LSU EE 4755 Homework 3 Due: 20 Oct 2024 (evening)

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/2024/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`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

In string `((I am balanced))` the parentheses are balanced, but in `((Not balanced)` they are not because a closing parenthesis is missing and so an opening parenthesis is unmatched. The modules in this assignment examine a string containing parentheses and report the number of unmatched closing, `)`, and opening, `(`, parentheses. Both modules put these numbers on outputs `left_out_n_unmat_close` and `right_out_n_unmat_open`. The parentheses in `a(b)c` and `(()a)()` are correctly matched. The parentheses in `())`, `((())`, and `))((` are not. For inputs like `()`, `(())`, and `()()` both outputs should be zero. For `)` and `())` the number of unmatched closing parentheses is one and so output `left_out_n_unmat_close` should be `1` and output `right_out_n_unmat_open` should be `0`. See the testbench output for more examples.

In both modules, `pmatch_base` (Problem 1) and `pmatch_mark` (Problem 2), the string appears on input `str`. In `pmatch_mark` there is an additional output, `str_marked`, which should be set to a version of the string in which the correctly matched parentheses are replaced by angle brackets. For example, for input `str='(()'` the output shold be `str_marked='(<>'`. See the testbench output for more examples.

The modules each have parameter `n`. Input `str` and output `str_marked` are `n`-element unpacked (ordinary) arrays of 4-bit quantities. For convenience enumeration constants are defined for the characters used in this assignment:

```
typedef enum logic [3:0]
  { Char_Blank, Char_Dot,
    Char_Open, Char_Close,
    Char_Open_Okay, Char_Close_Okay } Char;
```

The input `str` can consist of any of the first four values. There is no distinction between `Char_Blank` and `Char_Dot`, they are stand-ins for arbitrary characters and neither affects paren-

thesis matching. The last two, `Char_Open_Okay` and `Char_Close_Okay` are to be used in Problem 2 for replacing properly matched parenthesis.

Those who are unsure of how to work with `str` or of just what the modules are supposed to do should examine modules `pmatch_comb_base` and `pmatch_comb_mark`. Module `pmatch_comb_base` will pass the testbench for Problem 1 (if it were renamed `pmatch_base`) and `pmatch_comb_mark` would pass the testbench for Problem 2 (if renamed).

These `comb` modules compute their results by scanning the string from left to right. In their synthesized hardware the critical path appears to be proportional to n, the string length. That's too long a critical path for our purposes. In Problems 1 and 2 this is to be overcome by using a recursive module structure that describes tree-like hardware. In a correct solution the critical path will be closer to $\log n$, and the cost will be lower too.

The synthesis output below shows the result of synthesizing the `comb` base module and a correct solution to Problem 1 at exponentially increasing string lengths ($n = 4,\ 8,\ 16,\ 32$). Notice that in the `comb` version the delay too increases exponentially (in proportion to n) while the delay in the Problem 1 solution increases more linearly. Absolute costs are lower too. The differences are less stark with a larger delay target. The default synthesis script uses the larger delay target to save time.

Module Name	Area	Delay Actual	Delay Target	Synth Time
pmatch_comb_base_n4_13	43915	0.82	0.1 ns	24 s
pmatch_comb_base_n8_29	175285	2.31	0.1 ns	185 s
pmatch_comb_base_n16_61	221959	6.99	0.1 ns	256 s
pmatch_comb_base_n32_125	771830	15.69	0.1 ns	772 s
pmatch_base_n4	22979	1.22	0.1 ns	15 s
pmatch_base_n8	73381	1.93	0.1 ns	50 s
pmatch_base_n16	142421	3.24	0.1 ns	86 s
pmatch_base_n32	341921	4.49	0.1 ns	179 s
pmatch_comb_base_n4	11039	2.14	900.0 ns	8 s
pmatch_comb_base_n8	33748	7.11	900.0 ns	77 s
pmatch_comb_base_n16	93278	18.98	900.0 ns	53 s
pmatch_comb_base_n32	248862	48.57	900.0 ns	117 s
pmatch_base_n4	15550	3.10	900.0 ns	6 s
pmatch_base_n8	41187	5.99	900.0 ns	5 s
pmatch_base_n16	98336	9.91	900.0 ns	18 s
pmatch_base_n32	216143	14.60	900.0 ns	23 s

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 instantiations of modules `pmatch_base` and `pmatch_mark`. The instantiations differ in the length of the string. At the end of execution the number of errors for each module at each size are shown. The output below is from a correctly solved assignment:

```
Total pmatch_base n=4: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_base n=5: Errors: 0 cl, 0 op, 0 mk.
```

```
Total pmatch_base n=7: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_base n=8: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_base n=9: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_base n=17: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=4: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=5: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=7: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=8: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=9: Errors: 0 cl, 0 op, 0 mk.
Total pmatch_mark n=17: Errors: 0 cl, 0 op, 0 mk.
```

Each line starting with `Total` shows a tally of results. After `Total` the line shows the module name, either `pmatch_base` or `pmatch_mark`, and `n`, the length of the string. A tally of each output's error is shown after `Errors:`, `cl` is the number of incorrect closing-parentheses values, `op` is the number of opening-parenthesis values, and `mk` is the number of incorrectly marked strings.

Further up, the testbench shows sample output and error details. For each instantiation the first `n_errors_show = 5` incorrect outputs are shown on lines starting with `Error`. If it would help to see more then feel free to search for `n_errors_show` and increase the value. In addition the details of the first `n_samples_show = 6` correct outputs are shown on lines starting with `Sample`. The output below shows correct outputs:

```
Starting pmatch_base tests for n=5.
Sample pmatch_base n=5 '()   ': close = 0,  open = 0 (both correct)
Sample pmatch_base n=5 '.( ) ': close = 0,  open = 0 (both correct)
Sample pmatch_base n=5 ')(   ': close = 1,  open = 1 (both correct)
Sample pmatch_base n=5 ')    ': close = 1,  open = 0 (both correct)
Sample pmatch_base n=5 '))   ': close = 2,  open = 0 (both correct)
Sample pmatch_base n=5 '()) ': close = 2,  open = 0 (both correct)
Sample pmatch_base n=5 '())( ': close = 1,  open = 1 (both correct)
Sample pmatch_base n=5 '())((': close = 1,  open = 2 (both correct)
Sample pmatch_base n=5 '))(( ': close = 2,  open = 2 (both correct)
```

In the sample above the first `Sample` line indicates that for input `()` both the `left_out_n_unmat_close` and `right_out_n_unmat_open` outputs were 0, which is correct because there were no unmatched parentheses. The second sample is also properly matched. It consists of a dot (which is just an ordinary character), parentheses, and spaces. The third sample has both one unmatched opening parenthesis and an unmatched closing parenthesis.

The testbench starts applying patterns found in `str_special`. Feel free to add your own to help with debugging. After the patterns in `str_special` are used the testbench will make up random patterns.

The output below is of a run using a partially correct `pmatch_base`:

```
Sample pmatch_base n=5 '()   ': close = 0,  open = 0 (both correct)
Sample pmatch_base n=5 '.( ) ': close = 0,  open = 0 (both correct)
Sample pmatch_base n=5 ')(   ': close = 1,  open = 1 (both correct)
Sample pmatch_base n=5 ')    ': close = 1,  open = 0 (both correct)
Error pmatch_base n=5 '))   ': close  0 != 2 (correct)
Sample pmatch_base n=5 '()) ': close = 2,  open = 0 (both correct)
Error pmatch_base n=5 '))(( ': close  0 != 2 (correct)
Error pmatch_base n=5 '))   ': close  0 != 2 (correct)
```

The errors reported above seem to show that the `left_out_n_unmat_close` is wrong when the string starts with two consecutive closing parentheses.

The `str_marked` output of module `pmatch_mark` is supposed to show the string with the correctly matched parentheses replaced by angle brackets (actually less-than and greater-than symbols). Appearing below is sample output of the module in a correctly solved assignment. Two lines are used to show the result of each input. The first shows the input string, such as `()` in the first sample, the second line shows the marked string, such as `<>`.

```
Sample pmatch_mark n=5 '()   ': close = 0,  open = 0 (both correct)
Sample pmatch_mark n=5 '<>   ' (marked_outpuut)
Sample pmatch_mark n=5 '.( ) ': close = 0,  open = 0 (both correct)
Sample pmatch_mark n=5 '.< > ' (marked_outpuut)
Sample pmatch_mark n=5 ')(   ': close = 1,  open = 1 (both correct)
Sample pmatch_mark n=5 ')(   ' (marked_outpuut)
Sample pmatch_mark n=5 ')    ': close = 1,  open = 0 (both correct)
Sample pmatch_mark n=5 ')    ' (marked_outpuut)
Sample pmatch_mark n=5 '))   ': close = 2,  open = 0 (both correct)
Sample pmatch_mark n=5 '))   ' (marked_outpuut)
Sample pmatch_mark n=5 '()))' ': close = 2,  open = 0 (both correct)
Sample pmatch_mark n=5 '<>)) ' (marked_outpuut)
Sample pmatch_mark n=5 '())( ': close = 1,  open = 1 (both correct)
Sample pmatch_mark n=5 '<>)( ' (marked_outpuut)
Sample pmatch_mark n=5 '())((': close = 1,  open = 2 (both correct)
Sample pmatch_mark n=5 '<>)((' (marked_outpuut)
Sample pmatch_mark n=5 '))(( ': close = 2,  open = 2 (both correct)
Sample pmatch_mark n=5 '))(( ' (marked_outpuut)
```

The output below is of a run using an incorrect `pmatch_mark` module. An error line is printed for each incorrect output, `left_out_n_unmat_close`, `right_out_n_unmat_open`, and `str_marked`. For a particular input, say `()`, a module can have one incorrect output, such as `left_out_n_unmat_close`, while the other two outputs, `right_out_n_unmat_open` and `str_marked` are correct. That's the case in the first and last error below:

```
Starting pmatch_mark tests for n=5.
Error pmatch_mark n=5 '()   ': close  1 != 0 (correct)
Error pmatch_mark n=5: '<<>>)' != '<>   ' (correct)
Error pmatch_mark n=5 '.( ) ': open   3 != 0 (correct)
Error pmatch_mark n=5: '..(((' != '.< > ' (correct)
Error pmatch_mark n=5 ')(   ': close  2 != 1 (correct)
```

References and Helpful Examples

The modules in this assignment must be recursively defined, so that they describe a tree-like structure. See the CLZ module from 2019 Homework 2. The assignment, solution, and live version done in class are part of the 2024 assignment directory, look for the file names starting `2019`. Understanding the `clz_tree_fat` solution is sufficient. The trick used to avoid the adder in the clz module is not relevant to this 2024 assignment.

Problem 1: Module `pmatch_base` has one input, `str`, and two outputs `left_out_n_unmat_close` and `right_out_n_unmat_open`, and parameters `n` and `wn`. Input `str` is an `n`-element array of 4-bit quantities called characters, with `str[0]` being the leftmost character and `str[n-1]` being the rightmost character. Outputs `left_out_n_unmat_close` and `right_out_n_unmat_open` are each `wn` bits. They should be set to the number of unmatched parentheses as described in the introduction to the assignment represented as an unsigned integer. The default value of `wn`, $\lceil \lg(n+1) \rceil$, is the minimum value needed to correctly report `n` mismatched parentheses. (Setting `wn` to a larger values is a potential waste.)

Complete the module so that it produces these outputs and so that it describes tree-structured hardware by using recursion. The critical path should be proportional to $\log n$, which can be achieved by splitting the `str` input between two recursive instantiations and combining their outputs. It may help to examine `pmatch_comb_base`, which produces the same outputs, though not recursively.

- [ ] Complete the module so that the testbench reports zero errors.
- [ ] The module description must be recursive and describe tree-like hardware.
- [ ] Set `wn` to the smallest correct value in the recursive instantiations.
- [ ] Make sure the module is synthesizable using command `genus -files syn.tcl`.

Hint: Consider checking `str` *only in the base (terminal) case of the recursion.*

Problem 2: Module `pmatch_mark` has three inputs, `str`, `left_in_n_unmat_open`, `right_in_n_unmat_close`, and three outputs `left_out_n_unmat_close`, `right_out_n_unmat_open`, `str_marked`, and parameter `n`. Input `str` and output `str_marked` are `n`-element 4-bit arrays. Input `str` carries a string, and output `str_marked` is to be set to a version of the input string with each properly matched `Char_Open` replaced with a `Char_Open_Okay` and each properly matched `Char_Close` replaced with a `Char_Close_Okay`.

Input `left_in_n_unmat_open` is set to the number of unmatched opening parentheses to the left of `str`. For example, suppose `str='))'` and `left_in_n_unmat_open=1`. Then that means that one of the unmatched parentheses in `str` is matched by something to the left of `str`. For this example, the value of `str_marked='>)'` because `left_in_n_unmat_open=1`. Similarly input `right_in_n_unmat_close` is set to the number of unmatched closing parentheses to the right of the string.

The testbench will always set `left_in_n_unmat_open` and `right_in_n_unmat_close` to zero. Your module should set them correctly in connections to recursive instantiations so that they can determine which of their parentheses are matched.

Outputs `left_out_n_unmat_close`, `right_out_n_unmat_open` should have the same values as they would in Problem 1. That is, they should show the number of unmatched opening and closing parentheses in `str` *even if those parentheses are marked as matched due to values of* `left_in_n_unmat_open` *and* `right_in_n_unmat_close`. For example, suppose `str='))))()('` and `left_in_n_unmat_open=2` and `right_in_n_unmat_close=1`. The module should set output `left_in_n_unmat_open=3` (ignoring the 2 matches) and `right_in_n_unmat_close=1` (ignoring the one match), and set output `str_marked='>>)<><'`, showing the matched parentheses. Parentheses are set as matched both when they are matched by parentheses within `str` and when they are matched by parentheses reported by `left_in_n_unmat_open` and `right_in_n_unmat_close`.

- [ ] Complete the module so that the testbench reports zero errors.
- [ ] The module description must be recursive and describe tree-like hardware.

- [ ] Set `wn` to the smallest correct value in the recursive instantiations.
- [ ] Make sure the module is synthesizable using command `genus -files syn.tcl`.

Hint: Consider writing `str_marked` *only in the base (terminal) case of the recursion.*