Svlint Manual

DEVELOPMENT c27eda8c

${REPOSITORY}

T

Contents

Introduction

About This Document

This document is generated from the Markdown files in `md/*.md`, the rules' source code (`svlint/src/rules/*.rs`), and their testcases (`testcases/(fail|pass)/*.sv`) using the `mdgen` utility.

Purpose of Lint Checks

The authors of any works must consider their audience, particularly in how different sections of the audience will interpret the works. For example, an author of childrens books has two main sections of audience (children, and their adult parents) so they might aim to please both sections at once; Children with simple storylines and colorful pictures; Parents with cultural references and subtle innuendo. Authors writing in SystemVerilog also have two main sections of audience which they should aim to please: 1) other silicon engineers, 2) mechanical tools. Although the differences between human and mechanical readers are significant, both must be satisfied for the text to be nice/enjoyable to work with. While a simulation tool doesn't care about whitespace, indentation, or thoughtful comments, your human colleagues will dread working with messy code (rewiewing, modifying, building upon outputs, etc.), which ultimately wastes their time, money, and sanity. Human readers will usually be polite about sub-par work, but tools are much more direct, simply spitting back at you with warning messages and an outright refusal to work if you dare to mis-spell a variable name.

There are two main classes of rule for helping human readers:

1. Rules which codify naming conventions.
2. Rules which codify style/formatting conventions.

Further information on these concepts is provided in the `style` ruleset.

Just as individual human readers have their own preferences (in language, style, naming conventions, etc.), each tool has its own quirks and ways of interpreting things, particularly when the language specification is not fully explicit. The most prominent example of tools' differences in interpretation of SystemVerilog is between tools for simulation and tools for synthesis. The SystemVerilog language is specifed in IEEE1800-2017, also known as the Language Reference Manual (LRM). The LRM is clear that the specification is written in terms of simulation, but that some of its constructs may be synthesized into physical hardware. This distinction is the basis for a class of functional rules which aim to minimize the risk of introducing a mismatch between simulation and synthesis. Another class of functional rules is those which check for datatypes and constructs that avoid compiler checks for legacy compatibility.

Usage

This tool (svlint) works in a series of well-defined steps:

1. On startup, search for a configuration file or use a default configuration.
2. Examine the configuration to determine which rules should be enabled and load them into memory.
3. Parse a whole file for preprocessor constructs like `` `ifdef `` and `` `include ``.
4. Apply the preprocessor semantics to produce a source descrition text.
5. Parse the source description into a syntax tree. The grammatical structure of a syntax tree is described in IEEE1800-2017 Annex A using Backus-Naur Form.
6. Iterate over each node of the syntax tree in order.
7. For each node, apply each rule independently.
8. If a rule detects an undesirable quality in the syntax tree, then return a failure, otherwise return a pass.

Filelists

Specification of the files to be processed can be given on the command line by *either* a list of files (e.g. `svlint foo.sv bar/*.sv`), or via filelists (e.g. `svlint -f foo.fl -f path/to/bar.fl`). It is not supported to specify both files and filelists, primarily because concerns about usability due to the way command-line arguments are processed.

The following features are supported via the sv-filelist-parser crate.

- Lines beginning with `//` or `#` and empty lines are ignored.
- Specify include directories like `+incdir+path/to/foo`.
- Define preprocessor macros like `+define+FOO` or `+define+BAR=1`.
- Include other filelists like `-f path/to/foo.fl`
- All remaining lines are treated as file paths.
- Substitute of environment variables like `$FOO`, `${FOO}`, or `$(FOO)`.

For example:

```
aaa.sv
$FOO/bbb.sv
${FOO}/ccc.sv
$(FOO)/ddd.sv
+incdir+$PWD/header/src
+define+SYNTHESIS
-f anotherFilelist.fl
```

Plugin rules

svlint supports plugin rules, an example of which is available here.

A plugin rule is one which is compiled separately to the main svlint binary, and is loaded at runtime. In the same way as integrated rules, a plugin rule must

implement the `Rule` trait, i.e. `check`, `name`, `hint`, and `reason`. The `hint` and `reason` methods allow plugin rules to provide information back to the user on the terminal, but they do not require testcases or an explanation. All loaded plugin rules, via the `--plugin` option, are enabled and have access to all values in the TOML configuration.

Configuration

First of all, you must put a configuration file `.svlint.toml` to specify enabled rules. Configuration file is searched to the upper directory until `/`. So you can put configuration file (`.svlint.toml`) on the repository root alongside `.gitignore`. Alternatively, for project-wide rules you can set the environment variable `SVLINT_CONFIG` to something like `/cad/projectFoo/teamBar.svlint.toml`.

The example of configuration file is below:

```
[option]
exclude_paths = ["ip/.*"]
prefix_label = ""

[rules]
non_ansi_module = true
keyword_forbidden_wire_reg = true
```

The complete example can be generated by `svlint --example`

`[option]` section

- `exclude_paths` is a list of regular expressions. If a file path is matched with any regex in the list, the file is skipped.
- `prefix_(inout|input|output)` are strings which port identifiers must begin with. Only used when the corresponding rule is enabled. Defaults to `"b_"`, `"i_"`, and `"o_"` respectively.
- `prefix_label` is a string which generate labels must begin with. Applicable to `if/else`, `for`, and `case` generate constructs when the corresponding `generate_*_with_label` rule is enabled. Defaults to `"l_"`. To check only that a label exists, set this to `""`.
- `prefix_instance` is a string which instances must begin with. Defaults to `"u_"`.
- `prefix_(interface|module|package)` are strings which definitions must begin with. An alternative naming convention for interface, module, and package names is uppercase/lowercase first letter. This is similar to Haskell where types begin with uppercase and variables begin with lowercase. These alternative rules are called `(lower|upper)camelcase_(interface|module|package)`.
- `re_(forbidden|required)_*` are regular expressions for detailed naming conventions, used only when the corresponding rules are enabled. The

defaults for `re_required_*` are either uppercase, lowercase, or mixed-case starting with lowercase, i.e. just vaguely sensible. The defaults for `re_forbidden_*` are to forbid all strings, except those starting with "X", i.e. not at all sensible (configuration required).

`[rules]` section All rules are disabled unless explicitly enabled in the `[rules]` section. To enable a rule, assign `true` to its name, e.g. `case_default = true`.

Where no configuration file can be found, all rules are implicitly enabled which will most likely result in errors from conflicting rules, e.g. **keyword__forbidden__generate** and **keyword__required__generate**.

If you need to turn off specific rules for a section, then you can use special comments within your SystemVerilog source code:

```
/* svlint off keyword_forbidden_always */
always @* foo = bar;                        // <-- This line is special.
/* svlint on keyword_forbidden_always */
```

Configuration update If svlint is updated, `.svlint.toml` can be updated to the latest version with `svlint --update`.

Rule Documentation

Each rule is documented with 5 pieces of information:

- Hint: A brief instruction on how to modify failing SystemVerilog. Also displayed in supported editors using svls.
- Reason: A one sentence explanation of the rule's purpose. Also displayed in supported editors using svls.
- Pass Example: A valid piece of SystemVerilog which is known to pass the rule. Ideally, this will show an example of best-practice.
- Fail Example: A valid piece of SystemVerilog which is known to fail the rule. In some cases the code shows multiple commented examples.
- Explanation: A full explanation of the rule's purpose with references to any other relevant information sources.

In each rule's explanation there is a "see also" list of other rules, each with a short reason why it should be seen.

- "suggested companion" - Suggestions are given for rules which do not check semantics, i.e suggestions are for style and naming conventions only.
- "potential companion" - These are noted where the named rule is given primarily out of completeness, but their use may cause other issues. For example, **style__keyword__datatype** exists to ensure all SystemVerilog keywords are captured in the `style_keyword_*` set, but its use is not suggested because it is visually appealing (and common practice) to align the identifiers in adjacent declarations.

- "useful companion" - Enabling the named rule provides an additional set of properties which are useful for reasoning about the function and semantics of code which passes. For example, the conjunction of **localparam__type__twostate** and **localparam__explicit__type** allows for stronger confidence that the author has properly considered the type of each constant.
- "alternative" - The named rule *should* not be used in conjunction, i.e. enabling both rules is, at best, a waste compute power.
- "mutually exclusive alternative" - The named rule *can* not be used in conjunction, i.e. enabling both rules is nonsensical because a failure on one implies a pass on the other rule.

You are welcome to suggest a new rule through Issues or Pull Requests.

Functional Rules

Rule: `action_block_with_side_effect`

Hint

Do not specify side effects within `assert` or `wait_order` action blocks.

Reason

Side effects may cause undefined event ordering.

Pass Example (1 of 1)

```
module M;
  always @(posedge clk)
    assert (A)
      else $error("A should be high.");

  // Simulator must report line number, label, and time on each violation.
  asrt_b1: assert property (@(posedge clk) B1)
    else $error("B1 should be high.");
  asrt_b2: assert property (@(posedge clk) B2)
    else $error("B2 should be high.");
endmodule
```

Fail Example (1 of 1)

```
module M;
  always @(posedge clk)
    assert (A) // These are legal, but potentially confusing.
    else begin
      $display("A should be high."); // Write to STDOUT.

      // Update global variable.
      errorId = 5; // What value if multiple immediate assertions fail?
      errorCount++; // Hopefully simulator blocks between processes.
    end

  // In what order do these action blocks occur?
  asrt_b1: assert property (@(posedge clk) B1)
    else begin
      $display("B1 should be high.");
      errorId = 1;
      errorCount++;
    end;
  asrt_b2: assert property (@(posedge clk) B2)
    else begin
      $display("B2 should be high.");
```

```
      errorId = 2;
      errorCount++;
    end;
endmodule
```

Explanation

Simulator event ordering between concurrent action blocks is undefined, so observed behavior is simulator-dependent. While assertions with side-effects may appear to work on a single-threaded simulator, they may interact in unexpected ways on a multi-threaded simulator. On encountering side-effect code in action blocks, a simulator can either implement inter-thread locking (with a hit to performance) or allow a race-condition to occur, neither of which are desirable.

Specifically, action blocks should not contain blocking assignments:

- Blocking assignment operator, e.g. `foo = 123;`
- Increment/decrement operators, e.g. `foo++;`, `foo--;`.
- Sequential IO, e.g. `$display();`, `$write();`. The full list of IO system tasks and system functions is given on page 624 of IEEE1800-2017.

See also:

- **non_blocking_assignment_in_always_comb** - Useful companion rule.
- **blocking_assignment_in_always_ff** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 15.5.4 Event sequencing: wait_order()
- 16 Assertions
- 21 Input/output system tasks and system functions

Rule: `blocking_assignment_in_always_ff`

Hint

Do not use blocking assignments within `always_ff`.

Reason

Blocking assignment in `always_ff` may cause undefined event ordering.

Pass Example (1 of 1)

```
module M;
  always_ff @(posedge clk) q1 <= d; // Correct.

  /* svlint off blocking_assignment_in_always_ff */
  always_ff @(posedge clk) q2 = d;  // Control comments avoid failure.
  /* svlint on blocking_assignment_in_always_ff */
endmodule
```

Fail Example (1 of 1)

```
module M;
/* svlint off blocking_assignment_in_always_ff */
always_ff @(posedge clk) q1 = d;   // Control comments avoid failure.
/* svlint on blocking_assignment_in_always_ff */

always_ff @(posedge clk) q2 = d;   // Failure.
endmodule
```

Explanation

Simulator event ordering between blocking and non-blocking assignments is undefined, so observed behavior is simulator-dependent. As all examples in IEEE1800-2017 show, `always_ff` should only contain non-blocking assignments in order for sampling and variable evaluation to operate in a defined order.

Specifically, `always_ff` constructs should not contain blocking assignments:

- Blocking assignment operator, e.g. `foo = 123;`
- Increment/decrement operators, e.g. `foo++;`, `foo--;`.

See also:

- **non_blocking_assignment_in_always_comb** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 4.9.3 Blocking assignment
- 4.9.4 Non-blocking assignment

- 9.2.2.4 Sequential logic always_ff procedure
- 9.4.2 Event control
- 10.4.1 Blocking procedural assignments
- 10.4.2 Nonblocking procedural assignments
- 16.5.1 Sampling

Rule: `case_default`

Hint

Use a `default` expression in `case` statements.

Reason

Incomplete case may cause simulation/synthesis mismatch in `always_comb` and `function`.

Pass Example (1 of 1)

```
module M;
  always_comb
    case (x)
      1: y = 0;
      default: y = 0;
    endcase

  always_ff
    case (x)
      1: y = 0;
    endcase
endmodule
```

Fail Example (1 of 1)

```
module M;
  always_comb
    case (x)
      1: a = 0;
    endcase
endmodule
```

Explanation

IEEE1800-2017 (clause 9.2.2.2) comments that tools should *warn* if an `always_comb` procedure infers memory. However, simulators and synthesis tools are not required to enforce that `always_comb` procedures only infer combinational logic. This allows for simulators and synthesis tools to interpret these procedures differently, which results in a mismatch between simulation and synthesis.

An incomplete case statement may be interpreted as latched logic, e.g: `always_comb case (foo) '0: a = 5; endcase`. Only the case where `foo == 0` is specified, to update variable `a` to the value `5`. When `foo` is non-zero, this example may be interpreted in at least two ways:

- `a = 'x;` - As the new value is not specified, it is unknown. A synthesis tool may allow node `a` to be undriven, or choose to drive `a` equivalently to one of the explicitly specified case expressions.
- `a = a;` - As the new value is not specified, do not change `a`. A synthesis tool may produce a latching circuit.

See also:

- **explicit_case_default** - Useful companion rule.
- **explicit_if_else** - Useful companion rule.
- **legacy_always** - Useful companion rule.
- **sequential_block_in_always_comb** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 9.2.2.2 Combinational logic `always_comb` procedure
- 12.5 Case statement
- 13.4 Functions

Rule: `default_nettype_none`

Hint

Place `` `default_nettype none `` at the top of source code.

Reason

Compiler directive `` `default_nettype none `` detects unintentional implicit wires.

Pass Example (1 of 1)

```
`default_nettype none
module M;
endmodule
```

Fail Example (1 of 1)

```
module M;
endmodule
```

Explanation

The `` `default_netype `` compiler directive can be used to specify the net type of implicit nets, i.e. where a signal is referenced, or assigned to, without being declared. IEEE1800-2017 clause 22.8 stipulates "When no `` `default_nettype `` directive is present or if the `` `resetall `` directive is specified, implicit nets are of type `wire`."

SystemVerilog makes a distinction between variables (only 0 or 1 drivers) and nets (0 or more drivers). IEEE1364-2001 (Verilog) uses variables as abstractions for data storage elements (`reg`, `integer`, `real`, `time`, `realtime`). In contrast, IEEE1800-2017 (SystemVerilog) the distinction between nets and variables is defined by how a simulator must calculate a value. In a simulator, a variable stores a value, but a net's value is calculated by evaluating the strength of all drivers. To keep compatibility with Verilog, the default net type of an undeclared net in SystemVerilog is `wire` (a net, not a variable), which requires evaluating a list of values with strengths, rather than simply looking up a value. The distinction between data storage elements and physical wires is therefore made in using `always_comb`, `always_ff`, and (less commonly) `always_latch` keywords.

Variables are preferred over nets for most digital logic for 2 reasons:

- Only 0 or 1 drivers allowed, so an accidental multi-driving is caught by a compile time error.
- Simulator performance (dependent on implemetation). Value can be found by lookup, rather than evaluation of drivers.

When `` `default_nettype none `` is used, all signals must be declared, thus forcing the author to consider whether they mean a variable or a net.

See also:

- **inout_with_tri** - Useful companion rule.
- **input_with_var** - Useful companion rule.
- **output_with_var** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 6.5 Nets and variables
- 22.8 default nettype

Note: One prominent paper (Cliff Cummings, HDLCON 2002) recommends *against* using `` `default_nettype none `` on the basis that concise, typeless code has fewer opportunities for mistakes. This attitude was popular at the time, e.g. Python's dynamic typing, but modern attitudes are now favouring explicit types, e.g. Python's new type checking syntax and tooling. Additionally, the reasoning behind this guideline only applies principally to IEEE1364, but not strongly to IEEE1800.

Rule: `enum_with_type`

Hint

Specify an explicit `enum` base type.

Reason

The default `enum` base type is `int` (32b, 2-state).

Pass Example (1 of 1)

```
module M;
  typedef enum int {
    i
  } E;
endmodule
```

Fail Example (1 of 1)

```
module M;
  typedef enum
    { i
    } E;
endmodule
```

Explanation

SystemVerilog has both 2-state types (each bit can take the values 0 or 1), and 4-state types (each bit can take the values 0, 1, x, or z). 2-state types are useful for holding constants, and programming non-synthesizable simulation constructs. 4-state types are useful for modelling physical hardware because undriven, multiply-driven, or improperly-driven wires can hold unknown states that cannot be sufficiently modelled by only 2 states. Therefore, it is important to use the 4-state types when writing SystemVerilog which will be used to infer physical hardware.

For example, a counter described as `always_ff @(posedge clk) count_q <= count_q + 'd1;` should be declared like `logic [4:0] count_q;`. This infers 5 non-reset flip-flops where the initial value is unknown, and in a 4-state simulation the value of `count_q` is always unknown (`'x`, because there's no initialization). Instead, if it was declared as `bit [4:0] count_q;`, then the initial value is `5'd0`, so a simulation will show `count_q` changing on every positive edge of `clk`. When describing physical hardware, it would be useful to know that the inferred flip-flops have no reset, i.e., you want to be *able* to see x's when a mistake is made even if you don't want to see x's.

An `enum` is a set of named values of a single type. If no datatype is specified, then the default `int` (32b, 2-state) is implied. For example, `enum {RED, BLACK} m;`

`assign m = foo ? BLACK : RED;` describes a multiplexor, but a simulator is unable to sufficiently model the behavior of `m` when the value of `foo` is unknown. A more appropriate declaration is `typedef enum int {RED, BLACK} color; integer m;`.

Note: Comparison of 4-state variables against 2-state constants/enums *is* appropriate, e.g. `logic a; a = (m == RED);`.

See also:

- **localparam_explicit_type** - Useful companion rule.
- **localparam_type_twostate** - Useful companion rule.
- **parameter_explicit_type** - Useful companion rule.
- **parameter_type_twostate** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 6.8 Variable declarations
- 6.11 Integer data types
- 6.19 Enumerations
- Table 6.7 Default variable initial values
- Table 6.8 Integer data types

Rule: `explicit_case_default`

Hint

Add a `default` arm to the `case` statement.

Reason

Fully-specified case clarifies design intent.

Pass Example (1 of 1)

```
module M;
  always_comb
    case (x)
      1: y = 0;
      default: y = 0;
    endcase

  always_ff @(clk)
    case (x)
      1: y = 0;
      default: y = 0;
    endcase
endmodule
```

Fail Example (1 of 2)

```
module M;
  always_comb
    case (x)
      1: a = 0; // Incompletely specified case implies memory.
    endcase
endmodule
```

Fail Example (2 of 2)

```
module M;
  always_ff @(clk) begin
    case (x)
      1: a = 0;
      default: a = 0; // Explicit default arm is good.
    endcase

    case (y)
      1: b = 0; // Implicit default arm.
    endcase
```

```
  end
endmodule
```

Explanation

The reasoning behind this rule are different between combinatial constructs (`always_comb`, `always @*`) vs sequential constructs (`always_ff`, `always_latch`). The reasoning behind this rule is equivalent to that of **explicit_if_else**.

For combinational constructs, the reasoning behind this rule is equivalent to that of the rule **case_default**. To summarize, an incompletely-specified case statement may infer sequential behavior (i.e. memory), thus causing a mismatch between simulation and synthesis tools. Due to the slightly different formulations, it is recommended that both this rule and **case_default** are enabled.

For sequential constructs, the reasoning behind this rule is equivalent to those of the rules **sequential_block_in_always_ff** and **sequential_block_in_always_latch**. To summarize, fully-specified case statements make the design intent explicit and clear through some useful redundancy.

NOTE: The legacy keyword `always` can infer both combinational and sequential constructs in the same block, which can be confusing and should be avoided. Use of the legacy keyword can be detected with the rule **legacy_always**.

See also:

- **case_default** - Useful companion rule.
- **explicit_if_else** - Useful companion rule.
- **legacy_always** - Useful companion rule.
- **sequential_block_in_always_comb** - Useful companion rule.
- **sequential_block_in_always_ff** - Useful companion rule.
- **sequential_block_in_always_latch** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 12.5 Case statement

Rule: `explicit_if_else`

Hint

Add an `else` clause to the `if` statement.

Reason

Fully-specified conditional clarifies design intent.

Pass Example (1 of 1)

```
module M;
  always_ff @(clk)
    if (x) y <= 0;
    else   y <= z;

  always_comb
    if (x) y = 0;
    else   y = z;
endmodule
```

Fail Example (1 of 2)

```
module M;
  always_comb
    if (x) y = 0; // Incompletely specified condition implies memory.
endmodule
```

Fail Example (2 of 2)

```
module M;
  always_ff @(clk) begin
    if (a)
      b <= c;
    else // Explicit else clause is good.
      b <= d;

    if (b)
      c <= d; // Implicit else clause.
  end
endmodule
```

Explanation

The reasoning behind this rule are different between combinatial constructs (`always_comb`, `always @*`) vs sequential constructs (`always_ff`, `always_latch`). The reasoning behind this rule is equivalent to that of **explicit_case_default**.

For combinational constructs, the reasoning behind this rule is equivalent to that of the rule **case_default**. To summarize, an incompletely-specified case statement may infer sequential behavior (i.e. memory), thus causing a mismatch between simulation and synthesis tools.

For sequential constructs, the reasoning behind this rule is equivalent to those of the rules **sequential_block_in_always_ff** and **sequential_block_in_always_latch**. To summarize, fully-specified case statements make the design intent explicit and clear through some useful redundancy.

NOTE: The legacy keyword `always` can infer both combinational and sequential constructs in the same block, which can be confusing and should be avoided. Use of the legacy keyword can be detected with the rule **legacy_always**.

See also:

- **explicit_case_default** - Useful companion rule.
- **legacy_always** - Useful companion rule.
- **sequential_block_in_always_comb** - Useful companion rule.
- **sequential_block_in_always_ff** - Useful companion rule.
- **sequential_block_in_always_latch** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 12.4 Conditional if-else statement

Rule: `function_same_as_system_function`

Hint

Rename `function` to something other than the name of a built-in function.

Reason

Name clashes may cause confusion amongst tools and readers.

Pass Example (1 of 1)

```
module M;
  function my_clog2;
  endfunction
endmodule
```

Fail Example (1 of 1)

```
module M;
  function clog2;
  endfunction
endmodule
```

Explanation

IEEE1800-2017 provides a variety of built-in functions, which must be implemented in simulation and synthesis tools. This rule is designed to catch (possibly incorrect) re-implementations of these functions which may have different behavior and confuse readers. Additionally, some tools may (wrongly) confuse user-defined functions with the built-in system of the same name (except of the leading `$`) which may lead to inconsistent results between tools.

See also:

- **function__with__automatic** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 13.7 Task and function names
- 20 Utility system tasks and system functions
- 23.8.1 Task and function name resolution

Rule: `function_with_automatic`

Hint

Add the `automatic` lifetime specifier to function.

Reason

Static lifetime of function items causes a simulation/synthesis mismatch.

Pass Example (1 of 1)

```
module M;
  function automatic F;
  endfunction
endmodule

module automatic M; // Default lifetime.
  function F;
  endfunction
endmodule

interface automatic I;
  function F;
  endfunction
endinterface

program automatic P;
  function F;
  endfunction
endprogram

package automatic P;
  function F;
  endfunction
endpackage

module static M;
  function automatic F; // Override default lifetime.
  endfunction
endmodule

interface static I;
  function automatic F;
  endfunction
endinterface
```

```
program static P;
  function automatic F;
  endfunction
endprogram

package static P;
  function automatic F;
  endfunction
endpackage

module M;
  class C;
    function F; // Function in class is automatic.
    endfunction
  endclass
endmodule

module automatic M;
  class C;
    function F;
    endfunction
  endclass
endmodule

module static M;
  class C;
    function F;
    endfunction
  endclass
endmodule
```

Fail Example (1 of 1)

```
module M;
  function F;
  endfunction
endmodule
```

Explanation

Functions defined within a module, interface, program, or package default to being static, with all declared items being statically allocated. These items shall be shared across all uses of the function executing concurrently. This causes a mismatch between simulation and synthesis.

Functions can be defined to use automatic storage by using the `automatic` keyword as part of the function declaration, i.e. in simulation each use of a

function is allocated dynamically for each concurrent function call. This behavior can be accurately inferred in synthesis.

See also:

- **function_same_as_system_function** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 13.4.2 Static and automatic functions

Rule: `genvar_declaration_in_loop`

Hint

Declare `genvar` inside a loop generate construct.

Reason

Minimized `genvar` scope makes code easier to read and review.

Pass Example (1 of 1)

```
module M;
  for(genvar i=0; i < 10; i++) begin: a
  end
endmodule
```

Fail Example (1 of 1)

```
module M;
  genvar i;
  for (i=0; i < 10; i++) begin
  end
endmodule
```

Explanation

The specification of genvar declarations in IEEE1800-2017 is not straightforward. The formal syntax of `genvar_initialization` specified in Annex A.4.2 (Generated instantiation) suggests that the `genvar` keyword is optional, but the second sentence of Clause 27.5 declares that "The loop index variable shall be declared in a genvar declaration prior to its use in a loop generate scheme". All 5 examples in Clause 27.4 (Loop generate constructs) declare the genvars outside of the generate loops, and the formal syntax of `genvar_declaration` in A.2.1.3 (Type declarations) is only applicable to declarations outside of loop generate constructs. That is, using syntax like `genvar i; for (i=0; ...)`. However, several examples of declarations inside loop generate constructs are present in other areas of the LRM like `for (genvar i=0; ...`:

- Clause 11.12 Let construct, example d, page 295.
- Clause 16.14.6.1 Arguments to procedural concurrent assertions, page 464.
- Clause 20.11 Elaboration system tasks, page 607.
- Clause 23.3.3.5 Unpacked array ports and arrays of instances, page 717.

Although it is not explicitly stated, a reasonable interpretation is that a genvar declared inside a generate loop may only be used within that specific loop generate construct, i.e. locally scoped. This interpretation matches C99 (ISO/IEC 9899:1999), while a requirement for the genvar to be declared outside would match ANSI C (ISO/IEC 9899:1990). This rule checks that genvars are declared

in a C99-like style so that the identifier is declared beside its use which has several advantages:

- The purpose of the genvar is immediately clear, e.g. it is easy to read that the `i` in `for (genvar i=0; i < N_BITS; i++) ...` refers to a bit index. In contrast, `genvar j; ...many lines... for (j=0; j < N_BITS; j++) ...` requires the reader to keep `j` in their head for a longer time.
- Only one comment is necessary, rather than splitting or duplicating the information.
- When a future revision of your code removes a generate loop, the genvar declaration is implictly removed too, which avoids lingering useless and distracting statements.
- A subsequent generate loop cannot accidentally use a "leftover" genvar which is intended for use only by a previous generate loop. The LRM only requires that "A genvar shall not be referenced anywhere other than in a loop generate scheme.".

Given the lack of clarity in the LRM, it is unsurprising that some tools might not support both ways of declaring genvars, so the related rule **genvar_declaration_out_loop** assumes a stricter interpretation of the LRM and checks that declarations must be separate from the generate loop syntax.

See also:

- **genvar_declaration_out_loop** - Opposite reasoning.

The most relevant clauses of IEEE1800-2017 are:

- 27.4 Loop generate constructs

Rule: `genvar_declaration_out_loop`

Hint

Declare `genvar` outside the loop generate construct.

Reason

Some tools don't support `genvar` declarations inside loop generate constructs.

Pass Example (1 of 1)

```
module M;
  genvar i;
  for (i=0; i < 10; i++) begin: a
  end
endmodule
```

Fail Example (1 of 1)

```
module M;
  for (genvar i=0; i < 10; i++) begin: l_foo
  end: l_foo
endmodule
```

Explanation

The specification of genvar declarations in IEEE1800-2017 is not straightforward. The formal syntax of `genvar_initialization` specified in Annex A.4.2 (Generated instantiation) suggests that the `genvar` keyword is optional, but the second sentence of Clause 27.5 declares that "The loop index variable shall be declared in a genvar declaration prior to its use in a loop generate scheme". All 5 examples in Clause 27.4 (Loop generate constructs) declare the genvars outside of the generate loops, and the formal syntax of `genvar_declaration` in A.2.1.3 (Type declarations) is only applicable to declarations outside of loop generate constructs. That is, using syntax like `genvar i; for (i=0; ...)`. However, several examples of declarations inside loop generate constructs are present in other areas of the LRM like `for (genvar i=0; ...`:

- Clause 11.12 Let construct, example d, page 295.
- Clause 16.14.6.1 Arguments to procedural concurrent assertions, page 464.
- Clause 20.11 Elaboration system tasks, page 607.
- Clause 23.3.3.5 Unpacked array ports and arrays of instances, page 717.

This rule assumes a strict interpretation of the LRM and checks that declarations must be separate from the generate loop syntax.

The related rule **genvar_declaration_in_loop** checks the opposite way because C99-like declarations inside loop generate constructs can lead to code

which is easier to read and review.

See also:

- **genvar_declaration_in_loop** - Opposite reasoning.

The most relevant clauses of IEEE1800-2017 are:

- 27.4 Loop generate constructs

Rule: `inout_with_tri`

Hint

Specify `tri` datakind on `inout` ports.

Reason

Explicit datakind of bi-directional ports should be consistent with input ports.

Pass Example (1 of 1)

```
module M
  ( inout tri a
  );
endmodule
```

Fail Example (1 of 1)

```
module M
  ( inout wire a
  );
endmodule
```

Explanation

This check mandates that each `inout` port must be explicitly declared as a tri-state net, rather than the default nettype.

The rules for determining port kind, datatype, and direction are specified in IEEE1800-2017 Clause 23.2.2.3 and facilitate various shorthand notations which are backwards compatible with the semantics of Verilog (IEEE1364-1995):

- `inout a` -> `inout tri logic a` The implicit datatype is `logic` and the default nettype is `tri` (without overriding via the `` `default_nettype `` compiler directive).
- `inout wire a` -> `inout tri logic a` Again, using the implicit datatype of `logic`; As `wire` is an alias for `tri`, this is equivalent to the above example.
- `inout logic a` -> `inout tri logic a` This time using an explicit datatype (`logic`) but relying on the default nettype for its datakind.
- `inout wire logic a` -> `inout tri logic a` Again, even with an explicit datatype (`logic`), the `wire` keyword is simply an alias for the datakind `tri`.

When the default nettype is overridden to none, i.e. with the compiler directive `` `default_nettype none ``, inout ports require an explicit datakind.

Although the semantics of `inout a` are equivalent in IEEE1364-1995, the intent is not clearly described. An author should use `inout` to declare ports which

are driven both internally and externally, but `input` to declare ports which should only be driven externally. In order to describe the intended bi-directional behavior, `inout` ports must be declared with an explicit `tri` datakind.

See also:

- **default_nettype_none** - Useful companion rule.
- **input_with_var** - Suggested companion rule.
- **output_with_var** - Suggested companion rule.
- **prefix_inout** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 6.5 Nets and variables
- 6.6 Net types
- 22.8 default nettype
- 23.2.2 Port declarations

Rule: `input_with_var`

Hint

Specify `var` datakind on `input` ports.

Reason

Default datakind of input port is a tri-state net.

Pass Example (1 of 1)

```
module M
  ( input var a
  );
endmodule
```

Fail Example (1 of 1)

```
module M
  ( input logic a
  );
endmodule
```

Explanation

This check mandates that each `input` port must be explicitly declared as a variable, rather than the default nettype.

The rules for determining port kind, datatype, and direction are specified in IEEE1800-2017 Clause 23.2.2.3 and facilitate various shorthand notations which are backwards compatible with the semantics of Verilog (IEEE1364-1995):

- `input a` -> `input tri logic a` The implicit datatype is `logic` and the default nettype is `tri` (without overriding via the `` `default_nettype `` compiler directive).
- `input wire a` -> `input tri logic a` Again, using the implicit datatype of `logic`; As `wire` is an alias for `tri`, this is equivalent to the above example.
- `input logic a` -> `input tri logic a` This time using an explicit datatype (`logic`) but relying on the default nettype for its datakind.
- `input wire logic a` -> `input tri logic a` Again, even with an explicit datatype (`logic`), the `wire` keyword is simply an alias for the datakind `tri`.

When the default nettype is overridden to none, i.e. with the compiler directive `` `default_nettype none ``, input ports require an explicit datakind.

Although the semantics of `input a` are equivalent in IEEE1364-1995, the intent is not clearly described. An author should use `input` to declare ports which

should only be driven externally, and `inout` to declare ports which may also be driven internally. In order to describe the intended uni-directional behavior, `input` ports must be declared with an explicit `var` datakind, thus requiring the compiler to check that the input is not driven from within the module (and if so, emit an error).

See also:

- **default_nettype_none** - Useful companion rule.
- **inout_with_tri** - Suggested companion rule.
- **output_with_var** - Suggested companion rule.
- **prefix_input** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 6.5 Nets and variables
- 6.6 Net types
- 22.8 default nettype
- 23.2.2 Port declarations

Rule: `interface_port_with_modport`

Hint

Specify the modport on the interface port.

Reason

Without a modport, the interface port signals are all implictly `inout`.

Pass Example (1 of 1)

```
module M
  ( test_if.a a
  , interface.b b
  );
endmodule
```

Fail Example (1 of 2)

```
module M
  ( test_if a
  );
endmodule
```

Fail Example (2 of 2)

```
module M
  ( interface b
  );
endmodule
```

Explanation

A SystemVerilog Interface (SVI) defines a set of named signals which can be used in many places within a design. For example, if modules `A` and `B` both instance an interface `I` as `A.u_I` and `B.u_I`, then both modules get their own collection of named signals, accessed like `u_I.x`. Each interface instance is separate, so `A.u_I.x` is independent of `B.u_I.x`. By adding another signal `y` to the interface, two new signals are created, `A.u_I.y` and `B.u_I.y`.

SVIs are useful for connecting hierarchical modules with a minimal amount of code, i.e. by using interface ports. To specify the direction of signals in an SVI, a `modport` is declared with and identifier and the directions of each signal declared from the perspective of inside a module. Without a `modport`, the default direction of interface port signals is `inout`. This is often undesirable for synthesizable digital designs, so this rule requires that each interface port includes a modport identifier.

See also:

- **inout__with__tri** - Useful companion rule.
- **input__with__var** - Useful companion rule.
- **non__ansi__module** - Useful companion rule.
- **output__with__var** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 25.4 Ports in interfaces
- 25.5 Modports

Rule: `keyword_forbidden_always`

Hint

Use `always_comb`/`always_ff`/`always_latch` instead of `always`.

Reason

General-purpose `always` cannot detect combinatorial/stateful (non-)blocking mistakes.

Pass Example (1 of 1)

```
module M;
  always_comb begin
  end
endmodule
```

Fail Example (1 of 1)

```
module M;
  always @* begin
  end
endmodule
```

Explanation

In Verilog (IEEE1364), there are two language constructs which can be used to model combinatorial logic:

1. Continuous assignment to `wire` signals is specified with the `assign` keyword.
2. `reg` signals are assigned to with an `always` block, which is evaluated whenever anything in the sensitivity list changes value.

The `always` keyword can also be used for modelling sequential logic by including the edge of a signal in the sensitivity list.

The semantics of these keywords in SystemVerilog are compatible with Verilog, but additional keywords (`always_comb`, `always_ff`, and `always_latch`) should be used to clarify intent of digital designs. The `always_*` keywords have slightly different semantics which are beneficial for synthesizable designs:

1. `always_*` processes require compiler checks that any signals driven on the LHS are not driven by any other process, i.e. `always_*` cannot infer multi-driven or tri-state logic.
2. `always_comb` processes require a compiler check that the process does not infer state.
3. `always_ff` processes require a compiler check that the process does infer state.

This rule forbids the use of the general-purpose `always` keyword, thus forcing authors of synthesizable design code to clarify their intent. In verification code to be used in simulation only, a general-purpose `always` process is a valid and useful way of scheduling events. Therefore, this rule is intended only for synthesizable design code, not for testbench code.

The alternative rule **level_sensitive_always** has similar reasoning but is slightly relaxed, requiring that `always` blocks have an explicit sensitivity list including an edge. It is possible to construct a full-featured testbench where all `always` blocks meet that requriment. Therefore, it is appropriate to use **keyword_forbidden_always** on synthesizable design code, but on verification code use **level_sensitive_always** instead.

See also:

- **level_sensitive_always** - Alternative rule.
- **sequential_block_in_always_comb** - Useful companion rule.
- **sequential_block_in_always_if** - Useful companion rule.
- **sequential_block_in_always_latch** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 9.2.2 Always procedures
- 9.5 Process execution threads

Rule: `keyword_forbidden_generate`

Hint

Remove `generate`/`endgenerate` keywords.

Reason

Keywords `generate`/`endgenerate` do not change semantics of generate blocks.

Pass Example (1 of 1)

```
module M;
endmodule
```

Fail Example (1 of 1)

```
module M;
  generate
  endgenerate
endmodule
```

Explanation

The `generate`/`endgenerate` keywords may be used in a module, interface, program, or checker to define a generate region. A generate region is a textual span in the module description where generate constructs may appear. Use of generate regions is optional. There is no semantic difference in the module when a generate region is used. A parser may choose to recognize the generate region to produce different error messages for misused generate construct keywords.

As the semantics of generate blocks are unchanged by the `generate`/`endgenerate` keywords, the keywords can be argued to be visual noise, simply distracting the reader. Therefore, this rule is designed to detect and forbid their use.

NOTE: Some non-compliant tools may require the use of these keywords, which provides an argument against this rule.

See also:

- **keyword__required__generate** - Opposite reasoning.

The most relevant clauses of IEEE1800-2017 are:

- 27.3 Generate construct syntax

Rule: `keyword_forbidden_priority`

Hint

Remove `priority` keyword, perhaps replace with an assertion.

Reason

Priority-case/if constructs may mismatch between simulation and synthesis.

Pass Example (1 of 1)

```
module M;
  initial
    case (a)
      default: b = 1;
    endcase
endmodule
```

Fail Example (1 of 2)

```
module M;
  initial
    priority case (a)
      default: b = 1;
    endcase
endmodule
```

Fail Example (2 of 2)

```
module M;
  initial
    priority if (a)
      b = 1;
    else if (a)
      b = 2;
    else
      b = 3;
endmodule
```

Explanation

The keyword `priority` may be used on `if`/`else` or `case` statements to enable *violation checks* in simulation, and describe design intent for synthesis.

A `priority if` statement without an explicit `else` clause will produce a *violation report* in simulation if the implicit `else` condition is matched. A `priority if` statement with an explicit `else` clause cannot produce a violation report. In

synthesis, the `priority` keyword makes no difference to an `if/else` statement, because the semantics of bare `if/else` statements already imply priority logic.

A `priority case` statement without a `default` arm will produce a violation report in simulation if the `default` condition is matched. A `priority case` statement with an explicit `default` arm cannot produce a violation report. In synthesis, the `priority` keyword indicates that the designer has manually checked that all of the possible cases are specified in the non-default arms. This is equivalent to the use of the informal `full_case` directive comment commonly seen in older Verilog code.

Violation checks only apply in simulation, not in synthesized hardware, which allows for mismatches to occur. For example, where violation reports are produced but ignored for whatever reason, but the simulation does not otherwise check for the erroneous condition, the synthesis tool may produce a netlist with the invalid assumption that the condition cannot be met.

See also:

- **case__default** - Useful companion rule.
- **explicit__case__default** - Useful companion rule.
- **keyword__forbidden__unique** - Useful companion rule.
- **keyword__forbidden__unique0** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 12.4 Conditional if-else statement
- 12.5 Case statement

Rule: `keyword_forbidden_unique`

Hint

Remove `unique` keyword, perhaps replace with an assertion.

Reason

Unique-case/if constructs may mismatch between simulation and synthesis.

Pass Example (1 of 1)

```
module M;
  initial
    case (a)
      default: b = 1;
    endcase
endmodule
```

Fail Example (1 of 2)

```
module M;
  initial
    unique case (a)
      default: b = 1;
    endcase
endmodule
```

Fail Example (2 of 2)

```
module M;
  initial
    unique if (a)
      b = 1;
    else if (a)
      b = 2;
    else
      b = 3;
endmodule
```

Explanation

The keyword `unique` may be used on `if`/`else` or `case` statements to enable *violation checks* in simulation, describe design intent for synthesis, and change the semantics of condition priority.

A `unique if` statement without an explicit `else` clause will produce a *violation report* in simulation if the implicit `else` condition is matched, or more than one `if` conditions are matched. A `unique if` statement with an explicit `else` clause

will produce a violation report when more than one of the `if` conditions are matched. Thus, the conditions in a `unique if` statement may be evaluated in any order. A `unique case` statement will produce a violation report if multiple arms match the case expression.

In synthesis, the `unique` keyword on an `if/else` statement specifies that priority logic (between the conditions) is not required - a significant change in semantics vs a bare `if/else` statement. Similarly, priority logic is not required between arms of a `unique case` statement. The `unique` keyword indicates that the designer has manually checked that exactly 1 of the specified conditions must be met, so all conditions may be safely calculated in parallel. This is equivalent to the use of the informal `parallel_case` and `full_case` directive comments commonly seen in older Verilog code.

In simulation, after finding a uniqueness violation in a `unique if`, the simulator is not required to evaluate or compare the rest of the conditions. However, in a `unique case`, all case item expressions must be evaluated even once a matching arm is found. These attributes mean that the presence of side effects, e.g. `$display()` or `foo++`, may cause non-deterministic results.

Violation checks only apply in simulation, not in synthesized hardware, which allows for mismatches to occur. For example, where violation reports are produced but ignored for whatever reason, but the simulation does not otherwise check for the erroneous condition, the synthesis tool may produce a netlist with the invalid assumption that the conditions can be safely evaluated in parallel.

See also:

- **case_default** - Useful companion rule.
- **explicit_case_default** - Useful companion rule.
- **keyword_forbidden_priority** - Useful companion rule.
- **keyword_forbidden_unique0** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 12.4 Conditional if-else statement
- 12.5 Case statement

Rule: `keyword_forbidden_unique0`

Hint

Remove `unique0` keyword, perhaps replace with an assertion.

Reason

Unique0-case/if constructs may mismatch between simulation and synthesis.

Pass Example (1 of 1)

```
module M;
  initial
    case (a)
      default: b = 1;
    endcase
endmodule
```

Fail Example (1 of 2)

```
module M;
  initial begin
    unique0 case (a)
      default: b = 1;
    endcase
  end
endmodule
```

Fail Example (2 of 2)

```
module M;
  initial
    unique0 if (a)
      b = 1;
    else if (a)
      b = 2;
    else
      b = 3;
endmodule
```

Explanation

The keyword `unique0` may be used on `if`/`else` or `case` statements to enable *violation checks* in simulation, describe design intent for synthesis, and change the semantics of condition priority.

A `unique0 if` statement will produce a *violation report* in simulation if more than one `if` condition is matched. Thus, the conditions in a `unique0 if`

statement may be evaluated in any order. In synthesis, the `unique0` keyword specifies that priority logic (between the conditions) is not required - a significant change in semantics vs a bare `if/else` statement.

In synthesis, the `unique0` keyword on an `if/else` statement specifies that priority logic (between the conditions) is not required - a significant change in semantics vs a bare `if/else` statement. Similarly, priority logic is not required between arms of a `unique0 case` statement. The `unique0` keyword indicates that the designer has manually checked that exactly 0 or 1 of the specified conditions must be met, so all conditions may be safely calculated in parallel. This is equivalent to the use of the informal `parallel_case` and `full_case` directive comments commonly seen in older Verilog code.

In simulation, after finding a uniqueness violation in a `unique0 if`, the simulator is not required to evaluate or compare the rest of the conditions. However, in a `unique0 case`, all case item expressions must be evaluated even once a matching arm is found. These attributes mean that the presence of side effects, e.g. `$display()` or `foo++`, may cause non-deterministic results.

Violation checks only apply in simulation, not in synthesized hardware, which allows for mismatches to occur. For example, where violation reports are produced but ignored for whatever reason, but the simulation does not otherwise check for the erroneous condition, the synthesis tool may produce a netlist with the invalid assumption that the conditions can be safely evaluated in parallel.

See also:

- **case__default** - Useful companion rule.
- **explicit__case__default** - Useful companion rule.
- **keyword__forbidden__priority** - Useful companion rule.
- **keyword__forbidden__unique** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 12.4 Conditional if-else statement
- 12.5 Case statement

Rule: `keyword_forbidden_wire_reg`

Hint

Replace `wire` or `reg` keywords with `logic`, `tri` and/or `var`.

Reason

Explicit datatype `logic` and/or datakind `var`/`tri` better describes intent.

Pass Example (1 of 1)

```
module M;
  logic a;
endmodule
```

Fail Example (1 of 1)

```
module M;
  wire a;
  reg b;
endmodule
```

Explanation

The keywords `wire` and `reg` are present in SystemVerilog primarily for backwards compatibility with Verilog (IEEE1364-1995). In SystemVerilog, there are additional keywords, such as `logic` and `tri` with more refined semantics to better express the programmer's intent.

The LRM covers the use of `wire`: > The net types `wire` and `tri` shall be identical in their syntax and > functions; two names are provided so that the name of a net can indicate the > purpose of the net in that model.

The LRM covers the use of `reg`: > The keyword `reg` does not always accurately describe user intent, as it > could be perceived to imply a hardware register. The keyword `logic` is a > more descriptive term. `logic` and `reg` denote the same type.

See also:

- **default_nettype** - Useful companion rule.
- **inout_with_tri** - Useful companion rule.
- **input_with_var** - Useful companion rule.
- **output_with_var** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 6.6.1 Wire and tri nets
- 6.11.2 2-state (two-value) and 4-state (four-value) data types

Rule: `keyword_required_generate`

Hint

Use `generate`/`endgenerate` keywords to define generate regions.

Reason

Omitting `generate`/`endgenerate` keywords may cause issues with non-compliant tools.

Pass Example (1 of 1)

```
module M;
  generate

  if (a) begin
  end

  case (a)
      default: a;
  endcase

  for(i=0; i < 10; i++) begin
  end

  endgenerate
endmodule
```

Fail Example (1 of 3)

```
module M;
  if (a) begin
  end
endmodule
```

Fail Example (2 of 3)

```
module M;
  case (a)
      default: a;
  endcase
endmodule
```

Fail Example (3 of 3)

```
module M;
  for (genvar i=0; i < 10; i++) begin
```

```
  end
endmodule
```

Explanation

The `generate/endgenerate` keywords may be used in a module, interface, program, or checker to define a generate region. A generate region is a textual span in the module description where generate constructs may appear. Use of generate regions is optional. There is no semantic difference in the module when a generate region is used. A parser may choose to recognize the generate region to produce different error messages for misused generate construct keywords.

Some non-compliant tools may require the use of these keywords. Therefore, this rule is designed to mandate their use.

NOTE: The visual noise introduced by these keywords provides an argument against this rule.

See also:

- **keyword_forbidden_generate** - Opposite reasoning.

The most relevant clauses of IEEE1800-2017 are:

- 27.3 Generate construct syntax

Rule: `level_sensitive_always`

Hint

Replace level-sensitive `always` with `always_comb`.

Reason

Level-sensitive `always` cannot detect combinatorial/stateful (non-)blocking mistakes.

Pass Example (1 of 1)

```
module M;
  always_comb begin
  end
  always @(posedge a) begin
  end
endmodule
```

Fail Example (1 of 2)

```
module M;
  always @* begin // No sensitivity list.
  end
endmodule
```

Fail Example (2 of 2)

```
module M;
  always @ (a or b) begin // No sensitivity to posedge, negedge, or edge.
  end
endmodule
```

Explanation

In Verilog (IEEE1364), there are two language constructs which can be used to model combinatorial logic:

1. Continuous assignment to `wire` signals is specified with the `assign` keyword.
2. `reg` signals are assigned to with an `always` block, which is evaluated whenever anything in the sensitivity list changes value.

The `always` keyword can also be used for modelling sequential logic by including the edge of a signal in the sensitivity list.

The semantics of these keywords in SystemVerilog are compatible with Verilog, but additional keywords (`always_comb`, `always_ff`, and `always_latch`) should

be used to clarify intent of digital designs. The `always_*` keywords have slightly different semantics which are beneficial for synthesizable designs:

1. `always_*` processes require compiler checks that any signals driven on the LHS are not driven by any other process, i.e. `always_*` cannot infer multi-driven or tri-state logic.
2. `always_comb` processes require a compiler check that the process does not infer state.
3. `always_ff` processes require a compiler check that the process does infer state.

This rule requires that general-purpose `always` blocks have an explicit sensitivity list which includes at least one edge, thus forcing the use of `assign` or `always_comb` to specify combinatorial logic. It is possible to construct a full-featured testbench where all `always` blocks meet that requriment. The alternative rule **keyword__forbidden__always** has similar reasoning but is more strict, completely forbidding the use of general-purpose `always` blocks. It is appropriate to use **keyword__forbidden__always** on synthesizable design code, but on verification code use **level__sensitive__always** instead.

See also:

- **keyword__forbidden__always** - Alternative rule.

The most relevant clauses of IEEE1800-2017 are:

- 9.2.2 Always procedures
- 9.5 Process execution threads

Rule: `localparam_explicit_type`

Hint

Provide an explicit type in `localparam` declaration.

Reason

Explicit parameter types clarify intent and improve readability.

Pass Example (1 of 1)

```
module M;
  localparam int A = 0;
endmodule
```

Fail Example (1 of 1)

```
module M;
  localparam L = 0;
endmodule
```

Explanation

The type of a parameter is more fundmental to express intent than its value. By analogy, asking a shopkeeper for "5 oranges" is more likely to be correctly understood than simply asking for "5" without clarification. This rule requires that authors consider and specify the type of each `localparam` elaboration-time constant. Explicit types help readers to understand exactly what effects the constant might have, thus reducing the effort they need to expend reading how the parameter is used.

Without an explicit type, a localparam will take a type compatible with its constant expression. Implict types can thereby introduce discrepencies between what the author intends and how tools interpret the code. For example, interactions between the default datatype `logic`, constant functions, and case expressions can result in mismatches between simulation and synthesis. A detailed investigation into the semantics of implicit vs explicit types on SystemVerilog `parameter` and `localparam`s can be found in a tutorial paper here: https://github.com/DaveMcEwan/dmpvl/tree/master/prs/paper/ParameterDatatypes

See also:

- **localparam_type_twostate** - Useful companion rule.
- **parameter_explicit_type** - Useful companion rule.
- **parameter_type_twostate** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 6.3 Value set

- 6.11 Integer data types
- 6.20.2 Value parameters
- 6.20.4 Local parameters (localparam)

Rule: `localparam_type_twostate`

Hint

Declare `localparam` with an explicit 2-state type.

Reason

Design constants with Xs or Zs may cause simulation/synthesis mismatch.

Pass Example (1 of 1)

```
module M;
  localparam byte     A = 8'b0;
  localparam shortint B = 16'b0;
  localparam int      C = 32'b0;
  localparam longint  D = 64'b0;
  localparam bit      E = 1'b0;
endmodule
```

Fail Example (1 of 3)

```
module M;
  localparam integer A = 32'b0; // 32b
endmodule
```

Fail Example (2 of 3)

```
module M;
  localparam logic B = 1'b0; // 1b
endmodule
```

Fail Example (3 of 3)

```
module M;
  localparam reg C = 1'b0; // 1b
endmodule
```

Explanation

In order to avoid subtle bugs related to 2-state vs 4-state types and X-propagation, constants should be declared with an explicit 2-state type. Separately, all synthesizable signals should be declared with an explicit 4-state type so that a simulation can detect unknown values (Xs). For complex types such as nested packed structs, that means you need two versions of each type: a 2-state version for constants, and a 4-state version for signals. The need for this rule stems from the fact that SystemVerilog includes the concepts of both equivalence and partial

equivalence, with interactions between 2-state and 4-state structure members which invite mismatching behavior between simulation and synthesis.

The relevant quote about implicit conversion of packed structure members from 2-state to 4-state is found on page 140 of IEEE1800-2017: If all datatypes within a packed structure are 2-state, the structure as a whole is treated as a 2-state vector. If any datatype within a packed structure is 4-state, the structure as a whole is treated as a 4-state vector. If there are also 2-state members in the structure, there is an implicit conversion from 4-state to 2-state when reading those members and from 2-state to 4-state when writing them.

For constants of simple datatypes, it is trivial to visually check that their values do not contain Xs or Zs. However, for constants of more complex datatypes, e.g. nested packed structures, the use of constant functions may infer Xs as (accidentally) unassigned members will take their default values. Default values are specified in IEEE1800-2017 Table 6-7. This can be particularly subtle when a single member of a deeply nested packed struct is wrongly declared with a 4-state type, e.g. `logic`, thus forcing all other (previously 2-state) members to have a default value of `'X` instead of the expected `'0`.

The equivalence operators ("case" equality/inequality) are written as 3 characters each (`===`, `!==`) and can only return false or true, e.g. `4'b01XZ === 4'b01XZ -> 1'b1` (true). The partial equivalence operators ("logical" equality/inequality) are written as 2 characters each (`==`, `!=`) and may return false, true, or unknown, e.g. `4'b01XZ === 4'b01XZ -> 1'bx` (unknown).

Let `w` be a 4-state signal which a systhesis tool will implement with a collection of wires. Let `c2` and `c4` be constants with 2-state and 4-state types respectively. Without loss of generality, only the case/logical equality operators are required to demonstrate troublesome expressions.

- `w === c2` Result may be false (`1'b0`), true (`1'b1`). If `w` contains any Xz or Zs, then the result is false (`1'b0`). This is *not* desired behavior as Xs in `w` are hidden and simulation is likely, but not certain, to mismatch synthesized hardware.
- `w === c4` Result may be false (`1'b0`), true (`1'b1`). If `w` contains any Xz or Zs, then the result is is true iff the constant `c4` has been defined with corresponding Xs and Zs. Comparison between unknown and unknown is all but certain, to mismatch synthesized hardware.
- `w == c2` Result may be false (`1'b0`), true (`1'b1`), or unknown (`1'bX`). If `w` contains any Xs or Zs, then the result is unknown. This is desired behavior as it sufficiently models synthesized physical hardware.
- `w == c4` Result may be false (`1'b0`), true (`1'b1`), or unknown (`1'bX`). If `c4` contains any Xs or Zs, then the result will always be unknown. While that may be noticed early in simulation, unwitting designers may be tempted to prevent X-propagation on the result, thus hiding any issues with Xs or Zs on `w`.

The use of 4-state constants with wildcard equality operators is a slightly different

usecase. If wildcard equality operators are used with 4-state constants in your code, this rule should be considered on a case-by-case basis.

See also:

- **localparam_explicit_type** - Useful companion rule.
- **parameter_explicit_type** - Useful companion rule.
- **parameter_type_twostate** - Useful companion rule, equivalent reasoning.

The most relevant clauses of IEEE1800-2017 are:

- 6.8 Variable declarations
- 6.11 Integer data types
- 7.2.1 Packed structures
- 11.4.5 Equality operators
- 11.4.6 Wildcard equality operators

NOTE: The reasoning behind this rule invites the use of other rules:

1. Check that members of a packed structure definition are either all 2-state or all 4-state.
2. Check for the use of case equality operators.
3. Check that functions are not declared with a 4-state type.

Rule: `loop_variable_declaration`

Hint

Declare the loop variable within the loop, i.e. `for (int i`.

Reason

Minimizing the variable's scope avoids common coding errors.

Pass Example (1 of 1)

```
module M;
  initial
    for(int i=0; i < 10; i++) begin
    end
endmodule
```

Fail Example (1 of 1)

```
module M;
  initial begin
    int i;
    for(i=0; i < 10; i++) begin
    end
  end
endmodule
```

Explanation

A loop variable may be declared either inside the loop, e.g. `for (int i = 0; i < 5; i++)`, or outside the loop, e.g. `int i; ... for (i = 0; i < 5; i++)`. This rule mandates that the scope of a loop variable, e.g. `i`, is minimized to avoid a common class of coding mistake where `i` is erroneously used outside the loop.

See also:

- **function_with_automatic** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 12.7 Loop statements

Rule: `multiline_for_begin`

Hint

Add `begin`/`end` around multi-line `for` statement.

Reason

Without `begin`/`end`, the loop statement may be confusing.

Pass Example (1 of 1)

```
module M;
  always_comb begin
    for (int a=0; a < 10; a++) begin
      a = 0;
    end

    for (int a=0; a < 10; a++) a = 0;
  end
endmodule
```

Fail Example (1 of 2)

```
module M;
  always_comb begin
    for (int i=0; i < 10; i++)
      a = 0;
  end
endmodule
```

Fail Example (2 of 2)

```
module M;
  always_comb begin
    for (int i=0; i < 10; i++) a = 0; // This is okay.

    for (int i=0; i < 10; i++) // Catch any for-loop, not only the first.
      a = 0;
  end
endmodule
```

Explanation

This rule is to help prevent a common class of coding mistake, where a future maintainer attempts to add further statements to the loop, but accidentally writes something different.

See also:

- **multiline_if_begin** - Useful companion rule.
- **style_indent** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 12.7 Loop statements

Rule: `multiline_if_begin`

Hint

Add `begin`/`end` around multi-line `if` statement.

Reason

Without `begin`/`end`, the conditional statement may be confusing.

Pass Example (1 of 1)

```
module M;
  always_comb begin
    if (a) begin
        a = 0;
    end

    if (a) begin
        a = 0;
    end else if (a) begin
        a = 0;
    end

    if (a) begin
        a = 0;
    end else if (a) begin
        a = 0;
    end else begin
        a = 0;
    end

    if (a) a = 0;
    else if (a) a = 0;
    else a = 0;
  end
endmodule
```

Fail Example (1 of 5)

```
module M;
  always_comb
    if (a)
      a = 0; // Missing begin/end.
endmodule
```

Fail Example (2 of 5)

```
module M;
  always_comb
    if (a) begin
      a = 0;
    end else if (a)
      a = 0; // Missing begin/end.
endmodule
```

Fail Example (3 of 5)

```
module M;
  always_comb
    if (a) begin
      a = 0;
    end else if (a) begin
      a = 0;
    end else
      a = 0; // Missing begin/end.
endmodule
```

Fail Example (4 of 5)

```
module M;
  always_comb begin
    if (a)
      a = 0; // Missing begin/end.
  end
endmodule
```

Fail Example (5 of 5)

```
module M;
  always_comb begin
    if (a) a = 0; // This conditional statement is okay.
    else if (a) a = 0;
    else a = 0;

    if (a)   // Check all if-statements, not only the first.
      a = 0; // Missing begin/end.
  end
endmodule
```

Explanation

This rule is to help prevent a common class of coding mistake, where a future maintainer attempts to add further statements to the conditional block, but accidentally writes something different.

See also:

- **multiline_for_begin** - Useful companion rule.
- **style_indent** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 12.4 Conditional if-else statement

Rule: `non_ansi_module`

Hint

Declare `module` header in ANSI style.

Reason

Non-ANSI module headers are visually noisy and error-prone.

Pass Example (1 of 3)

```
module M      // An ANSI module has ports declared in the module header.
  ( input  a
  , output b
  );
endmodule
```

Pass Example (2 of 3)

```
module M;     // A module with no ports is also ANSI.
endmodule
```

Pass Example (3 of 3)

```
module M      // Declaring ports in the header with default direction (inout)
  ( a         // also specifies an ANSI module.
  , b
  );
endmodule
```

Fail Example (1 of 1)

```
module M
  ( a
  , b
  );
  input  a;   // Declaring ports outside the module header declaration
  output b;   // makes this a non-ANSI module.
endmodule
```

Explanation

There are two ways to declare a module header in SystemVerilog:

1. ANSI style - newer, neater, more succinct, compatible with IEEE1364-2001.
2. non-ANSI style - additionally compatible with older Verilog (IEEE1364-1995).

Examples of both styles are given in IEEE1364-2001 (e.g. pages 180 vs 182) and IEEE1800-2017 (e.g. pages 702 vs 700).

The non-ANSI style separates the declaration of ports, their direction, and their datatype. In addition to requiring more text, and visual noise, to convey the same information, the non-ANSI style encourages simple coding mistakes where essential attributes may be forgotten. This rule requires that module headers are declared using the ANSI style.

See also:

- No related rules.

The most relevant clauses of IEEE1800-2017 are:

- 23.2 Module definitions

Rule: `non_blocking_assignment_in_always_comb`

Hint

Remove non-blocking assignment in `always_comb`.

Reason

Scheduling between blocking and non-blocking assignments is non-deterministic.

Pass Example (1 of 1)

```
module M;
  always_comb
    x = 0;
endmodule
```

Fail Example (1 of 1)

```
module M;
  always_comb
    x <= 0;
endmodule
```

Explanation

Simulator event ordering between blocking and non-blocking assignments is undefined, so observed behavior is simulator-dependent. This rule forbids the use of non-blocking assigments (using the `<=` operator) in `always_comb` blocks. Instead, use the blocking assignment operator `=`.

An excellent paper detailing the semantics of Verilog blocking and non-blocking assignments is written by Clifford E Cummings and presented at SNUG-2000, "Nonblocking Assignments in Verilog Synthesis, Coding Styles that Kill".

See also:

- **blocking_assignment_in_always_ff** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 4.9.3 Blocking assignment
- 4.9.4 Non-blocking assignment
- 9.2.2.2 Combinational logic `always_comb` procedure
- 9.4.2 Event control
- 10.4.1 Blocking procedural assignments
- 10.4.2 Nonblocking procedural assignments

Rule: `output_with_var`

Hint

Specify `var` datakind on `output` ports.

Reason

Explicit datakind of output ports should be consistent with input ports.

Pass Example (1 of 1)

```
module M
  ( output var logic a
  );
endmodule
```

Fail Example (1 of 1)

```
module M
  ( output logic a
  );
endmodule
```

Explanation

This check mandates that each `output` port must be explicitly declared as a variable, rather than the default nettype or implict datakind.

The rules for determining port kind, datatype, and direction are specified in IEEE1800-2017 Clause 23.2.2.3 and facilitate various shorthand notations which are backwards compatible with the semantics of Verilog (IEEE1364-1995):

- `output a` -> `output tri logic a` The implicit datatype is `logic` and the default nettype is `tri` (without overriding via the `` `default_nettype `` compiler directive).
- `output wire a` -> `output tri logic a` Again, using the implicit datatype of `logic`; As `wire` is an alias for `tri`, this is equivalent to the above example.
- `output wire logic a` -> `output tri logic a` Again, even with an explicit datatype (`logic`), the `wire` keyword is simply an alias for the datakind `tri`.
- `output logic a` -> `output var logic a` This time the datakind is implicit, but the datatype is *explicit*, so the inferred datakind is `var`.

When the datatype is implicit and the default nettype is overridden to none, i.e. with the compiler directive `` `default_nettype none ``, output ports require an explicit datakind.

Although the semantics of `output a` are equivalent in IEEE1364-1995, the intent is not clearly described, and the difference to `output logic a` is unintuitive. An author should use `output` to declare ports which should only be driven internally, and `inout` to declare ports which may also be driven externally. In order to describe the intended uni-directional behavior, `output` ports must be declared with an explicit `var` datakind, thus requiring the compiler to check that the output is only driven from within the module (otherwise, emit an error).

See also:

- **default_nettype_none** - Useful companion rule.
- **inout_with_tri** - Suggested companion rule.
- **output_with_var** - Suggested companion rule.
- **prefix_output** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 6.5 Nets and variables
- 6.6 Net types
- 22.8 default nettype
- 23.2.2 Port declarations

Rule: `parameter_default_value`

Hint

Specify `parameter` with an explicit default value.

Reason

Default values are required by some tools and clarify intent.

Pass Example (1 of 1)

```
module M
  #(parameter int P = 0
  ) ();
endmodule
```

Fail Example (1 of 3)

```
module M
  #(parameter int P // Type is specified (good), but default value isn't (bad).
  ) ();
endmodule
```

Fail Example (2 of 3)

```
module M
  #(parameter Q // Neither type or default value are specified (very bad).
  ) ();
endmodule
```

Fail Example (3 of 3)

```
module M
  #(parameter int P = 0
  , R // Legal, but even less clear about the author's intention.
  ) ();
endmodule
```

Explanation

It is legal for parameters declared in a parameter port list to omit a default value (an elaboration-time constant), thus setting to parameter value to the default value of its type when not overridden. This language feature can be used by module authors to force integrators to choose an override value, by ensuring that the default is invalid.

```
module M #(parameter int NE0) ();
  if (0 == NE0) $error("MUST_OVERRIDE must not be zero.");
```

```
endmodule

module Parent ();
  M u_bad (); // This causes elaboration error.
  M #(NE0=1) u_good ();
endmodule
```

The example above uses a system elaboration task to explicitly force an elaboration error, but there are several ways to implictly cause elaboration errors. Relying on the type's default value can cause problems for two reasons:

1. Some tools do not support this syntax.
2. Simply omitting the default value is unclear about the author's intention, particularly when the type is also omitted.

This rule checks that all parameter ports have an explicit default value.

See also:

- **parameter_explicit_type** - Useful companion rule.
- **parameter_type_twostate** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 6.20.1 Parameter declaration syntax
- 6.20.2 Value parameters
- 23.2.3 Parameterized modules
- A.10 Footnotes (normative), number 18.

Rule: `parameter_explicit_type`

Hint

Provide an explicit type in `parameter` declaration.

Reason

Explicit parameter types clarify intent and improve readability.

Pass Example (1 of 1)

```
module M
  #(parameter int a = 0
  ) ();
endmodule
```

Fail Example (1 of 2)

```
module M
  #(parameter a = 0
  ) ();
endmodule
```

Fail Example (2 of 2)

```
module M
  #(parameter a = int'(0)
  ) ();
endmodule
```

Explanation

The type of a parameter is more fundmental to express intent than its value. By analogy, asking a shopkeeper for "5 oranges" is more likely to be correctly understood than simply asking for "5" without clarification. This rule requires that authors consider and specify the type of each module `parameter` port. Explicit types help readers, particularly large-scale integrators, to understand exactly what values are expected, thus reducing the effort they need to expend reading how the parameter is used.

Without an explicit type, a module parameter will take a type compatible with its default assignment, or a type compatible with any override values. Implict types can thereby introduce discrepencies between what the author intends and how tools interpret the code. For example, interactions between the default datatype `logic`, constant functions, and case expressions can result in mismatches between simulation and synthesis. A detailed investigation into the semantics of implicit vs explicit types on SystemVerilog `parameter` and `localparam`s can be found in

a tutorial paper here: https://github.com/DaveMcEwan/dmpvl/tree/master/prs/paper/ParameterDatatypes

See also:

- **localparam_explicit_type** - Useful companion rule.
- **localparam_type_twostate** - Useful companion rule.
- **parameter_type_twostate** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 6.3 Value set
- 6.11 Integer data types
- 6.20.2 Value parameters
- 23.2.3 Parameterized modules

Rule: `parameter_in_generate`

Hint

Replace `parameter` keyword with `localparam`.

Reason

In a generate block, `localparam` properly describes the non-overridable semantics.

Pass Example (1 of 1)

```
module M;
  for (genvar i=0; i < 5; i++) begin
    localparam int P1 = 1;
  end

  if (1) begin
    localparam int P2 = 2;
  end else begin
    localparam int P3 = 3;
  end

  case (1)
    0: begin
      localparam int P4 = 4;
    end
    default: begin
      localparam int P5 = 5;
    end
  endcase
endmodule
```

Fail Example (1 of 1)

```
module M;
  for (genvar i=0; i < 5; i++) begin
    parameter int P1 = 1;
  end

  if (1) begin
    parameter int P2 = 2;
  end else begin
    parameter int P3 = 3;
  end
```

```
  case (1)
    0: begin
      parameter int P4 = 4;
    end
    default: begin
      parameter int P5 = 5;
    end
  endcase
endmodule
```

Explanation

In the context of a generate block, the `parameter` keyword is a synonym for the `localparam` keyword. This rule encourages the author to consider that the constant may not be overridden and convey that explictly.

See also:

- **parameter_in_package**

The most relevant clauses of IEEE1800-2017 are:

- 6.20.4 Local parameters (localparam)
- 27 Generate constructs, particularly 27.2 Overview.

Rule: `parameter_in_package`

Hint

Replace `parameter` keyword with `localparam`.

Reason

In a package, `localparam` properly describes the non-overridable semantics.

Pass Example (1 of 1)

```
package P;
  localparam int A = 1;
endpackage
```

Fail Example (1 of 1)

```
package P;
  parameter int A = 1;
endpackage
```

Explanation

In the context of a package, the `parameter` keyword is a synonym for the `localparam` keyword. This rule encourages the author to consider that the constant may not be overridden and convey that explictly.

See also:

- **parameter_in_generate**

The most relevant clauses of IEEE1800-2017 are:

- 6.20.4 Local parameters (localparam)
- 26 Packages

Rule: `parameter_type_twostate`

Hint

Declare `parameter` with an explicit 2-state type.

Reason

Design constants with Xs or Zs may cause simulation/synthesis mismatch.

Pass Example (1 of 1)

```
module M
  #(parameter byte     A = 8'b0
  , parameter shortint B = 16'b0
  , parameter int      C = 32'b0
  , parameter longint  D = 64'b0
  , parameter bit      E = 1'b0
  ) ();
endmodule
```

Fail Example (1 of 3)

```
module M
  #(parameter integer A = 32'b0
  ) ();
endmodule
```

Fail Example (2 of 3)

```
module M
  #(parameter logic B = 1'b0
  ) ();
endmodule
```

Fail Example (3 of 3)

```
module M
  #(parameter reg C = 1'b0
  , logic         Z = 1'b0 // TODO: Z isn't caught.
  ) ();
endmodule
```

Explanation

The reasoning behind this rule is equivalent to that of **localparam_type_twostate**. Please see the explanation for **localparam_type_twostate**.

See also:

- **localparam_explicit_type** - Useful companion rule.
- **localparam_type_twostate** - Useful companion rule, equivalent reasoning.
- **parameter_explicit_type** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 6.8 Variable declarations
- 6.11 Integer data types
- 7.2.1 Packed structures
- 11.4.5 Equality operators
- 11.4.6 Wildcard equality operators

Rule: `sequential_block_in_always_comb`

Hint

Keywords `begin` and `end` are forbidden within `always_comb`.

Reason

Sequential blocks within `always_comb` introduce sequential dependencies.

Pass Example (1 of 1)

```
module M;
  always_comb
    a = b;

  always_comb
    if (x)
      a = b;
    else
      a = c;

  always_comb
    case (x)
      one:     a = x;
      two:     a = y;
      default: a = z;
    endcase
endmodule
```

Fail Example (1 of 4)

```
module M;
  always_comb begin
    a = z;
  end
endmodule
```

Fail Example (2 of 4)

```
module M;
  always_comb
    if (bar) begin
      b = z;
    end
endmodule
```

Fail Example (3 of 4)

```
module M;
  always_comb
    if (bar) c = z;
    else begin
      c = z;
    end
endmodule
```

Fail Example (4 of 4)

```
module M;
  always_comb
    case (bar)
      one: begin
        d = z;
      end
      two: d = z;
      default: d = z;
    endcase
endmodule
```

Explanation

This rule has two purposes:

1. Prevent mismatches between simulation and synthesis.
2. Avoid unnecessarily restricting the simulator's scheduler.

An `always_comb` block is scheduled for execution whenever any of the RHS variables (or nets) change value, which can lead to unnecessary sequential dependencies. For example, the following block is requires that the "expensive" (in terms of CPU time) function must be called to update `a` whenever `z` changes value, in addition to whenever `y` changes value.

```
always_comb begin
  a = expensive(y);
  b = z;
end
```

The above example can be reformed to allow the simulator more flexibility in how it schedules processes. Logical equivalence is maintained, and a synthesis tool will interpret these examples equivalently. Note that continuous assignment (using `assign`) is not sensitive to changes in `y` because functions are not transparent.

```
always_comb a = expensive(y);
assign b = z;
```

This rule is intended for synthesisable code only, not testbench code. Testbenches often necessarily rely on sequential dependencies, but a synthesis tool for digital synchronous logic will produce a netlist without sequential dependencies. That can lead to a mismatch between simulation and synthesis.

See also:

- **style_indent** - Useful companion rule.
- **sequential_block_in_always_ff** - Similar rule, different purpose.
- **sequential_block_in_always_latch** - Similar rule, different purpose.

The most relevant clauses of IEEE1800-2017 are:

- 4.6 Determinisim
- 9.2.2.2 Combinational logic always_comb procedure
- 9.3.1 Sequential blocks
- 10.3 Continuous assignments
- 10.4 Procedural assignments

Rule: `sequential_block_in_always_ff`

Hint

Keywords `begin` and `end` are forbidden within `always_ff`.

Reason

Sequential blocks within `always_ff` may encourage overly-complex code.

Pass Example (1 of 1)

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

  always_ff @(posedge clk)
    if (x) q <= d;

  always_ff @(posedge clk)
    if (rst) q <= 0;
    else     q <= d;

  always_ff @(posedge clk)
    case (foo)
      one:     q <= x;
      two:     r <= y;
      default: s <= z;
    endcase
endmodule
```

Fail Example (1 of 4)

```
module M;
  always_ff @(posedge clk) begin
    a <= z;
  end
endmodule
```

Fail Example (2 of 4)

```
module M;
  always_ff @(posedge clk)
    if (x) begin
      a <= z;
    end
endmodule
```

Fail Example (3 of 4)

```
module M;
  always_ff @(posedge clk)
    if (x) a <= z;
    else begin
      a <= z;
    end
endmodule
```

Fail Example (4 of 4)

```
module M;
  always_ff @(posedge clk)
    case (x)
      foo: begin
        a <= z;
      end
      bar: a <= z;
      default: a <= z;
    endcase
endmodule
```

Explanation

The consequences/purposes of this rule are perhaps subtle, particulaly in how it works with companion rules **default_nettype_none**, **explicit_case_default**, **explicit_if_else**, **style_indent**, and a guideline to avoid `for` within `always_ff`.

In conjunction with these companion rules and guidelines, a nice consequence is that editing code after the fact is "safe", i.e. not error prone. Without `begin`/`end` adding another statement to a single-statement conditional block may be error prone. This is why coding styles for C-style languages often forbid writing `if (a) foo;`, instead requiring `if (a) { foo; }` - because it's easy to forget to add braces with an additional statement like `if (a) { foo; bar; }`. While a simple rule is to require the use of `begin` and `end` (or `{` and `}`), this introduces visual noise. The goal is to guard programmers from making a simple and easy mistake. This rule, in conjunction with the companion rules, achieves the same goal using a different approach, in addition to providing other nice properties.

With a sequential block (marked by `begin` and `end`) you can assign to multiple signals in a leaf conditon which can easily result in difficult-to-comprehend logic, e.g.:

```
always_ff @(posedge clk) begin
  if (cond) begin
    foo_q <= foo_d;       // Block was originally written for foo.
```

```
    bar_q <= bar_d;       // This was added later.
  end
  bar_q <= bar_d;         // What happens to bar_q?
end
```

By forbidding sequential blocks, you enforce that exactly signal is assigned to per leaf condition. A nice consequence is that exactly one signal is updated on each evaluation of the `always_ff` block. IEEE1800-2017 specifies that if a signal is assigned to in an `always_ff` block, then it shall not be assigned to by any other block (compile error).

An example with multiple signals in the `always_ff` is a ping-pong buffer (AKA shunt buffer, storage of a 2-entry fifo). Due to the construction, you can be sure that you never update both entries at the same time, except when that is clearly explicit.

```
  // Enforced exclusive updates, with reset and clockgate.
  always_ff @(posedge clk)
    if (rst)
      {ping_q, pong_q} <= '0; // Assignment to multiple signals is explicit.
    else if (clkgate)
      if (foo) ping_q <= foo;
      else     pong_q <= foo;
    else // Optional explicit else.
      {ping_q, pong_q} <= {ping_q, pong_q};
```

Another example with multiple signals is an address decoder. Due to the construction, you can be sure that you aren't accidentally updating multiple registers on a write to one address.

```
  // Enforced exclusivity of address decode.
  always_ff @(posedge clk)
    if (write)
      case (addr)
        123:        red_q   <= foo;
        456:        blue_q  <= foo;
        789:        green_q <= foo;
        default:    black_q <= foo; // Optional explicit default.
      endcase
```

When you don't need those exclusivity properties, only one signal should be updated per `always_ff`. That ensures that the code doesn't get too deep/complex/unintuitive and drawing a logical diagram is straightforward. This is the expected form for most signals.

```
  always_ff @(posedge clk)
    if (rst)          ctrl_q <= '0;
    else if (clkgate) ctrl_q <= ctrl_d;
    else              ctrl_q <= ctrl_q; // Optional explicit else.
```

See also:

- **default__nettype__none** - Useful companion rule.
- **explicit__case__default** - Useful companion rule.
- **explicit__if__else** - Useful companion rule.
- **style__indent** - Useful companion rule.
- **sequential__block__in__always__comb** - Similar rule, different purpose.
- **sequential__block__in__always__latch** - Similar rule, different purpose.

The most relevant clauses of IEEE1800-2017 are:

- 4.6 Determinisim
- 9.2.2.4 Sequential logic always_ff procedure
- 9.3.1 Sequential blocks
- 9.4.2 Event control
- 12.4 Conditional if-else statement
- 12.5 Case statement
- 12.7 Loop statements

Rule: `sequential_block_in_always_latch`

Hint

Keywords `begin` and `end` are forbidden within `always_latch`.

Reason

Sequential blocks within `always_latch` may encourage overly-complex code.

Pass Example (1 of 1)

```
module M;
  always_latch
    if (foo) a <= b;

  always_latch
    if (foo) b <= y;
    else     b <= z;

  always_latch
    case (foo)
      one:     a <= x;
      two:     b <= y;
      default: c <= z;
    endcase
endmodule
```

Fail Example (1 of 4)

```
module M;
  always_latch begin
    a <= z;
  end
endmodule
```

Fail Example (2 of 4)

```
module M;
  always_latch
    if (x) begin
      a <= z;
    end
endmodule
```

Fail Example (3 of 4)

```
module M;
  always_latch
    if (x) a <= z;
    else begin
      a <= z;
    end
endmodule
```

Fail Example (4 of 4)

```
module M;
  always_latch
    case (x)
      foo: begin
        a <= z;
      end
      bar: a <= z;
      default: a <= z;
    endcase
endmodule
```

Explanation

The explanation of **sequential_block_in_always_ff**, and much of the explanation of **sequential_block_in_always_comb**, also applies to this rule. Main points are that avoiding `begin`/`end` helps protect the programmer against simple mistakes, provides exclusivity properties by construction, and avoids restricting simulator scheduling decisions.

See also:

- **default_nettype_none** - Useful companion rule.
- **explicit_case_default** - Useful companion rule.
- **explicit_if_else** - Useful companion rule.
- **style_indent** - Useful companion rule.
- **sequential_block_in_always_comb** - Similar rule, different purpose.
- **sequential_block_in_always_ff** - Similar rule, different purpose.

The most relevant clauses of IEEE1800-2017 are:

- 4.6 Determinisim
- 49.2.2.3 Latched logic always_latch procedure
- 49.3.1 Sequential blocks
- 49.4.2 Event control
- 412.4 Conditional if-else statement
- 412.5 Case statement
- 412.7 Loop statements

Naming Convention Rules

Rules for checking against naming conventions are named with either the suffix `_with_label` or one of these prefixes:

- `prefix_`
- `(lower|upper)camelcase_`
- `re_(forbidden|required)_`

Naming conventions are useful to help ensure consistency across components in large projects. A naming convention might be designed with several, sometimes competing, points of view such as:

- Enable simple identification of code's owner, e.g. "Prefix all module identifiers with `BlueTeam_` at the point of declaration". This makes it easy for the blue team to review their own code, without being distracted by other team's code. Not specific to SystemVerilog, i.e also applicable to VHDL.
- Enhance readability of netlists, e.g. "Prefix all module instances with `u_`, interface instances with `uin_`, and generate blocks with `l_`". This facilitates straightforward translation from a netlist identifier to its corresponding identifier in SystemVerilog. Not specific to SystemVerilog, i.e also applicable to VHDL.
- Enhance readability of code for integrators and reviewers, e.g. "Prefix all ports with `i_`, `o_`, or `b_` for inputs, outputs, and bi-directionals respectively". This allows a reader to glean important information about how ports and internal logic are connected without the need to scroll back-and-forth through a file and/or memorize the portlist.
- Add redundancy to capture design intent, e.g. "Suffix every signal which should infer a flip-flop with `_q`". By using conventional terminology (`d` for input, `q` for output) readers will be alerted to investigate any flip-flops without this prefix as the tools may not be treating the code as the original author intended. Some example suffixes include:
  - `_d`: Input to a flip-flop.
  - `_q`: Output from a flip-flop.
  - `_lat`: Output from a latch.
  - `_mem`: Memory model.
  - `_a`: Asynchronous signal.
  - `_n`: Active-low signal.
  - `_dp`, `_dn`: Differential positive/negative pair.
  - `_ana`: Analog signal.
  - `_55MHz`: A signal with a required operating frequency.
- On the above two points, prefixes are redundant re-statements of information which must is explicit in SystemVerilog semantics, and suffixes are redundant clarifications of information which can only be specified implictly in SystemVerilog.

The rules `re_forbidden_*` can also be used to restrict language features. For example, if a project requires that interfaces must never be used, you can enable

the rule `re_forbidden_interface` and configure it to match all identifier strings. By forbidding all possible identifiers at the point of declaration, no interfaces may be specified. For example:

```
[option]
re_forbidden_interface = ".*"

[rules]
re_forbidden_interface = true
```

Rule: `generate_case_with_label`

Hint

Use a label with prefix "l_" on conditional generate block.

Reason

Unnamed generate blocks imply unintuitive hierarchical paths.

Pass Example (1 of 1)

```
module A;
  generate case (2'd3)
    2'd1:     begin: l_nondefault wire c = 1'b0; end
    default:  begin: l_default    wire c = 1'b0; end
  endcase endgenerate
endmodule
```

Fail Example (1 of 5)

```
module M;
  case (2'd0)             // No begin/end delimiters.
    2'd1:
      logic a = 1'b0;
    default:
      logic a = 1'b0;
  endcase
endmodule
```

Fail Example (2 of 5)

```
module M;
  case (2'd1)             // begin/end delimiters, but no label.
    2'd1: begin
      logic b = 1'b0;
    end
    default: begin
      logic b = 1'b0;
    end
  endcase
endmodule
```

Fail Example (3 of 5)

```
module M;
  case (2'd2)             // With label, but no prefix.
    2'd1: begin: foo
```

```
      logic c = 1'b0;
    end: foo               // NOTE: With optional label on end.
    default: begin: bar
      logic c = 1'b0;
    end                    // NOTE: Without optional label on end.
  endcase
endmodule
```

Fail Example (4 of 5)

```
module M;
  case (2'd4)              // Without default arm.
    2'd1: begin: foo
      logic e = 1'b0;
    end
  endcase
endmodule
```

Fail Example (5 of 5)

```
module M;
  case (2'd5)              // Without non-default arm.
    default: begin: bar
      logic f = 1'b0;
    end
  endcase
endmodule
```

Explanation

Conditional generate constructs select zero or one blocks from a set of alternative generate blocks within a module, interface, program, or checker. The selection of which generate blocks are instantiated is decided during elaboration via evaluation of constant expressions. Generate blocks introduce hierarchy within a module, whether they are named or unnamed. Unnamed generate blocks are assigned a name, e.g. `genblk5`, which other tools can use and depend on. For example, to find a specific DFF in a netlist you could use a hierarchical path like `top.genblk2[3].u_cpu.genblk5.foo_q`. The naming scheme for unnamed generated blocks is defined in IEEE1800-2017 clause 27.6.

These implicit names are not intuitive for human readers, so this rule is designed to check three things:

1. The generate block uses `begin`/`end` delimiters.
2. The generate block has been given a label, e.g. `begin: mylabel`.
3. The label has an appropriate prefix, e.g. `begin: l_mylabel` starts with the string `l_`.

The prefix is useful to when reading hierarchical paths to distinguish between module/interface instances and generate blocks. For example, `top.l_cpu_array[3].u_cpu.l_debugger.foo_q` provides the reader with more useful information than `top.genblk2[3].u_cpu.genblk5.foo_q`.

See also:

- **generate_for_with_label** - Similar reasoning, useful companion rule.
- **generate_if_with_label** - Equivalent reasoning, useful companion rule.
- **prefix_instance** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 27.5 Conditional generate constructs
- 27.6 External names for unnamed generate blocks

Rule: `generate_for_with_label`

Hint

Use a label with prefix "l_" on loop generate block.

Reason

Unnamed generate blocks imply unintuitive hierarchical paths.

Pass Example (1 of 1)

```
module M;
  for(genvar i=0; i < 10; i++) begin: l_a
  end
endmodule
```

Fail Example (1 of 3)

```
module M;
  for (genvar i=0; i < 10; i++) // No begin/end delimeters.
    assign a[i] = i;
endmodule
```

Fail Example (2 of 3)

```
module M;
  for (genvar i=0; i < 10; i++) begin // begin/end delimiters, but no label.
    assign a[i] = i;
  end
endmodule
```

Fail Example (3 of 3)

```
module M;
  for (genvar i=0; i < 10; i++) begin: foo // With label, but no prefix.
    assign a[i] = i;
  end
endmodule
```

Explanation

A loop generate construct allows a single generate block to be instantiated multiple times within a module, interface, program, or checker. The selection of which generate blocks are instantiated is decided during elaboration via evaluation of constant expressions. Generate blocks introduce hierarchy within a module, whether they are named or unnamed. Unnamed generate blocks are assigned a name, e.g. `genblk5`, which other tools can use and depend on. For

example, to find a specific DFF in a netlist you could use a hierarchical path like `top.genblk2[3].u_cpu.genblk5.foo_q`. The naming scheme for unnamed generated blocks is defined in IEEE1800-2017 clause 27.6.

These implicit names are not intuitive for human readers, so this rule is designed to check three things:

1. The generate block uses `begin`/`end` delimiters.
2. The generate block has been given a label, e.g. `begin: mylabel`.
3. The label has an appropriate prefix, e.g. `begin: l_mylabel` starts with the string `l_`.

The prefix is useful to when reading hierarchical paths to distinguish between module/interface instances and generate blocks. For example, `top.l_cpu_array[3].u_cpu.l_debugger.foo_q` provides the reader with more useful information than `top.genblk2[3].u_cpu.genblk5.foo_q`.

See also:

- **generate_case_with_label** - Similar reasoning, useful companion rule.
- **generate_if_with_label** - Similar reasoning, useful companion rule.
- **prefix_instance** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 27.4 Loop generate constructs
- 27.6 External names for unnamed generate blocks

Rule: `generate_if_with_label`

Hint

Use a label with prefix "l_" on conditional generate block.

Reason

Unnamed generate blocks imply unintuitive hierarchical paths.

Pass Example (1 of 1)

```
module M;
  if (a) begin: l_abc
  end else if (b) begin: l_def
  end else begin: l_hij
  end
endmodule
```

Fail Example (1 of 8)

```
module M;
  if (x)                          // No begin/end delimeters.
    assign a = 0;                 // if condition.
  else if (x) begin: l_def
    assign a = 1;
  end else begin: l_hij
    assign a = 2;
  end
endmodule
```

Fail Example (2 of 8)

```
module M;
  if (x) begin: l_abc
    assign a = 0;
  end else if (x)                 // No begin/end delimeters.
    assign a = 1;                 // else-if condition.
  else begin: l_hij
    assign a = 2;
  end
endmodule

// TODO: This isn't caught.
module M;
  if (x) begin: l_abc
    assign a = 0;
  end else if (x) begin: l_def
```

```
    assign a = 1;
  end else                        // No begin/end delimeters.
    assign a = 2;                 // else condition
endmodule
```

Fail Example (3 of 8)

```
module M;
  if (x) begin                    // begin/end delimiters, but no label.
    assign a = 0;                 // if condition.
  end else if (x) begin: l_def
    assign a = 1;
  end else begin: l_hij
    assign a = 2;
  end
endmodule
```

Fail Example (4 of 8)

```
module M;
  if (x) begin: l_abc
    assign a = 0;
  end else if (x) begin           // begin/end delimiters, but no label.
    assign a = 1;                 // else-if condition.
  end else begin: l_hij
    assign a = 2;
  end
endmodule
```

Fail Example (5 of 8)

```
module M;
  if (x) begin: l_abc
    assign a = 0;
  end else if (x) begin: l_def
    assign a = 1;
  end else begin                  // begin/end delimiters, but no label.
    assign a = 2;                 // else condition
  end
endmodule
```

Fail Example (6 of 8)

```
module M;
  if (x) begin: foo               // With label, but no prefix.
    assign a = 0;                 // if condition.
  end else if (x) begin: l_def
```

```
    assign a = 1;
  end else begin: l_hij
    assign a = 2;
  end
endmodule
```

Fail Example (7 of 8)

```
module M;
  if (x) begin: l_abc
    assign a = 0;
  end else if (x) begin: foo    // With label, but no prefix.
    assign a = 1;               // else-if condition.
  end else begin: l_hij
    assign a = 2;
  end
endmodule
```

Fail Example (8 of 8)

```
module M;
  if (x) begin: l_abc
    assign a = 0;
  end else if (x) begin: l_def
    assign a = 1;
  end else begin: foo           // With label, but no prefix.
    assign a = 2;               // else condition
  end
endmodule
```

Explanation

Conditional generate constructs select zero or one blocks from a set of alternative generate blocks within a module, interface, program, or checker. The selection of which generate blocks are instantiated is decided during elaboration via evaluation of constant expressions. Generate blocks introduce hierarchy within a module, whether they are named or unnamed. Unnamed generate blocks are assigned a name, e.g. `genblk5`, which other tools can use and depend on. For example, to find a specific DFF in a netlist you could use a hierarchical path like `top.genblk2[3].u_cpu.genblk5.foo_q`. The naming scheme for unnamed generated blocks is defined in IEEE1800-2017 clause 27.6.

These implicit names are not intuitive for human readers, so this rule is designed to check three things:

1. The generate block uses `begin`/`end` delimiters.
2. The generate block has been given a label, e.g. `begin: mylabel`.

3. The label has an appropriate prefix, e.g. `begin: l_mylabel` starts with the string `l_`.

The prefix is useful to when reading hierarchical paths to distinguish between module/interface instances and generate blocks. For example, `top.l_cpu_array[3].u_cpu.l_debugger.foo_q` provides the reader with more useful information than `top.genblk2[3].u_cpu.genblk5.foo_q`.

See also:

- **generate_case_with_label** - Equivalent reasoning, useful companion rule.
- **generate_for_with_label** - Similar reasoning, useful companion rule.
- **prefix_instance** - Useful companion rule.

The most relevant clauses of IEEE1800-2017 are:

- 27.5 Conditional generate constructs
- 27.6 External names for unnamed generate blocks

Rule: `lowercamelcase_interface`

Hint

Begin `interface` name with lowerCamelCase.

Reason

Naming convention simplifies audit.

Pass Example (1 of 1)

```
interface fooBar;
endinterface
```

Fail Example (1 of 1)

```
interface FooBar;
endinterface
```

Explanation

There are 3 usual types of SystemVerilog file for synthesizable design code (module, interface, package) and having a simple naming convention helps distinguish them from a filesystem viewpoint. In Haskell, types/typeclasses must start with an uppercase letter, and functions/variables must start with a lowercase letter. This rule checks part of a related naming scheme where modules and interfaces should start with a lowercase letter, and packages should start with an uppercase letter.

See also:

- **lowercamelcase_module** - Suggested companion rule.
- **lowercamelcase_package** - Potential companion rule.
- **prefix_interface** - Alternative rule.
- **uppercamelcase_interface** - Mutually exclusive alternative rule.
- **uppercamelcase_module** - Potential companion rule.
- **uppercamelcase_package** - Suggested companion rule.

Rule: `lowercamelcase_module`

Hint

Begin `module` name with lowerCamelCase.

Reason

Naming convention simplifies audit.

Pass Example (1 of 1)

```
module fooBar;
endmodule
```

Fail Example (1 of 1)

```
module FooBar;
endmodule
```

Explanation

There are 3 usual types of SystemVerilog file for synthesizable design code (module, interface, package) and having a simple naming convention helps distinguish them from a filesystem viewpoint. In Haskell, types/typeclasses must start with an uppercase letter, and functions/variables must start with a lowercase letter. This rule checks part of a related naming scheme where modules and interfaces should start with a lowercase letter, and packages should start with an uppercase letter.

See also:

- **lowercamelcase_interface** - Suggested companion rule.
- **lowercamelcase_package** - Potential companion rule.
- **prefix_module** - Alternative rule.
- **uppercamelcase_interface** - Potential companion rule.
- **uppercamelcase_module** - Mutually exclusive alternative rule.
- **uppercamelcase_package** - Suggested companion rule.

Rule: `lowercamelcase_package`

Hint

Begin `package` name with lowerCamelCase.

Reason

Naming convention simplifies audit.

Pass Example (1 of 1)

```
package fooBar;
endpackage
```

Fail Example (1 of 1)

```
package FooBar;
endpackage
```

Explanation

There are 3 usual types of SystemVerilog file for synthesizable design code (module, interface, package) and having a simple naming convention helps distinguish them from a filesystem viewpoint. In Haskell, types/typeclasses must start with an uppercase letter, and functions/variables must start with a lowercase letter. This rule checks part of a related naming scheme where modules and interfaces should start with an uppercase letter, and packages should start with an lowercase letter.

See also:

- **lowercamelcase_interface** - Potential companion rule.
- **lowercamelcase_module** - Potential companion rule.
- **prefix_package** - Alternative rule.
- **uppercamelcase_interface** - Suggested companion rule.
- **uppercamelcase_module** - Suggested companion rule.
- **uppercamelcase_package** - Mutually exclusive alternative rule.

Rule: `prefix_inout`

Hint

Prefix `inout` port identifier with "b_".

Reason

Port prefixes help readers to follow signals through modules.

Pass Example (1 of 1)

```
module M
  ( inout var b_foo
  , input var logic [FOO-1:0] b_bar
  );
endmodule
```

Fail Example (1 of 3)

```
module M
  ( inout var foo // `foo` is missing prefix.
  );
endmodule
```

Fail Example (2 of 3)

```
module M
  ( inout var logic [A-1:0] bar // `bar` is missing prefix, not `A`.
  );
endmodule
```

Fail Example (3 of 3)

```
module M
  ( inout var i_foo
  , inout var bar // `bar` is missing prefix.
  );
endmodule
```

Explanation

There are 4 kinds of SystemVerilog port (`inout`, `input`, `output`, and `ref`), though `ref` is not generally used for synthesisable code. For a new reader, unfamiliar with a large module, it is useful to be able to distinguish at a glance between which signals are ports and internal ones. This is especially useful for an integrator who needs to read and understand the boundaries of many modules quickly and accurately. To use a visual analogy, prefixing port

names is like adding arrowheads to a schematic - they're not essential, but they speed up comprehension. This rule requires the prefix `b_` (configurable) on bi-directional signals, i.e, ports declared with direction `inout`, which is also the default direction.

See also:

- **prefix_input** - Suggested companion rule.
- **prefix_instance** - Suggested companion rule.
- **prefix_output** - Suggested companion rule.

Rule: `prefix_input`

Hint

Prefix `input` port identifier with "i_".

Reason

Port prefixes help readers to follow signals through modules.

Pass Example (1 of 1)

```
module M
  ( input var i_foo
  , input var logic [FOO-1:0] i_bar
  );
endmodule
```

Fail Example (1 of 1)

```
module M
  ( input var foo
  , input var logic [FOO-1:0] bar
  );
endmodule
```

Explanation

There are 4 kinds of SystemVerilog port (`inout`, `input`, `output`, and `ref`), though `ref` is not generally used for synthesisable code. For a new reader, unfamiliar with a large module, it is useful to be able to distinguish at a glance between which signals are ports and internal ones. This is especially useful for an integrator who needs to read and understand the boundaries of many modules quickly and accurately. To use a visual analogy, prefixing port names is like adding arrowheads to a schematic - they're not essential, but they speed up comprehension. This rule requires the prefix `i_` (configurable) on `input` signals.

See also:

- **prefix_inout** - Suggested companion rule.
- **prefix_instance** - Suggested companion rule.
- **prefix_output** - Suggested companion rule.

Rule: `prefix_instance`

Hint

Prefix instance identifier with "u_".

Reason

Naming convention helps investigation using hierarchical paths.

Pass Example (1 of 1)

```
module M;
  I #() u_foo (a, b, c);
endmodule
```

Fail Example (1 of 1)

```
module M;
  Foo #() foo (a, b, c);
endmodule
```

Explanation

This rule requires that instances of modules or interfaces are prefixed with `u_` (configurable) which allows readers to quickly find instances and connections of interest. Prefixing instances also allows components of a hierarchical path to be easily identified as modules/interfaces rather than generate blocks, which is especially useful when reading netlists and synthesis reports. The default value of `u_` comes from the historical use of `U` for the PCB reference designator of an inseparable assembly or integrated-circuit package, as standardized in IEEE315-1975.

See also:

- **generate_case_with_label** - Suggested companion rule.
- **generate_for_with_label** - Suggested companion rule.
- **generate_if_with_label** - Suggested companion rule.
- **prefix_inout** - Suggested companion rule.
- **prefix_input** - Suggested companion rule.
- **prefix_output** - Suggested companion rule.
- https://en.wikipedia.org/wiki/Reference_designator

Rule: `prefix_interface`

Hint

Prefix `interface` identifier with "ifc_".

Reason

Naming convention simplifies audit.

Pass Example (1 of 1)

```
interface ifc_withPrefix;
endinterface
```

Fail Example (1 of 1)

```
interface noPrefix;
endinterface
```

Explanation

There are 3 usual types of SystemVerilog file for synthesizable design code (module, interface, package) and having a simple naming convention helps distinguish them from a filesystem viewpoint. This rule requires that interface identifiers are declared with a prefix of `ifc_` (configurable) which allows a reader to easily distinguish between module and interface instances.

See also:

- **lowercamelcase_interface** - Alternative rule.
- **prefix_module** - Potential companion rule.
- **prefix_package** - Suggested companion rule.
- **uppercamelcase_interface** - Alternative rule.

Rule: `prefix_module`

Hint

Prefix `module` identifier with "mod_".

Reason

Naming convention simplifies audit.

Pass Example (1 of 1)

```
module mod_withPrefix; // Module identifier of declaration has prefix.
  I #(.A(1)) u_M (.a); // Module identifier of instance doesn't require prefix.
endmodule
```

Fail Example (1 of 1)

```
module noPrefix; // Module identifier of declaration should have prefix.
endmodule
```

Explanation

There are 3 usual types of SystemVerilog file for synthesizable design code (module, interface, package) and having a simple naming convention helps distinguish them from a filesystem viewpoint. This rule requires that module identifiers are declared with a prefix of `mod_` (configurable) which allows a reader to easily distinguish between module and interface instances.

See also:

- **lowercamelcase_module** - Alternative rule.
- **prefix_interface** - Suggested companion rule.
- **prefix_package** - Suggested companion rule.
- **uppercamelcase_module** - Alternative rule.

Rule: `prefix_output`

Hint

Prefix `output` port identifier with "o_".

Reason

Port prefixes help readers to follow signals through modules.

Pass Example (1 of 1)

```
module M
  ( output var o_foo
  , output var logic [FOO-1:0] o_bar
  );
endmodule
```

Fail Example (1 of 1)

```
module M
  ( output var foo
  , output var logic [FOO-1:0] bar
  );
endmodule
```

Explanation

There are 4 kinds of SystemVerilog port (`inout`, `input`, `output`, and `ref`), though `ref` is not generally used for synthesisable code. For a new reader, unfamiliar with a large module, it is useful to be able to distinguish at a glance between which signals are ports and internal ones. This is especially useful for an integrator who needs to read and understand the boundaries of many modules quickly and accurately. To use a visual analogy, prefixing port names is like adding arrowheads to a schematic - they're not essential, but they speed up comprehension. This rule requires the prefix `o_` (configurable) on `output` signals.

See also:

- **prefix_inout** - Suggested companion rule.
- **prefix_input** - Suggested companion rule.
- **prefix_instance** - Suggested companion rule.

Rule: `prefix_package`

Hint

Prefix `package` identifier with "pkg_".

Reason

Naming convention simplifies audit.

Pass Example (1 of 1)

```
package pkg_withPrefix;
endpackage
```

Fail Example (1 of 1)

```
package noPrefix;
endpackage
```

Explanation

There are 3 usual types of SystemVerilog file for synthesizable design code (module, interface, package) and having a simple naming convention helps distinguish them from a filesystem viewpoint. This rule requires that package identifiers are declared with a prefix of `pkg_` (configurable). When used in conjunction with a file naming scheme like "There should be one package declaration per file, and a package `pkg_foo` must be contained in a file called `pkg_foo.sv`.", this aids a reader in browsing a source directory.

See also:

- **lowercamelcase_package** - Alternative rule.
- **prefix_interface** - Suggested companion rule.
- **prefix_module** - Potential companion rule.
- **uppercamelcase_package** - Alternative rule.

Rule: `re_forbidden_assert`

Hint

Use an immediate assertion identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 3)

```
module M;
  initial begin
    Xfoo: // Identifier doesn't match default forbidden regex (X prefix).
      assert (p) else $error(); // Simple immmediate assertion statement.
  end
endmodule
```

Pass Example (2 of 3)

```
module M;
  initial begin
    Xfoo: // Identifier doesn't match default forbidden regex (X prefix).
      assert #0 (p) else $error(); // Deferred immmediate assertion statement.
  end
endmodule
```

Pass Example (3 of 3)

```
module M;
  Xfoo: // Identifier doesn't match default forbidden regex (X prefix).
    assert #0 (p) else $error(); // Deferred immmediate assertion item.
endmodule
```

Fail Example (1 of 3)

```
module M;
  initial begin
    foo: // Unconfigured forbidden regex matches (almost) anything.
      assert (p) else $error(); // Simple immmediate assertion statement.
  end
endmodule
```

Fail Example (2 of 3)

```
module M;
  initial begin
```

```
    foo: // Unconfigured forbidden regex matches (almost) anything.
      assert #0 (p) else $error(); // Deferred immmediate assertion statement.
  end
endmodule
```

Fail Example (3 of 3)

```
module M;
  foo: // Unconfigured forbidden regex matches (almost) anything.
    assert #0 (p) else $error(); // Deferred immmediate assertion item.
endmodule
```

Explanation

Immediate assertions, including deferred immediate assertions, must not have identifiers matching the regex configured via the `re_forbidden_assert` option.

See also:

- **re_required_assert**

Rule: `re_forbidden_assert_property`

Hint

Use a concurrent assertion identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 2)

```
module M;
  Xfoo: // Identifier doesn't match default forbidden regex (X prefix).
    assert property (@(posedge c) p); // Concurrent assertion.
endmodule
```

Pass Example (2 of 2)

```
module M;
  initial begin
    Xfoo: // Identifier doesn't match default forbidden regex (X prefix).
      assert property (@(posedge c) p); // Concurrent assertion.
  end
endmodule
```

Fail Example (1 of 2)

```
module M;
  foo: // Unconfigured forbidden regex matches (almost) anything.
    assert property (@(posedge c) p); // Concurrent assertion.
endmodule
```

Fail Example (2 of 2)

```
module M;
  initial begin
    foo: // Unconfigured forbidden regex matches (almost) anything.
      assert property (@(posedge c) p); // Concurrent assertion.
  end
endmodule
```

Explanation

Concurrent assertions must not have identifiers matching the regex configured via the `re_forbidden_assert_property` option.

See also:

- **re__required__assert__property**

Rule: `re_forbidden_checker`

Hint

Use a checker identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
checker Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endchecker
```

Fail Example (1 of 1)

```
checker foo; // Unconfigured forbidden regex matches (almost) anything.
endchecker
```

Explanation

Checkers must not have identifiers matching the regex configured via the `re_forbidden_checker` option.

See also:

- **re__required__checker**

Rule: `re_forbidden_class`

Hint

Use a class identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
class Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endclass
```

Fail Example (1 of 1)

```
class foo; // Unconfigured forbidden regex matches (almost) anything.
endclass
```

Explanation

Classes must not have identifiers matching the regex configured via the `re_forbidden_class` option.

See also:

- **re_required_class**

Rule: `re_forbidden_function`

Hint

Use a function identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
package P;
  function Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
  endfunction
endpackage
```

Fail Example (1 of 1)

```
package P;
  function foo; // Unconfigured forbidden regex matches (almost) anything.
  endfunction
endpackage
```

Explanation

Functions must not have identifiers matching the regex configured via the `re_forbidden_function` option.

See also:

- **re__required__function**
- **function__same__as__system__function**

Rule: `re_forbidden_generateblock`

Hint

Use a generate block identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 3)

```
module M;
  if (0) begin: Xfoo // Identifier doesn't match default forbidden regex (X prefix).
    assign a = 0;
  end: Xfoo
  else begin: Xbar // Identifier doesn't match default forbidden regex (X prefix).
    assign a = 1;
  end: Xbar
endmodule
```

Pass Example (2 of 3)

```
module M;
  // Identifier doesn't match default forbidden regex (X prefix).
  for (genvar i=0; i < 5; i++) begin: Xfoo
    assign b[i] = 0;
  end: Xfoo
endmodule
```

Pass Example (3 of 3)

```
module M;
  case (0)
    0: begin: Xfoo // Identifier doesn't match default forbidden regex (X prefix).
      assign c = 0;
    end: Xfoo
    1: begin: Xbar // Identifier doesn't match default forbidden regex (X prefix).
      assign c = 1;
    end: Xbar
    default: begin: Xbaz // Identifier doesn't match default forbidden regex (X prefix).
      assign c = 2;
    end: Xbaz
  endcase
endmodule
```

Fail Example (1 of 3)

```
module M;
  if (0) begin: foo // Unconfigured forbidden regex matches (almost) anything.
    assign a = 0;
  end: foo
  else begin: bar // Unconfigured forbidden regex matches (almost) anything.
    assign a = 1;
  end: bar
endmodule
```

Fail Example (2 of 3)

```
module M;
  // Unconfigured forbidden regex matches (almost) anything.
  for (genvar i=0; i < 5; i++) begin: foo
    assign b[i] = 0;
  end: foo
endmodule
```

Fail Example (3 of 3)

```
module M;
  case (0)
    0: begin: foo // Unconfigured forbidden regex matches (almost) anything.
      assign c = 0;
    end: foo
    1: begin: bar // Unconfigured forbidden regex matches (almost) anything.
      assign c = 1;
    end: bar
    default: begin: baz // Unconfigured forbidden regex matches (almost) anything.
      assign c = 2;
    end: baz
  endcase
endmodule
```

Explanation

Generate blocks must not have identifiers matching the regex configured via the `re_forbidden_generateblock` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_required_generateblock**

- **generate_case_with_label**
- **generate_for_with_label**
- **generate_if_with_label**

Rule: `re_forbidden_genvar`

Hint

Use a genvar identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 2)

```
module M;
  genvar Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endmodule
```

Pass Example (2 of 2)

```
module M;
  // Identifier doesn't match default forbidden regex (X prefix).
  for (genvar Xbar=0; Xbar < 5; Xbar++) begin
  end
endmodule
```

Fail Example (1 of 2)

```
module M;
  genvar foo; // Unconfigured forbidden regex matches (almost) anything.
endmodule
```

Fail Example (2 of 2)

```
module M;
  // Unconfigured forbidden regex matches (almost) anything.
  for (genvar bar=0; bar < 5; bar++) begin
  end
endmodule
```

Explanation

Genvars must not have identifiers matching the regex configured via the `re_forbidden_genvar` option.

See also:

- **re__required__genvar**

Rule: `re_forbidden_instance`

Hint

Use an instance identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M;
  A #(
  ) Xfoo (); // Identifier doesn't match default forbidden regex (X prefix).
endmodule
```

Fail Example (1 of 1)

```
module M;
  A #(
  ) foo (); // Unconfigured forbidden regex matches (almost) anything.
endmodule
```

Explanation

Instances must not have identifiers matching the regex configured via the `re_forbidden_instance` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re__required__package**
- **prefix__instance**

Rule: `re_forbidden_interface`

Hint

Use a interface identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
interface Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endinterface
```

Fail Example (1 of 1)

```
interface foo; // Unconfigured forbidden regex matches (almost) anything.
endinterface
```

Explanation

Interfaces must not have identifiers matching the regex configured via the `re_forbidden_interface` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_required_interface**
- **prefix_interface**
- **uppercamelcase_interface**
- **lowercamelcase_interface**

Rule: `re_forbidden_localparam`

Hint

Use a localparam identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
package P;
  localparam Xfoo = 0; // Identifier doesn't match default forbidden regex (X prefix).
endpackage
```

Fail Example (1 of 1)

```
package P;
  localparam foo = 0; // Unconfigured forbidden regex matches (almost) anything.
endpackage
```

Explanation

Local parameters must not have identifiers matching the regex configured via the `re_forbidden_localparam` option.

See also:

- **re_required_localparam**
- **localparam_explicit_type**
- **localparam_type_twostate**
- **parameter_explicit_type**
- **parameter_in_package**
- **parameter_type_twostate**

Rule: `re_forbidden_modport`

Hint

Use a modport identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
interface I;
  modport Xfoo // Identifier doesn't match default forbidden regex (X prefix).
  ( input i
  );
endinterface
```

Fail Example (1 of 1)

```
interface I;
  modport foo // Unconfigured forbidden regex matches (almost) anything.
  ( input i
  );
endinterface
```

Explanation

Modports must not have identifiers matching the regex configured via the `re_forbidden_modport` option.

See also:

- **re__required__modport**
- **interface__port__with__modport**

Rule: `re_forbidden_module_ansi`

Hint

Use a module identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endmodule
```

Fail Example (1 of 1)

```
module foo; // Unconfigured forbidden regex matches (almost) anything.
endmodule
```

Explanation

Modules declared with an ANSI header must not have identifiers matching the regex configured via the `re_forbidden_module_ansi` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re__required__module__ansi**
- **re__forbidden__module__nonansi**
- **re__required__module__nonansi**
- **prefix__module**
- **uppercamelcase__module**
- **lowercamelcase__module**
- **non__ansi__module**

Rule: `re_forbidden_module_nonansi`

Hint

Use a module identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module Xfoo // Identifier doesn't match default forbidden regex (X prefix).
  ( a
  );
  input a;
endmodule
```

Fail Example (1 of 1)

```
module foo // Unconfigured forbidden regex matches (almost) anything.
  ( a
  );
  input a;
endmodule
```

Explanation

Modules declared with a non-ANSI header must not have identifiers matching the regex configured via the `re_forbidden_module_nonansi` option. Non-ANSI modules are commonly used where compatability with classic Verilog (IEEE1364-1995) is required, such as low-level cells and macros.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re__required__module__nonansi**
- **re__forbidden__module__ansi**
- **re__required__module__ansi**
- **prefix__module**
- **uppercamelcase__module**
- **lowercamelcase__module**
- **non__ansi__module**

Rule: `re_forbidden_package`

Hint

Use a package identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
package Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endpackage
```

Fail Example (1 of 1)

```
package foo; // Unconfigured forbidden regex matches (almost) anything.
endpackage
```

Explanation

Packages must not have identifiers matching the regex configured via the `re_forbidden_package` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_required_package**
- **prefix_package**
- **uppercamelcase_package**
- **lowercamelcase_package**

Rule: `re_forbidden_parameter`

Hint

Use a parameter identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M
  #( Xfoo // Identifier doesn't match default forbidden regex (X prefix).
  ) ();
endmodule
```

Fail Example (1 of 1)

```
module M
  #( foo // Unconfigured forbidden regex matches (almost) anything.
  ) ();
endmodule
```

Explanation

Parameters must not have identifiers matching the regex configured via the `re_forbidden_parameter` option.

See also:

- **re__required__parameter**
- **localparam__explicit__type**
- **localparam__type__twostate**
- **parameter__explicit__type**
- **parameter__in__package**
- **parameter__type__twostate**

Rule: `re_forbidden_port_inout`

Hint

Use a port identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 2)

```
module M
  ( inout Xfoo // Identifier doesn't match default forbidden regex (X prefix).
  );
endmodule
```

Pass Example (2 of 2)

```
module M_nonansi
  ( Xfoo
  );
  inout Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endmodule
```

Fail Example (1 of 2)

```
module M
  ( inout foo // Unconfigured forbidden regex matches (almost) anything.
  );
endmodule
```

Fail Example (2 of 2)

```
module M_nonansi
  ( foo
  );
  inout foo; // Unconfigured forbidden regex matches (almost) anything.
endmodule
```

Explanation

Bidirectional ports must not have identifiers matching the regex configured via the `re_forbidden_port_inout` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re__required__port__inout**
- **prefix__inout**

Rule: `re_forbidden_port_input`

Hint

Use a port identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 2)

```
module M
  ( input Xfoo // Identifier doesn't match default forbidden regex (X prefix).
  );
endmodule
```

Pass Example (2 of 2)

```
module M_nonansi
  ( Xfoo
  );
  input Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endmodule
```

Fail Example (1 of 2)

```
module M
  ( input foo // Unconfigured forbidden regex matches (almost) anything.
  );
endmodule
```

Fail Example (2 of 2)

```
module M_nonansi
  ( foo
  );
  input foo; // Unconfigured forbidden regex matches (almost) anything.
endmodule
```

Explanation

Input ports must not have identifiers matching the regex configured via the `re_forbidden_port_input` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re__required__port__input**
- **prefix__input**

Rule: `re_forbidden_port_interface`

Hint

Use a port identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 2)

```
module M
  ( I Xfoo // Identifier doesn't match default forbidden regex (X prefix).
  );
endmodule
```

Pass Example (2 of 2)

```
module M_nonansi
  ( Xfoo
  );
  I.i Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endmodule
```

Fail Example (1 of 2)

```
module M
  ( I.i foo // Unconfigured forbidden regex matches (almost) anything.
  );
endmodule
```

Fail Example (2 of 2)

```
module M_nonansi
  ( foo
  );
  I.i foo; // Unconfigured forbidden regex matches (almost) anything.
endmodule
```

Explanation

Interface ports must not have identifiers matching the regex configured via the `re_forbidden_port_interface` option.

See also:

- **re__required__port__interface**

Rule: `re_forbidden_port_output`

Hint

Use a port identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 2)

```
module M
  ( output Xfoo // Identifier doesn't match default forbidden regex (X prefix).
  );
endmodule
```

Pass Example (2 of 2)

```
module M_nonansi
  ( Xfoo
  );
  output Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endmodule
```

Fail Example (1 of 2)

```
module M
  ( output foo // Unconfigured forbidden regex matches (almost) anything.
  );
endmodule
```

Fail Example (2 of 2)

```
module M_nonansi
  ( foo
  );
  output foo; // Unconfigured forbidden regex matches (almost) anything.
endmodule
```

Explanation

Output ports must not have identifiers matching the regex configured via the `re_forbidden_port_output` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re__required__port__output**
- **prefix__output**

Rule: `re_forbidden_port_ref`

Hint

Use a port identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M
  ( ref Xfoo // Identifier doesn't match default forbidden regex (X prefix).
  );
endmodule
```

Fail Example (1 of 1)

```
module M
  ( ref foo // Unconfigured forbidden regex matches (almost) anything.
  );
endmodule
```

Explanation

Reference ports must not have identifiers matching the regex configured via the `re_forbidden_port_ref` option.

See also:

- **re__required__port__ref**

Rule: `re_forbidden_program`

Hint

Use a program identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
program Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endprogram
```

Fail Example (1 of 1)

```
program foo; // Unconfigured forbidden regex matches (almost) anything.
endprogram
```

Explanation

Programs must not have identifiers matching the regex configured via the `re_forbidden_program` option.

See also:

- **re__required__program**

Rule: `re_forbidden_property`

Hint

Use a property identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M;
  property Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
    @(posedge c) p; // Concurrent assertion.
  endproperty
endmodule
```

Fail Example (1 of 1)

```
module M;
  property foo; // Unconfigured forbidden regex matches (almost) anything.
    @(posedge c) p; // Concurrent assertion.
  endproperty
endmodule
```

Explanation

Properties must not have identifiers matching the regex configured via the `re_forbidden_property` option.

See also:

- **re_required_property**

Rule: `re_forbidden_sequence`

Hint

Use a sequence identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M;
  sequence Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
    @(posedge c) a ##1 b
  endsequence
endmodule
```

Fail Example (1 of 1)

```
module M;
  sequence foo; // Unconfigured forbidden regex matches (almost) anything.
    @(posedge c) a ##1 b
  endsequence
endmodule
```

Explanation

Sequences must not have identifiers matching the regex configured via the `re_forbidden_sequence` option.

See also:

- **re__required__sequence**

Rule: `re_forbidden_task`

Hint

Use a task identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M;
  task Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
  endtask
endmodule
```

Fail Example (1 of 1)

```
module M;
  task foo; // Unconfigured forbidden regex matches (almost) anything.
  endtask
endmodule
```

Explanation

Tasks must not have identifiers matching the regex configured via the `re_forbidden_task` option.

See also:

- **re__required__task**

Rule: `re_forbidden_var_class`

Hint

Use a class-scoped variable identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
class C;
  int Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
endclass
```

Fail Example (1 of 1)

```
class C;
  int foo; // Unconfigured forbidden regex matches (almost) anything.
endclass
```

Explanation

Class-scoped variables must not have identifiers matching the regex configured via the `re_forbidden_var_class` option.

See also:

- **re__required__var__class**

Rule: `re_forbidden_var_classmethod`

Hint

Use a method-scoped variable identifier not matching regex `^[^X](UNCONFIGURED|.*)$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
class C;
  function F;
    int Xfoo; // Identifier doesn't match default forbidden regex (X prefix).
  endfunction
endclass
```

Fail Example (1 of 1)

```
class C;
  function F;
    int foo; // Unconfigured forbidden regex matches (almost) anything.
  endfunction
endclass
```

Explanation

Method-scoped variables must not have identifiers matching the regex configured via the `re_forbidden_var_classmethod` option.

See also:

- **re__required__var__classmethod**
- **re__required__var__class**
- **re__forbidden__var__class**

Rule: `re_required_assert`

Hint

Use an immediate assertion identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 3)

```
module M;
  initial begin
    mn3: // Identifier matches default required regex (lowercase).
      assert (p) else $error(); // Simple immmediate assertion statement.
  end
endmodule
```

Pass Example (2 of 3)

```
module M;
  initial begin
    mn3: // Identifier matches default required regex (lowercase).
      assert #0 (p) else $error(); // Deferred immmediate assertion statement.
  end
endmodule
```

Pass Example (3 of 3)

```
module M;
  mn3: // Identifier matches default required regex (lowercase).
    assert #0 (p) else $error(); // Deferred immmediate assertion item.
endmodule
```

Fail Example (1 of 3)

```
module M;
  initial begin
    Mn3: // Identifier doesn't match default required regex (lowercase).
      assert (p) else $error(); // Simple immmediate assertion statement.
  end
endmodule
```

Fail Example (2 of 3)

```
module M;
  initial begin
```

```
    Mn3: // Identifier doesn't match default required regex (lowercase).
      assert #0 (p) else $error(); // Deferred immmediate assertion statement.
  end
endmodule
```

Fail Example (3 of 3)

```
module M;
  Mn3: // Identifier doesn't match default required regex (lowercase).
    assert #0 (p) else $error(); // Deferred immmediate assertion item.
endmodule
```

Explanation

Immediate assertions, including deferred immediate assertions, must have identifiers matching the regex configured via the `re_required_assert` option.

See also:

- **re_forbidden_assert**

Rule: `re_required_assert_property`

Hint

Use a concurrent assertion identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 2)

```
module M;
  mn3: // Identifier matches default required regex (lowercase).
    assert property (@(posedge c) p); // Concurrent assertion.
endmodule
```

Pass Example (2 of 2)

```
module M;
  initial begin
    mn3: // Identifier matches default required regex (lowercase).
      assert property (@(posedge c) p); // Concurrent assertion.
  end
endmodule
```

Fail Example (1 of 2)

```
module M;
  Mn3: // Identifier doesn't match default required regex (lowercase).
    assert property (@(posedge c) p); // Concurrent assertion.
endmodule
```

Fail Example (2 of 2)

```
module M;
  initial begin
    Mn3: // Identifier doesn't match default required regex (lowercase).
      assert property (@(posedge c) p); // Concurrent assertion.
  end
endmodule
```

Explanation

Concurrent assertions must have identifiers matching the regex configured via the `re_required_assert_property` option.

See also:

- **re__forbidden__assert__property**

Rule: `re_required_checker`

Hint

Use a checker identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
checker mn3; // Identifier matches default required regex (lowercase).
endchecker
```

Fail Example (1 of 1)

```
checker Mn3; // Identifier doesn't match default required regex (lowercase).
endchecker
```

Explanation

Checkers must have identifiers matching the regex configured via the `re_required_checker` option.

See also:

- **re_forbidden_checker**

Rule: `re_required_class`

Hint

Use a class identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
class mn3; // Identifier matches default required regex (lowercase).
endclass
```

Fail Example (1 of 1)

```
class Mn3; // Identifier doesn't match default required regex (lowercase).
endclass
```

Explanation

Classes must have identifiers matching the regex configured via the `re_required_class` option.

See also:

- **re_forbidden_class**

Rule: `re_required_function`

Hint

Use a function identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
package P;
  function mn3; // Identifier matches default required regex (lowercase).
  endfunction
endpackage
```

Fail Example (1 of 1)

```
package P;
  function Mn3; // Identifier doesn't match default required regex (lowercase).
  endfunction
endpackage
```

Explanation

Functions must have identifiers matching the regex configured via the `re_required_function` option.

See also:

- **re_forbidden_function**
- **function_same_as_system_function**

Rule: `re_required_generateblock`

Hint

Use a generate block identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M;
  if (0) begin: mn3 // Identifier matches default required regex (lowercase).
    assign a = 0;
  end: mn3
  else begin: mn4 // Identifier matches default required regex (lowercase).
    assign a = 1;
  end: mn4

  // Identifier matches default required regex (lowercase).
  for (genvar i=0; i < 5; i++) begin: mn5
    assign b[i] = 0;
  end: mn5

  case (0)
    0: begin: mn6 // Identifier matches default required regex (lowercase).
      assign c = 0;
    end: mn6
    1: begin: mn7 // Identifier matches default required regex (lowercase).
      assign c = 1;
    end: mn7
    default: begin: mn8 // Identifier matches default required regex (lowercase).
      assign c = 2;
    end: mn8
  endcase
endmodule
```

Fail Example (1 of 1)

```
module M;
  if (0) begin: Mn3 // Identifier doesn't match default required regex (lowercase).
    assign a = 0;
  end: Mn3
  else begin: Mn4 // Identifier doesn't match default required regex (lowercase).
    assign a = 1;
  end: Mn4
```

```
  // Identifier doesn't match default required regex (lowercase).
  for (genvar i=0; i < 5; i++) begin: Mn5
    assign b[i] = 0;
  end: Mn5

  case (0)
    0: begin: Mn6 // Identifier doesn't match default required regex (lowercase).
      assign c = 0;
    end: Mn6
    1: begin: Mn7 // Identifier doesn't match default required regex (lowercase).
      assign c = 1;
    end: Mn7
    default: begin: Mn8 // Identifier doesn't match default required regex (lowercase).
      assign c = 2;
    end: Mn8
  endcase
endmodule
```

Explanation

Generate blocks must have identifiers matching the regex configured via the `re_required_generateblock` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_forbidden_generateblock**
- **generate_case_with_label**
- **generate_for_with_label**
- **generate_if_with_label**

Rule: `re_required_genvar`

Hint

Use a genvar identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M;
  genvar mn3; // Identifier matches default required regex (lowercase).

  // Identifier matches default required regex (lowercase).
  for (genvar mn4=0; mn4 < 5; mn4++) begin
  end
endmodule
```

Fail Example (1 of 1)

```
module M;
  genvar Mn3; // Identifier doesn't match default required regex (lowercase).

  // Identifier doesn't match default required regex (lowercase).
  for (genvar Mn4=0; Mn4 < 5; Mn4++) begin
  end
endmodule
```

Explanation

Genvars must have identifiers matching the regex configured via the `re_required_genvar` option.

See also:

- **re_forbidden_genvar**

Rule: `re_required_instance`

Hint

Use an instance identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M;
  A #(
  ) mn3 (); // Identifier matches default required regex (lowercase).
endmodule
```

Fail Example (1 of 1)

```
module M;
  A #(
  ) Mn3 (); // Identifier doesn't match default required regex (lowercase).
endmodule
```

Explanation

Instances must have identifiers matching the regex configured via the `re_required_instance` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re__forbidden__instance**
- **prefix__instance**

Rule: `re_required_interface`

Hint

Use a interface identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
interface mn3; // Identifier matches default required regex (lowercase).
endinterface
```

Fail Example (1 of 1)

```
interface Mn3; // Identifier doesn't match default required regex (lowercase).
endinterface
```

Explanation

Interfaces must have identifiers matching the regex configured via the `re_required_interface` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_forbidden_interface**
- **prefix_interface**
- **uppercamelcase_interface**
- **lowercamelcase_interface**

Rule: `re_required_localparam`

Hint

Use a localparam identifier matching regex `^[A-Z]+[A-Z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
package P;
  localparam MN3 = 0; // Identifier matches default required regex (uppercase).
endpackage
```

Fail Example (1 of 1)

```
package P;
  localparam Mn3 = 0; // Identifier doesn't match default required regex (uppercase).
endpackage
```

Explanation

Local parameters must have identifiers matching the regex configured via the `re_required_localparam` option.

See also:

- **re_forbidden_localparam**
- **localparam_explicit_type**
- **localparam_type_twostate**
- **parameter_explicit_type**
- **parameter_in_package**
- **parameter_type_twostate**

Rule: `re_required_modport`

Hint

Use a modport identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
interface I;
  modport mn3 // Identifier matches default required regex (lowercase).
  ( input i
  );
endinterface
```

Fail Example (1 of 1)

```
interface I;
  modport Mn3 // Identifier doesn't match default required regex (lowercase).
  ( input i
  );
endinterface
```

Explanation

Modports must have identifiers matching the regex configured via the `re_required_modport` option.

See also:

- **re_forbidden_modport**
- **interface_port_with_modport**

Rule: `re_required_module_ansi`

Hint

Use a module identifier matching regex `^[a-z]+[a-zA-Z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module mN3; // Identifier matches default required regex (mixed-case).
endmodule
```

Fail Example (1 of 1)

```
module Mn3; // Identifier doesn't match default required regex (mixed-case).
endmodule
```

Explanation

Modules declared with an ANSI header must have identifiers matching the regex configured via the `re_required_module_ansi` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_forbidden_module_ansi**
- **re_forbidden_module_nonansi**
- **re_required_module_nonansi**
- **prefix_module**
- **uppercamelcase_module**
- **lowercamelcase_module**
- **non_ansi_module**

Rule: `re_required_module_nonansi`

Hint

Use a module identifier matching regex `^[A-Z]+[A-Z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module MN3 // Identifier matches default required regex (uppercase).
  ( a
  );
  input a;
endmodule
```

Fail Example (1 of 1)

```
module mn3 // Identifier doesn't match default required regex (uppercase).
  ( a
  );
  input a;
endmodule
```

Explanation

Modules declared with a non-ANSI header must have identifiers matching the regex configured via the `re_required_module_nonansi` option. Non-ANSI modules are commonly used where compatability with classic Verilog (IEEE1364-1995) is required, such as low-level cells and macros.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_forbidden_module_nonansi**
- **re_forbidden_module_ansi**
- **re_required_module_ansi**
- **prefix_module**
- **uppercamelcase_module**
- **lowercamelcase_module**
- **non_ansi_module**

Rule: `re_required_package`

Hint

Use a package identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
package mn3; // Identifier matches default required regex (lowercase).
endpackage
```

Fail Example (1 of 1)

```
package Mn3; // Identifier doesn't match default required regex (lowercase).
endpackage
```

Explanation

Packages must have identifiers matching the regex configured via the `re_required_package` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_forbidden_package**
- **prefix_package**
- **uppercamelcase_package**
- **lowercamelcase_package**

Rule: `re_required_parameter`

Hint

Use a parameter identifier matching regex `^[A-Z]+[A-Z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M
  #( MN3 // Identifier matches default required regex (uppercase).
  ) ();
endmodule
```

Fail Example (1 of 1)

```
module M
  #( Mn3 // Identifier doesn't match default required regex (uppercase).
  ) ();
endmodule
```

Explanation

Parameters must have identifiers matching the regex configured via the `re_required_parameter` option.

See also:

- **re_forbidden_parameter**
- **localparam_explicit_type**
- **localparam_type_twostate**
- **parameter_explicit_type**
- **parameter_in_package**
- **parameter_type_twostate**

Rule: `re_required_port_inout`

Hint

Use a port identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M
  ( inout mn3 // Identifier matches default required regex (lowercase).
  );
endmodule

module M_nonansi
  ( mn3
  );
  inout mn3; // Identifier matches default required regex (lowercase).
endmodule
```

Fail Example (1 of 1)

```
module M
  ( inout Mn3 // Identifier doesn't match default required regex (lowercase).
  );
endmodule

module M_nonansi
  ( Mn3
  );
  inout Mn3; // Identifier doesn't match default required regex (lowercase).
endmodule
```

Explanation

Bidirectional ports must have identifiers matching the regex configured via the `re_required_port_inout` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_forbidden_inout**
- **prefix_inout**

Rule: `re_required_port_input`

Hint

Use a port identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M
  ( input mn3 // Identifier matches default required regex (lowercase).
  );
endmodule

module M_nonansi
  ( mn3
  );
  input mn3; // Identifier matches default required regex (lowercase).
endmodule
```

Fail Example (1 of 1)

```
module M
  ( input Mn3 // Identifier doesn't match default required regex (lowercase).
  );
endmodule

module M_nonansi
  ( Mn3
  );
  input Mn3; // Identifier doesn't match default required regex (lowercase).
endmodule
```

Explanation

Input ports must have identifiers matching the regex configured via the `re_required_port_input` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_forbidden_input**
- **prefix_input**

Rule: `re_required_port_interface`

Hint

Use a port identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M
  ( I.i mn3 // Identifier matches default required regex (lowercase).
  );
endmodule

module M_nonansi
  ( mn3
  );
  I.i mn3; // Identifier matches default required regex (lowercase).
endmodule
```

Fail Example (1 of 1)

```
module M
  ( I.i Mn3 // Identifier doesn't match default required regex (lowercase).
  );
endmodule

module M_nonansi
  ( Mn3
  );
  I.i Mn3; // Identifier doesn't match default required regex (lowercase).
endmodule
```

Explanation

Interface ports must have identifiers matching the regex configured via the `re_required_port_interface` option.

See also:

- **re__forbidden__interface**

Rule: `re_required_port_output`

Hint

Use a port identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M
  ( output mn3 // Identifier matches default required regex (lowercase).
  );
endmodule

module M_nonansi
  ( mn3
  );
  output mn3; // Identifier matches default required regex (lowercase).
endmodule
```

Fail Example (1 of 1)

```
module M
  ( output Mn3 // Identifier doesn't match default required regex (lowercase).
  );
endmodule

module M_nonansi
  ( Mn3
  );
  output Mn3; // Identifier doesn't match default required regex (lowercase).
endmodule
```

Explanation

Output ports must have identifiers matching the regex configured via the `re_required_port_output` option.

NOTE: For performance reasons, particularly within text-editor integrations (i.e. svls), the `re_(required|forbidden)_` should only be used where the simpler naming rules are not sufficient.

See also:

- **re_forbidden_output**
- **prefix_output**

Rule: `re_required_port_ref`

Hint

Use a port identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M
  ( ref mn3 // Identifier matches default required regex (lowercase).
  );
endmodule

module M_nonansi
  ( mn3
  );
  ref var mn3; // Identifier matches default required regex (lowercase).
endmodule
```

Fail Example (1 of 1)

```
module M
  ( ref Mn3 // Identifier doesn't match default required regex (lowercase).
  );
endmodule

module M_nonansi
  ( Mn3
  );
  ref var Mn3; // Identifier doesn't match default required regex (lowercase).
endmodule
```

Explanation

Reference ports must have identifiers matching the regex configured via the `re_required_port_ref` option.

See also:

- **re__forbidden__ref**

Rule: `re_required_program`

Hint

Use a program identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
program mn3; // Identifier matches default required regex (lowercase).
endprogram
```

Fail Example (1 of 1)

```
program Mn3; // Identifier doesn't match default required regex (lowercase).
endprogram
```

Explanation

Programs must have identifiers matching the regex configured via the `re_required_program` option.

See also:

- **re_forbidden_program**

Rule: `re_required_property`

Hint

Use a property identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M;
  property mn3; // Identifier matches default required regex (lowercase).
    @(posedge c) p; // Concurrent assertion.
  endproperty
endmodule
```

Fail Example (1 of 1)

```
module M;
  property Mn3; // Identifier doesn't match default required regex (lowercase).
    @(posedge c) p; // Concurrent assertion.
  endproperty
endmodule
```

Explanation

Properties must have identifiers matching the regex configured via the `re_required_property` option.

See also:

- **re_forbidden_property**

Rule: `re_required_sequence`

Hint

Use a sequence identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M;
  sequence mn3; // Identifier matches default required regex (lowercase).
    @(posedge c) a ##1 b
  endsequence
endmodule
```

Fail Example (1 of 1)

```
module M;
  sequence Mn3; // Identifier doesn't match default required regex (lowercase).
    @(posedge c) a ##1 b
  endsequence
endmodule
```

Explanation

Sequences must have identifiers matching the regex configured via the `re_required_sequence` option.

See also:

- **re_forbidden_sequence**

Rule: `re_required_task`

Hint

Use a task identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
module M;
  task mn3; // Identifier matches default required regex (lowercase).
  endtask
endmodule
```

Fail Example (1 of 1)

```
module M;
  task Mn3; // Identifier doesn't match default required regex (lowercase).
  endtask
endmodule
```

Explanation

Tasks must have identifiers matching the regex configured via the `re_required_task` option.

See also:

- **re_forbidden_task**

Rule: `re_required_var_class`

Hint

Use a class-scoped variable identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
class C;
  int mn3; // Identifier matches default required regex (lowercase).
endclass
```

Fail Example (1 of 1)

```
class C;
  int Mn3; // Identifier doesn't match default required regex (lowercase).
endclass
```

Explanation

Class-scoped variables must have identifiers matching the regex configured via the `re_required_var_class` option.

See also:

- **re_forbidden_var_class**

Rule: `re_required_var_classmethod`

Hint

Use a method-scoped variable identifier matching regex `^[a-z]+[a-z0-9_]*$`.

Reason

Identifiers must conform to the naming scheme.

Pass Example (1 of 1)

```
class C;
  function F;
    int mn3; // Identifier matches default required regex (lowercase).
  endfunction
endclass
```

Fail Example (1 of 1)

```
class C;
  function F;
    int Mn3; // Identifier doesn't match default required regex (lowercase).
  endfunction
endclass
```

Explanation

Method-scoped variables must have identifiers matching the regex configured via the `re_required_var_classmethod` option.

See also:

- **re__forbidden__var__classmethod**
- **re__forbidden__var__class**
- **re__required__var__class**

Rule: `uppercamelcase_interface`

Hint

Begin `interface` name with UpperCamelCase.

Reason

Naming convention simplifies audit.

Pass Example (1 of 1)

```
interface FooBar;
endinterface
```

Fail Example (1 of 1)

```
interface fooBar;
endinterface
```

Explanation

There are 3 usual types of SystemVerilog file for synthesizable design code (module, interface, package) and having a simple naming convention helps distinguish them from a filesystem viewpoint. In Haskell, types/typeclasses must start with an uppercase letter, and functions/variables must start with a lowercase letter. This rule checks part of a related naming scheme where modules and interfaces should start with an uppercase letter, and packages should start with an lowercase letter.

See also:

- **lowercamelcase_interface** - Mutually exclusive alternative rule.
- **lowercamelcase_module** - Potential companion rule.
- **lowercamelcase_package** - Suggested companion rule.
- **prefix_interface** - Alternative rule.
- **uppercamelcase_module** - Suggested companion rule.
- **uppercamelcase_package** - Potential companion rule.

Rule: `uppercamelcase_module`

Hint

Begin `module` name with UpperCamelCase.

Reason

Naming convention simplifies audit.

Pass Example (1 of 1)

```
module FooBar;
endmodule
```

Fail Example (1 of 1)

```
module fooBar;
endmodule
```

Explanation

There are 3 usual types of SystemVerilog file for synthesizable design code (module, interface, package) and having a simple naming convention helps distinguish them from a filesystem viewpoint. In Haskell, types/typeclasses must start with an uppercase letter, and functions/variables must start with a lowercase letter. This rule checks part of a related naming scheme where modules and interfaces should start with an uppercase letter, and packages should start with an lowercase letter.

See also:

- **lowercamelcase_interface** - Potential companion rule.
- **lowercamelcase_module** - Mutually exclusive alternative rule.
- **lowercamelcase_package** - Suggested companion rule.
- **prefix_module** - Alternative rule.
- **uppercamelcase_interface** - Suggested companion rule.
- **uppercamelcase_package** - Potential companion rule.

Rule: `uppercamelcase_package`

Hint

Begin `package` name with UpperCamelCase.

Reason

Naming convention simplifies audit.

Pass Example (1 of 1)

```
package FooBar;
endpackage
```

Fail Example (1 of 1)

```
package fooBar;
endpackage
```

Explanation

There are 3 usual types of SystemVerilog file for synthesizable design code (module, interface, package) and having a simple naming convention helps distinguish them from a filesystem viewpoint. In Haskell, types/typeclasses must start with an uppercase letter, and functions/variables must start with a lowercase letter. This rule checks part of a related naming scheme where modules and interfaces should start with a lowercase letter, and packages should start with an uppercase letter.

See also:

- **lowercamelcase_interface** - Suggested companion rule.
- **lowercamelcase_module** - Suggested companion rule.
- **lowercamelcase_package** - Mutually exclusive alternative rule.
- **prefix_package** - Alternative rule.
- **uppercamelcase_interface** - Potential companion rule.
- **uppercamelcase_module** - Potential companion rule.

Style/Whitespace Convention Rules

Most rules for checking style/whitespace are named with the prefix `style_`, but `tab_character` is also in this class. These rules do not reference any clause in the LRM (IEEE1800-2017).

Rule: `style_commaleading`

Hint

Follow each comma with a single space (comma-leading format).

Reason

Consistent style enhances readability.

Pass Example (1 of 1)

```
module M
  #(bit FOO = 1 // comment
  , int BAR = 2 /* comment */
  , bit [31:0] BAZ = 2
  )
  ( input  var logic i_abc // comment
  , output var logic o_ghi /* comment */
  );

  assign {foo, bar} =
    { i_abc
    , 12'h345
    , b_def     // comment
    , 16'h3456  /* comment */
    };

  assign singleline2D = {{foo, bar}, {foo, bar}, {foo, bar}};

  function F
    ( input a
    , input b
    );
  endfunction
endmodule
```

Fail Example (1 of 1)

```
module M
  #( bit FOO = 1 // Space after `#(` causes misalignment.
  , int BAR = 2
  ,  bit [31:0] BAZ = 2 // Too many spaces after comma.
  )
  (input  var logic i_abc // Missing space after `(` causes misalignment.
  ,output var logic o_ghi // Missing space after comma.
  );
```

```
  assign {foo, bar} = { // One-line style is okay.
      i_abc
    ,12'h345 // Missing space.
    ,  b_def // Too many spaces after comma.
    };

  function foo
  (input a // Missing space after `(` causes misalignment.
  ,  input b // Too many spaces after comma.
  );
  endfunction

endmodule
```

Explanation

This rule is intended to enforce consistent formatting of comma-separated lists such as parameter/signal port declarations, concatenations, assignment patterns, and function arguments. The rule is very simple: Each comma must be followed by exactly 1 space.

Comma-leading style is seen extensively in other languages, e.g. Haskell, and lends itself well to SystemVerilog, as seen in the following examples.

```
/* Module declaration without parameter ports.
*/
module Mod_A
  ( input  var logic i_abc // comment
  , inout  tri logic b_def /* comment */
  , output var logic o_ghi
  );
endmodule

/* Module declaration with parameter ports.
*/
module Mod_B
  #(int FOO = 1 // comment
  , bit BAR = 2 /* comment */
  , bit [31:0] BAZ = 2
  , parameter int BUZZ = 4
  )
  ( input  var logic i_abc // comment
  , inout  tri logic b_def /* comment */
  , output var logic o_ghi
  );
```

```
/* Each item on its own line.
- Short lines.
- Every list indented to same level.
- Single-line LHS can be any length without indent issue.
*/
assign {foo, bar} =
  { i_abc
  , 12'h345
  , b_def     // comment
  , 16'h3456  /* comment */
  };


/* Everything can fit on one line.
- No space after opening parenthesis/bracket/brace.
*/
assign singleline1D = {i_abc, 12'h345, b_def, 16'h3456};
assign singleline2D = {{foo, bar}, {foo, bar}, {foo, bar}};

/* Multi-dimensional concatenation with innermost array on one line.
*/
assign matrix2D_A =
  { {elem21, elem20}
  , {elem11, elem10} // comment
  , {elem01, elem00} /* comment */
  };
assign matrix3D_A =
  { { {elem211, elem210}
    , {elem201, elem200}
    }
  , { {elem111, elem110} // comment
    , {elem101, elem100} /* comment */
    }
  , { {elem011, elem010}
    , {elem001, elem000}
    }
  };

/* Multi-dimensional concatenation with one element per line.
*/
assign matrix2D_B =
  { { elem21
    , elem20_with_long_name
    }
  , { elem11 // comment
```

```
    , elem10 /* comment */
    }
  , { elem01_note_no_misalignment
    , elem00
    }
  };

/* Module instance without parameter ports.
*/
Mod_A u_instanceA
  ( .i_abc(foo) // comment
  , .b_def({bar, bar}) /* comment */
  , .o_ghi
  );

/* Module instance with parameter ports.
*/
Mod_B
  #(.FOO(1) // comment
  , .BAR(2) /* comment */
  , .BUZZ(2)
  ) u_instanceB
  ( .i_abc(foo) // comment
  , .b_def({bar, bar}) /* comment */
  , .o_ghi
  );

endmodule
```

See also:

- **style__indent** - Suggested companion rule.

Rule: `style_indent`

Hint

Follow each newline with an integer multiple of 2 spaces.

Reason

Consistent indentation is essential for readability.

Pass Example (1 of 1)

```
module M;
  if (a)
    a = 0;
  else
    a = 1;
  // comment
/*
  comment
*/
endmodule
```

Fail Example (1 of 1)

```
module M;
 if (a)
   a = 0;
   else
     a = 1;
   // comment
/*
 comment
   */
endmodule
```

Explanation

Consistent indentation is essential for efficient reading by your human colleagues. This rule simply checks that any newline (outside of string literals) is followed by an integer multiple of 2 (configurable) space characters.

See also:

- **tab_character** - Suggested companion rule.

Rule: `style_keyword_0or1space`

Hint

Follow keyword with a symbol or exactly 1 space.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
  function F;
    if (a)
      return; // semicolon immediately after `return`.
    else
      return a; // 1 space then expression after `return`.
  endfunction
endmodule
```

Fail Example (1 of 1)

```
module M;
  function F();
    if (a)
      return  ; // Multiple spaces after `return`.
  endfunction
endmodule
```

Explanation

This rule checks the whitespace immediately following the `return` keyword. The `return` keyword can be used without an argument for void functions, in which case there should be no space between the keyword and the following symbol, i.e. `return;`. The `return` keyword can also be used with an argument, in which case there should be exactly 1 space between the keyword and the following identifier, e.g. `return foo;`.

See also:

- **style_keyword_0space** - Suggested companion rule.
- **style_keyword_1or2space** - Suggested companion rule.
- **style_keyword_1space** - Suggested companion rule.
- **style_keyword_construct** - Suggested companion rule.
- **style_keyword_datatype** - Potential companion rule.
- **style_keyword_end** - Suggested companion rule.
- **style_keyword_maybelabel** - Suggested companion rule.
- **style_keyword_newline** - Suggested companion rule.

Rule: `style_keyword_0space`

Hint

Remove all whitespace between keyword and following symbol.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
  always_comb
    case (a)
      123:
        b = c;
      default: // no space between `default` and colon.
        b = d;
    endcase

  function F;
    for (;;)
      if (a)
        break; // no space between `break` and semicolon.
  endfunction
endmodule
```

Fail Example (1 of 1)

```
module M;
  always_comb
    case (a)
      123:
        b = c;
      default : // Space between `default` and colon.
        b = d;
    endcase
  function foo ();
    for (;;)
      if (a) break  ; // Spaces between `break` and semicolon.
  endfunction
endmodule
```

Explanation

This rule checks the whitespace immediately following these keywords: `break` , `continue` , `default` , `new` , `null` , `super` , and `this`. Uses of these keywords

should never have any whitespace between the keyword and the following symbol, e.g. `break;` , `continue;` , `default:` , `new[5]` , `(myexample == null)` , or `super.foo`.

See also:

- **style_keyword_indent** - Suggested companion rule.
- **style_keyword_0or1space** - Suggested companion rule.
- **style_keyword_1or2space** - Suggested companion rule.
- **style_keyword_1space** - Suggested companion rule.
- **style_keyword_construct** - Suggested companion rule.
- **style_keyword_datatype** - Potential companion rule.
- **style_keyword_end** - Suggested companion rule.
- **style_keyword_maybelabel** - Suggested companion rule.
- **style_keyword_newline** - Suggested companion rule.

Rule: `style_keyword_1or2space`

Hint

Follow keyword with exactly 1 or 2 spaces.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M
  ( input a
  , inout b  // 1 space after `input` or `inout` keywords
  , output c // makes port identifiers unaligned.

  , input  d
  , inout  e // 2 spaces after `input` or `inout` keywords
  , output f // makes port identifiers aligned.
  );
endmodule
```

Fail Example (1 of 1)

```
module M
  ( input   a
  , inout   b // multiple spaces after `input` or `inout` keywords
  );
endmodule
```

Explanation

This rule checks the whitespace immediately following the `inout` and `input` keywords. These keywords specify the direction of signal ports, and are frequently used alongside the `output` keyword which is 1 character longer. The suggested companion rule **style_keyword_1space** checks that `output` is followed by a single space, and this rule allows `inout`/`input` to be followed by a single space too. However, it is common and visually appealing to have port definitions vertically aligned, so this rule also allows 2 following spaces, e.g:

```
module foo
  ( input  var logic i_foo // aligned, 2 spaces
  , output var logic o_bar
  , inout tri logic b_baz // unaligned, 1 space
  );
endmodule
```

See also:

- **style__keyword__indent** - Suggested companion rule.
- **style__keyword__0or1space** - Suggested companion rule.
- **style__keyword__0space** - Suggested companion rule.
- **style__keyword__1space** - Suggested companion rule.
- **style__keyword__construct** - Suggested companion rule.
- **style__keyword__datatype** - Potential companion rule.
- **style__keyword__end** - Suggested companion rule.
- **style__keyword__maybelabel** - Suggested companion rule.
- **style__keyword__newline** - Suggested companion rule.

Rule: `style_keyword_1space`

Hint

Follow keyword with exactly 1 space.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;                     // 1 space after `module`.
  for (i = 0; i < 5; i++)     // 1 space after `for`.
    assign foo = bar;         // 1 space after `assign`.
  always_ff @(posedge clk)    // 1 space after `always_ff`.
    if (a)                    // 1 space after `if`.
      case (a)                // 1 space after `case`.
        1: foo <= bar;
      endcase
endmodule
```

Fail Example (1 of 1)

```
module  M;                    // Multiple spaces after `module`.
  for(genvar i = 0; i < 5; i++)    // No spaces after `for`.
    assign  a = b;          // Multiple spaces after `assign`.
  always_ff@(posedge clk)     // No spaces after `always_ff`.
    if  (a)                   // Multiple spaces after `if`.
      case(a)                 // No spaces after `case`.
        1: a <= b;
      endcase
endmodule
```

Explanation

This rule checks the whitespace immediately following these keywords: `accept_on` , `alias` , `always` , `always_ff` , `and` , `assert` , `assume` , `automatic` , `before` , `bind` , `bins` , `binsof` , `bit` , `buf` , `bufif0` , `bufif1` , `case` , `casex` , `casez` , `cell` , `checker` , `class` , `clocking` , `cmos` , `config` , `const` , `constraint` , `context` , `cover` , `covergroup` , `coverpoint` , `cross` , `deassign` , `defparam` , `design` , `disable` , `dist` , `do` , `edge` , `enum` , `eventually` , `expect` , `export` , `extends` , `extern` , `first_match` , `for` , `force` , `foreach` , `forever` , `forkjoin` , `function` , `genvar` , `global` , `highz0` , `highz1` , `if` , `iff` , `ifnone` , `ignore_bins` , `illegal_bins` , `implements` , `implies` , `import` , `incdir` , `include` , `inside` , `instance` , `interconnect` , `interface` , `intersect` , `large` , `let` , `liblist` , `library` , `local` , `localparam` , `macromodule` , `matches` , `medium` , `modport` , `module` , `nand` , `negedge` , `nettype` , `nexttime` , `nmos` , `nor`

, `noshowcancelled` , `not` , `notif0` , `notif1` , `or` , `output` , `package` , `packed` , `parameter` , `pmos` , `posedge` , `primitive` , `priority` , `program` , `property` , `protected` , `pull0` , `pull1` , `pulldown` , `pullup` , `pulsestyle_ondetect` , `pulsestyle_onevent` , `pure` , `rand` , `randc` , `randcase` , `randsequence` , `rcmos` , `reject_on` , `release` , `repeat` , `restrict` , `rnmos` , `rpmos` , `rtran` , `rtranif0` , `rtranif1` , `s_always` , `s_eventually` , `s_nexttime` , `s_until` , `s_until_with` , `scalared` , `sequence` , `showcancelled` , `small` , `soft` , `solve` , `specparam` , `static` , `strong` , `strong0` , `strong1` , `struct` , `sync_accept_on` , `sync_reject_on` , `tagged` , `task` , `throughout` , `timeprecision` , `timeunit` , `tran` , `tranif0` , `tranif1` , `trireg` , `type` , `typedef` , `union` , `unique` , `unique0` , `until` , `until_with` , `untyped` , `use` , `var` , `vectored` , `virtual` , `wait` , `wait_order` , `weak` , `weak0` , `weak1` , `while` , `wildcard` , `with` , `within` , `xnor` , and `xor`. This rule covers the majority of SystemVerilog keywords, ensuring that they are followed by a single space, e.g. `if (foo)`, `always_ff @(posedge clk)`, or `typedef struct packed {`.

See also:

- **style__keyword__indent** - Suggested companion rule.
- **style__keyword__0or1space** - Suggested companion rule.
- **style__keyword__0space** - Suggested companion rule.
- **style__keyword__1or2space** - Suggested companion rule.
- **style__keyword__construct** - Suggested companion rule.
- **style__keyword__datatype** - Potential companion rule.
- **style__keyword__end** - Suggested companion rule.
- **style__keyword__maybelabel** - Suggested companion rule.
- **style__keyword__newline** - Suggested companion rule.

Rule: `style_keyword_construct`

Hint

Follow keyword with a newline or exactly 1 space.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
  always_comb a = b;  // 1 space after `always_comb`.

  initial begin       // 1 space after `initial`.
    foo = bar;
  end

  always_latch
    if (a) b = c;     // newline after `always_latch`.
    else d = e;       // 1 space after `else`.

  final // 1 space then comment after `final`.
    foo = bar;
endmodule
```

Fail Example (1 of 1)

```
module M;
  always_comb   a = b;  // Multiple spaces after `always_comb`.
  initial     begin       // Multiple spaces after `initial`.
    a = b;
  end
  always_latch
    if (a) b = c;
    else      d = e;  // Multiple spaces after `else`.
  final  // Multiple spaces then comment after `final`.
    a = b;
endmodule
```

Explanation

This rule checks the whitespace immediately following these keywords: `always_comb` , `always_latch` , `assign` , `else` , `final` , `generate` , and `initial`. These keyword open constucts and should always be followed by a newline, exactly 1 space then another keyword/identifier, or exactly 1 space then a comment, e.g:

```
// Followed by 1 space then another keyword.
always_comb begin
  foo = '0;
  foo[0] = 5;
end

// Followed by 1 space then an identifier.
always_comb bar = 5;

// Followed by a newline.
always_comb
  if (x < y)
    z = 5;
  else // Followed by 1 space then this comment.
    z = 6;

// Assign to a concatenation.
assign // You could use `always_comb` instead.
  { foo
  , bar
  , baz[i][j][k]
  } = '0;
```

See also:

- **style__keyword__indent** - Suggested companion rule.
- **style__keyword__0or1space** - Suggested companion rule.
- **style__keyword__0space** - Suggested companion rule.
- **style__keyword__1or2space** - Suggested companion rule.
- **style__keyword__1space** - Suggested companion rule.
- **style__keyword__datatype** - Potential companion rule.
- **style__keyword__end** - Suggested companion rule.
- **style__keyword__maybelabel** - Suggested companion rule.
- **style__keyword__newline** - Suggested companion rule.

Rule: `style_keyword_datatype`

Hint

Follow datatype keyword with a symbol or exactly 1 space.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
  localparam bit A = 0;    // 1 space after `bit`.
  localparam int B = 0;    // 1 space after `int`.
  logic a;                 // 1 space after `logic`.
  reg b;                   // 1 space after `reg`.
  wire b;                  // 1 space after `wire`.
endmodule
```

Fail Example (1 of 1)

```
module M;
  localparam bit  A = 0;  // Multiple spaces after `bit`.
  localparam int
    B = 0;                // Newline after `int`.
  logic // foo
    a;                    // Single-line comment after `logic`.
  reg /* bar */ b;        // Multi-line after `reg`.
  wire        c;          // Multiple spaces after `wire`.
endmodule
```

Explanation

This rule checks the whitespace immediately following these keywords: `byte` , `chandle` , `event` , `int` , `integer` , `logic` , `longint` , `real` , `realtime` , `ref` , `reg` , `shortint` , `shortreal` , `signed` , `string` , `supply0` , `supply1` , `time` , `tri` , `tri0` , `tri1` , `triand` , `trior` , `unsigned` , `uwire` , `void` , `wand` , `wire` , and `wor`. These keywords are used to declare the datatype of signals/variables (like `logic foo`), and cast expressions (like `int'(foo)`).

See also:

- **style_keyword_indent** - Suggested companion rule.
- **style_keyword_0or1space** - Suggested companion rule.
- **style_keyword_0space** - Suggested companion rule.
- **style_keyword_1or2space** - Suggested companion rule.
- **style_keyword_1space** - Suggested companion rule.
- **style_keyword_construct** - Suggested companion rule.

- **style__keyword__end** - Suggested companion rule.
- **style__keyword__maybelabel** - Suggested companion rule.
- **style__keyword__newline** - Suggested companion rule.

Rule: `style_keyword_end`

Hint

Follow keyword with a colon, newline, or exactly 1 space.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
  initial begin
    if (foo) begin: l_foo
      a = b;
    end: l_foo           // colon immediately after `end`.

    if (foo) begin
      a = c;
    end else begin       // 1 space after `end`.
      a = d;
    end
//  ^^^ newline after `end`.
  end // 1 space then comment after `end`.
endmodule
```

Fail Example (1 of 1)

```
module M;
  initial begin
    if (foo) begin: l_foo
      a = b;
    end   : l_foo          // Spaces between `end` and colon.

    if (foo) begin
      a = c;
    end   else begin       // Multiple spaces after `end`.
      a = d;
    end
  end   // Multiple spaces then comment after `end`.
endmodule
```

Explanation

This rule checks the whitespace immediately following the `end` keyword. The keyword `end` always be followed by a newline, exactly 1 space then another keyword, a colon, or exactly 1 space then a comment, e.g:

```
// Followed by a newline.
if (FOO) begin
  ...
end

// Followed by 1 space then a keyword.
if (FOO) begin
  ...
end else ...

// Followed by a colon.
if (FOO) begin: l_foo
  ...
end: l_foo

// Followed by a comment.
if (FOO) begin // {{{ An opening fold marker.
  ...
end // }}} A closing fold marker.
```

See also:

- **style_keyword_indent** - Suggested companion rule.
- **style_keyword_0or1space** - Suggested companion rule.
- **style_keyword_0space** - Suggested companion rule.
- **style_keyword_1or2space** - Suggested companion rule.
- **style_keyword_1space** - Suggested companion rule.
- **style_keyword_construct** - Suggested companion rule.
- **style_keyword_datatype** - Potential companion rule.
- **style_keyword_maybelabel** - Suggested companion rule.
- **style_keyword_newline** - Suggested companion rule.

Rule: `style_keyword_maybelabel`

Hint

Follow keyword with a colon, newline, or exactly 1 space plus comment.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
endmodule: M // colon immediately after `endmodule`
package P;
    function F;
    endfunction
//  ^^^^^^^^^^^ newline after `endfunction`
endpackage // 1 space then comment after `endpackage`
```

Fail Example (1 of 3)

```
module M;
endmodule  : M // spaces immediately after `endmodule`
```

Fail Example (2 of 3)

```
package P;
endpackage  // multiple spaces then comment after `endpackage`
```

Fail Example (3 of 3)

```
interface I;
endinterface interface J; // space instead of newline after `endinterface`
endinterface
```

Explanation

This rule checks the whitespace immediately following these keywords: `begin` , `endchecker` , `endclass` , `endclocking` , `endconfig` , `endfunction` , `endgroup` , `endinterface` , `endmodule` , `endpackage` , `endprimitive` , `endprogram` , `endproperty` , `endsequence` , `endtask` , `fork` , `join` , `join_any` , and `join_none`. These keywords are used to delimit code blocks and should always be followed by a colon, a newline, or exactly 1 space then a comment, e.g:

```
if (FOO) begin: l_foo // Followed by a colon.
  ...
end
```

```
module top;
  ...
endmodule: top  // Followed by a colon.

// Followed by a newline.
if (FOO) begin
  ...
end

if (FOO) begin // Followed by a comment.
  ...
end
```

See also:

- **style__keyword__indent** - Suggested companion rule.
- **style__keyword__0or1space** - Suggested companion rule.
- **style__keyword__0space** - Suggested companion rule.
- **style__keyword__1or2space** - Suggested companion rule.
- **style__keyword__1space** - Suggested companion rule.
- **style__keyword__construct** - Suggested companion rule.
- **style__keyword__datatype** - Potential companion rule.
- **style__keyword__end** - Suggested companion rule.
- **style__keyword__newline** - Suggested companion rule.

Rule: `style_keyword_newline`

Hint

Follow keyword with a newline or exactly 1 space plus comment.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
  generate
    case (foo)
      123: a = b;
    endcase
//  ^^^^^^^ newline after `endcase`
  endgenerate // 1 space then comment after `endgenerate`
endmodule
```

Fail Example (1 of 1)

```
module M;
  generate
    case (x)
      123: a = b;
    endcase if (x) a = b; // No newline after `endcase`.
  endgenerate   // Multiple spaces then comment after `endgenerate`.
endmodule
```

Explanation

This rule checks the whitespace immediately following these keywords: , `endcase` , `endgenerate` , `endspecify` , `endtable` , `specify` , and `table`. These keywords are used to delimit code blocks and should always be followed by a newline or exactly 1 space then a comment, e.g:

```
case (FOO)
  ...
endcase // Followed by a comment.

// Followed by a newline.
case (FOO)
  ...
endcase
```

See also:

- **style__keyword__indent** - Suggested companion rule.
- **style__keyword__0or1space** - Suggested companion rule.
- **style__keyword__0space** - Suggested companion rule.
- **style__keyword__1or2space** - Suggested companion rule.
- **style__keyword__1space** - Suggested companion rule.
- **style__keyword__construct** - Suggested companion rule.
- **style__keyword__datatype** - Potential companion rule.
- **style__keyword__end** - Suggested companion rule.
- **style__keyword__maybelabel** - Suggested companion rule.

Rule: `style_operator_arithmetic`

Hint

Follow operator with a symbol, identifier, newline, or exactly 1 space.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
  localparam bit [a-1:0] P1 = b; // No space around `-`.

  localparam int P2 = a + b; // Single space around `+`.

  localparam int P3 =
    a *
    b; // Newline following `*`.

  localparam int P4 =
    a * // Single space then comment following `*`.
    b;
endmodule
```

Fail Example (1 of 1)

```
module M;
  localparam int P2 = a  +  b; // Multiple spaces around `+`.

  localparam int P3 =
    a *

    b; // Multiple newlines following `*`.

  localparam int P4 =
    a *  // Multiple spaces then comment following `*`.
    b;
endmodule
```

Explanation

This rule checks the whitespace immediately following any arithmetic operator: `+` , `-` , `*` , `/` , `%` , and `**`. Uses of these operators may have a single space or newline between the operator's symbol and the following symbol or identifier, e.g. `a + b`, , or `a+b`.

In relation to Annex A of IEEE1800-2017, this rule applies to the specific variants of `binary_operator` specified in Table 11-3.

See also:

- **style__operator__boolean** - Suggested companion rule.
- **style__operator__integer** - Suggested companion rule.
- **style__operator__unary** - Suggested companion rule.

Rule: `style_operator_boolean`

Hint

Follow operator with a exactly 1 space.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
  localparam bit P1 = a && b; // Single space around `&&`.

  localparam bit P2 = a < b; // Single space around `<`.

  for (genvar i=0; i < 5; i++) begin // Single space around `<`.
  end
endmodule
```

Fail Example (1 of 1)

```
module M;
  localparam bit P1 = a&&b; // No space around `&&`.

  localparam bit P2 =
    a <
    b; // Newline after `<`.

  for (genvar i=0; i<5; i++) begin // No space around `<`.
  end
endmodule
```

Explanation

This rule checks the whitespace immediately following any binary operator whose operation returns a boolean: `==` , `!=` , `===` , `!==` , `==?` , `!=?` , `&&` , `||` , `<` , `<=` , `>` , `>=` , `->` , and `<->`. Uses of these operators must have a single space between the operator's symbol and the following symbol or identifier, e.g. `a && b`, , `c !== d` , or `0 < 5`.

In relation to Annex A of IEEE1800-2017, this rule applies to specific variants of `binary_operator` and `binary_module_path_operator`.

See also:

- **style__operator__arithmetic** - Suggested companion rule.
- **style__operator__integer** - Suggested companion rule.

- **style__operator__unary** - Suggested companion rule.

Rule: `style_operator_integer`

Hint

Follow operator with a newline or exactly 1 space.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
  localparam int P1 = a | b; // Single space around `|`.

  localparam int P2 =
    a &
    aMask; // Newline following `&`.

  localparam int P3 =
    a & // Single space then comment following `&`.
    aMask;
endmodule
```

Fail Example (1 of 1)

```
module M;
  localparam int P1 = a|b; // No space around `|`.

  localparam int P2 =
    a &

    aMask; // Multiple newlines following `&`.

  localparam int P3 =
    a &  // Multiple spaces then comment following `&`.
    aMask;
endmodule
```

Explanation

This rule checks the whitespace immediately following any binary operator whose operation returns an integer (except arithmetic operators): `&` , `|` , `^` , `^~` , `~^` , `>>` , `<<` , `>>>` , and `<<<`. Uses of these operators must have single space or a newline between the operator's symbol and the following symbol or identifier, e.g. `1 << 5`, , or `8'hAA | 8'h55`.

In relation to Annex A of IEEE1800-2017, this rule applies to specific variants of `binary_operator` and `binary_module_path_operator`.

See also:

- **style__operator__arithmetic** - Suggested companion rule.
- **style__operator__boolean** - Suggested companion rule.
- **style__operator__unary** - Suggested companion rule.

Rule: `style_operator_unary`

Hint

Remove all whitespace following the operator.

Reason

Consistent use of whitespace enhances readability by reducing visual noise.

Pass Example (1 of 1)

```
module M;
  localparam bit P1 = &{a, b}; // No space after `&`.

  for (genvar i=0; i < 5; i++) begin // No space after `++`.
  end
endmodule
```

Fail Example (1 of 1)

```
module M;
  localparam bit P1 = & {a, b}; // No space after `&`.

  for (genvar i=0; i < 5; i++ ) begin // Space after `++`.
  end
endmodule
```

Explanation

This rule checks the whitespace immediately following any unary operator: `++` , `--` , `+` , `-` , `!` , `~` , `&` , `~&` , `|` , `~|` , `^` , `~^` , and `^~`. Uses of these operators must never have any whitespace between the operator's symbol and the following symbol or identifier, e.g. `i++`, `!FOO`, , `&{a, b, c}` , or `$info("%d", -5);`.

In relation to Annex A of IEEE1800-2017, this rule applies to all variants of `unary_operator`, `unary_module_path_operator`, and `inc_or_dec_operator`.

See also:

- **style__operator__arithmetic** - Suggested companion rule.
- **style__operator__boolean** - Suggested companion rule.
- **style__operator__integer** - Suggested companion rule.

Rule: `style_trailingwhitespace`

Hint

Remove trailing whitespace.

Reason

Trailing whitespace leads to unnecessary awkwardness with version control.

Pass Example (1 of 1)

```
module        M;
// End of line ^
endmodule
```

Fail Example (1 of 1)

```
module M;

// End of line ^
endmodule

module M;
// End of line ^
endmodule
```

Explanation

Trailing whitespace, i.e. space characters immediately followed by a newline, lead to unnecessary differences in version control because some/many/most developer's editors are setup to remove this on writing to disk. This rule simply checks that any newline (outside of string literals) is not immediately preceeded by a space character. You can

See also:

- **style_indent** - Suggested companion rule.
- **tab_character** - Suggested companion rule.
- Vim: https://vimtricks.com/p/vim-remove-trailing-whitespace/
- Emacs: https://www.emacswiki.org/emacs/WhiteSpace
- VSCode: `files.trimTrailingWhitespace: true,`
- Notepad++: "Trim Trailing Space" on https://npp-user-manual.org/docs/editing/

Rule: `tab_character`

Hint

Replace tab characters with spaces.

Reason

Tabs may cause misalignment depending on editor setup.

Pass Example (1 of 1)

```
module M;
  logic a;
endmodule
```

Fail Example (1 of 1)

```
module M;
    logic a;
endmodule
```

Explanation

Tab characters appear as different widths in dependent on editor/viewer setup, leading to confusion for readers with a different setup. Spaces are all but essential, but tabs are not, so this rule simply forbids the use of tabs.

NOTE: `sv-parser`, the basis of svlint and svls requires files to be encoded in UTF-8. See `man iconv` for details on how to convert legacy encodings to UTF-8.

See also:

- **style_indent** - Suggested companion rule.

The most relevant clauses of IEEE1800-2017 are:

- Not applicable.

Rulesets

Some pre-configured rulesets are provided in `rulesets/*.toml`. A pre-configured ruleset can be used in the three standard ways (rename to `.svlint.toml` and place in the project root, the `--config` argument, or via the `SVLINT_CONFIG` environment variable). Pre-configured rulesets reside in `rulesets/*.toml`. There are two methods of specifying those TOML files:

1. Simply copy your existing `.svlint.toml` configuration into that directory. Ideally, add some comments to explain the background of the configuration and open a pull request to have it included as part of this project. This is the (initially) lower-effort approach, best suited to small personal projects with low requirements for documentation.
2. Create a definition in Markdown to compose a ruleset from a sequence of TOML fragments , i.e. write `md/ruleset-foo.md` to describe how the configuration in `rulesets/foo.toml` should be formed. Again, please open a pull request to have it included as part of this project. This approach is initially higher-effort but on larger projects, users will appreciate a good explanation of why configurations are necessary.

The rest of this section refers to the second method, which is a variant of literate programming.

If you only use one configuration, there isn't much benefit in having wrapper scripts, i.e. the benefits appear when you regularly use several configurations. For example, say you work on two projects called "A" and "B", and each project has its own preferences for naming conventions. This situation can be troublesome because there are many ways to get confused about which configuration file should be used on which files. Wrapper scripts help this situation by providing convenient commands like `svlint-A` and `svlint-B`. Another case for convenient access to specific rulesets is where you want to check that some files adhere to a particular set of rules, e.g. rules to reduce synthesis/simulation mismatches should apply to `design/*.sv` but not apply to `verif/*.sv`.

Each ruleset specification (in `md/ruleset-*.md`) is processed individually. A ruleset specification is freeform Markdown containing codeblocks with TOML (toml), POSIX shell (sh), or Windows batch (winbatch) language markers. Each ruleset specifies exactly one TOML configuration, one POSIX shell script, and one Windows batch script. For example, let this ruleset specification be placed in `md/ruleset-an-example.md`:

````
This is freeform Markdown.

Some explanation of how the **foo** and **bar** rules work together
with the associated option **blue**.
```toml
rules.foo = true
rules.bar = true
````
````

```
option.blue = "ABC"
```

Next, some text about the **baz** rule and another option **red**.
```toml
# A comment here.
rules.baz = true
option.red = "DEF"
```

Maybe some more Markdown text here.

This example will produce three files under `rulesets/` when cargo builds this crate: `an-example` (a POSIX shell script), `an-example.cmd` (a Windows batch script), and `an-example.toml`. A ruleset's TOML configuration is generated by concatenating all TOML codeblocks into one file, so the above example will produce this TOML file:

```toml
rules.foo = true
rules.bar = true
option.blue = "ABC"
# A comment here.
rules.baz = true
option.red = "DEF"
```

POSIX shell scripts begin with this header, where "an-example" is replaced by the ruleset's name:

```
#!/usr/bin/env sh
set -e

# If flag/options are given that don't use the ruleset config, simply run
# svlint with the given arguments.
NONRULESET="-h|--help|-V|--version|--dump-filelist|-E|--example|--update"
if printf "%b\n" " $*" | grep -Eq " (${NONRULESET})";
then
  svlint $*
  exit $?
fi

SVLINT_CONFIG="$(dirname $(command -v svlint-an-example))/an-example.toml"

# Delete ANSI control sequences that begin with ESC and (usually) end with m.
# Delete ASCII control characters except line feed ('\n' = 0o12 = 10 = 0x0A).
SANS_CONTROL="| sed -e 's/\\o33\\[[0-9;]*[mGKHF]//g'"
SANS_CONTROL="${SANS_CONTROL} | tr -d '[\\000-\\011\\013-\\037\\177]'"

# Combine the above output sanitization fragments into variables which can be
```

```
# evaluated and processed with xargs, e.g:
#   eval "${SVFILES}" | xargs -I {} sh -c "grep foo {};"
# NOTE: Creating a variable with the result (instead of the command) would lead
# to undefined behavior where the list of file paths exceeds 2MiB.
SVFILES="svlint --dump-filelist=files $* ${SANS_CONTROL}"
SVINCDIRS="svlint --dump-filelist=incdirs $* ${SANS_CONTROL}"
```

Next, any codeblocks with the `sh` language marker are concatenated to the header in order before, finally, this footer is appended:

```
env SVLINT_CONFIG="${SVLINT_CONFIG}" svlint $*
```

The final command calls the main `svlint` executable, wherever it is on your `$PATH`, with the environment variable `SVLINT_CONFIG` pointing to a TOML configuration in the same directory as the wrapper script. Any command line arguments given to the wrapper script are passed on to the main executable (via `$*`). When svlint searches for a configuration (`src/main.rs::search_config()`), the environment variable is chosen in preference to the `--config` flag which prevents confusing cases:

1. Where the script is given the option, e.g. `svlint-foo --config=bar *.sv`. As the environment variable is set, the option `--config=bar` is ignored. If a user wishes to pass a different configuration, they'll need to call the main executable like `svlint --config=bar *.sv`.
2. Where the environment variable is not set or is invalid, the `--config` value (defaulting to `.svlint.toml`) is searched for hierarchically, beginning in the current directory then moving upwards to filesystem ancestors.

If instead the `--config` option was used in wrapper scripts, this could lead to confusion where TOML files exist elsewhere in the hierarchy.

It isn't essential for all ruleset scripts to be POSIX compliant, but POSIX compliance should be encourage because it allows for consistent behavior across the widest range of systems. The utilities used in the POSIX wrappers are specified in the current POSIX standard (IEEE1003.1-2017, Volume 3: Shell and Utilities). Some resources related to these components:

- `env` Included in the Single Unix Specification since X/Open Portability Guide Issue 2 (1987).
- `sh` Included in the Single Unix Specification since X/Open Portability Guide Issue 2 (1987).
- `set` Specified in Section 2.14 Special Built-In Utilities, and available since (at least) X/Open Portability Guide Issue 2 (1987).
- `printf` Included in the Single Unix Specification since X/Open Common Application Environment (CAE) Specification Issue 4 (1994).
- `grep` Included in the Single Unix Specification since X/Open Portability Guide Issue 2 (1987).
- `command` Included in the Single Unix Specification since X/Open Common Application Environment (CAE) Specification Issue 4 (1994).

- `dirname` Included in the Single Unix Specification since X/Open Portability Guide Issue 2 (1987).
- `sed` Included in the Single Unix Specification since X/Open Portability Guide Issue 2 (1987).
- `tr` Included in the Single Unix Specification since X/Open Portability Guide Issue 2 (1987).

Windows batch scripts begin with this header, where "an-example" is replaced by the ruleset's name:

```
@echo off
for /f %%E in ('where.exe /f svlint-an-example') do (
    set "SVLINT_CONFIG=%%~dpEan-example.toml"
)
```

Next, any codeblocks with the `winbatch` language marker are then concatenated to the header in order before, finally, this footer is appended:

```
svlint %*
```

The batch script template is designed for Windows XP and later, using the `cmd.exe` shell. Some useful resources for Windows batch script commands:

- `echo`
- `for`
- `where`
- `set`

These wrapper scripts can then be used with svlint's usual arguments like `svlint-foo path/to/design/*.sv`. Note that this style of wrapper script allows you to use `$PATH` environment variable in the usual way, and that the wrapper scripts will simply use the first version of `svlint` found on your `$PATH`.

This method of generating a configuration and wrapper scripts enables full flexibility for each ruleset's requirements, while encouraging full and open documentation about their construction. The process, defined in `src/mdgen.rs`, is deterministic so both the Markdown specifications and the TOML configurations are tracked by versions control. However, wrapper scripts are not installed alongside the `svlint` binary created via `cargo install svlint` (or similar). Instead, you must either add `rulesets/` to your `$PATH` environment variable, or copy the wrapper scripts to somewhere already on your `$PATH`.

Ruleset: *designintent*

Rules that forbid suspicious constructions, i.e. ways of specifying hardware that are legal according to the LRM, but may express their intention unclearly.

This ruleset is a superset of **ruleset-simsynth**. These rules don't depend on each other or interact to provide additional properties.

```
rules.blocking_assignment_in_always_ff = true
rules.non_blocking_assignment_in_always_comb = true
rules.case_default = true
rules.enum_with_type = true
rules.function_with_automatic = true
rules.keyword_forbidden_priority = true
rules.keyword_forbidden_unique = true
rules.keyword_forbidden_unique0 = true
rules.level_sensitive_always = true # Redundant with keyword_forbidden_always.
```

This ruleset has further rules which don't depend on each other or combine to provide additional properties. Please see their individual explanations for details. Note, in the related **ruleset-verifintent**, the rule **keyword_forbidden_always** is not enabled because it is perfectly reasonable for a simulation testbench to schedule assignments, tasks, and functions in ways that wouldn't make sense for synthesizable hardware.

```
rules.action_block_with_side_effect = true
rules.default_nettype_none = true
rules.function_same_as_system_function = true
rules.keyword_forbidden_always = true
rules.keyword_forbidden_wire_reg = true
rules.non_ansi_module = true
```

When synthesised into a netlist, generate blocks should have labels so that their inferred logic can be detected in hierarchical paths. Although the LRM is clear about the implict naming of unlabelled generate blocks, see IEEE1800-2017 clause 27.6, using a well-named label provides some clarification about the author's intention for that logic. In the similar **ruleset-verifintent**, these rules are not enabled because they (mostly) relate to synthesizable hardware.

```
rules.generate_case_with_label = true
rules.generate_for_with_label = true
rules.generate_if_with_label = true
```

Generally, elaboration-time constant (`parameter`, `localparam`) should be 2-state types and always supplied with some default value. Additionally, where the context defines that `parameter` is an alias for `localparam`, author's should demonstate that they understand the constant cannot be overriden by using the `localparam` keyword.

```
rules.localparam_type_twostate = true
```

```
rules.parameter_type_twostate = true
rules.localparam_explicit_type = true
rules.parameter_explicit_type = true
rules.parameter_default_value = true
rules.parameter_in_generate = true
rules.parameter_in_package = true
```

Genvars, which are also elaboration-time constants, should be declared within generate `for` loops to reduce their scope. This allows readers to be confident that they can see all of the relevant information about a genvar in one place, i.e. declaration and usage. A notable advantage of declaring genvars in each generate loop is that authors are encouraged to give their genvars suitably descriptive names. Rules on the use of the `generate` and `endgenerate` keywords is similarly subjective, but this ruleset forbids their use because readers should be aware that all `case`, `for`, and `if` blocks outside of assignment processes are generate blocks. Further, the use of `generate` and `endgenerate` is entirely optional with no semantic difference to not using them.

```
rules.genvar_declaration_in_loop = true
rules.genvar_declaration_out_loop = false
rules.keyword_forbidden_generate = true
rules.keyword_required_generate = false
```

Rules in the below subset combine to provide an important property for the robust design of synthesizable hardware - that you can easily draw a schematic of what the synthesis result should look like. The two rules of thumb are to always fully specify decision logic, and never use sequential models for (what will be synthesized to) parallel logic.

```
rules.explicit_case_default = true
rules.explicit_if_else = true
rules.sequential_block_in_always_comb = true
rules.sequential_block_in_always_ff = true
rules.sequential_block_in_always_latch = true
```

Where sequential modelling of parallel logic is an unavoidable pragmatic approach, using the `begin` and `end` keywords should be done carefully with proper indentation. Note, this ruleset does *not* check the amount of indentation like **style_indent**.

```
rules.multiline_for_begin = true
rules.multiline_if_begin = true
```

The semantics around port declarations are, perhaps, unintuitive but were designed for backward compliance with Verilog (IEEE1364-1995). The below subset ensures that port declarations clearly convey important information about the direction and update mechanism of each signal port.

```
rules.inout_with_tri = true
rules.input_with_var = true
```

```
rules.output_with_var = true
rules.interface_port_with_modport = true
```

Ruleset: *parseonly*

If a file passes this ruleset you have these pieces of information:

- The file is valid UTF-8.
- svlint's preprocessor can successfully parse and emit text.
- The emitted text is valid SystemVerilog adhering to Annex A of IEEE1800-2017, i.e. there are no syntax errors.

Disable All Rules

All rules are implicitly disabled, and all options are implicitly set to their default values. Despite non of svlint's rules being enabled, this instructs the files to be preprocessed and parsed, i.e. internally processed from text to a syntax tree.

```
[option]
[rules]
```

Ruleset: *simsynth*

The set of checks which detect potential mismatches between simulation and synthesis.

Unlike the rules in, for example, **ruleset-style**, the rules in this ruleset do not depend on each other or combine to check additional properties. See the explanations of individual rules for their details.

```
rules.blocking_assignment_in_always_ff = true
rules.non_blocking_assignment_in_always_comb = true
rules.case_default = true
rules.enum_with_type = true
rules.function_with_automatic = true
rules.keyword_forbidden_priority = true
rules.keyword_forbidden_unique = true
rules.keyword_forbidden_unique0 = true
rules.level_sensitive_always = true
```

Ruleset: *style*

The set of whitespace-only checks which are "suggested" in the explanations of the **style__** rules.

Motivation

Style conventions also help a human reader to quickly and efficiently comprehend large bodies of code. Indeed, that is exactly what a reader wants to do when they're working with code written by other people, often complete strangers. The reader simply wishes to open the file, extract the necessary information, close the file, and get on with their life. Unlike mechanical tools, people process code visually (by translating their view of the screen into a mental model) and any noise which obscures the useful information will require extra mental effort to process. When code is written with consistent and regular whitespace, the important details like operators and identifiers are easily extracted. In contrast, when little attention is paid to indentation or spaces around keywords, operators, or identifers, the readers must waste their energy performing a mental noise reduction. Therefore, the main motivation behind this ruleset is to avoid visual noise.

Two notable style conventions help with a change-review process, i.e. comparing multiple versions of a file, rather than reading one version:

- Line length limited to a fixed number of characters, usually 80.
  - Excessively long lines may indicate problems with a program's logic.
  - Excessively long lines prevent viewing differences side-by-side.
  - Side-by-side reading is awkward when sideways scrolling is involved.
  - Code which is printed on paper cannot be scrolled sideways, and soft-wrap alternatives interrupt indentation.
- Trailing whitespace is forbidden.
  - Changes to trailing whitespace are not usually visible to human readers, but are found by version control tools.
  - Editors are often configured to remove trailing whitespace, resulting in unnecessary differences.
  - Git, a popular version control tool will (by default) warn against trailing whitespace with prominent markers specifically because of the unnecessary noise introduced to a repository's history.

These conventions help give a consistent view over different ways of viewing files which include the writer's text editor (Vim, VSCode, Emacs, etc.), consumer's text editor, reviewer's web-based tools (GitHub, BitBucket, GitLab, etc.), printed material (e.g. via PDF), and logfiles from CI/CD tools (GitHub Actions, Bamboo, Jenkins, etc).

Test Each File for Excessively Long Lines

To get a list of all the files examined by a particular invocation of svlint, use the variable `${SVFILES}`, which is provided in all POSIX wrapper scripts.

The `grep` utility can be used to detect, and report, lines longer than a given number of characters.

```
TEXTWIDTH='80'
LINELEN="grep -EvIxHn --color '.{0,${TEXTWIDTH}}' {};"
LINELEN="${LINELEN} if [ \"\$?\" -eq \"0\" ]; then"
LINELEN="${LINELEN}   echo '!!! Lines longer than ${TEXTWIDTH} characters !!!';"
LINELEN="${LINELEN}   exit 1;"
LINELEN="${LINELEN} else"
LINELEN="${LINELEN}   exit 0;"
LINELEN="${LINELEN} fi"
eval "${SVFILES}" | xargs -I {} sh -c "${LINELEN}"
```

Another use of `grep` is to report obfuscated statements where semicolons are pushed off the RHS of the screen.

```
OBFUSTMT="grep -EIHn --color '[ ]+;' {};"
OBFUSTMT="${OBFUSTMT} if [ \"\$?\" -eq \"0\" ]; then"
OBFUSTMT="${OBFUSTMT}   echo '!!! Potentially obfuscated statements !!!';"
OBFUSTMT="${OBFUSTMT}   exit 1;"
OBFUSTMT="${OBFUSTMT} else"
OBFUSTMT="${OBFUSTMT}   exit 0;"
OBFUSTMT="${OBFUSTMT} fi"
eval "${SVFILES}" | xargs -I {} sh -c "${OBFUSTMT}"
```

On Windows, the default environment does not contain utilities such as `grep`, so some system-specific scripting may be more appropriate.

Indentation

An indent of 2 spaces, not tabs, is chosen. For better or worse, contemporary computer language styles have moved decisively away from using tabs for indentation. The most likely reason behind this is that tab display width is configurable so tab indentations are shown differently, depending on the reader's personal configuration.

```
option.indent = 2
rules.tab_character = true
rules.style_indent = true
```

Note that the **style__indent** rule does not check that indentations are the correct level - only that the indentation is an integer multiple of 2 spaces.

In SystemVerilog, most of the language is independent of whitespace characters, so readers are (hopefully) aware that they should be careful not to interpret

indentation with semantic meaning, but its human nature to do so. Therefore, author care is still required to use the correct indent, i.e. **style__indent** only points out indentations which are obviously wrong, but does not understand the logical semantics of any SystemVerilog constructs.

```
always_comb begin
  x = 0;
  y = 123;

  if (a)
    x = 1;
  else
    x = 2;
    y = 666;

  z = y + x;
end
```

Above is a simple demonstration of how the human eye can be misled in ways that mechanical tools like compilers are immune to. Depending on the value of expression `a`, the variable `z` takes the value either `667` or `668`, but never `124`. To mitigate the risk of confusion around multi-line conditional statements and loops, two further rules are enabled to check that either `begin`/`end` keyword delimiters are used, or the statement is moved to the same line as the condition.

```
rules.multiline_if_begin = true
rules.multiline_for_begin = true
```

Indentation Preprocessor Considerations

A potential source of confusion is in the use of the preprocessor to accidentally introduce whitespace. In these examples, a dot character (`.`) is used to visually present a space character where it's important.

```
`ifdef A
..foo();
`endif.// A space between the "endif" directive and the line comment
```

If `A` is defined, the above example will be emitted from the preprocessor as this text:

```
foo();
.// A space between the "endif" directive and the line comment
```

The line after `foo()` begins with a 1 space, which violates the **style__indent** check. Note that the violation occurs even if `A` is not defined.

To further confuse things, the following example will *not* cause a violation when `A` is undefined!

```
.`ifdef A
..foo();
.`endif.// A space between the "endif" directive and the line comment
```

The 1 space on the `ifdef` line is joined to the 1 space after `endif` to make a line with a 2 space indent like this:

```
..// A space between the "endif" directive and the line comment
```

Confusing situations like these arise from the fact that SystemVerilog is a combination of two languages;

1. A text processing language (defining the preprocessor) in specified informally in IEEE1800-2017 Clause 22 among other compiler directives.
2. The rest of SystemVerilog syntax is formally called `source_text`, is specified formally in IEEE1800-2017 Annex A.

Svlint rules operate on the `source_text` part of SystemVerilog, i.e. after the preprocessor has been applied. As with other languages with similar text-based templating features, most notably C, use of the preprocessor is discouraged except where absolutely necessary. To avoid confusion with preprocessor, here are two recommendations:

1. Don't indent compiler directives, especially preprocessor statements containing any `source_text`.
2. Don't put any spaces between compiler directives and comments on the same line.

These are some examples of confusion-ridden style, not recommended.

```
`define Z // Space then comment
`ifdef A // Space then comment
..`ifdef B// Indented ifdef
....foo(); // Indent of source_text mixed with preprocessor
..`endif// Indented endif
`endif // Space then comment
```

The above examples can be reformed like this:

```
`define Z// No space then comment
`ifdef A// No space then comment
`ifdef B
..foo();
`endif// B
`endif// A
```

Where no `source_text` is contained in the ifdef block, i.e. only preprocessor definitions, these may be indented without causing confusion:

```
`ifdef A
..`ifdef B
....`define Z
```

```
..`endif// B
`endif// A
```

For clarification, when both `A` and `B` are defined, the above block will be emitted from the svlint preprocessor as shown below.

```
`define Z
..// B
// A
```

One method which can help catch unintended whitespace, both from the preprocessor and written by hand, is to forbid trailing spaces, i.e. space characters followed immediately by a newline.

```
rules.style_trailingwhitespace = true
```

Problems around indented preprocessor directives must be caught before svlint's preprocessor stage, so searching with `grep` beforehand is appropriate.

```
PPDIRECTIVES="define|undef|undefineall|resetall"
PPDIRECTIVES="${PPDIRECTIVES}|ifdef|ifndef|elsif|else|endif"
PPDIRECTIVES="${PPDIRECTIVES}|include"

PPINDENT="grep -EIHn --color '[ ]+\`(${PPDIRECTIVES})' {};"
PPINDENT="${PPINDENT} if [ \"\$?\" -eq \"0\" ]; then"
PPINDENT="${PPINDENT}   echo '!!! Indented preprocessor directives !!!';"
PPINDENT="${PPINDENT}   exit 1;"
PPINDENT="${PPINDENT} else"
PPINDENT="${PPINDENT}   exit 0;"
PPINDENT="${PPINDENT} fi"
eval "${SVFILES}" | xargs -I {} sh -c "${PPINDENT}"
```

Operators and Keywords

Consistent use of whitespace around operators and keywords makes it easier to read expressions quickly and accurately.

```
rules.style_operator_arithmetic = true
rules.style_operator_boolean = true
rules.style_operator_integer = true
rules.style_operator_unary = true

rules.style_keyword_0or1space = true
rules.style_keyword_0space = true
rules.style_keyword_1or2space = true
rules.style_keyword_1space = true
rules.style_keyword_construct = true
rules.style_keyword_datatype = false # Overly restrictive.
rules.style_keyword_end = true
```

```
rules.style_keyword_maybelabel = true
rules.style_keyword_newline = true
```

Comma-Separated Lists

SystemVerilog code has many uses for comma-separated lists of items specified in IEEE1800-2017 Annex A. Most of these uses can be found by searching for BNF symbols containing the string `list_of_`, but uses are specified in BNF expressions for other symbols, e.g. `modport_declaration` and `data_type`.

Without careful review processes in place, the large variety semantics and syntax surrounding comma-separated lists can easily lead authors writing in a large variety of styles. To make matters worse, the use of comma-separated lists varies is common in other languages - but with significant subtle differences. For example, while Python and Rust allow an extra comma after the last argument in a function call, C and SystemVerilog do not allow this.

The desire for consistent formatting and readability provides motivation for a simple rule which can be easily remembered by authors. The most common style in functional programming language Haskell provides inspiration for such a rule: "Every comma must be followed by exactly one space".

```
rules.style_commaleading = true
```

This rule leads to the comma-leading style which, although perhaps unfamiliar to authors with a background in C or Python, has a number of advantages.

- The rule is extremely simple, especially in comparison to the multitude of rules requried to format comma-trailing lists consistently.
- A comma character is visually similar to bullet-point.
- When changing code over time, it's more common to add items to the end of a list than the beginning. This means that comma-leading style often leads to diffs which are easier to review. Closely related to this is that comma-leading style makes it less likely to introduce an extra comma at the end of a list (which would be a syntax error).
- Multi-dimensional arrays are easier to read, because it's natural to put a line without elements (only the closing `}`) between elements of the more-significant axis.
- Comma is visually similar to bulletpoint (a common symbol for introducing an item of a list in prose).
- Comma-leading style can be said to be more closely aligned with BNF specification, e.g. `list_of_genvar_identifiers ::= genvar_identifier { , genvar_identifier }`. This is reflected by how sv-parser attaches `Comment` nodes (which contain whitespace) to the RHS of comma symbols.

For some examples, please see the explanation of the **style_commaleading** rule.

Ruleset: *verifintent*

Rules that forbid suspicious constructions, i.e. those which are legal according to the LRM, but may express their intention unclearly. This ruleset is similar to **ruleset-designintent**, but with some rules enabled or disabled where they are applicable to testbench code (instead of synthesizable digital logic).

While this ruleset is *not* a superset of **ruleset-simsynth**, some of those rules are also useful for testbench/verification code. A clean separation of (non-)blocking assignments and `always_(comb|ff)` processes is useful to prevent the specification of processes with scheduling semantics which are difficult to reason about.

```
rules.blocking_assignment_in_always_ff = true
rules.non_blocking_assignment_in_always_comb = true
rules.enum_with_type = true
rules.keyword_forbidden_priority = true
rules.keyword_forbidden_unique = true
rules.keyword_forbidden_unique0 = true
```

This ruleset has further rules which don't depend on each other or combine to provide additional properties. Please see their individual explanations for details. Note, in the related **ruleset-designintent**, an additional rule **keyword_forbidden_always** is enabled.

```
rules.action_block_with_side_effect = true
rules.default_nettype_none = true
rules.function_same_as_system_function = true
rules.keyword_forbidden_wire_reg = true
rules.non_ansi_module = true
```

Generally, elaboration-time constant (`parameter`, `localparam`) should be 2-state types and always supplied with some default value. Additionally, where the context defines that `parameter` is an alias for `localparam`, author's should demonstate that they understand the constant cannot be overriden by using the `localparam` keyword.

```
rules.localparam_type_twostate = true
rules.parameter_type_twostate = true
rules.localparam_explicit_type = true
rules.parameter_explicit_type = true
rules.parameter_default_value = true
rules.parameter_in_generate = true
rules.parameter_in_package = true
```

Genvars, which are also elaboration-time constants, should be declared within generate `for` loops to reduce their scope. This allows readers to be confident that they can see all of the relevant information about a genvar in one place, i.e. declaration and usage. A notable advantage of declaring genvars in each

generate loop is that authors are encouraged to give their genvars suitably descriptive names. Rules on the use of the `generate` and `endgenerate` keywords is similarly subjective, but this ruleset forbids their use because readers should be aware that all `case`, `for`, and `if` blocks outside of assignment processes are generate blocks. Further, the use of `generate` and `endgenerate` is entirely optional with no semantic difference to not using them.

```
rules.genvar_declaration_in_loop = true
rules.genvar_declaration_out_loop = false
rules.keyword_forbidden_generate = true
rules.keyword_required_generate = false
```

To prevent difficult-to-read procedural code, using the `begin` and `end` keywords should be done carefully with proper indentation. Note, this ruleset does *not* check the amount of indentation like **style__indent**.

```
rules.multiline_for_begin = true
rules.multiline_if_begin = true
```

The semantics around port declarations are, perhaps, unintuitive but were designed for backward compliance with Verilog (IEEE1364-1995). The below subset ensures that port declarations clearly convey important information about the direction and update mechanism of each signal port.

```
rules.inout_with_tri = true
rules.input_with_var = true
rules.output_with_var = true
rules.interface_port_with_modport = true
```