CS 310H **Computer Organization and Programming** **Spring 2008**
Professor Fussell **Due: Friday Feb. 22, 2008**

Assignment #4

Instructions: Jeff will be presenting at a conference from Tuesday, Feb 19^{th} for the rest of the week. As a result, **there will be no discussion section on Thursday, Feb** 21^{st}. Instead, turn in your homework to Don during Friday's class. Because BOTH discussion sections will be turning in your hwk at once, PLEASE clearly indicate whether you are in the 9am or 11am section on your paper. As always, make sure your assignment is neat, stapled, and is *entirely your own work* ^{TM}.

1. You are happily walking along to your CS310H discussion section, when all of a sudden a strange man in a dark suit runs up to you and hands you a brief case. He is suddenly gunned down from at least 7 different directions and killed. Thinking nothing of it at the time, you simply run to class. Later that night you remember about the brief case, and you open it to find it contains high level logic components! You carefully lay out the contents of the brief case:

   - A single 1-bit full adder.
   - Seven master slave flip flops, each built with two D-latches and an inverter.
   - Lots and lots of wires.
   - The back of the brief case has wire connections called "Ain, Bin, and CLK".

   A note in the back of the brief case says that unless someone can hook up the pieces to create a 4-bit adder by February 22, the World will be destroyed!

   Hints: This is called a **bit-serial adder**. Assume that the input signals are A (1 bit) and B (1 bit) and a clock. On each rising edge of the clock, the next bit of the four bit input for A and B arrives on the A and B wires (at the start of cycle 0, A[0] arrives on the A wire, at the next cycle rising edge, A[1] appears on the wire, then A[2] on the next cycle, and finally A[3]. Your four-bit adder should end up with each of the sum bits S[0] through S[3] in a separate flip-flop. You may assume that the values of all the flip-flops are initially zero. No need to determine when the 4 bits are done.

2. **Games of State:** Give an upper bound for the number of states it would take to implement (naively) the entire game of tic-tac-toe. (This was implemented in real life using nothing but tinker toys as computational elements! It was also implemented in 1952 for the EDSAC computer under the name "Noughts and Crosses") We will do something a little simpler. Write a state machine that will win at a game of NIM starting with just 2 piles of sticks. (Look up the rules of NIM and winning strategies.) Last person to take a stick loses. The only information (state) you know about the 2 piles of sticks is whether they have an ODD, EVEN, or ZERO amount of sticks in each pile. Your OUTPUT at each state consists of "REMOVE ONE FROM i" or "REMOVE TWO FROM i" (where i is the pile number, 0 or 1) or "I WON" or "I LOST", with an appropriate change of state. In this example, assume that the "input" from the other player merely serves to magically change your state. For full credit, your computer must win if at all possible.

3. P&P 3.41

4. P&P 3.43

5. Some Odds and Ends

   a. **Circuits:** You have a number of power outlets in your house connected to the same circuit. In the circuit box in your house is a fuse for that circuit. Your circuit supplies a constant 110 volts with varying current. Assume all plugs have a ground wire. What are the implications if all the plugs in a given circuit are wired in parallel? What if they are wired in serial? Which way are they wired? (Give a hypothetical example as your evidence.)

   b. **Binary algorithms:** You have a 16-bit unsigned integer ALU that can multiply two 8 bit numbers and produce a 16 bit product. Your ALU can also do arbitrary shifts on 16-bit numbers as well as (unsigned) adds. Write pseudo code (or an equation) that would allow your ALU to multiply two 8 bit numbers representing scaled, real numbers from [0,1], that is, $00 = 0.0 and $FF = 1.0. For example, you could use the simple formula:
   Sum16 = A8 * B8, Sum8 = (Sum16 >> 8)
   This would work for "0.8" fixed point numbers, but has the undesirably property that ($FF x $FF) = $FE, so $FF isn't really equal to 1.0 here, but rather, 255/256. Find a simple algorithm that scales correctly.

   c. **Logic:** You'd like to augment the unsigned 16-bit adder to deal with 16-bit signed numbers in two's complement notation. It would be a neat feature to generate a signal if you have signed overflow as a result of the add. Create a logic diagram to implement the function for the line OVFLR = ???. You will need to decide what it means to have an overflow when adding 2's complement numbers and then determine how to detect it.

   d. **Logic Proof:** Using ONLY NAND gates, implement a "naive" or straightforward implementation of an XOR gate, i.e., convert the logical definition of XOR to NANDs. Now go to wikipedia and look up "XOR gate". There is a clever implementation of XOR that uses only four NAND gates. Use boolean algebra to show that this really produces XOR. Remember all the logic identities you've learned in explaining your derivation.

   e. **Transmission Gates:** Your reading in Uyemura, chapter 2.5 talks about transmission gates, also known as "pass transistors". In conventional CMOS (complementary) design, we use logic signals to gate transistors, but the transistors simply pass through a strong 0 or strong 1. With transmission gates, we are turning things on their side - we are actually passing logic values through the transistor's source and drain. Would you consider this to be "restoring logic" (that is, logic that produces strong 1's and 0's as output)? Compare the transistor count of a 2:1 mux implemented using conventional gates vs transmission gates as in Figure 2.61.