LSU EE 4755 Homework 6 Due: 4 December 2019

For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`*. For the complete Verilog for this assignment without visiting the lab follow* `https://www.ece.lsu.edu/koppel/v/2019/hw06.v.html`*.*

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account (if for whatever reason you haven't done so or need to do it again), copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw06.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

Module `add_accum` keeps a running total of values appearing at its inputs. A 1-bit input `ai_valid` indicates whether the value on w-bit input `ai` should be added to the total (`ai_valid==1`) or ignored (`ai_valid==0`). These signals should be examined on the positive edge of input `clk`. The module places a running sum of these values on output `sum`. The sum should be reset to 0 when input `reset` is 0 at a positive edge. The Verilog below implements the behavior described so far.

```
module add_accum
  #( int w = 20, n_stages = 3 )
   ( output logic [w-1:0] sum,
     output logic sum_valid,
     input uwire [w-1:0] ai,
     input uwire ai_valid, reset, clk );

   always_ff @( posedge clk )
     if ( reset ) sum = 0; else if ( ai_valid ) sum += ai;

   always_comb sum_valid = 1;

endmodule
```

A student at this point might wonder if this is going to be a dull assignment. No, of course not! Did you notice the parameter `n_stages`? That indicates that the module shall [I understand why shall is used instead of should in some contexts, but still it sounds too bossy to me] use a pipelined adder of `n_stages` stages. The point of this assignment is to modify the module above so that it uses the provided (and pre-instantiated) pipelined adder.

There are two challenges here. The straightforward challenge of connecting the pipelined input and output ports properly. Then there's the perhaps unexpected and interesting challenge of properly updating the running sum when input values arrive even while the calculation of a sum is still in the pipeline. The module has an output `sum_valid` that should only be set to `1` when output `sum` is the correct sum of all arriving valid values since the most recent reset.

After a reset `sum` should be set to zero and `sum_valid` to `1`. When the first value arrives `sum` might change to that arriving value by the next clock cycle (no adder needed). But when the second value arrives it will be necessary to add it to the first (the current sum) and since the pipelined adder takes several cycles `sum_valid` will have to be set to zero while the adder is computing. If no other new values arrive before the adder is finished `sum` can be set to the sum and `sum_valid` should again be set to `1`. Suppose instead that while the adder is operating on the first two values, a third value arrives? Then when the adder is finished the third value will have to be added to the just-completed sum. There is no restriction on when values can arrive. They may arrive every cycle or with large gaps between arrivals. If values arrive frequently then `sum_valid` may remain

0. But if values stop arriving sum_valid should eventually be set to 1 and sum should be set to the correct sum.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing $\boxed{\texttt{F9}}$, tests add_accum instantiated for different pipeline lengths. It will check that the output values are correct, and that they don't appear too early or too late. Initially the testbench will report that there were 0 incorrect values but that they all arrived too early. The testbench will report the first four errors of each time for each pipeline length. The error message is followed by a string describing when the module was last reset and when values have since arrived. For example:

```
At cyc 7, value ready too soon, 0, cyc. (Min cyc 8.)
 R(4)+42(5)+40(7)
```

This indicates that at cycle 7 the value arrived too soon, after 0 cycles instead of after a minimum of 8 cycles. (The first value can appear after 0 cycles since there's nothing to add.) The R(4) indicates that the most recent reset was in cycle 4. The +42(5) indicates that the value 42 was at the input to the module in cycle 5.

A tally of errors and other information is shown after each pipeline length:

```
Done with 6-stage tests, 10000 series.
 Correct, 65271; errors : 0 not done, 0 val, 45273/0 early/late.
For 6 stages average latency 0.15 cycles.
```

The number after correct was the number of correct values found. To the left of "not done" is the number of tests skipped due to unresponsiveness. The number to the left of val is the number of incorrect results. The numbers to the left of early/late indicate the number of values appearing too early (45273 in the example above) or too late (0 in the example).

The testbench enforces a minimum time for all but the first value after a reset. The minimum time, n_stages, is assigned to parameter lat_min_empty in module testbench_n. The testbench enforces two maximum times. If the module is asserting sum_valid and a new value arrives, the updated sum should appear within lat_limit_empty = n_stages + 2 cycles. (That's also a testbench parameter.) If sum_valid is 0 and a new value arrives the testbench will patiently wait lat_limit_full = 2 + (1+$clog2(n_stages)) * (n_stages + 1) cycles. These testbench parameters can be changed to help with debugging, but they should be set back. The ta-bot will test the code using a different copy of the testbench module.

Following the error tally an average latency is shown, in this case less than 1 cycle. A low number is good so long as the pipelined adder is being used (which it isn't in the example above).

The following is output if the problem is solved correctly:

```
Starting tests for 2-stage pipeline.
Done with 2-stage tests, 10000 series.
 Correct, 35763; errors : 0 not done, 0 val, 0/0 early/late.
For 2 stages average latency 3.26 cycles.
Starting tests for 3-stage pipeline.
Done with 3-stage tests, 10000 series.
 Correct, 32338; errors : 0 not done, 0 val, 0/0 early/late.
For 3 stages average latency 4.64 cycles.
Starting tests for 5-stage pipeline.
Done with 5-stage tests, 10000 series.
 Correct, 28774; errors : 0 not done, 0 val, 0/0 early/late.
For 5 stages average latency 7.77 cycles.
Starting tests for 6-stage pipeline.
```

```
Done with 6-stage tests, 10000 series.
 Correct, 27737; errors : 0 not done, 0 val, 0/0 early/late.
For 6 stages average latency 9.48 cycles.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
Total number of errors: 0
```

Use Simvision to debug your modules. Finding errors in sequential code without a debugger is time consuming and tedious. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

Synthesis

The synthesis script, `syn.tcl`, will synthesize `add_pipe` (for reference) and `add_accum`. Each module will be synthesized at several pipeline depths, and with two delay targets, a delay-is-nothing-to-worry-about 10 ns and an unachievable 0.1 ns. If a module doesn't synthesize −.001 s is shown for its delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus. If you would like to synthesize additional modules or sizes edit `syn.tcl` near the bottom.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew-file.log`.

Problem 1: Modify module `add_accum` so that it keeps an accumulated sum (see the intro above) using a pipelined adder. The module must be synthesizable. A pipelined adder has been instantiated and some starter solution code has been included:

```
module add_accum
  #( int w = 20, n_stages = 3 )
   ( output logic [w-1:0] sum,  output logic sum_valid,
     input uwire [w-1:0] ai,  input uwire ai_valid, reset, clk );

   always_ff @ ( posedge clk )
     if ( reset ) sum = 0; else if ( ai_valid ) sum += ai;

   always_comb sum_valid = 1;

   /// The code above must be removed and the pipelined adder, add_p0, used instead.

   uwire [w-1:0] aout;
   uwire [w-1:0] a0 = ai;  // May need other connections.
   uwire [w-1:0] a1;

   add_pipe #(w,n_stages) add_p0(aout,a0,a1,clk);

   logic [n_stages:0] st_occ; // Indicate which stage of add_p0 is occupied.

   uwire aout_valid = st_occ[n_stages-1];

   always_ff @( posedge clk ) if ( reset ) begin
      st_occ <= 0;
   end else begin

      // Keep track of which stage of add_p0 is occupied.
      st_occ[0] <= ai_valid;  // Lets initially assume all values enter pipe.
      // Advance other occupied signals.
      for ( int i=1; i<=n_stages; i++ ) st_occ[i] <= st_occ[i-1];
   end
endmodule
```

The module above correctly computes the accumulated sum, however it does not use the pipelined adder. The pipelined adder has been instantiated and one input has been connected (though it may need to be connected to additional items).

Beneath the pipelined adder is code needed to keep track of which stages of the adder have values. Bit `st_occ[i]` is 1 if stage `i` of the adder has a valid value. Stage 0 is initialized with the module input's valid signal. Values are advanced one position per cycle. Net `aout_valid` is 1 if the adder output is valid, which will be true `n_stages` cycles after `ai_valid` is 1.

As described in the introduction, this problem would be easy if new values arrived at least `n_stages` cycles apart, because in that case the accumulated sum and the new value could be placed in the adder without worry. But a new value can arrive while the adder is busy with one or more computations, so the new value must be buffered until there is something to add it to, either a second new value or something emerging from the pipeline.

See the checkbox items in the Verilog code for additional items to look for. A diagram like the one below might help in solving this problem.

```
Cycle      0  1  2  3  4  5  6
ai_valid   1        1

a0         ai       ai
a1         sum      sum

aout_valid 0     1  0     1
sum              =ao   =ai
Cycle      0  1  2  3  4  5  6
```