Name Solution

Digital Design using HDLs

LSU EE 4755

Solve-Home Final Examination

Wednesday, 9 December 2020 to Friday, 11 December 2020 16:30 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(10 pts)
Problem 5		(35 pts)
Exam Total		(100 pts)

Alias mRNA!

Good Luck!

Problem 1: [20 pts] Module `prob1_seq`, below, is based on the solution to 2016 Final Exam Problem 1 (also appearing in problem set `https://www.ece.lsu.edu/koppel/v/guides/pset-syn-seq-main.pdf`, please look at that solution). In that problem an incomplete diagram of the hardware was given, similar to the one on the next page, and a module was to be completed so that it computes `v0*v0 + v0*v1 + v1*v1` consistent with the hardware. The completed module appears below, with minor simplifications. *If you must know, the simplifications include omitting the floating-point modules' round inputs and status outputs. Also, the* `case` *statement was replaced by an* `if/else` *statement. In case anyone is concerned, this wordy aside would be omitted from an in-class exam.*

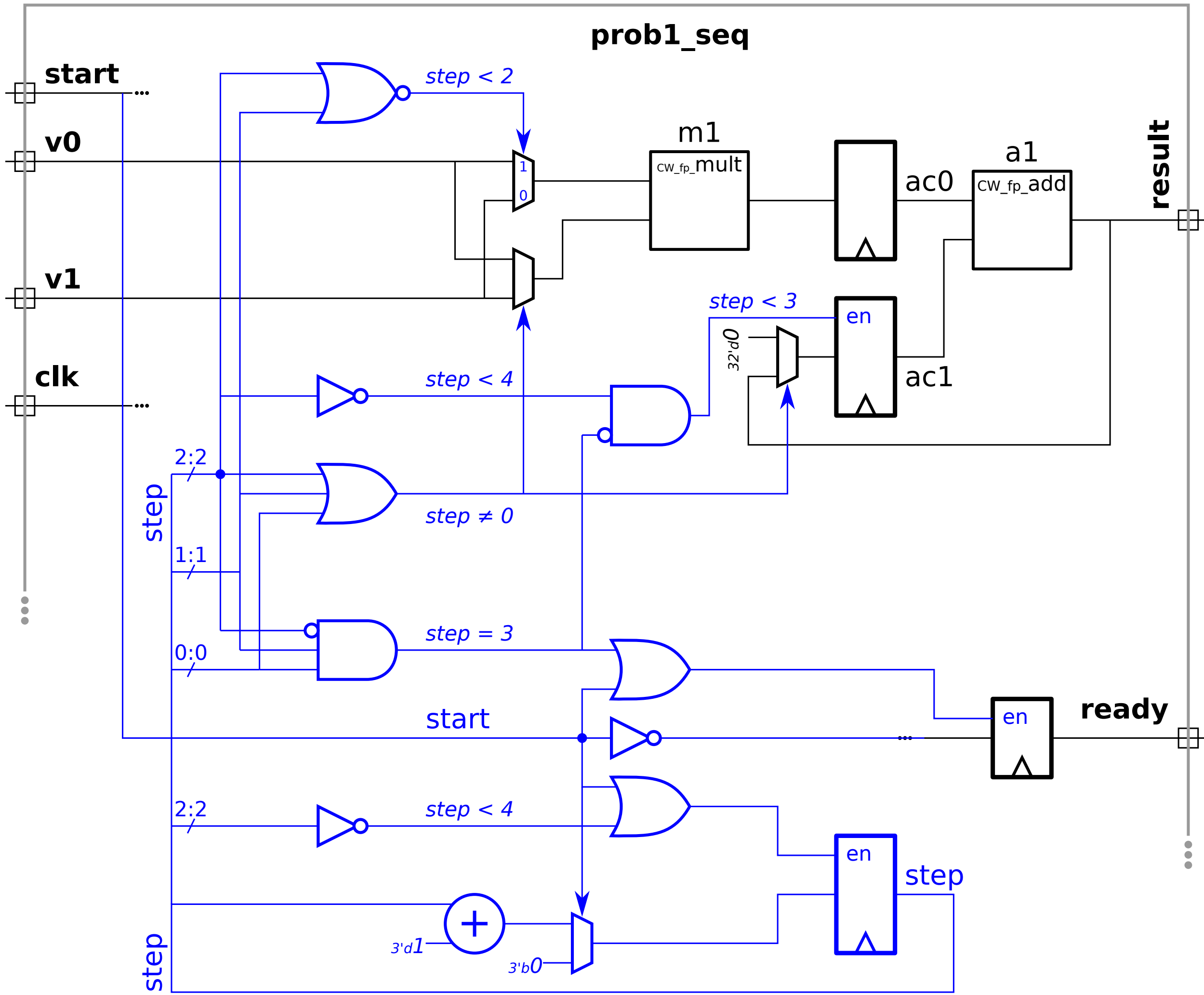
Though module `prob1_seq` is now complete, the hardware diagram isn't. In this problem complete the diagram of the synthesized hardware based on the module below. The diagram omits the hardware for `step`, select signals for the multiplexors, enable signals for some of the registers, etc. Optimize the hardware that compares `step` to a constant. Do so by showing individual gates rather than an equality or comparison unit.

☑ Complete the diagram so that it shows inferred hardware after some optimization.

☑ Where `step` is compared to a constant, show individual gates, not a comparison unit.

```
module prob1_seq
  ( output logic [31:0] result,  output logic ready,
    input uwire [31:0] v0, v1,   input uwire start, clk );

   uwire [31:0] mul_a, mul_b, add_a, add_b, prod, sum;

   logic [2:0]  step;
   logic [31:0] ac0, ac1;

   localparam int last_step = 4;

   always_ff @( posedge clk )
     if ( start ) step <= 0;
     else if ( step < last_step ) step <= step + 1;

   CW_fp_mult m1( .a(mul_a), .b(mul_b),  .z(prod) );
   CW_fp_add  a1( .a(add_a), .b(add_b),  .z(sum) );

   assign mul_a = step < 2  ? v0 : v1;
   assign mul_b = step == 0 ? v0 : v1;
   assign add_a = ac0,  add_b = ac1;

   always_ff @( posedge clk )
     begin
        ac0 <= prod;
        if ( step < 3 ) ac1 <= step ? sum : 0;
        if ( start ) ready <= 0; else if ( step == last_step-1 ) ready <= 1;
     end

   assign result = sum;

endmodule
```

Solution appears below in blue. A register was added to hold `step`. The value of `step` and `start` are used to determine multiplexor select signals and register enable inputs. The solution is labeled with some step comparison results, such as `step < 2`. Those who are unsure of how the illustrated logic computes these values are strongly urged to draw a truth table.



Problem 2: [20 pts] Consider again that module from Problem 1 of the 2016 final exam. Appearing below is the start of a Verilog description of a pipelined version of this module. The ports are the same as in the sequential version from the previous problem, however the module must operate in pipelined fashion, meaning that a new `v0`, `v1` pair could arrive at the inputs each cycle.

Complete the module. Two floating-point units are instantiated for your convenience. Add floating-point and other hardware as needed.

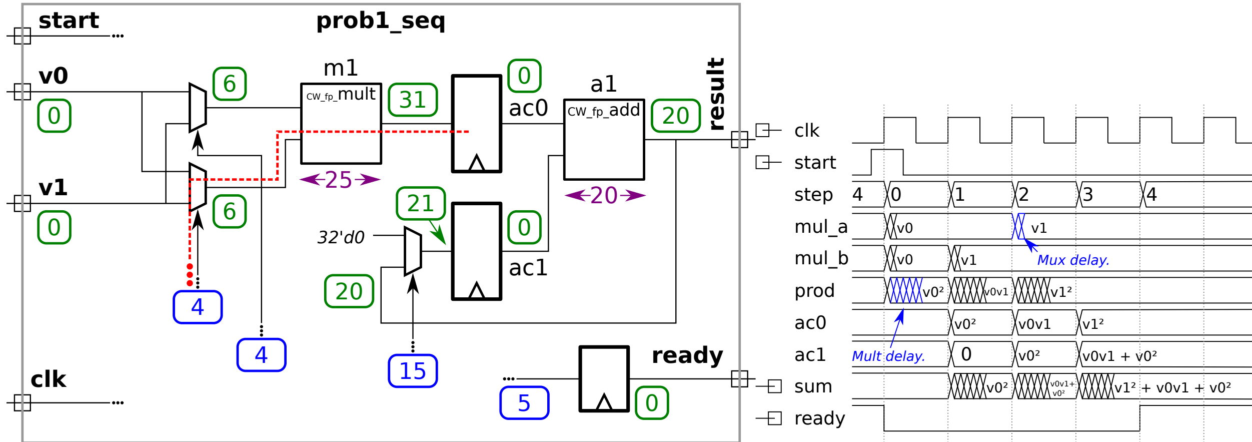
☑ Complete module so that it operates in pipelined fashion.

The solution that appears below is what would be expected on an exam. This problem was assigned as 2021 Homework 6 as a programming assignment. See that solution for additional details. In the solution below notice that the **`start`** signal is carried along the pipeline and finally connected to the **`ready`** output port.

The sequential hardware uses the value of register **`step`** so determine multiplexor and enable settings. That's not needed here because each stage does a particular step, in effect make step a constant. (Also, because there are more functional units fewer steps are needed.) For that reason there is no equivalent to the step register in the pipelined solution.

```
module prob1_pipe( output logic [31:0] result,   output logic ready,
                   input uwire [31:0] v0, v1,    input uwire start, clk);

   /// SOLUTION

   uwire [31:0] v00, v01, v11, s1, s2;
   logic [31:0] pl_1_v00, pl_1_v01, pl_1_v11;
   logic [31:0] pl_2_v0001, pl_2_v11;
   logic pl_1_occ, pl_2_occ;

   CW_fp_mult m00( .a(v0), .b(v0), .z(v00) );
   CW_fp_mult m01( .a(v0), .b(v1), .z(v01) );
   CW_fp_mult m11( .a(v1), .b(v1), .z(v11) );

   CW_fp_add  a1(.a(pl_1_v00),   .b(pl_1_v01), .z(s1) );
   CW_fp_add  a2(.a(pl_2_v0001), .b(pl_2_v11), .z(s2) );

   always_ff @( posedge clk ) begin

      pl_1_v00 <= v00;
      pl_1_v01 <= v01;
      pl_1_v11 <= v11;
      pl_1_occ <= start;

      pl_2_v0001 <= s1;
      pl_2_v11 <= pl_1_v11;
      pl_2_occ <= pl_1_occ;

      result <= s2;
      ready <= pl_2_occ;

   end
endmodule
```

Problem 3: [15 pts] Yet again, consider the solution to 2016 Final Exam Problem 1. (The solution appears in the sequential problem set, `https://www.ece.lsu.edu/koppel/v/guides/pset-syn-seq-main.pdf`, feel free to look at it.) Appearing below is an incomplete diagram of the hardware with some timing information shown, and a timing diagram. In this problem several performance measures will be computed based on the simple model.



Let $t_m = 25\,\mathrm{u_t}$ denote the delay of the `CW_fp_mult` unit and let $t_a = 20\,\mathrm{u_t}$ denote the delay of the `CW_fp_add` unit. The arrival times of signals at the multiplexor select inputs and at the `ready` register are shown boxed in blue. Base the delay of the registers and multiplexors on the simple model.

(*a*) Determine the clock period for this module using the assumptions above and show the critical path on which this clock period is based.

☑ Determine the clock period. ☑ Show critical path used to determine the clock period.

☑ Show work, and state any assumptions.

The arrival times of stable values are shown in the diagram boxed in green and a critical path is shown as a red dashed line. Another critical path (which must of the same length) pass through the upper multiplexor. Note that the critical path is through the multiplexor select signal, not through the data inputs. The critical path length is $31\,\mathrm{u_t}$, adding on the register delay, we get the clock period of $[31+6]\,\mathrm{u_t} = 37\,\mathrm{u_t}$.

(*b*) Based on your answers above determine the latency and throughput for this calculation.

☑ The latency is:

Since it takes four cycles to compute a result the latency is $4 \times 37\,\mathrm{u_t} = 148\,\mathrm{u_t}$.

☑ The throughput is:

The unit of work is computing a $v_0^2 + v_0v_1 + v_1^2$ value. It takes four cycles to do so, so in this case the throughput is one over the latency or $\frac{1}{4\times 37\,\mathrm{u_t}} = \frac{1}{148\,\mathrm{u_t}}$. If $\mathrm{u_t} = 1\,\mathrm{ns}$ then the throughput would be $\frac{1}{148\,\mathrm{ns}}$ or 6756756 calculations per second.

Problem 4: [10 pts] The `mult_tree_bfas` module below has a flaw: It won't compile if `wp < wa+wb`. That's a big deal, because in many—perhaps most—cases when one multiplies two w-bit integers all one wants is the w least significant bits of the product. *Note: In the original exam some object names were different and there was unused code setting high bits of the product to zero.*

☑ Modify the module so that it will work correctly for values of `wp<=wa+wb`. ☑ Do so in a way that generates less hardware even without optimization of unconnected nets and unread variables.

```
module mult_tree_bfas #( int wa = 16, int wb = wa, int wp = wa + wb )
   ( output uwire [wp-1:0] prod,
     input uwire [wa-1:0] a,      input uwire [wb-1:0] b );

   if ( wa == 1 ) begin
      assign prod = a ? b : 0;
   end else begin

      localparam int wa_re = wa / 2;
      localparam int wp_re = wb + wa_re;

      uwire [wp_re-1:0] prod_lo;


      uwire [wp_re-1:0] prod_hi;


      mult_tree_bfas #(wa_re,wb) mlo( prod_lo, a[wa_re-1:0],  b );


      mult_tree_bfas #(wa_re,wb) mhi( prod_hi, a[wa-1:wa_re], b );


      assign prod[wa_re-1:0] = prod_lo[wa_re-1:0];


      uwire c[wp-1:wa_re-1];
      assign c[wa_re-1] = 0;


      for ( genvar i=wa_re; i<wp_re; i++ )
        bfa b(c[i], prod[i], prod_lo[i], prod_hi[i-wa_re], c[i-1] );


      for ( genvar i=wp_re; i<wp_re+wa_re; i++ )
        bha b(c[i], prod[i], prod_hi[i-wa_re], c[i-1] );

   end
endmodule
```

The solution is on the next page.

```
module mult_tree_bfas #( int wa = 16, int wb = wa, int wp = wa + wb )
   ( output uwire [wp-1:0] prod,
     input uwire [wa-1:0] a,      input uwire [wb-1:0] b );

   if ( wa == 1 ) begin
      assign prod = a ? b : 0;
   end else begin

      localparam int wa_re = wa / 2;
      localparam int wp_re = wb + wa_re;

      // SOLUTION:
      // Compute the width of the product actually needed from the lo and hi modules.
      localparam int wp_lo = min( wp_re, wp );
      localparam int wp_hi = min( wp_re, wp - wa_re );

      // SOLUTION: Possibly use fewer than wp_re bits for the product.
      uwire [wp_lo-1:0] prod_lo;
      uwire [wp_hi-1:0] prod_hi;

      // SOLUTION: Compute how many bits of b are needed in the hi module.
      localparam int wb_hi = min( wb, wp_hi );

      // SOLUTION: Instantiate using the smaller values for the number
      // of bits in the product (wp_lo, wp_hi) and a smaller value for
      // the number of bits in b (wb_hi).
      mult_tree_bfas #(wa_re, wb,    wp_lo) mlo( prod_lo, a[wa_re-1:0],  b );
      mult_tree_bfas #(wa_re, wb_hi, wp_hi) mhi( prod_hi, a[wa-1:wa_re], b[wb_hi-1:0] );

      assign prod[wa_re-1:0] = prod_lo[wa_re-1:0];

      uwire c[wp-1:wa_re-1];
      assign c[wa_re-1] = 0;

      // SOLUTION: Use wp_lo and wp_hi in the loop bounds so that
      // there are only as many BFA and BHA modules as needed.
      for ( genvar i=wa_re; i<wp_lo; i++ )
        bfa b(c[i], prod[i], prod_lo[i], prod_hi[i-wa_re], c[i-1] );
      for ( genvar i=wp_lo; i<wp_hi+wa_re; i++ )
        bha b(c[i], prod[i], prod_hi[i-wa_re], c[i-1] );

   end
endmodule
```

Problem 5: [35 pts] Answer each question below.

(*a*) When is it less expensive to implement design X using an FPGA, and when is it less expensive to implement design X (the same design) using an ASIC? Cost here refers to the purchase price, not something computed using the simple model.

☑ An FPGA is less expensive for design X when ... ☑ Explain.

... when only a small number will be fabricated, say 1. An FPGA is a moderately priced mass-produced component, so you are sharing the development costs with many other customers. An ASIC is made just for you, so even if you want one you are paying for a whole wafer full of chips, plus the cost of the masks needed for fabrication.

☑ An ASIC is less expensive for design X when ... ☑ Explain.

... when a large number will be fabricated, say 10,000. An ASIC contains just the logic you need, unlike an FPGA which has customizable logic (such as little look-up tables), customizable connections between the logic, not to mention left-over stuff that you didn't use. Therefore, if you fabricate enough ASIC chips the per-chip cost will be less than an FPGA.

(*b*) A testbench is written to verify whether a Verilog module does what it is supposed to do. (It's not just for homework assignments.) Consider a component that could quickly and thoroughly be tested after it has been manufactured.

☑ Is a testbench still necessary for the Verilog description of this component?

☑ Explain.

Strictly speaking a testbench is not necessary, but practically speaking it is very necessary. A testbench can let the engineer know if the HDL has an error in a short time, perhaps seconds. The testbench might even provide information that can be used to find the flaw in the HDL. If there was no testbench then the component would need synthesized, fabricated, then tested. At best that would take minutes (say, for an FPGA), but for an ASIC it might take weeks. Even if it were just minutes, that would add up until the design were working correctly.

A company has two testbench teams, the good team, and the okay team. (The good team is much better than the okay team.) Is it better to use the good team (rather than the okay team) for the testbench when the design is being made into an FPGA or when the design is being made into an ASIC?

☑ Better to use the good team for writing the testbench when fabricating an ◯ *FPGA* or ⊗ *ASIC* .

☑ Explain.

Suppose the testbench written by the good team finds a flaw that the okay team's testbench missed. For the ASIC, that discovered flaw would result in weeks of lost time while the flawed design was fabricated and then tested. It would also cost lots of money. For the FPGA, perhaps only hours of time are wasted by synthesizing and downloading a flawed design. So for that reason it better to use the good team for the ASIC designs.

(*c*) In each code fragment below indicate whether the non-blocking assignments are necessary, must be replaced by a blocking assignment, or whether it does not matter which is used. Assume typical use of Verilog.

☑ Are the non-blocking assignments ◯ *necessary,* ◯ *must be replaced by blocking assignments,* ⊗ *either one will work* .

☑ Explain.

```
// Fragment A
always_comb begin x <= a + y; end   // Line 1
always_comb begin a <= b + c; end   // Line 2
```

Short answer: The value of x will be updated either way (with or without the non-blocking assignments) in the same time step.

Discussion: Notice that a is referenced in Line 1 and written in Line 2. Each is an always_comb, and so each line executes whenever its live-in objects change. The live-in objects for Line 1 are a and y, and the live-in objects for Line 2 are b and c. If y and b both change, then Line 1 might be executed before Line 2. But because Line 2 changes a Line 1 will execute a second time. Because a is assigned using a non-blocking assignment a is not actually changed until all the active-region work is complete. But once that happens a is changed and that leads to an execution of Line 1.

☑ Are the non-blocking assignments ⊗ *necessary,* ◯ *must be replaced by blocking assignments,* ◯ *either one will work* .

☑ Explain.

```
// Fragment B
always_ff @( posedge clk ) begin x <= a + y; end   // Line 1
always_ff @( posedge clk ) begin a <= b + c; end   // Line 2
```

The non-blocking assignments are necessary because each line will execute just once in reaction to the positive edge. Without the non-blocking assignment results would depend on whether Line 1 was executed before or after Line 2.

(*d*) Consider three ways of designing digital hardware: combinational, sequential, and pipelined.

Sequential hardware is the lowest-cost alternative for many designs. (Some of which appear on this test.) Provide an example of some non-trivial hardware for which a sequential design would not be less expensive than a combinational design. The hardware might compute an arithmetic expression, as does the hardware in Problem 1.

☑ Non-trivial hardware that can't be made less expensive with a sequential design compared with a combinational design. ☑ Explain.

Short Answer: Hardware for computing $a * b + c$, because the each operation is performed once. (Assuming a sequential adder and multiplier are not practical.)

Explanation: A sequential design has a lower cost than a combinational design when something in the combinational design can be used multiple times. Expression $v_0^2 + v_0 v_1 + v_1^2$ can be computed by a sequential circuit consisting of one multiplier (used three times) and one adder (used twice). But for $a * b + c$ there would be one multiplier and one adder in both the combinational and sequential designs, so there is no cost benefit for the sequential design.

(*e*) Both modules below have an input port providing an array of unsigned integers, and an output port, `elt_min`, which is set to the smallest of these numbers. The two modules are nearly identical, the difference is that in `min_b_s` (the s is for shortcut) the loop ends when a value of 0 is found (because there can't be anything smaller, so why bother looking), while in `min_b` the loop always iterates for `n-1` iterations. Consider a situation in which most inputs contain a zero. Which module has a shorter critical path (meaning that it is faster in a typical digital design)?

```
module min_b #( int w = 4, int n = 8 )
   ( output logic [w-1:0] elt_min, input uwire [w-1:0] elts[n] );
   always_comb begin
      elt_min = elts[0];
      for ( int i=1; i<n; i++ )
        if ( elts[i] < elt_min ) elt_min = elts[i];
   end
endmodule

module min_b_s #( int w = 4, int n = 8 )
   ( output logic [w-1:0] elt_min,  input uwire [w-1:0] elts[n] );
   always_comb begin
      elt_min = elts[0];
      for ( int i=1; i<n && elt_min > 0; i++ )
        if ( elts[i] < elt_min ) elt_min = elts[i];
   end
endmodule
```

☑ Which module has a shorter critical path, ⊗ `min_b` or ○ `min_b_s` ?

☑ Explain.

The hardware in `min_b` is simpler so it likely has a shorter critical path. For hardware there is no benefit in ending the loop early.