Name ______________________________

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 ______________________________

Good Luck!

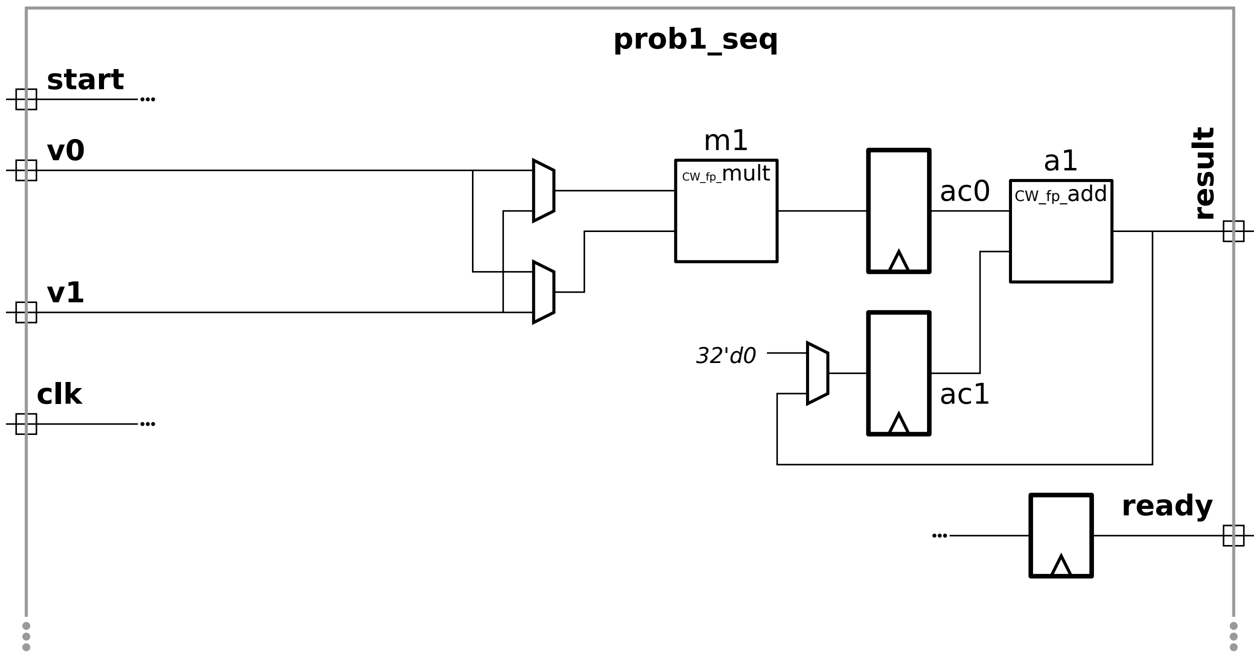
Problem 1: [20 pts] Module `prob1_seq`, below, is based on the solution to 2016 Final Exam Problem 1 (also appearing in problem set `https://www.ece.lsu.edu/koppel/v/guides/pset-syn-seq-main.pdf`, please look at that solution). In that problem an incomplete diagram of the hardware was given, similar 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
```


prob1_seq
start
v0
v1
clk
m1
CW_fp_mult
ac0
a1
CW_fp_add
result
32'd0
ac1
ready

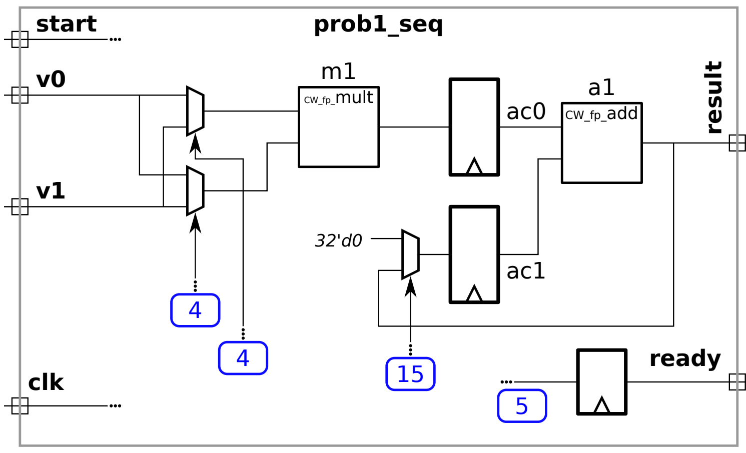
Problem 2: [20 pts] Consider again that module from Problem 1 of the 2016 final exam. Appearing below is the start of a Verilog description of a pipelined version of this module. The ports are the same as in the sequential version from the previous problem, however the module must operate in pipelined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

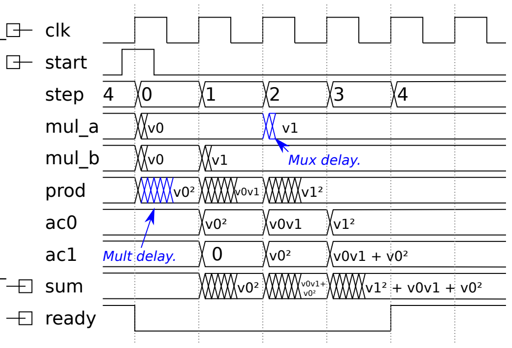
Complete the module. Two floating-point units are instantiated for your convenience. Add floating-point and other hardware as needed.

☐ Complete module so that it operates in pipelined fashion.

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

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

   // Add or modify FP units and other hardware.

   CW_fp_mult m1( .a(mul_a), .b(mul_b), .z(prod) );

   CW_fp_add  a1( .a(add_a), .b(add_b), .z(sum) );
















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.

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

☐ The latency is:

☐ The throughput is:

Problem 4: [10 pts] The `bfa_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.

☐ 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 wn = wa / 2;
      localparam int wx = wb + wn;

      uwire [wx-1:0] prod_lo;


      uwire [wx-1:0] prod_hi;


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


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


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

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


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


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


      localparam int wz = wp - wx - wn;
      if ( wz > 0 ) assign prod[wp-1 :- wz] = 0;

   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.

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

(*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.

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.

(*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
```

☐ 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
```

(*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.

(*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.