Name Solution *Formatted For Two-Sided Printing*

Staple This Side

Digital Design using HDLs
LSU EE 4755
Final Examination

Thursday, 7 December 2023 15:00-17:00 CST

Problem 1		(28 pts)
Problem 2		(25 pts)
Problem 3		(27 pts)
Problem 4		(20 pts)
Exam Total		(100 pts)

Alias $Q\star$?

Good Luck!

Staple This Side

Problem 1: [28 pts] Appearing below is the solution to Homework 5.

(*a*) On the facing page show the inferred hardware for an instantiation with `n=4`.

(*b*) Explain why the cost of the hardware corresponding to the line `n_match += match` is much lower than one would expect for hardware performing `wc`-bit addition.

☑ The `n_match += match` is much less expensive because:

Because `match` is only one bit. Also, `n_match` starts off with an initial value of `1` so the number of bits needed for the early `i` iterations is small.

Grading Note: A very similar question was asked in Homework 4, Problem 1b.

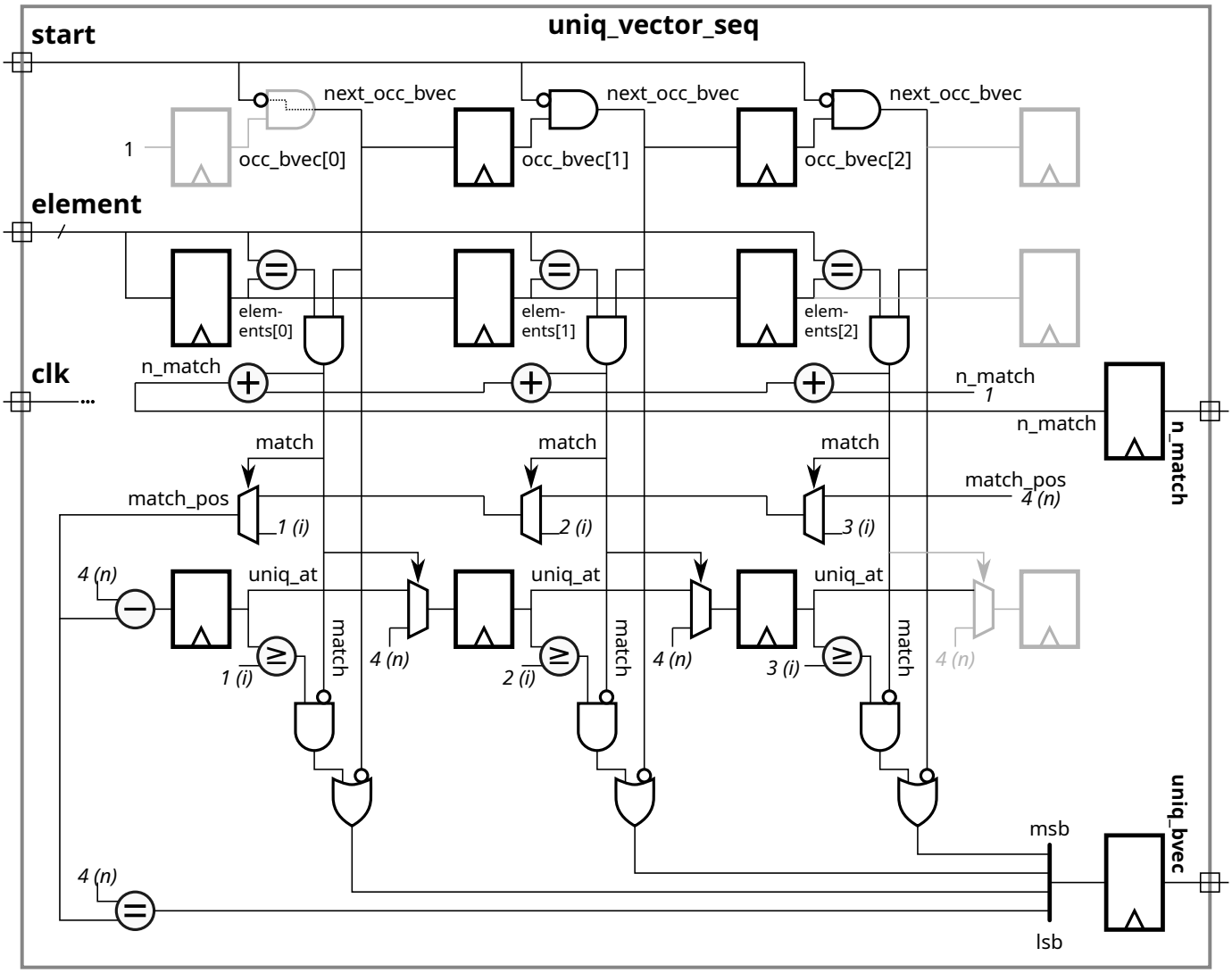
```
module uniq_vector_seq
  #( int we = 10, n = 4, wc = $clog2(n+1) )
   ( output logic [n-1:0] uniq_bvec,     output logic [wc-1:0] n_match,
     input uwire [we-1:0] element,       input uwire start, clk );

   logic [we-1:0] elements [n-1:0];
   logic [n-1:0] occ_bvec;
   logic [wc-1:0] uniq_at [n-1:0];

   always_ff @( posedge clk ) begin

      automatic logic [wc-1:0] match_pos = n;
      n_match = 1;

      for ( int i=n-1; i>=1; i-- ) begin

         automatic logic next_occ_bvec = !start && occ_bvec[i-1];
         automatic logic match = next_occ_bvec && element == elements[i-1];

         n_match += match;

         if ( match ) match_pos = i;

         elements[i] <= elements[i-1];
         occ_bvec[i] <= next_occ_bvec;

         uniq_at[i] <= match ? n : uniq_at[i-1];
         uniq_bvec[i] <= !next_occ_bvec || !match && i >= uniq_at[i-1];

      end

      elements[0] <= element;
      occ_bvec[0] <= 1;
      uniq_at[0] <= n - match_pos;
      uniq_bvec[0] <= match_pos == n;
   end

endmodule
```


Staple This Side

☑ Show inferred hardware for `n=4`.

☑ Do not confuse ports with parameters. ☑ Do not confuse elaboration-time computation with computation hardware.

Solution appears below. Registers and gates shown in gray can be eliminated by optimization.



Problem 2: [25 pts] Illustrated on the facing page is a diagram showing inferred hardware similar to the `word_count` module from last year's final exam. An important difference is that it is shown for `n_avg_of=n`, not the specific value of 4. Assume that `n` is a power of 2.

☑ In terms of `n`, `wl`, `wn`, and `v` show simple-model arrival times at each wire and ☑ show a critical path.

☑ Account for cascaded ripple units ☑ constant inputs, and ☑ remember that `n` can be any power of 2, not neccesarily 4.

The arrival times are shown in purple, and a critical path is shown as a dashed red line.

To solve this correctly one must pay attention to bit widths. Most bit widths are directly marked, such as w_l for `lword` and w_n for `nwords`, but one had to infer that `lsum` is $w_l + v$ bits wide. (In the original problem `lsum` was $w_l + 4$ bits wide.) That width has been directly added to the solution but in the unsolved problem it must be inferred from the `wl+v-1:v` (in the original problem `wl+3:2`) bit slice used at the input to the `lavg` mux.

Constant inputs affect the delay of adders, multiplexors, and the comparison unit. The delay of a w-bit constant-input adder is $\lceil w - 1 \rceil \, \mathrm{u_t}$. To see how, set carry-in to zero in the constant adder shown in 2023 Midterm Exam Problem 2. Because n is a power of 2 the least-significant $\lg n$ bits of n are zero. Since $v = \lg n$ (see the top of the module illustration) the comparison $n_{\mathrm{words}} \geq n$ can skip the first v bits and so the delay of the ripple comparison circuit is $w_n - v - 1$ (a recursive construction would have a logarithmic delay).

The subtractor and adder computing a result for `lsum` are cascaded and so their respective delays are not added. To make it a bit more tricky one input to the adder is w_l bits and the other is $w_l + v$ bits. So the adder consists of approximately w_l BFAs, with a carry delay of 2 each, and v BHAs with a carry delay of 1 each, plus the final sum XOR with a total delay of $2(w_l + 1) + v$. The adder can start when the least-significant bit arrives at $2\lg(n) + 4$ so the arrival time of the result at the output of the adder (input to the register) is $2\lg(n) + 2(w_l + 1) + v + 4 = 3v + 2(w_l + 1) + 4$. If the problem were solved for an `lsum` width of $w_l + 4$ the arrival time would be $2\lg(n) + 2(w_l + 1) + 4 + 4 = 2v + 2(w_l + 1) + 8$.

The illustrated critical path assumes $2\lg(n) + 2(w_l + 1) + v + 4 \geq w_n + 1$. The registers also have delays, which are not shown. The delays are $6 \, \mathrm{u_t}$ for registers without enables and $8 \, \mathrm{u_t}$ for registers with enables. The time unit, $\mathrm{u_t}$, has been omitted in the diagram to save space and in this discussion to save time.

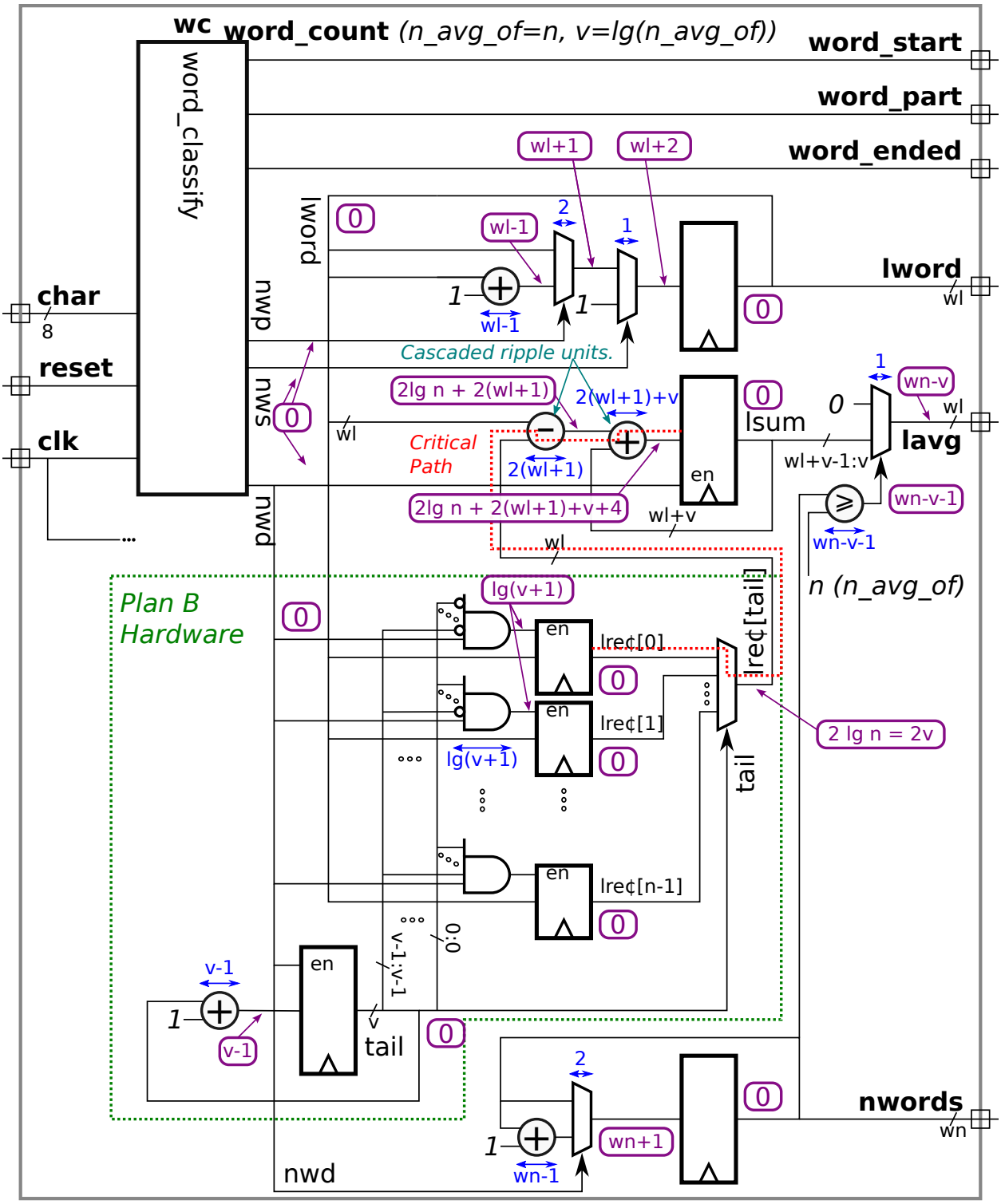
☑ In terms of `n`, `wl`, `wn`, and `v` compute the simple-model cost of the Plan B hardware, assuming `n` is a power of 2. ☑ Account for constant inputs.

The costs appear in the table below. For the costs of the constant-input adder see 2023 Midterm Exam Problem 2.

Item	Count	Each	Total
Constant Input v-bit Adder	1	$4v$	$4v$
v-bit Register with Enable (`tail`)	1	$10v$	$10v$
$(v+1)$-Input AND Gates	n	v	nv
w_l-Bit Registers with Enable (`lrec`)	n	$10w_l$	$10nw_l$
w_l-Bit, n-Input Multiplexor	1	$3w_l(n-1)$	$3w_l(n-1)$

Grading Note: A common mistake was using the wrong number of bits for the registers and multiplexor.

Staple This Side



Problem 3: [27 pts] The two modules below look for a match of input `target` in an `n`-element array `elts` but only check elements 0 to `i_limit-1`. Output `n_match` is the number of matching elements and `match_i` is the lowest `i` for which `elts[i]==target` and `i<i_limit`, or `n` if there is no match. (These modules could be used in the `uniq_vector` module.) Module `fmatch_comb` is complete and works correctly.

(*a*) Module `fmatch_rec` has some code for a recursive implementation. Complete it so that it performs the same calculation as `fmatch_comb`.

☑ Complete `fmatch_rec` so that it computes the same values as `fmatch_comb`.

☑ Don't forget to show the bit ranges of `elts` in the connections to the recursive instantiations.

Solution appears on the next page.

```
module fmatch_comb
  #( int n = 22, w = 12, wn = $clog2(n+1) )
   ( output logic [wn-1:0] n_match, match_i,
     input uwire  [w-1:0] elts[n-1:0], target,   input uwire [wn-1:0] i_limit );

   // Do not modify this module. It is correct.
   always_comb begin

      n_match = 0;
      match_i = n;

      for ( int i=n-1; i>=0; i-- ) if (  i < i_limit  &&  elts[i] == target  ) begin
         n_match++;
         match_i = i;
      end

   end

endmodule
```


Staple This Side

```
module fmatch_rec
  #( int n = 22, w = 12, wn = $clog2(n+1) )
   ( output uwire [wn-1:0] n_match, match_i,
     input uwire  [w-1:0] elts[n-1:0], target,   input uwire [wn-1:0] i_limit );

   if ( n == 1 ) begin

      // Do not modify the n==1 code, it works.
      uwire match = i_limit != 0 && elts[0] == target;
      assign n_match = match;
      assign match_i = match ? 0 : 1;

   end else begin

      // SOLUTION: Split elements between recursive instances.
      //
      localparam int nlo = n/2;
      localparam int nhi = n - nlo;

      localparam int wnr = $clog2(nhi+1);
      uwire [wnr-1:0] nm_lo, nm_hi, mi_lo, mi_hi;

      // SOLUTION: For lo instance make sure i_limit does not overflow.
      //
      uwire [wnr-1:0] il_lo = i_limit <= nlo ? i_limit : nlo;

      // SOLUTION: For hi instance adjust i_limit because elts[nlo]
      // for this instance is the same as elts[0] for the lo instance.
      //
      uwire [wnr-1:0] il_hi = i_limit <= nlo ? 0 : i_limit - nlo;

      // SOLUTION: Connect low nlo elements of elts to lo instance and
      // SOLUTION: remaining elements of elts to hi instance.
      //
      fmatch_rec #(nlo,w,wnr) ilo( nm_lo, mi_lo, elts[nlo-1:0], target, il_lo );
      fmatch_rec #(nhi,w,wnr) ihi( nm_hi, mi_hi, elts[n-1:nlo], target, il_hi );

      // SOLUTION: Add number of matches found by each recursive instance.
      //
      assign n_match = nm_lo + nm_hi;

      // SOLUTION: If lo instance did not find a match ( mi_lo == nlo )
      // then use match_i from high instance, adjusting the value. Otherwise
      // use match_i from lo instance.
      //
      assign match_i = mi_lo == nlo ? mi_hi + nlo : mi_lo;

   end

endmodule
```

Problem 4: [20 pts] Answer each question below.

(*a*) Consider two technology targets, FabFab A1000, an ASIC, and LÜTeq FXL9000, an FPGA. Floating-point multipliers are available on the A1000 and the FXL9000 targets.

☑ On one of these targets a design can have as many multipliers as will fit on the chip. Which target is it? ☑ Explain.

An ASIC. The synthesized design uses components from the ASIC technology target, in the same way the printed page of a text document uses characters from a font. If a text document consists of the letter L repeated 1600 times, that's no problem because there is no way to run out of the letter L. So, if the synthesis program placed 1000 copies of the FP multiplier from the FabFab A1000 technology kit all over the ASIC chip, that's not a problem.

☑ On the other target there is a fixed number of FP multipliers, say 5. Does that mean a design that needs 7 FP multipliers can't use the target? ☑ Explain. ☑ The number of needed multipliers can't be reduced.

An FPGA consists of a fixed number of components that can be configured and interconnected to implement the synthesized design. Usually those components are simple look-up tables that can be configured to perform simple logic functions (such as AND). But an FPGA might have a small number of larger components, such as FP functional units.

The FXL9000 just has 5 FP multipliers, but it presumably has plenty of other logic. That other logic can be configured to implement a FP multiplier. So, the design would use the 5 FP multipliers on the FPGA plus use the other logic to "make" another 2 FP multipliers.

(*b*) The output of the module below will be `lt=1` for inputs `a=100, b=40, amt=20`, indicating that $100+40 < 20$, which is wrong of course. It works correctly for `a=100, b=40, amt=5`, meaning the output is `lt=0`.

```
module less_than( output uwire lt, input uwire [6:0] a, b, amt );
   assign lt = a + b < amt;
endmodule
```

☑ Why is the output wrong?

Since `a`, `b`, and `amt` are all 7 bits the addition will be performed at a precision of 7 bits, and so there will be an overflow. (The largest value is $2^7 = 127$.) Rather than using the correct sum, $100 + 40 = 140 = 10001100_2$, the Verilog simulator and the synthesized hardware will use the lower 7 bits, $0001100_2 = 12_{10}$.

☑ What is the largest value of `amt` for which the module output is correct when the other inputs are `a=100, b=40`?

The output will be correct for values of `amt` less than or equal to 12 (because both $140 < 12$ and $12 < 12$ are false), so the largest value for `amt` is 12.

Staple This Side

Staple This Side

(*c*) The `hw` output of the module below is supposed to be set to the number of `1`s in input `vec` at the positive edge of the clock. Due to a beginner's Verilog error it does not work.

```
module pop #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] hw,  input uwire [n-1:0] vec,  input uwire clk );

   always_ff @( posedge clk ) begin

      hw <= 0;
      for ( int i=0; i<n; i++ ) hw <= hw + vec[i];

   end
endmodule
```

☑ Describe the problem. ☑ Describe how it's possible that `hw` can be greater than `n` with this error. ☑ Fix the problem.

Short Answer: The code uses non-blocking assignments but because it refers to values of `hw` that are assigned in the `always_ff` block it should have used blocking assignments. The computed value of `hw` may be large because the value of `hw` used in `hw + vec[i]` will be from the last `i` iteration in *previous clock cycle*, and so the initialization to zero will not be seen. The problem is fixed by changing `hw <=` to `hw =` in both places.

Longer Explanation: The value assigned to `hw` in the non-blocking assignments will not be visible in the execution of the `always_ff` in which they were assigned. Therefore the `hw + vec[i]` expression will use the value of `hw` from *before* the positive edge. (To be precise, since the last time the simulator event queue processed the non-blocking assignment (NBA) region.) So, each of the `n` iterations uses the same `hw` value. That value would be the value assigned in the last (`i=n-1`) iteration from the previous clock cycle. The other `n-1` assignments in the loop and the `hw <= 0` assignment never make it into `hw`. Since `hw` is never set to `0` its value will only increase, and so will exceed `n`.

(*d*) Consider the population module below.

```
module pop_comb #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] hw,  input uwire [n-1:0] vec );
   always_comb begin
      hw = 0;
      for ( int i=0; i<n; i++ ) hw = hw + vec[i];
   end
endmodule
```

☑ The loop above is procedural. Re-write the module below so that it is a generate loop. The array `s` should come in handy.

Solution appears below. Note that a wire must be provided, `s[i]`, for each iteration to handle the intermediate sum.

```
module pop_comb #( int n = 5, wn = $clog2(n+1) )
   ( output uwire [wn-1:0] hw,  input uwire [n-1:0] vec );

 // SOLUTION
   uwire [wn-1:0] s [n-1:0];
   assign s[0] = vec[0];
   for ( genvar i=1; i<n; i++ )
     assign s[i] = s[i-1] + vec[i];
   assign hw = s[n-1];
endmodule
```

Staple This Side