (g7, a[1], b[1]);
   nand #1 (g8, a[1], b[1]);
   and #3 (g6, ~g7, g8);
   and #3 (g9, ~cin, g3, g8);
   and #3 (g11, g8, g2);
   nor #1 (g10, g9, g11,g7); // g10 is the cin for the third bit adder
   xor #4 (s[1], g5, g6);

   nor #1 (g13, a[2], b[2]);
   nand #1 (g14, a[2], b[2]);
   and #3 (g12, ~g13, g14);
   and #3 (g15, ~cin, g3, g8, g14);
   and #3 (g17, g8, g14, g2);
   and #3 (g18, g14, g7);
   nor #1 (g16, g15, g17, g18, g13); // g16 is the cin for the fourth bit adder
   xor #4 (s[2], g10, g12);

   nor #1 (g20, a[3], b[3]);
   nand #1 (g21, a[3],b[3]);
   and #3 (g19, ~g20, g21);
   and #3 (g22, ~cin, g3, g8, g14, g21);
   and #3 (g23, g8, g14, g21, g2);
   and #3 (g24, g7, g21, g14);
   and #3 (g25, g21, g13);
   nor #1 (cout, g22, g23, g24, g25, g20);
   xor #4(s[3], g16, g19);

endmodule
```

Code 6

Task 7

In task 7, we were asked to modify the carry-look-ahead adder of task 6 using the def param statements. We did not have to physically test the code. We were only asked to develop and simulate the testbench. The code can be seen below in Code 7.

```
 parameter and_delay = 0;
  parameter xor_delay = 0;
  parameter inv_delay = 0;

   wire g1, g2, g3, g4, g5, g6,
      g7, g8, g9, g10, g11, g12,
```

```
g13, g14, g15, g16, g17, g18,
g19, g20, g21, g22, g23, g24, g25;

    nor #(and_delay, inv_delay)(g2, a[0], b[0]);
    nand #(and_delay, inv_delay)(g3, a[0], b[0]);
    and #and_delay(g1, ~g2, g3);
    and #and_delay(g4, ~cin, g3);
    nor #(and_delay, inv_delay)(g5, g4, g2); // g5 is the cin for the second bit
adder
    xor #xor_delay(s[0], g1, cin);

    nor #and_delay(g7, a[1], b[1]);
    nand #(and_delay, inv_delay)(g8, a[1], b[1]);
    and #and_delay(g6, ~g7, g8);
    and #and_delay(g9, ~cin, g3, g8);
    and #and_delay(g11, g8, g2);
    nor #(and_delay, inv_delay)(g10, g9, g11,g7); // g10 is the cin for the third
bit adder
    xor #xor_delay(s[1], g5, g6);

    nor #and_delay(g13, a[2], b[2]);
    nand #(and_delay, inv_delay)(g14, a[2], b[2]);
    and #and_delay(g12, ~g13, g14);
    and #and_delay(g15, ~cin, g3, g8, g14);
    and #and_delay(g17, g8, g14, g2);
    and #and_delay(g18, g14, g7);
    nor #(and_delay, inv_delay)(g16, g15, g17, g18, g13); // g16 is the cin for the
fourth bit adder
    xor #xor_delay(s[2], g10, g12);

    nor #and_delay(g20, a[3], b[3]);
    nand #(and_delay, inv_delay)(g21, a[3],b[3]);
    and #and_delay(g19, ~g20, g21);
    and #and_delay(g22, ~cin, g3, g8, g14, g21);
    and #and_delay(g23, g8, g14, g21, g2);
    and #and_delay(g24, g7, g21, g14);
    and #and_delay(g25, g21, g13);
    nor #(and_delay, inv_delay)(cout, g22, g23, g24, g25, g20);
    xor #xor_delay(s[3], g16, g19);

endmodule
```

Code 6

```
module Task7_tb();
reg D;
reg Clk;
reg ce;
reg reset;
wire Q;
integer k;

Task7 DUT (.Clk(Clk), .reset(reset), .D(D), .ce(ce),
.Q(Q));
    initial begin
      #300 $finish; //runs simulation 300 times
      end

      initial begin
      D = 0;
      ce = 0;
      reset = 0;
      #20 //20
      D = 1;
      #40 //60
      ce = 1;
      #20 //80
      ce = 0;
      #20 //100
      D = 0;
      #20 //120
      reset = 1;
      #20 //140
      reset = 0;
      #40 //180
      ce = 1;
      #20 //200
      ce = 0;
      #20 //220
      D = 1;
      end

      //Clock
          initial begin
             for (k = 0; k <= 15; k = k+1)
             begin
                Clk = 0;
                #10 Clk = 1;
                #10;
                Clk = 0;
                end
            end
endmodule
```

Testbench 7

Task 8

In task 8, we were asked to model a T flip-flop with synchronous negative-logic reset and clock enable using the above given to us in *Code 7.* All we had to do for this task was physically verify the design on the board. The design ran as expected.

```
module Task8(
   input Clk,
   input reset_n,
   input T,
   output reg Q
   );

   always @(negedge Clk)
   if (!reset_n)
   Q <= 1'b0;
   else if (T)
   Q <= ~Q;
endmodule
```

Code 7

Conclusion

In conclusion, this lab was very helpful for us to understand different types of latches and flip-flops. This lab was also an introduction to coding workbenches. While I wish the lab went into more detail on how to correctly write workbenches and provided examples, it was overall a simple process. The lab manual was well written and gave plenty of coding examples for latches and flip-flops and latches. Overall, I would say the lab was fairly easy but did take a very long time to write all the code for since there were so many tasks. Other than the length, I would say this lab was a good intro to writing our own testbenches, latches, and flip-flops.