LSU LSU EE 4755 Homework 3 Due: 24 October 2014

Updated 18 October 2014, 18:00:29 CDT

The Homework 3 code package contains a simple behavioral multiplier and several sequential multipliers. It also contains a synthesis script in file `syn.cmd`*.*

Problem 0: Copy the code package from `/home/faculty/koppel/pub/ee4755/hw/2014f/hw03`. Verify that everything is working by running the simulation on the unmodified file. It should report a 0% error rate for all modules.

Problem 1: The module `mult_seq_csa` is a sequential multiplier that instantiates an adder, however unlike `mult_seq_ga` shown in class, `mult_seq_csa` instantiates a carry-save adder from the Chipware library, `CW_csa`. The carry save adder computes the sum of three integers, `a`, `b`, and `c` (those are the port names). It produces two sums, which we'll call `sum_a` and `sum_b` (the port names for these are `carry` and `sum`). All of these ports are w bits wide, where w is a parameter. The actual sum of `a`, `b`, and `c` is obtained by adding together outputs `sum_a` and `sum_b` using a conventional adder. Carry save adders are used when there many integers to be added. Some arrangement (linear, tree) of many carry-save adders will produce a `sum_a` and `sum_b`, which will be added by a single conventional (called carry-propagate) adder.

The advantage of a carry save adder is that it can compute a sum of w-bit numbers in $O(1)$ time (the amount of time is not affected by w), which of course is much better than the $O(w)$ time for a ripple adder or the $O(\log w)$ time for much more expensive carry look-ahead adders. The performance advantage of a CSA is lost for `mult_seq_csa` because the module only computes one partial product at a time.

(*a*) Sketch the hardware that will be synthesized for `mult_seq_csa`. Show the carry-save adder and other major units as boxes, but be sure to show registers, multiplexors, and other such components. **Do not** show the actual output produced by an actual synthesis program. (It's okay if you look at a synthesis program's output.)

(*b*) Based on this sketch of synthesized hardware, explain why the benefit of using a CSA is lost. Also explain how the module can be made a little faster (with a small change), but is still not a good way to use a CSA.

Problem 2: Module `mult_seq_csa_m` initially contains the m-partial-products-per-cycle module that we did in class. In this problem modify it to use CSA's, and avoid the issue identified in the previous problem.

(*a*) Modify `mult_seq_csa_m` so that it uses the carry-save adder to compute m partial products per cycle. Use `generate` statements to instantiate the CSA's, and of course, connect them appropriately. (In class we used generate statements for the pipelined adder to instantiate stages, that code is in `mult_pipe_ia` in the same file as the assignment.)

(*b*) Sketch the hardware that you expect to be synthesized for an $m = 2$ version. Make sure that your design does not do something foolish with the conventional adder.

Problem 3: Run the synthesis program to compare the cost and performance of `mult_seq_csa_m` to `mult_seq_m`. The synthesis script `syn.cmd` can be used to synthesize these modules at different sizes. To run it use the command `rc -files syn.cmd`. Feel free to modify the script. (It is written in TCL, it should be easy to find information on this language.)

(*a*) Show the cost and performance versus m for these modules.

(*b*) If you solved the previous problem correctly the total delay shown for `mult_seq_csa_m` should be wrong. Explain why, and (optional) if you like try modifying syn.cmd to fix it.

(*c*) Explain how you might expect the delay of `mult_seq_csa_m` to change with increasing m? Explain your expectation and whether the synthesis results bear that out.