In this assignment, we will revisit some of the problems from Homework 4 and implement them in Verilog.

Using Blackboard, please submit a zip file containing the Verilog sources for your modules. The zip file should contain seven files, each one with the same name as the module that it implements (e.g. less_than_three.v).

You should write test bench code to test your Verilog modules, but you don’t need to submit your test bench code. I will run your modules against my own test bench.

1) Design a Verilog module
   
   module less_than_three(A, Y);

   that takes as input a 3-bit binary number $A$ and outputs a single bit $Y$ which is equal to 1 if and only if $A$ is less than three. Note that if you carefully study the “Operators” section of the cheat sheet linked to from the course website, this could be a very short module!

2) Design a Verilog module

   module majority(A, Y);

   that takes a 3-bit input $A$ and outputs a single bit $Y$ which is equal to 1 if and only the majority of the input bits are 1’s.

3) Design a Verilog module

   module hw4_3a(X, Y);

   that takes as input a 3-bit binary number $X$ and outputs a 3-bit binary number $Y$, according to the following: when $X$ is less than 4, outputs two greater than $X$; when $X$ is greater than or equal to 4, outputs three less than $X$.

4) Design a Verilog module

   module sort_bits(X, Y);

   that takes a 3-bit input $X$ and outputs a 3-bit output $Y$ which is equal to the bits of $X$ sorted from least to greatest (0’s, then 1’s).

5) Design a Verilog module

   module binary_to_gray(B, G);

   that takes as input a 4-bit binary number $B$ and returns the corresponding $B^{th}$ 4-bit Gray code $G$. 
6) Modify the 2-to-4 decoder Verilog example from the course website to produce a module

```verilog
module 2_to_4_with_enable(EN, I, Y);
```

that implements a 2-to-4 decoder with enable.

7) Design a module

```verilog
module 4_to_16_with_enable(EN, I, Y);
```

that implements a 4-to-16 decoder with enable by instantiating your 2-to-4 decoder module from the previous problem five times.

8) For the function

\[ F(a, b, c, d) = \Sigma(1, 2, 3, 6, 7, 8, 9, 11, 12, 15) \]

a. Use Shannon's expansion theorem to implement \( F \) with a 4x1 multiplexer, using \( a \) and \( b \) as the address line inputs. Simplify the remaining 4 inputs by combining product terms. Draw a diagram.

b. Do the same, except now using \( c \) and \( d \) as the address line inputs. Again, draw a diagram.

c. Draw a K-map for the function. Each non-address input to your multiplexer corresponds to a particular 4-cell region of the K-map. What regions correspond to the 4 inputs of the first multiplexer? The second?

d. Can you explain, using the K-map, why the inputs to the second multiplexer are less complex than the inputs to the first?

9) In class we saw the Shannon Expansion theorem, which states that for a Boolean function of \( n \) variables,

\[ F(w_1, w_2, ..., w_n) = w_1' \cdot F(0, w_2, ..., w_n) + w_1 \cdot F(1, w_2, ..., w_n) \]

a. Prove that the Shannon expansion theorem works for any Boolean function expressed as sum of product terms.

b. State why it works for any Boolean function, not just SOP functions.

c. Consider a Boolean function \( F(a, b, c, d) \). We will implement it with an 8x1 multiplexer, using the variables \( a, b, \) and \( c \) as inputs to the address lines. What are the only four possible inputs that we can send into the remaining 8 input lines, and why?