CSCI 2500 — Computer Organization
Lab 08 (document version 1.0)

- This lab is due by the end of your lab session on Wednesday, October 23, 2019.
- This lab is to be completed **individually**. Do not share your code with anyone else.
- You **must** show your code and your solutions to a TA or mentor to receive credit for each checkpoint.
- Labs are available by Mondays before your lab sessions. Plan to start each lab early and ask questions during office hours, in the discussion forum on Submitty, and during your lab session.

1. **Checkpoint 1:** For the first (and second) checkpoint, you will use C to finish implementing a simulation of logical NOT, logical OR, logical AND, and logical XOR.

   Download the `lab08.c` code, which provides fill-in-the-blank skeletal code for these logical gates, as well as unit test code in `main()`. Fill in the `not_gate()`, `or_gate()`, `and_gate()`, and `xor_gate()` functions. Verify that the truth table outputs are correct.

2. **Checkpoint 2:** For the second checkpoint, continue to add to the `lab08.c` code by implementing the `multiplexer()`, `decoder()`, and `alu_1bit()` functions. Make sure that whenever possible you reuse functions that you already implemented. Also add code to `main()` to comprehensively test your `decoder()` and `alu_1bit()` functions. As above, verify that the truth table outputs are correct.

   Then, implement the `alu_4bit()` function reusing `alu_1bit()` function. Verify the correctness of your implementation of `alu_4bit()` by comparing the results with the "expected" values.

3. **Checkpoint 3:** For the third checkpoint, write a function called `ieee754encode()` that has the function prototype shown below and generates the encoding of a single-precision floating-point value in its 32-bit binary form.

   ```
   void ieee754encode( float value, char * encoded );
   ```

   This function accepts two arguments. The `value` argument is the actual single-precision floating-point value to be encoded (e.g., `57.75`). The `encoded` argument points to where the normalized binary string should be stored, with the binary string representing the IEEE 754-1985 form (e.g., `"01000010011001110000000000000000"`).

   You can assume that the `encoded` argument points to a valid chunk of memory of at least 33 bytes. And as a character string, you will need to generate the correct series of `'0'` and `'1'` characters. **Be sure to use the normalized form.**

   Your function should output the following debugging information:

   ```
   input: 57.750000
   sign: 0
   exponent: 10000100
   fraction: 11001110000000000000000
   output: 01000010011001110000000000000000
   ```