Combinational Block for 4-bit BCD Counter (Incrementer)

Vincent Martin
TUID: 913012274
ECE 2613
Lab #: 5 (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.

<u>The Theory of our 4-bit binary coded decimal counter/incrementer</u>
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.

<u>Applying the Theory to Hardware:</u>
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.

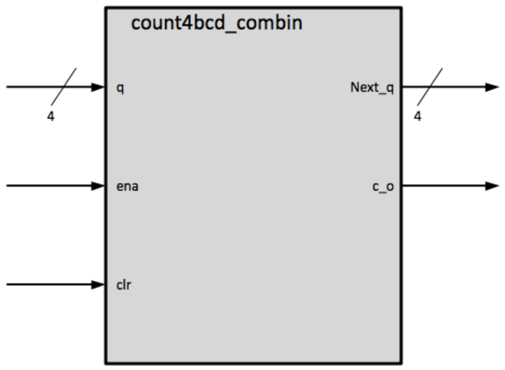

Figure 1: count4bcd_combin module

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.

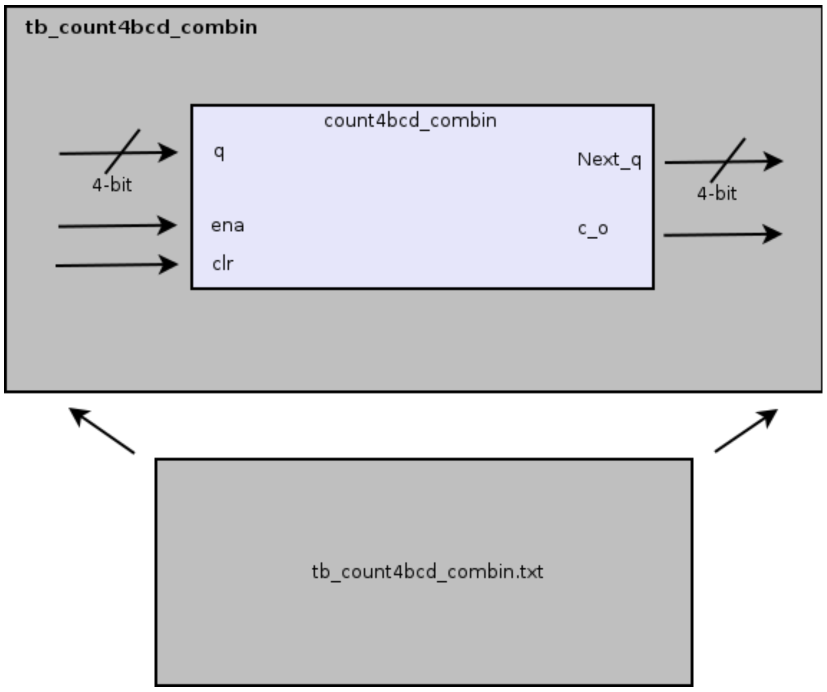

Figure 2: tb_count4bcd_combin test module

Procedure for Part A:

Create the Truth Table and Formula

1. Create a truth table for our module (figure 3).
2. Calculate the equations that will result in the appropriate incremented output.
3. Use the results of the truth table to create the equations found below in Figure 4.

Implement the Design in Software for part A.

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 lab5a project in ~/Xilinx/lab5a/ directory.
4. Create the count4bcd_combin.v module with the following input and output settings
    a. input [3:0] q,
    b. input ena,
    c. input clr,
    d. output reg [3:0] Next_q
    e. output reg c_o
5. Modify the source to include an always block with the appropriate logic to match the results dictated by the truth table.
6. Save all files.

Prepare for Testing the Design

1. Verify the tb_count4bcd_combin.txt testing to suit the needs of our truth table by navigating to
    - View: Implementation
2. Verify the tb_count4bcd_combin.txt file to contain all possible input bit combinations.
3. Verify the tb_count4bcd_combin.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_count4bcd_combin
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. Lab5_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.
3. Right click design summary/reports.
4. Click on Run.
5. Record # of 4 input LUTs used by lab 5a.

Procedure for Part B:

Implement the Design in Software for part B.

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 lab5a project in ~/Xilinx/lab5a/ directory.
4. Copy the lab5b project to ~/Xillinx/lab5b/ directory.

5. Modify the count4bcd_combin.v module with the following input and output
6. Modify the count4bcd_combin.v to have the following code in the portion which increments the q input bits.
    a.
    ```
    if ((ena == 1) && (clr == 0)) begin
         Next_q = q + 1;
         if (q == 9) c_o = 1;
         if (q >= 9) Next_q = 0;
    end  // end of if ena
    ```
7. Save all files.

Prepare for Testing the Design

1. Verify the tb_count4bcd_combin.txt testing to suit the needs of our truth table by navigating to
    - View: Implementation
2. Verify the count4bcd_combin.txt file to contain all possible input bit combinations.
3. Verify the count4bcd_combin.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_count4bcd_combin
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. Lab5_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.
3. Right click design summary/reports.
4. Click on Run.
5. Record # of 4 input LUTs used by lab 5b.

Results:
Below you will find the truth table and equation representing our 4-bit BCD counter/incrementer. The results on the physical board matched what was expected from the truth table.

Figure 3: Truth Table

Inputs						Outputs				
clr	ena	Q[3]	Q[2]	Q[1]	Q[0]	c_o	N_q[3]	N_q[2]	N_q[1]	N_q[0]
0	1	0	0	0	0	0	0	0	0	1
0	1	0	0	0	1	0	0	0	1	0
0	1	0	0	1	0	0	0	0	1	1
0	1	0	0	1	1	0	0	1	0	0
0	1	0	1	0	0	0	0	1	0	1
0	1	0	1	0	1	0	0	1	1	0
0	1	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	1	0	0	0
0	1	1	0	0	0	0	1	0	0	1
0	1	1	0	0	1	1	0	0	0	0
0	1	1	0	1	0	0	0	0	0	0
0	1	1	0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0

0	1	1	1	0	1	0	0	0	0	0
0	1	1	1	1	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	1	0	0	0	0	1
0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	1	1	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	0	1	0	1	0	0	1	0	1
0	0	0	1	1	0	0	0	1	1	0
0	0	0	1	1	1	0	0	1	1	1
0	0	1	0	0	0	0	1	0	0	0
0	0	1	0	0	1	0	1	0	0	1
0	0	1	0	1	0	0	1	0	1	0
0	0	1	0	1	1	0	1	0	1	1
0	0	1	1	0	0	0	1	1	0	0
0	0	1	1	0	1	0	1	1	0	1
0	0	1	1	1	0	0	1	1	1	0
0	0	1	1	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0
1	0	0	0	1	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0	0
1	0	0	1	1	1	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0
1	0	1	0	1	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	0	1	1	0	1	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0
1	0	1	1	1	1	0	0	0	0	0
1	1	0	0	0	0	1	1	0	0	1
1	1	0	0	0	1	1	1	0	0	1
1	1	0	0	1	0	1	1	0	0	1
1	1	0	0	1	1	1	1	0	0	1
1	1	0	1	0	0	1	1	0	0	1
1	1	0	1	0	1	1	1	0	0	1
1	1	0	1	1	0	1	1	0	0	1
1	1	0	1	1	1	1	1	0	0	1
1	1	1	0	0	0	1	1	0	0	1
1	1	1	0	0	1	1	1	0	0	1
1	1	1	0	1	0	1	1	0	0	1
1	1	1	0	1	1	1	1	0	0	1
1	1	1	1	0	0	1	1	0	0	1
1	1	1	1	0	1	1	1	0	0	1
1	1	1	1	1	0	1	1	0	0	1
1	1	1	1	1	1	1	1	0	0	1

<u>Number of 4-bit LUTs used:</u>

Lab 5a: 10 4-bit LUTs
Lab 5b: 10 4-bit LUTs

Each method of creating 4-bit BCD counter used the same number of 4-bit LUTs.

Discussion:

This lab taught me that we do not always have to use the case statement to create our logic blocks. We can instead do a more simple case such as just adding as we did in part b of this lab. This was much less work and in my opinion is easier to read and understand to someone else who may be looking at your design.

It is also interesting that doing it this shorter way resulted in the same number of 4-bit LUTs being used in both designs. This shows me that the software package is smart enough to know that both ways of the design result in the same input/output combinations and that while a designer may have their own preference as of how to syntactically write their Verilog code, that often times there is going to be no difference once the design is put onto the hardware.

Finally, this was a good exercise to let me know that I should be very careful when it comes to writing my logic. If I accidentally miss some of my logic while coding the module it can result in latch scenarios, which will not allow the module to be built and put onto the hardware board.

I learned more from trouble shooting this latch error than any other lab so far.

Source Code:
Please see attached documents for part 5a and 5b of this lab.

Count4bcd_combin.v module code PART A

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:31:53 09/26/2012
// Design Name:
// Module Name:    count4bcd_combin
// Project Name:  Part A
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module count4bcd_combin(
   input [3:0] q,
   input ena,
   input clr,
   output reg [3:0] Next_q, // I had to make this a reg so that I could use my always block.
   output reg c_o                              // I also had to make this a reg so that I can use it in my always block.
   );

         //Let us do some logic on our input switches


         always @(q or ena or clr)
                  begin

                           Next_q = 4'b0000;
                           c_o = 4'b0;


                           // ** HOLD **
                           //Logic when ena and clr are set to 0 to hold current values
                           if( (ena == 1'b0) && (clr == 1'b0))
                                    begin
                                             c_o = 1'b0;
                                             Next_q[0] = q[0];
                                             Next_q[1] = q[1];
                                             Next_q[2] = q[2];
                                             Next_q[3] = q[3];
                                    end



                           // ** MAX **
                           //Logic to handle when ena =1 and clr = 1 to get ouput
                           //of 9

                           if ( (ena ==1'b1) && (clr == 1'b1) )
```

```
                    begin
                            c_o = 1'b1;
                            Next_q = 4'b1001;
                    end



            // ** NEXT COUNT **
            //Logic when ena = 1 to calculate our next bit

            if ((ena == 1'b1) && (clr == 1'b0))
                    begin

                    // Set the carry bit to 0 by default unless changed by 4'b1001 case.
                    //c_o = 1'b0;

                            case ({q})

                                    //Begin 0 through 9
                                    4'b0000: Next_q = 4'b0001; // 0 input
                                    4'b0001: Next_q = 4'b0010; // 1 input
                                    4'b0010: Next_q = 4'b0011; // 2 input
                                    4'b0011: Next_q = 4'b0100; // 3 input
                                    4'b0100: Next_q = 4'b0101; // 4 input
                                    4'b0101: Next_q = 4'b0110; // 5 input
                                    4'b0110: Next_q = 4'b0111; // 6 input
                                    4'b0111: Next_q = 4'b1000; // 7 input
                                    4'b1000: Next_q = 4'b1001; // 8 input

                                    4'b1001:
    // 9 input, make output 0000 and set carry bit.
                                            begin
                                                    Next_q = 4'b0000;
                                                    c_o = 1'b1;
                                            end

                                    default: Next_q =   4'b0000; // All other combinations of
ena 1 and Next_q just sets Next_q to 0000

                            endcase
                    end // end of if handling the next bit



            // ** CLEAR **
            //Logic when clr = 1 and ena = 0 to clear
            if ((clr == 1'b1) && (ena == 1'b0))
                    begin
                            Next_q = 4'b0000; // set to 0000.
                            c_o = 1'b0;                 // set the carry bit to 0.
                    end // end if to set Next_1 and c_o to 0.


        end  // End my entire always block



endmodule
```

Count4bcd_combin.v module code PART B

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:31:53 09/26/2012
// Design Name:
// Module Name:    count4bcd_combin
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module count4bcd_combin(
   input [3:0] q,
   input ena,
   input clr,
   output reg [3:0] Next_q, // I had to make this a reg so that I could use my always block.
   output reg c_o                              // I also had to make this a reg so that I can use it in my always block.
   );

         //Let us do some logic on our input switches


         always @(q or ena or clr)
                 begin

                         Next_q = 4'b0000;
                         c_o = 4'b0;


                         // ** HOLD **
                         //Logic when ena and clr are set to 0 to hold current values
                         if( (ena == 1'b0) && (clr == 1'b0))
                                 begin
                                         c_o = 1'b0;
                                         Next_q[0] = q[0];
                                         Next_q[1] = q[1];
                                         Next_q[2] = q[2];
                                         Next_q[3] = q[3];
                                 end



                         // ** MAX **
                         //Logic to handle when ena =1 and clr = 1 to get ouput
                         //of 9

                         if ( (ena ==1'b1) && (clr == 1'b1) )
                                 begin
```

```
                        c_o = 1'b1;
                        Next_q = 4'b1001;
                end



        // ** NEXT COUNT **
        //Logic when ena = 1 to calculate our next bit

        if ((ena == 1'b1) && (clr == 1'b0))
                begin

                        Next_q = q + 1;

                        if ( q == 9) c_o = 1;
                        if ( q >= 9) Next_q = 0;

                end // end of if handling the next bit



        // ** CLEAR **
        //Logic when clr = 1 and ena = 0 to clear
        if ((clr == 1'b1) && (ena == 1'b0))
                begin
                        Next_q = 4'b0000; // set to 0000.
                        c_o = 1'b0;                     // set the carry bit to 0.
                end // end if to set Next_1 and c_o to 0.


    end  // End my entire always block



endmodule
```

tb_count4bcd_combin.txt module test file

```
//
// lab5a : version 09/23/2012
//
// This file contains the test vectors for the
// bcd combinational block for a 4 bit BCD counter.
// The first column is the input clr signal
// The second column is the input ena signal
// The next four columns are the inputs: q[3:0]
// The next column is the carry out, c_o
// The next 4 columns are the outputs: Next_q[3:0]
//
// This needs to be 64 lines long to cover all possibilities
//

//Plus 1
01_0000_00001
01_0001_00010
01_0010_00011
01_0011_00100
01_0100_00101
01_0101_00110
01_0110_00111
01_0111_01000
01_1000_01001
01_1001_10000
01_1010_00000
01_1011_00000
01_1100_00000
01_1101_00000
01_1110_00000
01_1111_00000

//Hold
00_0000_00000
00_0001_00001
00_0010_00010
00_0011_00011
00_0100_00100
00_0101_00101
00_0110_00110
00_0111_00111
00_1000_01000
```

```
00_1001_01001
00_1010_01010
00_1011_01011
00_1100_01100
00_1101_01101
00_1110_01110
00_1111_01111

//Clear to 0
10_0000_00000
10_0001_00000
10_0010_00000
10_0011_00000
10_0100_00000
10_0101_00000
10_0110_00000
10_0111_00000
10_1000_00000
10_1001_00000
10_1010_00000
10_1011_00000
10_1100_00000
10_1101_00000
10_1110_00000
10_1111_00000

//Clear to 9 MAX
11_0000_11001
11_0001_11001
11_0010_11001
11_0011_11001
11_0100_11001
11_0101_11001
11_0110_11001
11_0111_11001
11_1000_11001
11_1001_11001
11_1010_11001
11_1011_11001
11_1100_11001
11_1101_11001
11_1110_11001
11_1111_11001
```