Name _____ *Formatted For Two-Sided Printing*

Staple This Side

Digital Design using HDLs
LSU EE 4755
Final Examination

Friday, 9 December 2022 15:00-17:00 CST

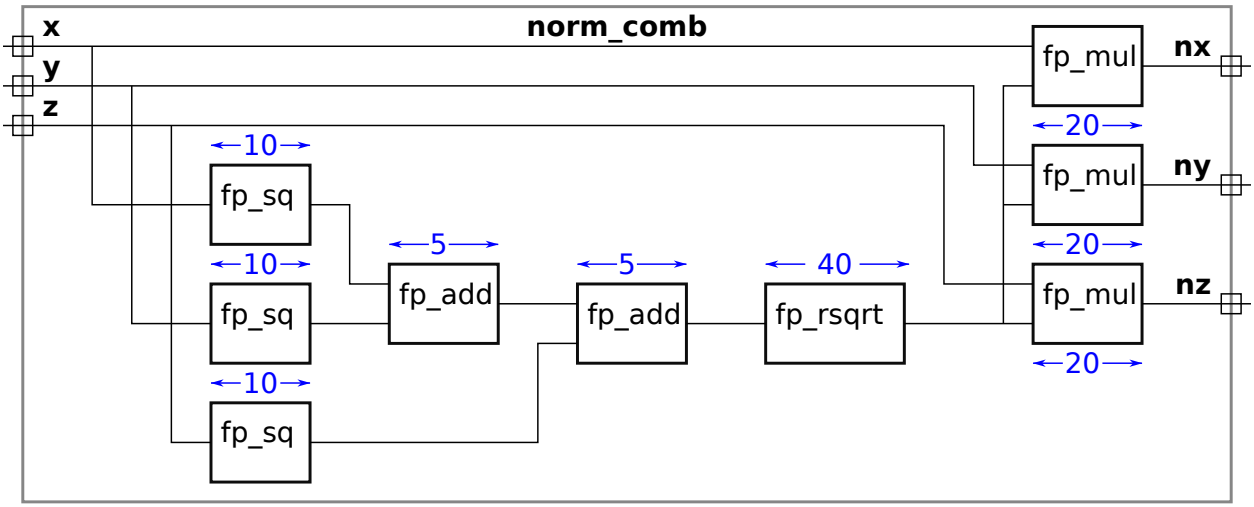
Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Staple This Side

Problem 1: [20 pts] Module `norm_comb`, below, computes the normal of a vector using floating-point arithmetic units from a library. The delay through each unit in nanoseconds is shown in the diagram.



(*a*) Compute the latency and throughput `norm_comb` given the timings shown in the diagram.

☐ Compute the arrival time (delay) at each module output.

☐ Show the critical path.

☐ The latency of this module is:

☐ The throughput of this module is:

Staple This Side

Staple This Side

(*b*) Draw a diagram of a pipelined implementation of the norm module. The goal is to maximize throughput first then minimize latency **given the delays shown in the diagram from part a**. Give some thought as to what arithmetic units go in what stage. Show the latency and throughput of your pipelined implementation.

☐ Draw a diagram (not Verilog) of a pipelined version of this `norm` module. ☐ Be sure to show pipeline latches.

☐ For the given delays: Maximize throughput. ☐ Avoid a hasty solution that has a higher latency than is necessary.

☐ The latency of this pipelined implementation is:

☐ The throughput of this pipelined implementation is:

Staple This Side

Problem 2: [20 pts] Incomplete module `norm_comb_n` is a version of the norm module from the previous problem, now written for vectors of any length, not just 3. (Output $u_i = n_i \left(\sum_{j=0}^{n-1} v_j^2\right)^{-\frac{1}{2}}$.) It makes use of module `norm_sos` to compute the sum $\sum_{j=0}^{n-1} v_j^2$. (That is, $v_0^2 + v_1^2 + \cdots + v_{n-1}^2$.) Complete the modules so that they compute their output combinationally. Use a recursive implementation for `norm_sos` and use generate loops for the needed code in `norm_comb_n`.

☐ Complete `norm_comb_n` so that it computes `u` in part by using `norm_sos`. ☐ Use a generate loop. ☐ Use `fp_mul`, ☐ don't use arithmetic operators.

```
module norm_comb_n #( int w = 32, int n = 8 )
   ( output uwire [w-1:0] u[n],   input uwire [w-1:0] v[n] );

   uwire [w-1:0] sos; // Sum Of Squares
   norm_sos #(w,n) ns( sos, v );    // This part is correct, don't modify it.

   uwire [w-1:0] rmag, rs_in;
   fp_rsqrt r(  rmag,  rs_in  );  // [ ] Rename rs_in, or connect it to something.

   // [ ] Compute u[0] = v[0] * rmag;  u[1] = v[1] * rmag; ...




endmodule
```

☐ Complete `norm_sos` so that it computes $\sum_{j=0}^{n-1} v_j^2$. ☐ Describe the module recursively. ☐ Use `fp_sq` and `fp_add`, ☐ do not use arithmetic operators.

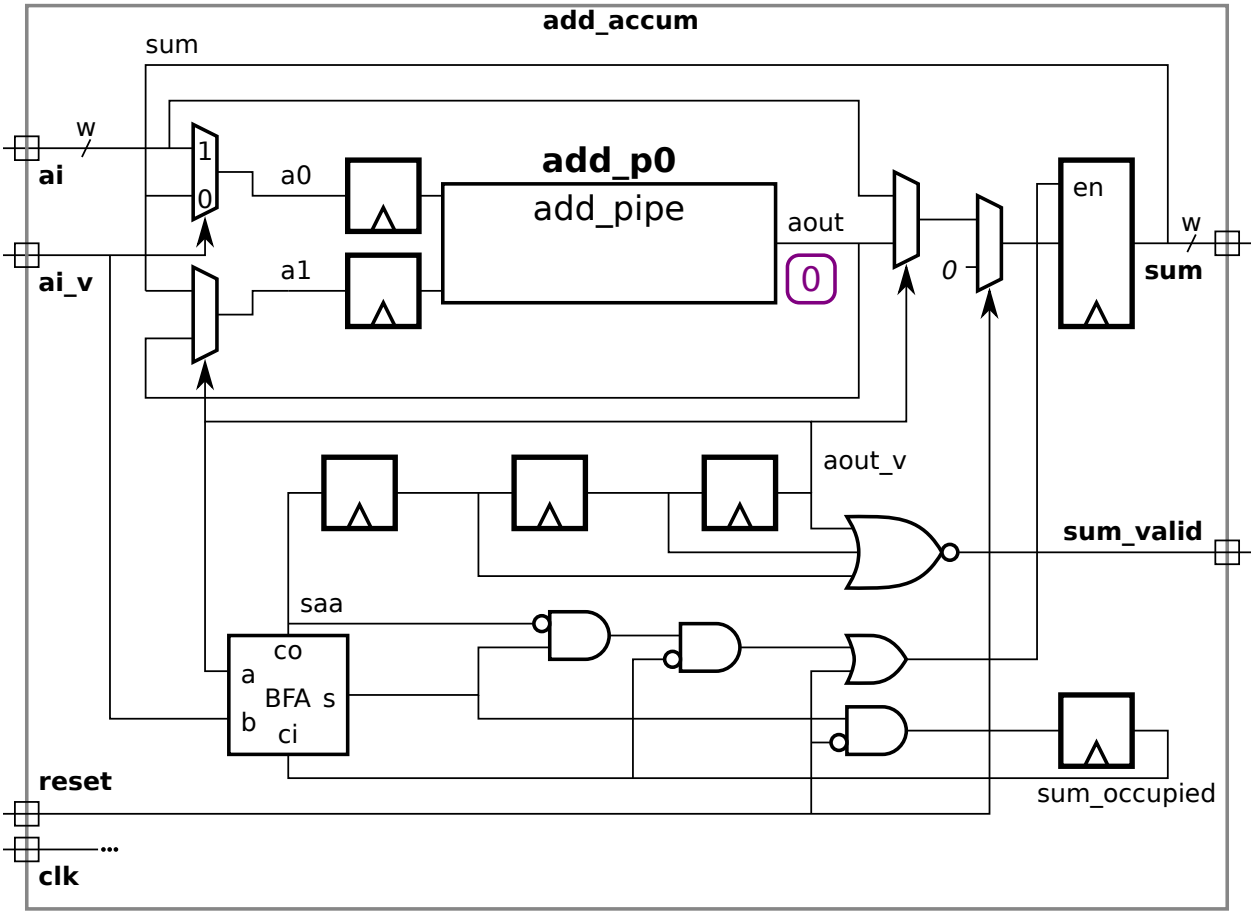
```
module norm_sos #( int w = 32, int n = 4 )
                 ( output uwire [w-1:0] sos,  input uwire [w-1:0] v[n-1:0] );
   // [ ] Recursively compute: sos = v[0]^2 + v[1]^2 + ...




endmodule
```


Staple This Side

Problem 3: [15 pts] Appearing below is the inferred hardware from the pipelined add accumulate module covered in class. Based on the simple model, show the timing, including the critical path, and compute the cost. The BFA module is, of course, a binary full adder. If you don't remember its cost and delay, just work it out.



☐ Show the timing (signal arrival time at each component output) and ☐ the critical path. ☐ Note that `aout` arrives at $t = 0$.

☐ Compute the cost using the simple model. Do not include the cost of `add_pipe` but ☐ include the cost of the BFA. ☐ Pay attention to bit widths.

Problem 4: [20 pts] Appearing below are simplified solutions to Homework 4.

(*a*) Below is a simplified version of the "official" solution. (Reset hardware is not shown, ignore its absence. Some object names shortened.) Show the hardware that will be inferred for this module when instantiated with `n_avg_of=4`. (Some of the hardware will be similar to the `r_avg2` module from the 2021 final exam.)

```
module word_count
  #( int wl = 5, wn = 6, n_avg_of = 10 )
   ( output logic word_start, word_part, word_ended,
     output logic [wl-1:0] lword, lavg,          output logic [wn-1:0] nwords,
     input uwire [7:0] char,                     input uwire reset, clk );

   uwire nws, nwp, nwd;
   word_classify wc( word_start, word_part, word_ended,
         nws, nwp, nwd, char, clk, reset );

   logic [wl-1:0] lrecent[n_avg_of];   // len_recent
   logic [wl+$clog2(n_avg_of):0] lsum; // len_sum

   assign lavg = nwords >= n_avg_of ? lsum / n_avg_of : 0;

   always_ff @ ( posedge clk ) begin
      lword <= nws ? 1 : nwp ? lword+1 : lword;
      nwords <= nwd ? nwords + 1 : nwords;
   end

   // Plan A Code (Referred to in next subproblem.)
   always_ff @ ( posedge clk ) if ( nwd ) begin

      lsum += lword - lrecent[n_avg_of-1];
      for ( int i=n_avg_of-1; i>0; i-- ) lrecent[i] = lrecent[i-1];
      lrecent[0] = lword;

   end
endmodule
```


Staple This Side

- [ ] Show inferred hardware for `n_avg_of=4`.
- [ ] Show `word_classify` as a box, don't attempt to show its contents.

Staple This Side

(*b*) The `word_count_plan_b` module below uses a different approach to keeping track of `lsum`. The only difference is the hardware under the `Plan B Code` comment. This version avoids a loop! That's great, right? Show the hardware that will be inferred for the `Plan B Code` for `n_avg_of = 4` and indicate impact on cost and performance.

```
module word_count_plan_b
  #( int wl = 5, wn = 6, n_avg_of = 10 )
   ( output logic word_start, word_part, word_ended,
     output logic [wl-1:0] lword, lavg,          output logic [wn-1:0] nwords,
     input uwire [7:0] char,                     input uwire reset, clk );

   uwire nws, nwp, nwd;
   word_classify wc( word_start, word_part, word_ended,
         nws, nwp, nwd, char, clk, reset );

   logic [wl-1:0] lrecent[n_avg_of];
   logic [wl+$clog2(n_avg_of):0] lsum;
   logic [$clog2(n_avg_of):0] tail;

   assign lavg = nwords >= n_avg_of ? lsum / n_avg_of : 0;

   always_ff @ ( posedge clk ) begin
      lword <= nws ? 1 : nwp ? lword+1 : lword;
      nwords <= nwd ? nwords + 1 : nwords;
   end

   // Plan B Code
   always_ff @ ( posedge clk ) if ( nwd ) begin

      lsum += lword - lrecent[tail];
      lrecent[tail] = lword;
      tail = tail == n_avg_of - 1 ? 0 : tail + 1;

   end

endmodule
```

☐ Describe impact on cost of Plan B compared to Plan A.

☐ Describe impact on performance of Plan B compared to Plan A.

Staple This Side

☐ Show inferred hardware for `Plan B Code`. (No need to show hardware for code above the `Plan B Code` comment.)

☐ Consider using an enable (`en`) signal on the registers to simplify the hardware.

Staple This Side

Problem 5: [25 pts] Answer each question below.

(a) Show a sketch of the hardware for an 8-bit left shift module, using the logarithmic approach presented in class.

☐ Show hardware for 8-bit left shift module. Include the ☐ 3-bit shift amount input, ☐ the 8-bit data ☐ input and 8-bit data output.

(b) Provide the following delays based on the simple model.

☐ What is the delay for a w-bit ripple adder for ☐ the LSB and ☐ the MSB.

☐ What is the delay for the sum of three w-bit values, say $a + b + c$, when computed using two ripple adders and accounting for cascading. Delay of the sum's ☐ LSB and ☐ MSB.

Staple This Side

Staple This Side

(*c*) In the code fragment below there is an error in one of the last two lines.

```
module examples( input uwire [31:0] a, b );
   localparam logic [31:0] la = a + b;
   uwire logic [31:0] ua = a + b;
```

☐ Which line above is incorrect? ☐ Why?

(*d*) The code fragment below lacks declarations.

☐ Declare objects `aa`, `ca`, and `fa` so that the code below is correct.

```
module examples( input uwire [31:0] a, b,  input uwire clk );

   assign aa = a + b;
   always_comb ca = a + b;
   always_ff @( posedge clk ) fa = a + b;
```

(*e*) Again consider the code above that assigns `aa`, `ca`, and `fa`. Draw a timing diagram that includes values of `a`, `b`, and `clk` for which at least one of the values `aa`, `ca`, and `fa` will at times differ from the others.

☐ Draw a timing diagram showing how `aa`, `ca`, and `fa` won't all be the same all the time.

Staple This Side