LSU EE 4755 **Homework 3** Solution **Due: 24 October 2014**

Updated 7 November 2014, 13:49:47 CST

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

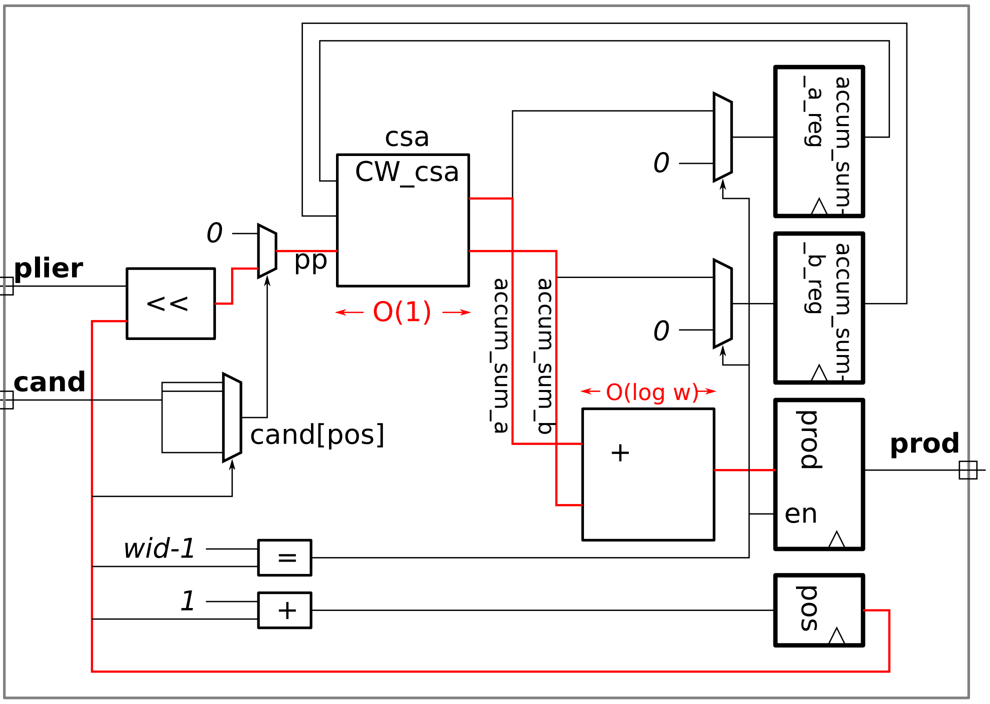
Problem 0: Copy the code package from `/home/faculty/koppel/pub/ee4755/hw/2014f/hw03`. Verify that everything is working by running the simulation on the unmodified file. It should report a 0% error rate for all modules.

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.)

The hardware appears below. In the diagram the critical path is shown in red. Notice that the critical path goes through *both* the CSA and conventional adders.



(*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.

The clock frequency is based on the (longest) critical path. For the module the critical path is the *sum* of the delay through the CSA and carry-propagate (regular) adder. If a carry-propagate adder were used

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.)

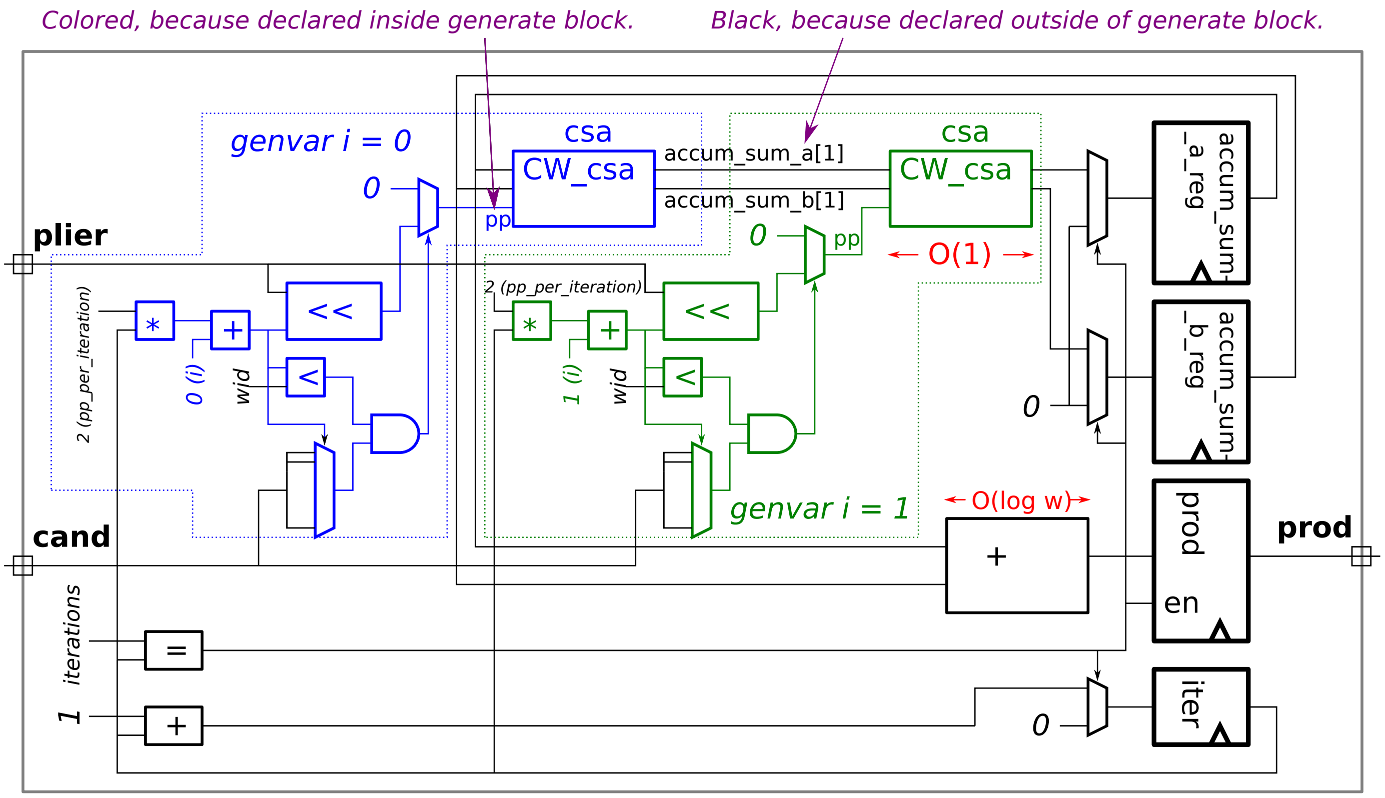
Solution appears below.

```
module mult_seq_csa_m #( int wid = 16, int pp_per_cycle = 2 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier,   input logic [wid-1:0] cand,   input clk);

   localparam int iterations = ( wid + pp_per_cycle - 1 ) / pp_per_cycle;
   localparam int iter_lg = $clog2(iterations);
   localparam int wid_lg = $clog2(wid);

   logic [iter_lg:0] iter;
   wire [2*wid-1:0] accum_sum_a[0:pp_per_cycle], accum_sum_b[0:pp_per_cycle];
   logic [2*wid-1:0] accum_sum_a_reg, accum_sum_b_reg;

   initial iter = 0;
   assign           accum_sum_a[0] = accum_sum_a_reg;
   assign           accum_sum_b[0] = accum_sum_b_reg;

   for ( genvar i=0; i<pp_per_cycle; i++ ) begin

      wire [wid_lg:1] pos = iter * pp_per_cycle + i;
      wire            co; // Unconnected.

      // The "pos < wid" below is needed in case wid is not an integer multiple of pp_per_cycle.
      wire [2*wid-1:0] pp = pos < wid && cand[pos] ? plier << pos : 0;

      CW_csa #(2*wid) csa
        ( .sum(accum_sum_a[i+1]), .carry(accum_sum_b[i+1]), .co(co),
          .a(accum_sum_a[i]), .b(accum_sum_b[i]), .c(pp), .ci(1'b0) );
   end

   always @( posedge clk ) if ( iter == iterations ) begin

         prod <= accum_sum_a_reg + accum_sum_b_reg;
         accum_sum_a_reg <= 0;
         accum_sum_b_reg <= 0;
         iter <= 0;
      end else begin

         accum_sum_a_reg <= accum_sum_a[pp_per_cycle];
         accum_sum_b_reg <= accum_sum_b[pp_per_cycle];
         iter <= iter + 1;
      end
endmodule
```

(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.

The hardware appears below. Coloring has been used to emphasize the hardware corresponding to each iteration of the generate loop (blue and green) and hardware corresponding to Verilog outside of the generate loop (black). Pay close attention to `accum_sum_a[i]` and `accum_sum_b[i]`. They are declared outside the generate loop but are used to interconnect items in different generate loop iterations.

The diagram shows the inferred hardware, before any optimization. Note that the conventional adder (the big box with the plus) receives its inputs from the outputs of register `accum_sum_a_reg` and `accum_sum_b_reg`, rather than the CSA outputs. This gives the adder the entire clock period to produce its sum.



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.

The cost and performance appear below. The first table shows the results using the unmodified synthesis script, in which area was minimized. The second table shows the results using a synthesis script in which the synthesis program was set to minimize delay.

```
--------------- Area Optimization -------------------------------------
Module Name                              Area   Clock     Total    Init.
                                               Period     Delay    Interv
mult_seq_csa_m_wid16_pp_per_cycle1     110308   14170    226720   226720
mult_seq_csa_m_wid16_pp_per_cycle2     135192   13692    109536   109536
mult_seq_csa_m_wid16_pp_per_cycle4     157668   12828     51312    51312
mult_seq_csa_m_wid16_pp_per_cycle8     195212   11110     22220    22220
mult_seq_m_wid16_pp_per_cycle1          74092   16444    263104   263104
mult_seq_m_wid16_pp_per_cycle2          99884   17470    139760   139760
mult_seq_m_wid16_pp_per_cycle4         112664   16508     66032    66032
mult_seq_m_wid16_pp_per_cycle8         154744   16463     32926    32926


-------------- Delay Optimization ----------------------------------
Module Name                              Area   Clock     Total    Init.
                                               Period     Delay    Interv
mult_seq_csa_m_wid16_pp_per_cycle1     164940    2054     32864    32864
mult_seq_csa_m_wid16_pp_per_cycle2     195408    2255     18040    18040
mult_seq_csa_m_wid16_pp_per_cycle4     239340    2756     11024    11024
mult_seq_csa_m_wid16_pp_per_cycle8     316748    4043      8086     8086
mult_seq_m_wid16_pp_per_cycle1         125408    3062     48992    48992
mult_seq_m_wid16_pp_per_cycle2         166488    3368     26944    26944
mult_seq_m_wid16_pp_per_cycle4         202096    3777     15108    15108
mult_seq_m_wid16_pp_per_cycle8         263772    4285      8570     8570
```

(*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.

The TCL script computes the total delay by multiplying the clock period by w/m. (In the TCL script w/m is computed by the routine `get_stages`. In that routine variable `bits` is used for w and `deg` for m.) The values of m and w used by the script are chosen so that m always divides w, so the problem has nothing to do with integer truncation errors.

The module designed for the solution to Problem 2 uses an extra cycle to compute the sum, so it takes $m/w + 1$ cycles, and the TCL script does not take this into account. (Of course, that would be easy enough to fix.)

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

The clock period is determined by either the delay of one carry-propagate (conventional) adder or the delay of m carry-save adders, whichever is larger. For small values of m the carry-propagate adder would have the larger delay. So, one might expect that the clock period for the modules with $m = 1$ and $m = 2$ would be the same. However, the time needed to compute a product, $T(m)$, would go from $T(1) = (w/1 + 1)\,t_{\mathrm{clk}} \approx wt_{\mathrm{clk}}$ to $T(2) = (w/2 + 1)\,t_{\mathrm{clk}} \approx \frac{w}{2}t_{\mathrm{clk}}$ which is nearly half the time. For these small values of m the clock period $t_{\mathrm{clk}} = t_{\mathrm{latch}} + t_{\mathrm{adder}}$, where t_{latch} is the setup time needed for the registers and t_{adder} is the time needed for the carry-propagate adder. When m is increased further the clock period time will be more like $t_{\mathrm{clk}} = t_{\mathrm{latch}} + mt_{\mathrm{csa}}$ where t_{csa} is the delay for one carry-save adder. At that point, further increases in m will not improve total performance by as much:

$$\begin{aligned} T(m) &= (w/m + 1)\,(t_{\mathrm{latch}} + mt_{\mathrm{csa}}) \\ &= (w+1)t_{\mathrm{csa}} + \left(\frac{w}{m} + 1\right) t_{\mathrm{latch}} \end{aligned}$$

When the synthesis program is optimizing delay, results are consistent with this analysis: Performance improvement with increasing m is much better when m is small than when m is large.