LSU EE 4755 Homework 5 Due: 10 November 2017

For instructions visit `http://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab visit* `http://www.ece.lsu.edu/koppel/v/2017/hw05.v.html`.

Problem 1: Module `lookup_char` has a w-bit input `char`, and two outputs, `found` and `idx`. The module has parameter `chars`, which is an n-element array of w-bit characters. Complete `lookup_char` so that output `found` is logic `1` iff `char` is equal to one of the elements of `chars`. Set `idx` to the index of that character. (That is, if `found` is `1` then `chars[idx] == char`.) It does not matter what `idx` is if the character is not found. The module should synthesize into combinational logic.

See the Verilog Problem 1 code for details on the parameters and ports and review the comment checkboxes at the top of the problem for additional tips.

Module `lookup_char` will be used in the next problem and the testbench will be able to test `lookup_char` even if no other parts of `nest` are finished.

Note: There is a 2016 EE 4755 homework assignment in which a module a lot like `lookup_char` had to be designed. The major difference is that in 2016 the `chars` array was a port, here it is a parameter. Feel free to look at the solutions. It should go without saying that the `chars` array should remain a parameter in this assignment.

Problem 2: Module `nest`, when completed, will monitor a sequence of characters that includes bracketing characters (such as parentheses) and indicate whether these bracketing characters are properly nested. For example, sequence "a()[d()e[f]]" is properly nested but "a(]" is not.

The module has input parameters `char_open` and `char_close`, each of these is an n-element array of w-bit characters listing characters that are to be treated as opening and closing bracketing characters. See the Verilog code for details. The module has three inputs, `clk`, `reset`, and `in_char`. The module has five outputs, `level`, `awaiting`, `is_open`, `is_close`, and `bad`.

Output `is_open` should be set to `1` iff `in_char` is one of the characters in `char_open`, and `is_close` should be set to `1` iff `in_char` is one of the characters in `char_close`. These outputs should be generated by instantiations of `lookup_char` (the module from the first problem). The logic for computing `is_open` and `is_close` should be combinational.

(*a*) Complete the logic for `is_open` and `is_close` as described above. The testbench checks these outputs for correctness, look for `op` and `cl` in the trace. They are correct if `er` does not appear to the right of the `0` or `1`. The module must be synthesizable.

The module has an output `level` which should operate as follows. On a positive clock edge if `reset` is `1`, `level` is set to zero. Otherwise, if `in_char` is in `char_open` then `level` should be incremented and if `in_char` is in `char_close` then `level` should be decremented. If `in_char` is in neither list then `level` is left unchanged. (`level` provides the current nesting level. A value of 0 indicates the current character is not "inside" any bracketing characters, or put another way, that we are not awaiting something like a closing parenthesis.)

The module has an output `bad` which indicates whether the sequence seen since the last reset is improperly nested or if the nesting level exceeded `d`, a module parameter. Set `bad` to 0 when `reset` is 1 (at a positive clock edge). Set `bad` to 1 if a closing character is seen when `level` is 0 or if an opening character is seen when `level` is `d`.

Also set `bad` to 1 if the wrong closing character is seen. For example, for "(]" set `bad` to 1 when the "]" is seen because a ")" was expected.

Output `awaiting` should be set to the next valid closing character. For example, if the sequence so far is "()[" `awaiting` should be set to "]".

When `bad` is `1` outputs `level` and `awaiting` can be set to any value.

Note that `bad`, `level`, and `awaiting` should be updated at the positive clock edge.

(*b*) Complete `nest` so that it works as described above. The module must be synthesizable and show no errors.

The testbench checks `nest` for correctness and at the end of a run it shows the number of errors. As of this writing it will test `nest` on 1000 different sequences, see variable `num_groups` in the testbench. It will print details on up to 2 sequences with zero errors and up to 3 sequences with at least one error. Feel free to edit the testbench to change these numbers.

Consider the following sample of testbench output:

```
ncsim> run

cyc    2  s.c  0. 0  i  op 0     cl 0     bad 0     lev  0  0     await ' )'
cyc    3  s.c  0. 1  J  op 0     cl 0     bad 0     lev  0  0     await ' )'
```

The text `cyc 2` indicates the cycle number. That can be used with SimVision or some other tool to locate the place in execution. The text `s.c 0. 1` shows the sequence number, 0, and the number of previous characters in the sequence, 1. Next shown is the character at `in_char`, `J` in cycle 3. The text `op 0 cl 0 bad 0` show the values of the `is_open`, `is_close`, and `bad` outputs that `nest` has produced. If these values are wrong then the text `er` appears to the right of the value. For example if the value at the `is_open` port were wrong the text would be `op 0 er` . Note that `bad 1` is fine but `bad 0 er` indicates that the `bad` port value is wrong. The text `lev 0 0` shows both the module `level` output (the first `0` here) and the known correct value (the second `0`). Finally, `await` shows the module output followed by the correct value. They are between quotes to make spaces and other non-printable characters obvious.

Note that when level is zero the value of await is irrelevant. Also, when `bad` is 1, the values of level and await are both irrelevant.

The example below shows the trace output when there are errors:

```
cyc   54  s.c  5. 0  L  op 0     cl 0     bad 0     lev  0  0     await ' )'
cyc   55  s.c  5. 1  (  op 1     cl 0     bad 0     lev  0  1 er  await ' )'
cyc   56  s.c  5. 2  (  op 1     cl 0     bad 0     lev  0  2 er  await ' )'
cyc   57  s.c  5. 3  q  op 0     cl 0     bad 0     lev  0  2 er  await ' )'
cyc   58  s.c  5. 4  Z  op 0     cl 0     bad 0     lev  0  2 er  await ' )'
cyc   59  s.c  5. 5  )  op 0     cl 1     bad 1 er  lev  7  1     await ' )'
cyc   60  s.c  5. 6  )  op 0     cl 1     bad 1 er  lev  6  0     await ' )'
```

At cycle 55 level should have been incremented for the "(", but it was not. Notice the `er` to the right of `lev`. Also, at cycle 59 the module set `bad` to 1 which is an error because the sequence has not violated any rules.

Problem 3: Run the synthesis script, using command `rc -files syn.tcl`. If it runs correctly, a file `spew-file.log` will be created which contains a timing report for a design. On paper or in comments in the submission file, indicate where the critical path is in your design.

Provide suggestions on making it faster, or explain what you actually did for a high clock frequency.