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`.

Staple This Side

Staple This Side

```
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
```

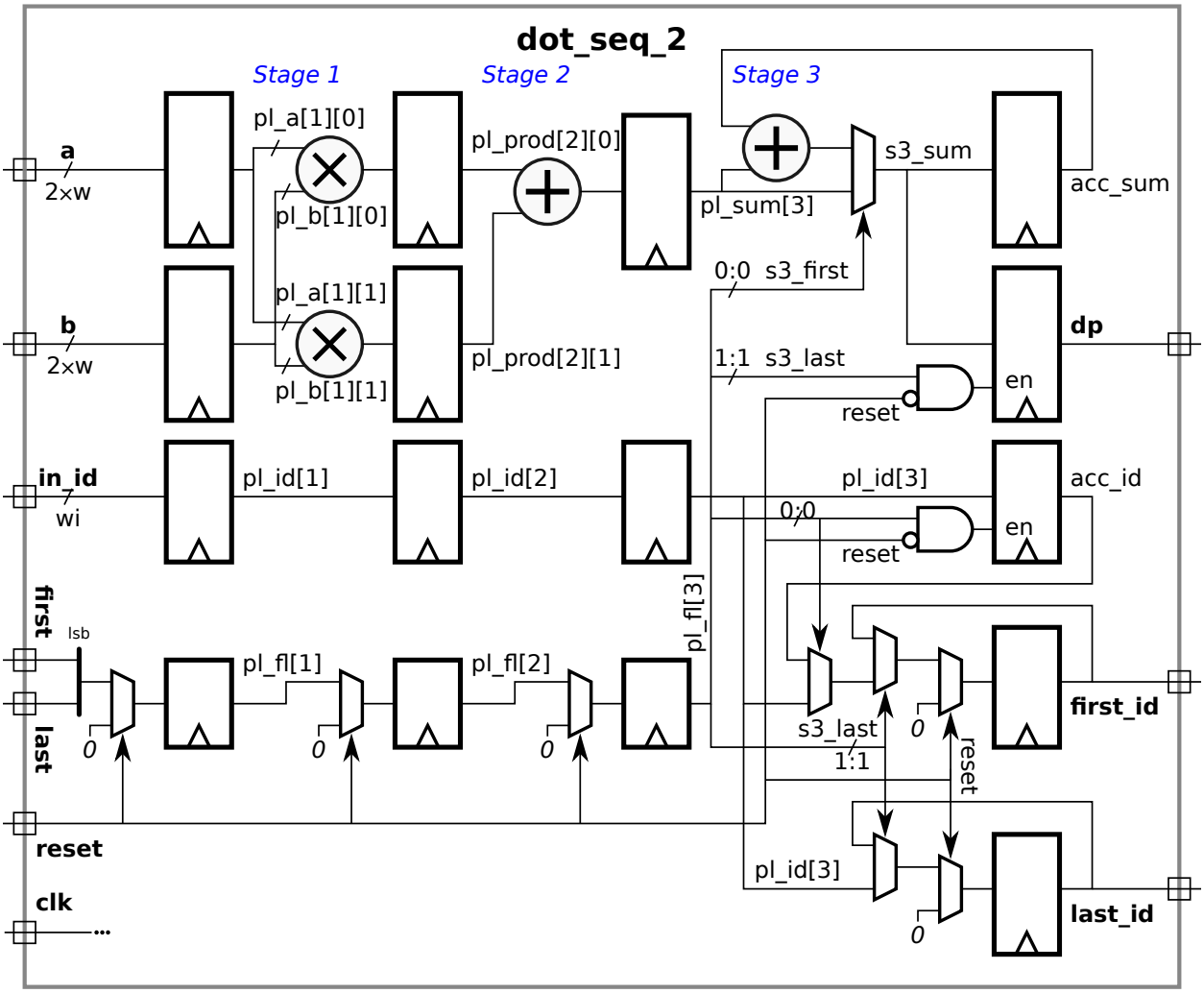

Staple This Side

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

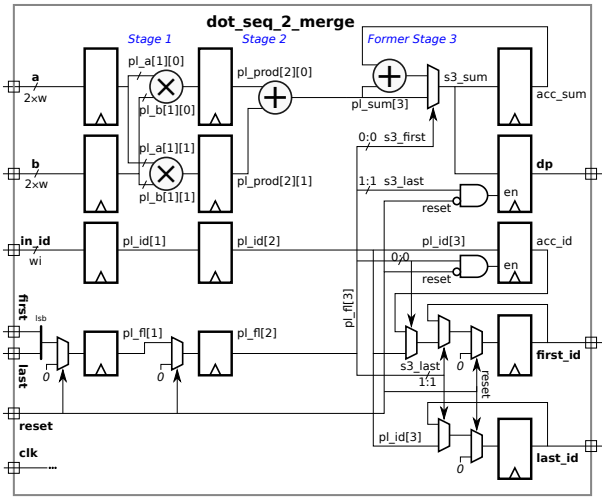
Staple This Side

☐ 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.

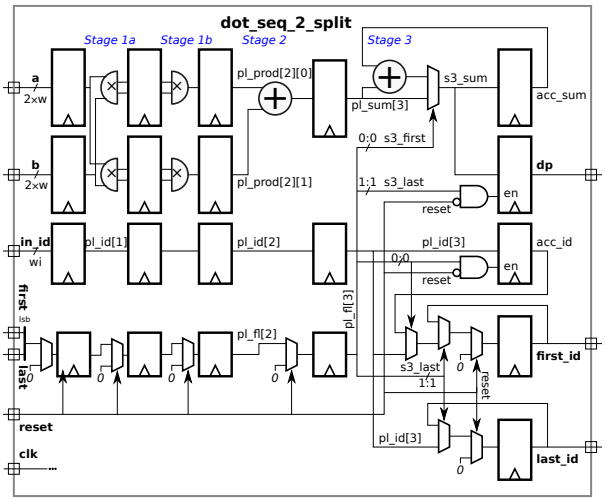
☐ What is the critical path length of `dot_seq_2_merge`?



☐ 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`?

Staple This Side

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.

Staple This Side

(*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.

Staple This Side

Staple This Side

```
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
```

Staple This Side