Coding Guidelines

Mandatory Guidelines

It's not enough that your design work. You will lose marks if you do not follow the guidelines in this section.

File-level Comments

Include, near the beginning of each Verilog file, comments showing: the file name, a line describing the purpose of the file, the author's name, and the date[1].

Why? These help to quickly identify the source and purpose of your code.

Example:

```
// lab1.sv
// Display digits of ID on 7-segment display.
// Jane Doe, 2020-9-15
```

Synchronous Design

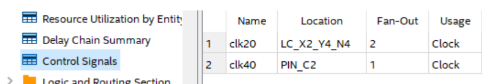
Use a single clock. The same clock signal must appear in every **`always_ff @(posedge`** *<clock>*`)` expression and only that signal should appear as a Clock in **Processing > Compilation Report > Fitter > Resource Section > Control Signals**.

Why? Programmable hardware and design tools assume synchronous (one clock) design. "Computed" clocks, such as in the ripple counter below, are inefficient and difficult to verify.

Example:

```
// NOT allowed:
  always_ff @(posedge clk40) clk20 = ~clk20 ;
  always_ff @(posedge clk20) clk10 = ~clk10 ;
```

Example: A report showing multiple clocks:



	Name	Location	Fan-Out	Usage
1	clk20	LC_X2_Y4_N4	2	Clock
2	clk40	PIN_C2	1	Clock

[1] Don't confuse this with the information on your report's cover page.

`logic` Type

Use the **`logic`** type for synthesis.

Why? System Verilog's **`logic`** can replace both **`wire`** and **`reg`**. Use it in new code.

Example:

```
// NOT allowed:
  wire clk ;
  reg [15:0] cnt ;

// OK:
  logic clk ;
  logic [15:0] cnt ;
```

Consistent Indentation

A line with **`end`** should be indented the same as the line with the corresponding **`begin`**. Each level of indentation should increase by the same amount (for example, by 4 spaces).

Why? This makes it much easier to find errors.

Example:

```
// NOT allowed:
   assign x = { a, b, c, d } ;
always_comb begin
   cnt_next = cnt ;
   if ( enable )
     begin
        cnt_next = cnt + 1'b1 ;  end
 end

// OK:
assign x = { a, b, c, d } ;
always_comb begin
   cnt_next = cnt ;
   if ( enable ) begin
      cnt_next = cnt + 1'b1 ;
   end
end
```

Use `always_ff` and `always_comb`

Use **`always_ff`** or **`always_comb`** instead of **`always`**.

Why? This avoids incomplete sensitivity lists and unintended latches.

Example:

```
// NOT allowed:
always @(enable,cnt)
  cnt_next = enable ? cnt + 1'b1 : cnt ;
always @(posedge clk)
  cnt = cnt_next ;

// OK:
always_comb
  cnt_next = enable ? cnt + 1'b1 : cnt ;
always_ff @(posedge clk)
  cnt = cnt_next ;
```

Single Assignment in `always_ff`

An **always_ff** block that defines a register may only contain one simple assignment and it must be of the form $x = x$**_next** where x is a signal name.

Don't follow examples that use sequential statements such as **if** or **case** within the **always_ff**.

Why? This ensures each **always_ff** corresponds to one register and that you can access the signals at both the input (x**_next**) and output (x) of a register.

Example:

```
// NOT allowed:
always_ff @(posedge clk) begin
  cnt = cnt + 1'b1 ;
  neg = cnt[15] ;
end

// OK:
assign cnt_next = cnt + 1'b1 ;
always_ff @(posedge clk) cnt = cnt_next ;
assign neg_next = cnt_next[15] ;
always_ff @(posedge clk) neg = neg_next ;
```

Recommended Guidelines

The following guidelines make your code easier to understand and help you avoid mistakes. However, their use won't be marked.

Add comments next to port and signal declarations and for non-obvious parts of your design. These should explain why you're doing something rather than repeating what is obvious from the code.

```
// AVOID redundant or missing comments:
module ...
  (
  input logic reset_n, clk, // active—low reset, clk
  input logic [15:0] a, b, c,
  ...
  // set usea to 1 if a>b and a<c or a>=c and a <=b
  assign usea = a > b && a < c || a >= c && a <= b ;
```

```
// DO describe non—obvious signals and code
module ...
  (
  input logic reset_n, clk,
  input logic [15:0] a, b, c, // filter inputs
  ...
  // is a the median value?
  assign usea = a > b && a < c || a >= c && a <= b ;
```

Use consistent signal naming to avoid confusion. The following are widely-recognized conventions:

- append **_n** to active-low signals
- append **_t** to type names
- append **_in** to names of input ports that have a corresponding internal signal such as a synchronised or debounced version
- append **_next** to the name of a register output to derive the name of the register input

Use enumerated types for states to make your code easier to read and to help the synthesizer optimize the design.

The garage door controller below is an example of a state machine. The synthesizer will choose an appropriate representation for the state variables.

```
module controller
  (
   input logic clk, pb_in,   // clock, pushbutton
   input logic top, bottom,  // position sensors
   output logic up, down     // motor control
   ) ;

  // declare and use an enum state type
  typedef enum int unsigned
               { off, opening, closing } state_t ;
  state_t state = off, state_next ;

  logic pb ;
  debounce db0 ( pb_in, clk, pb ) ;

  assign state_next
    = state == off && pb ?
        ( top ? closing : opening ) :
      state == opening && top ? off :
      state == closing && bottom ? off :
      state ;
  always @(posedge clk) state = state_next ;
  assign up = state == opening ;
  assign down = state == closing ;

endmodule
```