

## Introduction to Verilog - Number Lock (Part II)

### 1. Synopsis:

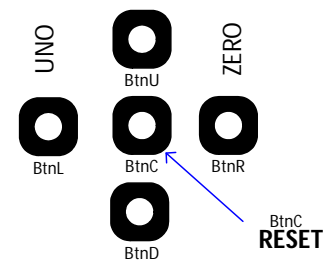
This lab introduces you to Verilog coding. You can consider this as an “alternative” to designing schematics for now. Your efforts will focus around the number-lock design you saw in a previous lab. You will have the chance to see a complete solution for the Detour lab written entirely in Verilog. Designing circuits in Verilog allows you to express your implementation at a much higher level. After completing this lab you should have a working knowledge of the syntax, you should possess the ability to debug Verilog code, and know the procedure to realize your Verilog code on an FPGA.

The content from sections 2 & 3 is an excerpt from your earlier lab. You can refer to it later as necessary but pay special attention to the state machine diagram in section 3.

### 2. Description of the Circuit:

In this design you will implement a slightly larger state machine than the simple Detour Signal state machine. In your design you will have two push buttons -- UNO and ZERO; *UNO in Spanish means ONE*. The two signals come out of the push button unit and into your state machine as inputs -- called **u** and **z**. The **u** signal goes high when UNO is pressed and the **z** signal goes high when the ZERO is pressed. Both signals remain low if neither is pressed. *Assume* that your state machine is clocked by approximately *10Hz clock* (0.1 second per clock cycle).

Humans tend to press a push button usually anywhere between a quarter second to half a second. *So once your state machine detects that a push button is pressed it should wait until the button is released* -- your design should not interpret a long push as multiple pushes.



The binary Number Lock secret code is **1 0 1 1**.

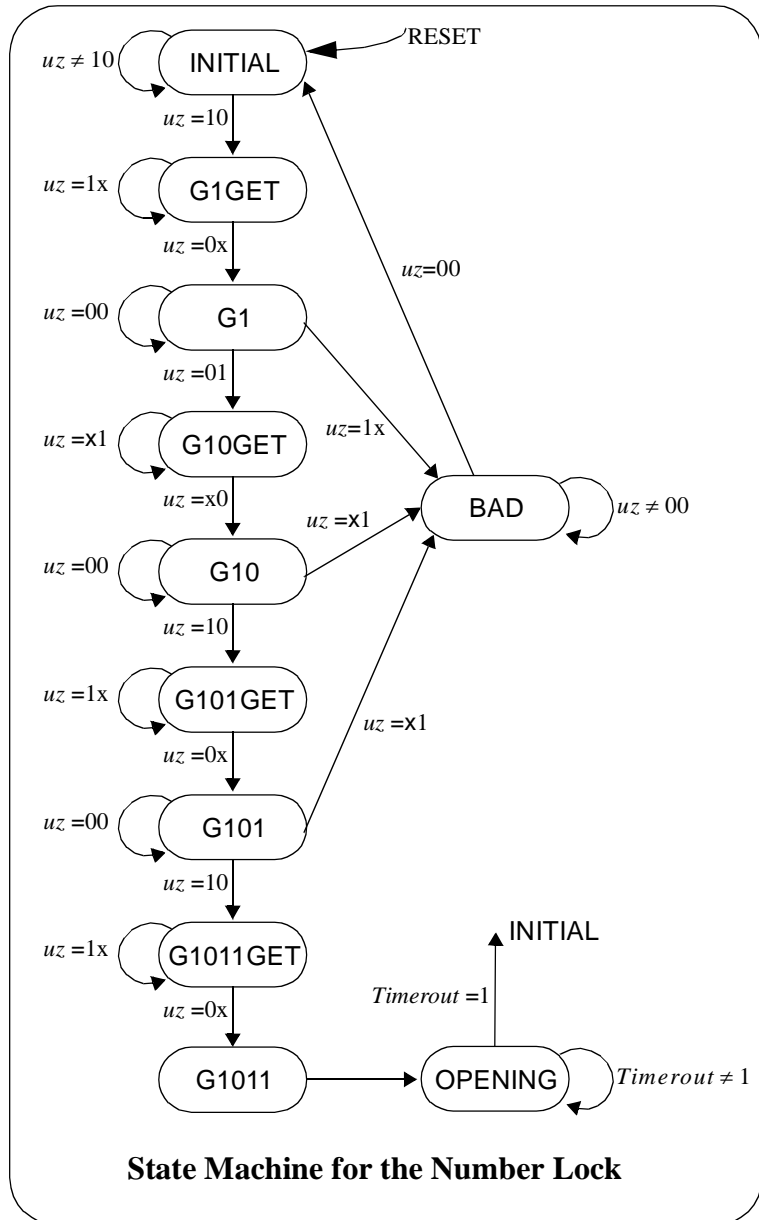
If the entered sequence is wrong, the state machine should return to the **INITIAL** state and start looking from the beginning. That is, if **1 0 1 0 1 1** is pressed, the number lock will not open even though the last four bits match with the code. This is because after **1 0 1 0** the machine returns to the **INITIAL**. We assume that the user will not press both the buttons together. This assumption simplifies the design. *One should not succeed in opening the lock by pressing both the buttons together every time though.*

### 3. The State Diagram

You are provided with a complete (and correct) state diagram. The state machine starts in the **INITIAL** state and as the user enters the Number Lock Code (by pressing UNO and ZERO buttons) the state machine moves through its states. Note the naming convention followed in the state machine: state **G1** means “got a 1”. Before this we have **G1GET** which means that we are in the process of “getting a 1”, meaning that the UNO button was pressed but has not been released yet.

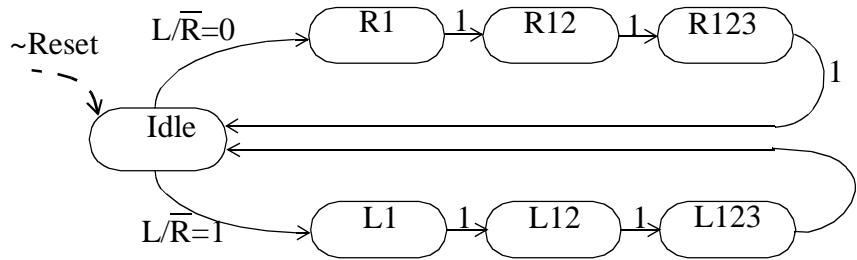
At **RESET**, the state machine enters the **INITIAL** state and waits for valid input. If the UNO button (**BTNL** on our Nexys 3 board) is pressed while the ZERO button (**BTNR**) is not, then the state machine enters **G1GET**. So the combination that takes you to **G1GET** is **UZ=10**. **G1GET** state means that you are still holding down the UNO button. Thus as long as **UZ=1x** (which means that UNO is pressed and you don't care about ZERO) you remain in **G1GET**. When you release the UNO button then you go to state **G1** (releasing a button sends a 0 and hence **UZ** must be **0x** for the state machine to transit from **G1GET** to **G1**).

This process continues if you keep entering the correct sequence, i.e. 1 0 1 1. Otherwise, the state machine moves to the **BAD** state and then back to **INITIAL**. If the entered sequence is correct then the state machine enters the **G1011** state which means that it got (received) 1011 -- the correct code. This can also be thought of as the “**DONE**” state for the system. The system remains in this state for only one clock cycle and then moves to the **OPENING** state. It stays in **OPENING** state until a counter times out. When the timer times out (i.e., **TIMEROUT =1**), the state machine moves to the **INITIAL** state. Notice that while the machine is in the **OPENING** state the push button are ignored.



## 4. Introduction to Verilog:

There are many ways to learn Verilog: reading a book, watching a lecture, just sitting down and hacking through it. This lab will introduce you to the fundamentals by dissecting a sample design implementing the earlier detour lab. It consists of two primary files: `ee254_detour_sm.v` (the state machine code) and `ee254_detour_top.v` (the top level design to interface the FPGA). In the following sections we will digest the major content of each. You should have a basic understanding of Verilog syntax and have your Cadence/Esperan Verilog guide as reference.



```
`timescale 1ns / 1ps
```

```
module ee254_detour_sm(q_I, q_R1, q_R12, q_R123, q_L1, q_L12, q_L123,
L_Rbar, Clk, reset, GLL, GL, GR, GRR);
```

```
    /* INPUTS */
```

```
    // Clock & Reset
```

```
    input Clk, reset;
```

```
    input L_Rbar;
```

```
    /* OUTPUTS */
```

```
    // our output leds (leftleft, left, right, rightright)
```

```
    output GLL, GL, GR, GRR;
```

```
    // store current state
```

```
    output q_I, q_R1, q_R12, q_R123, q_L1, q_L12, q_L123;
```

```
    reg [6:0] state;
```

```
    assign {q_L123, q_L12, q_L1, q_R123, q_R12, q_R1, q_I} = state;
```

```
// lets make accessing the state information easier within the state machine code
```

```
// each line aliases the appropriate state bits and sets up a 1-hot code
```

```
localparam
```

```
    QI      = 7'b00000001,
```

```
    QR1     = 7'b00000010,
```

```
    QR12    = 7'b00000100,
```

```
    QR123   = 7'b00001000,
```

```
    QL1     = 7'b00010000,
```

```
    QL12    = 7'b00100000,
```

```
    QL123   = 7'b01000000,
```

```
    UNK     = 7'bXXXXXXXX;
```

```

// NSL AND SM
always @ (posedge Clk, posedge reset)
begin
    if(reset)
        state <= QI;
    else
    begin
        case(state)
            QI:
                // don't worry about async reset here because 'if' statement considers this first
                if(L_Rbar)
                    // switch left
                    state <= QL1;
                else
                    // switch right
                    state <= QR1;
                // these are pretty boring, just unconditionals
                QR1: state <= QR12;
                QR12: state <= QR123;
                QR123: state <= QI;
                QL1: state <= QL12;
                QL12: state <= QL123;
                QL123: state <= QI;
                default: state <= UNK;
            endcase
        end
    end
end

// OFL
assign GLL = q_L123;
assign GL  = q_L123 | q_L12 | q_R1 | q_R12 | q_R123;
assign GR  = q_R123 | q_R12 | q_L1 | q_L12 | q_L123;
assign GRR = q_R123;

endmodule

```

#### 4.1 Module: ee254\_detour\_sm

Our state machine acts like a block (called a module). It has a number of inputs and outputs (“ports”). The port list includes each of the input signals, clock, reset, and also an output connection for each state. We indicate the *direction* of each port by defining them as input or output.

##### 4.1.1 State Memory & tentative 1-hot coding

In HDL coding, having each state as a separate wire is tedious and also error prone. More over we do not want to design the next state logic by hand. We equate the symbolic state names to 7-bit one-hot codes as shown below. The 7-bit one-hot codes here are only a suggestion here. We will later teach you how to tell the synthesis tool to synthesize a one-hot coded state machine or encoded or gray-coded state machine.

```

reg [6:0] state;
// aliases the appropriate state bit and sets up a 1-hot parameter
assign {q_L123, q_L12, q_L1, q_R123, q_R12, q_R1, q_I} = state;

localparam
    QI      = 7'b0000001,
    QR1     = 7'b0000010,
    QR12    = 7'b0000100,

```

The first line (`reg [6:0] state;`) declares a 7-bit register (7 flip-flops) to store the current state. Notice how there are 7 of them. This register represents our state memory. The next line takes each of the registers and gives the output a different name. So `q_I = state[0]`, `q_R1 = state[1]`, and so forth. Now we use something like “`state <= QR12`” to set all of the FFs at once.

#### 4.1.2 Next State Logic (NSL)

We do not implement the Next state logic at gate level manually in HDL coding. We use a `case` statement within an `always` block that is only sensitive to `clk` and `reset`. This indicates that we only want this code to “execute” only when either `clk` or `reset` changes. Our block is a little more specific though and says only run this block on the positive edge of the clock (`posedge`) or reset on the positive level of `reset`.

To design a NSL up to now we looked at all of the arrows GOING IN to a specific state. The implementation in Verilog differs because we look at all the arrows GOING OUT of the state:

```

case(state)
    QI:
        if(L_Rbar)
            // switch left
            state <= QL1;
        else
            // switch right
            state <= QR1;
    QR1: state <= QR12;
    QR12: state <= QR123;
    QR123: state <= QI;

```

The Verilog `case` statement allows us to “match” the current state (i.e. look at one particular state on the state diagram). You can see the trivial (unconditional) transitions from `QR1`, `QR12`, `QR123` above. If the current `state` matches any of these, we unconditionally load `state` with next state value. Confirming with our state diagram, we see that we have a choice in `QI` (is the switch `L` or `R`?).

What happens when we first turn our device on? We should force the circuit into `Q_I` (the initial state) when reset goes active. We were smart enough to include `posedge reset` in our sensitivity list so the block is run when reset goes active high. We accomplish that by adding:

```
if(reset)
    state <= QI;
else
```

You should ask yourself: “why does reset take priority over other state transitions?” (answer: asynchronous reset has high-priority over clock).

Notice that all of our assignments in this `always` statement use “non-blocking” ( `<=` ) assignment. We should use non-blocking assignments when assigning to registers (Flip-Flops) (physical registers). Otherwise we will use “blocking” ( `=` ) assignments. There are many complex reasons to this that your TA or Instructor might be able to help you see. For now we will use this as a general rule.

### 4.1.3 Output Function Logic (OFL)

To finish our design we need to implement the output function logic. Recall from your earlier lab that we generated 4 “group” signals (GLL, GL, GR, GRR). Each controlled a “group” of 2 LEDs. In this lab we have very simple OFL (simple combinational logic) and handle each in a single line using `assign`:

```
assign GLL = q_L123;
assign GL  = q_L123 | q_L12 | q_R1 | q_R12 | q_R123;
assign GR  = q_R123 | q_R12 | q_L1 | q_L12 | q_L123;
assign GRR = q_R123;
```

Notice that these lines exist OUTSIDE of an `always` block. The `assign` statements are concurrent. Each statement defines a part of the combinational logic. Collectively, they define the OFL.

## 4.2 Module: ee254\_detour\_top

Our top level design will also be a module. Just like the state machine we will have a number of “ports” (inputs = buttons, switches, Clk and outputs = LEDs, SSDs, etc.). Our usage of input output resources on the Nexys 3 board is identical to the schematic design except for LD2 as shown below.

### 4.2.1 Global Signals and Clock Division

Before we can use the 100MHz clock we need to use a `BUFPGP` to buffer and produce the global clock signal using:

```
BUFPGP BUFPGP1 (board_clk, ClkPort);
```

This creates a BUFGP with input=ClkPort and output=board\_clk. The label BUFGP1 gives this particular BUFGP a name (called an “instance name”).

In the previous labs we used a cascade of wide (8-bit and 16-bit) counters to divide the board clock. In Verilog we practice “behavioral” coding by describing the input-output relations and letting the Synthesizer choose the best way to achieve this. For instance the following implements the clock divider by creating a counter:

```
always @ (posedge board_clk, posedge reset)
begin : CLOCK_DIVIDER
    if (reset)
        DIV_CLK <= 0;
    else
        DIV_CLK <= DIV_CLK + 1'b1;
    end

    assign sys_clk = DIV_CLK[25];
```

The final line ties sys\_clk to the DIV\_CLK[25] (~ 1.5Hz).

## 4.2.2 Switch Inputs and LED Outputs

The following lines connect (assign) the signals from our design to the output ports for our top module:

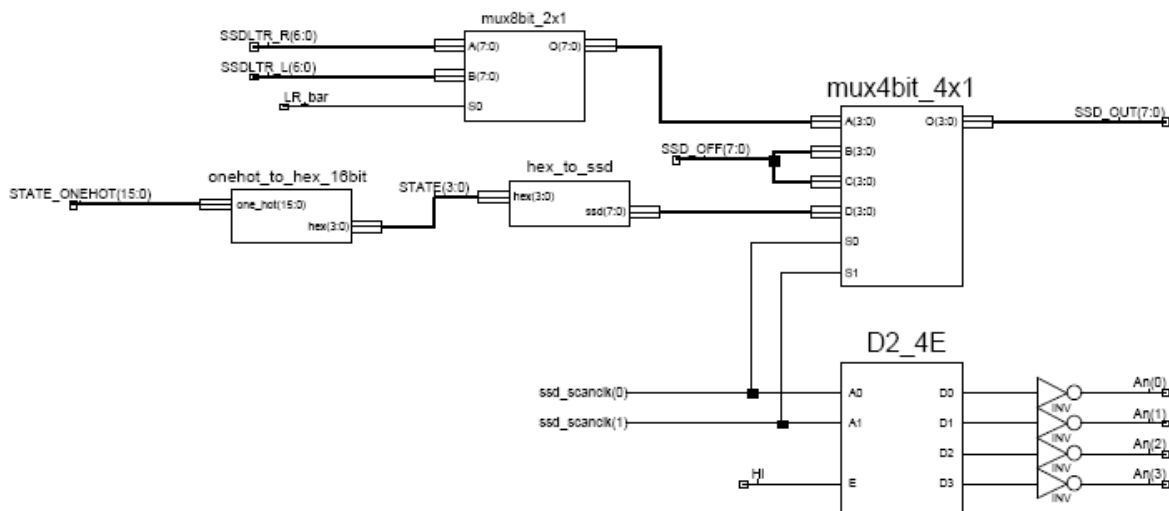
```
wire L_Rbar;
assign L_Rbar = Sw0;

assign {Ld0,Ld1} = {GRR, GRR};
assign {Ld2,Ld3} = {GR, GR};
...
```

You should use these as a template for your number-lock design. For this lab you can just think of these assign statements like buffers in your schematic. They emulate having a net with multiple labels (i.e. aliases).

### 4.2.3 SSD Outputs

One way to drive the SSDs in the detour schematic lab is shown below. This design is translated into Verilog in your `ee254_detour_top.v`



We have to interpret the 1-hot signals from our state machine before we can display them on the SSD. First we needed to **encode** the state information ( $q_I = 0000$ ,  $q_{L1} = 0001$ ,  $q_{L2} = 0010$ , etc.). The Verilog sample does this in the `always` block labeled `ONE_HOT_TO_HEX`:

```
reg [3:0] state_num;

always @ ( q_I, q_L1, q_L12, q_L123, q_R1, q_R12, q_R123 )
begin : ONE_HOT_TO_HEX
    case ( {q_I, q_L1, q_L12, q_L123, q_R1, q_R12, q_R123} )
        7'b1000000: state_num = `QI_NUM;
        7'b0100000: state_num = `QL1_NUM;
        7'b0010000: state_num = `QL12_NUM;
        ...
    endcase
end
```

Notice the sensitivity list includes all of the states. This will cause the simulator (or FPGA) to invoke this code anytime the state changes. Also notice that we needed to create a “`reg [3:0] state_num;`” to store the result. If you have experience with C/C++ or Java you can think of these statements like variable declarations in those languages.

Before we send the hex digit to the SSD we need to convert it into the appropriate combination of on’s and off’s (i.e. for 3 we need Ca, Cb, Cc, Cd, Cg). We used the device `hex_to_ssd` in the earlier lab (see SSD schematic figure above) to convert the 4-digit hex code to 7-digit SSD. In



Verilog we implement this block in the form of a big case statement. We set the block to trigger anytime the `state_num` changes and then match the input to the desired output like:

```
reg [6:0]  SSD_STATENUM;
// Following is Hex-to-SSD conversion
always @ (state_num)
begin : HEX_TO_SSD
    case (state_num)
        4'b0000: SSD_STATENUM = 7'b0000001 ; // 0
        4'b0001: SSD_STATENUM = 7'b1001111 ; // 1
        4'b0010: SSD_STATENUM = 7'b0010010 ; // 2
        ...
    endcase
end
```

Find the representation of the 2x1 mux from your schematic that selected L or R to go to the SSD.

Because the detour lab uses only 2 of the SSD digits we need to make sure the other two get turned off. We defined an alias called `SSD_OFF` for this purpose. Remember SSD cathodes are active LOW. ``define SSD_OFF 7'b1111111`

We still have to worry about the “scanning” procedure to deal with the multiplexing of the SSD controls. We choose `ssdscan_clk = DIV_CLK[19:18]` to scan over the SSDs. The `An0`, `An1`, ... signals are just combinational logic (`An0=0` when `ssdscan_clk = 00`, `An1=1` when `ssdscan_clk = 01`, etc.) and the following lines take care of that nicely:

```
assign An0 = !(~(ssdscan_clk[1]) && ~(ssdscan_clk[0])); // 00
assign An1 = !(~(ssdscan_clk[1]) && (ssdscan_clk[0])); // 01
...
```

To finish the SSD implementation we need to sync the proper SSD data as we scan through `An0`, `An1`, `An2`, `An3` using a 4x1 mux:

```
always @ (ssdscan_clk, SSD0, SSD1, SSD2, SSD3)
begin : SSD_SCAN_OUT
    case (ssdscan_clk)
        2'b00: SSD = SSD0;
        2'b01: SSD = SSD1;
        2'b10: SSD = SSD2;
        2'b11: SSD = SSD3;
    endcase
end
```

By including the `SSD0-SSD3` signals in the sensitivity list we make sure that the digit updates even if it changes during the middle of a clock. This illustrates putting all the input signals our combinational (mux) block uses into the sensitivity list (also called event expression) of the `always` construct.

## 5. Prelab:

- Q 5. 1: Fix the following code by removing the `initial` block from the synthesizable code and instead use `reset` to clear the D-FF. (5pts)

```
module D_FF(Clk, reset, D, Q);
    input Clk, reset, D;

    output Q;
    reg Q

    initial
    begin
        Q = 0;
    end

    always @ (posedge Clk)
    begin
        Q <= D;
    end
endmodule
```

- Q 5. 2: Finish the code implementing the voting machine design (you may use NAND-NAND or AND-OR). (3pts)

```
module ee254_voting(A, B, C, D, Result);
    input A, B, C, D;

    output Result;

    assign Result = _____;
endmodule
```

- Q 5. 3: We use 7-bits to control the SSD cathodes. We could choose to turn each off or on individually but this is cumbersome and error prone. Finish the code below that assigns the 7 cathodes to a vector (CATHODES). Refer to the use of concatenate operator on page 107 of Cadence (Esperan) Verilog guide. (2pts)

```
reg Ca, Cb, Cc, Cd, Ce, Cf, Cg;

wire [_____:0] CATHODES;

assign CATHODES = _____;
```

- 5.3.B: Now rewrite the assign statement so that the CATHODES always displays “3”. (2pts)

```
`define SSD_NUM_3 _____

assign CATHODES = _____;
```

- Q 5. 4: Finish the code to initialize the flag registers x,y, and z to “0” in a SINGLE statement. Again refer to the use of concatenate operator on page 107 of Cadence (Esperan) Verilog guide.(3pts)

```
reg x,y,z;
always @ (posedge Clk, posedge reset)
begin
    if(reset)
        _____
    else
        X <= X | FoundX;
        Y <= Y | FoundY;
        Z <= Z | FoundZ;
end
```

## 6. Procedure:

### Part 0: Detour Lab Sample Verilog Design

6.1 Download the zip file containing the example Xilinx ISE project. Extract the zipped project directory into the projects folder (`C:\xilinx_projects\`). Open the project in the Xilinx Project Navigator. The example project implements the Detour Lab design entirely in Verilog. Notice the basic layout consists of two files: (1) a top design (`ee254_detour_top.v`) and (2) the detour state machine (`ee254_detour_sm.v`).

6.2 Synthesize the design like you have in previous labs. Transfer the `.bit` file to the FPGA board and confirm the implementation matches the earlier detour lab.

### Part 1: Completing the number-lock state machine

6.3 Create a new project. Select “**HDL**” from the “Top-Level Source Type” and give the project a good name (`ee254l_number_lock_verilog`). And proceed to the “Create New Source” Page. Add a new “Verilog Module” source called “`ee254_numlock_sm`”. You do not need to modify the options in the wizard so click through to finish. Note: Some of us prefer to use **Notepad++** to create the `ee254_numlock_sm.v` as it provides better text editing features.

6.4 You will implement the number lock state machine in the source file you just created. You should refer to the state machine sample from Part 0. We will design this portion first and test the design with ModelSim.

6.5 Recall the input and output wires you used on the state machine block implemented during the earlier detour lab. To refresh:

- Inputs: `clk`, `reset`, `U` (Uno), `z` (Zero)
- Outputs: `unlock`, and also a wire for each state

You must add each input and output to the port list of the `ee254_numlock_sm` module (inside the parentheses like in the example). This will allow you to access these signals from your top design.

6.6 To add the state memory (remember 1 FF / state) lets use “`reg [10:0] state`” vector. But we want our design to send these signals out by name to our top level. So use an `assign` statement like in the example. Use `localparam` to make symbolic names stand for the bit-coded states to make changing states easier.

6.7 Implement the NSL for the state machine in an `always` block. Make sure you consider the start conditions (i.e. during `reset`). Most of the conditions from the Detour State Machine example were unconditional so the code you write will be a little more complex. You should implement the first 2 or 3 state transitions and show your TA. Save writing the transitions from the `OPENING` state until later. Some starting help is given below.

```

case(state)
  QI:
    if (UZ == 2'b10)
      state <= QG1GET;
    // another way to say the above (but not quite preferred) is
    // state <= (UZ == 10) ? QG1GET : QI ;
  QG1GET:
    if (U == 0)
      state <= QG1;
  ....
endcase

```

6.8 We can use the `assign` statement to generate the OFL. You should refer to page 103 of the Cadence guide for the list of possible bitwise operations including AND, NOT, XOR, and OR.

6.9 Now you can finish the design by handling the `OPENING->INITIAL` transition. These transitions require a little more thought because they require a trigger (Timerout) to transition. We want to design a counter that increments every clock but only while in the `OPENING` state:

a) How many clocks do we want to wait? How many bit counter should we use? Create a new reg vector called `Timerout_count` to hold the count.

b) Now create a new `always` block. What should your sensitivity list contain?

c) Instead of using a structural adder component, let's use a behavioral implementation. Use the '+' operator to give `Timerout_count <= Timerout_count + 1;`. If we accidentally use a blocking operator for this step will the circuit function (though perhaps non-ideally)? Add lines to this block to ensure `Timerout_count` starts at zero when you enter `q_Opening`. How do we do this? (hint: when do we **NOT** want to count and in all those states can we keep clearing the counters synchronously?). Do we need an asynchronous clear on this counter controlled by `reset` signal? If it is not necessary, you should avoid it!

6.10 You just finished your first Verilog module! Let's debug it. Create a new "Verilog Test Fixture" called `ee254_numlock_sm_tb`. Make sure you associate the test fixture with your state machine. First test to make sure your circuit responds to a good combination "U-Z-U-U". Something like:

```

U = 0; Z = 0;

#100;
U = 1; Z = 0;

#20;
U = 0; Z = 1;

#20;
U = 1; Z = 0;

```

```
#20;
U = 1; Z = 0;
```

Think carefully. Did you provide a stimulus in the above testbench to release both buttons before operating another button? Also, are going faster than your clock?

Test other combinations and verify that your design handles BAD combinations also.

6.11 Now simulate your ee254\_numlock\_sm\_tb behaviorally to test your state machine design. Check your waveform and confirm your implementation. NOTE:



**HINT:** The waveform automatically shows the inputs and outputs from your module. What if we want to look at an internal signal like timerout? You should use the waveform browser (objects panel) and drag that signal over. But wait... no data shows up? That is because Xilinx only records the module I/O signals by default. We can tell Xilinx to record ALL signals by editing .udo file. Open ee254\_numlock\_sm\_tb\_v.udo in Notepad++. At the end add: "log -r /\*" on a new line. Then in the .fdo file move "do {ee254\_numlock\_sm\_tb\_v.udo}" above the "run 1us" line. Now re-run the simulation in ModelSim and adding signals to the waveform will show you the history!

An alternative is to write a .do file. Let us call it ee254\_numlock\_sm.do . After arranging what you need in the waveform in what order and radix, etc. save the waveform format in a wave.do file or preferably named ee254\_numlock\_sm\_wave.do and invoke it in as shown ee254\_numlock\_sm.do below.

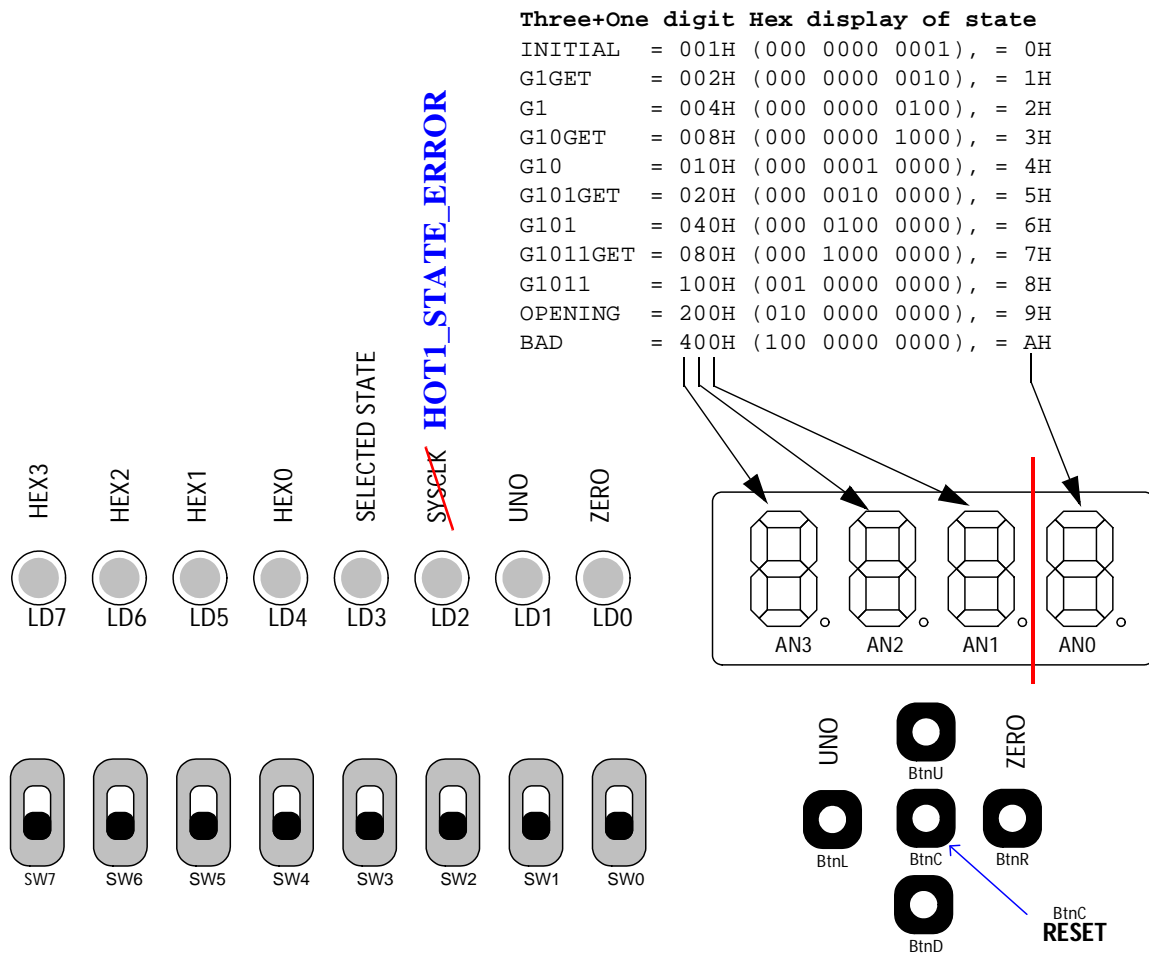
```
# ee254_numlock_sm.do

quit -sim

vlib work
vlog +acc "ee254_numlock_sm.v"
vlog +acc "ee254_numlock_sm_tb.v"
# vsim work.ee254_numlock_sm_tb
vsim -novopt -t lps -lib work ee254_numlock_sm_tb
do {ee254_numlock_sm_wave.do}
view wave
view structure
view signals
log -r *
run 2 us
WaveRestoreZoom {0 ps} {2000 ns}
```

## Part 2: Completing the “top” design

Reproduced below is the Nexys-3 I/O resources allocation diagram from your schematic lab. There is one important change. The LD2 is used indicate if the state machine goes into any illegal state (other than the legal one-hot states).



Recall (or look at) your schematic design and code your `ee254_numlock_top.v`

We will refer heavily to the example Verilog project top (`ee254_detour_top.v`) for this part. In this part you will create a module that you can put directly onto the FPGA. We will give you a skeleton top file `ee254_numlock_top.v` for you to modify.

6.12 Add the source `ee254_numlock_top.v` to your project. Look through the skeleton top file and identify the major portions (clock division, inputs, outputs, SSD display, etc).

6.13 Use the “find” feature and search for the lines with “TODO”. These sections require completion.

6.14 Before synthesizing your design we need to add a `.ucf` file to define the I/O connections (constraints). Copy the `.ucf` file (user constraint file) from the detour sample project to your number lock project directory. Now add the `.ucf` file (“Add Source”) to your project. Comment out or

uncomment the necessary lines in this file so that each of the inputs and outputs in your top level is tied to a pin. This is a 4-minute task. Try not to copy .ucf file given with the schematic lab.

6.15 Before implementing, do

Generate Programming File => Process Properties => Startup Options =>  
FPGA Start-Up Clock => JTAG Clock

You can now synthesize and implement your design and then produce .bit file. Connect your FPGA to the computer and configure the FPGA using Digilent's Adept tool. Demonstrate it to your TA.



## 7. Lab Report:

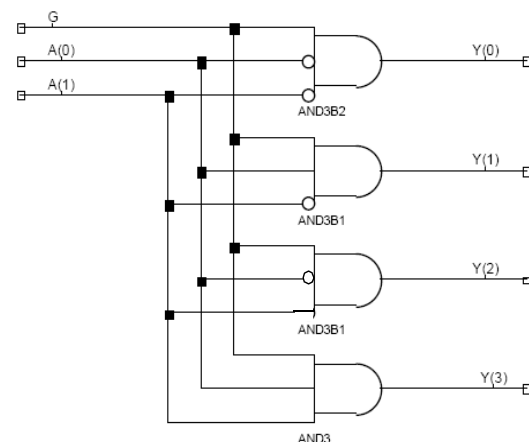
Name: _____	Date: _____
Lab Session: _____	TA's Signature: _____

<b>For TAs:</b> Pre-lab (15): _____ Implementation (25): _____ Report (out of 60): _____
Comments:

Q 7. 1: Attach your Verilog files (ee254\_numlock\_top, ee254\_numlock\_sm) and waveform from part II. (25pts = 10+10+5)

Q 7. 2: What is wrong with the following `always` combinational block implementing a 2x4 decoder? How would you fix it? (6pts = 3+3)

```
always @ (A)
begin
    Y[3] = ~A[1] * ~A[0] * G;
    Y[2] = ~A[1] * A[0] * G;
    Y[1] = A[1] * ~A[0] * G;
    Y[0] = A[1] * A[0] * G;
end
```



Q 7. 3: Refer to the 4-bit counter you implemented to generate `Timeout_count`. If you use a blocking assignment instead of a non-blocking assignment will (circle the answer and explain your answer below). Consider in your explanation, what happens if you coded the counter not in a separate `always` block but in the main `always` block for the state machine.: (4pts)

- A. not operate
- B. operate, but this method is not preferred
- C. operate, this is the preferred method

Q 7. 4: What is wrong with the following block of code intended to implement a 4-bit counter? Write your corrected version to the below. (4pts = 2+2)

```
// assume sys_clk & reset were already defined above
reg count;
always @ (posedge sys_clk, posedge reset)
begin : COUNTER4BIT
    count = count + 1;
end
```

Q 7. 5: What is one benefit to labeling `always` blocks (such as `COUNTER4BIT` in the previous question)? (2pts)

Q 7. 6: After you run your simulation you find a few errors. You want to check a few internal signals within your state machine module. After finding the signals in the signal browser you drag them over to the waveform but ModelSim reports “No Data”. What should you do? (4pts)

Q 7. 7: While implementing the Hex2SSD conversion in your schematic we used 7 separate 16x1 bit ROMs (1 representing each cathode). In our Verilog code we use a "case" statement. But this looks a lot like a multiplexor (see the 7-bit 4x1 Mux in the detour lab example). How can that be? (hint: is there really a difference from an I/O perspective? Explain?). (6pts)

Q 7. 8: While debugging Mr. Bruin's design (Bruin is one of your junior engineers), you localize the problem to a few lines. Mr. Bruin needs to save a decremented version of an 8-bit number X (i.e.  $Y=X-1$ ) using 2's complement addition, i.e.  $Y = X + (-1)$ . But on the waveform, you see that Y is showing as per  $Y=X+1$ . What is wrong with his code and how would you correct it? (do NOT change the " $Y = X + \text{offset}$ " line). (6pts)

```
module f1(X, Y);
    input [7:0] X;

    output Y;
    reg [7:0] Y;

    wire offset;

    `define NEG_1 8'b11111111
    assign offset = `NEG_1;

    always @ (X)
    begin : DECREMENT_X
        Y = X + offset;
    end
endmodule
```