Utilizing the Seven Segment Display on a Nexys2 Board with Case

Vincent Martin
TUID: 913012274
ECE 2613
Lab #: 4 (9/21/2012)

Introduction:
The objective of this lab is to utilize our understanding of the case function and procedural methods to create a Verilog module that will use input based on switches to drive the seven segment display built into our Xilinx test board. Then we will test it, and finally implement it on the board itself.

The primary difference between this lab and our previous lab is that we will be using a procedural method along with a case statement to define the logic for our svn segment display. This will result in the use of less code and also will prove to be easier to read.

Finally, verification will be done via reporting methods to determine that both ways of creating our module, the way in lab 3 and this current one in lab 4, will result in the identical number of 4 input LUTs.

The Theory of our Seven Segment Display:
To drive the display we will need to create a module that has two inputs. One being a 4 bit bus that will represent the 4 switches we are using to to create binary numbers. The second input will be the 1 bit status of a 5th switch. These inputs will result in an output of 7 bits into the seg_out bus.

When the 5th switch is on and the remaining 4 switches are set the end user will be able to control what gets sent to the seven segment display components by the seg_out bus. Otherwise we will see no display if the 5th switch is off.

Using the case statement
The previous lab required the usage of a large number of assign statements to define the logic used to drive our seven segment display. This lead to code that could at times be difficult to understand.

The previous assign statements were all run in parallel. This is where procedural statements are different. While the procedural block will run in parallel along with other assign statements, the code within the procedural block will not. It will run more as a programming language, such as C. Therefor, order is important in the always block.

By using this we will be able to determine what differences other than aesthetic may result from its usage.

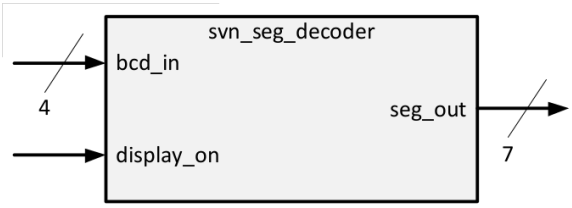
Applying the Theory to Hardware:

In order to transfer our understanding of theory to our Nexys2 hardware board we will have to write a Verilog code module that represents the block diagram seen in figure 1.

Module Description:

- Input
  - bcd_in: 4 bit mapped to switches 0 to 3 on the board.
  - display_on: 1 bit mapped to switch 4 on the board.
- Output
  - seg_out: 7 bit mapped to each particular segment of the seven segment display as seen in figure 2.

Figure 1 : Block diagram for the svn_seg_decoder



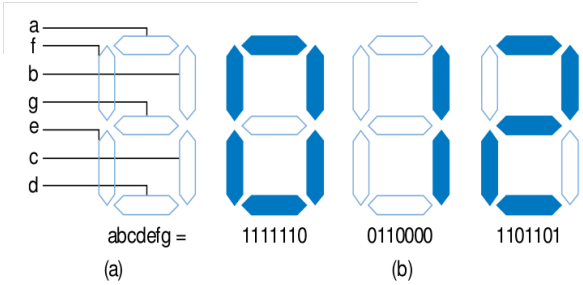


Figure 2 : Seven Segment Display Diagram

Additionally we will want to implement a testing module based on Figure 3. This scheme will utilize a .txt file, based on our truth table, which will allow us to test all of our expected outcomes. This text file will be included in the lab report.

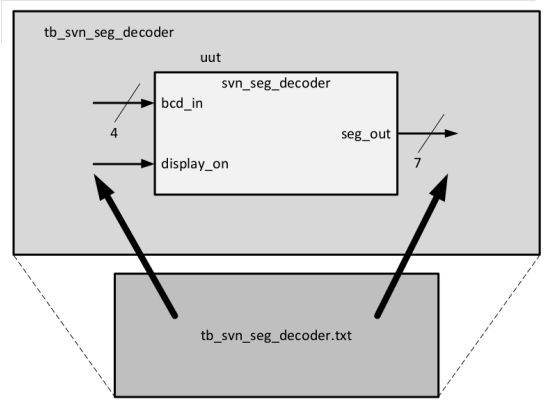


Figure 3: Testing methods

Procedure:

Create the Truth Table and Formula

1. Create a truth table for our module (figure 4).
2. Calculate the equations that will result in seg_out lighting the appropriate segments in regards to the input of bcd_in and display_on.
3. Use the results of the truth table to create a Equations found below in Figure 5.

Implement the Design in Software

1. Make a secure connection to electro9.eng.temple.edu using the no machine client.
2. Once terminal opens on the local workstation type the 'remote_xilinx.sh' shell command to launch the ISE development environment.
3. Open the lab3 seven segment project in ~/Xilinx/lab3/ directory.
4. Copy the lab3 seven segment project to ~/Xilinx/lab4/ directory.
5. Modify the svn_seg_out.v source code to implement your algorithm by navigating to
    - View: Implementation
        - xc3s500e-4f6320
    - Delete and or comment out the old assign code.
    - Modify output seg_out to be a reg type.
    - Create an always block
    - Put a case statement handling logic within this block
6. Save all files.

Prepare for Testing the Design

1. Verify the tb_svn_seg_out.txt testing to suit the needs of our truth table by navigating to
    - View: Implementation
2. Verify the tb_svn_seg_out.txt file to contain all possible input bit combinations.
3. Verify the tb_svn_seg_out.txt to contain all expected output bit combinations.
4. Save all files.

Test the Design with iSim

1. Switch to Simulation mode by clicking on
    a. View:Simulation
        i. Xc3s500e-4fg320.
        ii. Tb_svn_seg_out
2. Run iSim simulator by clicking on
    a. iSim Simulator
    b. Right click Simulate Behavioral Model and then run.
3. Once iSim runs, verify that the Mismatch—index messages match what you are expecting in your test bench text file.
4. If the results are not what you expect either edit your module code or your test bench code and then attempt to test again.
5. If the results are what you expected move on to the Compile to .bit file step.

Compile to .bit file

1. Compile to .bit file by navigating to
    a. Implementation
        i. Xc3s500e-4g320
            1. Lab4_top_io_wrapper
                a. Implement design
                b. Generation programming file

Transfer .bit file to Board

1. Use your favorite network transfer program to move the .bit file from the development server to your local workstation.
2. Plug the board into USB port.
3. Launch the Digilent Adept application on your local workstation.
4. Click the config tab.
5. Click on browse by the PROM icon.
6. Select your transferred .bit file.
7. Click program.
8. Once complete press the reset button on the board.

9. Test your outcome physically on the board to make sure that it matches expectations.

Run reports to determine the number of 4 bit LUTs

1. Navigate to simulation mode.
2. Click on the top_io_module for each lab.
3. Right click design summary/reports.
4. Click on Run.
5. Record # of 4 input LUTs used by lab 3 and lab 4.

Results:

Below you will find the truth table and equation representing our seven segment display. The results on the physical board matched what was expected from the truth table.

Figure 4: Truth Table

Display	BCD IN				g	f	e	d	c	b	a
Display_ON	bcd_in[3]	bcd_in[2]	bcd_in[1]	bcd_in[0]	seg_out[6]	seg_out[5]	seg_out[4]	seg_out[3]	seg_out[2]	seg_out[1]	seg_out[0]
1	0	0	0	0	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	1	1
1	0	0	1	1	1	1	1	1	0	0	1
1	0	1	0	0	1	1	1	0	1	0	0
1	0	1	0	1	1	1	0	1	1	0	1
1	0	1	1	0	1	1	0	1	1	1	1
1	0	1	1	1	0	1	1	1	0	0	0
1	1	0	0	0	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	0	1
1	1	0	1	0	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0
0	1	0	1	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0	0	0	0

Create Equation that Satisfies the Truth Table:

Each bit of the seg_out bus will need to be set via the logic determined from the trth table. In our previous lab we had to do this via a rather large list of assign statements. However, this time we will have the luxury of using procedural logic and the case statement as seen in Figure 5.

Number of 4 bit LUTs used:

Lab3: 12 4 bit LUTs
Lab 4: 12 4 bit LUTs

Each method of creating our seven segment display driver used the same number of 4 bit LUTs.

Figure 5: Verilog Equations

```
always @(display_on, bcd_in[3] or bcd_in[2] or bcd_in[1] or bcd_in[0]) begin

case({display_on,bcd_in[3],bcd_in[2],bcd_in[1],bcd_in[0]})

    // Takes into account all combinations with the display turned off

        //Number 0000 Decimal Val: 0 Display: on *
        5'b10000: seg_out = 7'b0111111;

        //Number 0001 Decimal Val: 1 Display: on *
        5'b10001: seg_out = 7'b0110000;

        //Number 0010 Decimal Val: 2 Display: on *
        5'b10010: seg_out = 7'b1011011;

        //Number 0011 Decimal Val: 3 Display: on
        5'b10011: seg_out = 7'b1111001;

        //Number 0100 Decimal Val: 4 Display: on
        5'b10100: seg_out = 7'b1110100;

        //Number 0101 Decimal Val: 5 Display: on
        5'b10101: seg_out = 7'b1101101;

        //Number 0110 Decimal Val: 6 Display: on
        5'b10110: seg_out = 7'b1101111;

        //Number 0111 Decimal Val: 7 Display: on
        5'b10111: seg_out = 7'b0111000;

        //Number 1000 Decimal Val: 8 Display: on
```

```
        5'b11000: seg_out = 7'b1111111;

        //Number 1001 Decimal Val: 9 Display: on
        5'b11001: seg_out = 7'b1111101;

        default: seg_out = 7'b0000000;

    endcase

end // end of always block
```

Discussion:

This was a very good introduction to the benefits of using the always statement to write procedural code to define our logic. It seems that using this method will allow for the writing of code that will often times be more simple to write and even read.

I am much more familiar to this sort of coding rather than the parallel Verilog model. While I understand both, I could see that more often than not if something is somewhat complex, that I would be more apt to use the procedural method of describing these modules. Perhaps things are done more this way in the professional world as well. However, I can not say for certain yet as I am new to the whole digital circuits realm.

Finally, it is interesting to see that while the code inside of the always block will run procedurally, anything out of that block will run in parallel still, along with the results of the logic from the always block. Either method will result in the same outcome.

It is because of this that we can see that each way to write our design will always result in the same number of 4 input LUTs, the building blocks of our Verilog logic board. This was verified when I ran a report for each and got back the same number of 12 for each method of describing our seven segment display module.

Source Code:
Please see attached documents.

SVN_SEG_DECODER.V module code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:16:21 09/11/2012
// Design Name:        seven segment decoder module
// Module Name:    svn_seg_decoder
// Project Name:   lab 04 seven segment decoder with case statement
// Target Devices:  xilinx board
// Tool versions:
// Description: Take in 4 bits as a descriptor of the number you want to show and
also
// 1bit as an on/off switch and then output the appropriate signal to drive
// the seven segment display.
//
// Dependencies:
//
// Revision: 2 (now using case statement)
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module svn_seg_decoder(
   input [3:0] bcd_in,
   input display_on,
   output reg [6:0] seg_out
   );



// My always logic to replace old code
always @(display_on, bcd_in[3] or bcd_in[2] or bcd_in[1] or bcd_in[0]) begin

         case({display_on,bcd_in[3],bcd_in[2],bcd_in[1],bcd_in[0]})
         // Takes into account all combinations with the display turned off

                  //Number 0000 Decimal Val: 0 Display: on *
                  5'b10000: seg_out = 7'b0111111;
```

```
                //Number 0001 Decimal Val: 1 Display: on *
                5'b10001: seg_out = 7'b0110000;

                //Number 0010 Decimal Val: 2 Display: on *
                5'b10010: seg_out = 7'b1011011;

                //Number 0011 Decimal Val: 3 Display: on
                5'b10011: seg_out = 7'b1111001;

                //Number 0100 Decimal Val: 4 Display: on
                5'b10100: seg_out = 7'b1110100;

                //Number 0101 Decimal Val: 5 Display: on
                5'b10101: seg_out = 7'b1101101;

                //Number 0110 Decimal Val: 6 Display: on
                5'b10110: seg_out = 7'b1101111;

                //Number 0111 Decimal Val: 7 Display: on
                5'b10111: seg_out = 7'b0111000;

                //Number 1000 Decimal Val: 8 Display: on
                5'b11000: seg_out = 7'b1111111;

                //Number 1001 Decimal Val: 9 Display: on
                5'b11001: seg_out = 7'b1111101;

                default: seg_out = 7'b0000000;

        endcase
end

endmodule
```

tb_svn_seg_module.txt

```
//
// lab4 : version 09/06/2012
//
// This file contains the test vectors for the
// 7 segment decoder
// The first column is the input display_on signal
// The next four columns are the inputs: bcd_in[3:0]
// The next 7 columns are the signals to the display:
// seg_out[6:0], representing the g,f,e,d,c,b,a segments.
//
// This needs to be 32 lines long to cover all possibilities
//
/* This is what they gave me, but I am going to just comment it all
out and then start over with my own generated file.

1_0000_0111111
1_0001_0110000

I basically copied this data from the excel sheet truth table and then used
tr -d '\t' <test.txt >> tb_svn_seg_decoder.txt
to strip it of the tab delimiters.
*/


1_0000_0111111
1_0001_0110000
1_0010_1011011
1_0011_1111001
1_0100_1110100
1_0101_1101101
1_0110_1101111
1_0111_0111000
1_1000_1111111
1_1001_1111101
1_1010_0000000
1_1011_0000000
1_1100_0000000
1_1101_0000000
1_1110_0000000
1_1111_0000000
0_0000_0000000
0_0001_0000000
```

0_0010_0000000
0_0011_0000000
0_0100_0000000
0_0101_0000000
0_0110_0000000
0_0111_0000000
0_1000_0000000
0_1001_0000000
0_1010_0000000
0_1011_0000000
0_1100_0000000
0_1101_0000000
0_1110_0000000
0_1111_0000000