# Verilog Coding Style Guide

https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md#parameterized-objects-modules-etc

### Basic Style Elements

* one file per module, files named as module.sv, C++ style comments //.
* **two** spaces **(no** tabs) per indent for all paired keywords, **four** spaces for line wrap indent.
* begin must be on the same line as the preceding keyword and end the line, and end must start a new line.
* **one** space must separate the comma and the next character, **whitespace** around keywords and binary operators.
* **no** space between case item and colon, function/task/macro call and open parenthesis.

### 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
* 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
* Use of latches is discouraged, use flip-flops when possible
* Interfaces are discouraged
* Sequential logic must use **non-blocking** assignments
* Combinational blocks must use **blocking** assignments
* Prefer assign statements wherever practical.
* The use of X assignments in RTL is strongly discouraged, make use of SVAs to check invalid behavior instead.
* 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, the combinational process should first define **default value** of all outputs in the process, the default value for next state variable should be the current state.

### Basic Template

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

| **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 |

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

### Indentation

Use spaces to indent or align text. Do not use tabs. You should set your editor to emit spaces when you hit the tab key.

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

// Wrapped procedural block by begin and end.

always\_ff @(posedge clk) begin

q <= d;

end

// Omitted begin and end when the entire structure fits on a single line.

always\_ff @(posedge clk) q <= d;

// "end else begin" are on the same line.

if (condition) begin

foo = bar;

end else begin

foo = bum;

end

// fit on a single line may omit begin and end.

if (condition) foo = bar;

else foo = bum;

// 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

When wrapping a long expression, indent the continued part of the expression by four spaces.

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};

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;

### Spacing

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

bus = {addr, parity, data};

a = myfunc(lorem, ipsum, dolor, sit, amet, consectetur, adipiscing, elit,

rhoncus);

mymodule mymodule(.a(a), .b(b));

Include whitespace on both sides of all binary operators.

assign a = ((addr & mask) == My\_addr) ? b[1] : ~b[0]; // good

**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

Adding whitespace to cause related things to align is encouraged.

logic [7:0] my\_interface\_data;

logic [15:0] my\_interface\_address;

logic my\_interface\_enable;

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];

//以下不推荐：

// There must not be a space between dimensions.

logic [7:0] [3:0] data[128] [2];

// There must 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];

### Module Declaration

Use the Verilog-2001 combined port and I/O declaration style. The port declaration in the module statement should fully declare the port name, type, and direction.

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

);

**Parameterized Module Instantiation**

my\_module #(

.Height(5),

.Width(10)

) my\_instance (

.clk\_i (clk\_i),

.rst\_ni(rst\_ni),

.d\_i (from\_here),

.q\_o (to\_there)

);

my\_reg #(16) my\_reg0 (.clk\_i, .rst\_ni, .d\_i(data\_in), .q\_o(data\_out));

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)

);

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)).

Do not use positional arguments to connect signals to ports. Instantiate ports in the same order as they are defined in the module.

### 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.

### Delay Modeling

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

### Sequential Logic (Latches)

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

### Sequential Logic (Registers)

***Use the standard format for declaring sequential blocks.***

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

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

### Combinational Logic

Use always\_comb for SystemVerilog combinational blocks.

Use always @\* if only Verilog-2001 is supported.

Prefer assign statements wherever practical.

Synthesizable combinational logic blocks should only use blocking assignments.

Do not use three-state logic (Z state) to accomplish on-chip logic.

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

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

### Finite State Machines

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.

// 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

### Enumerations

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;