ECE2029: Introduction to Digital Circuit Design

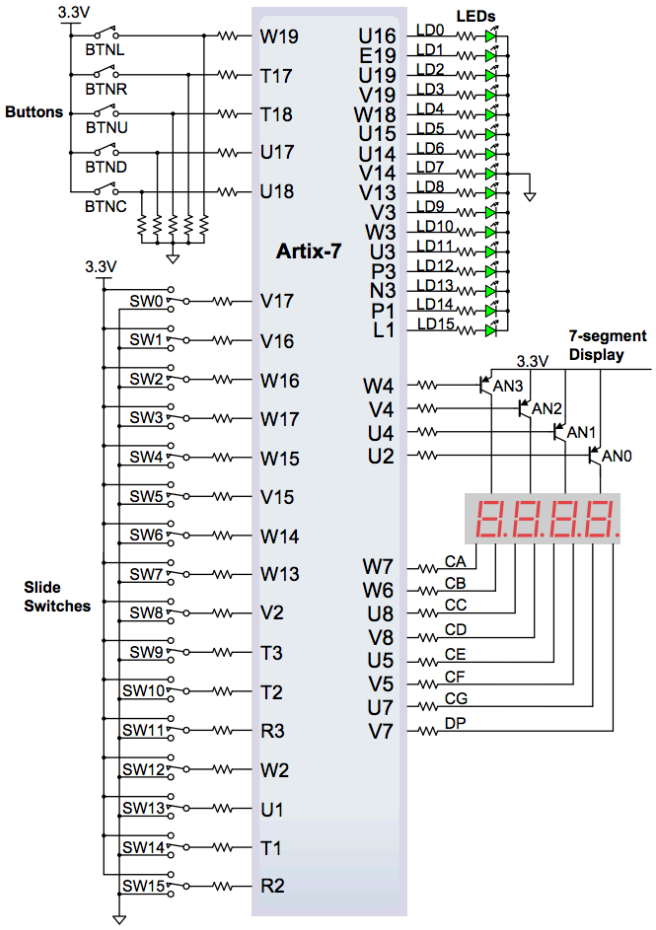
Lab 4 - Convert 8-Bit Binary to BCD Display

1. Objective

The objective of this lab is to build complicated combinational logic circuits, such as the binary to BCD converter. This lab will also be a transition from combinational logic to sequential logic circuits, such as counters. We will also apply the techniques learned from lectures on multiplexers and decoders to implement 4-digit 7-segment LED display.

The lab signoff is to use 8 dip switches to input a binary number and the 7-segment display shows the corresponding number in 4-digit decimal format. It is fine to have leading 0's in display, e.g. 8_{10} is shown as "0008" in display. The input number range is $(00\sim FF)_{16}$ and the decimal number on display is $(000\sim255)_{10}$.

2. Understand how multiple 7-segment LEDs work on Basys 3 board

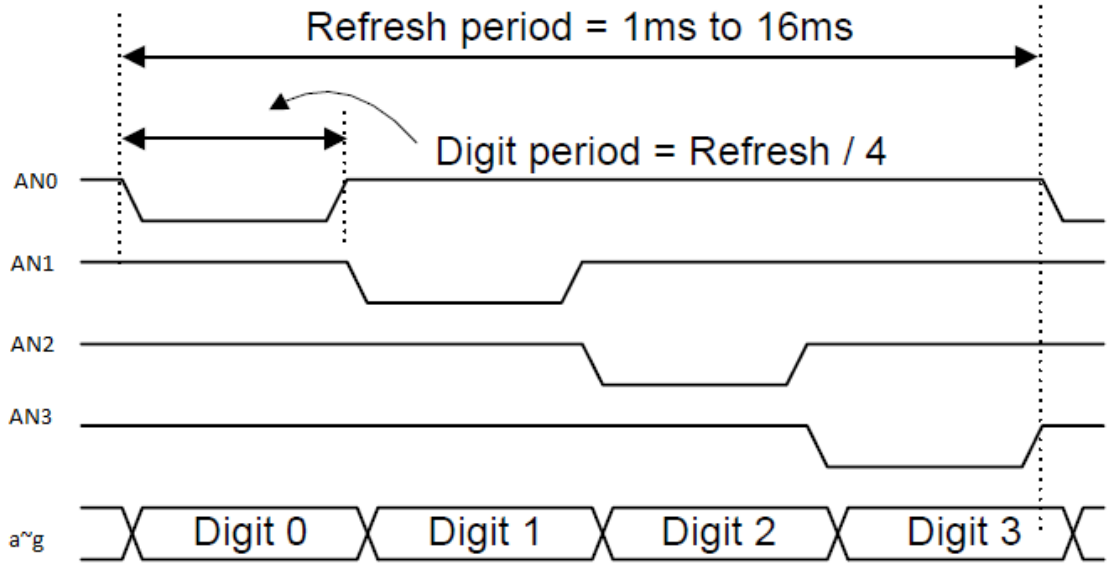


From your experience in Lab 3, all 4-digit of 7-segment display the same number. This is because all of their inputs a~g are tied together. We can turn each individually off by set the AN0 or AN1 or AN2 or AN3 to 1. Then how can we display 4 different numbers on this display?

The idea is to "trick your eyes". You will display one digit at one LED for a short period of time, by turning on only 1 LED and turn off the other 3 LEDs. Then you repeat for the next digit. If you do this fast enough, human eyes cannot catch the on/off switching activities. The following information is extracted from the Basys 3 reference manual on page 16.

A scanning display controller circuit can be used to show a four-digit number on this display. This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession, at an update rate that is faster than the human eye can detect. Each digit is illuminated just one-quarter of the time, but because the eye cannot perceive the darkening of a digit before it is illuminated again, the digit appears continuously illuminated. If the update or "refresh" rate is slowed to around 45 hertz, most people will begin to see the display flicker.

In order for each of the four digits to appear bright and continuously illuminated, all four digits should be driven once every 1 to 16ms, for a refresh frequency of 1KHz to 60Hz. For example, in a 60Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for ¼ of the refresh cycle, or 4ms.



In this lab, we will only need 3 decimal digits for BCD display. So the left most LED is actually turned off by setting AN4 = '1'. We will design a decoder and a mux module to implement the switch of data and AN signals simultaneously.

3. System Block Diagram

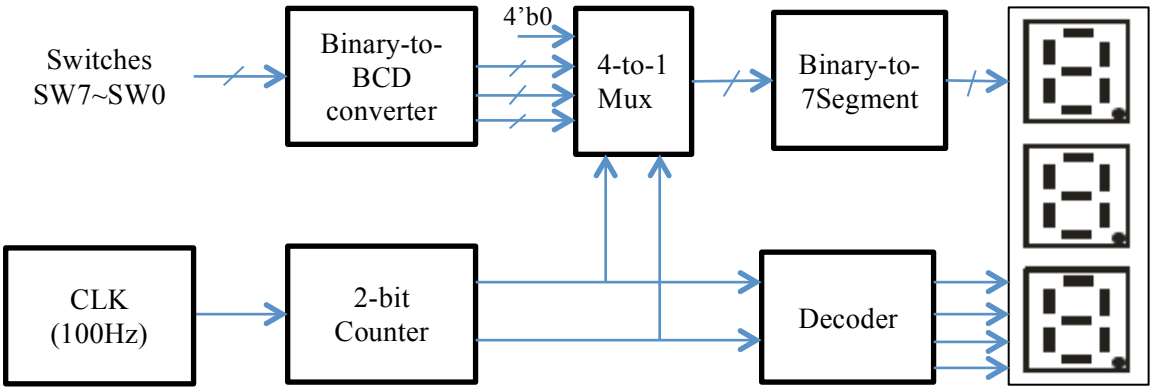


Figure 1. System diagram for BCD display on 7-segment LEDs

Among the blocks in Fig. 1, the binary-to-BCD is given and tested in the prelab. The binary-to-7segment display decoder was built in Lab 2. If you did not complete Lab2, you can use the following Verilog code to implement the binary-to-7segment decoder.

```
module bcd7seg(input[3:0] Y,
               output reg[6:0] disp); //Declaring outputs as register.
     always@(Y) //Describing an event that should happen to Y when a certain condition is met.
     begin
          case(Y)   //The case  statement is a decision instruction that executes the statement.
            0: disp=7'b0000001; //Here 7 bits represent 7 segments (a to g) on the display. Remember we
            1: disp=7'b1001111; provide active-low logic to the display. To display 0 , all segments (or
            2: disp=7'b0010010; bits) are set to 0 except for g. To display 1, b and c are set to zero, and
            3: disp=7'b0000110; rest are all 1s.
            4: disp=7'b1001100;
            5: disp=7'b0100100;
            6: disp=7'b0100000;
            7: disp=7'b0001111;
            8: disp=7'b0000000;
            9: disp=7'b0000100;
            10: disp=7'b0001000;
            11: disp=7'b1100000;
            12: disp=7'b0110001;
            13: disp=7'b1000010;
            14: disp=7'b0110000;
            15: disp=7'b0111000;
          endcase
     end
endmodule
```

4. Slow Clock at 100 Hz

In order to rotate the LED display properly, we need to generate a slow clock. As stated in the Basys 3 reference manual, the refresh clock should be in the range of 60 Hz to 1 KHz. As you can experiment later in this lab, the LED display is not stable if switching too fast. If the switch frequency is slower than 45 Hz, you can clearly see the "shifting digits". We opt to use a slow clock at 100 Hz, or a clock period of 10ms. The default system clock on the Basys 3 board is 100 MHz or a clock period of 10ns. The following code implements a counter to count the numbers of clock cycles to 500,000 that is equivalent to a time period of 5ms. If we hold the output signal '0' for 5ms, '1' for another 5ms and repeat, we generate a 100 Hz output clock.

```
module slowclock (input clk_in,
                  output reg clk_out);   //Clock input of the design, output clock

reg [20:0] period_count = 0;

always @ (posedge clk_in) //We trigger the clock with respect to positive (rising) edge of the clock.
     if (period_count!= 500000 - 1) //If the statement is not true, period_count not equivalent to 499000.
          begin
               period_count<= period_count + 1; //Increment period_count by 1.
               clk_out <= 0; //clk_out gets 0.
          end
     else  //If above statement is true
          begin
               period_count <= 0;    period_count gets 0
               clk_out <= 1;    clk_out gets 1.
          end
endmodule
```

5. 2-bit Counter

In order to send the 4 BCD digits to display iteratively, we need to design a 2-bit counter. The following code implements a 2-bit counter. We will cover the design of counters and more complicated sequential logic circuits in class in the following weeks.

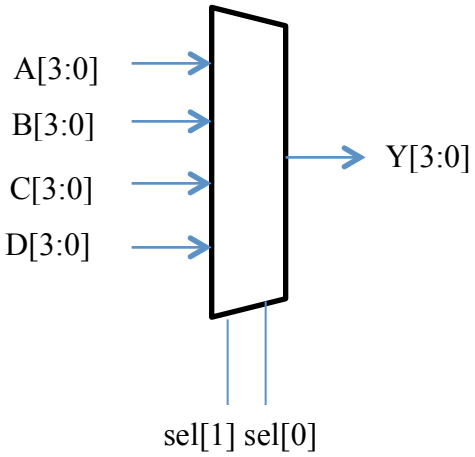
```
module my_counter(input clk,
                  output [1:0] Q); //clk is input, two outputs Q[1]--MSB, Q[0]--LSB.

     reg [1:0] temp = 0;
     always @(posedge clk) begin
          temp = temp + 1;
     end
     assign Q = temp;
endmodule
```

The purpose of the 2-bit counter is to generate a control signal. Table below shows the function of the counter in the mux and decoder.

2-bit Counter Output	4-to-1 Mux Output	Decoder Output [3:0] AN
00	ONES	1110
01	TENS	1101
10	HUNDREDS	1011
11	THOUSANDS (4'b0)	1111

6. 4-to-1 Mux

In this section, you will design and implement a 4-to-1 mux as in the diagram.



There are many different ways to implement a 4-to-1 mux in Verilog. The following is an example code for an implementation. You can design your 4-to-1 mux with different signal names.

```
module mux4to1( input [3:0] A,
                 input [3:0] B,
                input [1:0] C,
                input [3:0] D,
                input [1:0]
                sel,  output
                [3:0] Y);
     assign Y = (sel==0)?A : (sel==1)?B : (sel==2)?C : D; //selector switches decide what data at the input data lines go through to the output (Y).

endmodule
```

Note that we only need 3 digits for BCD display in this lab (ONES, TENS and HUNDRES). You can modify the 4-to-mux code to set one input signal D[3:0] to all 0's. Alternatively, you can map the input signal D to all 0's when you connect the modules together at the top level.

7. Decoder

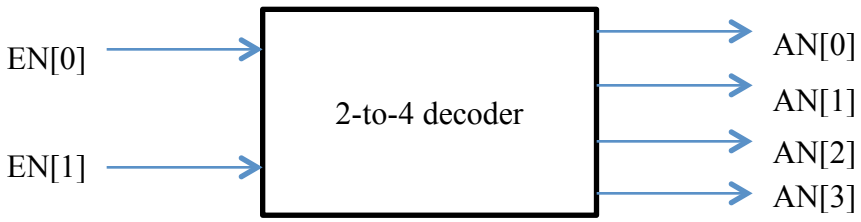
As we learned in class, we also need to build a 2-to-4 decoder to turn the 7-segment LED on/off alternately. The following Verilog code is a standard 2-to-4 decoder.

```
module decoder2to4 (input [1:0] en,
                    output reg [3:0] an);
     always@(en) begin
        case (en)
          0: an=4'b1110;
          1: an=4'b1101;
          2: an=4'b1011;
          3: an=4'b0111;
        endcase
     end
endmodule
```

//Declaring output (4, each used to enable each 7-segment display) as registers.

//When both en[0] and en[1] are 0, an[0] is 0, and rest of the o/ps are 1.

//When en[0] is 0 and en[1] is 1, an[1] is 0, and rest of the o/ps are 1.



You need to modify the code above to turn the left-most LED permanently off. Then it only shows 3 BCD digits in display. Hint: a very small modification will do the trick.

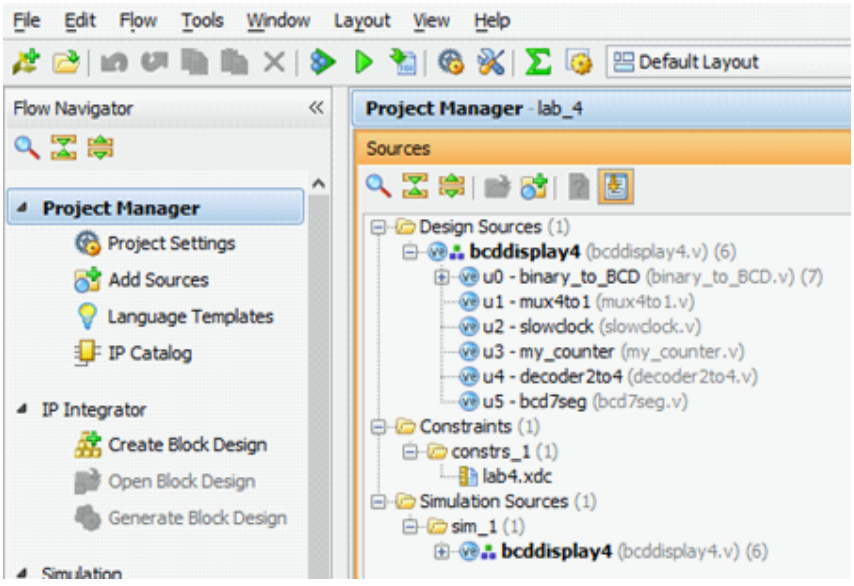
8. Put the system together

Finally, we will need to have a top-level design to put all. This will be your top module, add all modules as sources to this as shown below:

```
module bcddisplay4(input clk, input [7:0] sw,      //See Figure 1. Clock and 8-switches are inputs.2 outputs
                              output [3:0] an,     (enable, one each for each 7-segment display, and; seg [6:0]
                              output [6:0] seg);   for a to g).

      parameter zero = 4'b0000;   //Left most display is set to 0 because the range  for 8-bit is 0 to 255.
      wire clk_out;               //Declaring everything (all modules) that connects input and output modules as wires.
      wire [3:0] mux_out;
      wire [1:0] counter_out;
      wire [3:0] ones, tens, hundreds;

      binary_to_BCD u0(sw, ones, tens, hundreds);

      mux4to1 u1(ones, tens, hundreds, zero, counter_out, mux_out);

      slowclock u2(clk, clk_out);

      my_counter u3(clk_out, counter_out);

      decoder2to4 u4(counter_out, an);

      bcd7seg u5(mux_out, seg);

endmodule
```

The .xdc file (constraint file) is similar to Lab3, except for the addition of clock signal. It should include 8 input switches (sw[0] to sw[7]), clock signal (clk), 7 segment display (seg[0] to seg[7]), and 4 enables (an[0] to an[3]).

Generate a programming (.bit) file and download your design to the Basys 3 board. Show the lab TA that your completed design is working for sign-off.



Verilog HDL Operators

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	case equality	equality
!=	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional