Coding Guidelines

References:

1. Clifford E. Cummings, The Fundamentals of Efficient Synthesizable Finite State Machine Design
2. Clifford E. Cummings, Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog.
3. OpenCores, HDL modeling guidelines
4. Lattice Semiconductor, HDL Coding Guidelines
5. Microsemi, Actel HDL Coding, Style Guide

Introduction

Coding style has a significant impact on how a digital system design is implemented and ultimately how it performs. However, synthesis tools have significantly improved, it is still the designer’s responsibility to generate HDL code that guides the synthesis tools and achieves the best result for a given architecture. This document provides Verilog HDL design guidelines for both novice and experienced designers. Adopting these guidelines will reduce the amount of time required to get high quality IP cores and will reduce possibilities for functional problems. Following these guidelines will, also, improve reusability and readability of the code.

The document outlines set of

- Rules (R): A rule is a required practice that enables good designs; so, it ***Must*** be implemented.
- Guidelines (G): A guideline is a recommended practice that enhances the design. So, it is ***Up to*** the designer to implement them.

General

- R: Use Indentation to Improve the Readability
- R: One Verilog statement per line.
- R: Keep the line length 80 characters or less.

Commenting

- R: Comment your code. Be reasonable. Too many comments and too few comments have their drawbacks.
- G: Use "`/*..*/`" for multi-line comments.
- R: Add the following header to the top of all of your source files

```
/*********************************************************************
 *
 * Module: module_name.v
 * Project: Project_Name
 * Author: name and email
 * Description: put your description here
 *
 * Change history: 01/01/17 – Did something
 *                 10/29/17 – Did something else
 *
 **********************************************************************/
```

Naming

- R: Create a Naming Convention for the design. Document it and use it consistently throughout the design.
- G: Use capitals to identify constants: e.g. **`STATE_S0`**, **`GO`**, **`ZERO`**, ….
- G: Use lowercase letters for all signal names, variable names, and port names.
- G: All parameter names must be in capitals
- R: Do not Use Hard Coded Numeric Values. Instead, use **`` `define``** to define constants.
- R: Do not declare **`` `define``** statements in individual modules. Place all global **`` `define``**'s in external definition files that are included in the project (using **`` `include``**).
- G: For active low signals, end the signal name with an underscore followed by a lowercase character (for example, **`_b`** or **`_n`**).
- R: When describing multibit buses, use a consistent ordering of bits

Module declaration/instantiation

- R: At most one module per file.
- R: Each module should be defined in a separate file.
- R: Declare inputs and outputs one per line.
- G: Declare the ports in the following order:
    - Clocks
    - Resets
    - Enables
    - Other control signals
    - Data and address lines
- G: Group ports logically by their function.

- R: Ports with **`inout`** cannot be used to create tri-state buses. Yosys synthesis tool does not support tri-state logic.
- R: Do not instantiate modules using positional arguments. Always use the dot notation. In other words, connect ports by name in module instantiations.
- R: Port connection widths must match.
- R: Expressions are not allowed in port connections.
- R: Drive all unused module inputs
- R: Connect unused module outputs
- G: Parameterize modules if appropriate and if readability is not degraded.

Clocking

- R: Core clock signal for a module is **`'clk'`**.
- G: All clocks, other than the core clock, should have a description of the clock and the frequency (e.g. **`clk_ssp_625`**).
- G: Use one clock per module if possible.
- G: Do not use mixed clock edges if possible
- R: Do not gate the clocks

Resetting

- R: Always reset sequential modules. Reset makes a design more deterministic and easier to verify.
- R: Use asynchronous active high reset.
- G: Use 'rst' for a module reset signal.

Modeling

- R: Blocking assignments should only be used for modeling combinational logic (used in **`always_comb`** blocks)
- R: Non-blocking assignments should only be used for modeling sequential logic (used in **`always_ff`** blocks)
- R: Don't mix blocking and non-blocking assignments within a code block. Separate Combinational from Sequential.
- R: Do not assign to the same variable from different **`always`** blocks to avoid race conditions
- R: One clock per always sensitivity list.
- R: Use @* for the sensitivity list for combinational logic always blocks:
- `R: $signed()` can only be used around the operands `to >>>, >, >=, <, <=` to ensure that these operators perform the signed equivalents.
- R: Wires must be declared before using them
- G: If you would like to generate hardware using loops, then you should use generate blocks.
- G: Multipliers can be synthesized and small multipliers can even be synthesized efficiently, but we recommend not to use the multiplication operator. Design your multiplier to be optimized for the design.
- G: Follow one of the recommended templates outlined by [1] to code an FSM.
- R: Avoid LATCHES! Check the results of synthesis and ensure you do not have any inferred latches—it usually means you missed something in your coding.
    - Every **`if`** statement has **`else`**
    - Every **`case`** statement has a default
    - All signals are initialized within a combinational block

- R: Do not use **`casex`** in RTL modeling.
- R: **`casez`** can only be used in very specific situations to compactly implement priority encoder style hardware structures.
- R: No asynchronous logic/combinational feedback loops/self-timed logic.
- R: Don't hand-instantiated standard library cells within your RTL.
- R: Synchronize any asynchronous input signal. Use brute force synchronizer (2 flip flops) for single signals. Consult [2] for synchronizing buses.
- R: Synchronize any signal that crosses clock domains. Use brute force synchronizer to synchronize a signal coming from slower domain to a faster domain. Consult [2] for other scenarios.
- G: Eliminate Glue Logic at the Top Level
- R: Do not assign **`x`** value to signals
- R: Operand sizes must match
- R: Verilog primitives cannot be used.
- R: Use parentheses in complex equations
- R: Avoid unbounded loops

Memory

- R: Do not use large arrays. If your design needs SRAM memory block(s), consider using one of the provided memory blocks and interface it to the module that needs it.

Test Benching

- R: **`initial`** shall not be used in RTL modeling. It can be used only to develop test benches. Use a reset signal to initialize registers if needed.
- R: Delays shall not be used in RTL modeling. They can be used in test benching only.
- G: In the test bench sample before the clock edge and drive after the clock edge.