System Verilog Style Guide

Benjamin Davis, allrisc.dev@gmail.com

2023-03-06

Contents

Chapter 1

Summary

Robert C. Martin, author of *Clean Code: A Handbook of Agile Software Craftsmanship*, famously said:

> Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ... [Therefore,] making it easy to read makes it easier to write.

This quote certainly holds true whether the code is traditional software, RTL, or Verification frameworks. Thus this document puts forward a set of rules and guidelines for writing easy to understand and consistent SystemVerilog.

A vast majority of these rules apply to SystemVerilog for synthesis and for verification. When this is not the case it will be noted and further guidance shall be provided.

While these are referred to as rules, they may not be the correct solution for all code-bases. When an instance arises where a developer feels that there is a method of writing the code which is clearer they should be empowered to do so. But, in general these moments are few and far-between, so the developer is asked to think critically about whether this approach is truly better.

1.1 Terminology

Unless otherwise stated, the following terminology conventions apply to the guidance layed out in this document:

- The word **must** indicates a mandatory requirement.
- Similarly, **do not** indcations a strict prohibition.
- The word **recommended** indicates a strong suggestion that should be applied, unless the implications and reasons for taking an alternative approach are completely understood.
- The word **may** indicates a course of action is permitted and optional.
- The word **can** indicates a course of action is possible given material, physical, or causal constraints.

1.2 General

The following general style constraints must be adhered:

- Generally, names should be descriptive and avoid abbreviations.
- Non-ASCII characters are forbidden.
- Indentation uses spaces, no tabs. Indentation is two spaces for nesting.
- Place a space between `if` and the parenthesis in conditional expressions.
- Use horizontal whitespace around operators, and avoid trailing whitespace at the end of lines.

1.3 Which Verilog to Use

Prefer SystemVerilog-2017.

All RTL and tests should be developed in SystemVerilog, following the IEEE 1800-2017 (SystemVerilog-2017) standard, except for prohibited features.

The standards document is available free of cost through IEEE GET (a registration is required).

Chapter 2

SystemVerilog Conventions

This chapter seeks to standardize aesthetic aspects of SystemVerilog.

2.1 General File Rules

2.1.1 File Extensions

Use the `.sv` extension for SystemVerilog files (or `.svh` for files that are included via the preprocessor).

File extensions have the following meanings:

- `.sv` indicates a SystemVerilog file.
- `.svh` indicates a SystemVerilog header file intended to be included in another file using the `‘include` directive.
- `.v` indicates a Verilog file.
- `.vh` indicates a Verilog header file.

Only `.sv` and `.v` files are intended to be compilation units. `.svh` and `.vh` files may only be `‘include`d into other files.

With exceptions of netlist files, each `.sv` or `.v` file should contain only one module, and the name should be associated. For instance, file `foo.sv` should contain only the module `foo`.

2.1.2 Characters

Use only ASCII characters with UNIX-style line endings(¨`\n`¨).

2.1.3 POSIX File Endings

All lines on non-empty files must end with a newline (\¨n¨).

2.1.4 Line Length

Wrap the code at 100 characters per line.

The maximum line length for style-compliant Verilog code is 100 characters per line.

Exceptions:

- Any place where line wraps are impossible (for example, an include path might extend past 100 characters).

Line-wrapping contains additional guidelines on how to wrap long lines.

2.1.5 No Tabs

Do not use tabs anywhere.

Use spaces to indent or align text. See Indentation for rules about indentation and wrapping.

To convert tabs to spaces on any file, you can use the UNIX expand utility.

2.1.6 No Trailing Spaces

Delete trailing whitespace at the end of lines.

2.2 Begin and End

Use `begin` and `end` unless the entire statement fits on one line.

If a statement cannot fit on a single line then the statement(s) must be enclosed in a `begin` and `end` block.

Correct

```
always_ff @(posedge clk) begin
  q <= d;
end
```

Correct

```
always_ff @(posedge clk) q <= d;
```

Incorrect

```
always_ff @(posedge clk)
  q <= d;
```

The `begin` must be on the same line as the prior keyword. `begin` must end the line. `end` must occupy it's own line. When multiple `end` are found sequentially, they should be accommodated by a label (or comment) to clarify which `begin` this `end` corresponds with.

Correct

```
if (a == 1'b1) begin
  q <= '0;
end
else begin
  q <= d;
end
```

Incorrect

```
if (a == 1'b1) begin
  q <= '0;
end else begin
  q <= d;
end
```

The above `begin` this `end` principles also apply to case statements.

Correct

```
casez (state)
  IDLE_STATE: state <= STATE_1;
  STATE_1: state <= STATE_2;
  STATE_2: begin
    out <= 1'b1;
    state <= IDLE_STATE;
  end
  default: begin
    out <= 1'b0;
    state <= IDLE_STATE;
  end
endcase
```

Incorrect

```
casez (state)
  IDLE_STATE:
    state <= STATE_1;
  STATE_1:
    state <= STATE_2;
  STATE_2: begin
    out <= 1'b1;
    state <= IDLE_STATE;
  end
  default: begin
    out <= 1'b0;
    state <= IDLE_STATE;
  end
endcase
```

2.3 Indentation

Indentation must be two spaces per level. Tab characters must be expanded to spaces.

Modern editors can/should be set to produce spaces when the tab key is hit.

An additional level of indentation must be added between paired keywords. (e.g. `begin` / `end`, `module` / `endmodule`, `task` / `endtask`, `function` / `endfunction`, etc.)

2.3.1 Wrapping Lines

When wrapping long expressions to multiple lines one of the following approaches should be used:

- Indent the continuation line by 4-spaces.
- Align the line continuation with an opening parentheses or brace on the first line.

Additionally, opening parentheses or braces which end a line in the wrapped expression should be closed with the respective character on their own line.

Correct

```
assign valid = enabled && (
    addr <  8'h7   ||
    addr == 8'h7F
);

assign error = error && (error_enabled &&
    addr >= 8'h7   &&
    addr != 8'h7F);

assign some_struct = '{
    addr: addr,
    we:   1'b0,
    data: data
};

assign addr = find_address(something, other_thing,
                           x, y, some_long_name);
```

2.3.2 Justification

Virtually every coding standard in the world has moved to using spaces instead of tabs. This has mainly been driven by a desire for consistency.

While tabs allow individual developers to choose their personal preference for spacing of tabs, this invariably impacts the spacing of wrapped lines. Additionally, when adding tabbed lines to a code base which uses spaces, it is likely that the number of tabs will be chosen to match the current code base, rather than the match the true indentation level.

To avoid these issues spaces are required.

So far as the number of spaces is concerned this is more personal preference. Python's official guidelines specify 4-spaces, while most of Google's coding standards enforce 2-spaces (the Linux kernel guidelines even specify 8!). In this document 2 spaces have been chosen because they appear to be the generally more readable method to the authors of these guidelines.

2.4 Spacing

Use spacing which aids readability.

2.4.1 Tabular Alignment

Tabular alignment is the practice of aligning specific parts of multiple related lines. This allows for ease of reading and comparing multiple related lines.

The use of tabular alignment is strongly encouraged.

Correct

```
module example (
  input                clk_i,
  input                rst_low_i,
  input  logic [ 7:0] data_i,
  output logic [31:0] repeated_data_o,

  // Scan is not directly related to above, so it can be on it's own "table"
  input        scan_clk_i,
  input        scan_rstn_i,
  input        scan_i,
  output logic scan_o
);
```

2.4.2 Expressions

Whitespace should be included on both sides of all non-trivial binary operators.

Correct

```
wire [WIDTH-1:0] reg;
wire [WIDTH - 1:0] reg2; // Acceptable, but unnecessary

assign reg_val = ((addr == reg_addr) && enabled) ? reg_r : ~reg_r;
```

Incorrect

```
assign reg_val=((addr==reg_addr)&&enabled)?reg_r:~reg_r;
```

2.4.3 Keywords

Whitespace should be included on both sides of SystemVerilog keywords.

The following exceptions apply:

- Before a keyword at the beginning of a line
- After a keyword at the end of a line
- Before keywords which are immediately preceded by an opening parentheses or brace

2.4.4 Comma Delimited Lists

When multiple items on a line are comma delimited each comma must be followed by a space.

2.4.5 Definitions

Add a space around the packed dimensions.

Do not add a space -

- Between identified and unpacked dimensions
- Between multiple dimensions

2.4.6 Parameterized Types

Add a single space before type parameters, except when part of a qualified name.

2.4.7 Labels and Case Items

Do not add a space prior to the colon in a Label or Case item. Add at least one space after the colon in a Label or Case item.

2.4.8 Calls

Functions, Task, and Macro calls must not have any space between the name and the the opening parenthesis.

2.5 Parentheses

Parentheses should be used to make operations unambiguous.

In any situation where most people would have to spend some modicum of thought, parentheses should be used instead to make the order of operations clear.

2.6 Ternary Expressions

Nested ternary expressions must be enclosed in parentheses unless they are in the false condition. **At most** 3 ternary expressions may be nested together.

2.7 Comments

C++ style comments (`// comment`) are preferred. C style comments (`/* comment */` may also be used)

The following principles should be followed regarding comments:

- A comment on its own line describes the code that immediately follows.
- Inline comments describe that line of code
- Header style comments (see below) can be used to seperate different functional parts within a module or class.

Correct

```
// The following module does: something
module something (...);

  localparam int SOME_MAGIC_NUMER = 10; // MAGIC_NUMBER is from ...

  ////////////////////
  // Controller FSM //
  ////////////////////
  ...

  ///////////////
  // Data Path //
  ///////////////
  ...

endmodule
```

2.8 Declarations

Signals must be declared before use. Implicit net declarations **must** not be used.

Chapter 3

Naming

3.1 Summary

Construct	Style
Module and Package Declaration	lower_snake_case
Class Declaration	UpperCamelCase
Interface Declaration	UpperCamelCase_if
Instance names	lower_snake_case
Signals (nets and ports)	lower_snake_case
Variables, functions, tasks	lower_snake_case
Named code blocks	lower_snake_case
d̀efine macros	ALL_CAPS
Tunable parameters for parameterized modules, classes, and interfaces	ALL_CAPS
Constants	ALL_CAPS
Enumeration types	lower_snake_case_e
Other typedef types	lower_snake_case_t
Enumerated value names	UpperCamelCase

3.2 Constants

3.3 Macros

3.4 Suffixes

3.5 Enumerations

3.6 Signals

3.6.1 Clocks

3.6.2 Resets

Chapter 4

Language Features

4.1 Prefer SystemVerilog Constructs

4.2 Package Dependencies

4.3 Module Declarations

4.4 Module Instantiations

4.5 Constants

4.6 Problematic Language Features

Chapter 5

Design Conventions

Appendix A

Condensed Style Guide

Appendix B

Basic Template