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)

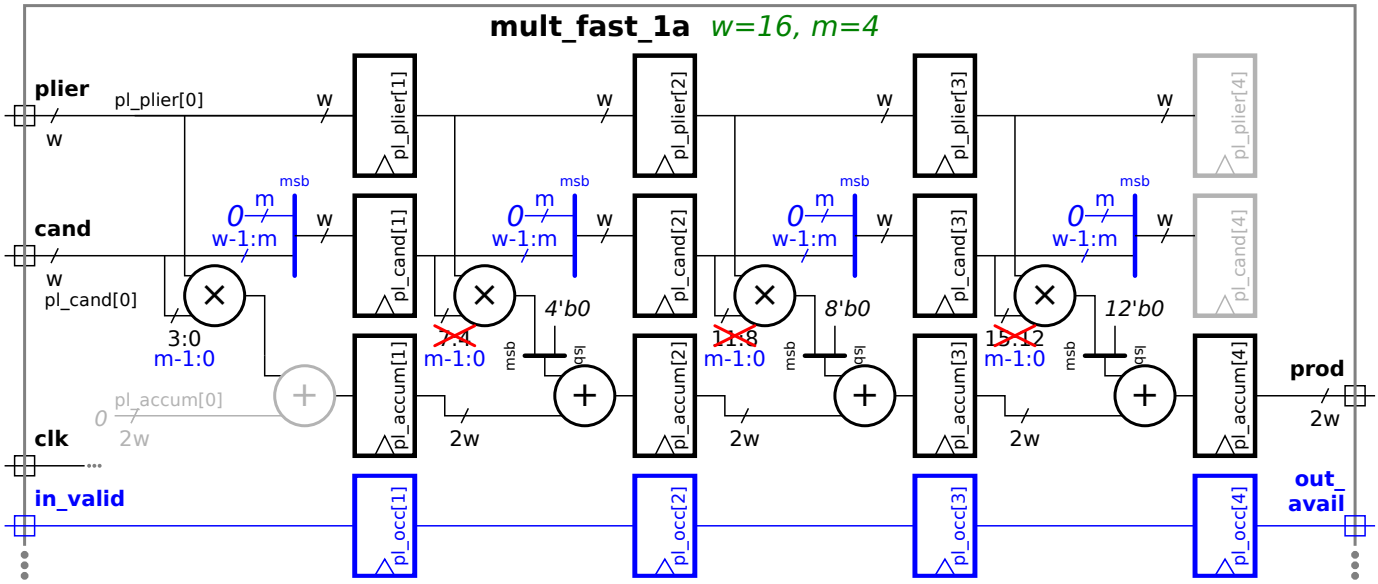
Alias Pie Plain

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.

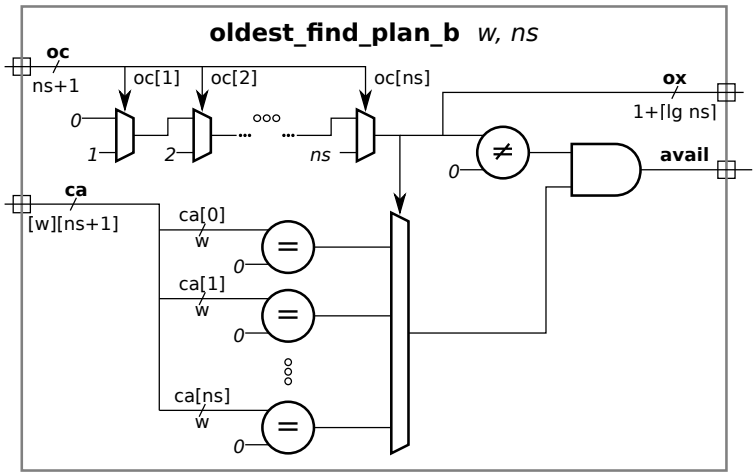
```
module mult_fast_1a #( int w = 16, int m = 4 )
   ( output uwire [2*w-1:0] prod,
     output uwire out_avail,       input uwire clk, in_valid,      //  ☑ DIFFERS
     input uwire [w-1:0] plier, cand );
   localparam int nstages = ( w + m - 1 ) / m;
   logic [2*w-1:0] pl_accum[0:nstages];
   logic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

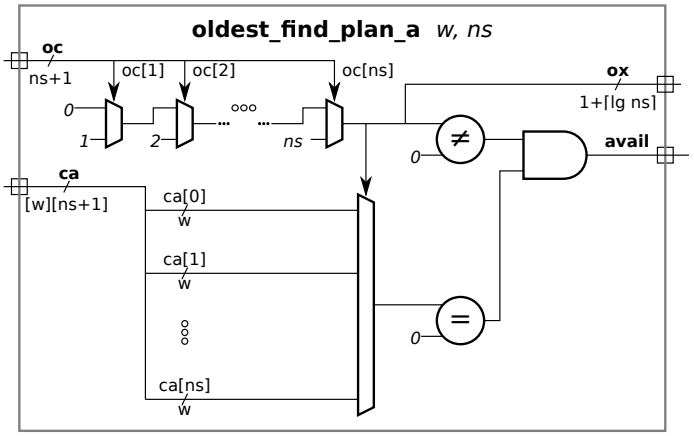
Problem 3: [20 pts] Appearing below are two variations on the oldest index module from the previous problem. The Plan A version is based on the code from the posted Homework 7 solution. The Plan B module is slightly different.

(*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.

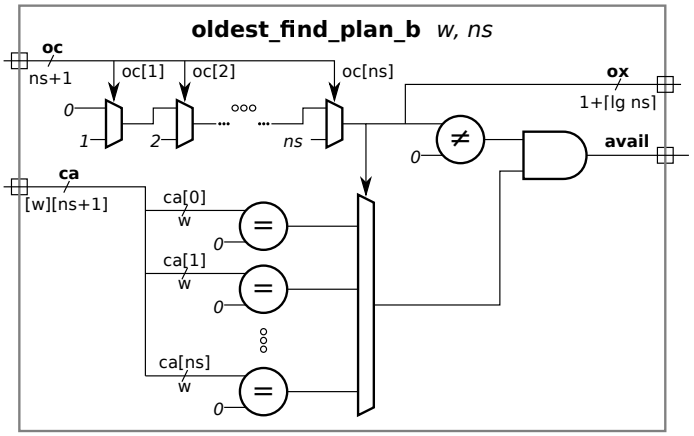


The $\neq 0$ unit can be realized using a $\lceil \lg n \rceil$-input OR gate, and the $= 0$ unit can be realized using a w-input NOR gate. The costs are the number of inputs minus one. The total cost is:

$$\overbrace{(n-2)\lceil \lg n \rceil}^{\text{2-input muxen}} + \overbrace{\lceil \lg n \rceil - 1}^{\neq 0} + \overbrace{3nw}^{\text{Big Mux}} + \overbrace{w-1}^{=0} + \overbrace{1}^{\text{AND}}$$

☑ 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:

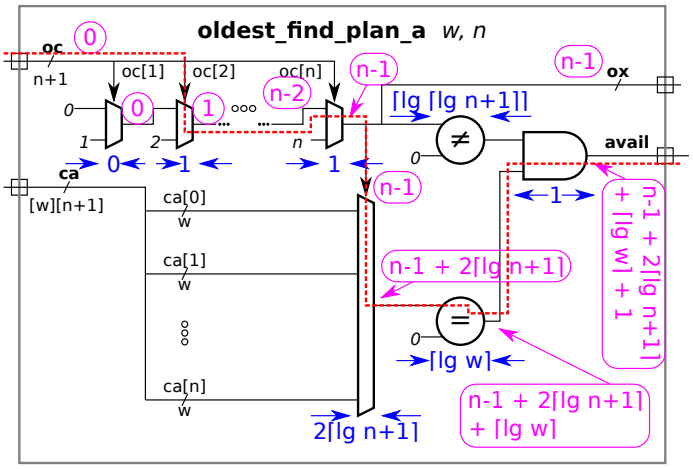
$$\overbrace{(n-2)\lceil \lg n \rceil}^{\text{2-input muxen}} + \overbrace{\lceil \lg n \rceil - 1}^{\neq 0} + \overbrace{(n+1)(w-1)}^{=0} + \overbrace{3n}^{\text{Big Mux}} + \overbrace{1}^{\text{AND}}$$

(*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.

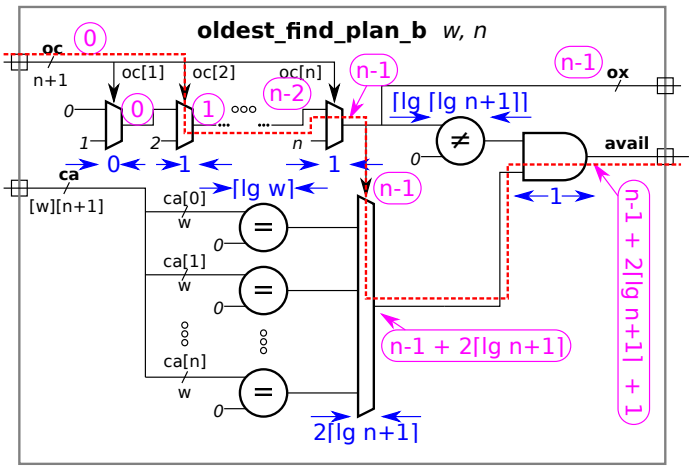
A common mistake was to overlook the possibility that the critical path can pass through a multiplexor select input, as it does here.



☑ 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.

Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9
$t = 0$	$t = 0$	$t = 0$	$t = 0$	$t = 0$	$t = 0$	$t = 0$	$t = 0$	$t = 2$
Active **C5a**↗	*Active*	*Active* **C1**↗ C2 C3	*Active* **C2**↗ C3	*Active* **C3**↗	*Active*	*Active* **C4**↗	*Active*	*Active* **C5b**↗
Inactive	*Inactive* C1 C2 C3	*Inactive*	*Inactive*	*Inactive*	*Inactive* C4	*Inactive*	*Inactive*	*Inactive*
NBA	*NBA*	*NBA*	*NBA*	*NBA*	*NBA*	*NBA*	*NBA*	*NBA*
	$t = 2$	$t = 2$	$t = 2$	$t = 2$	$t = 2$	$t = 2$	$t = 2$	
	Inactive C5b	*Inactive* C5b	*Inactive* C5b	*Inactive* C5b	*Inactive* C5b	*Inactive* C5b	*Inactive* C5b	

Step 10	Step 11	Step 12	Step 13	Step 14	Step 15	Step 16	Step 17	Step 18
$t = 2$	$t = 2$	$t = 2$	$t = 2$	$t = 2$	$t = 2$	$t = 2$	$t = 2$	$t = 2$
Active	*Active* **C1**↗ C2	*Active* **C2**↗	*Active*	*Active* b← 11↗	*Active*	*Active* **C1**↗ C2 C3	*Active* **C2**↗ C3	*Active* **C3**↗
Inactive C1 C2	*Inactive*	*Inactive*	*Inactive*	*Inactive*	*Inactive* C1 C2 C3	*Inactive*	*Inactive*	*Inactive*
NBA b← 11	*NBA* b← 11	*NBA* b← 11	*NBA* b← 11	*NBA*	*NBA*	*NBA*	*NBA*	*NBA*
$t = 4$	$t = 4$	$t = 4$	$t = 4$	$t = 4$	$t = 4$	$t = 4$	$t = 4$	$t = 4$
Inactive C5c	*Inactive* C5c	*Inactive* C5c	*Inactive* C5c	*Inactive* C5c	*Inactive* C5c	*Inactive* C5c	*Inactive* C5c	*Inactive* C5c

Step 19	Step 20	Step 21	Step 22
$t = 2$	$t = 2$	$t = 2$	$t = 4$
Active	*Active* **C4**↗	*Active*	*Active* **C5c**↗
Inactive C4	*Inactive*	*Inactive*	*Inactive*
NBA	*NBA*	*NBA*	*NBA*
$t = 4$	$t = 4$	$t = 4$	
Inactive C5c	*Inactive* C5c	*Inactive* C5c	

(*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.