Name Solution _____ *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 The other one. _____

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`.

Solution appears on the facing page. In stage 1 the loop has been increased to four iterations. In stage 2 two sums are computed, not just one. Because the critical path must be limited to one arithmetic operation a new stage is needed to add together the sums computed in stage 2, that's stage 3 (which is new, the old stage 3 is now stage 4). In stage 4 hardware was added to compute the length of the vector.

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][4];             // Vector products.
   logic [w-1:0] pl_sum[3:4][2];              // Dot prod of 2-element segment.
   logic [1:0] pl_fl[1:4];                    // The first and last signals.
   logic [w-1:0] acc_sum;
   logic [wi-1:0] acc_dim; // SOLUTION: Use to keep track of vector length.

   always_ff @( posedge clk ) begin

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

      // Stage 1
      for ( int i=0; i<4; 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][0] <= pl_prod[2][0] + pl_prod[2][1];
      pl_sum[3][1] <= pl_prod[2][2] + pl_prod[2][3]; // SOLUTION
      pl_fl[3] <= reset ? 2'h0 : pl_fl[2];

      // Stage 3
      pl_sum[4][0] <= pl_sum[3][0] + pl_sum[3][1];   // SOLUTION
      pl_fl[4] <= reset ? 2'h0 : pl_fl[3];

      // Stage 4 -- (Was stage 3 in unsolved problem.)
      begin
         // Declare intermediate values in this block.
         automatic logic s3_first = pl_fl[4][0]; // For readability.
         automatic logic s3_last =  pl_fl[4][1]; // For readability.

         automatic logic [w-1:0] s3_sum = s3_first ? pl_sum[4][0] : pl_sum[4][0] + acc_sum;

         // SOLUTION - Compute new length of vector.
         automatic logic [w-1:0] s3_dim = s3_first ? 4 : 4 + acc_dim;

         acc_sum <= s3_sum;
         acc_dim <= s3_dim; // SOLUTION

         if ( !reset && s3_last ) begin
            dp <= s3_sum;
            dim <= acc_dim; // SOLUTION
         end
      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 $4w\,\mathrm{u_t}$.

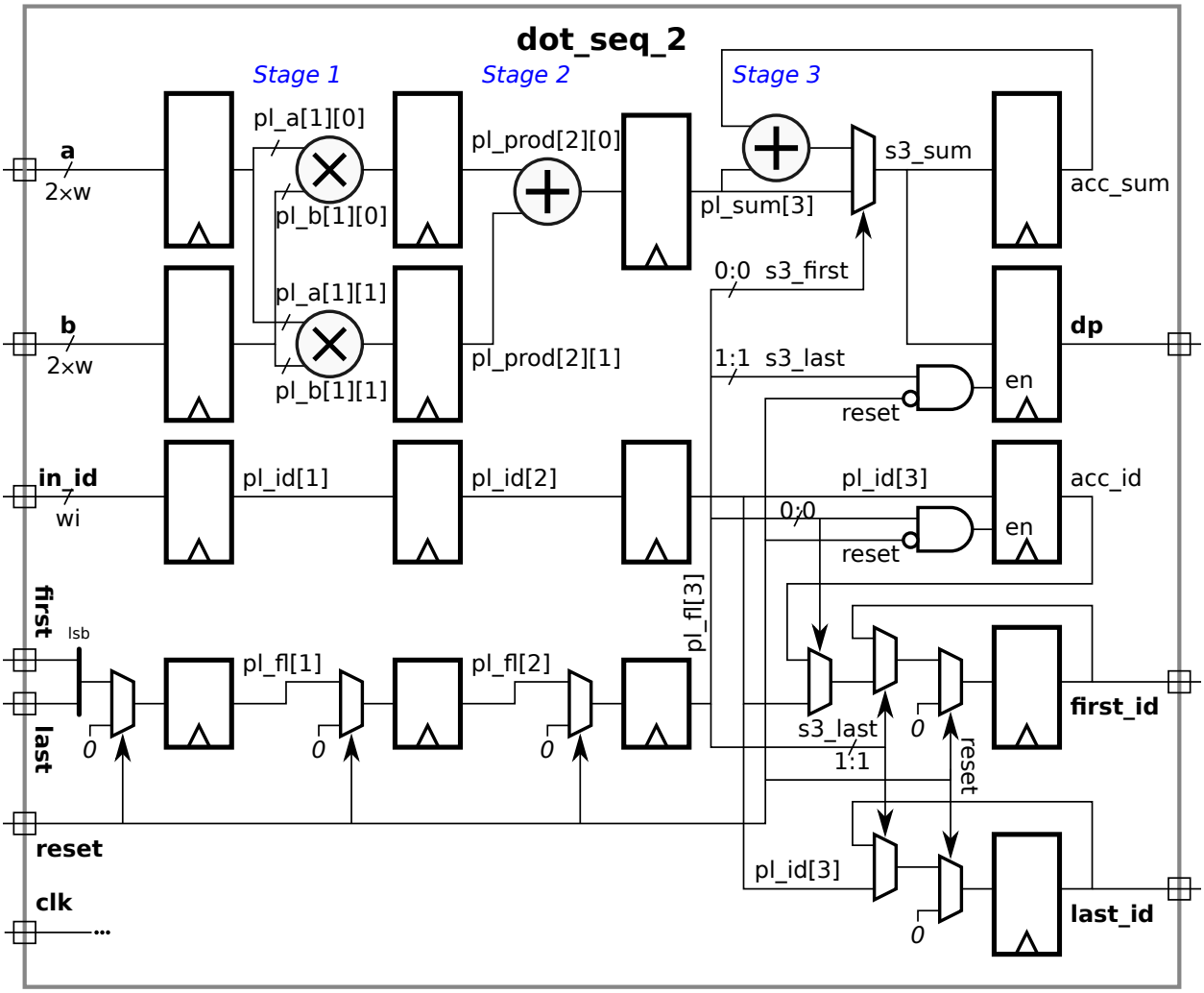
☐ Show the arrival time at each wire, especially at the capture points. ☐ Be sure to account for constant inputs.

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

The critical path is through the multiplier $t_{\mathrm{crit}} = 4w\,\mathrm{u_t}$.

☑ 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.

Assume that register setup time is small and can be ignored. (Otherwise it would add $6\,\mathrm{u_t}$ to the critical path length.) The clock frequency $\phi = \frac{1}{t_{\mathrm{crit}}} = \frac{1}{4w\,\mathrm{ns}} = \frac{1}{4w}\,\mathrm{GHz}$.



Staple This Side

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

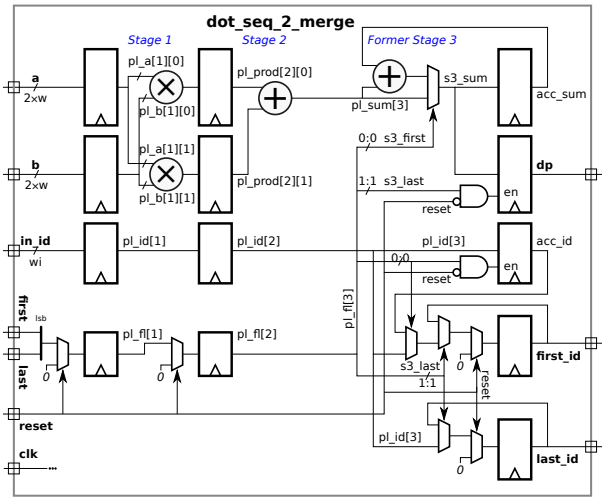
The unit of work is computing the dot product of 2-element vectors. The latency includes the four cycles it takes from the time the inputs are presented until the result is at the output. So the latency is $4t_{\text{crit}} = 16w\,\text{u}_\text{t}$.

Since a new 2-element vector can arrive each clock cycle the throughput is the clock frequency, $\phi = \frac{1}{4w\,\text{u}_\text{t}}$.

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

It takes $\frac{8}{2} = 4$ cycles for an 8-element vector to be read by the module. Assume that the latency here starts when the first 2 elements arrive and ends when the dot product is at the `dp` output. This takes $4 + 3$ cycles, the plus 3 is for the three extra cycles (compared to a 2-element vector). The latency is then $7t_{\text{crit}} = 28w\,\text{u}_\text{t}$. Since it takes four cycles for an 8-element vector to be read, the throughput for 8-element vectors is $\frac{\phi}{4} = \frac{1}{16w\,\text{u}_\text{t}}$.

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



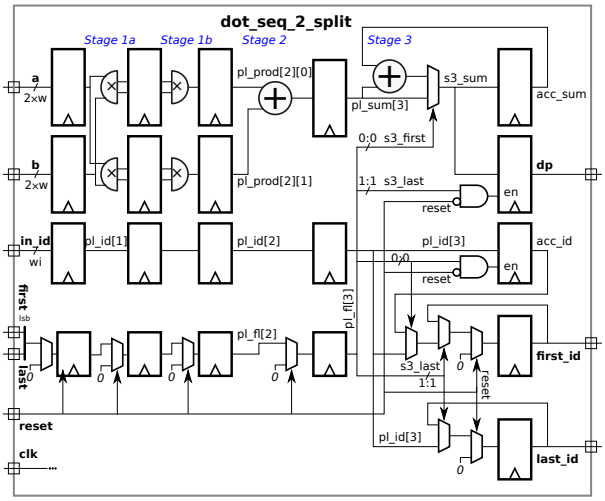
☑ What is the critical path length of `dot_seq_2_merge`?

If those adders are cascadable, then their combined delay is $[2w + 4]\,\text{u}_\text{t}$ which leaves the critical path passing through the multiplier. But, if they aren't cascadable the critical path is $[2 \times 2w + 2]\,\text{u}_\text{t} = [4w + 2]\,\text{u}_\text{t}$, slightly higher than before.

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

The number of cycles drops to three. The latency is $3t_{\text{crit}} = 3 \times 4w\,\text{u}_\text{t} = 12w\,\text{u}_\text{t}$. A big improvement. The throughput, however, is the same because a new pair of vectors can still arrive each cycle and the clock frequency hasn't changed.

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



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

Now the critical path is in the last stage, length $t_{\text{crit}} = [2w + 2]\,\text{u}_\text{t}$.

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

There are now five stages, but the clock frequency nearly doubled! The latency is $5t_{\text{crit}} = 5(2w+2)\,\text{u}_\text{t} = [10w+10]\,\text{u}_\text{t}$, better than the other modules! The throughput is also better, now $\frac{1}{[2w+2]\,\text{u}_\text{t}}$, almost double.

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`?

The recursion depth will be $\lg n$ levels. Each level adds a delay of t_2. So the total delay is $t_2 \lg n$. (The delay of the $n = 1$ level is zero, of course. But the $n = 1$ level isn't "counted" because $\lg 1 = 0$. So for $n = 2$, $\lg 2 = 1$.)

(*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.

Solution appears on the facing page. Notice that an `n==2` case was needed. Also notice that there are now two `min_2` modules at each level, their instance names are `m2` and `m3`.

(*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`?

The delay is $2t_2 \log_3 n$.

☑ 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.

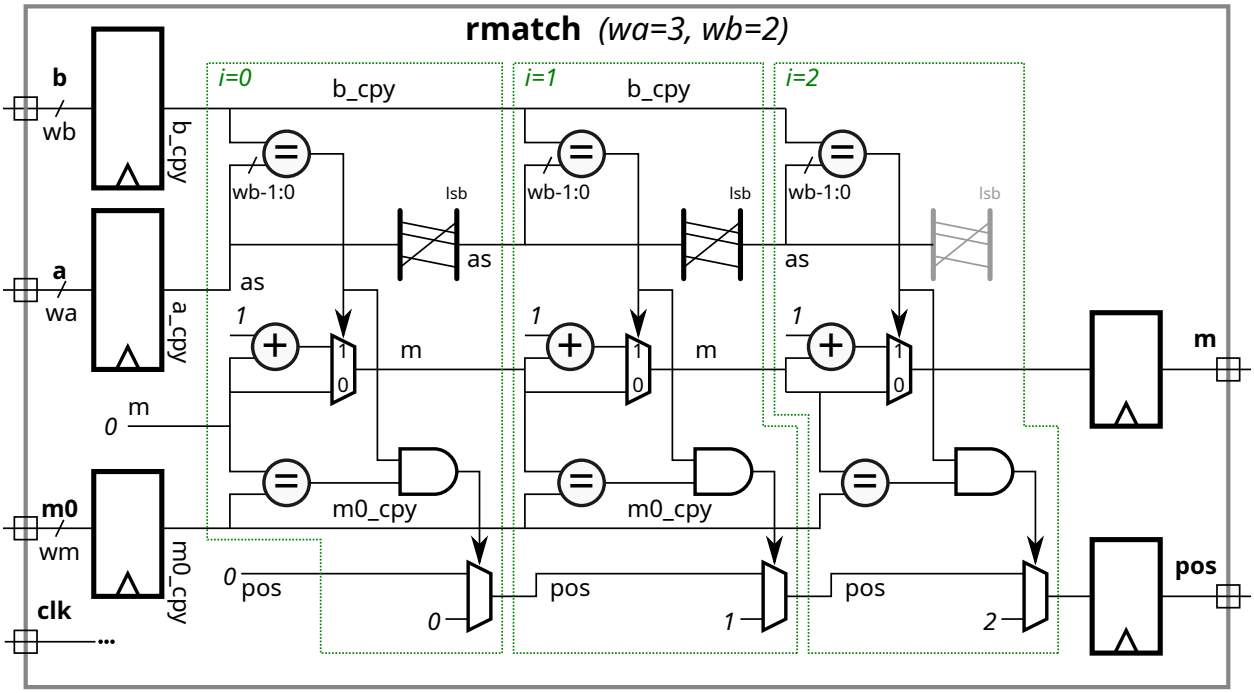
Delay is larger because the critical path now passes through two `min_2` modules at each level. The binary `min_t` has delay $t_2 \lg n = t_2 \frac{\ln n}{\ln 2}$. The trianary version has delay $2t_2 \frac{\ln n}{\ln 3}$. Since $\frac{2}{\ln 3} > \frac{1}{\ln 2}$ (or $1.820 > 1.443$) the delay of the trianary version is larger.

```
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 if ( n == 2 ) begin

      // SOLUTION -- Add an n == 2 case.

      min_2 #(w) m2( elt_min, elts[0], elts[1] );

   end else begin

      // SOLUTION -- Compute three sizes for recursive instantiations.
      localparam int n_hi = n / 3;
      localparam int n_mi = ( n - n_hi ) / 2;
      localparam int n_lo = n - n_hi - n_mi;

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

      min_t #(w,n_hi) mhi( elt_lo, elts[  n-1   :  n_lo + n_mi  ]);
      min_t #(w,n_mi) mmi( elt_mi, elts[ n_lo + n_mi -1  :  n_lo   ]); // SOLUTION
      min_t #(w,n_lo) mlo( elt_hi, elts[  n_lo-1  :  0   ]);

      min_2 #(w) m2( elt_lh, elt_lo, elt_hi );
      // SOLUTION -- Add a second min_2.
      min_2 #(w) m3( elt_min, elt_lh, elt_mi );

   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). *Note: In the original problem* `m` *was not initialized.*

```
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;
      m = 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 = 1   ☑ b = 2   ☑ c = 4   ☑ d = 1   ☑ e = 5   ☑ f = 6

    #1;

    // ☑ a = 2   ☑ b = 1   ☑ c = 4   ☑ d = 1   ☑ e = 13  ☑ f = 105

 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.

For the `x` version to work correctly, all live-in variables must be in the sensitivity list in the event control. If, say, `b`, were omitted then different hardware would be described, hardware using a latch, not combinational logic. With `always_comb` all of the live-in objects are automatically put in the sensitivity list.

Another feature of `always_comb` is that an object that is written in an `always_comb`, such as `y` here, cannot be written anywhere else. Since the intent is to describe combinational logic, we don't want it to be changed anywhere else, especially by accident.

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.

First error: The genvar loop iterates `n` times. In each iteration the output of `fp_madd` connects to `akk`. That means `akk` has `n` drivers. Since it was declared `uwire` it can only have one driver, and so there is a compile error (or should be). And remember in this class we never want more than one driver.

Second error: There can not be a delay control, `#1` in this case, in an `always_ff` (nor in an `always_comb` nor `always_latch`).

Third error: The `fp_madd` instantiation is in procedural code. There can never be instantiations in procedural code.

☑ 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."

No, it's not faster. The amount of time for output `dp` to change is determined by the clock frequency. Under any reasonable circumstance the clock frequency is fixed and won't change with the value of `use_fp`. The maximum possible clock frequency is determined by the synthesis program based on the critical path. The critical path is based on the hardware, but not on the values carried by wires. So if the critical path is through the floating-point hardware, that's what the clock frequency is limited by. It doesn't matter what `use_fp` is set to.

☑ 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.

No, of course not. The synthesis program will synthesize all the logic specified in the module. It would expect that `use_fp` can change and so it won't assume it's always one value. Things would be totally different if `use_fp` were a parameter.

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