Name Solution

# Digital Design using HDLs
# EE 4755
# Midterm Examination

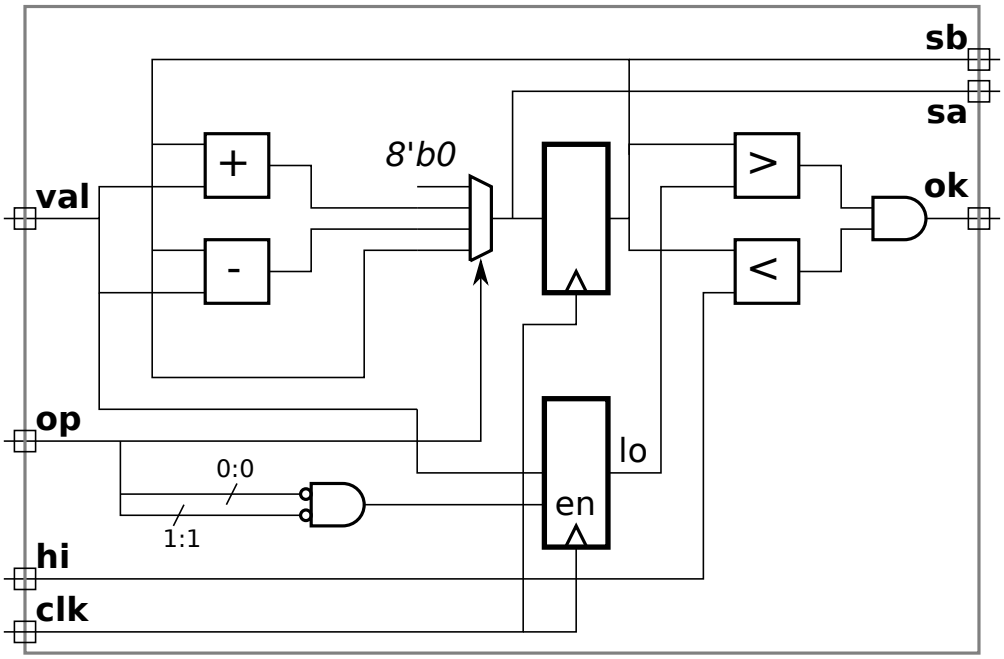
Monday, 10 November 2014 11:30–12:20 CST

| | | |
|---|---|---|
| Problem 1 | | (20 pts) |
| Problem 2 | | (20 pts) |
| Problem 3 | | (10 pts) |
| Problem 4 | | (15 pts) |
| Problem 5 | | (13 pts) |
| Problem 6 | | (22 pts) |
| Exam Total | | (100 pts) |

Alias Over-reactive region

*Good Luck!*

Problem 1: [20 pts] Write a Verilog description of the hardware shown below.




☑ Write a Verilog module corresponding to the hardware above.

☑ Be sure to declare module ports and ☑ any wires and vars (`logic`) used inside.

☑ Pay attention to the differences between `lo` and `hi` and ☑ the differences between `sa` and `sb`.

```
// SOLUTION
module prob1(output logic [7:0] sa, sb,   output uwire ok,
             input uwire [1:0] op,        input uwire [7:0] val, hi,   input uwire clk );

   always_comb
     case ( op )
       0: sa = 0;
       1: sa = sb + val;
       2: sa = sb - val;
       3: sa = sb;
     endcase

   always_ff @( posedge clk ) sb <= sa;

   logic [7:0] lo;
   always_ff @( posedge clk ) if ( op == 0 ) lo <= val;

   assign ok = sb > lo && sb < hi;

endmodule
```

The Verilog appears above.

Discussion of `sa` /`sb` differences.

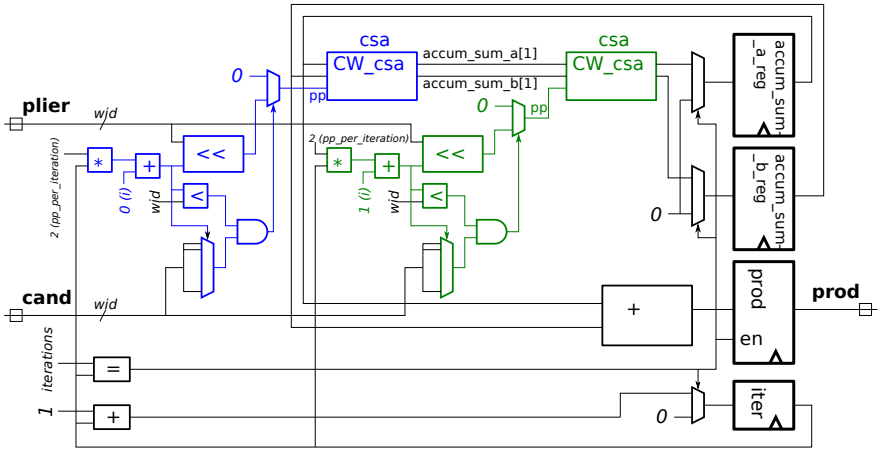
In the diagram notice that `sa` is produced in part by signals connected to the module inputs. That means if, for example, input `op` changes then `sa` must change as soon as it can. For that reason it is **not** assigned in an always block controlled by `posedge clk`, instead it is assigned in an `always` block sensitive to all live-in objects, namely `op`, `val`, and `sb`. In contrast, output `sb` is connected to the output of an edge-triggered register which means it can *only* change on the positive edge of `clk`. For that reason it is assigned in an `always` block sensitive to `posedge clk`.

The `lo` register is written on the positive edge of the clock when bit 0 (notice the `0:0` label next to the tic mark) of `op` is zero and when bit 1 of `op` is zero. In Verilog that's cleanly shown as `if ( op == 0 )`. It would be correct though cumbersome to replace the `if` condition with `op[0] == 0 && op[1] == 1`. An even more cumbersome solution would instantiate an AND gate and two NOT gates.

Problem 2: [20 pts] Appearing below is the multiply circuit from the solution to Homework 3, in Verilog (slightly simplified) and as a diagram showing what hardware a synthesis program might infer.

```
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 uwire 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;
   uwire [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;

   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

      uwire [wid_lg:1] pos = iter * pp_per_cycle + i;
      uwire [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]), .a(accum_sum_a[i]), .b(accum_sum_b[i]), .c(pp));
   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
         prod <= prod;
         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
```

USE NEXT PAGE FOR SOLUTION




USE NEXT PAGE FOR SOLUTION

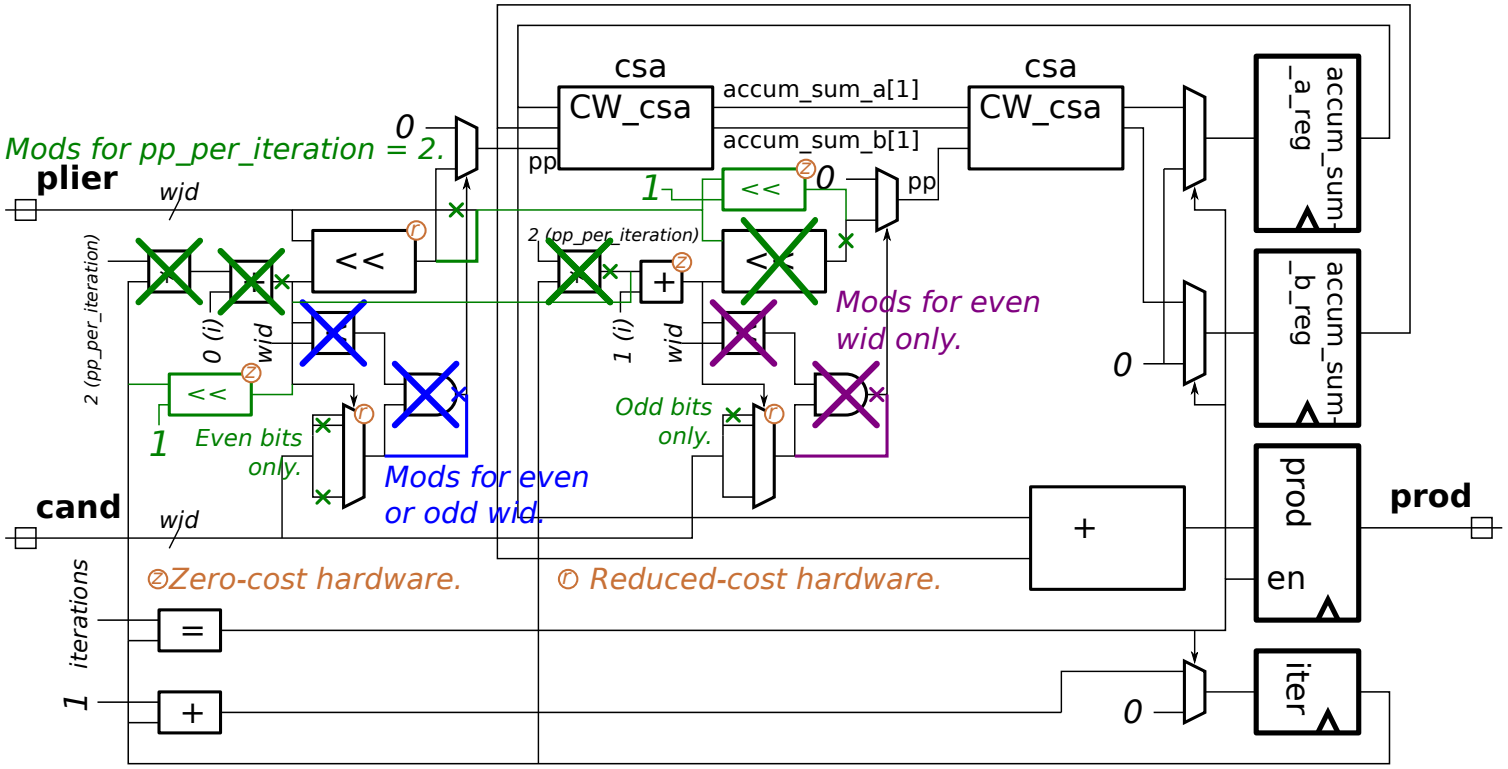
(*a*) Show optimizations that might be performed that exploit the value $m = 2$ (that is, `pp_per_iteration=2`).

(*b*) Show the optimizations that might be performed assuming that `wid` is odd, and assuming that `wid` is even, both for $m = 2$.

☑ Modify diagram to show optimizations for `pp_per_iteration` = $m = 2$ and arbitrary `wid`.

☑ Modify diagram to show optimizations for `pp_per_iteration` = $m = 2$ and odd `wid`.

☑ Modify diagram to show optimizations for `pp_per_iteration` = $m = 2$ and even `wid`.




Solution appears above. Three sets of changes are shown. The changes in green are optimizations possible with `pp_per_iteration=2`, the changes in blue are possible with any value of `wid` (odd or even), and the changes in purple are possible only when `wid` is even. Hardware labeled with a circled z is zero-cost, meaning that the outputs are either constants or are connected directly to the inputs. (In class this was called *renaming bits*.) The hardware labeled with a circled r is a lower cost version of the hardware depicted. In particular, the shift so labeled is lower cost because it only needs to shift by an even number of positions. The r-labeled multiplexors are lower cost because half of their inputs are unused (and so will not be synthesized).

If we know that `pp_per_iteration` is 2 then the shift amounts for `plier` just shifting `iter` by one bit (for `i=0`) or shifting and placing a `1` in the LSB position (for `i=1`). These observations are used to eliminate the multipliers and adders. (One of the adders is shown as zero-cost.) Further, in the `i=0` section we know that only even-numbered bit positions are from `cand`, reducing the cost of the multiplexor; a similar optimization is made for the `i=1` section.

The $<$ modules are used to determine if the `cand` bit position is valid. For the `i=1` section the `cand` bit position in the last iteration will be invalid if `wid` is odd. (For example, suppose `wid=5` and consider the third iteration, when `iter` is 2. The bit position sought by the `i=0` section will be $2 \times 2 = 4$, the MSB of `cand`. The `i=1` section will look for bit $2 \times 2 + 1 = 5$ which is invalid, though with a typical adder the mulitplexor might be commanded to look at bit 0, which is wrong. The less-than module and AND gate prevent the bit from being used.)

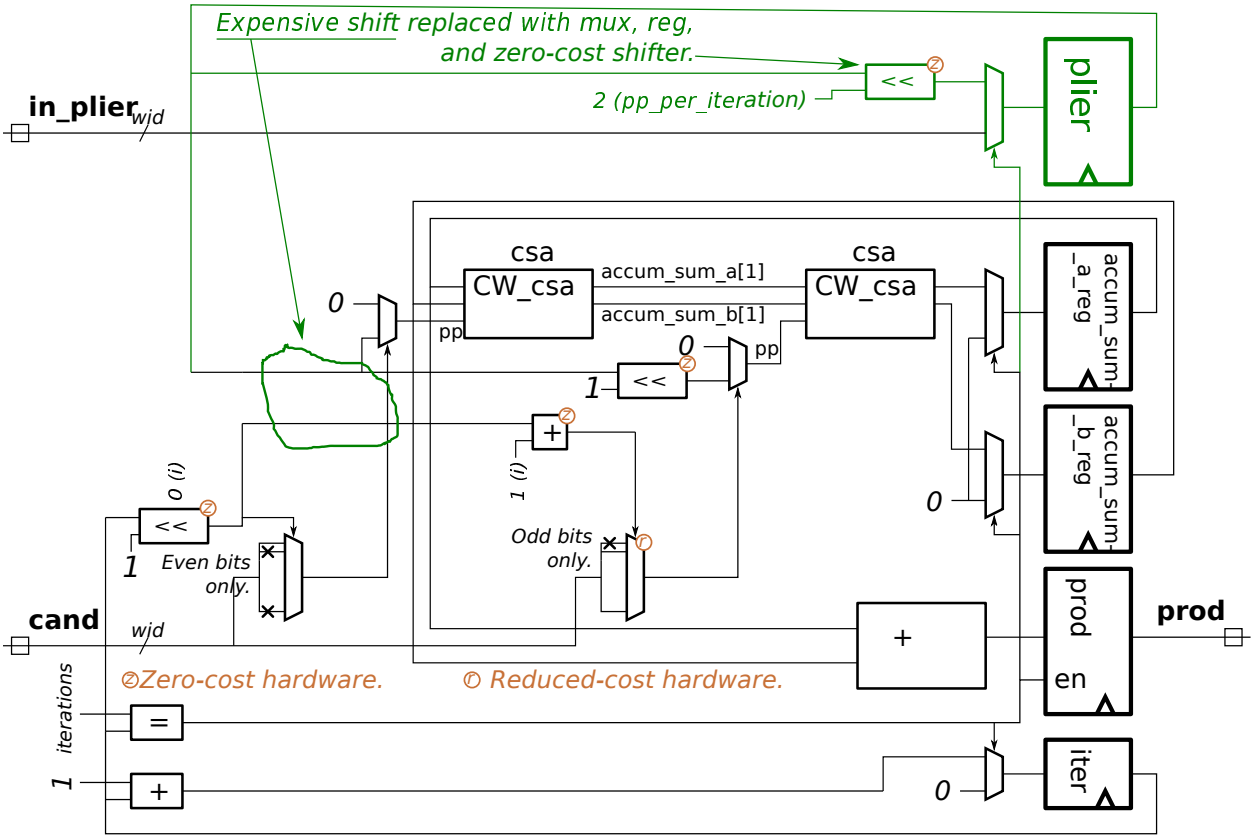
In contrast, there will never be an invalid bit position when `wid` is even. So, when `wid` is even both the optimizations shown in blue and purple can be made. If `wid` is odd then the blue optimizations can be made but the purple optimizations **cannot** be made.

Problem 2, continued:

(*c*) The cost of the shifters with input `plier` in the design on the previous pages is significant. Explain how these shifters can be eliminated by adding a register. Quickly sketch the hardware to illustrate your answer.

☑ Show how a register can be used to eliminate the costly shifters.

Solution appears below in green, based on the optimized even-`wid` multiplier. The shift-by-any-amount (or at least any even amount) shifter (which would occupy the hand-drawn circle in the diagram) is replaced by a shifter that shifts by exactly `pp_per_iteration` positions, which is a zero-cost device. The output of this shifter is stored in a register and used in the next iteration. The shift amounts that are needed in a particular iteration can be obtained using only these zero-cost shifters. The multiplexor and register that we've added is not free, but their should be less than the shifters when about three or more iterations are needed.




(*d*) Explain how the streamlined multiplier described in class eliminated the `plier` shifter *without* having to add a register.

☑ Show how the streamlined multiplier does not need an extra register to eliminate the shifter.

The streamlined multiplier shifts the accumulated product rather than the multiplier. (This may not be possible using CSAs.)

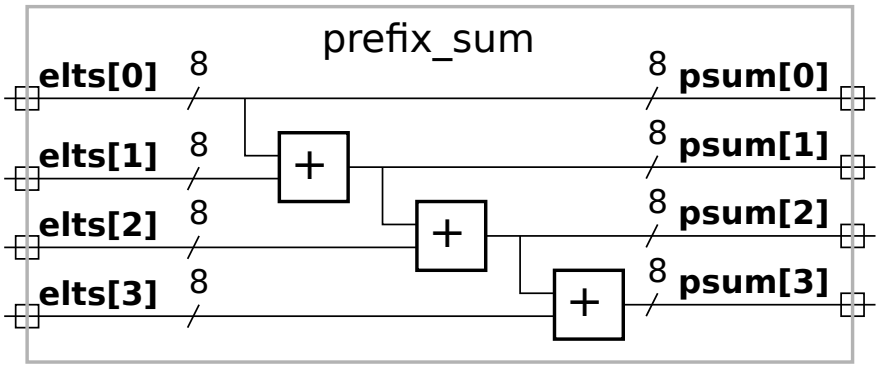
Problem 3: [10 pts] The module below computes the prefix sum of a sequence of integers at its input.

```
module prefix_sum #( int len=8, int wid = 8)
  (output logic [wid:1] psum [len], input uwire [wid:1] elts[len]);

   always @* begin
      psum[0] = elts[0];
      for ( int i=1; i<len; i++ ) psum[i] = psum[i-1] + elts[i];
   end
endmodule
```

(*a*) Show the hardware that would be synthesized for the module before optimization, elaborated with parameters `len=4` and `wid=8`. Label the input ports `elts[0]`, `elts[1]`, `elts[2]`, and `elts[3]`; and label the output ports `psum[0]`, `psum[1]`, `psum[2]`, and `psum[3]`.

☑ Show synthesized hardware.

Synthesized hardware appears below.




(*b*) Estimate the delay for the synthesized hardware before optimization. Use $w$ for the value of `wid` and $L$ for `len`. Assume that a $w$-bit adder has delay $w$.

☑ Delay in terms of $w$ and $L$:

The delay is $(L-1)w$.

Problem 4: [15 pts] Answer the following questions about the Verilog module below.

```
module timing();
   logic [7:0] a, e, f2, g, g1, g2;   logic clk;      uwire [7:0] e1, f, f1;
   initial begin
      clk = 0;
      a = 11;
      #1;
      a = 1;
      a <= 22;
      a <= #5 a + 1;

      #9;
      a = 7;
      e = 10;
      f2 = 30;
      g = 40;
      g1 = 50;
      g2 = 60;

      #10;
                                          // B0
      a <= 700;
      clk = 1;

      #1;
      // POINT X (See subproblem.)
   end

   always @( posedge clk ) e = a;           // B1
   always @*               e1 = a;          // B2
   always @*               f = e + 1;       // B3
   always @*               f1 = e1 + 1;     // B4
   always @( posedge clk ) f2 <= e + 1;     // B5
   always @( posedge clk ) begin            // B6
                            g = f;   g1 = f1;   g2 = f2;   end
endmodule
```

(*a*) Show values for `a` versus time in the table below. For this part, **only** `a`. The table already shows that `a` has value 11 from time 0 to time 1. Extend the table as long as necessary, and be sure to show values for both $t$ and `a`. *Note: The original exam did not provide the table. Also, in the original exam there were differences in how* `a` *was assigned.*

☑ Complete the table.

| $t$ | 0 | 1 | 6 | 10 | 20 |
|---|---|---|---|---|---|
| $a$ | 11 | 22 | 2 | 7 | 700 |

Solution appears above. After $t = 1$ `a` gets the value 22 because of the non-blocking assignment. However, the delayed assignment (`a <= #5 a + 1;`) uses the value of `1` for `a` since the non-blocking assignment of `22` at that point had not taken effect. (It must wait until the scheduler gets to the NBA region of the event queue.)

Notice that delays (such as `#10` are relative to the current time, **not to** $t = 0$).

(*b*) Show the values that will be present on `g`, `g1`, `g2` when execution reaches the `POINT X` comment in the module above. For partial credit also show intermediate values for other signals used to compute the `g`'s. (Look at next part before solving this one.)

☑ At `POINT X` `g`=__11__, `g1`=__8__, `g2`=__8__.

Solution appears above. The `g`'s are assigned on the positive edge of the clock in block `B6`. To solve the problem one needs to figure out what has already executed. Before point `B0` in the code `a` is 7, `e` is 10, and `f2` is 30, set by the procedural code, and the combinational (`@*`) always blocks would have set `f1` to 8 and `f` to 11. After the procedural code assigns `clk=1` and reaches the `#1` the scheduler will schedule the `posedge clk` blocks in arbitrary order. After `B0` finishes the three newly scheduled blocks `B1`, `B5`, and `B6`, are placed in the active region of the event queue. Block `B1` changes `e` to 7, which causes `B3` to be scheduled in the inactive region. The scheduler continues with the active region, next executing `B5`, which schedules an update event in the NBA region that will set `f2` to 8. Next `B6` is executing, assigning the `g`'s. Variable `g` is assigned 11, notice that `B3` is still in the active region and so has not gotten its chance to modify `f`. Variable `g1` is assigned 8. Variable `g2` is assigned 30. Notice that `B5` adds 1 to `e` before `B6` executes but the update is done afterwards. Block `B3` executes after `B6`. Therefore the "old" values are used for `g` and `g2`.

(*c*) Recall that the event queue used for Verilog simulation has *active*, *inactive*, and *NBA* regions, among others. Just before `B1` starts execution in module `timing` above the active region might contain `B1`, `B5`, and `B6` (see the comments on the right). (What the other regions contain is part of this problem.) Show the contents of the three regions when `B5` starts. Assume that events in a region are scheduled in order.

☑ When `B5` starts: Active = {__B6__}. Inactive = {__B3__}. NBA = {__a <= 700__}.

Solution appears above. When `B5` starts only `B6` remains in the active region (see the solution to the previous problem). Block `B3` has been scheduled in the inactive region due to the assignment of `e` by `B1`. The update to `a` was scheduled in the NBA (non-blocking assignment) region by `B0` in the `initial` block.

Problem 5: Answer each question below.

(*a*) [5 pts] Module `add3` is supposed to compute the sum of its three inputs using instances of `our_adder`, but it won't work. Fix the problem. The fixed module should still use `our_adder`.

☑ Fix `add3`.

```
module add3(output uwire [15:0] sum, input uwire [15:0] a,b,c);



   our_adder a1(  sum  ,  a  ,  b  );

   our_adder a2(  sum  ,  sum  ,  c  );

endmodule
```

```
   /// SOLUTION

module add3(output uwire [15:0] sum, input uwire [15:0] a,b,c);
   uwire [15:0] sum1;

   our_adder a1( sum1, a,    b );
   our_adder a2( sum,  sum1, c );

endmodule
```

Solution appears above. The problem was that the same object, `sum`, was connected to the output of both adders. Its value therefore is undefined. In the solution a new wire, `sum1`, is declared and used as the output of the first adder.

(*b*) [8 pts] The output of the module below is like the input except the bit positions are reversed (after enough clock cycles). Re-write the module so that it synthesizes to combinational logic (the `clk` input will no longer be needed). Add a parameter to indicate the input and output bit width.

```
module bitrev(output logic [7:0] x, input uwire [7:0] a, input uwire clk);
   logic [2:0] pos;
   initial pos = 0;

   always @( posedge clk ) begin
      x[pos] = a[7-pos];
      pos++;
   end
endmodule
```

☑ Re-write so that it is combinational.

☑ Include a parameter `wid` to specify the size.

```
// SOLUTION
module bitrev_s #(int wid = 8) (output logic [wid-1:0] x, input [wid-1:0] a);

   always @* for ( int i=0; i<wid; i++ ) x[ i ] = a[ wid-i-1 ];

endmodule
```

Solution appears above. Since the logic is combinational there is no need for a clock input. Notice that this will synthesize into a module that contains no logic. All it does is rename signals. This would not be a problem if it were part of a larger design, but if this module were the only thing fabricated on a chip money could have been better spent.

Problem 6: Answer each question below.

(*a*) [5 pts] A Verilog module computes a result in one clock cycle. In our design we need that result in 3 ns, which can easily be achieved. The right way to achieve that in Cadence Encounter is to use the `define_clock` command to set the target clock period to 3 ns. Suppose instead we used `define_clock` to set the period to 1 ps, an impossible goal. *Note: The original exam did not have the "can easily be achieved" phrase.*

☑ Would the synthesized design meet our 3 ns performance goal?

Yes. Even though 1 ps is impossible, the synthesis program will synthesize a circuit with as short a delay as it's capable of, and according to the problem it can easily create a circuit with a delay less than 3 ns. *Note: For those taking the original exam the answer would be: Yes, if the synthesis program is capable of reaching the* 3 ns *goal.*

☑ Considering typical design goals, what would be the disadvantage of setting the period to 1 ps for our design even though we needed 3 ns?

Short answer: The disadvantage is that the cost of the synthesized circuit might be higher than would be obtained when setting the clock period to our performance target, 3 ns.

Suppose the synthesis program generates a circuit with a delay of 2.1 ns. That meets our performance goal, but so would a 3 ns circuit. However the 2.1 ns circuit might have a higher cost than the 3 ns circuit since the optimization program tries to minimize cost while meeting design constraints. Since cost minimization is a typical design goal, setting the clock period to 1 ps would result in a worse design.

(*b*) [10 pts] In the module below, `translate` directives are used to prevent the synthesis program from reading the line with `initial`.

```
module mult_seq( output logic [311:0] prod, input logic [15:0] plier, cand, input uwire clk);

   logic [3:0] pos;    logic [31:0] accum;

   // cadence translate_off          <-- The translate synthesizer directive.
   initial pos = 0;
   // cadence translate_on           <-- The translate synthesizer directive.

   always @( posedge clk ) begin
      if ( pos == 0 ) begin  prod = accum;  accum = 0;  end
      if ( cand[pos] == 1 ) accum += plier << pos;
      pos++;
   end
endmodule
```

☑ Why shouldn't the synthesis program see the line with `initial`?

☑ What would happen if the synthesis program saw the `initial` line?

Short answer: The synthesis program should not see the `initial` line because it has no way to synthesize corresponding hardware, if it saw the line it would generate an error message.

The synthesis program should not see the `initial` line because it is unsynthesizable, and so would result in an error message. It is unsynthesizable because the developers of the synthesis program (this semester Cadence Encounter RTL Compiler) and the developers of probably every other HDL synthesis program do not think it's worth the trouble to generate special "initial" hardware that only does something when, say, the power is turned on. The correct way of achieving that kind of behavior is by providing a reset input to the module.

☑ What would happen if the simulation program *didn't* see the line with `initial`?

The value of `pos` would remain at `x` (undefined).

(*c*) [7 pts] All four variables below have a size of 32 bits, but there are differences between them.

```
logic [31:0] a;
logic b [31:0];
logic [0:31] c;
int e;
```

All four variables above hold 32 bits. (Unlike C, SystemVerilog sets the size of `int` to be 32 bits.)

Variable `a` is called a packed vector. It is interpreted as a single 32-bit quantity, and so can conveniently be used in expressions such as `a+x`.

☑ Difference between `a` and `b`?

Variable `b` is interpreted as a 32-element array of 1-bit elements.

☑ Difference between `a` and `c`?

Both `a` and `c` are packed vectors and are interpreted as 32-bit quantities. However the bit numbering of the two are different. That makes a difference in expressions that refer to bit positions, such as `y = a[10];` , but it does not make a difference in expressions that don't refer to bit positions, such as `y = a + x;`.

☑ Difference between `a` and `e`?

Each bit in a `logic` object can have four states, `0`, `1`, `x`, and `z`. Type `int` is a 32-bit quantity in which each bit is either `0` or `1`. The `logic` type is intended for objects that will synthesize into hardware, while `int` is intended for other uses such as in testbenches.