Name: **Jonathan Lemarroy**

ECE 3544: Digital Design 1
Homework Assignment 4 (80 points)

When writing a Verilog module or naming a file, replace YOURPID with your Virginia Tech PID. Name the file according to the module name that you use.

For each design that you represent in a Verilog module, copy your source code into the document. For each design that you simulate in ModelSim, include waveforms displaying the correct operation of each module

In addition, submit the .v file containing each design and each test bench that you write. Your files should contain header information as described in Project 1 and be neatly formatted and commented. Submit your files separately on Canvas. Do not put them into an archive. *Make sure that you upload all of your files.*

Helpful Hints for Implementing Combinational Logic Functions using Combinational Logic Blocks

- To implement a combinational logic function having n input variables, we need a decoder having n inputs and an OR gate. Each of the decoder outputs represents a possible minterm of the function. Connect the decoder outputs that correspond to the function's minterms to inputs of the OR gate. The output of the OR gate represents the output of the function.

  We can use the same decoder to implement multiple functions of the same set of input variables. As long as we have one OR gate per function output, we can connect the decoder outputs that correspond to minterms of each function to the inputs of the OR gate for that function output.

- One way to implement an n-variable combinational logic function using a multiplexer is to use a multiplexer having the same number of select lines as the function has input variables. Each of the multiplexer inputs corresponds to a possible minterm of the function. Connect logic-1 to the multiplexer inputs that correspond to the function's minterms. Connect logic-0 to the multiplexer inputs that do not correspond to the function's minterms. This will cause the multiplexer to output 1 or 0 for each combination of the input variables, as determined by the function's minterm set. The output of the multiplexer is also the output of the function.

  Unlike the decoder implementation, this method can only implement one function at a time. But it doesn't need any additional logic.

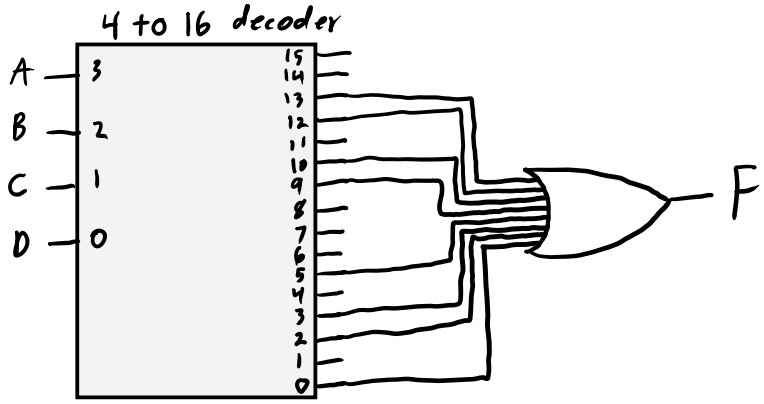
- We can implement an n-variable combinational logic function using a multiplexer that has fewer select lines than the function has variables. If we apply one input variable to each available multiplexer select, we can then use the remaining variables to generate functions that – when applied to the multiplexer inputs – cause the multiplexer's output to be the function's output, as before. The general approach is to use the choice of multiplexer inputs to partition a truth table, and then to use the partitioned truth table to determine how the function relates to the remaining variables.

Problem 1 (12 points)
Implement the function F(A, B, C, D) = Σm(0, 2, 3, 5, 9, 10, 12, 13) using:

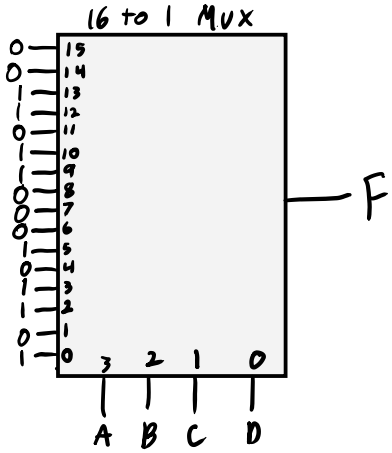
a. A 4-to-16 decoder and an OR gate.
b. A 16-to-1 multiplexer.
c. An 8-to-1 multiplexer and (at most) one inverter. Use the method described in class.

For each part of Problem 1, your solution should include a neatly-drawn schematic.

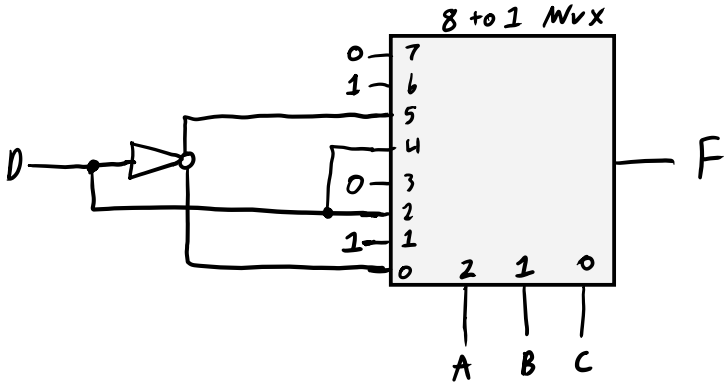
a)



b)



c)



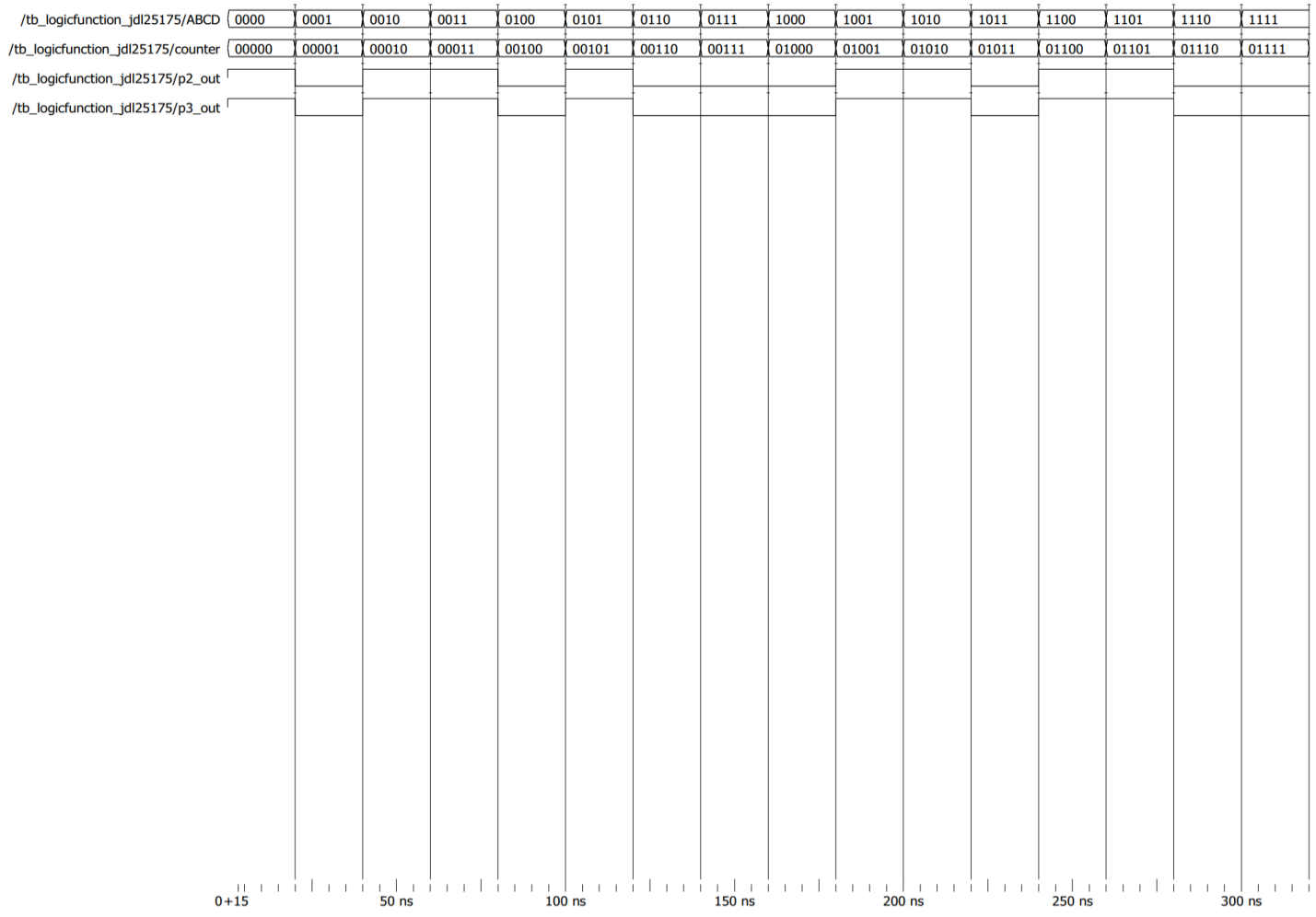
Problem 2 (14 points)
Implement your solution for Problem 1(a) in Verilog and verify its operation.

a. (6 points) First, create a behavioral module for a 4-to-16 decoder. Each scalar component of `dec_out` represents one of the sixteen decoder outputs. When `enable` is asserted and the value of `dec_in` equals *k*, `dec_out[k]` should equal 1. All other components of `dec_out` should equal 0. When `enable` is not asserted, all components of `dec_out` should equal 0 regardless of *k*'s value.

```
// Filename:    decoder4to16_jdl25175.v
// Author:      Jonathan Lemarroy
// Date:        15 October 2020
// Version:     1
// Description: This is a behavioral 4 to 16 decoder whose default output value is 0
//              unless enable and dec_in pins are correctly active.

module decoder4to16_jdl25175(enable, dec_in, dec_out);
    input           enable; // active-high enable
    input     [3:0] dec_in; // decoder inputs
    output   [15:0] dec_out;// decoder outputs
    reg      [15:0] temp;

    always @(enable, dec_in) begin
        if(enable) begin
            case(dec_in)
                4'b0000: temp = 16'b0000000000000001;
                4'b0001: temp = 16'b0000000000000010;
                4'b0010: temp = 16'b0000000000000100;
                4'b0011: temp = 16'b0000000000001000;
                4'b0100: temp = 16'b0000000000010000;
                4'b0101: temp = 16'b0000000000100000;
                4'b0110: temp = 16'b0000000001000000;
                4'b0111: temp = 16'b0000000010000000;
                4'b1000: temp = 16'b0000000100000000;
                4'b1001: temp = 16'b0000001000000000;
                4'b1010: temp = 16'b0000010000000000;
                4'b1011: temp = 16'b0000100000000000;
                4'b1100: temp = 16'b0001000000000000;
                4'b1101: temp = 16'b0010000000000000;
                4'b1110: temp = 16'b0100000000000000;
                4'b1111: temp = 16'b1000000000000000;
                default: temp = 16'b0000000000000000;
            endcase
        end
        else
            temp = 16'b0000000000000000;
    end

    assign dec_out = temp;
endmodule
```

b. (8 points) Next, create a module to implement the logic function. Use the following module declaration. This module must instantiate one instance of your 4-to-16 decoder module. Your instantiation must properly use the input variables of the logic function as the decoder inputs. Your module must also use an OR-function to collect the appropriate decoder outputs. It need not be a primitive OR-gate.

```
// Filename:    problem2_jdl25175.v
// Author:      Jonathan Lemarroy
// Date:        15 October 2020
// Version:     1
// Description: This is the solution to Problem 2 part b, which implements the
//              function F using a 4 to 16 decoder.

module problem2_jdl25175(a, b, c, d, f);
    input  a, b, c, d;      // The input variables for the function
    output f;               // The output of the function

    wire [15:0] o;

    decoder4to16_jdl25175 den(1'b1, {a,b,c,d}, o);

    or out1(f, o[13], o[12], o[10], o[9], o[5], o[3], o[2], o[0]);

endmodule
```



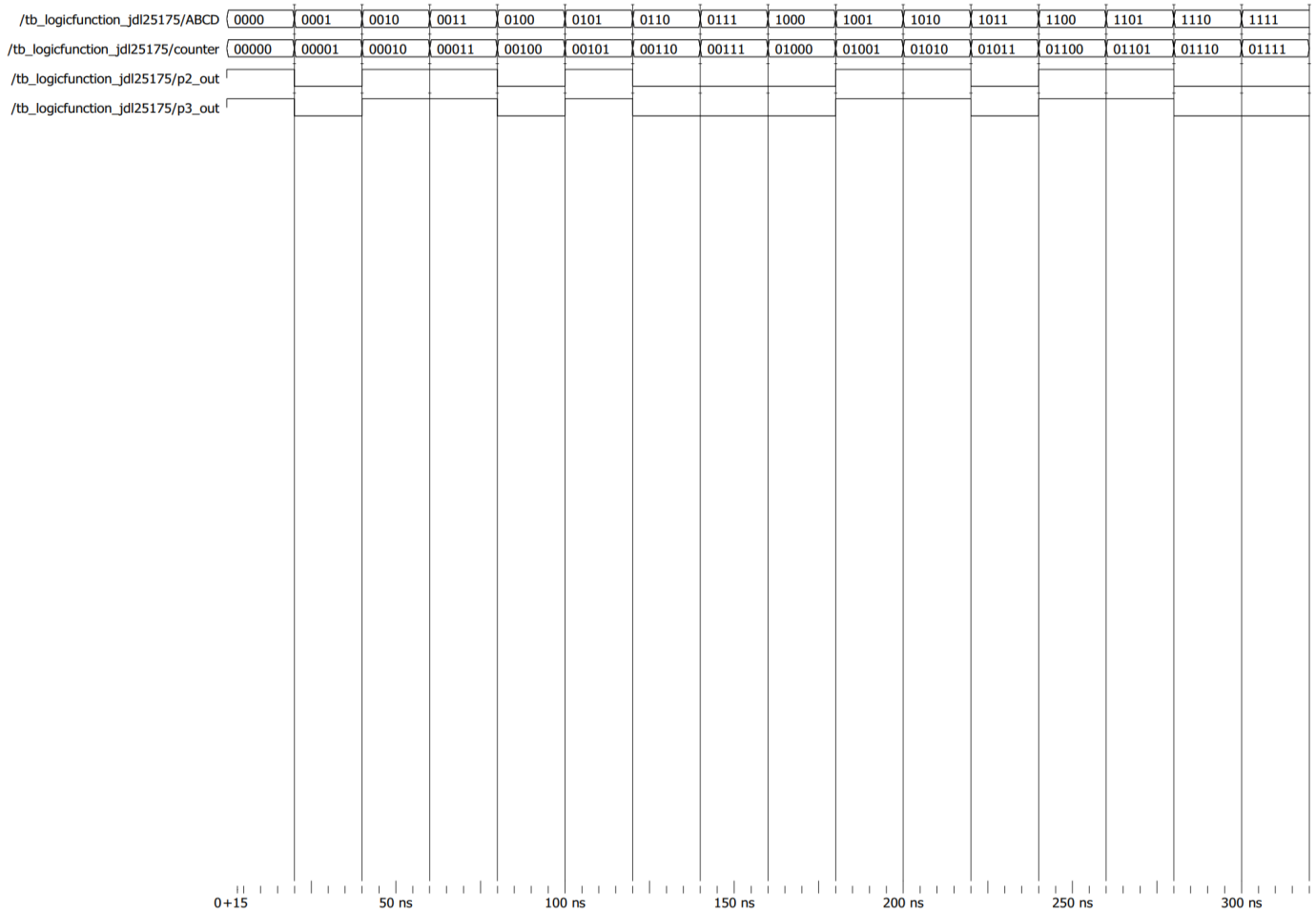
Problem 3 (14 points)
Implement your solution for Problem 1(c) in Verilog and verify its operation.

a. (6 points) First, create a behavioral module for an 8-to-1 multiplexer. Each scalar component of `mux_in` represents one of the eight multiplexer inputs. When `enable` is asserted and the value of `select` equals *k*, `mux_out` should equal `mux_in[k]`. When `enable` is not asserted, `mux_out` should equal 0 regardless of *k*'s value.

```
// Filename:    mux8to1_jdl25175.v
// Author:      Jonathan Lemarroy
// Date:        15 October 2020
// Version:     1
// Description: This is a behavioral 8 to 1 multiplexer whose default value is 0
//              unless enable and select pins are correctly active.

module mux8to1_jdl25175(enable, select, mux_in, mux_out);
    input       enable;     // active-high enable
    input [2:0] select;     // multiplexer select lines
    input [7:0] mux_in;     // multiplexer input lines
    output      mux_out;    // multiplexer output
    reg         temp;

    always @(enable, select, mux_in) begin
        if(enable) begin
            case(select)
                3'b000: temp = mux_in[0];
                3'b001: temp = mux_in[1];
                3'b010: temp = mux_in[2];
                3'b011: temp = mux_in[3];
                3'b100: temp = mux_in[4];
                3'b101: temp = mux_in[5];
                3'b110: temp = mux_in[6];
                3'b111: temp = mux_in[7];
                default: temp = 1'b0;
            endcase
        end
        else
            temp = 1'b0;
    end

    assign mux_out = temp;
endmodule
```

b. (8 points) Next, create a module to implement the logic function. This module must instantiate one instance of your 8-to-1 multiplexer module. Your instantiation must properly use the input variables of the logic function as the select lines and input lines of the multiplexer. You may invert any of the input variables of the function, but your implementation should be simple in form.

```
// Filename:    problem3_jdl25175.v
// Author:      Jonathan Lemarroy
// Date:        15 October 2020
// Version:     1
// Description: This is the solution to Problem 3 part b, which implements the
//              function F using a 8 to 1 multiplexer.

module problem3_jdl25175(a, b, c, d, f);
    input  a, b, c, d;      // The input variables for the function
    output f;               // The output of the function

    mux8to1_jdl25175 den(1'b1, {a,b,c}, {1'b0, 1'b1, ~d, d, 1'b0, d, 1'b1, ~d}, f);

endmodule
```



Problem 4 (16 points)

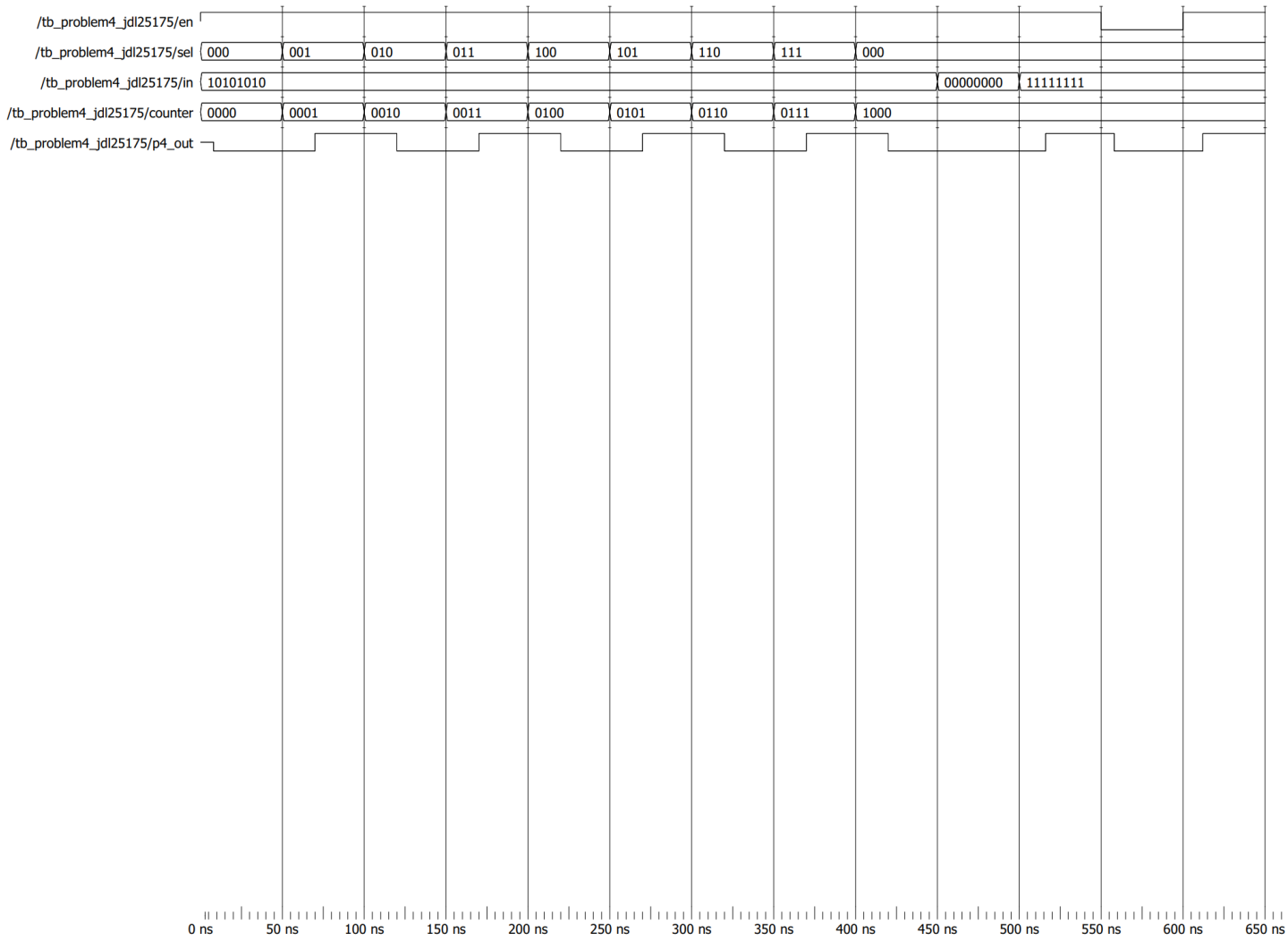
Using the 8-to-1 multiplexer that you wrote in Problem 3, use a `specify` block to give the multiplexer the following delay characteristics:

Propagation Path	t_{PHL}	t_{PLH}
`enable` to `mux_out`	8 ns	12 ns
`select` to `mux_out`	20 ns	20 ns
`mux_in` to `mux_out`	12 ns	16 ns

Create a behavioral model for an 8-to-1 multiplexer named `mux8to1_delay_YOURPID`. The model for this multiplexer should be identical to the 8-to-1 multiplexer that you wrote in Problem 3, with the addition of the `specify` block that models the input-to-output delays described in the table above.

```
// Filename:    mux8to1_delay_jdl25175.v
// Author:      Jonathan Lemarroy
// Date:        15 October 2020
// Version:     1
// Description: This is a behavorial 8 to 1 multiplexer whose default value is 0
//              unless enable and select pins are correctly active.

// Time Unit = 1 ns (#1 means 1 ns)
// Simulation Precision = 1 ns
`timescale 1ns/1ns

module mux8to1_delay_jdl25175(enable, select, mux_in, mux_out);
    input       enable;     // active-high enable
    input [2:0] select;     // multiplexer select lines
    input [7:0] mux_in;     // multiplexer input lines
    output      mux_out;    // multiplexer output
    reg         temp;

    always @(enable, select, mux_in) begin
        if(enable) begin
            case(select)
                3'b000: temp = mux_in[0];
                3'b001: temp = mux_in[1];
                3'b010: temp = mux_in[2];
                3'b011: temp = mux_in[3];
                3'b100: temp = mux_in[4];
                3'b101: temp = mux_in[5];
                3'b110: temp = mux_in[6];
                3'b111: temp = mux_in[7];
                default: temp = 1'b0;
            endcase
        end
        else
            temp = 1'b0;
    end
```

```
    assign mux_out = temp;

    specify
        (enable => mux_out) = (12, 8);
        (select[0] => mux_out) = (20, 20);
        (select[1] => mux_out) = (20, 20);
        (select[2] => mux_out) = (20, 20);
        (mux_in[0] => mux_out) = (16, 12);
        (mux_in[1] => mux_out) = (16, 12);
        (mux_in[2] => mux_out) = (16, 12);
        (mux_in[3] => mux_out) = (16, 12);
        (mux_in[4] => mux_out) = (16, 12);
        (mux_in[5] => mux_out) = (16, 12);
        (mux_in[6] => mux_out) = (16, 12);
        (mux_in[7] => mux_out) = (16, 12);
    endspecify

endmodule
```



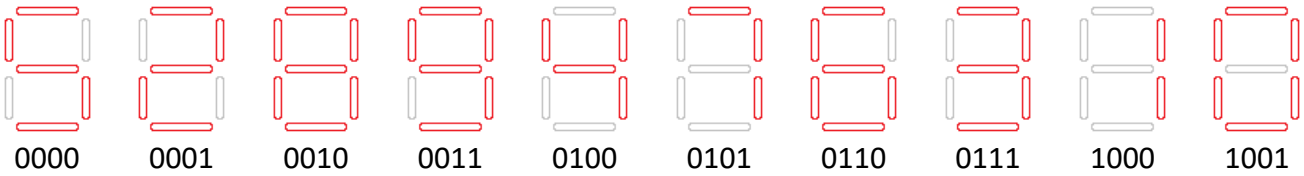
Problem 5 (14 points)
A decimal decoder (also sometimes called a 4-to-10 decoder) accepts a 4-bit input and asserts one of **ten** outputs based on the input value.

a. (6 points) Write a behavioral Verilog model for a decimal decoder. As before, each scalar component of `dec_out` represents one of the ten decoder outputs. When `enable` is asserted and the value of `dec_in` equals *k*, `dec_out[k]` should equal 1. All other components of `dec_out` should equal 0. For "invalid" values of `dec_in`, all of the components of `dec_out` should equal 0. When `enable` is not asserted, all components of `dec_out` should equal 0 regardless of *k*'s value.

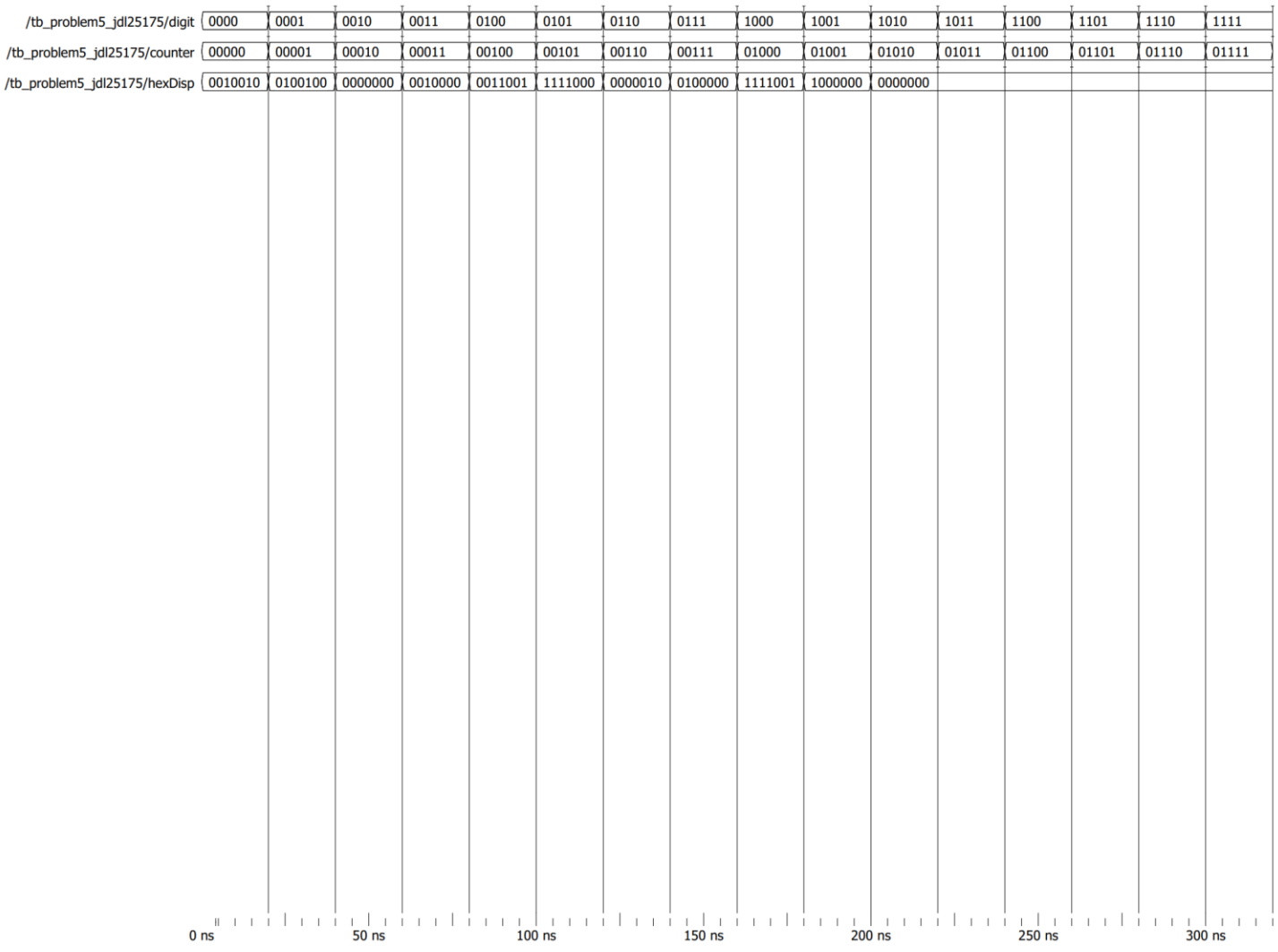
```
// Filename:    decoder4to10_jdl25175.v
// Author:      Jonathan Lemarroy
// Date:        15 October 2020
// Version:     1
// Description: This is a behavioral 4 to 10 decoder whose default output value is 0
//              unless enable and dec_in pins are correctly active.

module decoder4to10_jdl25175(enable, dec_in, dec_out);
    input           enable; // active-high enable
    input     [3:0] dec_in; // decoder inputs
    output    [9:0] dec_out;// decoder outputs
    reg       [9:0] temp;   // temporarily holds the decoders output.

    always @(enable, dec_in) begin
        if(enable) begin
            case(dec_in)
                4'b0000: temp = 10'b0000000001;
                4'b0001: temp = 10'b0000000010;
                4'b0010: temp = 10'b0000000100;
                4'b0011: temp = 10'b0000001000;
                4'b0100: temp = 10'b0000010000;
                4'b0101: temp = 10'b0000100000;
                4'b0110: temp = 10'b0001000000;
                4'b0111: temp = 10'b0010000000;
                4'b1000: temp = 10'b0100000000;
                4'b1001: temp = 10'b1000000000;
                default: temp = 10'b0000000000;
            endcase
        end
        else
            temp = 10'b0000000000;
    end

    assign dec_out = temp;
endmodule
```

b. (8 points) Create a module to implement a set of driver circuits for the seven-segment display described in Homework Assignment 3. This module must instantiate one instance of your decimal decoder. Your instantiation must properly use the input variables of the logic function as the decoder inputs. Your module must also use an OR-function to collect the appropriate decoder outputs to implement each of the seven driver functions described below. They need not be primitive OR-gates.

The value applied as the decoder input should cause the seven-segment display to show the following characters. Assume that the user will not apply the invalid BCD cases. *The number order shown is intentional; I like "alphabetizing" the numbers*.




```
// Filename:    problem5_jdl25175.v
// Author:      Jonathan Lemarroy
// Date:        15 October 2020
// Version:     1
// Description: This is the solution to Problem 5 part b, which implements the
//              seven-segment display using a 4 to 10 decoder.

//              ****IMPORTANT**** --> This implements a 7 segement display that
//              is active when logic 0 is applied. The display will enable all
//              segments if digits are z or x.

module problem5_jdl25175(digit, hex_display);
    input  [3:0] digit;
    output [6:0] hex_display;

    wire   [9:0] dOut;

    decoder4to10_jdl25175 dec(1'b1, digit, dOut);

    or or0(hex_display[0], dOut[4], dOut[8]);
    or or1(hex_display[1], dOut[0], dOut[6]);
    or or2(hex_display[2], dOut[1], 1'b0);
    or or3(hex_display[3], dOut[4], dOut[5], dOut[8]);
    or or4(hex_display[4], dOut[0], dOut[3], dOut[4], dOut[5], dOut[8]);
    or or5(hex_display[5], dOut[1], dOut[5], dOut[7], dOut[8]);
    or or6(hex_display[6], dOut[5], dOut[8], dOut[9]);

endmodule
```


/tb_problem5_jdl25175/digit
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
/tb_problem5_jdl25175/counter
00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
01101
01110
01111
/tb_problem5_jdl25175/hexDisp
0010010
0100100
0000000
0010000
0011001
1111000
0000010
0100000
1111001
1000000
0000000
0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns

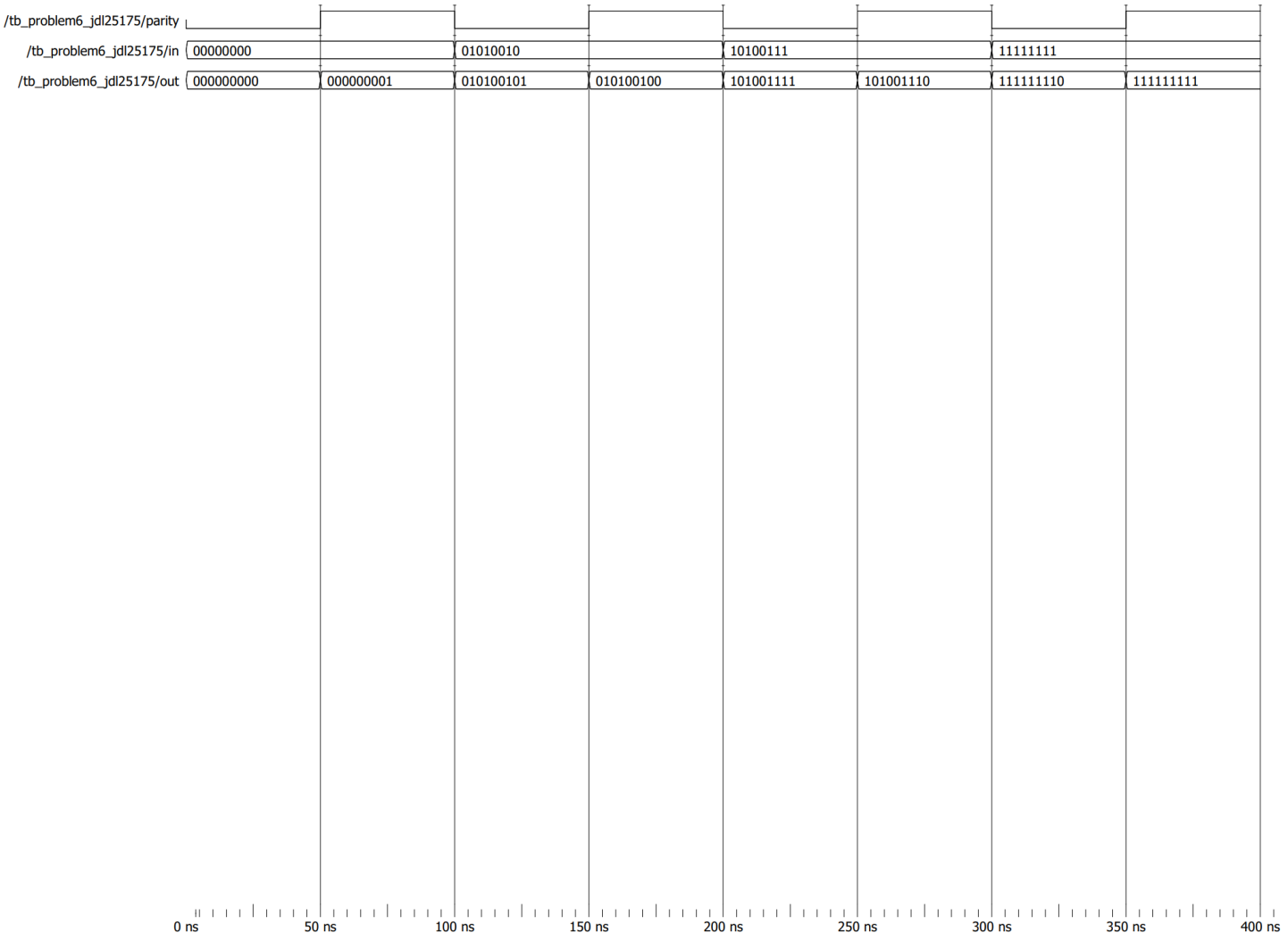
Problem 6 (10 points)

As part of the creation of a hypothetical UART, we might wish to create a module that takes an 8-bit input word (`input_word`) and generates a 9-bit word (`output_word`) for transmission. The transmitted word consists of the eight bits of the input word, followed by a parity bit in the LSB position. The module uses a control bit (`parity_control`) to determine the kind of parity with which the word will be transmitted: 0 for even parity, and 1 for odd parity.

For example, given the input word 0x68 (01101000), the module would generate the output word 0x0D1 (011010001) if the parity control bit equaled 0, and 0x0D0 (011010000) if the parity control bit equaled 1.

Write a behavioral Verilog module that implements the parity generation scheme described. *For "fun": see how "compact" you can make your module.*

```
// Filename:    problem6_jdl25175.v
// Author:      Jonathan Lemarroy
// Date:        15 October 2020
// Version:     1
// Description: This is the solution to Problem 6, which implements a
//              circuit designed to add a parity bit on the end;

module problem6_jdl25175(parity_control, input_word, output_word);
    input        parity_control;    // 0 - even, 1 - odd
    input  [7:0] input_word;
    output [8:0] output_word;
    wire         odd, parityBit;
    assign odd = ((input_word[0] ^ input_word[1]) ^ (input_word[2] ^ input_word[3])) ^
                 ((input_word[4] ^ input_word[5]) ^ (input_word[6] ^ input_word[7]));

    assign parityBit = odd ^ parity_control;
    assign output_word = {input_word, parityBit};

endmodule
```



Input Word	Output Word (odd parity)	Output Word (even parity)
00000000	000000001	000000000
01010010	010100100	010100101
10100111	101001110	101001111
11111111	111111111	111111110