Coding Guideline

Purpose of Having Coding Standard

1. A coding standard gives a uniform appearance to the codes written by different engineers.
2. It improves readability, and maintainability of the code and it reduces complexity also.
3. It helps in code reuse and helps to detect error easily.
4. It promotes sound programming practices and increases efficiency of the programmers.

Coding Guideline

Some of the coding standard is given below

1.1. Standard Header

For better understanding and maintenance of the code, the header of different modules should follow some standard format and information. The header format must contain below items

- Name of the module
- Date of module creation
- Author of the module
- A brief description the module about what the module does

Template Header

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:        MICRO-ELECTRONICS RESEARCH LABORATORY                                                                           //
//                                                                                                                                 //
// Engineers:      Auringzaib Sabir - Verification                                                                                 //
//                                                                                                                                 //
// Additional contributions by:                                                                                                    //
//                                                                                                                                 //
// Create Date:      08-MARCH-2022                                                                                                 //
// Design Name:    xyz                                                                                                             //
// Module Name:   xyz.sv                                                                                                           //
// Project Name:    VIPs for different peripherals                                                                                 //
// Language:         SystemVerilog - UVM                                                                                           //
//                                                                                                                                 //
// Description:                                                                                                                    //
//     - Driver recieves the transacions from the sequencers(through TLM connections- Transaction....                             //
//                                                                                                                                 //
// Revision Date:                                                                                                                 //
//                                                                                                                                 //
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

1.2. Naming conventions

Meaningful and understandable variable name helps anyone to understand the reason for using it.

1.2.1. General

- All variables should use lower case with "_" between words.
  Ex. *header_valid, start_bank_no*
- Defines and parameters should use all uppercase
  Ex. *`define BUS_WIDTH 32*
  parameter IDLE_STATE = 0
- Avoid using single letter name
  Ex. *eth_packet p; // Not a good naming practice*
- Prefix inputs with "i_" and outputs with "o_"
  Ex. *input i_slave_ready, output o_php_valid*

1.2.2. SystemVerilog

- typedef should end with "_t"
  Example
  typedef int_array_t int[]
  ...
  function int_array_t sort_list (int arr1[], int arr2[])
- constraint name should end with "_c"
  Example
  constraint payload_c {payload.size() inside {[64:2000]};}

1.3. Commenting

Comments are one of the most important part of the code. It is a good practice to write the comments at the time of code development. Please DO NOT leave the comments to be added later, after the completion of the code. This will lead to either poor quality of commenting, or may require you to invest a lot of time to revisit the code and write the comment. The comments are, most of the time, either given before the logic or at end of the line. Consider the following two examples:

***//* Latching the data into an internal register at valid**
always @ (posedge clk) begin
* if (valid) data_int <= data;*
end
assign out = (state_reg == DONE); ***// Assert the output when FSM is in DONE state***

If the comment is large enough, greater than say 100 characters, then it should be broken into multiple lines. Sometimes, it is a good idea to also include equations, timing diagrams and other relevant logic in the comment.

1.4. Alignment

One key factor in writing a good, readable code is using proper alignment of the code. There are a number of things in your code which can be properly aligned to improve the aesthetics and readability of the code. Examples are Ports list, Instances, If...else and case statements. Following are two examples of the instance of a full adder.

// Instantiating full adder – **Example 1** - **Not recommended**

```
full_adder full_adder (.a(a), .b(b), .c_in(c_in), sum(sum), c_out(c_out));
```

// Instantiating full adder – **Example 2** - **Recommended**

```
full_adder full_adder
  (.a        (a),
   .b        (b),
   .c_in     (c_in ),
   .sum      (sum ),
   .c_out    (c_out)
  );
```

1.5. Concise Coding Style

It is a good practice to use begin...end for blocks that contain multiple lines even if it is not required by the language. It improves readability by clearly demarcating the block.
Consider the following example

Example 1 - No begin-end - **Not recommended**

```
always @ (posedge clk or negedge rst_n)
  if(!rst_n) control_int <= 32'h0;
  else if (valid) control_int <= control;
```

Example 2 - with begin-end - **Recommended**

```
always @ (posedge clk or negedge rst_n) begin
  if(!rst_n) control_int <= 32'h0;
  else if (valid) control_int <= control;
end
```

However if the block contains only one line of code, begin … end is not required and only adds coding effort

Example 3 - begin … end is a overhead
initial force dut.pll_out = 0;

With SystemVerilog, multiple statements can be written between the task declaration and endtask, or function and endfunction, which means that the begin .... end can be omitted. Having begin … end inside task
curly-braces {}. But they should only be used if they are really needed. Consider the code snippet or function only adds to coding effort and increases indentation level unnecessarily.

Example 4 - No begin … end for task

```
task display_valid();
  if(valid) $display("Inside Task : Valid is high");
  else $display("Inside Task : Valid is low");
endtask
```

1.6. State Encoding in FSM

The states of the FSM should be properly encoded using localparam. This allows the FSM design to be very much organized and easier to understand. For example, if a state machine has three states IDLE, READ and WRITE, this can be encoded as follows:

```
localparam IDLE  = 2'b00;
localparam READ  = 2'b01;
localparam WRITE = 2'b10;
```

1.7. Indentation

Proper indentation is very important to increase the readability of the code. Following a consistent indentation throughout the code is of great significance

- **Level of indentation:**
  Level of indentation should be **two-space** for every level. For example

```
always @ (posedge clk) begin
  wdata_reg <= wdata;
  wdata_valid <= wvalid;
end
```

  Use of "Tab" to indent is not allowed because different text editors interpret "Tab" differently. It is recommended to set the Tab key's interpretation to two-spaces in the editor of your choice
- **Inline begin … end:**
  Verilog allows to write, begin inline with the condition or as a separate line. writing begin in a seperate line, adds a extra line unnecessary and increases overall indentation level

Example 1 - Seperate begin line - **Not recommended**

```
always @ (posedge clk or negedge rst_n)
  begin
    if(!rst_n) control_int <= 32'h0;
    else if (valid) control_int <= control;
  end
```

Example 2 - Inline begin - **Recommended**

```
always @ (posedge clk or negedge rst_n) begin
  if(!rst_n) control_int <= 32'h0;
  else if (valid) control_int <= control;
end
```

- **Use of brackets:**
  Brackets should be used to make the intention of the code clear.

  Example 1 - No brackets - **Not recommended**

  ```
  assign out = state_reg == DONE && count == 2;
  ```

  Example 2 - With brackets - **Recommended**

  ```
  assign out = (state_reg == DONE) && (count == 2);
  ```

1.8. Module and File Name

It is a nice idea to keep the file name, same as the Module name. For example, if the module name is dual_port_ram, then the file name should be dual_port_ram.sv. Please note that design files can also be named as .sv as use of SystemVerilog features for design is highly encouraged. It is also important to note that it is a good practice to keep only one module/class/interface per file.