**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.

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.

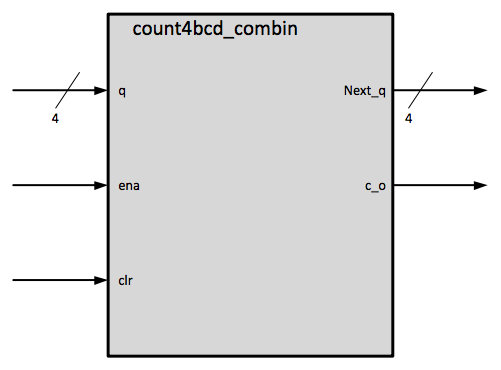


Figure : count4bcd\_combin module

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.

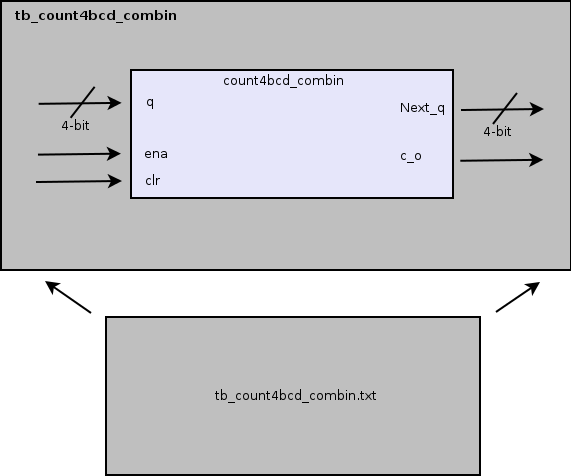


Figure : tb\_count4bcd\_combin test module

**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
   1. input [3:0] q,
   2. input ena,
   3. input clr,
   4. output reg [3:0] Next\_q
   5. 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

1. Verify the tb\_count4bcd\_combin.txt file to contain all possible input bit combinations.
2. Verify the tb\_count4bcd\_combin.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\_count4bcd\_combin
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. Lab5\_top\_io\_wrapper
            1. Implement design
            2. Generation programming file

Transfer .bit file to Board

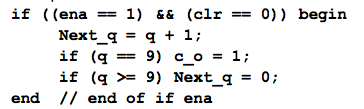
1. Use your favorite network transfer program to move the .bit file from the development server to your local workstation.
2. Plug the board into USB port.
3. Launch the Digilent Adept application on your local workstation.
4. Click the config tab.
5. Click on browse by the PROM icon.
6. Select your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

1. Verify the count4bcd\_combin.txt file to contain all possible input bit combinations.
2. Verify the count4bcd\_combin.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\_count4bcd\_combin
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. Lab5\_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.
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 |

Number of 4-bit LUTs used:

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