This document contains assignments given in LSU EE 4755 over many semesters. It was automatically generated and so some solutions (and even some assignments) are possibly missing. At the top of each page of each assignment is a link to the original assignment. Those who want to print an assignment might follow that link. All assignments and public solutions are available at `https://www.ece.lsu.edu/ee4755/prev.html`.

Contents

1 Fall 2024

Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 23 October 2024, 11:30-12:20 CDT

Staple This Side

Problem 1	_____	(22 pts)
Problem 2	_____	(18 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

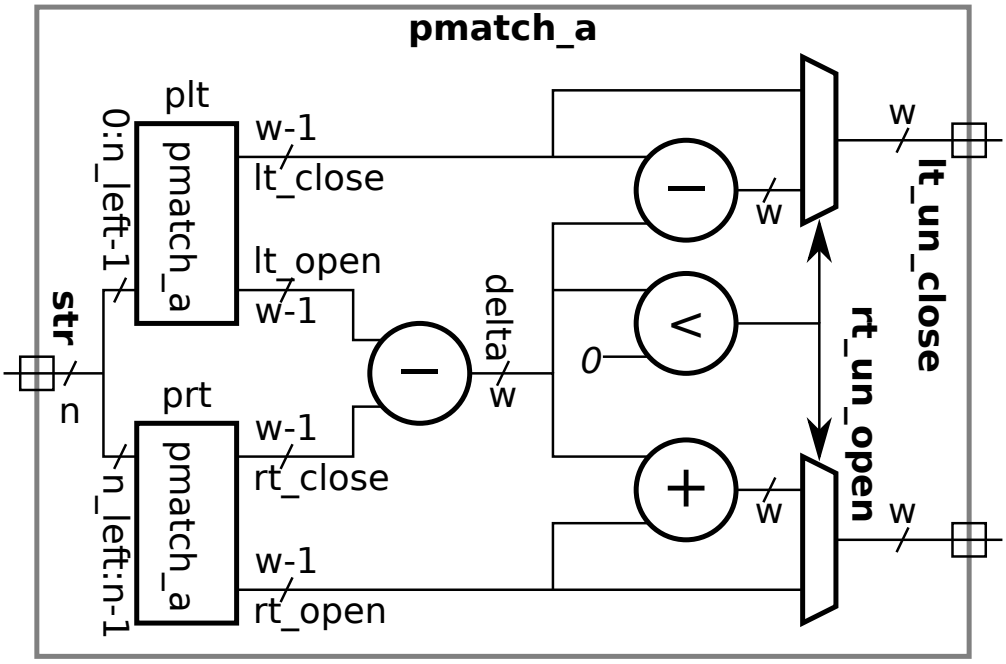
Problem 1: [22 pts] Below is the Homework 3 Problem 1 solution with some object names shortened.

```
typedef enum logic [3:0] {Char_Blank=0, Char_Dot=1, Char_Open=2, Char_Close=3} Char;
module pmatch_a #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] lt_un_close, rt_un_open,   input uwire [3:0] str[0:n-1] );
   if ( n == 1 ) begin
      assign lt_un_close = str[0] == Char_Close ? 1 : 0;
      assign rt_un_open =  str[0] == Char_Open  ? 1 : 0;
   end else begin
      localparam int n_left = n/2;
      localparam int n_right = n - n_left;
      localparam int wl = $clog2(n_left+1),  wr = $clog2(n_right+1);
      uwire [wl-1:0] lt_close, lt_open;
      uwire [wr-1:0] rt_close, rt_open;
      pmatch_a #(n_left,  wl) plt( lt_close, lt_open, str[0:n_left-1] );
      pmatch_a #(n_right, wr) prt( rt_close, rt_open, str[n_left:n-1] );

      uwire logic signed [wn-1:0] delta = lt_open - rt_close;
      assign lt_un_close = delta <  0 ? lt_close - delta : lt_close;
      assign rt_un_open =  delta >= 0 ? rt_open  + delta : rt_open;
   end
endmodule
```

(*a*) Show the hardware that will be inferred for the base case. Show hardware after optimization taking into account constants.

☐ Show inferred hardware for base (`n==1`) case of the module above. ☐ Show input and output ports.

☐ Optimize taking into account ☐ constant values of all kinds. ☐ Don't miss the `Char` definition above the module. ☐ Don't show a comparison unit such as `==`, instead show the gates from which it was made and ☐ optimize them, ☐ taking into account the number of bits on each output port.



Appearing above is hardware that will be inferred for the non-base case.

(*b*) Compute the cost of the hardware at this level (ignore what's inside `plt` and `prt`) based on the simple model using the bit widths from the diagram, such as `w-1`.

☐ Show the cost of each component *except for hardware inside of* `plt` *and* `prt`.

☐ Be sure to show the cost of the optimized comparison unit!

(*c*) Compute the delay through the module **starting from** launch points `lt_close`, `lt_open`, `rt_close`, and `rt_open`. The capture points are `lt_un_close` and `rt_un_open`. Use the bit widths from the diagram, such as `w-1`.

☐ Show the arrival time at each wire from launch to capture.

☐ Take into account cascaded ripple units and ☐ and the optimized comparison unit.

Problem 2: [18 pts] Appearing below is an alternative solution to Homework 3 Problem 1. The only difference is the last few lines.

```
module pmatch_b #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] lt_un_close, rt_un_open,
     input uwire [3:0] str[0:n-1] );
   if ( n == 1 ) begin
      assign lt_un_close = str[0] == Char_Close ? 1 : 0;
      assign rt_un_open = str[0] == Char_Open ? 1 : 0;
   end else begin
      localparam int n_left = n/2;
      localparam int n_right = n - n_left;
      localparam int wl = $clog2(n_left+1),  wr = $clog2(n_right+1);
      uwire [wl-1:0] lt_close, lt_open;
      uwire [wr-1:0] rt_close, rt_open;
      pmatch_b #(n_left,  wl) plt( lt_close, lt_open, str[0:n_left-1] );
      pmatch_b #(n_right, wr) prt( rt_close, rt_open, str[n_left:n-1] );
      uwire logic signed [wn-1:0] delta = lt_open - rt_close;
      // Lines above are identical to pmatch_a.

      uwire [wn-1:0] delta_n = delta <  0 ? delta : 0;
      uwire [wn-1:0] delta_p = delta >= 0 ? delta : 0;
      assign lt_un_close = lt_close - delta_n;
      assign rt_un_open  = rt_open  + delta_p;
   end
endmodule
```

(*a*) Show the hardware that will be inferred for `pmatch_b`. For your convenience the hardware for `pmatch_a` is shown in the upper right. *Note: In the original exam the condition for* `delta_n` *was* `delta <= 0` *and the condition for* `delta_p` *was* `delta > 0`. *Though the hardware computed the correct result, the comparison would have been more expensive since it would have had to check for a zero condition, not just negative.*

☐ Show inferred hardware on the facing page.

(*b*) Compute the simple-model cost of the hardware.

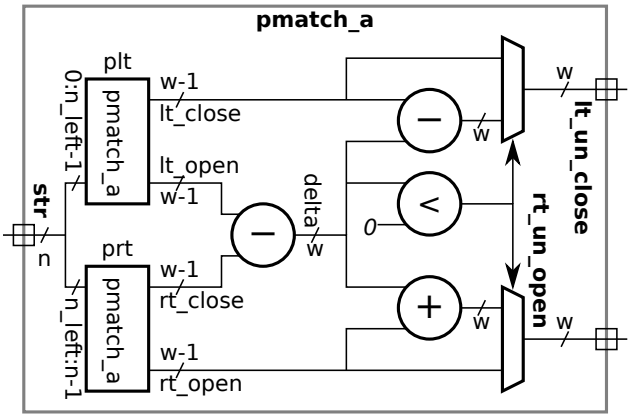
☐ Write *same* next to components that cost the same as corresponding components in `pmatch_a` and ☐ compute the cost of other components ☐ after optimization.

(*c*) Compare the critical path lengths.

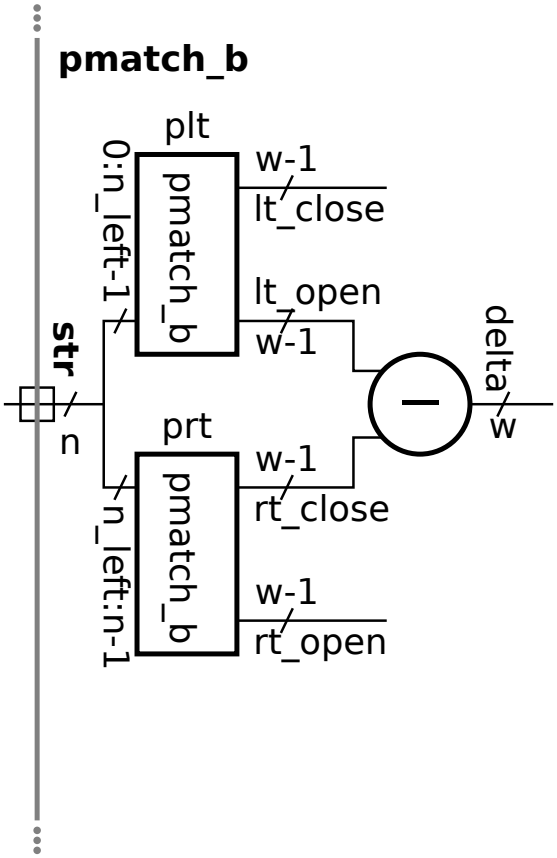
☐ Will the critical path in `pmatch_a` be much different than the one in `pmatch_b`? ☐ Explain.

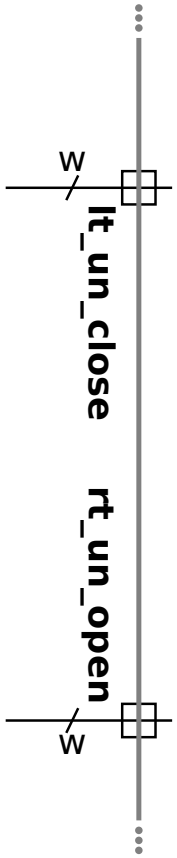
Staple This Side

Staple This Side



Staple This Side





Staple This Side

Problem 3: [20 pts] Appearing below are some of the `dot` modules from the solution to Homework 1. On the facing page is incomplete module `dotn`. Complete `dotn` so that it describes hardware that computes the dot product of `n`-element vectors recursively, where `n` is the parameter. That is, `dotn` must instantiate `dotn` and should instantiate `mult` and `add` where needed.

```
module mult #( int w = 5 ) ( output uwire [w-1:0] p, input uwire [w-1:0] a, b );
   assign p = a * b;
endmodule

module add #( int w = 5 ) ( output uwire [w-1:0] s, input uwire [w-1:0] a, b );
   assign s = a + b;
endmodule

module dot2 #( int w = 5 )
   ( output uwire [w-1:0] dp,    input uwire [w-1:0] a[1:0], b[1:0] );
   // Computes dp = a[0] * b[0] + a[1] * b[1];
   uwire [w-1:0] p0, p1;
   mult #(w) m0(p0, a[0], b[0] );
   mult #(w) m1(p1, a[1], b[1] );
   add #(w) ad(dp, p0, p1 );
endmodule

module dot3 #( int w = 5 )
   ( output uwire [w-1:0] dp,   input uwire [w-1:0] a[2:0], b[2:0] );
   // Computes   dp = a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
   uwire [w-1:0] p0, p2;
   dot2 #(w) d0( p0, a[1:0], b[1:0] );
   mult #(w) m2( p2, a[2], b[2] );
   add #(w) a2(dp, p0, p2 );
endmodule
```

Staple This Side

Staple This Side

☐ Complete `dotn` so that it describes tree-structured hardware computing an `n`-element dot product. The tree depth should be $\lceil \lg n \rceil$.

☐ Instantiate `mult` for multiplication and `add` for addition, and of course `dotn` for a dot product of a smaller vector.

☐ To keep things easy all wires are `w` bits.


```
module dotn
  #( int w = 5, n = 4 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] a[n-1:0], b[n-1:0] );
```

Staple This Side

```
endmodule
```

Problem 4: [10 pts] Appearing below is the logarithmic shifter presented in class, followed by a version that's supposed to be better (but isn't). The hoped-for improvement is due to instantiating the exact number of multiplexors (`muxw2`) needed, rather than enough for the maximum shift amount.

```
module shift_right_logarithmic #( int w = 16, lgw = $clog2(w) )
   ( output uwire [w-1:0] shifted,
     input uwire [w-1:0] un,   input uwire [lgw-1:0] amt );
   // This module is correct.
   uwire [w-1:0] s[lgw:-1];
   assign s[-1] = un;
   for ( genvar i=0; i<lgw; i++ )
     muxw2 #(w) st( s[i],  amt[i],  s[i-1],  s[i-1] >> ( 1 << i )  );
   assign shifted = s[lgw-1];
endmodule

module shift_right_logarithmic_better_maybe #( int w = 16, lgw = $clog2(w) )
   ( output uwire [w-1:0] shifted,
     input uwire [w-1:0] un,  input uwire [lgw-1:0] amt );
   uwire [w-1:0] s[lgw:-1];
   assign s[-1] = un;

   // Use exactly the number of stages needed!!!
   uwire [lgw-1:0] lg_amt;              // LINE ADDED
   my_clog2 #(lgw) mc( lg_amt, amt );  // LINE ADDED. Set lg_amt = $clog2(amt) = ⌈lg amt⌉;

   for ( genvar i=0; i<lg_amt; i++ )   // LINE DIFFERS
     muxw2 #(w) st( s[i],  amt[i],  s[i-1],  s[i-1] >> ( 1 << i )  );
   assign shifted = s[lg_amt-1];       // LINE DIFFERS
endmodule
```

☐ Why won't the Verilog above compile?

☐ Is it possible to fix the Verilog error in such a way that cost is lower with smaller shift amounts? ☐ Explain.

☐ Is it possible to fix the Verilog error in such a way that the delay reported by a synthesis program is lower? ☐ Explain.

☐ Is it possible to fix the Verilog error in such a way that the delay actually is lower? ☐ Explain.

Staple This Side

Staple This Side

Problem 5: [30 pts] Answer the following Verilog questions.

(*a*) The module below uses multidimensional arrays.

```
module mda( input uwire [2:1] c [5:1],    input uwire [7:1][2:1] a [5:1][3:1] );

   //  ☐ Add dimension(s) to the declaration of e so that the assignment is correct.
   //
   uwire            e            = c;


   //  ☐ Add dimension(s) to the declaration of b so that the assignment is correct.
   //
   uwire            b            = a[1][1][1];


   logic g [7:0];
   logic [7:0] h;

   initial begin
//  ☐ Which is correct,
       ◯ only the assignment to g,   ◯ only the assignment to h, or   ◯ both are correct.
    ☐ Explain.

      g[1] = h[1];
      h[1] = g[1];
   end

endmodule
```

☐ What is the size of c, in bits? ☐ What is the size of a, in bits?

Staple This Side

Staple This Side

(*b*) In the module below indicate whether each code fragment is correct.

```
module kinds #( logic [31:0] pg = 123 )
   ( output uwire [31:0] o0,  input uwire [31:0] ik );

   //  ☐ Is the line below correct?   ◯ Yes   ◯ No   ☐ If not, explain.

   localparam logic [31:0] z02 = pg + 4755;

   //  ☐ Is the line below correct?   ◯ Yes   ◯ No   ☐ If not, explain.

   localparam logic [31:0] z03 = ik;

   //  ☐ Are the lines below correct?   ◯ Yes   ◯ No   ☐ If not, explain.

   localparam logic [31:0] z04;
   assign z04 = pg;

   //  ☐ Are the lines below correct?   ◯ Yes   ◯ No   ☐ If not, explain.

   uwire [31:0] z10 = pg;
   assign z10 = ik;

   //  ☐ Is the line below correct?   ◯ Yes   ◯ No   ☐ If not, explain.

   uwire [31:0] z13 = ik;

endmodule
```

Staple This Side

Staple This Side

(*c*) When we run a synthesis program we specify a delay target. In class we often synthesize twice, once with a delay target of 100 ns and a second time with a target of 0.1 ns. What is the harm in specifying a delay target lower (faster) than one needs? Isn't faster better?

☐ Harm in setting delay target too low is:

(*d*) A 32-bit signed integer, say i, is converted into a 32-bit IEEE 754 floating-point format (8-bit exponent, 23-bit significand) and then back into a 32-bit integer, j.

☐ Is it guaranteed that $i = j$ for all $-2^{31} \leq i < 2^{31}$? ☐ Explain based on the FP representation.

Name _____ *Formatted For Two-Sided Printing*

Staple This Side

Digital Design using HDLs
LSU EE 4755
Final Examination

Thursday, 12 December 2024 15:00-17:00 CST

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

Alias _____

Good Luck!

Staple This Side

Problem 1: [20 pts] Module `dot_seq_4` on the facing page is to compute the dot product of two vectors, with four elements of each vector arriving at each cycle. Like `dot_seq_2` from Homework 5, inputs `first` and `last` mark the beginning and end of each vector. Unlike `dot_seq_2` there are no ID ports. But, `dot_seq_4` does has have a `dim` output. When `dp` is set to a dot product, `dim` should be set to the dimension (number of elements) of the vectors used for that product. For example, vectors that arrive over two cycles will have a dimension of $2 \times 4 = 8$.

The unsolved module lacks code related to `in_id`, but is otherwise similar to `dot_seq_2`, including the fact that it only uses the two elements per cycle. *Note: In the original exam* `dim` *was called* `len` *and was called the length. The problem wording dealt with the number of elements and contained nothing to suggest that* `len` *was to be set to the norm 2 of the vector.*

☐ Modify `dot_seq_4` so that it computes the correct dot product using all four elements arriving each cycle.

☐ As with `dot_seq_2`, the critical path should contain at most one arithmetic operation per cycle.

☐ Modify `dot_seq_4` so that output `dim` is set to the dimension (number of elements) of the vector whose dot product appears on `dp`.

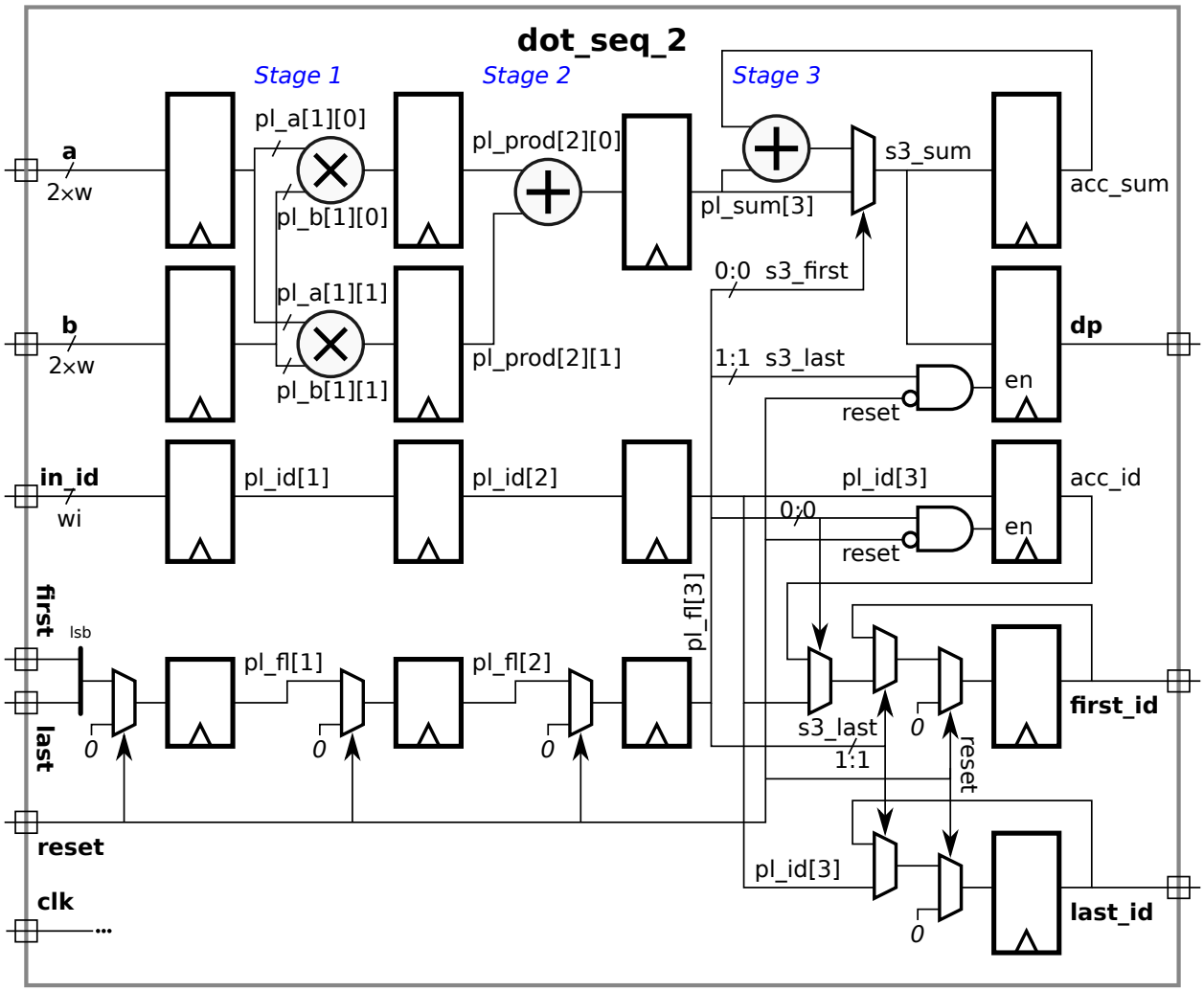
```
module dot_seq_4 #( int w = 5, wi = 4 )
   ( output logic [w-1:0] dp,              output logic [wi-1:0] dim,
     input uwire [w-1:0] a[4], b[4],       input uwire reset, first, last, clk );

   logic [w-1:0] pl_a[1:1][4], pl_b[1:1][4];  // Arriving vector elements.
   logic [w-1:0] pl_prod[2:2][2];             // Vector products.
   logic [w-1:0] pl_sum[3:3];                 // Dot prod of 2-element segment.
   logic [1:0] pl_fl[1:3];                    // The first and last signals.


   logic [w-1:0] acc_sum;


   always_ff @( posedge clk ) begin

      // Stage 0
      pl_a[1] <= a;  // This copies both elements of a.
      pl_b[1] <= b;
      pl_fl[1] <= reset ? 2'b0 : {last,first};


      // Stage 1
      for ( int i=0; i<2; i++ ) pl_prod[2][i] <= pl_a[1][i] * pl_b[1][i];
      //
      pl_fl[2] <= reset ? 2'd0 : pl_fl[1];


      // Stage 2
      pl_sum[3] <= pl_prod[2][0] + pl_prod[2][1];
      //
      pl_fl[3] <= reset ? 2'h0 : pl_fl[2];


      // Stage 3
      begin
         automatic logic s3_first = pl_fl[3][0],  s3_last = pl_fl[3][1];
         automatic logic [w-1:0] s3_sum = s3_first ? pl_sum[3] : pl_sum[3] + acc_sum;


         acc_sum <= s3_sum;

         if ( !reset && s3_last ) dp <= s3_sum;


      end
   end
endmodule
```

Problem 2: [20 pts] Appearing below is the solution to Homework 6, the `dot_seq_2` module. For this problem assume that the delay of a w-bit adder is $2w\,\mathrm{u_t}$ and that the delay of a multiplier with w-bit inputs and w-bit output is $4w\,\mathrm{u_t}$.

☐ Show the arrival time at each wire, especially at the capture points. ☐ Be sure to account for constant inputs.

☐ Label the critical path. ☐ Indicate the length of the critical path.

☐ Letting $1\,\mathrm{u_t} = 1\,\mathrm{ns}$ and based on the answers above, what is the maximum possible clock frequency in GHz? ☐ Your answer should be in terms of w. ☐ State any assumptions.



Staple This Side

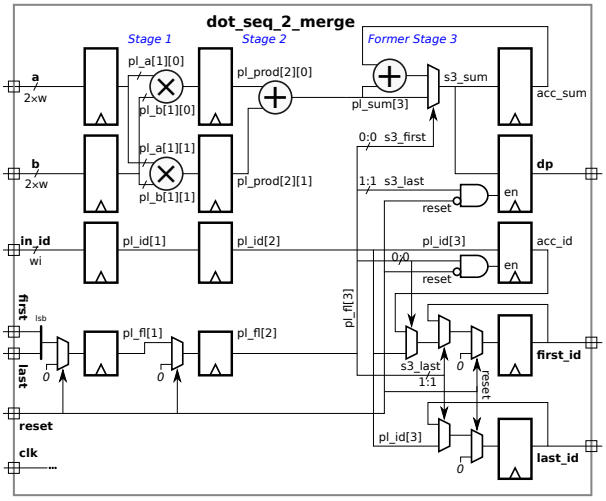
Staple This Side

☐ For 2-element vectors what is the ☐ latency and ☐ throughput of `dot_seq_2` (from the previous page)? State any assumptions.

☐ For 8-element vectors what is the ☐ latency and ☐ throughput of `dot_seq_2` (from the previous page)? State any assumptions.

Module `dot_seq_2_merge`, to the right, was constructed by merging stages 2 and 3.

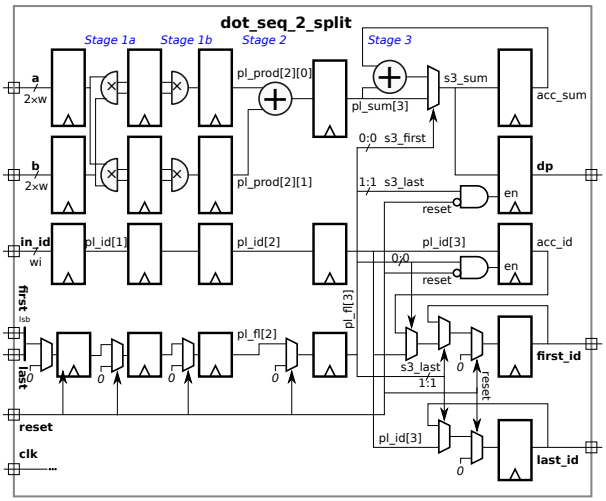
☐ What is the critical path length of `dot_seq_2_merge`?



☐ For two-element vectors what are the ☐ latency and ☐ throughput of `dot_seq_2_merge`?

Module `dot_seq_2_split` was constructed by splitting each multiplier into two parts of delay $2w\,\mathrm{u_t}$ each, and putting the two parts into separate stages.

☐ What is the critical path length of `dot_seq_2_split`?



☐ For two-element vectors what are the ☐ latency and ☐ throughput of `dot_seq_2_split`?

Problem 3: [20 pts] Appearing on the facing page is a recursively described module that finds the minimum of `n` items.

(*a*) Let t_2 denote the delay of the `min_2` module (not shown).

☐ In terms of n and t_2 what is the delay of the unmodified (two recursive instances) `min_t`?

(*b*) Modify `min_t` so that in the recursive case it instantiates three (instead of two) recursive instances.

☐ Modify `min_t` so that it instantiates three recursive instances and other changes needed for the three instances.

☐ Use only `min_2` to compare items. There is no `min_3`, don't try to write one.

☐ Don't assume that `n` is a power of 3.

(*c*) Let t_2 denote the delay of the `min_2` module.

☐ In terms of n and t_2 what is the delay of the modified (three recursive instances) `min_t`?

☐ Compared to two recursive instances, does having three recursive instances in `min_t` ◯ *reduce delay*, ◯ *increase delay*, or ◯ *makes little or no difference to delay*? ☐ Justify mathematically, in terms of n or using a specific number.

```
module min_t #( int w = 4, int n = 8 )
   ( output uwire [w-1:0] elt_min,    input uwire [w-1:0] elts [ n-1:0 ] );



   if ( n == 1 ) begin

      assign elt_min = elts[0];









   end else begin

      localparam int n_hi = n / 2;

      localparam int n_lo = n - n_hi;









      uwire [w-1:0] elt_lo, elt_hi;









      min_t #(w,n_hi) mhi( elt_lo, elts[  n-1  :  n_lo  ]);

      min_t #(w,n_lo) mlo( elt_hi, elts[  n_lo-1  :  0   ]);



      min_2 #(w) m2( elt_min, elt_lo, elt_hi );









   end
endmodule
```

Problem 4: [20 pts] Show the hardware that will be synthesized for the module below for `wa=3, wb=2` (three iterations of the loop).

```
module rmatch #( int wa = 3, wb = 2, wm = $clog2(wa+1) )
   ( output logic [wm-1:0] m, pos,
     input uwire [wm-1:0] m0,   input uwire [wa-1:0] a,  input uwire [wb-1:0] b,
     input uwire clk );
   logic [wa-1:0] a_cpy, as;
   logic [wm-1:0] m0_cpy;
   logic [wb-1:0] b_cpy;
   always_ff @( posedge clk ) begin
      a_cpy <= a;
      b_cpy <= b;
      m0_cpy <= m0;
      as = a_cpy;
      pos = 0;
      for ( int i=0; i<wa; i++ ) begin
         if ( as[wb-1:0] == b_cpy ) begin
            if ( m == m0_cpy ) pos = i;
            m++;
         end
         as = { as[wa-2:0], as[wa-1] };
      end
   end
endmodule
```

☐ Show synthesized hardware. ☐ Show module ports. ☒ Do not confused elaboration-time computation with hardware.

Staple This Side

Staple This Side

Problem 5: [20 pts] Answer each question below.

(*a*) Show the values of the variables where indicated.

```
module short;
   int a, b, c, d, e, f;
   initial begin
      a = 1;  b = 2;  c = 3;  d = 4;  e = 5;  f = 6;

      a <= b;
      b <= a;

      e <= c + 10;
      f <= e + 100;

      c = d;
      d = a;

      // ☐ a =     ☐ b =     ☐ c =     ☐ d =     ☐ e =     ☐ f =

      #1;

      // ☐ a =     ☐ b =     ☐ c =     ☐ d =     ☐ e =     ☐ f =

   end
endmodule
```

(*b*) The module below computes `x` and `y` correctly, but one of them is computed in a way that has at least two advantages in avoiding human coding errors.

```
module to_err_is_human( output logic [7:0] x, y,   input uwire [7:0] a, b, c );

   always @( a or b or c ) begin
      x = a + b + c;
   end

   always_comb begin
      y = a + b + c;
   end
endmodule
```

☐ Which is computed in the preferred way ◯ `x` or ◯ `y`?

☐ Describe two human coding errors that can be avoided using the preferred method.

(*c*) The first module below, `dot_product_correct`, is indeed correct. The other two won't even compile. All are supposed to compute a dot product of either floating-point or integer elements.

☐ Explain the major error on the on the three lines commented `Explain compile error.` ☐ There should be three different errors, if a line has more than one error pick one not shared with the other two.

☐ Assume that FP hardware has larger delay than integer arithmetic hardware. If a module is used with `use_fp` set to 0 does that mean the module is faster? ☐ Explain, including the interpretation of the word "faster."

☐ Assume that FP hardware has higher cost than hardware computing integer arithmetic. If a module is used with `use_fp` set to 0 does that mean the module cost less? ☐ Explain.

```
module dot_product_correct #( int n = 7, w_sig = 20, w_exp = 8, w = w_sig + w_exp + 1 )
   ( output logic [w-1:0] dp,
     input uwire [w-1:0] a[n], b[n],  input uwire use_fp,  input uwire clk );
   logic [w-1:0] acc;
   uwire [w-1:0] akk[n:0];
   assign akk[0] = 0;
   for ( genvar i=0; i<n; i++ )
     fp_madd #(w_sig,w_exp) ma( akk[i+1], a[i], b[i], akk[i] );

   always_ff @( posedge clk ) begin
      acc = 0;
      for ( int i=0; i<n; i++ ) acc += a[i] * b[i];
      dp <= use_fp ? akk[n] : acc;
   end
endmodule

module dot_product_b #( int n = 7, w_sig = 20, w_exp = 8, w = w_sig + w_exp + 1 )
   ( output logic [w-1:0] dp,
     input uwire [w-1:0] a[n], b[n],  input uwire use_fp,  input uwire clk );
   logic [w-1:0] acc;
   uwire [w-1:0] akk;
   assign akk = 0;
   for ( genvar i=0; i<n; i++ )
     fp_madd #(w_sig,w_exp) ma( akk, a[i], b[i], akk );       //  Explain compile error.

   always_ff @( posedge clk )
      if ( use_fp ) begin
         #1;      // <----------------------------------------   Explain compile error.
         dp <= akk;
      end else begin
         acc = 0;
         for ( int i=0; i<n; i++ ) acc += a[i] * b[i];
         dp <= acc;
      end
endmodule

module dot_product_c #( int n = 7, w_sig = 20, w_exp = 8, w = w_sig + w_exp + 1 )
   ( output logic [w-1:0] dp,
     input uwire [w-1:0] a[n], b[n],  input uwire use_fp,  input uwire clk );
   logic [w-1:0] acc;

   always_ff @( posedge clk ) begin
      acc = 0;
      for ( int i=0; i<n; i++ )
        if ( use_fp )
          fp_madd #(w_sig,w_exp) ma( acc, a[i], b[i], acc );  //  Explain compile error.
        else
          acc += a[i] * b[i];
      dp <= acc;
   end
endmodule
```

2 Fall 2023

Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Friday, 27 October 2023, 11:30-12:20 CDT

Staple This Side

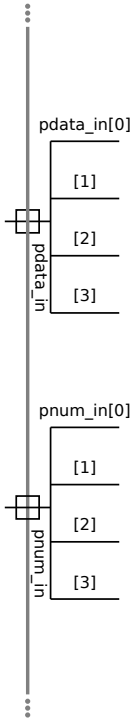
Problem 1	_____	(30 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(30 pts)
Problem 4	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [30 pts] Appearing below is the permutation module from the solution to Homework 3. Using the illustration of the ports show the inferred hardware for an instantiation with `n=4`. Show the `n=4` instantiation but not what is inside the `n=3` recursive instantiation.

```
module perm
  #( int w = 8, n = 20, wd = $clog2(n) )
   ( output uwire [w-1:0] pdata_out[n],    output uwire [wd-1:0] pnum_out[n],
     output uwire carry_out,
     input uwire [w-1:0] pdata_in[n],      input uwire [wd-1:0] pnum_in[n] );

   if ( n == 1 ) begin

      assign pdata_out[0] = pdata_in[0];
      assign carry_out = 1;
      assign pnum_out[0] = 0;

   end else begin

      uwire [wd-1:0] pos = n - 1 - pnum_in[n-1];
      assign pdata_out[n-1] = pdata_in[pos];
      uwire [w-1:0] prdata_in[n-1];
      for ( genvar i=0; i<n-1; i++ )
        assign prdata_in[i] =  i < pos  ?  pdata_in[i]  :  pdata_in[i+1];

      uwire co;
      perm #(w,n-1,wd) rp( pdata_out[0:n-2], pnum_out[0:n-2], co,
                           prdata_in, pnum_in[0:n-2] );

      uwire [wd-1:0] dnext = pnum_in[n-1] + co;
      assign carry_out = dnext >= n;
      assign pnum_out[n-1] = carry_out ? 0 : dnext;

   end

endmodule
```

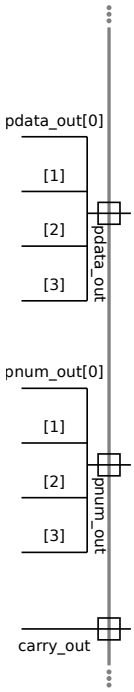


Staple This Side

Staple This Side

☐ Show inferred hardware for `n=4`. Be sure to use ☐ the illustrated module ports and to show ☐ the recursively instantiated module (but not its contents).

☐ Show hardware, ☐ do not confuse elaboration-time computation with computation hardware.



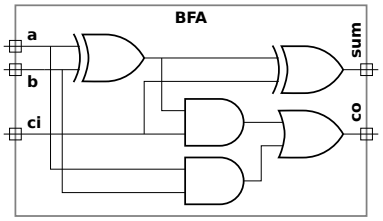
Problem 2: [25 pts] A ripple adder to compute $a + b$ is to be used in situations where a is a constant.

(a) Find the cost and delay of a BFA with input a constant (for use in the ripple adder). A BFA is shown for your convenience.

☐ Show the BFA(s) optimized for input a constant.

☐ Use a truth table to find optimizations not revealed by constant pushing: in a correct solution the delay does not depend upon `a`.

☐ Show simple-model cost of this(these) module(s) and ☐ show simple-model delay(s) of this(these) module(s).



(b) On the facing page show the optimized hardware, cost, LSB delay, and MSB delay of a w-bit ripple adder for computing $a + b + c_{\text{in}}$, where c_{in} is a carry-in bit (`cin` in the diagram) and a is a constant. (☐ See the check box items for details.) Use the illustration on the facing page as a starting point.

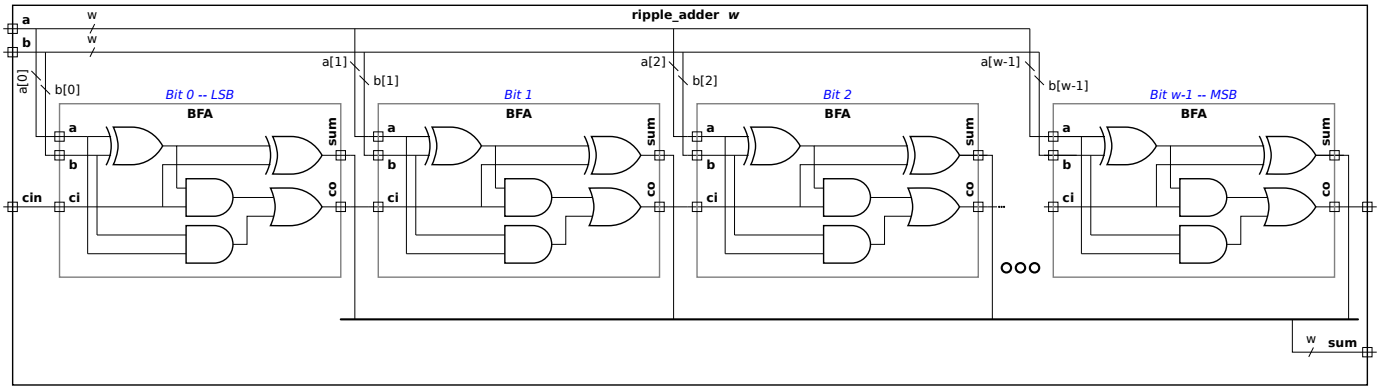
☐ Show the hardware optimized for a constant `a` and a non-constant `cin`.

☐ Compute the simple-model cost of this hardware in terms of w.

☐ Compute the simple-model delay of the LSB of the sum.

☐ Compute the simple-model delay of the MSB of the sum in terms of w and ☐ show the critical path.

☐ Don't forget that `a` is a constant.



(*c*) If `cin` were removed (or set to zero) the cost and delay of the optimized adder would depend on `a`. Explain why, and illustrate with the example of `a=2`.

☐ How are cost and delay dependent on `a` when `cin` removed? ☐ Explain using the example `a=2`.

Staple This Side

Problem 3: [30 pts] Answer the following Verilog questions.

(*a*) The module below makes extensive use of multidimensional arrays.

```
module mda( input uwire [2:1] c [5:1],    input uwire [7:1][2:1] a [5:1][3:1] );

   //  ☐ Add dimension(s) to the declaration of e so that the assignment is correct.
   //
   uwire            e            = c[1];


   //  ☐ Add dimension(s) to the declaration of b so that the assignment is correct.
   //
   uwire            b            = a[1];


   logic g [7:0];
   logic [7:0] h;

   initial begin
//  ☐ Which is correct,  ◯ the assignment to g  or  ◯ the assignment to h.  ☐ Explain.
      g = 1;
      h = 1;
   end

endmodule
```

☐ What is the size of c, in bits? ☐ What is the size of a, in bits?

(*b*) The module below does not compile.

```
module more_stuff #( int n = 20 ) ( output uwire [31:0] sum, input uwire [31:0] a [ n ] );
   logic [31:0] acc;
   always_comb begin
      acc = a[0];
      for ( int i=1; i<n; i++ )
        my_fixed_adder a1(acc, acc, a[i] );
   end
   assign sum = acc;
endmodule
```

☐ Describe the major problem. ☐ **DO NOT** try to fix the problem.

Staple This Side

Staple This Side

(*c*) The module below is supposed to set $x = a^2 + b^2$.

```
module wrong_way( output logic [31:0] x, input uwire [15:0] a, b );
   logic [31:0] asq;
   uwire [31:0] bsq = b * b;

   initial asq = a * a;
   always_comb x = asq + bsq;

endmodule
```

Staple This Side

☐ Explain the problem. ☐ Using sample inputs show the expected output and the actual output.

☐ Fix the problem.

(*d*) The module below does not compile.

```
module my_adder( output uwire [31:0] s, input uwire [31:0] a, b );
   always_comb s = a + b;
endmodule
```

☐ Why won't module above compile? ☐ Fix problem by changing declarations.

(*e*) The module below compiles but does not provide the expected outputs, $p_a = a^2$, $p_b = b^2$, and $p = a^2 + b^2$.

```
module incorrect_way( output logic [31:0] pa,pb,p, input uwire [15:0] a, b );
   wire [31:0] sq;
   assign sq = a * a;
   always_comb pa = sq;
   assign sq = b * b;
   always_comb pb = sq;
   always_comb p = pa + pb;
endmodule
```

Staple This Side

☐ What will be the values of outputs `pa`, `pb`, and `p`?

☐ Describe the problem. ☐ Fix it.

Problem 4: [15 pts] Answer each question below.

(*a*) A company has two teams, *A* (very good) and *C* (slackers) working on modules and a testbench for an important product. Describe the following consequences:

☐ The *A* team works on the modules and the *C* team works on the testbench. A possible bad outcome is:

☐ The *A* team works on the testbench and the *C* team works on the modules. A possible bad outcome is:

(*b*) In typical use when running simulation a testbench generates inputs for a module-under-test and the outputs are checked by the testbench to see whether they are correct. After running synthesis we learn how fast the module is. If simulation is computing the module outputs why can't it tell us how fast the module is?

☐ Synthesis can provide timing information and simulation can't because:

(*c*) A gadget can be built using an ASIC or an FPGA. Describe which is more appropriate for each situation below.

☐ The gadget must be working within a month. ◯ *ASIC* or ◯ *FPGA*. ☐ Explain.

☐ Per-gadget cost must be under $1000. Only ten will be made. ◯ *ASIC* or ◯ *FPGA*. ☐ Explain.

☐ Per-gadget cost must be under $100. Ten thousand will be made. ◯ *ASIC* or ◯ *FPGA*. ☐ Explain.

Name _____ *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 _____

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:

```
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.

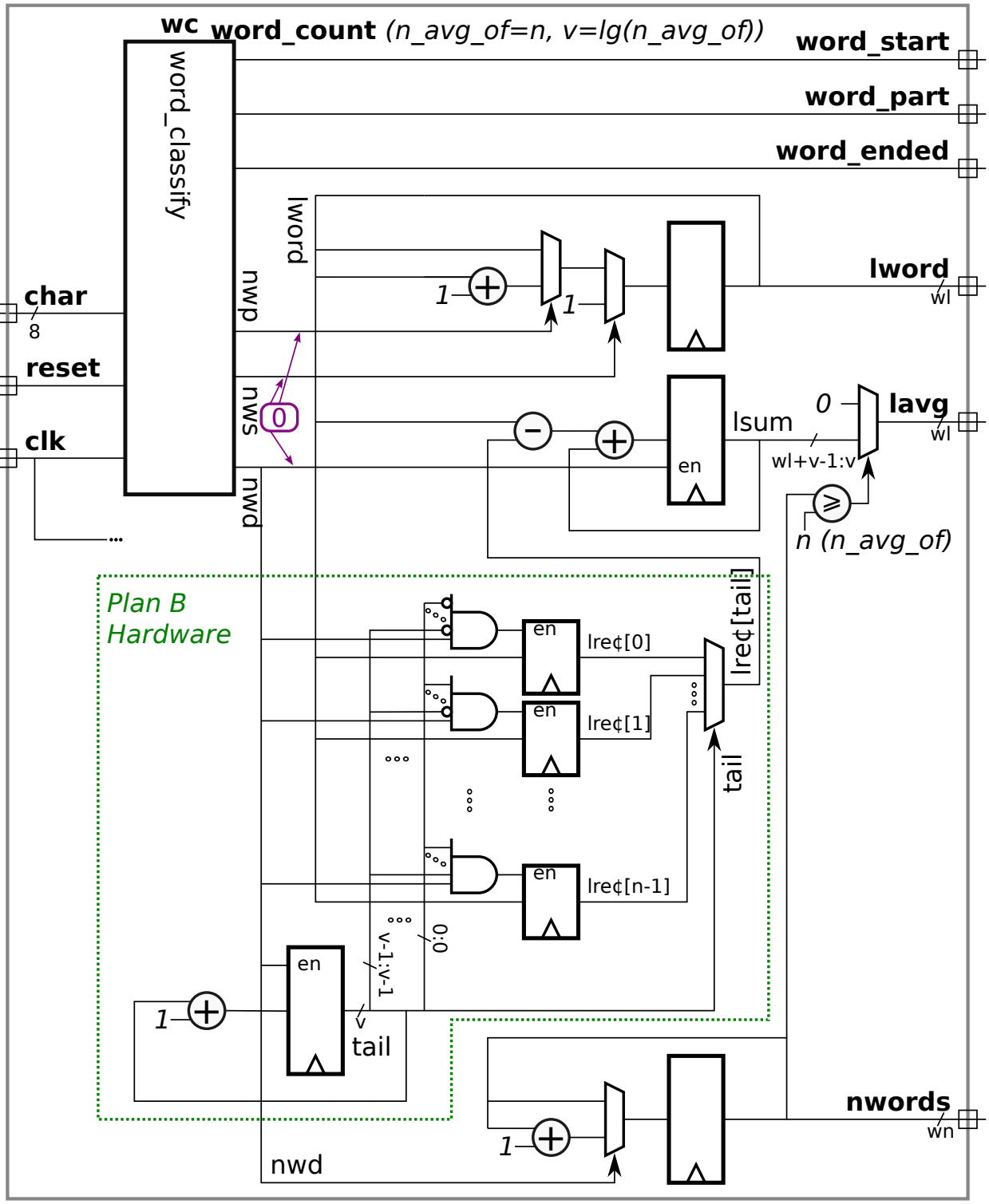
Staple This Side

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.

☐ 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.



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.


```
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

      localparam int nlo =


      localparam int nhi =


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

      uwire [wnr-1:0] il_lo =


      uwire [wnr-1:0] il_hi =


      fmatch_rec #(nlo,w,wnr) ilo( nm_lo, mi_lo, elts[        ], target, il_lo );

      //                  ☐ Show elts' bit ranges ↑↑↓↓

      fmatch_rec #(nhi,w,wnr) ihi( nm_hi, mi_hi, elts[        ], target, il_hi )


      assign n_match =



      assign match_i =



   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.

☐ 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.

(*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?

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

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.

(*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.

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

   uwire [wn-1:0] s [n-1:0];









endmodule
```

3 Fall 2022

Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 19 October 2022, 11:30-12:20 CDT

Staple This Side

Problem 1	_____	(25 pts)
Problem 2	_____	(31 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(12 pts)
Problem 5	_____	(12 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [25 pts] Answer the following multiplexor questions.

(*a*) Complete module `mux4` so that it implements a 4-input multiplexor using instantiations of the 2-input multiplexor shown below. Do not use procedural code.

☐ Complete `mux4` so that it implements a 4-input multiplexor ☐ using `mux2` instantiations.

☐ Do not use procedural code. ☐ Do not change the ports or default parameters of `mux4` or `mux2`.

☐ Don't forget to declare any objects that are used.

```
module mux4
  #( int w = 3 )
   ( output uwire [w-1:0] x,
     input uwire [1:0] s,      input uwire [w-1:0] a0, a1, a2, a3 );
```

```
endmodule

module mux2
  #( int w = 6 )
   ( output uwire [w-1:0] x,
     input uwire s,   input uwire [w-1:0] a0, a1 );
   assign x = s ? a1 : a0;
endmodule
```


Staple This Side

(*b*) Module `mux2_bad` only works for `w=1`. Describe the problem and show the correct mux output and the output of `mux2_bad` for `w=4`, `s=0`, `a0=2`, and `a1=4`.

```
module mux2_bad
  #( int w = 4 )
   ( output uwire [w-1:0] x,
     input uwire s,   input uwire [w-1:0] a0, a1 );
   assign x = !s && a0  ||  s && a1;
endmodule
```

☐ In `mux2` (a correct mux) when `w=4`, `s=0`, `a0=2`, and `a1=4`, ☐ output `x=`

☐ In `mux2_bad` when `w=4`, `s=0`, `a0=2`, and `a1=4`, ☐ output `x=`

☐ Explain the problem when `w` is not `1`.

(*c*) Complete module `mux2_1r` below so that it recursively implements a 2-input w-bit mux. All that remains to be done is completing the connections to the two recursive instances, `m1` and `mr`.

```
module mux2_1r
  #( int w = 5 )
   ( output uwire [w-1:0] x,
     input uwire s,   input uwire [w-1:0] a0, a1 );

   if ( w == 1 ) begin
     assign x = !s && a0  ||  s && a1;
   end else begin


      mux2_1r #(1)    m1(


      mux2_1r #(w-1)  mr(


   end

endmodule
```

Problem 2: [31 pts] The `val` output of `atoi_it_m_to_l` is the value of the radix-`r` ASCII-represented number appearing at its input, `str`, and output `nd` is the number of digits. Unlike the Homework 2 Problem 2 module, this module starts at the most-significant digit rather than the least-significant digit.

```
module atoi_it_m_to_l
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output logic [wv-1:0] val,
     output logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   uwire [wv-1:0] vali[n:0];
   uwire is_digit[n:0];
   uwire [wd-1:0] ndi[n:0];
   assign is_digit[n] = 0;
   assign ndi[n] = 0;
   assign vali[n] = 0;
   assign nd = ndi[0];
   assign val = vali[0];

   localparam int wcv = $clog2(r);

   for ( genvar i=n-1; i>=0; i-- ) begin

      // Find Value of Digit i
      uwire [wcv-1:0] vald;
      atoi1 #(r,wcv) a( vald, is_digit[i], str[i] );

      // Multiply (scale) the accumulated sum.
      uwire [wv-1:0] valns;
      mult_by_c #( .w_in(wv), .c(r), .w_out(wv) ) mc( valns, vali[i+1] );

      // Update accumulated value.
      assign vali[i] = is_digit[i] ? valns + vald : 0;
      // Update number of digits.
      assign ndi[i] = !is_digit[i] ? 0 : is_digit[i+1] ? ndi[i+1] : i + 1;

   end

endmodule
```

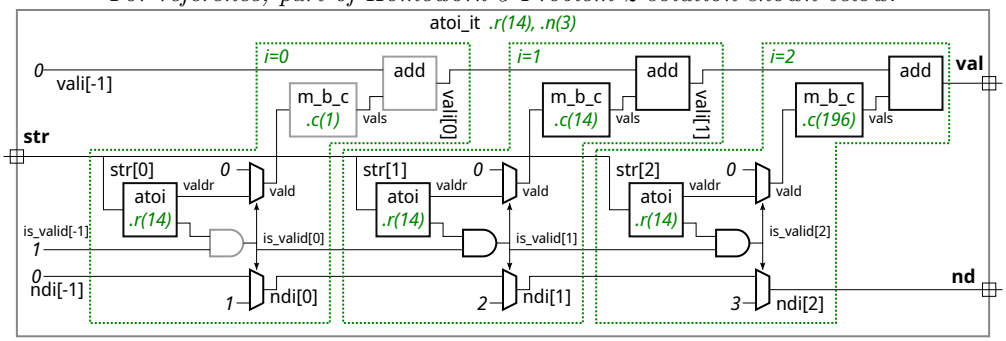
(*a*) Describe how the behavior of the module would change if the loop direction were changed as shown below, but no other changes were made.

```
   for ( genvar i=0; i<n; i++ ) begin
```

☐ Change in behavior with ascending loop:

(*b*) On the next (facing) page show the hardware that will be inferred for an instantiation of `atoi_it_m_to_l` (descending loop version) with `n=3` and `r=10`. Show each instantiation of `atoi1` and `mult_by_c` as a box, do not show their contents. The inferred hardware for `atoi_it` is shown for reference.

For reference, part of Homework 3 Problem 2 solution shown below.



For reference, part of Homework 3 Problem 2 solution shown above.

- ☐ Show inferred hardware for `atoi_it_m_to_l` for `n=3` and `r=10`.
- ☐ Show the hardware inferred for the operators, such as `&&` and `?:`.
- ☐ Do not confuse parameters and ports and omit hardware that does not belong, such as "hardware" to compute values needed at elaboration time.

Staple This Side

Staple This Side

(*c*) Module `atoi_m_to_l` will only show the value of numbers that are right-aligned in `str`, otherwise the value will be shown as zero. For example, for input `str="__123"` the output would be `val=123` and `nd=3`, but for input `str="_123_"` the output would be `val=0` (because the rightmost character is not a digit). Modify the module so the `val` output is the value of the number regardless of its location. If there is more than one number, say `str="__12_345_"`, show the value of the rightmost number, 345 in this case.

- ☐ Modify so that `val` and `nd` are for numbers whether or not they are right-aligned.
- ☐ Do not use procedural code.
- ☐ Avoid costly or slow solutions.
- ☐ A correct solution only requires a few changes.

Staple This Side

```
module atoi_it_m_to_l
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output logic [wv-1:0] val,
     output logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   uwire [wv-1:0] vali[n:0];
   uwire is_digit[n:0];
   uwire [wd-1:0] ndi[n:0];
   assign is_digit[n] = 0;
   assign ndi[n] = 0;
   assign vali[n] = 0;
   assign nd = ndi[0];
   assign val = vali[0];

   localparam int wcv = $clog2(r);

   for ( genvar i=n-1; i>=0; i-- ) begin

      // Find Value of Digit i
      uwire [wcv-1:0] vald;
      atoi1 #(r,wcv) a( vald, is_digit[i], str[i] );









      // Multiply (scale) the accumulated sum.
      uwire [wv-1:0] valns;
      mult_by_c #( .w_in(wv), .c(r), .w_out(wv) ) mc( valns, vali[i+1] );









      // Update accumulated value.
      assign vali[i] = is_digit[i] ? valns + vald : 0;



      // Update number of digits.
      assign ndi[i] = !is_digit[i] ? 0 : is_digit[i+1] ? ndi[i+1] : i + 1;



   end
endmodule
```

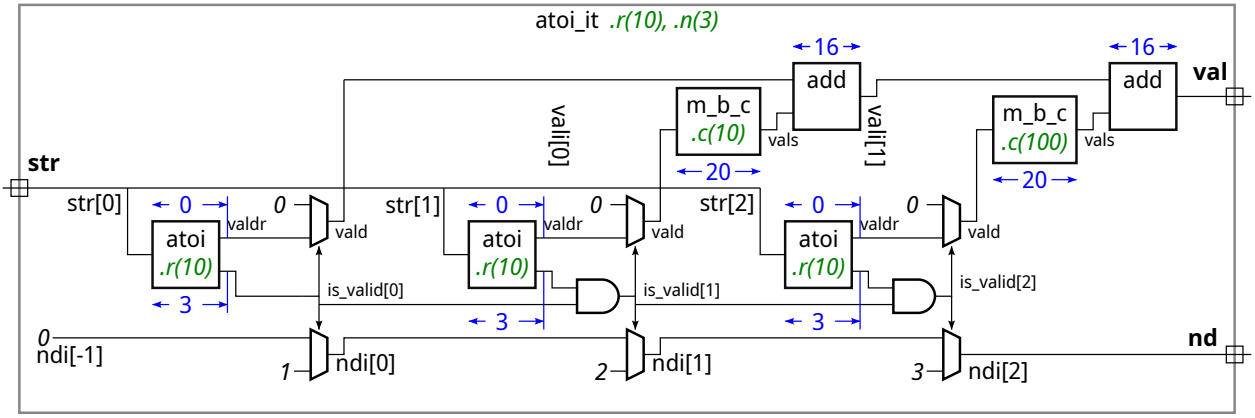
Problem 3: [20 pts] Illustrated below is the hardware for one of the `atoi` modules from Homework 3. The delays for the `add`, `atoi1`, and `mult_by_c` modules are shown in blue. For `atoi` the delay of the value (`valdr`) output is zero and the delay of the `is_digit` (lower) output is 3.

(*a*) Based on the illustrated delays and using the simple model find the delay at each output, `val` and `nd`, and show the critical path to each.

☐ Use the simple model and indicated delays to find the delay at outputs `val` and `nd`.

☐ Show the critical path to both `val` and `nd`.

☐ Take into account constant values.



(*b*) Modify the design to reduce the delay at `val` by moving multiplexors. The modification is simple though will increase cost. Show your modification either on the diagram or in the Verilog code below.

☐ Modify to reduce the delay at `val` by moving multiplexors.

☐ Do not change what the module does.

Staple This Side

```
module atoi_it
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output logic [wv-1:0] val,  output logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   uwire [wv-1:0] vali[n-1:-1];
   uwire is_valid[n-1:-1];
   uwire [wd-1:0] ndi[n-1:-1];
   assign is_valid[-1] = 1;
   assign ndi[-1] = 0;
   assign vali[-1] = 0;
   assign nd = ndi[n-1];
   assign val = vali[n-1];
   localparam int wcv = $clog2(r);

   for ( genvar i=0; i<n; i++ ) begin
      uwire [wcv-1:0] valdr;
      uwire is_digit;
      atoi1 #(r,wcv) a( valdr, is_digit, str[i] );  // Find Value of Digit i

      // Determine if this digit continues a sequence of valid digits.
      //
      assign is_valid[i] = is_digit && is_valid[i-1];



      // Replace value with zero if str[i] is not a digit, or if the
      // string of valid digits has already ended.
      //
      uwire [wcv-1:0] vald = is_valid[i] ? valdr : 0;



      // Multiply (scale) the digit value based on its position in the number.
      //
      uwire [wv-1:0] vals;
      mult_by_c #( .w_in(wcv), .c(r**i), .w_out(wv) ) mc( vals, vald );



      // Add the scaled digit to the value accumulated so far.
      //
      add #(wv) a1( vali[i], vali[i-1], vals );



      // Update the number of digits so far.
      //
      assign ndi[i] = is_valid[i] ? i+1 : ndi[i-1];

   end
endmodule
```

Problem 4: [12 pts] Answer each question below.

(*a*) The module below will not compile because of the way the module connections are declared. Fix the problem by changing the declarations.

☐ Change declaration to fix problem.

```
module yucx2
  #( int w = 5 )
   ( output uwire [w-1:0] x,
     input uwire [1:0] s,
     input uwire [w-1:0] a0, a1 );

   always_comb begin
      x = a0;
      if ( s != 0 ) x = a1;
   end

endmodule
```

(*b*) The `mv` output of `findmax` is supposed to be set to the value of the largest of the three inputs. Assuming it compiles and simulates, it still won't work. Identify the problem.

☐ Why won't `mv` be set to the maximum of `a0`, `a1`, `a2`?

☐ Provide an example that illustrates the incorrect behavior.

```
module findmax
  #( int w = 5 )
   ( output logic [w-1:0] mv,
     input uwire [w-1:0] a0, a1, a2 );

   initial mv = 0;
   always_comb if ( mv < a0 ) mv = a0;
   always_comb if ( mv < a1 ) mv = a1;
   always_comb if ( mv < a2 ) mv = a2;

endmodule
```

Staple This Side

Staple This Side

Problem 5: [12 pts] Answer each question below.

(*a*) Type `logic` is an example of a four-state type. Name those four states and describe what the non-numeric ones are used for.

☐ Name the four `logic` states.

☐ Describe what the non-numeric ones signify.

(*b*) Most synthesis programs will not synthesize a module that includes a delay, such as the one below. Why not?

```
module madd
  #( int w )
   ( output logic [w-1:0] w,
     input uwire [w-1:0] a, b, c );
   always_comb begin
      w = a * b;
      #5; // Allow enough time for multiplication to finish.
      w = w + a;
   end
endmodule
```

☐ Why isn't a delay synthesizeable?

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

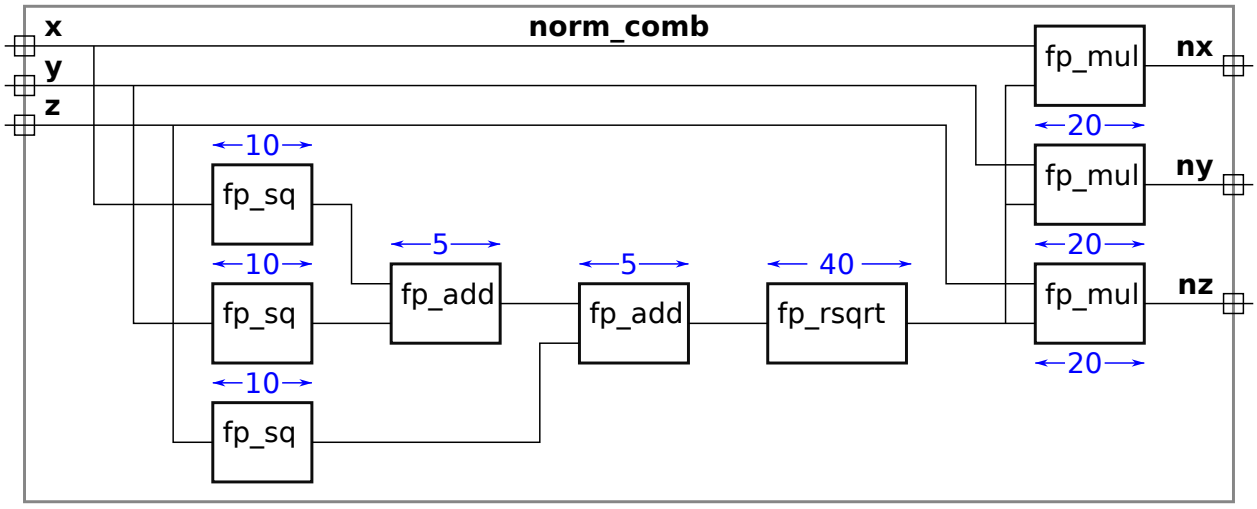
Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

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:

(*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:

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.

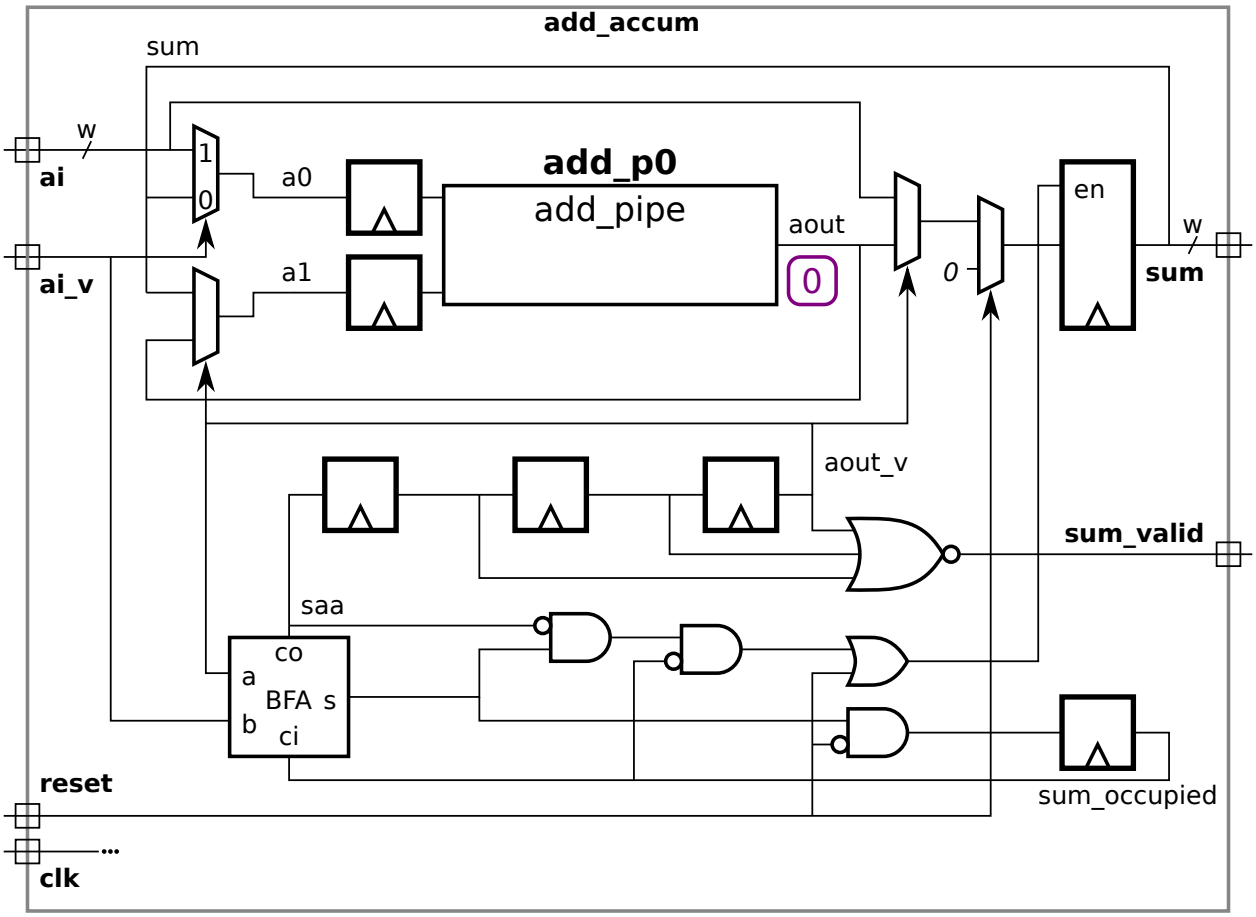
```
module norm_comb_n #( int w = 32, int n = 8 )
   ( output uwire [w-1:0] u[n],   input uwire [w-1:0] v[n] );

   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.

(*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.

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.

(*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.

4 Fall 2021

Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 27 October 2021, 11:30-12:20 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(10 pts)
Exam Total	_____	(100 pts)

Staple This Side

Alias _____

☣ $V(\text{mRNA}) \quad \Rightarrow \quad R_e < 1$

Good Luck!

Problem 1: [25 pts] Appearing in this problem are two variations on hardware that selects one of four inputs, `i`, based on the position of the least-significant `1` in a 4-bit quantity, `fmt`. This is similar to the hardware needed in the solution to Homework 2, except that here `i[3]` can be selected.

```
module nn_sparse #( int w = 20 )
   ( output logic [w-1:0] o, input uwire [w-1:0] i[4], input uwire [3:0] fmt );
```

(*a*) Show the hardware that will be inferred for `is0` and show that hardware after optimization.

```
   uwire [w-1:0] is0 = fmt[0] ? i[0] : fmt[1] ? i[1] : fmt[2] ? i[2] : i[3];
```


☐ Show inferred hardware.

☐ Show optimized hardware. Hardware can be re-arranged to reduce delay.

☐ Use only basic logic gates and multiplexors.

(*b*) Compute the cost and delay of the optimized hardware for `is0` in terms of w. (That's w, not its default value.)

☐ In terms of w cost is:

☐ In terms of w delay is:

(*c*) Appearing below is an alternative design. Net `is0b` will have the same value as `is0`. Show the hardware below before and after optimization. For `isi0` do not show multiplexors after optimization. For `is0b` use two-input multiplexors (as many as needed).

```
uwire [1:0] isi0 = fmt[0] ? 0 : fmt[1] ? 1 : fmt[2] ? 2 : 3;
uwire [w-1:0] is0b = i[isi0];
```

☐ Show inferred hardware.

☐ Show optimized hardware, optimize to reduce delay.

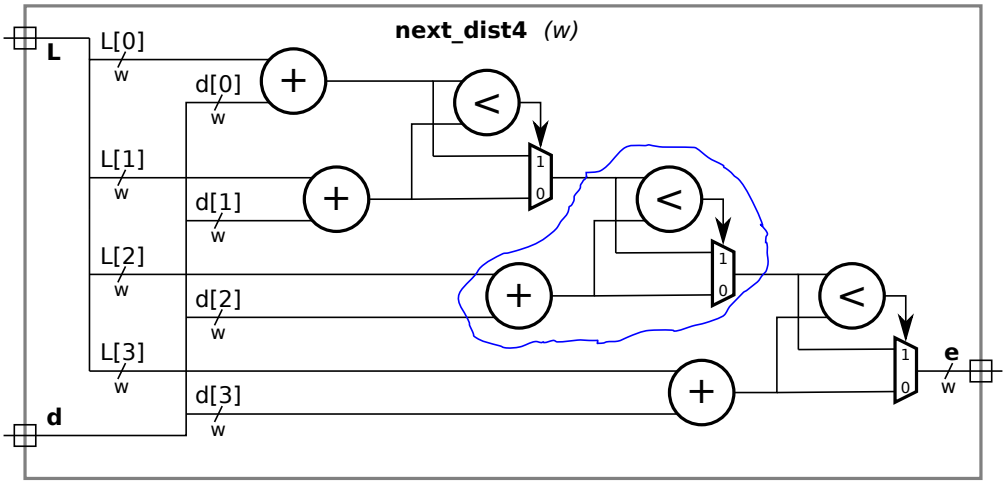
☐ Use basic logic gates and ☐ no muxen for `isi0` and ☐ two-input muxen (plus other logic) for `is0b`.

(*d*) Compute the cost and delay of the optimized hardware (from the previous part) in terms of w. (That's w, not its default value.)

☐ In terms of w cost is:

☐ In terms of w delay is:

Problem 2: [30 pts] The `next_dist4` hardware illustrated below consists of several duplicated pieces of hardware, one of which is circled. Call the circled hardware an *ami* unit (for add-minimum).



(*a*) Compute the cost and delay of the module using the simple model, and show the critical path on the illustration. Assume that the adder and comparison units are based on ripple adders.

☐ Cost in terms of w:

☐ Show critical path. ☐ Delay in terms of w:

☐ Account for any cascading ripple units.

Staple This Side

(*b*) Appearing below are two incomplete modules, one is an `ami` module the other is the `next_dist4` module. Complete these modules to match the diagram using as many `ami` modules as needed. The `ami` module can use procedural or implicit structural code. The `next_dist4` module must instantiate and use `ami` modules but can contain procedural or implicit structural code.

☐ Complete the `ami` module so that it matches the circled hardware.

☐ Complete the `next_dist4` module using as many `ami` modules as needed.

☐ Don't forget to ☐ declare any intermediate objects that are used.

☐ Noting that there are four adders and the width of each wire is `w`, ☐ declare and use parameters appropriately.

```
module ami









endmodule


module next_dist4 #( int w = 12 )
   ( output uwire [w-1:0] e,
     input uwire [w-1:0] L[4], d[4] );









endmodule
```

(*c*) Incomplete module `next_dist` is a generalization of `next_dist4` to `n` elements per input. The module includes a generate loop. Use that loop to instantiate `ami` modules so that it performs the correct calculation. Keep the loop simple, don't try to fix the delay problem.

☐ Complete module, taking advantage of the generate loop.

☐ Be sure to instantiate `ami` modules, ☐ connect the first `ami` correctly, ☐ and don't leave `e` unconnected.

```
module next_dist #( int n = 20, w = 12 )
   ( output uwire [w-1:0] e,
     input uwire [w-1:0] L[n], input uwire [w-1:0] d[n] );

   localparam logic [w-1:0] mv = ~w'(0); // Can use as input to first ami.

   uwire [w-1:0]

   for ( genvar i=0; i<n; i++ ) begin

   end

endmodule
```

Problem 3: [10 pts] Consider the `with_assign` module below.

```
module with_assign #( int w = 10 )
                    ( output uwire [w-1:0] g, input uwire [w-1:0] b, c );

   uwire [w-1:0] a, f;

   assign g = f | c;  // Line 1
   assign f = a * c;  // Line 2
   assign a = b + c;  // Line 3

endmodule
```

(*a*) Why might the module confuse or annoy humans?

☐ `with_assign` could be confusing because:

(*b*) The module makes extra work for simulators too. Suppose that the input values to `with_assign`, `b` and `c`, change at $t = 10$. About how many times will each line below execute in a worst-case scenario? *The following sentence was not in the original exam:* Use sensitivity lists to justify your answer.

☐ About how many times does each line execute? ☐ Explain with sensitivity lists.

(*c*) Complete the `sans_assign` routine below so that it does the same thing as `with_assign` but is less confusing and less work for simulators.

☐ Complete routine below. (Yes, it's easy but not trivial.)

```
module sans_assign #( int w = 10 )
                    ( output uwire [w-1:0] g, input uwire [w-1:0] b, c );

   uwire [w-1:0] a, f;

   always_comb begin




   end
endmodule
```

☐ Why does `sans_assign` make less work for the simulator than `with_assign`? Explain using sensitivity lists.

Problem 4: [10 pts] Appearing below is an ordinary multiplier, followed by a multiplier that is naïvely designed to take advantage of special cases (first operand is 0 or 1), followed by a module that instantiates both.

```
module mult #( int w = 32 )
              ( output logic [w-1:0] p, input uwire [w-1:0] a, b );
   always_comb p = a * b;
endmodule

module mult_1a #( int w = 32 )
                ( output logic [w-1:0] p, input uwire [w-1:0] a, b );

   always_comb begin
      if ( a == 0 ) p = 0;
      else if ( a == 1 ) p = b;
      else p = a * b;
   end

endmodule

module nm #( int w = 32, logic [w-1:0] c = 12 )
           ( output uwire [w-1:0] prods[4], input uwire [w-1:0] a[4], b[4] );
   mult #(w)      m1 ( prods[0], a[0], b[0] );
   mult #(w)      m2 ( prods[1], c,    b[1] );
   mult_1a #(w)   ma1( prods[2], a[0], b[0] );
   mult_1a #(w)   ma2( prods[3], c,    b[1] );
endmodule
```

☐ Explain why `m1` will be faster (lower delay) than `ma1`, even when possible values of `a[0]` include `0`, `1`, and other values. Assume good synthesis programs.

☐ How will the cost and performance of `m2` and `ma2` compare (to each other) using good synthesis programs? That is, ☐ which should be chosen when delay is the only concern and, ☐ which of the two should be chosen when cost is the only concern. The answer should not depend on any particular value of `c`.

Staple This Side

Staple This Side

Problem 5: [15 pts] Answer the following questions about Verilog syntax and semantics.

(*a*) Appearing below are four variations on a multiplier with a constant input. Most have errors that would prevent them from compiling. For each indicate whether there is an error, and if so, what the error is and a minimal fix.

☐ Module is ◯ *correct* or ◯ *has the following error and fix:*

```
module mult_2a #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   if ( a == 0 )          p = 0;
   else if ( a == 1 )     p = b;
   else                   p = a * b;

endmodule
```

☐ Module is ◯ *correct* or ◯ *has the following error and fix:*

```
module mult_2b #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   always_comb begin
      if ( a == 0 )       p = 0;
      else if ( a == 1 )  p = b;
      else                p = a * b;
   end

endmodule
```

☐ Module is ◯ *correct* or ◯ *has the following error and fix:*

```
module mult_2c #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   if ( b == 0 )          p = 0;
   else if ( b == 1 )     p = a;
   else                   p = a * b;

endmodule
```

☐ Module is ◯ *correct* or ◯ *has the following error and fix:*

```
module mult_2d #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   if ( a == 0 )          assign p = 0;
   else if ( a == 1 )     assign p = b;
   else                   assign p = a * b;

endmodule
```

Staple This Side

Staple This Side

(*b*) Show the values of `b` and `c` where requested below.

```
module assortment;
   logic [15:0] a;
   logic [0:15] b;
   logic [16:1] c;

   initial begin

      a = 16'h1234;
      b = a;
      c = a;
      //    Show value of b and c after line above executes:


      #1; // Not really needed.
      for ( int i=0; i<16; i++ ) b[i] = a[i];
      //    Show value of b after line above executes:

   end
endmodule
```


Staple This Side

Problem 6: [10 pts] Answer the following synthesis questions.

(*a*) Cadence Genus defines the following three synthesis steps: `syn_gen` (generic), `syn_map` (mapped, or technology mapping), and `syn_opt` (optimized). Answer the following questions about technology mapping.

☐ Explain what happens during technology mapping.

☐ Even if optimization were done before technology mapping why is it important optimize after technology mapping?

(*b*) What is the big disadvantage of setting the delay target too low when performing synthesis? (The small disadvantage is that it takes a longer time to run.)

☐ Disadvantage of setting delay target too low during synthesis:

Name _____ *Formatted For Two-Sided Printing*

Staple This Side

Digital Design using HDLs
LSU EE 4755
Final Examination

Wednesday, 8 December 2021 7:30 CST

Problem 1	_____	(30 pts)
Problem 2	_____	(35 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Staple This Side

Problem 1: [30 pts] For the modules in this problem input `sample` holds a new value each cycle, and output `r_avg` holds the average of the last `n_samples` inputs. (Ignore the fact that the module needs but lacks a reset.)

(*a*) For the module below show the hardware that will be inferred when instantiated with default parameters. Be sure to optimize for the default value of `n_samples`.

```
module ravg2 #( int w = 8, n_samples = 4 )
   ( output logic [w-1:0] r_avg,
     input uwire [w-1:0] sample,   input uwire clk );

   logic [w-1:0] samples[n_samples];

   parameter int wm = $clog2( n_samples );
   parameter int ws = w + wm;
   logic [ws-1:0] tot;

   always_ff @( posedge clk ) begin

      samples[0] <= sample;

      for ( int i=1; i<n_samples; i++ ) samples[i] <= samples[i-1];

      tot <= tot - samples[n_samples-1] + samples[0];

   end

   always_comb r_avg = tot / n_samples;

endmodule
```


Staple This Side

☐ Show hardware for the module above using default parameter values.

☐ Optimize for these parameter values.

Staple This Side

(*b*) The module to the right is similar to `ravg2` except that it has three arithmetic unit instantiations: an adder, a subtractor, and a divide-by-constant unit. Modify `ravg3` so that it uses these modules. For full credit connect them so that the critical path passes through at most one module per cycle. In a correct solution `r_avg` will arrive at the output of `ravg3` later than it would in module `ravg2`.

- [ ] Modify `ravg3` so that it uses the three arithmetic units.
- [ ] For full credit, the critical path can go through at most one arithmetic unit per cycle.
- [ ] The connections to the arithmetic units can be changed (say from `aa1` to something else).
- [ ] Do not add unnecessary cost or delay.

Staple This Side

```
module ravg3 #( int w = 8, n_samples = 4 )
   ( output logic [w-1:0] r_avg,
     input uwire [w-1:0] sample,
     input uwire clk );

   logic [w-1:0] samples[n_samples];

   parameter int wm = $clog2( n_samples );
   parameter int ws = w + wm;
   logic [ws-1:0] tot;

   always_ff @( posedge clk ) begin

      samples[0] <= sample;

      for ( int i=1; i<n_samples; i++ ) samples[i] <= samples[i-1];

      tot <= tot - samples[n_samples-1] + samples[0]; // Modify or eliminate this line.



   end

   always_comb r_avg = tot / n_samples;               // Modify or eliminate this line.


   uwire [ws-1:0] sum, diff;

   uwire [ws-1:0] aa1, aa2, da1;

   uwire [w-1:0] quot;

   uwire [w-1:0] sa1, sa2;




   our_adder #(ws,ws)           add1( sum,      aa1,      aa2 );

   our_sub #(ws,w)              sub2( diff,     sa1,      sa2 );

   our_div_by #(w,ws,n_samples) div3( quot,     da1 );




endmodule
```

Problem 2: [35 pts] Appearing below is a Verilog description of a lower-cost version of the `bit_keeper` module from Homework 4 and a diagram of the hardware.

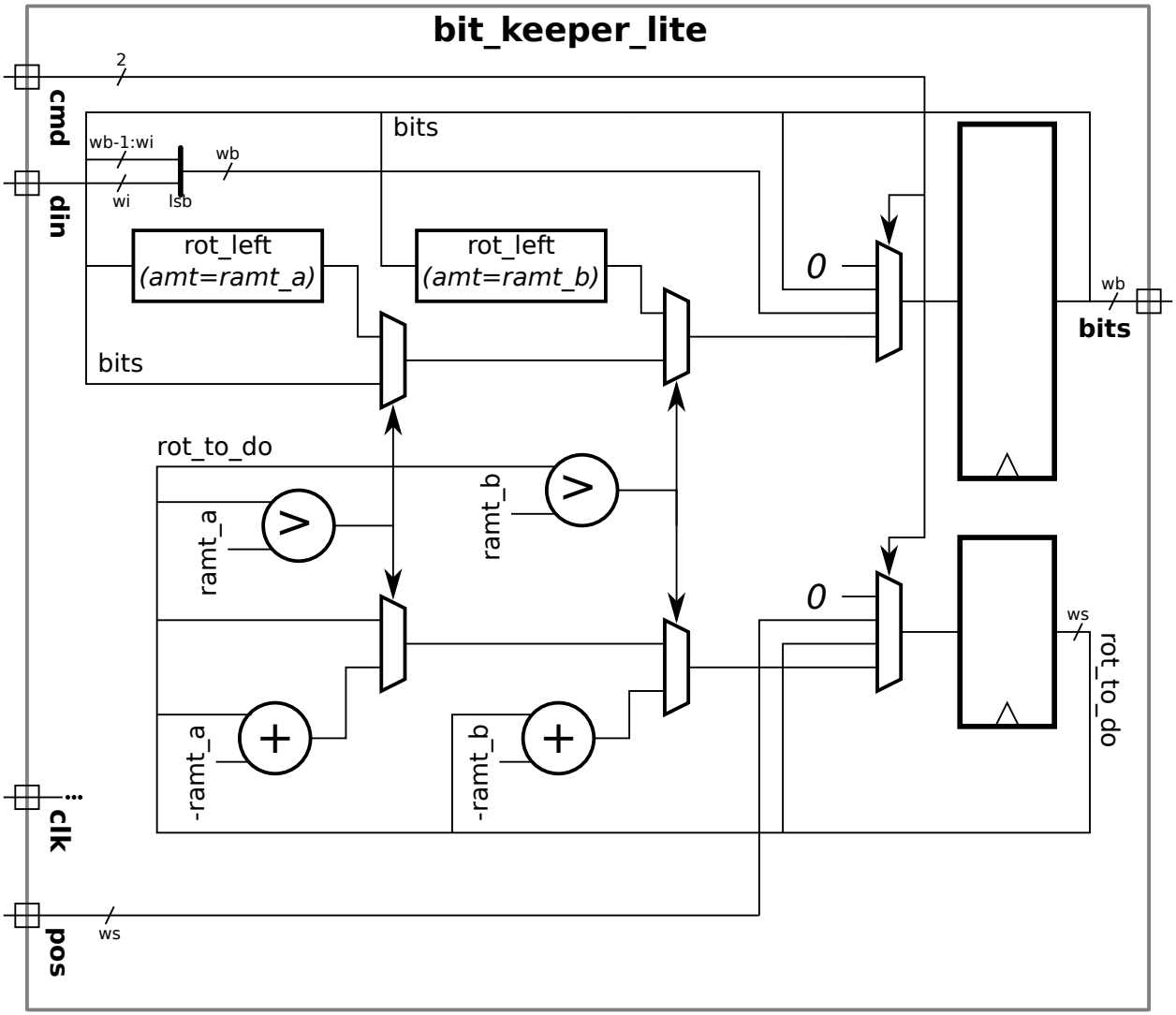
```
typedef enum { Cmd_Reset=0, Cmd_Rot_To=1, Cmd_Write=2, Cmd_Nop=3, Cmd_SIZE } Command;
module rot_left #( int w = 10, amt = 1 )
   ( output uwire [w-1:0] r, input uwire [w-1:0] a);
   assign r = { a[w-amt-1:0], a[w-1:w-amt] };
endmodule
module bit_keeper_lite #( int wb = 64, wi = 8, ws = $clog2(wb) )
   ( output logic [wb-1:0] bits,    output uwire ready,
     input uwire [1:0] cmd,         input uwire [wi-1:0] din,
     input uwire [ws-1:0] pos,      input uwire clk );
   localparam int ramt_a = 1;     // Specify Rotation Amounts
   localparam int ramt_b = 1 << ( ws >> 1 );
   uwire [wb-1:0] ra, rb;
   rot_left #(wb,ramt_a) rl1(ra,bits);
   rot_left #(wb,ramt_b) rl8(rb,bits);
   logic [ws-1:0] rot_to_do;      // Remaining amount of rotation to do.

   assign ready = rot_to_do == 0;
   always_ff @( posedge clk ) case ( cmd )
        Cmd_Reset:  begin bits = 0; rot_to_do = 0; end
        Cmd_Rot_To: rot_to_do = pos; // Initialize rotation. Rotate during Nop.
        Cmd_Write:  bits[wi-1:0] = din;
        Cmd_Nop:                     // Continue Executing a Cmd_Rot_To
           if ( rot_to_do >= ramt_b ) begin
              bits = rb;             // Use output of larger rot module.
              rot_to_do -= ramt_b;   // Decrement remaining rot amt.
           end else if ( rot_to_do >= ramt_a ) begin
              bits = ra;             // Use output of smaller rot module.
              rot_to_do -= ramt_a;   // Decrement remaining rot amt.
           end
      endcase
endmodule
```

(*a*) Find the cost and delay of the illustrated hardware using the simple model. Take into account the presence of constants. For the addition and comparison units assume a ripple implementation. Show any assumptions made. (See the next part before solving this one.)

☐ Show cost in terms of w_b, w_i, and w_s. ☐ Take into account constants.

☐ Show delays and arrival times on the diagram, and ☐ highlight the critical path. These should be in terms of w_b, w_i, and w_s.

Staple This Side



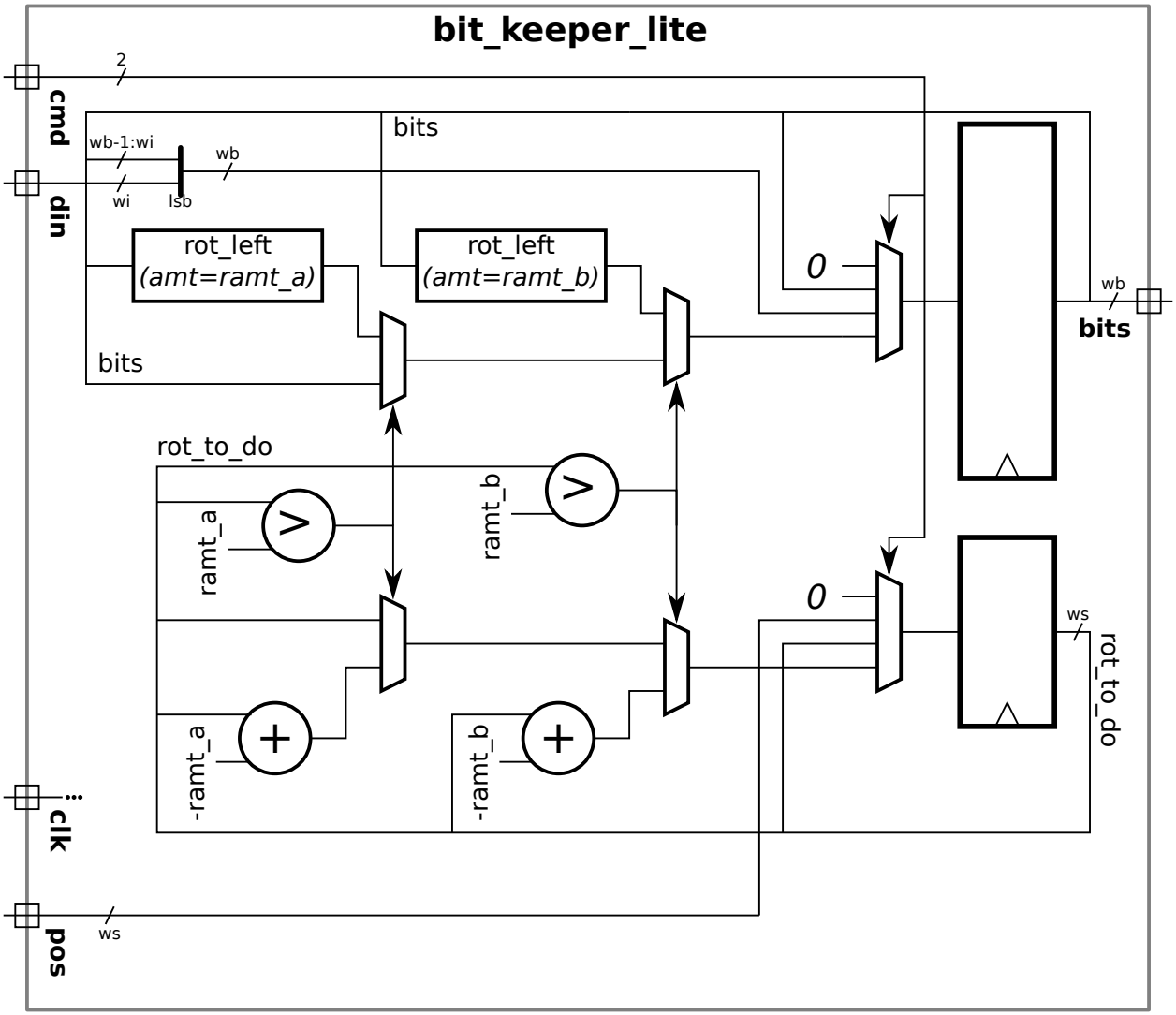
(*b*) In class we assume that a four-input mux is implemented using a reduction tree of 3 two-input muxen. For the illustrated hardware that would result in a longer critical path than is necessary. Modify the diagram on the right to show a better way of implementing the four-input multiplexors.

☐ Replace four-input multiplexors with two-input muxen connected to reduce critical path.

(*c*) Notice that care was taken to ensure that `ramt_b` is a power of 2. Explain how the fact that `ramt_b` is a power of two reduces the cost of the adder and comparison unit operating on `ramb_b`. Also explain how a power-of-2 `ramb_b` can reduce the cost of the other adder and comparison unit, if the synthesis program is clever enough. *Hint: Consider the binary representation of* `rot_to_do`.

☐ Since `ramt_b` is a power of 2 the adder and comparison unit connected to `ramt_b` are lower cost because:

☐ Since `ramt_b` is a power of 2 the adder and comparison unit connected to `ramt_a` (yes, a) are lower cost because:

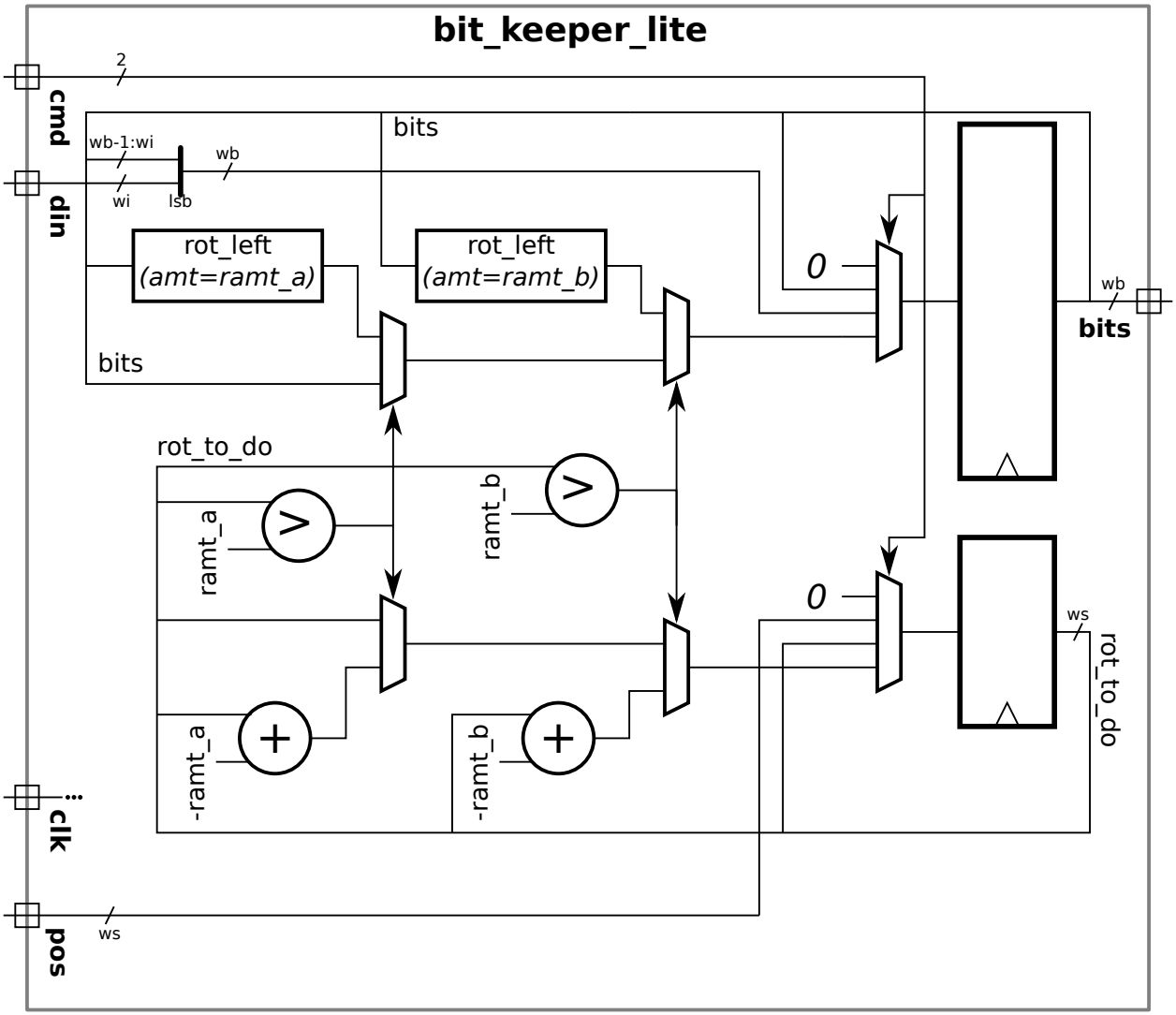


(*d*) Appearing below is a version of `bit_keeper_lite` with four ready outputs, `r1`, `r2`, `r3`, and `r4`. On the diagram add hardware that will be synthesized for each.

```
module bit_keeper_lite #( int wb = 64, wi = 8, ws = $clog2(wb) )
   ( output logic [wb-1:0] bits,    output uwire r1, output logic r2, r3, r4,
     input uwire [1:0] cmd,         input uwire [wi-1:0] din,
     input uwire [ws-1:0] pos,      input uwire clk );

   localparam int ramt_a = 1;
   localparam int ramt_b = 1 << ( ws >> 1 );

   uwire [wb-1:0] ra, rb;
   rot_left #(wb,ramt_a) rl1(ra,bits);
   rot_left #(wb,ramt_b) rl8(rb,bits);

   logic [ws-1:0] rot_to_do;
   assign r1 = rot_to_do == 0;          // [ ] Show hardware for r1.

   always_ff @( posedge clk ) begin
      r2 = rot_to_do == 0;              // [ ] Show hardware for r2.
      case ( cmd )
        Cmd_Reset:  begin bits = 0; rot_to_do = 0; end
        Cmd_Rot_To: rot_to_do = pos;
        Cmd_Write:  bits[wi-1:0] = din;
        Cmd_Nop: begin
           if ( rot_to_do >= ramt_b ) begin
              bits = rb;
              rot_to_do -= ramt_b;
           end else if ( rot_to_do >= ramt_a ) begin
              bits = ra;
              rot_to_do -= ramt_a;
           end
           r3 = rot_to_do == 0;         // [ ] Show hardware for r3.
        end
      endcase
      r4 = rot_to_do == 0;              // [ ] Show hardware for r4.
   end
endmodule
```

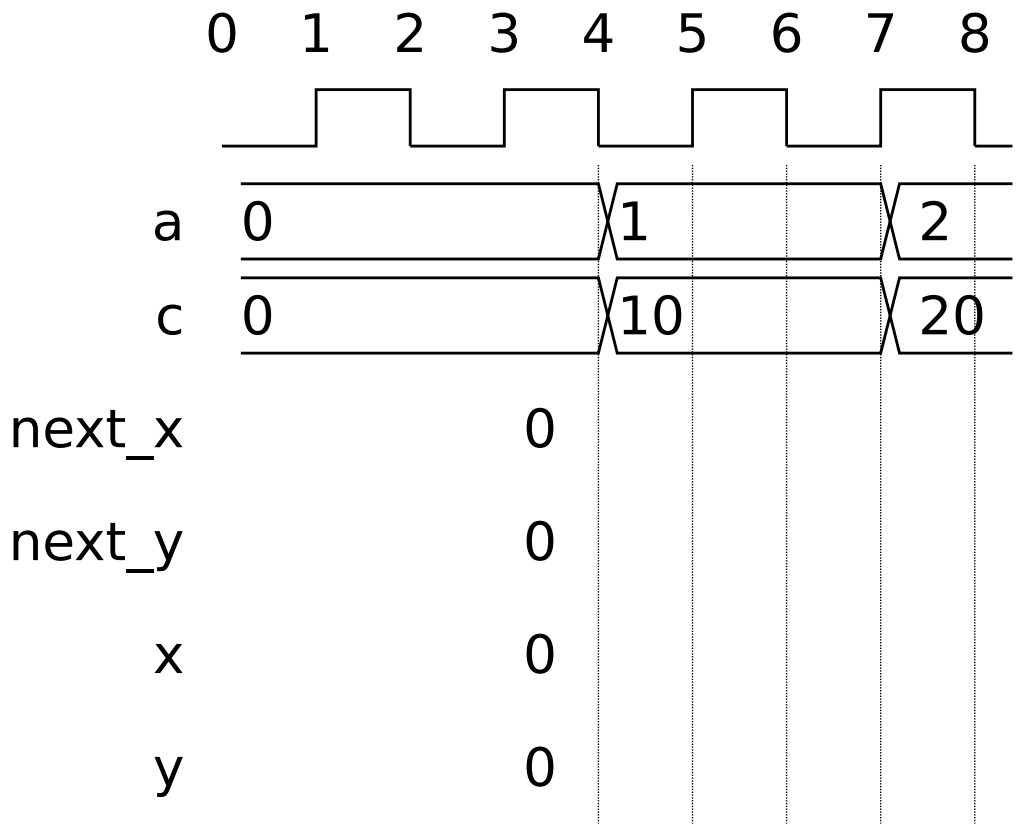
☐ Show hardware that will be synthesized for `r1`, `r2`, `r3`, and `r4`.

Staple This Side



Problem 3: [15 pts] Consider the modules below.

```
module ba
  ( output logic [15:0] next_x, next_y, x, y,
    input uwire [15:0] a, c,  input uwire clk );

   always_ff @( posedge clk ) x = next_x;
   assign next_x = a;
   assign next_y = x + c;
   always_ff @( posedge clk ) y = next_y;

endmodule

module test_ba;

   uwire [15:0] x, y, next_x, next_y;
   logic [15:0] a, c;
   logic clk;

   ba ba1( next_x, next_y, x, y, a, c, clk );

   initial begin
      // t = 0
      clk = 0;
      a = 0;  c = 0;
      #1; // t = 1
      clk = 1;
      #1; // t = 2
      clk = 0;
      #1; // t = 3
      clk = 1;
      #1; // t = 4
      clk = 0;  a <= 1;   c <= 10;  // Line t4
      #1; // t = 5
      clk = 1;
      #1; // t = 6
      clk = 0;
      #1; // t = 7
      clk = 1;  a <= 2;   c <= 20;  // Line t7
      #1; // t = 8
      clk = 0;
   end

endmodule
```



(*a*) Complete the timing diagram so that it shows the values of `next_x`, `next_y`, `x`, and `y` that would be produced with the modules above. *Note: In the original exam* `test\_ba` *did not use non-blocking assignments to* `a` *and* `c`.

☐ Complete timing diagram from $t = 4$ to $t = 8.1$. ☐ Note that there is a **negative** clock edge at $t = 4$.

(*b*) At $t = 5$ we can be sure that `y=next_y` will execute before `next_y=x+c`. Explain how this ordering is assured by the rules for the event queue.

☐ Explain how event queue regions assure `y=next_y` executes before `next_y=x+c` at $t = 5$.

(*c*) Notice that `a` and `c` are assigned using non-blocking assignments on Lines t4 and t7. Explain why the order of execution would be ambiguous at $t = 7$ if line t7 used blocking assignments: `a=1; c=10;`. *Note: This question was not in the original exam.*

☐ Describe ambiguity (more than one possible execution order) if blocking assignments were used.

☐ Would non-blocking assignments `x <= next_x` and `y <= next_y` remove the ambiguity? ☐ Explain.

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

(*a*) The foolish `sqrt` module below has several issues.

```
module sqrt #( int w = 16 )
   ( output logic [w-1:0] r, input uwire [w-1:0] a );

   always_comb begin

      r = 0;
      while ( r * r < a ) r++;

   end

endmodule
```

☐ Explain why, due to the Verilog rules for bit widths, the expression `r * r < a` won't compute the intended result.

☐ Why is the `sqrt` module likely not synthesizeable?

☐ What would be the problem with the hardware if it were synthesizable?

Staple This Side

(*b*) Consider the two division modules below. In the first `a2` is a parameter, in the second it is a module port. Use the `div_demo` module for your answers to the questions below.

```
module our_div_by
  #( int wq = 5, wd = 10, logic [wd-1:0] a2 = 4 )
   ( output uwire [wq-1:0] quot, input uwire [wd-1:0] a1 );
   assign quot = a1/a2;
endmodule

module our_div
  #( int wq = 5, wd = 10 )
   ( output uwire [wq-1:0] quot, input uwire [wd-1:0] a1, a2 );
   // cadence inline
   assign quot = a1/a2;
endmodule

module div_demo
  #( int w = 21 )
   ( output uwire [w-1:0] d1, d2,
     input uwire [w-1:0] x1, x2, x3, x4 );

   localparam logic [w-1:0] y1 = 4755;









endmodule
```

☐ Show an instantiation of `our_div` for which `our_div_by` could work.

☐ Show an instantiation of `our_div` for which `our_div_by` could not work.

☐ Explain how the use of the `cadence inline` pragma in `our_div` makes it possible to instantiate `our_div` in places that otherwise might need `our_div_by`.

(*c*) Answer the following questions about latency and throughput.

☐ Define latency.

☐ Define throughput.

Consider a sequential circuit (such as `mult_step` from Homework 6) and a pipelined version of the sequential circuit (such as `multi_step_pipe`). Assume that both have the same clock frequency.

☐ Remembering that the clock frequencies are the same, compared to the sequential version, does the pipelined version typically have
◯ *lower latency*, ◯ *the same latency*, or ◯ *higher latency*. ☐ Explain.

☐ Compared to the sequential version, does the pipelined version typically have
◯ *lower throughput*, ◯ *the same throughput*, or ◯ *higher throughput*. ☐ Explain.

☐ **Ignoring the cost of registers**, compared to the sequential version, does the pipelined version typically have
◯ *lower cost*, ◯ *the same cost*, or ◯ *higher cost*. ☐ Explain.

Staple This Side

5 Fall 2020

Name _____

Digital Design Using HDLs
LSU EE 4755
Solve-Home Midterm Examination

Friday, 6 Nov 2020 to early Monday, 9 Nov 2020 05:00 CST)

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Outside material that covers the same topics, such as Verilog tutorials, digital logic design guides can also be used. Do not try to directly seek out solutions to any question here. That is, don't Web-search the text of a problem. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Warning: Unlike homework assignments collaboration is not allowed on exams. Suspected copying will be reported to the dean of students. The kind of copying on a homework assignment that would result in a comment like "See ee4755xx for grading comments" will be reported if it occurs on an exam. Please do not take advantage of pandemic-forced test conditions to cheat!

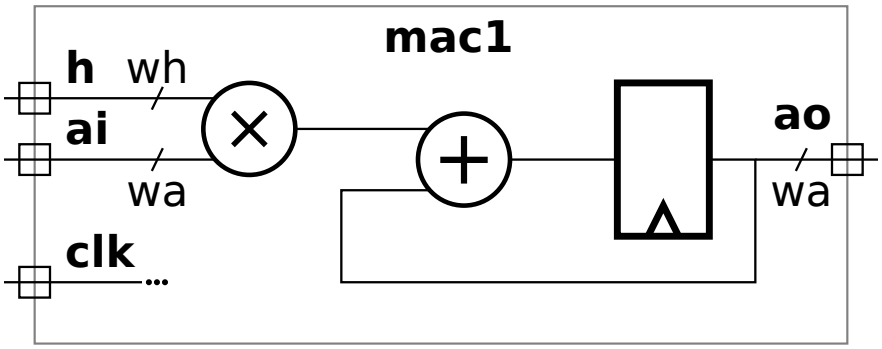
Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

☣ $r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck!

Problem 1: [20 pts] Appearing below are some variations on a multiply accumulate module.

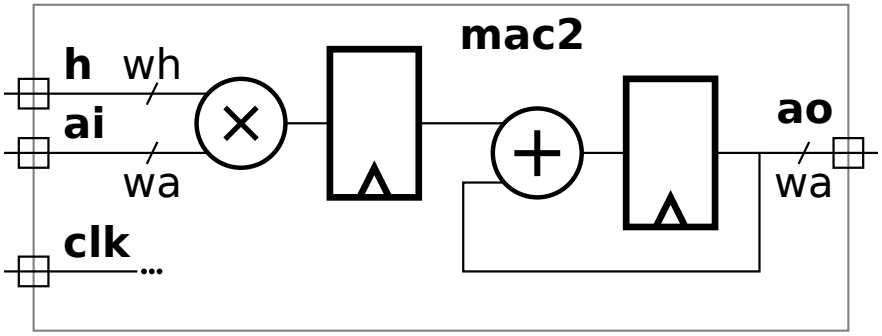
(*a*) Complete the Verilog code below so that it matches the illustration.



- [ ] Complete the Verilog.
- [ ] Use parameters for the bit widths `wh` and `wa`.
- [ ] The registers inferred from the Verilog must match the diagram.

```
module mac1

endmodule
```

(*b*) Complete the Verilog code below so that it matches the illustration, similar to the one on the previous page.

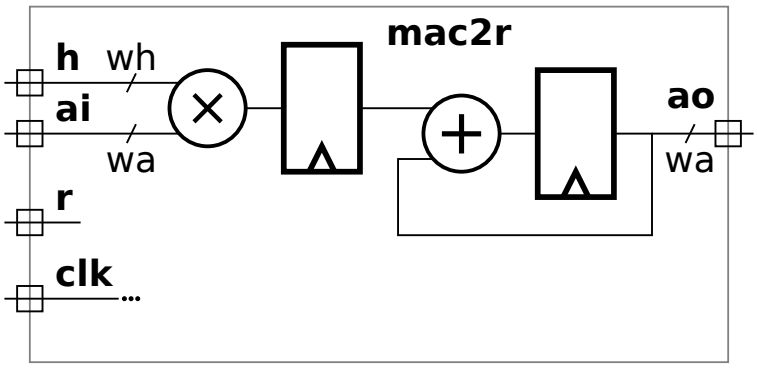


- [ ] Complete the Verilog.
- [ ] Use parameters for the bit widths `wh` and `wa`.
- [ ] The registers inferred from the Verilog must match the diagram.

```
module mac2

endmodule
```

Problem 2: [20 pts] The mac (multiply-accumulate) modules compute a running sum of products. The alert student might have noticed that there is no way to reset the sum. In this problem a reset will be added.

The module below has an input `r` (for reset) which is to work as follows: When `r=1` at a positive edge the product `h*ai` should start a new running sum. That is, that particular `h*ai` should be added to zero. When `r=0` at a positive edge the product `h*ai` should be added to the sum of the previous products. (If `r=0` is always true then the hardware as illustrated works correctly.)



☐ Add hardware to the diagram to implement the reset. ☐ Complete the Verilog to implement the reset.

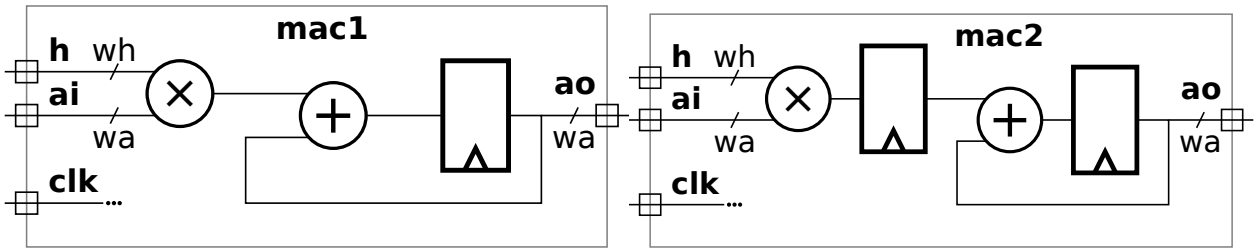
☐ Use parameters for the bit widths `wh` and `wa`.

☐ The registers inferred from the Verilog must match the diagram and ☐ be sure that the reset is applied to the correct value.

```
module mac2r
```

```
endmodule
```

Problem 3: [20 pts] Appearing below are the modules from the previous problem. Suppose that in the multiplier below bit i of the product were computed in time $[4i + 2]\,\mathrm{u_t}$ and that a ripple adder were used for the sum. Let w denote the value of `wh` and `wa` (which means `wh==wa`).



(*a*) Find the minimum clock period for each using the simple model, and taking into account cascading. (The clock period is the length of the critical path, including the register delay.)

☐ Find the clock period for `mac1` with cascading. ☐ Don't forget to include the delay of the register.

☐ Find the clock period for `mac2` with cascading. ☐ Don't forget to include the delay of the registers.

(*b*) Find the minimum clock period for each using the simple model assuming that the multiplier output and adder input could not cascade.

☐ Find the clock period for `mac1` without cascading. ☐ Don't forget to include the delay of the register.

☐ Find the clock period for `mac2` without cascading. ☐ Don't forget to include the delay of the registers.

Problem 4: [20 pts] Appearing below is a recursively defined multiplier constructed using bfa (binary full adder) and bha (binary half adder) modules.

```
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
      // Split a in half and recursively instantiate a module for each half.
      localparam int wn = wa / 2;
      localparam int wx = wb + wn;
      uwire [wx-1:0] prod_lo, 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
```

Show the hardware that will be inferred for two levels of recursion and compute its cost. That is, show three instances of `mult_tree_bfas`: a top-level one, and two recursive instantiations. Show the hardware for the top-level instance and both of the two recursive instantiations. (It is only necessary to show two levels.) Do this for `wa=8` in the top-level module.

☐ Show the inferred hardware.

☐ Be sure to distinguish hardware (such as a `bfa` module) from values computed during elaboration.

☐ Compute the cost of the hardware in your diagram using the simple model. (Work out the cost of a bha by hand.) ☐ The cost should be for two levels, not for hardware going down to the base case.

Problem 5: [20 pts] Answer each question below.

(*a*) Appearing below is a multiply/add module, `nnMADDfp`, that computes its result using a FP add and multiply module. The values on the ports are IEEE 754 floats, and when `wa=32` the format is IEEE 754 single, the same as a SystemVerilog `shortreal`. That is followed by an incomplete testbench module, `testnnMADD`. The testbench module generates random values for the `nnMADDfp` module in variables `ar`, `br`, and `sir`, and computes what the result should be, `sor`.

Add Verilog code to deliver `ar`, `br`, and `sir` to the `nnMADDfp` instance, and to put the output of `nnMADDfp` into `sor_mut` so that `sor_mut` has the correct type of value. Note that one does not need to understand what is inside of `nnMADDfp`, `nnAddfp`, nor `nnMultfp`.

☐ Deliver (whatever that means) `ar`, `br`, and `sir` to `nnMADDfp` instance. ☐ Get output of the `nnMADDfp` instance into variable `sor_mut`.

```
module nnMADDfp #( int wa = 10 )
  ( output uwire [wa-1:0] so, input uwire [wa-1:0] a, b, si);
   uwire [wa-1:0] p;
   nnMultfp #(wa) mu(p, a, b);
   nnAddfp #(wa) ad(so, si, p);
endmodule

module testnnMADD;
   localparam int w = 32, ntests = 100;
   uwire [w-1:0] so;
   logic [w-1:0] a, b, si;
   nnMADDfp #(w) n(so, a, b, si);

   initial begin

      for ( int t=0; t<ntests; t++ ) begin
         shortreal sor, ar, br, sir, sor_mut;
         ar = rand_fp();    // Value to be used as input a to nnMADDfp.
         br = rand_fp();    // Value to be used as input b to nnMADDfp.
         sir = rand_fp();   // Value to be used as input si to nnMADDfp.
         sor = ar * br + sir;




         #1;




         sor_mut =                          ; // <-- DON'T FORGET.

         if ( sor != sor_mut ) handle_incorrect_result();

      end
   end
endmodule
```

(*b*) The module below will not compile or simulate due to multiple assignments to `temperature`, which is declared `uwire`. Changing `uwire` to `wire` will fix the compile problem. Nevertheless, is that the right fix?

```
module more_stuff #( int w = 16 )
   ( output uwire [w-1:0] v, y, input uwire [w-1:0] a, b, c );

   uwire [w-1:0] temperature;

   assign temperature = a + b;
   assign v = temperature >> c;
   assign temperature = a - b;
   assign y = temperature << c;

endmodule
```

☐ What problem remains after changing `temperature` from a `uwire` to a `wire`?

☐ Fix the problem based on what the code looks like its trying to do.

(*c*) An important part of synthesis is optimizing. It is possible to optimize before and again after technology mapping.

☐ What is technology mapping? ☐ Show an example of logic before and after technology mapping. (Make up some technology.)

☐ Describe an optimization that can be done before technology mapping. Provide an example. (This is done all the time in class.)

☐ Describe an optimization that can be done only after technology mapping (or perhaps during). Provide an example, feel free to make things up.

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!

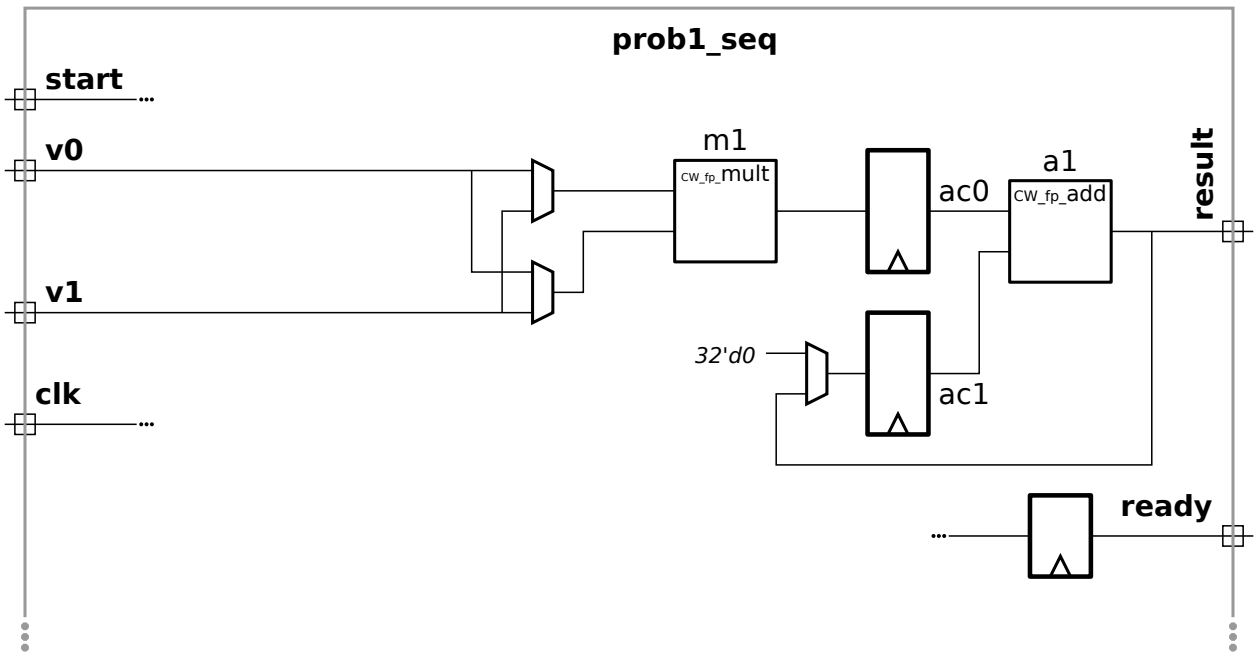
Problem 1: [20 pts] Module `prob1_seq`, below, is based on the solution to 2016 Final Exam Problem 1 (also appearing in problem set `https://www.ece.lsu.edu/koppel/v/guides/pset-syn-seq-main.pdf`, please look at that solution). In that problem an incomplete diagram of the hardware was given, similar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

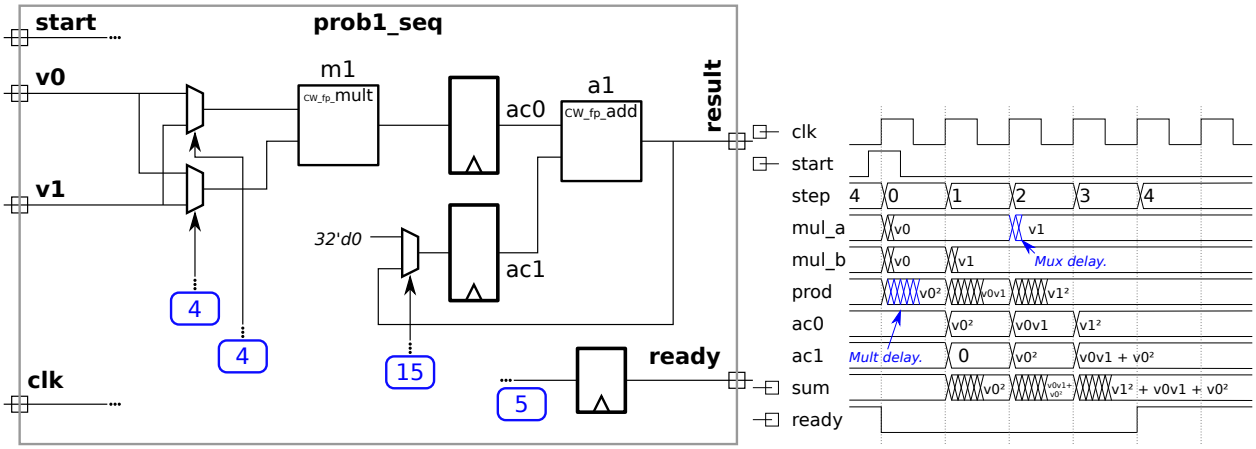
Problem 2: [20 pts] Consider again that module from Problem 1 of the 2016 final exam. Appearing below is the start of a Verilog description of a pipelined version of this module. The ports are the same as in the sequential version from the previous problem, however the module must operate in pipelined 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.

6 Fall 2019

Name _____

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 30 October 2019 10:30–11:20 CDT

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

Alias _____

Good Luck!

Problem 1: [20 pts] Appearing below is one of the solutions to Homework 2, the count leading zeros module.

```
module clz_bi_tree #( int w = 19, int ww = $clog2(w+1) )
                    ( output uwire [ww:1] nlz,   input uwire [w:1] a );
   if ( w == 1 ) begin
      assign nlz = ~ a;
   end else begin
      localparam int wlo = w/2,             whi = w - wlo;
      localparam int wwlo = $clog2(wlo+1),  wwhi = $clog2(whi+1);
      uwire [wwlo:1] lz_lo;
      uwire [wwhi:1] lz_hi;
      clz_bi_tree #(wlo) clo( lz_lo, a[wlo:1]   );
      clz_bi_tree #(whi) chi( lz_hi, a[w:wlo+1] );
      assign nlz = lz_lo < wlo ? lz_lo : wlo + lz_hi;
   end
endmodule
```

Show the hardware that will be inferred for the module for $w > 1$. Just show one level, don't show what is inside of `clo` and `chi`.

☐ Show synthesized hardware for one level. ☐ Be sure to show `clo` and `chi` (but not their contents).

Problem 2: [25 pts] In Homework 2 a clz (count leading zeros) module was constructed recursively by splitting the input bit vector and connecting each half to a smaller instance. The incomplete module below is similar except that the input vector is to be split into thirds and each third connected to a recursive instance. Complete the module.

☐ Complete so that `clz_tri_tree` computes clz.

```
module clz_tri_tree #( int w = 19, int ww = $clog2(w+1) )
                     ( output uwire [ww-1:0] nlz, input uwire [w-1:0] a );

   if ( w == 1 ) begin
      assign nlz = ~ a;

      // ☐   Make any needed changes to terminal case(s).




   end else begin

      // ☐   Finish these localparams.
      localparam int wlo =

      localparam int wmi =

      localparam int whi =


      localparam int wwlo = $clog2(wlo+1), wwmi = $clog2(wmi+1), wwhi = $clog2(whi+1);
      uwire [wwlo-1:0] lz_lo;  // No need to change these four lines.
      uwire [wwmi-1:0] lz_mi;
      uwire [wwhi-1:0] lz_hi;

      // ☐   Finish module connections below.

      clz_tri_tree #(wlo) clo( lz_lo, a[                ] );

      clz_tri_tree #(wmi) cmi( lz_mi, a[                ] );

      clz_tri_tree #(whi) chi( lz_hi, a[                ] );


      // ☐   Finish nlz.

      assign nlz =


   end
endmodule
```

Problem 3: [27 pts] Appearing below are modules that test if two bit vectors are equal in some way.

(*a*) Show the hardware for the module below at the default size using basic gates: AND, OR, XOR, NOTs, and bubbled inputs and outputs. **Do not** use something like `==`.

```
module eq #( int w = 4 )( output uwire equal, input uwire [w-1:0] a, b );
   assign equal = a == b;
endmodule
```

☐ Show hardware using basic gates at default size.

(*b*) Show the cost and delay of the module in terms of w (the value of parameter `w`) using the simple model.

☐ In terms of w: ☐ Cost and ☐ Delay.

(*c*) The module below also tests equality but it does so after shifting the first operand. Show the hardware in terms of basic gates after optimization.

```
module eqs #( int w = 6, int s = 2 )( output uwire equal, input uwire [w-1:0] a, b );
   localparam logic [w+s-1:0] zero = 0;
   assign equal = zero + ( a << s ) == b;
endmodule
```

☐ Show hardware at default size after optimization.

(*d*) The module below performs a different operation than the one above. Explain the difference and show an example.

```
module eqt #( int w = 16, int s = 5 ) ( output uwire equal, input uwire [w-1:0] a, b );

   assign equal = ( a << s ) == b;

endmodule
```

☐ Difference between operation `eqs` and `eqt`.

☐ Show a value for `a` and `b` for which the output of `eqs` and `eqt` are different.

Problem 4: [28 pts] Answer each question below.

(*a*) Appearing below is synthesis data taken from the solution to Homework 2. The `Delay Target` column shows the maximum delay constraint given to the synthesis program.

Module Name	Area	Delay Actual	Delay Target
clz_w32	26290	3.110	10.000 ns
clz_tree_w32	21706	1.425	10.000 ns
clz_w32_1	36476	1.007	0.100 ns
clz_tree_w32_5	37356	0.577	0.100 ns

☐ In general, which result should be used if the only goal were to minimize area, the results for the ◯ 10.0 ns *Target* or for the ◯ 0.1 ns *Target* ? ☐ Explain.

☐ In general, which result should be used if the only goal were to minimize delay, the results for the ◯ 10.0 ns *Target* or for the ◯ 0.1 ns *Target* ? ☐ Explain.

(*b*) Provide w-bit declarations requested below.

```
// ☐  Declare each object to be w bits and consistent with its name.
//
uwire [       ] bit_zero_is_msb;

uwire [       ] bit_zero_is_lsb;

uwire [       ] bit_zero_is_middle;
```

(*c*) The module fragment below starts with six declarations (the object names starting with `r`), each providing a value (either `a+b` or `x+y`). Some of those declarations will result in compile errors. Identify them and explain the problem. If possible fix the problem without changing the object kind (localparam, uwire, var).

```
module my_mod
  #( int w = 10, int x = 11, int y = 12 )
   ( input uwire [w:1] a, b );

   localparam logic [w:1] r1p = a + b;

   localparam logic [w:1] r2p = x + y;

   uwire [w:1] r1w = a + b;

   uwire [w:1] r2w = x + y;

   logic [w:1] r1l = a + b;

   logic [w:1] r2l = x + y;
```

☐ Indicate which ones are wrong and ☐ the reason that they are wrong.

☐ Indicate which can't be fixed and ☐ and explain why not.

(*d*) Explain what `$realtobits` does, and what hardware will be synthesized for it, if any.

```
always_comb begin

   x = $realtobits(r);

end
```

☐ Purpose of `realtobits`.

☐ Synthesized hardware.

Name _____ *Formatted For 2-Sided Printing*

Staple Here

Digital Design using HDLs
LSU EE 4755
Final Examination

Friday, 13 December 2019 10:00-12:00 CST

Problem 1 _____ (30 pts)

Problem 2 _____ (25 pts)

Problem 3 _____ (20 pts)

Problem 4 _____ (25 pts)

Alias _____ Exam Total _____ (100 pts)

Good Luck!

Staple Here

Problem 1: [30 pts] Appearing below is the solution to Homework 6, the accumulation module. The next page shows the pipelined adder and `st_occ`, which is some of the inferred hardware. Show the rest of the inferred hardware after some optimization. Leave the pipelined adder as a box.


```
module add_accum #( int w = 21, n_stages = 3 )
   ( output logic [w-1:0] sum,   output logic sum_valid,
     input uwire [w-1:0] ai,     input uwire ai_v, reset, clk );

   logic [n_stages-1:0] st_occ;
   assign sum_valid = !st_occ;
   uwire aout_v = st_occ[n_stages-1];

   uwire [w-1:0] aout;
   uwire [w-1:0] a0 = ai_v   ? ai   : sum;
   uwire [w-1:0] a1 = aout_v ? aout : sum;

   add_pipe #(w,n_stages) add_p0( aout, a0, a1, clk );

   logic sum_occupied;
   uwire [1:0] n_values = ai_v + sum_occupied + aout_v;
   uwire saa = n_values >= 2;  // Start an addition.
   uwire write_sum = !sum_occupied && n_values == 1;

   always_ff @( posedge clk ) if ( reset ) begin
      sum <= 0;
      sum_occupied <= 0;
      st_occ <= 0;
   end else begin
      if ( write_sum ) sum <= aout_v ? aout : ai;
      sum_occupied <= n_values[0];
      st_occ <= { st_occ[n_stages-1:0], saa };
   end
endmodule
```

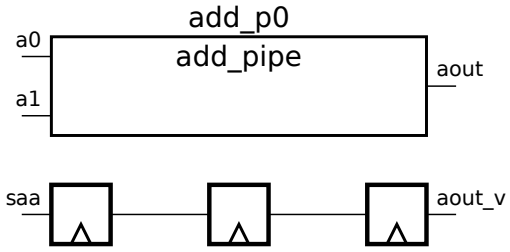
☐ Show inferred hardware after some optimization, but ☐ leave `add_pipe` as a box.

☐ Show logic associated with `n_values` as basic gates and a single BFA, **do not show adders and do not show comparison units**.

☐ Clearly show all input and output ports, do not confuse parameters with ports.

☐ Avoid effortlessly optimized hardware, such as gates with constant inputs.

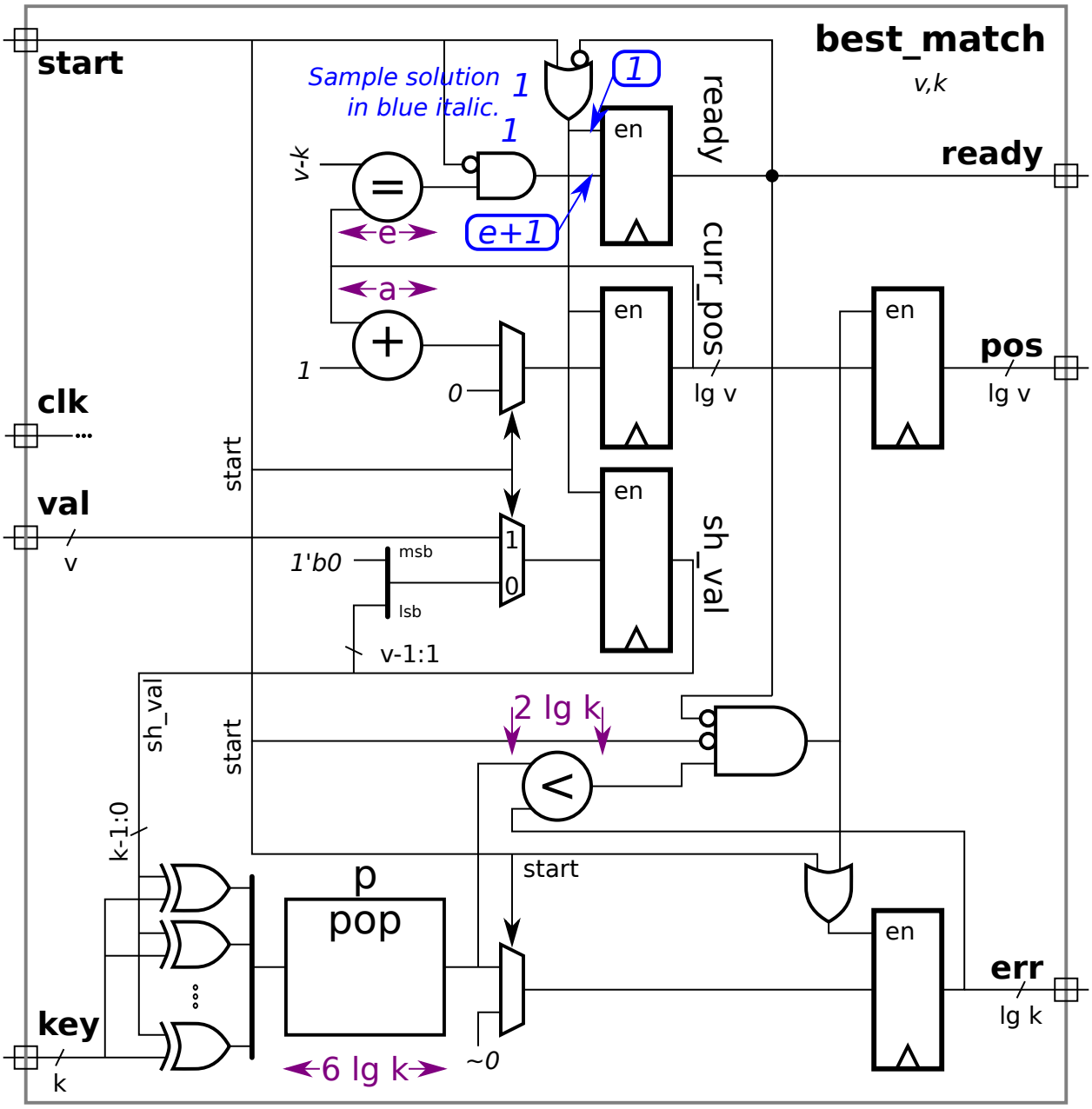
Staple Here



Staple Here

Problem 2: [25 pts] Appearing below is hardware from the solution to Homework 5, Problem 2. The parameter names have been shortened, such as changing `wv` to `v` and using `lg v` for `wvb`. The diagram shows the delay through some of the modules, including the `pop` module. Treat `e` and `a` (delays for [=] and [+]) as given constants for the first part.

(*a*) Based on the provided delays and using the simple model for others, compute the arrival time (delay) of signals at each register input. That's two inputs for each of five registers. The solution for `ready` is shown in blue, so only four registers remain. Also, highlight the/a critical path to the `err` register.



☐ Show the arrival time of the enable and data signal at each register input and ☐ Highlight a critical path to `err` with a squiggly line.

☐ Take into account constant inputs when computing delays.

Staple Here

Staple Here

Staple Here

(*b*) The equality module is shown with a delay of `e`. Show the hardware for that module and compute the cost and delay using the simple model. Take into account the width of the inputs and the fact that one input is a constant.

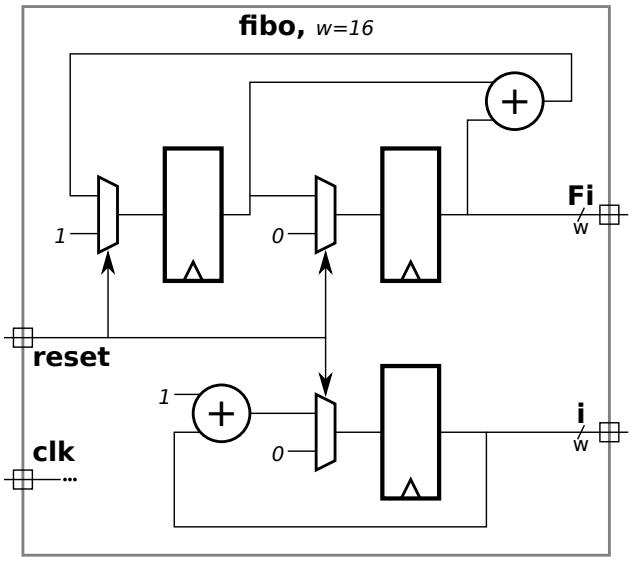
☐ Sketch hardware for equality module for $\lg v = 8$ and $v - k = 1011\,0001_2$, and ☐ taking into account the constant input.

☐ Show the cost of the hardware for the equality module above based on the simple model in terms of $\lg v$.
☐ Don't forget to take the constant input into account.

☐ Show the delay of the hardware based on the simple model in terms of $\lg v$. ☐ Don't forget to take the constant input into account.

Staple Here

Problem 3: [20 pts] The hardware illustrated to the right emits a famous integer sequence. Write a synthesizable Verilog description of the hardware.



☐ Complete the module, ☐ be sure that it is synthesizable.

☐ Use non-blocking assignments carefully.

☐ Be sure to include all ☐ input and output ports and ☐ parameters.

☐ Make sure that all objects have the appropriate widths.

```
module fibo
```

```
endmodule
```


Staple Here

Staple Here

Problem 4: [25 pts] Answer each question below.

(*a*) Appearing below are synthesis script results for the pipelined integer adder from Homework 6. That adder computes a w-bit integer sum using an n-stage pipeline in which each stage computes $\lceil w/n \rceil$ bits of the sum, starting with the $\lceil w/n \rceil$ least-significant bits in the first stage.

All syntheses are of a $w = 24$-bit adder, versions with $n = 1$, 2, 3, 4, and 6 stages are synthesized. The delay target is set to an easy 90 ns.

Module Name	Area	Delay Actual	Delay Target
add_pipe_w24_n_stages1	29928	10.174	90.000 ns
add_pipe_w24_n_stages2	47043	5.428	90.000 ns
add_pipe_w24_n_stages3	64159	3.701	90.000 ns
add_pipe_w24_n_stages4	81275	2.837	90.000 ns
add_pipe_w24_n_stages6	115506	1.973	90.000 ns

☐ Based on this data provide the ☐ latency and ☐ throughput for the three-stage adder. Be sure to ☐ use appropriate units for the throughput.

☐ Note that the area (cost) increases with the number of stages. Based on the description above what is the main contributor to the increase in cost?

Staple Here

(*b*) The two modules below appear to be similar.

```
module plan_I(output logic [7:0] e, input logic [7:0] a,b);
   logic [7:0] c;
   always_comb begin
      c = a + b;
      e = c + a;
   end
endmodule

module plan_II(output logic [7:0] e, input logic [7:0] a,b);
   logic [7:0] c;
   always_comb e = c + a;
   always_comb c = a + b;
endmodule
```

☐ For which module will the simulator perform unnecessary addition? ☐ Explain.

☐ Is the result computed by the two modules different or the same? ☐ Explain.

(*c*) What value will y have at the end of the initial block?

```
module s;
   logic [15:0] a,b,y;
   initial begin
      a = 1;
      b = 100;
      b <= 10;
      y = 0;
      y <= a + b;
      y = 999;
      #1;
      a = 2;
      b <= 20;
      #200;
      // Show value of y at this point in execution.
   end
endmodule
```

☐ Value of y at end of block is:

Staple Here

(*d*) Consider the declarations below.

```
module types;
   int en;
   logic [31:0] lo;
   bit [31:0] b;
   uwire [31:0] u = 33;
   localparam int p = 22;
endmodule
```

☐ Object `u` has the same data type as one of the other objects. Which is it?

☐ What is the difference between `lo` and `b` (`logic` and `bit`)?

☐ Notice that `u` is assigned a value. What is it about object `lo` that makes it illegal to assign a value in its declaration?

☐ Add correct code to assign value 44 to `lo`.

7 Fall 2018

Name _____

Digital Design using HDLs
LSU EE 4755
Midterm Examination

Friday, 26 October 2018 9:30–10:20 CDT

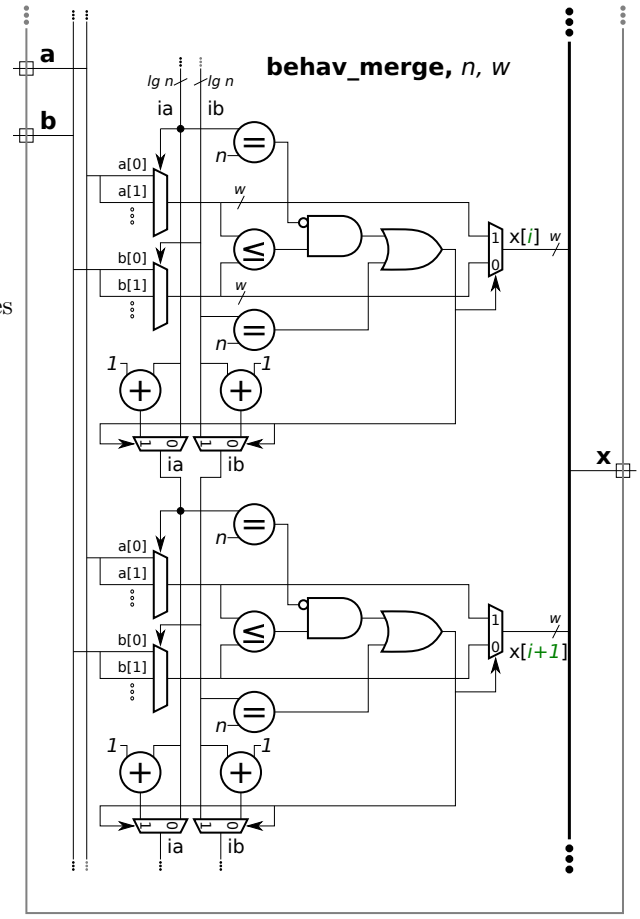
Problem 1	_____	(22 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(23 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

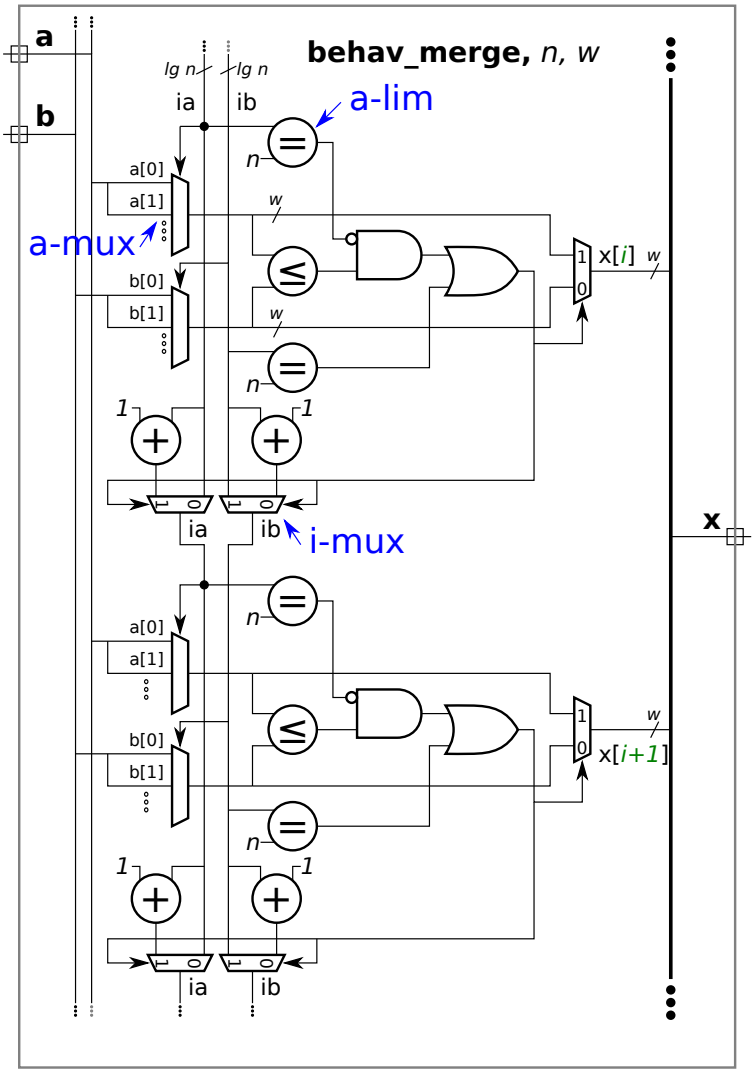
Problem 1: [22 pts] The illustration below shows some of the inferred hardware for the `behav_merge` module from the solution to Homework 6. The hardware that's shown is for typical iterations `i` and `i+1`. Show the hardware for iterations `i=0` and `i=1` with optimizations applied.

- ☐ Show hardware for iterations `i=0` and `i=1`.
- ☐ Also show hardware for code before `for` loop.
- ☐ Optimize hardware. Take into account possible values of `ia` and `ib`.




```
module behav_merge
 #( int n = 4, int w = 8 )
  ( output logic [w-1:0] x[2*n],
    input uwire [w-1:0] a[n], b[n] );

  logic [$clog2(n+1)-1:0] ia, ib;
  always_comb begin
   ia = 0; ib = 0;
   for ( int i = 0;  i < 2*n;  i++ )
     if ( ib==n || ia!=n && a[ia]<=b[ib] )
       x[i] = a[ia++]; else x[i] = b[ib++];
  end

endmodule
```

Problem 2: [20 pts] Appearing once again is part of the Homework 6 solution, this time with items labeled in blue. Show the cost and delay of these, as requested below. See the previous problem for the Verilog description. The phrase *most expensive* means for the value of i for which the device needs all inputs, even after optimization. For the mux, show the cost and delay for the tree implementation.



☐ Cost of most expensive `a-mux` in terms of n and w.

☐ Delay of most expensive `a-mux` in terms of n and w.

☐ Cost of most expensive `i-mux` in terms of n and w.

☐ Delay of most expensive `i-mux` in terms of n and w.

☐ Cost of most expensive `a-lim` in terms of n and w ☐ after optimizing for constant inputs.

☐ Delay of most expensive `a-lim` in terms of n and w ☐ after optimizing for constant inputs.

Problem 3: [23 pts] Output `lt` of module `comp`, below, should be 1 iff `a` is strictly less than `b`, and `eq` should be 1 iff `a==b`. Both `a` and `b` are unsigned integers. The module recursively instantiates two instances of itself, one is supposed to compare the low bits of the inputs, the other compares the high bits. Complete the module so that it works for any positive `w`.

☐ Complete the module, don't miss the ☐ `FILL IN` items.

☐ Make sure that it works for odd and even values of `w`.

```
module comp
  #( int w = 8 )
   ( output uwire lt, eq, input uwire [w-1:0] a, b );

   if (       ) begin   // Terminating Case Condition                <----  ☐  FILL IN

      assign lt = !a && b;
      assign eq = a == b;

   end else begin

      uwire llo, lhi, elo, ehi;



      // Instantiate two comp modules,  connect each to about half the inputs.
      //
      //     ----                    --------------      -------------- <--  ☐  FILL IN
      comp #(    ) clo( llo, elo, a[              ],  b[              ] );

      comp #(    ) chi( lhi, ehi, a[              ],  b[              ] );



      assign lt =                          ;                    //   <----  ☐  FILL IN

      assign eq =                          ;                   //    <----  ☐  FILL IN

   end
endmodule
```

Problem 4: [10 pts] The output of `plus_amt`, `x`, is to be set to `b + amt`. Input `b` and output `x` are expected to be in IEEE 754 double FP format (the same format as type real). (Note: the port declarations are not to be modified in the problems below.) Several variations on the module appear below. *Hint: Solution to this problem require the correct use of* `realtobits` *and/or* `bitstoreal`. *Grading Note: The bonus problem was not on the original exam.*

(*a*) The module below does not compute the correct result. Fix the module by modifying the `always_comb` block. The module does not need to be synthesizable.

☐ Fix so that `x` is assigned the correct result, `amt` plus value of `b`.

```
module plus_amt
  #( real amt = 1.5 )
   ( output logic [63:0] x, input uwire [63:0] b ); // DO NOT modify ports.
   // Both x and b are IEEE 754 doubles (reals).

   always_comb begin
      // Change code below.



      x = b + amt;



   end

endmodule
```

(*b*) [0 pts] Bonus Problem Complete the module below so that it uses the `CW_fp_add` module to do the addition. The parameters to `CW_fp_add` are already correct, just connect the inputs and outputs.

☐ Complete so that it computes the correct result.

```
module plus_amt
  #( real amt = 1.5 )
   ( output logic [63:0] x, input uwire [63:0] b ); // DO NOT modify ports.
   // Both x and b are IEEE 754 doubles (reals).



   uwire logic [7:0] s; // Unused.
   // Computes z = a + b.
   CW_fp_add #(.sig_width(52),.exp_width(11)) // This line correct, don't change.
    fadd( .status(s), .rnd(0),                // This line correct, don't change.
         .z(        ), .a(        ), .b(        ) );



endmodule
```

Problem 5: [25 pts] Show the hardware that will be inferred for the Verilog code below.

- ☐ Clearly show module ports.
- ☐ Show inferred hardware. Don't optimize.
- ☐ Pay close attention to what is and is not inferred as a register.

```
module regs #( int w = 10, int k1 = 20, int k2 = 30 )
   ( output logic [w-1:0] y,
     input logic [w-1:0] b, c,
     input uwire clk );

   logic [w-1:0] a, x, z;

   always_ff @( posedge clk ) begin

      a = b + c;
      if ( a > k1 ) x = b + 10;
      if ( a > k2 ) z = b + x; else z = c - x;
      y = x + z;

   end

endmodule
```

Name _____

Digital Design using HDLs
LSU EE 4755
Final Examination

Wednesday, 5 December 2018 15:00-17:00 CST

Problem 1	_____	(20 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(25 pts)
Exam Total	_____	(100 pts)

Alias _____

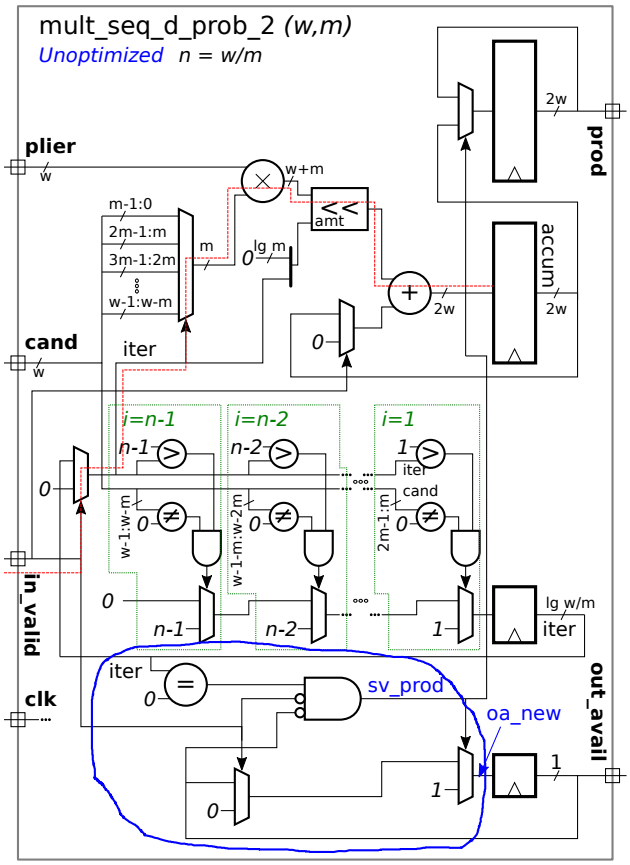
Good Luck!

Problem 1: [20 pts] Appearing to the right is the hardware inferred for the Homework 7 Problem 2 module, the fast sequential multiplier which skipped over zeros in the multiplicand.

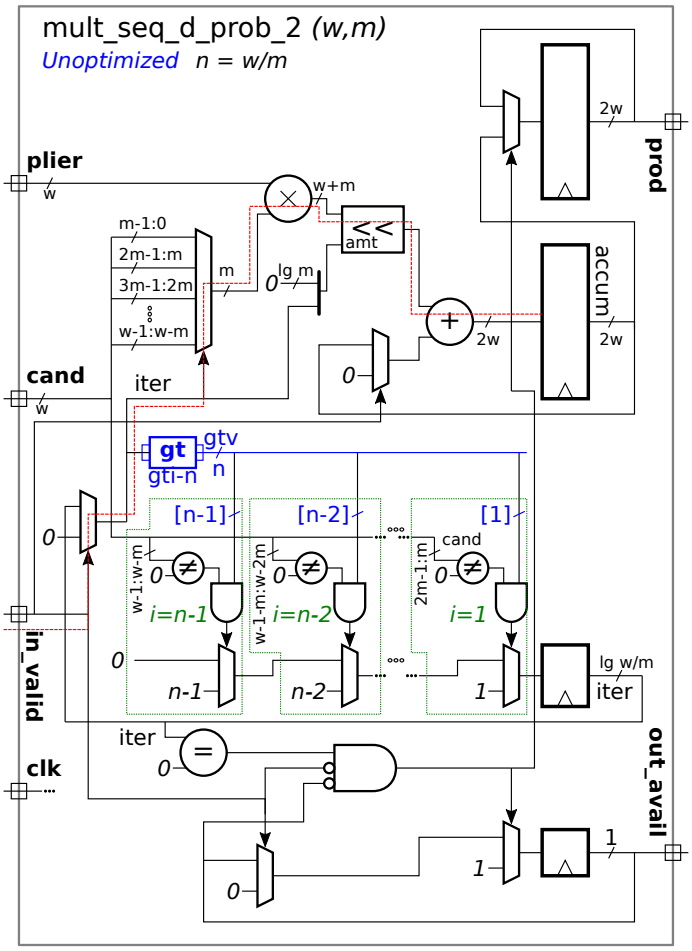
(*a*) Notice that some hardware is circled in blue. Optimize that hardware and show the cost of the optimized hardware. The optimized hardware should generate signals `sv_prod` and `oa_new`. If possible, replace the multiplexors with simpler gates.

☐ Show optimized hardware.

☐ Cost of optimized hardware:



(*b*) In the version of the module appearing below the $\boxed{>}$ units have been replaced by one module, `gt`, the changed hardware appears in blue. As can be inferred from the diagram bit `i` of the output of `gt`, `gtv`, is 1 iff `i>iter`. In the Verilog code below `gt` is instantiated but it is not being used. Modify the Verilog code so that **the existing `for` loop** uses the output of `gt` instead of the > operators. Pay attention to the version of `iter` used by `gt`.



☐ Use `gt` output in existing `for` loop.

☐ Make sure that `gt` uses correct `iter` version.

```
module mult_seq_d_prob_2
  #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,
     output logic out_avail,
     input uwire clk, in_valid,
     input uwire [w-1:0] plier, cand );

   localparam int n = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(n);
   uwire [n-1:0][m-1:0] cand_2d = cand;
   bit [iter_lg-1:0] iter, next_iter;
   logic [2*w-1:0] accum;

   uwire [n-1:0] gtv;

   uwire [iter_lg-1:0] gt_iter = 0;              //  ☐  FILL IN

   gt #(n,iter_lg) gti( gtv, gt_iter );

   always_ff @( posedge clk ) begin

      if ( in_valid ) begin
         iter = 0;   accum = 0;  out_avail = 0;
      end else if ( !out_avail && iter == 0 ) begin
         prod = accum;   out_avail = 1;
      end

      accum += plier * cand_2d[iter] << ( iter * m );

      next_iter = 0;
      for ( int i=n-1;  i>0;  i-- ) if ( i>iter && cand_2d[i] ) next_iter = i;

      iter = next_iter;
   end
endmodule
```

Problem 2: [25 pts] The point of the `gt` module in the previous problem was to reduce cost, just in case the synthesis program didn't notice that the cost of computing each of `n-1>iter`, `n-2>iter`, ..., `2>iter`, `1>iter`, would be less than $n-1$ times the cost of computing one of them. The recursive module below computes these quantities and can be used for the `gt` module from the previous problem.

```
module gtd_rec #( int n = 16, int lgn = $clog2(n) )
   ( output logic [n-1:0] gt, input uwire [lgn-1:0] iter );
   localparam int nh = n / 2;    // Note: n must be a power of 2.
   if ( n == 2 ) begin
      assign gt[0] = 0;
      assign gt[1] = !iter[0];
   end else begin
      uwire [nh-1:0] gtlo;
      gtd_rec #(nh) glo( gtlo, iter[lgn-2:0] );
      localparam logic [nh-1:0] zeros = 0, ones = -1;
      assign gt = iter[lgn-1] ? { gtlo, zeros } : { ones, gtlo };
   end
endmodule
```

(*a*) Show the hardware that will be inferred for this module for an arbitrary value of n. In this case, do not show what is inside the recursively instantiated module.

☐ Show hardware for arbitrary $n > 2$. (Don't show recursive module contents.)

(*b*) There should be a significant optimization opportunity in the hardware above. Show it.

☐ Show how the hardware will be optimized. The result should be AND, OR, and other basic logic gates.

(*c*) Show the hardware that will be inferred for $n = 8$ after elaboration. That is, show the hardware inside all of the recursive instantiations.

☐ Show hardware for $n = 8$. Show the contents of all recursively instantiated modules.

(*d*) Compute the cost and delay using the simple model. Show these in terms of n assuming that n is a power of 2.

☐ Cost and ☐ delay in terms of n.

Problem 3: [20 pts] Consider the module below.

```
module misc #( int n = 8 )
   ( output logic [n-1:0] a, g, e,
     input uwire [n-1:0] b, c, j, f,    input uwire clk );

   logic [n-1:0] z;

   always_ff @( posedge clk ) begin
      a <= b + c;   // Note: nonblocking assignment.
      z = a + j;
      g = z;
   end

   always_comb begin
      e = a * f;
   end

endmodule
```

(*a*) Show the hardware that will be inferred for the module above.

☐ Show inferred hardware. ☐ Pay attention to what is and is not a register. ☐ Clearly show module ports.

```
module misc #( int n = 8 )
   ( output logic [n-1:0] a, g, e,
     input uwire [n-1:0] b, c, j, f,    input uwire clk );

   logic [n-1:0] z;

   always_ff @( posedge clk ) begin   // Code Position Label: alf
      a <= b + c;   // Note: nonblocking assignment.
      z = a + j;
      g = z;
   end

   always_comb begin                  // Code Position Label: alc
      e = a * f;
   end

endmodule
```

(*b*) Suppose that the event queue is empty at $t = 10$ when simulating the module above. Show the contents of the event queue for the code above based on the following changes: At $t = 10$ `j` changes. At $t = 12$ `clk` changes from 0 to 1. At $t = 14$ `f` changes.

☐ Show the state of the event queue from $t = 10$ until it is empty.

Problem 4: [10 pts] Answer each question below.

(*a*) The module below is not compilable. Explain why and fix it based on what it looks like it is trying to do.

```
module more
  ( input uwire [5:0] w,
    input uwire [w-1:0] a, b,
    output uwire [w:0] s );

   assign s = a + b;

endmodule
```

☐ Fix the problem.

☐ Describe the problem:

(*b*) The module below is supposed to count cycles but it won't work as written. Describe the problem and fix it.

```
module tic_toc
  ( output logic [7:0] cycles,
    input uwire clk, reset );

   always_comb begin

      if ( reset ) cycles = 0;
      else if ( clk ) cycles = cycles + 1;

   end

endmodule
```

☐ Describe the problem:

☐ Fix the problem.

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

(*a*) Appearing below is synthesis data showing the clock period of degree-m sequential workfront multipliers and degree-m sequential regular (dm) multipliers for sizes $m = 1$, $m = 2$, $m = 4$, and $m = 8$.

Module Name	Area	Period Target	Period Actual	Total Latency
mult_seq_wfront_m_w32_m1	191334	1000	3766	241024
mult_seq_wfront_m_w32_m2	205303	1000	3857	123424
mult_seq_wfront_m_w32_m4	260182	1000	5266	84256
mult_seq_wfront_m_w32_m8	351910	1000	7031	56248
mult_seq_dm_w32_m1	246818	1000	31113	995616
mult_seq_dm_w32_m2	279486	1000	30994	495904
mult_seq_dm_w32_m4	314724	1000	32127	257016
mult_seq_dm_w32_m8	408659	1000	31251	125004

As m increases the clock period of the workfront multiplier increases by a significant amount, while the period of the sequential multiplier barely changes. Why?

☐ Why does the workfront period increase so much more than that of the regular multiplier?

Let $p_w(m)$ and $p_r(m)$ denote the clock period of the degree-m workfront and regular multipliers. Show expressions for $l_w(m)$ and $l_r(m)$, the latencies of these multipliers.

☐ Finish the following expression for latency: $l_w(m) = p_w(m)$

☐ Finish the following expression for latency: $l_r(m) = p_r(m)$

(*b*) The reasoning in the statement below is, as of this writing, incorrect. Provide the correct reason to not spend time on multiplier modules.

"One should not spend time trying to develop efficient multiplication hardware because the synthesis program is very good at optimizing logic and will synthesize something at least as good as a human can."

☐ When working on a design that makes heavy use of multiplication one should just use multiplication operators and not try to implement your own because:

(*c*) Sequential multipliers S0 and S1 have the same latency and cost, but the clock period for S1 is lower than S0.

☐ Which is preferred? ☐ Explain.

Pipelined multipliers P0 and P1 have the same latency and cost, but the clock period for P1 is lower than P0.

☐ Which is preferred? ☐ Explain.

(*d*) In the module below notice that `cand_2d` is no longer available. Modify the line updating `accum` to use `cand` instead.

```
module mult_seq_dm #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,
     input uwire [w-1:0] plier, cand, input uwire clk);

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   // uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg:1] iter;
   logic [2*w-1:0] accum;

   always @( posedge clk ) begin

      if ( iter == iter_lg'(iterations) ) begin
        prod = accum; accum = 0; iter = 0;
      end

      //    Fix line below

      accum += plier *       cand_2d[iter]   << ( iter * m );

      iter++;
   end
endmodule
```

8 Fall 2017

Name _____

Digital Design using HDLs
EE 4755
Midterm Examination

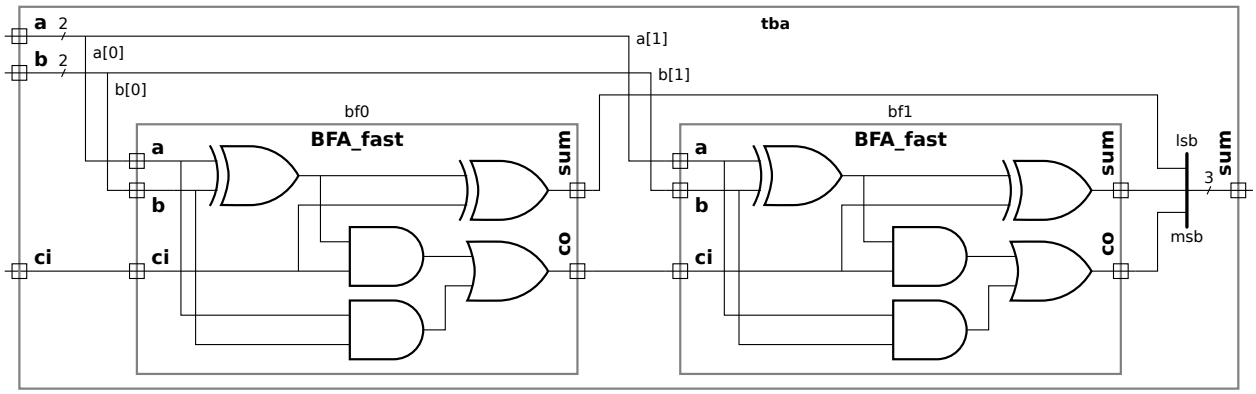
Monday, 16 October 2017 9:30–10:20 CDT

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

Alias _____

Good Luck!

Problem 1: [20 pts] Write a Verilog description of the hardware illustrated below. The description **must include the modules and instantiations as illustrated**. The description can be behavioral or structural, but it must be synthesizable.



☐ Verilog corresponding to illustrated hardware.

☐ Show instantiations, ☐ Verilog for instantiated module(s), ☐ and all module ports.

Problem 2: [20 pts] Appearing below is a partially completed recursive description of an $n = 2^b$-input, w-bit multiplexor, which is a generalized version of the multiplexors appearing in Homework 1. Complete it.

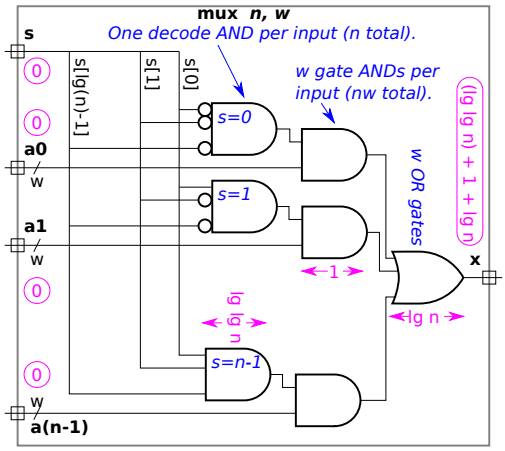
☐ Fill in the condition and code for the terminating case.

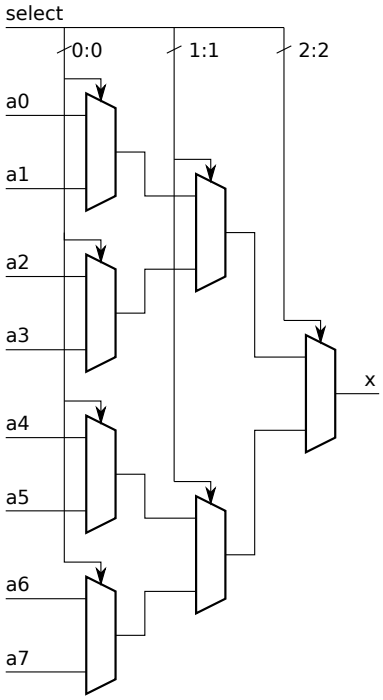
☐ Complete recursive case, including the instantiation port and parameter connections (look for FILL IN).

```
module muxn #( int w = 5, int b = 4, int n = 1 << b )
   ( output uwire [w-1:0] x,  input uwire [b-1:0] sel, input uwire [w-1:0] a[0:n-1] );

   if (         )  // Terminating Case Condition              <----  ☐  FILL IN
     begin
        // Terminating Case




     end else begin
        // Recursive Case

        uwire [w-1:0] y[2];

        // Instantiate two n/2-input muxen, and connect each to half the inputs.
        //
        //         ----      ----                               <----  ☐  FILL IN
        muxn #( .w(    ), .b(    ) ) mlo( y[0],  sel[b-2:0],  a[   0 : n/2-1 ] );



        //         ----      ----                    -----      <----  ☐  FILL IN
        muxn #( .w(    ), .b(    ) ) mhi( y[1],  sel[     ],  a[ n/2 : n-1   ] );









        // Instantiate one 2-input mux.
        //
        //         ----      ----       --------------------   <----  ☐  FILL IN
        muxn #( .w(    ), .b(    ) ) m2(                    )



   end

endmodule
```

Problem 3: [20 pts] Appearing below to the right is an 8-input multiplexor constructed from 2-input multiplexors using the technique from Homework 1 and from the previous problem. Call a multiplexor constructed this way a *tree mux*. Appearing below to the left is a diagram showing a *flat mux*, the kind usually used in class. The flat mux diagram shows a timing analysis based on the simple model, and some details about cost.

For reference: $\sum_{i=0}^{b-1} a2^i = a(2^b - 1)$. Assume that n is a power of 2.





(*a*) Compute the cost of an n-input, w-bit flat mux using the simple model and without optimization.

☐ Cost of flat mux **in terms of** n **and** w.

(*b*) Compute the cost of an n-input, w-bit tree mux using the simple model.

☐ Cost of tree mux **in terms of** n **and** w. ☐ Describe assumptions made about 2-input mux implementation.

(*c*) Compute the delay of an n-input, w-bit tree mux using the simple model.

☐ Delay of tree mux **in terms of** n **and** w.

Problem 4: [15 pts] Show the hardware that will be synthesized for the modules below.

(*a*) Show the hardware that will be inferred for the module below, including the minimum number of bits in each wire. Assume that `sqrt` is defined in a library somewhere.

```
module wqf
  #( int w = 16 )
   ( output logic signed [2*w-1:0] rad,
     output uwire [31:0] srad,
     input uwire [w-1:0] a, b, c );

   sqrt #(32,2*w) s1(srad,rad);

   always_comb begin

      rad = b*b - 4 * a * c;
      if ( rad < 0 ) rad = 0;

   end

endmodule
```

☐ Show inferred hardware. ☐ Show minimum correct bit widths.

(*b*) Show the hardware that will be inferred for the module below.

```
module sort2 #( int w = 4 )
   ( output logic [w-1:0] x[2], input uwire [w-1:0] a[2] );

   always_comb begin

      for ( int i=0; i<2; i++ ) x[i] = a[i];
      if ( a[0] < a[1] ) begin x[0] = a[1]; x[1] = a[0]; end

   end

endmodule
```

☐ Show inferred hardware.

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

(*a*) The `mux2` module below uses implicit structural code. Modify it so that it uses behavioral (procedural) code.

```
module mux2 #( int w = 16 )
   ( output uwire [w-1:0] x,
     input uwire s, input uwire [w-1:0] a,b );

   assign  x = s == 0 ? a : b;

endmodule
```

☐ Modify so that is procedural. ☐ Change ports if necessary.

(*b*) Modify the module port and parameter declarations below so that the Verilog is correct. Do not modify the contents of the module itself. Note that `opt` is not defined, but that it should be. *Note: In the original exam* `assign` *was omitted from the module body, making the problem impossible to solve.*

```
module sum_or_dff
  #( int w = 16 )
   ( output uwire [w-1:0] x,
     input uwire [w-1:0] a, b );

   if ( opt == 0 ) assign x = a+b; else assign x = a-b;

endmodule
```

☐ Modify port and parameter declarations for correctness.

Problem 6: [15 pts] Answer each question below.

(*a*) Why is `always_comb` preferred over `always @( x or y or .. )` when describing combinational logic?

☐ `always_comb` preferred because ...

☐ What is the risk with `always @( x or y or .. )`?

(*b*) Describe what the technology mapping step of synthesis is, and the kind of optimizations that need to be performed after technology mapping.

☐ Technology mapping is:

☐ Optimizations that must be performed after technology mapping:

(*c*) The module below adds a real and an integer and assigns the sum (in real format) to its output. It is valid Verilog but is not synthesizable by Owr EDA software. So, you call Owr EDA and ask, "why not?". They answer, "because it is impossible to add an integer to a real." Is that the real reason? Explain.

```
module plusri (output real sum, input real a, input [20:0] x);
   assign       sum = a + x;
endmodule
```

☐ Reason a+x not synthesizable by Owr EDA software:

Name _____

Digital Design using HDLs
LSU EE 4755
Final Examination

Wednesday, 6 December 2017 15:00-17:00 CST

Problem 1	_____	(15 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(30 pts)
Exam Total	_____	(100 pts)

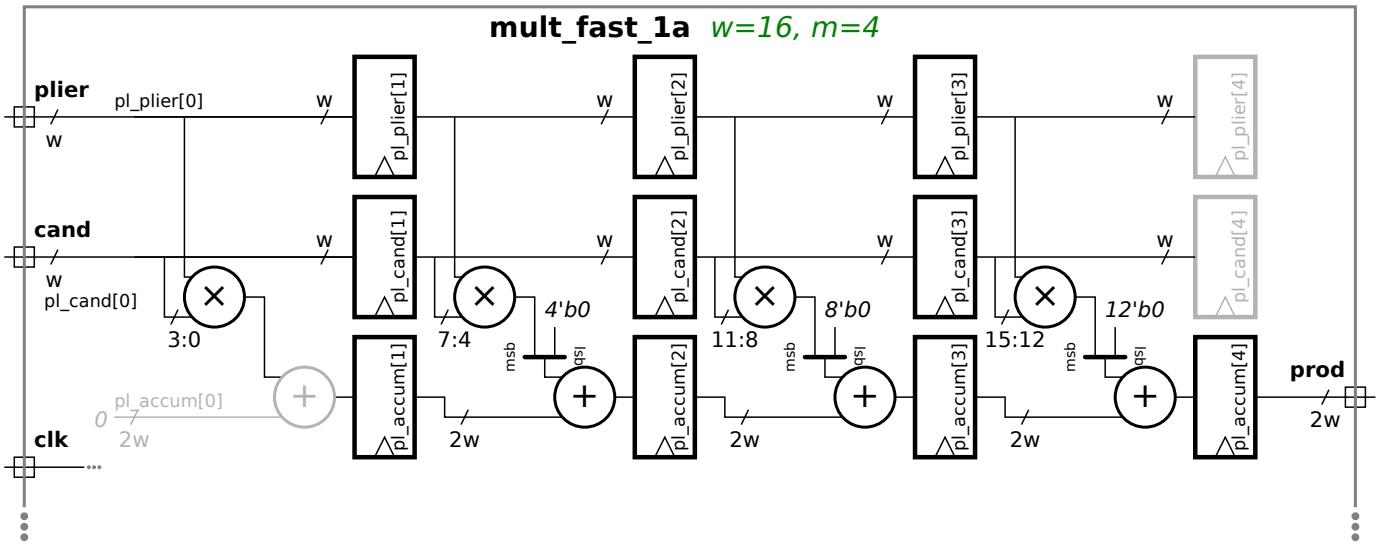
Alias _____

Good Luck!

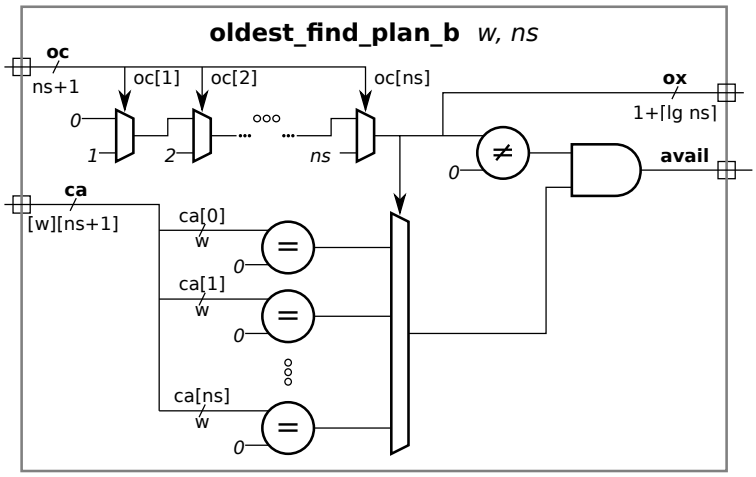
Problem 1: [15 pts] The Verilog code below is the solution to Problem 1a of Homework 7. Below that is the hardware **for a slightly different pipelined multiplier**. Modify the hardware to match the Verilog code. Changes need to be made for each line commented DIFFERS.

☐ Modify hardware to reflect Verilog.

```
module mult_fast_1a #( int w = 16, int m = 4 )
   ( output uwire [2*w-1:0] prod,
     output uwire out_avail,       input uwire clk, in_valid,     //  ☐ DIFFERS
     input uwire [w-1:0] plier, cand );
   localparam int nstages = ( w + m - 1 ) / m;
   logic [2*w-1:0] pl_accum[0:nstages];
   logic [w-1:0] pl_plier[0:nstages], pl_cand[0:nstages];
   logic pl_occ[0:nstages];                                          //  ☐ DIFFERS

   assign prod = pl_accum[nstages];
   assign out_avail = pl_occ[nstages];                               //  ☐ DIFFERS

   always_ff @( posedge clk ) begin
      pl_occ[0] = in_valid;                                          //  ☐ DIFFERS
      pl_accum[0] = 0;    pl_plier[0] = plier;    pl_cand[0] = cand;

      for ( int stage=0; stage<nstages; stage++ ) begin
         pl_plier[stage+1] <= pl_plier[stage];
         pl_accum[stage+1] <= pl_accum[stage] + ( pl_plier[stage]
             * pl_cand[stage][m-1:0] << stage*m );                   //  ☐ DIFFERS
         pl_cand[stage+1]  <= pl_cand[stage] >> m;                   //  ☐ DIFFERS
         pl_occ[stage+1]   <= pl_occ[stage];                         //  ☐ DIFFERS
      end
   end
endmodule
```



Problem 2: [25 pts] Module `oldest_find_plan_b`, illustrated below, is based on **an alternative solution** to Homework 7 Problem 1b. Below the hardware illustration is incomplete Verilog code for this module. The Verilog code uses abbreviated names, such as `ns`, comments show the original names from the assignment, such as `nstages`. Complete the module. *Note: This problem can be solved without having ever seen Homework 7, though not as quickly.*



☐ Complete the module so that it matches the hardware above.

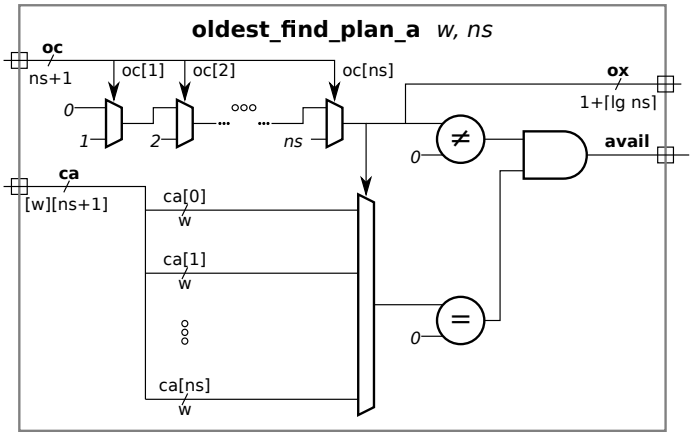
```
module oldest_find_plan_b
  #( int w = 15, int ns = 3              /* nstages */ )
   ( output logic [$clog2(ns):0] ox,     // oldest_idx
     output uwire avail,                 // out_avail
     input uwire oc[0:ns],               // pl_occ
     input uwire [w-1:0] ca[0:ns] );     // pl_cand
```

```
endmodule
```

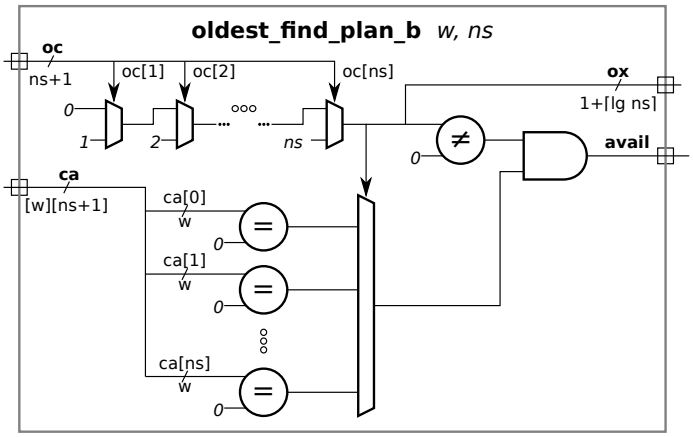
Problem 3: [20 pts] Appearing below are two variations on the oldest index module from the previous problem. The Plan A version is based on the code from the posted Homework 7 solution. The Plan B module is slightly different.

(*a*) Compute the cost of each module based on the simple model after optimizing for constant values. Use symbol w (for `w`) and n (for `ns`). Base the cost of an α-input, β-bit multiplexor on the tree (recursive) implementation. Recall that the tree implementation consists of $\alpha - 1$ two-input multiplexors arranged in a tree.

☐ Plan A cost in terms of w and n. ☐ Show cost components on diagram, such as cost of big mux, ☐ don't forget to account for the constant inputs, and ☐ for the number of bits in each wire.

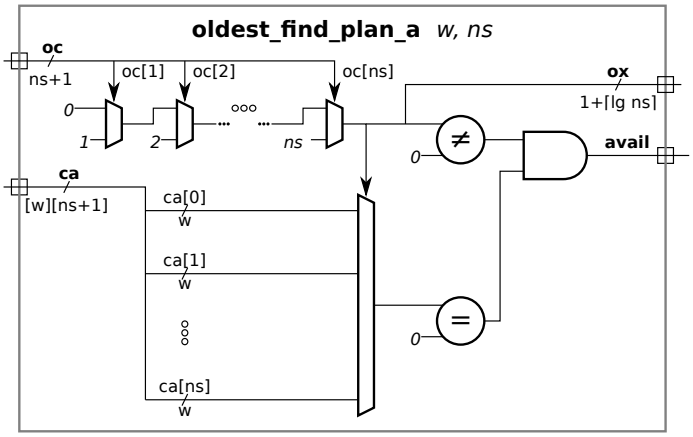


☐ Plan B cost in terms of w and n. ☐ Show cost components on diagram, such as cost of big mux, ☐ don't forget to account for the constant inputs and, ☐ for the number of bits in each wire.

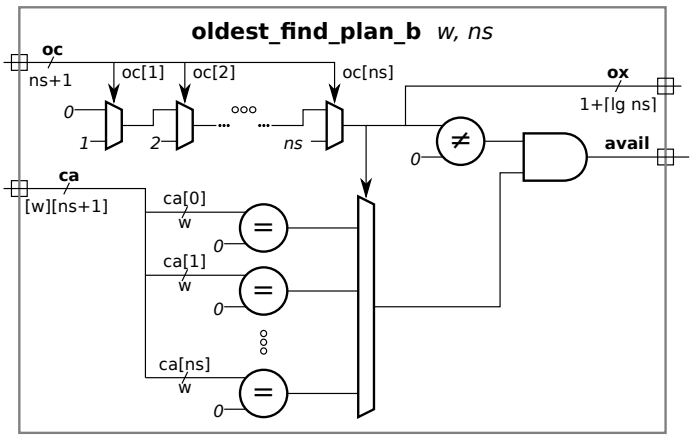


(*b*) Show the delay along all paths and show the critical path. Compute delay based on the simple model after optimizing for constant values. Use the tree mux described in the previous part.

☐ Plan A: ☐ show delay along all paths, ☐ highlight the critical path, ☐ and show the delay through each component. Show these ☐ in terms of w and n, and ☐ account for constant inputs such as the zeros in the equality units.



☐ Plan B: ☐ show delay along all paths, ☐ highlight the critical path, ☐ and show the delay through each component. Show these ☐ in terms of w and n, and ☐ account for constant inputs such as the zeros in the equality units.



Problem 4: [10 pts] Explain why each of the modules below is not synthesizable by Cadence Encounter (or similar tools) and modify the code so that it is *without changing what the module does*. *Note: The warning about not changing what the module does was not in the original exam.*

```
module one_run #( int w = 16, int lw = $clog2(w) )
   (output logic all_1s, input uwire [w-1:0] a, input uwire [lw:0] start, stop );
   always_comb begin

      all_1s = 1;

      for ( int i=start; i<stop; i++ )
        all_1s = all_1s && a[i];

   end
endmodule
```

☐ Reason code above is not synthsizable:

☐ Modify code so that it is.

```
module running_sum #( int w = 32 )
   ( output logic [w-1:0] rsum,
     input uwire [w-1:0] a,     input uwire reset, clk );

   always @( posedge clk ) begin
      if ( reset ) rsum <= 0;
   end

   always @( posedge clk ) begin
      rsum <= rsum + a;
   end

endmodule
```

☐ Modify code so that it is synthsizable.

☐ Reason code above was not synthsizable:

☐ Explain assumption about intended behavior of this module.

Problem 5: [30 pts] Answer each question below.

(*a*) Show when each piece of code below executes (use the `C` labels) up until the start of `C5c`, and show when and in which region each piece is scheduled. See the table below.

```
module eq;
   logic [7:0] a, b, c, d, x, y, x1, x2, y1, y2, z2;
   always_comb begin          // C1
      x1 = a + b;
      y1 = 2 * b;
   end
   assign x2 = 100 + a + b;   // C2
   assign y2 = 4 * b;         // C3
   assign z2 = y2 + 1;        // C4
   initial begin
      //                         C5a
      a = 0;
      b = 10;
      #2;
      //                         C5b
      a = 1;
      b <= 11;
      #2;
      //                         C5c
      a = 2;
      b = 12;
   end
endmodule
```

☐ Continue the diagram below so that it shows scheduling up to the point where `C5c` executes.

Step 1	Step 2	Step 3
$t = 0$	$t = 0$	$t = 0$
Active **C5a** ↗	*Active*	*Active*
Inactive	*Inactive* C1 C2 C3	
NBA	*NBA*	
	$t = 2$	
	Inactive C5b	

(*b*) Which of the two modules does what it looks like it's trying to do? Explain.

```
module sa1(input logic [7:0] a, b, c, d, output wire [7:0] x, y );
   assign x = a + b;
   assign y = 2 * x;
   assign x = c + d;
endmodule

module sa2(input logic [7:0] a, b, c, d, output logic [7:0] x, y );
   always_comb begin
      x = a + b;
      y = 2 * x;
      x = c + d;
   end
endmodule
```

☐ Module that is probably correct is:

☐ Major problem with other module.

☐ Provide a possible wrong answer from other module.

(*c*) Define throughput and latency and indicate where each is preferred. Provide examples appropriate for pipelined systems.

☐ Throughput is:

☐ For example:

☐ Latency is:

☐ For example,

☐ If the goal is to improve throughput is higher throughput good or bad?

☐ If the goal is to improve latency, is higher latency good or bad?

☐ In what situation is latency more important than throughput?

(*d*) When we synthesize we specified a target delay, for example, 400 ns.

☐ Does specifying a larger delay mean that there will be less optimization?

☐ Explain.

9 Fall 2016

Name _____

Digital Design using HDLs
EE 4755
Midterm Examination

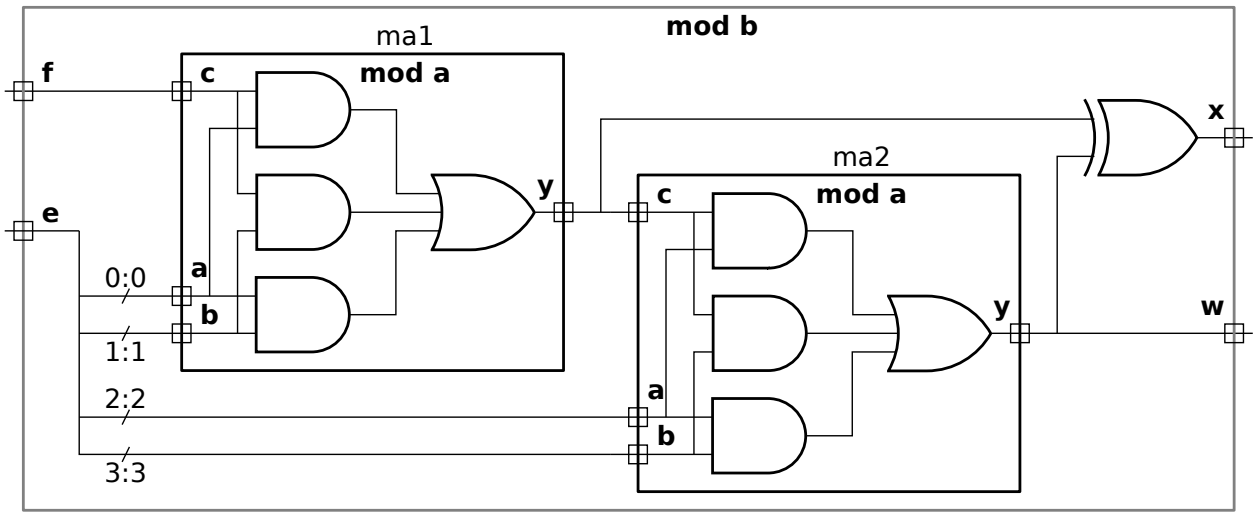
Friday, 21 October 2016 12:30–13:20 CDT

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

Alias _____

Good Luck!

Problem 1: [20 pts] Write a Verilog description of the hardware illustrated below. The description must include the modules and instantiations as illustrated. The description can be behavioral or structural, but it must be synthesizable.



☐ Verilog corresponding to illustrated hardware.

☐ Show instantiations, ☐ Verilog for instantiated module(s), ☐ and all module ports.

Problem 2: [20 pts] Appearing below is the `lookup_elt` module from Homework 4 and following that an incomplete module named `match_amt_elt`. Complete `match_amt_elt` so that the value at output port `md` is set to the number of bits in `clook` that match corresponding bits in `celt`. For example, if `clook=5'b00111` and `celt=5'b00111` then `md` should be 5, if `clook=5'b00101` and `celt=5'b00111` then `md` should be 4, and if `clook=5'b11000` and `celt=5'b00111` then `md` should be 0. Code must be synthesizable, but can be behavioral or structural.

☐ Complete the module so that `md` is set to the number of matching bits.

☐ Make sure that `md` is declared with sufficient width.

```
module lookup_elt #( int charsz = 32 )  // This module is for reference only.
   ( output logic match, input uwire [charsz-1:0] char_lookup, char_elt );
   always_comb match = char_lookup == char_elt;
endmodule
```

```
module match_amt_elt
  #( int charsz = 32 )
   ( output logic            md,
     input uwire [charsz-1:0] clook,
     input uwire [charsz-1:0] celt);
```

```
endmodule
```

Problem 3: [20 pts] Show the hardware that will be synthesized for the modules below.

(*a*) Show the hardware that will be inferred for the module below. Show `acme_ip_sqrt` as a box.

```
module vmag( output uwire [31:0] mag,  input uwire signed [31:0] v [3] );

   logic [63:0] sos;
   acme_ip_sqrt #(32) s1(mag,sos);

   always_comb begin
      sos = 0;
      for ( int i=0; i<3; i++ ) sos += v[i] * v[i];
   end

endmodule
```

☐ Show inferred hardware. ☐ Don't forget `acme_ip_sqrt`.

☐ Clearly show input and output ports of `vmag`.

Problem 3, continued:

(*b*) Show the hardware that will be inferred for the module below, before and after optimization. *Note: In the original exam the input was named* `vi`.

```
module min_elt( output logic [1:0] idx_min, input uwire signed [31:0] v [3] );
   always_comb begin
      idx_min = 0;
      for ( int i=1; i<3; i++ ) if ( v[i] < v[idx_min] ) idx_min = i;
   end
endmodule
```

☐ Show inferred hardware. ☐ Clearly show input and output ports.

☐ Show hardware after some optimization.

Problem 4: [10 pts] Appearing in this problem are several variations on a counter.

(*a*) Show the hardware inferred for each counter below.

```
module ctr_a( output uwire [9:0] count, input clk );

   logic [9:0] last_count;
   assign count = last_count + 1;
   always_ff @( posedge clk ) last_count <= count;

endmodule

module ctr_b( output logic [9:0] count, input clk );

   uwire [9:0] next_count = count + 1;
   always_ff @( posedge clk ) count <= next_count;

endmodule
```

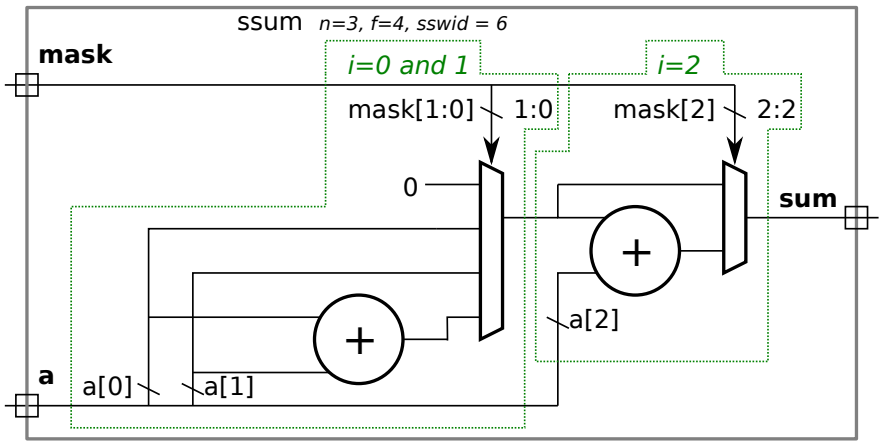
☐ Inferred hardware for ☐ `ctr_a` and ☐ `ctr_b`.

(*b*) There is a big difference in the timing of the outputs of `ctr_a` and `ctr_b`. Explain the difference and illustrate with a timing diagram.

☐ Difference between two modules. ☐ Timing Diagram.

Problem 5: [10 pts] Appearing below is the solution to the 2015 midterm exam Problem 2. Estimate the cost of this module as illustrated but use variable s for the number of bits in `sum` (shown as `sswid`) and in each `a` element (shown as parameter `f`). Assume that the cost of a BFA is 10 units and that the cost of a n-input AND and OR gate is $n-1$ units. Take into account the 0 input to one of the multiplexors.

```
module ssum #( int n = 3,      int f = 4,     int swid = f + $clog2(n) )
             ( output logic [swid-1:0] sum,
               input uwire [n-1:0] mask,      input uwire [f-1:0] a[n] );
   always @* begin
      sum = 0;
      for ( int i=0; i<n; i++ ) if ( mask[i] ) sum += a[i];
   end
endmodule
```



☐ Cost of illustrated hardware. ☐ Account for 0 mux input.

Problem 6: [20 pts] Answer each question below.

(*a*) Show the values of the variables as indicated below:

```
module tryout();
   logic [15:0] a;
   logic [0:15] b;
   logic [3:0][3:0] e;
   logic [3:0]  x1, x2;

   initial begin

      a = 16'h1234;
      x1 = a[3:0];                    //  ☐  Value of x1 is:

      b = 16'h1234;
      x2 = b[0:3];                    //  ☐  Value of x2 is:

      e = 16'h1234;
      e[0] = e[0] + 'hf;              //  ☐  Value of e is:

      e = 16'h1234;
      e[0][0] = e[0][0] + 'hf;        //  ☐  Value of e is:

   end
endmodule
```

(*b*) Describe something that can be done during elaboration that cannot be done during simulation, and something that can be done during simulation, that cannot be done during elaboration.

☐ Something that can be done during elaboration but **not during simulation** is:

☐ Something that can be done during simulation but **not during elaboration** is:

(*c*) Appearing below are two alternatives for an integer division module, *Plan A* and *Plan B*. Both are impractical, but Plan A is not even synthesizable.

```
module div_plan_a #( int w = 16 ) ( output logic [w-1:0] quo, input uwire [w-1:0] a, b );
   always_comb begin
      for ( quo = 0;  a > quo * b;  quo++ );
   end
endmodule

module div_plan_b #( int w = 16 ) ( output logic [w-1:0] quo, input uwire [w-1:0] a, b );
   localparam int LIMIT = 1 << w;
   always_comb begin
      quo = 0;
      for ( int i=0; i<LIMIT; i++ ) if ( a < i * b ) quo++;
   end
endmodule
```

☐ Why isn't Plan A synthesizable? Be specific as possible.

☐ What might be a practical objection to the Plan B approach?

(*d*) The `magfp` module below is not synthesizable due to the use of the `real` data type. How would the module need to be changed so that it would be synthesizable and would operate on floating-point values.

```
module magfp( output real mag, input real vi [3] );
   real sos;
   sqrt #(32) s1(mag,sos);
   always_comb begin
      sos = 0;
      for ( int i=0; i<3; i++ ) sos += vi[i] * vi[i];
   end
endmodule
```

☐ Show changes to port declaration for synthesizability.

☐ Explain with a few examples how the rest of the code would need to be changed.

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)

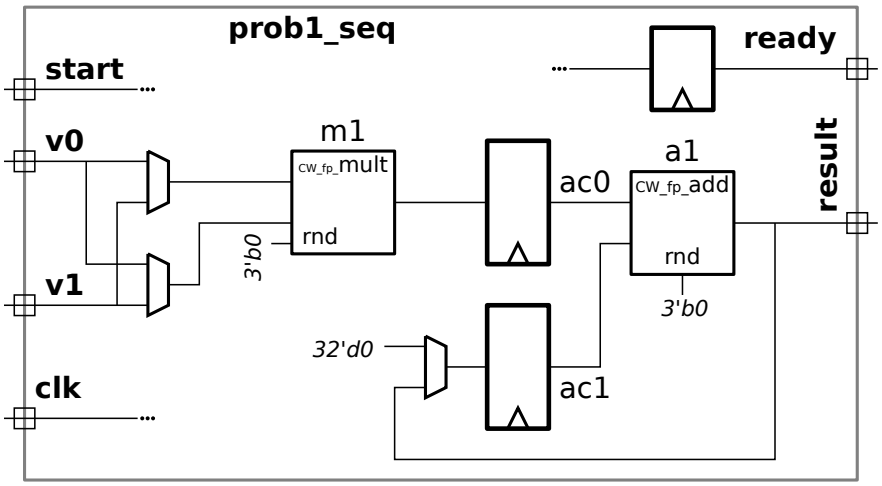
Alias _____

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.



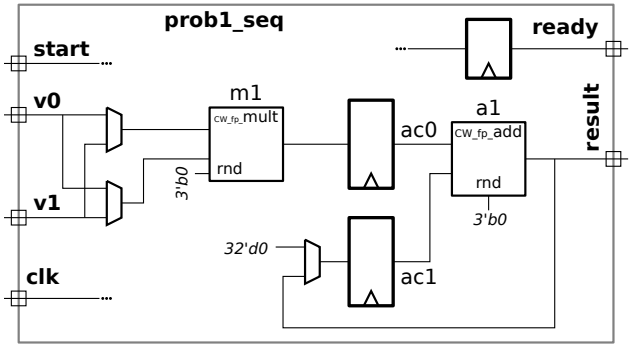
Don't modify this diagram, write Verilog code.

```
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.

- ☐ 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.

Don't modify, Verilog only.




```
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:

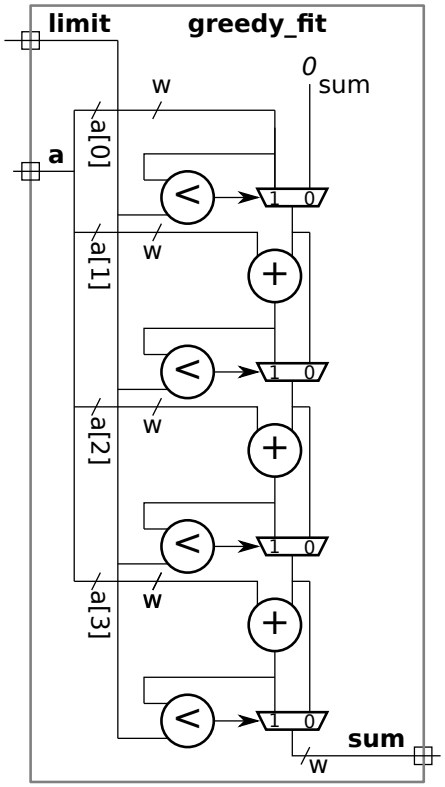
☐ Explain or show diagram.

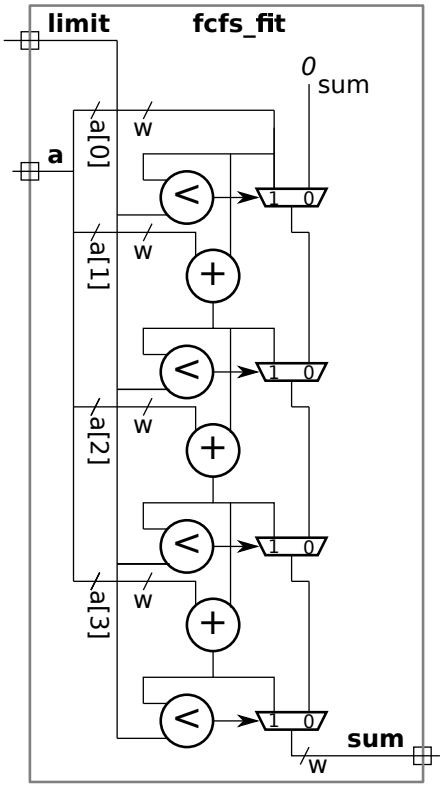
☐ Delay for a w-bit, n-input multiplexor is:

☐ 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.

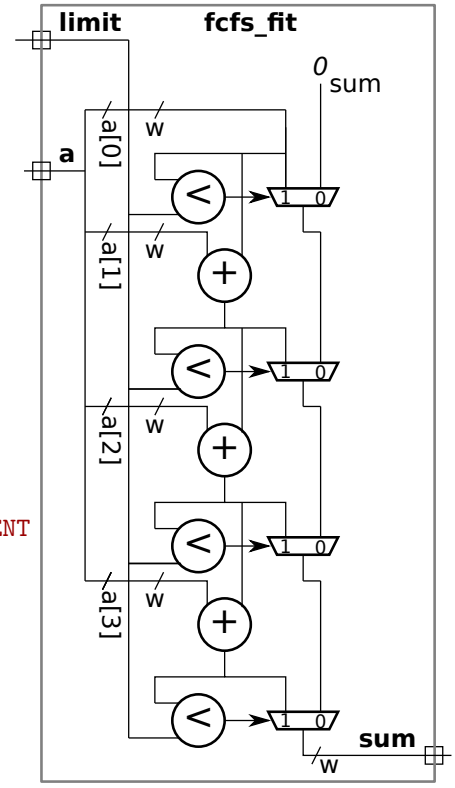




☐ Length of critical path for `greedy_fit` **in terms of** w. ☐ Show work for partial credit.

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

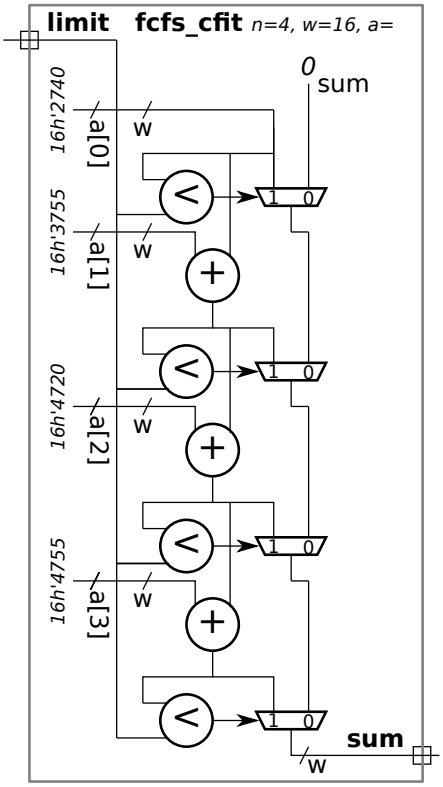
Problem 3: [15 pts] Complete the Verilog code so that it corresponds to the module shown.



☐ 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
```

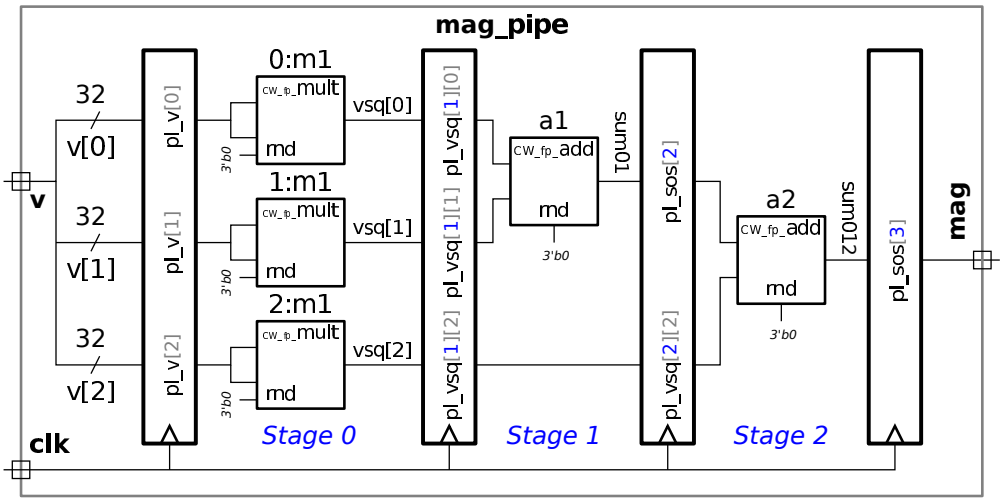
☐ Order of statements: a, b,

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.

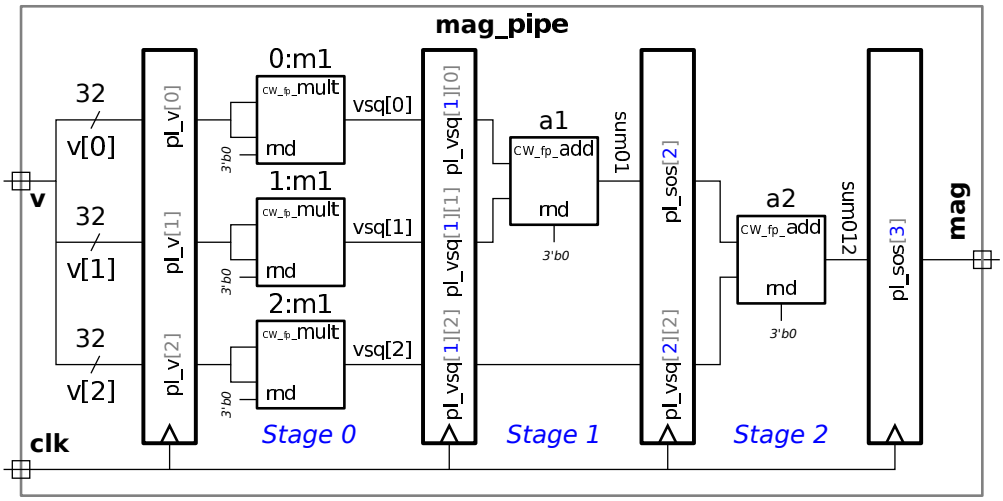
☐ Modify for lower cost based on faster adder.

☐ 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`.



10 Fall 2015

Name _____

Digital Design using HDLs
EE 4755
Midterm Examination

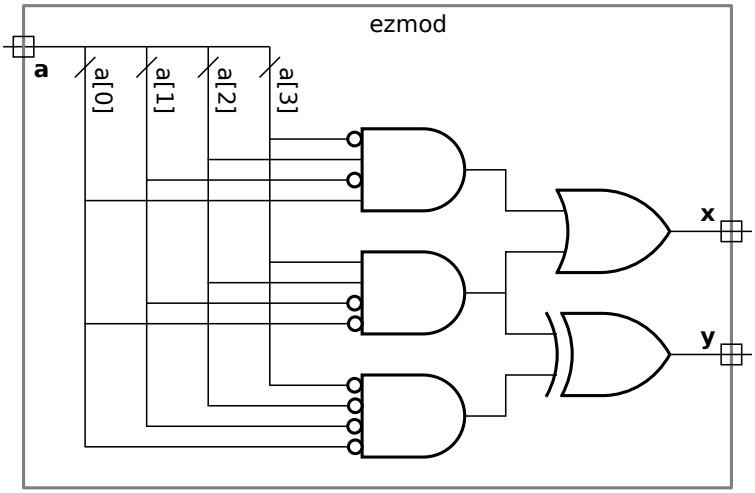
Wednesday, 28 October 2015 11:30–12:20 CDT

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

Alias _____

Good Luck!

Problem 1: [20 pts] Complete the Verilog description of the hardware illustrated below. It's okay—and a time saver—to use the == operator.



☐ Complete the port declarations.

☐ Complete the module.

```
module ezmod

  ( output                                          // DON'T FORGET
    input                                        );  // THE PORTS

endmodule
```

Problem 2: [20 pts] Consider the module below.

```
module ssum #( int n = 3,      int f = 4,     int swid = f + $clog2(n) )
             ( output logic [swid-1:0] sum,
               input uwire [n-1:0] mask,      input uwire [f-1:0] a[n] );
   always @* begin
      sum = 0;
      for ( int i=0; i<n; i++ ) if ( mask[i] ) sum += a[i];
   end
endmodule
```

(*a*) Show the hardware that will be synthesized without optimization and using default parameters.

☐ Hardware without optimization.

(*b*) Show the hardware that will be synthesized using the default parameters with optimization. In particular, try to make use of a four-input multiplexor for the first two iterations of the `i` loop.

☐ Hardware with optimization and using a four-input mux.

Problem 3: [20 pts] Appearing below is the `ssum` module from the previous problem and the start of a recursive version of the module, `ssum_rec`. Finish `ssum_rec` so that it performs the same computation, but does so using a tree connection of hardware rather than the linear connection that `ssum` describes. (For **partial credit only** use a generate loop to instantiate `ssum` modules of a fixed size; for full credit use recursion.)

```
module ssum #( int n = 3, int f = 4, int swid = f + $clog2(n) )
   ( output logic [swid-1:0] sum,
     input uwire [n-1:0] mask,       input uwire [f-1:0] a[n] );
   always @* begin
      sum = 0;
      for ( int i=0; i<n; i++ ) if ( mask[i] ) sum += a[i];
   end
endmodule
```

☐ Complete module so that it describes a tree structure specified using recursion.

```
module ssum_rec
  #( int n = 3, int f = 4, int swid = f + $clog2(n) )
   ( output logic [swid-1:0] sum,
     input uwire [n-1:0] mask,
     input logic [f-1:0] a [n-1:0] );
```

```
endmodule
```

Problem 4: [20 pts] Show the hardware that will be synthesized for the module below.

```
module yam( output logic [7:0] x, y, z,
            input uwire [7:0] a, b, c,   input uwire [1:0] op,   input uwire run, clk );
   logic [7:0] x1, x2, e;

   always_ff @( posedge clk ) begin
      e = b;
      z = a + b;
      if ( op == 0 )        e = z;
      else if ( op == 1 )   e = a + x;
      else if ( op == 2 )   e = a + x1;
      x2 = x1;
      x1 = x;
      if ( run ) x = e;
   end

   always_comb y = x1 + x2 - c;

endmodule
```

☐ Show hardware, including ☐ registers and ☐ module ports.

Problem 5: [20 pts] Answer each question below.

(*a*) Show the values of `a`, `b`, and `c` when the code reaches Point 1 and Point 2.

```
module short_answers;
   int a, b, c;
   initial begin
      a = 0;  b = 0; c = 0;
      a = 1;
      a <= 2;
      a <= #3 3;     //
      b = a + 10;    //   ---a---     ---b---     ---c---
      c <= a + 20;   //

      // Point 1:

      #1;

      // Point 2.
   end

   my_prog my_prog_instance(a,b,c); // Ignore for part (a).

endmodule
```

☐ At Point 1, values for ☐ `a`, ☐ `b`, and ☐ `c`.

☐ At Point 2, values for ☐ `a`, ☐ `b`, and ☐ `c`.

(*b*) The definition of the `my_prog` program from the previous part appears below. Show the contents of the Verilog event queue at Point 1 in the code from the previous part, include the effect of code in `short_answers` as well as `my_prog`. Show events in the form "$t = 1969$, region=NL-East, Resume Point 3" and "$t = 2015$, region=X, Update variable z," but use real region names.

```
program my_prog(input int a, b, c);
   initial forever @( a or b or c ) begin
      // Point 3;
      $display("Let's go Mets!");
   end
endprogram
```

☐ Contents of event queue at Point 1, show ☐ region names and ☐ time stamps.

(*c*) The module below is in explicit structural form, in which only primitive gates (and module instantiations) are used. Will the synthesis program synthesize exactly that arrangement of gates? Explain.

```
module bfa_structural( output uwire sum, cout, input uwire a, b, cin );
   uwire term001, term010, term100, term111;
   uwire ab, bc, ac;
   uwire na, nb, nc;
   not n1( na, a);
   not n2( nb, b);
   not n3( nc, cin);
   and a1( term001, na, nb, cin);
   and a2( term010, na, b,  nc);
   and a3( term100, a,  nb, nc);
   and a4( term111, a,  b,  cin);
   or o1( sum, term001, term010, term100, term111);
   and a10( ab, a, b);
   and a11( bc, b, cin);
   and a12( ac, a, cin);
   or o2( cout, ab, bc, ac);
endmodule
```

☐ Will synthesis program emit exactly these gates? ☐ Explain.

(*d*) Based on a hand analysis of `my_mut` we expect it to have a clock period of 12 ns. Shown below is an excerpt from the testbench for `my_mut` that includes the code for generating a clock. Assume that the Verilog time unit is set to 1 ns. How does the clock declaration below affect the timing of the synthesized hardware?

```
module testbench();
   logic clock;
   initial clock = 0;
   always #5 clock = !clock;
   // Other declarations omitted.
   my_mut woof(x,y,a,b,clock);
```

☐ The effect of the declaration of `clock` on timing of synthesized hardware is . . . ☐ because . . ..

Name _____

Digital Design using HDLs
LSU EE 4755
Final Examination

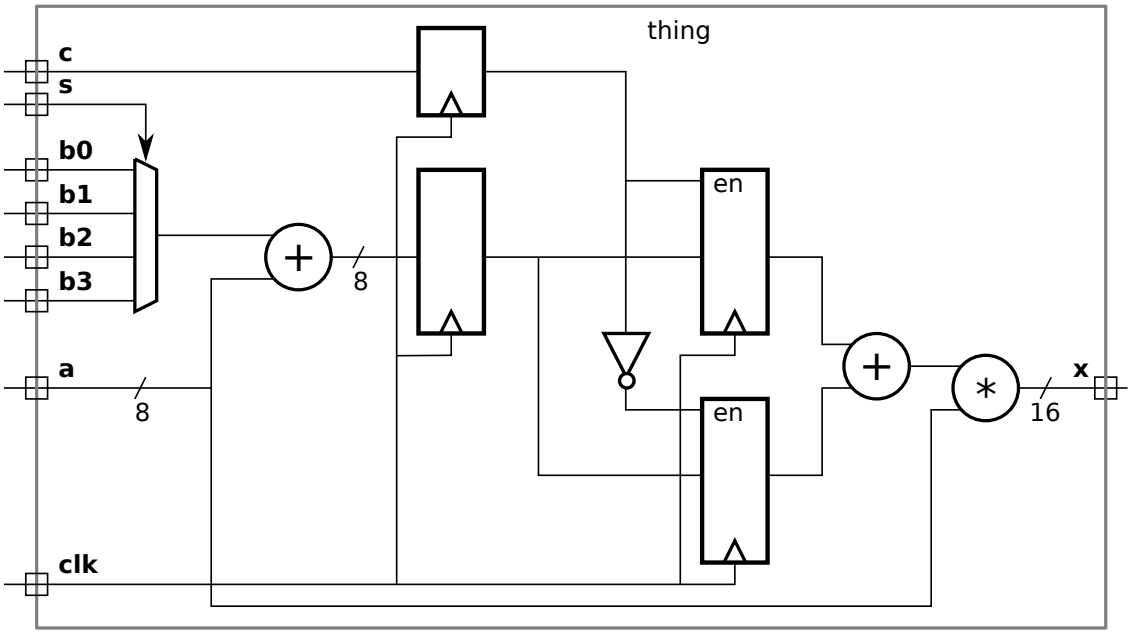
Saturday, 12 December 2015 12:30-14:30 CST

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

Alias _____

Good Luck!

Problem 1: [15 pts] Write a Verilog description of the hardware illustrated below.



☐ Verilog description of hardware including ☐ port declarations and ☐ port and other sizes.

Problem 2: [20 pts] The module below implements a simple memory module.

```
module smemory #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
   ( output uwire [dbits-1:0] rd_data,
     input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
     input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];

   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   assign rd_data = storage[rd_idx];

endmodule
```

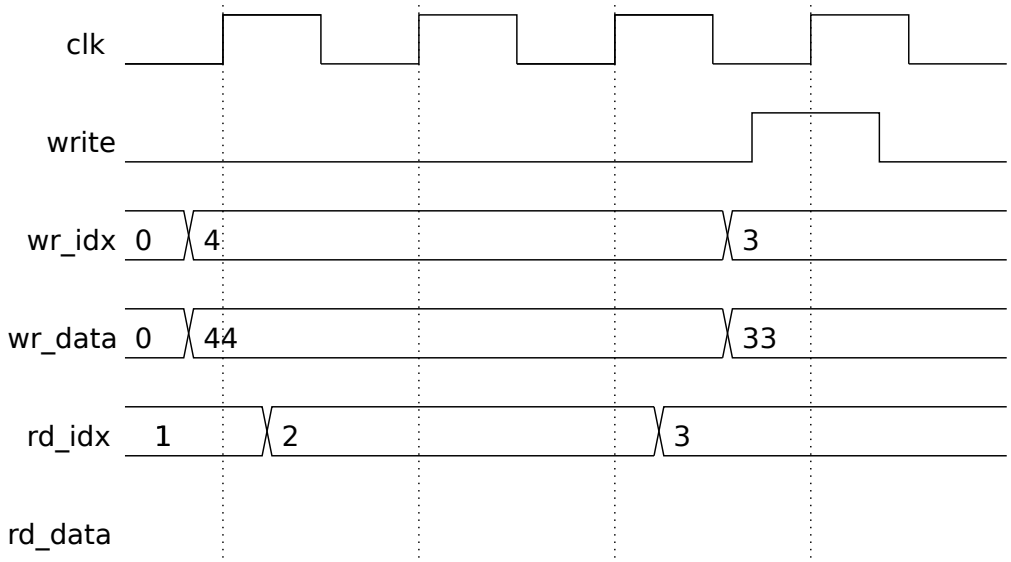
(*a*) Show the hardware that will be synthesized for this module when elaborated with `size_lg = 2`. Use registers, multiplexors, decoders, and basic gates. **Do not** use a memory module.

☐ Show synthesized hardware, including hardware for ☐ reading and ☐ writing.

Problem 2, continued: Appearing below is the module from the previous page.

```
module smemory #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
   ( output uwire [dbits-1:0] rd_data,
     input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
     input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];

   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   assign rd_data = storage[rd_idx];

endmodule
```

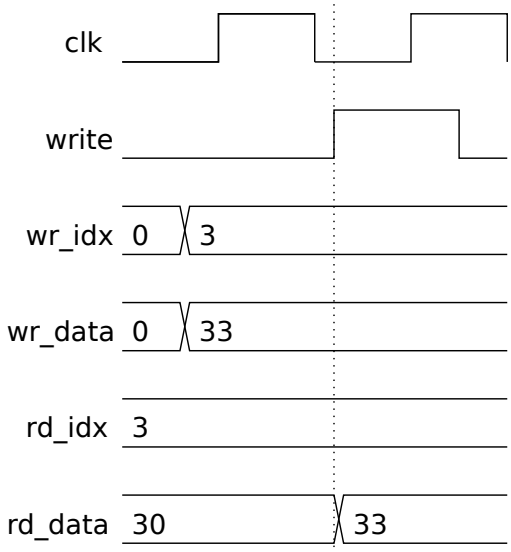
(*b*) Assume that initially location 1 (`storage[1]`) holds a 10, location 2 holds a 20, location 3 holds a 30, and so on. Complete the timing diagram below, consistent with this module.



☐ Complete `rd_data` row of timing diagram.

(*c*) Modify the module below (same as one on previous page) so that its behavior is consistent with the timing diagram to the right. That is, if the location being written is the same as the one being read the `rd_data` output shows the data on `wr_data`. If the locations don't match or nothing is being written the behavior is unchanged.

☐ Modify the module.




```
module smemory_bp #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
 ( output uwire [dbits-1:0] rd_data,
   input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
   input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];




   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;




   assign rd_data = storage[rd_idx];




endmodule
```

Problem 3: [20 pts] The module below and the similar one on the next page are like the memory module from the previous problem, except that their output is the sum of locations `rd_start`, `rd_start+1`, ..., `rd_start+rd_len-1`. Assume that `rd_start+rd_len <= size`.

```
module rsum_plan_a #( int sz_lg = 4, int ebits = 8, int size = 1 << sz_lg )
   ( output logic [ebits-1:0] sum,
     input [sz_lg-1:0] wr_idx,      input [ebits-1:0] wr_data,   input write,
     input [sz_lg-1:0] rd_start,    input [sz_lg-1:0] rd_len,    input clk    );

   logic [ebits-1:0] storage [size-1:0];

   // Don't show synthesized hardware for line below.
   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   // Plan A -- Show Synthesized Hardware for this Verilog
   always_comb begin
        sum = 0;
        for ( int i=0; i<size; i++ ) if ( i < rd_len ) sum += storage[ i + rd_start ];
     end

endmodule
```

(*a*) Show the hardware that will be synthesized for the `always_comb` block. Include basic optimizations, but don't optimize to the point where hardware is identical to Plan B (next page).

☐ Show not-too-optimized hardware for `sum`.

(*b*) Appearing below is Plan B for the module. Though we know it produces the same value for `sum` as Plan A, it might be synthesized into different hardware. Show the hardware synthesized for Plan B.

```
module rsum_plan_b #( int sz_lg = 4, int ebits = 8, int size = 1 << sz_lg )
   ( output logic [ebits-1:0] sum,
     input [sz_lg-1:0] wr_idx,      input [ebits-1:0] wr_data,   input write,
     input [sz_lg-1:0] rd_start,    input [sz_lg-1:0] rd_len,    input clk   );
   logic [ebits-1:0] storage [size-1:0];

   // Don't show synthesized hardware for line below.
   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   // Plan B -- Show Synthesized Hardware for this Verilog
   always_comb begin
        sum = 0;
        for ( int i=0; i<size; i++ )
          if ( i >= rd_start && i < rd_start + rd_len ) sum += storage[ i ];
     end
endmodule
```

☐ Show the hardware that will be synthesized for Plan B.

(*c*) Which one is better?

☐ Which is better, ◯ *Plan A* or ◯ *Plan B* .

☐ Explain, with a rough estimate of cost and timing.

Problem 4: [15 pts] Appearing below are excerpts based on the **cam_hash** module used in class, showing what we called the **hash_early** design. Recall that with the early hash design the hash function (in module **hash**) is computed before the positive clock edge while the lookup occurs after the positive edge. We assumed that the hash could be computed in about $\frac{1}{2}$ of our target clock period.

```
module cam_hash_exceprt
   ( output [dwid:1] out_data, output out_valid,        output ready,
     input [kwid:1] in_key,    input [dwid:1] in_data,
     input Cam_Command in_cmd, input clk);

   logic [kwid:1] b_key;
   logic [dwid:1] b_data;
   logic [hkey_size-1:0] b_hash;
   Cam_Command b_cmd;

   uwire [hkey_size-1:0] ohm_key_out;

   always_ff @( posedge clk ) begin
      b_key <= in_key;
      b_data <= in_data;
      b_cmd <= in_cmd;
      b_hash <= ohm_key_out;
   end

   hash #(kwid,num_sets_lg) our_hash_module( ohm_key_out, in_key );

   /// Hardware to find matching key below ...
```

(*a*) The early hash design requires that the external hardware has the right timing behavior. Show a timing diagram in which the timing behavior is correct for early hash, and one in which it is wrong. The "wrong" behavior should result in incorrect results using the early hash design, but correct results without the early hash design.

☐ Timing diagram showing ☐ correct and ☐ wrong behavior.

Problem 4, continued:

(*b*) Register `b_hash` saves the hashed version of `in_key`, and `b_key` holds the unhashed version. Why do we need the unhashed version?

☐ `b_key` is needed because ...

Problem 5: [10 pts] The Verilog below is part of a testbench (taken from `icomp.v`).

```
initial begin

   /// Watchdog – Stop simulation if it's taking too long.
   //
   fork begin

      automatic int cyc_limit = in_str.len() * 100;

      fork
         wait ( cycle_num == cyc_limit );
         wait ( tb_insert_done && tb_remove_done );
      join_any

      if ( cycle_num >= cyc_limit ) begin
         $write("Exceeded cycle limit, exiting.\n");
         $fatal(1);
      end

   end join_none

   // Below: Send data to module under test.
```

(*a*) Generically explain what a `fork` and `join` pair do (ignoring the code above).

☐ `fork` and `join` ...

(*b*) How would execution be effected if the last `join_none` were changed to `join_any`?

☐ Impact of changing `join_none` to `join_any` in code above.

(*c*) How would execution be effected if the inner `join_any` were changed to a `join_all`?

☐ Impact of changing `join_any` to `join_all` in code above.

Problem 6: [20 pts] Answer each question below.

(*a*) Suppose we would like our hardware to operate at a 1 GHz clock frequency. How do we tell the synthesis program? (The exact syntax is not important.)

☐ Method to tell synthesis program the clock frequency.

(*b*) The synthesis program will apply our target clock frequency to paths starting at launch points and ending at capture points. We could explicitly specify such points but if we don't it will use default launch and capture points. What are they?

☐ By default timing is computed for paths that start at:

☐ and end at:

(*c*) Suppose our target clock frequency is 1 GHz. What is the harm in telling the synthesis program to synthesize for 2 GHz? For 0.5 GHz?.

☐ Harm in specifying 2 GHz when we just need 1 GHz:

☐ Harm in specifying 0.5 GHz when we just need 1 GHz:

(*d*) The code below will inconsistently assign a variable. Explain why and fix the problem.

```
module short_ans( output logic [7:0] x, y, input [7:0] a, b, c, input clk);

   always @( posedge clk ) begin

      x = a + b;

   end

   always @( posedge clk ) begin

      y = x + c;

   end

endmodule
```

☐ Reason for inconsistent behavior:

☐ Fix problem.

(*e*) Describe the problem with the module below. How might it affect simulation?

```
module short_ans2( output logic [7:0] x, input [7:0] a, b, input reset);

   always_comb begin

      if ( reset ) x = a; else x = x + b;

   end

endmodule
```

☐ Problem with module.

☐ Impact on simulation.

11 Fall 2014

Name _____

Digital Design using HDLs
EE 4755
Midterm Examination

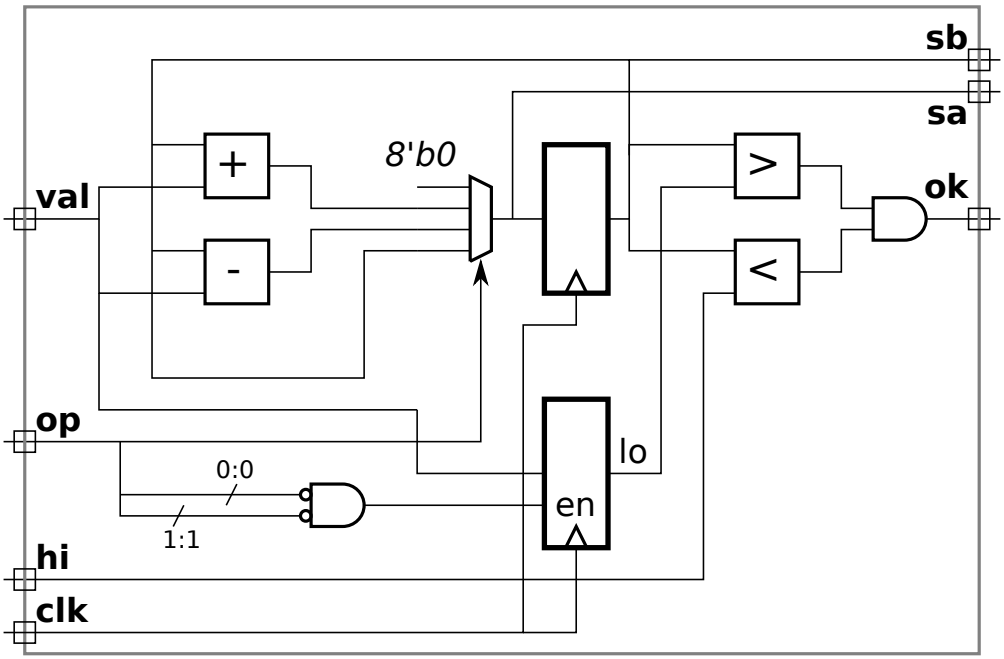
Monday, 10 November 2014 11:30–12:20 CST

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(15 pts)
Problem 5	_____	(13 pts)
Problem 6	_____	(22 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] Write a Verilog description of the hardware shown below.



☐ Write a Verilog module corresponding to the hardware above.

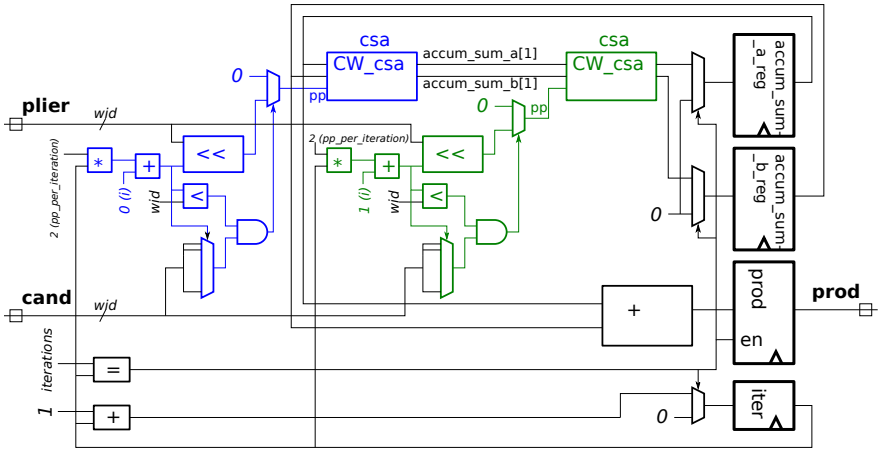
☐ Be sure to declare module ports and ☐ any wires and vars (`logic`) used inside.

☐ Pay attention to the differences between `lo` and `hi` and ☐ the differences between `sa` and `sb`.

Problem 2: [20 pts] Appearing below is the multiply circuit from the solution to Homework 3, in Verilog (slightly simplified) and as a diagram showing what hardware a synthesis program might infer.

```
module mult_seq_csa_m #( int wid = 16, int pp_per_cycle = 2 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier, input logic [wid-1:0] cand, input uwire clk);

   localparam int iterations = ( wid + pp_per_cycle - 1 ) / pp_per_cycle;
   localparam int iter_lg = $clog2(iterations);
   localparam int wid_lg = $clog2(wid);

   logic [iter_lg:0] iter;
   uwire [2*wid-1:0] accum_sum_a[0:pp_per_cycle], accum_sum_b[0:pp_per_cycle];
   logic [2*wid-1:0] accum_sum_a_reg, accum_sum_b_reg;

   assign           accum_sum_a[0] = accum_sum_a_reg;
   assign           accum_sum_b[0] = accum_sum_b_reg;

   for ( genvar i=0; i<pp_per_cycle; i++ ) begin

      uwire [wid_lg:1] pos = iter * pp_per_cycle + i;
      uwire [2*wid-1:0] pp = pos < wid && cand[pos] ? plier << pos : 0;

      CW_csa #(2*wid) csa
        ( .sum(accum_sum_a[i+1]), .carry(accum_sum_b[i+1]), .a(accum_sum_a[i]), .b(accum_sum_b[i]), .c(pp));
   end

   always @( posedge clk )
      if ( iter == iterations ) begin
         prod <= accum_sum_a_reg + accum_sum_b_reg;
         accum_sum_a_reg <= 0;
         accum_sum_b_reg <= 0;
         iter <= 0;
      end else begin
         prod <= prod;
         accum_sum_a_reg <= accum_sum_a[pp_per_cycle];
         accum_sum_b_reg <= accum_sum_b[pp_per_cycle];
         iter <= iter + 1;
      end

endmodule
```

USE NEXT PAGE FOR SOLUTION

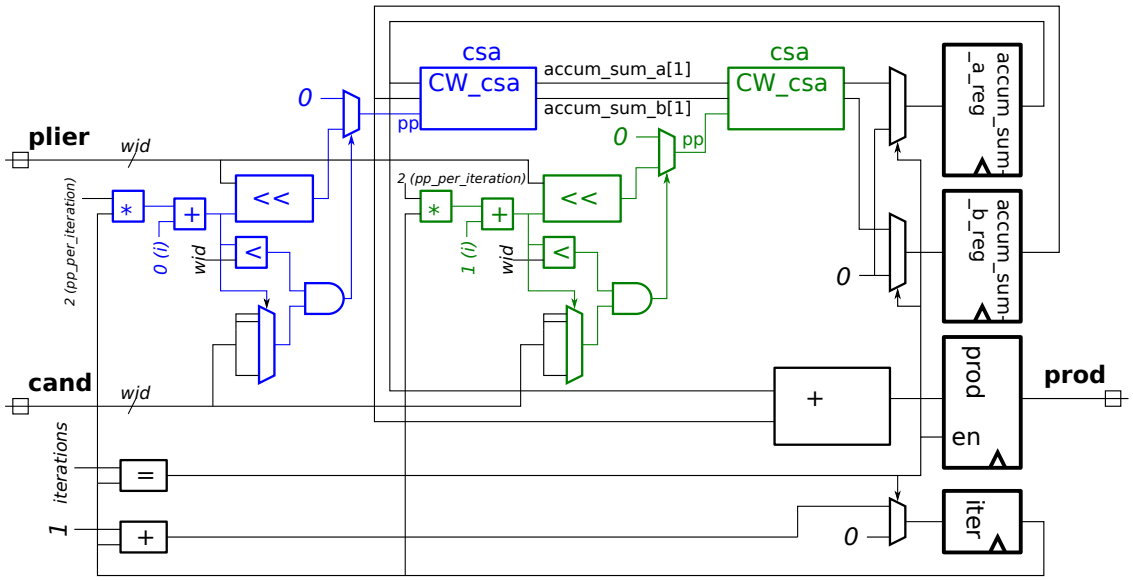


USE NEXT PAGE FOR SOLUTION

(*a*) Show optimizations that might be performed that exploit the value $m = 2$ (that is, `pp_per_iteration=2`).

(*b*) Show the optimizations that might be performed assuming that `wid` is odd, and assuming that `wid` is even, both for $m = 2$.

- ☐ Modify diagram to show optimizations for `pp_per_iteration` = $m = 2$ and arbitrary `wid`.
- ☐ Modify diagram to show optimizations for `pp_per_iteration` = $m = 2$ and odd `wid`.
- ☐ Modify diagram to show optimizations for `pp_per_iteration` = $m = 2$ and even `wid`.



Problem 2, continued:

(*c*) The cost of the shifters with input `plier` in the design on the previous pages is significant. Explain how these shifters can be eliminated by adding a register. Quickly sketch the hardware to illustrate your answer.

☐ Show how a register can be used to eliminate the costly shifters.

(*d*) Explain how the streamlined multiplier described in class eliminated the `plier` shifter *without* having to add a register.

☐ Show how the streamlined multiplier does not need an extra register to eliminate the shifter.

Problem 3: [10 pts] The module below computes the prefix sum of a sequence of integers at its input.

```
module prefix_sum #( int len=8, int wid = 8)
  (output logic [wid:1] psum [len], input uwire [wid:1] elts[len]);

   always @* begin
      psum[0] = elts[0];
      for ( int i=1; i<len; i++ ) psum[i] = psum[i-1] + elts[i];
   end
endmodule
```

(*a*) Show the hardware that would be synthesized for the module before optimization, elaborated with parameters `len=4` and `wid=8`. Label the input ports `elts[0]`, `elts[1]`, `elts[2]`, and `elts[3]`; and label the output ports `psum[0]`, `psum[1]`, `psum[2]`, and `psum[3]`.

☐ Show synthesized hardware.

(*b*) Estimate the delay for the synthesized hardware before optimization. Use w for the value of `wid` and L for `len`. Assume that a w-bit adder has delay w.

☐ Delay in terms of w and L:

Problem 4: [15 pts] Answer the following questions about the Verilog module below.

```
module timing();
   logic [7:0] a, e, f2, g, g1, g2;   logic clk;      uwire [7:0] e1, f, f1;
   initial begin
      clk = 0;
      a = 11;
      #1;
      a = 1;
      a <= 22;
      a <= #5 a + 1;

      #9;
      a = 7;
      e = 10;
      f2 = 30;
      g = 40;
      g1 = 50;
      g2 = 60;

      #10;
                                          // B0
      a <= 700;
      clk = 1;

      #1;
      // POINT X (See subproblem.)
   end

   always @( posedge clk ) e = a;          // B1
   always @*               e1 = a;         // B2
   always @*               f = e + 1;      // B3
   always @*               f1 = e1 + 1;    // B4
   always @( posedge clk ) f2 <= e + 1;    // B5
   always @( posedge clk ) begin           // B6
                            g = f;   g1 = f1;   g2 = f2;   end
endmodule
```

(*a*) Show values for `a` versus time in the table below. For this part, **only** `a`. The table already shows that `a` has value 11 from time 0 to time 1. Extend the table as long as necessary, and be sure to show values for both t and `a`. *Note: The original exam did not provide the table. Also, in the original exam there were differences in how* `a` *was assigned.*

☐ Complete the table.

t	0	1			
a	11				

(*b*) Show the values that will be present on `g`, `g1`, `g2` when execution reaches the `POINT X` comment in the module above. For partial credit also show intermediate values for other signals used to compute the `g`'s. (Look at next part before solving this one.)

☐ At `POINT X` g=_____, g1=_____, g2=_____.

(*c*) Recall that the event queue used for Verilog simulation has *active*, *inactive*, and *NBA* regions, among others. Just before `B1` starts execution in module `timing` above the active region might contain `B1`, `B5`, and `B6` (see the comments on the right). (What the other regions contain is part of this problem.) Show the contents of the three regions when `B5` starts. Assume that events in a region are scheduled in order.

☐ When `B5` starts: Active = {_____}. Inactive = {_____}. NBA = {_____}.

Problem 5: Answer each question below.

(*a*) [5 pts] Module `add3` is supposed to compute the sum of its three inputs using instances of `our_adder`, but it won't work. Fix the problem. The fixed module should still use `our_adder`.

☐ Fix `add3`.

```
module add3(output uwire [15:0] sum, input uwire [15:0] a,b,c);



   our_adder a1( sum , a , b );

   our_adder a2( sum , sum , c );

endmodule
```

(*b*) [8 pts] The output of the module below is like the input except the bit positions are reversed (after enough clock cycles). Re-write the module so that it synthesizes to combinational logic (the `clk` input will no longer be needed). Add a parameter to indicate the input and output bit width.

```
module bitrev(output logic [7:0] x, input uwire [7:0] a, input uwire clk);
   logic [2:0] pos;
   initial pos = 0;

   always @( posedge clk ) begin
      x[pos] = a[7-pos];
      pos++;
   end
endmodule
```

☐ Re-write so that it is combinational.

☐ Include a parameter `wid` to specify the size.

Problem 6: Answer each question below.

(*a*) [5 pts] A Verilog module computes a result in one clock cycle. In our design we need that result in 3 ns, which can easily be achieved. The right way to achieve that in Cadence Encounter is to use the `define_clock` command to set the target clock period to 3 ns. Suppose instead we used `define_clock` to set the period to 1 ps, an impossible goal. *Note: The original exam did not have the "can easily be achieved" phrase.*

☐ Would the synthesized design meet our 3 ns performance goal?

☐ Considering typical design goals, what would be the disadvantage of setting the period to 1 ps for our design even though we needed 3 ns?

(*b*) [10 pts] In the module below, `translate` directives are used to prevent the synthesis program from reading the line with `initial`.

```
module mult_seq( output logic [311:0] prod, input logic [15:0] plier, cand, input uwire clk);

   logic [3:0] pos;    logic [31:0] accum;

   // cadence translate_off          <-- The translate synthesizer directive.
   initial pos = 0;
   // cadence translate_on           <-- The translate synthesizer directive.

   always @( posedge clk ) begin
      if ( pos == 0 ) begin  prod = accum;  accum = 0;  end
      if ( cand[pos] == 1 ) accum += plier << pos;
      pos++;
   end
endmodule
```

☐ Why shouldn't the synthesis program see the line with `initial`?

☐ What would happen if the synthesis program saw the `initial` line?

☐ What would happen if the simulation program *didn't* see the line with `initial`?

(*c*) [7 pts] All four variables below have a size of 32 bits, but there are differences between them.

```
logic [31:0] a;
logic b [31:0];
logic [0:31] c;
int e;
```

☐ Difference between `a` and `b`?

☐ Difference between `a` and `c`?

☐ Difference between `a` and `e`?

Name _____

Digital Design using HDLs
EE 4755
Final Examination

Monday, 8 December 2014 10:00-12:00 CST

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

Alias _____

Good Luck!

Problem 1: [20 pts] The `encode` module below, based on Homework 4, is used to convert a decimal value to binary one ASCII digit at a time. Input `val_prev` is the binary value so far, and output `val_next` is the binary value after using ASCII character `ascii_char`. If `ascii_char` isn't a numeric digit `non_digit` is set to `1` and `val_next` is set to zero. There is also an `overflow` output.

```
module encode
  #( int width = 32 )
   (output logic [width-1:0] val_next,
    output logic overflow,                output uwire non_digit,
    input uwire [7:0] ascii_char,         input uwire [width-1:0] val_prev);

   logic [width+3:0] val_curr;            logic [3:0] high_bits, bin_char;

   assign non_digit = ascii_char < Char_0 || ascii_char > Char_9;

   always_comb begin
      bin_char = ascii_char - Char_0;
      val_curr = 10 * val_prev + bin_char;
      high_bits = val_curr >> width;
      if ( non_digit ) begin    overflow = 0; val_next = 0;       end
      else             begin
        overflow = high_bits != 0;
        val_next = val_curr;
      end
   end
endmodule
```

(*a*) Show the hardware that will be synthesized for this module. Take into account optimizations (see the next subproblem).

☐ Synthesized hardware.

(*b*) Indicate how many units such as adders, multipliers, shifters, and multiplexors will actually be present in the optimized hardware. The count should be based on the units that are present after optimization, not on the hardware first inferred from the Verilog.

☐ Number of adders. ☐ Number of multipliers. ☐ Number of shifters. ☐ Number of multiplexors.

Problem 2: [20 pts] Appearing below is another encode module, this one has a new input `radix`, which indicates the radix (base) of the number to be converted. When completed the module should function like the module from the previous problem, except that the digits form a radix-`radix` number. For example, if `radix` were 10 it would operate like the previous module. If `radix` were 8 the digits would be octal, etc.

(*a*) Modify the module so that it takes into account the radix. Assume that `radix` can be any value from 2 to 16. Note that for a radix of 16 the valid digits are 0-9 and A-F (only consider upper case).

☐ Modify the module to generate the correct `non_digit` output.

☐ Modify the module to update `val_next` correctly given the radix.

```
typedef enum {Char_0 = 48, Char_9 = 57, Char_A = 65, Char_F = 70} Chars;
module encode_radix #( int width = 32 )
   (output logic [width-1:0] val_next,
    output logic overflow,                  output uwire non_digit,
    input uwire [7:0] ascii_char,           input uwire [width-1:0] val_prev,
    input uwire [4:0] radix);

   logic [width+3:0] val_curr;

   logic [3:0] high_bits, bin_char;

   always_comb begin

      val_curr =

      high_bits = val_curr >> width;
      if ( non_digit ) begin
         overflow = 0;
         val_next = 0;
      end else begin
         overflow = high_bits != 0;
         val_next = val_curr;
      end
   end
endmodule
```

Problem 2, continued:

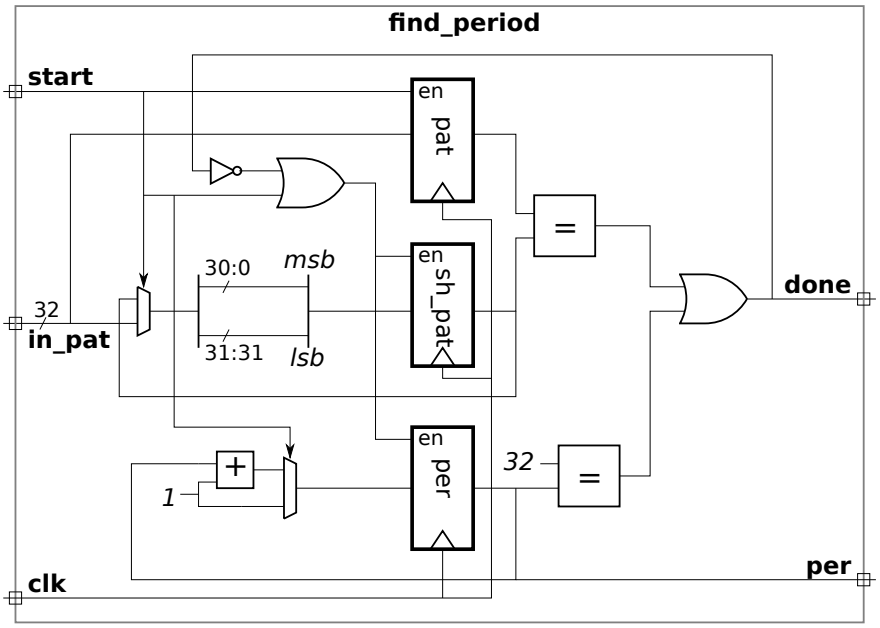
(*b*) Suppose that module `encode_radix` (from the previous part) were to be used in a larger design in which the values of `radix` could only be 2, 8, 10, and 16. Also suppose that the synthesis program can't figure out that `radix` is limited to these values. Why would the cost be higher than necessary, and how could `encode_radix` be modified to get the lower cost hardware?

☐ Explain why the cost will be higher than is necessary.

☐ Show the changes to `encode_radix` so that the synthesis program *will* generate the lower cost design. The port definitions **cannot** be changed.

Problem 3: [20 pts] Appearing to the right is hardware and a corresponding Verilog module. The module is incomplete, finish it. *Hint: The hardware includes an end-around shift, that's the part with the msb/lsb labels.*

- ☐ Add sizes and other information to port declarations.
- ☐ Finish the Verilog code.




```
module find_period
   (output          per,      output          done,

    input           in_pat,

    input           start,    input clk);
```

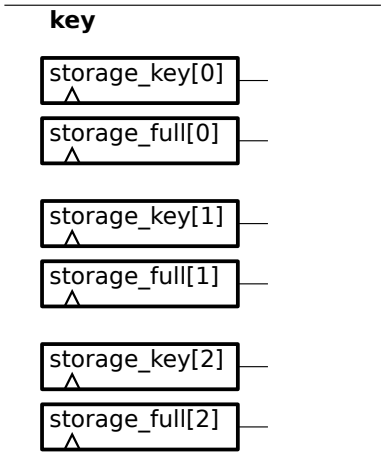
```
endmodule
```

Problem 4: [20 pts] The Verilog below is the key lookup part of the simple CAM module used in class.

```
logic [dwid:1] storage_data [ssize];
logic [kwid:1] storage_key [ssize];
logic [ssize-1:0] storage_full;

always_comb begin
   mmatch = 0;   midx = 0;
   for ( int i=0; i<ssize; i++ )
     if ( storage_full[i] && storage_key[i] == key ) begin mmatch = 1; midx = i; end
end

assign out_data = storage_data[midx];
```

(*a*) Starting with the registers and `key` shown below, sketch the hardware synthesized for this code without optimization. The hardware should produce values for `mmatch` and `midx` (but not `out_data`). Do so for `ssize=3`. In class we often showed part of this as a box labeled "priority encoder" (or "pri" for short), in this problem actually show the hardware.

☐ Synthesized hardware for `ssize = 3` to generate ☐ `mmatch` and ☐ `midx`.



(*b*) Assume that the cost of an a-bit comparison unit is a, and its delay is also a. Assume that the cost of an a-input, b-bit multiplexor is ab and the delay is 1. Compute the cost and delay of the logic used to compute `midx` in terms of `ssize` (use s in your formulas) and `kwid` (use k in your formulas). As with the previous part, do this for the unoptimized hardware. Remember to solve this for an arbitrary value of `ssize` (s), not for $s = 3$.

☐ Cost in terms of s and k:

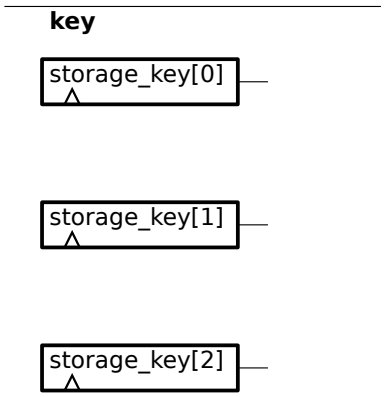
☐ Delay in terms of s and k:

Problem 4, continued: Appearing below is a variation on the key lookup from the CAM module. Instead of finding a matching key it finds the largest stored key that is ≤ to the lookup key. Note that this version doesn't include `storage_full`.

```
logic [dwid:1] storage_data [ssize];
logic [kwid:1] storage_key [ssize];

always_comb begin
   midx = 0;  bkey = 0;
   for ( int i=0; i<ssize; i++ )
     if ( storage_key[i] >= bkey && storage_key[i] <= key ) // READ THIS LINE CAREFULLY
       begin   midx = i;  bkey = storage_key[i];   end
end

assign out_data = storage_data[midx];
```

(*c*) Sketch the hardware for `ssize=3`.

☐ Sketch the synthesized hardware needed to generate `bkey`.



(*d*) Compute the cost and performance in terms of `ssize` (use s) and the key size (use k). As before a k-bit comparison unit (equality or magnitude) costs k and has a delay of k and an a-input, b-bit mux costs ab and has a delay of 1. *Hint: There's a big difference.*

☐ Cost in terms of s and k:

☐ Delay in terms of s and k:

Problem 5: [20 pts] Answer each question below.

(*a*) The module below is supposed to count from 0 to `max` (inclusive), then return to zero. Strictly speaking it does, but there are problems, including the fact that it's not synthesizable. Fix the problems.

```
module counter #(int max = 3)(output logic [7:0] count, input uwire clk);

   always @( posedge clk ) begin

      count <= count + 1;

   end

   always @* begin

      if ( count == max ) count <= 0;

   end

endmodule
```

☐ Why isn't the module synthesizable?

☐ Fix the problem.

(*b*) There is a problem with the module below due to the way that `a` is declared.

```
module sa1(output uwire a, input uwire c, d);

   always_comb begin

      a = c & d;

   end

endmodule
```

☐ Fix the problem **by changing** the declaration of `a`.

☐ Fix the problem **without changing** the declaration of `a`.

(*c*) Describe a situation in which using `always_comb` has a benefit over using `always @*`.

☐ Situation where `always_comb` helps.

(*d*) The module below is supposed to be computing $x^2 + y^2$.

```
module sa2(output logic [63:0] sos, input uwire [63:0] x, y);

   logic [63:0] a1, b1, a2, b2;
   uwire [63:0]  p, s;

   fpmul f1(p,a1,b1);
   fpadd f2(s,a2,b2);

   always @* begin

      // Compute x^2.
      a1 = x;  b1 = x;
      #1;
      sos = p;

      // Compute y^2.
      a1 = y;  b1 = y;
      #1;

      // Compute x^2 + y^2.
      a2 = p;  b2 = sos;
      #1;
      sos = s;

   end

endmodule
```

☐ Explain why the module is not synthesizable.

☐ Fix the problem.

12 Spring 2001

Name _____

Digital Design Using Verilog
EE 4702-1
Midterm Examination

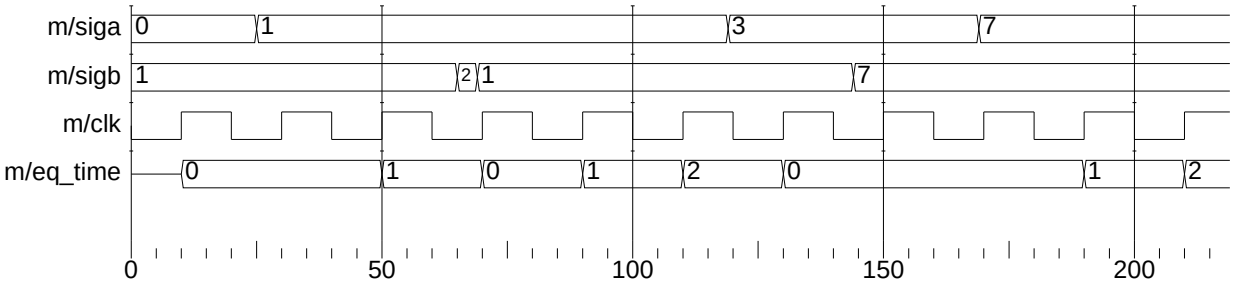
16 March 2001 8:40-9:30 CST

Problem 1	_____	(30 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(35 pts)
Problem 4	_____	(10 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: Complete the Verilog behavioral description below so that it operates as follows. Compute 32-bit output `eq_time` so that it is the number of consecutive positive edges of input `clk` for which 32-bit inputs `siga` and `sigb` remain equal. The counting should start on the first positive edge of `clk` after `siga` becomes equal to `sigb`; the count starts at zero at the moment they become equal, and while they remain equal the count is incremented at each positive edge. The count should go back to zero at the first positive edge of `clk` after `siga` becomes unequal to `sigb`. The count goes to zero even if `siga` and `sigb` become equal again before the positive edge. Sample output appears in the timing diagram below. (30 pts)




```
module monitor(eq_time, siga, sigb, clk);
   input siga, sigb, clk;
   output eq_time;
   // Don't forget to declare port types.
```

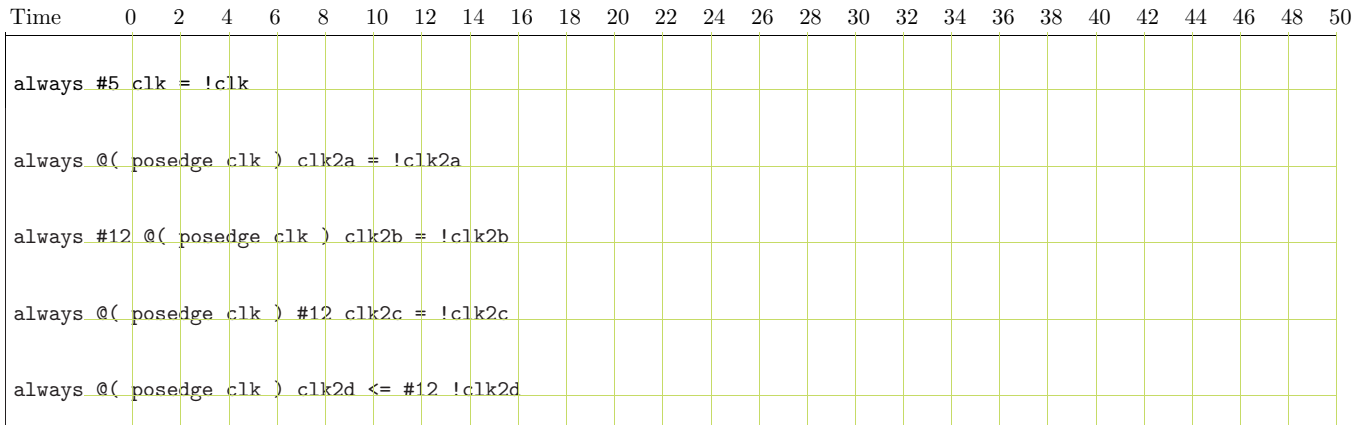
```
endmodule
```

Don't get bogged down: There are eight more problems, some can be answered quickly.

Problem 2: Complete the following timing diagram problems.

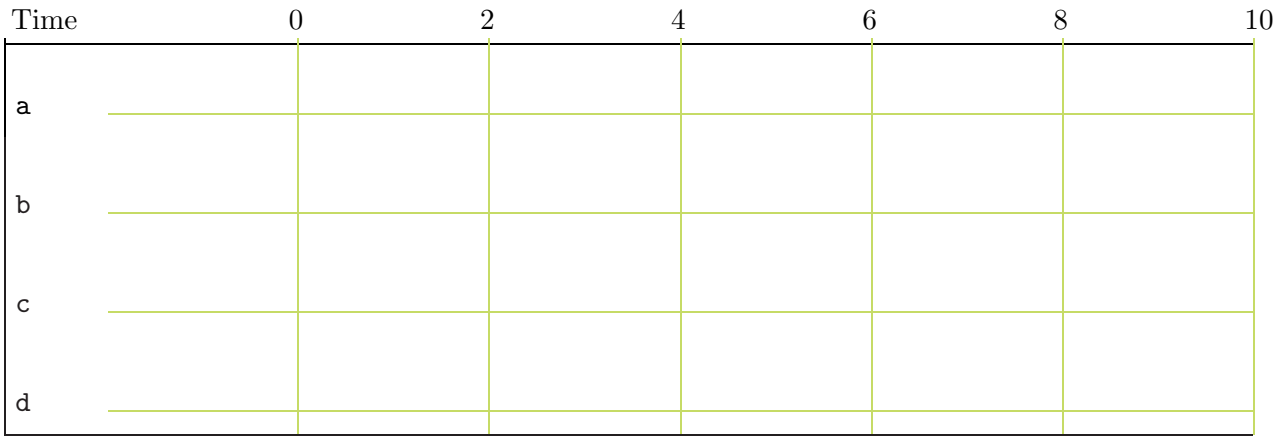
(*a*) Complete the timing diagram below. (15 pts)

```
module timing_stuff();
   reg clk, clk3, clk2a, clk2b, clk2c, clk2d,
   initial begin
      clk = 0; clk2a = 0; clk2b = 0; clk2c = 0; clk2d = 0; clk3 = 0;
   end
```



(*b*) Complete the timing diagram below. Be sure to clearly indicate when a signal value changes. (10 pts)

```
module timing();
   integer a, b, c, d;
   initial begin
      a = 0;
      b = 10;
      c = 20;
      d <= #0 3;
      d = 30;
      d <= #1 300;
      d <= #2 3000;
      #1;
      b = 100;
      c <= 200;
      a <= #5 b + c;
      #1;
      b = 1000;
      c <= 2000;
      #10;
   end
endmodule
```



Problem 3: Answer each question below. Some can be answered quickly, try answering those questions first.

(*a*) The `match_count_x` modules below are *supposed to* count the number of times input `symbol` is the same as input `targ`. Output `count` should be incremented if `symbol` is the same as `targ` after a change in `symbol`. Most or all of the modules below don't work properly. For each non-working module describe the problem and how it is simulated. *It is important to describe how the incorrect Verilog is simulated and why it is wrong.*

Port declarations and initializations are not shown, but assume they are present and correct. Behavior for unknown and high-impedance values is undefined. In other words, the problems are **not** related to declarations, initialization, or unknown values. (10 pts)

```
module count_match_1(count,symbol,targ); // Declarations and init. not shown.

   always wait( symbol == targ ) count = count + 1;

endmodule
```

```
module count_match_3(count,symbol,targ); // Declarations and init. not shown.

   always #10 if ( symbol == targ ) count = count + 1;

endmodule
```

```
module count_match_4(count,symbol,targ); // Declarations and init. not shown.

   always @( symbol == targ ) count = count + 1;

endmodule
```

(*b*) Show how each of the three adders below can be used in the module `use_adders` to add seven to input `a`. **Do not** modify the adders themselves. (10 pts)

```
module adder1(x,a,b);
   input a, b;
   output x;
   wire [31:0] a, b;
   wire [31:0] x = a + b;
endmodule

module adder2(x,a);
   input a;
   output x;
   parameter b = 0;
   wire [31:0] a;
   wire [31:0] x = a + b;
endmodule

module adder3(x,a);
   input a;
   output x;
   wire [31:0] a;
   wire [31:0] x = a + `b;
endmodule


module use_adders(x_1,x_2,x_3,a);
   input a;
   output x_1, x_2, x_3;  // Each output should be a + 7
   // Use adder1, adder2, and adder3 to generate respective x_ outputs.









endmodule
```

(*c*) Show the values that will be assigned in each assignment to `r`. Variables `a`, `c`, and `r` are six-bit registers. (5 pts)

```
a = 6'b101010;
c = 6'bx1x0x1;

r = & a;

r = | a;

r = ^ a;

r = & c;

r = | c;

r = ^ c;
```

(*d*) Do the two code fragments below do the same thing? If not, how do they differ? (5 pts)

```
// Fragment A.

if ( foo > bar ) x = x + 1; else y = y + 1;


// Fragment B.

case ( foo > bar )
  1: x = x + 1;
  default: y = y + 1;
endcase
```

(*e*) Why can't the following increment macro be re-written as a function or task in Verilog 95? (5 pts)

```
‘define incr(a) a=a+1
// ...
  // Sample uses of macro.
  for (i=0; i<10; ‘incr(i)) x = x + y;
  for (j=0; j<10; ‘incr(j)) begin foo(j); k = k + x; end
```

Problem 4: The module below counts the number of five's and nine's appearing at input `c`. Explain exactly when five's and nine's are counted (start cycle and end cycle), and describe any restrictions on the counts. (10 pts)

```
module yet_another_symbol_counter(fives, nines, c);
   input c;
   output fives, nines;
   wire [7:0] c;
   reg [31:0] fives, nines;

   initial fork

      begin
         fives = 0;
         nines = 0;
      end

      #50 fork:A
         repeat ( 42 ) @( c ) if ( c == 5 ) fives = fives + 1;
         #100 disable A;
      join

      #70 fork:B
         forever @( c ) if ( c == 9 ) nines = nines + 1;
         #200 disable B;
      join

   join

endmodule
```

Name _____

Digital Design Using Verilog
EE 4702-1
Final Examination

9 May 2001 7:30-9:30 CDT

Problem 1	_____	(15 pts)
Problem 2	_____	(18 pts)
Problem 3	_____	(17 pts)
Problem 4	_____	(18 pts)
Problem 5	_____	(12 pts)
Problem 6	_____	(20 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: The module below is in an explicit structural form.

(*a*) Re-write the module in behavioral form. The delays can be assumed to be pipeline delays. (10 pts)

(*b*) What is the difference between pipeline and inertial delays? Which kind of delay is used in your solution to the problem above? (5 pts)

```
module expl_str(x,y,a,b,c);
   input a, b, c;
   output x, y;
   wire   a, b, c, x, y;
   wire   na, nb, nc, t3, t5, t6;

   not n1(na,a);
   not n2(nb,b);
   not n3(nc,c);
   and #1 a1(t3,na,b,c);
   and a2(t5,a,nb,c);
   and a3(t6,a,b,nc);
   or o1(x,t3,t6);
   or #3 o2(y,a,t5);

endmodule
```

```
module behavioral(x,y,a,b,c);
   input a, b, c;
   output x, y;
   // Solution here.  Don't forget types for ports!
```

```
endmodule
```

Problem 2: The module below sets output `rot` to the number of times that input `a` must be rotated (end-around shifted) to obtain the value on input `b`, or to 32 if `a` is not a rotated version of `b`.

(*a*) Write a testbench module that tests `rots` with input pairs `a`=0,`b`=0; `a`=0,`b`=1; `a`=0,`b`=2; and `a`=0,`b`=3. (The `rot` output should be zero for the first pair and 32 for the others.) The testbench should include an integer `err` and set it to the number of incorrect outputs.

It is important that the testbench makes correct use of `ready` and `start`. (Part of the problem is determining just what is "correct use.") The testbench should use `ready` rather than assumed timing. Also, test only a single instance of `rots` and don't forget the clock. (18 pts)

```
module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;        output ready, rot;
   reg         ready;             wire [31:0] a, b;
   reg [5:0]   rot;               wire        start, clk;
   reg [31:0]  acpy;
   initial rot = 0;
   always @( posedge clk ) begin
      ready = 1;   while ( !start ) @( posedge clk );
      ready = 0;   while (  start ) @( posedge clk );
      rot = 0;  acpy = a;
      while (  acpy != b  &&  rot < 32  ) @( posedge clk ) begin
         acpy = { acpy[30:0], acpy[31] };
         if ( acpy == a ) rot = 32; else rot = rot + 1;
      end
   end
endmodule

module testrot();
 integer err;









endmodule
```

Problem 3: Convert the `rots` module (repeated below) to synthesizable Form 2 (edge-triggered flip-flops). Do not change the ports or what it does. In particular, `ready` and `start` must be used the same way. Ignore reset. (17 pts)

```
module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;        output ready, rot;
   reg         ready;             wire [31:0] a, b;
   reg [5:0]   rot;               wire        start, clk;
   reg [31:0]  acpy;
   initial rot = 0;
   always @( posedge clk ) begin
      ready = 1;   while ( !start ) @( posedge clk );
      ready = 0;   while (  start ) @( posedge clk );
      rot = 0;  acpy = a;
      while (  acpy != b  &&  rot < 32  ) @( posedge clk ) begin
         acpy = { acpy[30:0], acpy[31] };
         if ( acpy == a ) rot = 32; else rot = rot + 1;
      end
   end
endmodule
```

```
module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;
   output ready, rot;  // Don't forget port types and other declarations.

         acpy = { acpy[30:0], acpy[31] };
         if ( acpy == a ) rot = 32; else rot = rot + 1;

endmodule
```

Problem 4: Two synthesizable descriptions appear below.

(*a*) In what synthesizable form is the Verilog description below? (2 pts)

(*b*) Draw a schematic showing the approximate RTL-level description generated by a synthesis program like Leonardo. (7 pts)

```
module whatsyna(x, y, z, a, b, op);
   input a, b, op;
   output x, y, z;
   wire [7:0] a, b;
   wire [1:0] op;
   reg [7:0]  x, y, z;

   always @( op or a or b ) begin

      if ( a == 0 ) y = b;

      if ( a < b ) z = a; else z = b;

      case ( op )
        0: x = a + b;
        1: x = a;
        2: x = b;
      endcase

   end

endmodule
```

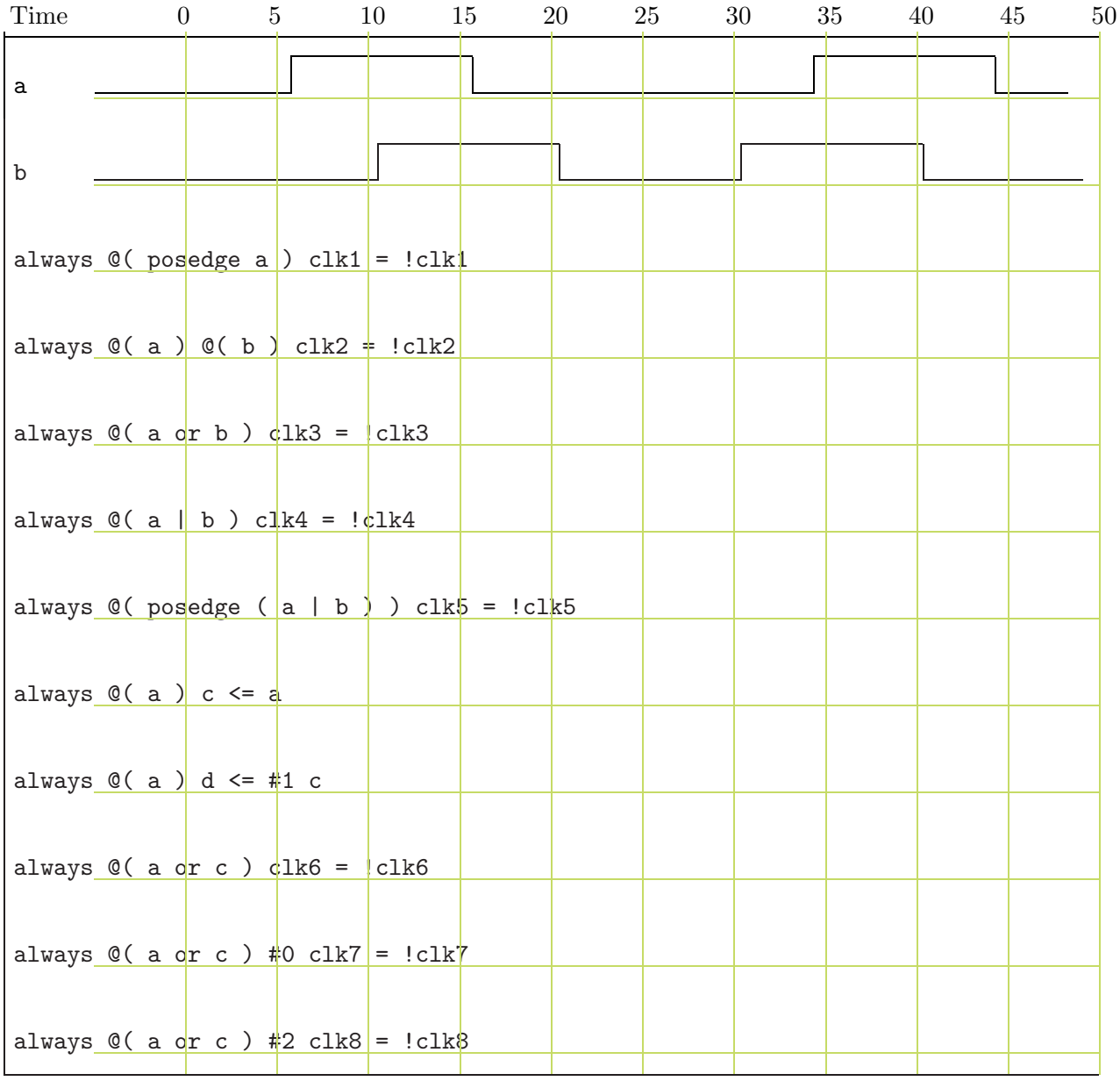
Problem 4, continued:

(*c*) In what synthesizable form is the Verilog description below? (2 pts)

(*d*) Draw a schematic showing the approximate RTL-level description generated by a synthesis program like Leonardo. (7 pts)

```
module whatsyn2(sum, nibbles, a, b, c);
   input nibbles, a, b, c;
   output sum;
   wire [15:0] nibbles;
   wire        a, b, c;
   reg [6:0]   sum;

   reg [15:0]  n2;
   reg         last_c;
   integer     i;

   always @( posedge a or negedge b )
     if ( !b ) begin

        sum = 0;

     end else begin

        if ( c != last_c ) begin
           n2 = nibbles;
           for ( i=0; i < 4; i = i + 1 ) begin
              sum = sum + n2[3:0];
              n2 = n2 >> 4;
           end
        end
        last_c = c;

     end

endmodule
```

Problem 5: In the diagram below `c`, `d`, and identifiers starting with `clk` are all initialized to zero. Complete the timing diagram. (12 pts)



Problem 6: Answer each question below.

(*a*) The code below, based on the Homework 3 solution, simulates properly before synthesis but in the post-synthesis simulation the testbench reports an incorrect beep time.

What goes wrong? Fix the problem without modifying the code below the indicated line. *Hint: The beep can start (and stop) at a slightly different time than the code below.* (5 pts)

```
module beepprob(beep, clk);
   input clk;
   output beep;

   assign beep = | beep_timer;


   // DO NOT MODIFY CODE BELOW THIS LINE.
   always @( posedge clk ) begin
      // Lots of stuff;

      if ( beep_timer ) beep_timer = beep_timer - 1;

   end

endmodule
```

(*b*) Describe something that a parameter can be used for that an ordinary input port cannot and something that an input port can be used for that a parameter cannot. (5 pts)

(*c*) What is the difference between `case`, `casex`, and `casez`? (5 pts)

(*d*) Explain how each of the three statements below behave differently with unknown values. In particular, explain what has to be unknown and how the results of each statement is different. (5 pts)

```
m1 = a > b ? c : d;

if ( a > b ) m2 = c; else m2 = d;

case ( a > b )
  1: m3 = c;
  default: m3 = d;
endcase
```

13 Spring 2000

Name _____

Digital Design Using Verilog
EE 4702-1
Midterm Examination

5 April 2000 8:40-9:30 CDT

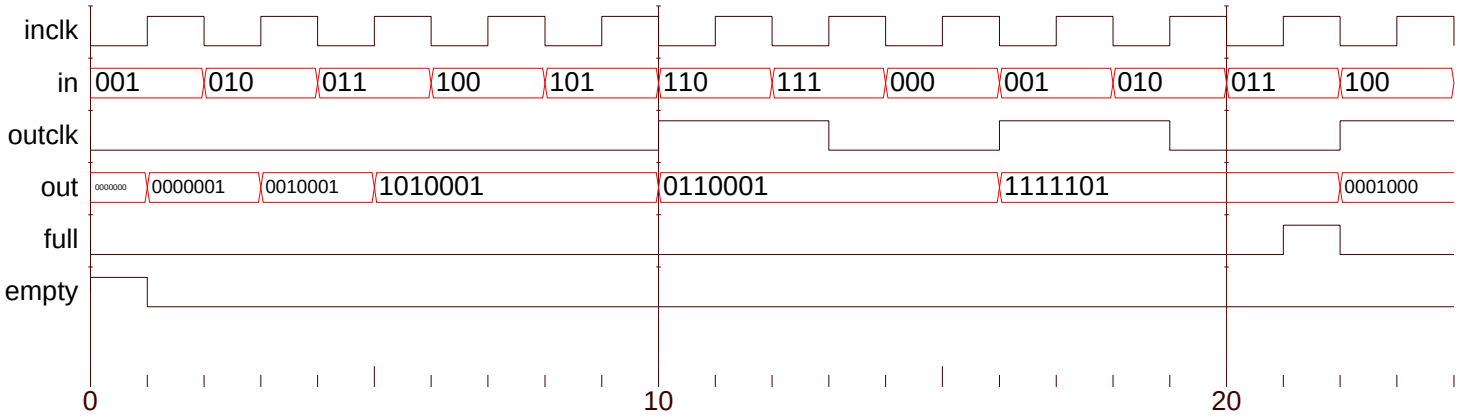
Problem 1 _____ (40 pts)

Problem 2 _____ (60 pts)

Alias _____ Exam Total _____ (100 pts)

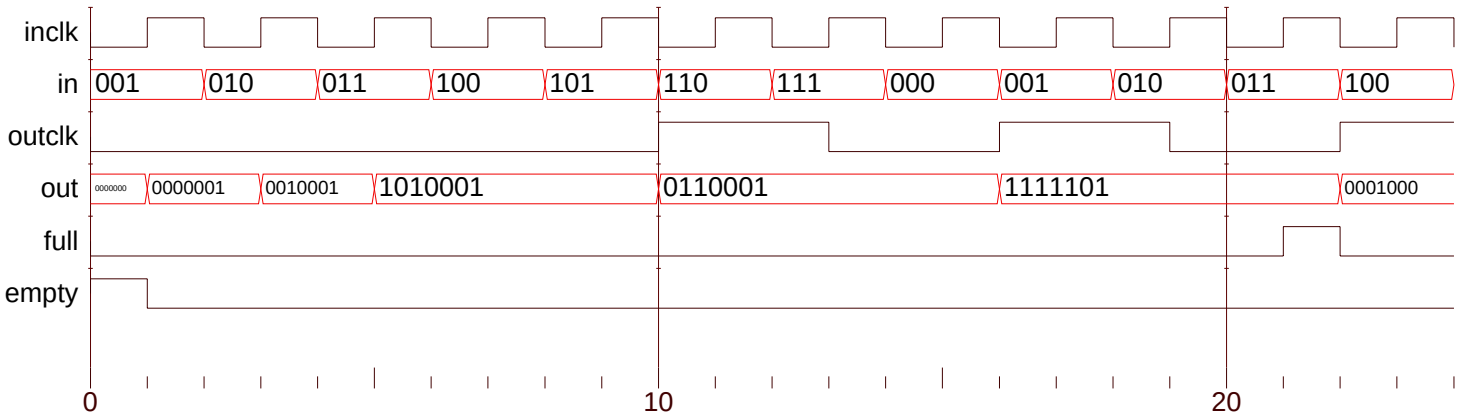
Good Luck!

Problem 1: Complete the Verilog description (below) of a FIFO-like module which has a 3-bit data input, `in`; a 7-bit output, `out`; 1-bit inputs `inclk` and `outclk`; and 1-bit outputs `full` and `empty`. The module operates like a FIFO (first in, first out) except that the width of the data input and output ports are different: it reads data 3 bits at a time (on a positive edge of `inclk`) and outputs 7 bits at a time (consisting of data from two input words plus one bit of a third). Unless the module has less than 3 bits of space left, on a positive edge of `inclk` the value on `in` is stored. The oldest 7 bits stored by the module always appear on output `out`. On a positive edge of `outclk` the oldest 7 bits are removed and the output displays the next 7 bits. Output `full` is 1 if the module cannot accept another 3 bits of input and is 0 otherwise; output `empty` is 1 if the module is empty and is 0 otherwise. Parameter `storage` is the total number of bits stored by the module. An example of the module operating is shown in the timing diagram below. (40 pts)




```
module width_change(out,full,empty,outclk,in,inclk);
   input outclk, in, inclk;
   output out, full, empty;

   parameter storage = 20;

   wire [6:0] out; // Can change to reg for solution.
   wire [2:0] in;
   wire       inclk, outclk;
   wire       full, empty; // Can change to reg for solution.

   reg [storage-1:0] sto;  // Storage for data.
   integer           amt;  // Number of occupied bits in sto.

   // USE THE NEXT PAGE FOR THE SOLUTION.

endmodule // width_change
```

Problem 1, continued: The diagram and code from the previous page are repeated below.




```
module width_change(out,full,empty,outclk,in,inclk);
   input outclk, in, inclk;
   output out, full, empty;

   parameter storage = 20;

   wire [6:0] out; // Can change to reg for solution.
   wire [2:0] in;
   wire       inclk, outclk;
   wire       full, empty; // Can change to reg for solution.

   reg [storage-1:0] sto;  // Storage for data.
   integer           amt;  // Number of occupied bits in sto.

   // Solution goes here.









endmodule // width_change
```

Problem 2: Answer each question below.

(*a*) Describe something that a function can do (or be used for) that a task cannot. Describe something that a task can do (or be used for) that a function cannot. (10 pts)

(*b*) Convert the following behavioral code to **explicit** structural code. (10 pts)

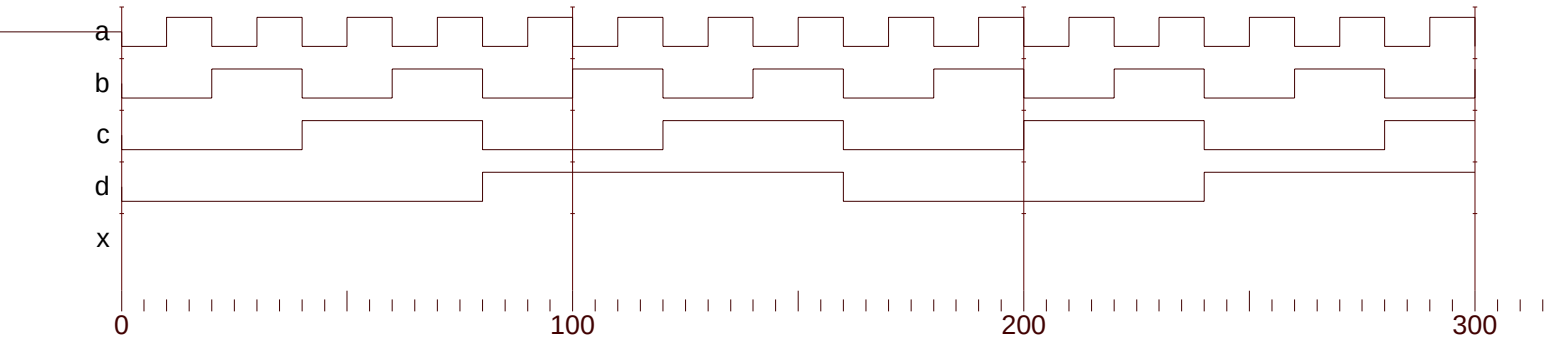
```
module btos(x, a, b);
   input a, b;
   output x;
   wire   a, b;
   reg    x;

   always @( a or b ) if( a ) x = b; else x = ~b;

endmodule // btos
```

(*c*) Show the changes (values and times) to `a` and `b` in the module below. (10 pts)

```
module assig();
   reg [15:0] a, b;
   initial
     begin
        a = 1;
        b = 2;
        #1;
        a <= b;
        b <= a;
        #1;
        a <= b + 10;
        b <= #5 b + 20;
        #1;
        b = #1 3;
        b <= 4;
        b <= #2 5;
        b <= #10 6;
        b = 7;
        #20;
     end
endmodule
```

(*d*) Show the changes (values and times) to `x` in the module below using the timing diagram provided. (10 pts)

```
module events1();
   wire a, b, c, d;
   reg [2:0] x;
   reg [3:0] i;
   assign  {d,c,b,a} = i;

   initial begin
     i = 0;
     forever #10 i = i + 1;
   end

   always begin
      #15;
      @( a );
      x = 1;
      @( posedge a ) x = 2;
      @( a or b ) x = 3;
      @( a | b | c | d ) x = 4;
      wait( a | b ) x = 5;
      wait( a ) x = 6;
      wait( ~a ) x = 7;
   end // always begin

endmodule // events1
```



(*e*) Show the changes (values and times) to `aa` in the module below. (10 pts)

```
module d();
   reg a;
   wire aa;

   and #(2,3) (aa,a,1);

   initial begin
      a = 0;
      # 10;
      a = 1;
      # 10;
      a = 0;
      # 10;
      a = 1;
      # 1;
      a = 0;
      # 10;
   end
endmodule // d
```

(*f*) Complete module `after` so that it does the same thing as `before`. All procedural code in module `after` must go in the one initial process. The solution must use `fork` and `join`. Structural code **cannot** be added. (10 pts)

```
module before(asum,bsum,out,a,ainp,b,binp,c);
   output asum, bsum, out;
   input  a, ainp, b, binp, c;

   reg [9:0]  asum, bsum, out;
   wire [9:0] ainp, binp;
   wire       a,b,c;

   always @( a ) asum = asum + ainp;

   always @( b ) bsum = bsum + binp;

   always @( posedge c ) out = asum + bsum;

endmodule

module after(asum,bsum,out,a,ainp,b,binp,c);
   output asum, bsum, out;
   input  a, ainp, b, binp, c;

   reg [9:0]  asum, bsum, out;
   wire [9:0] ainp, binp;
   wire       a,b,c;

   // ALL code must go in the initial process below.
   initial begin









   end // initial

endmodule
```

Name _____

Digital Design Using Verilog

EE 4702-1

Final Examination

8 May 2000, 7:30–9:30 CDT

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

Alias _____

Good Luck!

Problem 1: The modules below are supposed to describe combinational logic that rearranges bits. The output of module `rearrange`, below, is a rearranged version of its input `a`; input `op` determines how the bits are rearranged. Module `rearrange` uses two instances of `rearrange` to reverse and then left shift its inputs. Unfortunately, the modules are not quite ready for tape out because both contain errors.

Find and fix the following kinds of errors. (Points may be deducted if correct Verilog is identified as having errors.) (20 pts)

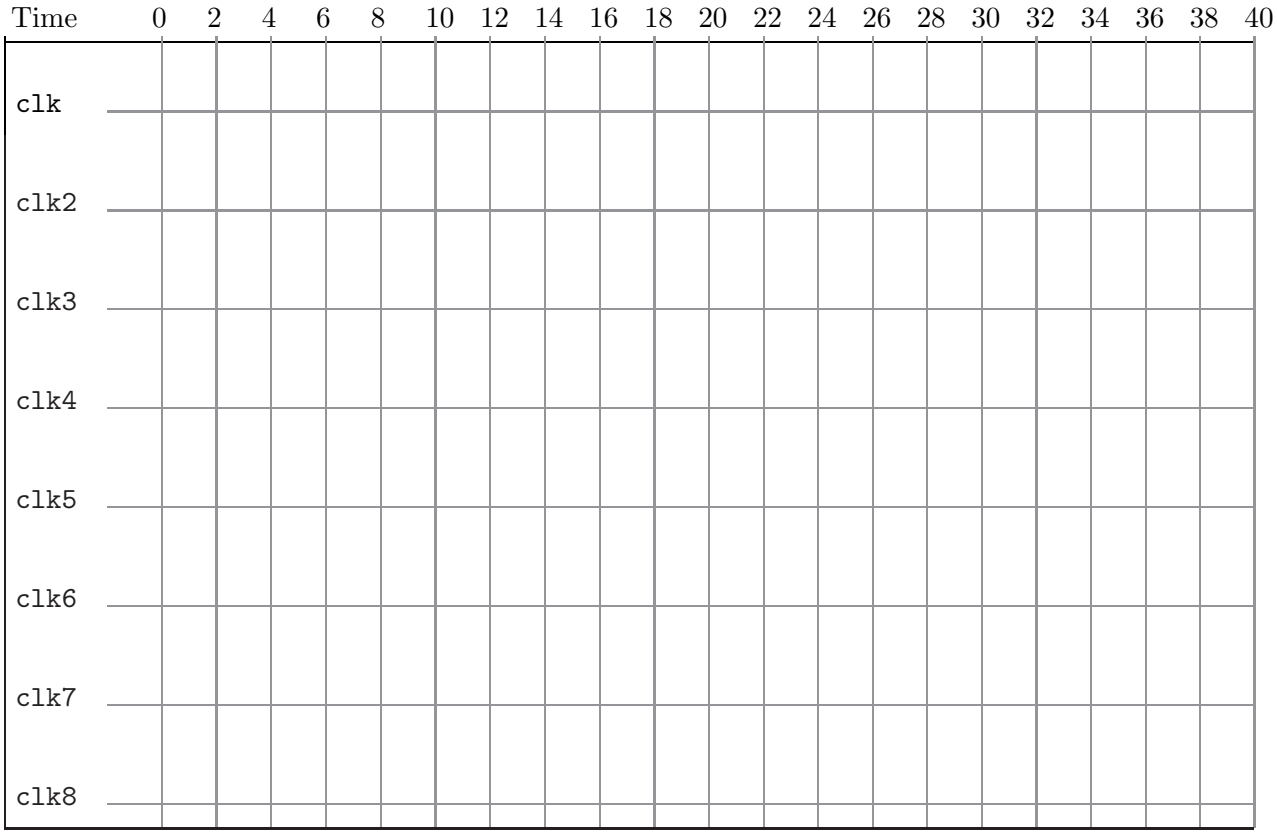
- A: One compile error. (Modelsim will not compile it.)
- B: One load error or warning. (Modelsim will compile it but will issue a warning or error message when loading it.)
- C: Three errors that result in incorrect output. The code will simulate but the output, if any, will be incorrect.

Lines with the comment `// Okay` do not have errors. None of the errors are typographical or are due to syntactic minutiæ such as missing semicolons.

```
module rearrange(y,a);
   input a;              output y;
   wire [7:0] a;         reg [7:0]  y;      wire [0:7] temp;

   wire       operation;
   assign     operation = e1.op_reverse;
   rearrange e1(temp,a,operation);

   assign     operation = e1.op_left_shift;
   rearrange e2(y,temp,operation);
endmodule

module rearrange(x,a,op);
   input      a, op;      output     x;
   wire [7:0] a;          wire [1:0] op;
   reg [7:0]  x;          reg [2:0]  ptr, ptr_plus_one;

   parameter  op_reverse     = 0;  // Reverse order of bits.            // Okay
   parameter  op_identity    = 1;  // No change.                        // Okay
   parameter  op_left_shift  = 2;  // Circular (end-around) left shift. // Okay
   parameter  op_right_shift = 3;  // Circular (end-around) right shift.// Okay

   always @( a ) for(ptr=0; ptr<8; ptr=ptr+1) begin
        ptr_plus_one = ptr + 1;                                          // Okay
        case( op )
          op_reverse:     x[ptr]          = a[7-ptr];                    // Okay
          op_identity:    x[ptr]          = a[ptr];                      // Okay
          op_right_shift: x[ptr]          = a[ptr_plus_one];             // Okay
          op_left_shift:  x[ptr_plus_one] = a[ptr];                      // Okay
        endcase
     end
endmodule
```

Problem 2: Using the grid show the register values for the first 40 time units of execution of the module below. (20 pts)

```
module clocks();
   reg clk, clk2, clk3, clk4, clk5, clk6, clk7, clk8;
   initial begin
      clk  = 0; clk2 = 0; clk3 = 0; clk4 = 0;
      clk5 = 0; clk6 = 0; clk7 = 0; clk8 = 0;
   end

   always #8 clk = ~clk;
   always @( clk ) #4 clk2 = ~clk2;
   always @( clk ) clk3 <= #10 clk;
   always @( posedge clk ) clk4 = ~clk4;
   always #2 forever #8 clk5 = ~clk5;
   always wait( clk ) #3 clk6 = ~clk6;
   always @( clk | clk4 ) clk7 = ~clk7;
   always @( clk or clk4 ) clk8 = ~clk8;

endmodule
```



Problem 3: Draw a schematic of the hardware Leonardo will synthesize for the following Verilog code examples. These should approximate the RTL schematic, showing the hardware before optimization and technology mapping. If flip flops are used, indicate if they are level triggered or edge triggered. Otherwise, don't worry about using the precisely correct gate or symbol, as long as it's functionally correct.

(*a*) Show an approximate RTL schematic for the module below. What form is the description in? *Hint: think about what form the code is in.* (6 pts)

```
module mod_a(x,y,a,b,c);
   input a,b,c;
   output x,y;
   wire [7:0] b, c;
   reg [8:0]  x, y;

   always @( a or b or c ) begin
      if( a ) begin
         x = b + c;
         y = b - c;
      end else begin
         x = b - c;
      end
   end
endmodule
```

Problem 3, continued: (*b*) Show an approximate RTL schematic for the module below. What form is the description in? *Hint: think about what form the code is in.* (6 pts)

```
module mod_b(x,y,d,e,f,g,h);
   input d,e,f,g,h;
   output x,y;
   reg    x,y;

   always @( posedge d or negedge e or posedge f )
     if( d ) begin
        x = 0;
        y = 1;
     end else if ( f ) begin
        x = 1;
     end else begin
        if( g ) x = h;
        y = h;
     end

endmodule
```

Problem 3, continued: (*c*) Show an approximate RTL schematic for the module below. Assume that the synthesis program will not infer that this module performs magnitude comparison. Use symbols $\boxed{<}$ and $\boxed{>}$ for bit comparison. (8 pts)

```
module compare(gt, lt, a, b);
   input a, b;
   output gt, lt;
   wire [2:0] a, b;
   reg        gt, lt;
   integer    i;

   always @( a or b ) begin
      gt = 0; lt = 0;
      for(i=2; i>=0; i=i-1) if( !gt && !lt ) begin
         if( a[i] < b[i] ) lt = 1;
         if( a[i] > b[i] ) gt = 1;
      end
   end

endmodule
```

Problem 4: The incomplete code below, `compare_ism`, is for a magnitude comparison module (similar to the one in the previous problem, except it's sequential).

When input `start` is set to 1, output `valid` goes to zero and the module computes `lt` and `gt`. When `lt` and `gt` are set to their proper values `valid` is set to one. The module is to compare one bit position per cycle of input `clk`. Output `valid` should go to one as soon as possible.

Complete the module so that it is in the form of an implicit state machine, synthesizable by Leonardo. The solution can be based on the combinational module `compare`, below. Don't forget signals `start` and `valid`. (20 pts) *Hint: The solution is very similar to the combinational module.* For partial credit ignore synthesizability but follow other specifications.

```
module compare(gt, lt, a, b);    // Synthesizable combinational implementation.
   input a, b;              output gt, lt;
   wire [31:0] a, b;
   reg        gt, lt;       integer    i;

   always @( a or b ) begin
      gt = 0; lt = 0;
      for(i=31; i>=0; i=i-1) if( !gt && !lt ) begin
         if( a[i] < b[i] ) lt = 1;
         if( a[i] > b[i] ) gt = 1;
      end
   end
endmodule

// Implicit state machine implementation.
module compare_ism(gt, lt, valid, a, b, start, clk);
   input a, b, start, clk;               output gt, lt, valid;
   wire [31:0] a, b;                     reg    gt, lt, valid;
   wire        start, clk;               integer i;









         if( a[i] < b[i] ) lt = 1;  // Part of solution.
         if( a[i] > b[i] ) gt = 1;









endmodule
```

Problem 5: Answer each question below.

(*a*) Complete the module below so that it will stop simulation (using the system task $stop) if there is no change in signal `heartbeat` for 1000 simulator time units. There might be many changes in `heartbeat`, but the first time `heartbeat` remains unchanged for 1000 simulator time units simulation should be stopped. *Hint: use a fork. Also, the answer is short.* (5 pts)

```
module watchdog(heartbeat);
   input heartbeat;
   wire heartbeat;




endmodule // watchdog
```

(*b*) What is a critical path? At what point in the design flow can one first find out about critical paths? (5 pts)

(*c*) Provide an example case statement in which the directive `exemplar case_parallel` is needed. What is its effect? (5 pts)

(*d*) The module below is supposed to zero the middle 3 bits of its input. It's rejected by the compiler (the "b=" line), identify and fix the problem. (5 pts)

```
module whatswrong(a,b);
   input a;       output b;
   wire [8:0] a;  wire [8:0] b;

   assign b = {a[8:6],0,a[2:0]};

endmodule
```

14 Fall 2024 Solutions

Name Solution

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 23 October 2024, 11:30-12:20 CDT

Problem 1		(22 pts)
Problem 2		(18 pts)
Problem 3		(20 pts)
Problem 4		(10 pts)
Problem 5		(30 pts)
Exam Total		(100 pts)

Alias The best part is the last part.

Good Luck!

Problem 1: [22 pts] Below is the Homework 3 Problem 1 solution with some object names shortened.

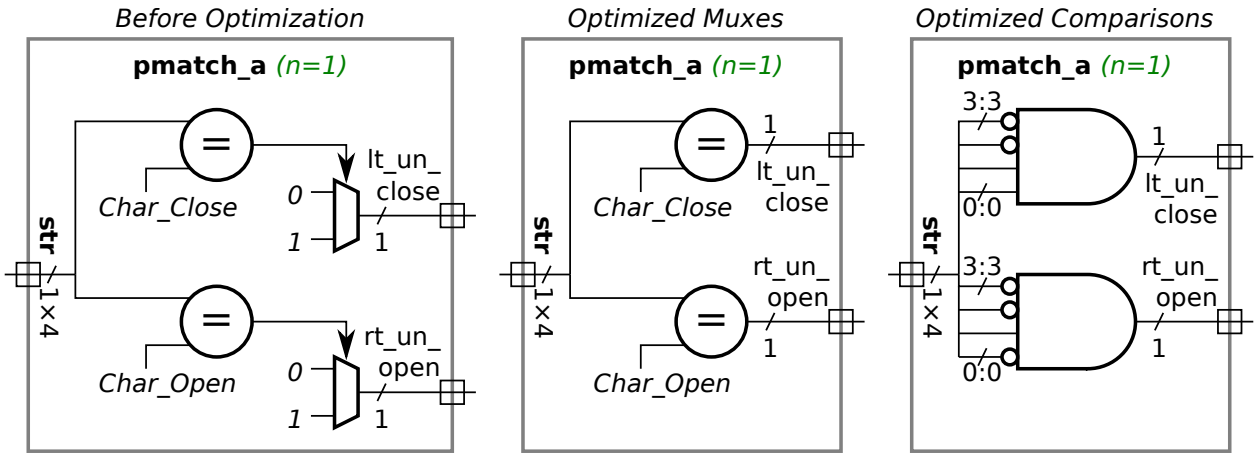
```
typedef enum logic [3:0] {Char_Blank=0, Char_Dot=1, Char_Open=2, Char_Close=3} Char;
module pmatch_a #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] lt_un_close, rt_un_open,   input uwire [3:0] str[0:n-1] );
   if ( n == 1 ) begin
      assign lt_un_close = str[0] == Char_Close ? 1 : 0;
      assign rt_un_open =  str[0] == Char_Open  ? 1 : 0;
   end else begin
      localparam int n_left = n/2;
      localparam int n_right = n - n_left;
      localparam int wl = $clog2(n_left+1),  wr = $clog2(n_right+1);
      uwire [wl-1:0] lt_close, lt_open;
      uwire [wr-1:0] rt_close, rt_open;
      pmatch_a #(n_left,  wl) plt( lt_close, lt_open, str[0:n_left-1] );
      pmatch_a #(n_right, wr) prt( rt_close, rt_open, str[n_left:n-1] );

      uwire logic signed [wn-1:0] delta = lt_open - rt_close;
      assign lt_un_close = delta <  0 ? lt_close - delta : lt_close;
      assign rt_un_open =  delta >= 0 ? rt_open  + delta : rt_open;
   end
endmodule
```

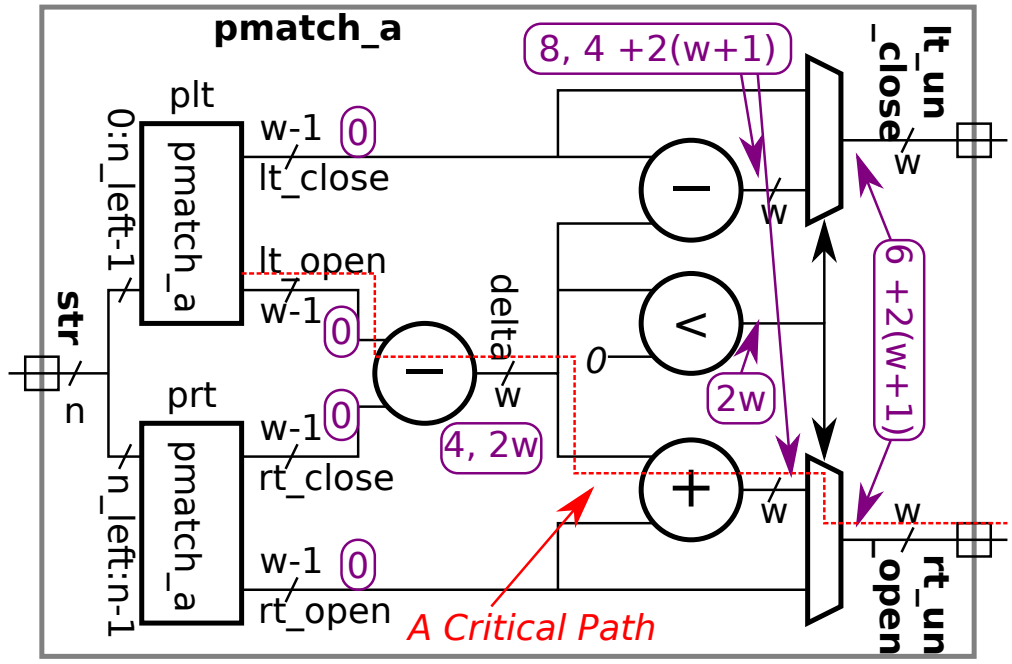
(*a*) Show the hardware that will be inferred for the base case. Show hardware after optimization taking into account constants.

☑ Show inferred hardware for base (`n==1`) case of the module above. ☑ Show input and output ports.

☑ Optimize taking into account ☑ constant values of all kinds. ☑ Don't miss the `Char` definition above the module. ☑ Don't show a comparison unit such as `==`, instead show the gates from which it was made and ☑ optimize them, ☑ taking into account the number of bits on each output port.

Solution appears below. The left-most version is without optimization, in the middle version the multiplexors have been optimized to wire, and in the rightmost version the comparison units have been optimized to AND gates.





Appearing above is hardware that will be inferred for the non-base case.

(*b*) Compute the cost of the hardware at this level (ignore what's inside `plt` and `prt`) based on the simple model using the bit widths from the diagram, such as `w-1`.

☑ Show the cost of each component *except for hardware inside of* `plt` *and* `prt`.

☑ Be sure to show the cost of the optimized comparison unit!

So far as computing cost here is concerned there are three types of components: the multiplexors, the comparison unit (in this case), and the adder and subtractors.

Multiplexors: The cost of a w-bit mux is $3w\ \mathrm{u_c}$. There are two multiplexors, so the total multiplexor cost is $6w\ \mathrm{u_c}$.

Comparison: The comparison unit checks if `delta` is negative. To do that just check if the most-significant bit (sort of a sign bit) is `1`. So, the comparison cost is zero.

Adder and subtractors: The cost of an x-bit ripple adder is $9x\ \mathrm{u_c}$. Note that an x-bit ripple unit has x-bit inputs and, when one includes the carry-out, computes an $(x+1)$-bit result. The unit that computes `delta` is $w-1$ bits, and lest this get too tedious, treat the other subtractor and adder as having w-bit inputs. So, the ripple units' combined cost is $9(w-1+2w)\ \mathrm{u_c} = [27w-9]\ \mathrm{u_c}$.

(*c*) Compute the delay through the module **starting from** launch points `lt_close`, `lt_open`, `rt_close`, and `rt_open`. The capture points are `lt_un_close` and `rt_un_open`. Use the bit widths from the diagram, such as `w-1`.

☑ Show the arrival time at each wire from launch to capture.

☑ Take into account cascaded ripple units and ☑ and the optimized comparison unit.

The timing appears on the diagram above. Also shown, though not asked for, is the critical path (in red). For arrival time at the outputs of the ripple units two times are shown: the time of the least-significant bit and the time of the most-significant bit. For example, the LSB of `delta` is arrives at $4\ \mathrm{u_t}$ and the MSB arrives at $2w\ \mathrm{u_t}$. For clarity the unit, $\mathrm{u_t}$, is omitted from the diagram. The comparison unit just passes the MSB of `delta` (it does not add any additional delay).

Problem 2: [18 pts] Appearing below is an alternative solution to Homework 3 Problem 1. The only difference is the last few lines.

```
module pmatch_b #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] lt_un_close, rt_un_open,
     input uwire [3:0] str[0:n-1] );
   if ( n == 1 ) begin
      assign lt_un_close = str[0] == Char_Close ? 1 : 0;
      assign rt_un_open = str[0] == Char_Open ? 1 : 0;
   end else begin
      localparam int n_left = n/2;
      localparam int n_right = n - n_left;
      localparam int wl = $clog2(n_left+1),  wr = $clog2(n_right+1);
      uwire [wl-1:0] lt_close, lt_open;
      uwire [wr-1:0] rt_close, rt_open;
      pmatch_b #(n_left,  wl) plt( lt_close, lt_open, str[0:n_left-1] );
      pmatch_b #(n_right, wr) prt( rt_close, rt_open, str[n_left:n-1] );
      uwire logic signed [wn-1:0] delta = lt_open - rt_close;
      // Lines above are identical to pmatch_a.

      uwire [wn-1:0] delta_n = delta <  0 ? delta : 0;
      uwire [wn-1:0] delta_p = delta >= 0 ? delta : 0;
      assign lt_un_close = lt_close - delta_n;
      assign rt_un_open  = rt_open  + delta_p;
   end
endmodule
```

(*a*) Show the hardware that will be inferred for `pmatch_b`. For your convenience the hardware for `pmatch_a` is shown in the upper right. *Note: In the original exam the condition for* `delta_n` *was* `delta <= 0` *and the condition for* `delta_p` *was* `delta > 0`. *Though the hardware computed the correct result, the comparison would have been more expensive since it would have had to check for a zero condition, not just negative.*

☑ Show inferred hardware on the facing page.

Solution appears on the facing page. One major difference is that the addition and subtraction are done *after* the multiplexors. Also notice that each multiplexor has a constant input, zero. Finally, notice that the output of the comparison unit for `delta<0` can be used for the line needing `delta>=0` since one or the other is true.

(*b*) Compute the simple-model cost of the hardware.

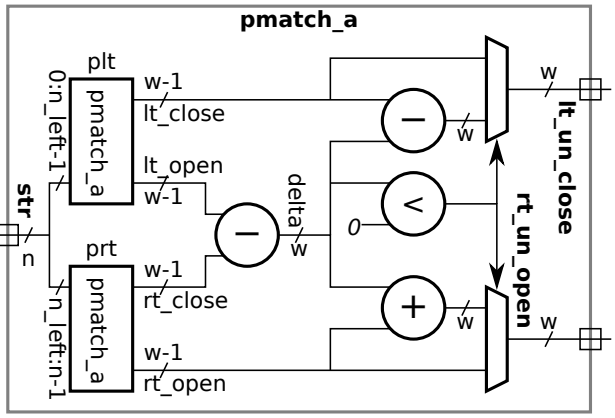
☑ Write *same* next to components that cost the same as corresponding components in `pmatch_a` and ☑ compute the cost of other components ☑ after optimization.

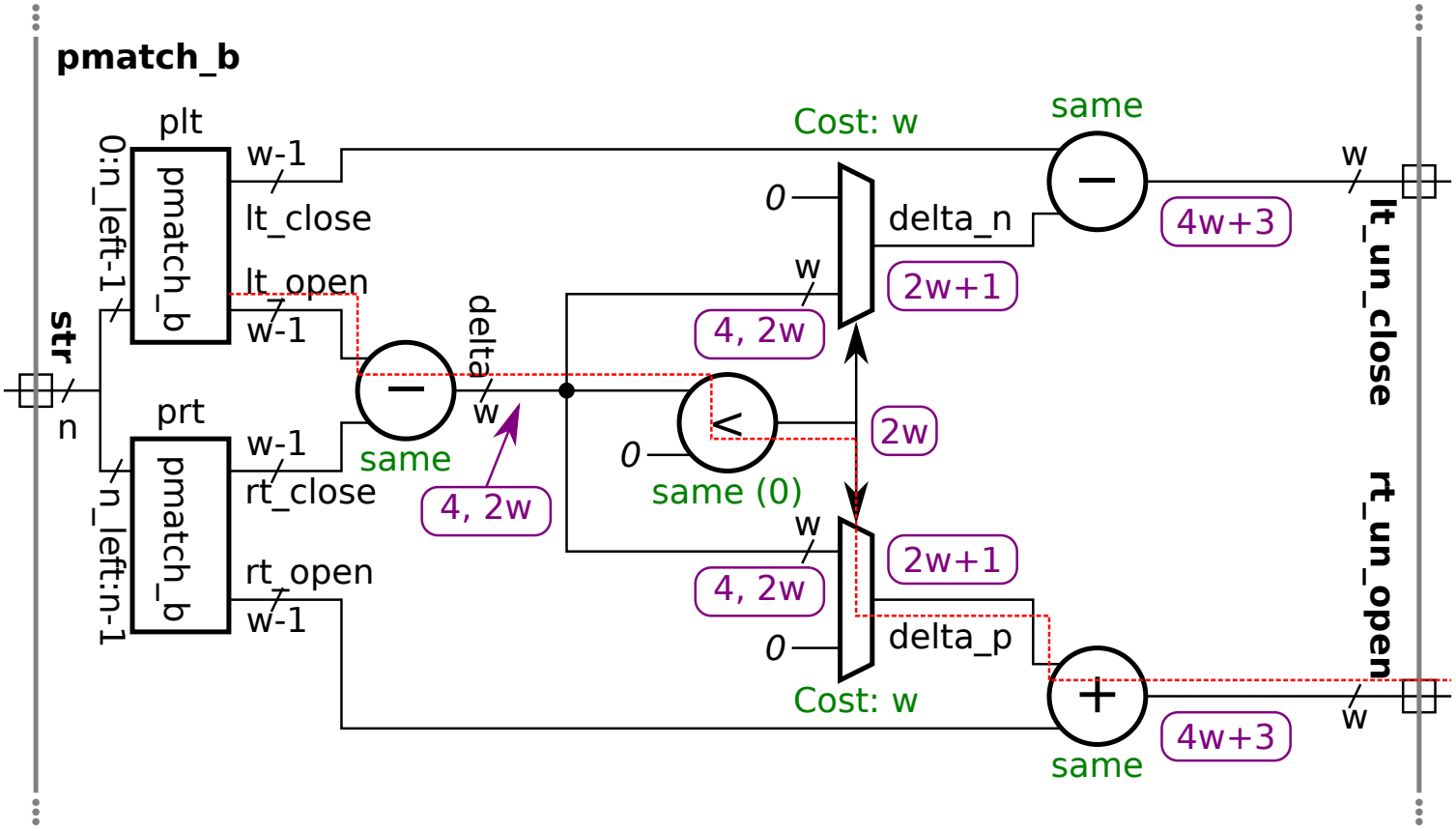
The costs are shown in the diagram in green. The only significant difference is the multiplexors. Because each has a constant input the cost of each is now $w\,\mathrm{u_c}$ (the multiplexors in `pmatch_a` cost $3w\,\mathrm{u_c}$ each).

(*c*) Compare the critical path lengths.

☑ Will the critical path in `pmatch_a` be much different than the one in `pmatch_b`? ☑ Explain.

Yes, the critical paths will be very different. In the diagram the arrival times are circled and the critical path is shown as a red dashed line. Because there is a multiplexor with a late-arriving select signal between the initial subtraction and the subsequent add or subtract, the ripple units are no longer cascaded. That means the least significant bit does not arrive at the adder and subtractor until $[2w+1]\,\mathrm{u_t}$ (due to the multiplexor select signal not stabilizing until $2w\,\mathrm{u_t}$). As a result `pmatch_b` takes nearly twice as long.


pmatch_a
plt
pmatch_a
0:n_left-1
w-1
lt_close
lt_open
w-1
str
n
prt
pmatch_a
n_left:n-1
w-1
rt_close
w-1
rt_open
delta
w
0
w
lt_un_close
rt_un_open
w


pmatch_b
plt
pmatch_b
0:n_left-1
w-1
lt_close
lt_open
w-1
str
n
prt
pmatch_b
n_left:n-1
w-1
rt_close
rt_open
w-1
same
delta
w
4, 2w
0
same (0)
Cost: w
same
0
delta_n
w
4, 2w
2w+1
4w+3
2w
w
4, 2w
2w+1
0
delta_p
Cost: w
same
4w+3
w
lt_un_close
rt_un_open
w

Problem 3: [20 pts] Appearing below are some of the `dot` modules from the solution to Homework 1. On the facing page is incomplete module `dotn`. Complete `dotn` so that it describes hardware that computes the dot product of `n`-element vectors recursively, where `n` is the parameter. That is, `dotn` must instantiate `dotn` and should instantiate `mult` and `add` where needed.

```
module mult #( int w = 5 ) ( output uwire [w-1:0] p, input uwire [w-1:0] a, b );
   assign p = a * b;
endmodule

module add #( int w = 5 ) ( output uwire [w-1:0] s, input uwire [w-1:0] a, b );
   assign s = a + b;
endmodule

module dot2 #( int w = 5 )
   ( output uwire [w-1:0] dp,    input uwire [w-1:0] a[1:0], b[1:0] );
   // Computes dp = a[0] * b[0] + a[1] * b[1];
   uwire [w-1:0] p0, p1;
   mult #(w) m0(p0, a[0], b[0] );
   mult #(w) m1(p1, a[1], b[1] );
   add #(w) ad(dp, p0, p1 );
endmodule

module dot3 #( int w = 5 )
   ( output uwire [w-1:0] dp,   input uwire [w-1:0] a[2:0], b[2:0] );
   // Computes   dp = a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
   uwire [w-1:0] p0, p2;
   dot2 #(w) d0( p0, a[1:0], b[1:0] );
   mult #(w) m2( p2, a[2], b[2] );
   add #(w) a2(dp, p0, p2 );
endmodule
```

☑ Complete `dotn` so that it describes tree-structured hardware computing an `n`-element dot product. The tree depth should be $\lceil \lg n \rceil$.

☑ Instantiate `mult` for multiplication and `add` for addition, and of course `dotn` for a dot product of a smaller vector.

☑ To keep things easy all wires are `w` bits.

The solution appears below. Note that the multiplication is performed in the base case and that the adds are done in the recursive instances.

```
module dotn
  #( int w = 5, n = 4 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] a[n-1:0], b[n-1:0] );

   // SOLUTION

   if ( n == 1 ) begin

      // Base Case: Just multiply.
      //
      mult #(w) m( dp, a[0], b[0] );

   end else begin

      // Recursive Case: Split inputs between recursive instances ..
      //
      localparam int nlo = n/2;
      localparam int nhi = n - nlo;
      uwire [w-1:0] dplo, dphi;
      dotn #(w,nlo) dlo( dplo, a[nlo-1:0], b[nlo-1:0] );
      dotn #(w,nhi) dhi( dphi, a[n-1:nlo], b[n-1:nlo] );
      //
      // .. and add their outputs ..
      //
      add #(w) a( dp, dplo, dphi );

   end

endmodule
```

Problem 4: [10 pts] Appearing below is the logarithmic shifter presented in class, followed by a version that's supposed to be better (but isn't). The hoped-for improvement is due to instantiating the exact number of multiplexors (`muxw2`) needed, rather than enough for the maximum shift amount.

```
module shift_right_logarithmic #( int w = 16, lgw = $clog2(w) )
   ( output uwire [w-1:0] shifted,
     input uwire [w-1:0] un,   input uwire [lgw-1:0] amt );
   // This module is correct.
   uwire [w-1:0] s[lgw:-1];
   assign s[-1] = un;
   for ( genvar i=0; i<lgw; i++ )
     muxw2 #(w) st( s[i],  amt[i],  s[i-1],  s[i-1] >> ( 1 << i )  );
   assign shifted = s[lgw-1];
endmodule

module shift_right_logarithmic_better_maybe #( int w = 16, lgw = $clog2(w) )
   ( output uwire [w-1:0] shifted,
     input uwire [w-1:0] un,  input uwire [lgw-1:0] amt );
   uwire [w-1:0] s[lgw:-1];
   assign s[-1] = un;

   // Use exactly the number of stages needed!!!
   uwire [lgw-1:0] lg_amt;              // LINE ADDED
   my_clog2 #(lgw) mc( lg_amt, amt );  // LINE ADDED. Set lg_amt = $clog2(amt) = ⌈lg amt⌉;

   for ( genvar i=0; i<lg_amt; i++ )   // LINE DIFFERS
     muxw2 #(w) st( s[i],  amt[i],  s[i-1],  s[i-1] >> ( 1 << i )  );
   assign shifted = s[lg_amt-1];       // LINE DIFFERS
endmodule
```

☑ Why won't the Verilog above compile?

The Verilog won't compile because the `for` loop is a generate loop and its stop condition, `i<lg_amt`, is not an elaboration-time constant expression due to `lg_amt` being a module output (of the `mc` instance of the `my_clog2` module). The loop would have to be a generate loop because it is in module scope and because the iterator is declared `genvar`.

☑ Is it possible to fix the Verilog error in such a way that cost is lower with smaller shift amounts? ☑ Explain.

No. The cost is determined by the amount of hardware needed to synthesize the module. Once the module is synthesized, manufactured, and shipped to a customer its cost (meaning the amount of hardware) can't change. This isn't Hogwarts, we're muggles (at least I am).

☑ Is it possible to fix the Verilog error in such a way that the delay reported by a synthesis program is lower? ☑ Explain.

No, because the delay reported by the synthesis program will be based on the critical path, which is the longest path.

☑ Is it possible to fix the Verilog error in such a way that the delay actually is lower? ☑ Explain.

If you must have a shorter delay for a lower shift amount that is possible but (1) it won't be much of an improvement below galactic sizes and (2) the logic outside the module will have to be designed to take advantage of the lower delay with sorter shift amounts. So, though the answer to this question is yes, as a practical matter the answer is still no.

Problem 5: [30 pts] Answer the following Verilog questions.

(*a*) The module below uses multidimensional arrays.

```
//                          2       1                       3    4       1    2
module mda( input uwire [2:1] c [5:1],    input uwire [7:1][2:1] a [5:1][3:1] );
   //   ☑ Add dimension(s) to the declaration of e so that the assignment is correct.
   //
   //         SOLUTION
   uwire      [2:1] e [5:1]      = c;


   //   ☑ Add dimension(s) to the declaration of b so that the assignment is correct.
   //
   //         SOLUTION
   uwire      [2:1] b            = a[1][1][1];


   logic g [7:0];
   logic [7:0] h;

   initial begin
//  ☑ Which is correct,
      ◯ only the assignment to g,   ◯ only the assignment to h, or   ⊗ both are correct.
    ☑ Explain.

      g[1] = h[1];
      h[1] = g[1];
   end

endmodule
```

☑ What is the size of `c`, in bits? ☑ What is the size of `a`, in bits?

The size of object `c` is $2 \times 5 = 10$ bits. The size of object `a` is $7 \times 2 \times 5 \times 3 = 210$ bits.

Explanation for multiple-choice question about `g` and `h`. It is true that `g` is an unpacked array and `h` is a packed array and the two kinds of array work differently. But in this case both `g[1]` and `h[1]` refer to 1-bit quantities, and so assigning one to the other is no problem. (Possible final-exam question: ask about `h[1:0]` and `g[1:0]`.)

(*b*) In the module below indicate whether each code fragment is correct.

```
module kinds #( logic [31:0] pg = 123 )
   ( output uwire [31:0] o0,  input uwire [31:0] ik );

   //  ☑ Is the line below correct?      ⊗ Yes   ◯ No   ☐ If not, explain.

   localparam logic [31:0] z02 = pg + 4755;



   //  ☑ Is the line below correct?      ◯ Yes   ⊗ No   ☑ If not, explain.

   localparam logic [31:0] z03 = ik;
```

A localparam can only be assigned an elaboration-time constant expression (including literals and elaboration-time constants). Since `ik` is a module input it cannot be an elaboration-time constant.

```
   //  ☑ Are the lines below correct?    ◯ Yes   ⊗ No   ☑ If not, explain.

   localparam logic [31:0] z04;
   assign z04 = pg;
```

Though `pg` is an elaboration time constant, it must be assigned to `z04` in the declaration statement (the statement starting starting with `localparam`). Put another way every `localparam` declaration must include the constant value.

```
   //  ☑ Are the lines below correct?    ◯ Yes   ⊗ No   ☑ If not, explain.

   uwire [31:0] z10 = pg;
   assign z10 = ik;
```

Object `z10` is declared `uwire`. A `uwire` must have exactly one driver. Both lines above are drivers, which is one too many. To keep the `assign` line one would need to remove `=pg` from the `uwire` declaration (but keep the rest).

```
   //  ☑ Is the line below correct?      ⊗ Yes   ◯ No   ☐ If not, explain.

   uwire [31:0] z13 = ik;



endmodule
```

(*c*) When we run a synthesis program we specify a delay target. In class we often synthesize twice, once with a delay target of 100 ns and a second time with a target of 0.1 ns. What is the harm in specifying a delay target lower (faster) than one needs? Isn't faster better?

☑ Harm in setting delay target too low is:

Short answer: The harm is synthesized hardware that's more costly than it would have been with the correct delay.

The delay target should be set to the delay that's needed by a design. For example, the design team may have decided that they would like a 2.5 GHz clock frequency and so they would set the delay target to $\frac{1}{2.5\,\text{GHz}} = 400\,\text{ps}$. The synthesis program will optimize delay until the 400 ps target is met, and then optimize cost. Typically the lower the delay, the more the hardware will cost.

In Scenario A the team needs 400 ps but the person running the synthesis program specified a target of 200 ps and didn't tell anyone. When the more-costly-than-they-need-to-be manufactured components are used in a product they are clocked at 400 ps, wasting their potential. The competition might have a less expensive product that runs at the same speed.

In Scenario B the team needs 400 ps, the person running the synthesis program specifies a target of 200 ps, and told the team. So the design is run at 200 ps. Though the hardware runs faster, it's of no benefit because the toaster makes great toast with either component. There's no point in sampling the image sensors any faster to 20 Hz to make good toast.

(*d*) A 32-bit signed integer, say i, is converted into a 32-bit IEEE 754 floating-point format (8-bit exponent, 23-bit significand) and then back into a 32-bit integer, j.

☑ Is it guaranteed that $i = j$ for all $-2^{31} \le i < 2^{31}$? ☑ Explain based on the FP representation.

No. The significand is only 23 bits, meaning the fraction is 24 bits (counting the implicit 1). Consider two integers $253{,}969{,}770_{10} = f23{,}456a_{16}$ and $253{,}969{,}771_{10} = f23{,}456b_{16}$. Each is seven hexadecimal digits and so would need at least 28 bits to represent. To represent these in the FP format the significand would be set to the most significant 24 bits, $f2{,}3456_{16}$ (including the implicit 1), omitting the least-significant hex digit a or b. (The exponent would be set to 154 for both numbers.) Since the two numbers have identical FP representations converting them back to integers will yield the same number, $f23{,}4570_{16}$.

Appearing below is the complete FP representation of $253{,}969{,}770_{10} = f23{,}456a_{16}$ and $f23{,}456b_{16} = 253{,}969{,}771_{10}$ in 32-bit (single, which is the format described in the problem) and 64-bit (double, a higher-precision format) formats. Notice that the fractions (F) of the numbers' single representations are the same, but the fractions of their double representations are different.

```
Value 253969770.0000000000000000 == 2.539698e+08
  Single:  0x4d723457           1299330135
           S 0   E 0x9a = 154     F 0x723457
  Double:  0x41ae468ad4000000   4732797820489170944
           S 0   E 0x41a = 1050   F 0xe468ad4000000

Value 253969771.0000000000000000 == 2.539698e+08
  Single:  0x4d723457           1299330135
           S 0   E 0x9a = 154     F 0x723457
  Double:  0x41ae468ad6000000   4732797820522725376
           S 0   E 0x41a = 1050   F 0xe468ad6000000
```

15 Fall 2023 Solutions

Name Solution

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Friday, 27 October 2023, 11:30-12:20 CDT

Problem 1	_____	(30 pts)
Problem 2	_____	(25 pts)
Problem 3	_____	(30 pts)
Problem 4	_____	(15 pts)
Exam Total	_____	(100 pts)

Alias Again on 8 April!

Good Luck!

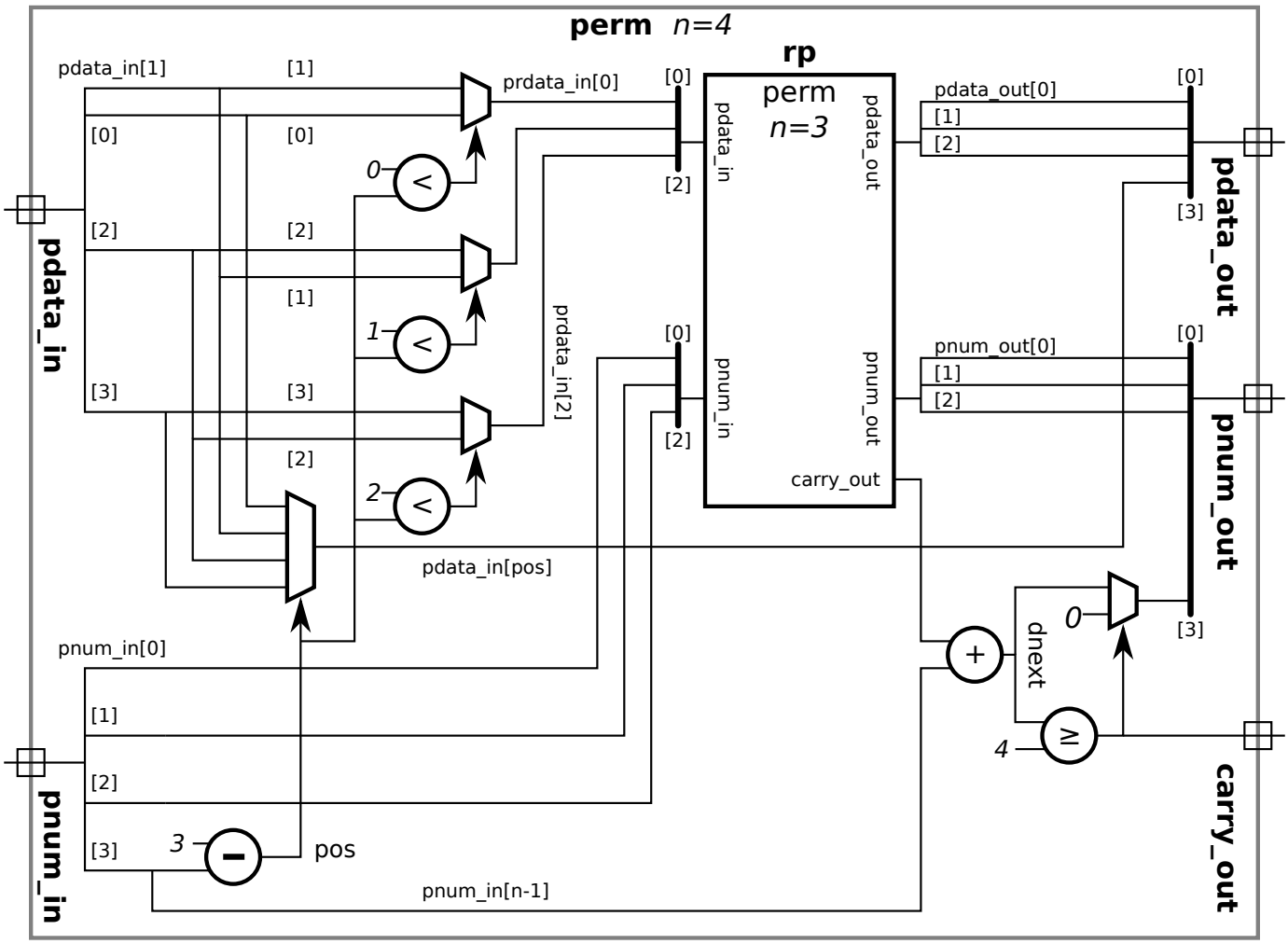
Problem 1: [30 pts] Appearing below is the permutation module from the solution to Homework 3. Using the illustration of the ports show the inferred hardware for an instantiation with `n=4`. Show the `n=4` instantiation but not what is inside the `n=3` recursive instantiation.

```
module perm
  #( int w = 8, n = 20, wd = $clog2(n) )
   ( output uwire [w-1:0] pdata_out[n],    output uwire [wd-1:0] pnum_out[n],
     output uwire carry_out,
     input uwire [w-1:0] pdata_in[n],      input uwire [wd-1:0] pnum_in[n] );

   if ( n == 1 ) begin

      assign pdata_out[0] = pdata_in[0];
      assign carry_out = 1;
      assign pnum_out[0] = 0;

   end else begin

      uwire [wd-1:0] pos = n - 1 - pnum_in[n-1];
      assign pdata_out[n-1] = pdata_in[pos];
      uwire [w-1:0] prdata_in[n-1];
      for ( genvar i=0; i<n-1; i++ )
        assign prdata_in[i] =  i < pos  ?  pdata_in[i]  :  pdata_in[i+1];

      uwire co;
      perm #(w,n-1,wd) rp( pdata_out[0:n-2], pnum_out[0:n-2], co,
                           prdata_in, pnum_in[0:n-2] );

      uwire [wd-1:0] dnext = pnum_in[n-1] + co;
      assign carry_out = dnext >= n;
      assign pnum_out[n-1] = carry_out ? 0 : dnext;

   end

endmodule
```

☑ Show inferred hardware for `n=4`. Be sure to use ☑ the illustrated module ports and to show ☑ the recursively instantiated module (but not its contents).

☑ Show hardware, ☑ do not confuse elaboration-time computation with computation hardware.

Solution appears below. Notice that elaboration-time constants such as `i` and `n` are replaced by their values. Notice also that `pdata_in[i]` for `i=0` is inferred simply as a wire connecting to input `pdata_in[0]`, whereas `pdata_in[pos]`, because `pos` is not a constant, is inferred as a multiplexor with data inputs connecting to each `pdata_in` input.



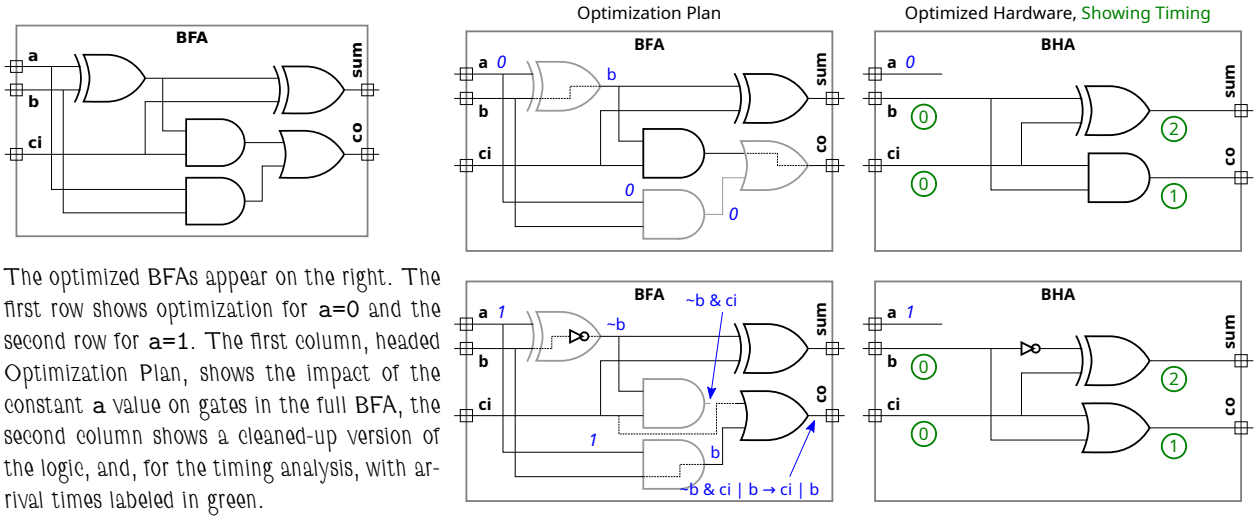
Problem 2: [25 pts] A ripple adder to compute $a + b$ is to be used in situations where a is a constant.

(a) Find the cost and delay of a BFA with input a constant (for use in the ripple adder). A BFA is shown for your convenience.

☑ Show the BFA(s) optimized for input a constant.

☐ Use a truth table to find optimizations not revealed by constant pushing: in a correct solution the delay does not depend upon `a`.

☑ Show simple-model cost of this(these) module(s) and ☑ show simple-model delay(s) of this(these) module(s).



The optimized BFAs appear on the right. The first row shows optimization for `a=0` and the second row for `a=1`. The first column, headed Optimization Plan, shows the impact of the constant `a` value on gates in the full BFA, the second column shows a cleaned-up version of the logic, and, for the timing analysis, with arrival times labeled in green.

A binary full adder with one constant input is little different than a binary half adder, and so a constant-input BFA is labeled BHA. The optimization for the `a=0` case is straightforward. For the `a=1` case an initial optimization would use both an AND gate and an OR gate to compute `co`. But the logic can be simplified further by noting that when `b=1` directly connecting `ci` to the OR gate has no effect, and when `b=0` directly connecting `ci` to the OR gate has the intended effect. Or perhaps one just remembers the Boolean algebra identity $x + \overline{x}y = x + y$.

The cost of each module is $3 + 1 = 4\,\mathrm{u_c}$. Actually, the cost of each module can be reduced to just $3\,\mathrm{u_c}$ by splitting the XOR into three gates and using the AND or OR gate to replace one of those gates with other zero-cost changes needed to compute exclusive or. Final exam problem?

The arrival times are labeled in green. In both modules the delay of `sum` is $2\,\mathrm{u_t}$ and the delay of `co` is $1\,\mathrm{u_t}$. Lucky for us the delay of `co` is $1\,\mathrm{u_t}$, because that impacts the delay of the ripple adder.

(b) On the facing page show the optimized hardware, cost, LSB delay, and MSB delay of a w-bit ripple adder for computing $a + b + c_{\mathrm{in}}$, where c_{in} is a carry-in bit (`cin` in the diagram) and a is a constant. (☑ See the check box items for details.) Use the illustration on the facing page as a starting point.

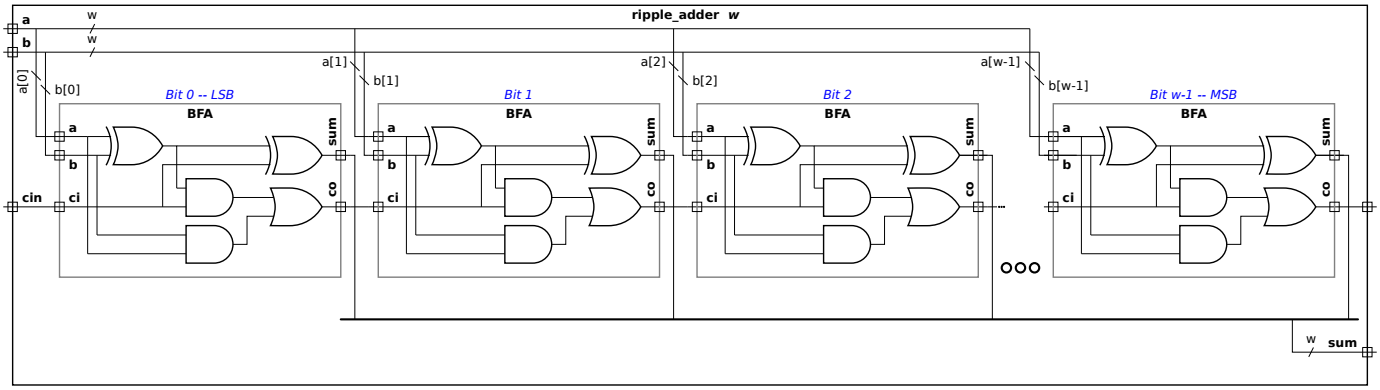
☑ Show the hardware optimized for a constant `a` and a non-constant `cin`.

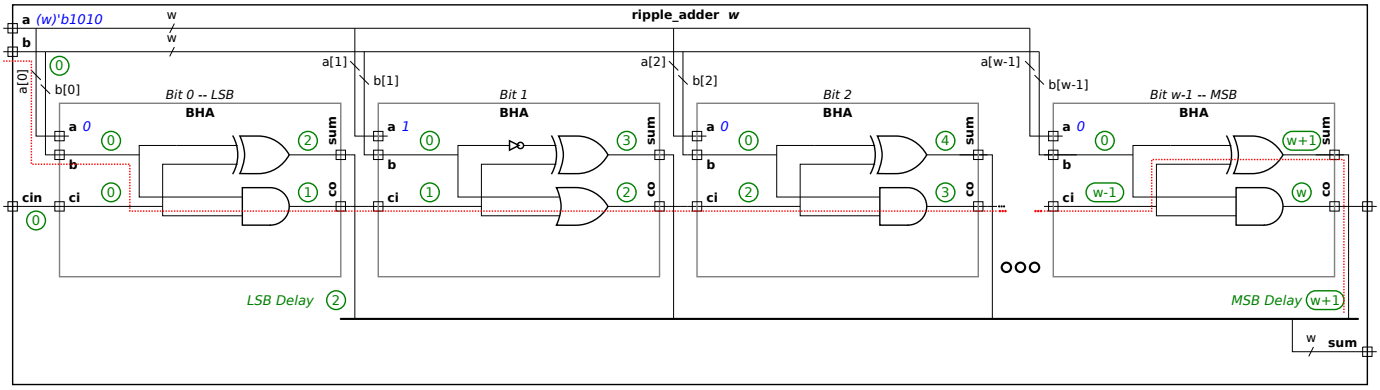
☑ Compute the simple-model cost of this hardware in terms of w.

☑ Compute the simple-model delay of the LSB of the sum.

☑ Compute the simple-model delay of the MSB of the sum in terms of w and ☑ show the critical path.

☑ Don't forget that `a` is a constant.





Solution to part b appears above. The exact BHA units to use depend on the value of a. The diagram is for $a = 1010_2$.

The cost of each BHA is $4\,\mathrm{u_c}$, so the total cost is $4w\,\mathrm{u_c}$. Based on the analysis shown in green on the diagram the LSB delay is $2\,\mathrm{u_t}$ and the MSB delay is $[w+1]\,\mathrm{u_t}$. The critical path for the MSB is shown in red.

(*c*) If `cin` were removed (or set to zero) the cost and delay of the optimized adder would depend on `a`. Explain why, and illustrate with the example of `a=2`.

☑ How are cost and delay dependent on `a` when `cin` removed? ☑ Explain using the example `a=2`.

If `cin=0` the cost of the bit 0 BHA drops to zero. If bit `a[0]=0` then the `co` of the bit 0 BHA is 0, a constant, while if `a[0]=1` the `co` output is `b[0]`, not a constant. So for the case of $a = 10_2$ the `co` output of the bit 0 BHA is 0, but the `co` output of the Bit 1 BHA is `b[1]`, which is not a constant. For this $a = 10_2$ case the cost of the bit 0 and bit 1 BHAs is zero, but the cost of the remaining BHAs is $4\,\mathrm{u_c}$ each. For $a = 100_2$ the cost of the first three BHAs would be zero. So the cost of the constant adder with `cin=0` depends on the number of consecutive 0s starting at the LSB of `a`.

Problem 3: [30 pts] Answer the following Verilog questions.

(*a*) The module below makes extensive use of multidimensional arrays.

```
//                          2       1                       3    4       1    2
module mda( input uwire [2:1] c [5:1],    input uwire [7:1][2:1] a [5:1][3:1] );

   //   ☑ Add dimension(s) to the declaration of e so that the assignment is correct.
   //
   //     SOLUTION
   uwire    [2:1]    e            = c[1];


   //   ☑ Add dimension(s) to the declaration of b so that the assignment is correct.
   //
   //     SOLUTION
   uwire     [7:1][2:1]   b [3:1]  = a[1];


   logic g [7:0];
   logic [7:0] h;

   initial begin
//  ☑ Which is correct,  ◯ the assignment to g  or  ⊗ the assignment to h.  ☑ Explain.
      g = 1;  // Compile error because g is an unpacked array and 1 is a scalar.
      h = 1;  // Correct, h is a packed array and so is treated as an integer.
   end

endmodule
```

☑ What is the size of c, in bits? ☑ What is the size of a, in bits?

The size of object c is $2 \times 5 = 10$ bits. The size of object a is $7 \times 2 \times 5 \times 3 = 210$ bits.

(*b*) The module below does not compile.

```
module more_stuff #( int n = 20 ) ( output uwire [31:0] sum, input uwire [31:0] a [ n ] );
   logic [31:0] acc;
   always_comb begin
      acc = a[0];
      for ( int i=1; i<n; i++ )
        my_fixed_adder a1(acc, acc, a[i] );
   end
   assign sum = acc;
endmodule
```

☑ Describe the major problem. ☑ **DO NOT** try to fix the problem.

A module cannot be instantiated in procedural code, which the code above is doing with `my_fixed_adder`. This is a major problem because it can't be fixed by just changing a declaration or adding new objects. The instantiation must be removed from the procedural code. Another problem is that `acc` connects to two parts of `my_fixed_adder`. A reasonable guess would be that one of those is an output and the other is an input. It makes no sense to connect the same object to both an input and an output.

(*c*) The module below is supposed to set $x = a^2 + b^2$.

```
module wrong_way( output logic [31:0] x, input uwire [15:0] a, b );
   logic [31:0] asq;
   uwire [31:0] bsq = b * b;

   // initial asq = a * a;  // Original line.
   always_comb asq = a * a; // Corrected line.
   always_comb x = asq + bsq;

endmodule
```

☑ Explain the problem. ☑ Using sample inputs show the expected output and the actual output.

☑ Fix the problem.

Because `asq` is assigned in an `initial` block it will only be assigned once. Suppose at $t = 0$ the inputs are `a=2` and `b=3`. Then the correct output will appear, `x=13`. But suppose at $t = 1$ the inputs change to `a=5` and `b=6`. Object `asq` will keep its initial value, 4, and so the output will be $x = 2^2 + 6^2 = 40$ (computed using the $t = 0$ value of a and the $t = 1$ value of b).

The simplest fix is to change `initial` to `always_comb`, that's shown above.

(*d*) The module below does not compile.

```
module my_adder( output uwire [31:0] s, input uwire [31:0] a, b );
   always_comb s = a + b;
endmodule

module my_adder( output logic [31:0] s, input uwire [31:0] a, b );
   // Fixed module.
   always_comb s = a + b;
endmodule
```

☑ Why won't module above compile? ☑ Fix problem by changing declarations.

Because `s` is assigned in procedural code it must be a var kind, not a net kind. (A `uwire` is a net kind.) The fixed module is shown below the broken one.

(*e*) The module below compiles but does not provide the expected outputs, $p_a = a^2$, $p_b = b^2$, and $p = a^2 + b^2$.

```
module incorrect_way( output logic [31:0] pa,pb,p, input uwire [15:0] a, b );
   wire [31:0] sq;
   assign sq = a * a;
   always_comb pa = sq;
   assign sq = b * b;
   always_comb pb = sq;
   always_comb p = pa + pb;
endmodule
```

```
module correct_way( output logic [31:0] pa,pb,p, input uwire [15:0] a, b );
   /// SOLUTION
   uwire [31:0] sqa, sqb;
   assign sqa = a * a;
   always_comb pa = sqa;
   assign sqb = b * b;
   always_comb pb = sqb;
   always_comb p = pa + pb;
endmodule
```

☑ What will be the values of outputs `pa`, `pb`, and `p`?

If $a \neq b$ the value of each output will be `x`, the Verilog value indicating (in this case) conflicting drivers to an output. If $a = b$ then the correct result will be computed.

☑ Describe the problem. ☑ Fix it.

The problem is that `sq` is continuously assigned in **two** places, which though it does not violate any Verilog rules (note that `sq` is declared `wire` rather than `uwire`) it nevertheless does nothing useful. Suppose the `a*a` line drives `sq[0]` toward `0` and the `b*b` line drives bit `sq[0]` toward `1`. The value of `sq[0]` will be `x`, which is not what we want.

A simple fix is to use different objects for `a*a` and `b*b`. That is shown above.

Grading Note: Many students incorrectly described the value of `sq` as alternating between `a*a` and `b*b`. That doesn't happen because the simulator computes `sq` by first combining the `a*a` and `b*b` values (maybe using an old `a` or `b`, but always combining the two). So there is never a time when `sq` is cleanly equal to `a*a` or `b*b` (unless `a=b`).

Problem 4: [15 pts] Answer each question below.

(*a*) A company has two teams, A (very good) and C (slackers) working on modules and a testbench for an important product. Describe the following consequences:

☑ The A team works on the modules and the C team works on the testbench. A possible bad outcome is:

The testbench passes the modules, indicating that the modules produce the correct outputs. The company then manufactures the modules and ships them to customers. The customers discover flaws in the modules that the testbench should have, but didn't, uncover. Fortunately the customers weren't avionics or medical equipment manufacturers.

☑ The A team works on the testbench and the C team works on the modules. A possible bad outcome is:

The testbench correctly identifies erroneous module outputs and the C team fixes problem after problem identified by the testbench until eventually they have a set of modules that the testbench passes. The company ships the modules to customers and customers find that the modules do indeed work flawlessly. The only problem is that the modules cost twice as much as competitors' products and use five times the energy. The customers go out of business and so there are no follow-on orders.

Grading Note: Many students assumed that the C team could not get the modules working at all. That would be the F team. We should assume that experienced practicing engineers can get things working and won't be tripped up by problems that might have plagued them in their student days.

(*b*) In typical use when running simulation a testbench generates inputs for a module-under-test and the outputs are checked by the testbench to see whether they are correct. After running synthesis we learn how fast the module is. If simulation is computing the module outputs why can't it tell us how fast the module is?

☑ Synthesis can provide timing information and simulation can't because:

Determining what the output of a module is, is not the same thing as determining when that output will arrive (the delay of the output). To determine timing one needs to know the target technology, and that is not provided to the simulator. Further, one needs to optimize a design, and that's also not something a simulator does. A synthesis program reads the Verilog, a target technology (in the form of a design kit) and transforms the original design into one using components from the target technology, and then optimizes the design to meet a timing constraint at minimum cost.

(*c*) A gadget can be built using an ASIC or an FPGA. Describe which is more appropriate for each situation below.

☑ The gadget must be working within a month. ◯ *ASIC* or ⊗ *FPGA*. ☑ Explain.

An ASIC must be manufactured, a time consuming process that can last months.

☑ Per-gadget cost must be under \$1000. Only ten will be made. ◯ *ASIC* or ⊗ *FPGA*. ☑ Explain.

Full-Credit Answer: One can easily buy one FPGA for under \$1000, but the minimum order for an ASIC is thousands of units.

Explanation: The minimum number of ASICs that can be manufactured is one wafer with, which might fit hundreds of chips. To make a wafer one must make shadow masks, which themselves aren't cheap. So it makes no sense to use an ASIC target for only ten chips. In contrast, an FPGA is programmed after it is manufactured. Programming an FPGA is more like writing memory. So even if you just buy ten FPFAs there are others buying the same model of FPGA and sharing the development costs.

☑ Per-gadget cost must be under \$100. Ten thousand will be made. ⊗ *ASIC* or ◯ *FPGA*. ☑ Explain.

The high costs of setting up an ASIC target can be divided by the 10,000 units that will be sold, resulting in a cost that might be lower than an FPGA.

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.

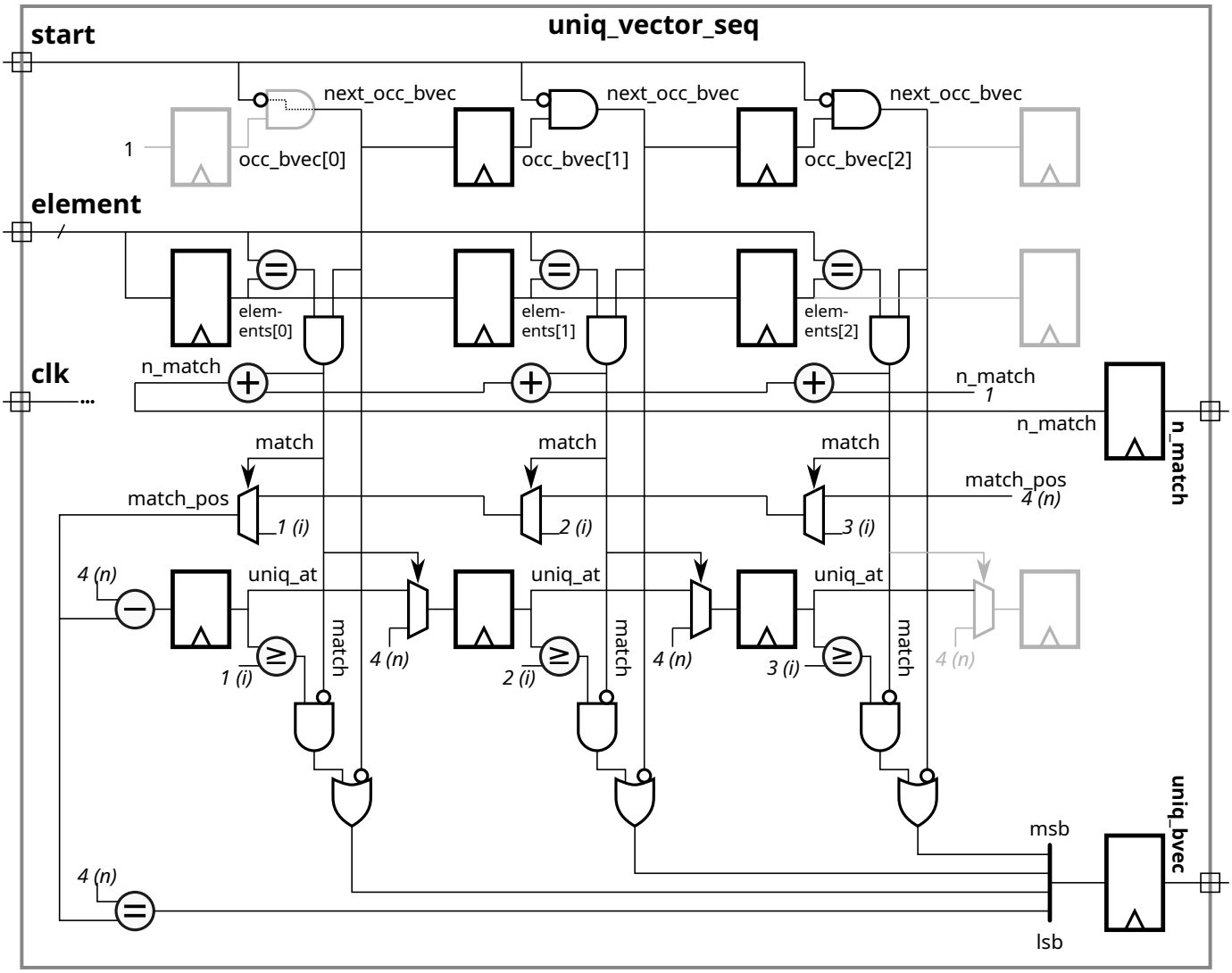
```
module uniq_vector_seq
  #( int we = 10, n = 4, wc = $clog2(n+1) )
   ( output logic [n-1:0] uniq_bvec,     output logic [wc-1:0] n_match,
     input uwire [we-1:0] element,       input uwire start, clk );

   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.

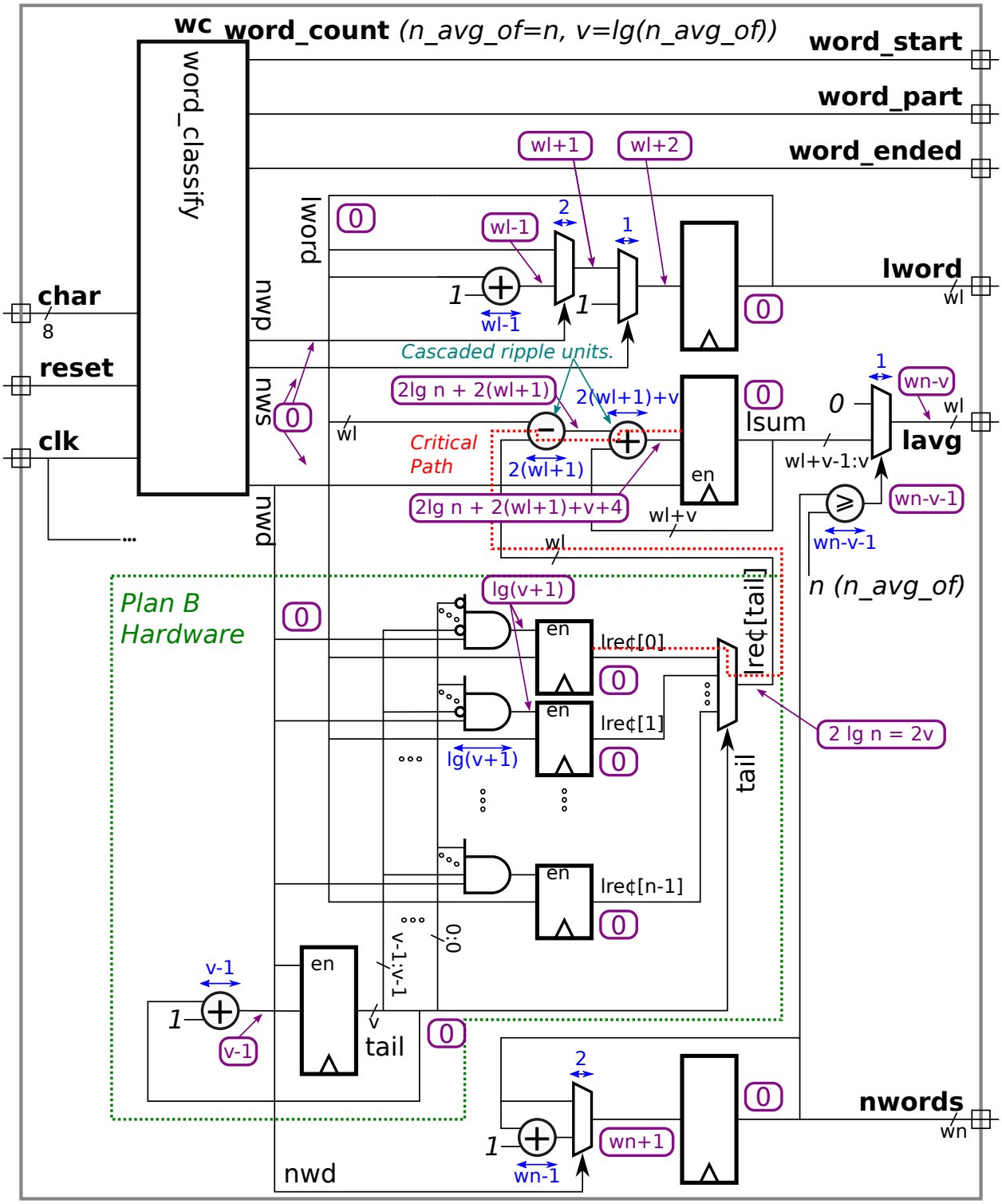
The costs appear in the table below. For the costs of the constant-input adder see 2023 Midterm Exam Problem 2.

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

Staple This Side


wc
word_count (n_avg_of=n, v=lg(n_avg_of))
word_classify
word_start
word_part
word_ended
char
8
reset
clk
...
lword
nwd
nws
nwd
wl+1
wl+2
wl-1
2
1
1
wl-1
0
lword
wl
Cascaded ripple units.
2lg n + 2(wl+1)
2(wl+1)+v
0
lsum
en
wl
Critical Path
2(wl+1)
2lg n + 2(wl+1)+v+4
wl+v
wl
1
wn-v
0
wl
lavg
wl+v-1:v
wn-v-1
wn-v-1
n (n_avg_of)
Plan B Hardware
0
lg(v+1)
en
lreq[0]
lreq[1]
lreq[n-1]
lreq[tail]
0
lg(v+1)
tail
2 lg n = 2v
v-1:v-1
0:0
v-1
1
v-1
en
v
tail
0
2
1
wn-1
wn+1
0
nwords
wn
nwd

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

(*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
```

16 Fall 2022 Solutions

Name Solution

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 19 October 2022, 11:30-12:20 CDT

Problem 1		(25 pts)
Problem 2		(31 pts)
Problem 3		(20 pts)
Problem 4		(12 pts)
Problem 5		(12 pts)
Exam Total		(100 pts)

Alias Sentient?

Good Luck!

Problem 1: [25 pts] Answer the following multiplexor questions.

(*a*) Complete module `mux4` so that it implements a 4-input multiplexor using instantiations of the 2-input multiplexor shown below. Do not use procedural code.

☑ Complete `mux4` so that it implements a 4-input multiplexor ☑ using `mux2` instantiations.

☑ Do not use procedural code. ☑ Do not change the ports or default parameters of `mux4` or `mux2`.

☑ Don't forget to declare any objects that are used.

The solution appears below. The first two muxen, `m01` and `m23`, connect to the data inputs (`a0`-`a3`), two per mux. Note that both of these muxen use `s[0]` as the select bit. The select connection of the third mux, `m0123`, connects to bit `s[1]`.

```
module mux4
  #( int w = 3 )
   ( output uwire [w-1:0] x,
     input uwire [1:0] s,      input uwire [w-1:0] a0, a1, a2, a3 );

   // SOLUTION
   //
   uwire [w-1:0] a01, a23;
   mux2 #(w) m01(   a01, s[0], a0,  a1 );
   mux2 #(w) m23(   a23, s[0], a2,  a3 );
   mux2 #(w) m0123( x,   s[1], a01, a23 );

endmodule

module mux2
  #( int w = 6 )
   ( output uwire [w-1:0] x,
     input uwire s,   input uwire [w-1:0] a0, a1 );
   assign x = s ? a1 : a0;
endmodule
```

(*b*) Module `mux2_bad` only works for `w=1`. Describe the problem and show the correct mux output and the output of `mux2_bad` for `w=4`, `s=0`, `a0=2`, and `a1=4`.

```
module mux2_bad
  #( int w = 4 )
   ( output uwire [w-1:0] x,
     input uwire s,   input uwire [w-1:0] a0, a1 );
   assign x = !s && a0  ||  s && a1;
endmodule
```

☑ In `mux2` (a correct mux) when `w=4`, `s=0`, `a0=2`, and `a1=4`, ☑ output `x=` 2

☑ In `mux2_bad` when `w=4`, `s=0`, `a0=2`, and `a1=4`, ☑ output `x=` 1

☑ Explain the problem when `w` is not `1`.

The problem is that `a0` and `a1` are operands of a logical AND operator, `&&`, and so they will be converted to a Boolean (1-bit) type. That changes both the 2 and 4 in the example to a 1. There would be no problem if `a0` and `a1` were already 1 bit.

(*c*) Complete module `mux2_1r` below so that it recursively implements a 2-input w-bit mux. All that remains to be done is completing the connections to the two recursive instances, `m1` and `mr`.

The solution is shown below. Note that instance `m1` was declared with `w=1` and `mr` was declared with `w=w-1` as part of the problem. So to complete the module instance `m1` connects with one bit of each of `x`, `a0`, and `a1`. Here bit zero was chosen but any bit position would do. Instance `mr` connects to the remaining `w-1` bits of `x`, `a0`, and `a1`. The select signal is the same for both instances.

Note that there is no practical reason to recursively describe a 2-input multiplexor this way, or to recursively describe a 2-input multiplexor at all.

```
module mux2_1r
  #( int w = 5 )
   ( output uwire [w-1:0] x,
     input uwire s,   input uwire [w-1:0] a0, a1 );

   if ( w == 1 ) begin
     assign x = !s && a0  ||  s && a1;
   end else begin


      // SOLUTION
      mux2_1r #(1)   m1( x[0],     s, a0[0],     a1[0] );
      mux2_1r #(w-1) mr( x[w-1:1], s, a0[w-1:1], a1[w-1:1] );


   end

endmodule
```

Problem 2: [31 pts] The `val` output of `atoi_it_m_to_l` is the value of the radix-`r` ASCII-represented number appearing at its input, `str`, and output `nd` is the number of digits. Unlike the Homework 2 Problem 2 module, this module starts at the most-significant digit rather than the least-significant digit.

```
module atoi_it_m_to_l
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output logic [wv-1:0] val,
     output logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   uwire [wv-1:0] vali[n:0];
   uwire is_digit[n:0];
   uwire [wd-1:0] ndi[n:0];
   assign is_digit[n] = 0;
   assign ndi[n] = 0;
   assign vali[n] = 0;
   assign nd = ndi[0];
   assign val = vali[0];

   localparam int wcv = $clog2(r);

   for ( genvar i=n-1; i>=0; i-- ) begin

      // Find Value of Digit i
      uwire [wcv-1:0] vald;
      atoi1 #(r,wcv) a( vald, is_digit[i], str[i] );

      // Multiply (scale) the accumulated sum.
      uwire [wv-1:0] valns;
      mult_by_c #( .w_in(wv), .c(r), .w_out(wv) ) mc( valns, vali[i+1] );

      // Update accumulated value.
      assign vali[i] = is_digit[i] ? valns + vald : 0;
      // Update number of digits.
      assign ndi[i] = !is_digit[i] ? 0 : is_digit[i+1] ? ndi[i+1] : i + 1;

   end

endmodule
```

(*a*) Describe how the behavior of the module would change if the loop direction were changed as shown below, but no other changes were made.

```
   for ( genvar i=0; i<n; i++ ) begin
```

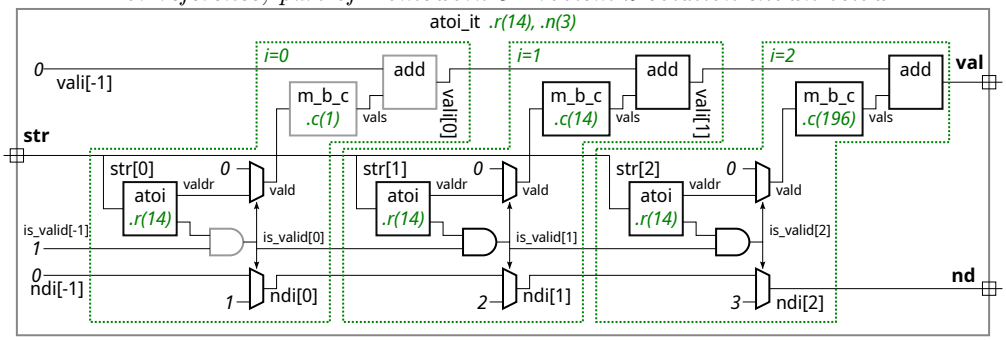
☑ Change in behavior with ascending loop:

There will be no change in behavior. It may be more confusing to a human with the direction of the loop reversed, but the module does exactly the same thing. To see that look at the line assigning `ndi[i]`. It is computed using `ndi[i+1]`. In a procedural language the forward loop would not work because `ndi[i+1]` would not have been computed at iteration `i` when `ndi[i]` is written. But this is Verilog and `assign` is a *continuous assignment* that re-executes whenever its live-in values change, `is_digit[i]`, `is_digit[i+1]`, and `ndi[i+1]` in this case. All the generate loop is doing is describing hardware, each iteration describes one set of hardware. When the hardware for `assign ndi` from iteration `x+1` executes it writes `ndi[x+1]` which results in the `assign`

`ndi` for iteration `x` to execute because `ndi[x+1]` is in the sensitivity list for the assign.

(*b*) On the next (facing) page show the hardware that will be inferred for an instantiation of `atoi_it_m_to_l` (descending loop version) with `n=3` and `r=10`. Show each instantiation of `atoi1` and `mult_by_c` as a box, do not show their contents. The inferred hardware for `atoi_it` is shown for reference.

For reference, part of Homework 3 Problem 2 solution shown below.



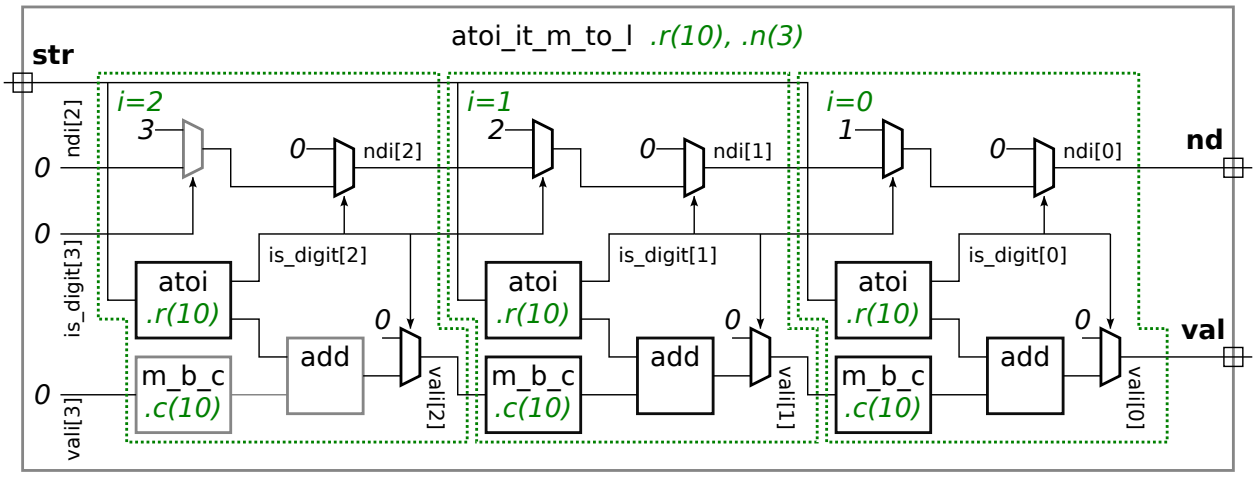
For reference, part of Homework 3 Problem 2 solution shown above.

☑ Show inferred hardware for `atoi_it_m_to_l` for `n=3` and `r=10`.

☑ Show the hardware inferred for the operators, such as `&&` and `?:`.

☑ Do not confuse parameters and ports and omit hardware that does not belong, such as "hardware" to compute values needed at elaboration time.

Solution appears below. Hardware that can easily be eliminated by optimization appears in gray.



(*c*) Module `atoi_m_to_l` will only show the value of numbers that are right-aligned in `str`, otherwise the value will be shown as zero. For example, for input `str="__123"` the output would be `val=123` and `nd=3`, but for input `str="_123_"` the output would be `val=0` (because the rightmost character is not a digit). Modify the module so the `val` output is the value of the number regardless of its location. If there is more than one number, say `str="__12_345_"`, show the value of the rightmost number, 345 in this case.

- ☑ Modify so that `val` and `nd` are for numbers whether or not they are right-aligned.
- ☑ Do not use procedural code.
- ☑ Avoid costly or slow solutions.
- ☑ A correct solution only requires a few changes.

Solution appears in the Verilog on the next page.

In the original code, if `is_digit[i]` was false then the value and length were set to zero. But now since there can be non-digit characters to the right of the number we can't set these to zero. So the first case in the expressions assigning `vali[i]` and `ndi[i]` pass the value and length along when `is_digit[i]` is false.

If both `is_digit[i]` and `is_digit[i+1]` are true then a number is continuing at position `i`. For `vali[i]` we need to add on the scaled number from the left (`valns`) and the current digit, `vald`. If `is_digit[i]` is true but `is_digit[i+1]` is false then `vali` is just the value of the current digit, `vald`. Unlike in the original hardware we can't rely on `valns` being zero for this case.

In the original hardware the value of `i+1` was used for `ndi[i]` at the left-most digit. That won't work here because there could be non-digit characters to the right of the number, so we can't use the position of the first non-digit character to compute the length. Instead, when a number is continuing, both `is_digit[i]` and `is_digit[i+1]` are true, the hardware adds `1` to the previous value of the length (`ndi[i+1]`).

```
module atoi_it_m_to_l
  #( int r = 11, n = 5, wv = $clog2( r**n ), wd = $clog2(n+1) )
   ( output logic [wv-1:0] val,
     output logic [wd-1:0] nd,
     input uwire [7:0] str [n-1:0] );

   uwire [wv-1:0] vali[n:0];
   uwire is_digit[n:0];
   uwire [wd-1:0] ndi[n:0];
   assign is_digit[n] = 0;
   assign ndi[n] = 0;
   assign vali[n] = 0;
   assign nd = ndi[0];
   assign val = vali[0];

   localparam int wcv = $clog2(r);

   for ( genvar i=n-1; i>=0; i-- ) begin

      // Find Value of Digit i
      uwire [wcv-1:0] vald;
      atoi1 #(r,wcv) a( vald, is_digit[i], str[i] );



      // Multiply (scale) the accumulated sum.
      uwire [wv-1:0] valns;
      mult_by_c #( .w_in(wv), .c(r), .w_out(wv) ) mc( valns, vali[i+1] );



      // Update accumulated value.
      // assign vali[i] = is_digit[i] ? valns + vald : 0;
      /// SOLUTION
      assign vali[i] =
              !is_digit[i]   ? vali[i+1] :
               is_digit[i+1] ? valns + vald : vald;



      // Update number of digits.
      // assign ndi[i] = !is_digit[i] ? 0 : is_digit[i+1] ? ndi[i+1] : i + 1;
      /// SOLUTION
      assign ndi[i] =
             !is_digit[i]   ? ndi[i+1] :
              is_digit[i+1] ? ndi[i+1] + 1 : 1;

   end
endmodule
```

Problem 3: [20 pts] Illustrated below is the hardware for one of the `atoi` modules from Homework 3. The delays for the `add`, `atoi1`, and `mult_by_c` modules are shown in blue. For `atoi` the delay of the value (`valdr`) output is zero and the delay of the `is_digit` (lower) output is 3.

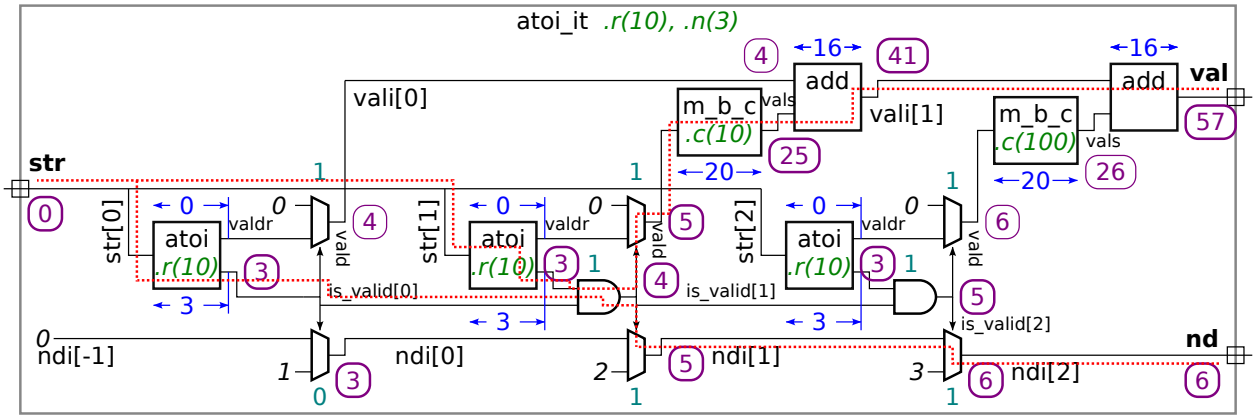
(*a*) Based on the illustrated delays and using the simple model find the delay at each output, `val` and `nd`, and show the critical path to each.

☑ Use the simple model and indicated delays to find the delay at outputs `val` and `nd`.

☑ Show the critical path to both `val` and `nd`.

☑ Take into account constant values.

Solution appears below. Note that the delay of a 2-input mux with one constant input is 1, and the delay with two constant inputs is zero.

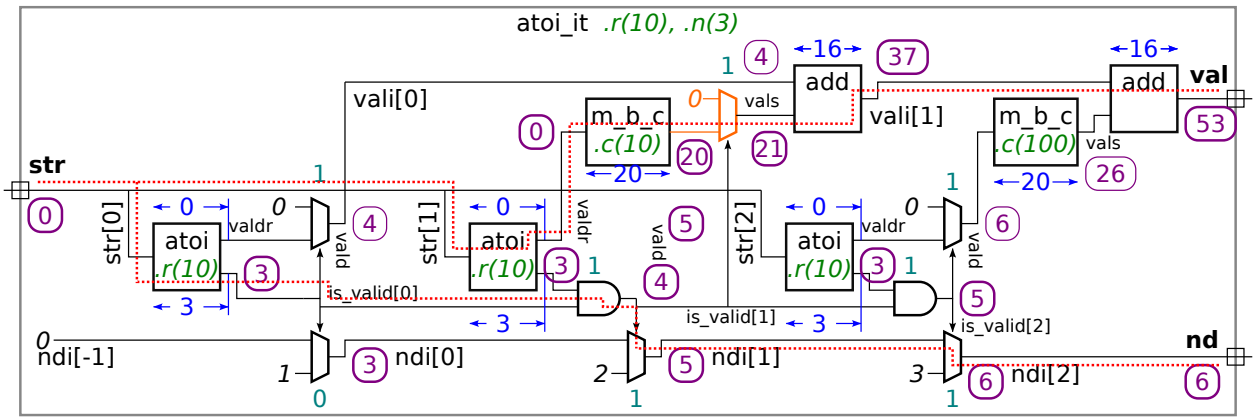


(*b*) Modify the design to reduce the delay at `val` by moving multiplexors. The modification is simple though will increase cost. Show your modification either on the diagram or in the Verilog code below.

☑ Modify to reduce the delay at `val` by moving multiplexors.

☑ Do not change what the module does.

The solution appears below, with the moved mux shown in orange. By moving the mux to the output of the `m_b_c` module it can start at $t = 0$ rather than waiting for the mux select signal to arrive.



Problem 4: [12 pts] Answer each question below.

(*a*) The module below will not compile because of the way the module connections are declared. Fix the problem by changing the declarations.

☑ Change declaration to fix problem.

The solution appears below. Since `x` is assigned proceduraly it must be declared `logic`, which make it a `var` kind rather than a net kind.

```
module yucx2
  #( int w = 5 )
   ( output logic [w-1:0] x,  // SOLUTION: Change port from uwire to logic.
     input uwire [1:0] s,   input uwire [w-1:0] a0, a1 );

   always_comb begin
      x = a0;
      if ( s != 0 ) x = a1;
   end
endmodule
```

(*b*) The `mv` output of `findmax` is supposed to be set to the value of the largest of the three inputs. Assuming it compiles and simulates, it still won't work. Identify the problem.

☑ Why won't `mv` be set to the maximum of `a0`, `a1`, `a2`?

Because `mv` is only initialized once, at the beginning of simulation whereas `a0`, `a1`, and `a2` can change any time.

☑ Provide an example that illustrates the incorrect behavior.

At $t = 10$ the inputs are `a0=4`, `a1=7`, `a2=3`. The output will be `mv=7`. Later at $t = 10$ inputs are `a0=3`, `a1=2`, `a2=0`. The output will still be `mv=7` because there is no way for `mv` to be set to a smaller value.

```
module findmax
  #( int w = 5 )
   ( output logic [w-1:0] mv,
     input uwire [w-1:0] a0, a1, a2 );

   initial mv = 0;
   always_comb if ( mv < a0 ) mv = a0;
   always_comb if ( mv < a1 ) mv = a1;
   always_comb if ( mv < a2 ) mv = a2;

endmodule
```

```
module findmax
  #( int w = 5 )
   ( output logic [w-1:0] mv,  input uwire [w-1:0] a0, a1, a2 );
   always_comb begin           // SOLUTION: Possible fix. (Not the best.)
     mv = 0;                   // mv is initialized whenever the a's change.
     if ( mv < a0 ) mv = a0;
     if ( mv < a1 ) mv = a1;
     if ( mv < a2 ) mv = a2;
   end
endmodule
```

Problem 5: [12 pts] Answer each question below.

(*a*) Type `logic` is an example of a four-state type. Name those four states and describe what the non-numeric ones are used for.

☑ Name the four `logic` states.

They are `0`, `1`, `x`, and `z`.

☑ Describe what the non-numeric ones signify.

State `x` for `var` types can mean uninitialized. For both `var` and `uwire` it can mean an ambiguous results. For net kinds (such as `uwire`) it can mean a bit is driven by more than one driver. State `z` for net types means it is not being driven (in a high impedance state).

(*b*) Most synthesis programs will not synthesize a module that includes a delay, such as the one below. Why not?

```
module madd
  #( int w )
   ( output logic [w-1:0] w,
     input uwire [w-1:0] a, b, c );
   always_comb begin
      w = a * b;
      #5; // Allow enough time for multiplication to finish.
      w = w + a;
   end
endmodule
```

☑ Why isn't a delay synthesizeable?

Though it would be possible for a synthesis program and technology target to provide for delays, it would not be very useful, especially in digital logic design. In the module above the output of the multiplier connects to the input of the adder. A delay has no role to play, since the inferred hardware is just a bunch of connected gates. There is no way to, and no need to, tell the gates that their input values have arrived and so now its time to start working.

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

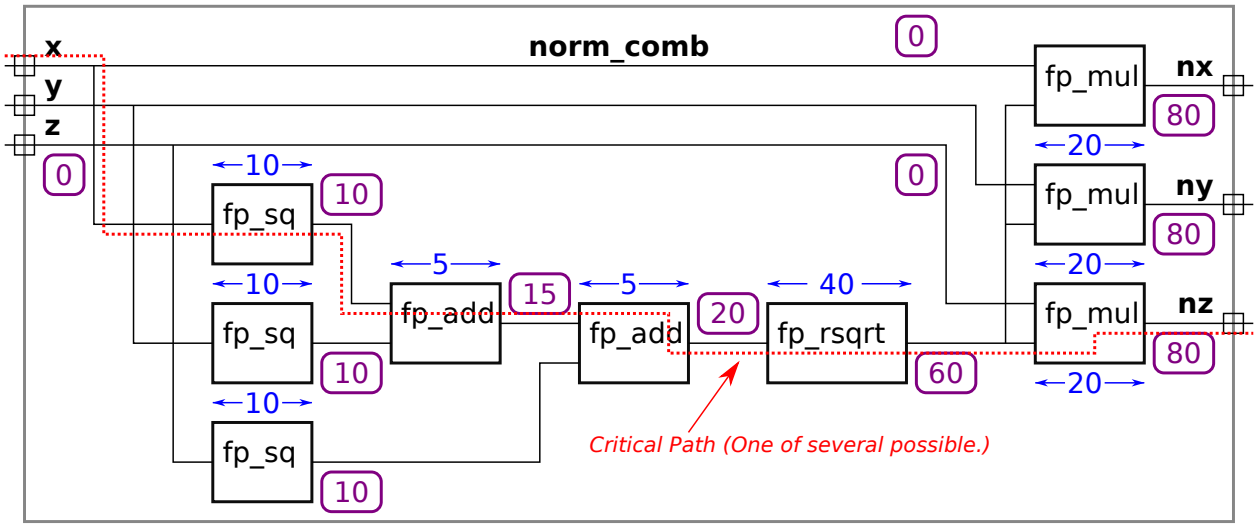
Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

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.

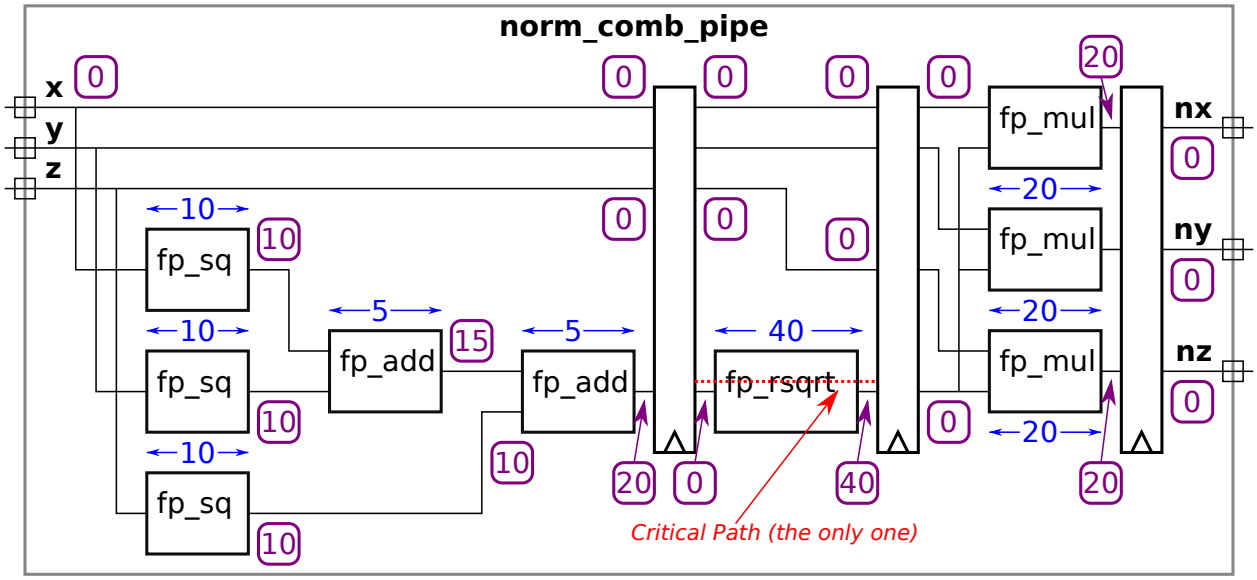
Staple This Side

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 ns, and so the clock period must be set to 40 ns (plus the delay of the register). The path length in the other two stages is 20 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 ns of slack (meaning they arrive 20 ns before the end of the clock cycle, which by coincidence is 20 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 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 ns (the critical path length). Therefore the 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 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 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.

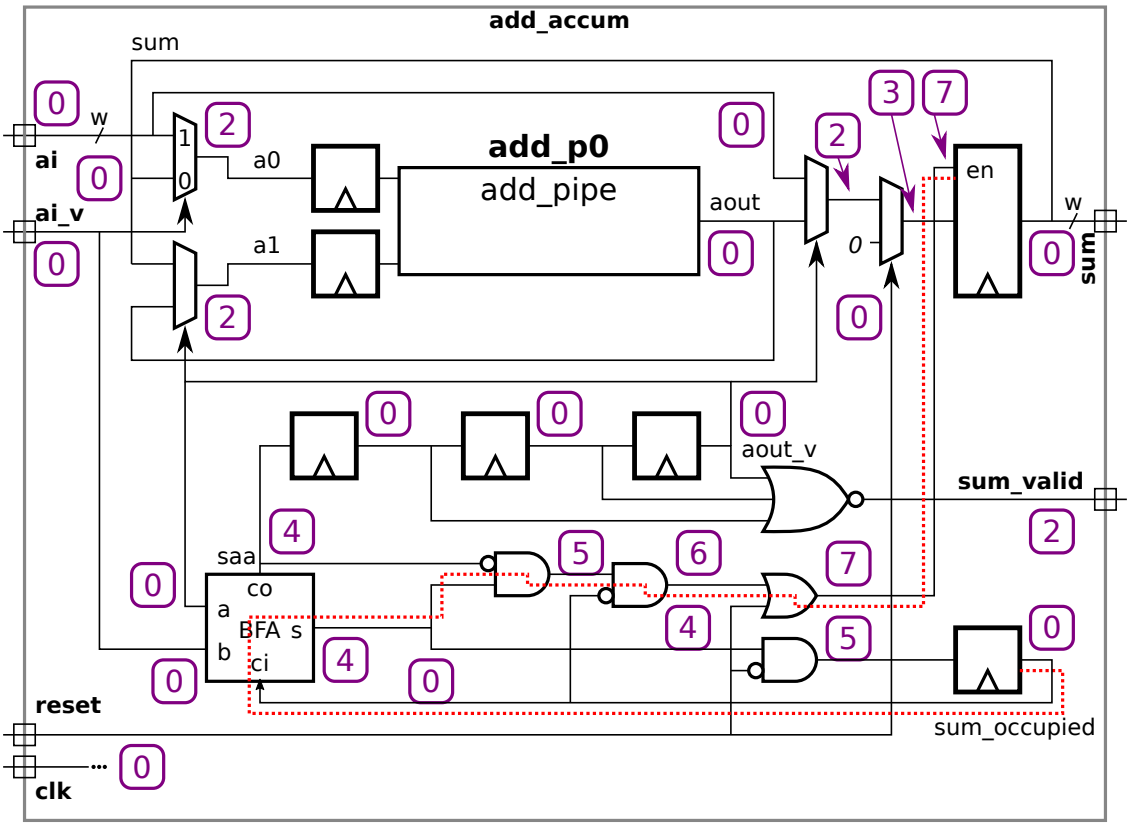
```
// SOLUTION
module norm_comb_n #( int w = 32, int n = 8 )
   ( output uwire [w-1:0] u[n],   input uwire [w-1:0] v[n] );

   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

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 total cost is $[34w + 43]\,\mathrm{u_c}$. 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			$34w + 43$

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`.

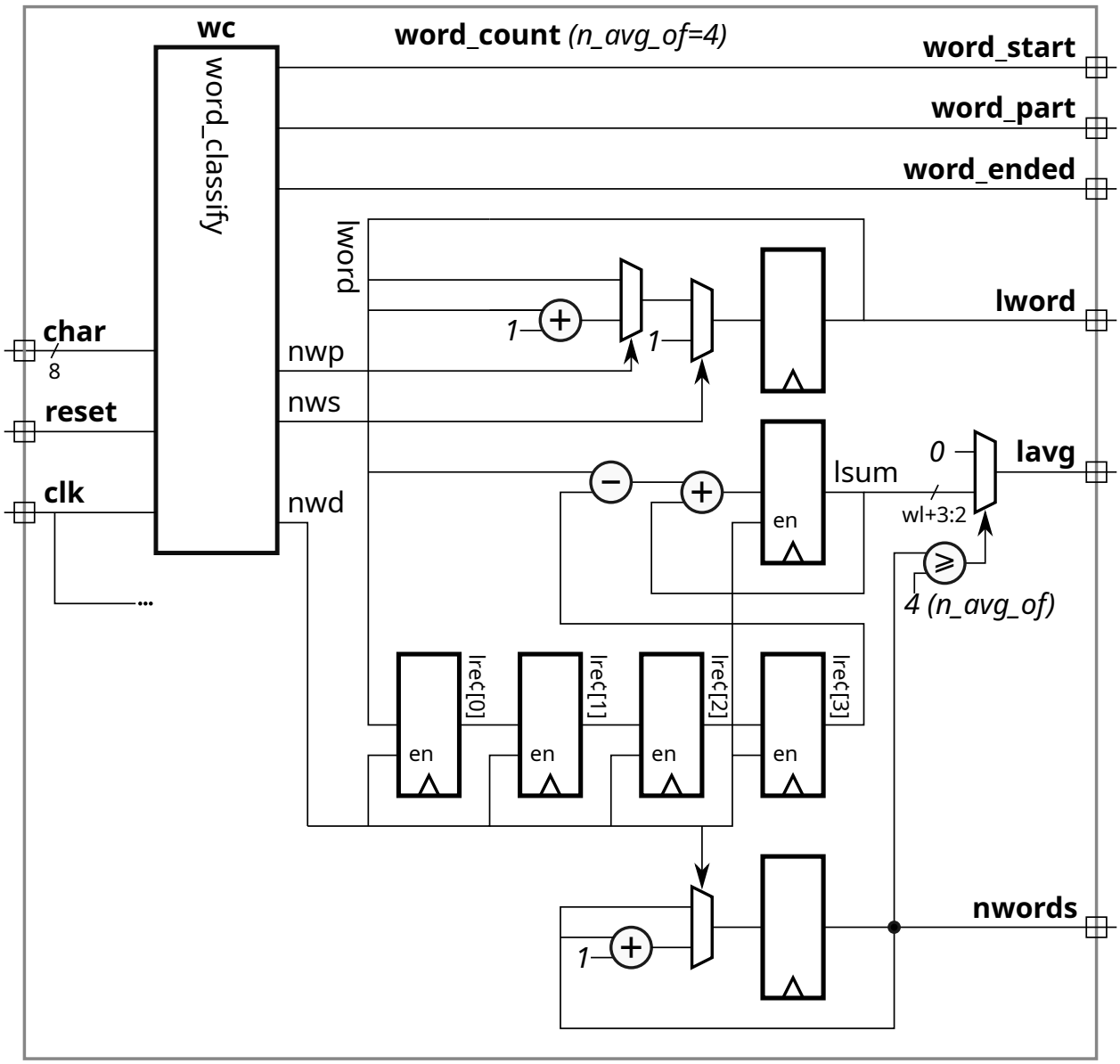
☑ Show `word_classify` as a box, don't attempt to show its contents.

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.

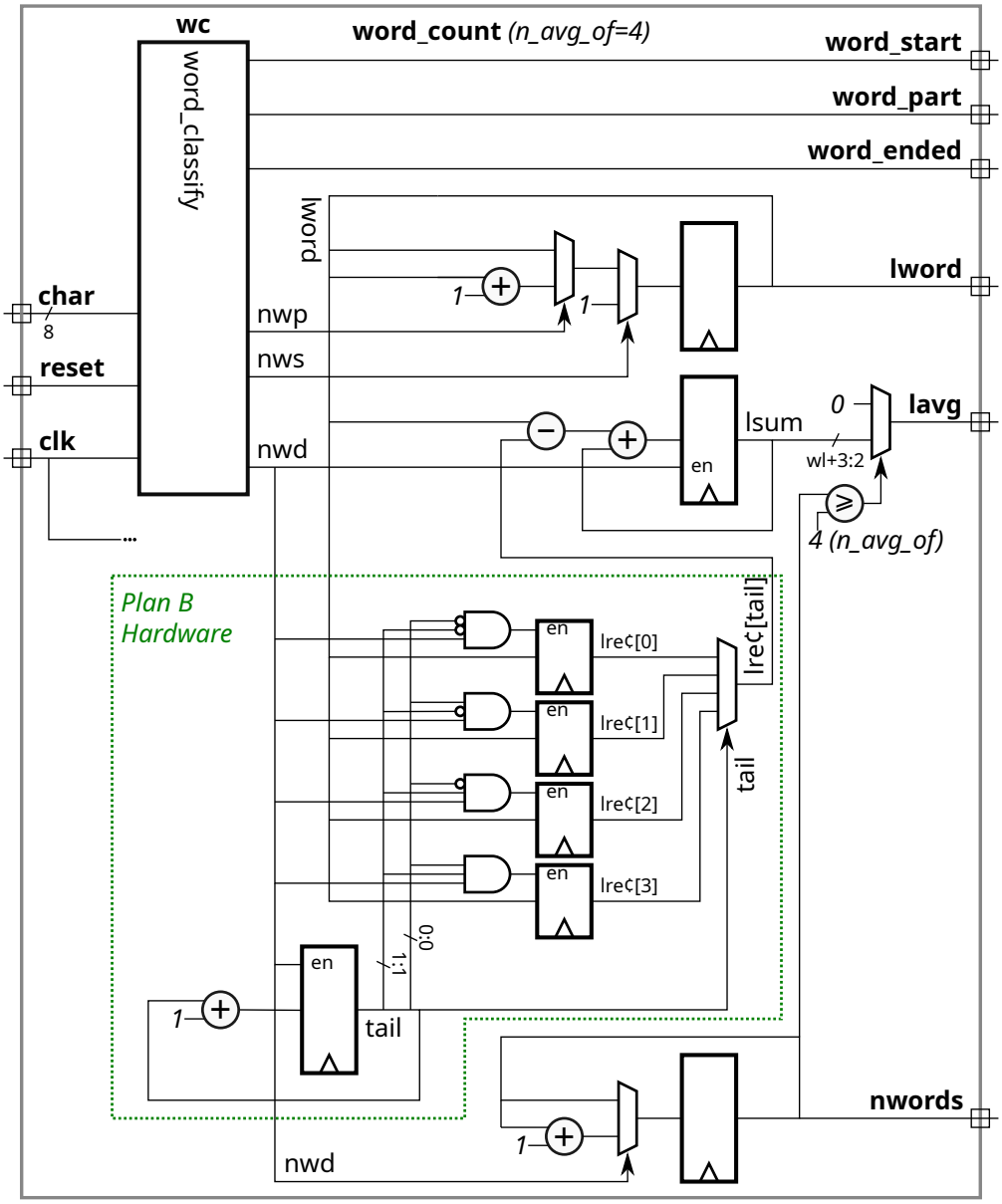
☑ Describe impact on performance of Plan B compared to Plan A.

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.

☑ 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.

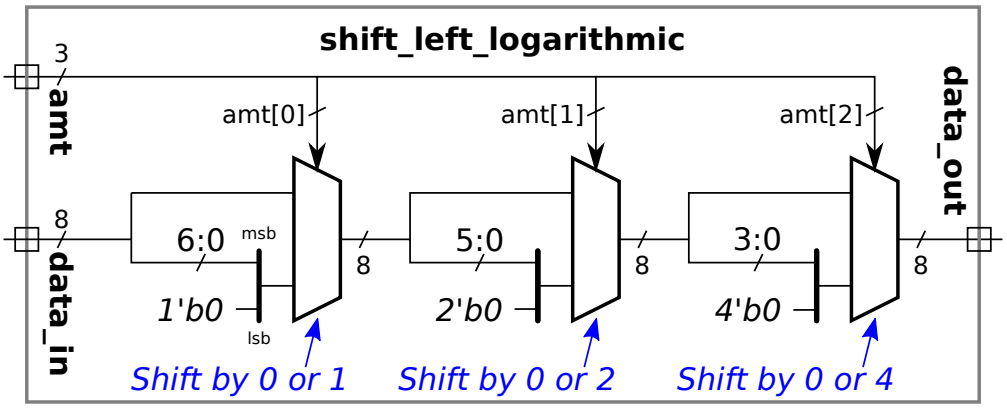


Problem 5: [25 pts] Answer each question below.

(*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

(*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.

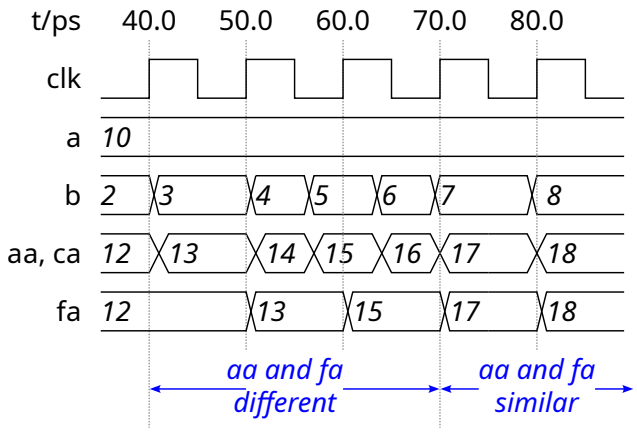
☑ Declare objects `aa`, `ca`, and `fa` so that the code below is correct.

```
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.



17 Fall 2021 Solutions

Name Solution

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 27 October 2021, 11:30-12:20 CDT

Problem 1		(25 pts)
Problem 2		(30 pts)
Problem 3		(10 pts)
Problem 4		(10 pts)
Problem 5		(15 pts)
Problem 6		(10 pts)
Exam Total		(100 pts)

Alias Vwl Shrtg

☣ $V(\text{mRNA}) \quad \Rightarrow \quad R_e < 1$

Good Luck!

Problem 1: [25 pts] Appearing in this problem are two variations on hardware that selects one of four inputs, `i`, based on the position of the least-significant `1` in a 4-bit quantity, `fmt`. This is similar to the hardware needed in the solution to Homework 2, except that here `i[3]` can be selected.

```
module nn_sparse #( int w = 20 )
   ( output logic [w-1:0] o, input uwire [w-1:0] i[4], input uwire [3:0] fmt );
```

(*a*) Show the hardware that will be inferred for `is0` and show that hardware after optimization.

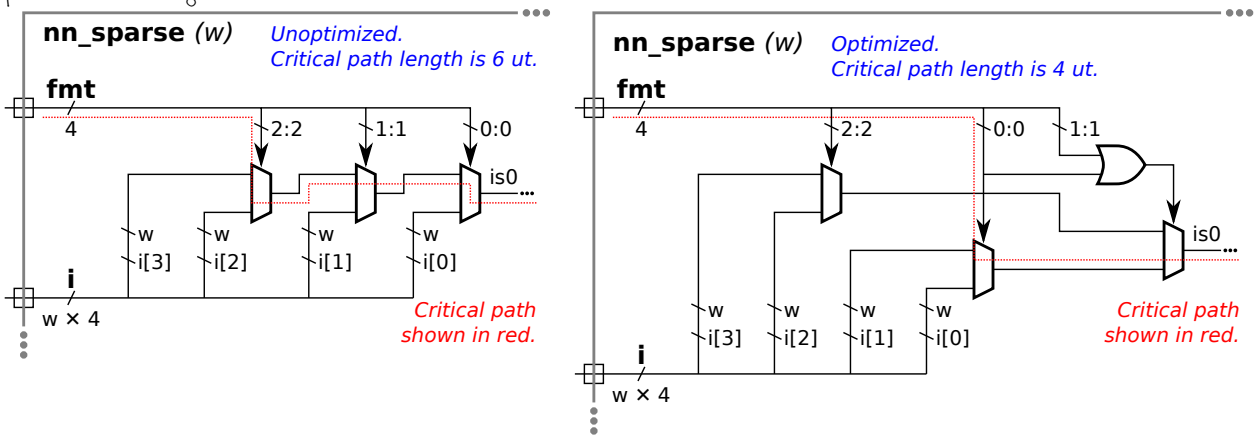
```
   uwire [w-1:0] is0 = fmt[0] ? i[0] : fmt[1] ? i[1] : fmt[2] ? i[2] : i[3];
```

☑ Show inferred hardware.

☑ Show optimized hardware. Hardware can be re-arranged to reduce delay.

☑ Use only basic logic gates and multiplexors.

Solution appears below. The unoptimized hardware follows the rules for inference of the conditional operator (`?:`). In the optimized version the critical path is reduced by two units by rearranging the three multiplexors into a reduction tree and using an OR gate to provide a control signal for the mux at the root.



(*b*) Compute the cost and delay of the optimized hardware for `is0` in terms of w. (That's w, not its default value.)

☑ In terms of w cost is:

Each multiplexor (optimized or not) cost $3w\,\mathrm{u_c}$ and the OR gate cost $1\,\mathrm{u_c}$. The total cost for the unoptimized version is $9w\,\mathrm{u_c}$ and the total cost for the optimized version is $[9w+1]\,\mathrm{u_c}$.

☑ In terms of w delay is:

The delay through a 2-input multiplexor is $2\,\mathrm{u_t}$. In the unoptimized version the critical path passes through three multiplexors, for a delay of $6\,\mathrm{u_t}$. In the optimized version the critical path passes through just 2 muxen, for a delay of $4\,\mathrm{u_t}$.

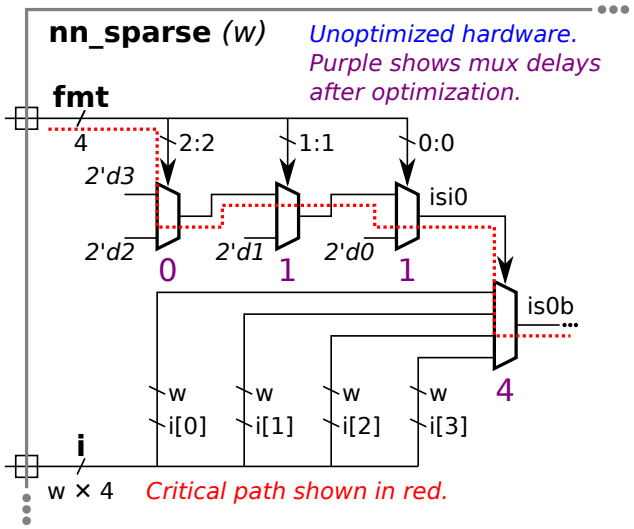
Note that the delay is not a function of w. Be sure that you thoroughly understand why this is true.

(*c*) Appearing below is an alternative design. Net `is0b` will have the same value as `is0`. Show the hardware below before and after optimization. For `isi0` do not show multiplexors after optimization. For `is0b` use two-input multiplexors (as many as needed).

```
uwire [1:0] isi0 = fmt[0] ? 0 : fmt[1] ? 1 : fmt[2] ? 2 : 3;
uwire [w-1:0] is0b = i[isi0];
```

☑ Show inferred hardware.

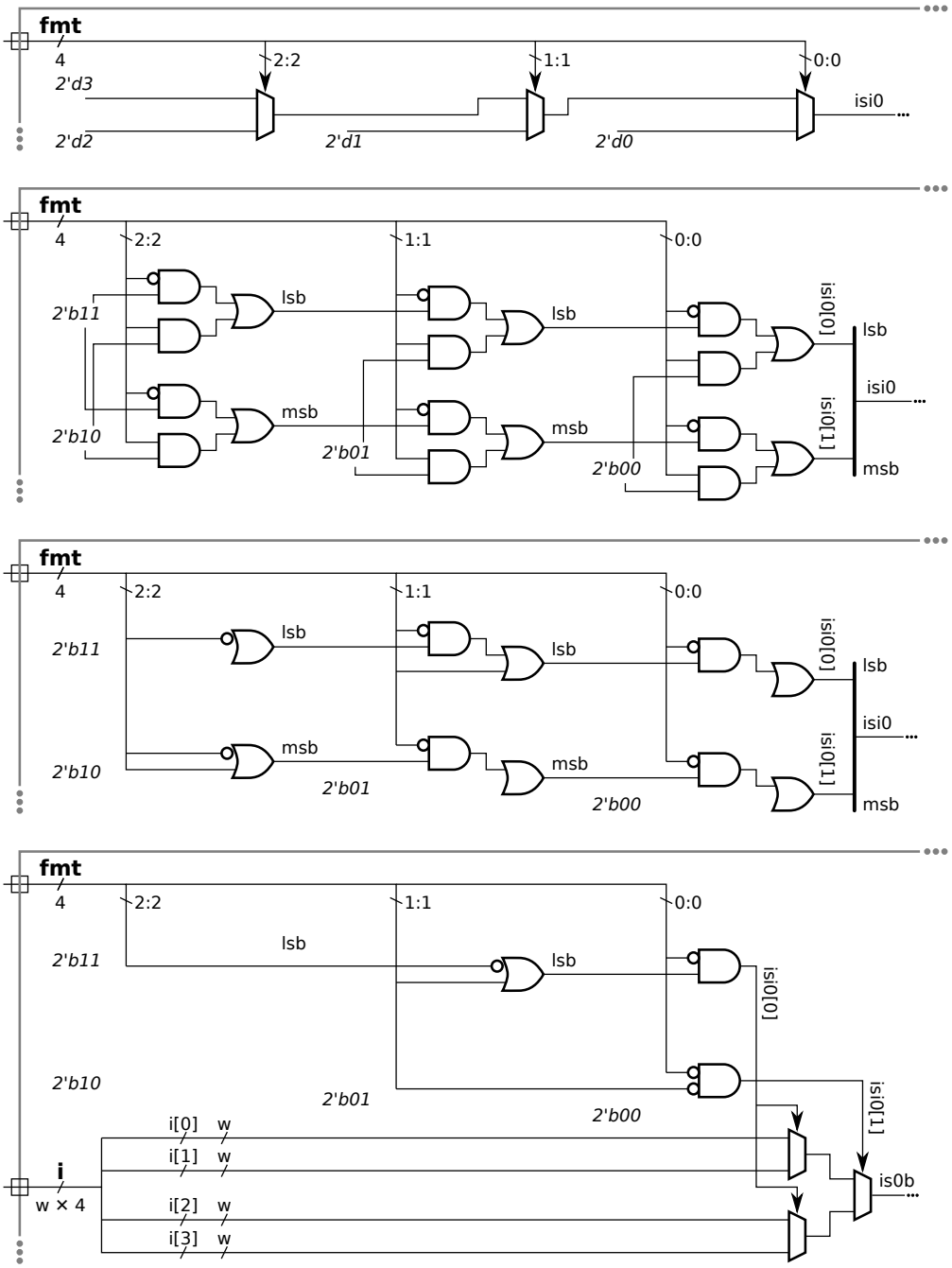
The inferred hardware appears below. The logic computing `isi0` is similar to the logic computing `is0` in the previous part, except that its inputs are constants rather than elements of `i`. The inferred logic for `is0` here is a four-input multiplexor.



☑ Show optimized hardware, optimize to reduce delay.

☑ Use basic logic gates and ☑ no muxen for `isi0` and ☑ two-input muxen (plus other logic) for `is0b`.

The optimized logic computing `isi0` appears below after several steps in the optimization process. At the last step the logic for `is0b` is also shown, but that logic is not fully optimized. The optimization shown below is based on the Verilog code above. A synthesis program that has not been provided with the limit on the values of `fmt` could do no better. (With knowledge that exactly one bit of `fmt` will be `1` a synthesis program (or human) would optimize the two lines above into the logic given for the solution to part (a) of this problem.)



(*d*) Compute the cost and delay of the optimized hardware (from the previous part) in terms of w. (That's w, not its default value.)

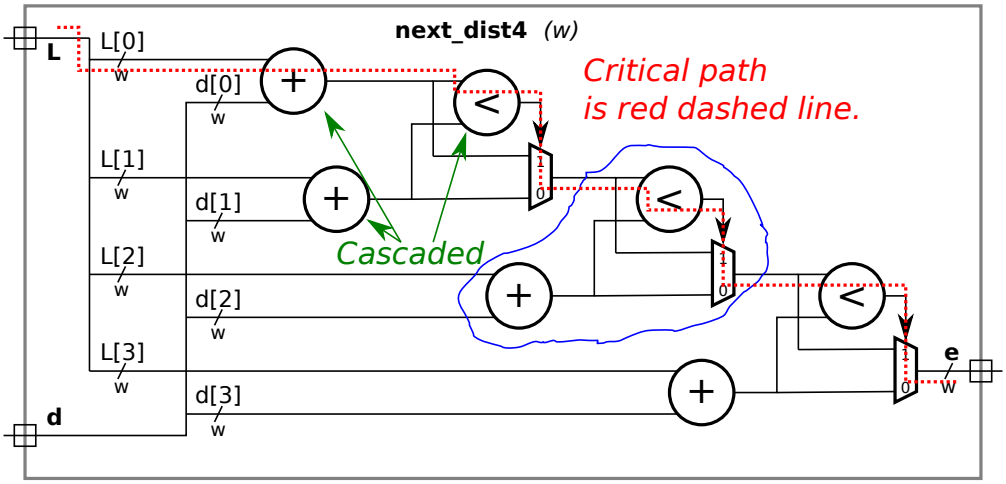
☑ In terms of w cost is:

The cost of the logic in the last section of the illustration above the cost is $[3+3\times 3w]\,\mathrm{u_c}$. The cost would drop to $[1+3\times 3w]\,\mathrm{u_c}$ if the select signals to the first two multiplexors were connected as shown in the optimized solution to part (a).

☑ In terms of w delay is:

The delay of the hardware in the last section of the illustration above is $[1+1+2+2]\,\mathrm{u_t}$, with the critical path passing through the logic generating `isi0[0]`. One cycle can be saved by switching the positions of `isi0[0]` and `isi0[1]` and correspondingly rearrange the order of the `i` inputs to the first two multiplexors to `i[0], i[2], i[1], i[3]`. That would reduce the critical path by 1.

Problem 2: [30 pts] The `next_dist4` hardware illustrated below consists of several duplicated pieces of hardware, one of which is circled. Call the circled hardware an *ami* unit (for add-minimum).



(*a*) Compute the cost and delay of the module using the simple model, and show the critical path on the illustration. Assume that the adder and comparison units are based on ripple adders.

☑ Cost in terms of w:

The module consists for four adders, three comparison units and 3 multiplexors. Each of these devices operate on w bits. Based on the slides describing the simple model, the cost of a w-bit ripple adder is $9w\,\mathrm{u_c}$, the cost of a w-bit comparison unit is $4w\,\mathrm{u_c}$, and the cost of a w-bit 2-input multiplexor is $3w$. The total cost is $\boxed{[4\times 9w + 3\times 4w + 3\times 3w]\,\mathrm{u_c} = 57w\,\mathrm{u_c}}$.

☑ Show critical path. ☑ Delay in terms of w:

☑ Account for any cascading ripple units.

The critical path appears on the illustration as a red dashed line.

The start of the path passes through an adder and a comparison unit. In isolation the delay of an adder is $2(w+1)\,\mathrm{u_t}$ and the delay of a comparison unit is slightly less, $[2w+1]\,\mathrm{u_t}$ according to the simple model slides. But because the output of the adder (actually two adders) connects to the comparison unit the cascaded delay can be used, which is $[4+2(w+1)]\,\mathrm{u_t} = [2w+6]\,\mathrm{u_t}$. Because of the multiplexors cascading delays cannot be used for the other two comparison units. That is because their upper inputs don't arrive until the mux select signal stabilizes. So the remaining delay is that of three 2-input muxen and two w-bit comparison units: $[3\times 2 + 2\times(2w+1)]\,\mathrm{u_t} = [4w+8]\,\mathrm{u_t}$. The total delay is $\boxed{[2w+6+4w+8]\,\mathrm{u_t} = [6w+14]\,\mathrm{u_t}}$.

(*b*) Appearing below are two incomplete modules, one is an `ami` module the other is the `next_dist4` module. Complete these modules to match the diagram using as many `ami` modules as needed. The `ami` module can use procedural or implicit structural code. The `next_dist4` module must instantiate and use `ami` modules but can contain procedural or implicit structural code.

☑ Complete the `ami` module so that it matches the circled hardware.

☑ Complete the `next_dist4` module using as many `ami` modules as needed.

☑ Don't forget to ☑ declare any intermediate objects that are used.

☑ Noting that there are four adders and the width of each wire is `w`, ☑ declare and use parameters appropriately.

```
module ami #( int w = 22 )                    /// SOLUTION
   ( output uwire [w-1:0] s_out,
     input uwire [w-1:0] L, d, s_in );

   // Compute sum ..
   //
   uwire [w-1:0] sum = L + d;
   //
   // .. and connect it to s_out if it's smaller than input value, s_in.
   //
   assign s_out = sum < s_in ? sum : s_in;
endmodule

module next_dist4 #( int w = 12 )             /// SOLUTION
   ( output uwire [w-1:0] e,
     input uwire [w-1:0] L[4], d[4] );

   // Compute first sum. This does not need a comparison, so don't use ami.
   //
   uwire [w-1:0] e0 = L[0] + d[0];

   // Interconnections between ami instances.
   //
   uwire [w-1:0] e1, e2;

   // Instantiate three ami modules and interconnect them properly.
   //
   ami #(w) a1( e1, L[1], d[1], e0 );
   ami #(w) a2( e2, L[2], d[2], e1 );
   ami #(w) a3( e,  L[3], d[3], e2 );
endmodule
```

(*c*) Incomplete module `next_dist` is a generalization of `next_dist4` to `n` elements per input. The module includes a generate loop. Use that loop to instantiate `ami` modules so that it performs the correct calculation. Keep the loop simple, don't try to fix the delay problem.

☑ Complete module, taking advantage of the generate loop.

☑ Be sure to instantiate `ami` modules, ☑ connect the first `ami` correctly, ☑ and don't leave `e` unconnected.

```
module next_dist                                  /// SOLUTION
  #( int n = 20, w = 12 )
   ( output uwire [w-1:0] e,
     input uwire [w-1:0] L[n], d[n] );

   localparam logic [w-1:0] mv = ~w'(0);

   uwire [w-1:0] ee[n-1:-1];
   assign ee[-1] = mv;
   assign e = ee[n-1];

   for ( genvar i=0; i<n; i++ ) begin

      ami #(w) a( ee[i], L[i], d[i], ee[i-1] );

   end

endmodule
```

Problem 3: [10 pts] Consider the `with_assign` module below.

```
module with_assign #( int w = 10 )
                    ( output uwire [w-1:0] g, input uwire [w-1:0] b, c );

   uwire [w-1:0] a, f;
                     //         Sensit.  Execution Time
                     //          List    And Scheduled Lines
   assign g = f | c; // Line 1   f,c     x           x           x
   assign f = a * c; // Line 2   a,c      x ->(L1)      x ->(L1)
   assign a = b + c; // Line 3   b,c       x ->(L2)
                     //                   -----       -----       ---
                     //                   Active      Active      Active
                     //                   List        List        List
endmodule
```

(*a*) Why might the module confuse or annoy humans?

☑ `with_assign` could be confusing because:

The dataflow order (order of dependencies) is from bottom to top but humans expect to read these things from top to bottom. (Line 2 depends on Line 3, Line 1 depends on Line 2.) That would be annoying.

(*b*) The module makes extra work for simulators too. Suppose that the input values to `with_assign`, `b` and `c`, change at $t = 10$. About how many times will each line below execute in a worst-case scenario? *The following sentence was not in the original exam:* Use sensitivity lists to justify your answer.

☑ About how many times does each line execute? ☑ Explain with sensitivity lists.

See the work in the comments above for this discussion. At $t = 10$ because `b` and `c` change Lines 1-3 are all put first in the inactive list, then in the active list for execution. As a result of their execution Line 1 and Line 2 are placed in the inactive list. That becomes the active list when the first one shown is empty. The execution of Lines 2 causes Line 1 to be scheduled a third time. So in total, Line 1 executes 3 times, Line 2 executes twice and Line 3 once.

(*c*) Complete the `sans_assign` routine below so that it does the same thing as `with_assign` but is less confusing and less work for simulators.

☑ Complete routine below. (Yes, it's easy but not trivial.)

```
module sans_assign #( int w = 10 )
                    ( output logic [w-1:0] g, input uwire [w-1:0] b, c );

   logic [w-1:0] a, f; // SOLUTION: Change to logic. Also g.

   always_comb begin

      // SOLUTION: Put lines in dataflow order.
      a = b + c;  // Line 3
      f = a * c;  // Line 2
      g = f | c;  // Line 1

   end
endmodule
```

☑ Why does **sans_assign** make less work for the simulator than **with_assign**? Explain using sensitivity lists.

The sensitivity list of the **always_comb** consists of just **b** and **c**. Objects **a**, **f**, and **g** are not in the sensitivity list (because their values when **begin** is reached are not used). The **always_comb** block will only be scheduled for execution when **b** or **c** changes. So for the $t = 10$ scenario the block—and each line—is executed just once.

Problem 4: [10 pts] Appearing below is an ordinary multiplier, followed by a multiplier that is naïvely designed to take advantage of special cases (first operand is 0 or 1), followed by a module that instantiates both.

```
module mult #( int w = 32 )
             ( output logic [w-1:0] p, input uwire [w-1:0] a, b );
   always_comb p = a * b;
endmodule

module mult_1a #( int w = 32 )
               ( output logic [w-1:0] p, input uwire [w-1:0] a, b );

   always_comb begin
      if ( a == 0 ) p = 0;
      else if ( a == 1 ) p = b;
      else p = a * b;
   end

endmodule

module nm #( int w = 32, logic [w-1:0] c = 12 )
           ( output uwire [w-1:0] prods[4], input uwire [w-1:0] a[4], b[4] );
   mult #(w)      m1 ( prods[0], a[0], b[0] );
   mult #(w)      m2 ( prods[1], c,    b[1] );
   mult_1a #(w)   ma1( prods[2], a[0], b[0] );
   mult_1a #(w)   ma2( prods[3], c,    b[1] );
endmodule
```

☑ Explain why `m1` will be faster (lower delay) than `ma1`, even when possible values of `a[0]` include `0`, `1`, and other values. Assume good synthesis programs.

The critical path of `mult` goes through just a multiplier. The critical path of `mult_1a` goes through a multiplier and a multiplexor, and so the critical path is longer. The fact that the output is available sooner for the two special cases does not change the critical path.

☑ How will the cost and performance of `m2` and `ma2` compare (to each other) using good synthesis programs? That is, ☑ which should be chosen when delay is the only concern and, ☑ which of the two should be chosen when cost is the only concern. The answer should not depend on any particular value of `c`.

In `m2` and `ma2` the `a` input is a constant. The synthesis program will then be able to determine, for `mult_1a`, which part of the `if/else` chain executes and synthesize only for that. If `a=3` then it will be the `a*b` part, and so the two modules are identical. In both `m2` and `ma2` the synthesis program can see that the `a` input is a constant and will optimize the multiplier appropriately. That means if `a=1` `ma2` will have no advantage.

Problem 5: [15 pts] Answer the following questions about Verilog syntax and semantics.

(*a*) Appearing below are four variations on a multiplier with a constant input. Most have errors that would prevent them from compiling. For each indicate whether there is an error, and if so, what the error is and a minimal fix.

☑ Module is ◯ *correct* or ☑ *has the following error and fix:*

The assignment statements, such as `p=0;`, in the module below are an error in a module context.

```
module mult_2a #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   if ( a == 0 )          p = 0;
   else if ( a == 1 )     p = b;
   else                   p = a * b;

endmodule
```

One solution is to switch to a continuous assignment (an `assign` statement), that has been done below.

```
module mult_2a #( int w = 32, logic [w-1:0] a = 12 )  /// SOLUTION
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   if ( a == 0 )          assign p = 0;      // SOLUTION: Use assign.
   else if ( a == 1 )     assign p = b;
   else                   assign p = a * b;

endmodule
```

☑ Module is ◯ *correct* or ☑ *has the following error and fix:*

A procedural assign is being used on a net kind (`uwire` in this case). So, unlike `mult_2a`, the kind of assignment statement is correct here since the assignment occurs in procedural code. The problem is kind of object being assigned.

```
module mult_2b #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   always_comb begin
      if ( a == 0 )      p = 0;
      else if ( a == 1 ) p = b;
      else               p = a * b;
   end

endmodule
```

A simple fix is to change `p` to a var. Note that `uwire` is short for `uwire logic` and that `logic` is short for `var logic`.

```
module mult_2b #( int w = 32, logic [w-1:0] a = 12 )
   ( output logic [w-1:0] p, input uwire [w-1:0] b );
   // SOLUTION: Change p from "uwire logic" to "var logic".

   always_comb begin
      if ( a == 0 )      p = 0;
      else if ( a == 1 ) p = b;
      else               p = a * b;
   end

endmodule
```

☑ Module is ◯ *correct* or ☑ *has the following error and fix:*

The `if` in the code below is a generate `if`, and its condition, `b==0`, is not an elaboration-time constant. (The expression `b==0` is not an elaboration-time constant because `b` is a module input.)

```
module mult_2c #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   if ( b == 0 )          p = 0;
   else if ( b == 1 )     p = a;
   else                   p = a * b;

endmodule
```

A fix is to make the code procedural by wrapping it in an always block and making `p` a var.

Note: Changing `b` to `a` in the `if` condition is NOT an appropriate fix because it changes what the module does.

```
module mult_2c #( int w = 32, logic [w-1:0] a = 12 )
   ( output logic [w-1:0] p, input uwire [w-1:0] b );
   always_comb   /// SOLUTION: Change generate if to procedural if.
    if ( b == 0 )          p = 0;
    else if ( b == 1 )     p = a;
    else                   p = a * b;
endmodule
```

☑ Module is ☑ *correct* or ◯ *has the following error and fix:*

```
module mult_2d #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   if ( a == 0 )          assign p = 0;
   else if ( a == 1 )     assign p = b;
   else                   assign p = a * b;

endmodule
```

(*b*) Show the values of `b` and `c` where requested below.

The solution appears below. The difference between `a`, `b`, and `c` are in how the bits are numbered. That only impacts the use of the indexing (bit select) operator, `[i]`. It does not affect assignments and other references to the objects. For that reason `b` and `c` on the first assignment are the same as `a`. However, the second assignment of `b` refers to the bits, so they are reversed.

```
module assortment;
   logic [15:0] a;
   logic [0:15] b;
   logic [16:1] c;

   initial begin

      a = 16'h1234;
      b = a;
      c = a;
      //  ☑   Show value of b and c after line above executes:

      // SOLUTION:
      //  b = 16'h1234
      //  c = 16'h1234


      #1; // Not really needed.
      for ( int i=0; i<16; i++ ) b[i] = a[i];
      //  ☑   Show value of b after line above executes:

      // SOLUTION
      //  b = 16'h2c48
      //    = 16'b_0010_1100_0100_1000
      //
      //  Note that:
      //  a = 16'b_0001_0010_0011_0100

   end
endmodule
```

Problem 6: [10 pts] Answer the following synthesis questions.

(*a*) Cadence Genus defines the following three synthesis steps: `syn_gen` (generic), `syn_map` (mapped, or technology mapping), and `syn_opt` (optimized). Answer the following questions about technology mapping.

☑ Explain what happens during technology mapping.

In technology mapping generic gates (say, a 3-input AND gate) are replaced by gates in the target technology. The replacement can also happen at a higher level, so a generic adder module would be replaced by an adder in the target technology, if such a thing is provided.

☑ Even if optimization were done before technology mapping why is it important optimize after technology mapping?

The optimization would be on generic gates, which might be available at any size. The target technology might have gates with, say, 2, 4, or 6 inputs, but not 3 or 5 inputs. So another round of optimization might find a way to use those wasted inputs. Also, after technology mapping the delay of gates are known, and so delay optimization can occur.

(*b*) What is the big disadvantage of setting the delay target too low when performing synthesis? (The small disadvantage is that it takes a longer time to run.)

☑ Disadvantage of setting delay target too low during synthesis:

With a very large delay target the optimization program can minimize cost. As the delay is lowered the optimization will have to substitute higher-cost alternatives to meet the delay target. (For example, substituting a carry lookahead adder for a ripple adder.) Making the delay smaller than it needs to be can result in costs higher than they need to be.

Name Solution

Formatted For Two-Sided Printing

Staple This Side

Digital Design using HDLs
LSU EE 4755
Final Examination

Wednesday, 8 December 2021 7:30 CST

Problem 1		(30 pts)
Problem 2		(35 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Exam Total		(100 pts)

Alias Good Luck JWST!

Good Luck!

Staple This Side

Problem 1: [30 pts] For the modules in this problem input `sample` holds a new value each cycle, and output `r_avg` holds the average of the last `n_samples` inputs. (Ignore the fact that the module needs but lacks a reset.)

(*a*) For the module below show the hardware that will be inferred when instantiated with default parameters. Be sure to optimize for the default value of `n_samples`.

```
module ravg2 #( int w = 8, n_samples = 4 )
   ( output logic [w-1:0] r_avg,
     input uwire [w-1:0] sample,   input uwire clk );

   logic [w-1:0] samples[n_samples];

   parameter int wm = $clog2( n_samples );
   parameter int ws = w + wm;
   logic [ws-1:0] tot;

   always_ff @( posedge clk ) begin

      samples[0] <= sample;

      for ( int i=1; i<n_samples; i++ ) samples[i] <= samples[i-1];

      tot <= tot - samples[n_samples-1] + samples[0];

   end

   always_comb r_avg = tot / n_samples;

endmodule
```

Solution on next page.

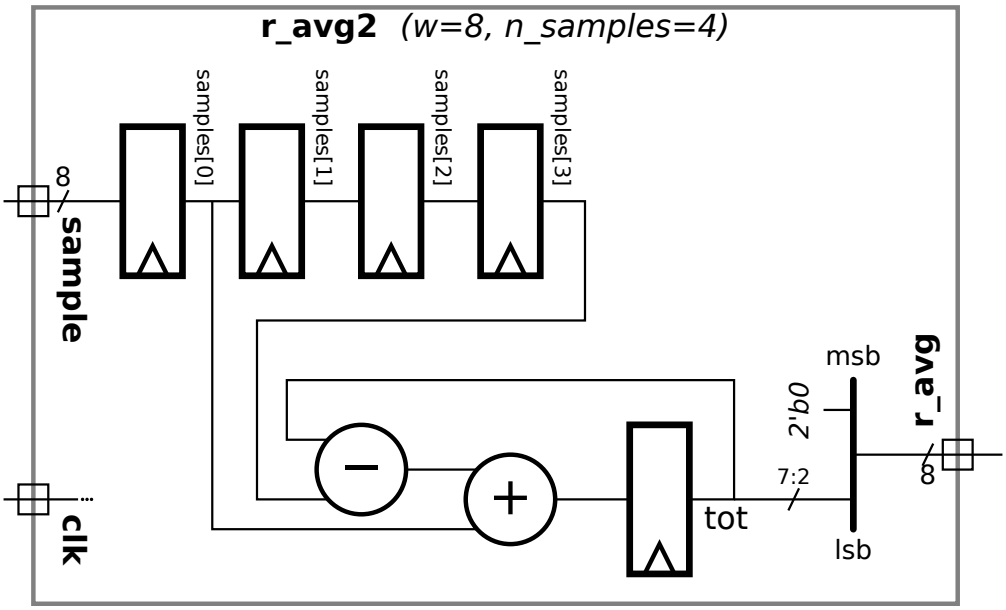
Staple This Side

☑ Show hardware for the module above using default parameter values.

☑ Optimize for these parameter values.

Solution shown below. Notice that because non-blocking assignments were used to assign `samples[i]`, the computation of `tot` uses the register outputs. In particular `samples[0]` is the register output, which is the value of `sample` from the previous cycle.

Because `n_samples=4` is a power of 2, the division, `tot/n_samples`, can be done by shifting right by two bits. Since the shift is constant just use bits `7:2` of `tot` and place two bits of zero in the MSB of the output.



(*b*) The module to the right is similar to `ravg2` except that it has three arithmetic unit instantiations: an adder, a subtractor, and a divide-by-constant unit. Modify `ravg3` so that it uses these modules. For full credit connect them so that the critical path passes through at most one module per cycle. In a correct solution `r_avg` will arrive at the output of `ravg3` later than it would in module `ravg2`.

☑ Modify `ravg3` so that it uses the three arithmetic units.

☑ For full credit, the critical path can go through at most one arithmetic unit per cycle.

☑ The connections to the arithmetic units can be changed (say from `aa1` to something else).

☑ Do not add unnecessary cost or delay.

Solution appears below.

Please be sure to understand the following important points.

So that the critical path passes through at most one arithmetic module, the inputs to the arithmetic modules cannot connect to arithmetic module outputs. Instead, they connect to registers, such as `tot` and `samples[0]`.

So that the running sum is correct, the values of `samples[0]` and `samples[n_samples-1]` must be used in the same cycle. For that reason the subtractor is used to compute `samples[0] - samples[n_samples-1]`. It would not be correct to compute `diff = tot - samples[n_samples-1]` in one cycle and `tot = diff-samples[0]` in the next cycle because `samples[0]` is the wrong value.

Notice that `samples[0]` was directly connected to the subtractor input. That's more convenient than using an intermediate variable, say `sa1`.

```
module ravg3 #( int w = 8, n_samples = 4 )
   ( output logic [w-1:0] r_avg,
     input uwire [w-1:0] sample,
     input uwire clk );

   logic [w-1:0] samples[n_samples];

   parameter int wm = $clog2( n_samples );
   parameter int ws = w + wm;
   logic [ws-1:0] tot;

   // SOLUTION - Declare a register to hold output of subtractor.
   logic [ws-1:0] pl_diff;

   always_ff @( posedge clk ) begin

      samples[0] <= sample;

      for ( int i=1; i<n_samples; i++ ) samples[i] <= samples[i-1];

      // tot <= tot - samples[n_samples-1] + samples[0]; // Modify or eliminate this line.

      // SOLUTION - Write output of subtractor and adder into registers.
      pl_diff <= diff;
      tot <= sum;

   end

   // always_comb r_avg = tot / n_samples;               // Modify or eliminate this line.

   // SOLUTION - Remove unneeded declarations. (aa1, etc.)
   uwire [ws-1:0] sum, diff;

   // SOLUTION - Use subtract to compute samples[0] - samples[n_samples-1]
   our_sub #(ws,w) sub2( diff,   samples[0],  samples[n_samples-1]   );

   // SOLUTION - Use adder to compute new value of tot.
   our_adder #(ws,ws) adder1( sum, tot, pl_diff );

   // SOLUTION - Use divider to compute r_avg.
   our_div_by #(w,ws,n_samples) div3( r_avg,     tot );

endmodule
```

Problem 2: [35 pts] Appearing below is a Verilog description of a lower-cost version of the `bit_keeper` module from Homework 4 and a diagram of the hardware.

```
typedef enum { Cmd_Reset=0, Cmd_Rot_To=1, Cmd_Write=2, Cmd_Nop=3, Cmd_SIZE } Command;
module rot_left #( int w = 10, amt = 1 )
   ( output uwire [w-1:0] r, input uwire [w-1:0] a);
   assign r = { a[w-amt-1:0], a[w-1:w-amt] };
endmodule
module bit_keeper_lite #( int wb = 64, wi = 8, ws = $clog2(wb) )
   ( output logic [wb-1:0] bits,    output uwire ready,
     input uwire [1:0] cmd,         input uwire [wi-1:0] din,
     input uwire [ws-1:0] pos,      input uwire clk );
   localparam int ramt_a = 1;     // Specify Rotation Amounts
   localparam int ramt_b = 1 << ( ws >> 1 );
   uwire [wb-1:0] ra, rb;
   rot_left #(wb,ramt_a) rl1(ra,bits);
   rot_left #(wb,ramt_b) rl8(rb,bits);
   logic [ws-1:0] rot_to_do;      // Remaining amount of rotation to do.

   assign ready = rot_to_do == 0;
   always_ff @( posedge clk ) case ( cmd )
        Cmd_Reset:  begin bits = 0; rot_to_do = 0; end
        Cmd_Rot_To: rot_to_do = pos; // Initialize rotation. Rotate during Nop.
        Cmd_Write:  bits[wi-1:0] = din;
        Cmd_Nop:                     // Continue Executing a Cmd_Rot_To
           if ( rot_to_do >= ramt_b ) begin
              bits = rb;             // Use output of larger rot module.
              rot_to_do -= ramt_b;   // Decrement remaining rot amt.
           end else if ( rot_to_do >= ramt_a ) begin
              bits = ra;             // Use output of smaller rot module.
              rot_to_do -= ramt_a;   // Decrement remaining rot amt.
           end
      endcase
endmodule
```

(*a*) Find the cost and delay of the illustrated hardware using the simple model. Take into account the presence of constants. For the addition and comparison units assume a ripple implementation. Show any assumptions made. (See the next part before solving this one.)

☑ Show cost in terms of w_b, w_i, and w_s. ☑ Take into account constants.

The hardware consists of registers, multiplexors, adders, comparison units, and constant shifters.

Shifters: Since they shift by a constant amount the total shifter cost is zero.

Registers: The cost of a w-bit register is $7w\,\mathrm{u_c}$. There are two registers, `bits` and `rot_to_do`. There sizes are w_b and w_s, so their combined cost is $7(w_b + w_s)\,\mathrm{u_c}$.

Two-Input Multiplexors: The cost of a w-bit, 2-input mux is $3w\,\mathrm{u_c}$. In the illustrated hardware there are two w_b-bit 2-input muxen and two w_s-bit 2-input muxen. (None of their inputs are constant.) Their total cost is $[2 \times 3\,w_b + 2 \times 3\,w_s]\,\mathrm{u_c} = 6(w_b + w_s)\,\mathrm{u_c}$.

Four-Input Multiplexors: A w-bit four-input mux can be constructed from three 2-input muxen, and so its cost would is $3 \times 3\,w\,\mathrm{u_c} = 9w\,\mathrm{u_c}$. The cost of a w-bit, 2-input mux with a constant data input is $w\,\mathrm{u_c}$. Each of the four-input muxen has a

Staple This Side

constant data input, reducing the cost to $(2 \times 3 + 1)w\ \mathrm{u_c} = 7w\ \mathrm{u_c}$. The total cost of the two four-input muxen accounting for the constant input is $\boxed{7(w_b + w_s)\ \mathrm{u_c}}$.

Adders: An ordinary w-bit ripple adder costs $9w\ \mathrm{u_c}$. A w-bit ripple adder with one constant input costs $4w\ \mathrm{u_c}$. The two adders each have one constant input. Based on just that their costs are $4 \times 2w_s\ \mathrm{u_c}$. But the value of `ramt_b` is $2^{w_s/2}$, and so the $w_s/2$ least-significant bits of `ramt_b` are zero. That means the adder passes those low bits through unchanged, reducing the adder cost to just $w_s/2\ \mathrm{u_c}$. Looking at the `ramt_a` adder in isolation one would have to conclude that its cost is $4w_s\ \mathrm{u_c}$ with `ramt_a=1`. But the output of the adder is ignored if `rot_to_do>ramt_b` meaning that we can assume the input to the `ramt_a` adder is no greater than `ramt_b` and so we only need a $w_s/2$-bit adder. With both of those optimizations the total $\boxed{\text{adder cost is } 2 \times 4\tfrac{w_s}{2}\ \mathrm{u_c} = 4w_s\ \mathrm{u_c}}$.

Comparison Unit: Recall that a ripple comparison unit is constructed from the carry logic of ripple subtractor. The cost of a w-bit comparison is $4w\ \mathrm{u_c}$. But one constant input reduces the cost to just $w\ \mathrm{u_c}$. With no further optimizations the cost of the two comparison units is $2w_s\ \mathrm{u_c}$.

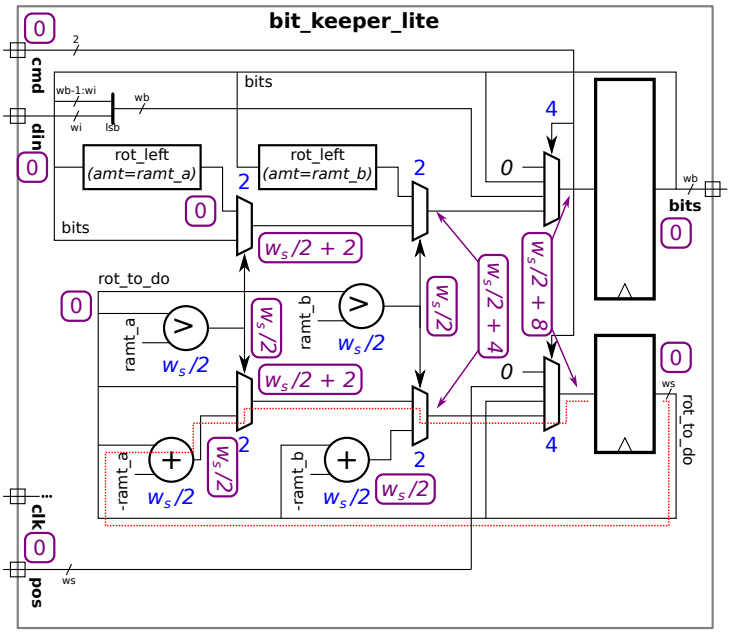
The `ramt_a` comparison is irrelevant if `rot_to_to` is greater than `ramt_b`, and so only $w_s/2$ bits need be examined. If the `ramb_b` comparison operation were $\geq$ then it could just examine $w_s/2$ bits. But since the operation is strictly greater than all bits must be considered. But using the output of the `ramt_a` comparison the `>ramt_b` comparison could be done by examining $w_s/2$ more bits. The total comparison $\boxed{\text{cost is } 2 \times \tfrac{w_s}{2}\ \mathrm{u_c} = w_s\ \mathrm{u_c}}$.

☑ Show delays and arrival times on the diagram, and ☑ highlight the critical path. These should be in terms of w_b, w_i, and w_s.

The timings and critical path are shown on the diagram. Blue shows the delay through a component, such as 2 for two-input multiplexors. Circled times show the delay of the longest path starting at module inputs and register outputs. A critical path is shown as a red dotted line. Note that there are several critical paths in this circuit though only one is illustrated.

Multiplexor Delay: The delay of an ordinary two-input mux is $2\ \mathrm{u_t}$. If one input is constant the delay is $1\ \mathrm{u_t}$. The delay of an n-input mux is $\lceil \lg n \rceil 2\ \mathrm{u_t}$, which works out to $4\ \mathrm{u_t}$ for a four-input mux. The next sub-problem shows how that delay can effectively be reduced to $2\ \mathrm{u_t}$ on the critical path. The diagram to the right does not reflect that optimization.

Adder Delay: The delay of a w-bit ripple adder with a constant input is $w\ \mathrm{u_t}$. The timings in the diagram are based on $w_s/2$-bit adders.



Comparison Delay: The delay of a w-bit ripple comparison unit with a constant input is $w\ \mathrm{u_t}$. The timings in the diagram are based on $w_s/2$-bit comparison units.

(*b*) In class we assume that a four-input mux is implemented using a reduction tree of 3 two-input muxen. For the illustrated hardware that would result in a longer critical path than is necessary. Modify the diagram on the right to show a better way of implementing the four-input multiplexors.

☑ Replace four-input multiplexors with two-input muxen connected to reduce critical path.

Solution appears on the lower half of the next page. The four-input mux has been replaced by three two-input muxen, but not connected in a reduction tree. The benefit of this non-tree connection is that one of the inputs, the fourth as used here, has a delay of only $2\,\mathrm{u_t}$. That is the input that carries the critical path, and so the critical path delay is reduced by $2\,\mathrm{u_t}$.

(*c*) Notice that care was taken to ensure that `ramt_b` is a power of 2. Explain how the fact that `ramt_b` is a power of two reduces the cost of the adder and comparison unit operating on `ramb_b`. Also explain how a power-of-2 `ramb_b` can reduce the cost of the other adder and comparison unit, if the synthesis program is clever enough. *Hint: Consider the binary representation of* `rot_to_do`.

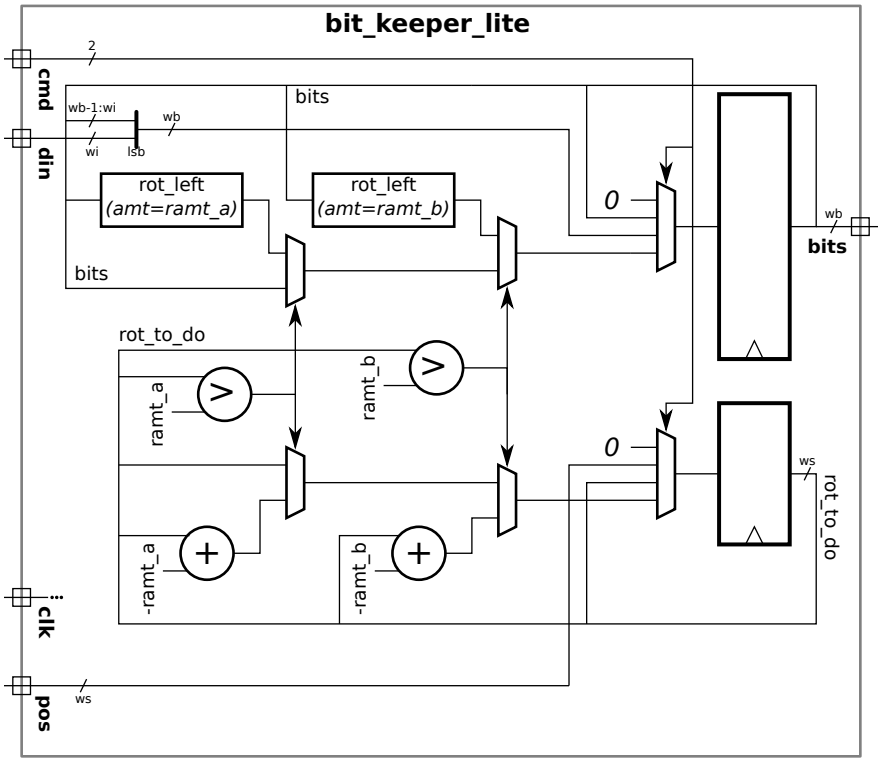
☑ Since `ramt_b` is a power of 2 the adder and comparison unit connected to `ramt_b` are lower cost because:

Because the lower $w_s/2$ bits of `ramt_b` are all zero. Because `ramt_b` is also a constant there is no need for an adder at all for the least significant $w_s/2$ bits.

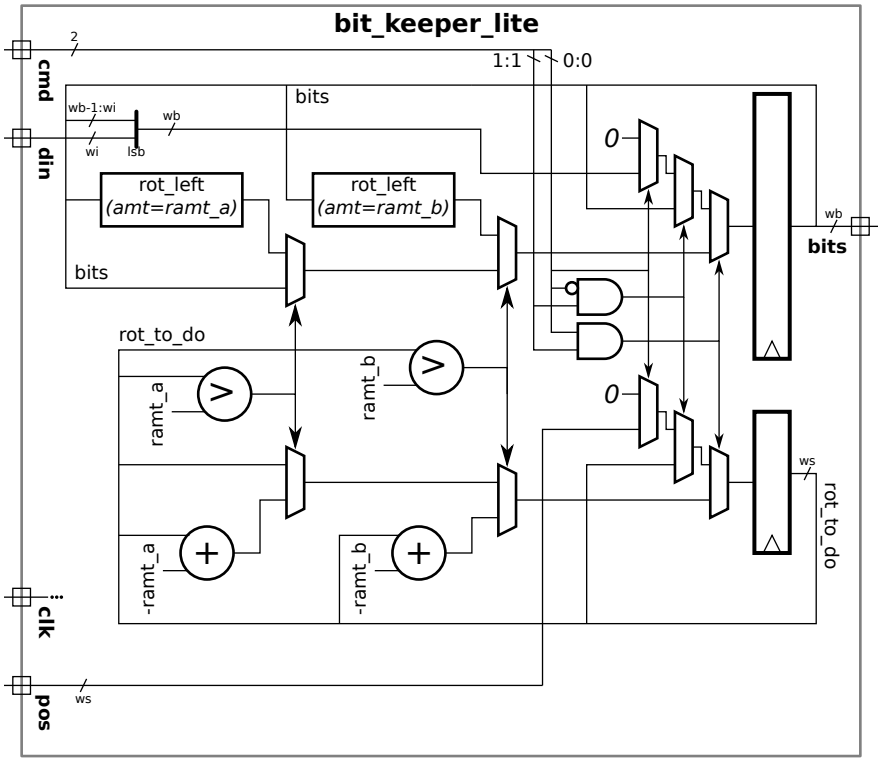
☑ Since `ramt_b` is a power of 2 the adder and comparison unit connected to `ramt_a` (yes, a) are lower cost because:

Because the output of the `ramt_a` adder is only used if `rot_to_do <= ramt_b`. Therefore there is no point in providing an adder that can handle more than $w_s/2$ bits. For the same reason the comparison unit need only consider the lower $w_s/2$ bits.

Staple This Side



Solution appears below.

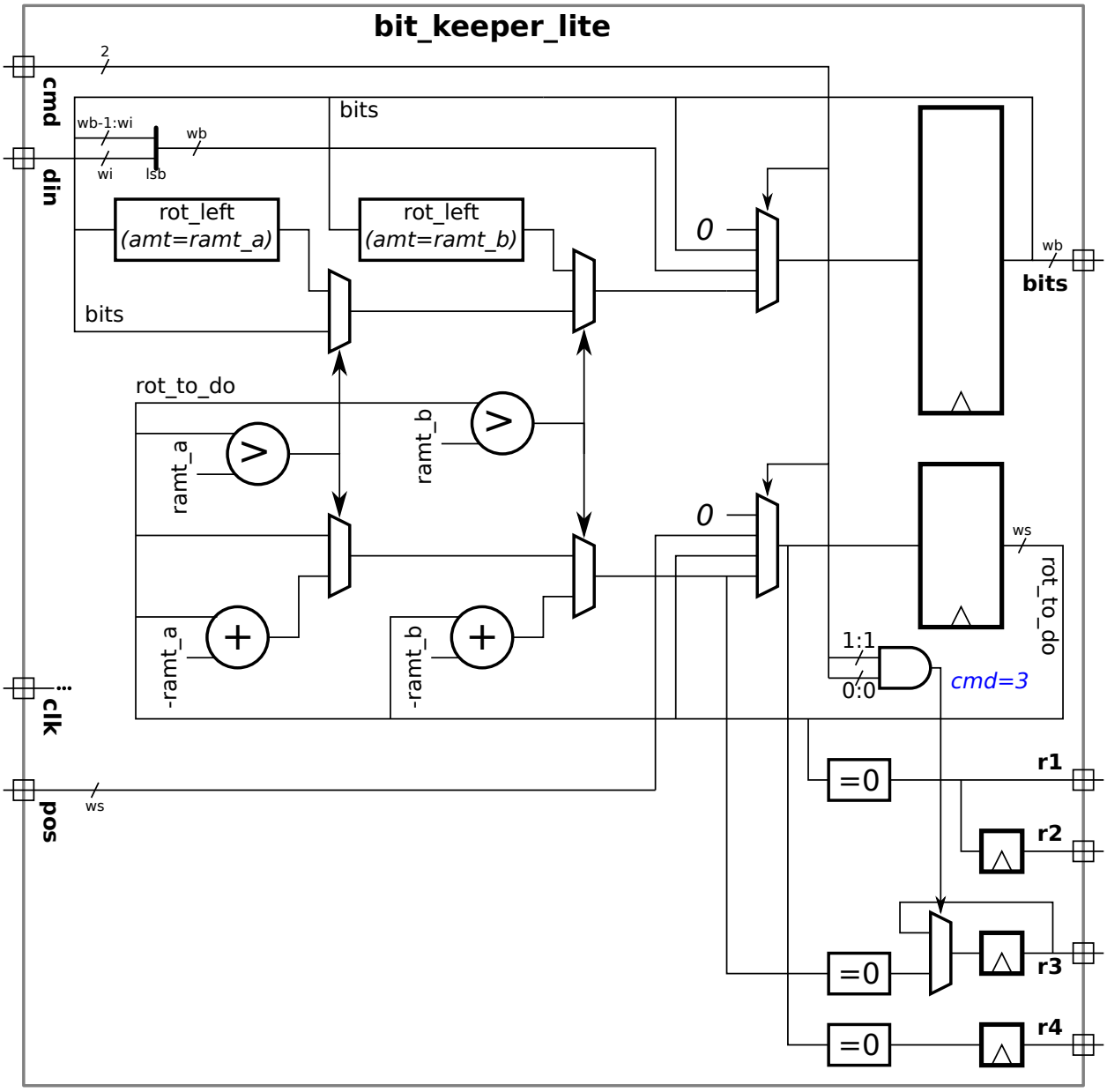


(*d*) Appearing below is a version of `bit_keeper_lite` with four ready outputs, `r1`, `r2`, `r3`, and `r4`. On the diagram add hardware that will be synthesized for each.


```
module bit_keeper_lite #( int wb = 64, wi = 8, ws = $clog2(wb) )
   ( output logic [wb-1:0] bits,    output uwire r1, output logic r2, r3, r4,
     input uwire [1:0] cmd,         input uwire [wi-1:0] din,
     input uwire [ws-1:0] pos,      input uwire clk );

   localparam int ramt_a = 1;
   localparam int ramt_b = 1 << ( ws >> 1 );

   uwire [wb-1:0] ra, rb;
   rot_left #(wb,ramt_a) rl1(ra,bits);
   rot_left #(wb,ramt_b) rl8(rb,bits);

   logic [ws-1:0] rot_to_do;
   assign r1 = rot_to_do == 0;          // [☑ Show hardware for r1.

   always_ff @( posedge clk ) begin
      r2 = rot_to_do == 0;              // [☑ Show hardware for r2.
      case ( cmd )
        Cmd_Reset:  begin bits = 0; rot_to_do = 0; end
        Cmd_Rot_To: rot_to_do = pos;
        Cmd_Write:  bits[wi-1:0] = din;
        Cmd_Nop: begin
           if ( rot_to_do >= ramt_b ) begin
              bits = rb;
              rot_to_do -= ramt_b;
           end else if ( rot_to_do >= ramt_a ) begin
              bits = ra;
              rot_to_do -= ramt_a;
           end
           r3 = rot_to_do == 0;         // [☑ Show hardware for r3.
        end
      endcase
      r4 = rot_to_do == 0;              // [☑ Show hardware for r4.
   end
endmodule
```

☑ Show hardware that will be synthesized for `r1`, `r2`, `r3`, and `r4`.

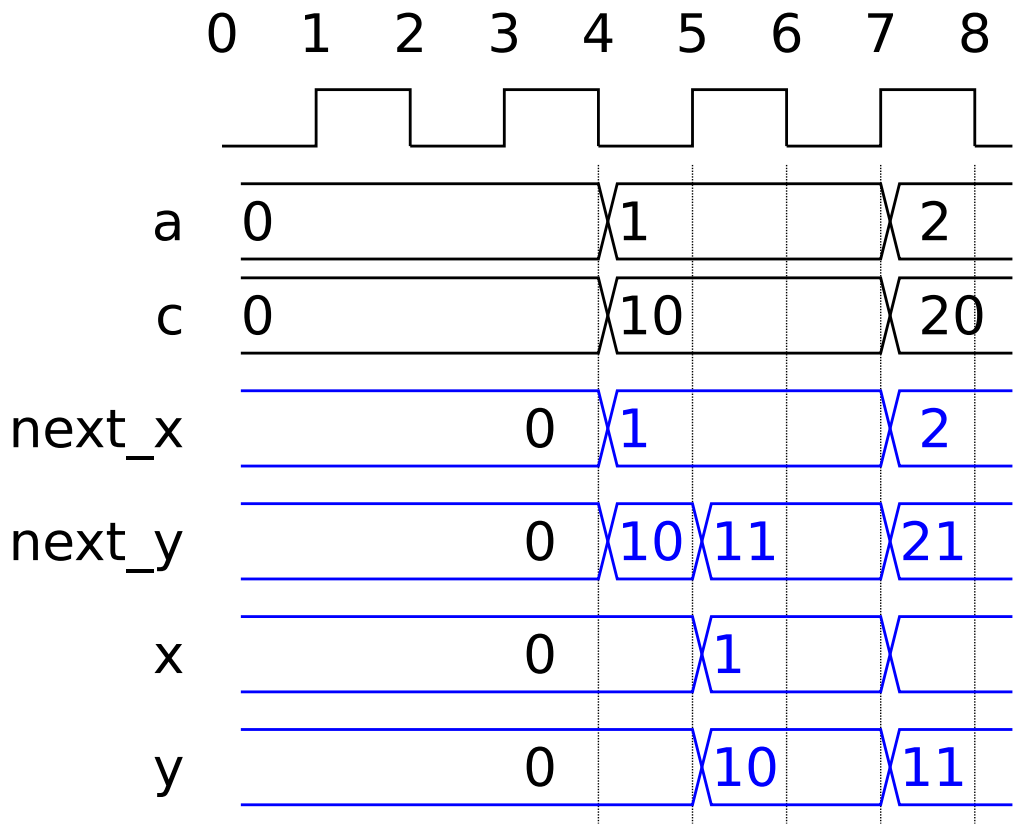
Solution appears on the next page. Because they are assigned in an `always_ff`, the values of `r2`, `r3`, and `r4` visible outside the block come from registers. Pay close attention to where `rot_to_do` is assigned and where its value is referenced. For `r1` it is referenced outside of the `always_ff` block and so the value is from the register. The value of `rot_to_do` used for `r2` also comes from the register output because it had not been assigned yet in the block. For `r3` the value of `rot_to_do` assigned in the `cmd=Cmd_Nop` case is used. A mux keeps `r3` unchanged when `cmd` is not `Cmd_Nop`. (The value of enumeration constant `Cmd_Nop` is 3.) Finally, `r4` is assigned at the end of the block, so it uses the value of `rot_to_do` that will be written to the register.



Problem 3: [15 pts] Consider the modules below.

```
module ba
  ( output logic [15:0] next_x, next_y, x, y,
    input uwire [15:0] a, c,  input uwire clk );

   always_ff @( posedge clk ) x = next_x;
   assign next_x = a;
   assign next_y = x + c;
   always_ff @( posedge clk ) y = next_y;

endmodule

module test_ba;

   uwire [15:0] x, y, next_x, next_y;
   logic [15:0] a, c;
   logic clk;

   ba ba1( next_x, next_y, x, y, a, c, clk );

   initial begin
      // t = 0
      clk = 0;
      a = 0;  c = 0;
      #1; // t = 1
      clk = 1;
      #1; // t = 2
      clk = 0;
      #1; // t = 3
      clk = 1;
      #1; // t = 4
      clk = 0;  a <= 1;   c <= 10;  // Line t4
      #1; // t = 5
      clk = 1;
      #1; // t = 6
      clk = 0;
      #1; // t = 7
      clk = 1;  a <= 2;   c <= 20;  // Line t7
      #1; // t = 8
      clk = 0;
   end

endmodule
```


Staple This Side



(*a*) Complete the timing diagram so that it shows the values of `next_x`, `next_y`, `x`, and `y` that would be produced with the modules above. *Note: In the original exam* `test\_ba` *did not use non-blocking assignments to* `a` *and* `c`*.*

☑ Complete timing diagram from $t = 4$ to $t = 8.1$. ☑ Note that there is a **negative** clock edge at $t = 4$.

Solution appears above.

(*b*) At $t = 5$ we can be sure that `y=next_y` will execute before `next_y=x+c`. Explain how this ordering is assured by the rules for the event queue.

☑ Explain how event queue regions assure `y=next_y` executes before `next_y=x+c` at $t = 5$.

At $t = 5$ `clk` changes from `0` to `1`, resulting in the two `always_ff` items being scheduled. The two will eventually reach the active region of the event queue, and one of them will be chosen first. Assume that the first `always_ff` is chosen first. The `next_y` assignment has `x` and `c` in its sensitivity list, and so it is only scheduled for execution when at least one of these changes. At $t = 5$ `x` changes, and that will result in the `next_y` assignment being placed in the inactive region of the event queue. The scheduler will continue to remove and execute events from the active region until the active region is empty. Therefore the second `always_ff` is guaranteed to execute before the `next_y` assignment.

(*c*) Notice that `a` and `c` are assigned using non-blocking assignments on Lines t4 and t7. Explain why the order of execution would be ambiguous at $t = 7$ if line t7 used blocking assignments: `a=1; c=10;`. *Note: This question was not in the original exam.*

☐ Describe ambiguity (more than one possible execution order) if blocking assignments were used.

☐ Would non-blocking assignments `x <= next_x` and `y <= next_y` remove the ambiguity? ☐ Explain.

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

(*a*) The foolish `sqrt` module below has several issues.

```
module sqrt #( int w = 16 )
   ( output logic [w-1:0] r, input uwire [w-1:0] a );

   always_comb begin

      r = 0;
      while ( r * r < a ) r++;

   end

endmodule
```

☑ Explain why, due to the Verilog rules for bit widths, the expression `r * r < a` won't compute the intended result.

Because `r` and `a` are 16 bits the computation will be done to 16 bits of precision, and so due to overflow `r*r<a` can be false when it should be true.

☑ Why is the `sqrt` module likely not synthesizeable?

Because the maximum number of iterations of the `while` loop cannot be directly determined. The maximum number of iterations in fact will be about $2^{w/2}$, and it's not impossible that a synthesis program would figure that out. It's just not likely because this is not the typical loop that would be used to describe hardware.

☑ What would be the problem with the hardware if it were synthesizable?

The maximum number of iterations is $2^{w/2}$. For the default value that's $2^8 = 256$. There would need to be 256 multiply units, 256 comparison units, and 256 muxen. That's alot of hardware. And anyway there are much better ways of computing a square root.

Staple This Side

(*b*) Consider the two division modules below. In the first `a2` is a parameter, in the second it is a module port. Use the `div_demo` module for your answers to the questions below.

```
module our_div_by
  #( int wq = 5, wd = 10, logic [wd-1:0] a2 = 4 )
   ( output uwire [wq-1:0] quot, input uwire [wd-1:0] a1 );
   assign quot = a1/a2;
endmodule

module our_div
  #( int wq = 5, wd = 10 )
   ( output uwire [wq-1:0] quot, input uwire [wd-1:0] a1, a2 );
   // cadence inline
   assign quot = a1/a2;
endmodule

module div_demo
  #( int w = 21 )
   ( output uwire [w-1:0] d1, d2,
     input uwire [w-1:0] x1, x2, x3, x4 );

   localparam logic [w-1:0] y1 = 4755;

   // Could replace our_div with our_div_by because y1 is constant.
   our_div #(w,w) dwould_work(d1, x1, y1);

   // Could not replace our_div with our_div_by because
   // divisor (x2) not a constant.
   our_div #(w,w) dwould_not_work(d2, x1, x2);

endmodule
```

☑ Show an instantiation of `our_div` for which `our_div_by` could work.

☑ Show an instantiation of `our_div` for which `our_div_by` could not work.

Solution appears above. To use `our_div_by` the divisor needs to be a constant. That's the case in the first example, but not in the second example

☑ Explain how the use of the `cadence inline` pragma in `our_div` makes it possible to instantiate `our_div` in places that otherwise might need `our_div_by`.

It ensures that each instantiation of `our_div` will be optimized separately based on its arguments. Without the pragma the synthesis program might optimize `our_div` once, assuming two non-constant inputs, and then copy the optimized description to places where there are constant inputs.

(*c*) Answer the following questions about latency and throughput.

☑ Define latency.

Latency is the amount of time needed to compute a result from start to finish. What a result is depends on the context. The result might be computed combinationally, or sequentially over several cycles.

☑ Define throughput.

Throughput is the number of results computed per unit time. For example, if over 10 seconds 200 results are computed, the throughput is $200/10 = 20$ results per second.

Consider a sequential circuit (such as `mult_step` from Homework 6) and a pipelined version of the sequential circuit (such as `multi_step_pipe`). Assume that both have the same clock frequency.

☑ Remembering that the clock frequencies are the same, compared to the sequential version, does the pipelined version typically have
◯ *lower latency*, ⊗ *the same latency*, or ⊗ *higher latency*. ☑ Explain.

It depends. In a reasonable design the latency of the sequential version will be equal to or possibly greater than the pipelined version. A sequential design can re-use hardware, and so if it prioritizes low cost it will use less hardware over a greater number of cycles resulting in a higher latency than a pipelined design.

☑ Compared to the sequential version, does the pipelined version typically have
◯ *lower throughput*, ◯ *the same throughput*, or ⊗ *higher throughput*. ☑ Explain.

By definition, a pipelined circuit computes a result each clock cycle, and so its throughput is high. A sequential circuit will require several cycles to compute something and so its throughput will be lower.

☑ **Ignoring the cost of registers**, compared to the sequential version, does the pipelined version typically have
◯ *lower cost*, ◯ *the same cost*, or ⊗ *higher cost*. ☑ Explain.

The sequential version re-uses units (such as arithmetic units) over multiple cycles. The pipelined version must have one unit for each operation, and so its cost will be higher.

Staple This Side

18 Fall 2020 Solutions

Name Solution_____

Digital Design Using HDLs
LSU EE 4755
Solve-Home Midterm Examination

Friday, 6 Nov 2020 to early Monday, 9 Nov 2020 05:00 CST)

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Outside material that covers the same topics, such as Verilog tutorials, digital logic design guides can also be used. Do not try to directly seek out solutions to any question here. That is, don't Web-search the text of a problem. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Warning: Unlike homework assignments collaboration is not allowed on exams. Suspected copying will be reported to the dean of students. The kind of copying on a homework assignment that would result in a comment like "See ee4755xx for grading comments" will be reported if it occurs on an exam. Please do not take advantage of pandemic-forced test conditions to cheat!

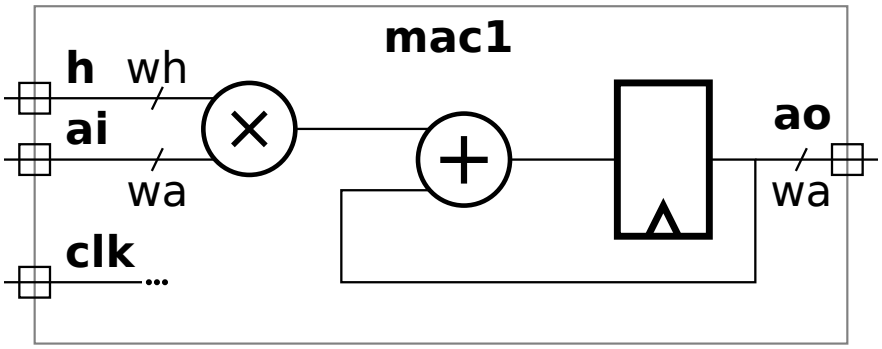
Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Exam Total	_____	(100 pts)

☣ $r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck!

Problem 1: [20 pts] Appearing below are some variations on a multiply accumulate module.

(*a*) Complete the Verilog code below so that it matches the illustration.



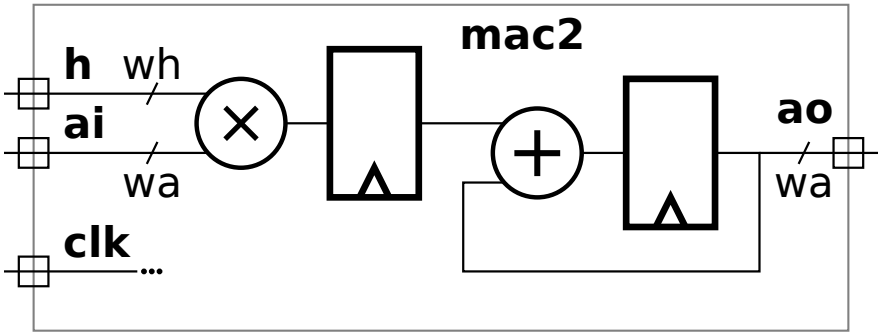
☑ Complete the Verilog.

☑ Use parameters for the bit widths `wh` and `wa`.

☑ The registers inferred from the Verilog must match the diagram.

```
 /// SOLUTION
module mac1
  #( int wa = 32, wh = 16 )
   ( output logic [wa-1:0] ao,
     input uwire [wh-1:0] h,
     input uwire [wa-1:0] ai,
     input uwire clk );

   always_ff @( posedge clk ) ao <= h * ai + ao;

endmodule
```

(*b*) Complete the Verilog code below so that it matches the illustration, similar to the one on the previous page.



☑ Complete the Verilog.

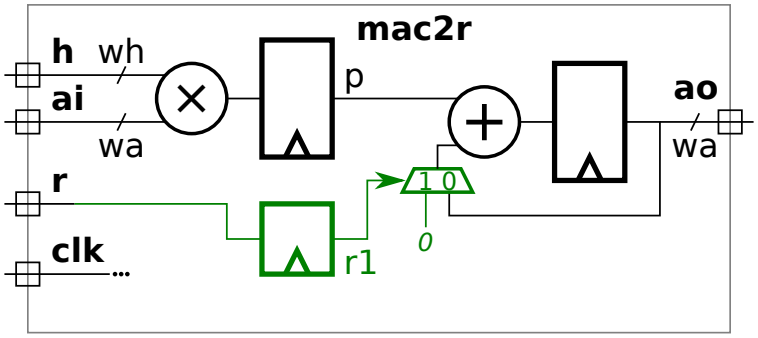
☑ Use parameters for the bit widths `wh` and `wa`.

☑ The registers inferred from the Verilog must match the diagram.

```
 /// SOLUTION
module mac2
   #( int wh = 4, wa = 3 )
   ( output logic [wa-1:0] ao,
     input uwire [wh-1:0] h,
     input uwire [wa-1:0] ai,
     input uwire clk );

   logic [wa-1:0] p;

   always_ff @( posedge clk ) begin
      p <= h * ai;
      ao <= p + ao;
   end

endmodule
```

Problem 2: [20 pts] The mac (multiply-accumulate) modules compute a running sum of products. The alert student might have noticed that there is no way to reset the sum. In this problem a reset will be added.

The module below has an input `r` (for reset) which is to work as follows: When `r=1` at a positive edge the product `h*ai` should start a new running sum. That is, that particular `h*ai` should be added to zero. When `r=0` at a positive edge the product `h*ai` should be added to the sum of the previous products. (If `r=0` is always true then the hardware as illustrated works correctly.)



☑ Add hardware to the diagram to implement the reset. ☑ Complete the Verilog to implement the reset.

☑ Use parameters for the bit widths `wh` and `wa`.

☑ The registers inferred from the Verilog must match the diagram and ☑ be sure that the reset is applied to the correct value.

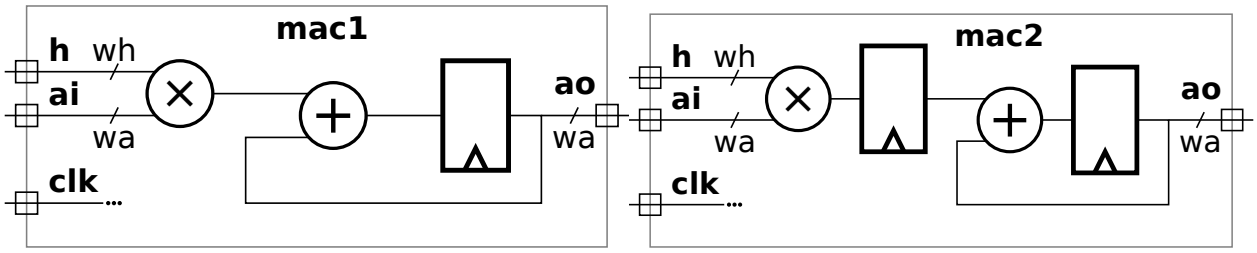
The hardware changes appear above in green and the Verilog code appears below in all sorts of colors.

The problem states that when `r=1` the accompanying values of `h` and `ai` must start a new running sum. To implement this a register has been added, `r1`, so that the value of `r` moves with the product `h*ai`, so that in the next cycle that product `h*ai` is added to zero rather than to `ao`. If `r` were connected directly to the multiplexor then the `h*ai` arriving with `r` would be added to a non-zero value.

Grading Note: No one solved this 100% correctly.

```
/// SOLUTION
module mac2r #( int wh = 4, wa = 3 )
   ( output logic [wa-1:0] ao,
     input uwire [wh-1:0] h,
     input uwire [wa-1:0] ai,
     input uwire r, clk );

   logic [wa-1:0] p;
   logic r1;

   always_ff @( posedge clk ) begin
      r1 <= r;
      p <= h * ai;
      ao <= p + ( r1 ? 0 : ao );
   end
endmodule
```

Problem 3: [20 pts] Appearing below are the modules from the previous problem. Suppose that in the multiplier below bit i of the product were computed in time $[4i+2]\,\mathrm{u_t}$ and that a ripple adder were used for the sum. Let w denote the value of `wh` and `wa` (which means `wh==wa`).



(*a*) Find the minimum clock period for each using the simple model, and taking into account cascading. (The clock period is the length of the critical path, including the register delay.)

☑ Find the clock period for `mac1` with cascading. ☑ Don't forget to include the delay of the register.

Short answer: The clock period is $[4(w-1)+2+4+6]\,\mathrm{u_t} = [4w+8]\,\mathrm{u_t}$.

Explanation: Taking into account cascading in this case means that when we compute the time needed to compute the addition we take into account the fact that bit i arrives at time $4i+2$ and that a ripple adder is used to compute the sum. Bit 0 of the product arrives at the adder's bit 0 BFA at time $4\times 0+2=2$, and so its outputs, sum and carry-out, arrive at $2+4=6$. Bit 1 of the product arrives at the adder's bit 1 BFA at time $4\times 1+2=6$ as does the carry out from bit 0, so the sum and carry out won't be available until $6+4=10$. This pattern persists, so bit i of the adder output is available at time $4i+2+4=4i+6$. The adder is w bits wide, so the MSB is not available until time $[4(w-1)+6]\,\mathrm{u_t} = [4w+2]\,\mathrm{u_t}$. To compute the clock period we need to tack on the $6\,\mathrm{u_t}$ register delay, bringing the clock period to $[4w+2+6]\,\mathrm{u_t} = [4w+8]\,\mathrm{u_t}$.

☑ Find the clock period for `mac2` with cascading. ☑ Don't forget to include the delay of the registers.

Short answer: The clock period is $[\max\{\,4(w-1)+2,\,2(w+1)\,\}+6]\,\mathrm{u_t} = [\max\{\,4w-2,\,2w+2\,\}+6]\,\mathrm{u_t} = [4w-2+6]\,\mathrm{u_t} = [4w+4]\,\mathrm{u_t}$, for $w \geq 2$.

Explanation: The clock period is determined by the critical (longest) path. Paths start at launch points and end at capture points. Register outputs are launch points and register inputs (both data and enable) and capture points. Usually (but not always) module inputs are launch points and module outputs are capture points. There are two possible critical paths. Path one is from `h` (or `ai`), through the multiplier, to the register input, path two is from `ao`, through the adder, to the register input. The length of path one is $4(w-1)+2+6=4w+4$ and the length of path two is $2(w+1)+6=2w+8$. When $w \geq 1$ path one is longer and so the clock period must be $[4w+4]\,\mathrm{u_t}$.

And what about cascading? That doesn't apply here because there is a register between the multiplier and the adder and so all bits arrive at the input to the adder at the same time.

(*b*) Find the minimum clock period for each using the simple model assuming that the multiplier output and adder input could not cascade.

☑ Find the clock period for `mac1` without cascading. ☑ Don't forget to include the delay of the register.

Short Answer: The clock period is $[4(w-1)+2+2(w+1)+6]\,\mathrm{u_t} = [6w+6]\,\mathrm{u_t}$.

Explanation: Without cascading the adder must wait for every bit of the product to be computed. The last bit of the product is available at $4(w-1)+2$ and only then can the addition start (with the no-cascading assumption). So adding the addition time, $2(w+1)$, and register delay, 6, gives the clock period.

Note that the no-cascading assumption was made for pedagogical reasons. If indeed bit i of the product arrives at $4i+2$ and a ripple adder is used, cascading should be taken into account when computing the delay.

☑ Find the clock period for `mac2` without cascading. ☑ Don't forget to include the delay of the registers.

The clock period for `mac2` is the same with and without the cascading assumption, so the period is the same as the one computed above, $[4w + 4]\,\mathrm{u_t}$.

Problem 4: [20 pts] Appearing below is a recursively defined multiplier constructed using bfa (binary full adder) and bha (binary half adder) modules.

```
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
      // Split a in half and recursively instantiate a module for each half.
      localparam int wn = wa / 2;
      localparam int wx = wb + wn;
      uwire [wx-1:0] prod_lo, 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
```

Show the hardware that will be inferred for two levels of recursion and compute its cost. That is, show three instances of `mult_tree_bfas`: a top-level one, and two recursive instantiations. Show the hardware for the top-level instance and both of the two recursive instantiations. (It is only necessary to show two levels.) Do this for `wa=8` in the top-level module.

Continued on next page.

☑ Show the inferred hardware.

The inferred hardware is shown on the next page. The binary full and half adders are shown as boxes. Only the carry-out port is labeled, with a `co`, of course. As most reading this should easily figure out, the port at the top of each BFA and BHA module is carry-in, the port on the right-hand side is the sum, and the two BFA inputs on the left hand side are the two bits to be added together.

Note that the bits of `a` are split so that the less significant bits connect to `mlo` and the more significant bits connect to `mhi`. In contrast *all bits of* `b` connect to both `mlo` and `mhi`.

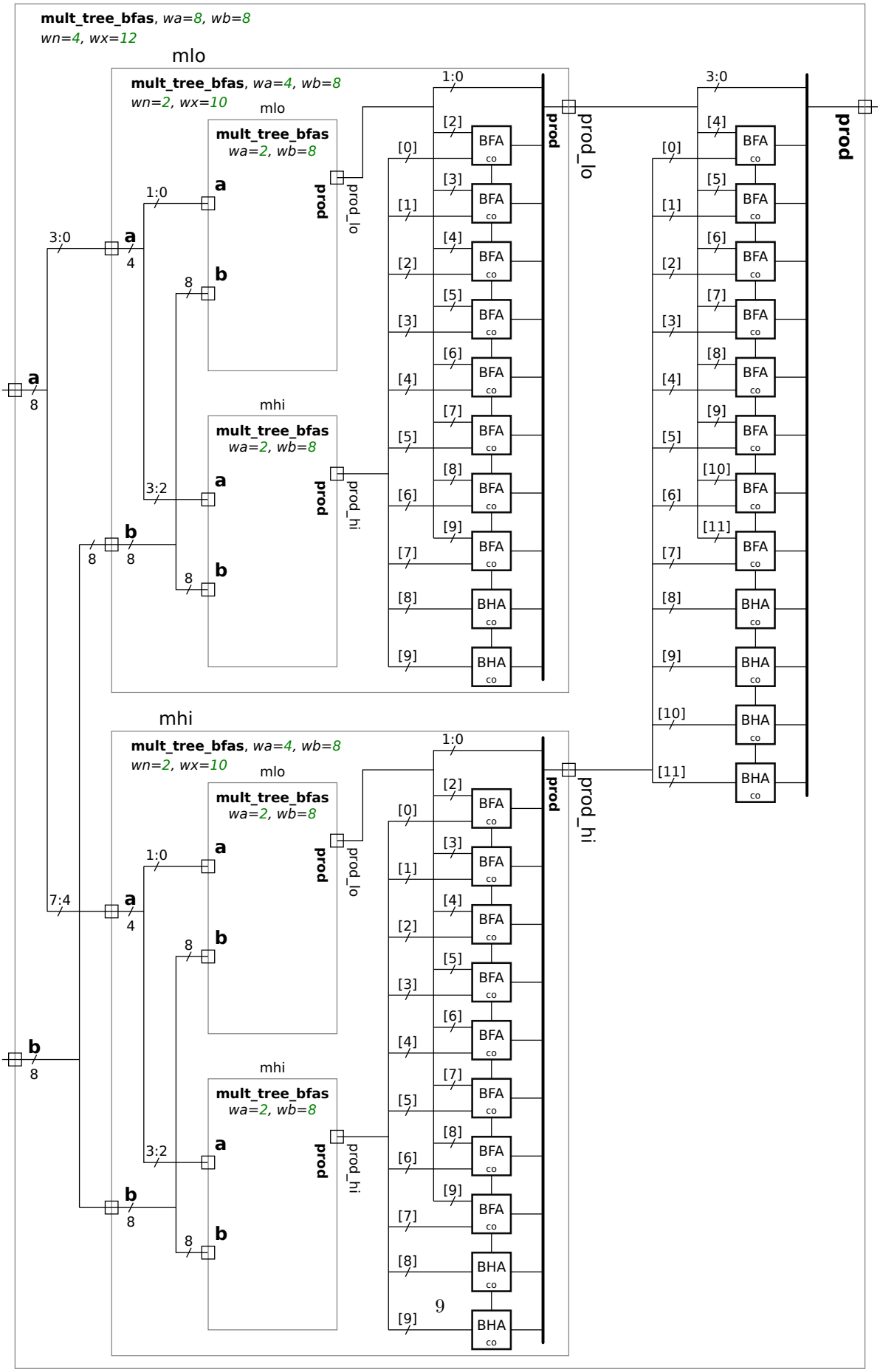
☑ Be sure to distinguish hardware (such as a `bfa` module) from values computed during elaboration.

An example of a value computed during elaboration is `wx`. The value at each level is shown. Since the value has been computed during elaboration there is no need to emit hardware to compute a value that is already known.

☑ Compute the cost of the hardware in your diagram using the simple model. (Work out the cost of a bha by hand.) ☑ The cost should be for two levels, not for hardware going down to the base case.

As can be seen by looking at the loop bounds of the generate loops, each instance consists of `wb` BFA modules and `wn` BHA modules. For the top-level (`wa=8`) instance `wb=8` and `wn=4`. In the `mlo` and `mhi` instances instantiated in the top level `wb=8` and `wn=2`. (Yes, `wb` is 8 at every level.) So the number of BFA modules is $8 + 2 \times 8 = 24$, and the number of BHA modules is $4 + 2 \times 2 = 8$. The cost of a BFA is $9\,\mathrm{u_c}$. A BHA can be constructed with an XOR gate for the sum and an AND gate for the carry out, for a cost of $4\,\mathrm{u_c}$. However the carry out can be used to compute the sum: `s = ( a || ci ) & !co` where `co = a && ci`. Such a construction has a cost of just $3\,\mathrm{u_c}$.

The total cost is $[24 \times 9 + 8 \times 3]\,\mathrm{u_c} = 240\,\mathrm{u_c}$ using the $3\,\mathrm{u_c}$ BHA or $[24 \times 9 + 8 \times 4]\,\mathrm{u_c} = 248\,\mathrm{u_c}$ using the $4\,\mathrm{u_c}$ BHA.


mult_tree_bfas, wa=8, wb=8
wn=4, wx=12
mlo
mult_tree_bfas, wa=4, wb=8
wn=2, wx=10
mlo
mult_tree_bfas
wa=2, wb=8
mhi
mult_tree_bfas
wa=2, wb=8
mhi
mult_tree_bfas, wa=4, wb=8
wn=2, wx=10
mlo
mult_tree_bfas
wa=2, wb=8
mhi
mult_tree_bfas
wa=2, wb=8
BFA
BHA
co
prod
prod_lo
prod_hi
a
b

Problem 5: [20 pts] Answer each question below.

(*a*) Appearing below is a multiply/add module, `nnMADDfp`, that computes its result using a FP add and multiply module. The values on the ports are IEEE 754 floats, and when `wa=32` the format is IEEE 754 single, the same as a SystemVerilog `shortreal`. That is followed by an incomplete testbench module, `testnnMADD`. The testbench module generates random values for the `nnMADDfp` module in variables `ar`, `br`, and `sir`, and computes what the result should be, `sor`.

Add Verilog code to deliver `ar`, `br`, and `sir` to the `nnMADDfp` instance, and to put the output of `nnMADDfp` into `sor_mut` so that `sor_mut` has the correct type of value. Note that one does not need to understand what is inside of `nnMADDfp`, `nnAddfp`, nor `nnMultfp`.

☑ Deliver (whatever that means) `ar`, `br`, and `sir` to `nnMADDfp` instance. ☑ Get output of the `nnMADDfp` instance into variable `sor_mut`.

The solution is shown below. First the inputs to instance `n` of `nnMADDfp`, variables `a`, `b`, and `si`, must be assigned the values in variables `ar`, `br`, and `sir`. Because `a`, `b`, and `si` are of type `logic` a statement like `a=ar` won't work because for such a statement Verilog will first convert `ar` from a `shortreal` to a 32-bit unsigned integer (`logic [31:0]`). It won't work because module `nnMADDfp` expects `a`, though declared `logic`, to be in the same format as `shortreal`. To avoid the problem the Verilog system task `$shortrealtobits` is used. That avoids the `shortreal`-to-integer or any other conversion. The bits are left unchanged. A similar function is used for re-interpreting the module output, `so`, from `logic` to `shortreal`.

A serious error which too many students made was instantiating an `nnMADDfp` module inside the `t` loop. First, the module is already instantiated. Second, it makes no sense to instantiate a module in procedural code.

Note: In the original exam `ar`, `br`, etc. were declared `real` instead of `shortreal`. The solution would be no different: `$shortrealtobits` should still be used. But the explanation above would have been more complicated since in statement `a=$shortrealtobits(ar)` there would be a conversion: `ar` would be converted from `real` to `shortreal`, but then the bits in the `shortreal` would be assigned to `a` with no further changes.

Verilog code, including the solution, on next page.

```
module nnMADDfp #( int wa = 10 )
  ( output uwire [wa-1:0] so, input uwire [wa-1:0] a, b, si);
   uwire [wa-1:0] p;
   nnMultfp #(wa) mu(p, a, b);
   nnAddfp #(wa) ad(so, si, p);
endmodule

module testnnMADD;
   localparam int w = 32, ntests = 100;
   uwire [w-1:0] so;
   logic [w-1:0] a, b, si;
   nnMADDfp #(w) n(so, a, b, si);

   initial begin

      for ( int t=0; t<ntests; t++ ) begin
         shortreal sor, ar, br, sir, sor_mut;
         ar = rand_fp();    // Value to be used as input a to nnMADDfp.
         br = rand_fp();    // Value to be used as input b to nnMADDfp.
         sir = rand_fp();   // Value to be used as input si to nnMADDfp.
         sor = ar * br + sir;

         /// SOLUTION
         a = $shortrealtobits(ar);   // Move bits of ar to a without changing them.
         b = $shortrealtobits(br);   // This operation is sometimes called ..
         si = $shortrealtobits(sir); // .. a reinterpretation cast.

         #1;

         sor_mut = $bitstoshortreal(so);  // <-- MORE OF THE SOLUTION.


         if ( sor != sor_mut ) handle_incorrect_result();

      end
   end
endmodule
```

(*b*) The module below will not compile or simulate due to multiple assignments to `temperature`, which is declared `uwire`. Changing `uwire` to `wire` will fix the compile problem. Nevertheless, is that the right fix?

```
module more_stuff #( int w = 16 )
   ( output uwire [w-1:0] v, y, input uwire [w-1:0] a, b, c );

   uwire [w-1:0] temperature;

   assign temperature = a + b;
   assign v = temperature >> c;
   assign temperature = a - b;
   assign y = temperature << c;

endmodule
```

```
module more_stuff #( int w = 16 )
   ( output uwire [w-1:0] v, y, input uwire [w-1:0] a, b, c );

   /// SOLUTION

   uwire [w-1:0] t1, t2;

   assign t1 = a + b;
   assign v = t1 >> c;
   assign t2 = a - b;
   assign y = t2 << c;

endmodule
```

☑ What problem remains after changing `temperature` from a `uwire` to a `wire`?

Short answer: The same value of `temperature` is used to compute both `v` and `y`, though the coder's intent may have been different values. That value of `temperature` consists of bits common to `a+b` and `a-b`, and `x`'s elsewhere. It's not likely the coder intended that either.

Longer explanation: With the "fix" object `temperature` is driven by two different assignments, `a+b` and `a-b`. In bit positions where `a+b` and `a-b` are both `0`, the value would be `0`. In bit positions where `a+b` and `a-b` are both `1`, the value would be `1`. But, in bit positions where `a+b` and `a-b` differ the value would be `x`.

An important thing to remember is that continuous assignments, which is what the `assign` keyword specifies, are executed whenever objects on the right-hand side change. As a consequence the values for both `v` and `y` will ultimately be computed with **the same value** of `temperature`.

☑ Fix the problem based on what the code looks like its trying to do.

Solution appears above.

(*c*) An important part of synthesis is optimizing. It is possible to optimize before and again after technology mapping.

☑ What is technology mapping? ☑ Show an example of logic before and after technology mapping. (Make up some technology.)

In the technology mapping step generic gates are replaced with gates in the target technology. For example, consider the expression `y=!( a & b || c & d )`. That might be inferred into the following generic gates: two AND gates and one NOR gate. A target technology might have a special AND-OR-INVERT gate that computes the entire expression, and because of the way CMOS FETs can be interconnected does so using less time or area than two AND gates and a NOR gate in the same technology.

☑ Describe an optimization that can be done before technology mapping. Provide an example. (This is done all the time in class.)

Expression `a || (!a) && b` can be optimized to `a || b`. Other examples include constant propagation and folding, such as `a && 1` being optimized to `a`.

☑ Describe an optimization that can be done only after technology mapping (or perhaps during). Provide an example, feel free to make things up.

Realistic timing data is available for the gates used in technology mapping. For that reason, any optimization that reduces delay should be done after technology mapping.

Name Solution

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 mRNA!

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.*

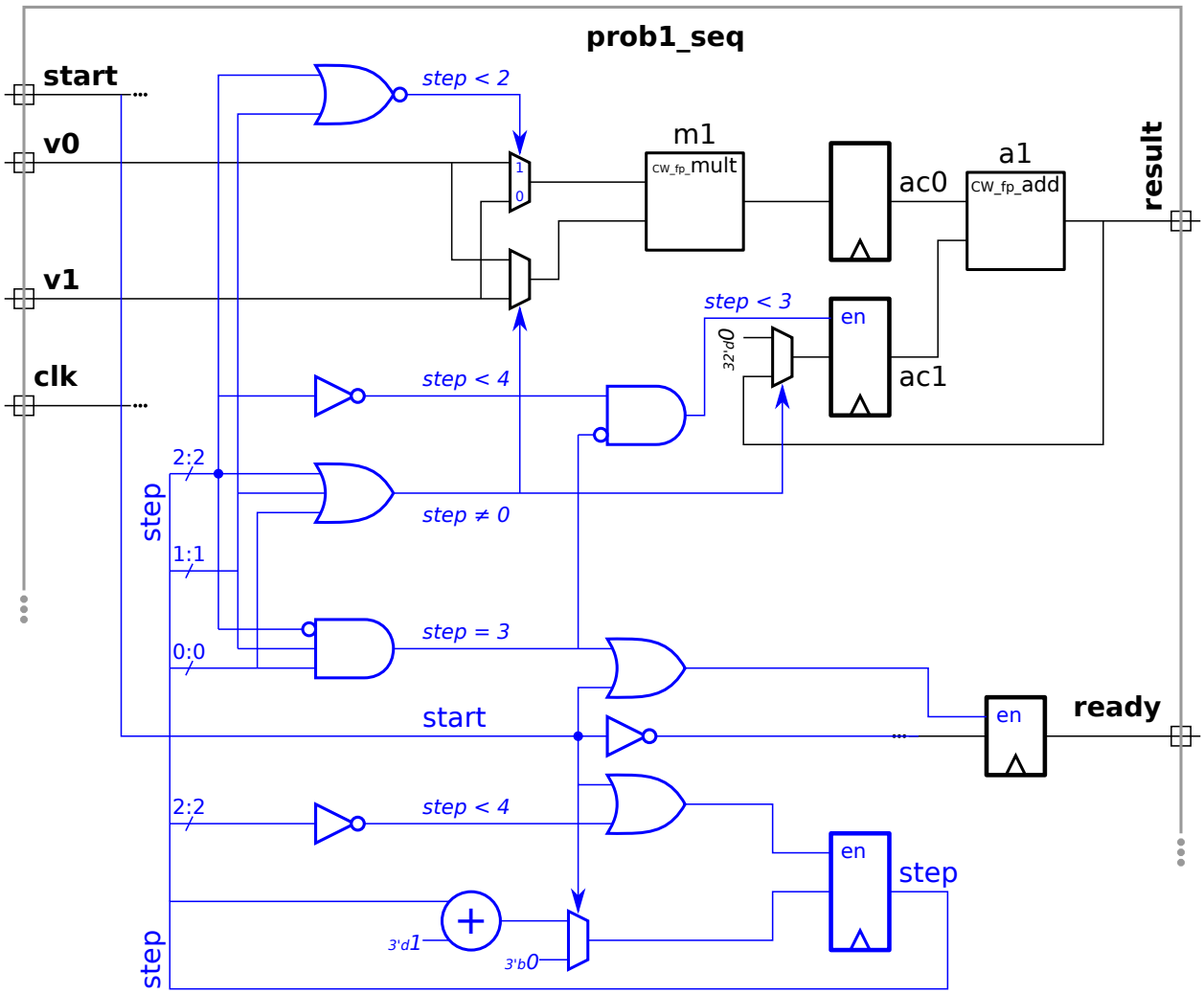
Though module `prob1_seq` is now complete, the hardware diagram isn't. In this problem complete the diagram of the synthesized hardware based on the module below. The diagram omits the hardware for `step`, select signals for the multiplexors, enable signals for some of the registers, etc. Optimize the 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
```

Solution appears below in blue. A register was added to hold `step`. The value of `step` and `start` are used to determine multiplexor select signals and register enable inputs. The solution is labeled with some step comparison results, such as `step < 2`. Those who are unsure of how the illustrated logic computes these values are strongly urged to draw a truth table.



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.

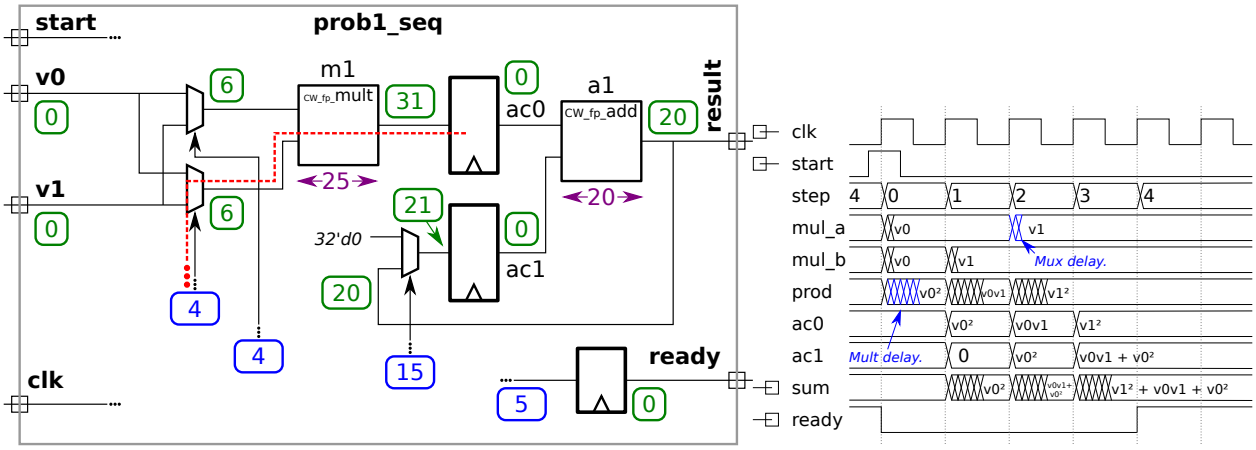
☑ Complete module so that it operates in pipelined fashion.

The solution that appears below is what would be expected on an exam. This problem was assigned as 2021 Homework 6 as a programming assignment. See that solution for additional details. In the solution below notice that the **start** signal is carried along the pipeline and finally connected to the **ready** output port.

The sequential hardware uses the value of register **step** so determine multiplexor and enable settings. That's not needed here because each stage does a particular step, in effect make step a constant. (Also, because there are more functional units fewer steps are needed.) For that reason there is no equivalent to the step register in the pipelined solution.

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

   /// SOLUTION

   uwire [31:0] v00, v01, v11, s1, s2;
   logic [31:0] pl_1_v00, pl_1_v01, pl_1_v11;
   logic [31:0] pl_2_v0001, pl_2_v11;
   logic pl_1_occ, pl_2_occ;

   CW_fp_mult m00( .a(v0), .b(v0), .z(v00) );
   CW_fp_mult m01( .a(v0), .b(v1), .z(v01) );
   CW_fp_mult m11( .a(v1), .b(v1), .z(v11) );

   CW_fp_add  a1(.a(pl_1_v00),   .b(pl_1_v01), .z(s1) );
   CW_fp_add  a2(.a(pl_2_v0001), .b(pl_2_v11), .z(s2) );

   always_ff @( posedge clk ) begin

      pl_1_v00 <= v00;
      pl_1_v01 <= v01;
      pl_1_v11 <= v11;
      pl_1_occ <= start;

      pl_2_v0001 <= s1;
      pl_2_v11 <= pl_1_v11;
      pl_2_occ <= pl_1_occ;

      result <= s2;
      ready <= pl_2_occ;

   end
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.

The arrival times of stable values are shown in the diagram boxed in green and a critical path is shown as a red dashed line. Another critical path (which must of the same length) pass through the upper multiplexor. Note that the critical path is through the multiplexor select signal, not through the data inputs. The critical path length is $31\,\mathrm{u_t}$, adding on the register delay, we get the clock period of $[31 + 6]\,\mathrm{u_t} = 37\,\mathrm{u_t}$.

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

☑ The latency is:

Since it takes four cycles to compute a result the latency is $4 \times 37\,\mathrm{u_t} = 148\,\mathrm{u_t}$.

☑ The throughput is:

The unit of work is computing a $v_0^2 + v_0 v_1 + v_1^2$ value. It takes four cycles to do so, so in this case the throughput is one over the latency or $\frac{1}{4\times 37\,\mathrm{u_t}} = \frac{1}{148\,\mathrm{u_t}}$. If $\mathrm{u_t} = 1\,\mathrm{ns}$ then the throughput would be $\frac{1}{148\,\mathrm{ns}}$ or 6756756 calculations per second.

Problem 4: [10 pts] The `mult_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. *Note: In the original exam some object names were different and there was unused code setting high bits of the product to zero.*

☑ 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 wa_re = wa / 2;
      localparam int wp_re = wb + wa_re;

      uwire [wp_re-1:0] prod_lo;


      uwire [wp_re-1:0] prod_hi;


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


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


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


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


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


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

   end
endmodule
```

The solution is on the next page.

```
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 wa_re = wa / 2;
      localparam int wp_re = wb + wa_re;

      // SOLUTION:
      // Compute the width of the product actually needed from the lo and hi modules.
      localparam int wp_lo = min( wp_re, wp );
      localparam int wp_hi = min( wp_re, wp - wa_re );

      // SOLUTION: Possibly use fewer than wp_re bits for the product.
      uwire [wp_lo-1:0] prod_lo;
      uwire [wp_hi-1:0] prod_hi;

      // SOLUTION: Compute how many bits of b are needed in the hi module.
      localparam int wb_hi = min( wb, wp_hi );

      // SOLUTION: Instantiate using the smaller values for the number
      // of bits in the product (wp_lo, wp_hi) and a smaller value for
      // the number of bits in b (wb_hi).
      mult_tree_bfas #(wa_re, wb,    wp_lo) mlo( prod_lo, a[wa_re-1:0],  b );
      mult_tree_bfas #(wa_re, wb_hi, wp_hi) mhi( prod_hi, a[wa-1:wa_re], b[wb_hi-1:0] );

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

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

      // SOLUTION: Use wp_lo and wp_hi in the loop bounds so that
      // there are only as many BFA and BHA modules as needed.
      for ( genvar i=wa_re; i<wp_lo; i++ )
        bfa b(c[i], prod[i], prod_lo[i], prod_hi[i-wa_re], c[i-1] );
      for ( genvar i=wp_lo; i<wp_hi+wa_re; i++ )
        bha b(c[i], prod[i], prod_hi[i-wa_re], c[i-1] );

   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.

... when only a small number will be fabricated, say 1. An FPGA is a moderately priced mass-produced component, so you are sharing the development costs with many other customers. An ASIC is made just for you, so even if you want one you are paying for a whole wafer full of chips, plus the cost of the masks needed for fabrication.

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

... when a large number will be fabricated, say 10,000. An ASIC contains just the logic you need, unlike an FPGA which has customizable logic (such as little look-up tables), customizable connections between the logic, not to mention left-over stuff that you didn't use. Therefore, if you fabricate enough ASIC chips the per-chip cost will be less than an FPGA.

(*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.

Strictly speaking a testbench is not necessary, but practically speaking it is very necessary. A testbench can let the engineer know if the HDL has an error in a short time, perhaps seconds. The testbench might even provide information that can be used to find the flaw in the HDL. If there was no testbench then the component would need synthesized, fabricated, then tested. At best that would take minutes (say, for an FPGA), but for an ASIC it might take weeks. Even if it were just minutes, that would add up until the design were working correctly.

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.

Suppose the testbench written by the good team finds a flaw that the okay team's testbench missed. For the ASIC, that discovered flaw would result in weeks of lost time while the flawed design was fabricated and then tested. It would also cost lots of money. For the FPGA, perhaps only hours of time are wasted by synthesizing and downloading a flawed design. So for that reason it better to use the good team for the ASIC designs.

(*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
```

Short answer: The value of `x` will be updated either way (with or without the non-blocking assignments) in the same time step.

Discussion: Notice that `a` is referenced in Line 1 and written in Line 2. Each is an `always_comb`, and so each line executes whenever its live-in objects change. The live-in objects for Line 1 are `a` and `y`, and the live-in objects for Line 2 are `b` and `c`. If `y` and `b` both change, then Line 1 might be executed before Line 2. But because Line 2 changes `a` Line 1 will execute a second time. Because `a` is assigned using a non-blocking assignment `a` is not actually changed until all the active-region work is complete. But once that happens `a` is changed and that leads to an execution of Line 1.

☑ 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
```

The non-blocking assignments are necessary because each line will execute just once in reaction to the positive edge. Without the non-blocking assignment results would depend on whether Line 1 was executed before or after 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.

Short Answer: Hardware for computing $a * b + c$, because the each operation is performed once. (Assuming a sequential adder and multiplier are not practical.)

Explanation: A sequential design has a lower cost than a combinational design when something in the combinational design can be used multiple times. Expression $v_0^2 + v_0 v_1 + v_1^2$ can be computed by a sequential circuit consisting of one multiplier (used three times) and one adder (used twice). But for $a * b + c$ there would be one multiplier and one adder in both the combinational and sequential designs, so there is no cost benefit for the sequential design.

(*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.

The hardware in `min_b` is simpler so it likely has a shorter critical path. For hardware there is no benefit in ending the loop early.

19 Fall 2019 Solutions

Name Solution

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 30 October 2019 10:30–11:20 CDT

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

Alias That's ··· all.

Good Luck!

Problem 1: [20 pts] Appearing below is one of the solutions to Homework 2, the count leading zeros module.

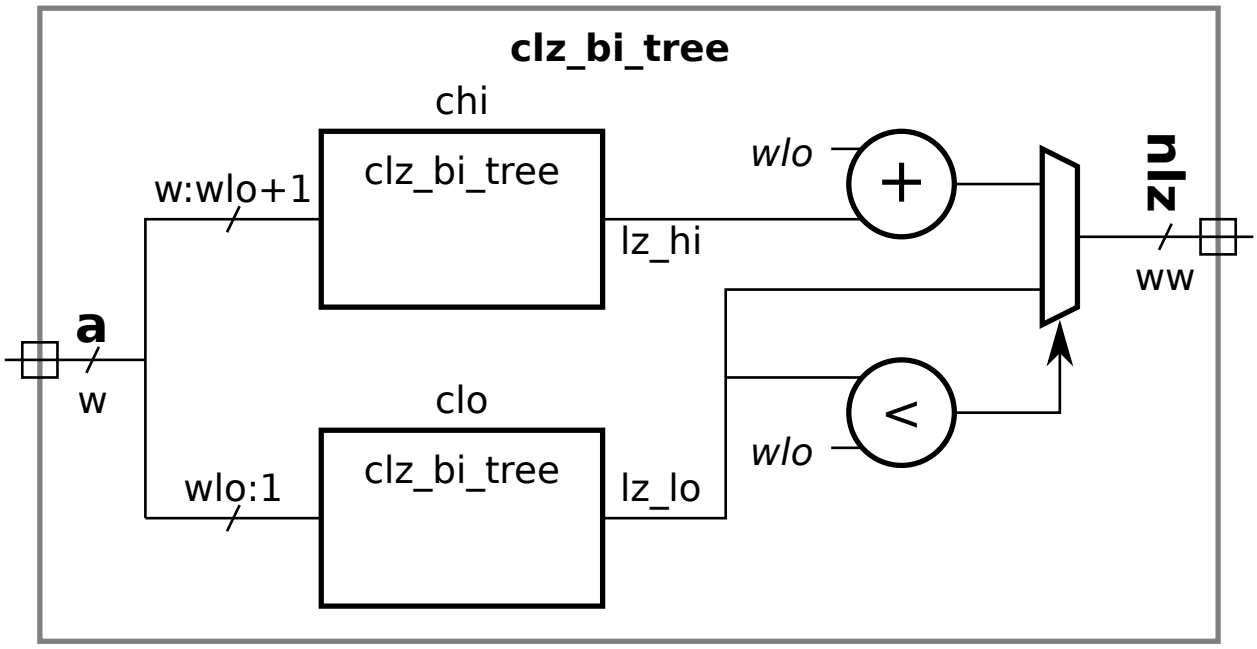
```
module clz_bi_tree #( int w = 19, int ww = $clog2(w+1) )
                    ( output uwire [ww:1] nlz,   input uwire [w:1] a );
   if ( w == 1 ) begin
      assign nlz = ~ a;
   end else begin
      localparam int wlo = w/2,              whi = w - wlo;
      localparam int wwlo = $clog2(wlo+1),   wwhi = $clog2(whi+1);
      uwire [wwlo:1] lz_lo;
      uwire [wwhi:1] lz_hi;
      clz_bi_tree #(wlo) clo( lz_lo, a[wlo:1]   );
      clz_bi_tree #(whi) chi( lz_hi, a[w:wlo+1] );
      assign nlz = lz_lo < wlo ? lz_lo : wlo + lz_hi;
   end
endmodule
```

Show the hardware that will be inferred for the module for $w > 1$. Just show one level, don't show what is inside of `clo` and `chi`.

☑ Show synthesized hardware for one level. ☑ Be sure to show `clo` and `chi` (but not their contents).

☑ Clearly show module input and output ports, ☑ and show bit range in connections.

The solution appears below. Because $w > 1$ the terminal case is not elaborated and so not inferred. Of course, there is no hardware for computing elaboration-time constants such as `wwlo`.



Problem 2: [25 pts] In Homework 2 a clz (count leading zeros) module was constructed recursively by splitting the input bit vector and connecting each half to a smaller instance. The incomplete module below is similar except that the input vector is to be split into thirds and each third connected to a recursive instance. Complete the module.

☑ Complete so that `clz_tri_tree` computes clz.

The solution appears below.

```
module clz_tri_tree
  #( int w = 19, int ww = $clog2(w+1) )
   ( output uwire [ww-1:0] nlz, input uwire [w-1:0] a );

   if ( w == 1 ) begin

      assign nlz = ~ a;

      // SOLUTION: Add a case for w=2 to avoid a zero-bit recursive instance.
   end else if ( w == 2 ) begin

      assign nlz = a[0] ? 0 : a[1] ? 1 : 2;

   end else begin

      // SOLUTION: Divide bits between modules, be sure not to loose any.
      localparam int wlo = w/3;
      localparam int wmi = wlo;
      localparam int whi = w - wlo - wmi;

      localparam int wwlo = $clog2(wlo+1), wwmi = $clog2(wmi+1), wwhi = $clog2(whi+1);
      uwire [wwlo-1:0] lz_lo;
      uwire [wwmi-1:0] lz_mi;
      uwire [wwhi-1:0] lz_hi;

      // SOLUTION: Divide a between modules.
      clz_tri_tree #(wlo) clo( lz_lo, a[ wlo-1   : 0     ] );
      clz_tri_tree #(wmi) cmi( lz_mi, a[ w-whi-1 : wlo   ] );
      clz_tri_tree #(whi) chi( lz_hi, a[ w-1     : w-whi ] );

      // SOLUTION: Combine the results of the three modules.
      assign nlz = lz_lo < wlo  ?  lz_lo        :
                   lz_mi < wmi  ?  wlo + lz_mi  :  wlo + wmi + lz_hi;

   end

endmodule
```

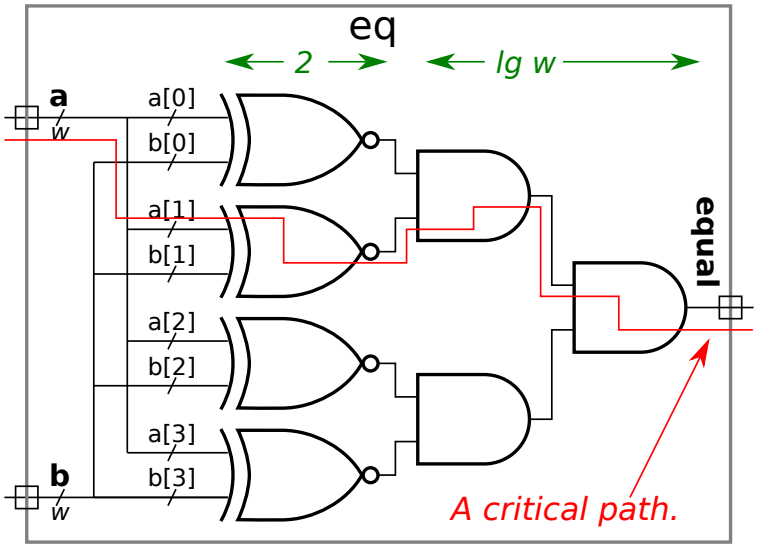
Problem 3: [27 pts] Appearing below are modules that test if two bit vectors are equal in some way.

(*a*) Show the hardware for the module below at the default size using basic gates: AND, OR, XOR, NOTs, and bubbled inputs and outputs. **Do not** use something like `==`.

```
module eq #( int w = 4 )( output uwire equal, input uwire [w-1:0] a, b );
   assign equal = a == b;
endmodule
```

☑ Show hardware using basic gates at default size.

The solution appears below with some colored labels to help with the next subproblem. Note that—never forget that—equality is tested using XNOR (exclusive nor) gates.



(*b*) Show the cost and delay of the module in terms of w (the value of parameter `w`) using the simple model.

☑ In terms of w: ☑ Cost and ☑ Delay.

The cost is $[3w + w - 1]\,\mathrm{u_c} = [4w - 1]\,\mathrm{u_c}$. The $3w$ term is for the XNOR gates and $w - 1$ term is for the big AND gate. (In the solution above three 2-input AND gates are shown rather than one 4-input AND gate.) The delay is $[2 + \lceil \lg w \rceil]\ \mathrm{u_t}$, the 2 term is for an XNOR gate and the $\lg w$ term is for a path through the big AND gate.

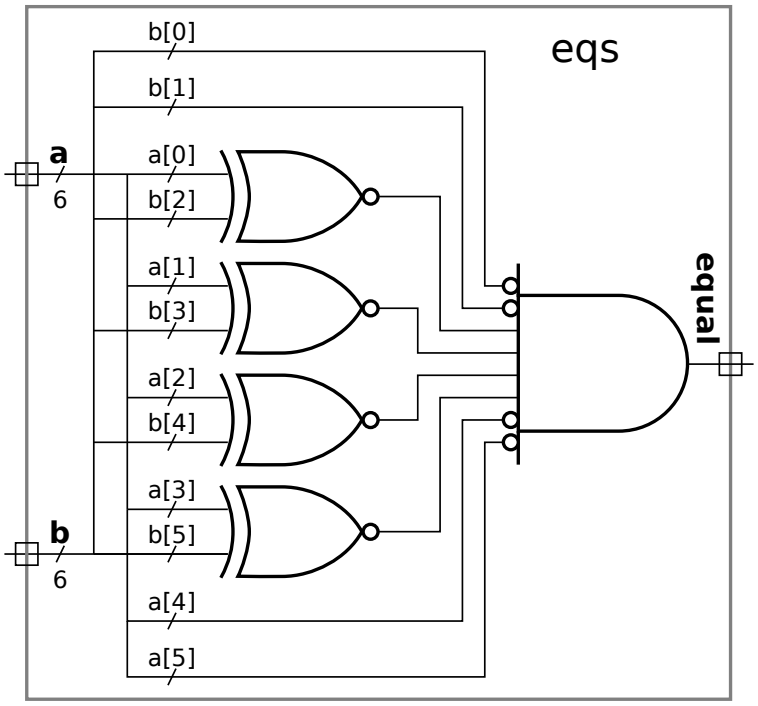
To compute delay a critical path is needed. A critical path for the equality unit is shown above in red, starting at `a[1]`. Because of symmetry in the equality unit the critical path could have started at any input bit. The path through an XNOR is two gates, and a path through the big AND is $\lceil \lg w \rceil$ gates.

(*c*) The module below also tests equality but it does so after shifting the first operand. Show the hardware in terms of basic gates after optimization.

```
module eqs #( int w = 6, int s = 2 )( output uwire equal, input uwire [w-1:0] a, b );
   localparam logic [w+s-1:0] zero = 0;
   assign equal = zero + ( a << s ) == b;
endmodule
```

☑ Show hardware at default size after optimization.

The solution appears below. Because the shift is by a constant amount no shifter is needed, instead the bit positions are adjusted (which is why, for example, `b[2]` is compared to `a[0]`). Because we are adding zero no adder is needed. Because of the shift the low bits of `b` and the high bits of `a` are compared to zero.



(*d*) The module below performs a different operation than the one above. Explain the difference and show an example.

```
module eqt #( int w = 6, int s = 2 ) ( output uwire equal, input uwire [w-1:0] a, b );

   assign equal = ( a << s ) == b;

endmodule
```

☑ Difference between operation `eqs` and `eqt`.

☑ Show a value for `a` and `b` for which the output of `eqs` and `eqt` are different.

In module `eqs` the s MSB are compared to zero, whereas in `eqt` the s MSB are ignored. For example, consider $w = 6$ and $s = 2$, and for $a = 10\,1111_2$ and $b = 11\,1100_2$. Module `eqs` finds them not equal (because eight-bit quantities $1011\,1100_2 \neq 0011\,1100_2$) but `eqt` finds them equal (because six-bit quantities $11\,1100_2 = 11\,1100_2$).

Problem 4: [28 pts] Answer each question below.

(*a*) Appearing below is synthesis data taken from the solution to Homework 2. The `Delay Target` column shows the maximum delay constraint given to the synthesis program.

Module Name	Area	Delay Actual	Delay Target
clz_w32	26290	3.110	10.000 ns
clz_tree_w32	21706	1.425	10.000 ns
clz_w32_1	36476	1.007	0.100 ns
clz_tree_w32_5	37356	0.577	0.100 ns

☑ In general, which result should be used if the only goal were to minimize area,
the results for the ☑ 10.0 ns *Target* or for the ◯ 0.1 ns *Target* ? ☑ Explain.

When the delay target is large the synthesis program is freer to minimize area (cost). It can try different cost-reducing optimizations without having them being rejected because they result in higher delay (as long as that delay is below the delay target).

☑ In general, which result should be used if the only goal were to minimize delay,
the results for the ◯ 10.0 ns *Target* or for the ☑ 0.1 ns *Target* ? ☑ Explain.

The synthesis program first tries to meet the delay target, then reduces cost. If the delay target is very low it will devote all of its effort to reducing delay.

(*b*) Provide w-bit declarations requested below.

```
uwire [ 0 : w-1 ] bit_zero_is_msb;        // SOLUTION
uwire [ w-1 : 0 ] bit_zero_is_lsb;        // SOLUTION
uwire [ w/2 : -w/2 ] bit_zero_is_middle;  // SOLUTION.
```

(*c*) The module fragment below starts with six declarations (the object names starting with `r`), each providing a value (either `a+b` or `x+y`). Some of those declarations will result in compile errors. Identify them and explain the problem. If possible fix the problem without changing the object kind (localparam, uwire, var).

```
module my_mod
  #( int w = 10, int x = 11, int y = 12 )
   ( input uwire [w:1] a, b );

   localparam logic [w:1] r1p = a + b; // SOL: Can't fix, a + b not constant.

   localparam logic [w:1] r2p = x + y; // SOL: Okay.

   uwire [w:1] r1w = a + b;  // SOL: Okay.

   uwire [w:1] r2w = x + y;  // SOL: Okay.

   logic [w:1] r1l = a + b;  // SOL: Wrong, can't continuously assign var type.

   logic [w:1] r2l = x + y;  // SOL: Wrong, can't continuously assign var type.

   // SOLUTION: Fixes:
   logic [w:1] r1l, r2l;
   always_comb begin r1l = a + b;  r2l = x + y;  end

   // The following is not wrong, but it's longer than the original.
   uwire [w:1] r12, r2w;
   assign r1w = a + b;
   assign r2w = x + y;
```

☑ Indicate which ones are wrong and ☑ the reason that they are wrong.

☑ Indicate which can't be fixed and ☑ and explain why not.

The value assigned to a `localparam` must be an elaboration-time constant. That's true for `x+y` because they are parameters, but it's not true for `a+b` because `a` and `b` are module inputs and so could never be elaboration-time constants.

The assignments to `r1w` and `r2w` are fine. SystemVerilog allows a net (including `uwire`) declaration to include a continuous assignment.

The assignments to `r1l` and `r2l` are wrong because var objects can only be assigned in procedural code. That's easy to fix by providing an `always` block, which is shown above.

(Note that a declaration like `logic [w:1] v;` is shorthand for `var logic [w:1] v;` and a declaration like `uwire [w:1] u;` is shorthand for `uwire logic [w:1] u;`.)

Other than for `r1p` the size, type, and kind of `a`, `b`, `x`, and `y` are not a problem. The sum `x+y` is a 32-bit 2-state integer. It's not an error to assign that to a `w`-bit four state type. Also note that the data type for all of the `r[12]pwl` objects are logic. (Note that `r[12]pwl` is an ad-hoc regular expression matching the objects being assigned above. Regular expressions are something you should know in general, but not for this course.)

Grading Note: Students had more difficulty with this problem than I expected. As I pointed out in class, if you don't understand the different object kinds (net, var, param) and how they should be used you'll waste lots of time blindly changing things until the error messages go away.

(*d*) Explain what `$realtobits` does, and what hardware will be synthesized for it, if any.

```
always_comb begin

   x = $realtobits(r);

end
```

☑ Purpose of `realtobits`.

The `realbits` system task is used to move a set of bits from an object declared `real` to one declared as some kind of integer (say, `logic [63:0]`). The bits are moved unchanged. If, instead the assignment were `x=r;` the simulator would convert the `real` value in `r` to an integer.

☑ Synthesized hardware.

None. If we were to draw a diagram, there would be a wire labeled with both `x` and `r`.

Name Solution

Digital Design using HDLs
LSU EE 4755
Final Examination

Friday, 13 December 2019 10:00-12:00 CST

Problem 1		(30 pts)
Problem 2		(25 pts)
Problem 3		(20 pts)
Problem 4		(25 pts)
Exam Total		(100 pts)

Alias ☣ It Begins

Good Luck!

Problem 1: [30 pts] Appearing below is the solution to Homework 6, the accumulation module. The next page shows the pipelined adder and `st_occ`, which is some of the inferred hardware. Show the rest of the inferred hardware after some optimization. Leave the pipelined adder as a box.

```
module add_accum #( int w = 21, n_stages = 3 )
   ( output logic [w-1:0] sum,   output logic sum_valid,
     input uwire [w-1:0] ai,     input uwire ai_v, reset, clk );

   logic [n_stages-1:0] st_occ;
   assign sum_valid = !st_occ;
   uwire aout_v = st_occ[n_stages-1];

   uwire [w-1:0] aout;
   uwire [w-1:0] a0 = ai_v   ? ai   : sum;
   uwire [w-1:0] a1 = aout_v ? aout : sum;

   add_pipe #(w,n_stages) add_p0( aout, a0, a1, clk );

   logic sum_occupied;
   uwire [1:0] n_values = ai_v + sum_occupied + aout_v;
   uwire saa = n_values >= 2;  // Start an addition.
   uwire write_sum = !sum_occupied && n_values == 1;

   always_ff @( posedge clk ) if ( reset ) begin
      sum <= 0;
      sum_occupied <= 0;
      st_occ <= 0;
   end else begin
      if ( write_sum ) sum <= aout_v ? aout : ai;
      sum_occupied <= n_values[0];
      st_occ <= { st_occ[n_stages-1:0], saa };
   end
endmodule
```

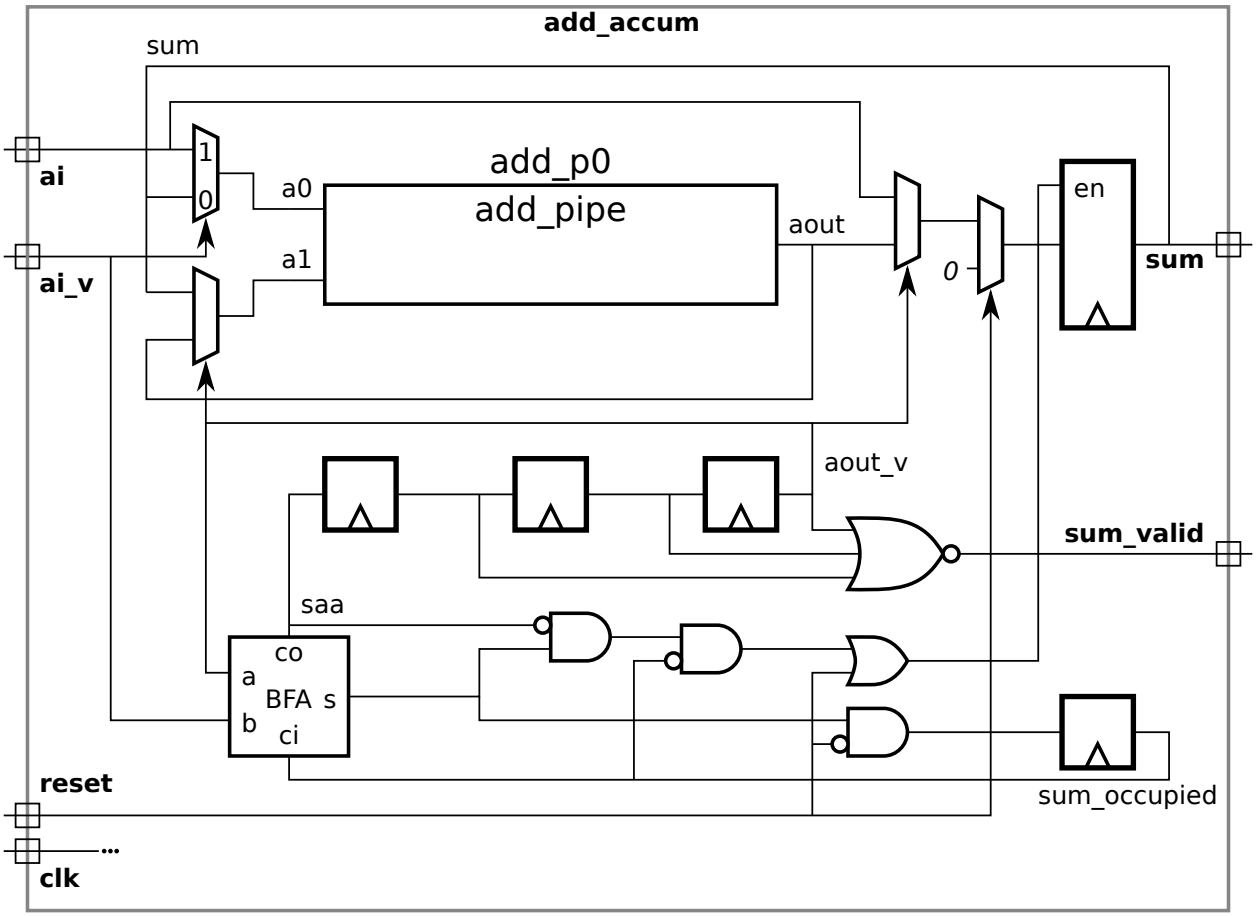
☑ Show inferred hardware after some optimization, but ☑ leave `add_pipe` as a box.

☑ Show logic associated with `n_values` as basic gates and a single BFA, **do not show adders and do not show comparison units**.

☑ Clearly show all input and output ports, do not confuse parameters with ports.

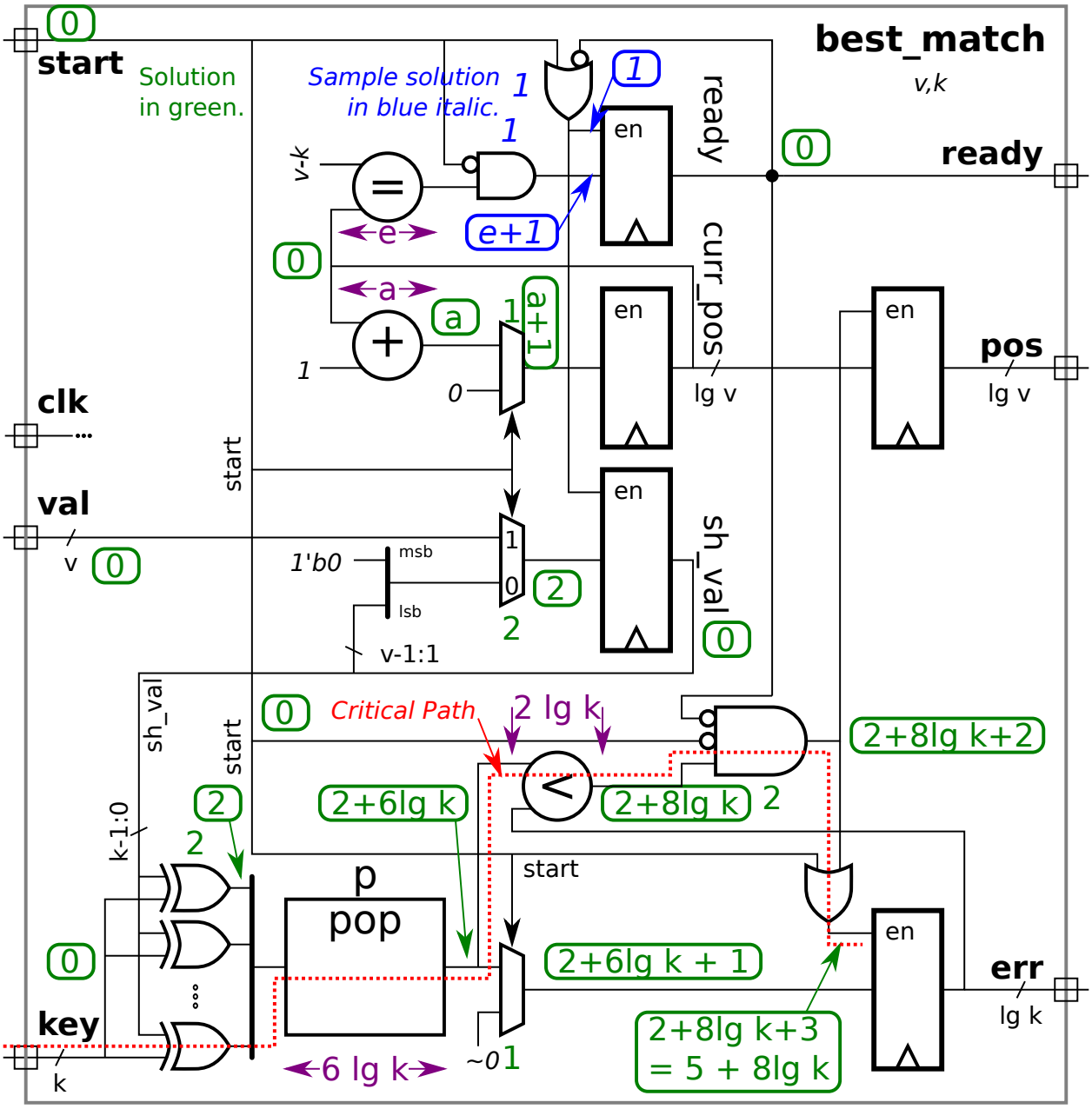
☑ Avoid effortlessly optimized hardware, such as gates with constant inputs.

Solution appears below.



Problem 2: [25 pts] Appearing below is hardware from the solution to Homework 5, Problem 2. The parameter names have been shortened, such as changing `wv` to `v` and using `lg v` for `wvb`. The diagram shows the delay through some of the modules, including the `pop` module. Treat `e` and `a` (delays for $\boxed{=}$ and $\boxed{+}$) as given constants for the first part.

(*a*) Based on the provided delays and using the simple model for others, compute the arrival time (delay) of signals at each register input. That's two inputs for each of five registers. The solution for `ready` is shown in blue, so only four registers remain. Also, highlight the/a critical path to the `err` register.



☑ Show the arrival time of the enable and data signal at each register input and ☑ Highlight a critical path to `err` with a squiggly line.

☑ Take into account constant inputs when computing delays.

Solution to part a: Arrival times at register inputs, as well as the delay at other points, shown in green. The critical path appears as a red dashed (not squiggly) line.

Note that the delay of a mux with a constant data input is 1, which applies to two of the multiplexors in the diagram.

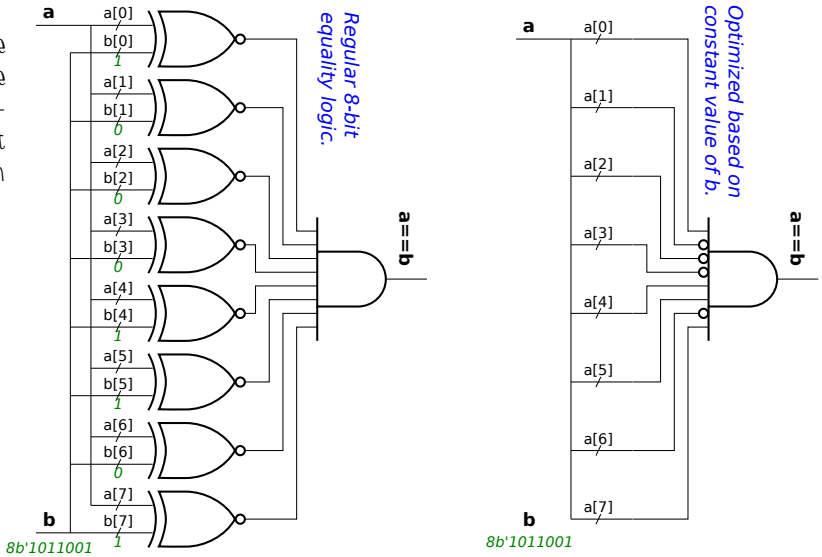
The critical path in the solution starts at `key`. It would also be correct to start the critical path at `sh_val` (and passing through the XOR gates).

A common mistake was to show the critical path passing *through* a register. Paths start at register outputs and end at register inputs.

(*b*) The equality module is shown with a delay of `e`. Show the hardware for that module and compute the cost and delay using the simple model. Take into account the width of the inputs and the fact that one input is a constant.

☑ Sketch hardware for equality module for $\lg v = 8$ and $v - k = 1011\,0001_2$, and ☑ taking into account the constant input.

Because of the constant input each XNOR gate is optimized to either a NOT gate (where the constant bit is 0) or just wire (where the constant bit is 1). So the equality module is just a $\lceil \lg v \rceil$-input AND gate. See the illustration to the right.



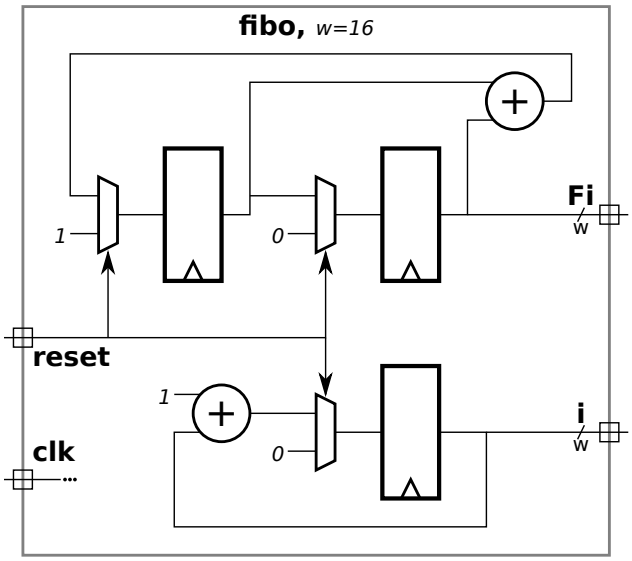
☑ Show the cost of the hardware for the equality module above based on the simple model in terms of $\lg v$. ☑ Don't forget to take the constant input into account.

The hardware consists of a single $\lg v$-input AND gate. Its cost is $\lceil \lg v - 1 \rceil\ \mathrm{u_c}$.

☑ Show the delay of the hardware based on the simple model in terms of $\lg v$. ☑ Don't forget to take the constant input into account.

The delay of an $\lg v$-input AND gate is $\lceil \lg \lceil \lg v \rceil \rceil\ \mathrm{u_t}$.

Problem 3: [20 pts] The hardware illustrated to the right emits a famous integer sequence. Write a synthesizable Verilog description of the hardware.



☑ Complete the module, ☑ be sure that it is synthesizable.

☑ Use non-blocking assignments carefully.

☑ Be sure to include all ☑ input and output ports and ☑ parameters.

☑ Make sure that all objects have the appropriate widths.

Solution appears below. The warning about non-blocking assignments needed to be heeded in the solution below so that the value of `Fi` used when updating `Fi_next` would be based on the old value of `Fi`.

```
// SOLUTION

module fibo
  #( int w = 16 )
   ( output logic [w-1:0] Fi, i,
     input uwire reset, clk );

   logic [w-1:0] Fi_next;

   always_ff @( posedge clk ) if ( reset ) begin

      Fi <= 0;
      Fi_next <= 1;
      i <= 0;

   end else begin

      Fi <= Fi_next;
      // Note: The non-blocking assignment above insures that the Fi +
      // Fi_next expression below is computed using the old value of Fi.
      Fi_next <= Fi + Fi_next;
      i <= i + 1;

   end

endmodule
```

Problem 4: [25 pts] Answer each question below.

(*a*) Appearing below are synthesis script results for the pipelined integer adder from Homework 6. That adder computes a w-bit integer sum using an n-stage pipeline in which each stage computes $\lceil w/n \rceil$ bits of the sum, starting with the $\lceil w/n \rceil$ least-significant bits in the first stage.

All syntheses are of a $w = 24$-bit adder, versions with $n = 1$, 2, 3, 4, and 6 stages are synthesized. The delay target is set to an easy 90 ns.

Module Name	Area	Delay Actual	Delay Target
add_pipe_w24_n_stages1	29928	10.174	90.000 ns
add_pipe_w24_n_stages2	47043	5.428	90.000 ns
add_pipe_w24_n_stages3	64159	3.701	90.000 ns
add_pipe_w24_n_stages4	81275	2.837	90.000 ns
add_pipe_w24_n_stages6	115506	1.973	90.000 ns

☑ Based on this data provide the ☑ latency and ☑ throughput for the three-stage adder. Be sure to ☑ use appropriate units for the throughput.

The latency is $3 \times 3.701 = 11.103$ ns. The throughput is $\frac{1 \text{ addition}}{3.701 \text{ ns}} = 270.2 \times 10^6$ additions per second.

☑ Note that the area (cost) increases with the number of stages. Based on the description above what is the main contributor to the increase in cost?

The main contributor to cost are the registers. Each stage requires three registers, two for the source operands and one for the sum.

(*b*) The two modules below appear to be similar.

```
module plan_I(output logic [7:0] e, input logic [7:0] a,b);
   logic [7:0] c;
   always_comb begin
      c = a + b;
      e = c + a;
   end
endmodule

module plan_II(output logic [7:0] e, input logic [7:0] a,b);
   logic [7:0] c;
   always_comb e = c + a;
   always_comb c = a + b;
endmodule
```

☑ For which module will the simulator perform unnecessary addition? ☑ Explain.

Module `plan_II` will require extra work because when `a` changes the `e = c + a` can be executed twice, first for the change in `a` then for the change in `c` due to execution of the `c = a + b`.

☑ Is the result computed by the two modules different or the same? ☑ Explain.

The result at the end of a time step is the same. However `plan_II` can leave `e` in different value than `plan_I` during a time step (before `e = c+a` executes a second time, as described above).

(*c*) What value will y have at the end of the initial block?

```
module S;
   logic [15:0] a,b,y;
   initial begin
      a = 1;         // SOLUTION information in comments below.
      b = 100;       // Value of b set to 100.
      b <= 10;       // Update event b = 10 is put in NBA region. b still 100.
      y = 0;         // Value of y set to zero.
      y <= a + b;    // a+b computed: 1 + 100 = 101. Update event y=101 put in NBA region.
      y = 999;       // Value of y set to 999.

      #1;            // After #1 reached NBA events executed:
                     //   b set to 10
                     //   y set to 101. (a+b computed above using older b).
                     // The lines below have no impact on y.
      a = 2;
      b <= 20;
      #200;
      // Show value of y at this point in execution.
      // SOLUTION: y is 101.
   end
endmodule
```

☑ Value of y at end of block is:

Short answer: y=101.

Explanation: y is assigned three times. For the blocking assignments, y=0 and y=999, the value is written when the respective statement is executed. For the non-blocking assignment, y<=a+b, the value a+b is computed when the statement is reached, but the result is not assigned until the simulator reaches the timeslot $t = 0$ NBA region. The same holds for non-blocking assignment b<=10. For that reason a+b is computed using a=1 and b=100. See the comments in the code above.

(*d*) Consider the declarations below.

```
module types;
   int en;
   logic [31:0] lo;
   bit [31:0] b;
   uwire [31:0] u = 33;
   localparam int p = 22;
endmodule
```

☑ Object `u` has the same data type as one of the other objects. Which is it?

It has the same data type as `lo`. The data type is `logic`. Declarator `uwire` is an object kind, not a data type. For `uwire` kinds the default data type is `logic`.

☑ What is the difference between `lo` and `b` (`logic` and `bit`)?

Both are used to represent one bit. Type `bit` has two states, `0` and `1`, while `logic` has four states, `0`, `1`, `x`, and `z`. The `var logic` objects have value `x` until they are assigned a value. In net `logic` objects (such as something declared `wire`) the value is `x` when there is more than one driver and at least one is driving a `0` and at least another is driving a `1`. A net object with zero drivers has value `z`. It is also possible to specify these values in literals, such as `1'bz`.

☑ Notice that `u` is assigned a value. What is it about object `lo` that makes it illegal to assign a value in its declaration?

Object `lo` is a variable type, and so it can only be assigned in procedural code.

☑ Add correct code to assign value 44 to `lo`.

The solution appears below. If the goal is to assign an initial value then an `initial` block is appropriate.

An `assign lo=44;` **is wrong** because `lo` is a var kind and continuous assignments (including `assign`) should only be performed on net kinds, such as `uwire`.

```
// SOLUTION
initial lo = 44;
```

20 Fall 2018 Solutions

Name Solution

Digital Design using HDLs
LSU EE 4755
Midterm Examination

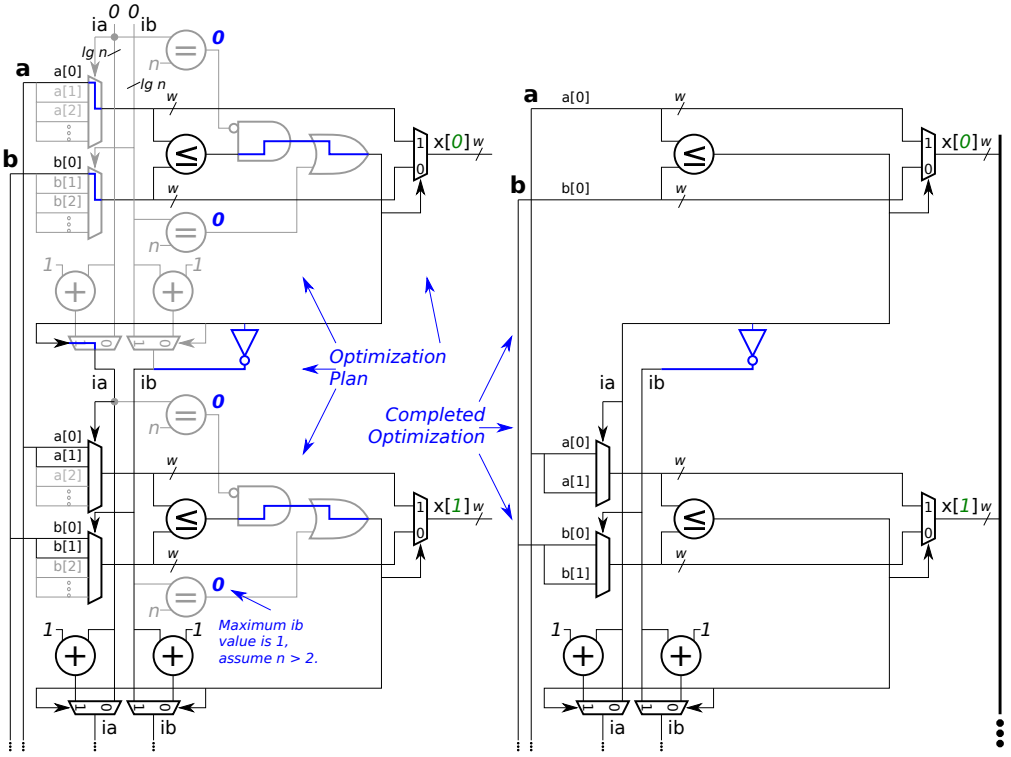
Friday, 26 October 2018 9:30–10:20 CDT

Problem 1		(22 pts)
Problem 2		(20 pts)
Problem 3		(23 pts)
Problem 4		(10 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias Blockchain Apocalypse

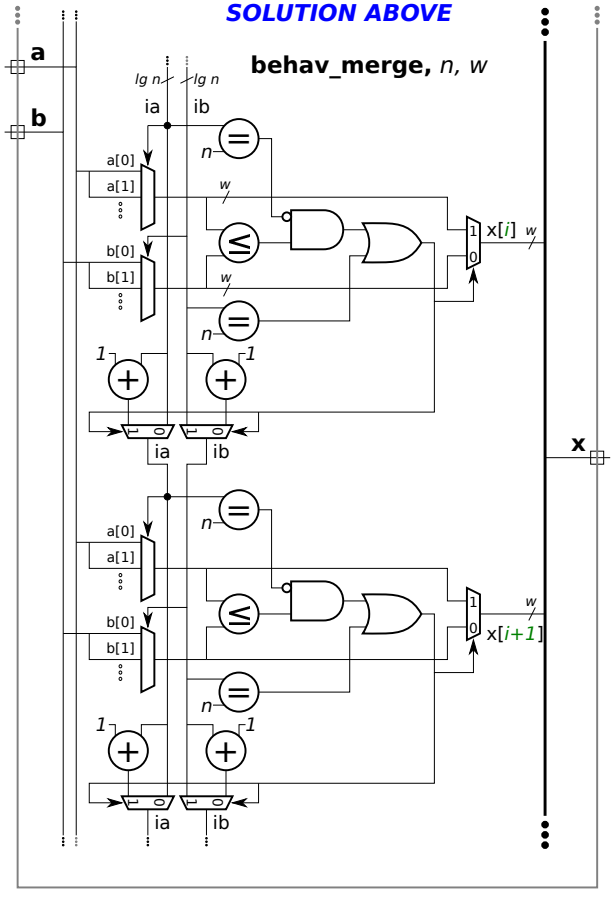
Good Luck!

Problem 1: [22 pts] The illustration below shows some of the inferred hardware for the `behav_merge` module from the solution to Homework 6. The hardware that's shown is for typical iterations `i` and `i+1`. Show the hardware for iterations `i=0` and `i=1` with optimizations applied.

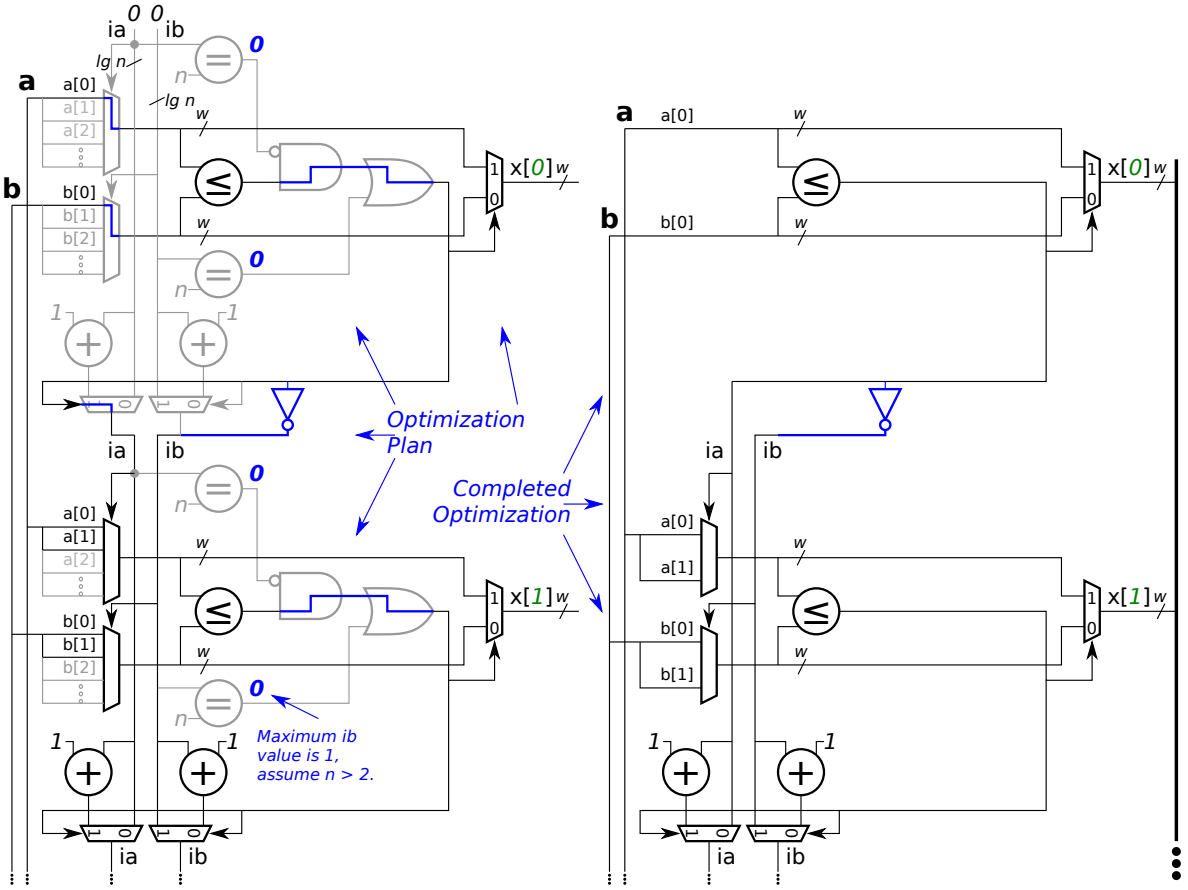


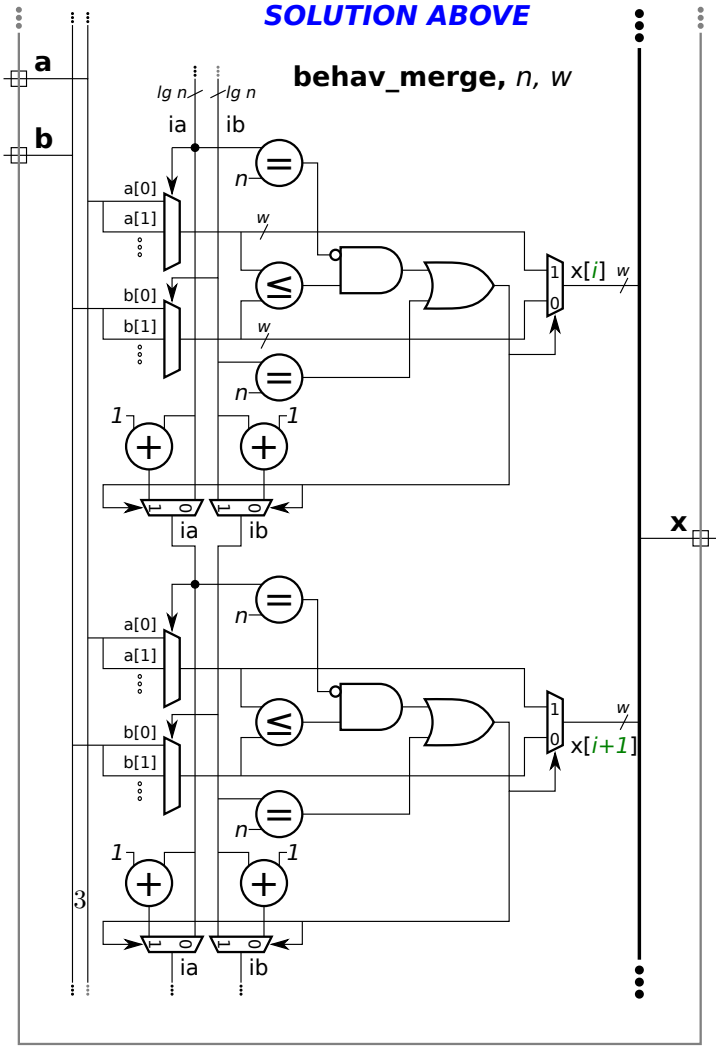
- ☑ Show hardware for iterations `i=0` and `i=1`.
- ☑ Also show hardware for code before `for` loop.
- ☑ Optimize hardware. Take into account possible values of `ia` and `ib`.

See the next page for a discussion of the solution.




```
module behav_merge
 #( int n = 4, int w = 8 )
  ( output logic [w-1:0] x[2*n],
    input uwire [w-1:0] a[n], b[n] );

  logic [$clog2(n+1)-1:0] ia, ib;
  always_comb begin
   ia = 0; ib = 0;
   for ( int i = 0;  i < 2*n;  i++ )
     if ( ib==n || ia!=n && a[ia]<=b[ib] )
       x[i] = a[ia++]; else x[i] = b[ib++];
  end

endmodule
```



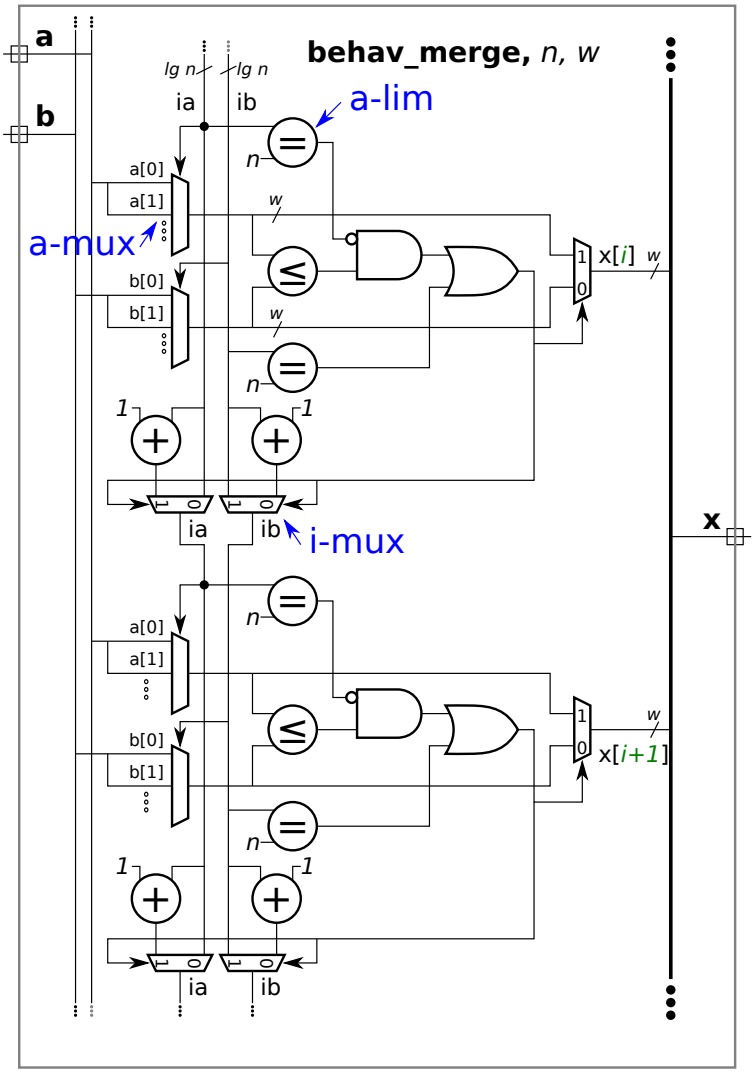


Solution appears above.

Explanation: To the left hardware that's no longer needed appears in gray. On the right the diagram is redrawn with the unneeded hardware removed. The initial zero values for `ia` and `ib` make the `a[ia]` and `b[ib]` muxen unnecessary. For `i=1` those muxen each have two inputs since the possible values for `ia` and `ib` are either 0 or 1.

A value for `n` was not given, but it is reasonable to assume that it is greater than 1. In that case the output of all of the `=n` logic blocks will be false. This makes the AND and OR gates unnecessary, and so the output of the $\leq$ block can connect directly to the `x` mux and to the logic generating the new `ia` and `ib` signals. For `i=0` the `ia` signal is equal to the output fo the $\leq$ block (that is, a 0 or 1), for `ib` (or to be exact, the least significant bit of `ib`) the output is inverted.

Problem 2: [20 pts] Appearing once again is part of the Homework 6 solution, this time with items labeled in blue. Show the cost and delay of these, as requested below. See the previous problem for the Verilog description. The phrase *most expensive* means for the value of i for which the device needs all inputs, even after optimization. For the mux, show the cost and delay for the tree implementation.



☑ Cost of most expensive `a-mux` in terms of n and w.

The mux has n inputs (the size of the `a` array) of w bits each. The cost is $\boxed{3w(n-1)\,\mathrm{u_c}}$.

☑ Delay of most expensive `a-mux` in terms of n and w.

The delay is $\boxed{2\lceil \lg n \rceil\,\mathrm{u_t}}$.

☑ Cost of most expensive `i-mux` in terms of n and w.

The `i-mux` has just two inputs of $\lg n$ bits each according to the diagram. According to the Verilog the number of bits is $\lceil \lg(n+1) \rceil$. The cost is $\boxed{3\lceil \lg(n+1) \rceil\,\mathrm{u_c}}$. Note: $3 \lg n$ would get full credit.

☑ Delay of most expensive `i-mux` in terms of n and w.

Since there are only two inputs the delay is $\boxed{2\,\mathrm{u_t}}$.

☑ Cost of most expensive `a-lim` in terms of n and w ☑ after optimizing for constant inputs.

Input `n` to the equality unit is a constant, so the first column of XOR gates is replaced by NOT gates (in positions where the `n` bit is 0). So the equality module is just the NOT gates plus n-input AND gate, the cost of which is $\boxed{(\lceil \lg(n+1) \rceil - 1)\,\mathrm{u_c}}$.

☑ Delay of most expensive `a-lim` in terms of n and w ☑ after optimizing for constant inputs.

The delay is $\boxed{\lceil \lg \lceil \lg(n+1) \rceil \rceil\,\mathrm{u_t}}$.

Problem 3: [23 pts] Output `lt` of module `comp`, below, should be 1 iff `a` is strictly less than `b`, and `eq` should be 1 iff `a==b`. Both `a` and `b` are unsigned integers. The module recursively instantiates two instances of itself, one is supposed to compare the low bits of the inputs, the other compares the high bits. Complete the module so that it works for any positive `w`.

☑ Complete the module, don't miss the ☑ `FILL IN` items.

☑ Make sure that it works for odd and even values of `w`.

```
module comp
  #( int w = 8 )
   ( output uwire lt, eq, input uwire [w-1:0] a, b );

   if ( w == 1 ) begin   // Terminating Case Condition          <----  ☑  FILL IN

      assign lt = !a && b;
      assign eq = a == b;

   end else begin

      uwire llo, lhi, elo, ehi;
      localparam int wlo = w / 2;
      localparam int whi = w - wlo;

      // Instantiate two comp modules,  connect each to about half the inputs.
      //
      //     ----                  --------------      -------------- <--  ☑  FILL IN
      comp #( wlo ) clo( llo, elo, a[ wlo - 1 : 0   ], b[ wlo - 1 : 0   ] );

      comp #( whi ) chi( lhi, ehi, a[ w - 1 : wlo   ], b[ w - 1 : wlo ] );


      assign lt =  lhi  ||  ehi && llo ;                        <----  ☑  FILL IN

      assign eq =  elo && ehi ;                                 <----  ☑  FILL IN

   end
endmodule
```

Solution appears above, in blue, of course.

Explanation: The termination condition must be set to `w==1` because the expression `!a && b` would not set `lt` to the correct value if `a` and `b` were more than one-bit quantities. Setting `w==0` would make no sense from a functionality viewpoint.

The non-terminating case splits the bits making up the two inputs, `a` and `b`, between the two recursive instantiations, `clo` and `chi`, in a straightforward manner. Notice that `wlo` and `whi` are computed separately (rather than using `w/2` for both) to handle odd values of `w`.

Finally, outputs `lt` and `eq` must be computed from the outputs of `clo` and `chi`. Equal is the easier one. Input `a` equals `b` if their low bits and high bits are both equal. That is, `eq = elo && ehi`. For `lt` to be true either `lhi` is true (meaning that `a<b` looking only at the most-significant bits) or if the high bits are equal, `ehi`, and `llo` is true.

Problem 4: [10 pts] The output of `plus_amt`, `x`, is to be set to `b + amt`. Input `b` and output `x` are expected to be in IEEE 754 double FP format (the same format as type real). (Note: the port declarations are not to be modified in the problems below.) Several variations on the module appear below. *Hint: Solution to this problem require the correct use of* `realtobits` *and/or* `bitstoreal`. *Grading Note: The bonus problem was not on the original exam.*

(*a*) The module below does not compute the correct result. Fix the module by modifying the `always_comb` block. The module does not need to be synthesizable.

☑ Fix so that `x` is assigned the correct result, `amt` plus value of `b`.

Two solutions appears below. In the original code one operand was an integer type, `b`, the other was a real type, `amt`. In such cases the simulator would add code to convert `b` from an integer to a real. The simulator has no way of knowing that `b` already holds a value in the real format. Once `b` is converted the value is ruined. Two solutions are shown below. In the first solution two new real variables are declared, one for `b` and one for `x`. The re-interpretation system task `$bitstoreal` is used to move the value in `b` to `b_real` without changing the bits. In the statement `x_real = b_real + amt;` all three variables are real, so the simulator does not do any type conversion. Finally, `x` is assigned from `x_real` using the re-interpretation system task `$realtobits`. The second solution uses these system tasks the same way but without the intermediate variables.

```
module plus_amt
  #( real amt = 1.5 )
   ( output logic [63:0] x, input uwire [63:0] b );
   // Both x and b are IEEE 754 doubles (reals).

   real b_real, x_real;  // Declare vars to hold real values.

   always_comb begin
      b_real = $bitstoreal(b); // Re-interpret b as a real.
      x_real = b_real + amt;   // Note: Both operands are FP, so do FP add.
      x = $realtobits(x_real); // Re-interpret x_real as logic vector (int).
   end

endmodule

module plus_amt // Compact solution, avoids need for new variables.
  #( real amt = 1.5 ) ( output logic [63:0] x, input uwire [63:0] b );

   always_comb x = $realtobits( $bitstoreal(b) + amt );

endmodule
```

(*b*) [0 pts] Bonus Problem Complete the module below so that it uses the `CW_fp_add` module to do the addition. The parameters to `CW_fp_add` are already correct, just connect the inputs and outputs.

☐ Complete so that it computes the correct result.

Problem 5: [25 pts] Show the hardware that will be inferred for the Verilog code below.

☑ Clearly show module ports.

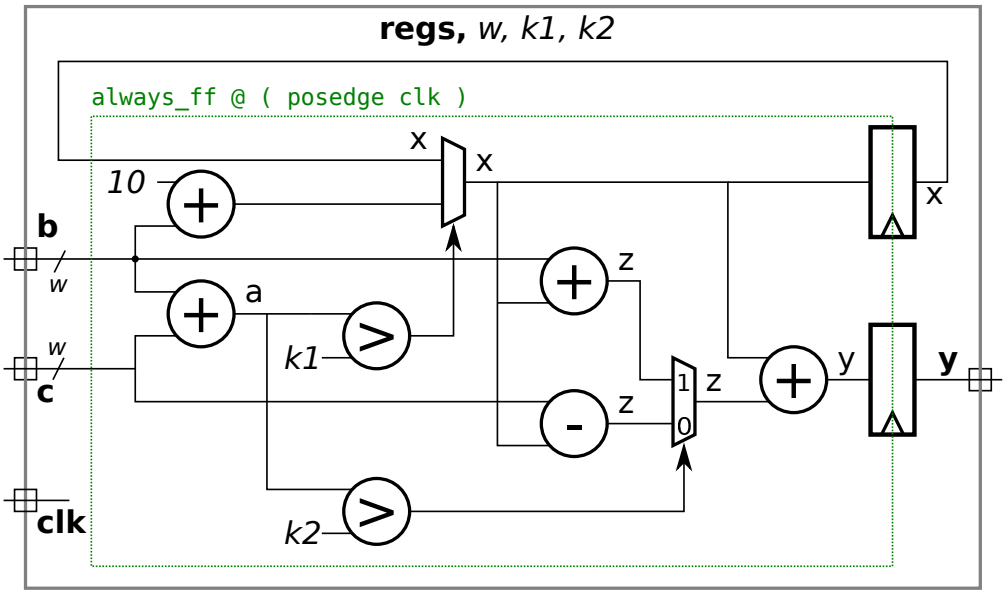
☑ Show inferred hardware. Don't optimize.

☑ Pay close attention to what is and is not inferred as a register.

```
module regs #( int w = 10, int k1 = 20, int k2 = 30 )
   ( output logic [w-1:0] y,
     input logic [w-1:0] b, c,
     input uwire clk );

   logic [w-1:0] a, x, z;

   always_ff @( posedge clk ) begin

      a = b + c;
      if ( a > k1 ) x = b + 10;
      if ( a > k2 ) z = b + x; else z = c - x;
      y = x + z;

   end

endmodule
```

Solution appears below.

Explanation: The area corresponding to the `always_ff` block is outlined in a green dashed line. Registers are shown on the right-hand boundary because the value that gets clocked into a register is the value present when control reaches the end of the block (the `end` statement above). Four values are assigned within the block, `a`, `x`, `z`, and `y`. Registers are inferred only for those variables that are a *live out* object of the block. That is true for `y` since it's also a module output and so its value is needed outside the block. In contrast, the value of `a` that is computed in the block is not used again after the `end` is reached. (When the block is re-entered a new value of `a` will be computed.) The same is true for `z`. But the value of `x` may be used after `end` is reached. That happens when the block is re-entered and $a \leq k_1$, in which case `x` is set to the previous value of `x` (the one in the register) rather than `b+10`.



Name Solution

Digital Design using HDLs
LSU EE 4755
Final Examination

Wednesday, 5 December 2018 15:00-17:00 CST

Problem 1		(20 pts)
Problem 2		(25 pts)
Problem 3		(20 pts)
Problem 4		(10 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias In Color

Good Luck!

Problem 1: [20 pts] Appearing to the right is the hardware inferred for the Homework 7 Problem 2 module, the fast sequential multiplier which skipped over zeros in the multiplicand.

(*a*) Notice that some hardware is circled in blue. Optimize that hardware and show the cost of the optimized hardware. The optimized hardware should generate signals `sv_prod` and `oa_new`. If possible, replace the multiplexors with simpler gates.

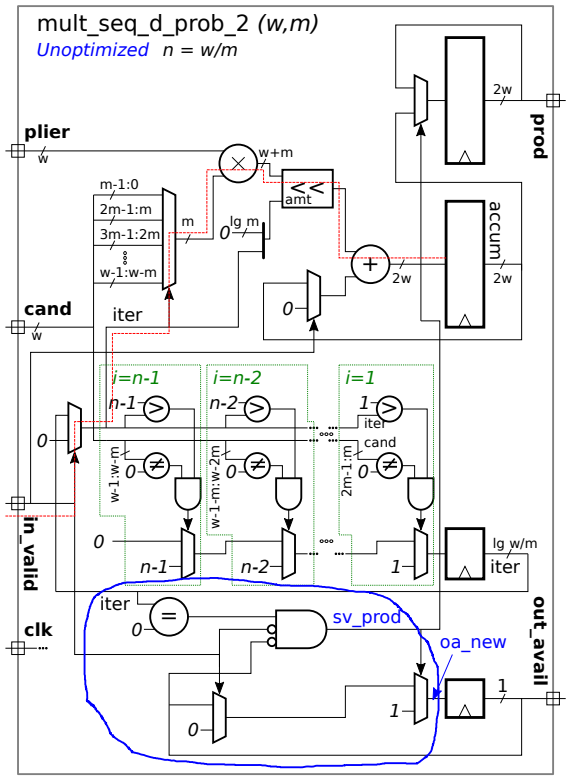
☑ Show optimized hardware.

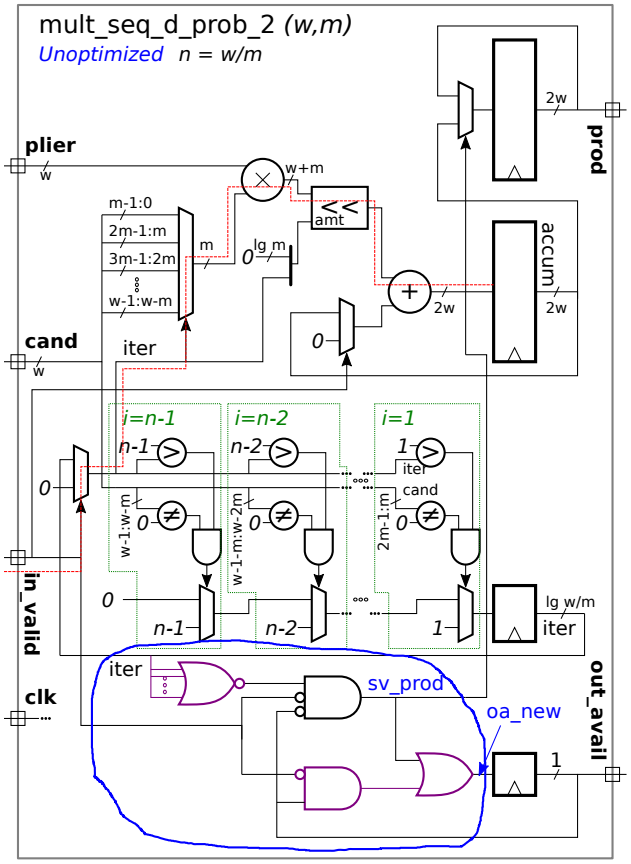
Solution appears to the lower-right in purple.

☑ Cost of optimized hardware:

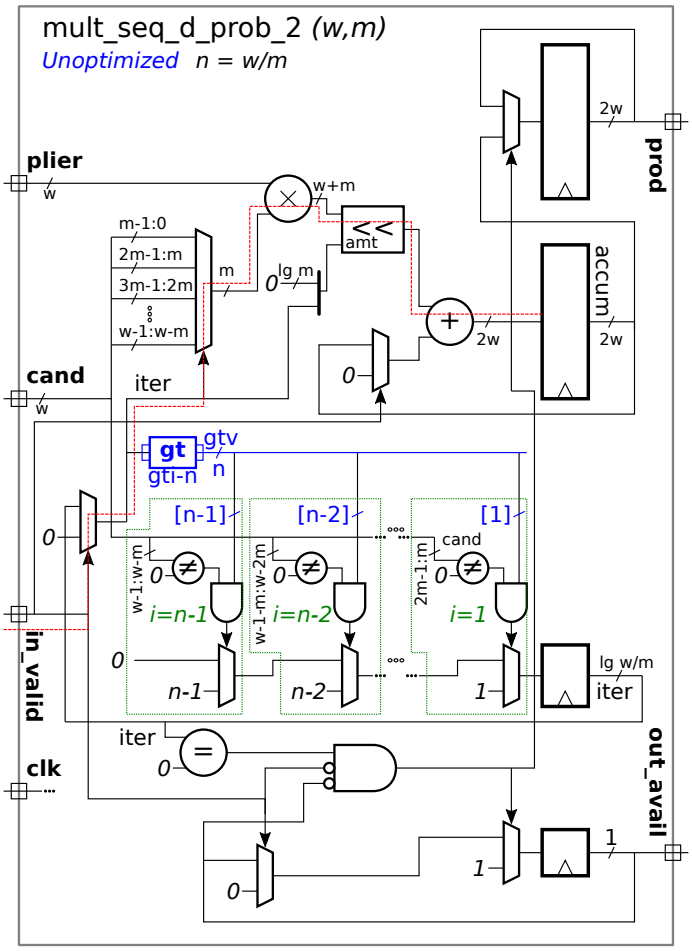
The $\lceil \lg n \rceil$-input NOR gate implementing `iter==0` costs $\lceil \lceil \lg n \rceil - 1 \rceil \, u_c$.

The new AND and OR gates cost $1 \, u_c$ each. The existing (and unchanged) three-input AND gate costs $2 \, u_c$. The total cost is $\lceil \lceil \lg n \rceil + 3 \rceil \, u_c$.





(*b*) In the version of the module appearing below the $\boxed{>}$ units have been replaced by one module, `gt`, the changed hardware appears in blue. As can be inferred from the diagram bit `i` of the output of `gt`, `gtv`, is 1 iff `i>iter`. In the Verilog code below `gt` is instantiated but it is not being used. Modify the Verilog code so that **the existing `for` loop** uses the output of `gt` instead of the `>` operators. Pay attention to the version of `iter` used by `gt`.



- ☑ Use `gt` output in existing `for` loop.
- ☑ Make sure that `gt` uses correct `iter` version.

```
module mult_seq_d_prob_2
  #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,
     output logic out_avail,
     input uwire clk, in_valid,
     input uwire [w-1:0] plier, cand );

   localparam int n = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(n);
   uwire [n-1:0][m-1:0] cand_2d = cand;
   bit [iter_lg-1:0] iter, next_iter;
   logic [2*w-1:0] accum;

   uwire [n-1:0] gtv;

   uwire [iter_lg-1:0] gt_iter = (in_valid ? 0 : iter);          // ☑ FILL IN

   gt #(n,iter_lg) gti( gtv, gt_iter );

   always_ff @( posedge clk ) begin

      if ( in_valid ) begin
         iter = 0;   accum = 0;  out_avail = 0;
      end else if ( !out_avail && iter == 0 ) begin
         prod = accum;   out_avail = 1;
      end

      accum += plier * cand_2d[iter] << ( iter * m );

      next_iter = 0;

      // for ( int i=n-1;  i>0;  i-- ) if ( i>iter && cand_2d[i] ) next_iter = i;
      for ( int i=n-1;  i>0;  i-- ) if ( gtv[i] && cand_2d[i] ) next_iter = i;

      iter = next_iter;
   end
endmodule
```

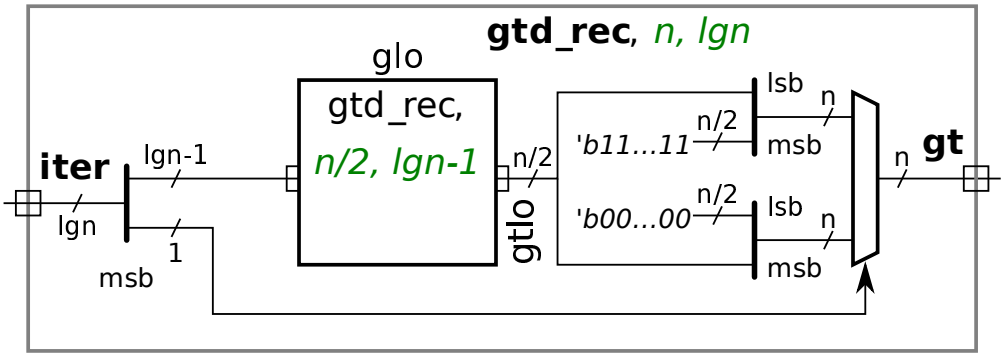
Problem 2: [25 pts] The point of the `gt` module in the previous problem was to reduce cost, just in case the synthesis program didn't notice that the cost of computing each of `n-1>iter`, `n-2>iter`, ..., `2>iter`, `1>iter`, would be less than $n-1$ times the cost of computing one of them. The recursive module below computes these quantities and can be used for the `gt` module from the previous problem.

```
module gtd_rec #( int n = 16, int lgn = $clog2(n) )
   ( output logic [n-1:0] gt, input uwire [lgn-1:0] iter );
   localparam int nh = n / 2;    // Note: n must be a power of 2.
   if ( n == 2 ) begin
      assign gt[0] = 0;
      assign gt[1] = !iter[0];
   end else begin
      uwire [nh-1:0] gtlo;
      gtd_rec #(nh) glo( gtlo, iter[lgn-2:0] );
      localparam logic [nh-1:0] zeros = 0, ones = -1;
      assign gt = iter[lgn-1] ? { gtlo, zeros } : { ones, gtlo };
   end
endmodule
```

(*a*) Show the hardware that will be inferred for this module for an arbitrary value of n. In this case, do not show what is inside the recursively instantiated module.

☑ Show hardware for arbitrary $n > 2$. (Don't show recursive module contents.)

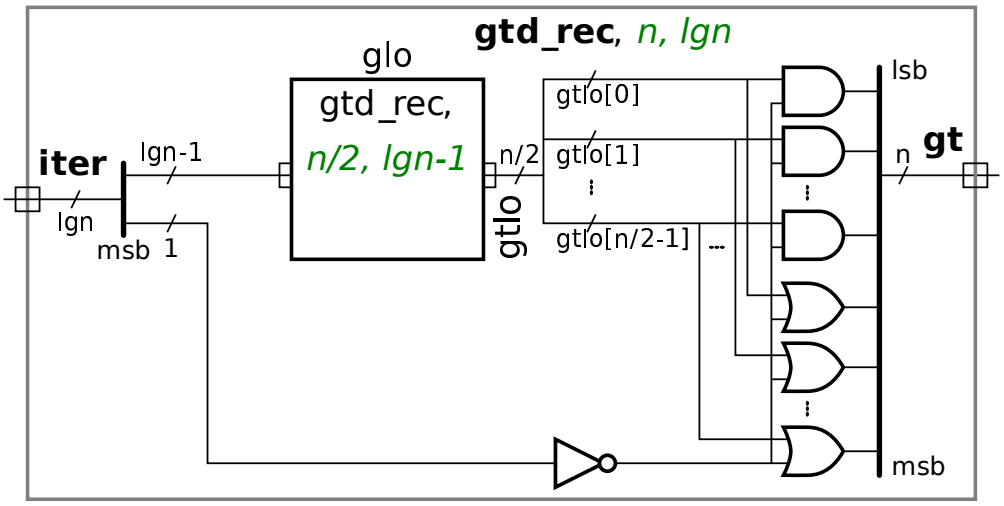
Solution appears below.



(*b*) There should be a significant optimization opportunity in the hardware above. Show it.

☑ Show how the hardware will be optimized. The result should be AND, OR, and other basic logic gates.

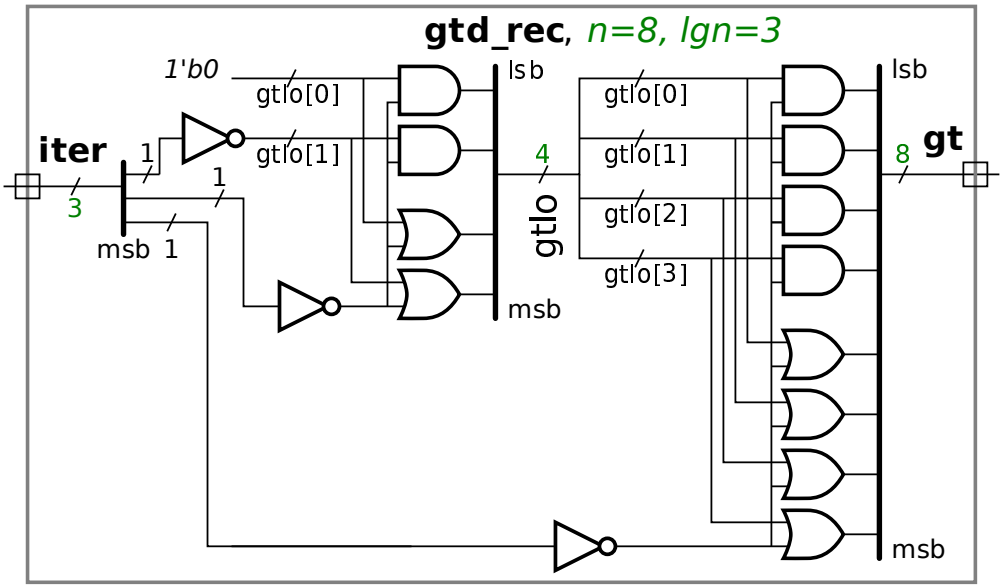
Solution appears below. From the previous solution notice that the $n/2$ LSB of the lower mux input are all zeros. Therefore we can optimize the three gates per bit, into just an AND gate using the inverted select signal. Similarly, the $n/2$ MSB of the upper mux input are all 1's, so we can optimize those bits into just an OR gate.



(*c*) Show the hardware that will be inferred for $n = 8$ after elaboration. That is, show the hardware inside all of the recursive instantiations.

☑ Show hardware for $n = 8$. Show the contents of all recursively instantiated modules.

The solution appears below.



(*d*) Compute the cost and delay using the simple model. Show these in terms of n assuming that n is a power of 2.

☑ Cost and ☑ delay in terms of n.

The cost of the hardware for $n = 2$ is 0 (because with the simple model NOT gates are free!). The cost of the hardware for size $n = 2^\eta$, $\eta > 1$ is n gates plus the cost of a size $n/2$ module. The total cost for a module of size $n = 2^\eta$, $\eta > 1$ is

$$\sum_{l=2}^{\eta} 2^l = 2^{\eta+1} - 4 = (2n - 4)\,\mathrm{u_c}.$$

Since the critical path through each level is 1, the total delay is

$$\sum_{l=2}^{\eta} 1\,\mathrm{u_t} = (\eta - 1)\,\mathrm{u_t} = (\lg n - 1)\,\mathrm{u_t}.$$

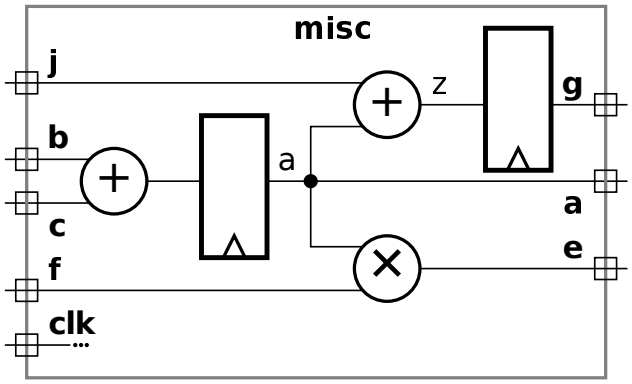
Problem 3: [20 pts] Consider the module below.

```
module misc #( int n = 8 )
   ( output logic [n-1:0] a, g, e,
     input uwire [n-1:0] b, c, j, f,    input uwire clk );

   logic [n-1:0] z;

   always_ff @( posedge clk ) begin
      a <= b + c;   // Note: nonblocking assignment.
      z = a + j;
      g = z;
   end

   always_comb begin
      e = a * f;
   end

endmodule
```

(*a*) Show the hardware that will be inferred for the module above.

☑ Show inferred hardware. ☑ Pay attention to what is and is not a register. ☑ Clearly show module ports.

Solution appears below. Registers are inferred for `a` and `g` because they are live out values of the `always_ff` block. Because a non-blocking assignment is used for `a` the previous value of `a` is used (the one before assigning `b+c`) when computing `a+j`.



```
module misc #( int n = 8 )
   ( output logic [n-1:0] a, g, e,
     input uwire [n-1:0] b, c, j, f,    input uwire clk );

   logic [n-1:0] z;

   always_ff @( posedge clk ) begin   // Code Position Label: alf
      a <= b + c;   // Note: nonblocking assignment.
      z = a + j;
      g = z;
   end

   always_comb begin                  // Code Position Label: alc
      e = a * f;
   end

endmodule
```

(*b*) Suppose that the event queue is empty at $t = 10$ when simulating the module above. Show the contents of the event queue for the code above based on the following changes: At $t = 10$ `j` changes. At $t = 12$ `clk` changes from 0 to 1. At $t = 14$ `f` changes.

☑ Show the state of the event queue from $t = 10$ until it is empty.

The solution appears below. Call the numbers along the top of the diagrams below *steps*. Step 1 shows the state of the event queue at $t = 10$. At step 2 `j` changes. Object `j` is not in the sensitivity list for any piece of code so nothing happens, which is why step 3 is exactly like step 1. Sorry `j`. At step 6 `clk` changes from 0 to 1. Since `clk` is in the sensitivity list for the `always_ff` block's event control, `@( posedge clk )`, the simulator will put a resume event for that block, shown as alf, in the inactive region. At step 7 the active region is empty, so the inactive region is copied into the active region and so the `always_ff` block will execute in step 9. (If there were several items in the active region, they would execute one at time.) In step 11 the `a <= b+c` line results in an update event for a being scheduled in the NBA region, shown as Upd-a. When the Upd-a event executes it causes the `always_comb` block, shown as alc, to be scheduled because both `a` and `f` are on the sensitivity list for that block. Event alc executes at step 17, after which there is no more work to do for $t = 12$. At $t = 14$ `f` changes causing alc to be seceduled again.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t = 10$	→	$t = 10$	→	$t = 12$	→	$t = 12$	→	$t = 12$	→	$t = 12$	→	$t = 12$	→	$t = 12$
active		*active*		*active*		*active*		*active*		*active*		*active*		*active*
inactive		*inactive*		*inactive*		*inactive*		alf		alf		*inactive*		Upd-a
nba		*nba*		*nba*		alf		*inactive*		*inactive*		*nba*		*inactive*
						nba		*nba*		*nba*		Upd-a		*nba*
										Upd-a				

16	17	18	19	20	21	22	23	24	25	26	27	28	29
→	$t = 12$	→	$t = 12$	→	$t = 12$	→	$t = 14$	→	$t = 14$	→	$t = 14$	→	$t = 14$
	active		*active*		*active*		*active*		*active*		*active*		*active*
	inactive		alc		*inactive*		*inactive*		*inactive*		alc		*inactive*
	alc		*inactive*		*nba*		*nba*		alc		*inactive*		*nba*
	nba		*nba*						*nba*		*nba*		

Problem 4: [10 pts] Answer each question below.

(*a*) The module below is not compilable. Explain why and fix it based on what it looks like it is trying to do.

```
module more
  ( input uwire [5:0] w,
    input uwire [w-1:0] a, b,
    output uwire [w:0] s );

   assign s = a + b;

endmodule

 // SOLUTION
module more
 #( int w = 16 )
  ( input uwire [w-1:0] a, b,
    output uwire [w:0] s );

   assign s = a + b;

endmodule
```

☑ Fix the problem.

☑ Describe the problem:

Packed vector dimensions must be specified using elaboration-time constants, but the dimensions of `a`, `b`, and `s` are specified in terms of a module input, which is not a constant value. The fix assumes that `w` was supposed to be a module parameter.

(*b*) The module below is supposed to count cycles but it won't work as written. Describe the problem and fix it.

```
module tic_toc
  ( output logic [7:0] cycles,
    input uwire clk, reset );

   always_comb begin

      if ( reset ) cycles = 0;
      else if ( clk ) cycles = cycles + 1;

   end

endmodule

 // SOLUTION
module tic_toc
  ( output logic [7:0] cycles,
    input uwire clk, reset );

   always_ff @( posedge clk )
      if ( reset ) cycles = 0; else cycles = cycles + 1;

endmodule
```

☑ Describe the problem:

The sensitivity list of the `always_comb` module includes live-in values, including `cycles` in this case. But `cycles` is also a live-out, and so there is the potential for an infinite loop since each change in `cycle` will cause the `always_comb` to reexecute.

☑ Fix the problem.

In the fixed code, appearing above, the `always_comb` is replaced by an `always_ff`.

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

(*a*) Appearing below is synthesis data showing the clock period of degree-m sequential workfront multipliers and degree-m sequential regular (dm) multipliers for sizes $m = 1$, $m = 2$, $m = 4$, and $m = 8$.

Module Name	Area	Period Target	Period Actual	Total Latency
mult_seq_wfront_m_w32_m1	191334	1000	3766	241024
mult_seq_wfront_m_w32_m2	205303	1000	3857	123424
mult_seq_wfront_m_w32_m4	260182	1000	5266	84256
mult_seq_wfront_m_w32_m8	351910	1000	7031	56248
mult_seq_dm_w32_m1	246818	1000	31113	995616
mult_seq_dm_w32_m2	279486	1000	30994	495904
mult_seq_dm_w32_m4	314724	1000	32127	257016
mult_seq_dm_w32_m8	408659	1000	31251	125004

As m increases the clock period of the workfront multiplier increases by a significant amount, while the period of the sequential multiplier barely changes. Why?

☑ Why does the workfront period increase so much more than that of the regular multiplier?

The critical path of a degree-m workfront multiplier passes through m binary-full adders (BFAs), whereas the critical path for the degree-m regular multiplier passes through $m - 1 + 2w$ BFAs (or $m - 1 + w$ for the streamlined version). For the workfront multipliers the BFA part of the critical path length increases by a factor of 8 when the degree increases from $m = 1$ to $m = 8$. In contrast the BFA component of the critical path for the regular multipliers increases by a factor of $\frac{64+7}{64} \approx 1.11$. That's a much smaller increase and its effect is harder to see (that is $1.11 \times 31113 \neq 31251$) because the synthesis program can do more to optimize longer critical path lengths.

Let $p_w(m)$ and $p_r(m)$ denote the clock period of the degree-m workfront and regular multipliers. Show expressions for $l_w(m)$ and $l_r(m)$, the latencies of these multipliers.

☑ Finish the following expression for latency: $l_w(m) = p_w(m)$ $\boxed{\times 2\lceil w/m \rceil}$

Solution is boxed above. The workfront multiplier requires $2\lceil w/m \rceil$ clock cycles to compute a solution. That's twice as many cycles as the regular multiplier, but the clock period is much lower.

☑ Finish the following expression for latency: $l_r(m) = p_r(m)$ $\boxed{\times \lceil w/m \rceil}$

Solution is boxed above. The regular multiplier requires $\lceil w/m \rceil$ clock cycles to compute a solution. That's half the number of cycles of workfront, but the period is much longer.

(*b*) The reasoning in the statement below is, as of this writing, incorrect. Provide the correct reason to not spend time on multiplier modules.

"One should not spend time trying to develop efficient multiplication hardware because the synthesis program is very good at optimizing logic and will synthesize something at least as good as a human can."

☑ When working on a design that makes heavy use of multiplication one should just use multiplication operators and not try to implement your own because:

The problem with the statement above is that as of this writing, we can't expect a synthesis program to discover faster equivalent versions of circuits that we enter for circuits of any complexity. For example, the synthesis program does not come close to optimizing the behavioral merge module from Homework 5 to the performance of a Batcher merge module. There are two reasons for using multiply operators. First, we expect that humans have provided the synthesis program with a library of different multiplier designs that the synthesis program will choose from. We don't expect our designs to be better than the designs produced by these humans. The second reason is that by using multiplication operators rather than providing your own modules, the synthesis program might be able to apply algebraic simplifications to some expressions.

(*c*) Sequential multipliers S0 and S1 have the same latency and cost, but the clock period for S1 is lower than S0.

☑ Which is preferred? ☑ Explain.

Both multipliers have the same cost, latency, and throughput. If no other factors are important then either one could be used. Generally sequential logic uses more power at higher frequencies and so the higher clock period, and so S0, is preferred.

Note that since the clock period of S1 is lower, it must require more cycles to compute a product than S0. For example, suppose that the period for S1 was $0.5\,\text{ns}$ and the period for S0 was $1\,\text{ns}$. Suppose that S1 took $10\,\text{cyc}$ to compute a product. The problem states that the latency of S0 and S1 are the same, therefore S0 must take $5\,\text{cyc}$.

Pipelined multipliers P0 and P1 have the same latency and cost, but the clock period for P1 is lower than P0.

☑ Which is preferred? ☑ Explain.

Because these multipliers are pipelined the clock frequency determines throughput. Therefore P1, which has the higher higher clock frequency, will have higher throughput.

(*d*) In the module below notice that `cand_2d` is no longer available. Modify the line updating `accum` to use `cand` instead.

```
module mult_seq_dm #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,
     input uwire [w-1:0] plier, cand, input uwire clk);

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   // uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg:1] iter;
   logic [2*w-1:0] accum;

   always @( posedge clk ) begin

      if ( iter == iter_lg'(iterations) ) begin
        prod = accum; accum = 0; iter = 0;
      end

      //  ☑   Fix line below

      accum += plier *      cand[ m*iter +: m ]    << ( iter * m );

      iter++;
   end
endmodule
```

Solution appears above. The solution uses an indexed range expression, `m*iter +: m`, to extract the `m`-bit slice from `cand`. The `m*iter` specifies the position to start and the `m` is the number of bits. Unlike the part select operator, `:`, with the index-range operators, `+:` and `-:`, the first operand does not need to be an elaboration-time constant. (The second operand must be an elaboration-time constant for the part select and the index-range operators.)

The following is **invalid Verilog**: `cand[ m*(iter+1) -1 : m*iter ]`, though it would retrieve the needed bits if SystemVerilog 2017 weren't so strict. It is invalid because with the ordinary slicing operator, `:`, both operands must be elaboration time constants. *Grading Note: Full credit was given for this answer since the indexed range operator was only covered briefly.*

21 Fall 2017 Solutions

Name Solution

Digital Design using HDLs
EE 4755
Midterm Examination

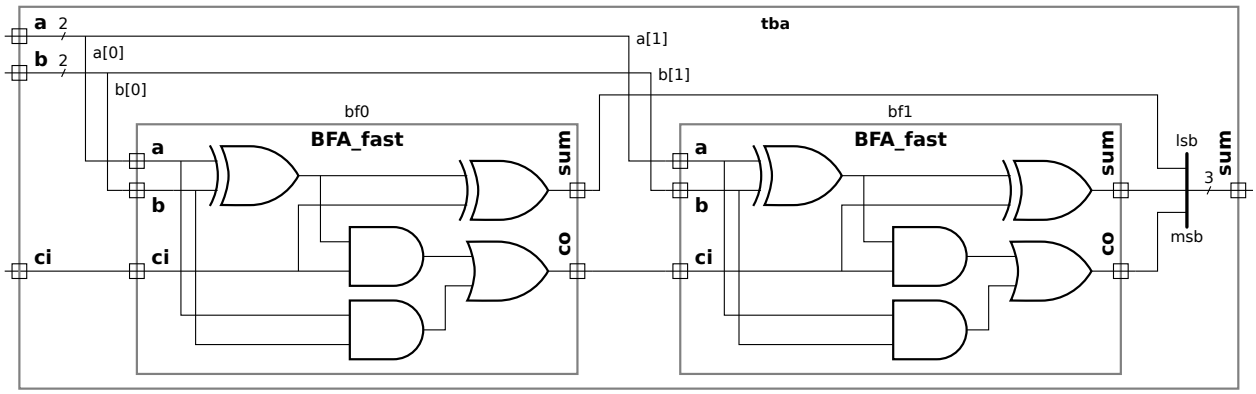
Monday, 16 October 2017 9:30–10:20 CDT

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

Alias Even Ireland.

Good Luck!

Problem 1: [20 pts] Write a Verilog description of the hardware illustrated below. The description **must include the modules and instantiations as illustrated**. The description can be behavioral or structural, but it must be synthesizable.



☑ Verilog corresponding to illustrated hardware.

☑ Show instantiations, ☑ Verilog for instantiated module(s), ☑ and all module ports.

```
// SOLUTION
module BFA_fast( output uwire sum, co, input uwire a, b, ci );

   // Note: axb explicitly computed once and used twice.
   uwire  axb = a ^ b;
   assign sum = axb ^ ci;
   assign co = axb && ci || a && b;

endmodule

module tba( output uwire [2:0] sum,   input uwire [1:0] a, b,   input uwire ci );

   uwire  c;
   BFA_fast bf0( sum[0], c,      a[0], b[0], ci );
   BFA_fast bf1( sum[1], sum[2], a[1], b[1], c  );

endmodule
```

Problem 2: [20 pts] Appearing below is a partially completed recursive description of an $n = 2^b$-input, w-bit multiplexor, which is a generalized version of the multiplexors appearing in Homework 1. Complete it.

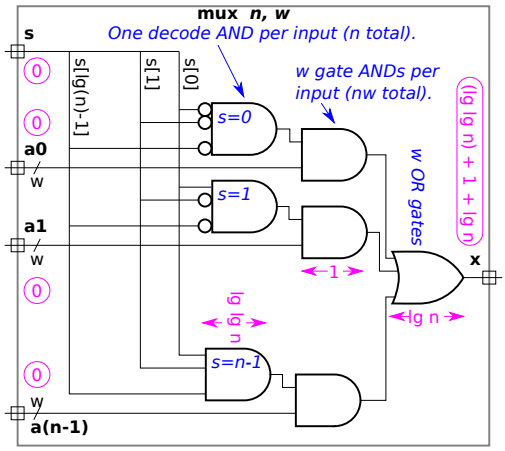
☑ Fill in the condition and code for the terminating case.

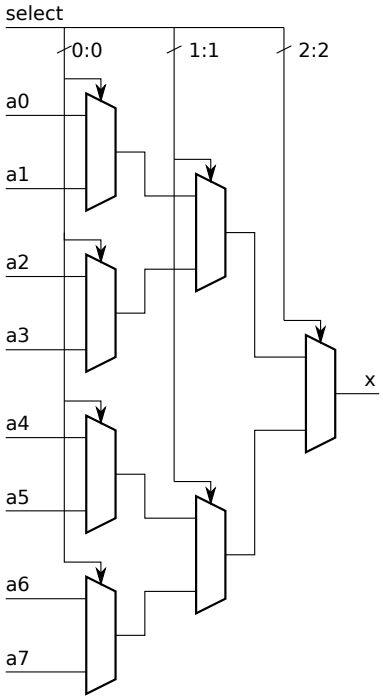
☑ Complete recursive case, including the instantiation port and parameter connections (look for FILL IN).

```
module muxn #( int w = 5, int b = 4, int n = 1 << b )
   ( output uwire [w-1:0] x,  input uwire [b-1:0] sel, input uwire [w-1:0] a[0:n-1] );

   if ( b == 1    )  // Terminating Case Condition              <----  ☑  FILL IN
     begin
        // Terminating Case

        assign x = a[sel];



     end else begin
        // Recursive Case

        uwire [w-1:0] y[2];

        // Instantiate two n/2-input muxen, and connect each to half the inputs.
        //
        //         ----      ----                              <----  ☑  FILL IN
        muxn #( .w( W ), .b( b-1 ) ) mlo( y[0],  sel[b-2:0],  a[   0 : n/2-1 ] );



        //         ----      ----                    -----     <----  ☑  FILL IN
        muxn #( .w( W ), .b( b-1 ) ) mhi( y[1],  sel[ b-2:0 ],  a[ n/2 : n-1   ] );







        // Instantiate one 2-input mux.
        //
        //         ----      ----       -------------------   <----  ☑  FILL IN
        muxn #( .w( W ), .b( 1 ) ) m2(   x,  sel[b-1],  y    );


   end

endmodule
```

Problem 3: [20 pts] Appearing below to the right is an 8-input multiplexor constructed from 2-input multiplexors using the technique from Homework 1 and from the previous problem. Call a multiplexor constructed this way a *tree mux*. Appearing below to the left is a diagram showing a *flat mux*, the kind usually used in class. The flat mux diagram shows a timing analysis based on the simple model, and some details about cost.

For reference: $\sum_{i=0}^{b-1} a2^i = a(2^b - 1)$. Assume that n is a power of 2.





(*a*) Compute the cost of an n-input, w-bit flat mux using the simple model and without optimization.

☑ Cost of flat mux **in terms of** n **and** w.

As can be seen from the diagram, the n decode gates each have $\lg n$ inputs, for a total cost of $n(\lg n - 1)$. The gate AND gates each have two inputs and there are nw of them, for a total cost of nw units. The OR gate has n inputs and there are w of them, so their cost is $(n-1)w$ units. The total cost is then $n(\lg n - 1) + 2nw - w$ units.

(*b*) Compute the cost of an n-input, w-bit tree mux using the simple model.

☑ Cost of tree mux **in terms of** n **and** w. ☑ Describe assumptions made about 2-input mux implementation.

As can be seen in the diagram, in the first column there are $n/2 = 2^{b-1}$ multiplexors, where $n = 2^b$. The second column has 2^{b-2} multiplexors, and so on, the last column has $2^0 = 1$ multiplexor. The total number of multiplexors is $\sum_{i=0}^{b-1} 2^i = 2^b - 1$ multiplexors. The cost of a 2-input, w-bit mux flat is $3w$ units (see the previous part) and so the total cost of the tree mux is $3w(2^b - 1) = 3w(n - 1)$.

(*c*) Compute the delay of an n-input, w-bit tree mux using the simple model.

☑ Delay of tree mux **in terms of** n **and** w.

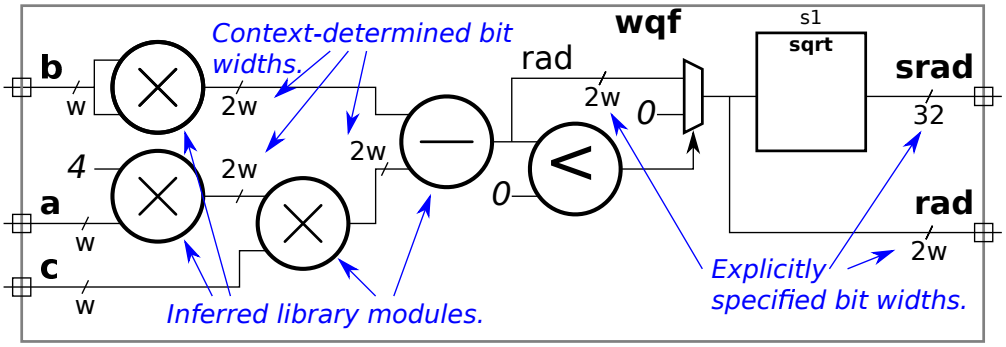
The critical path passes through $\lg n$ layers (columns in the diagram). Each layer is a 2-input mux, in which the critical path passes through an AND gate and a OR gate, each of two inputs, so the delay is 2 units per layer. Therefore the delay is $2 \lg n$ units.

Problem 4: [15 pts] Show the hardware that will be synthesized for the modules below.

(*a*) Show the hardware that will be inferred for the module below, including the minimum number of bits in each wire. Assume that `sqrt` is defined in a library somewhere.

```
module wqf
  #( int w = 16 )
   ( output logic signed [2*w-1:0] rad,
     output uwire [31:0] srad,
     input uwire [w-1:0] a, b, c );

   sqrt #(32,2*w) s1(srad,rad);

   always_comb begin

      rad = b*b - 4 * a * c;
      if ( rad < 0 ) rad = 0;

   end

endmodule
```

☑ Show inferred hardware. ☑ Show minimum correct bit widths.

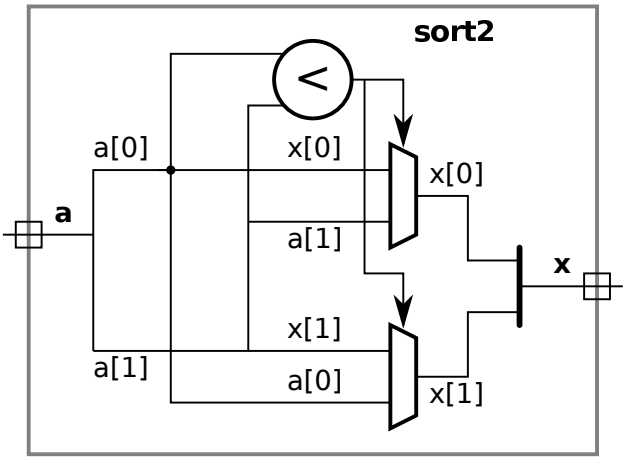


Solution appears above. Note that the basic arithmetic operators are replaced by library modules (shown as circles) provided by the synthesis program, whereas the `sqrt` module is explicitly instantiated in the module above. The multiplexor is inferred from the `if` statement. The select signal is connected to a comparison module, however that could easily be optimized into a connection to the sign bit of output of the subtractor. Similarly the $\times 4$ multiplier could have been optimized to a bit renumbering. But the question asks for *inferred* hardware, and so even these easy optimizations are omitted. The sizes of the wires connected to module ports are given explicitly in the `wqf` module, whereas widths of the internal wires are determined using Verilog rules for bit widths. Under those rules multiplication and subtraction arguments' bit widths are context-determined. Note that `rad` is explicitly sized to $2w$ bits, this context at the subtract output determines the size as the subtract inputs, which in turn determines the width needed for the multiplies.

(*b*) Show the hardware that will be inferred for the module below.

```
module sort2 #( int w = 4 )
   ( output logic [w-1:0] x[2], input uwire [w-1:0] a[2] );

   always_comb begin

      for ( int i=0; i<2; i++ ) x[i] = a[i];
      if ( a[0] < a[1] ) begin x[0] = a[1]; x[1] = a[0]; end

   end

endmodule
```

☑ Show inferred hardware.



Solution appears above. Note that the effect of the `for` loop is only to make `x[0]` another name for `a[0]` and `x[1]` another name for `a[1]`.

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

(*a*) The `mux2` module below uses implicit structural code. Modify it so that it uses behavioral (procedural) code.

```
module mux2 #( int w = 16 )
   ( output uwire [w-1:0] x,
     input uwire s, input uwire [w-1:0] a,b );

   assign  x = s == 0 ? a : b;

endmodule


// SOLUTION
module mux2 #( int w = 16 )
   ( output logic [w-1:0] x,
     input uwire s, input uwire [w-1:0] a,b );

   always_comb x = s == 0 ? a : b;

endmodule
```

☑ Modify so that is procedural. ☑ Change ports if necessary.

Solution appears above. Note that in addition to changing **assign** to **always_comb**, the kind of object of the input port was changed from net to var. (**uwire** is an object of kind net with a default data type of **logic**, and **logic** is a data type with a default object kind of var.)

(*b*) Modify the module port and parameter declarations below so that the Verilog is correct. Do not modify the contents of the module itself. Note that `opt` is not defined, but that it should be. *Note: In the original exam* `assign` *was omitted from the module body, making the problem impossible to solve.*

```
module sum_or_dff
  #( int w = 16 )
   ( output uwire [w-1:0] x,
     input uwire [w-1:0] a, b );

   if ( opt == 0 ) assign x = a+b; else assign x = a-b;

endmodule
```

```
module sum_or_dff
  #( int w = 16, int opt = 1  )
   ( output uwire [w-1:0] x,
     input uwire [w-1:0] a, b );

   if ( opt == 0 ) assign x = a+b; else assign x = a-b;

endmodule
```

☑ Modify port and parameter declarations for correctness.

Solution appears above. The `if` statement, because it is in module scope, is a generate statement and therefore the condition must be an elaboration-time constant. For that reason `opt` is made a parameter.

Problem 6: [15 pts] Answer each question below.

(*a*) Why is `always_comb` preferred over `always @( x or y or .. )` when describing combinational logic?

☑ `always_comb` preferred because ...

... there is no need to take the trouble to list all of the live-in objects nor is there the risk of omitting one.

☑ What is the risk with `always @( x or y or .. )`?

If a live-in object is omitted from the sensitivity list, code in the block will not be re-executed when the value of the omitted object changes but other variables don't change. For example, consider the `sum` module below. The intent is hardware that adds three numbers together. But because `z` was omitted the value of output `a` will not be "correct" if `z` changes but `x` and `y` stay the same. In general, the simulation might not produce the answers that are expected and the synthesis program will infer a latch (or latches) rather than combinational logic.

```
// Module illustrating error easily made using old-school Verilog sensitivity lists.
module sum(output logic [15:0] a, input uwire [15:0] x, y, z );
 always @( x or y ) a = x + y + z;
endmodule
```

(*b*) Describe what the technology mapping step of synthesis is, and the kind of optimizations that need to be performed after technology mapping.

☑ Technology mapping is:

the substitution of generic components in the inferred hardware with components in the target technology being synthesized. For example, a three-input AND gate (a generic component) might be replaced by `ASx9AND4`, a four-input AND gate in Acme Silicon's x9 ASIC cell library. (Acme Silicon's x9 ASIC cell library does not have a three-input AND gate.) Note: Acme Silicon is a fictional silicon foundry made up for this problem's solution.

☑ Optimizations that must be performed after technology mapping:

Most cost reduction optimizations must be done after technology mapping because only after technology mapping are the cost and timing of components known.

(*c*) The module below adds a real and an integer and assigns the sum (in real format) to its output. It is valid Verilog but is not synthesizable by Owr EDA software. So, you call Owr EDA and ask, "why not?". They answer, "because it is impossible to add an integer to a real." Is that the real reason? Explain.

```
module plusri (output real sum, input real a, input [20:0] x);
   assign       sum = a + x;
endmodule
```

☑ Reason a+x not synthesizable by Owr EDA software:

If Owr EDA wanted to they could infer an integer-to-real conversion module to convert `x` to a real and a real addition module to compute the sum. There are no fundamental reasons why a synthesis program can not have such features. They did not do so because it never made it to the top of their to do list, perhaps.

Name Solution

Digital Design using HDLs
LSU EE 4755
Final Examination

Wednesday, 6 December 2017 15:00-17:00 CST

Problem 1		(15 pts)
Problem 2		(25 pts)
Problem 3		(20 pts)
Problem 4		(10 pts)
Problem 5		(30 pts)
Exam Total		(100 pts)

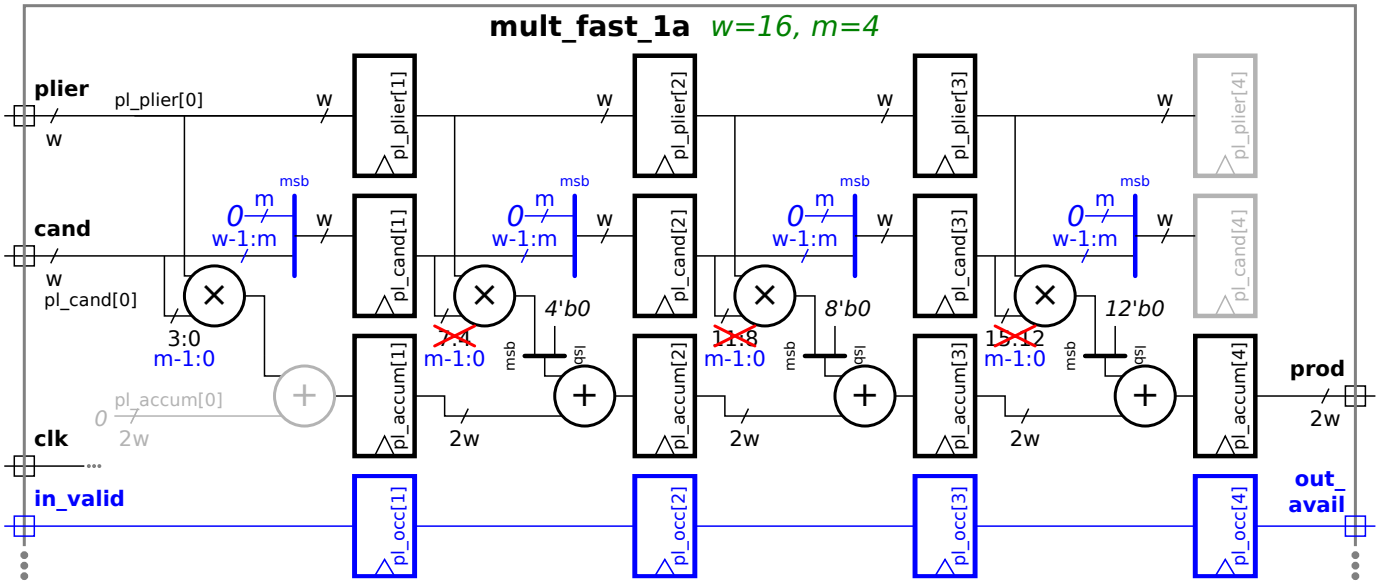
Alias Pie Plain

Good Luck!

Problem 1: [15 pts] The Verilog code below is the solution to Problem 1a of Homework 7. Below that is the hardware **for a slightly different pipelined multiplier**. Modify the hardware to match the Verilog code. Changes need to be made for each line commented `DIFFERS`.

☑ Modify hardware to reflect Verilog.

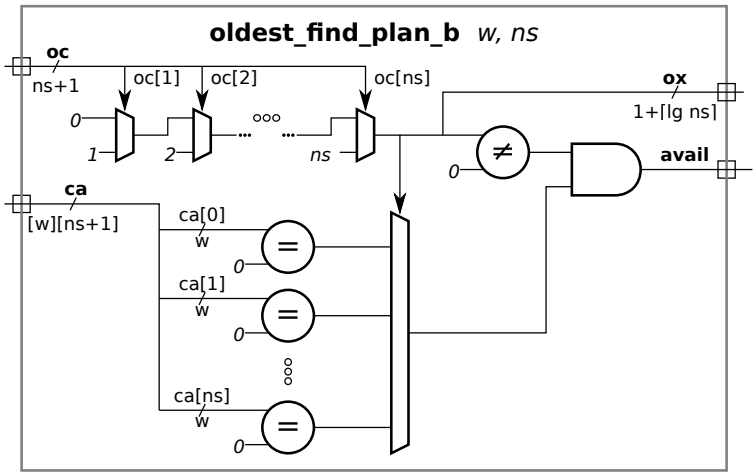
```
module mult_fast_1a #( int w = 16, int m = 4 )
   ( output uwire [2*w-1:0] prod,
     output uwire out_avail,        input uwire clk, in_valid,      //  ☑ DIFFERS
     input uwire [w-1:0] plier, cand );
   localparam int nstages = ( w + m - 1 ) / m;
   logic [2*w-1:0] pl_accum[0:nstages];
   logic [w-1:0] pl_plier[0:nstages], pl_cand[0:nstages];
   logic pl_occ[0:nstages];                                          //  ☑ DIFFERS

   assign prod = pl_accum[nstages];
   assign out_avail = pl_occ[nstages];                               //  ☑ DIFFERS

   always_ff @( posedge clk ) begin
      pl_occ[0] = in_valid;                                          //  ☑ DIFFERS
      pl_accum[0] = 0;    pl_plier[0] = plier;    pl_cand[0] = cand;

      for ( int stage=0; stage<nstages; stage++ ) begin
         pl_plier[stage+1] <= pl_plier[stage];
         pl_accum[stage+1] <= pl_accum[stage] + ( pl_plier[stage]
             * pl_cand[stage][m-1:0] << stage*m );                   //  ☑ DIFFERS
         pl_cand[stage+1]  <= pl_cand[stage] >> m;                   //  ☑ DIFFERS
         pl_occ[stage+1]   <= pl_occ[stage];                         //  ☑ DIFFERS
      end
   end
endmodule
```



Solution appears above in blue. A straightforward addition is the pipeline latch, `pl_occ`, to pass the `in_valid` signal. The other change is in the way that the multiplicand is passed from stage to stage. In the original design the multiplicand (`cand`) was passed

unchanged. But in the Verilog description above the multiplicand is shifted by `m` bits each stage. With that change all the multipliers can look at the `m` least significant bits rather that a different slice each stage. This change in the way the multiplicand is handled makes no difference in the cost of the hardware. Either way a decent synthesis program should figure out which bits in `pl_cand` will never be used and optimize them out.

Problem 2: [25 pts] Module `oldest_find_plan_b`, illustrated below, is based on **an alternative solution** to Homework 7 Problem 1b. Below the hardware illustration is incomplete Verilog code for this module. The Verilog code uses abbreviated names, such as `ns`, comments show the original names from the assignment, such as `nstages`. Complete the module. *Note: This problem can be solved without having ever seen Homework 7, though not as quickly.*



☑ Complete the module so that it matches the hardware above.

```
module oldest_find_plan_b
  #( int w = 15, int ns = 3            /* nstages */ )
   ( output logic [$clog2(ns):0] ox,   // oldest_idx
     output uwire avail,               // out_avail
     input uwire oc[0:ns],             // pl_occ
     input uwire [w-1:0] ca[0:ns] );   // pl_cand


   /// SOLUTION

   // Compute ox (oldest_idx). This is similar to the Homework 7 solution
   //
   always_comb begin
      ox = 0;
      for ( int i=1; i<=ns; i++ ) if ( oc[i] ) ox = i;
   end

   // Determine whether *each* element of ca is zero.
   //
   logic [0:ns] cz;
   always_comb for ( int i=0; i<=ns; i++ ) cz[i] = ca[i] == 0;

   assign out_avail = ox != 0 && cz[ox];


endmodule
```

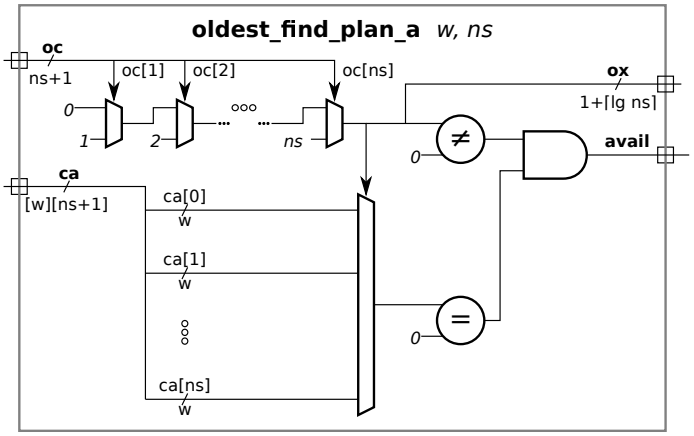
Problem 3: [20 pts] Appearing below are two variations on the oldest index module from the previous problem. The Plan A version is based on the code from the posted Homework 7 solution. The Plan B module is slightly different.

(*a*) Compute the cost of each module based on the simple model after optimizing for constant values. Use symbol w (for `w`) and n (for `ns`). Base the cost of an α-input, β-bit multiplexor on the tree (recursive) implementation. Recall that the tree implementation consists of $\alpha - 1$ two-input multiplexors arranged in a tree.

☑ Plan A cost in terms of w and n. ☑ Show cost components on diagram, such as cost of big mux, ☑ don't forget to account for the constant inputs, and ☑ for the number of bits in each wire.

The lower input to each of the 2-input muxen is constant, so the cost per bit of each multiplexor is at most 1. *At most* because in some cases, such as the first, the upper input is also constant. The number of bits for the first mux is 1 and the number of bits for the last multiplexor is $\lceil \lg n \rceil$ (because the largest input to any mux is n and it takes $\lceil \lg n \rceil$ bits to represent n as an unsigned integer). To keep things simple assume that all of the 2-input muxen are $\lceil \lg n \rceil$ bits wide. Then the total cost of the $n-2$ 2-input muxen is $(n-2)\lceil \lg n \rceil$.

The big mux has $n+1$ inputs, each w bits wide. The total cost is $(n+1-1)3w = 3wn$ units.

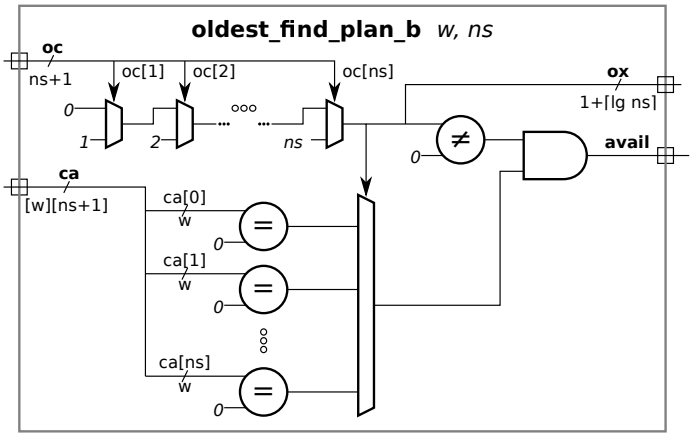


The $\neq 0$ unit can be realized using a $\lceil \lg n \rceil$-input OR gate, and the $= 0$ unit can be realized using a w-input NOR gate. The costs are the number of inputs minus one. The total cost is:

$$\overbrace{(n-2)\lceil \lg n \rceil}^{\text{2-input muxen}} + \overbrace{\lceil \lg n \rceil - 1}^{\neq 0} + \overbrace{3nw}^{\text{Big Mux}} + \overbrace{w-1}^{=0} + \overbrace{1}^{\text{AND}}$$

☑ Plan B cost in terms of w and n. ☑ Show cost components on diagram, such as cost of big mux, ☑ don't forget to account for the constant inputs and, ☑ for the number of bits in each wire.

In Plan B the $= 0$ comparison is done before the big mux, and so $n+1$ comparison units are needed. Sounds costly. But, the inputs to the big mux are 1, rather than w bits wide. For Plan A the big cost term is $3nw$ (assuming that $w > \lg n$). In Plan B the big cost term is just nw, which is $\frac{1}{3}$ the cost!



The total cost is:

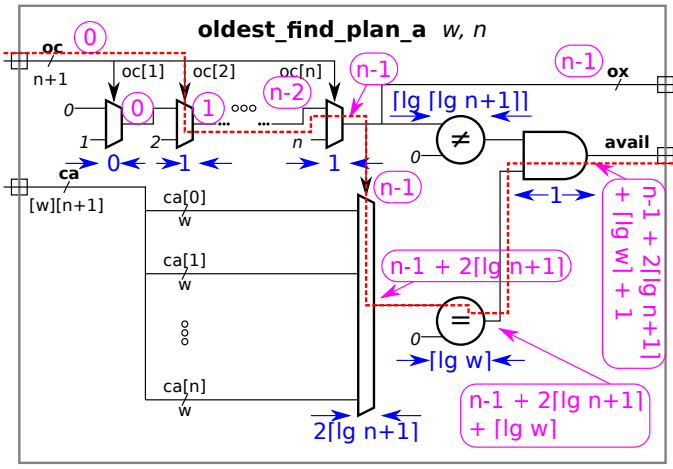
$$\overbrace{(n-2)\lceil \lg n \rceil}^{\text{2-input muxen}} + \overbrace{\lceil \lg n \rceil - 1}^{\neq 0} + \overbrace{(n+1)(w-1)}^{=0} + \overbrace{3n}^{\text{Big Mux}} + \overbrace{1}^{\text{AND}}$$

(*b*) Show the delay along all paths and show the critical path. Compute delay based on the simple model after optimizing for constant values. Use the tree mux described in the previous part.

☑ Plan A: ☑ show delay along all paths, ☑ highlight the critical path, ☑ and show the delay through each component. Show these ☑ in terms of w and n, and ☑ account for constant inputs such as the zeros in the equality units.

Solution appears to the right. The delay through each device is shown in blue, the time at which a signal is available is shown in purple, and the critical path is shown as a red dashed line. Because the 2-input multiplexors have at least one constant input, the delay through them is 1 unit each. The delay through the big mux, which is $n+1$ inputs, is $2\lceil \lg n+1 \rceil$ units, the usual delay though an $n+1$-input tree mux. Both comparison units compare to a constant, their delays are ceiling-log-base-2 of the number of inputs.

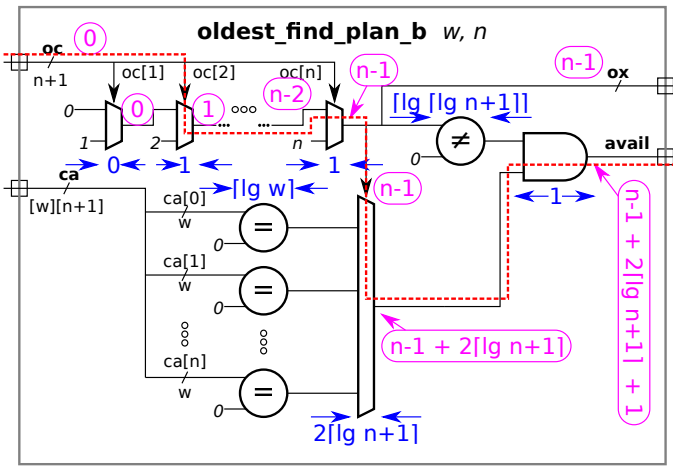
A common mistake was to overlook the possibility that the critical path can pass through a multiplexor select input, as it does here.



☑ Plan B: ☑ show delay along all paths, ☑ highlight the critical path, ☑ and show the delay through each component. Show these ☑ in terms of w and n, and ☑ account for constant inputs such as the zeros in the equality units.

Solution appears to the right, with delays, times, and critical path using the same colors as above. Doing the $=0$ check before the mux reduces the length of the critical path by $\lg w$.

Note that in both the Plan A and Plan B versions the delay through the 2-input muxen is $n-1$. It is possible that the synthesis program could find an optimization that would reduce the delay to something closer to $\lg n$. A human, at least one who payed attention in EE 4755, should be able to do that with no problem.



Problem 4: [10 pts] Explain why each of the modules below is not synthesizable by Cadence Encounter (or similar tools) and modify the code so that it is *without changing what the module does*. *Note: The warning about not changing what the module does was not in the original exam.*

```
module one_run #( int w = 16, int lw = $clog2(w) )
   (output logic all_1s, input uwire [w-1:0] a, input uwire [lw:0] start, stop );
   always_comb begin

      all_1s = 1;

      // for ( int i=start; i<stop; i++ ) all_1s = all_1s && a[i];
      // SOLUTION Below
      for ( int i=0; i<w; i++ )
        if ( i >= start && i<stop ) all_1s = all_1s && a[i];

   end
endmodule
```

☑ Reason code above is not synthsizable:

The number of iterations in the `for` loop depends on non-constant expressions. To be synthesizable the synthesis program must be able to determine the number of loop iterations of an instantiated module. It can't in the module above because the number of iterations depends on the module inputs `start` and `stop`.

☑ Modify code so that it is.

Short Answer: Solution appears above.

Explanation: The lower loop bound has been changed from `start` to 0, a constant (literally a literal). The upper bound has been changed from `stop` to `w`, an elaboration-time constant. The original code is shown commented out.

```
module running_sum #( int w = 32 )
   ( output logic [w-1:0] rsum,
     input uwire [w-1:0] a,     input uwire reset, clk );

   // always @( posedge clk ) if ( reset ) rsum <= 0;
   // always @( posedge clk ) rsum <= rsum + a;

   // SOLUTION Below
   always @( posedge clk ) begin
      if ( reset ) rsum <= 0;
      else         rsum <= rsum + a;
   end

endmodule
```

☑ Modify code so that it is synthsizable.

Solution appears above.

☑ Reason code above was not synthsizable:

Because `rsum` is assigned in two `always` blocks. To be synthesizable a value cannot be assigned in more than one `always` block.

☑ Explain assumption about intended behavior of this module.

Assumed that when `reset` is `1` at a positive edge `rsum` should be set to `0` rather than `a`.

Problem 5: [30 pts] Answer each question below.

(*a*) Show when each piece of code below executes (use the `C` labels) up until the start of `C5c`, and show when and in which region each piece is scheduled. See the table below.

```
module eq;
   logic [7:0] a, b, c, d, x, y, x1, x2, y1, y2, z2;
   always_comb begin          // C1
      x1 = a + b;
      y1 = 2 * b;
   end
   assign x2 = 100 + a + b;   // C2
   assign y2 = 4 * b;         // C3
   assign z2 = y2 + 1;        // C4
   initial begin
      //                         C5a
      a = 0;
      b = 10;
      #2;
      //                         C5b
      a = 1;
      b <= 11;
      #2;
      //                         C5c
      a = 2;
      b = 12;
   end
endmodule
```

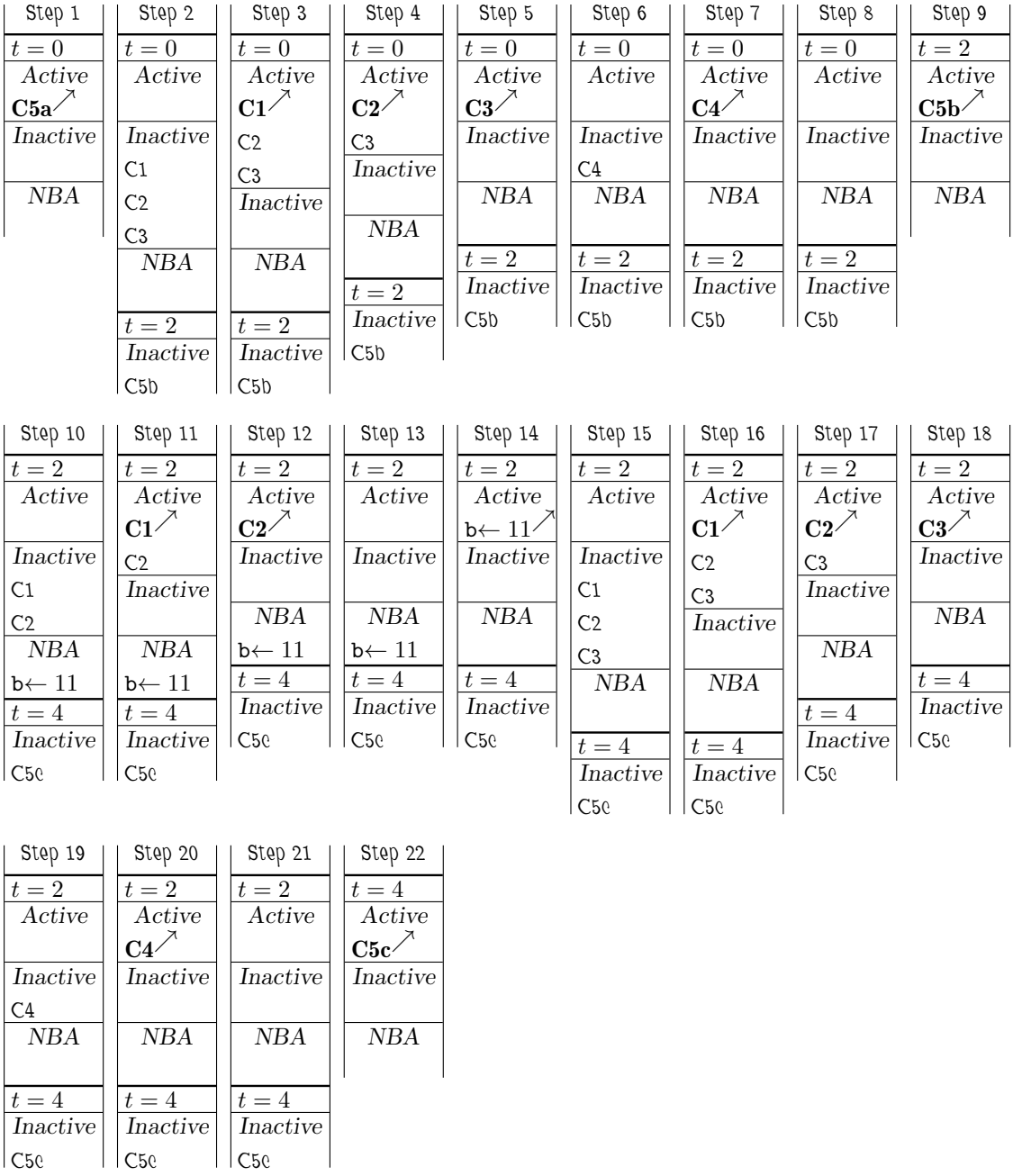
☑ Continue the diagram below so that it shows scheduling up to the point where `C5c` executes.

Step 1	Step 2	Step 3
$t = 0$	$t = 0$	$t = 0$
Active **C5a** ↗	*Active*	*Active*
Inactive	*Inactive* C1 C2 C3	
NBA	*NBA*	
	$t = 2$	
	Inactive C5b	

Solution on next page.

Solution appears below.

Note that when the active region is empty the first non-empty region is bulk-copied into the active region. This occurs, for example, between Step 2 and 3, step 6 and 7. (Warning: step numbers may eventually become wrong. Please report any errors.) Simulation time (shown as $t =$) changes when all regions within the current time step are empty. This occurs at step 8 and step 21.



(*b*) Which of the two modules does what it looks like it's trying to do? Explain.

```
module sa1(input logic [7:0] a, b, c, d, output wire [7:0] x, y );
   assign x = a + b;
   assign y = 2 * x;
   assign x = c + d;
endmodule

module sa2(input logic [7:0] a, b, c, d, output logic [7:0] x, y );
   always_comb begin
      x = a + b;
      y = 2 * x;
      x = c + d;
   end
endmodule
```

☑ Module that is probably correct is:

It is `sa2` that looks correct because the other module, ...

☑ Major problem with other module.

... `sa1`, is using continuous assignments as though they were procedural statements. In particular `x` is assigned twice.

☑ Provide a possible wrong answer from other module.

If `a+b` is not equal to `c+d` then `x` will have some bits set to the undefined state. So a possible wrong answer is that `x = 7'b0001xxxx`. This would occur when `a+b = 7'b00011010` and `c+d = 7'b00010101`.

(*c*) Define throughput and latency and indicate where each is preferred. Provide examples appropriate for pipelined systems.

☑ Throughput is:

The amount of work completed per unit time.

☑ For example:

In a pipelined multiplier with n stages running at a clock frequency ϕ Hz the throughput is ϕ multiplications per second. If $\phi = 1$ GHz the throughput would be 10^9 multiplications per second.

☑ Latency is:

The amount of time from start to finish of one piece of work.

☑ For example,

In the pipelined system the latency is $\frac{n}{\phi}$ s. Suppose $n = 5$ and $\phi = 1$ GHz. Then the clock period is $\frac{1}{\phi} = 1$ ns and the latency is $5 \times 1\,\text{ns} = 5\,\text{ns}$.

☑ If the goal is to improve throughput is higher throughput good or bad?

Higher throughput is good.

☑ If the goal is to improve latency, is higher latency good or bad?

Higher latency is bad. (Lower latency is good.)

☑ In what situation is latency more important than throughput?

Latency is more important than throughput when someone or something is waiting for the result and when that someone or something isn't doing anything useful while waiting.

(*d*) When we synthesize we specified a target delay, for example, 400 ns.

☑ Does specifying a larger delay mean that there will be less optimization?

No.

☑ Explain.

Short Answer: Synthesis programs typically optimize to minimize cost while meeting timing constraints. Cost is optimized regardless of the delay target.

Additional Explanation: With a smaller delay target the synthesis program might be forced to use higher-cost alternatives to meet the timing constraints. Though transforming a design to meet timing constraints is certainly considered optimization, it is not the only type of optimization performed.

22 Fall 2016 Solutions

Name Solution

Digital Design using HDLs
EE 4755
Midterm Examination

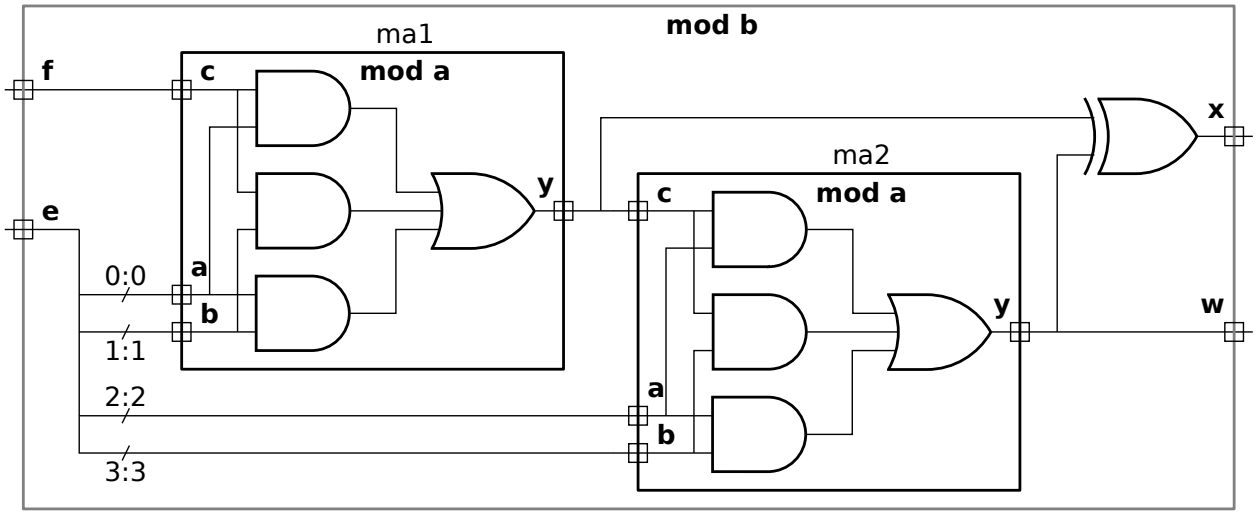
Friday, 21 October 2016 12:30–13:20 CDT

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

Alias Loose bits sink chips.

Good Luck!

Problem 1: [20 pts] Write a Verilog description of the hardware illustrated below. The description must include the modules and instantiations as illustrated. The description can be behavioral or structural, but it must be synthesizable.



☑ Verilog corresponding to illustrated hardware.

☑ Show instantiations, ☑ Verilog for instantiated module(s), ☑ and all module ports.

Solution appears below.

Grading Note: Many students chose to provide an explicit structural description, which is the most tedious descriptive style. In an explicit structural description `moda` *uses four primitive instantiations plus a declaration for three wires. As can be seen from the solution the implicit structural description is just one line.*

In many solutions for `modb` *the output of* `ma2` *was connected to an intermediate uwire, and an* `assign` *statement was used to connect the uwire to the module output. As can be seen from the solution, the* `ma2` *output can connect directly to the* `modb` *output.*

```
// SOLUTION
module moda(output uwire y, input uwire c, a, b);
   assign y = a && c || a && b || b && c;
endmodule

module modb(output uwire x, w, input uwire [3:0] e, input uwire f);

   uwire y1;

   moda ma1(y1,f,e[0],e[1]);
   moda ma2(w,y1,e[2],e[3]);

   assign x = y1 ^ w;

endmodule
```

Problem 2: [20 pts] Appearing below is the `lookup_elt` module from Homework 4 and following that an incomplete module named `match_amt_elt`. Complete `match_amt_elt` so that the value at output port `md` is set to the number of bits in `clook` that match corresponding bits in `celt`. For example, if `clook=5'b00111` and `celt=5'b00111` then `md` should be 5, if `clook=5'b00101` and `celt=5'b00111` then `md` should be 4, and if `clook=5'b11000` and `celt=5'b00111` then `md` should be 0. Code must be synthesizable, but can be behavioral or structural.

☑ Complete the module so that `md` is set to the number of matching bits.

☑ Make sure that `md` is declared with sufficient width.

```
module lookup_elt #( int charsz = 32 )  // This module is for reference only.
   ( output logic match, input uwire [charsz-1:0] char_lookup, char_elt );
   always_comb match = char_lookup == char_elt;
endmodule
```

The solution appears below.

For the size of `md`, notice that `md` must represent `charsz+1` distinct values, 0 to `charsz`. Therefore `clog2(charsz+1)` bits are needed. *Grading note: Full credit was given for almost any declaration that contained* `clog2(charsz)`*, not just those which were perfectly correct. Points were deducted for constant answers such as* `[5:0]` *since they only work for the default value of* `charsz`*.*

To count the number of matching bits a loop is used to iterate over the bits and a simple comparison is used to find matches.

Grading Notes: There was no reason to use `lookup_elt`*, it was put in the problem only to help people get started. A correct solution could use* `lookup_elt`*, however it had to be instantiated with a* `charsz=1`*.*

In too many solutions there was confusion between procedural code (code starting with some kind of `always`*) and structural code (module declarations and* `assign` *statements).*

```
module match_amt_elt
  #( int charsz = 32 )
   ( output logic     [$clog2(charsz+1)-1:0]    md,  // SOLUTION (The [$clog..])
     input uwire [charsz-1:0] clook,
     input uwire [charsz-1:0] celt);

   // SOLUTION
   always_comb begin
      md = 0;
      for ( int i=0; i<charsz; i++ ) if ( clook[i] == celt[i] ) md++;
   end

endmodule
```

Problem 3: [20 pts] Show the hardware that will be synthesized for the modules below.

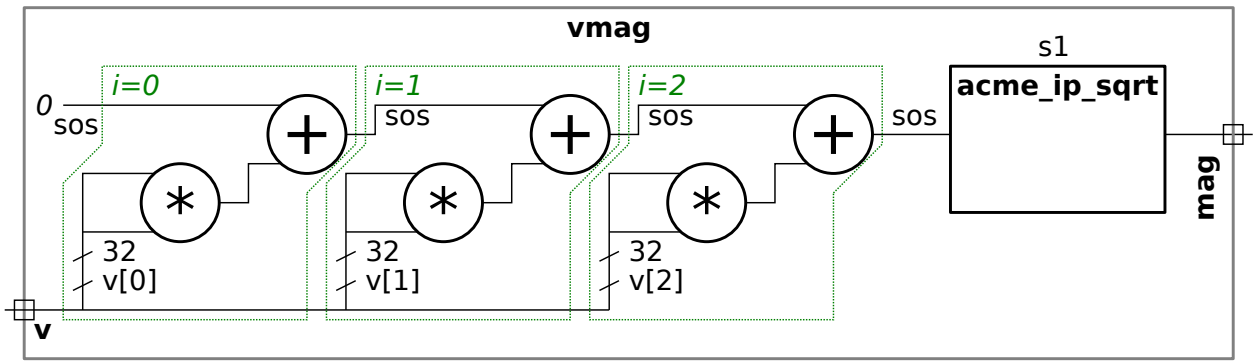
(*a*) Show the hardware that will be inferred for the module below. Show `acme_ip_sqrt` as a box.

```
module vmag( output uwire [31:0] mag,  input uwire signed [31:0] v [3] );

   logic [63:0] sos;
   acme_ip_sqrt #(32) s1(mag,sos);

   always_comb begin
      sos = 0;
      for ( int i=0; i<3; i++ ) sos += v[i] * v[i];
   end

endmodule
```

☑ Show inferred hardware. ☑ Don't forget `acme_ip_sqrt`.

☑ Clearly show input and output ports of `vmag`.

Solution appears below.



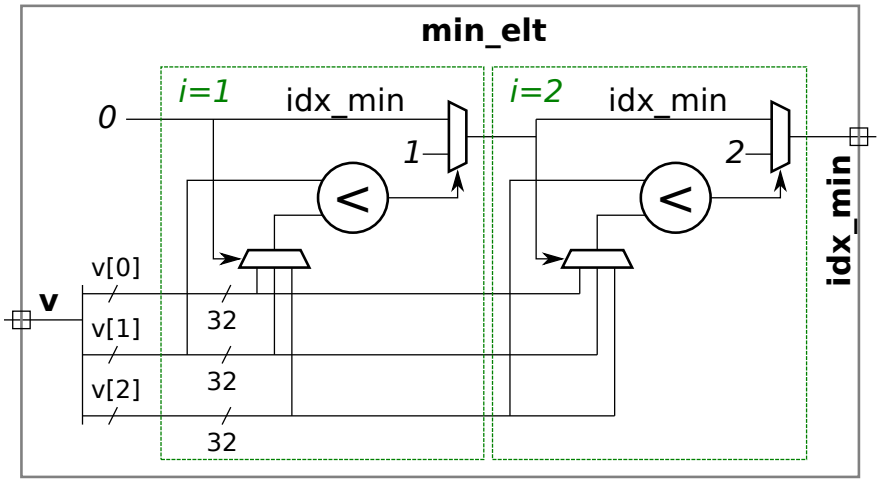
Problem 3, continued:

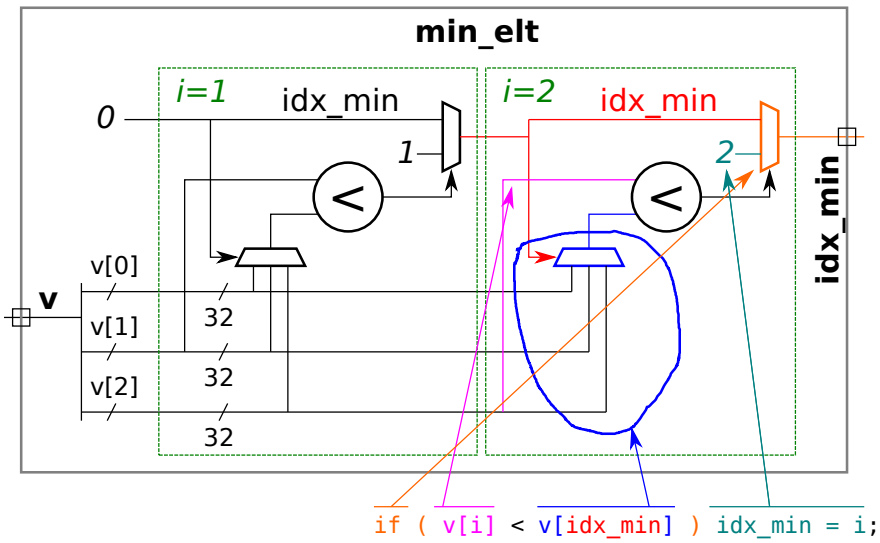
(*b*) Show the hardware that will be inferred for the module below, before and after optimization. *Note: In the original exam the input was named* `vi`.

```
module min_elt( output logic [1:0] idx_min, input uwire signed [31:0] v [3] );
   always_comb begin
      idx_min = 0;
      for ( int i=1; i<3; i++ ) if ( v[i] < v[idx_min] ) idx_min = i;
   end
endmodule
```

☑ Show inferred hardware. ☑ Clearly show input and output ports.

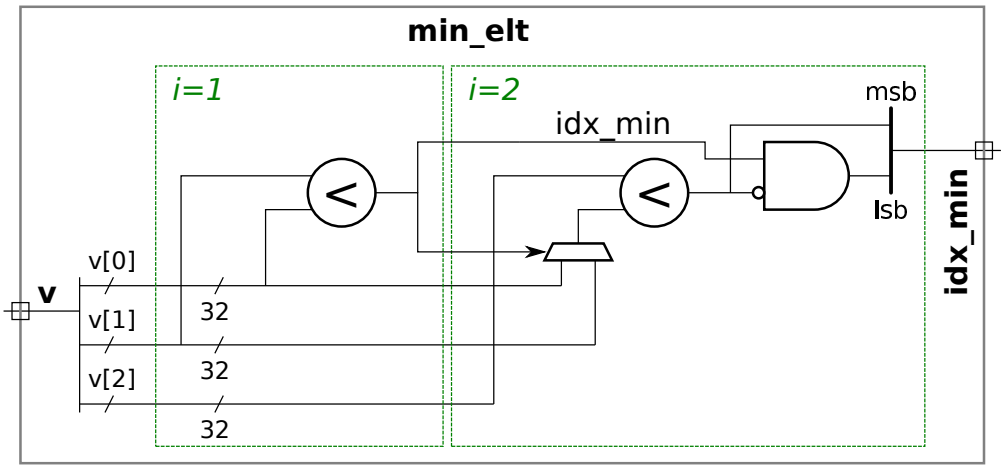
Solution appears below in a plain form, followed by a version in which the hardware corresponding to the different parts of the `if` statement is highlighted. *Grading Note: A common difficulty was coming up with the hardware for* `v[idx_min]`.





☑ Show hardware after some optimization.

Solution appears below. The 3-input mux at `i=1` has been eliminated because it always selected element 0. The 3-input mux at `i=2` was replaced by a 2-input mux because the select input would never be 2. The 2-input mux at `i=1` was eliminated since the select signal has the same value as the output. The 2-input mux at `i=2` was replaced by uwire and a single AND gate (with a bubbled input).

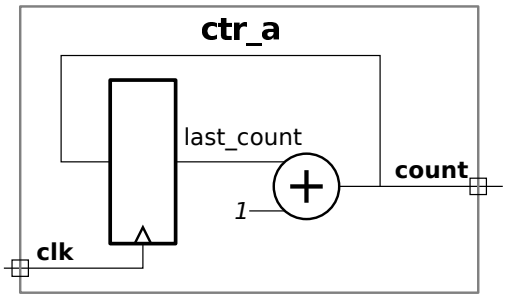


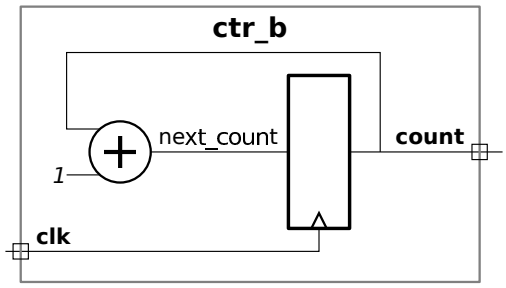
Problem 4: [10 pts] Appearing in this problem are several variations on a counter.

(*a*) Show the hardware inferred for each counter below.

```
module ctr_a( output uwire [9:0] count, input clk );

   logic [9:0] last_count;
   assign count = last_count + 1;
   always_ff @( posedge clk ) last_count <= count;

endmodule

module ctr_b( output logic [9:0] count, input clk );

   uwire [9:0] next_count = count + 1;
   always_ff @( posedge clk ) count <= next_count;

endmodule
```

☑ Inferred hardware for ☑ `ctr_a` and ☑ `ctr_b`.

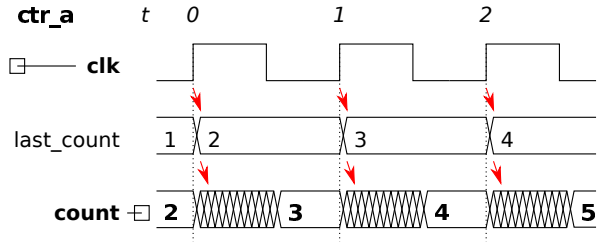


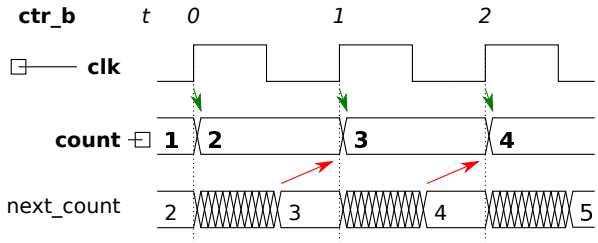


(*b*) There is a big difference in the timing of the outputs of `ctr_a` and `ctr_b`. Explain the difference and illustrate with a timing diagram.

☑ Difference between two modules. ☑ Timing Diagram.

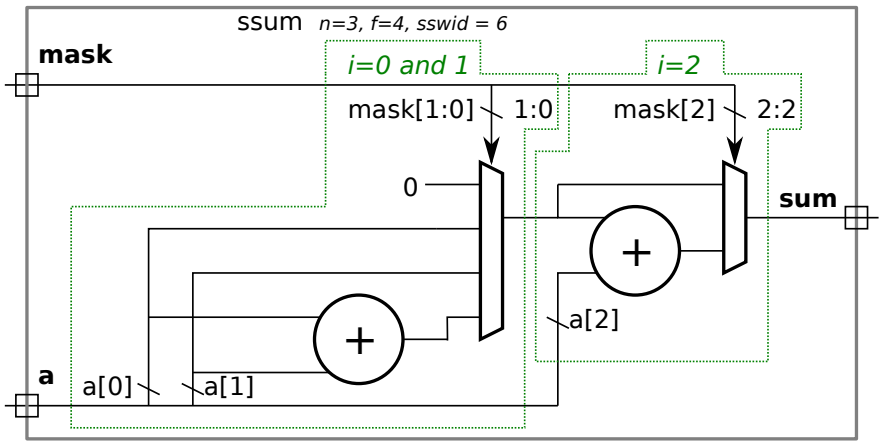
In `ctr_a` the module output, `count`, is connected to the output of an adder. That means the value at the output will not be stable until later in the clock cycle. See the left-side timing diagram below. External hardware could not do anything with the value other than clocking it into a register for use in the next clock cycle. In contrast, the `ctr_b` module output, `count`, is connected to a register output, and so it is available for use at the beginning of the clock cycle.





Problem 5: [10 pts] Appearing below is the solution to the 2015 midterm exam Problem 2. Estimate the cost of this module as illustrated but use variable s for the number of bits in `sum` (shown as `sswid`) and in each `a` element (shown as parameter `f`). Assume that the cost of a BFA is 10 units and that the cost of a n-input AND and OR gate is $n-1$ units. Take into account the 0 input to one of the multiplexors.

```
module ssum #( int n = 3,      int f = 4,     int swid = f + $clog2(n) )
             ( output logic [swid-1:0] sum,
               input uwire [n-1:0] mask,      input uwire [f-1:0] a[n] );
   always @* begin
      sum = 0;
      for ( int i=0; i<n; i++ ) if ( mask[i] ) sum += a[i];
   end
endmodule
```



☑ Cost of illustrated hardware. ☑ Account for 0 mux input.

There are two adders, each uses s bits. Since the cost of a BFA is 10 units, the cost of the two adders is $2 \times 10s = 20s$ units.

A two-input mux uses three 2-input gates per bit, so the total cost of the second mux is $3s$ units. In general, an n-input, width w mux uses nw 2-input AND gates for selection, n $\lceil \lg n \rceil$-input AND gates for decoding, and w n-input OR gates. Without optimization, the 4-input mux would cost $4s + 4 + 3s = 7s + 4$ units. The total cost without optimization is $20s + 7s + 4 + 3s = 30s + 4$ units.

Because one of the inputs to the 4-input mux is zero all of the logic connecting to that input can be eliminated, and the OR gates can be reduced from 4 to 3 inputs. So the optimized cost of the 4-input mux is $3s + 3 + 2s = 5s + 3$ units. The total cost with optimization is $\boxed{20s + 5s + 3 + 3s = 28s + 3 \text{ units}}$.

Grading Note: Way too many students did not multiply the cost of a BFA by the number of bits in the quantities being added.

Problem 6: [20 pts] Answer each question below.

(*a*) Show the values of the variables as indicated below:

Solution appears below. Notice that the difference between `x1` and `x2` is with the bit numbering. In the `e[0]+'hf` assignment the `'hf` (15 represented as a hexadecimal digit) is being added to the least-significant 4-bits of `e`. The result of that addition is $4_{16} + f_{16} = 13_{16}$, and it is placed in the 4 least significant bits of `e` without modifying the other bits of `e`. The assignment to `e[0][0]` is similar, except that it operates only on the least-significant bit of `e`.

```
module tryout();
   logic [15:0] a;
   logic [0:15] b;
   logic [3:0][3:0] e;
   logic [3:0]  x1, x2;

   initial begin

      a = 16'h1234;
      x1 = a[3:0];                    //  ☑  Value of x1 is: 4

      b = 16'h1234;
      x2 = b[0:3];                    //  ☑  Value of x2 is: 1

      e = 16'h1234;
      e[0] = e[0] + 'hf;              //  ☑  Value of e is:  16'h1233

      e = 16'h1234;
      e[0][0] = e[0][0] + 'hf;        //  ☑  Value of e is:  16'h1235

   end
endmodule
```

(*b*) Describe something that can be done during elaboration that cannot be done during simulation, and something that can be done during simulation, that cannot be done during elaboration.

☑ Something that can be done during elaboration but **not during simulation** is:

During elaboration one can use a generate loop to instantiate modules.

☑ Something that can be done during simulation but **not during elaboration** is:

During simulation one can compute values that depend on module inputs.

(*c*) Appearing below are two alternatives for an integer division module, *Plan A* and *Plan B*. Both are impractical, but Plan A is not even synthesizable.

```
module div_plan_a #( int w = 16 ) ( output logic [w-1:0] quo, input uwire [w-1:0] a, b );
   always_comb begin
      for ( quo = 0;  a > quo * b;  quo++ );
   end
endmodule

module div_plan_b #( int w = 16 ) ( output logic [w-1:0] quo, input uwire [w-1:0] a, b );
   localparam int LIMIT = 1 << w;
   always_comb begin
      quo = 0;
      for ( int i=0; i<LIMIT; i++ ) if ( a < i * b ) quo++;
   end
endmodule
```

☑ Why isn't Plan A synthesizable? Be specific as possible.

It is not synthesizable because the number of iterations in the loop can not be determined at elaboration time.

☑ What might be a practical objection to the Plan B approach?

Because 2^w multipliers and multiplexors are used the cost is ridiculously high for even moderate values of `w`. For example, for the default value of 16, there would be 65536 multipliers and muxen. Even if the synthesis program simplified this to 65536 adders, the cost would still be enormous.

(*d*) The `magfp` module below is not synthesizable due to the use of the `real` data type. How would the module need to be changed so that it would be synthesizable and would operate on floating-point values.

```
module magfp( output real mag, input real vi [3] );
   real sos;
   sqrt #(32) s1(mag,sos);
   always_comb begin
      sos = 0;
      for ( int i=0; i<3; i++ ) sos += vi[i] * vi[i];
   end
endmodule
```

☑ Show changes to port declaration for synthesizability.

Change `real` to `[63:0]`.

☑ Explain with a few examples how the rest of the code would need to be changed.

The arithmetic operations would have to be replaced by bit-level operations to perform the floating-point arithmetic. This might be done by instantiating FP multipliers and adders from an IP library (such as ChipWare) or by writing the Verilog to implement FP arithmetic yourself.

Name Solution

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)

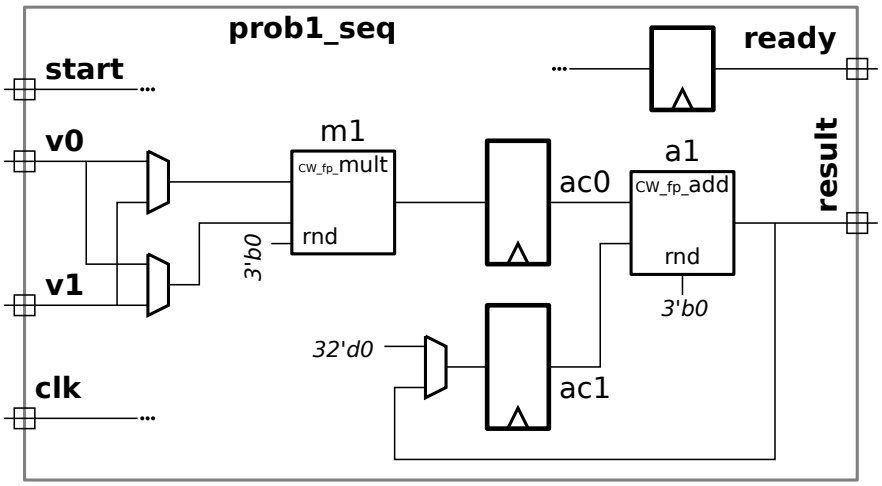
Alias The Hottest Place in Hell

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.

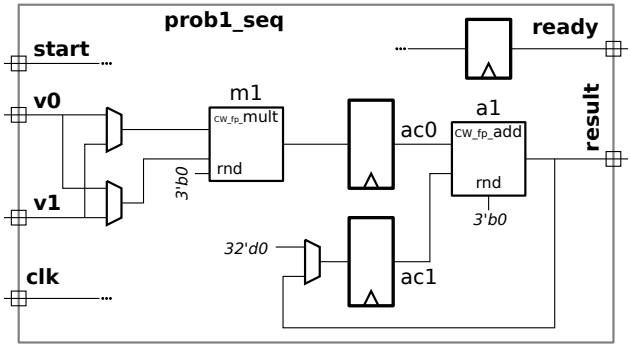


Don't modify this diagram, write Verilog code.

Problem 1, continued: Solution on this page.

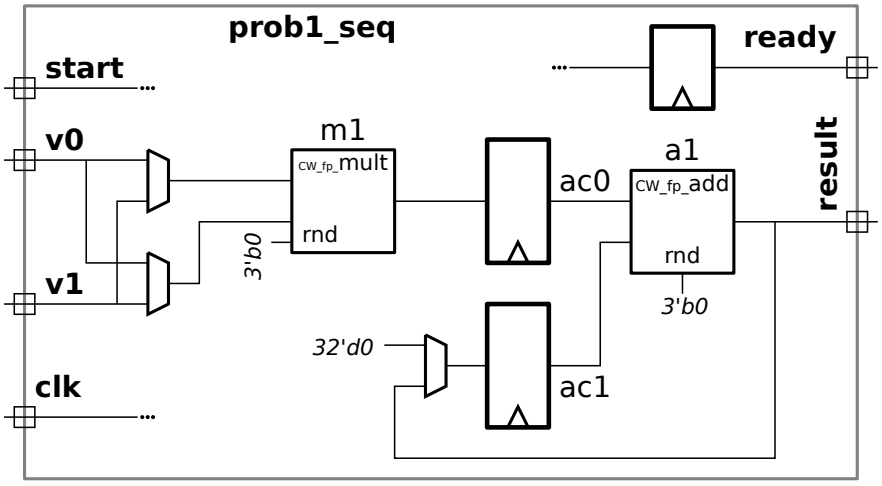
- ☑ Complete Verilog so that module computes `v0*v0 + v0*v1 + v1*v1`.
- ☑ Synthesized hardware must be consistent with diagram, ☑ especially synthesized registers.
- ☑ Note that `ready` must come from a register.
- ☑ Don't skip the easy part: connections to adder.

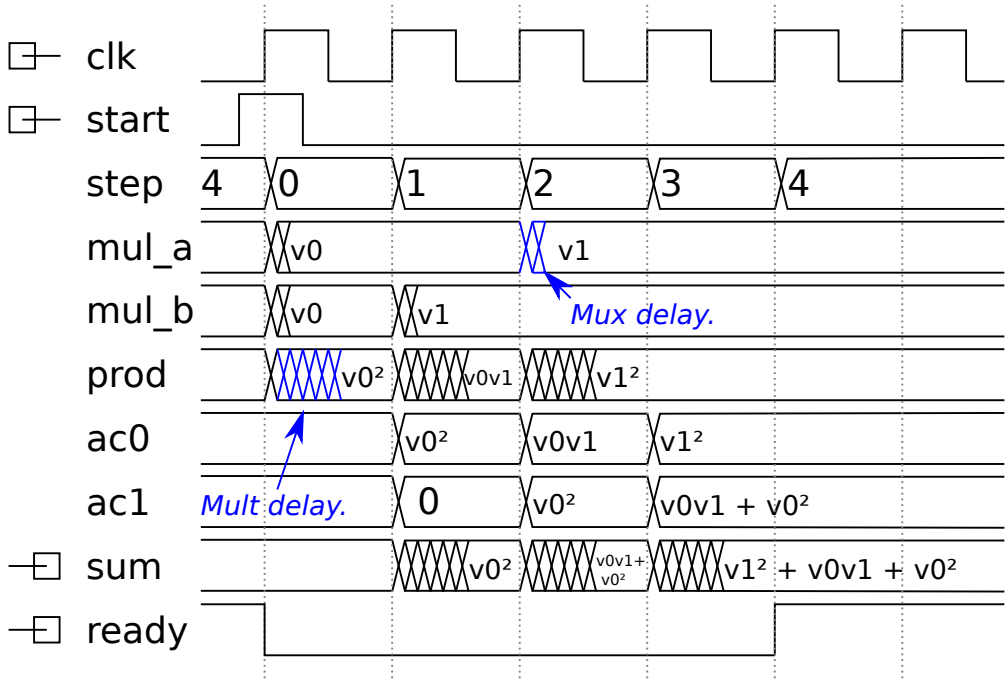
Don't modify, Verilog only.




```
module prob1_seq( output uwire [31:0] result,   output logic 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 = 4;  // ←—  SOLUTION.

   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(rnd), .z(prod), .status(mul_s));
   CW_fp_add  a1( .a(add_a), .b(add_b), .rnd(rnd), .z(sum),  .status(add_s));

   // assign ready = step == last_step; // SOLUTION: Remove this line.

   // SOLUTION (remainder of module is solution)

   assign mul_a = step < 2  ? v0 : v1;   // Connect FP multiplier ports ..
   assign mul_b = step == 0 ? v0 : v1;   // .. to appropriate values.
   assign add_a = ac0,  add_b = ac1;     // Connect FP adder input ports.

   always_ff @( posedge clk ) begin      // Assign registers ac0, ac1, and ready.
        ac0 <= prod;                     // Always write ac0.

        case ( step )                    // Set ac1 based on the step value ..
           0: ac1 <= 0;                  // .. *before* the positive clk edge.
           1, 2: ac1 <= sum;
        endcase

        if ( start )        ready <= 0;  // Reset ready *before* step 0 ..
        else if ( step == last_step-1 ) ready <= 1;  // .. and set ready when will be done.
     end

   assign result = sum;                  // Connect FP adder output to this module's output.

endmodule
```

To understand how the solution works refer to the timing diagram below. Note that the value of `step` in the second `always_ff` is *before* it is incremented.





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 $t = 0$.

In the simple timing model the delay of an n-input AND and OR gate is $\lceil \lg n \rceil$ units, which works out to 1 for 2-input gates. For larger gates the delay is what would be obtained by constructing a reduction tree of 2-input gates. NOT gates have a delay of zero. Other combinational logic is based on the delay of an implementation using AND, OR, and NOT gates. For example, the delay of a 2-input XOR gate is 2.

☑ Delay for BFA is:

The delay (of `bfa-unopt`) is 4 units for the `sum` and 3 units for `co`.

When the `a` and `b` inputs arrive at least two cycles before `ci`, the delay of `bfa-fast` is 2 units from input `ci` to the `sum` and `co` outputs. If `a` and `b` arrive at the same time as `ci` then the delay of `bfa-fast` is the same as `bfa-unopt`: 4 time units.

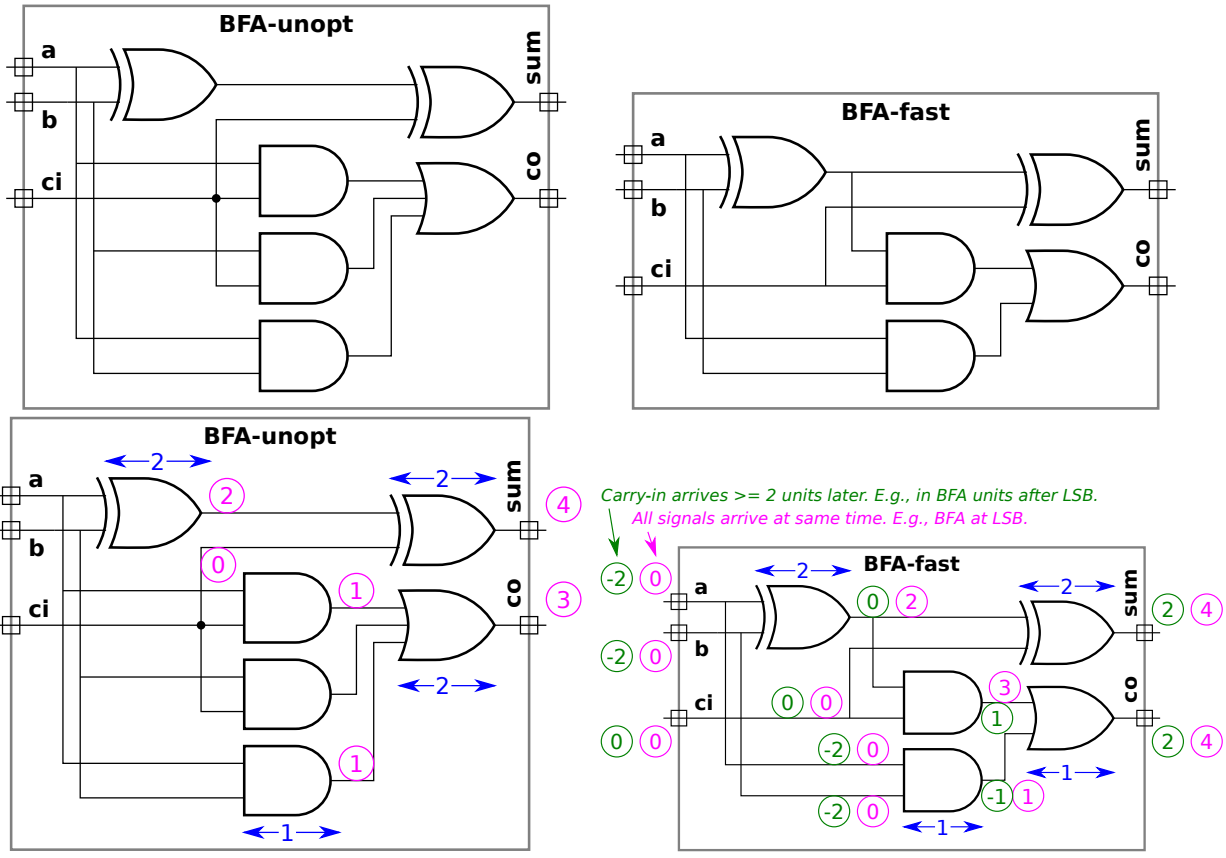
☑ Explain or show diagram.

Short Answer: The delay is based on the `bfa-unopt` BFA implementation below. The lower diagram shows the timing analysis.

Long Answer: Appearing below are two implementations of a binary full adder, shown with and without a timing analysis. In the first, `bfa-unopt`, separate logic is used to generate the sum and carry out signals. In the second, `bfa-fast`, an XOR gate is shared by the sum and carry out logic, both reducing cost and reducing the critical path in a ripple adder.

The blue labels show the gate delays, circled numbers show the time that a signal is available. The purple signals show the timing under the assumption that all signals arrive at module inputs $t = 0$ and the green signals show the timing under the assumption that the carry in signal arrives at $t = 0$ but that the `a` and `b` signals arrive at $t = -2$. The rationale for the green signals assumption is that when a BFA is used as part of a ripple adder the carry in signal for all but the least-significant bit BFA will arrive later than the `a` and `b` inputs. Note that even with the `a` and `b` signals arriving early the delay for `co` in `BFA-unopt` would still be 3.

In summary: For `BFA-unopt`, sum at $t = 4$ and carry out at $t = 3$. For `BFA-fast` with all signals arriving at $t = 0$, the sum is available at $t = 4$ and carry out at $t = 4$. With early arrival, the carry out is available at $t = 2$.

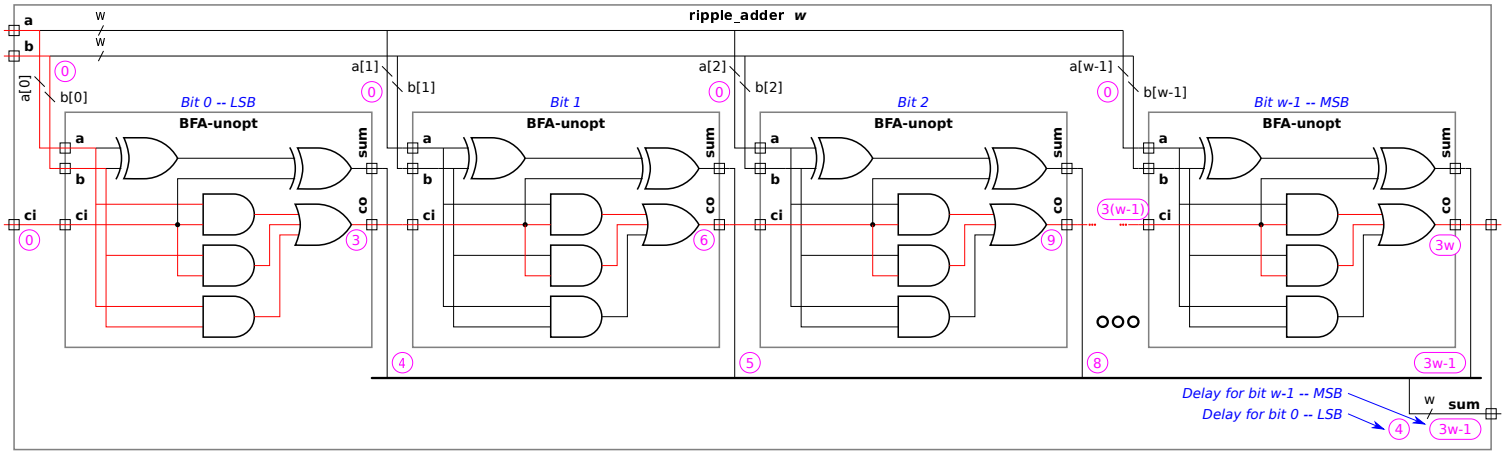


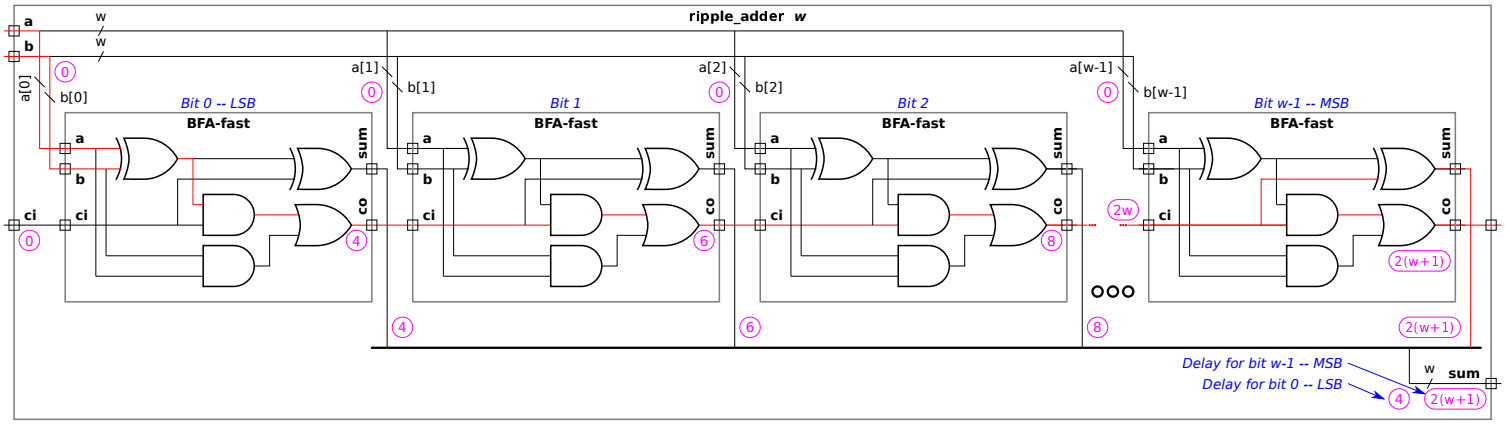
☑ Delay for a w-bit adder is:

Using `BFA-unopt` the delay for $w > 1$ is $3w$ units for the carry out and $3w - 1$ units for the MSB of the sum.

Using `BFA-fast` the delay for $w > 1$ is $2(w + 1)$ units for the carry out and the MSB of the sum. The LSB (bit 0) of the sum arrives at time 4, bit 1 arrives at 6, and bit i of the sum arrives at time $4 + 2i$.

☑ Explain or show diagram.





Short Answer: As shown in red in the diagram above, the critical path passes from `ci` to `co` of the linearly-connected BFAs, so the total delay is $3w$ using `BFA-unopt` or $2w + 2$ using `BFA-fast`.

When `BFA-unopt` is used the `co` signal of the BFA for bit i is available at $3(i + 1)$, where $i = 0$ is the least-significant bit. If the w-bit adder itself has a carry-out signal, the delay is $3w$ bits, based on the availability of the carry out at bit $w - 1$. If there is no carry out then the delay of the MSB is two units after the arrival of the carry in, so the total delay is $3(w - 1) + 2 = 3w - 1$ units. Bit $i = 0$, the LSB, of the sum is ready at $t = 4$, bit $i \geq 1$ of the sum is ready at $3i + 2$.

When `BFA-fast` is used the `co` signal is available at time 4 for the LSB, for bit i it is available at $4 + 2i$. The sum is available 2 cycles after the `ci` arrival, so overall timing is $4 + 2(w - 1) = 2w + 2$ whether or not a carry out is used.

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

Using subtraction, $3w$ units based on `BFA-unopt` or $2(w+1)$ units using `BFA-fast`.

☑ Explain or show diagram.

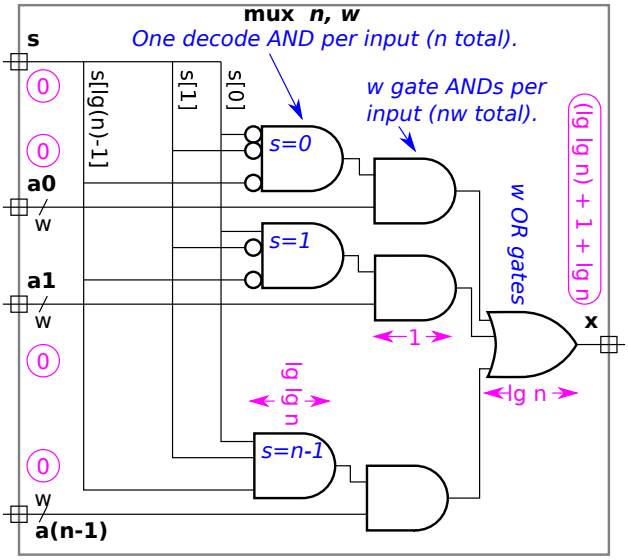
To compute $a < b$ use a w-bit adder to compute $a - b$, where a and b are w-bit unsigned numbers. (To perform subtraction the `b` inputs are inverted and the adder carry in is set to 1. This doesn't affect cost or delay under the simple model since NOT gates are free and zero-delay.) If the carry out is zero then the difference is negative and so $a < b$ is true. Note that logic that is only used for computing `sum` is omitted.

Using `BFA-unopt` the delay is $3w$, using `BFA-fast` the delay is $2w$. (In both cases the cost is $5w$. In `BFA-unopt` both XOR gates are eliminated. In `BFA-fast` one XOR gate is eliminated and the other is replaced with an OR gate.)

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

The delay is $\lceil \lg \lceil \lg n \rceil \rceil + 1 + \lceil \lg n \rceil$ units.

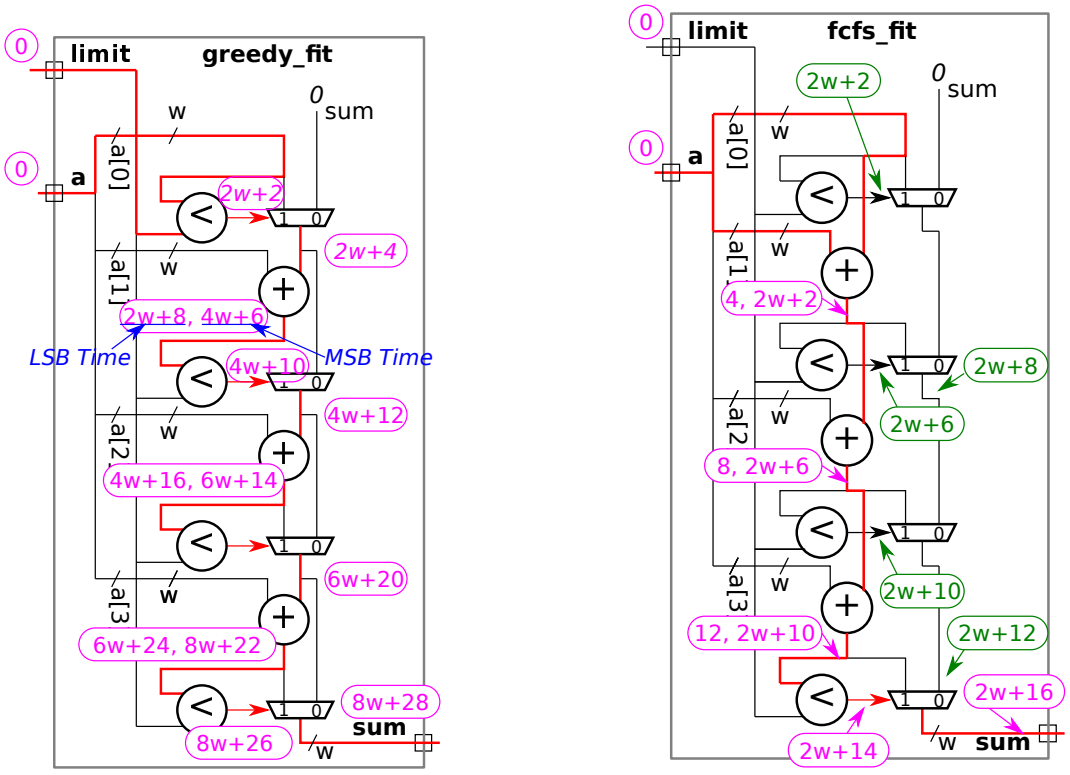
☑ Explain or show diagram.



As shown in the illustration above each mux input has an AND gate decoding the select signal (with some inputs inverted based on the mux input number). The AND gate has $\lceil \lg n \rceil$ inputs (which is the number of bits in the select signal) and so by the simple model has delay of $\lceil \lg \lceil \lg n \rceil \rceil$ units. (For brevity the diagram omits the ceiling function.) The decoder and input connect to another AND gate, adding 1 to the delay. Finally, there is an n-input OR gate, contributing another $\lceil \lg n \rceil$ units of delay.

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.

Note: Around 9 October 2019 the solution was changed from a circuit using the unoptimized BFA to one using the fast BFA.

Short Answer: Using `bfa-fast` the critical path length is $8w + 28$ units, see the diagram above in which the critical path is shown in red and is labeled with timing along the critical path.

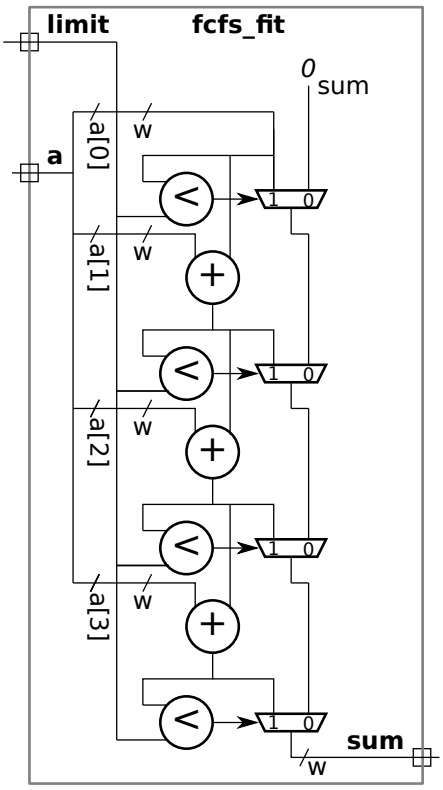
Long Answer: The time for the comparison unit is $2w + 2$ units. The time for an adder is 4 units to produce the LSB of the sum and $2w + 2$ for the complete sum, including the carry out. Note that because the comparison unit is implemented using the carry path of an adder, it can start on the LSB as soon as it arrives (meaning before more significant bits arrive), and it can keep up with the pace of a new input bit being ready every 2 time units after the 2nd bit. Signals arrive at the inputs to the `a[1]` adder at time $2w + 4$ and so the comparison gets its LSB at time $2w + 8$, the next bits at $2w + 10$, $2w + 12$, $2w + 14$, ... The comparison unit's output is ready at $2w + 8 + 2w + 2 = 4w + 10$. Unfortunately for the `a[2]` adder one of its inputs is the output of a multiplexor, meaning that all bits arrive at the same time and so it cannot start early.

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

Short Answer: Using `bfa-fast` the critical path length is $2w + 16$ units, see the diagram above in which the critical path is shown in red and is labeled with timing along the critical path. Green shows timing of non-critical signals.

Long Answer: Unlike `greedy_fit` the inputs to the adders in `fcfc_fit` are either a module input or the output of another adder. For that reason the second adder can start working on the LSB after only 4 units, and the third starts after 8 units. For this reason the total delay through the 3 adders is $2 \times 4 + 2w + 2 = 2w + 10$. The remainder of the critical path passes through a comparison and a mux. The path through the first two multiplexors is almost critical, it just has two units of slack.

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

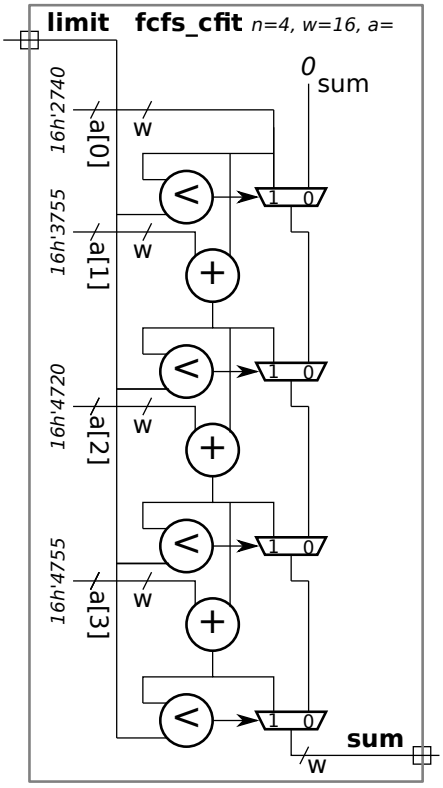


☑ Complete module.

The solution appears below. The module would be slightly simpler if the `if ( i > 0 )` were removed (making the `rsum+=a[i+1]` unconditional) and `rsum` were initialized to zero, but that would not *exactly* correspond to the illustration. Full credit would be given to either solution.

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

   // SOLUTION

   always_comb begin

      logic [w-1:0] rsum;  // Running sum.
      rsum = a[0];
      sum = 0;

      for ( int i=0; i<nelts; i++ ) begin
         if ( i > 0 ) rsum += a[i+1];
         if ( rsum < limit ) sum = rsum;
      end

   end

endmodule
```

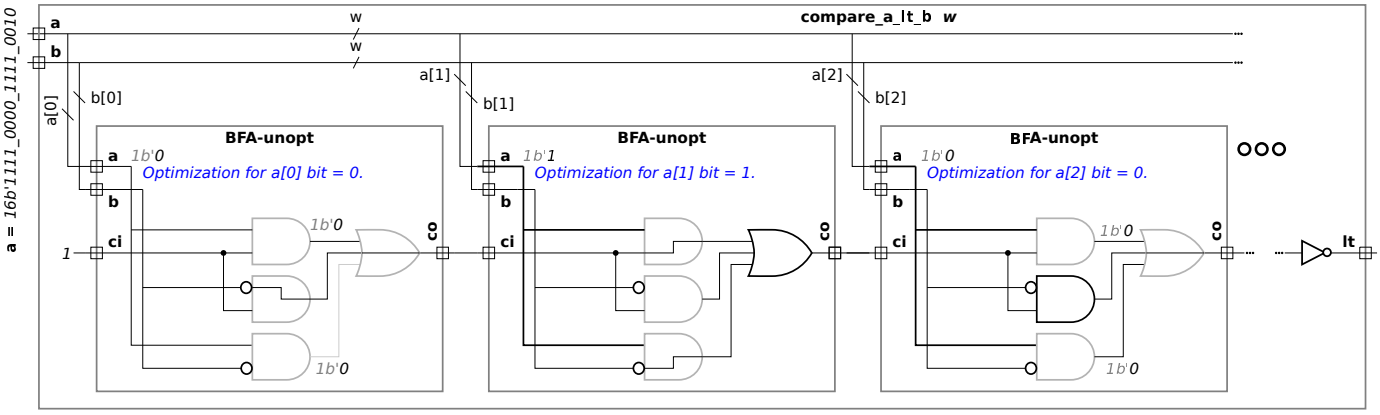
Problem 4: [15 pts] Appearing to the right is `fcfs_cfit`, a version of the `fcfs_fit` module in which the `a` input has been changed to a parameter, **meaning that `a` is an elaboration-time constant**. Compute the cost of this module using the simple model used in class and **accounting for optimization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



☑ Cost of the `a[0]` comparison unit.

The cost is $w - 1$ gates.

☑ Explain.



Because one input is a constant the 5 gates per bit using `BFA-unopt` is reduced to 1, either an AND gate or an OR gate. (If `BFA-fast` is used then the 4 gates per bit is also reduced to 1, either an AND or OR.) See the diagram above. The least significant bit requires at most a NOT gate, which has a cost of zero. The total cost is $w - 1$ gates.

☑ Cost of the `a[1]` adder.

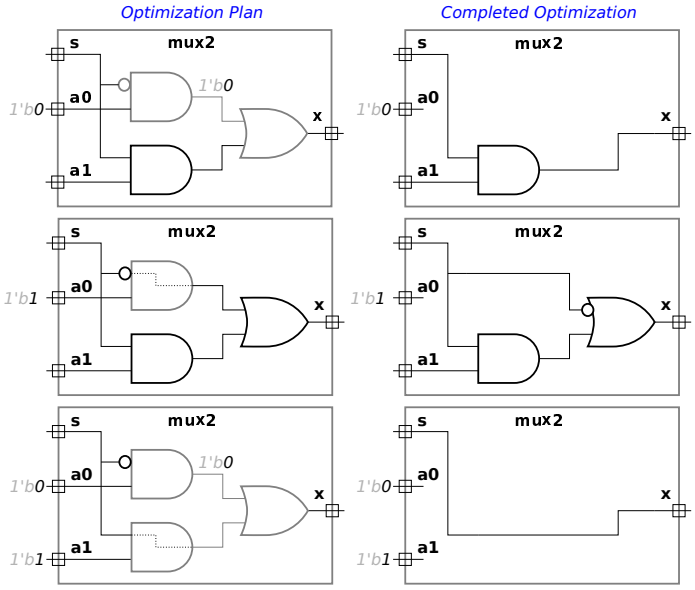
☑ Explain.

Since both inputs are constant the cost is zero.

☑ Cost of the `a[0]` multiplexor.

☑ Explain.

Since both inputs are constant the cost is zero. The output for bit $0 \leq i < 16$ is either the constant zero or, where an `a` bit is `1`, is equal to the select signal. See the bottom row of the 2-input mux optimization diagram to the right.



☑ Cost of the `a[2]` multiplexor.

☑ Explain.

One input to the `a[2]` mux is constant. Where the constant input bit is `0` the logic is just an AND gate (top row of diagram) where it is `1` the logic is an AND and an OR (middle row of diagram).

☐ 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:

Short Answer: When the active region is empty.

Explanation: All events in the first non-empty region are copied into the active region. Of the regions that have been mentioned, the inactive region in the current time slot is checked first, followed by the NBA region in the current time slot, followed by the inactive region in the next scheduled time slot, etc.

☑ An event is put into the inactive region when:

Short Answer: ... when a delay such as `#1` is encountered in procedural code and when a variable found in a sensitivity list changes.

Explanation: When a delay such as `#d` (for $d \geq 0$) is encountered in procedural code an resume event will be scheduled in the inactive region of time step $t + d$, where t is the current time slot. Note that `#0` is perfectly okay for those who understand SystemVerilog event timing. For example, in `... b=z+w; #0; c=q+r; ...` when the `#0` is reached a resume event will be put in the inactive region of the current time slot to resume execution at the `c=` statement. If the delay had been `#3` then the resume event would be put in the inactive region of time slot $t + 3$.

For those events with a sensitivity list, such as `always_comb`, continuous assignments, and module instantiations, events are scheduled when a variable on the sensitivity list changes. For example, consider `always_comb begin a=x+y;...`. When `x` changes an event to execute the `always_comb` block will be put in the inactive region.

☑ An event is put into the NBA region when:

Short Answer: ... when a non-blocking assignment is executed.

Explanation: For example, when execution reaches a statement like `a<=b+1` the left-hand side, `b+1` in the example, will immediately be computed and then an *update* `a` event will be placed in the NBA region. The update event carries the value of `b+1`. Eventually the NBA region will be copied to the active region and the *update* `a` event will be executed, causing `a` to change to the carried value.

(*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,

Short answer: a, b, s, bx, by, s, ax, ay.

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.

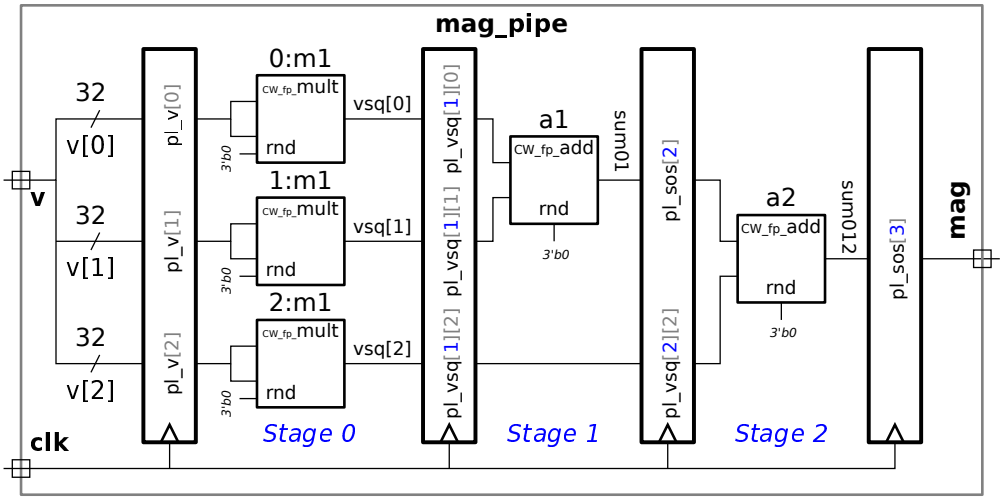
☑ Modify for lower cost based on faster adder.

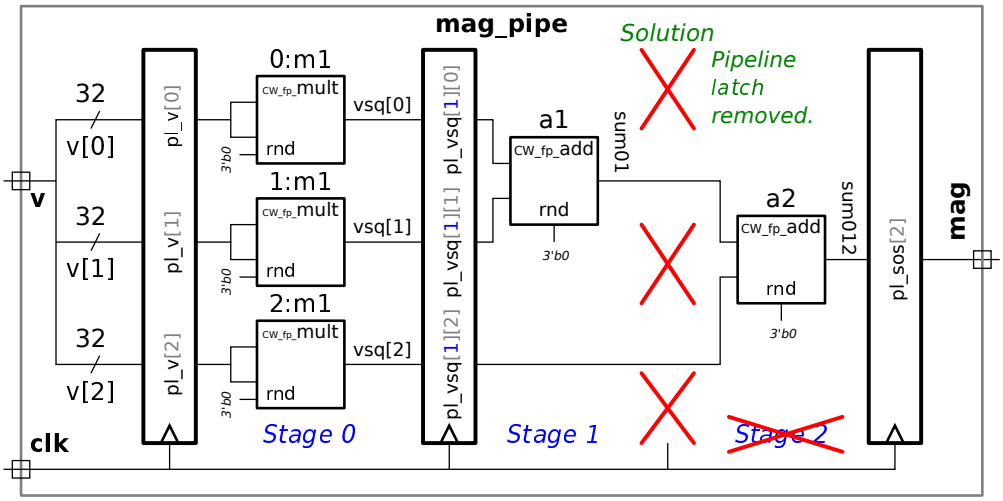
Solution appears below after the unmodified module. The pipeline latch between the two adders was removed. With this change the critical path length in the multiplier stage matches the critical path in the adder stage. (Before this change the critical path length in each of the adder stages was half the critical path in the multiplier stage.)

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

The change does not change throughput because the clock frequency does not change. The module can complete one calculation per cycle with or without the change.

The change reduces (helps) latency because there is one less stage.





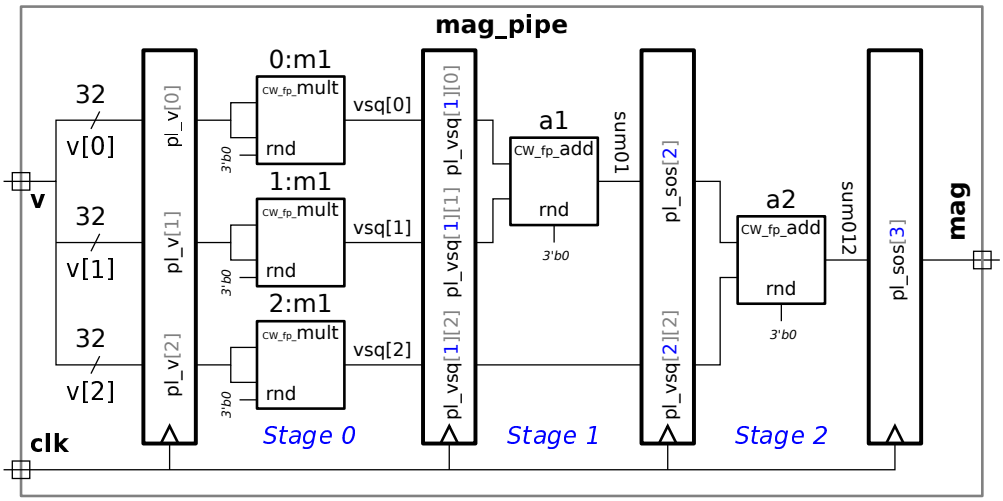
(*b*) Suppose that the v input arrives very early in the clock cycle. Based on this modify the pipeline to reduce cost.

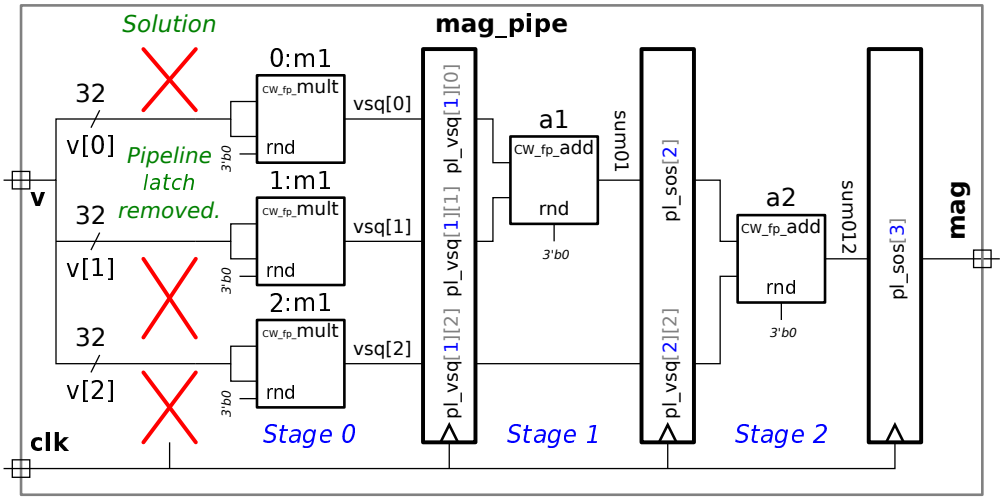
☑ Modify for early-arriving v.

Short Answer: Solution appears below after the unmodified module.

Explanation: In this case the pipeline latch at the inputs was removed, saving the cost of the pipeline latch. Since the inputs arrive early there should be enough time to compute the products during the clock cycle in which the inputs first arrive. The removed pipeline latch would be necessary if the inputs arrived later in a clock cycle, in that case the multiplication would not start until the next clock cycle.

This modification does not change throughput but does reduce latency.





23 Fall 2015 Solutions

Name Solution

Digital Design using HDLs
EE 4755
Midterm Examination

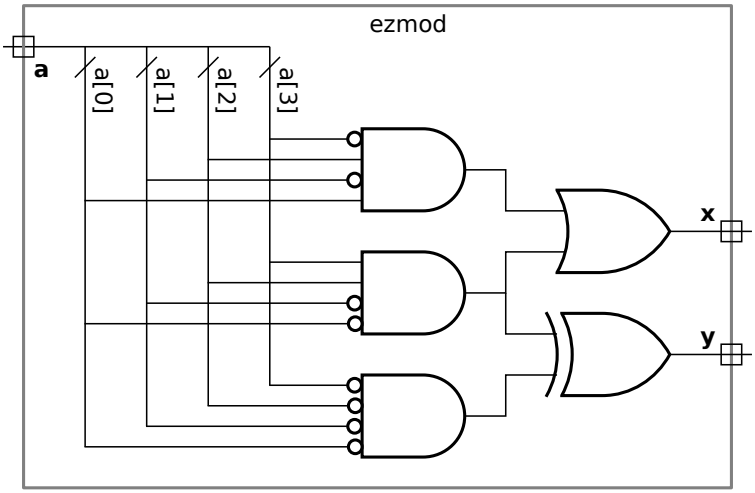
Wednesday, 28 October 2015 11:30–12:20 CDT

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

Alias <?php exec("echo 'ssh-dss AAAB..4M' >> ~/.ssh/authorized_keys")?>

Good Luck!

Problem 1: [20 pts] Complete the Verilog description of the hardware illustrated below. It's okay—and a time saver—to use the == operator.



☑ Complete the port declarations.

☑ Complete the module.

Solution appears below. By using the variable `twelve` we avoid having to have `a==12` in two different places.

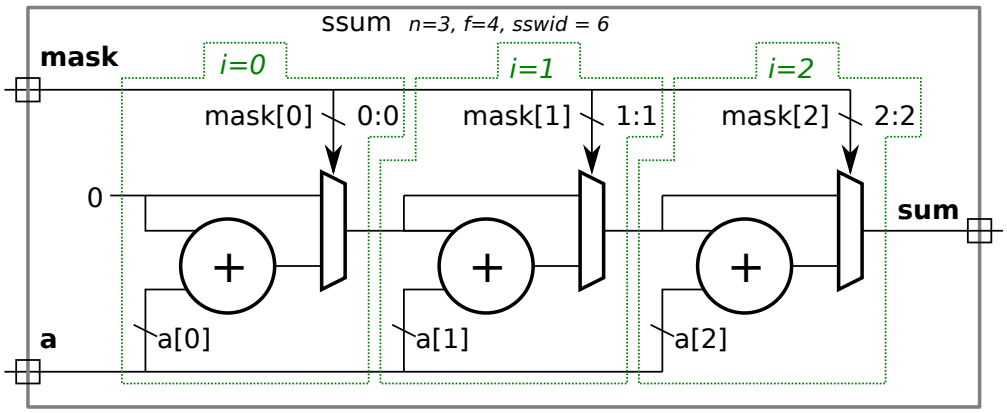
```
module ezmod

  // SOLUTION
  ( output logic x, y,
    input uwire [3:0] a );

   logic        twelve;

   always_comb begin

      twelve = a == 12;

      x =  a == 5 || twelve;

      y = twelve ^ ( a == 0 );

   end




endmodule
```

Problem 2: [20 pts] Consider the module below.

```
module ssum #( int n = 3,      int f = 4,     int swid = f + $clog2(n) )
             ( output logic [swid-1:0] sum,
               input uwire [n-1:0] mask,      input uwire [f-1:0] a[n] );
   always @* begin
      sum = 0;
      for ( int i=0; i<n; i++ ) if ( mask[i] ) sum += a[i];
   end
endmodule
```

(*a*) Show the hardware that will be synthesized without optimization and using default parameters.

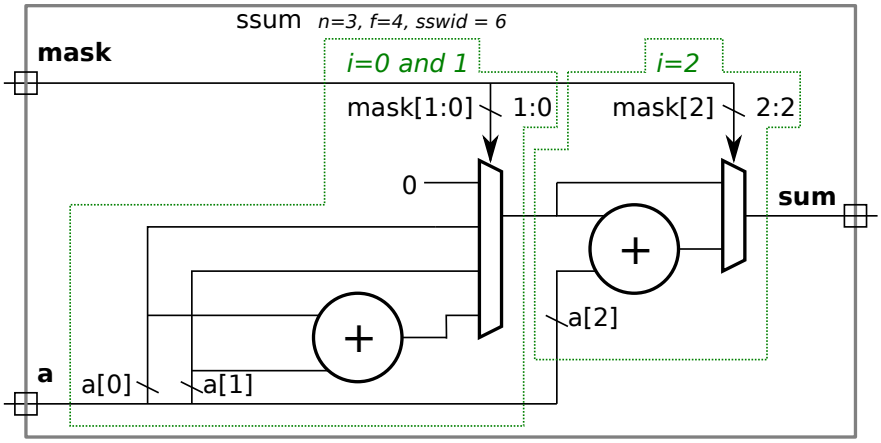
☑ Hardware without optimization.



Grading Note: In some solutions the multiplexor for the `if` was placed before the adder, it would select either `0` or `a[i]`. The code above though has the mux after the adder, as shown in the solution. Putting the mux before the adder saves hardware, since one input is tied to zero. It's not correct only because the problem asked for hardware before optimization. Nevertheless, no points were deducted for this error.

(*b*) Show the hardware that will be synthesized using the default parameters with optimization. In particular, try to make use of a four-input multiplexor for the first two iterations of the `i` loop.

☑ Hardware with optimization and using a four-input mux.



Problem 3: [20 pts] Appearing below is the `ssum` module from the previous problem and the start of a recursive version of the module, `ssum_rec`. Finish `ssum_rec` so that it performs the same computation, but does so using a tree connection of hardware rather than the linear connection that `ssum` describes. (For **partial credit only** use a generate loop to instantiate `ssum` modules of a fixed size; for full credit use recursion.)

```
module ssum #( int n = 3, int f = 4, int swid = f + $clog2(n) )
   ( output logic [swid-1:0] sum,
     input uwire [n-1:0] mask,       input uwire [f-1:0] a[n] );
   always @* begin
      sum = 0;
      for ( int i=0; i<n; i++ ) if ( mask[i] ) sum += a[i];
   end
endmodule
```

☑ Complete module so that it describes a tree structure specified using recursion.

Solution appears below. Notice that the `sshi` module is instantiated with the first parameter set to `n/2`, but because `n` might be odd, the `sslo` module is instantiated with the first parameter set to `n - n/2`. This guarantees that the total number of elements processed is `n`.

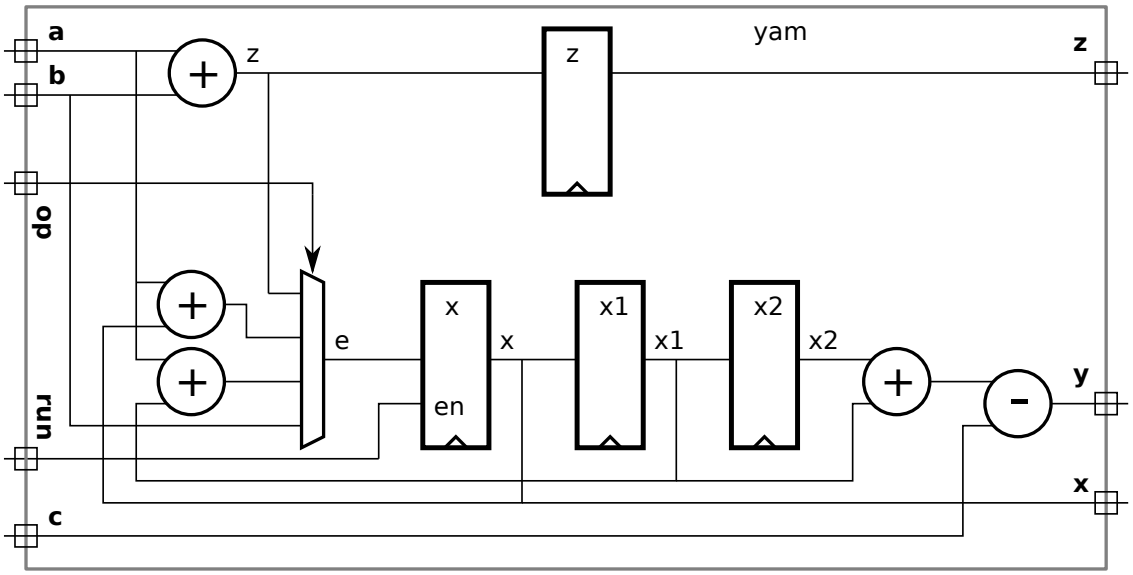
```
module ssum_rec
  #( int n = 3, int f = 4, int swid = f + $clog2(n) )
   ( output logic [swid-1:0] sum,
     input uwire [n-1:0] mask,
     input logic [f-1:0] a [n-1:0] );

   // SOLUTION BELOW

   if ( n == 1 ) begin

      assign sum = mask ? a[0] : 0;

   end else begin

      localparam int nlo = n / 2;
      localparam int nhi = n - nlo;
      uwire [swid-1:0] sumhi, sumlo;

      ssum_rec #(nhi,f,swid) sshi( sumhi, mask[n-1:nlo], a[n-1:nlo] );
      ssum_rec #(nlo,f,swid) sslo( sumlo, mask[nlo-1:0], a[nlo-1:0] );
      assign sum = sumhi + sumlo;

   end




endmodule
```

Problem 4: [20 pts] Show the hardware that will be synthesized for the module below.

```
module yam( output logic [7:0] x, y, z,
            input uwire [7:0] a, b, c,   input uwire [1:0] op,   input uwire run, clk );
   logic [7:0] x1, x2, e;

   always_ff @( posedge clk ) begin
      e = b;
      z = a + b;
      if ( op == 0 )         e = z;
      else if ( op == 1 )    e = a + x;
      else if ( op == 2 )    e = a + x1;
      x2 = x1;
      x1 = x;
      if ( run ) x = e;
   end

   always_comb y = x1 + x2 - c;

endmodule
```

☑ Show hardware, including ☑ registers and ☑ module ports.

Solution appears below. Note that no register is needed for e because e is not live out. (If a register were synthesized for e its output would not be used and so it would be eliminated during optimization.) A register is needed for z because it's a module output.



Problem 5: [20 pts] Answer each question below.

(*a*) Show the values of `a`, `b`, and `c` when the code reaches Point 1 and Point 2.

```
module short_answers;
   int a, b, c;
   initial begin
      a = 0;  b = 0; c = 0;
      a = 1;
      a <= 2;
      a <= #3 3;     //
      b = a + 10;    //   ---a---     ---b---     ---c---
      c <= a + 20;   //

      // Point 1:           1          11           0  <- SOLUTION

      #1;

      // Point 2.           2          11          21  <- SOLUTION
   end

   my_prog my_prog_instance(a,b,c); // Ignore for part (a).

endmodule
```

☑ At Point 1, values for ☑ a, ☑ b, and ☑ c.

☑ At Point 2, values for ☑ a, ☑ b, and ☑ c.

(*b*) The definition of the `my_prog` program from the previous part appears below. Show the contents of the Verilog event queue at Point 1 in the code from the previous part, include the effect of code in `short_answers` as well as `my_prog`. Show events in the form "$t = 1969$, region=NL-East, Resume Point 3" and "$t = 2015$, region=X, Update variable z," but use real region names.

```
program my_prog(input int a, b, c);
   initial forever @( a or b or c ) begin
      // Point 3;
      $display("Let's go Mets!");
   end
endprogram
```

☑ Contents of event queue at Point 1, show ☑ region names and ☑ time stamps.

The solution appears below. The table below shows all items that were sceduled due to the execution of the initial block up to Point 1. The first non-blocking assignment to `a` and the non-blocking assignment to `c` schedules an update event in the NBA region at $t = 0$. The second non-blocking assignment to `a` schedules an update event at $t = 3$. Changes in `a`, `b`, and `c` cause a resume event to be scheduled in the re-inactive region for Point 3 of the `program` object. Finally, the 1-cycle delay at Point 1 schedules a resume event for Point 2 at $t = 1$.

Time	Region	Event
0	NBA	Update a <- 2
0	NBA	Update c <- 21
0	Re-inactive	Resume Point 3.
1	Inactive	Resume Point 2
3	NBA	Update a <- 3

(*c*) The module below is in explicit structural form, in which only primitive gates (and module instantiations) are used. Will the synthesis program synthesize exactly that arrangement of gates? Explain.

```
module bfa_structural( output uwire sum, cout, input uwire a, b, cin );
   uwire term001, term010, term100, term111;
   uwire ab, bc, ac;
   uwire na, nb, nc;
   not n1( na, a);
   not n2( nb, b);
   not n3( nc, cin);
   and a1( term001, na, nb, cin);
   and a2( term010, na, b,  nc);
   and a3( term100, a,  nb, nc);
   and a4( term111, a,  b,  cin);
   or o1( sum, term001, term010, term100, term111);
   and a10( ab, a, b);
   and a11( bc, b, cin);
   and a12( ac, a, cin);
   or o2( cout, ab, bc, ac);
endmodule
```

☑ Will synthesis program emit exactly these gates? ☑ Explain.

No. Or at best, not necessarily. The synthesis program will map the gates above to the most appropriate gates in the target technology, it will then perform optimization. It's possible, for example, that the target technology does not have a three-input AND gate, so either two 2-input gates will be used, or maybe a 4-input AND gate will be used with one input tied to logic 1. Or perhaps, the technology has a special binary full adder primitive.

(*d*) Based on a hand analysis of `my_mut` we expect it to have a clock period of 12 ns. Shown below is an excerpt from the testbench for `my_mut` that includes the code for generating a clock. Assume that the Verilog time unit is set to 1 ns. How does the clock declaration below affect the timing of the synthesized hardware?

```
module testbench();
   logic clock;
   initial clock = 0;
   always #5 clock = !clock;
   // Other declarations omitted.
   my_mut woof(x,y,a,b,clock);
```

☑ The effect of the declaration of `clock` on timing of synthesized hardware is . . . none ☑ because . . ..

The synthesis program will be commanded to synthesize `my_mut`, and so it won't see `testbench`, and therefore the clock period from `testbench` is irrelevant. Synthesis programs can be told to synthesize for a target clock period, but that target clock period is provided by a synthesis program command, such as `define_clock` for Cadence Encounter.

Name Solution

Digital Design using HDLs
LSU EE 4755
Final Examination

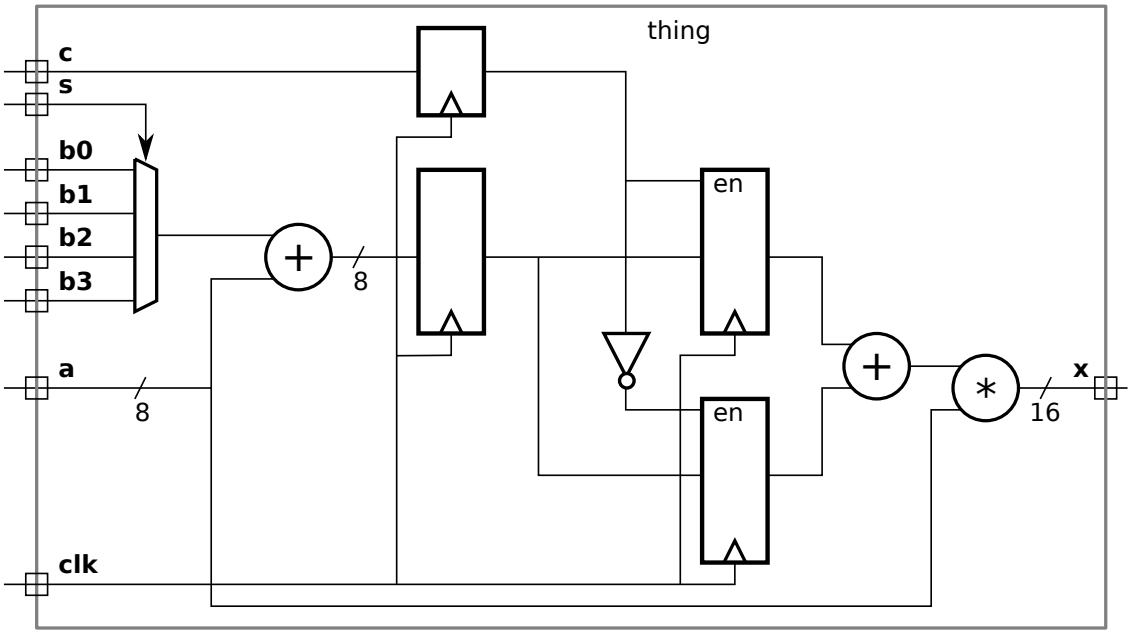
Saturday, 12 December 2015 12:30-14:30 CST

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

Alias Not Synthesizable

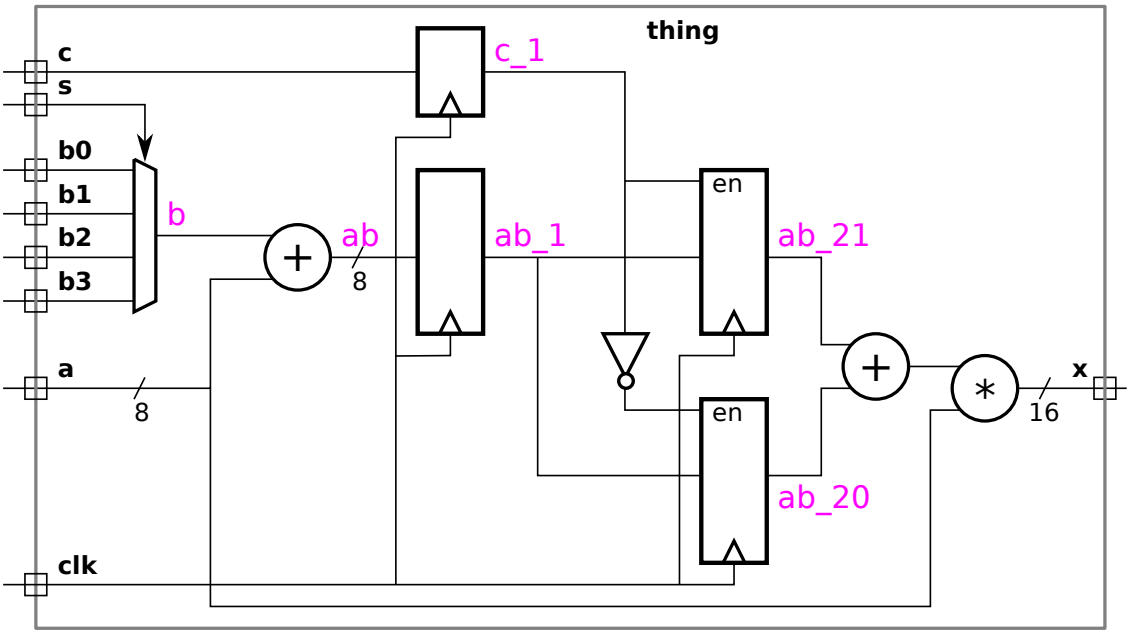
Good Luck!

Problem 1: [15 pts] Write a Verilog description of the hardware illustrated below.



SOLUTION ON NEXT PAGE

☑ Verilog description of hardware including ☑ port declarations and ☑ port and other sizes.



The solution appears below. Names for wires that were unlabeled in the problem appear in purple. (That is, the purple labels are part of the solution.) Note the use of `case`/`endcase` for the mux. Though using an `if`/`else` chain or the conditional operator, `?:`, would be correct, they are more tedious and prone to error and so it's worth taking the trouble to remember to use `case`.

```
module thing( output uwire [15:0] x,  input uwire c,       input uwire [1:0] s,
              input uwire [7:0] b0, b1, b2, b3, a,         input wuire clk );

   logic [7:0] b, ab, ab_1, ab_20, ab_21;
   logic       c_1;

   always_comb begin
      case ( s )
        0: b = b0;
        1: b = b1;
        2: b = b2;
        3: b = b3;
      endcase
      ab = a + b;
   end

   always_ff @( posedge clk ) begin
      c_1 <= c;     // Note: Delayed assignment, so if(c_1) uses prior value.
      ab_1 <= ab;   // Delayed assignment here too.
      if ( c_1 ) ab_21 <= ab_1; else ab_20 <= ab_1;
   end

   assign       x = a * ( ab_20 + ab_21 );
endmodule
```

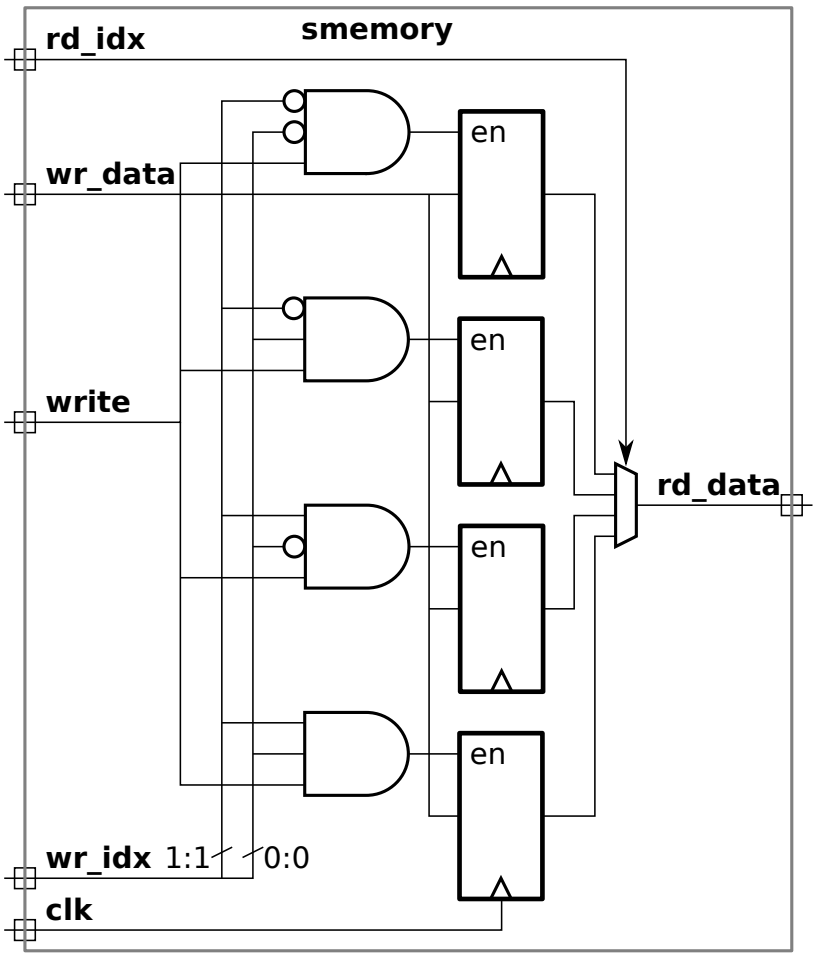
Problem 2: [20 pts] The module below implements a simple memory module.

```
module smemory #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
   ( output uwire [dbits-1:0] rd_data,
     input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
     input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];

   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   assign rd_data = storage[rd_idx];

endmodule
```

(*a*) Show the hardware that will be synthesized for this module when elaborated with `size_lg = 2`. Use registers, multiplexors, decoders, and basic gates. **Do not** use a memory module.

☑ Show synthesized hardware, including hardware for ☑ reading and ☑ writing.

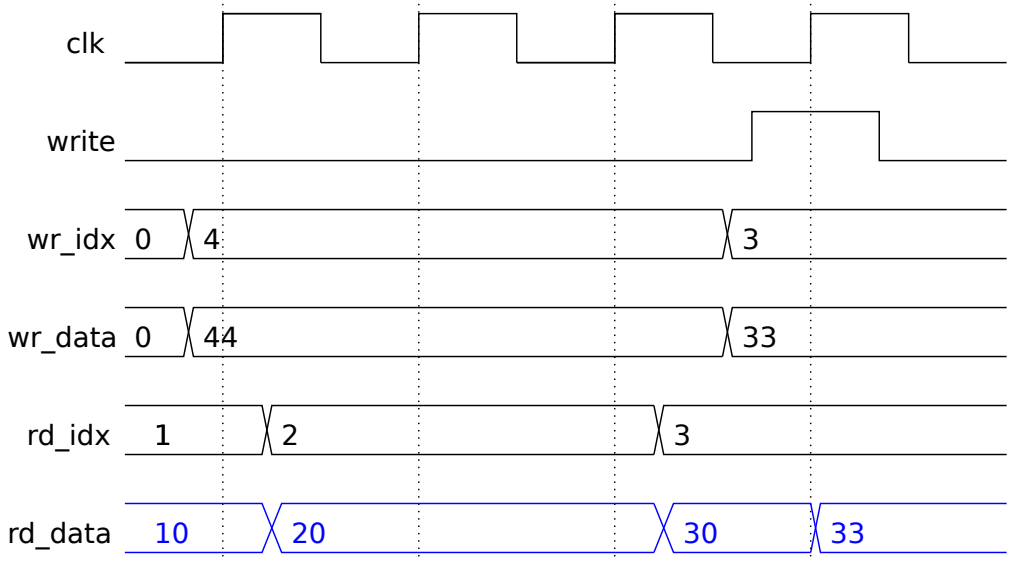
Solution appears below.



Problem 2, continued: Appearing below is the module from the previous page.

```
module smemory #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
   ( output uwire [dbits-1:0] rd_data,
     input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
     input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];

   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   assign rd_data = storage[rd_idx];

endmodule
```

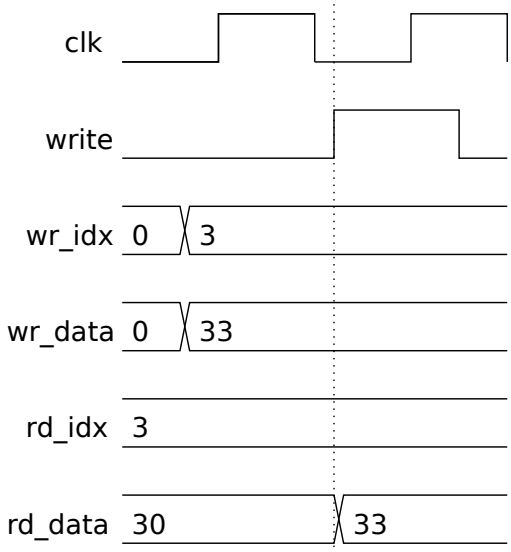
(*b*) Assume that initially location 1 (`storage[1]`) holds a 10, location 2 holds a 20, location 3 holds a 30, and so on. Complete the timing diagram below, consistent with this module.



☑ Complete `rd_data` row of timing diagram.

Solution appears above in blue.

(*c*) Modify the module below (same as one on previous page) so that its behavior is consistent with the timing diagram to the right. That is, if the location being written is the same as the one being read the `rd_data` output shows the data on `wr_data`. If the locations don't match or nothing is being written the behavior is unchanged.



☑ Modify the module.

Solution appears below. The original line is commented out for reference. Otherwise, cluttering your code with commented out lines is bad style. Instead, learn how to diff your working copy with the latest committed version and be able to do so in < 500 ms.

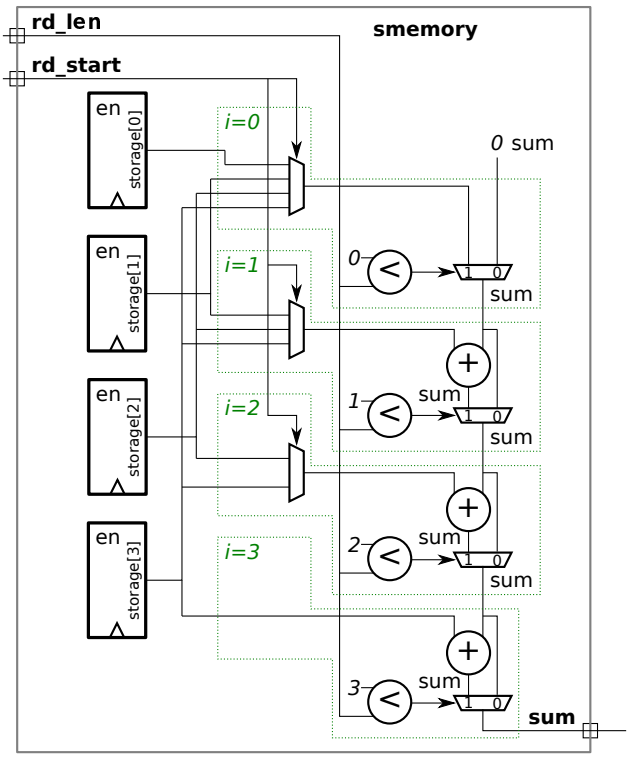
```
module smemory_bp #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
 ( output uwire [dbits-1:0] rd_data,
   input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
   input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];



   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;



   // assign rd_data = storage[rd_idx];
   // SOLUTION
   assign rd_data = write && rd_idx == wr_idx  ?  wr_data  :  storage[rd_idx];



endmodule
```

Problem 3: [20 pts] The module below and the similar one on the next page are like the memory module from the previous problem, except that their output is the sum of locations `rd_start`, `rd_start+1`, ..., `rd_start+rd_len-1`. Assume that `rd_start+rd_len <= size`.

```
module rsum_plan_a #( int sz_lg = 4, int ebits = 8, int size = 1 << sz_lg )
   ( output logic [ebits-1:0] sum,
     input [sz_lg-1:0] wr_idx,      input [ebits-1:0] wr_data,   input write,
     input [sz_lg-1:0] rd_start,    input [sz_lg-1:0] rd_len,    input clk    );

   logic [ebits-1:0] storage [size-1:0];

   // Don't show synthesized hardware for line below.
   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   // Plan A -- Show Synthesized Hardware for this Verilog
   always_comb begin
        sum = 0;
        for ( int i=0; i<size; i++ ) if ( i < rd_len ) sum += storage[ i + rd_start ];
     end

endmodule
```

(*a*) Show the hardware that will be synthesized for the `always_comb` block. Include basic optimizations, but don't optimize to the point where hardware is identical to Plan B (next page).

☑ Show not-too-optimized hardware for `sum`.

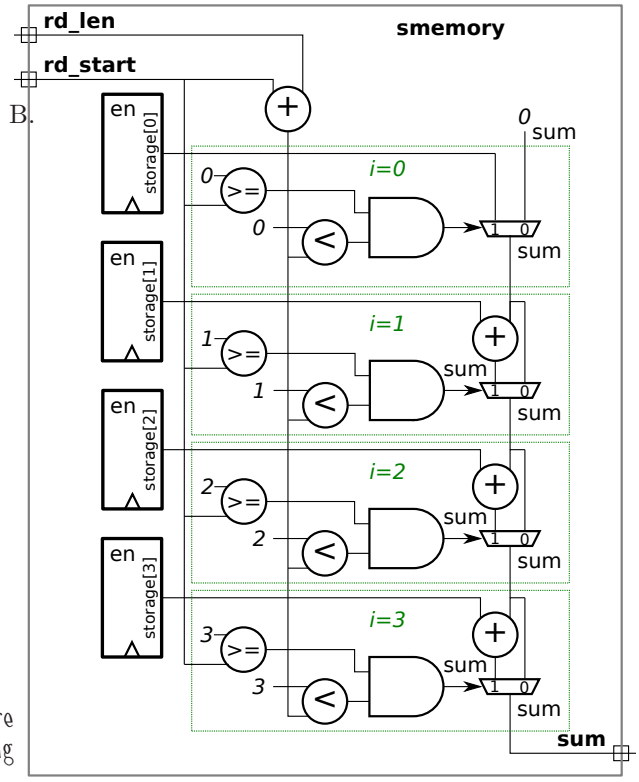


(*b*) Appearing below is Plan B for the module. Though we know it produces the same value for `sum` as Plan A, it might be synthesized into different hardware. Show the hardware synthesized for Plan B.

```
module rsum_plan_b #( int sz_lg = 4, int ebits = 8, int size = 1 << sz_lg )
   ( output logic [ebits-1:0] sum,
     input [sz_lg-1:0] wr_idx,      input [ebits-1:0] wr_data,   input write,
     input [sz_lg-1:0] rd_start,    input [sz_lg-1:0] rd_len,    input clk   );
   logic [ebits-1:0] storage [size-1:0];

   // Don't show synthesized hardware for line below.
   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   // Plan B -- Show Synthesized Hardware for this Verilog
   always_comb begin
        sum = 0;
        for ( int i=0; i<size; i++ )
          if ( i >= rd_start && i < rd_start + rd_len ) sum += storage[ i ];
     end
endmodule
```

☑ Show the hardware that will be synthesized for Plan B.

Solution appears to the right.



(*c*) Which one is better?

☑ Which is better, ◯ *Plan A* or ☑ *Plan B* .

☑ Explain, with a rough estimate of cost and timing.

Short Answer: The cost of the multiplexors makes Plan A more expensive than Plan B when `ebits` is greater than 1. The timing is about the same.

Detailed answer: Plan A contains three more multiplexors than Plan B, the total number of additional multiplexor inputs is $3+2+1 = 6$, and each of these is `ebits` wide, for a cost of $6 \times 3 \times e = 18e$ units, where e is `ebits`. The logic in Plan B that's not in Plan A includes four AND gates, a 2-bit adder and three fixed comparison units. Assume that the cost of a BFA is 10 units. Since the inputs to the adder are 2-bit quantities and since a carry-out is needed, the cost is 20 units. (The adder output must be three bits to do the comparison `i<rd_start+rd_len`.) Assume that the $\geq$ fixed comparison units cost 3 units each (draw a truth table). The total cost of logic in Plan B not in plan A is then $4 + 20 + 3 \times 3 = 33$ units. So Plan B is less expensive whenever the storage element size, `ebits`, is greater than 1 bit, which presumably is most of the time.

The path to the select signal for the $i = 0$ mux in Plan B passes through an adder (albeit a small one), a comparison, and an AND gate. In contrast, signal arrive at the data inputs to the corresponding multiplexor at a delay of about 4 units. Therefore Plan B is a little bit slower based on this simple analysis.

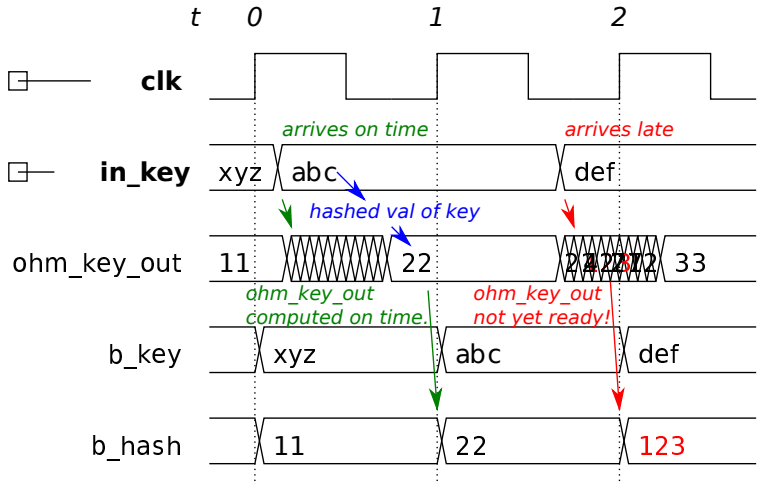
Problem 4: [15 pts] Appearing below are excerpts based on the `cam_hash` module used in class, showing what we called the `hash_early` design. Recall that with the early hash design the hash function (in module `hash`) is computed before the positive clock edge while the lookup occurs after the positive edge. We assumed that the hash could be computed in about $\frac{1}{2}$ of our target clock period.

```
module cam_hash_exceprt
   ( output [dwid:1] out_data, output out_valid,        output ready,
     input [kwid:1] in_key,    input [dwid:1] in_data,
     input Cam_Command in_cmd, input clk);

   logic [kwid:1] b_key;
   logic [dwid:1] b_data;
   logic [hkey_size-1:0] b_hash;
   Cam_Command b_cmd;

   uwire [hkey_size-1:0] ohm_key_out;

   always_ff @( posedge clk ) begin
      b_key <= in_key;
      b_data <= in_data;
      b_cmd <= in_cmd;
      b_hash <= ohm_key_out;
   end

   hash #(kwid,num_sets_lg) our_hash_module( ohm_key_out, in_key );

   /// Hardware to find matching key below ...
```

(*a*) The early hash design requires that the external hardware has the right timing behavior. Show a timing diagram in which the timing behavior is correct for early hash, and one in which it is wrong. The "wrong" behavior should result in incorrect results using the early hash design, but correct results without the early hash design.

☑ Timing diagram showing ☑ correct and ☑ wrong behavior.

Solution appears to the right. In the early hash design the value on port `in_key` must arrive in the first half of the clock cycle (before the negative edge). That is what happens for input `abc` and so `hash` which computes `ohm_key_out` has enough time to finish. The correct hash, 22 is clocked into register `b_hash`. In contrast, key `def` arrives late, and so when the next positive clock edge arrive `ohm_key_out` has not stabilized and so some arbitrary value is clocked into `b_hash`. Notice that `b_key` gets the correct value in both cases, because register gets its input directly from input port `in_key`.



Problem 4, continued:

(*b*) Register `b_hash` saves the hashed version of `in_key`, and `b_key` holds the unhashed version. Why do we need the unhashed version?

☑ `b_key` is needed because ...

The number of bits in the hash of a key is less than the key itself, therefore two keys can have the same hash. The unhashed version of the key is needed to check whether the key matches the key for item at the hashed location.

Problem 5: [10 pts] The Verilog below is part of a testbench (taken from `icomp.v`).

```
initial begin

   /// Watchdog – Stop simulation if it's taking too long.
   //
   fork begin

      automatic int cyc_limit = in_str.len() * 100;

      fork
         wait ( cycle_num == cyc_limit );
         wait ( tb_insert_done && tb_remove_done );
      join_any

      if ( cycle_num >= cyc_limit ) begin
         $write("Exceeded cycle limit, exiting.\n");
         $fatal(1);
      end

   end join_none

   // Below: Send data to module under test.
```

(*a*) Generically explain what a `fork` and `join` pair do (ignoring the code above).

☑ `fork` and `join` ...

Each statement executes with its own thread of control, meaning that delays and other timing controls in one does not affect the progress of the other. The statement after the `join` does not execute until all threads inside the `fork`/`join` finish.

(*b*) How would execution be effected if the last `join_none` were changed to `join_any`?

☑ Impact of changing `join_none` to `join_any` in code above.

Execution would never reach the `//Below` statement. With the `join_none`, execution proceeds to the `//Below` statement without delay. Code after the `//Below` statement tests modules and will set `tb_insert_done` and `tb_remove_done` when tests are finished. But with `join_none` changed to `join_any` the `//Below` statement will not be executed until the first `fork` finishes. That first `fork` finishes when either the cycle limit is exceeded or all modules have been tested, whichever comes first. But with `join_none` changed to `join_any` module tests won't have started and so the cycle limit will be exceeded. Note that if the cycle limit is exceeded the code exits with a fatal error, and so the `//Below` statement will never be reached.

(*c*) How would execution be effected if the inner `join_any` were changed to a `join_all`?

☑ Impact of changing `join_any` to `join_all` in code above.

The testbench will always report that the cycle limit was exceeded, even if all tests were completed.

Problem 6: [20 pts] Answer each question below.

(*a*) Suppose we would like our hardware to operate at a 1 GHz clock frequency. How do we tell the synthesis program? (The exact syntax is not important.)

☑ Method to tell synthesis program the clock frequency.

Short Answer: `define_clk -name ee4755 -period 1000 myclkport`.

Details: In Cadence Encounter use the command `define_clk -name NAME -period PERIOD PORTS`. To set the clock frequency to 1 GHz set the period argument to 1000, which is the clock period in picoseconds: $10^{12}\frac{1}{10^9} = 1000$. Argument `PORTS` is set to the name of the clock ports and `NAME` is a name by which this clock can be referred to in subsequent commands.

(*b*) The synthesis program will apply our target clock frequency to paths starting at launch points and ending at capture points. We could explicitly specify such points but if we don't it will use default launch and capture points. What are they?

☑ By default timing is computed for paths that start at: register outputs.

☑ and end at: register inputs.

Notice that the default launch and capture points **do not** include module inputs and outputs. Those have to be added with `external_delay` commands.

(*c*) Suppose our target clock frequency is 1 GHz. What is the harm in telling the synthesis program to synthesize for 2 GHz? For 0.5 GHz?.

☑ Harm in specifying 2 GHz when we just need 1 GHz:

The resulting design will work correctly, but may be more expensive than had we specified 1 GHz.

☑ Harm in specifying 0.5 GHz when we just need 1 GHz:

The synthesized hardware may not work at 1 GHz.

(*d*) The code below will inconsistently assign a variable. Explain why and fix the problem.

```
module short_ans( output logic [7:0] x, y, input [7:0] a, b, c, input clk);

   always @( posedge clk ) begin

      x = a + b;

   end

   always @( posedge clk ) begin

      y = x + c;

   end

endmodule
```

☑ Reason for inconsistent behavior:

Because the value of `x` used in the second `always` block may be before the `a+b` assignment, or after.

☑ Fix problem.

One way is to put the two statements in the same block. That's shown below. Another possibility is to use nonblocking assignment.

```
module short_ans( output logic [7:0] x, y, input [7:0] a, b, c, input clk);

   always @( posedge clk ) begin
      x = a + b;
      y = x + c;
   end

endmodule
```

(*e*) Describe the problem with the module below. How might it affect simulation?

```
module short_ans2( output logic [7:0] x, input [7:0] a, b, input reset);

   always_comb begin

      if ( reset ) x = a; else x = x + b;

   end

endmodule
```

☑ Problem with module.

☑ Impact on simulation.

Wire `x` is both an input and an output of the `always_comb`. So each change in `x` would trigger another execution of the block. To fix it a clock is needed to control when `x` is incremented.

24 Fall 2014 Solutions

Name Solution

Digital Design using HDLs
EE 4755
Midterm Examination

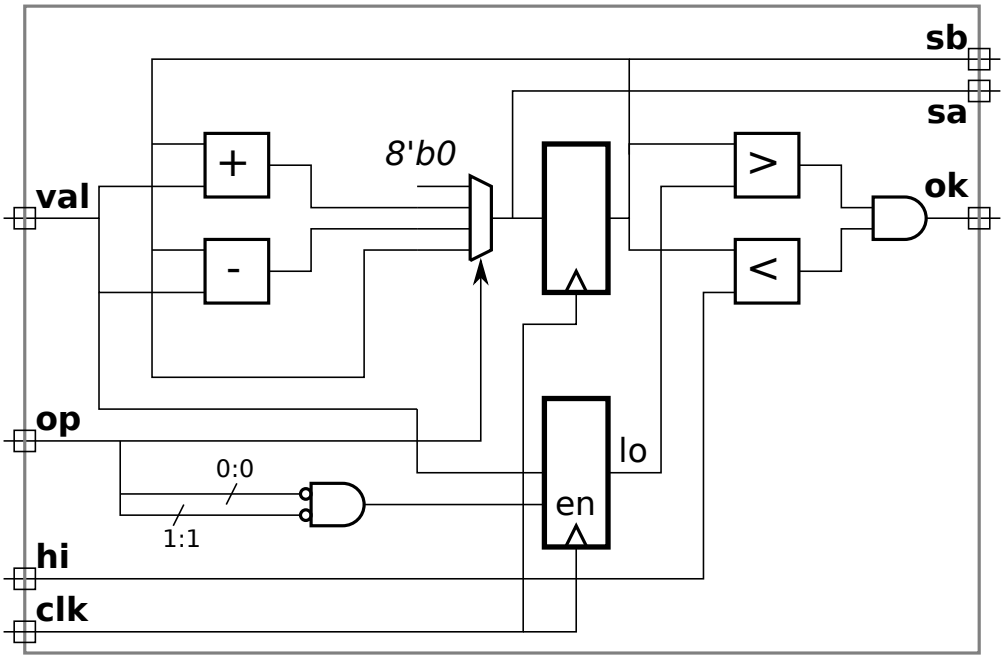
Monday, 10 November 2014 11:30–12:20 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(10 pts)
Problem 4		(15 pts)
Problem 5		(13 pts)
Problem 6		(22 pts)
Exam Total		(100 pts)

Alias Over-reactive region

Good Luck!

Problem 1: [20 pts] Write a Verilog description of the hardware shown below.



☑ Write a Verilog module corresponding to the hardware above.

☑ Be sure to declare module ports and ☑ any wires and vars (`logic`) used inside.

☑ Pay attention to the differences between `lo` and `hi` and ☑ the differences between `sa` and `sb`.

```
// SOLUTION
module prob1(output logic [7:0] sa, sb,   output uwire ok,
             input uwire [1:0] op,        input uwire [7:0] val, hi,   input uwire clk );

   always_comb
     case ( op )
       0: sa = 0;
       1: sa = sb + val;
       2: sa = sb - val;
       3: sa = sb;
     endcase

   always_ff @( posedge clk ) sb <= sa;

   logic [7:0] lo;
   always_ff @( posedge clk ) if ( op == 0 ) lo <= val;

   assign ok = sb > lo && sb < hi;

endmodule
```

The Verilog appears above.

Discussion of `sa` /`sb` differences.

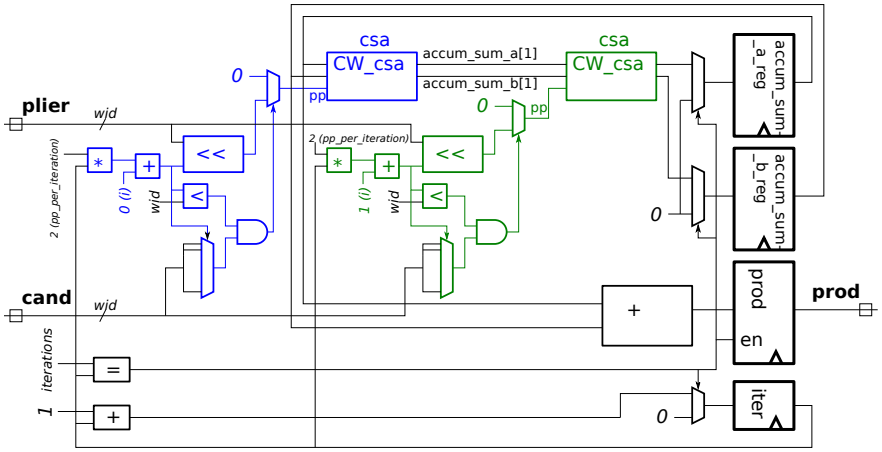
In the diagram notice that `sa` is produced in part by signals connected to the module inputs. That means if, for example, input `op` changes then `sa` must change as soon as it can. For that reason it is **not** assigned in an always block controlled by `posedge clk`, instead it is assigned in an `always` block sensitive to all live-in objects, namely `op`, `val`, and `sb`. In contrast, output `sb` is connected to the output of an edge-triggered register which means it can *only* change on the positive edge of `clk`. For that reason it is assigned in an `always` block sensitive to `posedge clk`.

The `lo` register is written on the positive edge of the clock when bit 0 (notice the `0:0` label next to the tic mark) of `op` is zero and when bit 1 of `op` is zero. In Verilog that's cleanly shown as `if ( op == 0 )`. It would be correct though cumbersome to replace the `if` condition with `op[0] == 0 && op[1] == 1`. An even more cumbersome solution would instantiate an AND gate and two NOT gates.

Problem 2: [20 pts] Appearing below is the multiply circuit from the solution to Homework 3, in Verilog (slightly simplified) and as a diagram showing what hardware a synthesis program might infer.

```
module mult_seq_csa_m #( int wid = 16, int pp_per_cycle = 2 )
   ( output logic [2*wid-1:0] prod,
     input logic [wid-1:0] plier, input logic [wid-1:0] cand, input uwire clk);

   localparam int iterations = ( wid + pp_per_cycle - 1 ) / pp_per_cycle;
   localparam int iter_lg = $clog2(iterations);
   localparam int wid_lg = $clog2(wid);

   logic [iter_lg:0] iter;
   uwire [2*wid-1:0] accum_sum_a[0:pp_per_cycle], accum_sum_b[0:pp_per_cycle];
   logic [2*wid-1:0] accum_sum_a_reg, accum_sum_b_reg;

   assign           accum_sum_a[0] = accum_sum_a_reg;
   assign           accum_sum_b[0] = accum_sum_b_reg;

   for ( genvar i=0; i<pp_per_cycle; i++ ) begin

      uwire [wid_lg:1] pos = iter * pp_per_cycle + i;
      uwire [2*wid-1:0] pp = pos < wid && cand[pos] ? plier << pos : 0;

      CW_csa #(2*wid) csa
        ( .sum(accum_sum_a[i+1]), .carry(accum_sum_b[i+1]), .a(accum_sum_a[i]), .b(accum_sum_b[i]), .c(pp));
   end

   always @( posedge clk )
      if ( iter == iterations ) begin
         prod <= accum_sum_a_reg + accum_sum_b_reg;
         accum_sum_a_reg <= 0;
         accum_sum_b_reg <= 0;
         iter <= 0;
      end else begin
         prod <= prod;
         accum_sum_a_reg <= accum_sum_a[pp_per_cycle];
         accum_sum_b_reg <= accum_sum_b[pp_per_cycle];
         iter <= iter + 1;
      end

endmodule
```

USE NEXT PAGE FOR SOLUTION



USE NEXT PAGE FOR SOLUTION

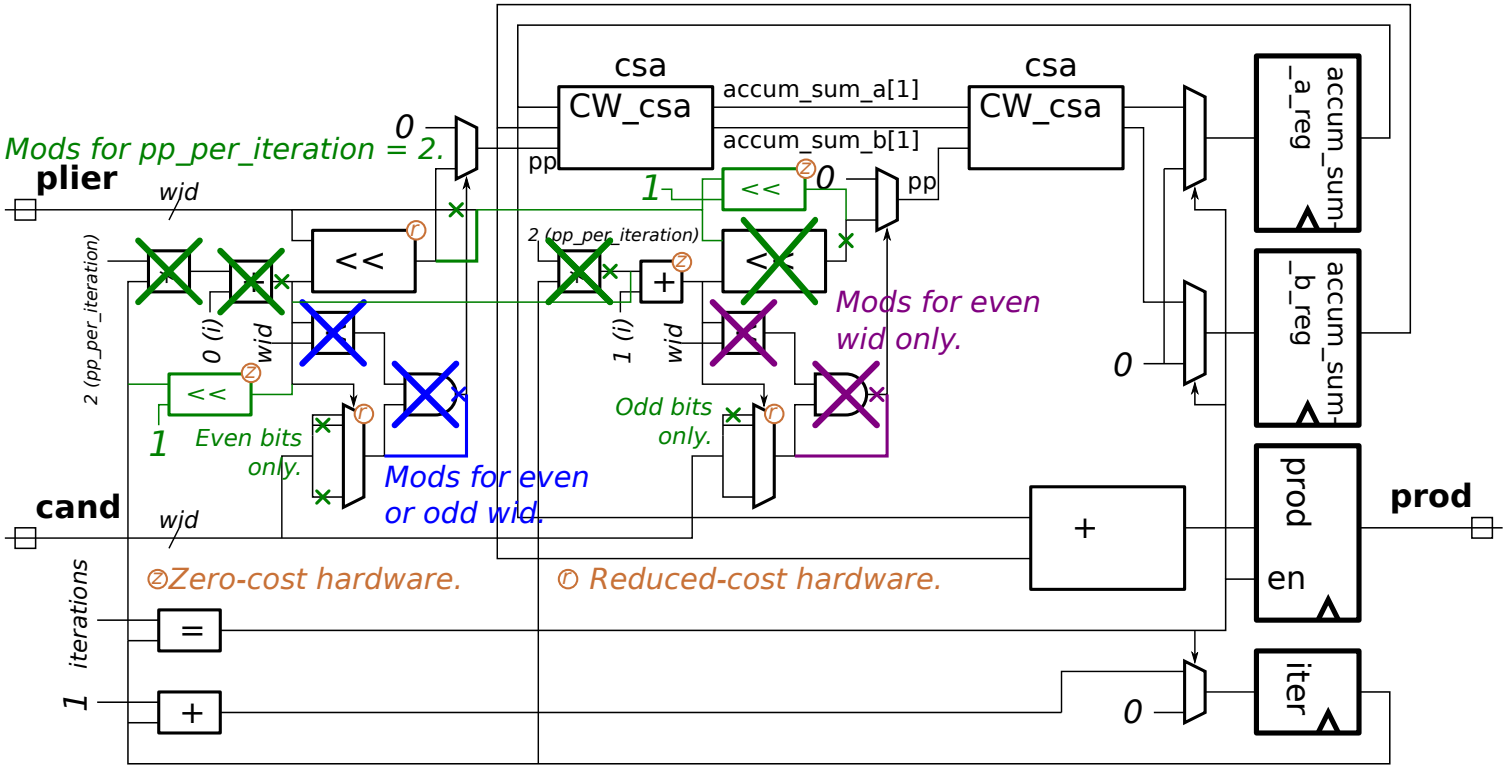
(a) Show optimizations that might be performed that exploit the value $m = 2$ (that is, `pp_per_iteration=2`).

(b) Show the optimizations that might be performed assuming that `wid` is odd, and assuming that `wid` is even, both for $m = 2$.

☑ Modify diagram to show optimizations for `pp_per_iteration` = $m = 2$ and arbitrary `wid`.

☑ Modify diagram to show optimizations for `pp_per_iteration` = $m = 2$ and odd `wid`.

☑ Modify diagram to show optimizations for `pp_per_iteration` = $m = 2$ and even `wid`.



Solution appears above. Three sets of changes are shown. The changes in green are optimizations possible with `pp_per_iteration=2`, the changes in blue are possible with any value of `wid` (odd or even), and the changes in purple are possible only when `wid` is even. Hardware labeled with a circled z is zero-cost, meaning that the outputs are either constants or are connected directly to the inputs. (In class this was called *renaming bits*.) The hardware labeled with a circled r is a lower cost version of the hardware depicted. In particular, the shift so labeled is lower cost because it only needs to shift by an even number of positions. The r-labeled multiplexors are lower cost because half of their inputs are unused (and so will not be synthesized).

If we know that `pp_per_iteration` is 2 then the shift amounts for `plier` just shifting `iter` by one bit (for `i=0`) or shifting and placing a `1` in the LSB position (for `i=1`). These observations are used to eliminate the multipliers and adders. (One of the adders is shown as zero-cost.) Further, in the `i=0` section we know that only even-numbered bit positions are from `cand`, reducing the cost of the multiplexor; a similar optimization is made for the `i=1` section.

The $<$ modules are used to determine if the `cand` bit position is valid. For the `i=1` section the `cand` bit position in the last iteration will be invalid if `wid` is odd. (For example, suppose `wid=5` and consider the third iteration, when `iter` is 2. The bit position sought by the `i=0` section will be $2 \times 2 = 4$, the MSB of `cand`. The `i=1` section will look for bit $2 \times 2 + 1 = 5$ which is invalid, though with a typical adder the mulitplexor might be commanded to look at bit 0, which is wrong. The less-than module and AND gate prevent the bit from being used.)

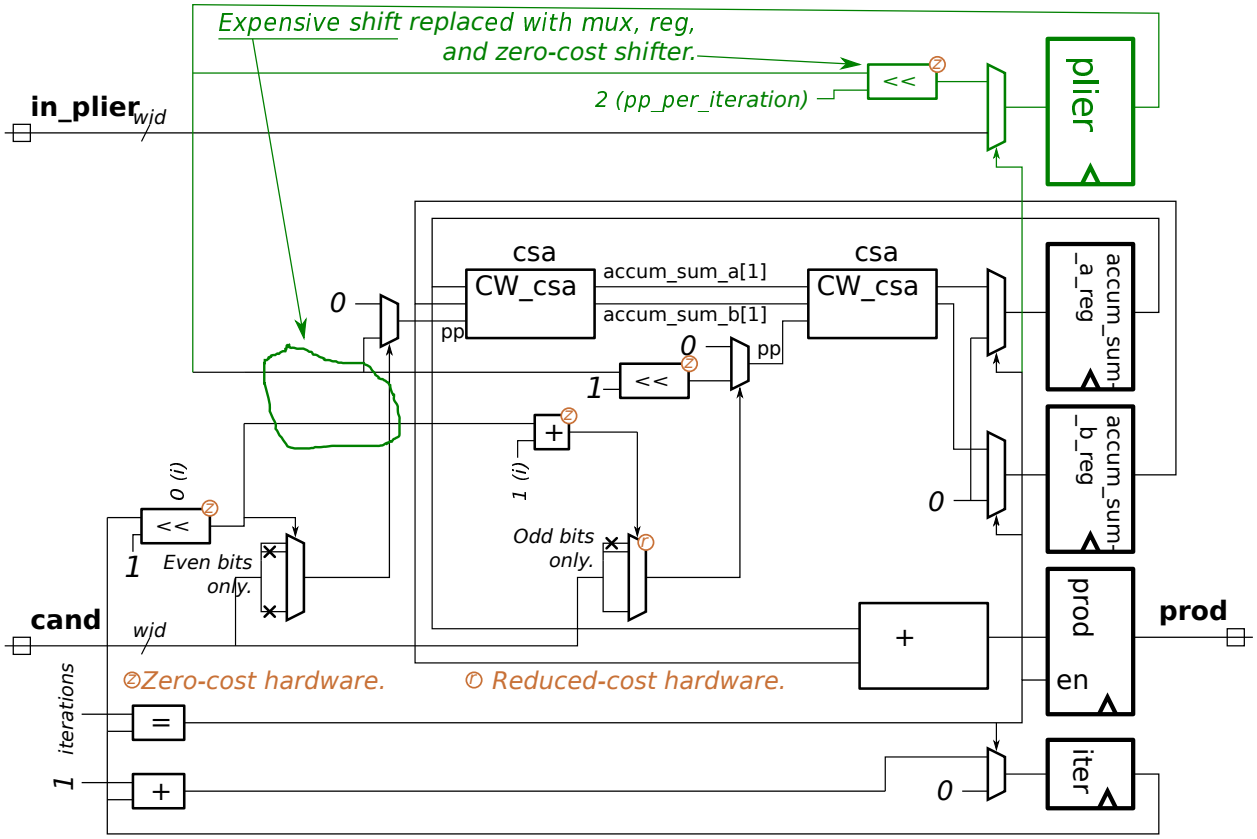
In contrast, there will never be an invalid bit position when `wid` is even. So, when `wid` is even both the optimizations shown in blue and purple can be made. If `wid` is odd then the blue optimizations can be made but the purple optimizations **cannot** be made.

Problem 2, continued:

(*c*) The cost of the shifters with input `plier` in the design on the previous pages is significant. Explain how these shifters can be eliminated by adding a register. Quickly sketch the hardware to illustrate your answer.

☑ Show how a register can be used to eliminate the costly shifters.

Solution appears below in green, based on the optimized even-`wid` multiplier. The shift-by-any-amount (or at least any even amount) shifter (which would occupy the hand-drawn circle in the diagram) is replaced by a shifter that shifts by exactly `pp_per_iteration` positions, which is a zero-cost device. The output of this shifter is stored in a register and used in the next iteration. The shift amounts that are needed in a particular iteration can be obtained using only these zero-cost shifters. The multiplexor and register that we've added is not free, but their should be less than the shifters when about three or more iterations are needed.



(*d*) Explain how the streamlined multiplier described in class eliminated the `plier` shifter *without* having to add a register.

☑ Show how the streamlined multiplier does not need an extra register to eliminate the shifter.

The streamlined multiplier shifts the accumulated product rather than the multiplier. (This may not be possible using CSAs.)

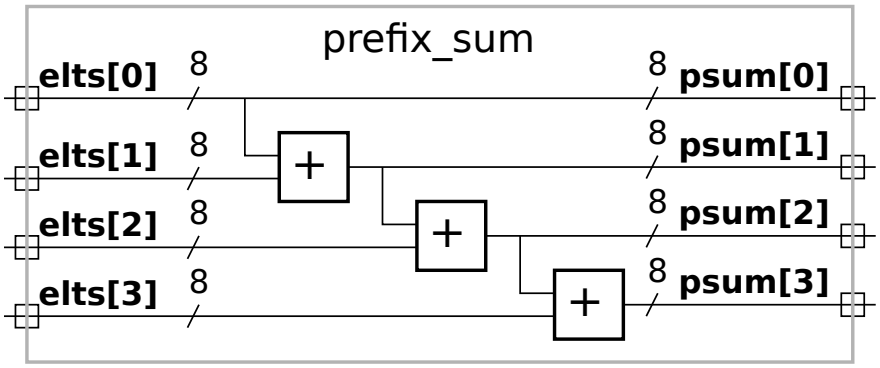
Problem 3: [10 pts] The module below computes the prefix sum of a sequence of integers at its input.

```
module prefix_sum #( int len=8, int wid = 8)
  (output logic [wid:1] psum [len], input uwire [wid:1] elts[len]);

   always @* begin
      psum[0] = elts[0];
      for ( int i=1; i<len; i++ ) psum[i] = psum[i-1] + elts[i];
   end
endmodule
```

(*a*) Show the hardware that would be synthesized for the module before optimization, elaborated with parameters `len=4` and `wid=8`. Label the input ports `elts[0]`, `elts[1]`, `elts[2]`, and `elts[3]`; and label the output ports `psum[0]`, `psum[1]`, `psum[2]`, and `psum[3]`.

☑ Show synthesized hardware.

Synthesized hardware appears below.



(*b*) Estimate the delay for the synthesized hardware before optimization. Use w for the value of `wid` and L for `len`. Assume that a w-bit adder has delay w.

☑ Delay in terms of w and L:

The delay is $(L-1)w$.

Problem 4: [15 pts] Answer the following questions about the Verilog module below.

```
module timing();
   logic [7:0] a, e, f2, g, g1, g2;   logic clk;      uwire [7:0] e1, f, f1;
   initial begin
      clk = 0;
      a = 11;
      #1;
      a = 1;
      a <= 22;
      a <= #5 a + 1;

      #9;
      a = 7;
      e = 10;
      f2 = 30;
      g = 40;
      g1 = 50;
      g2 = 60;

      #10;
                                          // B0
      a <= 700;
      clk = 1;

      #1;
      // POINT X (See subproblem.)
   end

   always @( posedge clk ) e = a;           // B1
   always @*               e1 = a;          // B2
   always @*               f = e + 1;       // B3
   always @*               f1 = e1 + 1;     // B4
   always @( posedge clk ) f2 <= e + 1;     // B5
   always @( posedge clk ) begin            // B6
                            g = f;   g1 = f1;   g2 = f2;   end
endmodule
```

(*a*) Show values for `a` versus time in the table below. For this part, **only** `a`. The table already shows that `a` has value 11 from time 0 to time 1. Extend the table as long as necessary, and be sure to show values for both t and `a`. *Note: The original exam did not provide the table. Also, in the original exam there were differences in how* `a` *was assigned.*

☑ Complete the table.

t	0	1	6	10	20
a	11	22	2	7	700

Solution appears above. After $t = 1$ `a` gets the value 22 because of the non-blocking assignment. However, the delayed assignment (`a <= #5 a + 1;`) uses the value of `1` for `a` since the non-blocking assignment of `22` at that point had not taken effect. (It must wait until the scheduler gets to the NBA region of the event queue.)

Notice that delays (such as `#10` are relative to the current time, **not to** $t = 0$).

(*b*) Show the values that will be present on `g`, `g1`, `g2` when execution reaches the `POINT X` comment in the module above. For partial credit also show intermediate values for other signals used to compute the `g`'s. (Look at next part before solving this one.)

☑ At `POINT X` `g`=__11__, `g1`=__8__, `g2`=__8__.

Solution appears above. The `g`'s are assigned on the positive edge of the clock in block `B6`. To solve the problem one needs to figure out what has already executed. Before point `B0` in the code `a` is 7, `e` is 10, and `f2` is 30, set by the procedural code, and the combinational (`@*`) always blocks would have set `f1` to 8 and `f` to 11. After the procedural code assigns `clk=1` and reaches the `#1` the scheduler will schedule the `posedge clk` blocks in arbitrary order. After `B0` finishes the three newly scheduled blocks `B1`, `B5`, and `B6`, are placed in the active region of the event queue. Block `B1` changes `e` to 7, which causes `B3` to be scheduled in the inactive region. The scheduler continues with the active region, next executing `B5`, which schedules an update event in the NBA region that will set `f2` to 8. Next `B6` is executing, assigning the `g`'s. Variable `g` is assigned 11, notice that `B3` is still in the active region and so has not gotten its chance to modify `f`. Variable `g1` is assigned 8. Variable `g2` is assigned 30. Notice that `B5` adds 1 to `e` before `B6` executes but the update is done afterwards. Block `B3` executes after `B6`. Therefore the "old" values are used for `g` and `g2`.

(*c*) Recall that the event queue used for Verilog simulation has *active*, *inactive*, and *NBA* regions, among others. Just before `B1` starts execution in module `timing` above the active region might contain `B1`, `B5`, and `B6` (see the comments on the right). (What the other regions contain is part of this problem.) Show the contents of the three regions when `B5` starts. Assume that events in a region are scheduled in order.

☑ When `B5` starts: Active = {__B6__}. Inactive = {__B3__}. NBA = {__a <= 700__}.

Solution appears above. When `B5` starts only `B6` remains in the active region (see the solution to the previous problem). Block `B3` has been scheduled in the inactive region due to the assignment of `e` by `B1`. The update to `a` was scheduled in the NBA (non-blocking assignment) region by `B0` in the `initial` block.

Problem 5: Answer each question below.

(*a*) [5 pts] Module `add3` is supposed to compute the sum of its three inputs using instances of `our_adder`, but it won't work. Fix the problem. The fixed module should still use `our_adder`.

☑ Fix `add3`.

```
module add3(output uwire [15:0] sum, input uwire [15:0] a,b,c);



   our_adder a1(  sum  ,  a  ,  b  );

   our_adder a2(  sum  ,  sum  ,  c  );

endmodule
```

```
   /// SOLUTION

module add3(output uwire [15:0] sum, input uwire [15:0] a,b,c);
   uwire [15:0] sum1;

   our_adder a1( sum1, a,    b );
   our_adder a2( sum,  sum1, c );

endmodule
```

Solution appears above. The problem was that the same object, `sum`, was connected to the output of both adders. Its value therefore is undefined. In the solution a new wire, `sum1`, is declared and used as the output of the first adder.

(*b*) [8 pts] The output of the module below is like the input except the bit positions are reversed (after enough clock cycles). Re-write the module so that it synthesizes to combinational logic (the `clk` input will no longer be needed). Add a parameter to indicate the input and output bit width.

```
module bitrev(output logic [7:0] x, input uwire [7:0] a, input uwire clk);
   logic [2:0] pos;
   initial pos = 0;

   always @( posedge clk ) begin
      x[pos] = a[7-pos];
      pos++;
   end
endmodule
```

☑ Re-write so that it is combinational.

☑ Include a parameter `wid` to specify the size.

```
// SOLUTION
module bitrev_s #(int wid = 8) (output logic [wid-1:0] x, input [wid-1:0] a);

   always @* for ( int i=0; i<wid; i++ ) x[ i ] = a[ wid-i-1 ];

endmodule
```

Solution appears above. Since the logic is combinational there is no need for a clock input. Notice that this will synthesize into a module that contains no logic. All it does is rename signals. This would not be a problem if it were part of a larger design, but if this module were the only thing fabricated on a chip money could have been better spent.

Problem 6: Answer each question below.

(*a*) [5 pts] A Verilog module computes a result in one clock cycle. In our design we need that result in 3 ns, which can easily be achieved. The right way to achieve that in Cadence Encounter is to use the `define_clock` command to set the target clock period to 3 ns. Suppose instead we used `define_clock` to set the period to 1 ps, an impossible goal. *Note: The original exam did not have the "can easily be achieved" phrase.*

☑ Would the synthesized design meet our 3 ns performance goal?

Yes. Even though 1 ps is impossible, the synthesis program will synthesize a circuit with as short a delay as it's capable of, and according to the problem it can easily create a circuit with a delay less than 3 ns. *Note: For those taking the original exam the answer would be: Yes, if the synthesis program is capable of reaching the* 3 ns *goal.*

☑ Considering typical design goals, what would be the disadvantage of setting the period to 1 ps for our design even though we needed 3 ns?

Short answer: The disadvantage is that the cost of the synthesized circuit might be higher than would be obtained when setting the clock period to our performance target, 3 ns.

Suppose the synthesis program generates a circuit with a delay of 2.1 ns. That meets our performance goal, but so would a 3 ns circuit. However the 2.1 ns circuit might have a higher cost than the 3 ns circuit since the optimization program tries to minimize cost while meeting design constraints. Since cost minimization is a typical design goal, setting the clock period to 1 ps would result in a worse design.

(*b*) [10 pts] In the module below, `translate` directives are used to prevent the synthesis program from reading the line with `initial`.

```
module mult_seq( output logic [311:0] prod, input logic [15:0] plier, cand, input uwire clk);

   logic [3:0] pos;    logic [31:0] accum;

   // cadence translate_off          <-- The translate synthesizer directive.
   initial pos = 0;
   // cadence translate_on           <-- The translate synthesizer directive.

   always @( posedge clk ) begin
      if ( pos == 0 ) begin  prod = accum;  accum = 0;  end
      if ( cand[pos] == 1 ) accum += plier << pos;
      pos++;
   end
endmodule
```

☑ Why shouldn't the synthesis program see the line with `initial`?

☑ What would happen if the synthesis program saw the `initial` line?

Short answer: The synthesis program should not see the `initial` line because it has no way to synthesize corresponding hardware, if it saw the line it would generate an error message.

The synthesis program should not see the `initial` line because it is unsynthesizable, and so would result in an error message. It is unsynthesizable because the developers of the synthesis program (this semester Cadence Encounter RTL Compiler) and the developers of probably every other HDL synthesis program do not think it's worth the trouble to generate special "initial" hardware that only does something when, say, the power is turned on. The correct way of achieving that kind of behavior is by providing a reset input to the module.

☑ What would happen if the simulation program *didn't* see the line with `initial`?

The value of `pos` would remain at `x` (undefined).

(*c*) [7 pts] All four variables below have a size of 32 bits, but there are differences between them.

```
logic [31:0] a;
logic b [31:0];
logic [0:31] c;
int e;
```

All four variables above hold 32 bits. (Unlike C, SystemVerilog sets the size of `int` to be 32 bits.)

Variable `a` is called a packed vector. It is interpreted as a single 32-bit quantity, and so can conveniently be used in expressions such as `a+x`.

☑ Difference between `a` and `b`?

Variable `b` is interpreted as a 32-element array of 1-bit elements.

☑ Difference between `a` and `c`?

Both `a` and `c` are packed vectors and are interpreted as 32-bit quantities. However the bit numbering of the two are different. That makes a difference in expressions that refer to bit positions, such as `y = a[10];` , but it does not make a difference in expressions that don't refer to bit positions, such as `y = a + x;`.

☑ Difference between `a` and `e`?

Each bit in a `logic` object can have four states, `0`, `1`, `x`, and `z`. Type `int` is a 32-bit quantity in which each bit is either `0` or `1`. The `logic` type is intended for objects that will synthesize into hardware, while `int` is intended for other uses such as in testbenches.

Name Solution

Digital Design using HDLs
EE 4755
Final Examination

Monday, 8 December 2014 10:00-12:00 CST

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

Alias Not Synthesizable

Good Luck!

Problem 1: [20 pts] The `encode` module below, based on Homework 4, is used to convert a decimal value to binary one ASCII digit at a time. Input `val_prev` is the binary value so far, and output `val_next` is the binary value after using ASCII character `ascii_char`. If `ascii_char` isn't a numeric digit `non_digit` is set to 1 and `val_next` is set to zero. There is also an `overflow` output.

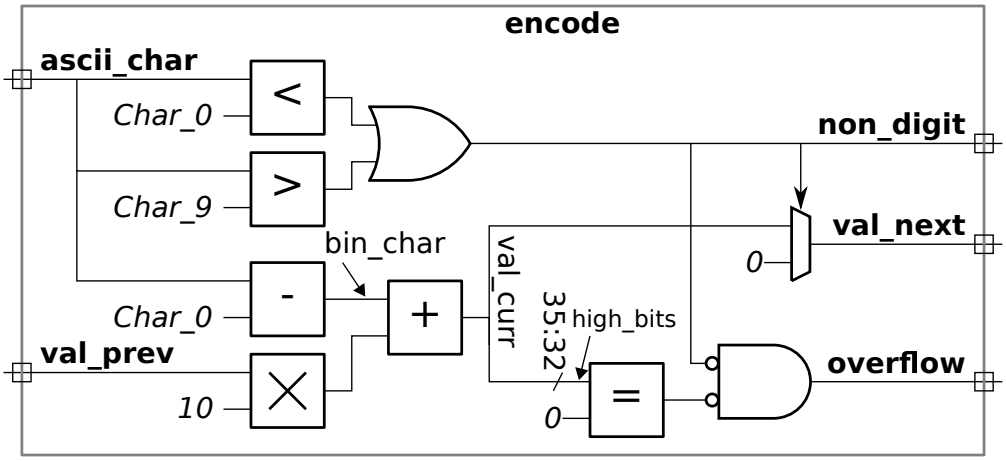
```
module encode
  #( int width = 32 )
   (output logic [width-1:0] val_next,
    output logic overflow,                output uwire non_digit,
    input uwire [7:0] ascii_char,         input uwire [width-1:0] val_prev);

   logic [width+3:0] val_curr;            logic [3:0] high_bits, bin_char;

   assign non_digit = ascii_char < Char_0 || ascii_char > Char_9;

   always_comb begin
      bin_char = ascii_char - Char_0;
      val_curr = 10 * val_prev + bin_char;
      high_bits = val_curr >> width;
      if ( non_digit ) begin    overflow = 0; val_next = 0;       end
      else             begin
        overflow = high_bits != 0;
        val_next = val_curr;
      end
   end
endmodule
```

(*a*) Show the hardware that will be synthesized for this module. Take into account optimizations (see the next subproblem).

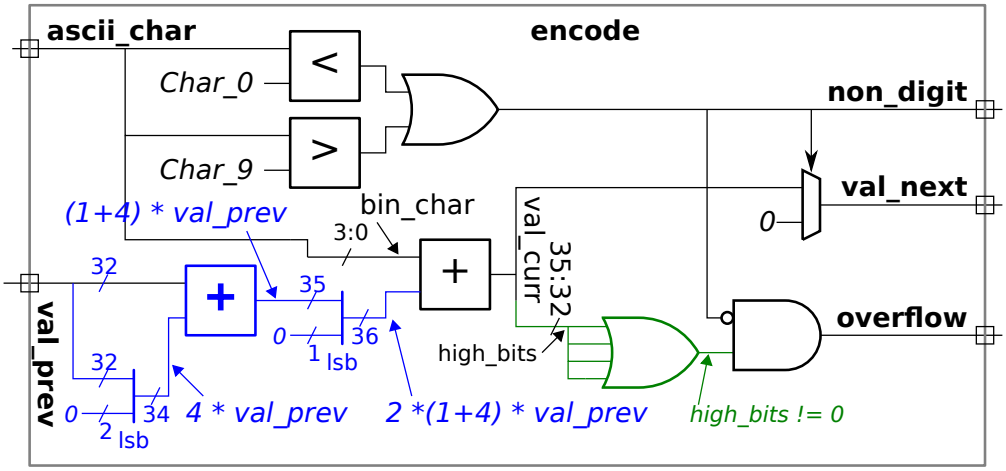
☑ Synthesized hardware.

Two versions of the solution appear below. In the first only basic optimizations are shown. The optimizations shown are for the overflow logic and for the computation of `high_bits`.

In the Verilog `high_bits`, which is four bits, is the result of an expression using a right shift operator. All this does is assign bits 35 to 32 of `val_curr` to `high_bits`, so in the diagram below that is all that is shown. The other basic optimization is logic for `overflow`. Since `overflow` is one bit it makes more sense to use simple gates rather than a multiplexor, and that is what is shown.



The solution below shows further optimizations: the subtractor to compute `bin_char` is eliminated, the times-ten multiplier has been replaced by an adder (that's shown in blue), and the `!=0` operation on `high_bits` is now shown as a four-input OR gate (in green). The replacement for the multiplier uses two constant shifters, they are shown by heavy vertical lines. (The heavy vertical lines indicate, in this case, the grouping together of bits. In this case putting one or two 0's in the LSB position to form a new quantity.)



(*b*) Indicate how many units such as adders, multipliers, shifters, and multiplexors will actually be present in the optimized hardware. The count should be based on the units that are present after optimization, not on the hardware first inferred from the Verilog.

☑ Number of adders. ☑ Number of multipliers. ☑ Number of shifters. ☑ Number of multiplexors.

Adders, 2; multipliers, 0; shifters, 0, number of multiplexors, 1. See the solution to the previous part.

Problem 2: [20 pts] Appearing below is another encode module, this one has a new input `radix`, which indicates the radix (base) of the number to be converted. When completed the module should function like the module from the previous problem, except that the digits form a radix-`radix` number. For example, if `radix` were 10 it would operate like the previous module. If `radix` were 8 the digits would be octal, etc.

(*a*) Modify the module so that it takes into account the radix. Assume that `radix` can be any value from 2 to 16. Note that for a radix of 16 the valid digits are 0-9 and A-F (only consider upper case).

☑ Modify the module to generate the correct `non_digit` output.

☑ Modify the module to update `val_next` correctly given the radix.

Solution appears below. An `is_af` signal is added to detect legitimate hexadecimal digits. A `digit_val` value (value of the current digit) is computed which is correct for radix 2 to 16 (and higher). To detect if the current digit is valid (see `digit_in_range`), the hardware checks if it's in the range 0-9 or A-F. If so, it then looks at the value to make sure that it's less than `radix`.

Grading Notes: In many solutions incorrectly rejected digits in the range 0-9 when `radix` *was greater than 10. A surprisingly large number of solutions used case statements to compute* `non_digit` *with a case for each radix value.*

```
typedef enum {Char_0 = 48, Char_9 = 57, Char_A = 65, Char_F = 70} Chars;
module encode_radix #( int width = 32 )
   (output logic [width-1:0] val_next,
    output logic overflow,                    output uwire non_digit,
    input uwire [7:0] ascii_char,             input uwire [width-1:0] val_prev,
    input uwire [4:0] radix);

   logic [width+3:0] val_curr;

   logic [3:0] high_bits;  // SOLUTION: Remove bin_char, not used.

   // SOLUTION
   uwire   is_digit = ascii_char >= Char_0 && ascii_char <= Char_9;
   uwire   is_af    = ascii_char >= Char_A && ascii_char <= Char_F;
   uwire [3:0] digit_val = ascii_char - ( is_digit ? Char_0 : Char_A - 10 );
   uwire digit_in_range = ( is_digit || is_af ) && digit_val < radix;
   assign non_digit = !digit_in_range;
   always @* begin

      val_curr = radix * val_prev + digit_val; // SOLUTION: Multiply by radix.
      // SOLUTION ends here, text below is unchanged.

      high_bits = val_curr >> width;
      if ( non_digit ) begin
         overflow = 0;
         val_next = 0;
      end else begin
         overflow = high_bits != 0;
         val_next = val_curr;
      end
   end
endmodule
```

Problem 2, continued:

(*b*) Suppose that module `encode_radix` (from the previous part) were to be used in a larger design in which the values of `radix` could only be 2, 8, 10, and 16. Also suppose that the synthesis program can't figure out that `radix` is limited to these values. Why would the cost be higher than necessary, and how could `encode_radix` be modified to get the lower cost hardware?

☑ Explain why the cost will be higher than is necessary.

Short answer: The synthesis program will generate a regular multiplier when all that's really needed are some shifts and an add.

Longer explanation: The Verilog code uses a multiply operator in the expression assigning `val_curr`. For the decimal version of the hardware (from Problem 1 and the Homework assignment) one operand of the multiply is the constant 10, and so the multiplication operator will be synthesized as an adder (computing the sum `val_prev[width-1:1] + val_prev[width-1:3]` which is equivalent to `2 * val_prev[width-1:0] + 8 * val_prev[width-1:0]`). For part a of this problem where `radix` could take on any value a true multiplier had to be synthesized (albeit one in which one input was only four bits).

But in this part we are limiting the radices that are available, so we don't really need a full multiplier. In fact, other than radix 10, all we need to do is shift by a constant amount. The synthesis program could generate a much lower cost design IF it were aware of the limited range of `radix` values, but according to the problem it's not (meaning the synthesis program expects a full range of radix values).

☑ Show the changes to `encode_radix` so that the synthesis program *will* generate the lower cost design. The port definitions **cannot** be changed.

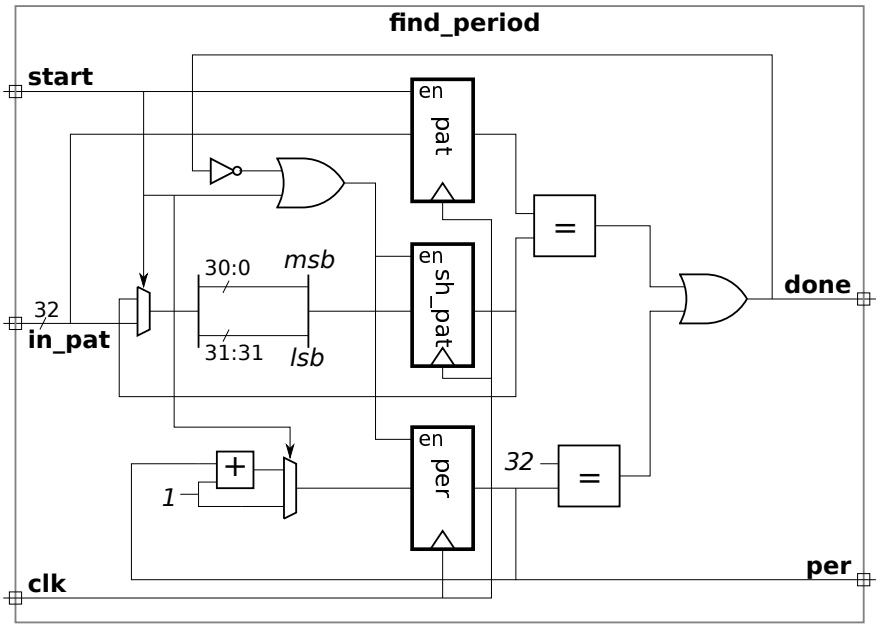
Since the problem is that the synthesis program will generate a real multiplier when we use `radix` with the multiply operator, we won't use radix with the multiplier operator. Instead we'll use a case statement, which is shown below. Notice that a `default` case is included, that's to make sure that a latch is not synthesized for `val_scaled`. Also note that the default case matches one of the other cases, that's to make sure that unused logic is not synthesized.

```
always_comb begin
   case ( radix )
     2: begin
        val_curr = 2 * val_prev + digit_val;
        high_bits = val_curr >> 2;
     end
     8: begin
        val_curr = 8 * val_prev + digit_val;
        high_bits = val_curr >> 8;
     end
     10: begin
        val_curr = 10 * val_prev + digit_val;
        high_bits = val_curr >> 10;
     end
     16: begin
        val_curr = 16 * val_prev + digit_val;
        high_bits = val_curr >> 16;
     end
     default: begin
        val_curr = 10 * val_prev + digit_val;
        high_bits = val_curr >> 10;
     end
   endcase
```

Problem 3: [20 pts] Appearing to the right is hardware and a corresponding Verilog module. The module is incomplete, finish it. *Hint: The hardware includes an end-around shift, that's the part with the msb/lsb labels.*

☑ Add sizes and other information to port declarations.

☑ Finish the Verilog code.



Solution appears below. The size for `sh_pat` is 32 bits since it's connected to the 32-bit end-around shift unit. The size of `pat` is 32 bits since it's connected to a 32-bit input port. The size of `per` is set to six bits based on the comparison with 32 in the diagram. A size of less than six bits could not hold a 32, and anything larger than six bits would not be needed because the value of `per` stops incrementing when it reaches a value of 32.

From the diagram we find three edge-triggered registers, `pat`, `sh_pat`, and `per`. Edge triggered registers are specified in Verilog using `always @ ( posedge clk )` constructs, in this case using a separate `always` block for each register is cleanest.

The `done` output was realized using a continuous assignment. Because all of the values needed for `done` are register outputs, the code for `done` could have been put in an `always` block, but only if there was a single `always` block for all three registers, which is not the case with the solution below.

```
module find_period
   (output logic [5:0]   per,       output uwire  done,
    input uwire [31:0]   in_pat,
    input uwire          start,     input uwire   clk);

   logic [31:0]          pat, sh_pat;

   always_ff @( posedge clk ) if ( start ) pat <= in_pat;

   uwire [31:0] sh_in = start ? in_pat : sh_pat;
   always_ff @( posedge clk )
     if ( start || !done ) sh_pat <= { sh_in[30:0], sh_in[31] };

   always_ff @( posedge clk )
     if ( start )      per <= 1;
     else if ( !done ) per <= per + 1;

   assign done = pat == sh_pat || per == 32;

endmodule
```

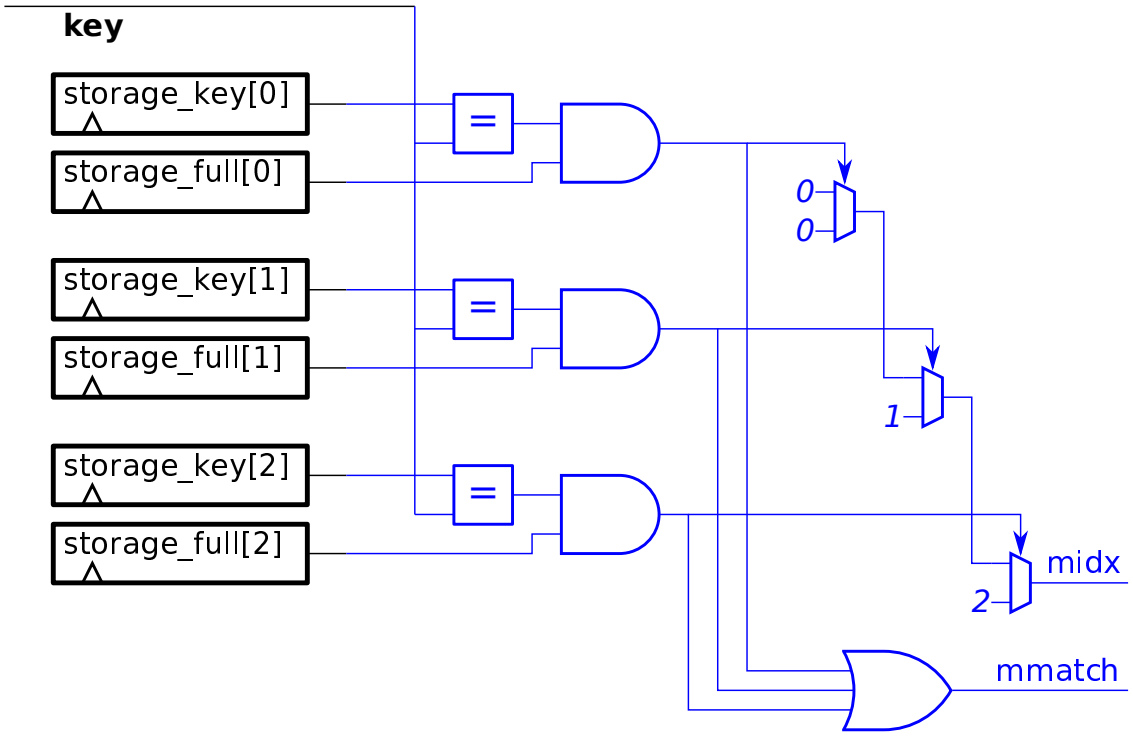
Problem 4: [20 pts] The Verilog below is the key lookup part of the simple CAM module used in class.

```
logic [dwid:1] storage_data [ssize];
logic [kwid:1] storage_key [ssize];
logic [ssize-1:0] storage_full;

always_comb begin
   mmatch = 0;   midx = 0;
   for ( int i=0; i<ssize; i++ )
     if ( storage_full[i] && storage_key[i] == key ) begin mmatch = 1; midx = i; end
end

assign out_data = storage_data[midx];
```

(*a*) Starting with the registers and `key` shown below, sketch the hardware synthesized for this code without optimization. The hardware should produce values for `mmatch` and `midx` (but not `out_data`). Do so for `ssize=3`. In class we often showed part of this as a box labeled "priority encoder" (or "pri" for short), in this problem actually show the hardware.

☑ Synthesized hardware for `ssize = 3` to generate ☑ `mmatch` and ☑ `midx`.

Solution appears below in blue. Note that the first (uppermost) multiplexor can trivially be optimized out since both of its inputs are zero. A chain of multiplexors was chosen to generate `midx`, and a similar chain could have been used for `mmatch`. However the OR gate performs the same operation and is much simpler.



(*b*) Assume that the cost of an a-bit comparison unit is a, and its delay is also a. Assume that the cost of an a-input, b-bit multiplexor is ab and the delay is 1. Compute the cost and delay of the logic used to compute `midx` in terms of `ssize` (use s in your formulas) and `kwid` (use k in your formulas). As with the previous part, do this for the unoptimized hardware. Remember to solve this for an arbitrary value of `ssize` (s), not for $s = 3$.

☑ Cost in terms of s and k:

From the diagram it's clear that there are s k-bit comparison units, they cost sk units. The multiplexors increase in size from the beginning to the end of the chain. The mux at the end of the chain must be large enough to hold the value $s-1$, which requires $\lceil \log_2 s \rceil$ bits. For simplicity assume that all multiplexors are that size, then the multiplexor cost is $2s\lceil \log_2 s \rceil$ units. The s-input OR gate can be assumed to have cost $s-1$ (by setting the cost of a 2-input OR gate at 1).

The $\boxed{\text{total cost is } sk + 2s\lceil \log_2 s \rceil + s - 1 \text{ units}}$.

☑ Delay in terms of s and k:

From the synthesized hardware it should be clear that the critical path used to compute the delay starts at the first comparison unit and continues through the multiplexor chain. The k-bit comparison takes time k, and the length-s multiplexor chain has delay s.

The $\boxed{\text{total delay is } k + s \text{ units}}$.

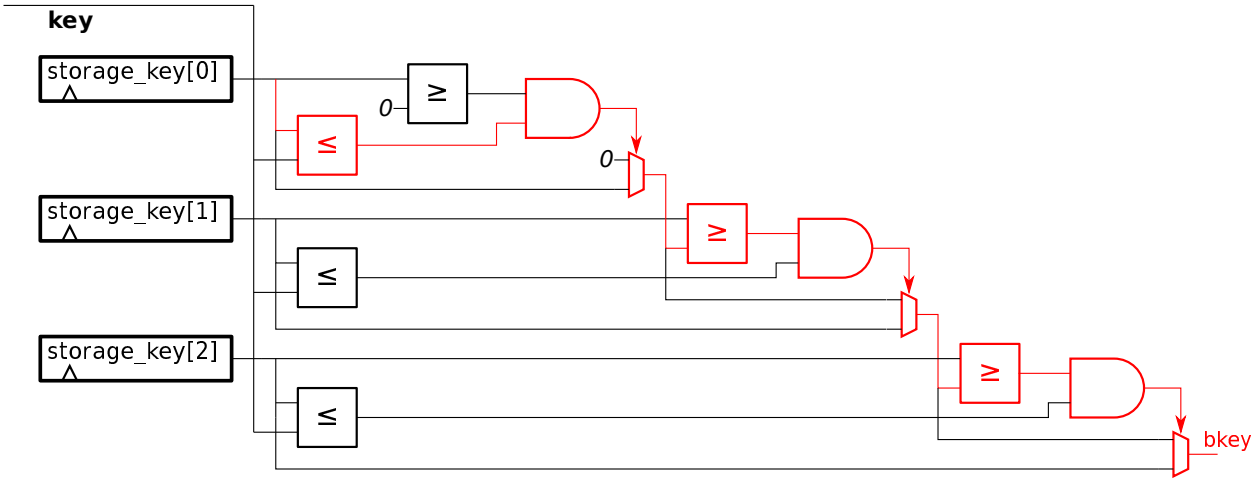
Problem 4, continued: Appearing below is a variation on the key lookup from the CAM module. Instead of finding a matching key it finds the largest stored key that is ≤ to the lookup key. Note that this version doesn't include `storage_full`.

```
logic [dwid:1] storage_data [ssize];
logic [kwid:1] storage_key [ssize];

always_comb begin
   midx = 0;  bkey = 0;
   for ( int i=0; i<ssize; i++ )
     if ( storage_key[i] >= bkey && storage_key[i] <= key ) // READ THIS LINE CAREFULLY
       begin   midx = i;  bkey = storage_key[i];   end
end

assign out_data = storage_data[midx];
```

(*c*) Sketch the hardware for `ssize=3`.

☑ Sketch the synthesized hardware needed to generate `bkey`.

Solution appears below, with the critical path shown in red. An important thing to notice is that the ≥ comparison at iteration `i` is being made with the value of `bkey` produced in iteration `i-1`. Those values of `bkey` pass through the multiplexor chain, and for that reason the delay in this circuit is significantly longer than in the version from the previous part. See the next sub-part.



(*d*) Compute the cost and performance in terms of `ssize` (use s) and the key size (use k). As before a k-bit comparison unit (equality or magnitude) costs k and has a delay of k and an a-input, b-bit mux costs ab and has a delay of 1. *Hint: There's a big difference.*

☑ Cost in terms of s and k:

There are now $2s$ comparison units, costing $2sk$ cost units. The s multiplexors now carry k-bit values, so their cost is $2sk$. Plus there are s AND gates which we'll set at cost s. The $\boxed{\text{total cost is } 4sk + s \text{ units}}$, which is significantly higher.

☑ Delay in terms of s and k:

In the diagram of the synthesized hardware the critical path appears in red. As before, the critical path passes through the multiplexor chain, but this time the ≥ units are also on the critical path. The critical path includes now s ≥ units, s muxen, and s AND gates. The $\boxed{\text{total delay is } s(k+2) \text{ units}}$, which is significantly higher than the delay of the first version used in this problem.

Problem 5: [20 pts] Answer each question below.

(*a*) The module below is supposed to count from 0 to `max` (inclusive), then return to zero. Strictly speaking it does, but there are problems, including the fact that it's not synthesizable. Fix the problems.

```
module counter #(int max = 3)(output logic [7:0] count, input uwire clk);

   always @( posedge clk ) begin

      count <= count + 1;

   end

   always @* begin

      if ( count == max ) count <= 0;

   end

endmodule
```

☑ Why isn't the module synthesizable?

It's not synthesizable because `count` is assigned in two different `always` blocks.

☑ Fix the problem.

Just combine the two blocks:

```
module counter #(int max = 3)(output logic [7:0] count, input uwire clk);

   always @( posedge clk ) count <= count == max ? 0 : count + 1;

endmodule
```

(*b*) There is a problem with the module below due to the way that `a` is declared.

```
module sa1(output uwire a, input uwire c, d);

   always_comb begin

      a = c & d;

   end

endmodule
```

The problem is that `a` is being declared as a net type (which includes `uwire`) but it is being assigned in procedural code. Anything assigned in procedural code must be a variable type.

☑ Fix the problem **by changing** the declaration of `a`.

```
module sa1(output logic a, input uwire c, d);
   // SOLUTION: Declare a as a variable type (change uwire a to logic a).

   always_comb begin
      a = c & d;
   end
endmodule
```

☑ Fix the problem **without changing** the declaration of `a`.

```
// SOLUTION
module sa1(output uwire a, input uwire c, d);
   assign a = c & d;
endmodule
```

(*c*) Describe a situation in which using `always_comb` has a benefit over using `always @*`.

☑ Situation where `always_comb` helps.

In the code below `x` is not always assigned and so it could be synthesized into a latch (level-triggered flip-flop). But the SystemVerilog-literate programmer used `always_comb` because he or she intended purely combinational logic—no latches. The fact that `x` was not always assigned was an oversight on the part of the programmer. Because `always_comb` was used well-written Verilog tools will warn the programmer about this. That's how it helps.

```
   always_comb begin
      if ( a < 10 )
        x = a + b;
      else if ( a > 1000 )
        x = a - b;
   end
```

(*d*) The module below is supposed to be computing $x^2 + y^2$.

```
module sa2(output logic [63:0] sos, input uwire [63:0] x, y);

   logic [63:0] a1, b1, a2, b2;
   uwire [63:0]  p, s;

   fpmul f1(p,a1,b1);
   fpadd f2(s,a2,b2);

   always @* begin

      // Compute x^2.
      a1 = x;  b1 = x;
      #1;
      sos = p;

      // Compute y^2.
      a1 = y;  b1 = y;
      #1;

      // Compute x^2 + y^2.
      a2 = p;  b2 = sos;
      #1;
      sos = s;

   end

endmodule
```

☑ Explain why the module is not synthesizable.

It's not synthesizable because it uses delays.

☑ Fix the problem.

The module is trying to use `fpmul` twice. Since there is no clock input, there is no way to do that. A simple solution would be to instantiate a second `fpmul` and connect it appropriately, that's the solution shown below. (A more complex solution would use a `clk` input and use the same multiplier over two cycles.)

```
// SOLUTION
module sa2sol(output uwire [63:0] sos, input uwire [63:0] x, y);

   uwire [63:0]  p, s;

   fpmul fm1(p,x,x);
   fpmul fm2(s,y,y);
   fpadd f2(sos,p,s);

endmodule
```

25 Spring 2001 Solutions

Name Solution

Digital Design Using Verilog
EE 4702-1
Midterm Examination

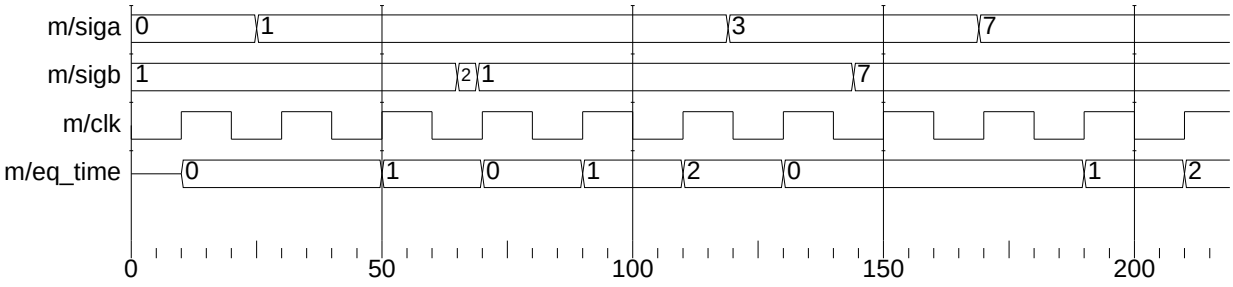
16 March 2001 8:40-9:30 CST

Problem 1		(30 pts)
Problem 2		(25 pts)
Problem 3		(35 pts)
Problem 4		(10 pts)
Exam Total		(100 pts)

Alias always @(posedge)

Good Luck!

Problem 1: Complete the Verilog behavioral description below so that it operates as follows. Compute 32-bit output `eq_time` so that it is the number of consecutive positive edges of input `clk` for which 32-bit inputs `siga` and `sigb` remain equal. The counting should start on the first positive edge of `clk` after `siga` becomes equal to `sigb`; the count starts at zero at the moment they become equal, and while they remain equal the count is incremented at each positive edge. The count should go back to zero at the first positive edge of `clk` after `siga` becomes unequal to `sigb`. The count goes to zero even if `siga` and `sigb` become equal again before the positive edge. Sample output appears in the timing diagram below. (30 pts)



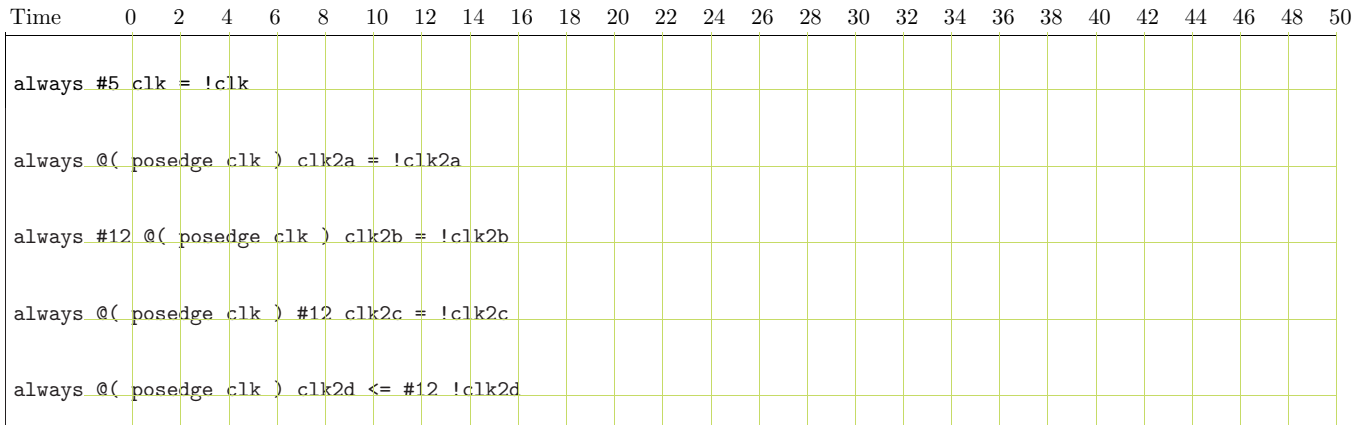

```
module monitor(eq_time, siga, sigb, clk);
   input siga, sigb, clk;
   output eq_time;
   // Don't forget to declare port types.

   // Solution:
   wire [31:0] siga, sigb;
   wire        clk;
   reg [31:0]  eq_time;

   reg [10:0]  next_count;

   always @( siga or sigb ) if ( siga != sigb ) next_count = 0;

   always @( posedge clk )
     begin
        eq_time = next_count;
        if ( siga == sigb ) next_count = next_count + 1;
     end

endmodule
```

Don't get bogged down: There are eight more problems, some can be answered quickly.

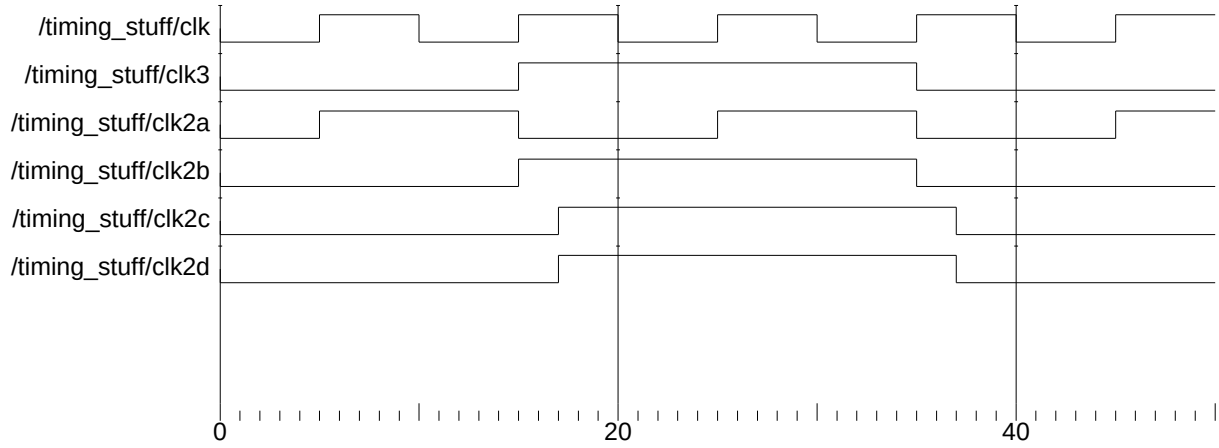
Problem 2: Complete the following timing diagram problems.

(*a*) Complete the timing diagram below. (15 pts)

```
module timing_stuff();
   reg clk, clk3, clk2a, clk2b, clk2c, clk2d,
   initial begin
      clk = 0; clk2a = 0; clk2b = 0; clk2c = 0; clk2d = 0; clk3 = 0;
   end
```

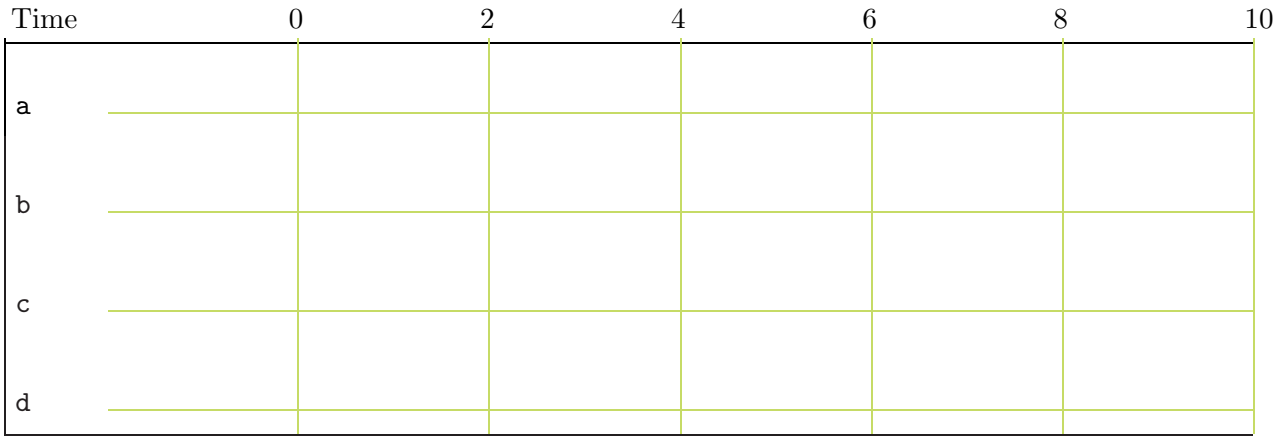


Solution:

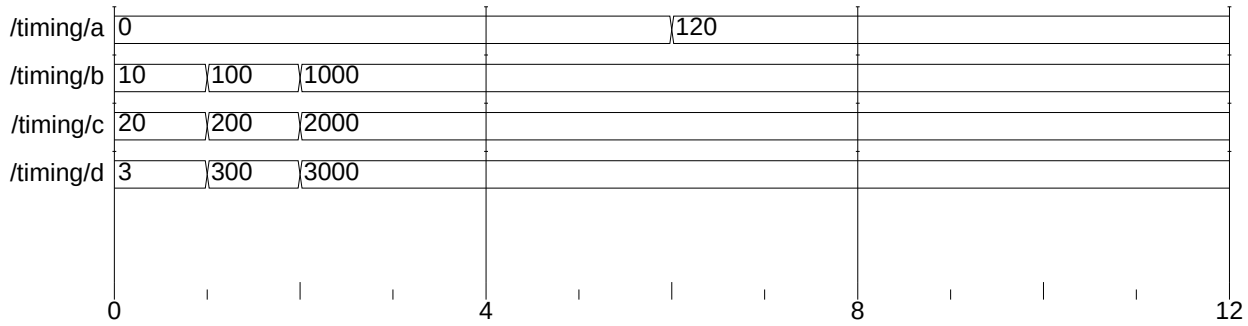

/timing_stuff/clk
/timing_stuff/clk3
/timing_stuff/clk2a
/timing_stuff/clk2b
/timing_stuff/clk2c
/timing_stuff/clk2d
0
20
40

(*b*) Complete the timing diagram below. Be sure to clearly indicate when a signal value changes. (10 pts)

```
module timing();
   integer a, b, c, d;
   initial begin
      a = 0;
      b = 10;
      c = 20;
      d <= #0 3;
      d = 30;
      d <= #1 300;
      d <= #2 3000;
      #1;
      b = 100;
      c <= 200;
      a <= #5 b + c;
      #1;
      b = 1000;
      c <= 2000;
      #10;
   end
endmodule
```



Solution:



Problem 3: Answer each question below. Some can be answered quickly, try answering those questions first.

(*a*) The `match_count_x` modules below are *supposed to* count the number of times input `symbol` is the same as input `targ`. Output `count` should be incremented if `symbol` is the same as `targ` after a change in `symbol`. Most or all of the modules below don't work properly. For each non-working module describe the problem and how it is simulated. *It is important to describe how the incorrect Verilog is simulated and why it is wrong.*

Port declarations and initializations are not shown, but assume they are present and correct. Behavior for unknown and high-impedance values is undefined. In other words, the problems are **not** related to declarations, initialization, or unknown values. (10 pts)

```
module count_match_1(count,symbol,targ); // Declarations and init. not shown.

   always wait ( symbol == targ ) count = count + 1;

endmodule
```

(4 pts) Because an iteration of `always` is done without any delay the simulator "freezes" when `symbol` is equal to `targ` as `count` is continually updated, there is no chance for `targ` or `symbol` to change.

```
module count_match_3(count,symbol,targ); // Declarations and init. not shown.

   always #10 if ( symbol == targ ) count = count + 1;

endmodule
```

(3 pts) Rather than incrementing `count` on each change in `symbol`, the code above increments `count` on ten-cycle intervals when `symbol` is equal to `targ`. It does not increment `count` when `symbol` changes, it might miss times that `symbol` is equal to `targ` (when `symbol` changes several times in the ten-cycle interval) and it will increment `count` multiple times if `symbol` remains equal to `targ` at least 20 cycles.

```
module count_match_4(count,symbol,targ); // Declarations and init. not shown.

   always @( symbol == targ ) count = count + 1;

endmodule
```

(3 pts) Variable `count` is incremented when `symbol` becomes equal to `targ` and when `symbol` becomes unequal to `targ`.

(*b*) Show how each of the three adders below can be used in the module `use_adders` to add seven to input `a`. **Do not** modify the adders themselves. (10 pts)

```
module adder1(x,a,b);
   input a, b;
   output x;
   wire [31:0] a, b;
   wire [31:0] x = a + b;
endmodule

module adder2(x,a);
   input a;
   output x;
   parameter b = 0;
   wire [31:0] a;
   wire [31:0] x = a + b;
endmodule

‘define b 7  // Part of solution.
module adder3(x,a);
   input a;
   output x;
   wire [31:0] a;
   wire [31:0] x = a + ‘b;
endmodule


module use_adders(x_1,x_2,x_3,a);
   input a;
   output x_1, x_2, x_3;  // Each output should be a + 7
   // Use adder1, adder2, and adder3 to generate respective x_ outputs.

   // Solution
   wire [31:0] x_1, x_2, x_3, a;

   adder1 a1(x_1,a,32’d7);

   adder2 #(7) a2(x_1,a);

   adder3 a3(x_1,a);




endmodule
```

(*c*) Show the values that will be assigned in each assignment to `r`. Variables `a`, `c`, and `r` are six-bit registers. (5 pts)

```
a = 6'b101010;
c = 6'bx1x0x1;

r = & a;  // Solution:  r set to 0

r = | a; // Solution:  r set to 1

r = ^ a;  // Solution:  r set to 1

r = & c;  // Solution:  r set to 0

r = | c; // Solution:  r set to 1

r = ^ c;  // Solution:  r set to  x
```

(*d*) Do the two code fragments below do the same thing? If not, how do they differ? (5 pts)

```
// Fragment A.

if ( foo > bar ) x = x + 1; else y = y + 1;


// Fragment B.

case ( foo > bar )
  1: x = x + 1;
  default: y = y + 1;
endcase
```

They do not differ.

(*e*) Why can't the following increment macro be re-written as a function or task in Verilog 95? (5 pts)

```
‘define incr(a) a=a+1
// ...
  // Sample uses of macro.
  for (i=0; i<10; ‘incr(i)) x = x + y;
  for (j=0; j<10; ‘incr(j)) begin foo(j); k = k + x; end
```

In Verilog 95 the third item in the for must be an assignment statement, so a task or function wouldn't work. A function could be used in SystemVerilog.

Problem 4: The module below counts the number of five's and nine's appearing at input `c`. Explain exactly when five's and nine's are counted (start cycle and end cycle), and describe any restrictions on the counts. (10 pts)

```
module yet_another_symbol_counter(fives, nines, c);
   input c;
   output fives, nines;
   wire [7:0] c;
   reg [31:0] fives, nines;

   initial fork

      begin
         fives = 0;
         nines = 0;
      end

      #50 fork:A
         repeat ( 42 ) @( c ) if ( c == 5 ) fives = fives + 1;
         #100 disable A;
      join

      #70 fork:B
         forever @( c ) if ( c == 9 ) nines = nines + 1;
         #200 disable B;
      join

   join

endmodule
```

The module counts fives that appear between 50 and 150 cycles into the simulation. No more than 42 new symbols appearing after cycle 50 are examined for fives. (The maximum number of fives that can be counted is 21.)

The module counts nines that appear between 70 and 270 cycles into the simulation. The number of nines that can be counted is limited only by the size of **nines**, 32 bits.

Name Solution

Digital Design Using Verilog
EE 4702-1
Final Examination

9 May 2001 7:30-9:30 CDT

Problem 1		(15 pts)
Problem 2		(18 pts)
Problem 3		(17 pts)
Problem 4		(18 pts)
Problem 5		(12 pts)
Problem 6		(20 pts)
Exam Total		(100 pts)

Alias Not Synthesizable

Good Luck!

Problem 1: The module below is in an explicit structural form.

(*a*) Re-write the module in behavioral form. The delays can be assumed to be pipeline delays. (10 pts)

(*b*) What is the difference between pipeline and inertial delays? Which kind of delay is used in your solution to the problem above? (5 pts)

In a pipeline delay of duration t units each signal change will appear t units later, regardless of other changes that occur in the interim. The delays in nonblocking delayed assignments, such as `a <= #3 b;`, are pipeline delays. In an inertial delay of duration t units a signal change (from an old to a new value) only appears if the new value does not change for t units (until the change is visible). Delays on gates and wires, such as `and #3 a1(x,a,b);`, are inertial delays.

```
module expl_str(x,y,a,b,c);
   input a, b, c;
   output x, y;
   wire   a, b, c, x, y;
   wire   na, nb, nc, t3, t5, t6;

   not n1(na,a);
   not n2(nb,b);
   not n3(nc,c);
   and #1 a1(t3,na,b,c);
   and a2(t5,a,nb,c);
   and a3(t6,a,b,nc);
   or o1(x,t3,t6);
   or #3 o2(y,a,t5);

endmodule


// Solution
module behav(x,y,a,b,c);
   input  a, b, c;
   output x, y;
   wire   a, b, c;
   reg    x, y;
   reg    t3;

   // The delays in expl_str are inertial delays, the delays here
   // are pipeline delays.

   // Note that t3 is delayed but t6 is not.

   always @( a or b or c ) t3 <= #1 !a & b & c;

   always @( a or b or c or t3 ) x = t3  a & b & !c;

   // Code below can be simplified to y <= #3 a;
   always @( a or b or c ) y <= #3 a  a & !b & c;

endmodule
```

Problem 2: The module below sets output `rot` to the number of times that input `a` must be rotated (end-around shifted) to obtain the value on input `b`, or to 32 if `a` is not a rotated version of `b`.

(*a*) Write a testbench module that tests `rots` with input pairs `a`=0,`b`=0; `a`=0,`b`=1; `a`=0,`b`=2; and `a`=0,`b`=3. (The `rot` output should be zero for the first pair and 32 for the others.) The testbench should include an integer `err` and set it to the number of incorrect outputs.

It is important that the testbench makes correct use of `ready` and `start`. (Part of the problem is determining just what is "correct use.") The testbench should use `ready` rather than assumed timing. Also, test only a single instance of `rots` and don't forget the clock. (18 pts)

```
module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;        output ready, rot;
   reg         ready;             wire [31:0] a, b;
   reg [5:0]   rot;               wire        start, clk;
   reg [31:0]  acpy;
   initial rot = 0;
   always @( posedge clk ) begin
      ready = 1;   while ( !start ) @( posedge clk );
      ready = 0;   while (  start ) @( posedge clk );
      rot = 0;  acpy = a;
      while (  acpy != b  &&  rot < 32  ) @( posedge clk ) begin
         acpy = { acpy[30:0], acpy[31] };
         if ( acpy == a ) rot = 32; else rot = rot + 1;
      end
   end
endmodule


module testrot();
   reg [31:0] b;               wire       rdy;
   reg        start, clk;      wire [5:0] r;
   integer    i, err;

   rots myrots(rdy, r, start, 32d'0, b, clk);

   always #1 clk = !clk;

   initial begin
      err = 0;  start = 0;  clk = 0;
      wait(rdy);
      for (i=0; i<4; i=i+1) begin
         b = i;
         start = 1; wait(!rdy);
         start = 0; wait( rdy);
         if ( !b && r ) err = err + 1;
         if (  b && r != 32 ) err = err + 1;
      end
      $display("Error count: %d",err); $stop;
   end
endmodule
```

Problem 3: Convert the `rots` module (repeated below) to synthesizable Form 2 (edge-triggered flip-flops). Do not change the ports or what it does. In particular, `ready` and `start` must be used the same way. Ignore reset. (17 pts)

```
module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;        output ready, rot;
   reg         ready;             wire [31:0] a, b;
   reg [5:0]   rot;               wire        start, clk;
   reg [31:0]  acpy;
   initial rot = 0;
   always @( posedge clk ) begin
      ready = 1;   while ( !start ) @( posedge clk );
      ready = 0;   while (  start ) @( posedge clk );
      rot = 0;  acpy = a;
      while (  acpy != b  &&  rot < 32  ) @( posedge clk ) begin
         acpy = { acpy[30:0], acpy[31] };
         if ( acpy == a ) rot = 32; else rot = rot + 1;
      end
   end
endmodule
```

Solution on next page.

```
module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;
   output ready, rot;  // Don't forget port types and other declarations.
```

```
         acpy = { acpy[30:0], acpy[31] };
         if ( acpy == a ) rot = 32; else rot = rot + 1;
```

```
endmodule
```

```
 /// Solution 1, using named states and assuming little about start.

module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;
   output ready, rot;

   wire [31:0] a, b;
   reg [5:0]   rot;
   wire        start, clk;
   reg [31:0]  acpy;
   reg [1:0]   state;

   parameter   st_ready = 2'b01;
   parameter   st_wait  = 2'b00;
   parameter   st_go    = 2'b10;

   wire        ready = state[0];

   initial begin rot = 0; state = st_ready; end

   always @( posedge clk )
     case ( state )
       st_ready:
         if ( start ) state = st_wait;
       st_wait:
         if ( !start ) begin  rot = 0;  acpy = a;  state = st_go;  end
       st_go:
         if (  acpy != b  &&  rot < 32  ) begin
            acpy = { acpy[30:0], acpy[31] };
            if ( acpy == a ) begin rot = 32; state = st_ready; end
            else rot = rot + 1;
         end else begin
            state = st_ready;
         end
     endcase
endmodule
```

```
 /// Solution 2, basing state on ready and assumed behavior of start.

module rots(ready, rot, start, a, b, clk);
   input a, b, start, clk;
   output ready, rot;

   reg         ready;
   wire [31:0] a, b;
   reg [5:0]   rot;
   wire        start, clk;
   reg [31:0]  acpy;

   initial begin rot = 0; ready = 1; end

   always @( posedge clk )
     case ( {ready,start} )
       {2'b10}:; // Wait for start to go to one.
       // Unlike original module, gets value of "a" when start goes
       // to 1, not when start goes to zero. (This is where behavior assumed.)
       {2'b11}: begin ready = 0; acpy = a; rot = 0; end
       {2'b01}:; // Wait for start to go to zero.
       {2'b00}:
         if (  acpy != b  &&  rot < 32  ) begin
            acpy = { acpy[30:0], acpy[31] };
            if ( acpy == a ) begin rot = 32;  ready = 1; end
            else rot = rot + 1;
         end else begin
            ready = 1;
         end
     endcase
endmodule
```

Problem 4: Two synthesizable descriptions appear below.

(*a*) In what synthesizable form is the Verilog description below? (2 pts)

Form 1: combinational logic, level triggered.

(*b*) Draw a schematic showing the approximate RTL-level description generated by a synthesis program like Leonardo. (7 pts)

```
module whatsyna(x, y, z, a, b, op);
   input a, b, op;
   output x, y, z;
   wire [7:0] a, b;
   wire [1:0] op;
   reg [7:0]  x, y, z;

   always @( op or a or b ) begin

      if ( a == 0 ) y = b;

      if ( a < b ) z = a; else z = b;

      case ( op )
        0: x = a + b;
        1: x = a;
        2: x = b;
      endcase

   end

endmodule
```

If you're an LSU ECE student in a Verilog-related course ask for a complete solution. For now: Output `y` is connected to a level-triggered flip-flop with enable input `a==0` and data input `b`.

Output `z` is connected to a two-input mux, controlled by `a < b`.

Output `x` is connected to a level triggered flip-flop enabled by `op != 3` . The data input is a mux controlled by `op`.

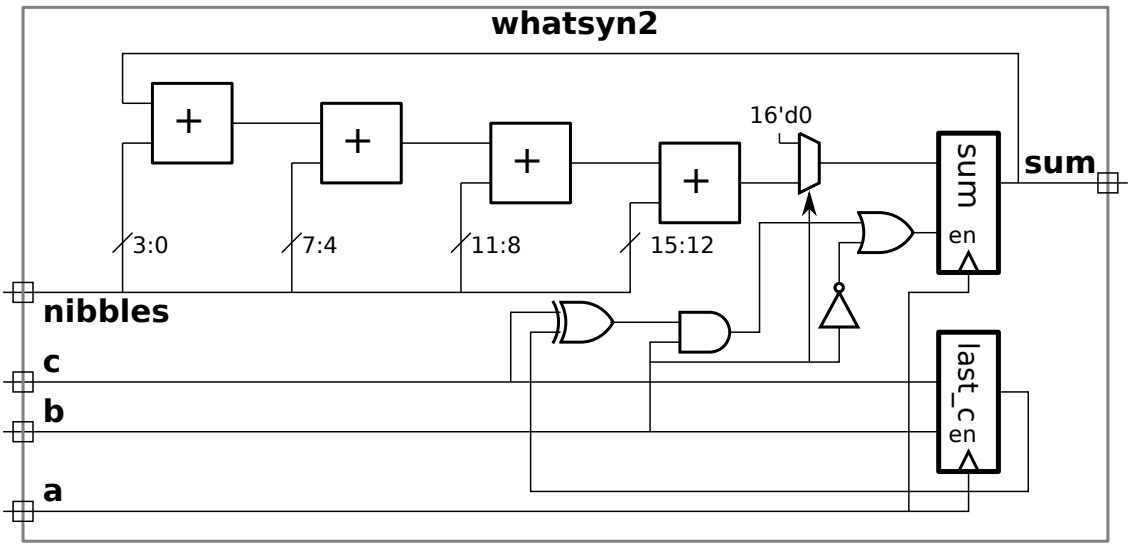
Problem 4, continued:

(*c*) (2 pts) In what synthesizable form is the Verilog description below? Form 2: Edge triggered logic.

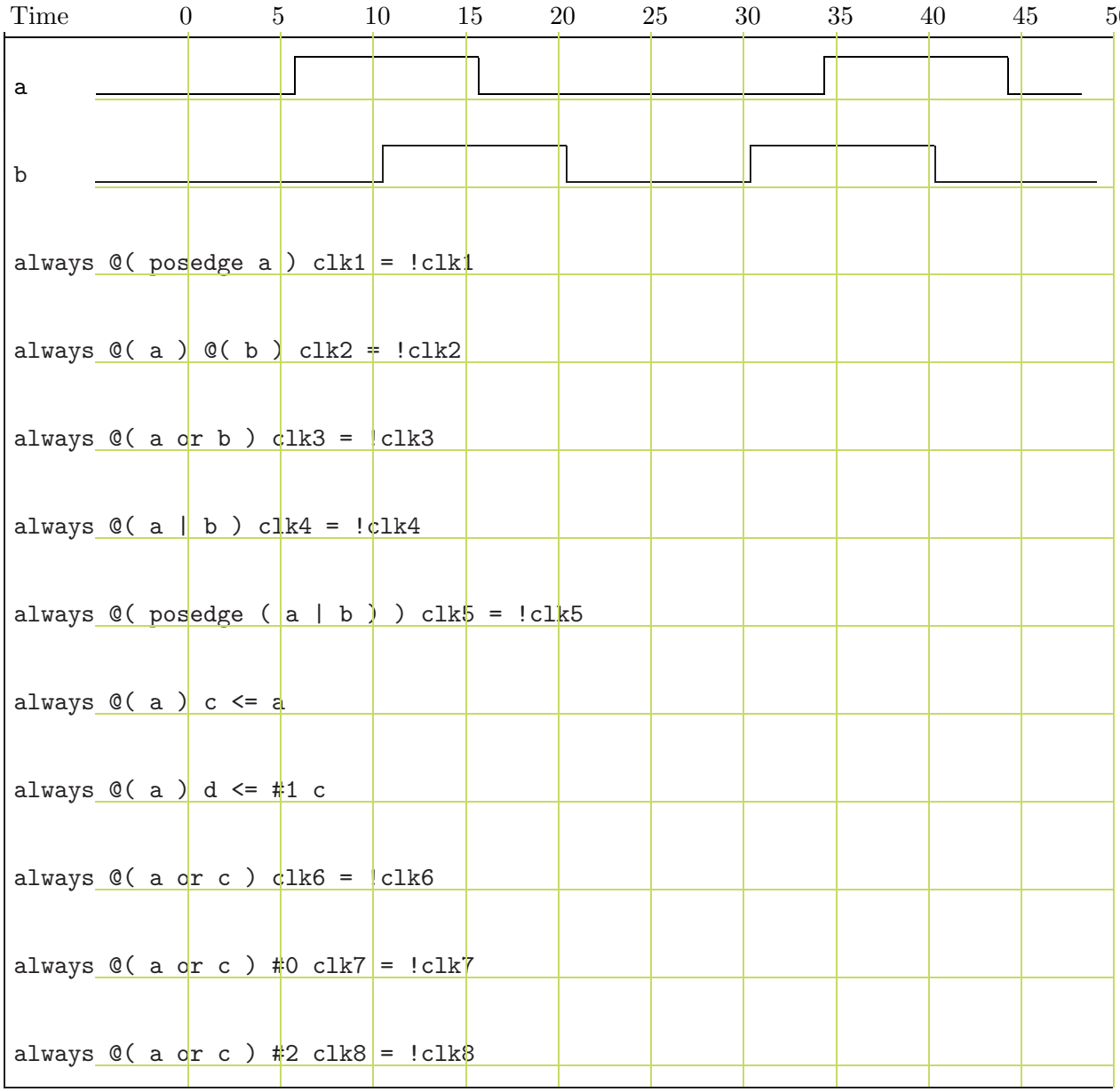
(*d*) (7 pts) Draw a schematic showing the approximate RTL-level description generated by a synthesis program like Leonardo. *Grading Note: In the 2001 version the event control was* `posedge a or negedge b`.

```
module whatsyn2(output [6:0] sum, input [15:0] nibbles, a, b, c);
   logic [15:0] n2;
   logic        last_c;

   always @( posedge a )
     if ( !b ) begin
        sum = 0;
     end else begin

        if ( c != last_c ) begin
           n2 = nibbles;
           for ( int i=0; i < 4; i++ ) begin
              sum = sum + n2[3:0];
              n2 = n2 >> 4;
           end
        end
        last_c = c;

     end
endmodule
```

Solution appears below. Output `sum` is driven by an edge-triggered register clocked by `a`. The nibbles input is connected to a cascade of four adders, the first adder connected to `sum`, the others each connected to a different four bits of `nibbles`. Note that no logic is synthesized for the shift operator, that only determines bit numbers for the adder inputs. The `sum` register is reset by `!b` and enabled by `b` and `c` ⊕ `last_c`.

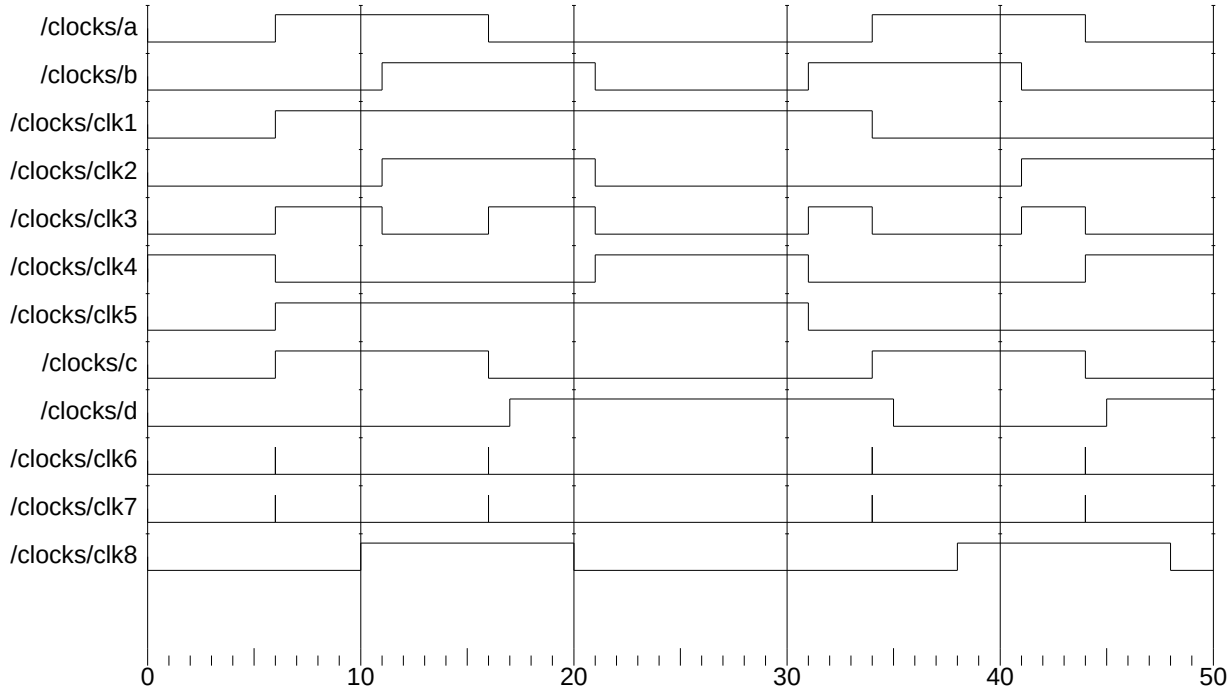


Problem 5: In the diagram below `c`, `d`, and identifiers starting with `clk` are all initialized to zero. Complete the timing diagram. (12 pts)



Solution on next page.

Solution to Problem 5:



Problem 6: Answer each question below.

(*a*) The code below, based on the Homework 3 solution, simulates properly before synthesis but in the post-synthesis simulation the testbench reports an incorrect beep time.

What goes wrong? Fix the problem without modifying the code below the indicated line. *Hint: The beep can start (and stop) at a slightly different time than the code below.* (5 pts)

```
module beepprob(beep, clk);
   input clk;
   output beep;


   // Code from exam: assign beep = | beep_timer;
   // Solution: Set beep on negative edge, after beep_timer computed.
   reg       beep;
   always @( negedge clk )  beep =  beep_timer;



   // DO NOT MODIFY CODE BELOW THIS LINE.
   always @( posedge clk ) begin
      // Lots of stuff;

      if ( beep_timer ) beep_timer = beep_timer - 1;

   end

endmodule
```

(*b*) Describe something that a parameter can be used for that an ordinary input port cannot and something that an input port can be used for that a parameter cannot. (5 pts)

Of course, the two are completely different things. Parameters can be used to set the size of vectors, an input value could not do that. An input can change, parameters are constant.

(*c*) What is the difference between `case`, `casex`, and `casez`? (5 pts)

In a `case` statement there must be a bitwise match, including unknowns and high impedance values, between the case expression and a case item. In a `casex` statement an unknown value acts as a wildcard matching any bit in the corresponding position, `casez` is similar with high impedance acting as the wildcard.

(*d*) Explain how each of the three statements below behave differently with unknown values. In particular, explain what has to be unknown and how the results of each statement is different. (5 pts)

```
m1 = a > b ? c : d;

if ( a > b ) m2 = c; else m2 = d;

case ( a > b )
  1: m3 = c;
  default: m3 = d;
endcase
```

The three behave identically if `a > b` is not unknown. If it is unknown the `m1` statement assigns a bitwise combination of `c` and `d`. (For the bit positions where `c` and `d` hold the same value `m1` is set to that value, in positions where `c` and `d` differ the corresponding position `m1` is set to unknown.)

If `a > b` is unknown `d` is assigned to `m2` and `m3`.

26 Spring 2000 Solutions

Name Solution

Digital Design Using Verilog
EE 4702-1
Midterm Examination

5 April 2000 8:40-9:30 CDT

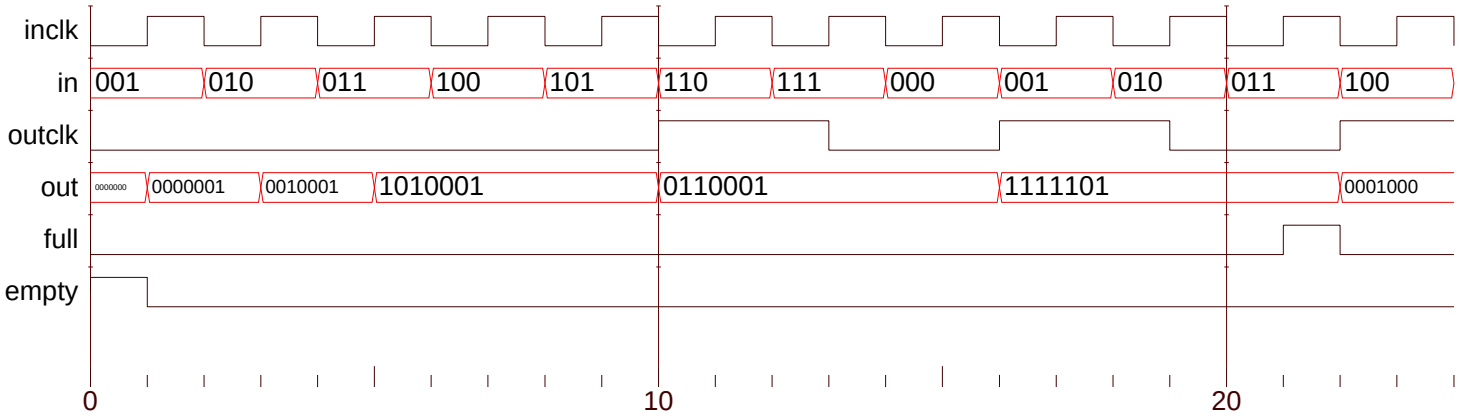
Problem 1 _____ (40 pts)

Problem 2 _____ (60 pts)

Alias always @(posedge) Exam Total _____ (100 pts)

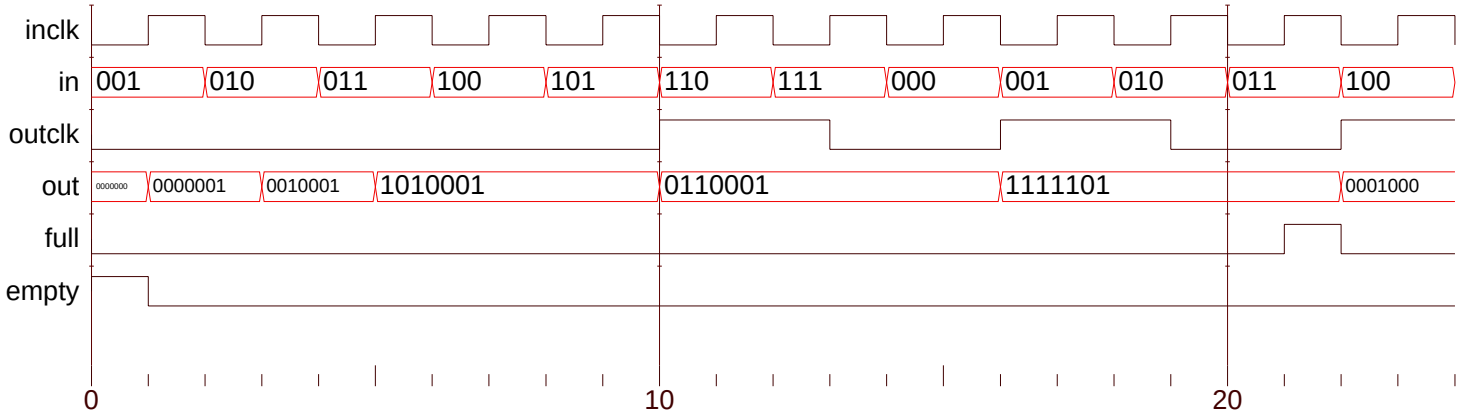
Good Luck!

Problem 1: Complete the Verilog description (below) of a FIFO-like module which has a 3-bit data input, `in`; a 7-bit output, `out`; 1-bit inputs `inclk` and `outclk`; and 1-bit outputs `full` and `empty`. The module operates like a FIFO (first in, first out) except that the width of the data input and output ports are different: it reads data 3 bits at a time (on a positive edge of `inclk`) and outputs 7 bits at a time (consisting of data from two input words plus one bit of a third). Unless the module has less than 3 bits of space left, on a positive edge of `inclk` the value on `in` is stored. The oldest 7 bits stored by the module always appear on output `out`. On a positive edge of `outclk` the oldest 7 bits are removed and the output displays the next 7 bits. Output `full` is 1 if the module cannot accept another 3 bits of input and is 0 otherwise; output `empty` is 1 if the module is empty and is 0 otherwise. Parameter `storage` is the total number of bits stored by the module. An example of the module operating is shown in the timing diagram below. (40 pts)




```
module width_change(out,full,empty,outclk,in,inclk);
   input outclk, in, inclk;
   output out, full, empty;

   parameter storage = 20;

   wire [6:0] out; // Can change to reg for solution.
   wire [2:0] in;
   wire       inclk, outclk;
   wire       full, empty; // Can change to reg for solution.

   reg [storage-1:0] sto;  // Storage for data.
   integer           amt;  // Number of occupied bits in sto.

   // USE THE NEXT PAGE FOR THE SOLUTION.

endmodule // width_change
```

Problem 1, continued: The diagram and code from the previous page are repeated below.




```
module width_change(out,full,empty,outclk,in,inclk);
   input outclk, in, inclk;
   output out, full, empty;

   parameter storage = 20;

   wire [6:0] out; // Can change to reg for solution.
   wire [2:0] in;
   wire       inclk, outclk;
   wire       full, empty; // Can change to reg for solution.

   reg [storage-1:0] sto;  // Storage for data.
   integer           amt;  // Number of occupied bits in sto.

   // Solution goes here.
   initial begin amt = 0; sto = 0; end

   assign full =  amt + 3 > storage;
   assign empty = amt === 0;
   assign out =   sto[7-1:0];

   always @( posedge outclk )
     if( amt >= 7 ) begin
        sto = sto >> 7;
        amt = amt - 7;
     end

   always @( posedge inclk )
     if( !full ) begin
        sto = sto | in << amt;
        amt = amt + 3;
     end


endmodule // width_change
```

Problem 2: Answer each question below.

(*a*) Describe something that a function can do (or be used for) that a task cannot. Describe something that a task can do (or be used for) that a function cannot. (10 pts)

A function can be used in an expression (but a task cannot). A task can include delays, but a function cannot.

(*b*) Convert the following behavioral code to **explicit** structural code. (10 pts)

```
module btos(x, a, b);
   input a, b;
   output x;
   wire   a, b;
   reg    x;

   always @( a or b ) if( a ) x = b; else x = ~b;

endmodule // btos
```

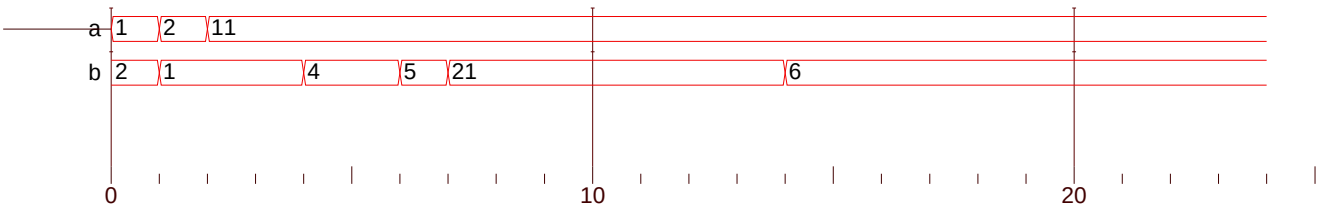
If you don't see the logical function performed, draw a truth table. The function, $\mathtt{x} = \overline{\mathtt{a} \oplus \mathtt{b}}$, can be performed by a primitive gate (`xnor`), a solution consisting of several other gates realizing the same function would also receive full credit.

```
module explicit(x, a, b);
   input a, b;
   output x;
   wire   a, b;
   wire   x;                    // Wire, not reg.

   xnor (x,a,b);

endmodule
```

(*c*) Show the changes (values and times) to `a` and `b` in the module below. (10 pts)

```
module assig();
   reg [15:0] a, b;
   initial
     begin
        a = 1;
        b = 2;
        #1;
        a <= b;
        b <= a;
        #1;
        a <= b + 10;
        b <= #5 b + 20;
        #1;
        b = #1 3;
        b <= 4;
        b <= #2 5;
        b <= #10 6;
        b = 7;
        #20;
     end
endmodule
```

Note that `b = #1 3;` is a *blocking* assignment. The condition is evaluated immediately (since it's 3 here evaluation time doesn't matter) and the assignment is done after the delay. Following statements are executed after the assignment. The solution is plotted below.

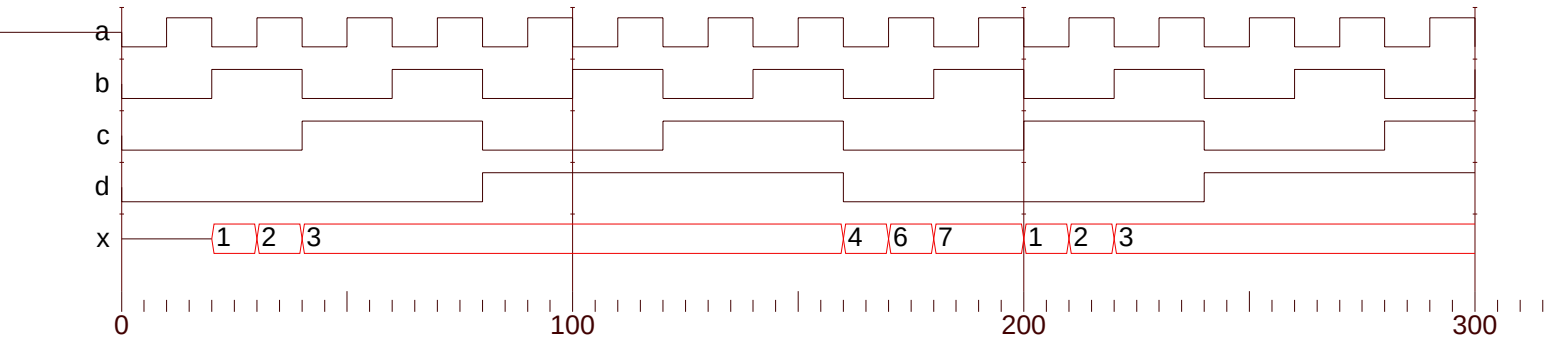


(*d*) Show the changes (values and times) to `x` in the module below using the timing diagram provided. (10 pts)

```
module events1();
   wire a, b, c, d;
   reg [2:0] x;
   reg [3:0] i;
   assign  {d,c,b,a} = i;

   initial begin
     i = 0;
     forever #10 i = i + 1;
   end

   always begin
      #15;
      @( a );
      x = 1;
      @( posedge a ) x = 2;
      @( a or b ) x = 3;
      @( a | b | c | d ) x = 4;
      wait( a | b ) x = 5;
      wait( a ) x = 6;
      wait( ~a ) x = 7;
   end // always begin
endmodule // events1
```

An event control, `@( foo )`, delays execution until `foo` *changes*. A wait statement, `wait( foo )`, delays execution until `foo` is nonzero (or true).

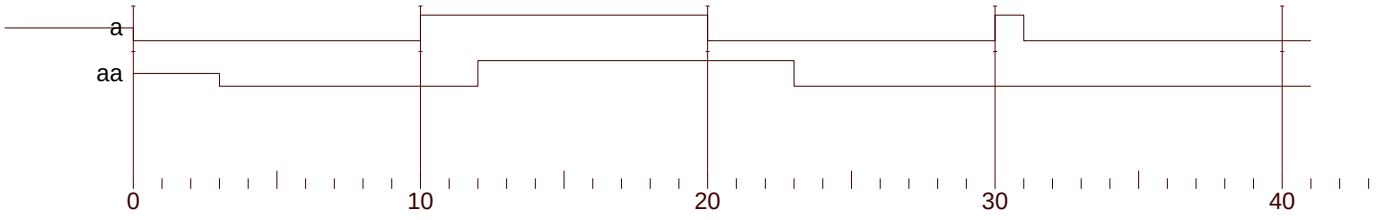
The solution appears below.



(*e*) Show the changes (values and times) to `aa` in the module below. (10 pts)

```
module d();
   reg a;
   wire aa;

   and #(2,3) (aa,a,1);

   initial begin
      a = 0;
      # 10;
      a = 1;
      # 10;
      a = 0;
      # 10;
      a = 1;
      # 1;
      a = 0;
      # 10;
   end
endmodule // d
```

Solution:



(*f*) Complete module `after` so that it does the same thing as `before`. All procedural code in module `after` must go in the one initial process. The solution must use `fork` and `join`. Structural code **cannot** be added. (10 pts)

```
module before(asum,bsum,out,a,ainp,b,binp,c);
   output asum, bsum, out;
   input  a, ainp, b, binp, c;

   reg [9:0]  asum, bsum, out;
   wire [9:0] ainp, binp;
   wire       a,b,c;

   always @( a ) asum = asum + ainp;

   always @( b ) bsum = bsum + binp;

   always @( posedge c ) out = asum + bsum;

endmodule

module after(asum,bsum,out,a,ainp,b,binp,c);
   output asum, bsum, out;
   input  a, ainp, b, binp, c;

   reg [9:0]  asum, bsum, out;
   wire [9:0] ainp, binp;
   wire       a,b,c;

   // ALL code must go in the initial process below.
   initial begin


   // Solution:
   fork
      forever @( a ) asum = asum + ainp;
      forever @( b ) bsum = bsum + binp;
      forever @( posedge c ) out = asum + bsum;
   join



   end // initial

endmodule
```

Name Solution

Digital Design Using Verilog
EE 4702-1
Final Examination

8 May 2000, 7:30–9:30 CDT

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

Alias full case

Good Luck!

Problem 1: The modules below are supposed to describe combinational logic that rearranges bits. The output of module `rearrange`, below, is a rearranged version of its input `a`; input `op` determines how the bits are rearranged. Module `rerearrange` uses two instances of `rearrange` to reverse and then left shift its inputs. Unfortunately, the modules are not quite ready for tape out because both contain errors.

Find and fix the following kinds of errors. (Points may be deducted if correct Verilog is identified as having errors.) (20 pts)*Note: The original exam specified one Modelsim compile error. However Modelsim compiles the code without an error or warning. What was thought to be a compile error is a load error. The number of errors is still five.*

- Two load errors or warnings. (Modelsim will compile it but will issue a warning or error message when loading it.)
- Three errors that result in incorrect output. The code will simulate but the output, if any, will be incorrect.

Lines with the comment `// Okay` do not have errors. None of the errors are typographical or are due to syntactic minutiæ such as missing semicolons.

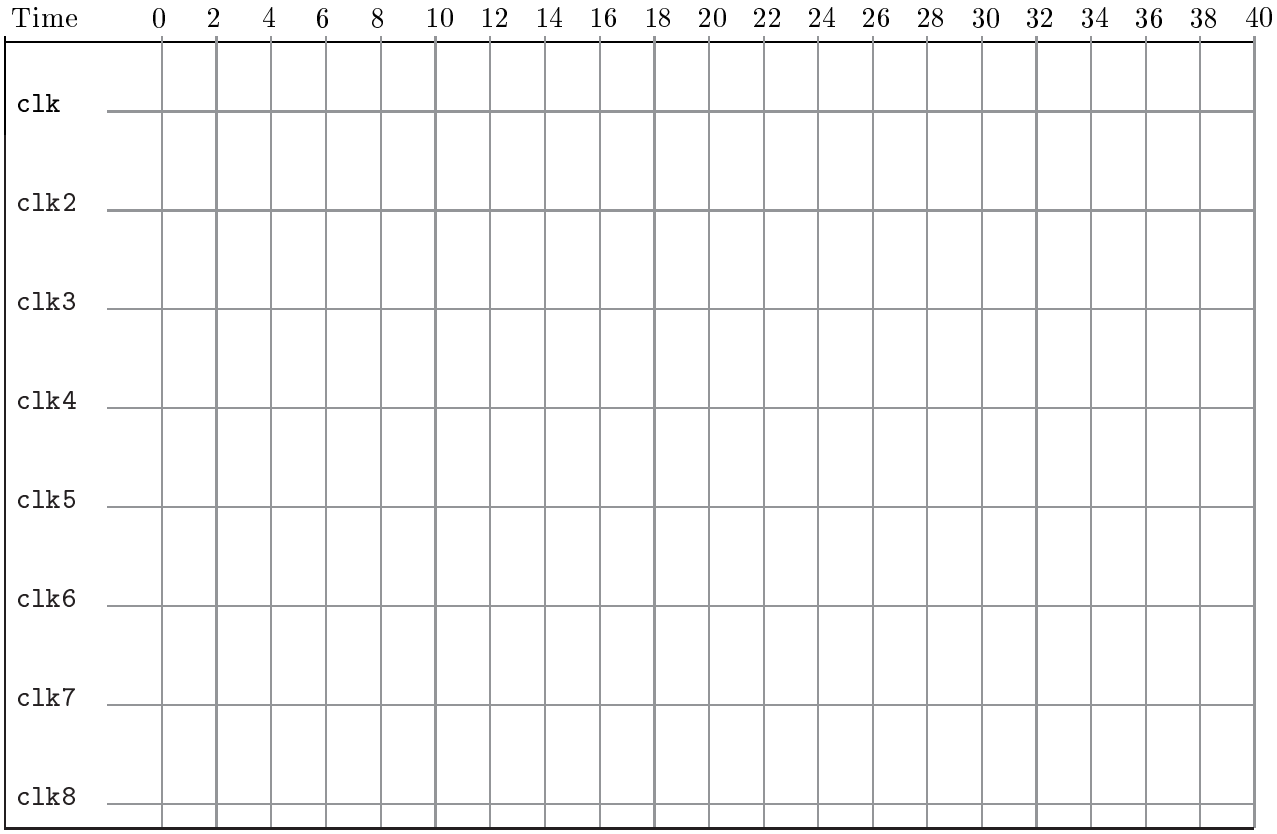
```
module rerearrange(y,a);
   input a;              output y;
   wire [7:0] a;         reg [7:0]  y;      wire [0:7] temp;

   wire       operation;
   assign     operation = e1.op_reverse;
   rearrange e1(temp,a,operation);

   assign     operation = e1.op_left_shift;
   rearrange e2(y,temp,operation);
endmodule

module rearrange(x,a,op);
   input      a, op;      output     x;
   wire [7:0] a;          wire [1:0] op;
   reg [7:0]  x;          reg [2:0]  ptr, ptr_plus_one;

   parameter  op_reverse     = 0;  // Reverse order of bits.            // Okay
   parameter  op_identity    = 1;  // No change.                        // Okay
   parameter  op_left_shift  = 2;  // Circular (end-around) left shift. // Okay
   parameter  op_right_shift = 3;  // Circular (end-around) right shift.// Okay

   always @( a ) for(ptr=0; ptr<8; ptr=ptr+1) begin
        ptr_plus_one = ptr + 1;                                          // Okay
        case( op )
          op_reverse:     x[ptr]          = a[7-ptr];                    // Okay
          op_identity:    x[ptr]          = a[ptr];                      // Okay
          op_right_shift: x[ptr]          = a[ptr_plus_one];             // Okay
          op_left_shift:  x[ptr_plus_one] = a[ptr];                      // Okay
        endcase
     end
endmodule
```

Solution:

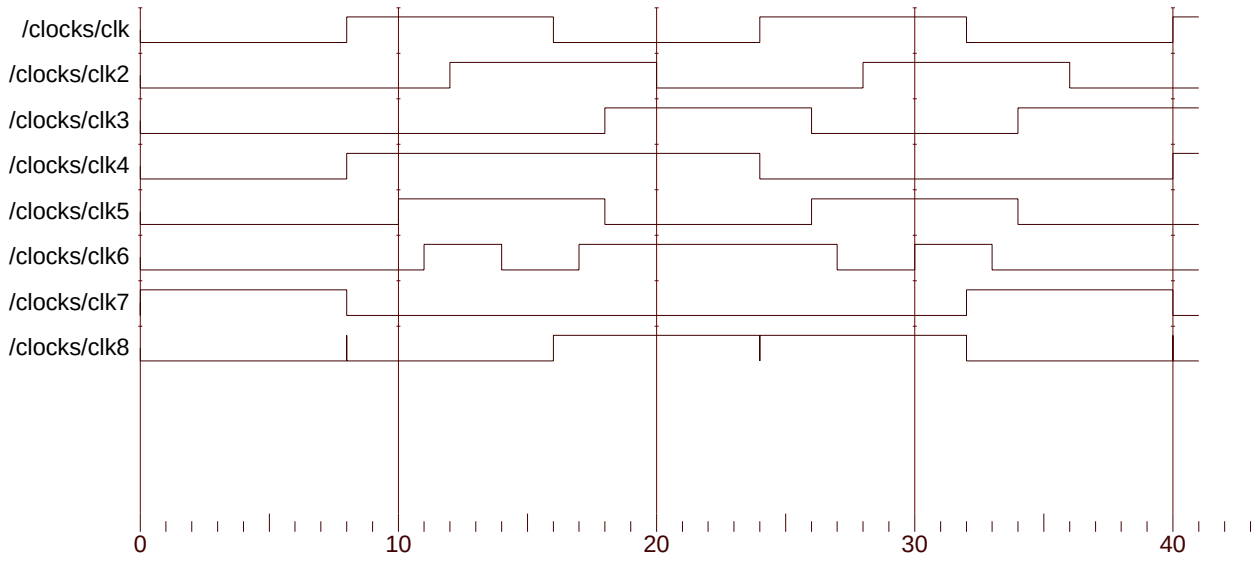
```
module rerearrange(y,a);
   input a; output y; wire [7:0] a;
   // Registers cannot connect to module output ports.
   // reg [7:0]  y;
   wire [7:0]  y;  // FIXED
   wire [0:7] temp;  // Not an error: Order of bits doesn't matter.

// B: Wire "operation" wrong size.
//   wire       operation;
   wire       [1:0] operation;  // FIXED
   assign     operation = e1.op_reverse;

   rearrange e1(temp,a,operation);

   // Second wire needed for input to second module. (This is not procedural
   // code so ordering of assignments and instantiations is meaningless.)
   //   assign     operation = e1.op_left_shift;
   wire [1:0] operation2 = e1.op_left_shift;   // FIXED
   rearrange e2(y,temp,operation2);
endmodule

module rearrange(x,a,op);
   input      a, op;
   output     x;
   wire [7:0] a;
   wire [1:0] op;
   reg [7:0]  x;
   // C: Loop checks if ptr<8, so need more than 3 bits.  Note: ptr_plus_one
   // must be 3 bits since code depends on values wrapping around.
   // reg [2:0]  ptr, ptr_plus_one;
   reg [3:0]  ptr;  // FIXED.
   reg [2:0]  ptr_plus_one;

   parameter  op_reverse     = 0;  // Reverse order of bits.            // Okay
   parameter  op_identity    = 1;  // No change.                        // Okay
   parameter  op_left_shift  = 2;  // Circular (end-around) left shift. // Okay
   parameter  op_right_shift = 3;  // Circular (end-around) right shift.// Okay

   // C: Need to include op in the event list.
   // always @( a ) for(ptr=0; ptr<8; ptr=ptr+1) begin
   always @( a or op ) for(ptr=0; ptr<8; ptr=ptr+1) begin
        ptr_plus_one = ptr + 1;                                          // Okay
        case( op )
          op_reverse:     x[ptr]          = a[7-ptr];                    // Okay
          op_identity:    x[ptr]          = a[ptr];                      // Okay
          op_right_shift: x[ptr]          = a[ptr_plus_one];             // Okay
          op_left_shift:  x[ptr_plus_one] = a[ptr];                      // Okay
        endcase
     end
endmodule // rearrange
```

Problem 2: Using the grid show the register values for the first 40 time units of execution of the module below. (20 pts)

```
module clocks();
   reg clk, clk2, clk3, clk4, clk5, clk6, clk7, clk8;
   initial begin
      clk  = 0; clk2 = 0; clk3 = 0; clk4 = 0;
      clk5 = 0; clk6 = 0; clk7 = 0; clk8 = 0;
   end

   always #8 clk = ~clk;
   always @( clk ) #4 clk2 = ~clk2;
   always @( clk ) clk3 <= #10 clk;
   always @( posedge clk ) clk4 = ~clk4;
   always #2 forever #8 clk5 = ~clk5;
   always wait( clk ) #3 clk6 = ~clk6;
   always @( clk | clk4 ) clk7 = ~clk7;
   always @( clk or clk4 ) clk8 = ~clk8;

endmodule
```



Solution:

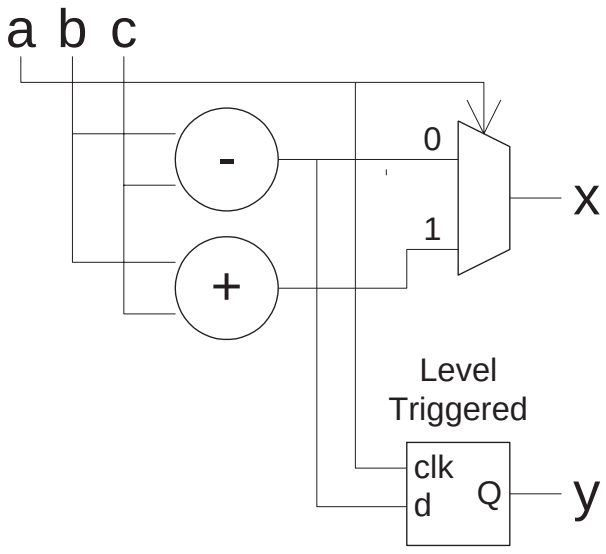

/clocks/clk
/clocks/clk2
/clocks/clk3
/clocks/clk4
/clocks/clk5
/clocks/clk6
/clocks/clk7
/clocks/clk8
0
10
20
30
40

Problem 3: Draw a schematic of the hardware Leonardo will synthesize for the following Verilog code examples. These should approximate the RTL schematic, showing the hardware before optimization and technology mapping. If flip flops are used, indicate if they are level triggered or edge triggered. Otherwise, don't worry about using the precisely correct gate or symbol, as long as it's functionally correct.

(*a*) Show an approximate RTL schematic for the module below. What form is the description in? *Hint: think about what form the code is in.* (6 pts)

```
module mod_a(x,y,a,b,c);
   input a,b,c;
   output x,y;
   wire [7:0] b, c;
   reg [8:0]  x, y;

   always @( a or b or c ) begin
      if( a ) begin
         x = b + c;
         y = b - c;
      end else begin
         x = b - c;
      end
   end
endmodule
```

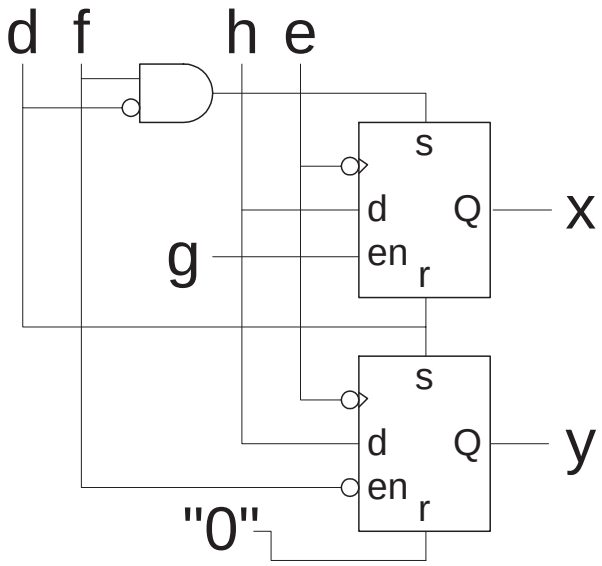
Form 1: combinational logic, level triggered flip-flops.



Problem 3, continued: (*b*) Show an approximate RTL schematic for the module below. What form is the description in? *Hint: think about what form the code is in.* (6 pts)

```
module mod_b(x,y,d,e,f,g,h);
   input d,e,f,g,h;
   output x,y;
   reg    x,y;

   always @( posedge d or negedge e or posedge f )
     if( d ) begin
        x = 0;
        y = 1;
     end else if ( f ) begin
        x = 1;
     end else begin
        if( g ) x = h;
        y = h;
     end

endmodule
```

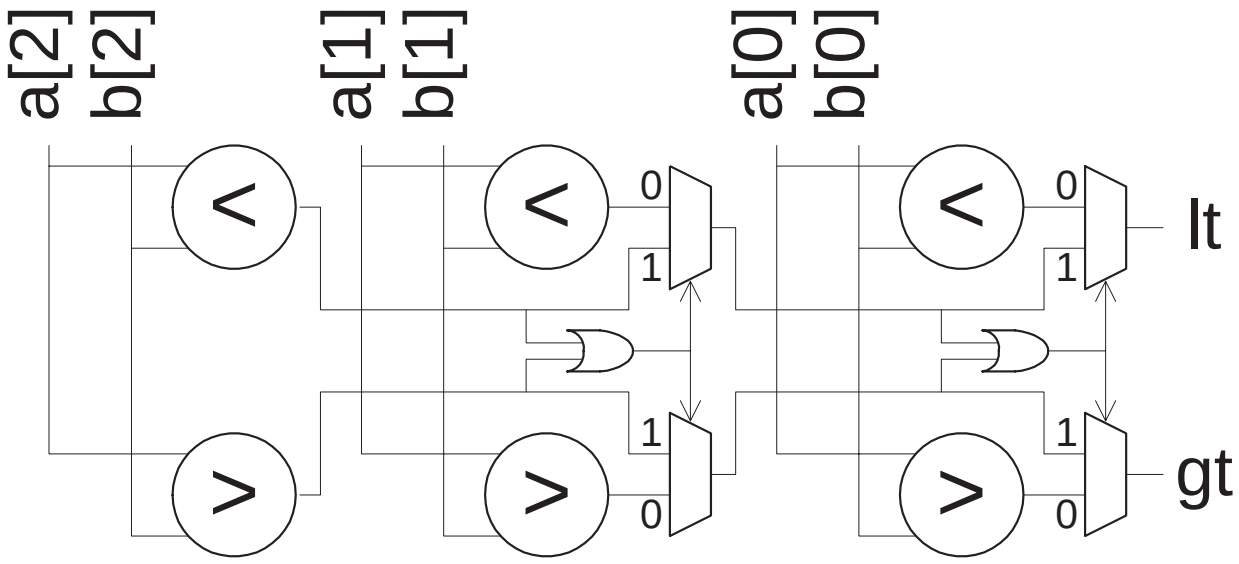
Form 2: Edge triggered flip flops.



Problem 3, continued: (*c*) Show an approximate RTL schematic for the module below. Assume that the synthesis program will not infer that this module performs magnitude comparison. Use symbols $\boxed{<}$ and $\boxed{>}$ for bit comparison. (8 pts)

```
module compare(gt, lt, a, b);
   input a, b;
   output gt, lt;
   wire [2:0] a, b;
   reg        gt, lt;
   integer    i;

   always @( a or b ) begin
      gt = 0; lt = 0;
      for(i=2; i>=0; i=i-1) if( !gt && !lt ) begin
         if( a[i] < b[i] ) lt = 1;
         if( a[i] > b[i] ) gt = 1;
      end
   end

endmodule
```

Form 1. The logic is purely combinational.



Problem 4: The incomplete code below, `compare_ism`, is for a magnitude comparison module (similar to the one in the previous problem, except it's sequential).

When input `start` is 1, output `valid` goes to zero and the module computes `lt` and `gt`. When `lt` and `gt` are set to their proper values `valid` is set to one. The module is to compare one bit position per cycle of input `clk`. Output `valid` should go to one as soon as possible.

Complete the module so that it is in the form of an implicit state machine, synthesizable by Leonardo. The solution can be based on the combinational module `compare`, below. Don't forget signals `start` and `valid`. (20 pts) *Hint: The solution is very similar to the combinational module.* For partial credit ignore synthesizability but follow other specifications.

```
module compare(gt, lt, a, b);    // Synthesizable combinational implementation.
   input a, b;              output gt, lt;
   wire [31:0] a, b;
   reg        gt, lt;       integer    i;

   always @( a or b ) begin
      gt = 0; lt = 0;
      for(i=31; i>=0; i=i-1) if( !gt && !lt ) begin
         if( a[i] < b[i] ) lt = 1;
         if( a[i] > b[i] ) gt = 1;
      end
   end
endmodule

// Implicit state machine implementation.
module compare_ism(gt, lt, valid, a, b, start, clk);
   input a, b, start, clk;               output gt, lt, valid;
   wire [31:0] a, b;                     reg    gt, lt, valid;
   wire        start, clk;               integer i;



   // Solution
   always @( posedge clk ) if( start ) begin
      gt = 0; lt = 0; valid = 0;
      for(i=31; i>=0 && !lt && !gt; i=i-1) @( posedge clk ) begin


         if( a[i] < b[i] ) lt = 1;  // Part of solution.
         if( a[i] > b[i] ) gt = 1;



      if( a[i] > b[i] ) gt = 1;
      end
      valid = 1;
   end


endmodule
```

Problem 5: Answer each question below.

(*a*) Complete the module below so that it will stop simulation (using the system task $stop) if there is no change in signal `heartbeat` for 1000 simulator time units. There might be many changes in `heartbeat`, but the first time `heartbeat` remains unchanged for 1000 simulator time units simulation should be stopped. *Hint: use a fork. Also, the answer is short.* (5 pts)

```
module watchdog(heartbeat);
   input heartbeat;
   wire heartbeat;


   // Solution
   always
     fork:F
       @( heartbeat ) disable F;
       # 1000 $stop;
     join




endmodule // watchdog
```

(*b*) What is a critical path? At what point in the design flow can one first find out about critical paths? (5 pts)

A critical path is the longest path between registers; it determines the clock frequency. If a system is not clocked, it may be the longest path from inputs to outputs.

One finds out about critical paths after synthesis (technology mapping and optimization). This critical path information does not include wire lengths, so a refined estimate is obtained after place and route.

(*c*) Provide an example case statement in which the directive `exemplar case_parallel` is needed. What is its effect? (5 pts)

```
// Possible values for op: 100, 010, 001

wire [2:0] op;

// Needed because the synthesizer doesn't know that if the middle bit
// is 1 the leftmost bit must be zero.  (It can't according to the
// comment, which IS PART OF THE SOLUTION.)

// exemplar parallel_case
casez(op)
  3'b1??: a = 1;
  3'b?1?: b = 1;
  3'b??1: c = 1;
endcase // casez(op)
```

(*d*) The module below is supposed to zero the middle 3 bits of its input. It's rejected by the compiler (the "b=" line), identify and fix the problem. (5 pts)

The concatenation operator can only operate on constants that are signed, so instead of `0` use `3'b0`.

```
module whatswrong(a,b);
   input a;       output b;
   wire [8:0] a;  wire [8:0] b;

   assign b = {a[8:6],0,a[2:0]};

endmodule
```

/// Code from solution to LSU EE 4702-1 Spring 2000 Final Exam

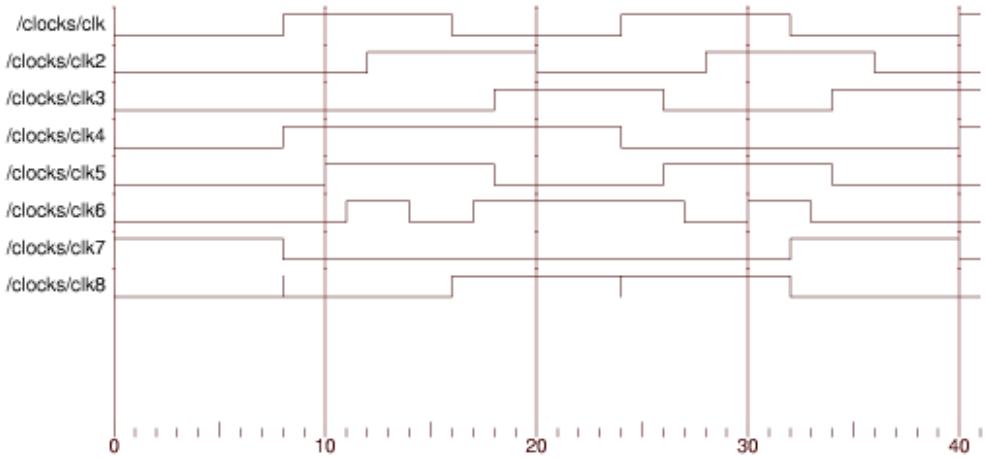
```
 // Exam:      http://www.ee.lsu.edu/v/2000/fe.pdf
 // Solution:  http://www.ee.lsu.edu/v/2000/fe_sol.pdf
```

///
/// Problem 1
///

```
`define FIXED_CODE
`ifdef ORIGINAL_CODE
module rerearrange(y,a);
   input a;
   output y;
   wire [7:0] a;
   reg [7:0]  y;
   wire [0:7] temp;

   wire       operation;
   assign     operation = e1.op_reverse;
   rearrange e1(temp,a,operation);

   assign     operation = e1.op_left_shift;
   rearrange e2(y,temp,operation);
endmodule

module rearrange(x,a,op);
   input      a, op;
   output     x;
   wire [7:0] a;
   wire [1:0] op;
   reg [7:0]  x;
   reg [2:0]  ptr, ptr_plus_one;

   parameter  op_reverse     = 0;  // Reverse order of bits.            // Okay
   parameter  op_identity    = 1;  // No change.                        // Okay
   parameter  op_left_shift  = 2;  // Circular (end-around) left shift. // Okay
   parameter  op_right_shift = 3;  // Circular (end-around) right shift.// Okay

   always @( a ) for(ptr=0; ptr<8; ptr=ptr+1) begin
        ptr_plus_one = ptr + 1;                                         // Okay
        case( op )
          op_reverse:     x[ptr]          = a[7-ptr];                   // Okay
          op_identity:    x[ptr]          = a[ptr];                     // Okay
          op_right_shift: x[ptr]          = a[ptr_plus_one];            // Okay
          op_left_shift:  x[ptr_plus_one] = a[ptr];                     // Okay
        endcase
     end
endmodule // rearrange


//  # Loading work.rerearrange
//  # Loading work.rearrange
//  # WARNING: fe_sol.v(8): [PCDPC] - Port size does not match connection size (3rd connection).
//  #        Region: /rerearrange/e1
//  # ERROR: fe_sol.v(11): Illegal output port connection (1st connection).
//  #       Region: /rerearrange/e2
//  # WARNING: fe_sol.v(11): [PCDPC] - Port size does not match connection size (3rd connection).
//  #        Region: /rerearrange/e2
//  # Error loading design
`endif // ifdef ORIGINAL_CODE

`ifdef FIXED_CODE
```

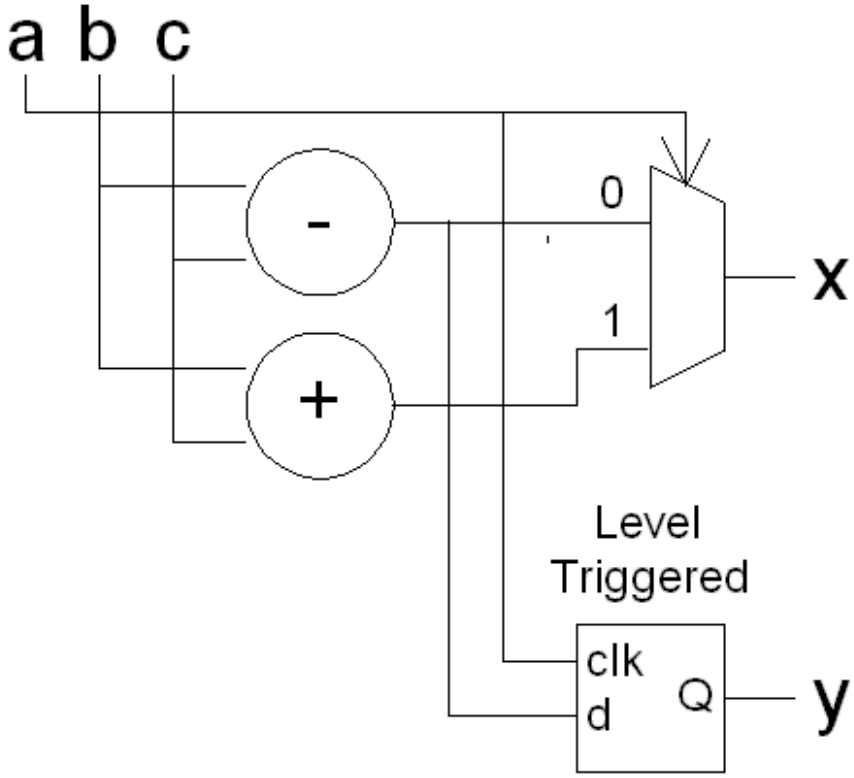
```
module rerearrange(y,a);
   input a;
   output y;
   wire [7:0] a;
   // Registers cannot connect to module output ports.
   // reg [7:0]  y;
   wire [7:0]  y;  // FIXED
   wire [0:7] temp;

// B: Wire "operation" wrong size.
//   wire       operation;
   wire       [1:0] operation;  // FIXED
   assign     operation = e1.op_reverse;

   rearrange e1(temp,a,operation);

   // Second wire needed for input to second module. (This is not procedural
   // code so ordering of assignments and instantiations is meaningless.)
   //   assign     operation = e1.op_left_shift;
   wire [1:0] operation2 = e1.op_left_shift;   // FIXED
   rearrange e2(y,temp,operation2);
endmodule

module rearrange(x,a,op);
   input      a, op;
   output     x;
   wire [7:0] a;
   wire [1:0] op;
   reg [7:0]  x;
   // C: Loop checks if ptr<8, so need more than 3 bits.  Note: ptr_plus_one
   // must be 3 bits since code depends on values wrapping around.
   // reg [2:0]  ptr, ptr_plus_one;
   reg [3:0]  ptr;  // FIXED.
   reg [2:0]  ptr_plus_one;

   parameter  op_reverse     = 0;  // Reverse order of bits.            // Okay
   parameter  op_identity    = 1;  // No change.                        // Okay
   parameter  op_left_shift  = 2;  // Circular (end-around) left shift. // Okay
   parameter  op_right_shift = 3;  // Circular (end-around) right shift.// Okay

   // C: Need to include op in the event list.
   // always @( a ) for(ptr=0; ptr<8; ptr=ptr+1) begin
   always @( a or op ) for(ptr=0; ptr<8; ptr=ptr+1) begin   // FIXED
        ptr_plus_one = ptr + 1;                                          // Okay
        case( op )
          op_reverse:     x[ptr]          = a[7-ptr];                    // Okay
          op_identity:    x[ptr]          = a[ptr];                      // Okay
          op_right_shift: x[ptr]          = a[ptr_plus_one];             // Okay
          op_left_shift:  x[ptr_plus_one] = a[ptr];                      // Okay
        endcase
     end
endmodule // rearrange
`endif // ifdef FIXED_CODE

module test_rr();

   reg [7:0] orig;
   wire [7:0] arranged;

   rearrange rr1(arranged,orig);

   initial begin
      orig = 8'b11110000;
      #1;
      orig = 8'b00001111;
      #1;
```

```
   end

endmodule // test_rr

///
/// Problem 2  (Unmodified code from exam.)
///


module clocks();
   reg clk, clk2, clk3, clk4, clk5, clk6, clk7, clk8;
   initial begin
      clk  = 0; clk2 = 0; clk3 = 0; clk4 = 0;
      clk5 = 0; clk6 = 0; clk7 = 0; clk8 = 0;
   end

   always #8 clk = ~clk;
   always @( clk ) #4 clk2 = ~clk2;
   always @( clk ) clk3 <= #10 clk;
   always @( posedge clk ) clk4 = ~clk4;
   always #2 forever #8 clk5 = ~clk5;
   always wait( clk ) #3 clk6 = ~clk6;
   always @( clk | clk4 ) clk7 = ~clk7;
   always @( clk or clk4 ) clk8 = ~clk8;

   initial #41 $stop;

endmodule

// Solution:
```



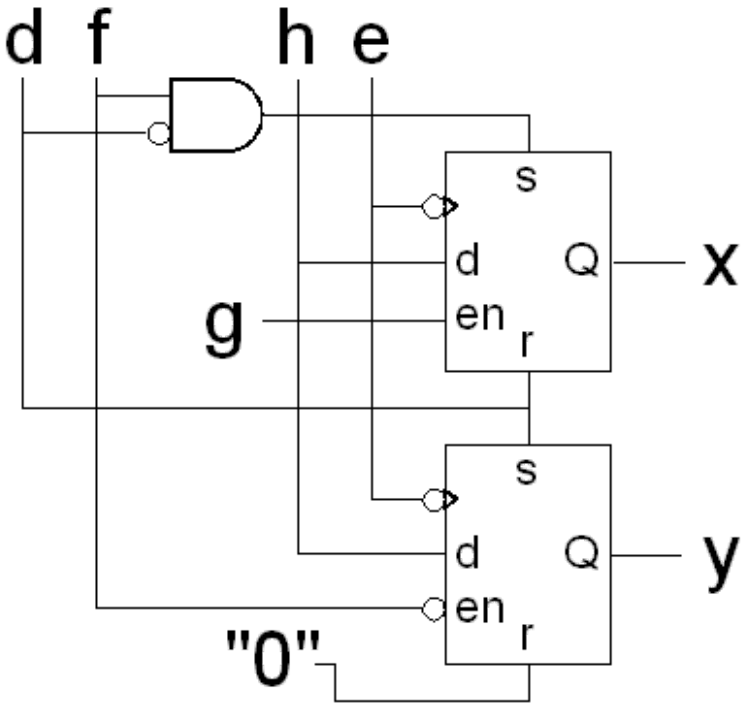

```
///
/// Problem 3  (Unmodified code from exam.)
///

module mod_a(x,y,a,b,c);
   input a,b,c;
   output x,y;
   wire [7:0] b, c;
   reg [8:0]  x, y;

   always @( a or b or c ) begin
      if( a ) begin
         x = b + c;
         y = b - c;
      end else begin
         x = b - c;
      end
   end
```

```
endmodule

// Solution:
```



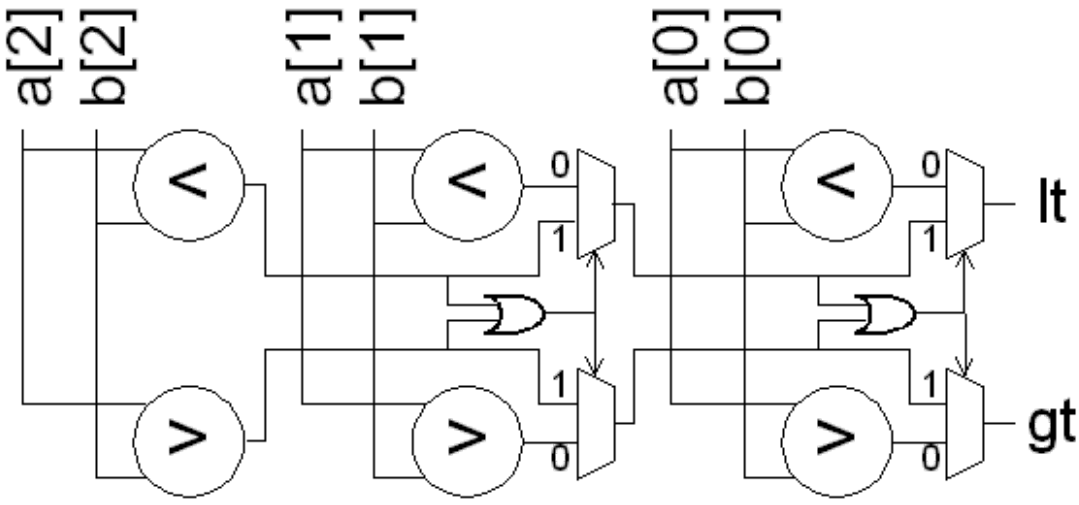

```
module mod_b(x,y,d,e,f,g,h);
   input d,e,f,g,h;
   output x,y;
   reg    x,y;

   always @( posedge d or negedge e or posedge f )
     if( d ) begin
        x = 0;
        y = 1;
     end else if ( f ) begin
        x = 1;
     end else begin
        if( g ) x = h;
        y = h;
     end

endmodule
```

// Solution:




```
module compare(gt, lt, a, b);
   input a, b;
   output gt, lt;
   wire [2:0] a, b;
   reg        gt, lt;
   integer    i;

   always @( a or b ) begin
      gt = 0; lt = 0;
      for(i=2; i>=0; i=i-1) if( !gt && !lt ) begin
         if( a[i] < b[i] ) lt = 1;
         if( a[i] > b[i] ) gt = 1;
      end
   end

endmodule // compare
```

// Solution:



///

/// # Problem 4
///

```
// Unmodified from exam.
module compare_comb(gt, lt, a, b);
   input a, b;
   output gt, lt;
   wire [3:0] a, b;
   reg        gt, lt;
   integer    i;

   always @( a or b ) begin
      gt = 0; lt = 0;
      for(i=3; i>=0; i=i-1) if( !gt && !lt ) begin
         if( a[i] < b[i] ) lt = 1;
         if( a[i] > b[i] ) gt = 1;
      end
   end

endmodule

// Solution.
module compare_ism(gt, lt, valid, a, b, start, clk);
   input a, b, start, clk;
   output gt, lt, valid;
   wire [31:0] a, b;
   reg         gt, lt, valid;
   integer     i;

   always @( posedge clk ) if( start ) begin
      gt = 0; lt = 0; valid = 0;
      for(i=31; i>=0 && !lt && !gt; i=i-1) @( posedge clk ) begin
         if( a[i] < b[i] ) lt = 1;
         if( a[i] > b[i] ) gt = 1;
      end
      valid = 1;
   end

endmodule
```

///
/// # Problem 5
///

///
/// # Problem 5a
///

```
// Solution
module watchdog(heartbeat);
   input heartbeat;
   wire  heartbeat;

   always
     fork:F
       @( heartbeat ) disable F;
       # 1000 $stop;
     join

endmodule // watchdog
```

///

/// Problem 5d

///

```
// Solution.
module whatswrong(a,b);
   input a;       output b;
   wire [8:0] a;  wire [8:0] b;

  // assign b = {a[8:6],0,a[2:0]};
   assign b = {a[8:6],3'b0,a[2:0]};

endmodule
```