Name _____ *Formatted For Two-Sided Printing*

Staple This Side

Digital Design using HDLs
LSU EE 4755
Final Examination

Friday, 12 December 2025 12:30-14:30 CST

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

Alias _____

Good Luck!

Staple This Side

Problem 1: [15 pts] Appearing below is an Verilog description of the rotate match module from the 2024 final exam. The facing page shows the inferred hardware for an instantiation at `wa=3` and `wb=2`.

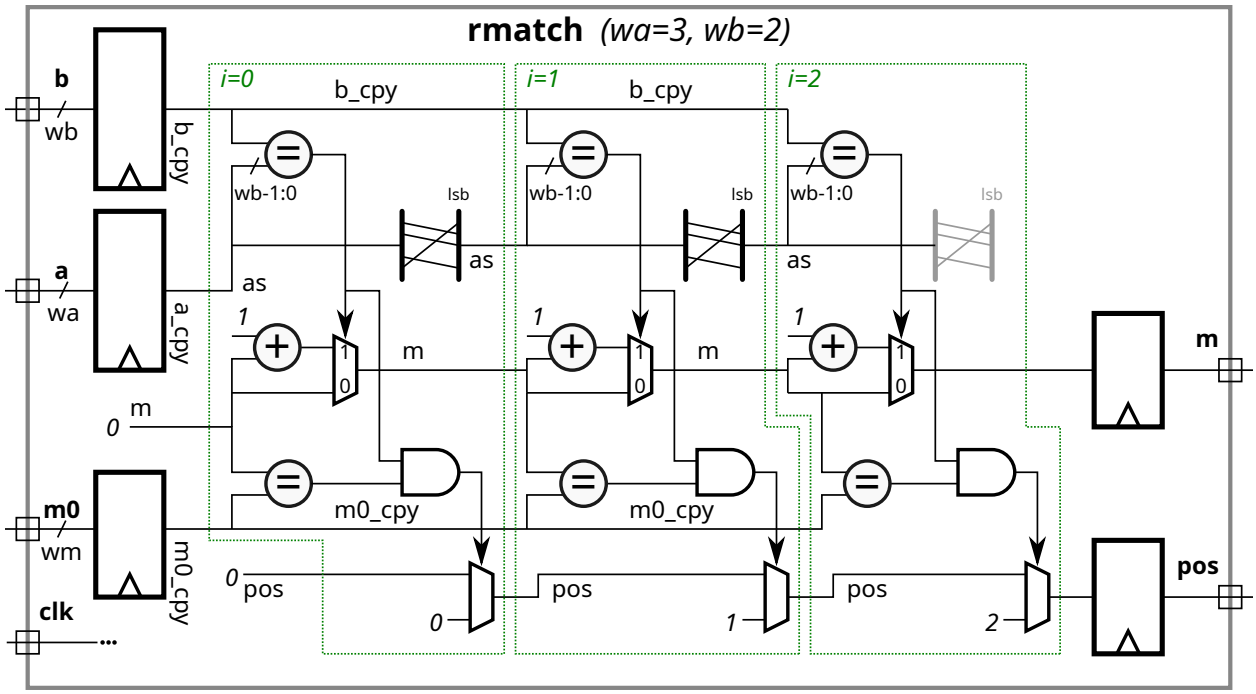
```
module rmatch #( int wa = 3, wb = 2, wm = $clog2(wa+1) )
   ( output logic [wm-1:0] m, pos,
     input uwire [wm-1:0] m0,   input uwire [wa-1:0] a,  input uwire [wb-1:0] b,
     input uwire clk );
   logic [wa-1:0] a_cpy, as;
   logic [wm-1:0] m0_cpy;
   logic [wb-1:0] b_cpy;
   always_ff @( posedge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

(*a*) Optimize the inferred hardware including the items below.

☐ Optimize for constants, there are plenty of them.

☐ If a unit (equality, adder, mux) as one-bit inputs replace the unit with gates (that implement the correct function).

☐ Show one of the upper comparison units as individual gates.


rmatch (wa=3, wb=2)
b
wb
a
wa
m0
wm
clk
b_cpy
a_cpy
m0_cpy
i=0
i=1
i=2
wb-1:0
lsb
as
m
0
1
pos
m0_cpy
m

Problem 2: [20 pts] On the facing page is the inferred hardware for the `rmatch` module, but this time the diagram does not show a particular size.

(*a*) Show the simple-model cost of the items requested below.

☐ Show costs requested below in terms of w_a, w_b, and w_m. ☐ Take care to **avoid** using a generic w or n in your final answers. When showing the cost ☐ **account for constant inputs**.

☐ Cost of the upper (`b_cpy`) comparison units.

☐ Cost of the bit rotation hardware.

☐ Cost of the adder.

☐ Cost of the `m` multiplexor.

☐ Cost of the lower (`m0_cpy`) comparison unit.

☐ Cost of the `pos` multiplexor.

(*b*) Show the simple-model delay of each component indicated below in terms of w_a, w_b, and w_m.

☐ Delay of the upper (`b_cpy`) comparison unit.

☐ Delay of the bit rotation hardware.

☐ Delay of the adder.

☐ Delay of the `m` multiplexor.

☐ Delay of the lower (`m0_cpy`) comparison unit.

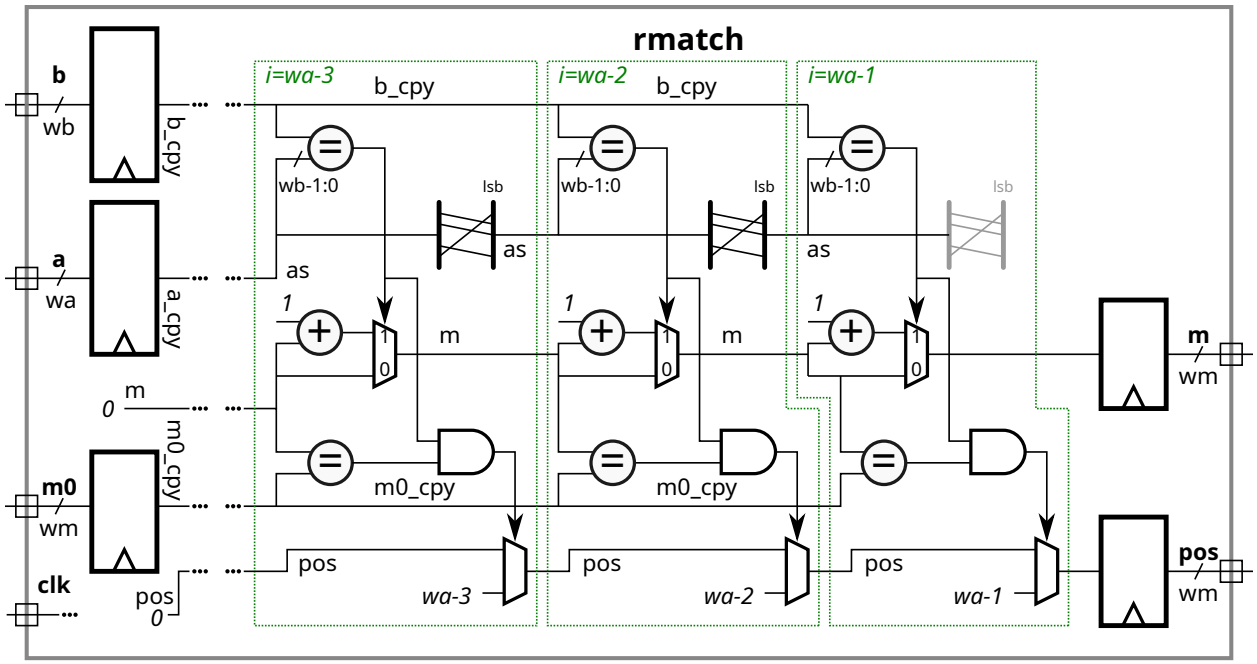
☐ Delay of the `pos` multiplexor.

(*c*) **Assume signals entering the `i=wa-3` section arrive at** $t = 0$.

☐ Show the arrival times, especially at the capture points ☐ in terms of w_a, w_b, and w_m.

☐ On the diagram show the critical path. No credit will be given for a list of numbers or components, instead show a path on the diagram.

Staple This Side


rmatch
i=wa-3
i=wa-2
i=wa-1
b
wb
a
wa
m0
wm
clk
b_cpy
a_cpy
m0_cpy
wb-1:0
lsb
as
m
1
0
pos
wa-3
wa-2
wa-1
m
wm
pos
wm

Problem 3: [20 pts] Appearing below is a simplified version of the solution to Homework 4, the sequential trend module. Show the hardware that will be inferred for this description.

```
module trend2_simple
  #( int wd = 20, wi = 10  )
   ( output logic [1:0] pr_trend,
     output logic [1:0] cr_trend,
     output logic [wi-1:0] cr_istart,
     input uwire [wd-1:0] samp,
     input uwire reset, clk );

   logic [wd-1:0] samp_prev;
   always_ff @( posedge clk ) samp_prev <= samp;

   uwire [1:0] trl2;
   ist_compare #(wd) c( trl2, samp_prev, samp );

   logic [wi-1:0] idx;
   uwire [wi-1:0] next_idx = reset ? 0 : idx + 1;
   always_ff @( posedge clk ) idx <= next_idx;

   logic [wi-1:0] lc_idx;
   always_ff @( posedge clk )
     if ( reset || trl2 ) lc_idx <= next_idx;

   uwire tr_new;
   magic_box mc( tr_new, cr_trend, trl2, reset, clk );
   //            <-- outputs --->, <--- inputs --->

   always_ff @( posedge clk )
     if ( reset ) begin
        cr_istart <= 0;
        pr_trend <= 0;
     end else if ( tr_new ) begin
        cr_istart <= lc_idx;
        pr_trend <= cr_trend;
     end
endmodule
```


Staple This Side

Staple This Side

- [ ] Show hardware that will be inferred for the module.
- [ ] Show all module input and output ports.
- [ ] Pay attention to whether a wire connects to a register output.
- [ ] Do not confuse elaboration-time items with synthesized hardware.

Staple This Side

Problem 4: [15 pts] Appearing below are variations on `best_rot_pipe` based on Homework 5.

```
module best_rot_pipe #( int wv = 17, wp = $clog2(wv+1) )
   ( output logic [wp-1:0] pos, dif,   input uwire [wv-1:0] val, key,   input uwire clk );
   logic [wv-1:0] pl_val_rot[wv:0], pl_key[wv:0];
   logic [wp-1:0] pl_pos[wv:1], pl_dif[wv:1], dif_here[wv-1:0];
   assign pos = pl_pos[wv], dif = pl_dif[wv];

   for ( genvar stage=0; stage<wv; stage++ )
     pop #(wv,wp) p( dif_here[stage], pl_val_rot[stage] ^ pl_key[stage] );

   always_ff @( posedge clk ) begin
      pl_val_rot[0] <= val;    pl_key[0] <= key;
      for ( int stage=0; stage<wv; stage++ ) begin
         automatic logic new_low = stage==0 || dif_here[stage] < pl_dif[stage];
         pl_dif[stage+1] <= new_low ? dif_here[stage] : pl_dif[stage];
         pl_pos[stage+1] <= new_low ? stage : pl_pos[stage];
         pl_key[stage+1] <= pl_key[stage];  // <---------------------- Hmm, key not changed.
         pl_val_rot[stage+1] <= { pl_val_rot[stage][0], pl_val_rot[stage][wv-1:1] };
      end
   end
endmodule
```

(*a*) Notice that in the module above `pl_key` is never changed. Variation Plan K, below, reduces cost by eliminating `pl_key`.

```
module best_rot_pipe_plan_K #( int wv = 17, wp = $clog2(wv+1) )
   ( output logic [wp-1:0] pos, dif,   input uwire [wv-1:0] val, key,   input uwire clk );
   logic [wv-1:0] pl_val_rot[wv:0];
   logic [wp-1:0] pl_pos[wv:1], pl_dif[wv:1], dif_here[wv-1:0];
   assign pos = pl_pos[wv], dif = pl_dif[wv];

   for ( genvar stage=0; stage<wv; stage++ )
     pop #(wv,wp) p( dif_here[stage], pl_val_rot[stage] ^ key ); // <--- Important change.

   always_ff @( posedge clk ) begin
      pl_val_rot[0] <= val;
      for ( int stage=0; stage<wv; stage++ ) begin
         automatic logic new_low = stage==0 || dif_here[stage] < pl_dif[stage];
         pl_dif[stage+1] <= new_low ? dif_here[stage] : pl_dif[stage];
         pl_pos[stage+1] <= new_low ? stage : pl_pos[stage];
         pl_val_rot[stage+1] <= { pl_val_rot[stage][0], pl_val_rot[stage][wv-1:1] };
      end
   end
endmodule
```

☐ Describe what's wrong with Plan K.

☐ Describe two sets of non-trivial inputs. For the first Plan K provides the wrong answers (different than the original), for the other Plan K and the original agree.

Staple This Side

Staple This Side

(*b*) The two variations below, Plan B, and Plan b (case sensitive), use blocking assignments.

```
module best_rot_pipe_plan_B #( int wv = 17, wp = $clog2(wv+1) )
   // Omitted the code above this point. It's the same as the original code.
   always_ff @( posedge clk ) begin
      pl_val_rot[0] = val;    pl_key[0] = key;
      for ( int stage=0; stage<wv; stage++ ) begin
         automatic logic new_low = stage==0 || dif_here[stage] < pl_dif[stage];
         pl_dif[stage+1] = new_low ? dif_here[stage] : pl_dif[stage];
         pl_pos[stage+1] = new_low ? stage : pl_pos[stage];
         pl_key[stage+1] = pl_key[stage];
         pl_val_rot[stage+1] = { pl_val_rot[stage][0], pl_val_rot[stage][wv-1:1] };
      end
   end
endmodule
```

☐ Will the `Plan_B` code, above, compile and if so, run correctly? ☐ Explain.

```
module best_rot_pipe_plan_b #( int wv = 17, wp = $clog2(wv+1) )
   // Omitted the code above this point. It's the same as the original code.
   always_ff @( posedge clk ) begin
      for ( int stage=wv-1; stage>=0; stage-- ) begin // <---- Note change in loop direction.
         automatic logic new_low = stage==0 || dif_here[stage] < pl_dif[stage];
         pl_dif[stage+1] = new_low ? dif_here[stage] : pl_dif[stage];
         pl_pos[stage+1] = new_low ? stage : pl_pos[stage];
         pl_key[stage+1] = pl_key[stage];
         pl_val_rot[stage+1] = { pl_val_rot[stage][0], pl_val_rot[stage][wv-1:1] };
      end
      pl_val_rot[0] = val;    pl_key[0] = key;
   end
endmodule
```

☐ Will the `Plan_b` code, above, compile and if so, run correctly? ☐ Explain.

(*c*) Consider Plan P:

```
module best_rot_pipe_plan_P #( int wv = 17, wp = $clog2(wv+1) )
   ( output logic [wp-1:0] pos, dif,   input uwire [wv-1:0] val, key,   input uwire clk );
   logic [wv-1:0] pl_val_rot[wv:0], pl_key[wv:0];
   logic [wp-1:0] pl_pos[wv:1], pl_dif[wv:1], dif_here[wv-1:0];
   assign pos = pl_pos[wv], dif = pl_dif[wv];
   always_ff @( posedge clk ) begin
      pl_val_rot[0] <= val;    pl_key[0] <= key;
      for ( int stage=0; stage<wv; stage++ ) begin
         automatic logic new_low = stage==0 || dif_here[stage] < pl_dif[stage];
         pop #(wv,wp) p( dif_here[stage], pl_val_rot[stage] ^ pl_key[stage] ); // <--- Change.
         pl_dif[stage+1] <= new_low ? dif_here[stage] : pl_dif[stage];
         pl_pos[stage+1] <= new_low ? stage : pl_pos[stage];
         pl_key[stage+1] <= pl_key[stage];
         pl_val_rot[stage+1] <= { pl_val_rot[stage][0], pl_val_rot[stage][wv-1:1] };
      end
   end
endmodule
```

☐ Will the `Plan_P` code, above, compile and if so, run correctly? ☐ Explain.

Staple This Side

Problem 5: [20 pts] Synthesis of modules based on Homework 5 are reported in the tables below. It's not important to understand how these modules compute their results. It is important to understand how their combinational, sequential, and pipelined organizations affect result timing. Let w denote the value of parameter `wv`, the number of bits in the `val` and `key` inputs.

Module `best_rot_procedural` describes combinational logic. Module `best_rot_seq` is sequential and requires w cycles to compute a result. Module `best_rot_pipe` is pipelined and requires w cycles to compute a result. Module `best_rot_pipe_extra_stage` is also pipelined but requires $w+1$ cycles to compute a result. Appearing below is synthesis data for instantiations with `wv=8`.

Module Name	Area	Delay Actual	Delay Target	Synth Time
best_rot_procedural_wv8	184379	2.34	0.1 ns	340 s
best_rot_seq_wv8	52448	2.02	0.1 ns	26 s
best_rot_pipe_wv8	255256	2.27	0.1 ns	66 s
best_rot_pipe_extra_stage_wv8	265623	1.63	0.1 ns	69 s

Based on this data:

☐ What is the ☐ latency and ☐ throughput of `best_rot_procedural`?

☐ What is the ☐ latency and ☐ throughput of `best_rot_seq`?

☐ What is the ☐ latency and ☐ throughput of `best_rot_pipe`?

☐ What is the ☐ latency and ☐ throughput of `best_rot_pipe_extra_stage`?

(*a*) Appearing below is another table of synthesis data for `wv=8` instantiations of the modules. The costs (areas) are lower compared to the syntheses appearing above.

Module Name	Area	Delay Actual	Delay Target	Synth Time
best_rot_procedural_wv8	77188	4.59	100.0 ns	51 s
best_rot_seq_wv8	31271	3.58	100.0 ns	3 s
best_rot_pipe_wv8	181612	3.81	100.0 ns	24 s
best_rot_pipe_extra_stage_wv8	198426	2.89	100.0 ns	25 s

☐ Why are the costs in this second set lower?

☐ Why is `best_rot_seq` less expensive than `best_rot_procedural`?

☐ Why is `best_rot_pipe` more expensive than `best_rot_procedural`?

Staple This Side

Staple This Side

Problem 6: [10 pts] Answer each question below.

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

```
module short;
 int a, b, c, d, e, f;

 always_comb b = a;
 always_comb e = f + b;
 always_comb f = a + 10;
 always_comb d <= f + b + 1;

 initial begin
    a = 1;  b = 2;  c = 3;  d = 4;  e = 5;  f = 6;

    c = b;

    a = 7;

    b <= 20;
    d = b;

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

    #1;

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

 end
endmodule
```

(*b*) In the simple model the time to compute the sum of three integers, such as $a+b+c$, is much less than twice the time to compute the sum of two items, such as $a+b$.

☐ Why is the time to compute $a+b+c$ much less than twice the time to compute $a+b$? ☐ What about the numbers and hardware are we assuming?

☐ Can the same technique be applied to floating-point numbers? ☐ Explain.

Staple This Side