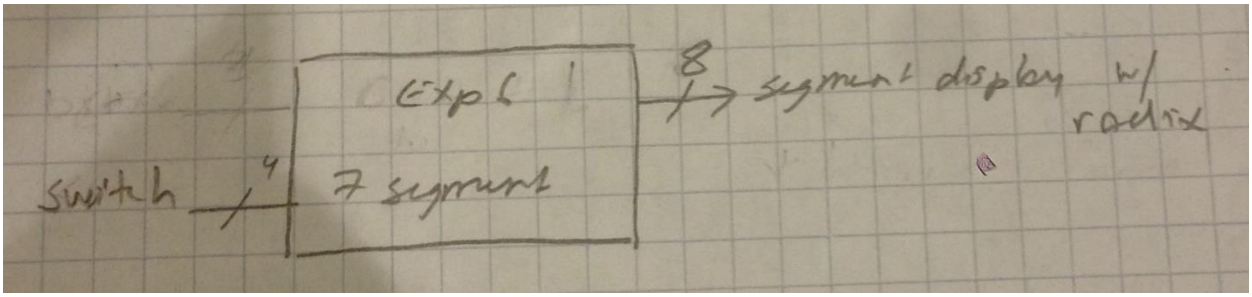
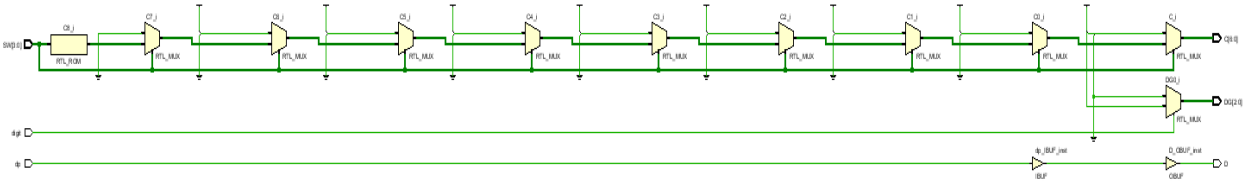
CPE 133 Lab 3 Report
Prof. John Callenes-Sloan
Carson Pepe, Luis Gomez

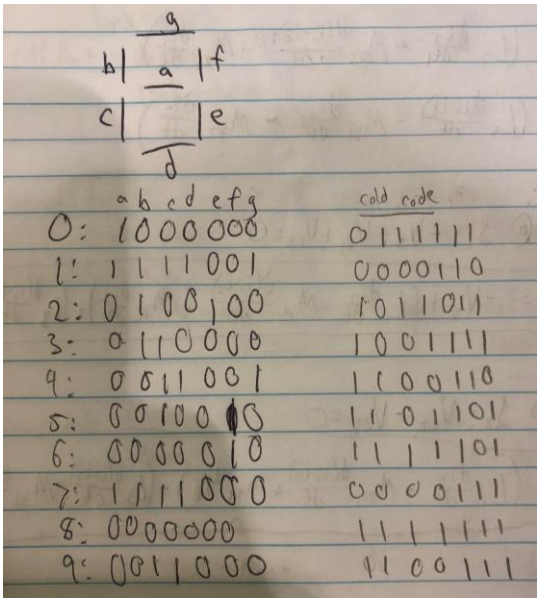
# Experiment 6: BCD to 7-segment Display Decoder


## Summary:

For experiment 6 we built a BCD to 7-Segment Display Decoder. Using logic engrained in Verilog, we designed the BASYS-3 board to display the numbers one through nine corresponding with the 4-bit BCD number generated by the left-most four switches on the board. In order to get our code to work, we had to inverse the zero's and one's for each segment (one-cold-codes).



***Above Figures: One Cold Code conversion & Black Box Diagram***

## Verification:

We provided a physical board demo exhaustively testing all possible outputs of the decoder circuit. Unfortunately, we were pressed for time while researching Verilog syntax and troubleshooting code, so were unable to provide a simulation test.

***VIDEO LINK:*** https://youtu.be/WgEJ3Ko8c1U

## Answers to Questions:

***1. How many K-maps would you have needed if you implemented this design using equations for each of the segment outputs? How much longer would this design have taken you if you had to implement it using K-maps?***

We would have needed seven K-maps. If we used K-maps instead of "the awesome power of Verilog" it would have taken us much longer. As we will demonstrate later in exp 7, utilizing K-maps for a 7-segment display considerably longer.

***2. How many different letters of the English alphabet could be represented with a standard 7-segment display? Which letters could not be displayed? For this question, letters can be displayed in either lower or upper-case formats***

.

You could display 18 out of 26 letters in the alphabet with a standard 7-segment display. The letters that could not be displayed are K, M, Q, T, V, W, X, and Z.

***3. When representing the decimal digits, what is the least used segment?***

Segment c, or the bottom left segment (orientation of segments is different on board than presented in lab instructions) , is the least used segment (not used five times).

***4. Why would an engineer even care about the previous question?***

The least used segment in a 7-segment display could be important to an engineer because if their particular 7-segment display didn't need to display the numbers that include that segment, then they could have it always turned off which saves power.

## Code:

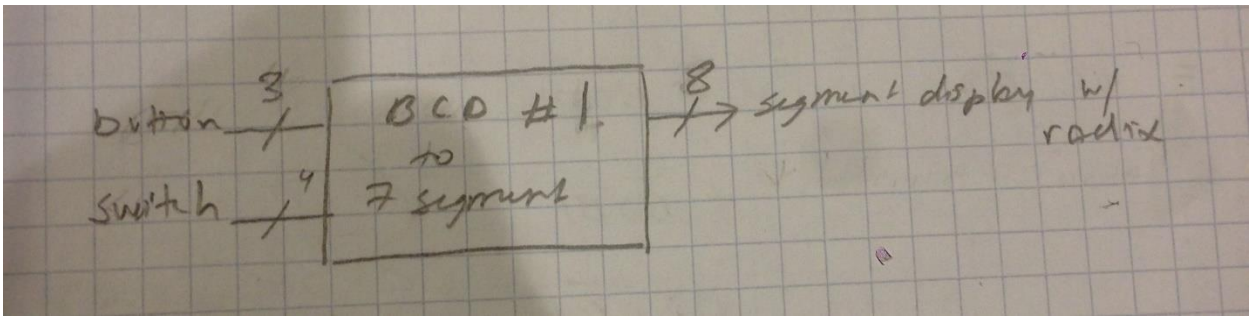
```
22
23  module exp6(
24      input dp, digit,
25      input [3:0] SW,
26      output D,
27      output [2:0] DG,
28      output [6:0] C
29      );
30
31      reg DP;
32      reg [2:0] DIGIT;
33
34      begin
35          assign C =
36          ((SW==0) ? 7'b1000000 :
37          ((SW==1) ? 7'b1111001 :
38          ((SW==2) ? 7'b0100100 :
39          ((SW==3) ? 7'b0110000 :
40          ((SW==4) ? 7'b0011001 :
41          ((SW==5) ? 7'b0010010 :
42          ((SW==6) ? 7'b0000010 :
43          ((SW==7) ? 7'b1111000 :
44          ((SW==8) ? 7'b0000000 :
45          ((SW==9) ? 7'b0011000 : 7'b1111111
46          )))))))))) ;
47          end
48
49      always @ (dp,digit)
50      begin
51          DP = (dp == 0) ? 1'b0 : 1'b1;
52          DIGIT = (digit == 1) ? 3'b000 : 3'b111;
53      end
54      assign D = DP;
55      assign DG = DIGIT;
56
57
58  endmodule
```

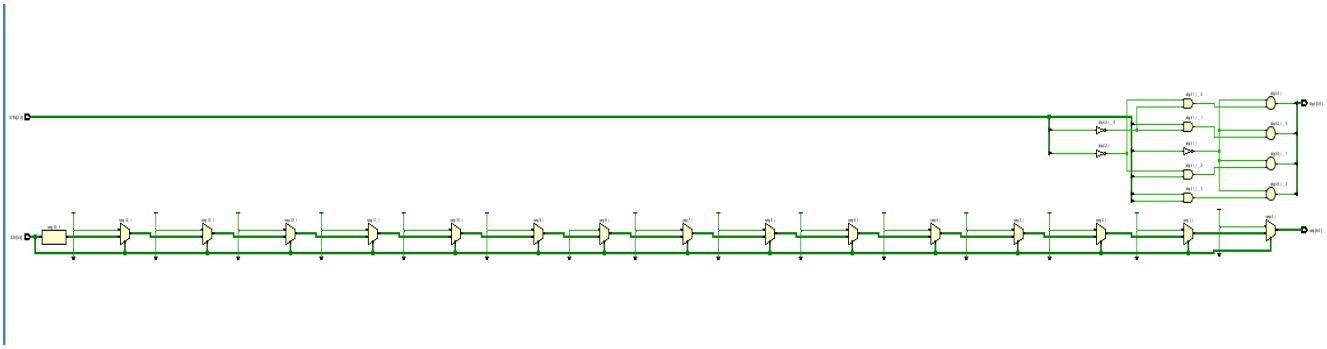
## Experiment 7: BCD to 7-segmnt Decoder (Behavioral Model)

Summary:

In this experiment we built a BCD to 7-Segment Decoder but this time using behavioral modeling. This Decoder includes the use of buttons that can toggle which one of the four 7-segment displays should be turned on and also can toggle all of them off. It essentially works the same way the previous experiment's decoder does but with an extra feature.

***Figures below include BBD, Circuit Diagram, Attempted Boolean Expressions for 7-Segment Display, & K-Maps for Buttons and Switches***





```
SEG[6] = ~((SW[3] & SW[2] & SW[1]) + (~SW[3] & ~SW[2] & SW[1] & SW[0]) +
    (SW[3] & ~SW[2] & ~SW[1] & ~SW[0]));

SEG[5] = ~((SW[3] & SW[2] & ~SW[0]) + (SW[3] & SW[2] & ~SW[1]) +
    (SW[3] & ~SW[1] & ~SW[0]) + (~SW[3] & ~SW[2] & SW[1] & ~SW[0]));

SEG[4] = ~((SW[3] & ~SW[0]) + (SW[3] & SW[2] & ~SW[1]) +
    (SW[3] & ~SW[1] & ~SW[0]) + (~SW[3] & ~SW[2] & SW[1] & ~SW[0]));

SEG[3] = ~((SW[3] & ~SW[2] & SW[1] & SW[0]) + (SW[2] & SW[1] & ~SW[0]) +
    (~SW[2] & ~SW[1] & ~SW[0]) + (~SW[3] & SW[2] & ~SW[1] & SW[0]));

SEG[2] = ~((~SW[3] & ~SW[2] & SW[1] & SW[0]) + (~SW[3] & ~SW[2] & ~SW[1]) +
    (~SW[3] & ~SW[1] & ~SW[0]) + (SW[3] & SW[2] & ~SW[1] & SW[0]));

SEG[1] = ~((~SW[3] & ~SW[2] & SW[0]) + (~SW[3] & ~SW[2] & ~SW[1]) +
    (SW[3] & ~SW[2] & SW[1] & ~SW[0]) + (~SW[3] & ~SW[1] & ~SW[0]) +
        (~SW[2] & ~SW[1] & SW[0]));

SEG[0] = ~((SW[3] & ~SW[2] & SW[1] & SW[0]) + (SW[3] & SW[2] & SW[1] & ~SW[0]) +
    (~SW[3] & ~SW[2] & SW[1] & ~SW[0]) + (~SW[3] & SW[2] & ~SW[1] & ~SW[0]));
```

***Figures below include K-Maps for 3-bit BTN (push button) input vector & 4-bit SW (switch) inputs. <u>K-MAP NOTE: Light colors are Essential PI, Bold are Implicants</u>***

| **Exp7** | BTN1/BTN0 | | | | |
|---|---|---|---|---|---|
| BTN2 | | 00 | 01 | 11 | 10 |
| digit [0] | 0 | 1 | 1 | 0 | 1 |
| | 1 | 0 | 0 | 0 | 0 |
| | | 00 | 01 | 11 | 10 |
| digit [1] | 0 | 1 | 0 | 1 | 1 |
| | 1 | 0 | 0 | 0 | 0 |
| | | 00 | 01 | 11 | 10 |
| digit [2] | 0 | 1 | 1 | 1 | 0 |
| | 1 | 0 | 0 | 0 | 0 |
| | | 00 | 01 | 11 | 10 |
| digit [3] | 0 | 0 | 1 | 1 | 1 |
| | 1 | 0 | 0 | 0 | 0 |

| **Exp7** | SW1/SW0 | | | | |
|---|---|---|---|---|---|
| SW3/SW2 | | 00 | 01 | 11 | 10 |
| seg [0] | 00 | 0 | 0 | 1 | 1 |
| | 01 | 1 | 1 | 0 | 1 |
| | 11 | 0 | 1 | 1 | 1 |
| | 10 | 1 | 1 | 1 | 1 |
| | | 00 | 01 | 11 | 10 |
| seg [1] | 00 | 1 | 0 | 0 | 0 |
| | 01 | 1 | 1 | 0 | 1 |
| | 11 | 1 | 0 | 1 | 1 |
| | 10 | 1 | 1 | 1 | 1 |
| | | 00 | 01 | 11 | 10 |
| seg [2] | 00 | 1 | 0 | 0 | 1 |
| | 01 | 0 | 0 | 0 | 1 |
| | 11 | 1 | 1 | 1 | 1 |
| | 10 | 1 | 0 | 1 | 1 |
| | | 00 | 01 | 11 | 10 |
| seg [3] | 00 | 1 | 0 | 1 | 1 |
| | 01 | 0 | 1 | 0 | 1 |
| | 11 | 1 | 1 | 0 | 1 |
| | 10 | 1 | 0 | 1 | x x |
| | | 00 | 01 | 11 | 10 |
| seg [4] | 00 | 1 | 1 | 1 | 0 |
| | 01 | 1 | 1 | 1 | 1 |
| | 11 | 0 | 1 | 0 | 0 |
| | 10 | 1 | 1 | 0 | 1 |
| | | 00 | 01 | 11 | 10 |
| seg [5] | 00 | 1 | 1 | 1 | 1 |
| | 01 | 1 | 0 | 1 | 0 |
| | 11 | 0 | 1 | 0 | 0 |
| | 10 | 1 | 1 | 0 | 1 |
| | | 00 | 01 | 11 | 10 |
| seg [6] | 00 | 1 | 0 | 1 | 1 |
| | 01 | 0 | 1 | 1 | 1 |
| | 11 | 1 | 0 | 1 | 1 |
| | 10 | 1 | 1 | 0 | 1 |

***The K-maps above were generated by mapping the inverted one cold code resulting from the logic governing the 7-Segment four digit display. See below for code inversions.***

```
reg [6:0] SEG;
reg [3:0] DIGIT;

always @ (BTN, SW) begin
    begin
    assign DIGIT =
    // one cold code !!!!
    ((BTN[1]==0 && BTN[0]==0) ? 4'b0001 : // 1110
    ((BTN[1]==1 && BTN[0]==0) ? 4'b0010 : // 1101
    ((BTN[1]==0 && BTN[0]==1) ? 4'b0100 : // 1011
    ((BTN[1]==1 && BTN[0]==1) ? 4'b1000 : // 0111
    ((BTN[2]==1) ? 4'b1111 : 4'b0000) // 0000 : 1111
    ))));
    end
    assign DIGIT = digit;

begin
    assign SEG =
    // One Cold Code
    ((SW==0) ? 7'b1000000 : // 0111111 X
    ((SW==1) ? 7'b1111001 : // 0000110 X
    ((SW==2) ? 7'b0100100 : // 1011011 X
    ((SW==3) ? 7'b0110000 : // 1001111 X
    ((SW==4) ? 7'b0011001 : // 1100110 X
    ((SW==5) ? 7'b0010010 : // 1101101 X
    ((SW==6) ? 7'b0000010 : // 1111101 X
    ((SW==7) ? 7'b1111000 : // 0000111 X
    ((SW==8) ? 7'b0000000 : // 1111111 X
    ((SW==9) ? 7'b0011000 : // 1100111 X
    ((SW==10) ? 7'b0001000 : // 1110111 X
    ((SW==11) ? 7'b0000111 : // 1111000 X
    ((SW==12) ? 7'b1000110 : // 0111001 X
    ((SW==13) ? 7'b0100001 : // 1011110 X
    ((SW==14) ? 7'b0000110 : // 1111001
    ((SW==15) ? 7'b0001110 : 7'b1111111) // 1110001 0000000
    ))))))))))))))) ;
    end
    assign SEG = seg;
end*/
```

## Verification:

Because of difficulties in implementing the logic for this lab we were only able to board test by way of a physical demo, found below on YouTube.

***VIDEO LINK: https://youtu.be/TOBpPBRjJOc***

## Answers to Questions:

**1. Think for a few minutes about how you would have implemented this project using a dataflow (RTL) model. Briefly describe whether you think the easier approach would be: dataflow (RTL) or behavioral.**

In our opinion, we think it would be an easier approach to implement this project using behavioral modeling. This is because behavioral modeling is used to describe the function of a design in an algorithmic manner. We inadvertently began this experiment using an RTL model via Boolean expressions, that was very difficult to troubleshoot because of one-cold-codes.

**2. Provide your well-supported opinion regarding the following statement: "Verilog behavioral models are much more powerful than Verilog dataflow(RTL) models."**

This is true because in general, dataflow models are used for simulations while behavioral modeling in Verilog is used for both simulation and synthesis. Behavioral code is easier to debug and is faster to simulate.

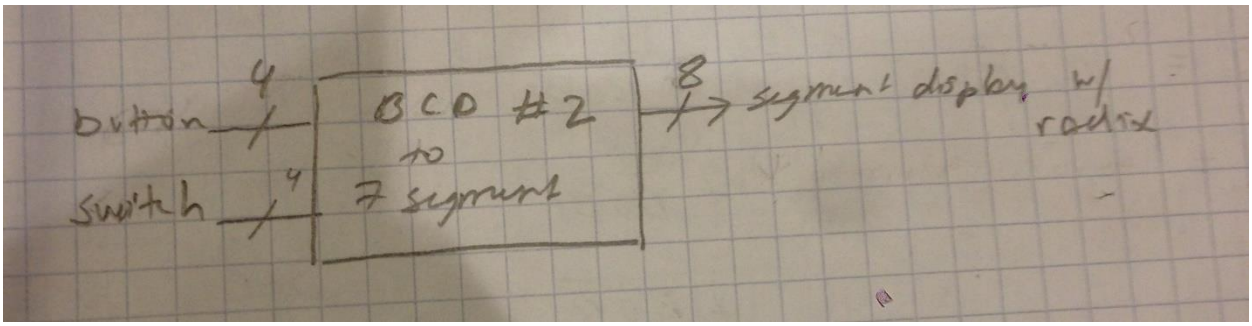
## Code:

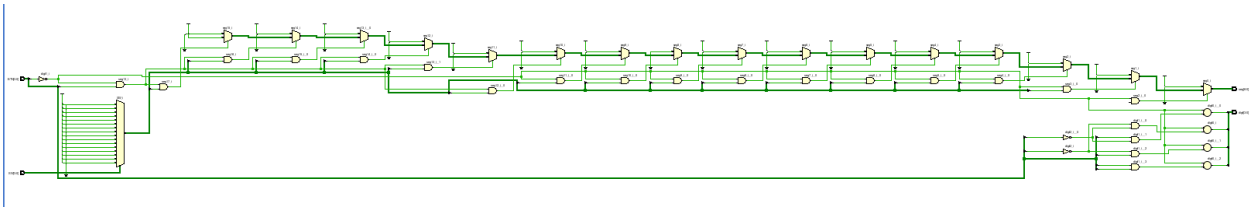
```
module bcd7segment(
    input [2:0] BTN,
    input [3:0] SW,
    output [3:0] digit,
    output [6:0] seg
    );

    reg [3:0] DIGIT;
    reg [6:0] SEG;

    always @ (BTN,SW) begin
        // Logic for Push Buttons
        DIGIT[0] = ((~BTN[2]) + (~BTN[1] & ~BTN[0])) ;
        DIGIT[1] = ((~BTN[2]) + (BTN[1] & ~BTN[0])) ;
        DIGIT[2] = ((~BTN[2]) + (~BTN[1] & BTN[0])) ;
        DIGIT[3] = ((~BTN[2]) + (BTN[1] & BTN[0])) ;

        SEG =
                // One Cold Code
                ((SW==0) ? 7'b1000000 : // 0111111 X
                ((SW==1) ? 7'b1111001 : // 0000110 X
                ((SW==2) ? 7'b0100100 : // 1011011 X
                ((SW==3) ? 7'b0110000 : // 1001111 X
                ((SW==4) ? 7'b0011001 : // 1100110 X
                ((SW==5) ? 7'b0010010 : // 1101101 X
                ((SW==6) ? 7'b0000010 : // 1111101 X
                ((SW==7) ? 7'b1111000 : // 0000111 X
                ((SW==8) ? 7'b0000000 : // 1111111 X
                ((SW==9) ? 7'b0011000 : // 1100111 X
                ((SW==10) ? 7'b0001000 : // 1110111 X
                ((SW==11) ? 7'b0000111 : // 1111000 X
                ((SW==12) ? 7'b1000110 : // 0111001 X
                ((SW==13) ? 7'b0100001 : // 1011110 X
                ((SW==14) ? 7'b0000110 : // 1111001
                ((SW==15) ? 7'b0001110 : 7'b1111111) // 1110001 0000000
                )))))))))))))))) ;
    end
    assign digit = DIGIT;
    assign seg = SEG;
    endmodule
        // Logic for switches
        /*SEG[0] = ~(~SW[3] & ~SW[2] & ~SW[1] & ~SW[0]);
```

## Experiment 8 BCD to 7-Segment Decoder w/ Modifications


Summary:

In this experiment we constructed a standard decoder utilizing 8-inputs (switches and buttons) outputting an 8 bit signal to the 7-segment display with radix point. The button & switch inputs worked the same as in experiment 7 except the added button BTN [3] toggled the hex values of the 7-segment display. The K-Maps for the pushbuttons in experiment 7 were valid here for Buttons 0-2. Button 3 worked sedately from these in the logic.

Verification: Because of time constraints we were only able to provide exhaustive physical board tests for all possible outputs.

***VIDEO LINK*** https://youtu.be/t0YBuWK5lXQ

Answers to Questions:

**1. There are many ways to complete this design. My feeling with this type of design is to do it "the easiest way I can think of" and let the Verilog synthesizer sort out the details. In your own terse words, what does this statement mean? Also, in your own terse words, list two ways you can tell whether this approach is "not too inefficient" or not.**

The statement means that one should minimize the work they must do to generate a digital design by leveraging the computational strength of the Verilog synthesizer. This approach is efficient but has drawbacks. It is efficient in the sense that a high level design can be rapidly created by a designer and implemented quickly by the software; however, the drawback is that the approach utilized by the software might not produced an optimized result in more complicated designs.

Code:

```
module bcd7segment(
    input [3:0] BTN,
    input [3:0] SW,
    output [3:0] digit,
    output [6:0] seg
    );

    reg [3:0] DIGIT;
    reg [6:0] SEG;

always @ (BTN,SW) begin
            // Logic for Push Buttons
            DIGIT[0] = ((~BTN[2]) + (~BTN[1] & ~BTN[0])) ;
            DIGIT[1] = ((~BTN[2]) + (BTN[1] & ~BTN[0])) ;
            DIGIT[2] = ((~BTN[2]) + (~BTN[1] & BTN[0])) ;
            DIGIT[3] = ((~BTN[2]) + (BTN[1] & BTN[0])) ;

            SEG = // One Cold Code
            ((~BTN[2] && SW==0) ? 7'b1000000 : // 0111111 X
            ((~BTN[2] && SW==1) ? 7'b1111001 : // 0000110 X
            ((~BTN[2] && SW==2) ? 7'b0100100 : // 1011011 X
            ((~BTN[2] && SW==3) ? 7'b0110000 : // 1001111 X
            ((~BTN[2] && SW==4) ? 7'b0011001 : // 1100110 X
            ((~BTN[2] && SW==5) ? 7'b0010010 : // 1101101 X
            ((~BTN[2] && SW==6) ? 7'b0000010 : // 1111101 X
            ((~BTN[2] && SW==7) ? 7'b1111000 : // 0000111 X
            ((~BTN[2] && SW==8) ? 7'b0000000 : // 1111111 X
            ((~BTN[2] && SW==9) ? 7'b0011000 : // 1100111 X
            ((~BTN[2] && BTN[3] && SW==10) ? 7'b0001000 : // 1110111 X
            ((~BTN[2] && BTN[3] && SW==11) ? 7'b0000011 : // 1111000 X
            ((~BTN[2] && BTN[3] && SW==12) ? 7'b1000110 : // 0111001 X
            ((~BTN[2] && BTN[3] && SW==13) ? 7'b0100001 : // 1011110 X
            ((~BTN[2] && BTN[3] && SW==14) ? 7'b0000110 : // 1111001
            ((~BTN[2] && BTN[3] && SW==15) ? 7'b0001110 : 7'b1111111) // 1110001 0000000
             ))))))))))))))));
        end

        assign digit = DIGIT;
        assign seg = SEG;

        endmodule
```