lowRISC Verilog Coding Style Guide

Basics

Summary

Verilog is the main logic design language for lowRISC Comportable IP.

Verilog and SystemVerilog (often generically referred to as just "Verilog" in this document) can be written in vastly different styles, which can lead to code conflicts and code review latency. This style guide aims to promote Verilog readability across groups. To quote the Google C++ style guide: "Creating common, required idioms and patterns makes code much easier to understand."

This guide defines the Comportable style for Verilog. The goals are to:

- promote consistency across hardware development projects
- promote best practices
- increase code sharing and re-use

This style guide defines style for both Verilog-2001 and SystemVerilog compliant code. Additionally, this style guide defines style for both synthesizable and test bench code.

See the Appendix for a condensed tabular representation of this style guide.

Table of Contents

Terminology Conventions

Unless otherwise noted, the following terminology conventions apply to this style guide:

- The word ***must*** indicates a mandatory requirement. Similarly, ***do not*** indicates a prohibition. Imperative and declarative statements correspond to ***must***.
- The word ***recommended*** indicates that a certain course of action is preferred or is most suitable. Similarly, ***not recommended*** indicates that a course of action is unsuitable, but not prohibited. There may be reasons to use other options, but the implications and reasons for doing so must be fully 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.

Default to C-like Formatting

Where appropriate, format code consistent with https://google.github.io/styleguide/cppguide.html

Verilog is a C-like language, and where appropriate, we default to being consistent with Google's C++ Style Guide.

In particular, we inherit these specific formatting guidelines:

- Generally, names should be descriptive and avoid abbreviations.
- Non-ASCII characters are forbidden.
- Indentation uses spaces, no tabs. Indentation is two spaces for nesting, four spaces for line continuation.
- 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.
- Maintain consistent and good punctuation, spelling, and grammar (within comments).
- Use standard formatting for comments, including C-like formatting for TODO and deprecation.

Style Guide Exceptions

Justify all exceptions with a comment.

No style guide is perfect. There are times when the best path to a working design, or for working around a tool issue, is to simply cut the Gordian Knot and create code that is at variance with this style guide. It is always okay to deviate from the style guide by necessity, as long as that necessity is clearly justified by a brief comment, as well as a lint waiver pragma where appropriate.

Which Verilog to Use

Prefer SystemVerilog-2012.

All RTL and tests should be developed in SystemVerilog, following the SystemVerilog-2012 standard, except for prohibited features.

Verilog/SystemVerilog Conventions

Summary

This section addresses primarily aesthetic aspects of style: line length, indentation, spacing, etc.

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 defining a module or package.
- `.svh` indicates a SystemVerilog header file intended to be included in another file using a preprocessor `` `include `` directive.
- `.v` indicates a Verilog-2001 file defining a module or package.
- `.vh` indicates a Verilog-2001 header file.

Only `.sv` and `.v` files are intended to be compilation units. `.svh` and `.vh` files may only be `` `include ``-ed 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`.

General File Appearance

Characters

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

POSIX File Endings

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

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.

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.

No Trailing Spaces

Delete trailing whitespace at the end of lines.

Begin / End

Use `begin` and `end` unless the whole statement fits on a single line.

If a statement wraps at a block boundary, it must use `begin` and `end.` Only if a whole semicolon-terminated statement fits on a single line can `begin` and `end` be omitted.

👍

```
// Wrapped procedural block requires begin and end.
always_ff @(posedge clk) begin
  q <= d;
end
```

👍

```
// The exception case, where begin and end may be omitted as the entire
// structure fits on a single line.
always_ff @(posedge clk) q <= d;
```

```
// Incorrect because a wrapped statement must have begin and end.
always_ff @(posedge clk)
  q <= d;
```

`begin` must be on the same line as the preceding keyword, and ends the line. `end` must start a new line. `end else begin` must be together on one line. The only exception is if `end` has a label, a following `else` should be on a new line.

```
// "end else begin" are on the same line.
if (condition) begin
  foo = bar;
end else begin
  foo = bum;
end
```

```
// begin/end are omitted because each semicolon-terminated statement fits on
// a single line.
if (condition) foo = bar;
else foo = bum;
```

```
// Incorrect because "else" must be on the same line as "end".
if (condition) begin
  foo = bar;
end
else begin
  foo = bum;
end
```

```
// An exception is made for labeled blocks.
if (condition) begin : a
  foo = bar;
end : a
else begin : b
  foo = bum;
end : b
```

The above style also applies to individual case items within a case statement. `begin` and `end` may be omitted if the entire case item (the case expression and the associated statement) fits on a single line. Otherwise, use the `begin` keyword on the same line as the case expression.

```
// Consistent use of begin and end for each case item is good.
unique case (state)
  StIdle: begin
    next_state = StA;
  end
  StA: begin
    next_state = StB;
  end
  StB: begin
    next_state = StIdle;
    foo = bar;
  end
  default: begin
    next_state = StIdle;
  end
endcase
```

```
// Case items that fit on a single line may omit begin and end.
unique case (state)
  StIdle: next_state = StA;
  StA: next_state = StB;
  StB: begin
    next_state = StIdle;
    foo = bar;
  end
  default: next_state = StIdle;
endcase
```

```
unique case (state)
  StIdle:              // These lines are incorrect because we should not wrap
    next_state = StA;  // case items at a block boundary without using begin
  StA:                 // and end.  Case items should fit on a single line, or
    next_state = StB;  // else the procedural block must have begin and end.
  StB: begin
    foo = bar;
    next_state = StIdle;
  end
  default: begin
    next_state = StIdle;
  end
endcase
```

Indentation

Indentation is two spaces per level.

Use spaces for indentation. Do not use tabs. You should set your editor to emit spaces when you hit the tab key.

Indented Sections

Always add an additional level of indentation to the enclosed sections of all paired keywords. Examples of SystemVerilog keyword pairs: `begin / end`, `module / endmodule`, `package / endpackage`, `class / endclass`, `function / endfunction`.

Line Wrapping

When wrapping a long expression, indent the continued part of the expression by four spaces, like this:

```
assign zulu = enabled && (
    alpha < bravo &&
    charlie < delta);

assign addr = addr_gen_function_with_many_params(
    thing, other_thing, long_parameter_name, x, y,
    extra_param1, extra_param2);

assign structure = '{
    src: src,
    dest: dest,
    default: '0};
```

Or, if it improves readability, align the continued part of the expression with a grouping open parenthesis or brace, like this:

```
assign zulu = enabled && (alpha < bravo &&
                          charlie < delta);

assign addr = addr_gen_function(thing, other_thing,
                                long_parameter_name,
                                x, y);

assign structure = '{src: src,
                     dest: dest,
                     default: '0};
```

Operators in a wrapped expression can be placed at either the end or the beginning of each line, but this must be done consistently within a file.

Preprocessor Directives

Keep branching preprocessor directives left-aligned and un-indented.

Keep branching preprocessor directives (`` `ifdef `` , `` `ifndef `` , `` `else `` , `` `elsif `` , `` `endif ``) aligned to the left, even if they are nested. Indent the conditional branches of text as if the preprocessor directives were absent. Non-branching preprocessor directives must follow the same indentation rules as the regular code.

👍

```
package foo;
`ifdef FOO              // good: branching directive left-aligned
  `include "foo.sv";    // normal indentation for non-branching directives
  parameter bit A = 1;  // normal indentation for the regular code
`ifdef BAR              // good: branching directive left-aligned
  parameter bit A = 2;
`else
  parameter bit A = 3;
`endif
`endif
endpackage : foo
```

Un-indented branching preprocessor directives disrupt the flow of reading to emphasize that there is conditional text. Leaving conditional branch text un-indented will result in post-preprocessed text looking properly indented.

Spacing

Comma-delimited Lists

For multiple items on a line, one space must separate the comma and the next character.

Additional whitespace is allowed for readability.

👍

```
bus = {addr, parity, data};
a = myfunc(lorem, ipsum, dolor, sit, amet, consectetur, adipiscing, elit,
           rhoncus);
mymodule mymodule(.a(a), .b(b));
```

👎

```
{parity,data} = bus;
a = myfunc(a,b,c);
mymodule mymodule(.a(a),.b(b));
```

Tabular Alignment

Adding whitespace to cause related things to align is encouraged.

Where it is reasonable to do so, align a group of two or more similar lines so that the identical parts are directly above one another. This alignment makes it easy to see which characters are the same and which characters are different between lines.

Use spaces, not tabs.

For example:

```
logic [7:0]  my_interface_data;
logic [15:0] my_interface_address;
logic        my_interface_enable;
```

Expressions

Include whitespace on both sides of all binary operators.

Use spaces around binary operators. Add sufficient whitespace to aid readability.

For example:

👍

```
assign a = ((addr & mask) == My_addr) ? b[1] : ~b[0];  // good
```

is better than

👎

```
assign a=((addr&mask)==My_addr)?b[1]:~b[0];  // bad
```

Exception: when declaring a bit vector, it is acceptable to use the compact notation. For example:

👍

```
wire [WIDTH-1:0] foo;   // this is acceptable
wire [WIDTH - 1 : 0] foo;  // fine also, but not necessary
```

When splitting alternation expressions into multiple lines, use a format that is similar to an equivalent if-then-else line. For example:

👍

```
assign a = ((addr & mask) == `MY_ADDRESS) ?
           matches_value :
           doesnt_match_value;
```

Array Dimensions in Declarations

Add a space around packed dimensions.

Do not add a space:

- between identifier and unpacked dimensions.
- between multiple dimensions.

Applies to packed and unpacked arrays as well as dynamic arrays, associative arrays, and queues.

👍

```
logic [7:0][3:0] data[128][2];
typedef logic [31:0] word_t;
bit bit_array[512];
data_t some_array[];
data_t some_map[addr_t];
data_t some_q[$];
```

👎

```
// There must not be a space between dimensions.
logic [7:0] [3:0] data[128] [2];
// There must not be a space around packed dimensions.
typedef logic[31:0]word_t;
// There must not be a space between identifier and unpacked dimension.
bit bit_array [512];
// Dynamic, associative, and queue "dimensions" are treated the same as unpacked
// dimensions.  There must not be a space.
data_t some_array [];
data_t some_map [addr_t];
data_t some_q [$];
```

Parameterized Types

Add one space before type parameters, except when the type is part of a qualified name.

A qualified name contains at least one scope `::` operator connecting its segments. A space in a qualified name would break the continuity of a reference to one symbol, so it must not be added. Parameter lists must follow the space-after-comma rule.

👍

```
my_fifo #(.WIDTH(4), .DEPTH(2)) my_fifo_nibble ...

class foo extends bar #(32, 8);  // unqualified base class
  ...
endclass
```

```
foo_h = my_class#(.X(1), .Y(0))::type_id::create("foo_h");  // static method call

my_pkg::x_class#(8, 1) bar;  // package-qualified name
```

👎

```
my_fifo#(.WIDTH(4), .DEPTH(2)) my_fifo_2by4 ...

class foo extends bar#(32, 8);  // unqualified base class
  ...
endclass

foo_h = my_class #(.X(1), .Y(0))::type_id::create("foo_h");  // static method call

my_pkg::x_class #(8, 1) bar;  // package-qualified name
```

Labels

When labeling code blocks, add one space before and after the colon.

For example:

👍

```
begin : foo
end : foo
```

👎

```
end:bar            // There must be a space before and after the colon.
endmodule: foobar  // There must be a space before the colon.
```

Case items

There must be no whitespace before a case item's colon; there must be at least one space after the case item's colon.

The `default` case item must include a colon.

For example:

👍

```
unique case (my_state)
  StInit:   $display("Shall we begin");
  StError:  $display("Oh boy this is Bad");
  default: begin
    my_state = StInit;
    interrupt = 1;
  end
endcase
```

👎

```
unique case (1'b1)
  (my_state == StError)  : interrupt = 1; // Excess whitespace before colon
  default:begin end                       // Missing space after colon
endcase
```

Function And Task Calls

Function and task calls must not have any spaces between the function name or task name and the open parenthesis.

For example:

👍

```
process_packet(pkt);
```

👎

```
process_packet (pkt);  // There must not be a space before "("
```

Macro Calls

Macro calls must not have any spaces between the macro name and the open parenthesis.

For example:


```
`uvm_error(ID, "you fail")
`ASSERT(name, a & b, clk, rst)
```



```
`uvm_error (ID, "you fail")  // There must not be a space before "("
`ASSERT (name, a & b, clk, rst)
```

Line Continuation

It is mandatory to right-align line continuations.

Aligning line continuations (' `\` ' character) helps visually mark the end of a multi-line macro. The position of alignment only needs to be beyond the rightmost extent of a multi-line macro by at least one space, when a space does not split a token, but should not exceed the maximum line length.

```
`define REALLY_LONG_MACRO(arg1, arg2, arg3) \
    do_something(arg1);                      \
    do_something_else(arg2);                 \
    final_action(arg3);
```

Space Around Keywords

Include whitespace before and after SystemVerilog keywords.

Do not include a whitespace:

- before keywords that immediately follow a group opening, such as an open parenthesis.
- before a keyword at the beginning of a line.
- after a keyword at the end of a line.

For example:

```
// Normal indentation before if.  Include a space after if.
if (foo) begin
end
// Include a space after always, but not before posedge.
always_ff @(posedge clk) begin
end
```

Parentheses

Use parentheses to make operations unambiguous.

In any instance where a reasonable human would need to expend thought or refer to an operator precedence chart, use parentheses instead to make the order of operations unambiguous.

Ternary Expressions

Ternary expressions nested in the true condition of another ternary expression must be enclosed in parentheses.

For example:

```
assign foo = condition_a ? (condition_a_x ? x : y) : b;
```

While the following nested ternary has only one meaning to the compiler, the meaning can be unclear and error-prone to humans:

```
assign foo = condition_a ? condition_a_x ? x : y : b;
```

Parentheses may be omitted if the code formatting conveys the same information, for example when describing a priority mux.

```
assign foo = condition_a ? a :
             condition_b ? b : not_a_nor_b;
```

Comments

C++ style comments (`// foo`) are preferred. C style comments (`/* bar */`) can also be used.

A comment on its own line describes the code that follows. A comment on a line with code describes that line of code.

For example:

```
// This comment describes the following module.
module foo;
  ...
endmodule : foo

localparam bit ValBaz = 1;  // This comment describes the item to the left.
```

It can sometimes be useful to structure the code using header-style comments in order to separate different functional parts (like FSMs, the main datapath or registers) within a module. In that case, the preferred style is a single-line section name, framed with `//` C++ style comments as follows:

```
module foo;

  ////////////////
  // Controller //
  ////////////////
  ...

  ///////////////////////
  // Main ALU Datapath //
  ///////////////////////
  ...

endmodule : foo
```

If the designer would like to use comments to mark the beginning/end of a particular section for better readability (e.g. in nested for loop blocks), the preferred way is to use a single-line comment with no extra delineators, as shown in the examples below.

```
// begin: iterate over foobar
for (...) begin
...
end
// end: iterate over foobar
```

```
for (...) begin // iterate over foobar
...
end // iterate over foobar
```

```
//------------------------- iterate over foobar -------------------------------
for (...) begin
...
end
//------------------------- iterate over foobar -------------------------------
```

👎

```
////////////////////////////
// begin iterate over foobar //
////////////////////////////
for (...) begin
...
end
////////////////////////////
// end iterate over foobar   //
////////////////////////////
```

Declarations

Signals must be declared before they are used. This means that implicit net declarations must not be used.

Within modules, it is **recommended** that signals, types, enums, and localparams be declared close to their first use. This makes it easier for the reader to find the declaration and see the signal type.

Basic Template

A template that demonstrates many of the items is given below.

Template:

```
// Copyright lowRISC contributors.
// Licensed under the Apache License, Version 2.0, see LICENSE for details.
// SPDX-License-Identifier: Apache-2.0
//
// One line description of the module

module my_module #(
  parameter Width = 80,
  parameter Height = 24
) (
  input              clk_i,
  input              rst_ni,
  input              req_valid_i,
  input  [Width-1:0] req_data_i,
  output             req_ready_o,
  ...
);

  logic [Width-1:0] req_data_masked;

  submodule u_submodule (
    .clk_i,
    .rst_ni,
    .req_valid_i,
    .req_data_i    (req_data_masked),
    .req_ready_o,
    ...
  );

  always_comb begin
    req_data_masked = req_data_i;
    case (fsm_state_q)
      ST_IDLE: begin
        req_data_masked = req_data_i & MASK_IDLE;
        ...
  end

  ...

endmodule
```

Naming

Summary

Construct	Style
Declarations (module, class, package, interface)	`lower_snake_case`
Instance names	`lower_snake_case`

Construct	Style
Signals (nets and ports)	`lower_snake_case`
Variables, functions, tasks	`lower_snake_case`
Named code blocks	`lower_snake_case`
`define macros	`ALL_CAPS`
Tunable parameters for parameterized modules, classes, and interfaces	`UpperCamelCase`
Constants	`ALL_CAPS` or `UpperCamelCase`
Enumeration types	`lower_snake_case_e`
Other typedef types	`lower_snake_case_t`
Enumerated value names	`UpperCamelCase`

Constants

Declare global constants using parameters in the project package file.

In this context, **constants** are distinct from tuneable parameters for objects such as parameterized modules, classes, etc.

Explicitly declare the type for constants.

When declaring a constant:

- within a package use `parameter`.
- within a module or class use `localparam`.

The preferred method of defining constants is to declare a `package` and declare all constants as a `parameter` within that package. If the constants are to be used in only one file, it is acceptable to keep them defined within that file rather than a separate package.

Define project-wide constants in the project's main package.

Other packages may also be declared with their own `parameter` constants to facilitate the creation of IP that may be re-used across many projects.

The preferred naming convention for all immutable constants is to use `ALL_CAPS`, but there are times when the use of `UpperCamelCase` might be considered more natural.

Constant Type	Style Preference	Conversation
`define	`ALL_CAPS`	Truly constant
module parameter	`UpperCamelCase`	truly modifiable by instantiation, not constant
derived localparam	`UpperCamelCase`	while not modified directly, still tracks module parameter
tuneable localparam	`UpperCamelCase`	while not expected to change upon final RTL version, is used by designer to explore the design space conveniently
true localparam constant	`ALL_CAPS`	Example `localparam OP_JALR = 8'hA0;`
enum member true constant	`ALL_CAPS`	Example `typedef enum ... { OP_JALR = 8'hA0;`
enum set member	`ALL_CAPS` or `UpperCamelCase`	Example `typedef enum ... { ST_IDLE, ST_FRAME_START, ST_DYN_INSTR_READ ...`, `typedef enum ... { StIdle, StFrameStart, StDynInstrRead...`. A collection of arbitrary values, could be either convention.

The units for a constant should be described in the symbol name, unless the constant is unitless or the units are "bits." For example, `FooLengthBytes`.

Example:

```
// package-scope
package my_pkg;

  parameter int unsigned NUM_CPU_CORES = 64;
  // reference elsewhere as my_pkg::NUM_CPU_CORES

endpackage
```

Parameterized Objects (modules, etc.)

Use `parameter` to parameterize, and `localparam` to declare module-scoped constants. Within a package, use `parameter`.

You can create parameterized modules, classes, and interfaces to facilitate design re-use.

Use the keyword `parameter` within the `module` declaration of a parameterized module to indicate what parameters the user is expected to tune at instantiation. The preferred naming convention for all parameters is `UpperCamelCase`. Some projects may choose to use `ALL_CAPS` to differentiate tuneable parameters from constants.

Derived parameters within the `module` declaration should use `localparam`. An example is shown below.

```
module modname #(
  parameter  int Depth  = 2048,         // 8kB default
  localparam int Aw     = $clog2(Depth) // derived parameter
) (
  ...
);

endmodule
```

`` `define `` and `defparam` should never be used to parameterize a module.

Use package parameters to transmit global constants through a hierarchy instead of parameters. To declare a constant whose scope is internal to the particular SystemVerilog module, use `localparam` instead.

Examples of when to use parameterized modules:

- When multiple instances of a module will be instantiated, and need to be differentiated by a parameter.
- As a means of specializing a module for a specific bus width.
- As a means of documenting which global parameters are permitted to change within the module.

Explicitly declare the type for parameters.

Use the type of the parameter to help constrain the legal range. E.g. `int unsigned` for general non-negative integer values, `bit` for boolean values. Any further restrictions on tuneable parameter values must be documented with assertions.

Tuneable parameter values should always have reasonable defaults.

For additional reading, see New Verilog-2001 Techniques for Creating Parameterized Models.

Macro Definitions

Macros should be ALL_CAPITALS with underscores.

Macros should be all capitals with underscores.

A **global define** is a tick-defined macro in a header file that is shared by all source files in a project. To reduce namespace collisions, global defines should be prefixed by the name of a group of related macros, followed by a pair of underscores:

```
// The following two constants are in the FOO namespace of the
// SN chip.
`define SN_FOO__ALPHA_BETA  5
`define SN_FOO__GAMMA_OMEGA 6
```

A **local define** is a tick-defined macro that should only be used within the scope of a single local file. It must be explicitly undefined after use, to avoid polluting the global macro namespace. To indicate that a macro is only meant to be used in the local scope, the macro name should be prefixed with a single underscore.

To ensure that local defines stay local, be careful not to `` `include `` other files between the macro definition and `` `undef ``.

Example:

```
`define _MAKE_THING(_x) \
    thing i_thing_##_x (.clk(clk), .i(i##_x) .o(o##_x));
`_MAKE_THING(a)
`_MAKE_THING(b)
`_MAKE_THING(c)
`undef _MAKE_THING
```

Suffixes

Suffixes are used in several places to give guidance to intent. The following table lists the suffixes that have special meaning.

Suffix(es)	Arena	Intent
`_e`	typedef	Enumerated types
`_t`	typedef	Other typedefs, including signal clusters
`_n`	signal name	Active low signal
`_n` , `_p`	signal name	Differential pair, active low and active high
`_d` , `_q`	signal name	Input and output of register
`_q2` , `_q3` , etc	signal name	Pipelined versions of signals; `_q` is one cycle of latency, `_q2` is two cycles, `_q3` is three, etc
`_i` , `_o` , `_io`	signal name	Module inputs, outputs, and bidirectionals

When multiple suffixes are necessary use the following guidelines:

- Guidance suffixes are added together and not separated by additional `_` characters (`_ni` not `_n_i`)
- If the signal is active low `_n` will be the first suffix
- If the signal is a module input/output the letters will come last.
- It is not mandatory to propagate `_d` and `_q` to module boundaries.

Example:

👍

```
module simple (
  input        clk_i,
  input        rst_ni,              // Active low reset

  // writer interface
  input [15:0] data_i,
  input        valid_i,
  output       ready_o,

  // bi-directional bus
  inout [7:0]  driver_io,          // Bi directional signal

  // Differential pair output
  output       lvds_po,            // Positive part of the differential signal
  output       lvds_no             // Negative part of the differential signal
);

  logic valid_d, valid_q, valid_q2, valid_q3;
  assign valid_d = valid_i; // next state assignment

  always_ff @(posedge clk or negedge rst_ni) begin
    if (!rst_ni) begin
      valid_q  <= '0;
      valid_q2 <= '0;
      valid_q3 <= '0;
    end else begin
      valid_q  <= valid_d;
      valid_q2 <= valid_q;
      valid_q3 <= valid_q2;
    end
  end

  assign ready_o = valid_q3; // three clock cycles delay

endmodule // simple
```

Enumerations

Name enumeration types `snake_case_e`. Name enumeration values `ALL_CAPS` or `UpperCamelCase`.

Always name `enum` types using `typedef`. The storage type of any enumerated type must be specified. For synthesizable enums, the storage type must be a 4-state data type (`logic` rather than `bit`).

Anonymous `enum` types are not allowed as they make it harder to use the type in other places throughout the project and across projects.

Enumeration type names should contain only lower-case alphanumeric characters and underscores. You must suffix enumeration type names with `_e`.

Enumeration value names (constants) should typically be `ALL_CAPS`, for example, `READY_TO_SEND`, to reflect their constant nature, especially for truly unchangeable values like defined opcode assignments. There are times when `UpperCamelCase` might be preferred, when the enumerated type's assigned value is effectively a don't care to the designer, like state machine values. See the conversation on constants for a discussion on how to think of this recommendation.

👍

```
typedef enum logic [7:0] {  // 8-bit opcodes
  OP_JALR = 8'hA0,
  OP_ADDI = 8'h47,
  OP_LDW  = 8'h0B
} opcode_e;
opcode_e op_val;
```

👍

```
typedef enum logic [1:0] {  // A 2-bit enumerated type
  ACC_WRITE,
  ACC_READ,
  ACC_PAUSE
} access_e; // new named type is created
access_e req_access, resp_access;
```

👍

```
typedef enum logic [1:0] {  // A 2-bit enumerated type
  AccWrite,
  AccRead,
  AccPause
} access_e; // new named type is created
access_e req_access, resp_access;
```

👎

```
enum {  // Typedef is missing, storage type is missing.
  Write,
  Read
} req_access, resp_access; // anonymous enum type
```

Signal Naming

Use `Lower_snake_case` when naming signals.

In this context, a **signal** is meant to mean a net, variable, or port within a SystemVerilog design.

Signal names may contain lowercase alphanumeric characters and underscores.

Signal names should never end with an underscore followed by a number (for example, `foo_1`, `foo_2`, etc.). Many synthesis tools map buses into nets using that naming convention, so similarly named nets can lead to confusion when examining a synthesized netlist.

Reserved Verilog or SystemVerilog-2012 standard keywords may never be used as names.

When interoperating with different languages, be mindful not to use keywords from other languages.

Use descriptive names

Names should describe what a signal's purpose is.

Use whole words. Avoid abbreviations and contractions except in the most common places. Favor descriptive signal names over brevity.

Prefixes

Use common prefixes to identify groups of signals that operate together. For example, all elements of an AXI-S interface would share a prefix: `foo_valid`, `foo_ready`, and `foo_data`.

Additionally, prefixes should be used to clearly label which signal is in which clock group for any module with multiple clocks. See the section on clock domains for more details.

Examples:

- Signals associated with controlling a blockram might share a `bram_` prefix.
- Signals that are synchronous with `clk_dram` rather than `clk` should share a `dram_` prefix.

Code example:

```
module fifo_controller (
  input         clk_i,
  input         rst_ni,

  // writer interface
  input [15:0]  wr_data_i,
  input         wr_valid_i,
  output        wr_ready_o,

  // reader interface
  output [15:0] rd_data_o,
  output        rd_valid_o,
  output [7:0]  rd_fullness_o,
  input         rd_ack_i,

  // memory interface:
  output [7:0]  mem_addr_o,
  output [15:0] mem_wdata_o,
  output        mem_we_o,
  input  [15:0] mem_rdata_i
);
```

This naming convention makes it easier to map port names onto similar signal names using simple and consistent rules. See the section on Hierarchical Consistency for more information.

Hierarchical consistency

The same signal should have the same name at any level of the hierarchy.

A signal that connects to a port of an instance should have the same name as that port. By proceeding in this manner, signals that are directly connected should maintain the same name at any level of hierarchy.

Exceptions to this convention are expected, such as:

- When connecting a port to an element of an array of signals.
- When mapping a generic port name to something more specific to the design. For example, two generic blocks, one with a `host_bus` port and one with a `device_bus` port might be connected by a `foo_bar_bus` signal.

In each exceptional case, care should be taken to make the mapping of port names to signal names as unambiguous and consistent as possible.

Clocks

All clock signals must begin with `clk`.

The main system clock for a design must be named `clk`. It is acceptable to use `clk` to refer to the default clock that the majority of the logic in a module is synchronous with.

If a module contains multiple clocks, the clocks that are not the system clock should be named with a unique identifier, preceded by the `clk_` prefix. For example: `clk_dram`, `clk_axi`, etc. Note that this prefix will be used to identify other signals in that clock domain.

Resets

Resets are active-low and asynchronous. The default name is `rst_n`.

Chip wide all resets are defined as active low and asynchronous. Thus they are defined as tied to the asynchronous reset input of the associated standard cell registers.

The default name is `rst_n`. If they must be distinguished by their clock, the clock name should be included in the reset name like `rst_domain_n`.

SystemVerilog allows either of the following syntax styles, but the style guide prefers the former.

```
// preferred
always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    q <= 1'b0;
  end else begin
    q <= d;
  end
end

// legal but not preferred
always_ff @(posedge clk, negedge rst_n) begin
  if (!rst_n) begin
    q <= 1'b0;
  end else begin
    q <= d;
  end
end
```

Language Features

Preferred SystemVerilog Constructs

Use these SystemVerilog constructs instead of their Verilog-2001 equivalents:

- `always_comb` is required over `always @*`.
- `logic` is preferred over `reg` and `wire`.
- Top-level `parameter` declarations are preferred over `` `define `` globals.

Package Dependencies

Packages must not have cyclic dependencies.

Package files may depend on constants and types in other package files, but there must not be any cyclic dependencies. That is: if package A depends on a constant from package B, package B must not depend on anything from package A. While cyclic dependencies are permitted by the SystemVerilog language specification, their use can break some tools.

For example:

```
package foo;

  // Package "bar" must not depend on anything in "foo":
  parameter int unsigned PageSizeBytes = 16 * bar::Kibi;

endpackage
```

Module Declaration

Use the Verilog-2001 full port declaration style, and use the format below.

Use the Verilog-2001 combined port and I/O declaration style. Do not use the Verilog-95 list style. The port declaration in the module statement should fully declare the port name, type, and direction.

The opening parenthesis should be on the same line as the module declaration, and the first port should be declared on the following line.

The closing parenthesis should be on its own line, in column zero.

Indentation for module declaration follows the standard indentation rule of two space indentation.

The clock port(s) must be declared first in the port list, followed by any and all reset inputs.

Example without parameters:

```
module foo (
  input              clk_i,
  input              rst_ni,
```

```
  input [7:0]        d_i,
  output logic [7:0] q_o
);
```

Example with parameters:

```
module foo #(
  parameter int unsigned Width = 8,
) (
  input                  clk_i,
  input                  rst_ni,
  input [Width-1:0]      d_i,
  output logic [Width-1:0] q_o
);
```

Do not use Verilog-95 style:


```
// WRONG:
module foo(a, b, c d);
input wire [2:0] a;
output logic b;
...
```

Parameterized Module Instantiation

Use named parameters for all instantiations.

When parameterizing an instance, specify the parameter using the named parameter style. An exception is if there is only one parameter that is obvious such as register width, then the instantiation can be implicit.

Indentation for module instantiation follows the standard indentation rule of two space indentation.

```
my_module #(
  .Height(5),
  .Width(10)
) my_module (
  ...etc...

my_reg #(16) my_reg0 (.clk_i, .rst_ni, .d_i(data_in), .q_o(data_out));
```

Do not specify parameters positionally, unless there is only one parameter and the intent of that parameter is obvious, such as the width for a register instance.

Do not use `defparam`.

Use named ports to fully specify all instantiations.

When connecting signals to ports for an instantiation, use the named port style, like this:

```
my_module i_my_instance (
  .clk_i (clk_i),
  .rst_ni(rst_ni),
  .d_i   (from_here),
  .q_o   (to_there)
);
```

If the port and the connecting signal have the same name, you can use the `.port` syntax (without parentheses) to indicate connectivity. For example:

```
my_module i_my_instance (
  .clk_i,
  .rst_ni,
  .d_i   (from_here),
  .q_o   (to_there)
);
```

All declared ports must be present in the instantiation blocks. Unconnected outputs must be explicitly written as no-connects (for example: `.output_port()`), and unused inputs must be explicitly tied to ground (for example: `.unused_input_port(8'd0)`)

`.*` is not permitted.

Do not use positional arguments to connect signals to ports.

Instantiate ports in the same order as they are defined in the module.

Do not instantiate recursively.

Modules may not instantiate themselves recursively.

Constants

It is recommended to use symbolicly named constants instead of raw numbers.

Try to give commonly used constants symbolic names rather than repeatedly typing raw numbers.

Local constants should always be declared using `localparam`.

Global constants should always be declared in a separate `.vh` or `.svh` include file.

For SystemVerilog code, global constants should always be declared as package parameters. For Verilog-2001 compatible code, top-level parameters are not supported and `` `define `` macros must be used instead.

Include the units for a constant as a suffix in the constant's symbolic name. The exceptions to this rule are for constants that are inherently unitless, or if the constant is describing the default unit type, "bits."

Example:

```
localparam int unsigned INTERFACE_WIDTH = 64;  // Bits
localparam int unsigned INTERFACE_WIDTH_BYTES = (INTERFACE_WIDTH + 7) / 8;
localparam int unsigned INTERFACE_WIDTH_64B_WORDS = (INTERFACE_WIDTH + 63) / 64;
localparam int unsigned IMAGE_WIDTH_PIXELS = 640;
localparam int unsigned MEGA = 1000 * 1000;  // Unitless
localparam int unsigned MEBI = 1024 * 1024;  // Unitless
localparam int unsigned SYSTEM_CLOCK_HZ = 200 * MEGA;
```

Signal Widths

Be careful about signal widths.

Always be explicit about the widths of number literals.

Examples:

```
localparam logic [3:0] bar = 4'd4;

assign foo = 8'd2;
```

```
localparam logic [3:0] bar = 4;

assign foo = 2;
```

Exceptions:

- When using parameterized widths, it is acceptable to simply use `1'b1` (e.g. when incrementing) rather than contrivances such as `{{(Bus_width-1){1'b0}},1'b1}`
- It is acceptable to use the '0 construct to create an automatic correctly sized zero.
- Literals assigned to integer variants (e.g. byte, shortint, int, integer, and longint) do not need an explicit width.

Port connections on module instances must always match widths correctly.

It is recommended to use explicit widths, rather than relying on Verilog's implicit zero-extension and truncation operations, whenever practical.

Examples:

👍

```
my_module i_module (
  .thirty_two_bit_input({16'd0, sixteen_bit_word})
);
```

👎

```
my_module i_module (
  // Incorrectly implicitly extends from 16 bit to 32 bit
  .thirty_two_bit_input(sixteen_bit_word)
);
```

Do not use multi-bit signals in a boolean context.

Rather than letting boolean operations and if expressions reduce a multi-bit signal to a single bit, explicitly compare the multi-bit signal to 0. The implicit conversion can hide subtle logic bugs.

Examples;

👍

```
logic [3:0] a, b;
logic out;

assign out = (a != '0) && (b == '0);

always_comb begin
  if (a != '0)
    ...
  else
    ...
end
```

👎

```
logic [3:0] a, b;
logic out;

// Incorrect because it implicitly converts 4-bit signals to 1-bit before AND.
// Also, !b is different from ~b and can be hard to catch.
assign out = a && !b;

// Incorrect use of a multi-bit signal in an if expression
always_comb begin
  if (a)
    ...
  else
    ...
end
```

Bit Slicing

Only use the bit slicing operator when the intent is to refer to a portion of a bit vector.

Examples:

👍

```
logic [7:0] a, b;
logic [6:0] c;

assign a = 8'd7;       // good

assign a[7:1] = 7'd5;  // good - it's partial assignment.
assign a = b;          // good - the parser would warn on width mismatch.
```

👎

```
logic [7:0] a, b;

assign a[7:0] = 8'd7;  // BAD - redundant and can mask linter warnings.
assign a = b[7:0];     // BAD - redundant and masks linter warnings.
```

Handling Width Overflow

Beware of shift operations, which can produce a result wider than the operand. Bit-selection and concatenation may be clearer than shifting by a constant amount.

Addition and negation operations produce a result one bit wider than the operands, due to carry. An allowable exception to the rule about matching widths is to silently drop the carry on assignment.

Example:

```
assign abc = abc + 4'h1;
```

Blocking and Non-blocking Assignments

Sequential logic must use non-blocking assignments. Combinational blocks must use blocking assignments.

Never mix assignment types within a block declaration.

A sequential block (a block that latches state on a clock edge) must exclusively use non-block assignments, as defined in the Sequential Logic section below.

Purely combinational blocks must exclusively use blocking assignments.

This is one of Cliff Cumming's Golden Rules of Verilog.

Delay Modeling

Do not use `#delay` in synthesizable design modules.

Synthesizable design modules must be designed around a zero-delay simulation methodology. All forms of `#delay`, including `#0`, are not permitted.

Sequential Logic (Latches)

The use of latches is discouraged - use flip-flops when possible.

Unless absolutely necessary, use flops/registers instead of latches.

If you must use a latch, use `always_latch` over `always`, and use non-blocking assignments (`<=`). Never use blocking assignments (`=`).

Sequential Logic (Registers)

Use the standard format for declaring sequential blocks.

In a sequential always block, only use non-blocking assignments (`<=`). Never use blocking assignments (`=`).

Designs that mix blocking and non-blocking assignments for registers simulate incorrectly because some simulators process some of the blocking assignments in an always block as occurring in a separate simulation event as the non-blocking assignment. This process makes some signals jump registers, potentially leading to total protonic reversal. That's bad.

Sequential statements for state assignments should only contain reset values and a next-state to state assignment, use a separate combinational-only block to generate that next-state value.

A correctly implemented 8-bit register with an initial value of "0xAB" would be implemented:

👍

```
logic foo_en;
logic [7:0] foo_q, foo_d;

always_ff @(posedge clk or negedge rst_ni) begin
  if (!rst_ni) begin
    foo_q <= 8'hab;
  end else if (foo_en) begin
    foo_q <= foo_d;
  end
end
```

Do not allow multiple non-blocking assignments to the same bit.

Example:

👎

```
if (cond1) begin
  abc <= 4'h1;
end

if (cond2) begin
  abc <= 4'h2;
end
```

If both cond1 and cond2 are true, the Verilog standard says that the second assignment will take effect, but this is a style violation.

Even if `cond1` and `cond2` are mutually exclusive, make the second `if` into an `else if`.

Exception: It is fine to set default values first, then specific values. However, it is preferred to do this work in a separate combinational block with explicit blocking assignments.

Example:

```
always_ff @(posedge clk or negedge rst_ni) begin
  if (!rst_ni) begin
    state_q <= StIdle;
  end else begin
    state_q <= state_d;
  end
end

always_comb begin
  state_d = state_q;    // default assignment next state is present state
  unique case (state_q)
    StIdle: state_d = StInit;       // Idle State move to Init
    StInit: begin                   // Initialize calculation
      if (conditional) begin
        state_d = StIdle;
      end else begin
        state_d = StCalc;
      end
    end
    StCalc: begin                   // Perform calculation
      if (conditional) begin
        state_d = StResult;
      end
    end
    StResult: state_d = Idle;
    default: ;
  endcase
end
```

Keep work in sequential blocks simple. If a sequential block becomes sufficiently complicated, consider splitting the combinational logic into a separate combinational (`always_comb`) block. Ideally, sequential blocks should contain only a register instantiation, with perhaps a load enable or an increment.

Don't Cares (`X` 's)

The use of `x` literals in RTL code is strongly discouraged. RTL must not assert `x` to indicate "don't care" to synthesis in any case. In order to flag and detect invalid conditions, rather than assign and propagate `x` values, designs should fully define all signal values and make extensive use of SVAs to indicate the invalid conditions.

If not strictly controlled, the use of `X` assignments in RTL to flag invalid or don't care conditions can lead to simulation/synthesis mismatches.

Instead of assigning and propagating `X` in order to flag and detect invalid conditions, it is encouraged to make **extensive use of SVAs**. The added benefits of this design practice are that:

- No special code style is required to properly propagate `X` conditions,
- The chance of accidentally introducing simulation/synthesis mismatches is systematically reduced,
- Simulation fails quickly and less signal backtracking is needed to root-cause bugs,
- In several cases, formal property verification (FPV) can be used to prove whether these SVAs can always be fulfilled,
- In a security context, deterministic/defined behavior is desired, even for illegal/invalid/unreachable input combinations (sometimes stated more tersely as "for security-critical designs, there are no don't-cares").

The solution presented here has similarities with the approaches presented in "Being Assertive With Your X" by Don Mills.

Note that although don't cares can be used to indicate possible optimization opportunities to the synthesis tool, it is debatable whether the gains in logic reduction are significant enough to outweigh the possible synthesis mismatch issues that the use of `X` literals may entail (especially with the gate-counts available in today's technologies).

Catching errors where invalid values are consumed

For an internally-generated signal that could be invalid (but not driven to `X`) and is used to trigger some action (such as a register write-enable), it is recommened to add an assert to check that when the enable is true, the signal is valid. This triggers a simple to diagnose failure when an invalid value has been accidentally used.

```
logic reg_addr;
logic reg_wr_en;

// internal logic which generates reg_addr/reg_wr_en reg_en_addr will never
// be X but must be ignored if reg_wr_en == 0
assign reg_addr = ...
assign reg_wr_en = ...

...

// trigger some specific action when a certain register is written
logic special_reg_en;

assign special_reg_en = (reg_addr == SPECIAL_REG_ADDR) & reg_wr_en;

// Aim to keep RHS of implication as broad as possible
`ASSERT(NoSpecialRegEnWithoutRegEn, special_reg_en |-> reg_wr_en);
```

Where the value and its validity signal are generated by a DV environment which will drive `X` on invalid signals an `` `ASSERT_KNOWN `` suffices.

```
module mymod (
  input [7:0] external_addr_i,
  input       external_wr_en_i
);

  logic special_action_en;

  assign special_action_en =
    (external_addr_i == SPECIAL_ADDR) & external_wr_en_i;

  `ASSERT_KNOWN(special_action_en);

endmodule
```

Specific Guidance on Case Statements and Ternaries

To comply with this style, RTL must place `` `ASSERT_KNOWN `` assertions on all module outputs, with the exception of signals that may implicitly be `X` at the beginning of the simulation, such as FIFO, SRAM or register file outputs.

```
module mymod (
  input        ina_i,
  input        inb_i,
  output logic out_o
);
  assign out_o = ina_i ^ inb_i;
  `ASSERT_KNOWN(OutKnown_A, out_o, clk_i, !rst_ni)
endmodule : mymod
```

Further, it is encouraged to add assertions to the signals forming conditions of case statements, ternaries or if/else statements. The assertion style is at the designer's discretion, and can range from simple `` `ASSERT_KNOWN `` to fully functional assertions, as shown in the following examples:

```
typedef enum logic [1:0] {mode0, mode1, mode2} state_e;
state_e sel;

// encouraged
`ASSERT_KNOWN(SelKnown_A, sel)
always_comb begin
  out0 = '0;
  out1 = '0;
  unique case (sel)
    mode1: out0 = foo;
    mode2: out1 = bar;
    default: ;
  endcase
```

```
end

// optional, but more explicit
// not always applicable
`ASSERT(MainFsmCase_A, sel inside {mode0, mode1, mode2}, clk_i, !rst_ni)
always_comb begin
  out0 = '0;
  out1 = '0;
  unique case (sel)
    mode1: out0 = foo;
    mode2: out1 = bar;
    default: ;
  endcase
end
```

In the context of ternary statements, the following are encouraged examples:

```
// encouraged
`ASSERT_KNOWN(ModeKnown_A, mode_i, clk_i, !rst_ni)
`ASSERT_KNOWN(LenKnown_A, len_i, clk_i, !rst_ni)
// assign '0 for all other combinations
assign val = (mode_i == ENC)                    ? 8'h01 :
             (mode_i == DEC && len_i == LEN128) ? 8'h36 :
             (mode_i == DEC && len_i == LEN192) ? 8'h80 :
             (mode_i == DEC && len_i == LEN256) ? 8'h40 : 8'h00;

// optional, but more explicit
`ASSERT(ValSelValid_A, mode_i == ENC || mode_i == DEC &&
    len_i inside {LEN128, LEN192, LEN256}, clk_i, !rst_ni)
// using one of the valid outputs for other combinations (saves logic)
assign val = (mode_i == ENC)                    ? 8'h01 :
             (mode_i == DEC && len_i == LEN128) ? 8'h36 :
             (mode_i == DEC && len_i == LEN192) ? 8'h80 :
             (mode_i == DEC && len_i == LEN256) ? 8'h40 : 8'h01;
```

Note that there are cases where the input into a case or ternary could be `X` but only under circumstances where it doesn't matter as the output will be ignored as some valid signal that qualifies the input is not set. For example the input may be fed directly from a memory or from a top-level input that a DV environment drives to `X`. A plain `` `ASSERT_KNOWN `` will not work under these circumstances and it is appropriate to use an assert with some qualifying valid instead:

```
`ASSERT(AddrKnownIfValid, addr_valid |-> !$isunknown(addr))
always_comb begin
  out = '0
  unique case (addr[1:0])
    ConstAddr1: out = foo;
    ConstAddr2: out = bar;
    default:    out = baz;
  endcase
end
```

The aim should be to make the qualifying valid signal as wide reaching as possible rather than narrowing down the `X` check more than is required:

👎

```
`ASSERT(AddrKnownIfValid,
  addr_valid & internal_condition_1 & internal_condition_2 |->
  !$isunknown(addr))
```

Dynamic Array Indexing

It should be noted that dynamic array indexing operations can implicitly lead to `X`. This should be avoided if possible by either aligning indexed arrays to powers of 2 or by adding guarding if statements around the indexing operation. These solutions are illustrated in the following examples.

👎

```
logic selected;
logic [3:0] idx;
logic [11:0] foo; // problematic

assign foo = {12'b1010_1111_0000};
assign selected = foo[idx];
```

```
logic selected;
logic [3:0] idx;
logic [15:0] foo; // aligned to powers of two

assign foo = {4'b0000, 12'b1010_1111_0000};
assign selected = foo[idx];
```

```
logic selected;
logic [3:0] idx;
logic [11:0] foo;

assign foo = {12'b1010_1111_0000};

// guarding if statement
assign selected = (idx < $bits(foo)) ? foo[idx] : 1'b0;
```

Combinational Logic

Avoid sensitivity lists, and use a consistent assignment type.

Use `always_comb` for SystemVerilog combinational blocks. Use `always @*` if only Verilog-2001 is supported. Never explicitly declare sensitivity lists for combinational logic.

Prefer assign statements wherever practical.

Example:

```
assign final_value = xyz ? value_a : value_b;
```

Where a case statement is needed, enclose it in its own `always_comb` block.

Synthesizable combinational logic blocks should only use blocking assignments.

Do not use three-state logic (`Z` state) to accomplish on-chip logic such as muxing.

Do not infer a latch inside a function, as this may cause a simulation / synthesis mismatch.

Case Statements

Avoid case-modifying pragmas. ***`unique case`*** ***is the best practice. Always define a default case.***

Never use either the `full_case` or `parallel_case` pragmas. These pragmas can easily cause synthesis-simulation mismatches.

Here is an example of a style-compliant full case statement:

```
always_comb begin
  unique casez (select)
    3'b000: operand = accum0 >> 0;
    3'b001: operand = accum0 >> 1;
    3'b010: operand = accum1 >> 0;
    3'b011: operand = accum1 >> 1;
    3'b1??: operand = regfile[select[1:0]];
    default: operand = '0; // assign a default
  endcase
end
```

The `unique` prefix is recommended before all case statements, as it creates simulation assertions that can catch certain mistakes. In some cases, `priority` may be used instead of `unique`, though in such cases, cascaded ternary structures should be the preferred way of representing priority encoders as they are a more readable representation for priority encoders.

Be sure to use `unique case` correctly. In particular, make sure that:

- a `default:` statement is **always** included in order to avoid accidental inference of latches, even if all cases are covered. In simulation, a case expression that evaluates to `X` will not match any case and will behave as a latch, leading to different behavior than synthesis if no default is specified.

- if no default assignments are given before the case statement as shown in the example above, any variables assigned in one case item must be assigned in all case items, including the `default:`. Failing to do this can lead to a simulation-synthesis mismatch as described in Don Mills' paper.

The following is a different example showing a style-compliant case statement variant that is frequently used for describing the next-state logic of a finite state machine. What is different from the previous example is that the default assignments are put before the `unique case` block, thus making it possible to omit common assignments in the individual cases further below. If it weren't for the common default assignments before the case statement, all variables would have to be assigned a value in all cases and in the `default:` in order to prevent simulation-synthesis mismatches.

```
always_comb begin
  // common default assignments
  state_d = state_q;
  outa = 1'b0;
  outb = 1'b0;
  outc = 1'b0;

  unique case (state_q)
    Idle: begin
      state_d = Work;
      outa = in0;
    end
    Work: begin
      state_d = Wait;
      outb = in1;
    end
    Wait: begin
      state_d = Idle;
      outc = in2;
    end
    // always include a default case
    // empty default permissible due to defaults before case block
    default: ;
  endcase
end
```

Wildcards in case items

Use `case` instead of `casez` whenever wildcard operator behavior is not required. When wildcard behavior is needed, use `casez`.

When expressing a wildcard in a case item, use the '?' character since it more clearly expresses the intent.

`casex` should not be used. `casex` implements a symmetric wildcard operator such that an `X` in the case expression may match one or more case items. `casez` only treats high-impedance states (`Z` or `?`) as a wildcard, and performs exact matches for undriven `X` inputs. While this does not completely fix the problems with symmetric wildcard matching, it is harder to accidentally produce a `Z` input than an `X` input, so this form is preferred.

References:

- Don Mills, Yet Another Latch and Gotchas Paper
- Clifford Cummings, full_case parallel_case, the Evil Twins of Verilog Synthesis
- Clifford Cummings, SystemVerilog's priority & unique
- Sutherland, Mills, and Spear, Gotcha Again: More Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know

Generate Constructs

Always name your generated blocks.

When using a generate construct, always explicitly name each block of generated code. Name each possible outcome of the generating if statement, and name the iterated block of a generating for statement.

This ensures that generated hierarchical signal names are consistent across different tools.

Generate and all named code blocks should use `lower_snake_case`. A space should be placed between `begin` and the code block name.

Example of a conditional generate construct:

```
if (TypeIsPosedge) begin : posedge_type
  always_ff @(posedge clk) foo <= bar;
end else begin : negedge_type
  always_ff @(negedge clk) foo <= bar;
end
```

Example of a loop generate construct:

```
for (genvar ii = 0; ii < NumberOfBuses; ii++) begin : my_buses
  my_bus #(.index(ii)) i_my_bus (.foo(foo), .bar(bar[ii]));
end
```

Do not wrap a generate construct with an additional `begin` block.

Do not use generate regions { `generate` , `endgenerate` }.

Signed Arithmetic

Use the available signed arithmetic constructs wherever signed arithmetic is used.

When it's necessary to convert from unsigned to signed, use the `signed'` cast operator (`$signed` in Verilog-2001).

If any operand in a calculation is unsigned, Verilog implicitly casts all operands to unsigned and generates a warning. There should not be any signed-to-unsigned warnings from either the simulation or synthesis tools if all unsigned variables are properly casted.

Example of implicit signed-to-unsigned casting:

```
logic signed [7:0]  a;
logic               incr;
logic signed [15:0] sum1, sum2;
initial begin
  a = 8'sh80;
  incr = 1'b1;
  sum1 = a + incr;                  // sum1 = 16'h0081
  sum2 = a + signed'({1'b0, incr});  // sum2 = 16'hFF81
end
```

In the above example, the fact that `incr` is unsigned causes `a` to be evaluated as unsigned as well. The `sum1` evaluation is surprising and is flagged by a warning that should not be ignored.

Number Formatting

Prefix printed binary numbers with `0b`. Prefix printed hexadecimal numbers with `0x`. Do not use prefixes for decimal numbers.

When formatting text representations of numbers for log files, make it clear what data you are including.

Make the base of a printed number clear. Only print decimal numbers without modifiers. Use a `0x` prefix for hexadecimal and `0b` prefix for binary.

Decode individual fields of large structures individually, instead of expecting the user to manually decode raw values.

```
$display("0x%0x", some_hex_value);
$display("0b%0b", some_binary_value);
$display("%0d",   some_decimal_value);
```

```
$display("%0x",   some_hex_value);
$display("%0b",   some_binary_value);
$display("0d%0d", some_decimal_value);
```

When assigning constant values, it is preferred to use underscore notation for hex or binary bit strengths of length beyond 8 for better readability. Zero prepending is not required unless it improves readability. Declare constants in the format (binary, hex, decimal) they are typically displayed in.

```
logic [15:0] val0, val1, val2;
logic [39:0] addr0, addr1;

always_comb begin
  val0 = 16'h0;
  if (condition1) begin
    val1  = 16'b0010_0011_0000_1101;
```

```
    val2  = 16'b0010_1100_0000_0000;
    addr1 = 40'h00_1fc0_0000;
    addr2 = 40'h00_efc0_0000;
  end else begin
    val0  = 16'hffff;
    val1  = 16'b1010_0011_0110_1001;
    val2  = 16'b1110_1100_1111_0110;
    addr1 = 40'h40_8000_0000;
    addr2 = 40'h41_c000_0000;
  end
end
```

Functions and Tasks

The following section applies to synthesizable RTL only. See the Coding Style Guide for Design Verification for DV usage.

In synthesizable RTL the use of functions is allowed, provided they are declared `automatic`. Tasks should not be used.

Functions must be declared in either a package or inside a module. A package is appropriate where the function relates to other definitions in the package and could be useful to multiple modules (even if it's currently only used by one). A module is appropriate where the function specifically relates to the internals of that module.

Functions should aim to conceptually represent a reusable block of combinational logic.

Storage types must be explicitly declared for all arguments and the function return value. All types must be 4-state data types, either `logic` or types derived from `logic` (such as appropriate `struct`, `enum` or `typedef` types).

Do not use `output`, `inout`, or `ref` on function arguments. All functions should only consume inputs and produce one output. `input` is the default and is not required on the function arguments.

👎

```
// - Doesn't have explicit storage type on `a` or `b` or return type
// - `b` being used as `output` argument
// - `input` not required on `a`
function automatic [2:0] foo(input [2:0] a, [2:0] b);
  b = b + 1;
  return a + b;
endfunction
```

👎

```
// - Doesn't have explicit storage type on `a`, `b` or `c`
// - Uses `output` on `c`
// - `input` not required on `a` and `b`
function automatic logic [2:0] foo(input [2:0] a, input [2:0] b, output [2:0] c);
  c = a - b;
  return a + b;
endfunction
```

👎

```
// - Uses 2-state data type `int` for `a`
function automatic logic [2:0] foo(int a, logic [2:0] b);
  return a + b;
endfunction
```

👍

```
function automatic logic [2:0] foo(logic [2:0] a, logic [2:0] b);
 return a ^ b;
endfunction
```

👍

```
typedef logic [2:0] bar_t;

typedef struct packed {
  logic [2:0] field;
} baz_t;

function automatic logic [2:0] foo(bar_t a, baz_t b);
```

```
  return a + b.field;
endfunction
```

Data should be returned from a function using an explicit `return result` style. Do not use a `function_name = result` style.

```
function automatic logic [2:0] foo(logic [2:0] a, logic [2:0] b);
  if (a == 3'd2) begin
    foo = b;
  end else begin
    foo = a ^ b;
  end
endfunction
```



```
function automatic logic [2:0] foo(logic [2:0] a, logic [2:0] b);
  logic [2:0] result;

  if (a == 3'd2) begin
    result = b;
  end else begin
    result = a ^ b;
  end

  return result;
endfunction
```

All local variables must be assigned in all code paths, either through an initial assignment or through the use of `else` and `default:` for `if` and `case` statements.

```
function automatic logic [2:0] foo(logic [2:0] a, logic [2:0] b);
  logic [2:0] local_var_1;
  logic [2:0] local_var_2;

  local_var_1 = 3'd0;

  if (a == 0) begin
    local_var_1 = 3'd2;
  end

  unique case(b)
    3'd0:    local_var_2 = 3'd1;
    3'd1:    local_var_2 = 3'd3;
    default: local_var_2 = 3'd0;
  endcase

  return local_var_1 + local_var_2;
endfunction
```

```
function automatic logic [2:0] foo(logic [2:0] a, logic [2:0] b);
  logic [2:0] local_var_1;
  logic [2:0] local_var_2;

  if (a == 0) begin
    local_var_1 = 3'd2;
  end

  unique case(b)
    3'd0:    local_var_2 = 3'd1;
    3'd1:    local_var_2 = 3'd3;
  endcase

  return local_var_1 + local_var_2;
endfunction
```

Problematic Language Features and Constructs

These language features are considered problematic and their use is discouraged unless otherwise noted:

- Interfaces.
- The `alias` statement.

Floating begin-end blocks

The use of generate blocks other than `for` loop, `if`, or `case` generate constructs is not LRM compliant. While such usage might be accepted by some tools, this guide prohibits such "bare" generate blocks. Note that the similar "sequential block" construct is LRM compliant and allowed.

👎

```
module foo (
  input bar,
  output foo
);
  begin // illegal generate block
    assign foo = bar;
  end
endmodule
```

Design Conventions

Summary

The key ideas in this section include:

- Declare all signals and use `logic`: `logic foo;`
- Packed arrays are little-endian: `logic [7:0] byte;`
- Unpacked arrays are big-endian: `byte_t arr[0:N-1];`
- Prefer to register module outputs.
- Declare FSMs consistently.

Declare all signals

Do not rely on inferred nets.

All signals **must** be explicitly declared before use. All declared signals must specify a data type. A correct design contains no inferred nets.

Use `logic` for synthesis

Use `logic` for synthesis. `wire` is allowed when necessary.

All signals in synthesizable RTL must be implemented in terms of 4-state data types. This means that all signals must ultimately be constructed of nets with the storage type of `logic`. While SystemVerilog does provide other data primitives with 4-state storage (ie. `integer`), those primitives are prone to misunderstandings and misuse.

For example:

👍

```
logic signed [31:0] x_velocity;  // say what you mean: a signed 32-bit integer.
typedef logic [7:0] byte_t;
```

👎

```
bit signed [63:0] stars_in_the_sky;  // 2-state logic doesn't belong in RTL
int grains_of_sand;  // Or wait, did I mean integer?  Easy to confuse!
```

It is permissible to use wire as a short-hand to both declare a net and perform continuous assignment. Take care not to confuse continuous assignment with initialization. For example:

👍

```
wire [7:0] sum = a + b;  // Continuous assignment

logic [7:0] acc = '0;  // Initialization
```

There are exceptions for places where `logic` is inappropriate. For example, nets that connect to bidirectional (`inout`) ports must be declared with `wire`. These exceptions should be justified with a short comment.

It is permissible for DV (Design Verification) to make use of 2-state logic, but all interfaces between 4-state and 2-state signals must assert a check for `X` on the 4-state net before resolving to a 2-state variable.

Logical vs. Bitwise

Prefer logical constructs for logical comparisons, bit-wise for data.

Logical operators (`!`, `||`, `&&`, `==`, `!=`) should be used for all constructs that are evaluating logic (true or false) values, such as if clauses and ternary assignments. Prefer bit-wise operators (`~`, `|`, `&`, `^`) for all data constructs, even if scalar. Exceptions can be made where it is clear that the evaluated expression is to be used in a logical context.

👍

```
always_ff @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    reg_q <= '0;
  end else begin
    reg_q <= reg_d;
  end
end

always_comb begin
  if (bool_a || (bool_b && !bool_c) begin
    x = 1'b1;
  end else begin
    x = 1'b0;
end

assign z = ((bool_a != bool_b) || bool_c) ? a : b;
assign y = (a & ~b) | c;
```

👎

```
always_ff @(posedge clk_i or negedge rst_ni) begin
  if (~rst_ni) begin
    reg_q <= '0;
  end else begin
    reg_q <= reg_d;
  end
end

always_comb begin
  if (bool_a | (bool_b & ~bool_c) begin
    x = 1'b1;
  end else begin
    x = 1'b0;
end

assign z = ((bool_a ^ bool_b) | bool_c) ? a : b;
assign y = (a && !b) || c;
```

👍

```
// allowed logical assignment for boolean test
assign request_valid = !fifo_empty && data_available;

always_comb begin
  if (request_valid) begin
    output_valid = 1'b1;
  end else begin
    output_valid = 1'b0;
  end
end
```

Packed Ordering

Bit vectors and packed arrays must be little-endian.

When declaring bit vectors and packed arrays, the index of the most-significant bound (left of the colon) must be greater than or equal to the least-significant bound (right of the colon).

This style of bit vector declaration keeps packed variables little-endian.

For example:

```
typedef logic [7:0] u8_t;
logic [31:0] u32_word;
u8_t [1:0] u16_word;
u8_t byte3, byte2, byte1, byte0;
assign u16_word = {byte1, byte0};
assign u32_word = {byte3, byte2, u16_word};
```

Unpacked Ordering

Unpacked arrays must be big-endian.

Declare unpacked arrays in big-endian fashion (for instance, `[n:m]` where `n <= m`). Never declare an unpacked array in little-endian order, such as `[size-1:0]`.

Declare zero-based unpacked arrays using the shorter notation `[size]`. It is understood that `[size]` is equivalent to the big-endian declaration `[0:size-1]`.

```
logic [15:0] word_array[3] = '{word0, word1, word2};
```

Finite State Machines

State machines use an enum to define states, and be implemented with two process blocks: a combinational block and a clocked block.

Every state machine description has three parts:

1. An enum that declares and describes the states.
2. A combinational process block that decodes state to produce next state and other combinational outputs.
3. A clocked process block that updates state from next state.

Enumerating States

The enum statement for the state machine should list each state in the state machine. Comments describing the states should be deferred to case statement in the combinational process block, below.

States should be named in `UpperCamelCase`, like other enumeration constants.

Barring special circumstances, the initial idle state of the state machines will be named `Idle` or `StIdle`. (Alternate names are acceptable if they improve clarity.)

Ideally, each module should only contain one state machine. If your module needs more than one state machine, you will need to add a unique prefix (or suffix) to the states of each state machine, to distinguish which state is associated with which state machine. For example, a module with a "reader" machine and a "writer" machine might have a `StRdIdle` state and a `StWrIdle` state.

Combinational Decode of State

The combinational process block should contain:

- A case statement that decodes state to produce next state and combinational outputs. For clarity, only cases where the output value deviates from the default should be coded.
- Before the case statement should be a block of code that defines default values for every combinational output, including "next state."
- The default value for the "next state" variable should be the current state. The case statement that decodes state will then only assign to "next state" when transitioning between states.
- Within the case statement, each state alternative should be preceded with a comment that describes the function of that state within the state machine.

The State Register

No logic except for reset should be performed in this process. The state variable should latch the value of the "next state" variable.

Other Guidelines

When possible, try to choose state names that differ near the beginning of their name, to make them more readable when viewing waveform traces.

Example

```
// Define the states
typedef enum {
  StIdle, StFrameStart, StDynInstrRead, StBandCorr, StAccStoreWrite, StBandEnd
} alcor_state_e;

alcor_state_e alcor_state_d, alcor_state_q;

// Combinational decode of the state
always_comb begin
  alcor_state_d = alcor_state_q;
  foo = 1'b0;
  bar = 1'b0;
  bum = 1'b0;
  unique case (alcor_state_q)
    // StIdle: waiting for frame_start
    StIdle:
      if (frame_start) begin
        foo = 1'b1;
        alcor_state_d = StFrameStart;
      end
    // StFrameStart: Reset accumulators
    StFrameStart: begin
      // ... etc ...
    end
    // may be empty or used to catch parasitic states
    default: alcor_state_d = StIdle;
  endcase
end

// Register the state
always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    alcor_state_q <= StIdle;
  end else begin
    alcor_state_q <= alcor_state_d;
  end
end
```

Active-Low Signals

The `_n` suffix indicates an active-low signal.

If active-low signals are used, they must have the `_n` suffix in their name. Otherwise, all signals are assumed to be active-high.

Differential Pairs

Use the `_p` and `_n` suffixes to indicate a differential pair.

For example, `in_p` and `in_n` comprise a differential pair set.

Delays

Signals delayed by a single clock cycle should end in a `_q` suffix.

If one signal is only a delayed version of another signal, the `_q` suffix should be used to indicate this relationship.

If another signal is then delayed by another clock cycle, the next signal should be identifed with the `_q2` suffix, and then `_q3` and so on.

Example:

```
always_ff @(posedge clk) begin
  data_valid_q <= data_valid_d;
  data_valid_q2 <= data_valid_q;
  data_valid_q3 <= data_valid_q2;
end
```

Wildcard import of packages

The wildcard import syntax, e.g. `import ip_pkg::*;` is only allowed where the package is part of the same IP as the module that uses that package. Wildcard import statement must be placed in the module header or in the module body.

```
// mod_a_pkg.sv and mod_a.sv are in the same IP.
// Packages can be imported in the module declaration if access to
```

```
// unqualified types is needed in the port list.

// mod_a_pkg.sv
package mod_a_pkg;

  typedef struct packed {
    ...
  } a_req_t;
endpackage

// mod_a.sv
module mod_a
  import mod_a_pkg::*;
(
  ...
  a_req_t a_req,
  ...
);

endmodule
```

👍

```
// mod_a
module mod_a ();

  // mod_a_pkg.sv and mod_a.sv are in the same IP.
  import mod_a_pkg::*;

  ...

  a_req_t a_req;

endmodule
```

Other than the cases above, wildcard imports are not allowed. For instance, the example below may create a name collision in the module following `mod_a` in the source list.

👎

```
// mod_a.sv
import mod_a_pkg::*; // not allowed: imported to $root scope.

module mod_a ();

endmodule
```

Other bad examples:

👎

```
// wildcard import for other packages outside of the IP
module mod_a import mod_b_pkg::*; ();
```

👎

```
module mod_a ();

  // not allowed: wildcard import of a package from a different IP
  import mod_b_pkg::*;

endmodule
```

Assertion Macros

It is encouraged to use SystemVerilog assertions (SVAs) throughout the design to check functional correctness and flag invalid conditions. In order to increase productivity and keep the assertions short and concise, the following assertion macros can be used:

```
// immediate assertion, to be placed within a process.
`ASSERT_I(<name>, <property>)
// immediate assertion wrapped within an initial block. can be used for things
// like parameter checking.
`ASSERT_INIT(<name>, <property>)
```

```
// concurrent assertion to be used for functional assertions.
`ASSERT(<name>, <property>, <clk>, <reset condition>)
// concurrent assertion that checks that a signal has a known value after reset
// (i.e. that the signal is not `X`).
`ASSERT_KNOWN(<name>, <signal>, <clk>, <reset condition>)
```

An implementation of these macros (including other useful variations thereof) can be found here:
https://github.com/lowRISC/opentitan/blob/master/hw/ip/prim/rtl/prim_assert.sv

A Note on Security Critical Applications

For security critical applications, the names of the assertion macros involved in guarding case statements and ternaries shall be postfixed with `_SEC`. This enables security-specific post-processing of these statements at a later stage in the design process. In terms of functionality these macros should be identical to the original assertions, i.e.,

```
`define ASSERT_SEC `ASSERT
`define ASSERT_I_SEC `ASSERT_I
`define ASSERT_KNOWN_SEC `ASSERT_KNOWN
```

More security assertion and coding style guidance will be given in a separate document, soon.

Appendix - Condensed Style Guide

This is a short summary of the Comportable style guide. Refer to the main text body for explanations examples, and exceptions.

Basic Style Elements

- Use SystemVerilog-2012 conventions, files named as module.sv, one file per module
- Only ASCII, **100** chars per line, **no** tabs, **two** spaces per indent for all paired keywords.
- C++ style comments `//`
- For multiple items on a line, **one** space must separate the comma and the next character
- Include **whitespace** around keywords and binary operators
- **No** space between case item and colon, function/task/macro call and open parenthesis
- Line wraps should indent by **four** spaces
- `begin` must be on the same line as the preceding keyword and end the line
- `end` must start a new line

Construct Naming

- Use **lower_snake_case** for instance names, signals, declarations, variables, types
- Use **UpperCamelCase** for tunable parameters, enumerated value names
- Use **ALL_CAPS** for constants and define macros
- Main clock signal is named `clk`. All clock signals must start with `clk_`
- Reset signals are **active-low** and **asynchronous**, default name is `rst_n`
- Signal names should be descriptive and be consistent throughout the hierarchy

Suffixes for signals and types

- Add `_i` to module inputs, `_o` to module outputs or `_io` for bi-directional module signals
- The input (next state) of a registered signal should have `_d` and the output `_q` as suffix
- Pipelined versions of signals should be named `_q2`, `_q3`, etc. to reflect their latency
- Active low signals should use `_n`. When using differential signals use `_p` for active high
- Enumerated types should be suffixed with `_e`
- Multiple suffixes will not be separated with `_`. `n` should come first `i`, `o`, or `io` last

Language features

- Use **full port declaration style** for modules, any clock and reset declared first
- Use **named parameters** for instantiation, all declared ports must be present, no `.*`
- Top-level parameters is preferred over `` `define `` globals
- Use **symbolically named constants** instead of raw numbers
- Local constants should be declared `localparam`, globals in a separate **.svh** file.
- `logic` is preferred over `reg` and `wire`, declare all signals explicitly
- `always_comb`, `always_ff` and `always_latch` are preferred over `always`

- Interfaces are discouraged
- Sequential logic must use **non-blocking** assignments
- Combinational blocks must use **blocking** assignments
- Use of latches is discouraged, use flip-flops when possible
- The use of `X` assignments in RTL is strongly discouraged, make use of SVAs to check invalid behavior instead.
- Prefer `assign` statements wherever practical.
- Use `unique case` and always define a `default` case
- Use available signed arithmetic constructs wherever signed arithmetic is used
- When printing use `0b` and `0x` as a prefix for binary and hex. Use `_` for clarity
- Use logical constructs (i.e `||`) for logical comparison, bit-wise (i.e `|`) for data comparison
- Bit vectors and packed arrays must be little-endian, unpacked arrays must be big-endian
- FSMs: **no logic** except for reset should be performed in the process for the state register
- A combinational process should first define **default value** of all outputs in the process
- Default value for next state variable should be the current state