Ben Paulson
CPE 133-80-2206

BCD-to-Seven-Segment Decoder

Objectives

The purpose of this lab was to design and implement a circuit to convert a binary-coded decimal (BCD) input into outputs that can drive a 7-segment display on the Basys 3 development board. This was done by creating a non-standard decoder look-up table (LUT) with values to display the digits 0-9 on the display.

Procedures

The design of this circuit started with drawing a top-level black box diagram. There are two 4-bit inputs, one is the BCD number to be displayed and the other is an "enable" to choose which 7-segment displays of the 4 available on the Basys 3 board are to be turned on. There is also a 4-bit output, the anodes of the 7-segment display, which is controlled directly by the enable input. The other 8-bit output is the cathodes of the segments, which are shared between all 4 displays. The diagram is shown in *Figure 1.*

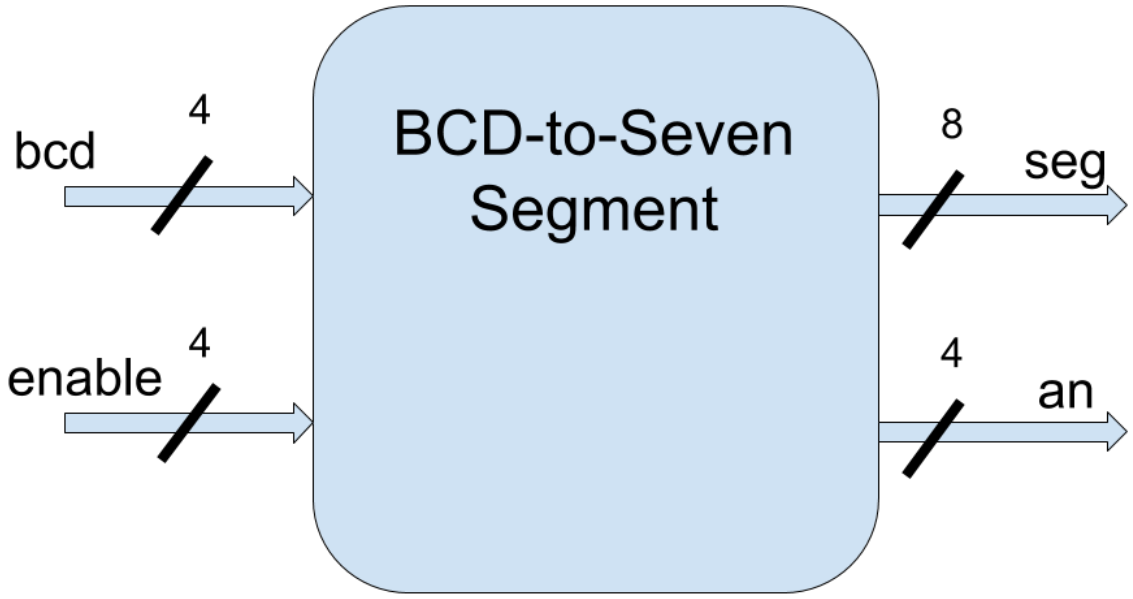


Figure 1 – Top level BBD

The next step was to create a LUT with the correct input and output bits to drive the 7-segment display. Since the input to the circuit is a 4-bit BCD number, there are a total of 16 output options. With an input above 9, the display is completely turned off because the number cannot be displayed on a single display. The LUT that was created can be seen in *Figure 2*.

abcd (BCD)	A	B	C	D	E	F	G	DP
0000	0	0	0	0	0	0	1	1
0001	1	0	0	1	1	1	1	1
0010	0	0	1	0	0	1	0	1
0011	0	0	0	0	1	1	0	1
0100	1	0	0	1	1	0	0	1
0101	0	1	0	0	1	0	0	1
0110	0	1	0	0	0	0	0	1
0111	0	0	0	1	1	1	1	1
1000	0	0	0	0	0	0	0	1
1001	0	0	0	1	1	0	0	1
1010	1	1	1	1	1	1	1	1
1011	1	1	1	1	1	1	1	1
1100	1	1	1	1	1	1	1	1
1101	1	1	1	1	1	1	1	1
1110	1	1	1	1	1	1	1	1
1111	1	1	1	1	1	1	1	1

Figure 2 – BCD to 7-Segment Decoder LUT

According to the Basys 3 board documentation, in order for a 7-segment display to turn on, both the anode and cathode of the segment must be a logical 0. So, every 0 in the LUT in *Figure 2* represents that the segment is on, whereas a 1 means it is off. Using this same principle, the anodes were assigned to be the inverse of the enable inputs.

$$an = \overline{enable}$$

Instead of deriving equations for each bit from this table and assigning the equations to each bit in code, the SystemVerilog module for this circuit uses an `always` block with a `case` statement for each possible combination, with a default of completely off if the input is too large. The HDL model written for this circuit can be seen in *Appendix A*, and the corresponding constraints file for mapping the inputs and outputs to physical hardware on the development board is in *Appendix B.*

Testing

To test correct operation of the circuit, another SystemVerilog module was created for the purpose of simulating all possible input and output combinations. The code for this simulation can be seen in *Appendix C*.

The output of the entire simulation, highlighting the BCD to 7-segment part, can be seen in the timing diagram in *Figure 4*. A zoomed view of the diagram, showing more specifically the enable and anode parts of the circuit, is in the timing diagram in *Figure 5*.

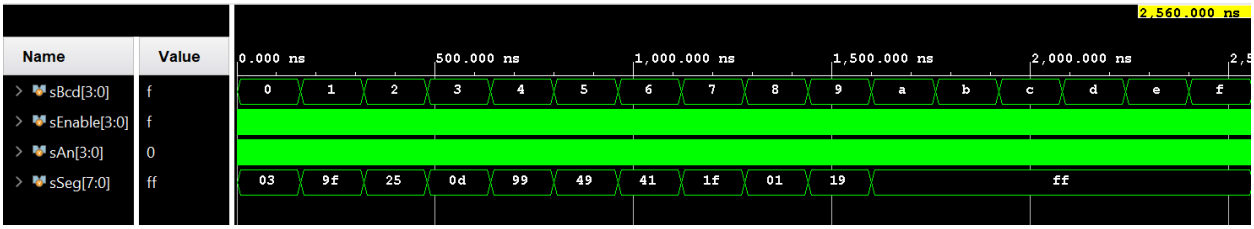


Figure 3 - Simulation highlighting BCD and 7-segment display

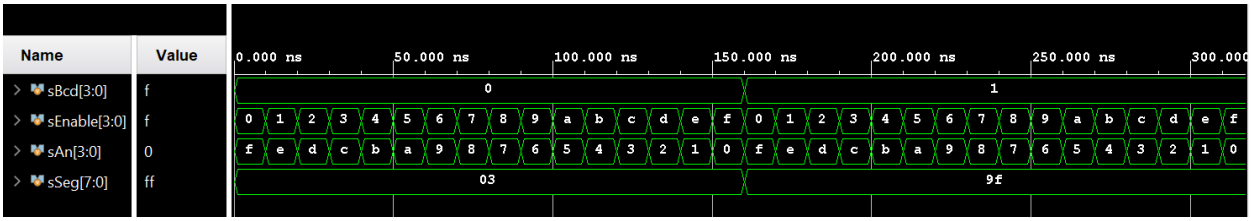


Figure 4 - Zoomed simulation highlighting enable and anode assignment

After simulation, the program was uploaded to the development board where it was visually verified that the circuit was operating correctly.

Conclusion

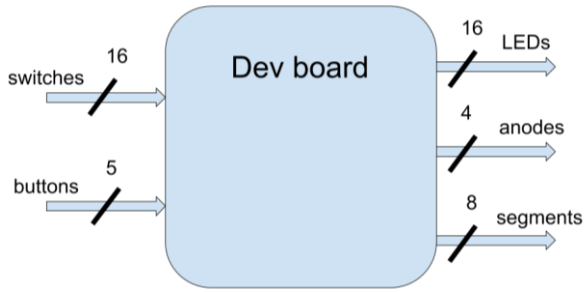
In this lab, a 4-bit binary-coded decimal input was converted to an 8-bit output to drive a 7-segment display using a non-standard decoder as a look-up table. The LUT allows for a more simple design by bypassing the need for an equation for each bit of the output. A second 4-bit input allows for selecting which of the four available 7-segment displays should be turned on. The circuit was tested by first writing a simulation to display all possible input and output combinations, and then by uploading the program to the development board for manual testing.

Questions

If you were not able to use a decoder in this experiment, who many concurrent signal assignments would you have needed to implement the segments portion of the seven segment display? Briefly explain.

Without a decoder, you would need to have an equation for each bit of the 7-segment display. With the decoder, we only have to worry about assigning one single value to the output.

Draw a black box diagram showing the anodes, LEDs, segments, switches, and buttons on the development board. Be sure to carefully labeled inputs and outputs on your diagram.



Similar to 7-segment displays, there are also 14-segment displays out there in the real world. Briefly describe the main purposed served by 14-segment displays.

A 14-segment display gives more options for characters that can be created. Instead of being limited to letters and numbers, you could display other kind of special characters with the additional LED segments.

This lab activity required that you use a generic decoder. You could have modeled this decoder using one of two possible types of procedural blocks. Provide the code for a generic decoder using one of the statements you did not use in this lab activity.

The BCD-to-seven-segment display module can be implemented with an `always_comb` block instead of an `always_ff @` block as it was originally. The difference between these modules is that the `always_comb` will run continuously and unconditionally, whereas `always_ff` will run continuously but only on a specific condition (a change in bcd input, in this case). It also works to replace the `case` statement with an `if` block.

```
// An example of how the BCD-7-segment decoder works
// using a different type of procedural block
`timescale 1ns / 1ps

/*
* bcd is the 4-bit BCD input
* seg is each LED in the 7-segment display
* where the MSB is segment A and LSB is
* the decimal point
*/
module BCDSevenSegmentDecoder_AlwaysComb(
    input [3:0] enable,
    input [3:0] bcd,
    output reg [7:0] seg,
    output [3:0] an
    );

    // 0 will turn it on, 1 off
    assign an = ~enable;

    /*
     * Using always_comb instead of always_ff allows
     * for continuous assignment without a changing bcd condition
     * (which always_ff required)
     * Using if statement instead of case also works
    */
```

```
    always_comb
        if (bcd == 0) seg <= 8'b00000011;
        else if (bcd == 1) seg <= 8'b10011111;
        else if (bcd == 2) seg <= 8'b00100101;
        else if (bcd == 3) seg <= 8'b00001101;
        else if (bcd == 4) seg <= 8'b10011001;
        else if (bcd == 5) seg <= 8'b01001001;
        else if (bcd == 6) seg <= 8'b01000001;
        else if (bcd == 7) seg <= 8'b00011111;
        else if (bcd == 8) seg <= 8'b00000001;
        else if (bcd == 9) seg <= 8'b00011001;
        else seg <= 8'b11111111;

endmodule
```

One of the important design approaches in modeling digital circuits is to use a LUT (decoder) whenever possible. Briefly describe why this is a good approach.

The use of a LUT allows us to bypass the need for equations for individual output bits. In this lab specifically, it allowed the segment values to be assigned all at once with 10 different cases instead of having to come up with equations for each segment individually.

How can the set of seven segment displays ever display a number such as "3948" if you can only ever display one number at a time on the set of displays?

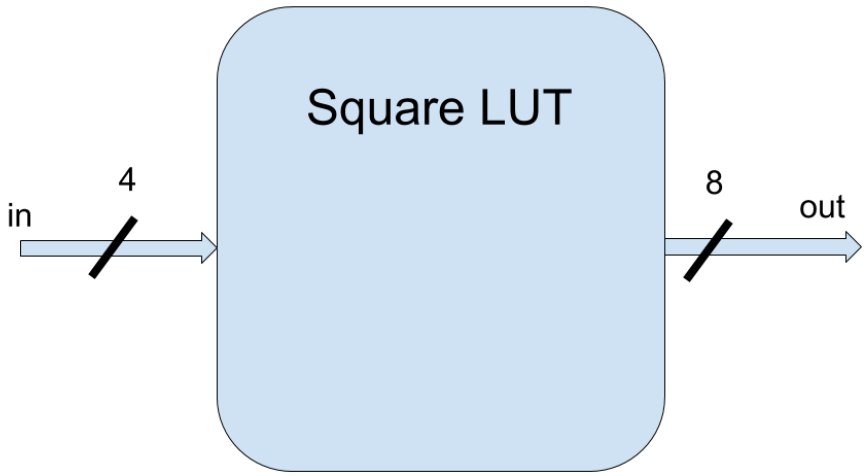
You would have to enable the first display and set the segment output to the correct number, and then disable it and enable the next one and change the segment output again to the next number, and keep doing this until the entire number is displayed. The rate at which you switch displays would have to be fast enough so that it your eye cannot see it flicker. The Basys 3 reference manual says that the refresh rate should be between 60Hz and 1KHz, and that at around 45Hz, the human eye can begin to see a flicker.

A four-digit seven-segment display has 32 LEDs, which seems to indicate we needs 32 outputs to control it. But… a four-digit seven-segment display is typically driven by 12 signals. Briefly explain the main reason this is done.

This is done for the reason explained in the previous question. The display that is active is cycled using the 4 anode bits until all have been active, while switching the 8 segment bits to the correct values, and then this is repeated at a fast rate so that it appears as though all segments are independent.

Design Problems

Non-standard decoders are essentially LUTs. As you know from computer programming, often times using a LUT for calculations is a great idea. For this problem, show the BBD and code for a 4-input decoder that outputs the square of the input. Consider both input and output to be unsigned binary numbers. Use as few signals for the output as possible but still be able to represent the largest possible value for the output.




```
`timescale 1ns / 1ps

/*
* Takes a 4-bit input and outputs
* the square of it using a lookup table.
* Output must be 8 bits to account for all possibilities
*/
module SquareLUT(
    input [3:0] in,
    output reg [7:0] out
    );

    always_ff @(in)
        case (in)
            0: out <= 8'b00000000; // 0
            1: out <= 8'b00000001; // 1
            2: out <= 8'b00000100; // 4
            3: out <= 8'b00001001; // 9
            4: out <= 8'b00010000; // 16
            5: out <= 8'b00011001; // 25
            6: out <= 8'b00100100; // 36
            7: out <= 8'b00110001; // 49
```

```
            8: out <= 8'b01000000; // 64
            9: out <= 8'b01010001; // 81
            10: out <= 8'b01100100; // 100
            11: out <= 8'b01111001; // 121
            12: out <= 8'b10010000; // 144
            13: out <= 8'b10101001; // 169
            14: out <= 8'b11000100; // 196
            15: out <= 8'b11100001; // 225
        endcase

endmodule
```

Appendix

A: SystemVerilog model for BCD-to-seven-segment decoder

```
`timescale 1ns / 1ps

/*
* bcd is the 4-bit BCD input
* seg is each LED in the 7-segment display
* where the MSB is segment A and LSB is
* the decimal point
*/
module BCDSevenSegmentDecoder(
    input [3:0] enable,
    input [3:0] bcd,
    output reg [7:0] seg,
    output [3:0] an
    );

    // 0 will turn it on, 1 off
    assign an = ~enable;

    /* Always be updating the value of seg
     * Using a case statement (decoder) instead of equations
    */
    always_ff @(bcd)
        case (bcd)
            0: seg <= 8'b00000011;
            1: seg <= 8'b10011111;
            2: seg <= 8'b00100101;
```

```
            3: seg <= 8'b00001101;
            4: seg <= 8'b10011001;
            5: seg <= 8'b01001001;
            6: seg <= 8'b01000001;
            7: seg <= 8'b00011111;
            8: seg <= 8'b00000001;
            9: seg <= 8'b00011001;
            default: seg <= 8'b11111111; // Default to turn it off
        endcase

endmodule
```

B: Constraints file

```
# Set enable input switches to leftmost switches 15-12

set_property PACKAGE_PIN R2 [get_ports {enable[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[3]}]

set_property PACKAGE_PIN T1 [get_ports {enable[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[2]}]

set_property PACKAGE_PIN U1 [get_ports {enable[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[1]}]

set_property PACKAGE_PIN W2 [get_ports {enable[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {enable[0]}]

# Map "an" output to 7-segment anodes

set_property PACKAGE_PIN W4 [get_ports {an[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[3]}]

set_property PACKAGE_PIN V4 [get_ports {an[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[2]}]

set_property PACKAGE_PIN U4 [get_ports {an[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[1]}]

set_property PACKAGE_PIN U2 [get_ports {an[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {an[0]}]
```

```
# Map BCD input to rightmost switches 3-0

set_property PACKAGE_PIN W17 [get_ports {bcd[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {bcd[3]}]

set_property PACKAGE_PIN W16 [get_ports {bcd[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {bcd[2]}]

set_property PACKAGE_PIN V16 [get_ports {bcd[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {bcd[1]}]

set_property PACKAGE_PIN V17 [get_ports {bcd[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {bcd[0]}]

# Map "seg" output to the correct LED in the 7-segment display

set_property PACKAGE_PIN W7 [get_ports {seg[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[7]}]

set_property PACKAGE_PIN W6 [get_ports {seg[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[6]}]

set_property PACKAGE_PIN U8 [get_ports {seg[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[5]}]

set_property PACKAGE_PIN V8 [get_ports {seg[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[4]}]

set_property PACKAGE_PIN U5 [get_ports {seg[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[3]}]

set_property PACKAGE_PIN V5 [get_ports {seg[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[2]}]

set_property PACKAGE_PIN U7 [get_ports {seg[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[1]}]

set_property PACKAGE_PIN V7 [get_ports {seg[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {seg[0]}]
```

C: Simulation code for testing BCD-to-seven-segment module

```
`timescale 1ns / 1ps

module BCDSevenSegmentDecoder_Simulation();

    logic [3:0] sBcd, sEnable, sAn;
    logic [7:0] sSeg;

    BCDSevenSegmentDecoder UUT(
        .bcd(sBcd), .enable(sEnable),
        .seg(sSeg), .an(sAn));

    initial begin
        /*
        * Loop through all possible values and combinations
        * for bcd and enable inputs. Both are 4-bits so 0-15.
        */
        for (byte i = 0; i < 16; i++) begin
            for (byte j = 0; j < 16; j++) begin
                sBcd = i;
                sEnable = j;
                #10;
            end
        end
    end

endmodule
```