Digital Design Lab #7: Decoders and Multiplexers in HDL

Charlie Coleman
Lab Partner: Luke Taylor
2Ø16 October 2Ø

Objective: The objective of this lab was to gain experience using Hardware Description Language while designing the seven segment decoder, a functions using a decoder, and multiplexers.

Design:
1A) In part one, we were to redesign the seven segment decoder, but use HDL instead of a schematic. The functions were the same as before.

Code:

```
entity lab7project is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : in  STD_LOGIC;
           D : in  STD_LOGIC;
           Sa : out  STD_LOGIC;
           Sb : out  STD_LOGIC;
           Sc : out  STD_LOGIC;
           Sd : out  STD_LOGIC;
           Se : out  STD_LOGIC;
           Sf : out  STD_LOGIC;
           Sg : out  STD_LOGIC);
end lab7project;

architecture Behavioral of lab7project is

begin
 Sa <= ((not A) and (not B) and (not C) and D) or ((not A) and B and (not C) and (not D)) or (A and B and (not C) and D) or (A and (not B) and C and D);
 Sb <= ((not A) and B and (not C) and D) or (A and B and (not D)) or (A and C and D) or (B and C and (not D));
 Sc <= ((not A) and (not B) and C and (not D)) or (A and B and (not D)) or (A and B and C);
 Sd <= ((not A) and B and (not C) and (not D)) or (B and C and D) or (A and (not B) and C and (not D)) or ((not B) and (not C) and D);
 Se <= ((not A) and D) or ((not A) and B and (not C)) or ((not B) and (not C) and D);
 Sf <= ((not A) and (not B) and D) or ((not A) and (not B) and C) or ((not A) and C and D) or (A and B and (not C) and D);
 Sg <= ((not A) and (not B) and (not C)) or ((not A) and B and C and D) or (A and B and (not C) and (not D));


end Behavioral;
```

2A) In part 2A, we were to design a 4:16 decoder using HDL. The decoder acted as a normal decoder would, activating the decimal equivalent line to the input values. The functions for each output were found via Karnaugh maps.

Code:

```
entity decoder4_16 is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : in  STD_LOGIC;
           D : in  STD_LOGIC;
           y : out  STD_LOGIC_vector(15 downto 0));

end decoder4_16;

architecture Behavioral of decoder4_16 is

begin
      y(0) <= not A and not B and not C and not D;
      y(1) <= not A and not B and not C and D;
      y(2) <= not A and not B and C and not D;
      y(3) <= not A and not B and C and D;
      y(4) <= not A and B and not C and not D;
      y(5) <= not A and B and not C and D;
      y(6) <= not A and B and C and not D;
      y(7) <= not A and B and C and D;
      y(8) <= A and not B and not C and not D;
      y(9) <= A and not B and not C and D;
      y(10) <= A and not B and C and not D;
      y(11) <= A and not B and C and D;
      y(12) <= A and B and not C and not D;
      y(13) <= A and B and not C and D;
      y(14) <= A and B and C and not D;
      y(15) <= A and B and C and D;

end Behavioral;
```

2C) For part 2C, the 4:16 decoder used in part 2A was to be used to implement 8 functions. The decoder was used as a component in the code, and the functions were implemented using OR gates with decoder outputs.

Code:

```
entity functions8 is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : in  STD_LOGIC;
           D : in  STD_LOGIC;
           f0 : out  STD_LOGIC;
           f1 : out  STD_LOGIC;
           f2 : out  STD_LOGIC;
           f3 : out  STD_LOGIC;
           f4 : out  STD_LOGIC;
           f5 : out  STD_LOGIC;
           f6 : out  STD_LOGIC;
           f7 : out  STD_LOGIC);
end functions8;

architecture Behavioral of functions8 is
COMPONENT decoder4_16
```

```
        PORT(
              A : IN std_logic;
              B : IN std_logic;
              C : IN std_logic;
              D : IN std_logic;
              y : OUT std_logic_vector(15 downto 0)
              );
        END COMPONENT;
    signal ysignal: std_logic_vector(15 downto 0);
    begin
        Inst_decoder4_16: decoder4_16 PORT MAP(
              A => A,
              B => B,
              C => C,
              D => D,
              y => ysignal
        );

        f0 <= ysignal(0) or ysignal(14) or ysignal(15);
        f1 <= ysignal(1) or ysignal(13) or ysignal(15);
        f2 <= ysignal(2) or ysignal(12) or ysignal(15);
        f3 <= ysignal(3) or ysignal(11) or ysignal(15);
        f4 <= ysignal(4) or ysignal(10) or ysignal(15);
        f5 <= ysignal(5) or ysignal(9) or ysignal(15);
        f6 <= ysignal(6) or ysignal(8) or ysignal(15);
        f7 <= ysignal(7) or ysignal(15);
    end Behavioral;
```

3A) For part 3, a quad 4:1 multiplexer was to be implemented. A quad 4:1 MUX takes 4 groups of 4 inputs and 2 select lines, and outputs the group of inputs who's decimal value is the same as the select line's decimal value. This code used when and else statements.

Code:

```
    entity multiplexer4_1 is
        Port ( s0 : in  STD_LOGIC;
               s1 : in  STD_LOGIC;
               a0 : in  STD_LOGIC;
               a1 : in  STD_LOGIC;
               a2 : in  STD_LOGIC;
               a3 : in  STD_LOGIC;
               b0 : in  STD_LOGIC;
               b1 : in  STD_LOGIC;
               b2 : in  STD_LOGIC;
               b3 : in  STD_LOGIC;
               c0 : in  STD_LOGIC;
               c1 : in  STD_LOGIC;
               c2 : in  STD_LOGIC;
               c3 : in  STD_LOGIC;
               d0 : in  STD_LOGIC;
               d1 : in  STD_LOGIC;
               d2 : in  STD_LOGIC;
               d3 : in  STD_LOGIC;
               y0 : out  STD_LOGIC;
               y1 : out  STD_LOGIC;
               y2 : out  STD_LOGIC;
               y3 : out  STD_LOGIC);
```

```
end multiplexer4_1;

architecture Behavioral of multiplexer4_1 is

begin
      y0 <= a0 when (s1 = '0' and s0 = '0') else
                  b0 when (s1 = '0' and s0 = '1') else
                  c0 when (s1 = '1' and s0 = '0') else
                  d0 when (s1 = '1' and s0 = '1');
      y1 <= a1 when (s1 = '0' and s0 = '0') else
                  b1 when (s1 = '0' and s0 = '1') else
                  c1 when (s1 = '1' and s0 = '0') else
                  d1 when (s1 = '1' and s0 = '1');
      y2 <= a2 when (s1 = '0' and s0 = '0') else
                  b2 when (s1 = '0' and s0 = '1') else
                  c2 when (s1 = '1' and s0 = '0') else
                  d2 when (s1 = '1' and s0 = '1');
      y3 <= a3 when (s1 = '0' and s0 = '0') else
                  b3 when (s1 = '0' and s0 = '1') else
                  c3 when (s1 = '1' and s0 = '0') else
                  d3 when (s1 = '1' and s0 = '1');
end Behavioral;
```

Procedure: To begin, we used the seven segment decoder lab and rewrote the design using VHDL. The Xilinx software was used to do this, and the code is shown above. To do this, we created a new source from project and used the VHDL Module setting. We set the 4 inputs A, B, C, and D to the in direction, and the outputs OUTPUTS set to out direction. This created the entity above and we simply added in the equations under architecture. We synthesized and tested the design and printed the result. We then coded a 4-16 decoder using VHDL. This was done by setting the four inputs A, B, C, and D to the in direction under the VHDL Module, and the outputs Y_0-Y_{15} to the out direction. We then coded the decoder into the architecture setting and compiled and tested. We then used the decoder as a component to implement the functions

$$F_0(A,B,C,D) = \sum m(0,14,15)$$

$$F_1(A,B,C,D) = \sum m(1,13,15)$$

$$F_2(A,B,C,D) = \sum m(2,12,15)$$

$$F_3(A,B,C,D) = \sum m(3,11,15)$$

$$F_4(A,B,C,D) = \sum m(4,10,15)$$

$$F_5(A,B,C,D) = \sum m(5,9,15)$$

$$F_6(A,B,C,D) = \sum m(6,8,15)$$

$$F_7(A,B,C,D) = \sum m(7,15)$$

This was done by creating a new VHDL source and setting the inputs A,

B, C, and D to in, and the outputs f_1-f_7 to out. The decoder was made into a component by clicking the View VHDL Insantitation Template. It was used in the new source by copying the component section and pasting it to the new one under architecture, and the instant section into the behavioral section. This function was then compiled simulated and printed. Lastly, we created a multiplexer circuit. This was done by writing a VHDL code to implement a 4-1 mux that multiplexes 4 bits at a time. This was done by connecting the $A_Ø$ $B_Ø$ $C_Ø$ and $D_Ø$ to the $Y_Ø$ output. The same was done for Y_1, Y_2, and Y_3. The mux was then compiled and simulated for the following inputs;

S(1:Ø) = ØØ A(3:Ø) = ØØØØ B(3:Ø) = ØØØ1 C(3:Ø) = ØØ1Ø D(3:Ø) = ØØ11
S(1:Ø) = ØØ A(3:Ø) = Ø11Ø B(3:Ø) = ØØØ1 C(3:Ø) = ØØ1Ø D(3:Ø) = ØØ11
S(1:Ø) = ØØ A(3:Ø) = 11ØØ B(3:Ø) = ØØØ1 C(3:Ø) = ØØ1Ø D(3:Ø) = ØØ11
S(1:Ø) = Ø1 A(3:Ø) = ØØØ1 B(3:Ø) = Ø1Ø1 C(3:Ø) = ØØ1Ø D(3:Ø) = ØØ11
S(1:Ø) = Ø1 A(3:Ø) = ØØØ1 B(3:Ø) = 11Ø1 C(3:Ø) = ØØ1Ø D(3:Ø) = ØØ11
S(1:Ø) = Ø1 A(3:Ø) = ØØØ1 B(3:Ø) = 1111 C(3:Ø) = ØØ1Ø D(3:Ø) = ØØ11
S(1:Ø) = 1Ø A(3:Ø) = ØØØ1 B(3:Ø) = ØØØ1 C(3:Ø) = Ø1ØØ D(3:Ø) = ØØ11
S(1:Ø) = 1Ø A(3:Ø) = ØØØ1 B(3:Ø) = ØØØ1 C(3:Ø) = Ø11Ø D(3:Ø) = ØØ11
S(1:Ø) = 1Ø A(3:Ø) = ØØØ1 B(3:Ø) = ØØØ1 C(3:Ø) = 1ØØ1 D(3:Ø) = ØØ11
S(1:Ø) = 11 A(3:Ø) = ØØØ1 B(3:Ø) = ØØØ1 C(3:Ø) = ØØ1Ø D(3:Ø) = 1Ø1Ø
S(1:Ø) = 11 A(3:Ø) = ØØØ1 B(3:Ø) = ØØØ1 C(3:Ø) = ØØ1Ø D(3:Ø) = 11ØØ
S(1:Ø) = 11 A(3:Ø) = ØØØ1 B(3:Ø) = ØØØ1 C(3:Ø) = ØØ1Ø D(3:Ø) = ØØ11

Data: See attached Xilinx simulation results.

Data Analysis: For each of our Xilinx designs, the simulated results matched the expected results. For the seven segment decoder and the decoder functions, we were able to test the program on the board, seeing that the results were as expected. The decoder was tested only using Xilinx, not put on the board. The truth table was checked for accuracy compared to the expected output of a decoder. For the MUX, the number of input lines was too large to test all possible combinations, so test vectors given in the lab were used and results were compared to expected output of a MUX. Each design was a success.

Conclusion: In this lab, we were to gain experience with HDL by implementing multiple different designs which we had previously done using schematics. Learning HDL is helpful because HDL is important for and makes some more advanced designs possible. Schematics can become very cluttered very quickly, and HDL code is often easier to troubleshoot and examine. Some challenges faced in this weeks lab was learning to use a previous code as a component instead of just copying the code, which was learned in lab. We also had some trouble using vectors.