**Combinational Block for 4-bit BCD Counter (Incrementer)**

Vincent Martin

TUID: 913012274

ECE 2613

Lab #: 4 (9/27/2012)

**Introduction:**

The objective of this lab is to create a 4-bit binary coded decimal counter/incrementer that will take a 4 bit binary number as its input and then output the input value incremented by 1 along with a carry bit if needed. This module will be created in two different ways using Verilog XISE. The first method will utilize case statements along with if statements in an always block.

The second manner will be quite similar, however, it will instead use addition in the assignment of the output rather than defining our result via the case statement. We will be able to see how this later method is more simple and easier to implement.

Another important objective is to learn how to avoid latch situations in our logic and finally, test our design via a test truth table and the implement it on a hardware boards.

The Theory of our 4-bit binary coded decimal counter/incrementer:

The main goal of our binary coded decimal counter/incrementer is to take in a 4 bit binary number from 4’b0000 to 4’b1001 and then increment it by 4’b0001, except when the input value is 4’b1001, in this case the output will be set to 4’b0000 along with a carry bit set to 1.

In addition to having a 4-bit input for our number to be incremented, the module will include a 1 bit clear and 1 bit enable input. If clear is set to 1 and enable is set to 0 our 4-bit output will be equal to 4’b0000 without a carry bit. In addition, if clear is set to 1 and enable is set to 1 then our 4-bit output will be equal to 4’b1001.

This explanation will be much more clear once the truth table is seen below.

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
  + q: 4 bit mapped to switches 0 to 3 on the board.
  + ena: 1 bit mapped to switch 4 on the board.
  + clr: 1 bit mapped to switch 5 on the board.
* Output
  + Next\_q: 4 bit mapped to the LEDs on the board.
  + c\_o: 1 bit carry bit mapped to a LED on the board.

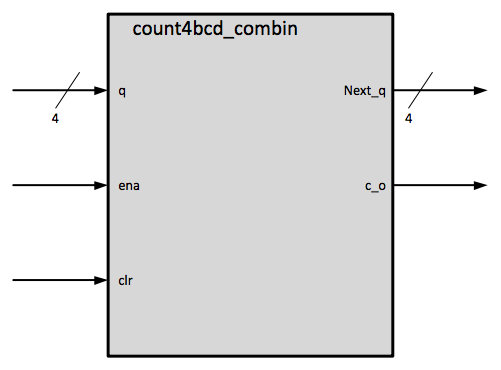
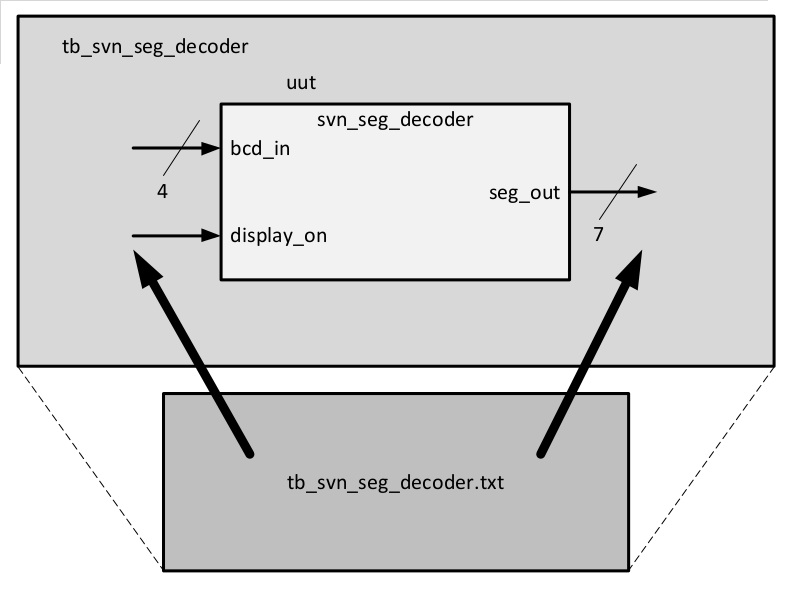


Figure : count4bcd\_combin module

Additionally we will want to implement a testing module based on Figure 2. 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

1. Verify the tb\_svn\_seg\_out.txt file to contain all possible input bit combinations.
2. Verify the tb\_svn\_seg\_out.txt to contain all expected output bit combinations.
3. Save all files.

Test the Design with iSim

1. Switch to Simulation mode by clicking on
   1. View:Simulation
      1. Xc3s500e-4fg320.
      2. Tb\_svn\_seg\_out
2. Run iSim simulator by clicking on
   1. iSim Simulator
   2. 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
   1. Implementation
      1. Xc3s500e-4g320
         1. Lab4\_top\_io\_wrapper
            1. Implement design
            2. 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.

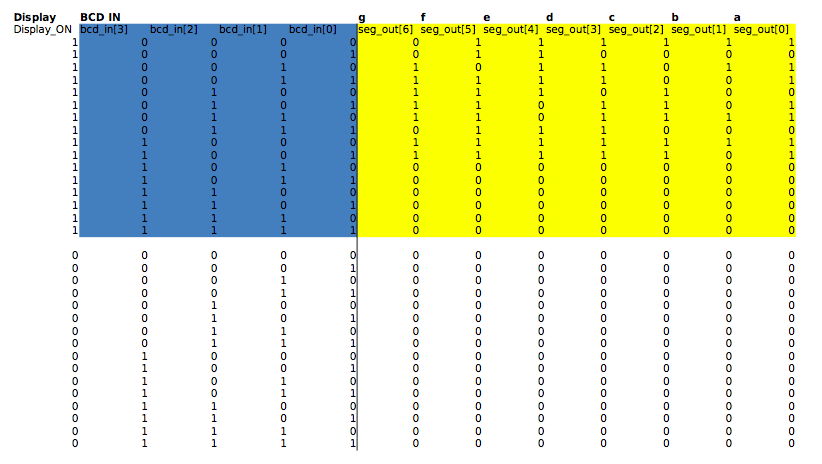
Run reports to determine the number of 4 bit LUTs

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



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