LSU EE 4755 Homework 5 Due: 27 November 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/hw05.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

In previous assignments there were modules that permuted their inputs, the one called `pdata_in`. What would happen if the `pdata_in` input to our permutation module, `perm`, did not consist of n distinct elements? Let's suppose there would be some dire consequences that we need to avoid. That's what this assignment is about, module `uniq_vector_seq` will be used to determine if elements are distinct.

Module `uniq_vector_seq` has one `we`-bit data input, `element`, where `we` is a module parameter. There are two outputs, `n`-bit output `uniq_bvec` and `wc`-bit output `n_match`, where `n` and `wc` are module parameters. There is also a 1-bit input `start`.

At each positive clock edge a new element will be placed on input `element`. The module output `uniq_bvec` indicates the elements arriving in the prior n cycles (or since the last `start`) that appear only once. An element that appears only once is called *unique*. Output `uniq_bvec` (unique bit vector) has one bit for each of the past n cycles, with the least significant bit corresponding to the previous cycle. Let t denote the current cycle and let e_t denote the element at the `element` input in cycle t. The previous cycle is $t-1$, the one before that is $t-2$, and so on. (If this is starting to get confusing look at the examples in the description of the testbench.)

First, consider the case where `start=0` for at least the last n cycles. If `uniq_bvec[i]` is `1` then e_{t-i-1} is unique, meaning that $e_{t-i-1} \neq e_{t-j-1}$ for $i, j \in [0, n-1]$ and $i \neq j$. If `uniq_bvec[i]` is `0` then $e_{t-i-1} = e_{t-j-1}$ for some $i \neq j$.

For example, suppose $n = 4$ and suppose the most recent elements are 4, 7, 5, 5, with 4 the least recent of those. Then `uniq_bvec` will be 1100_2 because 5 appears twice. For 7, 7, 2, 2 `uniq_bvec` will be 0000_2, for 4, 7, 2, 0 `uniq_bvec` will be 1111_2, and finally for 3, 7, 7, 0 `uniq_bvec` will be 1001_2. The testbench shows the recent elements and the provided (module output) and if different, the correct value of `uniq_bvec`.

Output `n_match` should be set to the number of elements that the most recent element matches, including itself. For 4, 7, 5, 5 `n_match=2`, for 4, 7, 2, 0 `n_match=1` and for 9, 0, 9, 9 `n_match=3` but for 8, 8, 8, 6 `n_match=1`.

In a cycle where `start=1` the element on `element` starts a new sequence. So for the purposes of computing `uniq_bvec` and `n_match` `element` is considered not equal to any element that arrived in a previous cycle.

Testbench

To compile your code and run the testbench press F9 in an Emacs buffer in a properly set up account. The testbench will apply inputs to several instantiation of module `uniq_vector_seq`. The instantiations differ in `n` and in whether the `start` signal will be set to `1` during testing.

The testbench will always show information about at least 5 (or the value of `trace_len`) sets of inputs for each instantiation. If there are errors it will show information on at least 4 inputs that generate each kind of error.

Here is sample testbench output from a working module:

```
** Starting tests for n=4, input start used = No **
Trace, uniq_bvec: t=33,  1001
[  ,   ],  1,  0,  0,  1  <-- uniq_bvec
[13,  8],  7,  9,  9,  4  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
Trace, uniq_bvec: t=34,  0011
[  ,   ],  0,  0,  1,  1  <-- uniq_bvec
[ 8,  7],  9,  9,  4, 14  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
Trace, uniq_bvec: t=35,  1100
[  ,   ],  1,  1,  0,  0  <-- uniq_bvec
[ 7,  9],  9,  4, 14, 14  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
```

The text above shows information on three inputs, they occur at $t = 33$ through $t = 35$. (Actually those numbers refers to test numbers, not cycles.) The rows labeled `Elements` show the elements that have arrived over the past six cycles. The rightmost one is the most recent. The output above was for a module instantiated with `n=4`, so only the last 4 elements should matter. As an aid in debugging two additional elements are shown. So, for $t = 33$ the module should only pay attention to elements 7, 9, 9, 4, and the module should ignore 13, 8. The value of `uniq_bvec` is shown on the lines that start with `Trace`. The same value is shown in the rows labeled `uniq_bvec`. Note that the value `1001` is the output at $t = 33$. But the values shown in the `Element` and `Start` rows are from the past $n + 2$ cycles.

Notice that a bit of `uniq_bvec` is 0 if the corresponding element appears more than once. That is the case for 9 in the $t = 33$ input. At $t = 35$ the 9 element becomes uniq because the other 9 has arrived more than n cycles ago. For the examples above `n_matches` should be 1 at $t = 33$ and $t = 34$ and 2 at $t = 35$ (because there are two 14s).

The `start` input is used to reset the module. When `start=1` the prior elements are forgotten or ignored. The output below shows the correct effect of start.

```
** Starting tests for n=6, input start used = Yes **
Trace, uniq_bvec: t=53,  000111
[  ,   ],  0,  0,  0,  1,  1,  1  <-- uniq_bvec
[13, 99],  1,  1,  1, 19, 95, 53  <-- Element
[ 0,  0],  0,  0,  0,  0,  0,  0  <-- Start
Trace, uniq_bvec: t=54,  111111
[  ,   ],  1,  1,  1,  1,  1,  1  <-- uniq_bvec
[99,  1],  1,  1, 19, 95, 53, 19  <-- Element
[ 0,  0],  0,  0,  0,  0,  0,  1  <-- Start
```

```
Trace, uniq_bvec: t=55,  111111
[  ,   ],  1,  1,  1,  1,  1,  1  <-- uniq_bvec
[ 1,  1],  1, 19, 95, 53, 19, 32  <-- Element
[ 0,  0],  0,  0,  0,  0,  1,  0  <-- Start
```

At $t = 53$ in the output above there are three elements equal to 1, setting the `uniq_bvec` bits to zero. A $t = 54$ the `start` input is asserted and so the positions with 1 elements become unique. Also arriving elment 19 is also considered uniq. If in $t = 56$ a 19 arrived then the 19 elements would no longer be unique.

Here is testbench output for a module with errors:

```
** Starting tests for n=4, input start used = No **
Error, uniq_bvec: t=9,  0100!= 1100 ( correct )
[  ,   ], E0,  1,  0,  0  <-- uniq_bvec
[ 9,  9],  9,  2, 13, 13  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
Error, uniq_bvec: t=11,  0001!= 0011 ( correct )
[  ,   ],  0,  0, E0,  1  <-- uniq_bvec
[ 9,  2], 13, 13,  2,  1  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
```

The `Error` line shows first the value of `uniq_bvec` exiting the module, and then the correct value. The `uniq_bvec` line shows the value from the module, preceded with an `E` if that value is wrong. At $t = 9$ the MSB should have been a 1 because 9 is unique. Perhaps it is being dubbed not unique because there was another 9 earlier, but that should be too early to matter. A common mistake is to leave an output unconnected. The value would be shown as `x`, say `Ex` for a `uniq_bvec` bit.

Error lines are also shown if `n_match` is wrong:

```
** Starting tests for n=4, input start used = No **
Trace, uniq_bvec: t=10,  0000
Error:  n_match: 1 != 2 (correct)
[  ,   ],  0,  0,  0,  0  <-- uniq_bvec
[ 9,  9],  7, 13, 13,  7  <-- Element
[ 0,  0],  0,  0,  0,  0  <-- Start
```

In the output above `n_match` should have been 2 (since element 7 appears twice), but the module output is 1.

The testbench checks instantiations with two values of `n`, and does one set of tests where `start` is always 0 (after initialization) and another set of tests where `start` is occasionally set to 1.

At the end of the testbench a summary of error counts is printed:

```
End of tests n= 4, s=0: 0 bvec errors, 33368 n_match errors for 99992 tests.
End of tests n= 4, s=1: 0 bvec errors, 19771 n_match errors for 99992 tests.
End of tests n= 6, s=0: 0 bvec errors, 18240 n_match errors for 99988 tests.
End of tests n= 6, s=1: 0 bvec errors, 9326 n_match errors for 99988 tests.
xmsim: *W,RNQUIE: Simulation is complete.
```

The output above shows lots of `n_match` errors but no `bvec_uniq` errors.

Helpful Examples

The demo module computing a running sum will probably be most helpful. That and other pipelined modules are in file `pipe.v` in the homework file and can be viewed, with images, at `https://www.ece.lsu.edu/koppel/v/2023/pipe.v.html`. Look for module `simple_pipe_avg`.

Problem 1: In the unmodified assignment module `uniq_vector_seq` has some starter code, and it will actually generate the correct outputs for the `start=0` tests. It does so using combinational module `uniq_vector_comb`. The problem with the combinational module is that it is too costly, and also slow. Also, it ignores the `start` signal. So for this problem remove the instantiation of `uniq_vector_comb` from `uniq_vector_seq` and complete `uniq_vector_seq` so that it operates as described above. It is important that the cost is reasonable. The reason that `uniq_vector_comb` is costly is that it does n^2 comparisons. Module `uniq_vector_seq` should only perform about n comparisons per clock cycle.

(*a*) Add code to `uniq_vector_seq` so that `n_match` works as described above. The code must be synthesizable. Use command `genus -file syn.tcl` to synthesize. This part is easy.

- ☐ Complete module so that `n_match` is correct.
- ☐ Follow the checkbox items in `hw05.v`.

(*b*) Add code to `uniq_vector_seq` so that `uniq_bvec` works as described above. This is trickier, at least for a low-cost solution. It might be easier to get the `start=0` version working first.

- ☐ Complete module so that `uniq_bvec` is correct.
- ☐ Pay attention to cost, cost should not be proportional to n^2.
- ☐ Follow the checkbox items in `hw05.v`.