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.

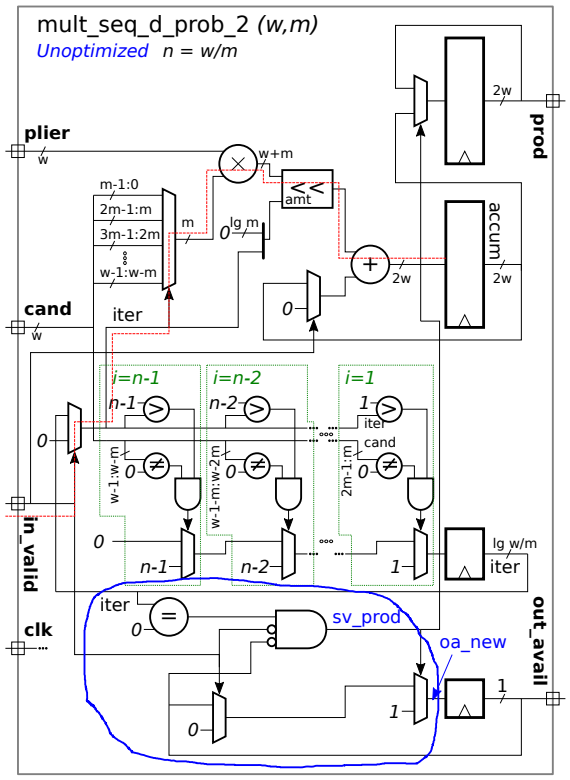
☑ Show optimized hardware.

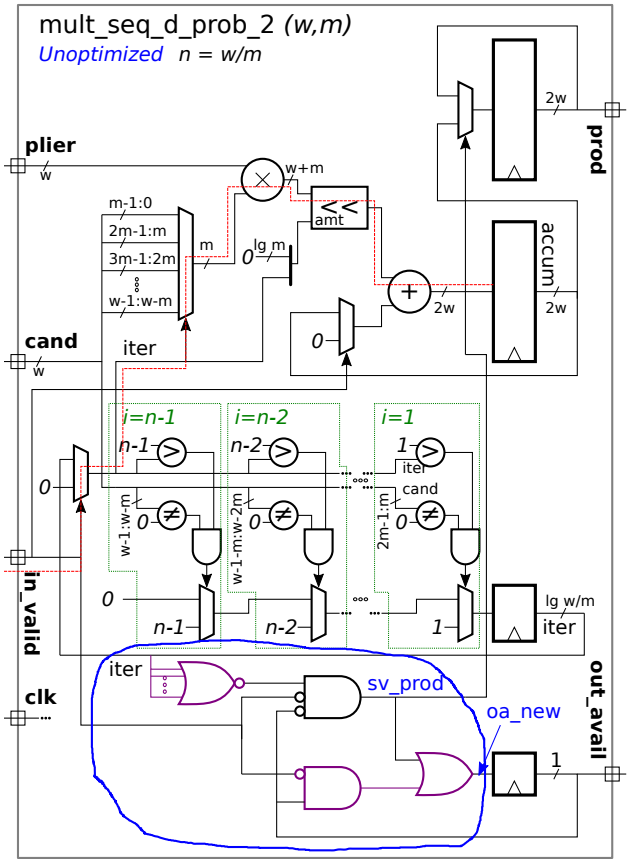
Solution appears to the lower-right in purple.

☑ Cost of optimized hardware:

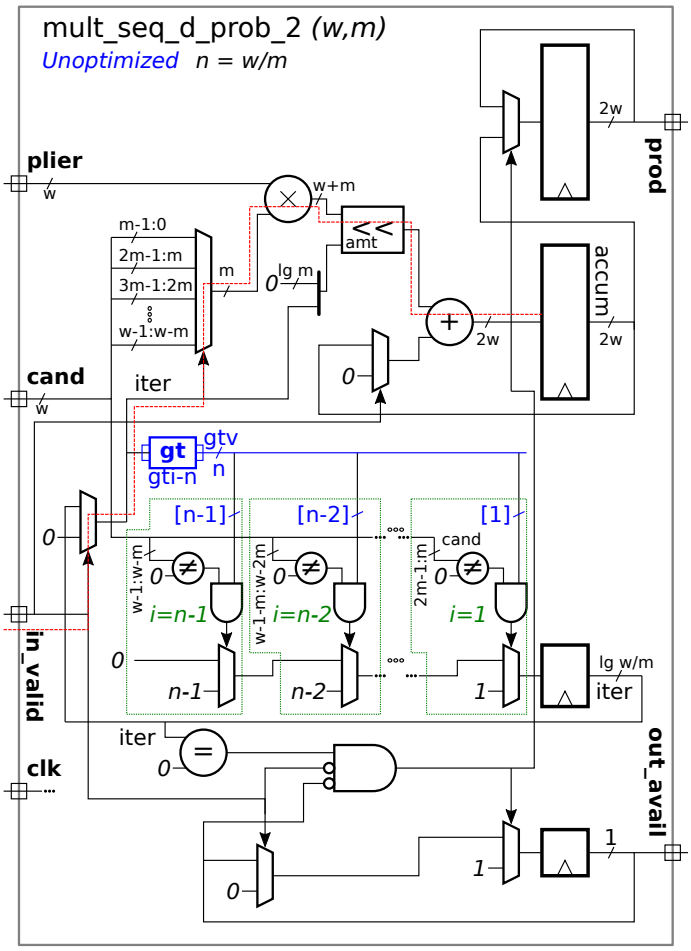
The $\lceil \lg n \rceil$-input NOR gate implementing `iter==0` costs $[\lceil \lg n \rceil - 1]\,\mathrm{u_c}$.

The new AND and OR gates cost $1\,\mathrm{u_c}$ each. The existing (and unchanged) three-input AND gate costs $2\,\mathrm{u_c}$. The total cost is $[\lceil \lg n \rceil + 3]\,\mathrm{u_c}$.





(*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}$ units have been replaced by one module, `gt`, the changed hardware appears in blue.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
```

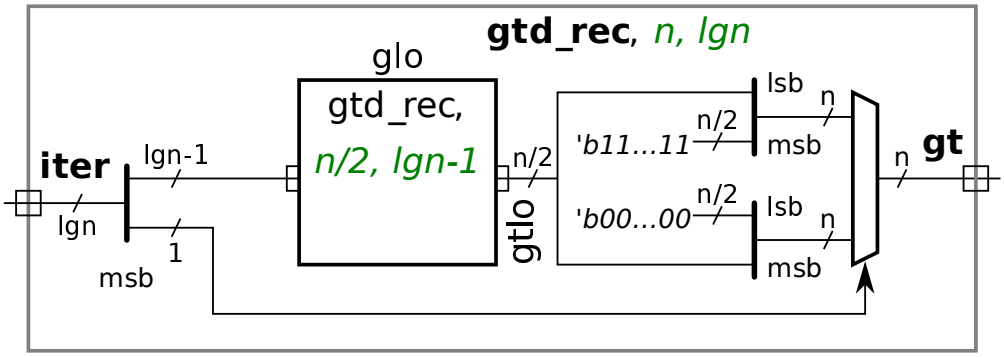
Problem 2: [25 pts] The point of the `gt` module in the previous problem was to reduce cost, just in case the synthesis program didn't notice that the cost of computing each of `n-1>iter`, `n-2>iter`, ..., `2>iter`, `1>iter`, would be less than $n-1$ times the cost of computing one of them. The recursive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

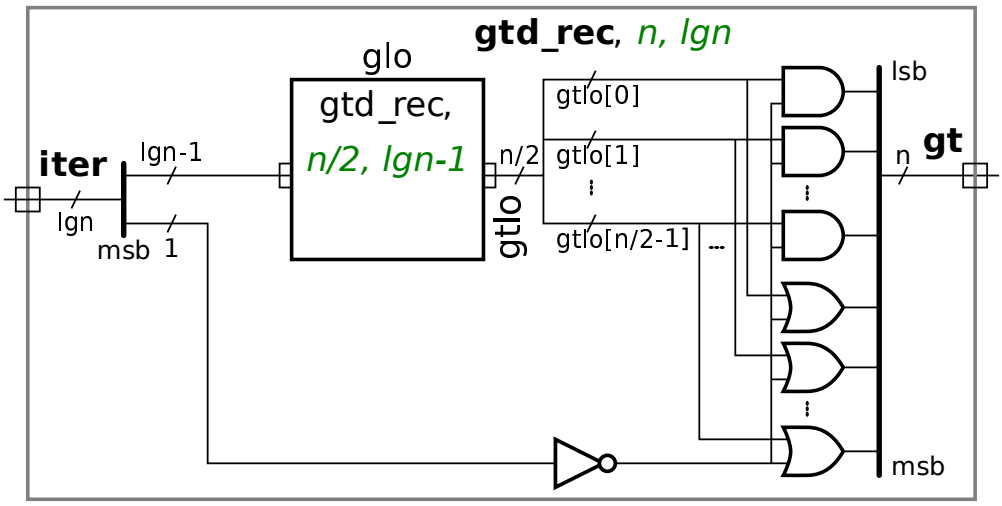
Solution appears below.



(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.

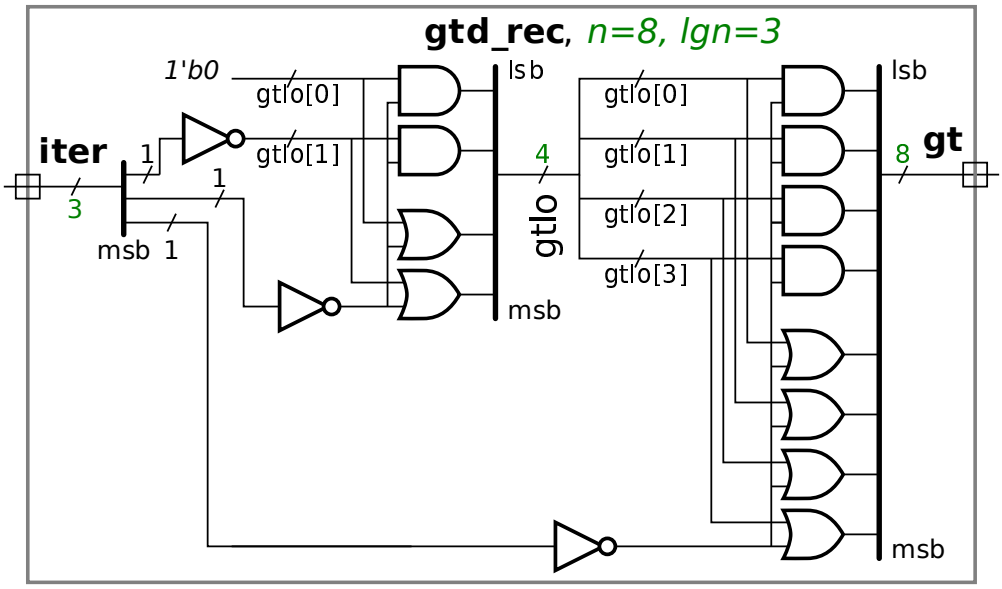
Solution appears below. From the previous solution notice that the $n/2$ LSB of the lower mux input are all zeros. Therefore we can optimize the three gates per bit, into just an AND gate using the inverted select signal. Similarly, the $n/2$ MSB of the upper mux input are all 1's, so we can optimize 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

$$\sum_{l=2}^{\eta} 1\,\mathrm{u_t} = (\eta - 1)\,\mathrm{u_t} = (\lg n - 1)\,\mathrm{u_t}.$$

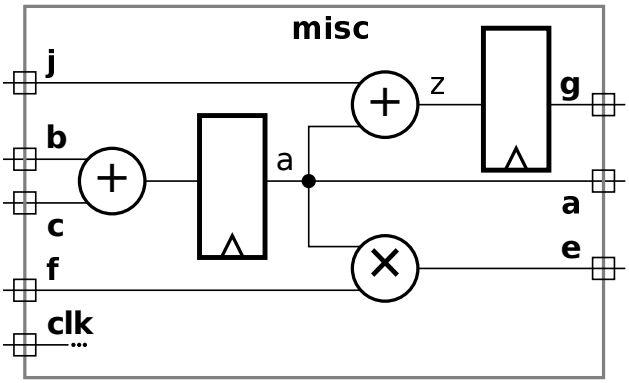
Problem 3: [20 pts] Consider the module below.

```
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.*