LSU EE 4755 **Homework 2** **Due: 26 September 2014**

The Homework 2 code package contains four unsigned integer floating point modules and a testbench. The first two modules, `mult_behav_1` *and* `mult_behav_2` *already work, the other two,* `mult_linear` *and* `mult_tree`*, are mostly empty and are to be completed as part of this assignment. The first two multipliers are synthesizable, though they were not written to be synthesized. If this assignment is completed correctly the other two multipliers will be synthesizable too.*

Multiplier `mult_behav_1` *is a simple-as-possible implementation, the intent is to provide a correct result to use to check the other modules. Nevertheless it is synthesizable with Cadence RC, which will substitute an integer multiply library function from the ChipWare library.*

Multiplier `mult_behav_2` *computes the multiplication itself by adding partial sums. (See* `http://www.ece.lsu.edu/ee3755/2013f/l07.v.html` *for a quick review of integer multiplication. Don't go beyond the long-hand procedure for this assignment.)*

Warning: DO NOT attempt to find Verilog code for multipliers and use them for the solution. You will learn nothing by doing so and will be unprepared for the midterm exam.

Problem 0: Copy the code package from `/home/faculty/koppel/pub/ee4755/hw/2014f/hw02`. Verify that everything is working by running the simulation on the unmodified file. It should report a 0% error rate for `mult_behav_2` and a 100% error rate for the linear and tree multipliers.

Problem 1: Synthesize `mult_behav_1` and `mult_behav_2` following the steps for synthesis on the course procedures page.

(*a*) Indicate the area and critical path delay for each module.

(*b*) Explain why one might be better than the other.

Problem 2: Complete `mult_linear` to that it performs a multiplication using `wid` instances of `good_adder` connected linearly. This module will be sort of a structural version of `mult_behav_2`. Use generate statements to instantiate the adders and make sure that the design is synthesiable.

Note that in this multiplier instance i of the adder cannot start until $i-1$ finishes (that's an oversimplification, but it's true enough).

Problem 3: Complete `mult_tree` so that the adders are connected in a tree-like fashion. Let a and b be the two w-bit operands of the multiplier. There should be $w/2$ adders near the leaves which add two partial products. (There are w partial products, partial product $i \in [0, w-1]$ is $a2^i$ if b_i is 1, or 0 if b_i is 0, where b_i is the digit at bit position i.) At the next level there will be $w/4$ adders which each add the sum of two adders from the lower level, and so on.

First try to solve this using $2w$-bit adders. If you are feeling clever optimize your solution by using $(w+2)$-bit adders for the first row, $(w+4)$-bit adders for the second row, etc.

As before, the design must be synthesiable.

Problem 4: Perform synthesis on your two modules.

(*a*) Indicate the area and delay of each module.

(*b*) Indicate which module you *expected* to be fastest and explain why. If that's different than the one that really is fastest, give a possible reason.