LSU EE 4755 Homework 3 Due: 13 October 2023

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/2023/hw03.v.html`.

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of Verilog syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out digital design resources for help on Verilog, digital design, etc. It is okay to make use of AI LLM tools such as ChatGPT and Copilot to generate sample Verilog code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account (if necessary), copy the assignment, and run the Verilog simulator on the unmodified homework file, `hw03.v`.

Homework Overview

As we probably know a *permutation* is a rearrangement of distinct objects. If there are n objects there are $n!$ permutations, including the identity permutation (which leaves the objects in their original positions). For example, the three-letter sequence `abc` can be permuted 6 ways: `abc, acb, bac, bca, cab, cba`. (In module `perm` input `pdata_in` is an unpermuted sequence and output `pdata_out` should be set to a permutation of the input.) There are many ways to specify which permutation we want. We could just say, I want permutation `acb`, meaning leave the first element unchanged and swap the next two. So permutation `acb` of `xyz` would be `xzy`. When specifying permutations this way it is more common to use digits, so rather than `acb` we would say permutation `021` of `xyz`. Here `021` indicates how we want things rearranged and `xyz` are the objects before being re-arranged and `xzy` are after the rearrangement.

Suppose we want to generate all permutations in, say, a loop. We might have a current permutation, `021`, and would like to generate the next one, say `102` (or `bac`). One way of doing that is using a *factorial number*. (This was the subject of `https://xkcd.com/2835`.) A factorial number is a mixed-radix number. (In module `perm` input `pnum_in` and output `pnum_out` are both factorial numbers. In the testbench the factorial numbers are called indices.) In an n-digit factorial number digit 0 (the LSD) is radix 1, digit 1 is radix 2 (binary), digit 2 is radix 3, and so on, to digit $n-1$. Digit 0, by the way, being radix 1, must always be zero. Digit 1 can be 0 or 1, digit 2 can be 0,1,2, etc. When all digits are zero the number specifies an identity permutation. Digit i of a factorial number specifies where to get the value to put in position i of the permuted sequence. To see examples of factorial numbers and the respective permutations look at the sample testbench outputs below.

With factorial numbers it's easy to compute the next permutation in a sequence: just add one. Start at the least significant digit. If there is a carry out (and there always is at the least significant digit) proceed to the next digit. Denote the value of digit i (radix $i+1$) as $d_i \in [0, i]$. Adding a carry in to the digit yields $d_i + 1$. If $d_i + 1 \leq i$ then that is the new value of of d_i and the carry out propagation stops. Otherwise the new value of d_i is zero and proceed to digit $i + 1$.

Code for computing the next permutation is shown in module `perm_behavioral`. That module also shows how to apply a factorial number (`pnum_in`) to permute items in `pdata_in` and connect them to `pdata_out`.

Testbench

To compile your code and run the testbench press $\boxed{\texttt{F9}}$ in an Emacs buffer in a properly set up account. The testbench will apply inputs to several instantiation of module `perm`. The instantiations differ on the number of items to permute, `n`, and the number of bits in each item, `w`. The testbench shows sample outputs and errors, and ends with a tally of errors for each instantiation. The output for an unmodified assignment includes:

```
Starting tests for w=8, n=3
Trace of permutation: 0 0 0  -> a b c
Error in next index:  0 0 0  -> 0 0 0   !=  0 1 0  (correct)
Error in permutation: 0 1 0  -> a b c   !=  a c b  (correct)
Error in next index:  0 1 0  -> 0 1 0   !=  1 0 0  (correct)
[snip]

Finished with n=10, 999 perm errors, 1000 next idx errors in 1000 tests.
End of tests n=3, 5 perm errors, 6 next idx errors for 6 tests.
End of tests n=4, 23 perm errors, 24 next idx errors for 24 tests.
End of tests n=8, 999 perm errors, 1000 next idx errors for 1000 tests.
End of tests n=10, 999 perm errors, 1000 next idx errors for 1000 tests.
xmsim: *W,RNQUIE: Simulation is complete.
```

In the unmodified assignment `perm` connects the permutation (`pdata_out`) output to the permutation input (`pdata_in`), which is wrong except for the identity permutation. That's why each module gets one permutation correct, as can be seen in the output above. (For example, for $n = 3$, 5 perm errors out of $3! = 6$ tests.)

The testbench always shows the first few outputs of each instance. For a correct assignment the output would include:

```
Starting tests for w=8, n=3
Trace of permutation: 0 0 0  -> a b c
Trace of permutation: 0 1 0  -> a c b
Trace of permutation: 1 0 0  -> b a c
Trace of permutation: 1 1 0  -> b c a
Trace of permutation: 2 0 0  -> c a b
Trace of permutation: 2 1 0  -> c b a
Finished with n=3, 0 perm errors, 0 next idx errors in 6 tests.
Starting tests for w=7, n=4
Trace of permutation: 0 0 0 0  -> a b c d
Trace of permutation: 0 0 1 0  -> a b d c
Trace of permutation: 0 1 0 0  -> a c b d
Trace of permutation: 0 1 1 0  -> a c d b
```

For $n = 3$ the testbench sets `pdata_in[0]='c'`, `pdata_in[1]='b'`, and `pdata_in[2]='a'`. The module needs to work for any settings for `pdata_in`, but the testbench sets `pdata_in` to values in `a,b,c,..` to make debugging easy.

Testbench output starting `Trace of permutation` shows the value of `pnum_in` (index, a factorial number) and `pdata_out` when `pdata_out` is correct. For the $n = 3$ module notice that all 6 permutations (permuted inputs) are shown, such as `a b c`. The sample above also shows the first few outputs of the $n = 4$ instance.

The digits of the permutation number are separated by spaces. The leftmost digit is (of course) the most significant, at position `n-1`. Being a permutation number, the least significant digit is always zero.

For each instance the first permutation is always identity (`pnum_in=0`), and the first 5 permutations are shown. For instances where $n! \leq 1000$ (or the value of `max_tests`) all permutations are tried. Otherwise, after showing 5 consecutive permutations a new random permutation is chosen. That can be seen below.

```
Starting tests for w=8, n=8
Trace of permutation: 0 0 0 0 0 0 0 0  -> a b c d e f g h
Trace of permutation: 0 0 0 0 0 0 1 0  -> a b c d e f h g
Trace of permutation: 0 0 0 0 0 1 0 0  -> a b c d e g f h
Trace of permutation: 0 0 0 0 0 1 1 0  -> a b c d e g h f
Trace of permutation: 0 0 0 0 0 2 0 0  -> a b c d e h f g
Trace of permutation: 5 2 3 2 1 0 0 0  -> f c e d b a g h
Trace of permutation: 5 2 3 2 1 0 1 0  -> f c e d b a h g
Trace of permutation: 5 2 3 2 1 1 0 0  -> f c e d b g a h
```

If there is an error the provided and correct outputs are shown. Here again is the output from the unmodified assignment:

```
Starting tests for w=8, n=3
Trace of permutation: 0 0 0  -> a b c
Error in next index:  0 0 0  -> 0 0 0   !=  0 1 0  (correct)
Error in permutation: 0 1 0  -> a b c   !=  a c b  (correct)
Error in next index:  0 1 0  -> 0 1 0   !=  1 0 0  (correct)
```

The first permutation output, `a b c`, is correct. The `pnum_out` value is wrong, that's shown in the line starting `Error in next index`. That line shows the value of `pnum_in` (the factorial number) to the left of the `->` and the value of `pdata_out` to the right of `->`. The correct value is shown to the right of `!=`. Similar information is shown for incorrect permutations.

Helpful Examples

An example that might help in computing `pnum_out` is from the class notes on generate statements. Module `ripple_w_r` recursively implements an adder. Pay attention to how `bfa` computes the LSB of the sum, and the recursive instance computes the remaining bits:

```
module ripple_w_r #( int w = 16 )
   ( output uwire [w-1:0] sum,   output uwire cout,
     input uwire [w-1:0] a, b,   input uwire cin);
   uwire c;

   // Instantiate a BFA to handle least-significant bit.
   //
   bfa bfa( sum[0], c, a[0], b[0], cin );

   if ( w == 1 )
     // If just one bit, we're done.
```

```
    //
    assign cout = c;
  else
    // Recursively instantiate this module to handle remaining bits.
    //
    ripple_w_r #(w-1) r(sum[w-1:1], cout, a[w-1:1], b[w-1:1], c);
endmodule
```

There's no need to use a BFA for this assignment. Use continuous assignments or procedural code to compute one digit of `pnum_out`.

In most examples of where we recursively describe a module we omit a particular bit (bit 0 in the ripple adder example above) in the connection to the recursive instance, or we have two recursive instances, each connected to half the inputs. The `perm` module is different because the digit to omit depends on the `pdata_in`. So, we need to use procedural code to compute an input to the recursive module and there are no good past assignment that do that. The closest is the Batcher merge module from 2018 Homework 5 where the odd and even elements of each of two the inputs were separated and recombined as inputs to two recursive instances:

```
module batcher_merge #( int n = 4, int w = 8 )
  ( output uwire [w-1:0] x[2*n], input uwire [w-1:0] a[n], b[n] );
  uwire [w-1:0] xlo[n], xhi[n];

  if ( n == 1 ) begin
     assign xlo[0] = a[0];
     assign xhi[0] = b[0];
  end else begin

     localparam int nh = n/2;
     uwire [w-1:0] ae[nh], ao[nh], be[nh], bo[nh];
     for ( genvar i=0; i<nh; i++ )
       begin
          assign ae[i] = a[2*i];
          assign ao[i] = a[2*i+1];
          assign be[i] = b[2*i];
          assign bo[i] = b[2*i+1];
       end

     batcher_merge #(nh,w) mlo( xlo, ae, bo );
     batcher_merge #(nh,w) mhi( xhi, ao, be );
  end

  for ( genvar i=0; i<n; i++ )
    sort2 #(w) s2( x[2*i], x[2*i+1], xlo[i], xhi[i] );
endmodule
```

Problem 1: Module `perm` has two data inputs, `pdata_in` and `pnum_in`. Input `pdata_in` is an array of `n` items, each `w` bits wide, where `n` and `w` are module parameters. Input `pnum_in` is an `n`-element array of `dw`-bit digits, where `dw` is a parameter. Module `perm` has three outputs, `pdata_out`, `pnum_out`, and `carry_out`. Like `pdata_in`, output `pdata_out` is an `n`-element array of `w`-bit items, and like `pnum_in`, output `pnum_out` is an `n`-element array of `dw`-bit digits. Output `carry_out` is one bit.

Output `pdata_out` is to be set to a permutation (rearrangement) of the elements of `pdata_in`. Suppose `pdata_in = {a,b,c}` (which means $n = 3$, and perhaps $w = 8$) and that the elements of `pdata_in` and `pdata_out` are ASCII characters. Then valid outputs could be `pdata_out = {a,b,c}`, `pdata_out = {b,a,c}`, etc. An invalid output would be `pdata_out = {a,a,c}`, it's invalid because `a` appears twice and `b` does not appear.

Input `pnum_in`, a factorial number, specifies how `pdata_in` should be permuted. A permutation is constructed iteratively, starting from the most-significant digit of `pnum_in`, which is `pnum_in[n-1]` and specifies where the value of `pdata_out[n-1]` should be drawn from. The Verilog code below (also part of the assignment file) shows how `pdata_out` is computed:

```
module perm_behavioral
  #( int w = 8, n = 20, dw = $clog2(n) )
   ( output logic [w-1:0] pdata_out[n], output logic [dw-1:0] pnum_out[n],
     output logic carry_out,
     input uwire [w-1:0] pdata_in[n], input uwire [dw-1:0] pnum_in[n] );

   always_comb begin

      pdata_out = pdata_in;

      for ( int i=n-1; i>0; i-- ) begin
         automatic logic [dw-1:0] pos = i-pnum_in[i];
         automatic logic [w-1:0] x = pdata_out[pos];
         for ( int j=pos; j<i; j++ ) pdata_out[j] = pdata_out[j+1];
         pdata_out[i] = x;
      end
   end
```

Notice that `pdata_out` is written multiple times, each iteration of the `i` loop permanently writes `pdata_out[i]` but also changes some other elements.

(*a*) Add code to `perm`, including recursive instantiation, so that `pdata_out` is a permutation of `pdata_in` as specified by `pnum_in`. Module `perm` with parameter `n>1` must recursively instantiate itself and the module must be synthesizable. Use command `genus -files syn.tcl` to synthesize.

(*b*) Add code to `perm`, including recursive instantiation, so that output `pnum_out` is the factorial number that follows `pnum_in`. The module must be synthesizable. As a reference the Verilog code below computes the next factorial number.

```
   always_comb begin
      // Compute next factorial (permutation) number.
      carry_out = 1;
      for ( int i=0; i<n; i++ ) begin
         automatic int radix = i + 1;
         automatic logic [dw:0] next_val = pnum_in[i] + carry_out;
         if ( next_val < radix ) begin
            pnum_out[i] = next_val;
            carry_out = 0;
         end else begin
            pnum_out[i] = 0;
         end
      end
   end
```