Name Solution _____ *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

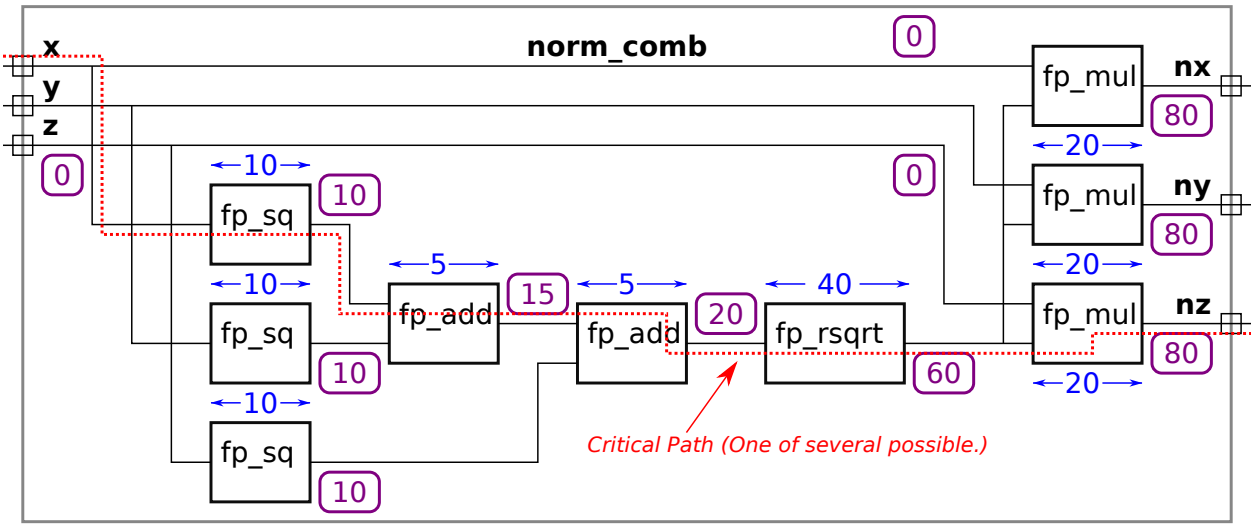
Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias Multiplexor Mayhem (Student Suggestion) _____

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.

Arrival times and delays at the outputs are shown as circled purple numbers.

☑ Show the critical path.

A critical path is shown as a red dotted line. Several others are possible, for example, another critical path starts at `y`. The illustrated critical path ends at `nz`, but it could have ended at `ny` or `nx`.

☑ The latency of this module is:

The latency is 80 ns.

(Because this is a combinational module, the latency is the same as the critical path.)

☑ The throughput of this module is:

Assuming that the clock period is the same as the critical path length, the throughput is $\frac{1\,\text{op}}{80\,\text{ns}} = 12.5\,\text{M}\frac{\text{op}}{\text{s}}$, where op refers to a normalization operation. (The throughput is given in units of work per unit time. The unit of work here is a normalization, and the unit of time is second.)

(*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 diagram appears below. Stage boundaries were chosen to minimize critical path, which is 40 ns due to the `fp_rsqrt` module.

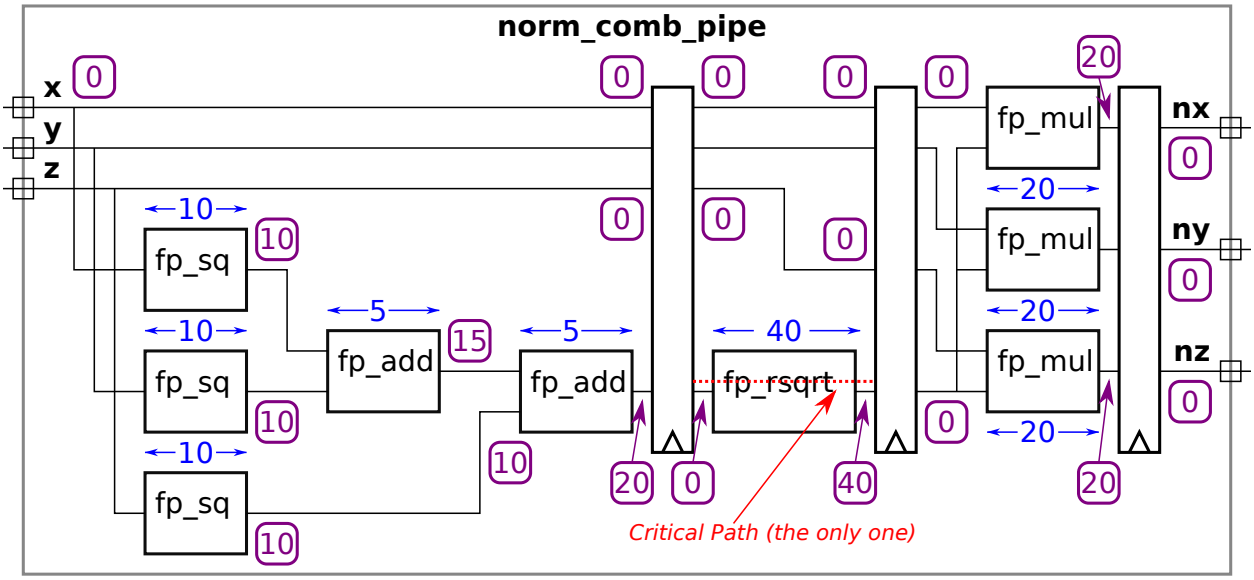
Staple This Side

The following discussion is to help future students understand the solution. Those taking the test need only show the diagram. To make it a pipeline, pipeline latches (collections of registers) have been inserted to divide the arithmetic units into three stages. The positioning of the pipeline latches has been chosen to minimize the critical path, and so maximizing clock frequency and throughput.

Recall from course material that the launch points are assumed to be module inputs and are always at register outputs. The arrival times at launch points are by definition zero. The capture points are always register inputs. In general they can be module outputs, but here we are assuming that the module outputs are not capture points, meaning that module outputs must be stable *at the beginning of a clock cycle*. (It would also be correct to assume that module outputs were capture points, so long as the computation of latency and throughput took this into account.) The diagram shows arrival times circled in purple including delays at the capture points.

The critical path, shown in a red dashed line, is $40\,\text{ns}$, and so the clock period must be set to $40\,\text{ns}$ (plus the delay of the register). The path length in the other two stages is $20\,\text{ns}$. Were it not for `fp_rsqrt` the clock period would be half (and so the clock frequency would be twice as high). But it is what it is, and so the calculations in the first and last stages finish with $20\,\text{ns}$ of slack (meaning they arrive $20\,\text{ns}$ before the end of the clock cycle, which by coincidence is $20\,\text{ns}$ after the start of the clock cycle).

In a correct solution the `fp_rsqrt` module must be in a stage by itself. For example, were an `fp_mul` moved into the stage with the `fp_rsqrt` then the critical path would increase to $60\,\text{ns}$, hurting performance. Though it would be possible to put the two adders in their own stage without changing the clock period, that would increase cost because another pipeline latch would be needed.



☑ The latency of this pipelined implementation is:

Latency refers to the time to complete a normalization operation. The pipeline has three stages and the clock period is at least $40\,\text{ns}$ (the critical path length). Therefore the $\boxed{\text{latency is } 3 \times 40\,\text{ns} = 120\,\text{ns}}$.

Notice that the latency is higher than the combinational module. That is due to the $20\,\text{ns}$ of slack in the first and last stages.

☑ The throughput of this pipelined implementation is:

Because the implementation is pipelined a new result is computed each clock cycle so the $\boxed{\text{throughput is } \frac{\text{cyc}}{40\,\text{ns}}\frac{\text{op}}{\text{cyc}} = 25\,\text{M}\frac{\text{op}}{\text{s}}}$. Notice that the throughput is higher than the combinational module. That's because the module simultaneously computes three operations.

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.

```
// SOLUTION
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,  sos );  // SOLUTION: Changed rs_in to sos.

   // SOLUTION: Use a genvar loop to instantiate one fp_mul per element.
   for ( genvar i=0; i<n; i++ )
     fp_mul mi( u[i], v[i], rmag );
endmodule
```

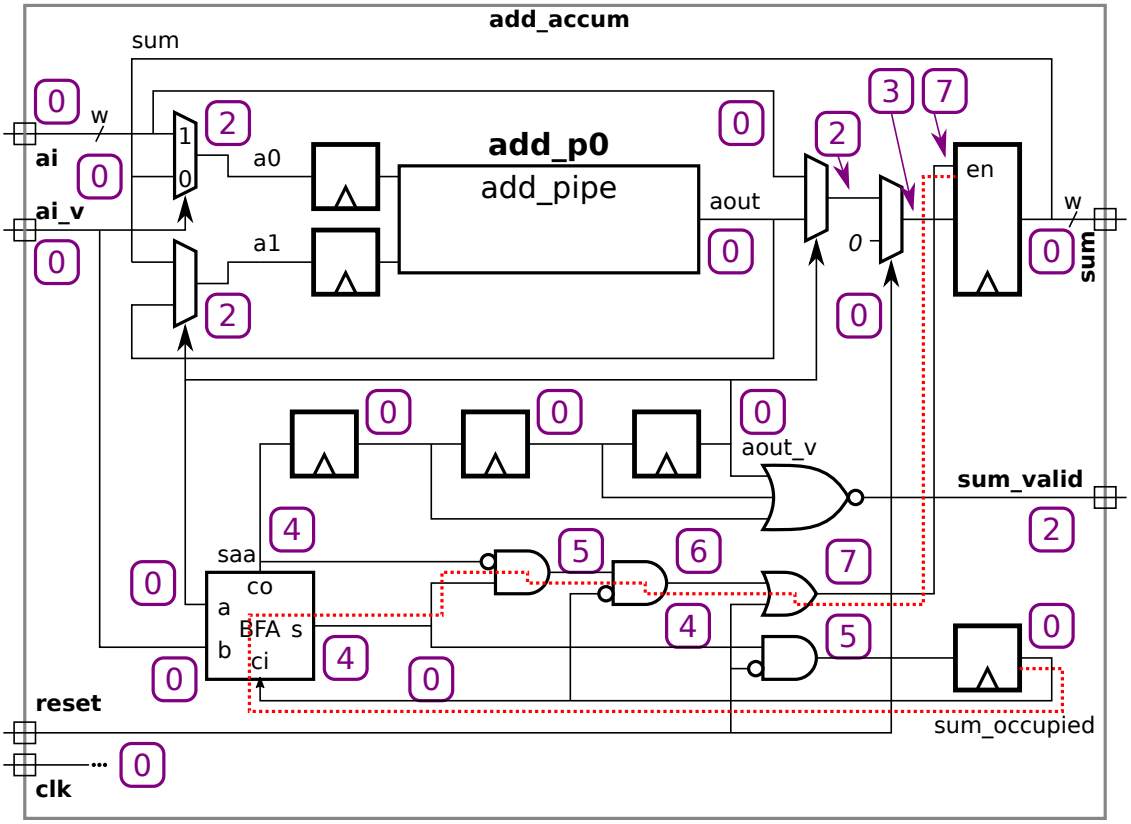
☑ Complete `norm_sos` so that it computes $\sum_{j=0}^{n-1} v_j^2$. ☑ Describe the module recursively. ☑ Use `fp_sq` and `fp_add`, ☑ do not use arithmetic operators.

```
module norm_sos #( int w = 32, int n = 4 )
                 ( output uwire [w-1:0] sos,  input uwire [w-1:0] v[n-1:0] );
   // SOLUTION
   if ( n == 1 ) begin

      fp_sq s( sos, v[0] );

   end else begin

      localparam int nlo = n/2;
      localparam int nhi = n - nlo;

      uwire [w-1:0] soslo, soshi;

      norm_sos #(w,nlo) slo( soslo, v[nlo-1:0] );
      norm_sos #(w,nhi) shi( soshi, v[n-1:nlo] );
      fp_add #(w) a( sos, soslo, soshi );

   end
endmodule
```


Staple This Side

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 $t = 0$.

Solution appears above. Arrival times are circled purple numbers and the critical path is a dashed red line.

☑ 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.

The $\boxed{\text{total cost is } [34w + 43]\ \text{u}_\text{c}}$. The table below shows the cost of each kind of component.

Item	Count	Each	Total
Non-Constant 2-input, w-bit Multiplexors	3	$3w$	$9w$
Constant 2-input, w-bit Multiplexor	1	w	w
w-bit Register with Enable	1	$10w$	$10w$
w-bit Registers without Enable	2	$7w$	$14w$
1-bit Registers	4	7	28
2-input Gates	4	1	4
3-input NOR Gate	1	2	2
BFA	1	9	9
Total Cost			$34w + 43$

Staple This Side

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`.

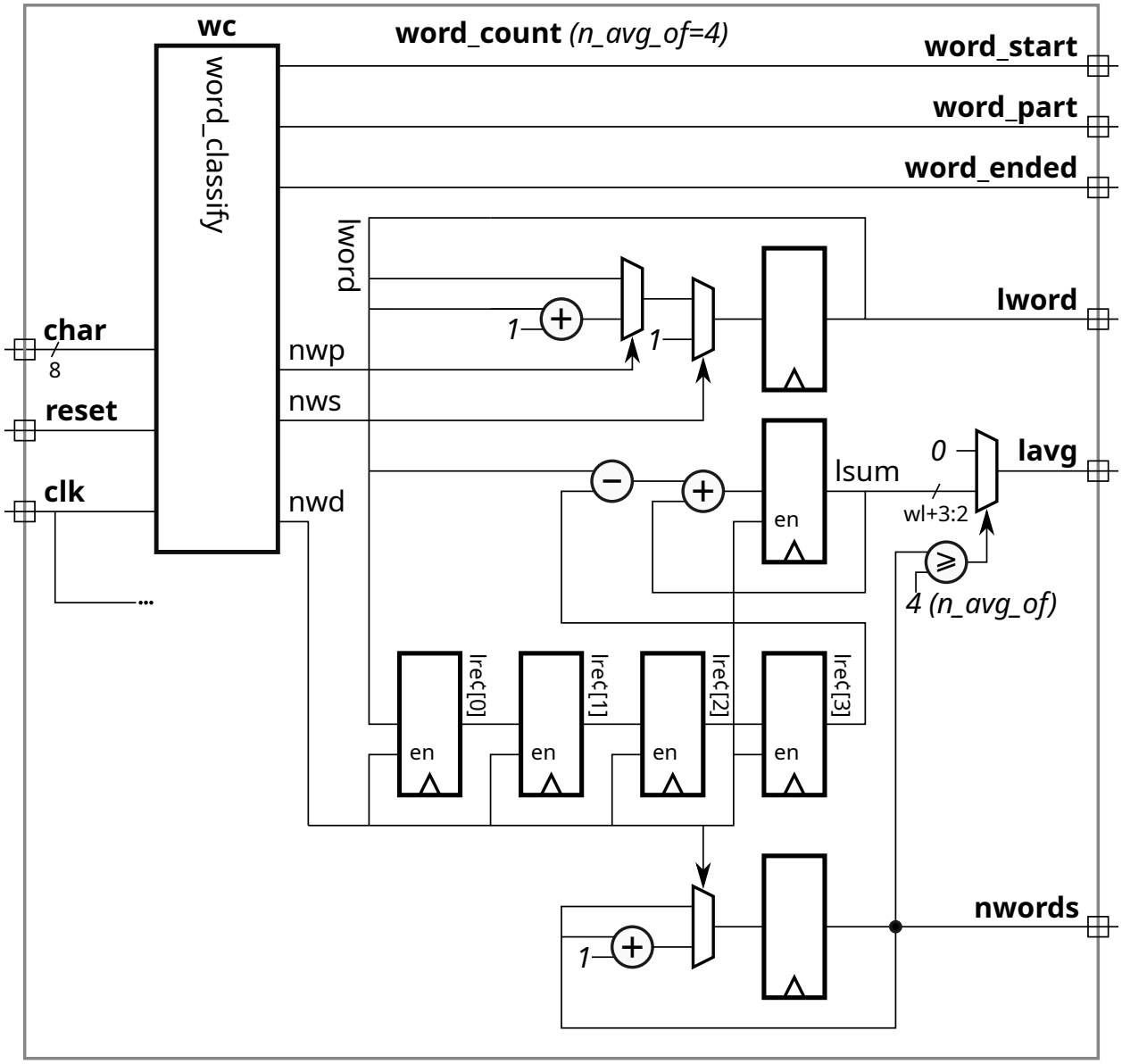
☑ Show `word_classify` as a box, don't attempt to show its contents.

Solution appears below.

Note that the value of `lword` used to compute `lsum += lword - lrecent[n_avg_of]` is the value at the register *output*. That's because `lword` is assigned using a non-blocking assignment. (It would have been wrong to assign `lword` using a blocking assignment because then whether the `lsum` expression used an old or new `lword` would depend on simulator timing.)

Because `n_avg_of = 4` (a power of 2) the term `lsum/n_avg_of` has been inferred as simply consisting of all but the two least-significant bits of `lsum`. Dividers are expensive so this is a good thing.

The body of the last `always_ff` block is guarded by a `if ( nwd )`. That is inferred as an enable on all of the registers inferred for that block, which is `lsum` and the `lrecent` registers.



(*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.

Plan B would be much more expensive due to the `lrecent[tail]` terms. The inferred hardware for `lrecent[tail]` used on the right-hand-side of an expression is an `n_avg_of`-input multiplexor. The cost of the hardware for `lrecent[tail]=lword` would be a decoder to provide enable inputs to the `lrecent` registers. There is also the cost of the `tail` register and the associated adder. None of this hardware is needed for Plan A.

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

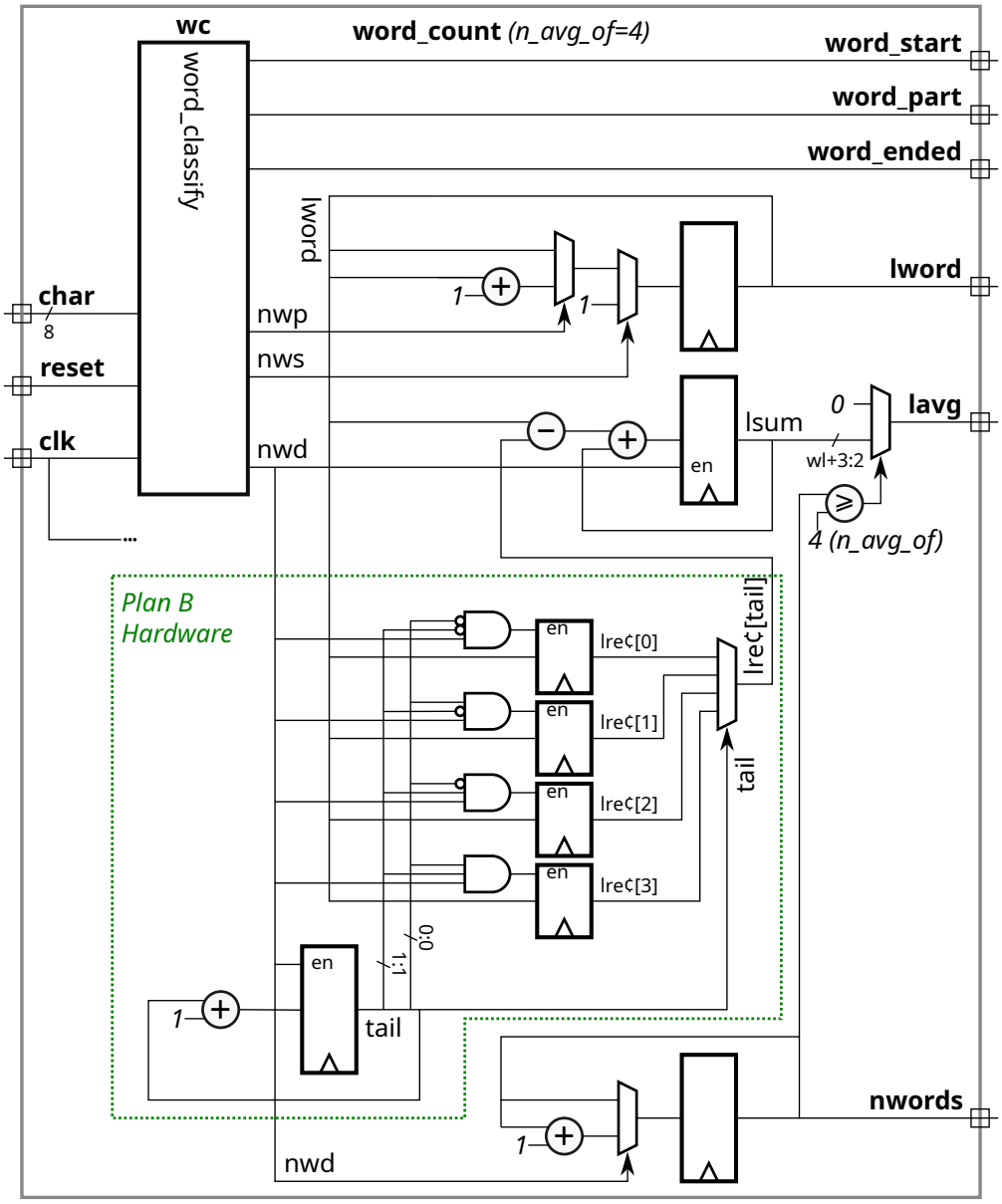
Because of the two arithmetic units (subtract and add) operating on non-constant values it is likely that `lrecent[tail]` and `lrecent[n_avg_of]` are on the critical paths in their respective modules. Plan B adds $2 \lg n_{\text{avg of}} \, \mathrm{u_t}$ to the critical path in comparison with Plan A, so it certainly hurts performance.

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.

The inferred hardware corresponding to the `Plan B Code` appears below, circled by a green dotted line. The four `lrecent` registers also appear in the `Plan A` design. Everything else is an added cost.

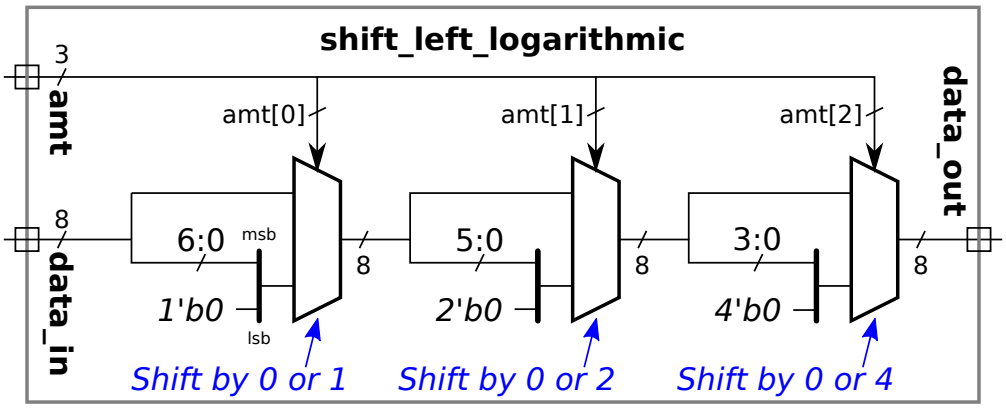


Problem 5: [25 pts] Answer each question below.

(*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.

Solution appears below.



(*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.

The delay of the LSB is $4\,\mathrm{u_t}$ and the delay of the MSB is $2(w+1)\,\mathrm{u_t}$.

☑ 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.

The general formula for the simple-model delay of bit i at the output of n cascaded ripple adders is $[4(n-1)+2(i+2)]\,\mathrm{u_t}$. For this case substitute $n \to 2$. For the LSB, $i \to 0$ and for the MSB, $i \to w-1$.

The delay of the LSB is $8\,\mathrm{u_t}$ and the delay of the MSB is $[8+2(w-1)]\,\mathrm{u_t}$.

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;  // Incorrect.
   uwire logic [31:0] ua = a + b;       // Correct.
```

☑ Which line above is incorrect? ☑ Why?

The first line is incorrect because the value assigned to `localparam` must be an elaboration-time constant. Since `a` and `b` are module inputs they are not elaboration time constants.

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

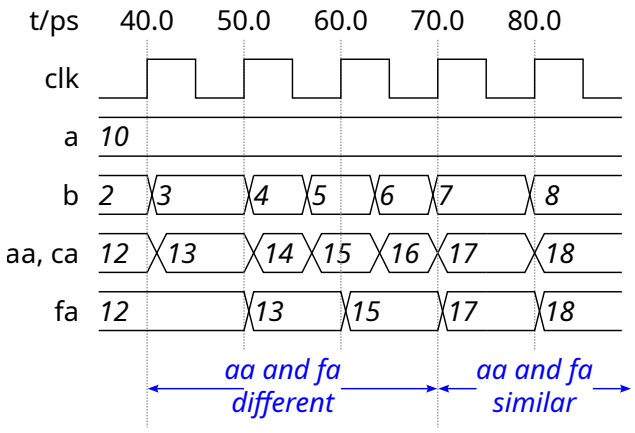
☑ Declare objects `aa`, `ca`, and `fa` so that the code below is correct.

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

   uwire [31:0] aa;      // SOLUTION
   logic [31:0] ca, fa;  // SOLUTION

   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.

The timing diagram appears to the right. The timing of the changes on input `b` before $t = 70.0$ result in the output `fa` being different than `aa` and `ca` for much of the time. This is because changes `b` occur well before the positive edge of `clk`. Outputs `aa` and `ca`, because they are driven by combinational logic, will start changing as soon as `b` starts changing. In contrast `fa` only starts changing at the positive edge of `clk`, and the changes are based on the values of `a` and `b` that were present at the positive edge. For example, `b` starts to change at $t = 40.0$, which is too late for `fa` to change immediately, it must wait until $t = 50.0$. Starting at $t = 70.0$ changes to `b` complete just before the positive edge, and so `aa` and `fa` have close to identical timing.



Staple This Side