Verilog Reference Guide for EECS 470

Purpose:

This document is intended to provide a quick reference for writing Verilog code for homework assignments and the final project in EECS 470. This is not a general purpose reference on the Verilog language, and does not describe how Verilog works.

1) Verilog is "like" C

It is case sensitive, same comments (C++ style // and C style /*, please use //), and treats whitespace in the same way.
A few more things are allowed in identifiers, and there are different reserved words, but for the most part identifiers are the same
Many of the operators are the same as well, and many of the control structures look very similar, and in some circumstances C code and verilog code mean pretty much the same thing

2) Verilog is NOT "like" C

Verilog describes hardware, which is inherently parallel. Only in very limited circumstances is the sequential nature of C preserved. Be aware that for the most part, everything is happening in parallel and you are really describing hardware structures. Verilog supports lots and lots of things, but that does not mean that the tools support them or support them properly. Unlike C, Verilog is commonly compiled, synthesized, and simulated. Each use entails very different functionality so being in the language doesn't mean a whole lot.

3) Modules

The basic organizing unit in verilog. Think of these as the "boxes." They have inputs and outputs and internals that the outside world does not see.

Declaration (defines what one of these boxes looks like):

```
module my_simple_mux(select_in,a_in,b_in,muxed_out);
  // all inputs and outputs listed
  input select_in, a_in, b_in;
  output muxed_out;  // defaults to wire but can ovverride

  assign muxed_out=select_in?b_in:a_in;

endmodule
```

Two instantiations with the same meaning (makes the boxes inside a bigger box):

```
my_simple_mux m1(.a_in(a),.b_in(b),.select_in(s),.muxed_out(m));
// note that the order of the operands does not matter

my_simple_mux m1(s,a,b,m); // note the order of s (which matters)
```

4) Variables

Two basic kinds of "variables," wires and regs. Wires can have no state. They are always evaluated in terms of other values. They are highly restrictive and so are very safe to use. Regs can have state and almost anything can be done with them so they are more flexible and more dangerous. Any output can be redeclared as a reg (default is wire). All variables have a size in bits. Default is one bit, size is given as a range from one bit number to another. In a sense, many variables are like an array of bits. Either endianness is possible, we suggest big-endian (as in these examples).

```
wire [7:0] an_8bit_wire, another_8bit_wire;
// an_8_bit_wire[0] is the rightmost bit, an_8_bit_wire[1:0] is
// the two rightmost bits
reg a_single_bit_register;
```

5) Operators

`+, -, *, >, <, >=, <=, ==, !=, <<, >>, (), &, |, ~, ^, ?:` are all like C and can be used in expressions in both always blocks and in assign statements. A short description of the ones less common in C:

- `&` Bitwise and
- `|` Bitwise or
- `~` Bitwise negation (can generally be combined with another operator so ~& is bitwise nand
- `^` Bitwise xor
- `<<` Left shift
- `>>` Right shift
- `? :` `a?b:c` yields b if a is true and c if a is false (note that just like in C, 0 is false and 1 is true)

Additionally there are the `{}` operators

- `{a,b,c}` concatenation puts a,b,and c after one another into a single value
- `{n{m}}` makes a single value that is n copies of m one after the other

6) Literals

Verilog allows many different bases to represent literals. A literal is a size in bits followed by the ' character (the "right" tick which should be on the same key as ") followed by a base indicator (most common are b=binary, d=decimal, h=hex) followed by the value. Some of these parts are optional, but it is good to get into the practice of including all of the parts.

Examples:

- `8'b00001010` a byte with decimal value 10
- `32'd10` a 32-bit word with decimal value 10
- `8'ha` a byte with decimal value 10

In general, the value is 0-extended to the left to fit the specified size.

To make matters more complicated, bits can be more than 0 and 1. x means unknown and z means disconnected. In general, these are going to be bad values that you will see as output when something is broken. However, it is perfectly possible to introduce x values into your code on purpose to help the synthesis make faster circuits. When a combinational circuit has an x value as output, the synthesis will pick whatever output it thinks makes for a faster circuit. So, if some value is never possible as input, you might set the outputs to x for that output. This is not recommended until you are comfortable with Verilog however.

7) Macros

Like a #define in C, but all uses must use the ` character (the "left" tick, should be on the same key as ~)

Definition and use:

```
`define MY_CONSTANT 3'b010
a = b + `MY_CONSTANT;  // really a = b + 3'b010;
```

8) Debugging with $display and $monitor

$display is like printf in C, even with very similar format strings (but no need for a trailing \n, that is included). It displays something every time the statement is executed
$monitor sets up a background watcher on some signals and prints something every time one of them changes.
These can be very useful methods of debugging because the GUI we have to look at things is much slower than simply running your simulation from the command-line. They should be ignored by the synthesis because they don't really have anything to do with actual hardware, just the simulation.

examples:

```
$monitor ("At time %4.0f: r=%d, u_d=%d", $time, r, u_d);

$display ("Got to else part and the instruction is %h", inst)
```

Example State Machine Code

A simple state machine that counts from 0 to 1 to 2 then back to 0 with an asynchronous reset to 0.

```
`define SD #1

module three_state_counter(clock, reset, enable, current_state);

  input clock, reset, enable;
  output [1:0]current_state;
  reg [1:0]current_state_reg;
  reg [1:0]next_state;

  assign current_state=current_state_reg;

  always @(current_state_reg or enable)
  begin
    if (enable)
      begin
        case(current_state_reg)
          2'b00: next_state=2'b01;
          2'b01: next_state=2'b10;
          2'b10: next_state=2'b00;
          2'b11: next_state=2'b11;  // Should never happen after reset
        endcase
      end
    else
      next_state=current_state_reg;
  end

  always @(posedge clock or posedge reset)
    if(reset)
      current_state_reg <= `SD 2'b00;
    else
      current_state_reg <= `SD next_state;

endmodule
```