ECE 2700 Lab 3
Decoders and MUXes

Due at the end of your registered lab session (110 points)

Objectives

- Theory Objectives:
  - Structure and application of N-to-2^N decoders
  - Using decoders to encode truth tables
  - Using decoders to build larger decoders
  - MUX arrays for vector (bus) switching
  - Verification by comparing to a "golden" reference design
  - Basic synchronous system design
- Verilog Syntax Objectives
  - Using `generate` blocks for module arrays
  - Parameterized modules for general-purpose module design
  - Unary vector reduction operators `|`, `&`, $\sim$, $\wedge$
  - Using decimal, octal or hexadecimal entry for more compact binary vectors
- Resource Objectives:
  - Reusable modules in RTL design (you will reuse the `clockdivider` module from Lab 1)
  - Satisfying design constraints from the Basys3 board's refresh timing specification
  - Use persistence-of-vision to make a multi-digit seven segment display

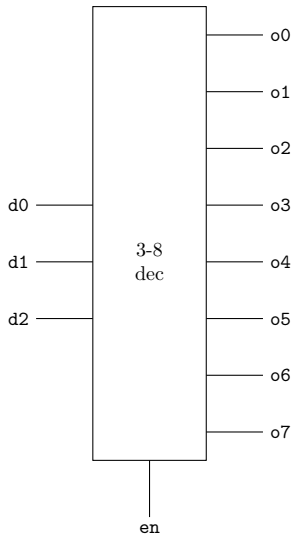
1 Introduction

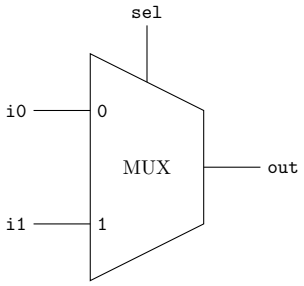
This lab makes use of two basic components: a decoder and multiplixer (MUX). A 3-to-8 decoder has address input `d` and *enable* input `en`. The decoder's output wires are labeled by an index from `0` up to `7`. The decoder produces all-zero output when `en==0`. When `en==1`, the decoder outputs zero on every wire *except* for the `d` $^{\text{th}}$ wire. In other words, *exactly one of the outputs is high* when `en` is high, and *all of the outputs are low* when `en` is low.

A MUX has one output `out`, two signal inputs `i0` and `i1`, and a *select* input `sel`. When `sel==0`, the output is assigned to equal `i0`. When `sel==1`, the output is assigned to equal `i1`. This behavior can be written algebraically as

$$\texttt{out} = \texttt{sel} \cdot \texttt{i1} + \overline{\texttt{sel}} \cdot \texttt{i0}.$$

The usual symbols for decoders and MUX modules are shown below.





2 Pre-Lab Exercises

1. Given the decoder behavior described above, show how you can make a 4-to-16 decoder by interconnecting two 3-to-8 decoders (with enable) and an inverter.

2. Use multiple 2-to-1 MUXes and inverters to construct a 4-to-1 MUX.

3. Suppose we want to switch between three-bit numbers `a` and `b` (i.e. `a` has digits `a0`, `a1`, and `a2`, and `b` is also a vector three bits). How would you interconnect 2-to-1 MUXes in order to select `a` if `sel==1`, or `b` if `sel==0`?

4. Suppose a `clk` signal operates with a frequency of 50 MHz, and we want to refresh two display units such that only one display can be active at a time, and each display needs to be active for exactly 1 ms. Calculate the period of the `clk` signal, and determine the numerator N for a clock divider so that the "ON" time is 1 ms.

5. Examine the truth table for the seven-segment display driver from Lab 2 (Example 2.3 on page 72 in the textbook). For each column `a`, `b`, ..., `f`, convert *vertically* from 16 binary digits to 4 hexadecimal digits. In each group of four digits, consider the *top bit* to be the MSB.

6. The function defined in the truth table below has three inputs (`a`, `b`, `c`) and two outputs (`x` and `y`). Use a 3-to-8 decoder with OR gates to implement the function. Show the complete schematic for your solution.

a	b	c	x	y
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

3 3-to-8 Decoder Module

Create a new folder named `Lab3`. Open Xilinx ISE and create a new project called `Decoder`. Then create a new Verilog source called `decoder3_8.v`, with the following I/O ports:

direction	name	vector	bits
input	d	yes	2:0
input	en	no	–
output reg	o	yes	7:0

Within your module, use a `case` statement within an `always` block to model the decoder behavior:

```
always @(*) begin
        if (en) begin
                case (d)
                        3'd7: o=8'd1;
                        3'd6: o=8'd2;
                        3'd5: o=8'd4;
                        3'd4: o=8'd8;
                        3'd3: o=8'd16;
                        3'd2: o=8'd32;
                        3'd1: o=8'd64;
                        3'd0: o=8'd128;
                endcase
        end
        else
                o=8'd0;
end
```

Notice here that we using **decimal numbers** to refer to binary vectors. By default, Verilog converts a decimal number to an **unsigned integer**, so `3'd0` is implicitly converted to `3'b000`, and `3'd1` gets converted to `3'b001`, and so on up to `3'd7` which gets converted to `3'b111`. On the output side, `8'd2` gets converted to `8'b00000010`, `8'd32` gets converted to `8'b00100000`, and so on. It's often more convenient or compact to write decimal, octal or hexadecimal numbers instead of full binary vectors.

Also notice that we enclose the `en` signal in an `if/else` block. **In a combinational logic module, it is important to define the output for all possible input combinations,** i.e. the `else` behavior needs to be explicitly defined. If anything is left out, Verilog will *infer* that you intend to have some kind of memory in your module. As a result, you would *accidentally* end up with a *sequential* logic module instead of a combinational logic module. This is bad because (1) we haven't learned about sequential logic yet; (2) the behavior could be unpredictable in some cases, and (3) the extra memory elements add unwanted complexity when your design is implemented on the FPGA.

Once your module is complete, you should create a Verilog Test Fixture to verify its behavior. Use the `New Source` wizard to create the test fixture template named `decoder_test.v`. We're going to setup a **clocked testbench**; in most future Xilinx projects you will use the same clocked testbench procedure, so pay attention. Within the test fixture, declare a `reg` signal named `clk`, and a four-bit `reg` signal named `count`. Edit the `initial` block to initialize signals and define the clock behavior:

```
reg        clk;
reg [3:0]  count;

initial begin
        // Initialize Inputs
        d = 0;
        en = 0;
        count = 0;
        clk = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here
```

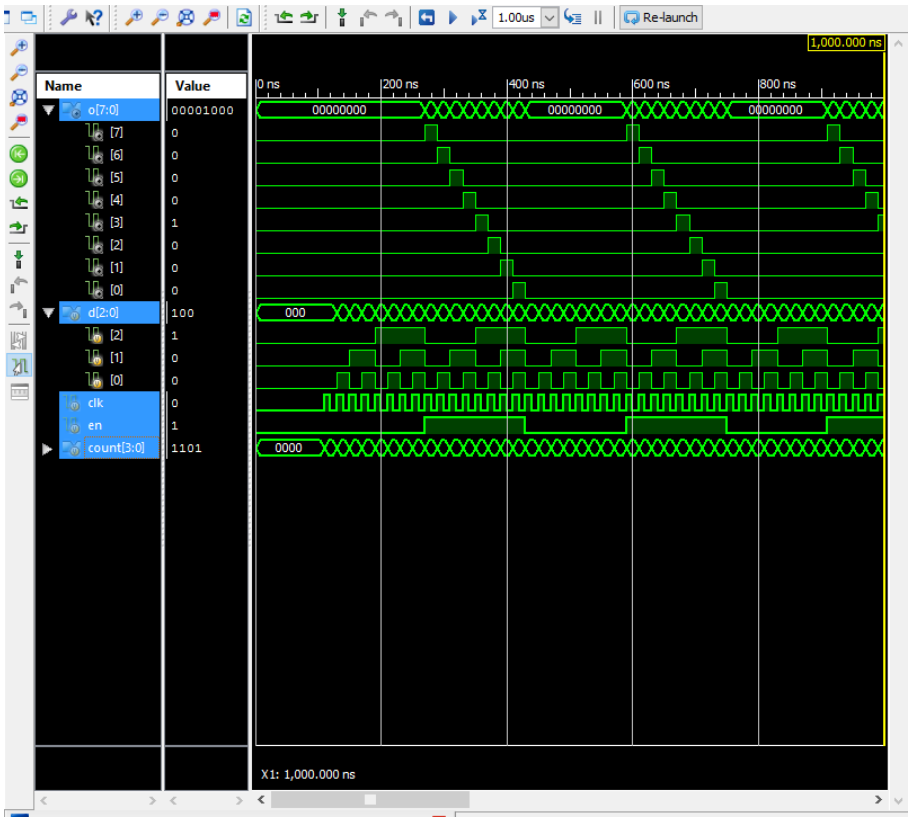
```
        forever #10 clk=~clk;
    end
```

Next, after the end of the `initial` block, create a clocked `always` block that is sensitive to the rising edge of `clk`, and increment the `count` signal by one each time the clock is triggered. Set `d` to equal the lower bits of `count`, and set `en` to equal the MSB of `count`:

```
    always @(posedge clk) begin
        count <= count + 1;
        d     <= count[2:0];
        en    <= count[3];
    end
```

In this testbench, we are treating `count` as though it were an integer. We increment `count` by one during each clock cycle, which will eventually step through all input combinations of `d` and `en`. When `count` reaches `4'b1111`, the next increment will roll it back to `4'b0000`. This testbench demonstrates a simple way of doing **exhaustive combinational testing**. This method is suitable for relatively simple designs, but becomes inefficient for larger designs. For example, if a module has 20 different input bits, we would have to test 2^{20} patterns (more than one million). You wouldn't want to look through a million patterns searching for a single error, so we'll use this procedure only for relatively small designs.

Your simulation results should look like the "stairstep" pattern shown below. Among the output wires, *at most* one should be active at any time. When `en` is low, none of the outputs should be active.



Once you have verified your decoder, import the `Basys3_Master.xdc` file and customize it for the port names in your design.Finally, implement the design, generate a bit file, program your board and physically test all the expected behavior. Demonstrate your final programmed design to the TA.

4 MUX Module

For the next part of this lab, **create a new project in Vivado** named `MUX`, and create a new module called `mux.v` with the following I/O signals:

direction	name	vector	bits
`input`	`i`	yes	1:0
`input`	`sel`	no	–
`output reg`	`o`	no	–

Define the correct MUX behavior using `if/then` or `case` statements in an `always` block. Create a clocked testbench similar to the one we made for the decoder module, and verify your design in simulation. If you are uncertain with your solution, ask the TA to confirm your simulation result. Then customize the XDC constraint file as before. Use `sw0` and `sw1` as the input `i`, `sw2` as `sel`, and `LED0` as the output. Program your board, verify the behavior, and demonstrate it to your TA.

5 Two-Digit Seven Segment Display

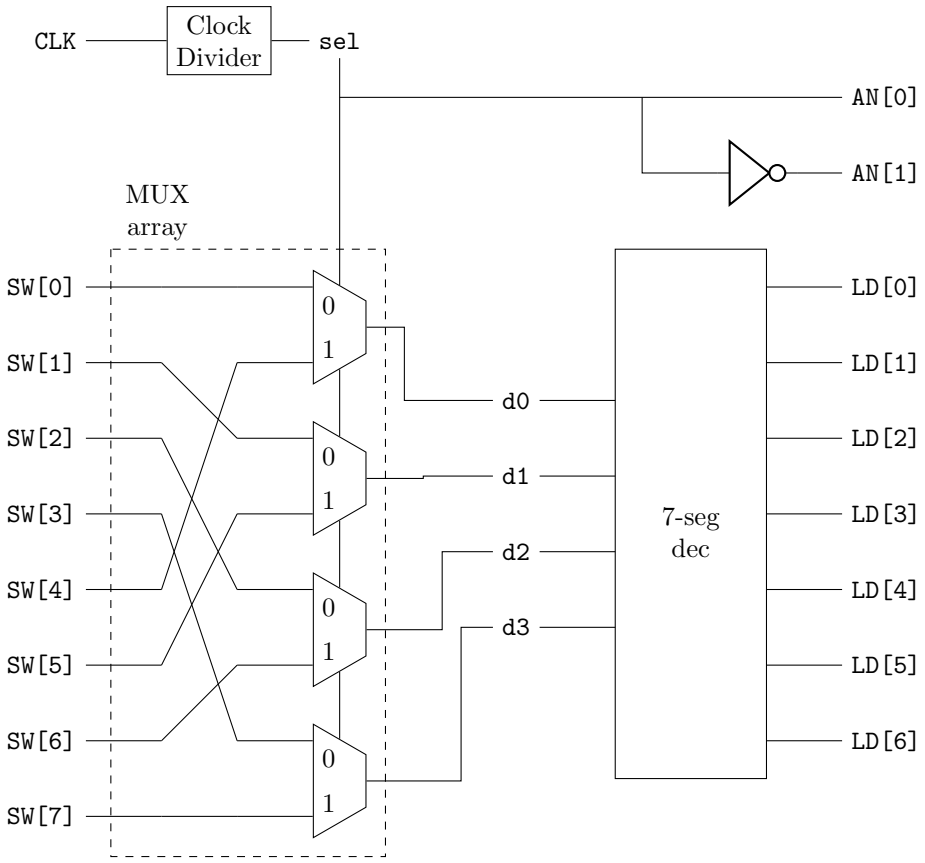
Now we're going to practice some hierarchical designs, and we're going to do some more work with the seven-segment display. Create a new project called `DoubleSevenSegment`. Then import sources from your previous projects by clicking `Add Source` (do not create new sources) and use the file browser to add in `decoder3_8.v`, `mux.v` and then go all the way back to Lab 1 to add in `clockdivider.v`. **Important: when adding sources make sure to add a copy of the source. Do not link the original source file since we will need to make some edits that are specific to this project.**

The Basys3 board allows us to define only one seven-segment display digit at a time, but there are four digits on the board. In your last lab, you were able to display the same value on all digits. How can we display separate values on all four digits at the same time? The answer has to do with *persistence of vision*, which is the basic idea behind almost all display hardware. A digital system is able to scan through an array of displays much faster than the human eye can perceive. If we rapidly display one digit, then the next, then the next, and so on, our eyes will perceive it as a simultaneous display. If the display is updated too slowly, then the eye will perceive the scanning process. If it is done too rapidly, then the display drivers will not have time to fully illuminate the digits and they will look dim or off.

So we will need to design a system with precise timing that carries out these steps:

1. Select data for digit 0, decode segments, and activate anode 0
2. Select data for digit 1, decode segments, and activate anode 1
3. Repeat these steps indefinitely

To achieve this, our top-level design will look like this:



To implement this design, we will create a 4-to-16 decoder and use it to make a new version of the seven-segment display decoder. We will use the clock divider to create the `sel` signal that selects between digit 0 and digit 1. We will create a **MUX array** module to switch between the inputs for digit 0 and digit 1, which will be taken from the top four switches and the bottom four switches, respectively.

5.1 4-to-16 Decoder Module

For your first new module in this project, create a new Verilog source named `decoder4_16.v`. This decoder will take a four-bit address input to assert one bit from a sixteen-bit output vector. Include an `en` input signal to enable your decoder. This module is going to be a **structural design**, meaning all signals are declared as `wire` type and we will not use any `always` block. Your design should implement the schematic you designed in Pre-Lab exercise 1. You will need to instantiate two `decoder3_8` modules, and you will need to define the `en` signals for those modules. The solution is partially completed for you in the code below.

```
    wire en1, en2;

    // Create an assign statement to define the logic for en1 and en2

    decoder3_8 d1(
                .en(en1),
                .d(),  // <--- fill in the d inputs
                .o(o[7:0])
        );
    decoder3_8 d2(
                .en(en2),
                .d(),  // <--- fill in the d inputs
                .o(o[15:8])
```

```
);
```

As before, simulate the design to verify it. **You don't need to program this one.**

6 Parameterized MUX Array Module

Your next module will be a flexible MUX array. Create a new source called `mux_array.v` with the following I/O signals:

direction	name	vector	bits
`input`	`a`	yes	3:0
`input`	`b`	yes	3:0
`input`	`sel`	no	–
`output reg`	`o`	yes	3:0

In this module, we are going to create an array of four MUX modules so that we can switch between two four-bit vectors. In order to do this efficiently, we will make use of Verilog's `generate` syntax which allows us to declare a regular array of modules. A bonus from using this technique is that we can make a **reusable general-purpose MUX array** by using Verilog's `parameter` syntax. This will make our MUX array adaptable to any size of input vectors, and will be useful for saving work in future designs.

The flexible syntax is demonstrated in the module shown below. Since we need to have parameterized input and output vector sizes, the parameter declaration must come first, using the syntax

```
module <module_name> #(parameter <param_name>=<default_value>) (
        <port definitions>
);
```

Then we can use the parameter in a `generate` statement as shown below:

```
module MUXarray #(parameter SIZE=4) (
        input sel,
        input [SIZE-1:0] a,
        input [SIZE-1:0] b,
        output [SIZE-1:0] o
);

        genvar i;

        generate
                for (i=0; i<SIZE; i=i+1) begin: MUXarray
                        mux m(
                                .i({a[i],b[i]}),
                                .sel(sel),
                                .o(o[i])
                        );
                end
        endgenerate

endmodule
```

The `generate` array contains three essential elements: (1) a `genvar` variable to be used in the `for` loop; (2) the actual `generate/endgenerate` block; and (3) a named `for` loop initiated with `begin: <name>` and terminated with `end`. In our case the loop is named "`MUXarray`", which happens to be the same as the module name (but we could name the loop anything, excluding Verilog keywords, if we were feeling creative.) Anything you instantiate inside the `for` loop will be repeated `SIZE` times. This is an incredibly useful shorthand for building structural designs.

When instantiating the parameterized MUX array, whether in a testbench or in a hierarchical design, you can use the default parameter setting (4 in this case) by declaring the instance as usual. To set an alternative value of the parameter, you need to specify it using the # syntax:

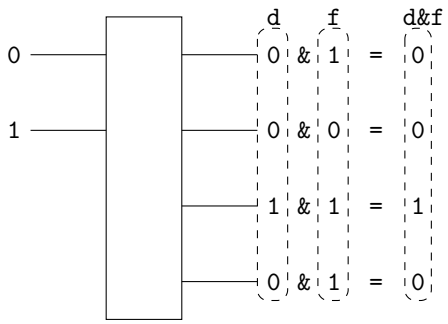
```
wire [3:0] a, b, o;
wire [7:0] big_a, big_b, big_o;

// Default parameter, size of 4:
MUXarray            mux1(.sel(sel),.a(a),.b(b),.o(o));

// Modified parameter, size of 8:
MUXarray #(.SIZE(8)) mux2(.sel(sel),.a(big_a),.b(big_b),.o(big_o));
```

In this design, it so happens we will only use the default size. In the future, however, you may need to reuse this `MUXarray` module for vectors with a different length.

6.1 New Seven-Segment Decoder

In Pre-Lab exercise 6, you were asked to implement a truth table by using a 3-to-8 decoder with OR gates. In exercise 5 you were asked to convert the seven-segment truth table columns into hexadecimal. Now, in this lab step, you will combine those approaches to create a seven-segment decoder based on the 4-to-16 decoder module. Create a new Verilog module in your project called `sevensegment.v`.

To efficiently model a truth table in Verilog, we can take advantage of the fact that the decoder outputs a 16-bit vector in which all bits are zero except for the selected signal. Therefore, if we apply a **bit-wise AND** operation over the decoder's output together with the truth table output, it will select precisely the desired row from the truth table:



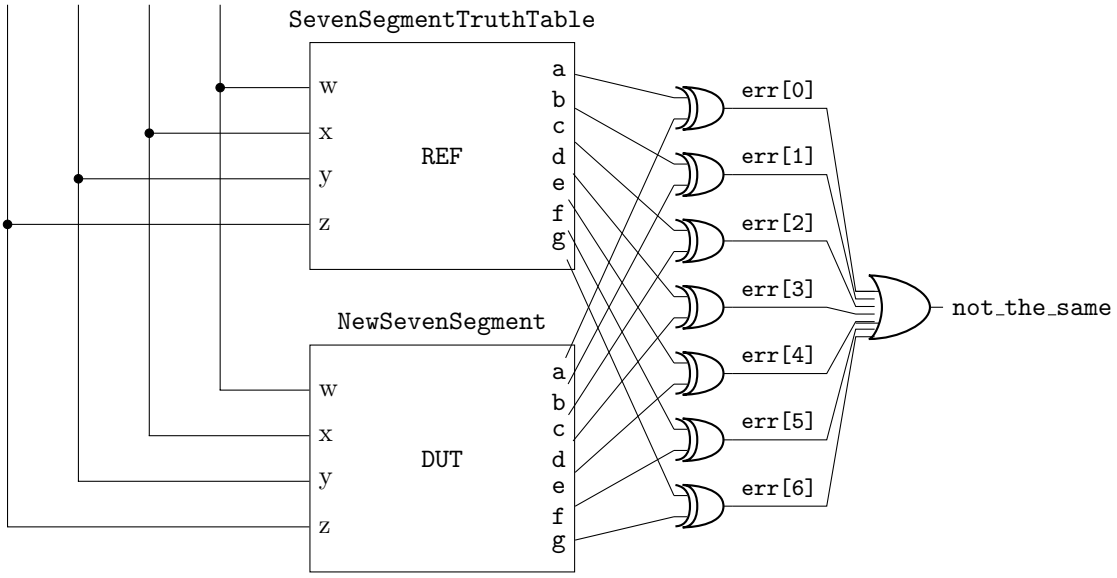
The module code is partially completed below; you will need to fill in the missing entries based on your Pre-Lab exercises.

```
module NewSevenSegment(
    input [3:0] wxyz,
    output [6:0] seg
    );

       wire [15:0] d;

       decoder4_16 D(.in(wxyz), .out(d));

       assign seg[0] = |(d & 16'hB7C0);  // a
       assign seg[1] = |(d & 16'hF9C0);  // b
       assign seg[2] = |(d & 16'hDFC0);  // c
       assign seg[3] = |(d & 16'hB6C0);  // d
       assign seg[4] = |(d & 16'hA280);  // e
       assign seg[5] = // ENTER YOUR SOLUTION for f
```

```
        assign seg[6] = // ENTER YOUR SOLUTION for g
endmodule
```

Verifying your new seven-segment module: Since you previously completed some alternative designs for the same seven-segment decoder function, you can check your new design against the old one. Specifically, you should copy the source for your `SevenSegmentTruthTable` module from Lab 2. Your new module should have exactly the same behavior as the old one. We can test their equivalence by creating instances of the new module and the old module, providing them both with the same input signals. If there is any error in your new module design, then at least one of its outputs will be different from `SevenSegmentTruthTable`. We can detect any difference by using XOR operations on the respective signals. So your test structure should look like this:

```
reg [3:0] wxyz;

always @(posedge clk) begin
        wxyz = wxyz+1;
end
```

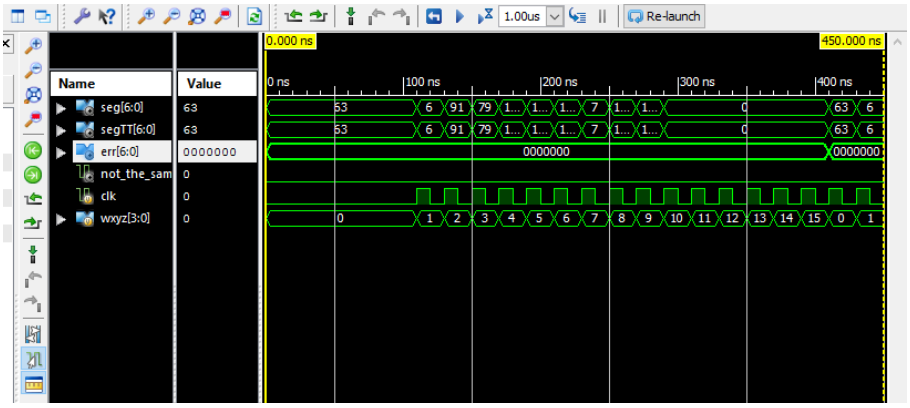


This verification is simpler than it appears. Follow these steps:

- First, modify your copy of `SevenSegmentTruthTable.v` so that its inputs and outputs are vectors, just like the I/O ports used in `NewSevenSegment.v`.
- Then create a `Verilog Text Fixture` using the `New Source` wizard and associate it with `NewSevenSegment.v`.
- Modify your text fixture to create a clock source, and create a four-bit `reg` signal named `wxyz`, initilized to zero. Then create three seven-bit `wire` signals named `segNew` and `segTruthTable` and `err`. Lastly create a single-bit `wire` signal named `not_the_same`.
- In the `DUT` instance statement, set the input to `wxyz` and the output to `segNew`.
- Copy and paste the instance declaration for your `DUT` module, which should be of type `NewSevenSegment`. In the pasted line, change the module type to `SevenSegmentTruthTable` and the instance name to `REF` (for "reference design"). Set `REF`'s input to `wxyz` and its output to `segTruthTable`.
- Now create an `assign` statement to set `err = segNew^segTruthTable`. This will compute the **bit-wise XOR operation** over all respective bits in the two vectors.
- Create another `assign` statement to set `not_the_same = |err`. This will compute the **reduction OR operation** over all the elements in `err`, reducing it to a single bit.

- Lastly, create an `always` block, sensitive to `posedge clk`, which increments `wxyz` through all of its binary values.

Now, when you simulate the design, you should see `wxyz` count through values from 0 up to 15, then roll over back to 0 and up to 15 again. Through all of this, you should see a lot of activity in `segNew` and `segTruthTable`, but the error signal `not_the_same` should stay flat at zero. If it stays at zero, this tells you the two modules are behaving exactly the same. The correct result should look like this:



Notice that `not_the_same` stays flat through the whole simulation. It detects no discrepancies between my two module designs.

NOTE: it's easy to deceive yourself by computing incorrect expressions. For example, if you XOR `segNew` with itself, the result will be forever zero. If you inadvertently assign `not_the_same` to zero, then obviously it will stay at zero. Make sure your assign statements are computing the right calculation before you declare victory.

Second NOTE: If you see a lot of errors, it *could be* due to one of your modules having reverse-order port signals. Study your module outputs and check if they are appearing in reverse order. If, for example, `segNew[0]` is identical to `segTruthTable[6]`, then one of your modules is wired backwards, and you need to modify it to reverse the order of signals in the output vector.

6.2 Multi-Digit Seven-Segment Display

Now create a file called `DoubleSevenSegment.v`, and create submodule instances to implement a structural description of the top-level design. You will use the `MUXarray`, the `decoder4_16` and the `clockdivider` modules.

Next, edit the `clockdivider.v` schematic to alter the divider ratio. Use the value that you calculated in the Pre-Lab to achieve a 1 ms positive phase for the clock (i.e. the total clock period should be 2 ms). Then customize an XDC constraint file to assign the board I/O signals as shown in the top-level design.

To verify your design, create a Verilog Test Fixture and create the appropriate test signals. Try to rigorously study and interpret the simulation traces to make sure your design does what is intended. Then implement the design and program your board. You should see two illuminated digits on the seven segment display. You can change one digit by altering the top four switches, and the other digit is set by the bottom four switches. For each digit, enter all binary values from 0 to 15. The displays should turn off (go blank) when the value is greater than 9.

7 TA Checkoff

- (24 points) Complete pre-lab work prior to start of the lab.
- (12 points) Correct design, simulation and demonstration of the 3-to-8 decoder module.
- (12 points) Correct design, simulation and demonstration of the 2-to-1 MUX module.
- (12 points) Correct design and simulation of the 4-to-16 decoder module.

- (12 points) Correct design and simulation of the parameterized array MUX module.
- (18 points) Correct design and simulation of the seven-segment decoder module.
- (20 points) Correct design, simulation and demonstration of the Top-Level Design for the Multi-Digit Seven-Segment Display.