# ECE 351 – Spring 2016
# Homework #1
# DUE to your D2L Dropbox by 10:00 PM on Tuesday, April 12

Complete all of the problems and submit your solutions to your Homework #1 dropbox in D2L. You may submit you solution as either a .doc (or .docx) file or as a .pdf file. If you decide to write your answers (instead of typing them in) please make sure you write legibly and please make it clear which part of which question your answer is for (this may seem like a strange request but you'd be surprised how often it is not clear). Verilog code should make liberal use of comments and signal names that convey meaning (a variable named x does not carry as much meaning as a variable named sum, for example). Indent your code appropriately.

Question 1 (40 pts) – Short Answers

a. (20 pts) Circle T if the answer is true or F for false (2 point for each correct answer)

| | | | |
|---|---|---|---|
| 1. | In Verilog an assignment to either an integer or net-type variable causes unsigned arithmetic to take place. | T | F |
| 2. | `x = ^~8'b11001110` will be equal to 0 if it is simulated in Verilog | T | F |
| 3. | The Verilog language supports the ability to mix different levels of abstraction in a single model | T | F |
| 4. | Consider the following Verilog code snippet:<br>`wire [32:0] x;`<br>`wire [15:0] y;`<br><br>`assign x = {b, 2{c,d},e};`<br>`assign y = x;`<br>`where b = 4'b0101, c = 4'hF, d = 8'D114, e = 5'h14`<br><br>After the assign statements execute x and y will have the following values<br>`x = 0101_1111_01110010_1111_01110010_10100`<br>`y = 0101_1111_01110010` | T | F |
| 5. | The following Verilog code is valid<br>`wire [7:0] a, b;`<br>`reg [7:0] sum`<br>`assign sum = a + b;` | T | F |
| 6. | The netlist produced by the synthesis tool is generally technology-independent | T | F |

| | | | |
|---|---|---|---|
| 7. | Numbers that are specified with neither a size nor a base(radix) always result in a 32-bit, unsigned, decimal number (ex: 23456) | T | F |
| 8. | Since the port definitions of a module allow that module to communicate with other modules, every module must have at least one port. | T | F |
| 9. | The system task $stop typically returns control of the simulator to the user | T | F |
| 10. | A string in Verilog is confined to a single line of Verilog code and must be surrounded by double quotes (ex: “Hello”) | T | F |

b. (10 pts) We discussed the four basic steps to create working hardware in class. These four steps are 1) Design entry and analysis, 2) Technology optimization and static timing analysis, 3) Place and route and 4) Validation. Briefly describe what happens during that step.

**During the validation step, the design is tested to see if:**

- **The design function correctly under normal and boundary cases**
- **The design meets its performance goals**
- **The design work over the specified environment conditions**
- **The design and safe**
- **And many more…**

**(These are all were obtained from the lecture notes)**

c. (10 pts). Verilog supports the capability to model an architecture or a circuit at different levels of abstraction. Briefly describe the attributes for each of these models of abstractions:

- Algorithmic (Behavioral)
  **No need to worry about hardware details (i.e delays, behavior). This is the highest level of abstraction. It's similar to programming in C specially**

- Dataflow
  **This comes after the algorithmic level. Must know how data flows by specifying how bits move between blocks**

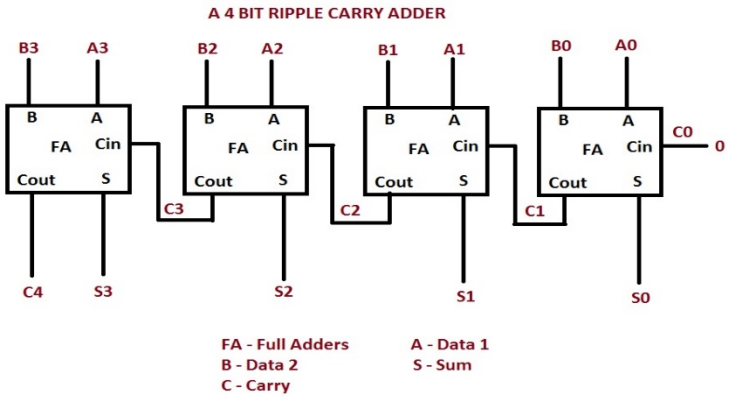
- Gate Level
  **A module is implemented using logic gates. For example, a module with 2 OR gates would be implemented using 2 OR gates, and of course connect them together in the desired way.**

- Switch Level
  **This is the lowest level of abstraction. A module is implemented using switches and storage nodes.**

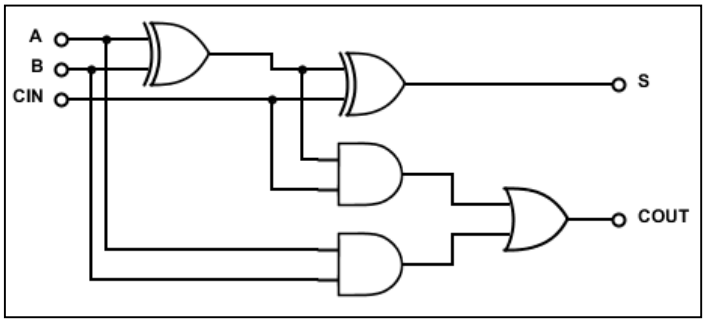
- Register-transfer Level (RTL)
  **This combines algorithmic and data flow to make a module. For example, using if else statements along with some bit manipulations.**

Question 2 (30 pts) – Hierarchical design in Verilog

In this problem we are going to build an 8-bit adder based on the 4-bit Ripple Adder that we started in class.




a. (10 pts*)* Implement the FA module as a gate-level model in Verilog. You may start with the following schematic:




```
module FA(A, B, CIN, S, COUT);

   input A, B, CIN;
   output S, COUT;
   wire S1, S2, S3;

   // xor gate for A and B
   xor x1(S1, A, B);
```

```
    // xor for S1 and CIN to produce the sum
    xor sum(S, S1, CIN);

    // and for S1 and CIN
    and a1(S2, S1, CIN);

    // and for A and B
    and a2(S3, A, B);

    // or to produce COUT
    or cout(COUT, S2, S3);

endmodule
```

b. (5 pts) Implement a 4-bit Ripple Carry Adder module by instantiating four (4) FA instances and wiring them together. This should be a complete module with input and output port declarations.

```
module fourBit_ripple_adder(Ain, Bin, Cin, Sum, Cout);

  input [3:0] Ain, Bin;
  input Cin;
  output [3:0] Sum;
  output Cout;
  wire C1, C2, C3;

  // instantiate FA
  FA ra1(
        .A(Ain[0]),
        .B(Bin[0]),
        .CIN(Cin),
        .S(Sum[0]),
        .COUT(C1)
  );

  FA ra2(
        .A(Ain[1]),
        .B(Bin[1]),
        .CIN(C1),
        .S(Sum[1]),
        .COUT(C2)
  );

  FA ra3(
        .A(Ain[2]),
```

```
        .B(Bin[2]),
        .CIN(C2),
        .S(Sum[2]),
        .COUT(C3)
    );

    FA ra4(
        .A(Ain[3]),
        .B(Bin[3]),
        .CIN(C3),
        .S(Sum[3]),
        .COUT(Cout)
    );
endmodule
```

c. (5 pts) Implement an 8-bit Ripple Carry Adder module by instantiating two (2) 4-bit Ripple Carry Adder modules and wiring them together. This should be a complete module with input and output port declarations.

```
module eightBit_ripple_adder(A, B, CIN, S, COUT);

  input [7:0] A, B;
  input CIN;
  output [7:0] S;
  output COUT;
  wire C1;

  // instantiate 2 four-bit ripple carry adders
  fourBit_ripple_adder r1(
    .Ain(A[3:0]),
    .Bin(B[3:0]),
    .Cin(CIN),
    .Sum(S[3:0]),
    .Cout(C1)
  );
```

```
    fourBit_ripple_adder r2(
        .Ain(A[7:4]),
        .Bin(B[7:4]),
        .Cin(C1),
        .Sum(S[7:4]),
        .Cout(COUT)
    );

endmodule
```

(5 pts) Describe a strategy for implementing a test bench for testing the 8-bit Ripple Carry Adder. Explain why the approach you describe does a good job of testing the adder. Be sure to note the method you will use to generate the test vectors and to confirm that the output of the adder is correct.

**Since this is an 8-bit adder, I would implement a self-check exhaustive test bench. The test vectors would be generated in a loop starting with A = 0, and B would start at 0 and goes all the way to 255. The next iteration would have A = 1 and B goes from 0 to 255. This continues all the way with the last iteration having A = 255, and B goes from 0 to 255. As these vectors are generated and applied to the module, the output would be checked in an if statement. The if statement would check the output of the module with the result of the built-in addition operator. If all outputs match, then we have a complete 8-bit adder.**

d. (5 pts) Assume the test bench you described in the previous problem (part d.). Estimate how much time it will take to run your simulation, assuming the simulator completes one iteration through your DUT (device under test) every 1msec (.001 sec) of simulation time. You do not need to implement the test bench…well, not in this homework assignment, at least.

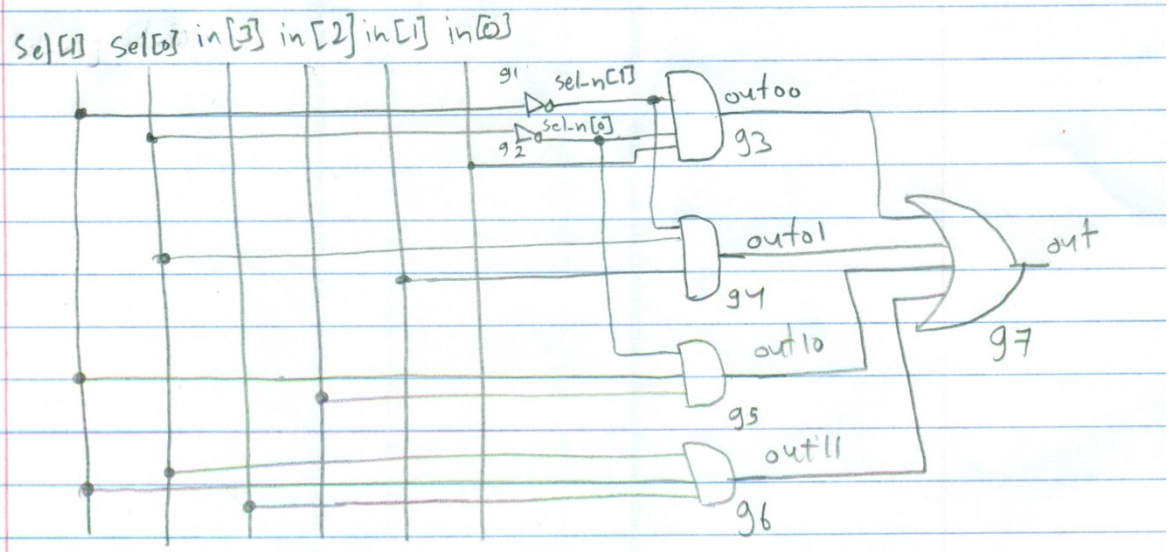
**My test bench would take 256 * 256 = 65,536 ms. or 65.5 secs.**

Question 3 (30 pts) – Verilog coding

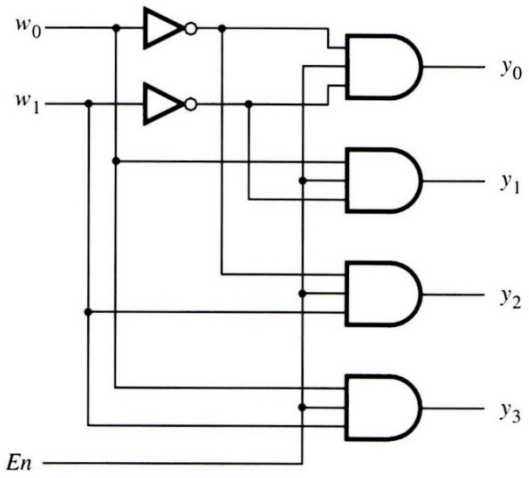
a. (10 pts) Draw a logic schematic for the following Verilog code:

```
// HW1 problem 3a
module hw1_3a (
      input [3:0]       in,
      input [1:0]       sel,
      output            out
);

wire [1:0] sel_n;
wire out00, out01, out10, out11;

not g1(sel_n[1], sel[1]);
not g2(sel_n[0], sel[0]);

and g3(out00, sel_n[1], sel_n[0]);
and g4(out01, sel_n[1], sel[0]);
and g5(out10, sel[1], sel_n[0]);
and g6(out11, sel[1], sel[0]);

or  g7(out, out00, out01, out10, out11);

endmodule
```

ECE 351
HW 1
Q3a



b. (10 pts) Write a gate level Verilog module that implements the following schematic:

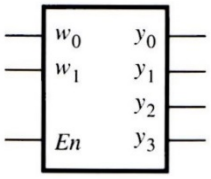



```
module two_to_four(w, En, y);

  input [1:0] w;
  input En;
  output [3:0] y;
  wire n0, n1;

  not g0(n0, w[0]);
  not g1(n1, w[1]);

  and g2(y[0], n0, En, n1);
  and g3(y[1], w[0], En, n1);
  and g4(y[2], n0, En, w[1]);
  and g5(y[3], w[0], En, w[1]);

endmodule
```

c. (10 pts) The circuit you implemented in part b is a 2-4 bit decoder that implements the following truth table:

| $En$ | $w_1$ | $w_0$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | x | x | 0 | 0 | 0 | 0 |

(a) Truth table

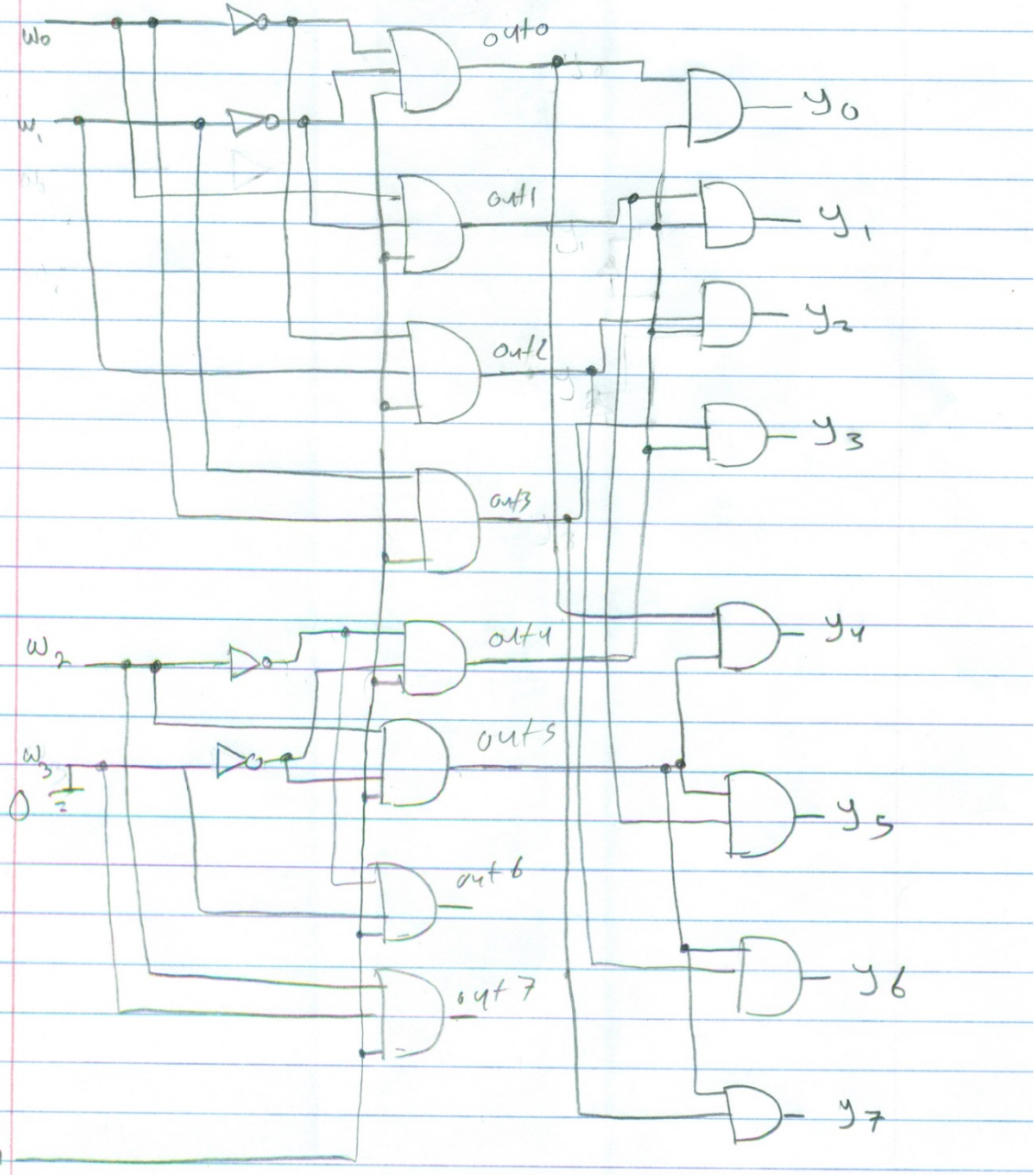



(b) Graphical symbol

Draw a schematic and write a Verilog module for a 3-8 decoder that **uses 2 instances of your 2-4 bit decoder and some additional gates**. A 3-8 decoder implements the following truth table:

| *En* | *w2* | *w1* | *w0* | *y0* | *y1* | *y2* | *y3* | *y4* | *y5* | *y6* | *y7* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | x | x | x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

# ECE 351
# HW1 Q3C



```
module three_to_eight(w, En, y);

   input [2:0] w;
   input En;
   output [7:0] y;
   wire [7:0] out;

   two_to_four d1(w[1:0], En, out[3:0]);
   two_to_four d2({1'b0, w[2]}, En, out[7:4]);

   // produce output with some additional gates
   and g1(y[0], out[0], out[4]);
   and g2(y[1], out[1], out[4]);
   and g3(y[2], out[2], out[4]);
   and g4(y[3], out[3], out[4]);

   and g5(y[4], out[0], out[5]);
   and g6(y[5], out[1], out[5]);
   and g7(y[6], out[2], out[5]);
   and g8(y[7], out[3], out[5]);

endmodule
```