

ECE 2700 Lab 2

Combinational Logic and Seven Segment Displays

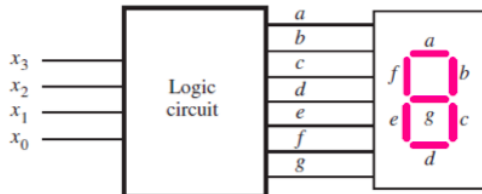
Due at the end of your registered lab session (120 points)

Objectives

- Design using truth tables and combinational logic.
- Derive efficient Boolean expressions from Truth Tables.
- Design and verify modules with multi-bit signals.
- Display information on a seven-segment display.
- Utilize hierarchy to simplify design of a basic system.
- Examine alternative Verilog syntax for Combinational and Truth-Table designs:
 - if-then vs case statements
 - always/reg vs assign/wire
 - individual signals vs vector assignments
 - single-bit logic operators (!, &&, ||)
 - bitwise logic operators (~, |, &, ^)
 - concatenation ({x}) and repetition ({7{x}}) operators

1 Pre-Lab Preparation

We will be implementing the seven-segment display using multiple techniques in this lab. The circuit view and the truth table from Figure 2.63 (a) and (b) of the textbook are shown below.



(a) Logic circuit and 7-segment display

	x_3	x_2	x_1	x_0	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

(b) Truth table

We can use our knowledge of *sum-of-products* (SOP), *product-of-sums* (POS), and Boolean algebra to derive and minimize expressions for each output signal from a to g . For example, study the truth table for signal a .

x_3	x_2	x_1	x_0	a
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1

Notice that **only two positions have zero values**. This suggests that using the POS form make a shorter expression for a .

$$\begin{aligned}
 a &= (x_3 + x_2 + x_1 + \bar{x}_0)(x_3 + \bar{x}_2 + x_1 + x_0) \\
 &= ((x_3 + x_1) + x_2 + \bar{x}_0)((x_3 + x_1) + \bar{x}_2 + x_0) && 10b. \text{ (Commutative)} \\
 &= (x_3 + x_1) + (x_2 + \bar{x}_0)(\bar{x}_2 + x_0) && 12b. \text{ (Distributive)} \\
 &= (x_3 + x_1) + (x_2\bar{x}_2 + \bar{x}_0\bar{x}_2 + x_2x_0 + \bar{x}_0x_0) && 12a. \text{ (Distributive)} \\
 &= (x_3 + x_1) + (0 + \bar{x}_0\bar{x}_2 + x_2x_0 + 0) && 8a \\
 &= x_3 + x_1 + x_0x_2 + \bar{x}_0\bar{x}_2 && 10a. \text{ (Commutative)}
 \end{aligned}$$

We can apply the same method to get a simplified expression for b :

$$b = \bar{x}_2 + x_1x_0 + \bar{x}_1\bar{x}_0$$

Your Assignment:

1. Using this technique, find minimal expressions for d , e , and f , and **verify your expression against the given truth table**. The remaining expressions for segments c and g will be provided for you.
2. Note that for four input signals, there should be $2^4 = 16$ input combinations. The truth table above is missing the last six rows. For these rows, the inputs do not represent a decimal digit and should be considered **invalid**. We will define a signal named **NAN** (short for “Not a Number”) to detect these cases. Derive a minimized logic expression for **NAN**. Verify that your expression detects **only digits greater than nine**.
3. Read all the Verilog syntax discussion in this document before coming your lab session.

2 Overview

In this lab, you will implement three seven-segment display modules using three different design techniques. Each of your modules will drive one digit of the Basys3 board’s seven-segment displays. In the first exercise, you will use the clock divider to slowly increment the number displayed on each digit; for this purpose, you will learn the clocked logic in Section 3 first, and then reuse this clock divider for this lab.

3 Introduction to Clocked Logic

Now let’s shift attention and experiment with the board’s built-in clock resource. The Basys3 has a system clock rate with frequency 100 MHz. In this exercise, we will create a **clock divider** module to

slow down the clock so that we can observe step-by-step events. We will reduce the clock rate to 2 Hz, i.e. two events per second, so that you can actually see the clock tick. A slow, observable clock can be useful for studying and debugging sequential logic circuits.

3.1 Design

In Xilinx Vivado, create a New Project named **ClockDivider**. Follow the wizard steps described in the first part of this lab, and create a new Verilog Design Source for a module named **ClockDivider.v**. Your module should have one input and one output, and the initial template code should look like this (header comments are not shown):

```
'timescale 1 ns/1 ns

module ClockDivider #(parameter PRESCALER = 25_000_000)(
    input clk_in,
    output reg clk_out // <--- add the "reg" keyword here
);

endmodule
```

Here is a quick note about the **parameter** keyword. It is placed between the module name and the opening parenthesis for the port declarations with its own set of parentheses with a leading **#** symbol. The keyword **parameter** is used to provide different customizations to a module instance. A default value can be assigned to the **parameter**, here, 25 million is used as the default. Later we will override this value in the simulation testbench. For a hierarchical design, parameters can be passed down from a top-level Verilog module to the instantiations of lower-level modules within the top-level module. It is similar to the **#define** statement in C++.

Notice that the keyword **reg** is inserted in the declaration of **clk_out**. This means that the **clk_out** signal will be defined *behaviorally* in an **always** block.

To divide the clock rate, we'll use a simple counter method. We will declare a **count** variable, initialize it at zero, and increment by one in each cycle of **clk_in**. Once the count adds up to a divisor **N**, we will flip **clk_out**. Then the frequency of **clk_out** should be

$$f_{\text{out}} = \frac{f_{\text{in}}}{2N}.$$

If the input frequency is $f_{\text{in}} = 100 \times 10^6$ Hz, then to get an output frequency of 2 Hz we need $N = 25 \times 10^6$. To represent such a big number in Verilog, we first need to know how many bits are required, which is

$$\lceil \log_2 N \rceil = 25 \text{ bits}.$$

To implement the clock divider, we add these lines into the module definition:

```
reg [24:0] count;

initial begin
    count = 0;
    clk_out = 0;
end

always @(posedge clk_in) begin
    if (count == PRESCALER) begin
        count <= 0;
        clk_out <= ~clk_out;
    end
    else begin
        count <= count + 1;
    end
end
```

```

                end
            end
end

```

In this code, the 25-bit variable `count` is treated, by default, as an integer. The code in the `always` block is executed **synchronously** with the rising edge of the input clock. At each clock, `count` is incremented by 1. When `count` reaches 25×10^6 , the PRESCALER value, `clkout` is **toggled** (the `~` symbol means “not”).

3.2 Simulate

Now we will create a **testbench** for the `ClockDivider` module. For simulation purposes, it will be easier to reduce N to a small number, such as eight. To do this, we are going to override the default parameter we gave to the `ClockDivider` module. This can be done like assigning ports: use a dot followed by the parameter name with the value you want to pass in enclosed parentheses. Now instead of having a value of 25 million, it will simulate with a value of 8 so that we do not have to simulate millions of clock cycles to verify the design. Parameters are a nice way to configure modules. The value 8 is only used in simulation. When we implement the design on the board, it will remain at 25 million. If we did not use the parameter altogether, then you would have to remember to comment out the existing value in the `ClockDivider.v` module and replace it with a smaller value. Then when you are ready to implement the design, you would have to remember to switch which lines were previously commented out. If you forgot to do so, the LED would appear to be always on, as it is blinking too fast.

Save the file, then click on **Add Sources**, select **Add Simulation Sources** and follow the procedures for making a simulation testbench. Open your new testbench file. You now need to setup the input clock. The easiest way is to define an infinite loop using the `forever` keyword in an `initial` block:

```

module testbench(
);
    reg clk;
    wire slow_clk;

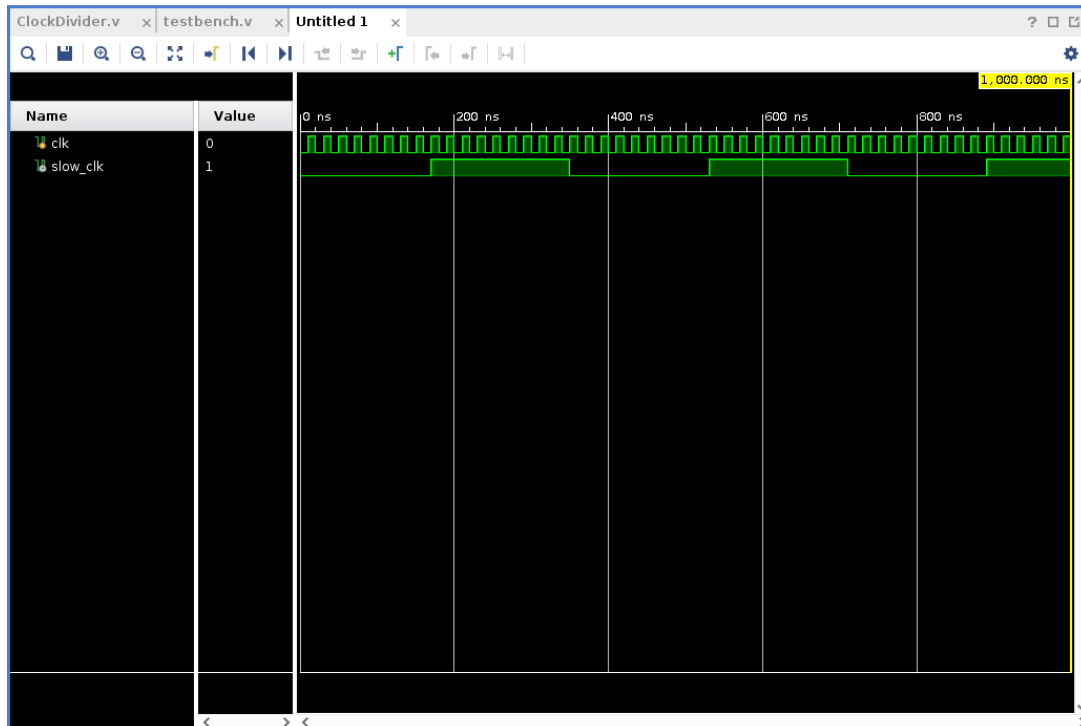
    // Instantiate the module to be tested.
    // Here we demonstrate named port connections
    // as an alternative to ordered port connections:
    // Also, the parameter is given a value of 8 instead
    // of 25 million
    ClockDivider #(.PRESCALER(8)) DUT(
        .clkin(clk),
        .clkout(slow_clk)
    );

    // Define the clock signal using the forever keyword:
    initial begin
        clk = 0;

        forever #10 clk = ~clk;
    end
end

```

Save your testbench file and run a behavioral simulation as we did before. In the waveform viewer, click the zoom-to-fit icon to see your simulation range. It should like this:



Now verify that `slow_clk` flips every nine cycles of `clk`. (just like arrays counting starts at zero. For this reason you will see nine rising clock edges not eight)

3.3 Implement

Now we need to configure the **Constraint File** for implementation. You should have already added **Basys3_Master.xdc** as a constraint to your project; if not, do so now. Open the constraint file. We will need to find lines that define the system clock, named `clk` in the default configuration. We will also need to associate `LED[0]` with `clkout`. Your completed constraint file should look like this:

```
## Clock signal
set_property PACKAGE_PIN W5 [get_ports clk_in]
set_property IOSTANDARD LVCMOS33 [get_ports clk_in]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk_in]

##LEDs
set_property PACKAGE_PIN U16 [get_ports {clkout}]
set_property IOSTANDARD LVCMOS33 [get_ports {clkout}]
```

You must be very careful to ensure that the names referenced with the `get_ports` keyword match precisely with the signal names in your design. Once your constraint is complete, run Synthesis, Implementation and Generate Bitstream. Program your device and verify that the LED blinks about twice per second. **Demonstrate your result to the TA.** You may need to program the Flash and carry your board to the TA in order to check it off.

4 Seven Segment Displays

In Example 2.15 of the textbook, you are asked to consider the relationship between a binary-coded decimal (BCD) number and the corresponding lights on a seven-segment display. The BCD digits are labeled x_3, x_2, x_1 and x_0 (with x_3 as the most significant bit), representing a number between 0 and 9. The display's individual segments are labeled a, b, \dots, g , with each signal corresponding to one illuminated edge in the display. The above truth table is used for a standard display.

There are a few different ways to model a truth table in Verilog. In each method, we will use an **always block**

```
always @(*) begin
    ...
end
```

In this syntax, the `@(*)` syntax declares implicit sensitivity. It literally means “execute the code in this block whenever *any* of the referenced signals change.” This type of sensitivity list is appropriate for **combinational logic**, and in this type of block we should usually use **blocking assignment** statements.

Within the always block, to define a truth table we need to explicitly define the outputs for every single pattern of inputs. There are several different syntax methods available to us; the most common methods are **if-else** and **case** statements:

1. **If-Else:** A C-style ‘==’ is used to detect **logical equality**

```
if (~x3 & ~x2 & ~ x1 & ~x0) begin
    a = 1; b=1; c=1; d=1; e=1; f=1; g=0;
end
else if (~x3 & ~x2 & ~ x1 & x0) begin
    a = 0; b=1; c=1; d=0; e=0; f=0; g=0;
end
//... and so on ...
// be sure to provide a generic "else" to catch any
// invalid or unexpected input patterns:
else begin
    a = 0; b=0; c=0; d=0; e=0; f=0; g=0;
end
```

2. **Compact If-Else:** before the always block, define bit vectors for the input and output signals:

```
wire [3:0] N = {x3,x2,x1,x0}; // Pack inputs into one signal
```

Then in the always block, you don’t need to type as much:

```
if (N==4'b0000) begin
    a = 1; b=1; c=1; d=1; e=1; f=1; g=0;
end
else if (N==4'b0001) begin
    a = 0; b=1; c=1; d=0; e=0; f=0; g=0;
end
//... and so on ...
else begin
    a = 0; b=0; c=0; d=0; e=0; f=0; g=0;
end
```

3. You can also create a compact signal for the output:

```
wire [3:0] N = {x3,x2,x1,x0}; // Pack inputs into a single signal
reg [6:0] D;
assign a = D[6];
assign b = D[5];
assign c = D[4];
```

```

assign d = D[3];
assign e = D[2];
assign f = D[1];
assign g = D[0];

always @(*) begin
    if (N==4'b0000) begin
        D = 7'b11111110;
    end
    else if (N==4'b0001) begin
        D = 7'b01100000;
    end
    //... and so on ...
    else begin
        D=0;
    end
end
end

```

4. **Compact Case statement:** the most efficient way to encode a truth table is with a case statement:

```

always @(*) begin
    case (N)
        4'b0000: D = 7'b11111110;
        4'b0001: D = 7'b01100000;
        4'b0010: D = 7'b1101101;
        4'b0011: D = 7'b1111001;
        4'b0100: D = 7'b0110011;
        4'b0101: D = 7'b1011011;
        4'b0110: D = 7'b1011111;
        4'b0111: D = 7'b1110000;
        4'b1000: D = 7'b1111111;
        4'b1001: D = 7'b1111011;
        // Always provide a "default" case to catch
        // unexpected or invalid inputs:
        default: D = 7'b00000000;
    endcase
end

```

5. **Boolean Expressions:** instead of entering the truth table contents (which can become very complex when many inputs are used), you can enter Boolean expressions for the output signals. The basic syntax is:

Bitwise Operators		Logic Operators	
Symbol	Operation	Symbol	Operation
&	AND	&&	AND
~	NOT	!	NOT
	OR		OR
^	XOR		

As a general rule, the **bitwise operators** are used for **Boolean expressions**, whereas the **logic operators** are used for **if/then expressions or other conditionals**. Using this method, the Boolean expressions from your pre-lab assignment can be entered directly into the always block:

```

a = x3 | x1 | x2&x0 | ~x2&~x0;
b = ~x2 | x1&x0 | ~x1&~x0;
c = x3 | x2 | ~x1 | x0;
d = // Provide your solutions to d, e, and f
e = //
f = //
g = x1&~x0 | x2&~x1 | x3 | ~x2&x1;

```

6. **Assign statements:** in a purely combinational design defined by Boolean expressions, you don't need to use an always block at all. You can instead use **assign** statements:

```

assign a = x3 | x1 | x2&x0 | ~x2&~x0;
assign b = ~x2 | x1&x0 | ~x1&~x0;
assign c = x3 | x2 | ~x1 | x0;
assign d = // Provide your solutions to d, e, and f
assign e = //
assign f = //
assign g = x1&~x0 | x2&~x1 | x3 | ~x2&x1;

```

Note: any signal that is assigned within an always block should be declared as a **reg** or **output reg** type, whereas any signal defined by an assign statement should be declared as a **wire** or **output** type (output ports are considered wires by default unless declared as reg type).

5 Demonstration Design

Now create a project in Vivado and define three Verilog modules:

1. **SevenSegmentTruthTable.v** (see section 4, part 4: Compact Case Statement for starter code)
2. **SevenSegmentCombinational.v** (see section 4, part 6: Assign Statements for starter code)
3. **SevenSegmentTop.v**

In the TruthTable and Combinational designs, you should specify the following I/O signals:

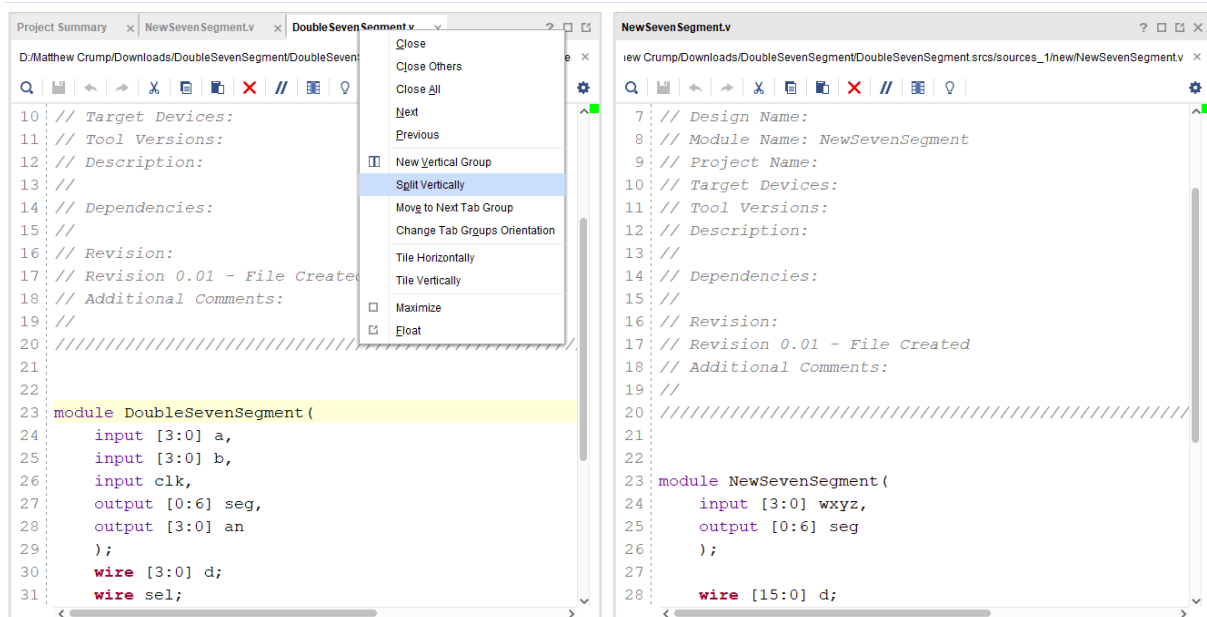
- inputs x3, x2, x1, x0
- outputs a, b, c, d, e, f, g

Complete the Verilog code for the TruthTable and Combinational modules using the methods described in the Section 4. Specifically, use compact case statements for the truth table, and use Boolean expressions for the combinational assignments.

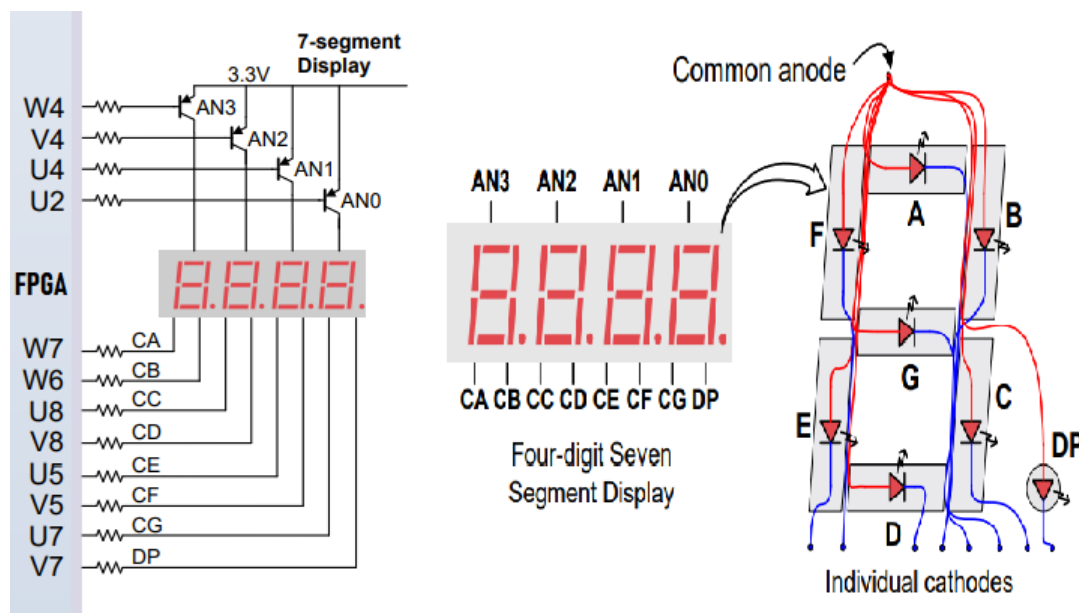
In the Top module, you should specify these I/Os:

- input [7:0] sw
- output [3:0] an
- output [6:0] seg

Having a hard time remembering what the port names were in another module? You can split the source windows by right-clicking the tabs then select **Split Vertically**. This makes it much easier to type out the ports for a submodule without having to go back and forth between the two modules.



On the Basys 3 boards, the seven segment signals are **active low**, meaning '0' is ON and '1' is OFF. See the following schematic for why that is. To illuminate a segment, the anode should be driven high while the cathode is driven low. However, since the Basys 3 uses transistors to drive enough current into the common anode point, the anode enables are inverted. Therefore, both the AN0..3 and the CA..G/DP signals are driven low when active.



We therefore need to invert the signal coming from your module. Additionally, the four **an** signal bits (also active low) determine which of the seven segment display digits are activated – only one digit can be displayed at a time. For now, we will assign the **an** bits directly via switches 4 through 7 on the board. We will assign the **x3**, **x2**, **x1**, **x0** bits from the four right-most switches (0 through 3) on the board. Putting it all together, the Top module should look like this:

```

module SevenSegmentTop(
    output [6:0] seg,
    output [3:0] an,

```

```

    input [7:0] sw
);

    wire [6:0] D;

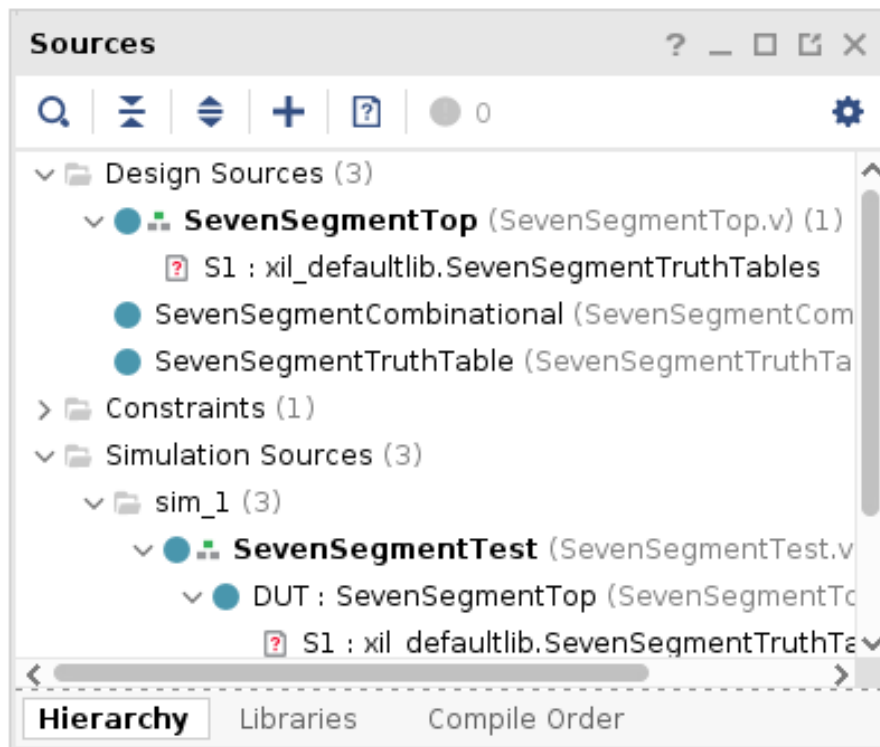
    assign seg = ~D;
    assign an  = ~sw[7:4];

//    SevenSegmentTruthTable S1(
    SevenSegmentCombinational S1(
        .x3(sw[3]),
        .x2(sw[2]),
        .x1(sw[1]),
        .x0(sw[0]),
        .a(D[0]),
        .b(D[1]),
        .c(D[2]),
        .d(D[3]),
        .e(D[4]),
        .f(D[5]),
        .g(D[6])
    );
endmodule

```

Notice that both the TruthTable and Combinational versions of the module are referenced. We will test both versions of the design. We can simply change the line comments to switch which version we want to use.

A useful tip: the hierarchy tree in the source window is very helpful for ensuring that submodules are in the correct place. In the following screenshot, we purposely made a mistake and misspelled the `SevenSegmentTruthTable.v` module in the top module. Notice how in the source window that the module shows up red with a question mark. This means that either the module does not exist or, more likely, you have made a typo in the top module. Once the typo has been fixed, they will show up blue under the module that they are instantiated in.



Next, to verify your design, create a new Verilog testbench module and instantiate your Top module as the Design Under Test. In this test, you will create a new `reg` signal named `clk`, and use it to trigger an always block that sweeps the `sw` signal from integer value zero, up to nine, and then back to zero.

```
module SevenSegmentTest;

    // Inputs
    reg [7:0] sw;
    reg      clk;

    // Outputs
    wire [6:0] seg;
    wire [3:0] an;

    // Instantiate the Design Under Test (DUT)
    SevenSegmentTop DUT (
        .seg(seg),
        .an(an),
        .sw(sw)
    );

    initial begin
        // Initialize Inputs
        sw = 0;
        clk = 0;

        // Wait 100 ns for global reset to finish
        #100;
        forever #10 clk = ~clk;
    end
end
```

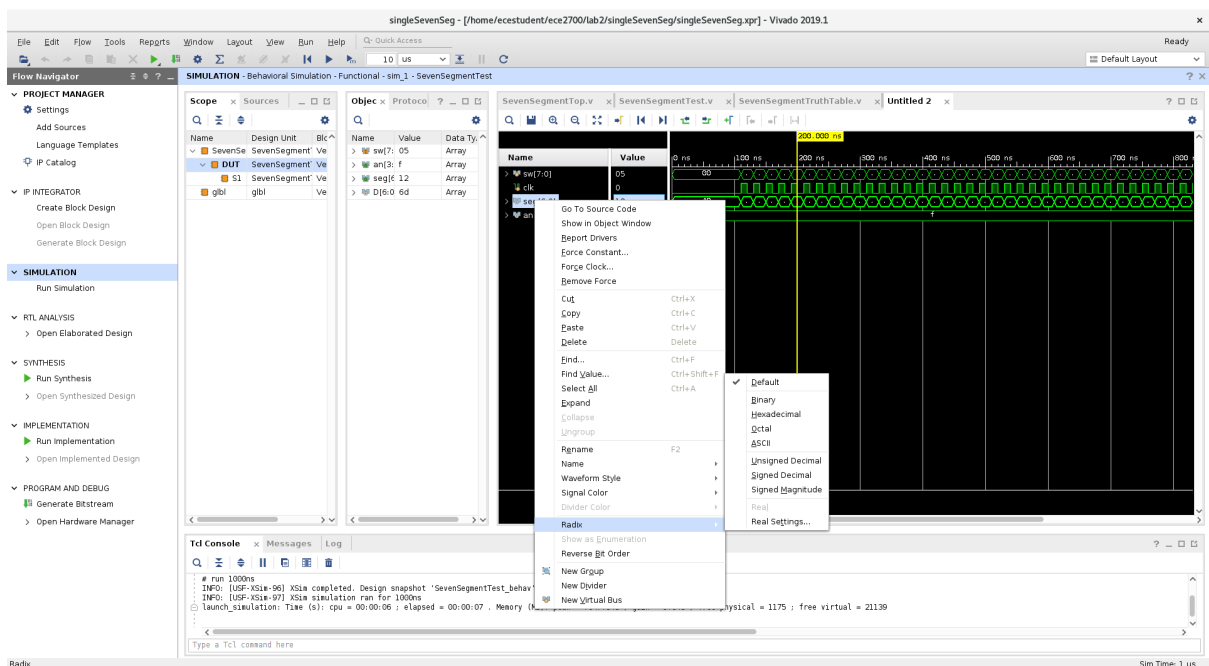
```

always @(posedge clk) begin
    if (sw >= 9)
        sw <= 0;
    else
        sw <= sw + 1;
end

endmodule

```

When your testbench module is ready, launch the simulation and zoom in so that you can see individual clock cycles. Click on the **seg** signal to expand its bits, and notice that they are both inverted and in reverse order (g down to a). Verify the first couple of truth table rows by inspecting the bits values. Since you have a lot of bits packed into the **seg** and **sw** signals, it will be helpful to represent them as integer numbers. To do this, **right-click on the signal's name in the waveform viewer, and select Radix→Unsigned Decimal**, like this:



Once you have changed the radix, you should see the signal values reported as numbers. The **sw** signal should increment as 0, 1, 2, ... 9 and then roll back to 0. The correct sequence for the **seg** signal should be 64, 121, 36, 48, 25, 18, 2, 120, 0, 16, and then back to 64. It is much easier to verify the integer sequence than to examine all the bits at all times, however if your design doesn't match the expected sequence, you will have to take a close look at the individual bits.

Name	Value	
sw[7:0]	05	66
clk	0	01
seg[6:0]	18	02
an[3:0]	f	03
		04
		05
		06
		07
		08
		09
		64
		121
		36
		48
		25
		18
		2
		120
		0
		16

Repeat this verification for both the TruthTable and Combinational versions of your design. Once the design is verified, proceed to program your Basys board and test the function on the physical device. You will need to define the pin mappings with an XDC constraints file. You can import the Master XDC file into your project and uncomment the lines corresponding to the switches and the seven-segment display. **Make sure that the names used in the “get_ports” statements match the actual port names used in your top-level module.** The end result should look something like this:

```

## Switches
set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
set_property PACKAGE_PIN V16 [get_ports {sw[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
set_property PACKAGE_PIN W16 [get_ports {sw[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[2]}]
set_property PACKAGE_PIN W17 [get_ports {sw[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[3]}]
set_property PACKAGE_PIN W15 [get_ports {sw[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[4]}]
set_property PACKAGE_PIN V15 [get_ports {sw[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[5]}]
set_property PACKAGE_PIN W14 [get_ports {sw[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[6]}]
set_property PACKAGE_PIN W13 [get_ports {sw[7]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sw[7]}]

# 7 segment display
set_property PACKAGE_PIN W7 [get_ports {seg[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {seg[0]}]
set_property PACKAGE_PIN W6 [get_ports {seg[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {seg[1]}]
set_property PACKAGE_PIN U8 [get_ports {seg[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {seg[2]}]
set_property PACKAGE_PIN V8 [get_ports {seg[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {seg[3]}]
set_property PACKAGE_PIN U5 [get_ports {seg[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {seg[4]}]
set_property PACKAGE_PIN V5 [get_ports {seg[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {seg[5]}]
set_property PACKAGE_PIN U7 [get_ports {seg[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {seg[6]}]

# Anode pins for 7-segment display:
set_property PACKAGE_PIN U2 [get_ports {an[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {an[0]}]
set_property PACKAGE_PIN U4 [get_ports {an[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {an[1]}]
set_property PACKAGE_PIN V4 [get_ports {an[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {an[2]}]
set_property PACKAGE_PIN W4 [get_ports {an[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {an[3]}]

```

6 Dealing with Invalid Inputs

In your pre-lab, you obtained minimized Boolean expressions only for input values that represent valid BCD digits (i.e. values 0 through 9). But “in the wild”, your design could conceivably encounter inputs for values 10 through 15. To see what happens, program your board with the Combinational version and give it some input values greater than 9. The results should look wrong, and case 11 looks like gibberish.

In the TruthTable version of the design, a **default** case is used to disable all of the segments when the

input isn't valid. Our combinational design doesn't have such a feature, but there are several ways we could screen for invalid input combinations. We could add more terms to the Boolean expressions within the Combinational module, or we could add that logic at a higher level of hierarchy in the Top module. The hierarchical method turns out to be pretty easy if we **implement the NAN logic that you solved in the pre-lab**. We can implement this function in just two lines:

```
wire    NAN = // INSERT YOUR SOLUTION HERE
assign seg = ~(D&~{7{NAN}});
```

Note that since we are working in the Top module, your logic solution for NAN will need to reference signals `sw[3]` instead of `w`, `sw[2]` instead of `x2`, and so on. In the second line, we make use of Verilog's **concatenation** and **repetition** syntax, which is useful for performing **bit-wise operations** across all the bits in a vector:

```
{7{NAN}}           // This means repeat NAN seven times
~{7{NAN}}          // This means negate seven copies of NAN
D&~{7{NAN}}        // This means {D[6]&~NAN,D[5]&~NAN,...,D[0]&~NAN}
~(D&~{7{NAN}})     // This means {~(D[6]&~NAN),~(D[5]&~NAN),...,~(D[0]&~NAN)}
```

Since D has seven bits, but NAN is only a one-bit signal, **we need to repeat NAN seven times** in order to do a bitwise operation. What do you think would happen if you tried to do this:

```
~(D&~NAN) // Bad code!
```

In this case, Verilog will automatically expand NAN into a seven-bit vector, but the default behavior is to **pad with zeros** from the left side. So if NAN equals 1, you will get

```
~(D&~NAN) // expands to ~(D&{0,0,0,0,0,0,1}) REALLY BAD
```

For this reason, you need to be explicit and tell Verilog that you intend to produce seven copies of NAN.

If you do this properly and implement the function on your board, **you should see the segment displays switch off when the input is greater than 9. Demonstrate this to your TA.**

7 Modify BCD to Build Hexadecimal Seven-Segment Decoder

Save a separate copy of all your BCD seven-segment decoder. You may modify your BCD decoder to implement a **hexadecimal** seven-segment decoder, showing digits A through F with `b` and `d` as lower case. You are required to use either the “compact case statement” or “assign statements” methods described in Section 4. **Demonstrate the working hexadecimal decoder FPGA implementation to your TA.**

8 TA Checkoff

- (18 points) Complete pre-lab work prior to start of the lab.
- (10 points) Correct simulation and FPGA implementation of the clock divider.
- (16 points) Correct simulation of the TruthTable module.
- (16 points) Correct simulation of the Combinational module.
- (20 points) Correct demonstration of **both modules** on the Basys board.
- (20 points) Correct solution and demonstration of the Combinational module with NAN detection.
- (20 points) Correct demonstration of hexadecimal decoder on the Basys board.

9 Extra Practice Problems

- Try connecting the clock divider to a 4-bit counter which is then connected to the seven segment display so it auto increments instead of using switches.
- Research activity: See if you can understand the **double-dabble algorithm** used to convert binary numbers to BCD representations. You will need to do some web research for this.
- If you're feeling really enterprising, try implementing the double-dabble algorithm in Verilog, and verify your implementation via simulation.