Name _____

Digital Design using HDLs
EE 4755
Final Examination

Thursday, 8 December 2016 12:30-14:30 CST

Problem 1	_____	(30 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(10 pts)
Problem 6	_____	(10 pts)
Exam Total	_____	(100 pts)

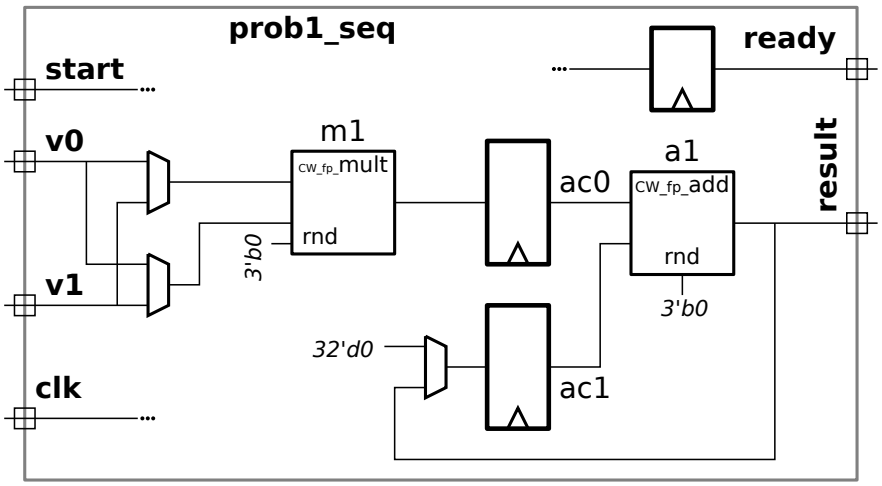
Alias _____

Good Luck!

Problem 1: [30 pts] The diagram and Verilog code below show incomplete versions of module `prob1_seq`. This module is to operate something like `mag_seq` from Homework 6. When `start` is `1` at a positive clock edge the module will set `ready` to `0` and start computing `v0*v0 + v0*v1 + v1*v1`, where `v0` and `v1` are each IEEE 754 FP single values. The module will set `ready` to `1` at the first positive edge after the result is ready.

Complete the Verilog code so that the module works as indicated and is consistent with the diagram. It is okay to change declarations from, say, `logic` to `uwire`. But the synthesized hardware cannot change what is already on the diagram, for example, don't remove a register such as `ac0` and don't insert any new registers in existing wires, such as those between the multiplier inputs and the multiplexors.

Don't modify this diagram, write Verilog code.

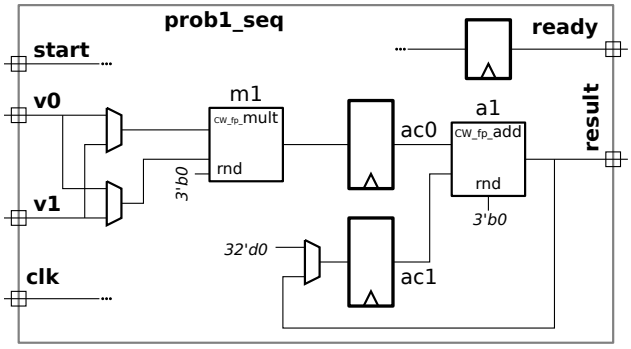


Don't modify this diagram, write Verilog code.

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

   uwire [7:0] mul_s, add_s;
   uwire [31:0] mul_a, mul_b;    uwire [31:0] add_a, add_b;    uwire [31:0]  prod, sum;

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

   localparam   int last_step = 1;
   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 ), .rnd(3'd0), .z( prod ), .status(mul_s));
   CW_fp_add  a1( .a( add_a ), .b( add_b ), .rnd(3'd0), .z( sum ),  .status(add_s));

   assign       ready = step == last_step; /// THIS MUST BE CHANGED.

   /// USE NEXT PAGE FOR SOLUTION!


endmodule
```

Problem 1, continued: Solution on this page.

Don't modify, Verilog only.



- ☐ Complete Verilog so that module computes `v0*v0 + v0*v1 + v1*v1`.
- ☐ Synthesized hardware must be consistent with diagram, ☐ especially synthesized registers.
- ☐ Note that `ready` must come from a register.
- ☐ Don't skip the easy part: connections to adder.

```
module prob1_seq( output uwire [31:0] result,   output uwire ready,
                  input uwire [31:0] v0, v1,    input uwire start, clk);
   uwire [7:0] mul_s, add_s;
   uwire [31:0] mul_a, mul_b;    uwire [31:0] add_a, add_b;    uwire [31:0]  prod, sum;
   logic [31:0] ac0, ac1;        logic [2:0]  step;

   localparam   int last_step = 1;                          //  ☐  MUST BE CHANGED.
   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 ), .rnd(3'd0), .z( prod ), .status(mul_s));
   CW_fp_add  a1( .a( add_a ), .b( add_b ), .rnd(3'd0), .z( sum ),  .status(add_s));

   assign       ready = step == last_step;                  //  ☐  MUST BE CHANGED.
```

```
endmodule
```

Problem 2: [20 pts] Analyze the timing of the two similar modules on the next page using the timing model used in class, as requested in the subproblems. Assume that **all adders are synthesized as a ripple connection of binary full adders** and that the **comparison units are also based on ripple hardware**.

(*a*) Before analyzing the modules, show the delay of each of the components listed below using the simple model given in class. For this part assume that all inputs are available at $t = 0$.

☐ Delay for BFA is:

☐ Explain or show diagram.

☐ Delay for a w-bit adder is:

☐ Explain or show diagram.

☐ Delay for a w-bit $<$ (less than) comparison unit is:

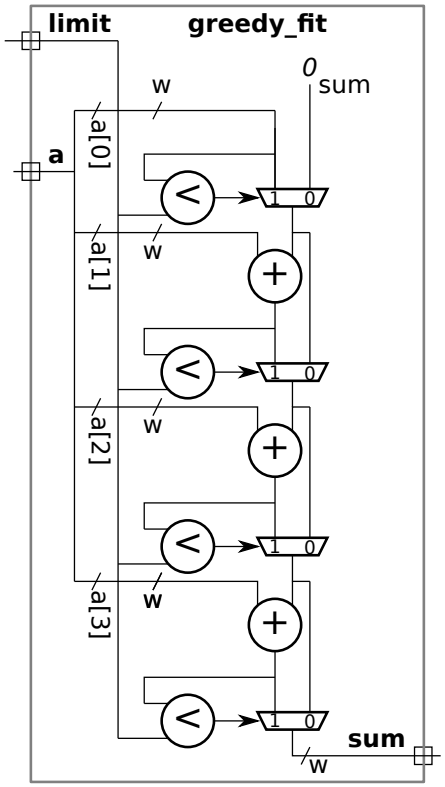
☐ Explain or show diagram.

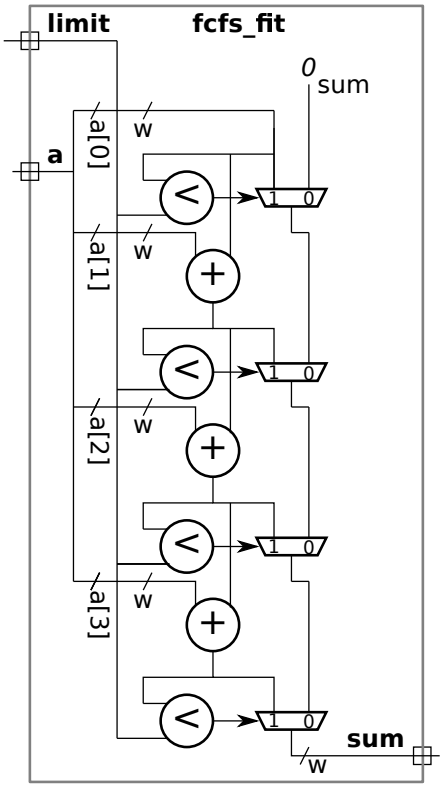
☐ Delay for a w-bit, n-input multiplexor is:

☐ Explain or show diagram.

Problem 2, continued:

(*b*) Find the length of critical path in the two modules below using the timings above. **Where applicable** make the reasonable assumption that a ripple adder can start when its lower bits arrive, not when all bits of its input are stable.

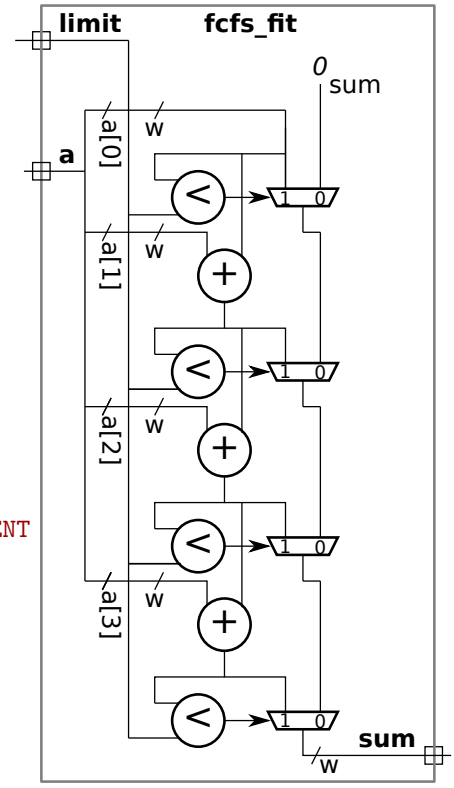




☐ Length of critical path for `greedy_fit` **in terms of** w. ☐ Show work for partial credit.

☐ Length of critical path for `fcfs_fit` **in terms of** w. ☐ Show work for partial credit.

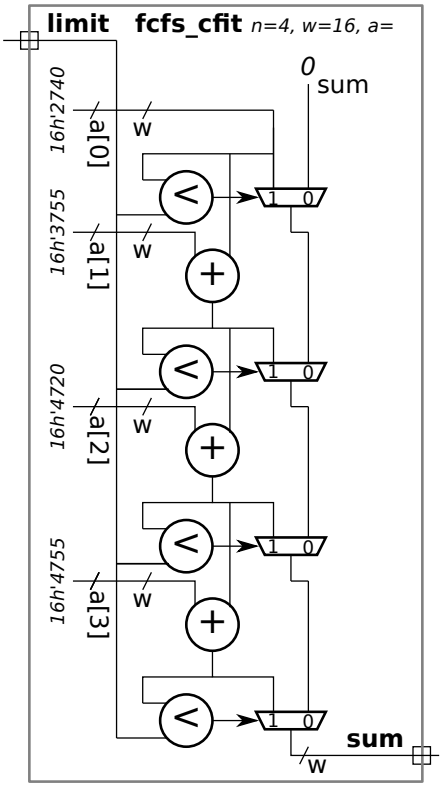
Problem 3: [15 pts] Complete the Verilog code so that it corresponds to the module shown.



☐ Complete module.

```
module fcfs_fit #( int nelts = 4, int w = 16 )
   ( output logic [w-1:0] sum,
     input uwire [w-1:0] a[nelts], limit );

   always              begin //  ☐  FINISH ALWAYS STATEMENT

      for ( int i=0; i<nelts; i++ ) begin

      end

   end

endmodule
```

Problem 4: [15 pts] Appearing to the right is `fcfs_cfit`, a version of the `fcfs_fit` module in which the `a` input has been changed to a parameter, **meaning that `a` is an elaboration-time constant**. Compute the cost of this module using the simple model used in class and **accounting for optimization based on the constant values**. As in an earlier problem, adders and comparision units are ripple-style.



☐ Cost of the `a[0]` comparison unit.

☐ Explain.

☐ Cost of the `a[1]` adder.

☐ Explain.

☐ Cost of the `a[0]` multiplexor.

☐ Explain.

☐ Cost of the `a[2]` multiplexor.

☐ Explain.

☐ Total cost.

Problem 5: [10 pts] Answer each question below.

(*a*) A time slot in the Verilog event queue contains many regions, among them *active*, *inactive*, and *NBA*. Explain how an event gets put in each region. (You can use the next subproblem for examples.)

☐ An event is put into the active region when:

☐ An event is put into the inactive region when:

☐ An event is put into the NBA region when:

(*b*) In the code fragment below show the order in which the statements are executed after the `posedge clk`. Identify a statement by the value that is assigned. The first two statements executed are `a` and `b`, that's shown. (Since `a` is a nonblocking assignment, the execution of `a` only means that `a+1` was computed, it doesn't mean that `a` was changed.) Complete the "Order of statements" list.

```
module regions;
   always_ff @( posedge clk ) begin
        a <= a + 1;
        b = b + 1;
    end

   always_comb s = a + b;
   always_comb ax = a + 2;
   always_comb ay = ax + 5;
   always_comb by = bx + 4;
   always_comb bx = b + 3;

endmodule
```

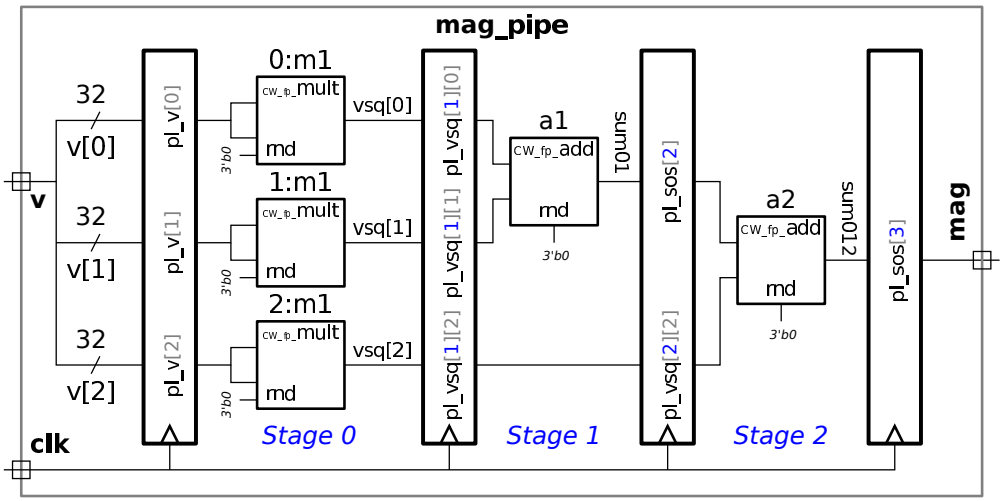
☐ Order of statements: a, b,

Problem 6: [10 pts] Appearing below is the pipelined mag module from Homework 6.

(*a*) Suppose it turns out that the multiply (`CW_fp_mult`) takes twice as long as the add (`CW_fp_add`). Based on this fact, modify the pipeline to reduce cost, but without affecting clock frequency. Draw in your changes, there's no need to write Verilog. Also, comment on latency and throughput changes.

☐ Modify for lower cost based on faster adder.

☐ Does the change ☐ help throughput? Does it help ☐ latency?



(*b*) Suppose that the `v` input arrives very early in the clock cycle. Based on this modify the pipeline to reduce cost.

☐ Modify for early-arriving `v`.