Synthesizable Verilog Guidelines for EECS 470

Purpose:

This document is intended to provide a quick reference for writing synthesizable 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. However, following the guidelines provided here should eliminate many problems and any deviation from these guidelines may lead to very strange and undesirable behavior from the tools we are using. If you encounter a strange problem, our first recommendation will be to look at these guidelines, and if we find a violation of these guidelines we will be very hesitant to investigate further.

Separate Sequential and Combinational Logic

Synthesizable verilog allows the design of hardware. Hardware registers and combinational logic are two very different things. The programming styles and guidelines for sequential and combinational logic are different but easily confused. Separating the sequential and combinational logic into different blocks allows one to check for errors much more easily.

There are two supported styles of combinational logic and one style for sequential logic (though you can think of it as two if you consider with asynchronous reset and without).

1) Sequential logic

All assignments use nonblocking <= operator with `SD

64bit register a_reg with input a (output is a_reg) no reset

```
reg [63:0] a_reg;

always @(posedge clock)
begin
  a_reg <= `SD a;
end
```

8 bit register b_reg with input b (output is b_reg) and asynchronous reset

```
reg [7:0]b_reg;

always @(posedge clock or posedge reset)
begin
  if(reset)
    b_reg <= `SD 0;
  else
    b_reg <= `SD b;
end
```

For the most part, we will not be using synchronous reset, and it can be considered part of the combinational logic.

2) Combinational logic via assign

- No always block.
- Outputs declared as wire
- Safest (no way to accidentally create state) due to limitations

1 bit wire do_something set to output of (do_a or do_b)

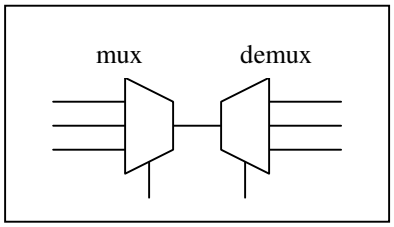
```
wire do_something;

assign do_something = do_a | do_b;
```

3) Combinational logic via always block

- All assignments use blocking = operator with NO ‘SD
- All inputs listed in always statement
- Output "wires" declared as reg (should synthesize to wires)
- All outputs must be assigned to on all paths through code
  (use of default initialization and/or defaults for cases recommended)
- Acts sequentially like C where the final assignment to an output is the output of the logic

This code routes one of 3 inputs (based on input_select) to one of 3 outputs (based on output_select) and puts a 0 on all non-selected outputs. When an invalid input is given an error output is set to 1. The circuit looks something like:
with the addition of some simple error detection.




```
‘define DEVA_ID 2’b00;
‘define DEVB_ID 2’b01;
‘define DEVC_ID 2’b10;

reg [63:0] deva_output, devb_output, devc_output;
reg [63:0] selected_output; //intermediate variable
reg control_error;

always @(deva_input or devb_input or devc_input or
         input_select or output_select)
begin
  control_error=0;
  deva_output=0;
  devb_output=0;
  devc_output=0;
  selected_output=0;
  case(input_select)
    ‘DEVA_ID: selected_output=deva_input;
    ‘DEVB_ID: selected_output=devb_input;
    ‘DEVC_ID: selected_output=devc_input;
    default:  control_error=1;
  endcase
  case(output_select)
    ‘DEVA_ID: deva_output=selected_output;
    ‘DEVB_ID: devb_output=selected_output;
    ‘DEVC_ID: devc_output=selected_output;
    default:  control_error=1;
  endcase
end
```