LSU EE 4755 Homework 4 Solution Due: 12 October 2015

Problem 0: Follow the instructions for account setup and homework workflow on the course procedures page, `http://www.ece.lsu.edu/koppel/v/proc.html`. Run the testbench on the unmodified file. There should be errors on the `shift_lt_seq_d_sol` module, but the others should run correctly. Run the Note: There are no points for this problem.

Problem 1: The homework Verilog file, `hw04.v`, contains a module `shift_lt_seq_d_sol` which is based on `shift_lt_seq_d`. It contains an `always_ff` block that assigns the same variables that are assigned in `shift_lt_seq_d`, however it assigns them from variables of the same name with `next_` prefixed:

```
always_ff @( posedge clk ) begin
   ready = next_ready;
   shifted = next_shifted;
   shift = next_shift;
   cnt = next_cnt;
end
```

Add code so that these `next_` objects will be assigned values from combinational logic, and so that the resulting module describes the same hardware as `shift_lt_seq_d`. A hand-drawn diagram of synthesized hardware should be identical, though it's possible that there will be small differences in the actual output of a synthesis program.

The added code can be implicit structural or behavioral, but it must synthesize to combinational logic.

The simplest approach is to start with the `always_ff` block from module `shift_lt_seq_d`. Change the always type to `always_comb` and rename some of the objects that are to synthesize to registers, namely `ready`, `shifted`, `shift`, and `cnt`.

If an assignment is made to any of these in the `always_comb` block, the assignment must be changed to write the `next_` version. For example change `cnt=amt;` to `next_cnt=amt;`. The right-hand side of an assignment should only use the `next_` version of a variable if it was assigned earlier in the block. For example, `next_shift` in the excerpt from the solution below:

```
        next_shift[i] = cnt[i] > 0;
        next_cnt[i] = next_shift[i] ? cnt[i] - 1 : cnt[i];
```

The code also has to be modified so that each of the `next_` variables is assigned at least once no matter what path is taken through the `always_comb` block. That is, they must be assigned for every possible outcome of the `if` statements. That's why there is no `if` statement in the assignment to `next_cnt` above. (That is, the following would be wrong: `if(next_shift[i])next_cnt[i]=cnt[i]-1`). (If a variable is not always assigned then its value will come from the output of a latch, rather than from combinational logic.)

The solution uses both continuous assign statements and an `always_comb` block. The complete solution appears below:

```
module shift_lt_seq_d_sol
  #( int wid_lg = 4, int num_shifters = 2, int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,       output logic ready,
     input [wid-1:0] unshifted,            input [wid_lg-1:0] amt,
     input start,                          input clk );

   logic [num_shifters-1:0] shift;
```

```
  wire [wid-1:0]           shin[num_shifters-1:-1];
  localparam int bits_per_seg = wid_lg / num_shifters;
  for ( genvar i=0; i<num_shifters; i++ ) begin
     localparam int fs_amt = 2 ** ( i * bits_per_seg );
     shift_fixed #( wid_lg, fs_amt ) sf( shin[i], shin[i-1], shift[i] );
  end

  assign shin[-1] = shifted;

  logic [num_shifters-1:0][bits_per_seg-1:0] cnt;
  logic [wid-1:0] next_shifted;
  logic next_ready;
  logic [num_shifters-1:0] next_shift;
  logic [num_shifters-1:0][bits_per_seg-1:0] next_cnt;

  always_comb begin

     if ( start == 1 ) begin

        next_cnt = amt;
        next_shift = 0;

     end else begin

        for ( int i=0; i<num_shifters; i++ ) begin
           next_shift[i] = cnt[i] > 0;
           // Note that next_cnt is always assigned, this avoids latches.
           next_cnt[i] = next_shift[i] ? cnt[i] - 1 : cnt[i];
        end
     end
  end

  // Use a continuous assignment for next_ready and next_shifted.
  assign next_ready = start ? 0 : cnt == 0 ? 1 : ready;
  assign next_shifted = start ? unshifted : shin[num_shifters-1];

  always_ff @( posedge clk ) begin
     shifted = next_shifted;
     ready = next_ready;
     shift = next_shift;
     cnt = next_cnt;
  end

endmodule
```

Problem 2: Module `shift_lt_seq_d_live` takes one more cycle to produce a result than module `shift_lt_seq_d`. Module `shift_lt_seq_d_p2` initially is identical to `shift_lt_seq_d_live`.

(*a*) Modify `shift_lt_seq_d_p2` so that it uses one less cycle to produce a result without changing the number of shifters per stage. There are two possible ways of doing this, performing some work in the same cycle that the `start` signal arrives, or doing work in the cycle when `ready` is set to 1. Either method is fine.

The original module, **shift_lt_seq_d_live**, does not start to shift until the cycle after **start** is set to 1. In the solution the logic generating the **shift** signal is moved so that it operates at every cycle. That was done by moving the **i** loop out of the **if/else** block, the logic generating the **ready** signal was also moved.

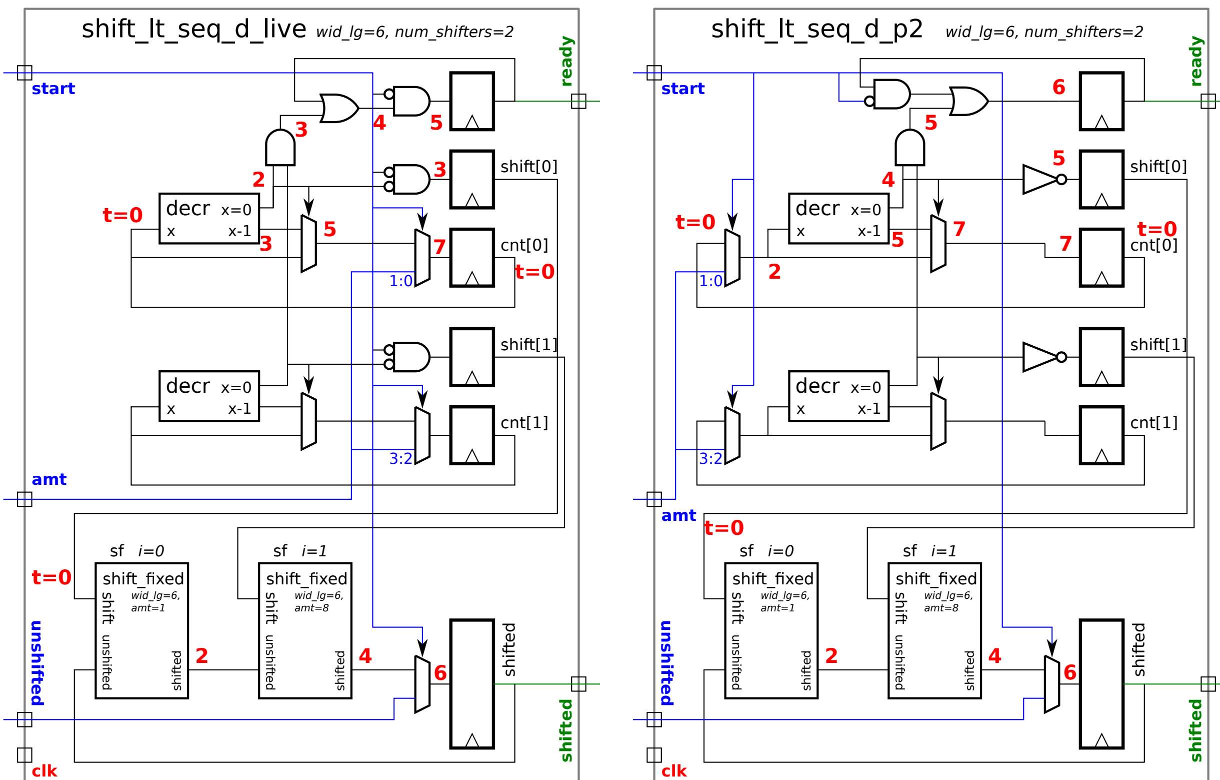
By doing this we are requiring **start** and **amt** to arrive early in the cycle. Before the change they could arrive late in the cycle.

```
module shift_lt_seq_d_p2
  #( int wid_lg = 6, int num_shifters = 1, int wid = 1 << wid_lg )
   ( output logic [wid-1:0] shifted,       output logic ready,
     input [wid-1:0] unshifted,            input [wid_lg-1:0] amt,
     input start,                          input clk );

   localparam int bits_per_seg = wid_lg / num_shifters;

   logic [num_shifters-1:0] shift;
   wire [wid-1:0] shin[num_shifters-1:-1];
   assign shin[-1] = shifted;

   for ( genvar i=0; i<num_shifters; i++ ) begin
      localparam int fs_amt = 2 ** ( i * bits_per_seg );
      shift_fixed #( wid_lg, fs_amt ) sf( shin[i], shin[i-1], shift[i] );
   end

   logic [num_shifters-1:0][bits_per_seg-1:0] cnt;

   always_ff @( posedge clk ) begin

      if ( start == 1 ) begin
         ready = 0;
         cnt = amt;
         shifted = unshifted;
      end else begin
         shifted = shin[num_shifters-1];
      end

      if ( cnt == 0 ) ready = 1;

      for ( int i=0; i<num_shifters; i++ ) begin
         shift[i] = cnt[i] > 0;
         if ( cnt[i] != 0 ) cnt[i]--;
      end

   end

endmodule
```

(*b*) Run **syn.tcl** and compare the cost and performance of your design and **shift_lt_seq_d_live**. Comment on the differences. An answer might start *"The cost was about the same because the same hardware was used..."*.

A table showing area (cost) and timing as reported by the synthesis program appears below. That's followed by a sketch of our guess of the synthesized hardware for each module, along with a timing analysis. These expectations are compared with the output of the synthesis program.

Module Name	Area	Delay Actual	Delay Target
shift_lt_seq_d_live_wid_lg6_num_shifters1	68368	1253	100
shift_lt_seq_d_p2_wid_lg6_num_shifters1	68428	1229	100
shift_lt_seq_d_live_wid_lg6_num_shifters2	77528	1355	100
shift_lt_seq_d_p2_wid_lg6_num_shifters2	78700	1348	100
shift_lt_seq_d_live_wid_lg6_num_shifters3	96648	1527	100
shift_lt_seq_d_p2_wid_lg6_num_shifters3	95820	1539	100
shift_lt_seq_d_live_wid_lg6_num_shifters6	143412	2002	100
shift_lt_seq_d_p2_wid_lg6_num_shifters6	142380	2007	100



To determine the expected area and timing differences between the two modules examine the sketches of the expected synthesized hardware for the two modules, which appears above. The change that enables us to save a cycle is moving the mux that selects a new value of **amt** from the input of **cnt** to the input of the decrement unit. That lets the shifter get started one cycle earlier.

Notice that by moving the hardware to compute **cnt** and **shift** out of the loop we are simplifying the logic at the input to those registers because they no longer have to check **start**. For this reason we would expect the cost to be slightly lower. The costs reported by the synthesis program are close and show no consistent pattern.

The sketches of the expected hardware include a simple timing analysis. The timing analysis is based on an assumed delay of two units for a mux, $\lceil \lg n \rceil$ units for an n-input gate and a delay of 3 for a 3-bit decrementor.

Based on this analysis the changes in the **p2** module don't affect the path that ends in the **shifted** register, that's the same 6 units in both cases.

Moving the **amt** mux from **cnt** to the decrementer inputs does not change the critical path. The move does delay the **shift** and **ready** signals by one or two units, but since they are not critical it doesn't matter. When **num_shifters** is 1 the path ending at **cnt** remains critical so moving the **mux** doesn't change anything. When **num_shifters** is larger the path ending at **shifted** is critical so moving the mux has no impact.

Based on this analysis we would not expect a change in the clock period. The output of the synthesis program shows only small changes.

The fact that the clock period is about the same is good news for us since one less clock cycle is needed. If the changes increased the clock period we may not actually get higher performance.