LSU EE 4755 Homework 1 Due: 21 September 2022

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/2022/hw01.v.html`*.*

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw01.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Background

The goal of this homework assignment and follow-on assignments is to convert an ASCII string into a number. For example, to convert `"12"` (equivalently `'h3132`) into 12 (equivalently `'b1100` or `'d12` or `'hc`). An ASCII string is a sequence of bytes, but in this assignment there is just one byte. The follow-on assignments there will be multiple bytes.

The input to the module for this assignment, `atoi1`, is the character. The module has two outputs, the value, `val`, and whether the character is a valid digit. For example, `"1"` is a valid digit, but `"#"` is not.

The module has a parameter `r` which indicates the radix of the number that's expected. If `r=2` and the character is `"3"` then it is not a valid digit and the returned value should be zero. Further details are provided in the problem description below. For `r=16` the valid characters are 0 to 9, A to F, and a to f, with a and A, b and B, ... treated equivalently. The module should work for any `r` up to 36. As of this writing the testbench evaluates radices 4, 8, 10, 14, 16, 19. The TA-bot might test with different radices. Feel free to modify the testbench to try different radices. (Search for `testbench` and figure out the code.)

This assignment exercises basic Verilog skills like instantiating modules and understanding the difference between structural and procedural code. In the follow-on assignment the `atoi` modules will be connected to handle longer strings.

Testbench

To run compile your code and run the testbench press $\boxed{\texttt{F9}}$ in an Emacs buffer in a properly set up account. In an unmodified assignment the testbench will generate output that includes the following near the end:

```
Radix  4, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix  8, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix 10, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix 14, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix 16, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix 19, done with 256 tests, 0 val errors, 0 is_digit errors.
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
Total number of errors: 0
```

There are zero errors because the procedural code in `atoi1` is correct. Notice that there are separate tallies for each radix plus a grand total. Detailed messages are printed for the first few errors, after which only a tally is provided. For example, here is what the error messages would look like if the conversion to upper case were wrong:

```
xcelium> run
Radix  4, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix  8, done with 256 tests, 0 val errors, 0 is_digit errors.
Radix 10, done with 256 tests, 0 val errors, 0 is_digit errors.
R 14  Error val 'h0d or D != A (correct) for string " a"
R 14  Error val 'h00 or 0 != B (correct) for string " b"
R 14  Error is_digit 0 != 1 (correct) for string " b"
R 14  Error val 'h00 or 0 != C (correct) for string " c"
R 14  Error is_digit 0 != 1 (correct) for string " c"
R 14  Error val 'h00 or 0 != D (correct) for string " d"
R 14  Error is_digit 0 != 1 (correct) for string " d"
Radix 14, done with 256 tests, 4 val errors, 3 is_digit errors.
R 16  Error val 'h0d or D != A (correct) for string " a"
R 16  Error val 'h0e or E != B (correct) for string " b"
R 16  Error val 'h0f or F != C (correct) for string " c"
R 16  Error val 'h00 or 0 != D (correct) for string " d"
R 16  Error is_digit 0 != 1 (correct) for string " d"
R 16  Error is_digit 0 != 1 (correct) for string " e"
R 16  Error is_digit 0 != 1 (correct) for string " f"
Radix 16, done with 256 tests, 6 val errors, 3 is_digit errors.
R 19  Error val 'h00d or D != A (correct) for string " a"
R 19  Error val 'h00e or E != B (correct) for string " b"
R 19  Error val 'h00f or F != C (correct) for string " c"
R 19  Error val 'h010 or G != D (correct) for string " d"
R 19  Error is_digit 0 != 1 (correct) for string " g"
R 19  Error is_digit 0 != 1 (correct) for string " h"
R 19  Error is_digit 0 != 1 (correct) for string " i"
Radix 19, done with 256 tests, 9 val errors, 3 is_digit errors.
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
Total number of errors: 28
```

Consider one of those lines with some ASCII-art underlining:

```
R 14  Error val 'h0d or D != A (correct) for string " a"
0000            1111    2    3                       44  <- ASCII art underlining
```

The part underlined with `0000` indicates that this result is for radix `r=14`. The character to be converted is a, though it's called a string and shown with a leading space. That's the part underlined with `44`. The part underlined with `3` is what the value should be in radix 14, and the part underlined with `2` is what the `atoi1` module's `val` output is, in radix 14. The part underlined in `1111` is the module output in hexadecimal. In this case, the value should be 10 (decimal) or A (base 14), but the module output is D (which is 13 in decimal).

That's one error line. Going back to the more complete testbench output notice that only strings with lower-case letters are wrong. This is what one would expect since we intentionally broke the lower-to-upper conversion.

The testbench only shows details for the first 4 errors of each type at each radix. If you want to see more errors feel free to edit the testbench. Search for `err < 5`. Feel free to edit the testbench in other ways to facilitate debugging. The TA-bot will run your code using its own testbench, so don't worry about being accused of cheating by modifying the testbench.

Problem 1: Appearing below (and in the assignment file `hw01.v`) is module `atoi1`, *ASCII to Integer of 1 character.* The module has an 8-bit input `char`, a 1-bit output `is_digit`, and a w-bit output `val`. There are also two parameters, `w` (width) and `r` (radix). Output `is_digit` is set to `1` iff (if and only if) `str` is a radix-r digit. If `char` is a digit output `val` is set to its value, otherwise `val` is set to zero.

```
module atoi1 #( int r = 32, w = $clog2(r) )
   ( output logic [w-1:0] val,  output uwire is_digit,  input uwire [7:0] char );
   logic [7:0] char_uc;
   logic [w-1:0] val_09, val_az;
   logic is_09, is_az;

   digit_valid_09 #(r,w) v09( is_09, val_09, char );
   assign is_digit = is_09 || is_az;

   always_comb begin
      char_uc = char >= Char_a && char <= Char_z ? char - Char_a + Char_A : char;
      val_az = 10 + char_uc - Char_A;
      is_az =  char_uc >= Char_A   &&  char_uc < Char_A + r - 10;
      if ( is_09 )       val = val_09;
      else if ( is_az )  val = val_az;
      else               val = 0;
   end
endmodule
```

For example, suppose `w=4` and `r=10`. If `char=51` (ASCII for the digit 3), then output `val` is set to 3 and `is_digit` is set to 1. If `char=58` (ASCII for `:` [colon]), then output `val` is set to 0 and `is_digit` is set to 0. If `char=65` (ASCII for `A`), then output `val` is set to 0 and `is_digit` is set to 0. Now suppose that `atoi1` is instantiated with `r=16` (hexadecimal). If `char=65` (ASCII for `A`), then output `val` is set to 10 and `is_digit` is set to 1. If `char=97` (ASCII for `a`), then output `val` is also set to 10 and `is_digit` is set to 1.

Module `atoi1` includes an instantiation of module `digit_valid_09`, a continuous assignment (of `is_digit`), and procedural code. Module `digit_valid_09`, which is finished, converts an ASCII character into a value if the character is a digit from 0 to 9, and if the value is valid (less than `r`). (Those who are not sure what `digit_valid_09` is doing might want to inspect module `atoi1_behavioral`, which uses only procedural code.)

Make the following changes to `atoi1`: Instantiate module `char_to_uc` (character to upper case) and use it to convert `char` to upper case. Instantiate module `digit_valid_az` and use it to compute `is_az` and `val_az`. Instantiate `mux2` modules and use them to route the correct value to the `val` output of `atoi1`. As you instantiate and connect these modules remove the procedural code that's no longer needed.

Also, add code to `digit_valid_az` and `char_to_uc` so that they compute their proper values.

To help with debugging, do this in small steps. For example, first complete the `char_to_uc` module and make sure there are no compilation errors. Then instantiate it in `atoi1`, and make sure there are no compilation errors and no testbench errors.

Pay attention to compilation errors and ask for help with any that you can't understand.

The code must by synthesizable. To synthesize your code issue the command `genus -files syn.tcl`. If there are no errors, running this command will generate output that includes like the following:

```
Module Name                                 Area   Delay   Delay
                                                  Actual  Target
atoi1_r2                                    1796   0.454  10.000 ns
atoi1_behavioral_r2                         1796   0.454  10.000 ns
atoi1_r8                                    2047   0.495  10.000 ns
atoi1_behavioral_r8                         2047   0.495  10.000 ns
atoi1_r10                                   2517   0.529  10.000 ns
atoi1_behavioral_r10                        2517   0.529  10.000 ns
atoi1_r16                                   5792   0.752  10.000 ns
atoi1_behavioral_r16                        5792   0.752  10.000 ns
atoi1_r2_3                                  3754   0.274   0.100 ns
atoi1_behavioral_r2_4                       3754   0.274   0.100 ns
atoi1_r8_3                                  5762   0.260   0.100 ns
atoi1_behavioral_r8_4                       5762   0.260   0.100 ns
atoi1_r10_3                                 7371   0.259   0.100 ns
atoi1_behavioral_r10_4                      6937   0.260   0.100 ns
atoi1_r16_3                                18302   0.363   0.100 ns
atoi1_behavioral_r16_4                     18302   0.363   0.100 ns
```

The synthesis script is synthesizing both the module for this assignment, `atoi1`, and the behavioral version, `atoi1_behavioral`. The radix at which it is instantiated is appended to the name.