Name:

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

Submitting Your Work
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.

Place your files into an archive. Name the archive hw4_YOURPID.zip.

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.

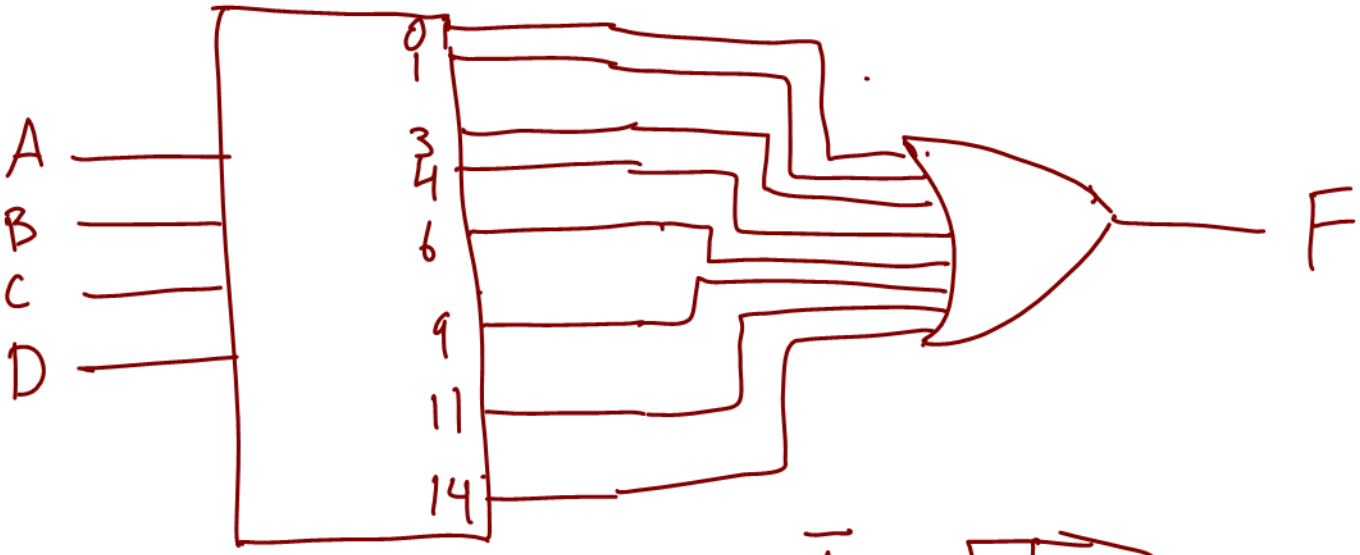
  Unlike the decoder implementation, this method can only implement one function at a time. But it doesn't need any additional logic.

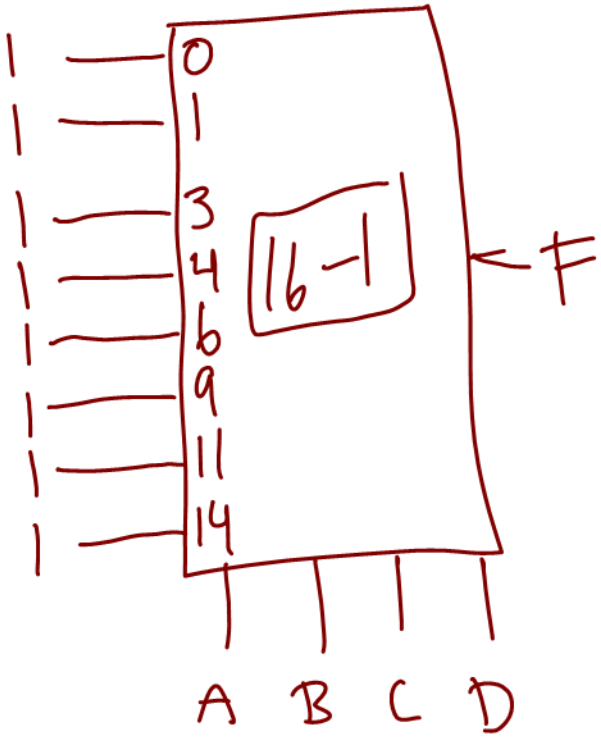
- We can implement an n-variable combinational logic function using a multiplexer that has fewer select lines than the function has variables. If we apply one input variable to each available multiplexer select, we can then use the remaining variables to generate functions that – when applied to the 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.

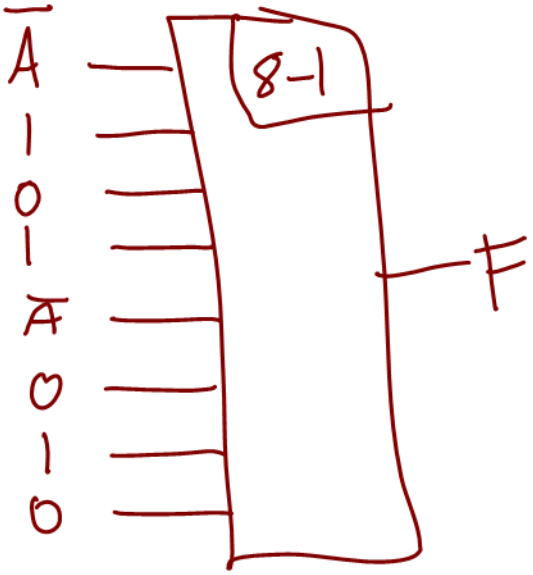
1. (12 points) Implement the function F(A, B, C, D) = Σm(0, 1, 3, 4, 6, 9, 11, 14) using:

   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.







$\bar{A}$ 0 1 2 3 9 5 6 7

A 8 9 10 11 12 13 14 15

$\bar{A}$ 1 0 1 $\bar{A}$ 0 1 0

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. Use the following module declaration. You may assign the `reg` type to an entity to make it into a variable separately from declaring the entity as an `output`. Remember that any value targeted by a procedure must be typed as a `reg`.

```
module decoder4to16_YOURPID(enable, dec_in, dec_out);
   input         enable;     // active-high enable
   input   [3:0] dec_in;     // decoder inputs
   output [15:0] dec_out;    // decoder outputs
```

   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.

   b. (8 points) Next, create a module to implement the logic function. Use the following module declaration.

```
module problem2_YOURPID(a, b, c, d, f);
   input  a, b, c, d;         // The input variables for the function
   output f;                  // The output of the function
```

   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.

   Create a test bench (`tb_logicfunction_YOURPID`) to validate your module and show correct operation for all input combinations. Your homework solution document should include waveforms displaying correct operation. You may use the same test bench to simulate and test your solutions to Problems 2 and 3. If you do it correctly, you can test both modules at the same time. *You would do well to write a test bench for your decoder module, but you need not submit it as part of your homework.*

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. Use the following module declaration. You may assign the `reg` type to an entity to make it into a variable separately from declaring the entity as an `output`. Remember that any value targeted by a procedure must be typed as a `reg`.

```
module mux8to1_YOURPID(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
```

   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.

   b. (8 points) Next, create a module to implement the logic function. Use the following module declaration.

```
module problem3_YOURPID(a, b, c, d, f);
   input  a, b, c, d;    // The input variables for the function
   output f;             // The output of the 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.

   Use the test bench `tb_logicfunction_YOURPID` to validate your module and show correct operation for all input combinations. Remember that if you do it correctly, you can test both modules at the same time. *You would do well to write a test bench for your decoder module, but you need not submit it as part of your homework.*

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_{PLH}	t_{PHL}
`enable` to `mux_out`	20 ns	15 ns
`select` to `mux_out`	30 ns	30 ns
`mux_in` to `mux_out`	25 ns	20 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.

Create a test bench (`tb_problem4_YOURPID`) to demonstrate that your delay model obeys the characteristics described in the table. Instantiate the model without delays and the model with delay so that you can compare how the two behave.

The waveforms you submit in your homework solution document should display the correct delay characteristics – that is, they should demonstrate that changes in the value of `mux_out` (from low-to-high and high-to-low) are delayed by the correct amount of time relative to the input events given in the table: changes in the value of `enable`, the value of `select`, and the value of `mux_in`.

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. Use the following module declaration. You may assign the `reg` type to an entity to make it into a variable separately from declaring the entity as an `output`. Remember that any value targeted by a procedure must be typed as a `reg`.

```
module decoder4to10_YOURPID(enable, decoder_input, decoder_output);
   input        enable;       // active-high enable
   input  [3:0] dec_in;       // decoder inputs
   output [9:0] dec_out;      // active-high decoder outputs.
```

   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.

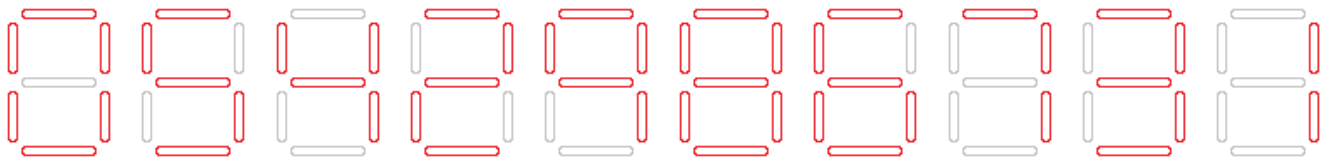
   b. (8 points) Create a module to implement a set of driver circuits for the seven-segment display described in Homework Assignment 3. Use the following module declaration:

```
module problem5_YOURPID(digit, hex_display);
   input  [3:0] digit;
   output [6:0] hex_display;
```

   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 inputs of the decoder. Your module must also use an OR-function to collect the appropriate decoder outputs. It need not be a primitive OR-gate.

   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. (The order of the digits is intentional, and follows an exceedingly secret order that I don't think anyone will figure out…) Assume that the user will not apply the invalid BCD cases.



   Create a test bench (`tb_problem5_YOURPID`) to validate your module and show correct operation for all input combinations. Your homework solution document should include waveforms displaying correct operation. *You would do well to write a test bench for your decoder module, but you need not submit it as part of your homework.*

6. (10 points) You don't need to know anything about UART or serial communications to solve this problem, but those of you who have taken or are taking ECE 2534 might recall that when transmitting bits over a serial channel, a user might wish to append a parity bit onto the end of the word that is being transmitted.

   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. Use the following module declaration:

```
module problem6_YOURPID(parity_control, input_word, output_word);
   input        parity_control; // 0 - odd, 1 - even
   input  [7:0] input_word;
   output [8:0] output_word;
```

   For "fun": see how "compact" you can make your module.

   Create a test bench (`tb_problem6_YOURPID`) to validate your module and show correct operation. You need not submit a waveform that shows all possible input cases (of which there are 512) even though I encourage you to fully verify your module by testing all cases. For your homework submission, use your module to complete the table below, and include waveforms displaying correct operation for these cases in your homework solution document.

Input Word	Output Word (odd parity)	Output Word (even parity)
00000000		
01000110		
11010101		
11111111		