LSU EE 4755 Homework 4 Solution Due: 11 November 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/hw04.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 neeed to do it again), copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw04.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

Module `best_match_behavioral` has two inputs, a longer vector, `val`, and a short vector, `k`. It sets `pos` to the start of a subvector of `val` that best matches `k` and sets `err` to the number of bit positions that don't match. For example, suppose `val = 8'b11110000` and `k=4'b1100`. Then `pos` would be set to 2 and `err` to 0 because there is an exact match at position 2 in `val`. If `k=4'b1101` then there isn't an exact match for `k` in `val`, but at position 2 there is a match with one error. If `k=2'b00` then there are matches at positions 0, 1, and 2, all with zero errors.

Module `best_match_behavioral` is combinational (and was written as a behavioral module). In this assignment a sequential version will be written and analyzed.

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 the modules. Initially, the testbench will exit because module `best_match` has not responded in sufficient time. When that happens one of the last lines of the testbench output shows that the final cycle count is the same as the cycle limit (128 below), and "CYCLE LIMIT EXCEEDED" is shown.

```
ncsim> run
Exit from clock loop at cycle 128, limit 128.  ** CYCLE LIMIT EXCEEDED **
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit

Compilation finished at Mon Nov  4 17:56:24
```

To get rid of this message `best_match` must handshake correctly, see Problem 1. If `best_match` responds in time, the testbench will check to see if `pos` is in the right range. The output below shows errors when `pos` is out of range:

```
Error in best_match, test #     3, pos out of range: 0xff
Error in best_match, test #     4, pos out of range: 0xff
Done with best_match_behavioral tests,           0 errors found.
Done with best_match tests,        1000 errors found.
Exit from clock loop at cycle 59001, limit 59069.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

The output `err` is supposed to be the number of non-matching bits at `pos`. If not, the testbench shows output like:

```
Error in best_match, test #     2, err wrong 1 != 3 (correct) pos   2  84 ^ 01
Error in best_match, test #     3, err wrong 1 != 2 (correct) pos  13  1f ^ 3d
Error in best_match, test #     4, err wrong 1 != 2 (correct) pos   4  78 ^ f9
Done with best_match_behavioral tests,           0 errors found.
Done with best_match tests,         972 errors found.
Exit from clock loop at cycle 59001, limit 59069.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

For test # 4, the testbench reports that `err` was 1 but should have been 2. The line also shows that `pos` was set to 4, and that `val` at that position was `78` (in hexadecimal) and that `k=f9`.

The testbench also checks whether the `err` returned is the minimum error for that value of `val` and `k`.

The testbench prints the details of the first few errors it finds. A grand total is printed at the end, see the transcript above.

Use Simvision to debug your modules. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

Synthesis

The synthesis script, `syn.tcl`, will synthesize `best_match_behavioral` (for reference) and `best_match` (your solution). Each module will be synthesized at three widths, and with two delay targets, an easy 90 ns and a un-achievable 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 on next page.

Problem 1: Complete module `best_match` so that it computes the best match sequentially as described below. In addition to `val` and `k`, the module has 1-bit inputs `start` and `clk` and 1-bit output `ready`.

Handshaking works as follows: When `start=1` at a positive edge the module should set `ready` to zero. It should then start scanning for the best match, checking one shifted position per cycle. The maximum number of cycles needed should be `wv-wk` plus one or two more needed for handshaking. (The testbench will wait `2*wv` cycles before giving up.) The module should set `err` and `pos` to their correct values and `ready` to 1.

The inputs, `val` and `k` will be held steady at least until `ready` is set to 1.

The module must use the `pop` (population) module (in `hw04.v`) to compute possible values for `err`. That is, don't use something like the `b` loop in `best_match_behavioral` to accumulate the sum `e`. Instead compute the XOR of the appropriate bit range and provide that to the `pop` module as an input.

For maximum credit avoid the use of large (such as `wv`-input) multiplexors in your design, or the use of a non-constant shifter.

The module must be synthesizable and correct.

The behavioral best match module is shown below for reference.

```
module best_match_behavioral
  #( int wv = 32, int wk = 10, int wvb = $clog2(wv), int wkb = $clog2(wk+1) )
   ( output logic [wvb:1] pos,  // Position of best match.
     output logic [wkb:1] err,  // Number of non-matching bits.
     input uwire [wv-1:0] val, input uwire [wk-1:0] k );

   always_comb begin

      automatic int best_err = wk + 1;
      automatic int best_pos = -1;

      for ( int p=0; p<=wv-wk; p++ ) begin
         automatic int e = 0;
         for ( int b=0; b<wk; b++ ) e += k[b] !== val[p+b];
         if ( e < best_err ) begin
            best_err = e;
            best_pos = p;
         end
      end
      err = best_err;
      pos = best_pos;

   end

endmodule
```

Solution on next page.

The solution appears below. The biggest difference between `best_match_behavioral` and `best_match` is that the `p`-loop has been eliminated, and the iteration variable, `p`, has been declared as a variable. The variable `p` is initialized to zero when `start` is asserted and then incremented each cycle until it points to the last position of a possible match, `wv-wp`.

Another difference is that the `b` loop, used to total the number of incorrect bit positions, has been replaced by the `pop` module. The input to the `pop` module is a bit vector, `e_vec`, which is constructed by exclusive-or'ing `k` with the low bits of `sh_val`. Bit `i` of `e_vec` is 1 if the bit `i` of `k` is different than bit `i` of `sh_val`, bit `i` is 0 if the bits are the same. Equivalently, `e_vec[i] = k[i] !== sh_val[i]`, or `e_vec[i] = k[i] ^ sh_val[i]`. Rather than iterating over `i` the entire value is computed using the bitwise exclusive OR operator: `e_vec = k ^ sh_val`.

The register `sh_val` is initialized to `val` and then shifted right by one bit each iteration. This avoids the need for a shifter. For example, if the error vector were computed using `e_vec = k ^ val[ p +: wk ];` a shifter would be needed for `val[p +: wk]`, to extract `wk` bits starting at position `p`.

```
module best_match
  #( int wv = 32, int wk = 10, int wvb = $clog2(wv), int wkb = $clog2(wk+1) )
   ( output logic [wvb:1] pos,  output logic [wkb:1] err,  output logic ready,
     input uwire [wv-1:0] val,  input uwire [wk-1:0] k,    input uwire start, clk );

   logic [wv-1:0] sh_val;
   logic [wvb-1:0] p;

   uwire [wk-1:0] e_vec = k ^ sh_val[wk-1:0];
   uwire [wkb-1:0] e;
   pop #(wk,wkb) p1( e, e_vec );

   always_ff @( posedge clk )

     if ( start == 1 ) begin

        ready = 0;
        sh_val = val;
        p = 0;
        err = wk;  // wk+1 might overflow err.

     end else if ( !ready ) begin

        if ( e < err ) begin  err = e;  pos = p;  end

        ready = p == wv - wk;
        p++;
        sh_val >>= 1;
     end

endmodule
```

Problem 2: Run the synthesis program and indicate how your module compares to the behavioral module.

Synthesis results appear below.

```
Module Name                                 Area   Delay   Delay
                                                   Actual  Target


best_match_wv16                            47923   3.786  90.000 ns
best_match_mux_wv16                        46566   5.181  90.000 ns
best_match_behavioral_wv16                 87155  10.862  90.000 ns

best_match_wv24                            60757   3.675  90.000 ns
best_match_mux_wv24                        60566   5.503  90.000 ns
best_match_behavioral_wv24                192546  21.535  90.000 ns

best_match_wv16_2                          63287   2.413   0.100 ns
best_match_mux_wv16_3                      77134   3.398   0.100 ns
best_match_behavioral_wv16_137            231102   3.504   0.100 ns

best_match_wv24_1                          79273   2.652   0.100 ns
best_match_mux_wv24_2                      89081   3.590   0.100 ns
best_match_behavioral_wv24_297            563769   6.667   0.100 ns
```

(*a*) Compare the amount of time needed for your module compared to the behavioral one. The answer to this question requires some manipulation of the values in the `Delay Actual` column. Indicate which results are expected, and which are not expected, and explain why.

The manipulation alluded to in the question is the multiplying of the delay by the number of cycles needed to compute the result (the position and error of the best match). The behavioral module is combinational, and so only one "cycle" is needed. (It's not really a cycle because the module isn't clocked.) The **best_match** module, in contrast, requires `wv-wk` cycles and so the delay must be multiplied by that number of cycles, $24-10 = 14$ for the 24-bit module and $16 - 10 = 6$ for the 16-bit module, in order to compute the time needed to find the best match.

Since we are comparing time we should look at the results for a delay target of 0.1 ns because it is in those runs that the synthesis program is optimizing delay. For the 24-bit module the behavioral module requires 6.667 ns to compute the best match. For `wv=24` and `wk=10` the **best_match** module in this solution requires at least $24 - 10 = 14$ cycles, for a total time of $14 \times 2.652\,\mathrm{ns} = 37.128\,\mathrm{ns}$. So the behavioral module will compute the error and position in much less time.

Some students submitted solutions that used fewer than `wv-wk` cycles when a perfect match (`err==0`) was found before bit `wv-wk` was reached. A student eager to showcase this clever shortcut could answer this question by describing a favorable situation: "For situations in which a perfect match occurs half the time and is uniformly distributed, ...

The question also asks for a discussion of whether the synthesis delay results were expected. That means we need to make some kind of a delay estimate for each module and compare it to the delays provided by the synthesis program. A starting point for the delay comparison is to recognize a key difference between the two modules: the behavioral module computes the error for each of $24 - 10 = 14$ positions (for the 24-bit module) in one cycle while the **best_match** module computes just one position per cycle. This factor of 14 (or $v - k$) difference would seem to put the behavioral module at a disadvantage. An important question to answer is whether the behavioral module's delay should be 14 times larger, $\lceil \lg 14 \rceil$ times larger, or something else. In the module generated by the synthesis program the behavioral delay is $6.667/2.652 \approx 2.5$ times larger.

The question did not explicitly ask us to compute the delay (say using the simple model), so that gives us some latitude for approximation. Full credit would have been given for all of the key points made so far in this solution. But having gotten this far, how can we not proceed further into the delay analysis? (Warning: EE 4755 Fall 2019 students are expected to read the entire solution. Exam questions will be based on homework assignments and the posted solutions, even the excessively wordy ones.)

First, consider the `b` loop in the behavioral module. It is doing the same thing as the assignment to `e_vec` and the hardware in `pop` module are doing in `best_match`. Let's assume that both will be synthesized to the same hardware after optimization (though in the case of the behavioral module there will be `wv-wk` copies of the hardware).

An important thing to remember is that the `p` loop and the `b` loop describe how synthesized hardware will be interconnected. They **do not execute** and don't even exist in the synthesized hardware. (The Verilog simulator does execute the loops as procedural code, but in this part we're considering synthesis.) The expression `k[b] !== val[p+b]` is synthesized into $(v-k)k$ pieces of hardware, one for each possible value of `p` ($(v-k)$ possible values) and `b` (k possible values). Since the values of `k` and `val` are available the beginning of a clock cycle all $(v-k)k$ comparisons are done simultaneously. The `b` loop describes a series of adders, computing the same sum as the `pop` module though describing the sum as a linear sequence of additions. If the synthesis program does its job well, meaning that it can re-associate the linear sequence of additions into a reduction tree, the delay for this will be $2 \lg k$ BFAs. Because of the way the BFAs are connected (possible final exam question?) we'll set the adder delay to 4 per BFA, for a total of $8 \lg k \,\mathrm{u_t}$. The input to each `b` loop is the same `val` and `k`, which are available at the start of a clock cycle. So taking into account the XOR delay each sum will be available at $[2 + 8 \lg k]\,\mathrm{u_t}$.

So far it looks like the time for the behavioral model to compute $v-k$ values of `e` is the same as the time needed by `best_match` to compute one. The difference in timing between the two is due to the code starting at `if ( e < best_err ) begin`. The problem is `best_err`. The value at iteration `p` depends on iteration `p-1` for `p>0`. Variable `best_err` is a live-in and live-out for an iteration. It's critical path passes through a comparison, `e < best_err`, and a mux (selecting the old or new `best_err`). If the comparison has delay $2 \lg k$ and the mux 2, the delay for $v-k$ iterations will be $[2 + 8 \lg k + (2(\lg k) + 2)(v-k)]\,\mathrm{u_t}$. The delay for `best_match` is roughly the of one iteration, $[2 + 8 \lg k + (2(\lg k) + 2)]\,\mathrm{u_t}$.

When $v-k$ is large the behavioral module would take $(v-k)$ times longer based on this analysis. For for $v-k = 24-10 = 14$ and $\lg k = 4$ the delays are much closer. For the behavioral delay $2+8\times4+(2\times8+2)(14) = 316\,\mathrm{u_t}$ and for `best_match` delay $2 + 8 \times 4 + (2 \times 8 + 2) = 82\,\mathrm{u_t}$, the modules have less than factor of 4 difference in delay. The synthesis program gives a difference of 2.5. Perhaps the synthesis program used a reduction tree for the `if ( e < best_err )` code. In that case the critical path would be through $\lceil \lg(v-k) \rceil = \lceil \lg 14 \rceil = 4$ layers, in which case delay would be $2 + 8 \times 4 + (2 \times 8 + 2)(4) = 136\,\mathrm{u_t}$, which works out to $136/82 = 1.66$ times longer than `best_match`. The difference in delays obtained from the synthesis program, 2.5, is somewhere between these two possibilities.

(*b*) Compare the area of your design to the behavioral one. Indicate which results are expected, and which are not expected, and explain why.

For the area comparison the $90\,\mathrm{ns}$ delay target runs should be used. For `wv=24` and `wk=10` the `p` loop iterates 14 times and so we would expect the behavioral code to have $14\times$ as much addition (including the `pop` module) and comparison hardware. The `best_match` module though needs a register for `sh_val`, something which the behavioral module does not need. Assume that the `pop` module and the expression totaling `e` use $2k$ BFAs each. (Approximated using $\sum_{i=1}^{\lg k} ik/2^i = 2(k-1) - \lg k$.) At a cost of $9\,\mathrm{u_c}$ per BFA, the cost of just the adders would be $18k\,\mathrm{u_c}$. For `best_match` there would be only one set of such adders, but for the behavioral module there would be $v-k$ such adders. For $v = 24$ and $k = 10$ the costs would be $180\,\mathrm{u_c}$ versus $2520\,\mathrm{u_c}$ for the behavioral module. Using $7\,\mathrm{u_c}$ per bit for `sh_val`, `pos`, and `err`, `best_match` would also require $7(v + \lg v + \lg(k+1)) = 7(24 + 5 + 4) = 231\,\mathrm{u_c}$ that the behavioral module lacks. The total so far is $411\,\mathrm{u_c}$ versus $2520\,\mathrm{u_c}$, a factor of 6 difference. The actual difference is closer to a factor of 3 when optimizing for area.