ALINT-PRO

ALDEC SV Rules

Table of Contents

Table of Contents

Table of Contents

Table of Contents

ALDEC SV Rules Overview

The ALDEC SV rules library includes 61 rule checkers given in the table below.

Legend for linting phases:

Parse	Checks can be executed after parse is finished.
Elaboration	Checks can be executed after elaboration is finished.
Synthesis	Checks can be executed after synthesis is finished.
Constraints	Checks can be executed after constraints phase is finished.
Chip	Checks can be executed after constraints linting is finished.

Name	Phase	Title	Base Rule
ALDEC_SV.1.1.1	Elaboration	Tasks and functions defined in the package should be declared as automatic.	ALINT_SV.2.5.1.6
ALDEC_SV.1.1.2	Elaboration	Tasks and functions defined in the package should not contain static variables.	ALINT_SV.2.5.1.7
ALDEC_SV.1.1.3	Parse	Do not use variables declared in packages for inter-module communication.	ALINT_SV.2.3.1.8
ALDEC_SV.1.2.6	Parse	Avoid placing global variables, static tasks/functions to '$unit' scope.	ALINT_SV.1.4.2.9
ALDEC_SV.1.2.7	Parse	Prefer declaring user-defined types in named packages instead of '$unit' scope.	ALINT_SV.1.4.2.9
ALDEC_SV.1.2.8	Parse	Prefer declaring automatic tasks/functions in named packages instead of '$unit'.	ALINT_SV.1.4.2.9
ALDEC_SV.1.2.9	Parse	Avoid using global parameters within the '$unit' scope.	ALINT_VLOG.1.4.2.1
ALDEC_SV.1.2.10	Parse	Use package import in the module scope only.	ALINT_SV.1.4.2.9
ALDEC_SV.1.3.1	Parse	Use ANSI C-styled module declaration.	ALINT_SV.1.2.4.11
ALDEC_SV.1.3.2	Elaboration	Reference module ports are non-synthesizable.	ALINT_VLOG.1.2.4.2
ALDEC_SV.1.3.3	Parse	Place nested modules into separate files included to the main module.	ALINT_SV.1.4.1.8
ALDEC_SV.1.3.4	Parse	Do not use the (.*) connection.	ALINT_SV.1.2.5.5
ALDEC_SV.1.3.5	Parse	Use .name to connect a net to a port with the same name.	ALINT_SV.1.2.5.7
ALDEC_SV.1.4.1	Elaboration	When similar groups of ports are passed through port maps of different modules, extract interfaces.	ALINT_SV.1.4.3.3
ALDEC_SV.1.4.3	Elaboration	Interface methods must be automatic to be synthesizable.	ALINT_SV.2.5.1.6
ALDEC_SV.1.4.4	Elaboration	Interface methods should not define static variables to be synthesizable.	ALINT_SV.2.5.1.7
ALDEC_SV.1.4.5	Elaboration	Exported methods are not synthesizable.	ALINT_SV.2.5.1.9
ALDEC_SV.1.4.6	Parse	Define modport in interface.	ALINT_SV.1.4.3.10
ALDEC_SV.1.4.7	Elaboration	Specify modport when calling the interface.	ALINT_SV.1.4.3.11
ALDEC_SV.1.5.1	Parse	Prefer module-level 'timeunit'/'timeprecision' declarations to file-level preprocessor directives.	ALINT_SV.1.2.7.12
ALDEC_SV.1.5.3	Parse	Prefer 'timeunit'/'timeprecision' declarations in design units instead of '$unit'.	ALINT_SV.1.4.2.9
ALDEC_SV.1.5.4	Parse	Avoid using default time unit and precision.	ALINT_SV.5.1.2.6

ALDEC_SV.2.1.1	Parse	In synthesizable code use 2-state types like 'int' only as a counter in 'for' loop.	ALINT_VLOG.1.3.1.1
ALDEC_SV.2.1.2	Elaboration	Avoid using 2-state type variables in 'casex'/'casez' statements due to undefined simulation semantics.	ALINT_SV.1.3.1.10
ALDEC_SV.2.1.3	Elaboration	Be careful with X-assignments to 2-state type variables.	ALINT_SV.5.2.3.9
ALDEC_SV.2.1.4	Synthesis	Reset logic is absolutely necessary for flip-flops based on 2-state type variables to avoid simulation vs synthesis mismatches.	ALINT_SV.3.1.1.10
ALDEC_SV.2.2.1	Parse	Use a fill value ('1,'0,'z,'x) for vector literals instead of a hard coded value.	ALINT_SV.1.2.2.9
ALDEC_SV.2.2.2	Elaboration	Avoid assign of non-constants to run-time constant in synthesizable description.	ALINT_SV.5.2.1.12
ALDEC_SV.2.3.1	Elaboration	Do not declare identical 'typedef' in multiple contexts.	ALINT_SV.1.4.2.8
ALDEC_SV.2.3.2	Elaboration	Never declare two different typedefs with the same name in different contexts.	ALINT_SV.1.4.2.11
ALDEC_SV.2.3.3	Elaboration	Follow naming conventions for user-defined types.	ALINT_SV.1.1.2.8
ALDEC_SV.2.3.4	Elaboration	Use enumeration methods (like .next, .prev) instead of arithmetic operations upon the enumeration variables or literals.	ALINT_SV.1.3.5.12
ALDEC_SV.2.3.5	Elaboration	Do not use class types in RTL description.	ALINT_SV.2.3.1.9
ALDEC_SV.2.3.6	Parse	Prefer named enumeration types to anonymous enumeration variables.	ALINT_SV.1.4.2.12
ALDEC_SV.2.3.7	Parse	Use 'var' keyword to emphasize variable declarations with user-defined types.	ALINT_SV.1.2.3.1
ALDEC_SV.2.3.8	Synthesis	Use enumerated type to define FSM states.	ALINT_SV.3.3.3.5
ALDEC_SV.2.3.9	Parse	Typedefs should be defined in packages only.	ALINT_SV.1.4.2.9
ALDEC_SV.2.4.1	Parse	Prefer named struct/union types to anonymous structure variables.	ALINT_SV.1.4.2.12
ALDEC_SV.2.4.4	Elaboration	Avoid out-of-range references when treating packed structures/unions as vectors.	ALINT_VLOG.1.3.3.4
ALDEC_SV.2.4.5	Elaboration	Avoid unpacked unions in synthesizable description.	ALINT_SV.2.3.1.13
ALDEC_SV.2.4.7	Elaboration	Prefer to use tagged packed unions and define a flag field member in structures with packed unions.	ALINT_SV.2.3.1.14
ALDEC_SV.2.5.2	Elaboration	Declare typedefs for identical arrays used more than once or when passing data through function/task ports.	ALINT_SV.1.3.3.8
ALDEC_SV.2.5.4	Elaboration	Avoid dynamic array types in synthesizable description.	ALINT_SV.2.3.1.10
ALDEC_SV.2.6.1	Parse	Specify port type and direction explicitly.	ALINT_SV.1.2.4.12
ALDEC_SV.2.6.2	Elaboration	Use wire type for multi-driven objects and logic type for others.	ALINT_SV.1.3.1.8
ALDEC_SV.3.1.2	Parse	Prefer 'always_comb' to 'always' with @* sensitivity.	ALINT_SV.5.2.2.7
ALDEC_SV.3.1.3	Synthesis	Use 'always_latch' block for generating latches only.	ALINT_SV.3.2.1.3
ALDEC_SV.3.2.1	Elaboration	Prefer void functions to tasks to enforce synthesizability checks.	ALINT_SV.2.5.1.8
ALDEC_SV.3.2.6	Parse	Functions should be defined in packages only.	ALINT_SV.1.4.2.9
ALDEC_SV.3.3.3	Parse	Do not use increment/decrement operators in expressions.	ALINT_SV.2.3.1.4

ALDEC_SV.3.3.4	Parse	Do not use new assignment operators (such as +=, -=) in expressions.	ALINT_SV.2.3.1.4
ALDEC_SV.3.3.5	Elaboration	Do not write same variable in different always_comb, always_ff, always_latch blocks.	ALINT_SV.2.6.1.6
ALDEC_SV.3.4.2	Elaboration	Use only constant expressions within 'inside' operator with wildcards.	ALINT_SV.2.3.1.7
ALDEC_SV.3.5.1	Elaboration	Avoid casts that narrow the width of the value.	ALINT_SV.1.3.4.6
ALDEC_SV.3.5.2	Parse	Do not use dynamic '$cast' function in RTL description.	ALINT_SV.2.3.1.11
ALDEC_SV.3.5.3	Elaboration	Avoid casting out-of-range numeric values to enumeration types.	ALINT_SV.1.3.4.7
ALDEC_SV.3.6.1	Parse	Prefer locally declared counters in 'for' loops.	ALINT_SV.2.7.1.8
ALDEC_SV.3.7.1	Elaboration	Use SystemVerilog 'case-inside' instead of Verilog 'casex' and 'casez'.	ALINT_SV.5.2.1.11
ALDEC_SV.3.7.2	Parse	Prefer 'unique-case' to full_case + parallel_case pragmas.	ALINT_SV.1.2.7.10
ALDEC_SV.3.7.3	Parse	Prefer 'priority-case' to full_case pragma.	ALINT_SV.1.2.7.10
ALDEC_SV.3.7.4	Parse	Prefer 'unique0-case' to parallel_case pragma.	ALINT_SV.1.2.7.10

Chapter 1 - Hierarchical Constructs

Section 1.1 - Packages

ALDEC_SV.1.1.1

Rule Name

Tasks and functions defined in the package should be declared as automatic.

Base Rule

ALINT_SV.2.5.1.6

Phase

Elaboration

Message

The "{SubprogramName}" static {SubprogramType} is declared in the "{UnitName}" {UnitType}.

Problem Description

It is recommended to declare tasks or functions defined in the package as automatic, and they should not contain static variables. When an automatic task/function is referenced in a module, the unique copy of such task/function is visible only for this module and there is no sharing of task/function items. Such approach prevents mismatches between pre-synthesis and post-synthesis behavior, when several modules reference same task/function.

Level Rule

Checker Behaviour

Checker scans packages:

- if a non-automatic subprogram is detected and it is called in synthesizable description => violation.

Example - Subprogram inside package

Description

In the following example the "adder" function is static (by default) and it is referenced in module => violation.

Sample Code

`package.sv`:

```
1:  package definitions;
2:    function [15:0] adder (input [15:0] a, b);
3:      return a + b;
4:    endfunction
```



```
 5:    //other declarations
 6:  endpackage
```

TOP.sv:

```
 1:
 2:  module top (
 3:    input [31:0] cod_l,
 4:    output reg [15:0] result
 5:    // ports list
 6:  );
 7:    import definitions::*;
 8:
 9:    always_comb
10:      result = adder(cod_l[15:0], cod_l[31:16]);
11:
12:    //statements
13:  endmodule
```

The "adder" static function is declared in the "definitions" package.

How to Fix

Change the function type from static to automatic.

Fixed Code

package.sv:

```
 1:  package definitions;
 2:    function automatic [15:0] adder (input [15:0] a, b);
 3:      return a + b;
 4:    endfunction
 5:    //other declarations
 6:  endpackage
```

TOP.sv:

```
 1:
 2:  module top (
 3:    input [31:0] cod_l,
 4:    output reg [15:0] result
 5:    // ports list
 6:  );
 7:    import definitions::*;
 8:
 9:    always_comb
10:      result = adder(cod_l[15:0], cod_l[31:16]);
11:
12:    //statements
13:  endmodule
```


ALDEC_SV.1.1.2

Rule Name

Tasks and functions defined in the package should not contain static variables.

Base Rule

ALINT_SV.2.5.1.7

Phase

Elaboration

Message

The "{SubprogramName}" automatic {SubprogramType} declared in the "{UnitName}" {UnitType} contains static variable(s).

The "{VariableName}" static variable is declared here.

Problem Description

Static variable can be defined in an automatic task/function and they will be kept as static. Such variable is re-used between different calls of the task/function. To prevent changing variable from outside the subprogram body and simulation-synthesis mismatches, do not use static variables within automatic tasks and functions defined in packages.

Level Rule

Checker Behaviour

The checker verifies automatic subprograms inside packages:

- if a static variable is detected inside an automatic subprogram that is called in synthesizable description => violation.

Example - Automatic function with static variable

Description

In the following example the "adder" automatic function contains static variable => violation.

Sample Code

`package.sv`:

```
1:  package definitions;
2:    function automatic [15:0] adder (input [15:0] a, b);
3:      static logic [2:0] temp;
4:      //statements
5:      return a + b;
6:    endfunction
```



```
 7:    //other declarations
 8:  endpackage
```

TOP.sv:

```
 1:
 2:  module top ( input [31:0] cod_l,
 3:    output reg [15:0] result
 4:    // ports list
 5:  );
 6:    import definitions::*;
 7:
 8:    always_comb
 9:      result = adder(cod_l[15:0], cod_l[31:16]);
10:
11:    //statements
12:  endmodule
```

The "adder" automatic function declared in the "definitions" package contains static variable(s).

The "temp" static variable is declared here.

How to Fix

Change lifetime of the variable from static to automatic.

Fixed Code

package.sv:

```
 1:  package definitions;
 2:    function automatic [15:0] adder (input [15:0] a, b);
 3:      logic [2:0] temp;
 4:      //statements
 5:      return a + b;
 6:    endfunction
 7:    //other declarations
 8:  endpackage
```

TOP.sv:

```
 1:
 2:  module top ( input [31:0] cod_l,
 3:    output reg [15:0] result
 4:    // ports list
 5:  );
 6:    import definitions::*;
 7:
 8:    always_comb
 9:      result = adder(cod_l[15:0], cod_l[31:16]);
10:
11:    //statements
12:  endmodule
```


ALDEC_SV.1.1.3

Rule Name

Do not use variables declared in packages for inter-module communication.

Base Rule

ALINT_SV.2.3.1.8

Phase

Parse

Message

The "{ObjectName}" {ObjectType} that declared inside "{PackageName}" package is used in synthesizable description.

The "{ObjectName}" {ObjectType} is used inside "{ModuleName}" module here.

Problem Description

When package variables are used for inter-module communication, they are visible and modifiable to all modules which import those variables. Such description is not synthesizable, therefore it is not recommended to use inter-module communication without passing values through module ports.

Level Rule

Checker Behaviour

The checker verifies variables and nets declared inside packages:

- if a variable or net is used outside of a package in synthesizable modules => violation.

Example - Use of package variable in modules

Description

In the following example the "inv_data" variable declared in the "definitions" package is used for inter-module communication => violation.

Sample Code

`package.sv`:

```
1:  package definitions;
2:    logic [1:0] inv_data;
3:    //other declarations
4:  endpackage
```

`module_1.sv`:


```
 1:  module ff (
 2:    input logic clk,
 3:    input logic rst,
 4:    output logic [1:0] q
 5:  );
 6:    import definitions::*;
 7:
 8:    always_ff @(posedge clk or posedge rst)
 9:      if (rst)
10:        q <= 'b0;
11:      else
12:        q <= inv_data;
13:
14:  endmodule
```

module_2.sv:

```
 1:  module inv (
 2:    input logic [1:0] data_in
 3:  );
 4:    import definitions::*;
 5:
 6:    assign inv_data = ~data_in;
 7:
 8:  endmodule
```

The "inv_data" variable that declared inside "definitions" package is used in synthesizable description.

The "inv_data" variable is used inside "ff" module here.

The "inv_data" variable is used inside "inv" module here.

How to Fix

Use a signal from the top-level unit to pass data between two instantiated modules.

Fixed Code

module_1.sv:

```
 1:  module ff (
 2:    input logic clk,
 3:    input logic rst,
 4:    input logic [1:0] data,
 5:    output logic [1:0] q
 6:  );
 7:
 8:    always_ff @(posedge clk or posedge rst)
 9:      if (rst)
10:        q <= 'b0;
11:      else
12:        q <= data;
13:
14:  endmodule
```

module_2.sv:

```
 1:  module inv (
 2:    input logic [1:0] data_in,
 3:    output logic [1:0] inv_data
 4:  );
 5:
```



```
 6:    assign inv_data = ~data_in;
 7:
 8:  endmodule
```

top.sv:

```
 1:  module top (
 2:    input logic clk,
 3:    input logic rst,
 4:    input logic [1:0] data_in,
 5:    output logic [1:0] q
 6:  );
 7:
 8:    logic [1:0] inv_data;
 9:
10:    inv u_inv (.data_in, .inv_data);
11:    ff u_ff (.*, .data(inv_data));
12:
13:  endmodule
```


Section 1.2 - $unit

ALDEC_SV.1.2.6

Rule Name

Avoid placing global variables, static tasks/functions to '$unit' scope.

Base Rule

ALINT_SV.1.4.2.9

Phase

Parse

Message

The "{DeclarationName}" {DeclarationType} is declared in the compilation-unit scope.

Problem Description

Objects declared in the **$unit** scope are not actually global. Such declarations are visible for all design units only if all files are compiled together. In this case an object declared in the **$unit** scope will be common for all compiled modules. However, in case of separate compilation, separate **$unit** scope is created for each file and declaration from the **$unit** scope of another file is not visible. This may lead to compilation error when there is a reference to a non-declared object. Also if this nonexistent object is a variable an implicit net can be created and no error occurs. As a result there will be different variables with the same name in different modules. All this leads to strict dependency of the files compilation order. Moreover, such description is non-synthesizable. So the global variables should not be declared in the **$unit** scope of the RTL description. Tasks and functions should be defined as automatic and placed into named packages instead of compilation-unit scope. Such approach helps to avoid errors when files are compiled separately, and makes the source code easier to debug and maintain.

Level Rule

Checker Behaviour

The checker scans compilation-unit scopes:

- if the **$unit** compilation scope contains declarations of global variables, static tasks/functions => violation.

Example - Static task in compilation-unit scope

Description

In the following example the "my_task" task (static by default) is declared in the compilation-unit scope => violation.

Sample Code

`TOP.sv`:


```
 1:  task my_task (input logic [9:0] data_in,
 2:    output logic data_out);
 3:    int temp;
 4:    // other declarations and statement
 5:  endtask
 6:
 7:  module top ( input clk, reset
 8:    // other port declarations
 9:  );
10:    // statements
11:  endmodule
```

The "my_task" task is declared in the compilation-unit scope.

How to Fix

Use package for shared declaration.

Fixed Code

`package.sv`:

```
 1:  package declarations;
 2:
 3:    task my_task (input logic [10:0] data_in,
 4:      output logic data_out);
 5:      int temp;
 6:      // other declarations and statements
 7:    endtask
 8:
 9:    // other declarations
10:
11:  endpackage
```

`TOP.sv`:

```
 1:
 2:  module top ( input clk, reset
 3:    // ports list
 4:  );
 5:    import declarations::my_task;
 6:
 7:    // other statements
 8:
 9:  endmodule
```


ALDEC_SV.1.2.7

Rule Name

Prefer declaring user-defined types in named packages instead of '$unit' scope.

Base Rule

ALINT_SV.1.4.2.9

Phase

Parse

Message

The "{DeclarationName}" {DeclarationType} is declared in the compilation-unit scope.

Problem Description

User-defined types declarations made in the **$unit** scope are synthesizable but they are not actually global. Such declarations are visible for all design units only if all files are compiled together. However, in case of separate compilation, separate **$unit** scope is created for each file and declaration from the **$unit** scope of another file is not visible. This may lead to compilation error when there is a reference to a non-declared type and, as a result, to strict dependency of the files compilation order. Therefore, it is recommended to define types in named packages instead of compilation-unit scope. Such approach helps to avoid errors when files are compiled separately, and makes the source code easier to debug and maintain.

Level Recommendation 1

Checker Behaviour

The checker scans compilation-unit scopes:

- if the **$unit** compilation scope contains declarations of user-defined types => violation.

Example - User-defined type in compilation-unit scope

Description

In the following example the "data_packet_t" type is declared in the compilation-unit scope => violation.

Sample Code

`top.sv`:

```
1:  typedef struct {int a, b;
2:    logic [1:0] c;
3:    logic d;
4:  } data_packet_t;
5:
6:  module top ( input clk, reset
7:    // other port declarations
```



```
 8:  );
 9:    //statements
10:  endmodule
```

The "data_packet_t" type is declared in the compilation-unit scope.

How to Fix

Use package for shared declaration.

Fixed Code

`package.sv`:

```
 1:  package declaration;
 2:
 3:    typedef struct {int a, b;
 4:      logic [1:0] c;
 5:      logic d;
 6:    } data_packet_t;
 7:
 8:    // other declarations
 9:
10:  endpackage
```

`top.sv`:

```
 1:
 2:  module top ( input clk, reset
 3:    // other port declarations
 4:  );
 5:
 6:    import declaration::data_packet_t;
 7:    // other statements
 8:
 9:  endmodule
```


ALDEC_SV.1.2.8

Rule Name

Prefer declaring automatic tasks/functions in named packages instead of '$unit'.

Base Rule

ALINT_SV.1.4.2.9

Phase

Parse

Message

The "{DeclarationName}" {DeclarationType} is declared in the compilation-unit scope.

Problem Description

Declarations of automatic tasks and functions in the compilation-unit scope are synthesizable, but such description should be avoided to prevent potential design errors when files are compiled separately. It is recommended to use named packages to define automatic tasks and functions in order to avoid problems with source code maintain and debug.

Level Recommendation 1

Checker Behaviour

The checker scans compilation-unit scopes:

- if the **$unit** compilation scope contains declaration of automatic task/function => violation.

Example - Automatic task in compilation-unit scope

Description

In the following example the "my_task" automatic task is declared in the compilation-unit scope => violation.

Sample Code

`top.sv`:

```
 1:  task automatic my_task (input logic [10:0] data_in,
 2:    output logic data_out);
 3:    int temp;
 4:    // other declarations and statement
 5:  endtask
 6:
 7:  module top ( input clk, reset
 8:    // other port declarations
 9:  );
10:    // statements
```



```
11:  endmodule
```

The "my_task" task is declared in the compilation-unit scope.

How to Fix

Use package for shared declaration.

Fixed Code

`package.sv`:

```
1:  package declaration;
2:
3:    task automatic my_task (input logic [10:0] data_in,
4:      output logic data_out);
5:      int temp;
6:      // other declarations and statement
7:    endtask
8:
9:  endpackage
```

`top.sv`:

```
1:
2:  module top ( input clk, reset
3:    // other port declarations
4:  );
5:
6:    import declaration::my_task;
7:    // statements
8:
9:  endmodule
```


ALDEC_SV.1.2.9

Rule Name

Avoid using global parameters within the '$unit' scope.

Base Rule

ALINT_VLOG.1.4.2.1

Phase

Parse

Message-1

The "{FileName}" file contains global parameter declaration(s) mixed with design unit declaration(s). Define global parameters in a separate file.

Message-2

The same "{ParameterName}" {ParameterType} is declared multiple times in the compilation-unit scope.

The "{ParameterName}" {ParameterType} is declared here.

Problem Description

Objects declared in the **$unit** scope are not actually global. Such declarations are visible for all design units only if all files are compiled together. In this case an object declared in the **$unit** scope will be common for all compiled modules. However, in case of separate compilation, separate **$unit** scope is created for each file and declaration from the **$unit** scope of another file is not visible. This may lead to compilation error when there is a reference to a non-declared type and, as a result, to strict dependency of the files compilation order. Therefore, it is recommended to declare global parameters in named packages instead of compilation-unit scope. Such approach helps to avoid errors when files are compiled separately, and makes the source code easier to debug and maintain.

Level Recommendation 1

Checker Behaviour

The checker detects parameter and **localparam** declarations all over the project:

- if a source file contains declaration of parameter or **localparam** in the **$unit** compilation scope and design unit declaration(s) => violation (message-1 for file);
- if the **$unit** compilation scope contains multiple declarations of parameter or **localparam** with the same name, type, data type and value => violation (message-2 for project + details for each parameter declaration).

Note
Checker analyzes all include files:

- if a parameter or **localparam** is included in different constructs from a single include file => no violation;

- if a parameter or **localparam** is included in different constructs from different include files => violation;
- if a parameter or **localparam** is declared in one construct and included from an include file in another construct => violation.

Example 1 - Global parameter and module in single file

Description

In the following example the file contains declaration of a parameter in the global scope and a module declaration => violation.

Sample Code

`top.sv`:

```
1:  parameter SIZE = 8;
2:
3:  module TOP (
4:    // ports list
5:  );
6:    // statements
7:  endmodule
```

The "top.sv" file contains global parameter declaration(s) mixed with design unit declaration(s). Define global parameters in a separate file.

How to Fix

Use package for shared declaration.

Fixed Code

`package.sv`:

```
1:  package declarations;
2:    parameter SIZE = 8;
3:    // other declarations
4:  endpackage
```

`top.sv`:

```
1:  module TOP (
2:    // ports list
3:  );
4:    import declarations::*;
5:    // statements
6:  endmodule
```

Example 4 - Identical parameters in compilation-unit scope

Description

In the following example the same "BUS_WIDTH" parameter is declared globally and inside a task in compilation-unit scope => violation.

Sample Code


```
1:  parameter BUS_WIDTH = 32;
2:
3:  task GEN_INPULSES ();
4:    parameter BUS_WIDTH = 32;
5:    logic [BUS_WIDTH-1:0] gen_data;
6:    // statements
7:  endtask
```

The same "BUS_WIDTH" parameter is declared multiple times in the compilation-unit scope.

The "BUS_WIDTH" parameter is declared here.

The "BUS_WIDTH" parameter is declared here.

How to Fix

Remove the "BUS_WIDTH" parameter declaration from the "GEN_INPULSES" task and place all declarations into a package.

Fixed Code

```
1:  package declarations;
2:
3:    parameter BUS_WIDTH = 32;
4:
5:    task GEN_INPULSES ();
6:      logic [BUS_WIDTH-1:0] gen_data;
7:      // statements
8:    endtask
9:
10:  endpackage
```


ALDEC_SV.1.2.10

Rule Name

Use package import in the module scope only.

Base Rule

ALINT_SV.1.4.2.9

Phase

Parse

Message

The "{DeclarationName}" {DeclarationType} is declared in the compilation-unit scope.

Problem Description

It is recommended to define package import in the module scope instead of compilation-unit one. When the package definition is imported into a module, then package items are visible within this module only. Such approach helps to avoid errors when files are compiled separately and conflicts when objects with same names are declared in multiple packages.

Level Recommendation 2

Checker Behaviour

The checker scans compilation-unit scopes:

- if the **$unit** compilation scope contains the package importing => violation.

Note
Package import declarations are checked in files that use IEEE Std 1800™-2009 SystemVerilog standard only.

Example - Package imported in the compilation-unit scope

Description

In the following example the package import is declared in the compilation-unit scope of the `module_1.sv` file => violation.

If the files will be compiled separately then declaration of the "data_packet_t" type will not be visible in the "module_2" module and its compilation will fail.

Sample Code

`package.sv`:

```
1:  package definitions;
2:    typedef struct {int a, b;
```



```
3:      logic [1:0] c;
4:      logic d;
5:    } data_packet_t;
6:  endpackage
```

module_1.sv:

```
1:  import definitions::*;
2:
3:  module module_1 ( input data_packet_t data
4:    // other port declarations
5:  );
6:    // statements
7:  endmodule
```

module_2.sv:

```
1:  module module_2 ( input clk, reset
2:    // other port declarations
3:  );
4:    data_packet_t pack;
5:    // statements
6:  endmodule
```

The package import is declared in the compilation-unit scope.

How to Fix

Import package in each module header to make the type declaration visible within the modules scopes (including port declarations).

Fixed Code

package.sv:

```
1:  package definitions;
2:    typedef struct {int a, b;
3:      logic [1:0] c;
4:      logic d;
5:    } data_packet_t;
6:  endpackage
```

module_1.sv:

```
1:  module module_1 import definitions::*; (
2:    input data_packet_t data
3:    // other port declarations
4:  );
5:    // statements
6:  endmodule
```

module_2.sv:

```
1:  module module_2 import definitions::*; (
2:    input clk, reset
3:    // other port declarations
4:  );
5:    data_packet_t pack;
6:    // statements
7:  endmodule
```


Section 1.3 - Modules

ALDEC_SV.1.3.1

Rule Name

Use ANSI C-styled module declaration.

Base Rule

ALINT_SV.1.2.4.11

Phase

Parse

Message

The "{ConstructName}" {ConstructType} is declared with non-ANSI header.

Problem Description

In order to reduce the length of the source code, and improve its readability and understandability, the ANSI style should be used for describing modules. It means that port names, directions, data types and bit width should be combined in the C-style module header.

Also it is recommended to use the ANSI style for subprograms description.

Level Recommendation 2

Checker Behaviour

The checker verifies constructs from CONSTRUCTS_TO_CHECK list:

- if non-ANSI header is used in construct declaration => violation.

Default Configuration

The CONSTRUCTS_TO_CHECK parameter defines the list of constructs that should have ANSI header. Possible values are: 'MODULE', 'FUNCTION', 'TASK'.

CONSTRUCTS_TO_CHECK = (MODULE)

Example 1 - Module declaration with non-ANSI header

Description

In the following example the "RAM" module is declared with non-ANSI header => violation.

Sample Code


```
 1:  module RAM (
 2:    CLK,
 3:    WE,
 4:    ADDR,
 5:    DATA_IN,
 6:    DATA_OUT
 7:  );
 8:    input        CLK;
 9:    input        WE;
10:    input  [4:0] ADDR;
11:    input  [3:0] DATA_IN;
12:    output [3:0] DATA_OUT;
13:
14:    reg    [3:0] ram [31:0];
15:
16:    always @(posedge CLK) begin
17:      if (WE)
18:        ram[ADDR] <= DATA_IN;
19:    end
20:
21:    assign DATA_OUT = ram[ADDR];
22:
23:  endmodule
```

The "RAM" module is declared with non-ANSI header.

How to Fix

Rewrite the "RAM" module declaration using ANSI port declaration.

Fixed Code

```
 1:  module RAM (
 2:    input        CLK,
 3:    input        WE,
 4:    input  [4:0] ADDR,
 5:    input  [3:0] DATA_IN,
 6:    output [3:0] DATA_OUT
 7:   );
 8:    reg    [3:0] ram [31:0];
 9:
10:    always @(posedge CLK) begin
11:      if (WE)
12:        ram[ADDR] <= DATA_IN;
13:    end
14:
15:    assign DATA_OUT = ram[ADDR];
16:
17:  endmodule
```

Example 2 - User-defined configuration

Description

In the following example the "mux" function is declared with non-ANSI header => violation.

The "TOP" module is declared with ANSI header => no violation.

ConfigurationCONSTRUCTS_TO_CHECK = "('MODULE','FUNCTION')"Sample Code


```
 1:  module TOP (
 2:    input [1:0] SEL,
 3:    input       D0,
 4:    input       D1,
 5:    input       D2,
 6:    input       D3,
 7:    output      DO
 8:  );
 9:
10:    function mux;
11:      input [3:0] i;
12:      input [1:0] s;
13:
14:        mux = i[s];
15:
16:    endfunction
17:
18:    assign DO = mux( {D0, D1, D2, D3}, SEL );
19:
20:  endmodule
```

The "mux" function is declared with non-ANSI header.

How to Fix

Rewrite the "mux" function declaration with usage of ANSI header declaration.

Fixed Code

```
 1:  module TOP (
 2:    input [1:0] SEL,
 3:    input       D0,
 4:    input       D1,
 5:    input       D2,
 6:    input       D3,
 7:    output      DO
 8:  );
 9:
10:    function mux(
11:      input [3:0] i,
12:      input [1:0] s
13:    );
14:
15:        mux = i[s];
16:
17:    endfunction
18:
19:    assign DO = mux( {D0, D1, D2, D3}, SEL );
20:
21:  endmodule
```


ALDEC_SV.1.3.2

Rule Name

Reference module ports are non-synthesizable.

Base Rule

ALINT_VLOG.1.2.4.2

Phase

Elaboration

Message-1

The "{RefPortList}" module port(s) is of {PortMode} mode.

Message-2

The "{ModportName}" interface 'modport' with "{RefPortList}" {PortModesList} port(s) is used in synthesizable description.

The 'modport' is used here.

Problem Description

It is not recommended to pass references to variables through module ports, since such description is not synthesizable. The **ref** ports should be used for abstract modeling levels only.

Level Rule

Checker Behaviour

The checker verifies synthesizable description:

- if a module has **ref** port => violation (message-1);
- if an interface **modport** has **ref** port and the interface is used in a module port declaration or module instance => violation (message-2).

Example 1 - Module with port of reference type

Description

In the following example the "TOP" module has "sh1" and "sh2" **ref** ports => violation.

Sample Code

```
1:  module TOP (input clk,
2:    ref wire sh1, sh2
3:    // other port declarations
4:  );
```



```
5:    // statements
6:  endmodule
```

The "sh1", "sh2" module port(s) is of ref mode.

How to Fix

Remove "sh1" and "sh2" non-synthesizable ports or change their direction.

Fixed Code

```
1:  module TOP (input clk
2:    // other port declarations
3:  );
4:    // statements
5:  endmodule
```

Example 2 - Interface with 'ref' ports

Description

In the following example "mp" modport of "IF1" interface has "sh" **ref** port and it is used in the "TOP" module port declaration => violation.

Sample Code

```
1:  interface IF1;
2:    wire sh;
3:    // other declarations
4:    modport mp (ref sh /* other modport definitions */ );
5:  endinterface
6:
7:  module TOP (IF1.mp i
8:    // other port declarations
9:  );
10:   // statements
11:  endmodule
```

The "mp" interface 'modport' with "sh" ref port(s) is used in synthesizable description.

The 'modport' is used here.

How to Fix

Change the "sh" port direction to other than **ref**.

Fixed Code

```
1:  interface IF1;
2:    wire sh;
3:    // other interface declarations
4:    modport mp (inout sh /* other modport definitions */ );
5:  endinterface
6:
7:  module TOP (IF1.mp i
8:    // other port declarations
9:  );
10:   // statements
11:  endmodule
```


ALDEC_SV.1.3.3

Rule Name

Place nested modules into separate files included to the main module.

Base Rule

ALINT_SV.1.4.1.8

Phase

Parse

Message

The "{NestedModuleName}" nested module is declared inside the "{UpperModuleName}" module. Place nested modules into separate files included to the main module.

Problem Description

For big and complicated projects it is recommended to avoid nesting of a lower-level module description into the declarations of the upper-level module. In such case the best practice is to keep each module declared in a separate file. Such approach makes the debugging more efficient, as it helps to understand project structure composed from multiple parts.

Level Recommendation 3

Checker Behaviour

The checker scans module (macromodule) design units:

- if a module contains nested module declarations => violation.

Example - Module with nested modules

Description

In the following example the upper-level module "TOP" contains nested module "SUB" that also contains one more nested module "SUB2" => violation.

Sample Code

`TOP.sv`:

```
1:  module TOP (output R);
2:
3:    module SUB (output R);
4:
5:      module SUB2 (output R);
6:        //module statements
7:      endmodule
8:
```



```
 9:      SUB2 SUB2_1 (.*);
10:     //module statements
11:
12:   endmodule
13:
14:   SUB SUB_1 (.*);
15:   //module statements
16:
17: endmodule
```

The "SUB" nested module is declared inside the "TOP" module. Place nested modules into separate files included to the main module.

The "SUB2" nested module is declared inside the "SUB" module. Place nested modules into separate files included to the main module.

How to Fix

Describe nested modules in separate files and add corresponding **`include** directives to the upper modules.

Fixed Code

`TOP.sv`:

```
1:  module TOP (output R);
2:
3:    `include "SUB.sv"
4:
5:    SUB SUB_1 (.*);
6:    //module statements
7:
8:  endmodule
```

`SUB.sv`:

```
1:  module SUB (output R);
2:
3:    `include "SUB2.sv"
4:
5:    SUB2 SUB2_1 (.*);
6:    //module statements
7:
8:  endmodule
```

`SUB2.sv`:

```
1:  module SUB2 (output R);
2:    //module statements
3:  endmodule
```


ALDEC_SV.1.3.4

Rule Name

Do not use the (.*) connection.

Base Rule

ALINT_SV.1.2.5.5

Phase

Parse

Message

The wildcard connection is detected in the "{InstanceName}" instantiation statement.

Problem Description

It is recommended to define a named port connection instead of a wildcard one (.*) for any connection between a port and a net with the same name. Using the (.*) connection worsens the readability of the source code, makes it difficult to understand, and can easily lead to mistakes.

Level Rule

Checker Behaviour

The checker scans instantiation statements:

- if wildcard connection (.*) is detected in the instantiation statement => violation.

Example - Wildcard connection

Description

In the following example the "UDFF" instantiation contains wildcard port connection => violation.

Sample Code

```
 1:  module DFF(
 2:    input            CLK,
 3:    input      [2:0] D,
 4:    output reg [2:0] Q
 5:  );
 6:
 7:    always @( posedge CLK )
 8:      Q <= D;
 9:
10:  endmodule
11:
12:
13:  module TOP (
14:    input        CLK,
```



```
15:    input  [2:0] D,
16:    output [2:0] Q
17:  );
18:
19:    DFF UDFF (.*);
20:
21:  endmodule
```

The wildcard connection is detected in the "UDFF" instantiation statement.

How to Fix

Use named connection instead of wildcard.

Fixed Code

```
 1:  module DFF(
 2:    input                 CLK,
 3:    input      [2:0] D,
 4:    output reg [2:0] Q
 5:  );
 6:
 7:    always @( posedge CLK )
 8:      Q <= D;
 9:
10:  endmodule
11:
12:
13:  module TOP (
14:    input         CLK,
15:    input  [2:0] D,
16:    output [2:0] Q
17:  );
18:
19:    DFF UDFF (
20:      .CLK(CLK),
21:      .D(D),
22:      .Q(Q)
23:    );
24:
25:  endmodule
```


ALDEC_SV.1.3.5

Rule Name

Use .name to connect a net to a port with the same name.

Base Rule

ALINT_SV.1.2.5.7

Phase

Parse

Message

The "{PortName}" port and a connected object in the "{InstanceName}" instantiation statement have the same names. Use .name to connect a net to a port with the same name.

Problem Description

In SystemVerilog, in cases when the port name and size matches the connecting net or bus name and size it is recommended to use **.name** implicit port connections. Such approach allows to simplify the code readability, for example, in cases when there are lots of clocks and resets.

Level Recommendation 2

Checker Behaviour

The checker scans named port connections in the instantiation statements:

- if a port and object connected to this port have the same names written with the same letter case => violation.

Example - Named port connection

Description

In the following example the "UDFF" instantiation contains named port connections where same names are used both for ports and connected objects => violation.

Sample Code

```
 1:  module DFF(
 2:    input CLK,
 3:    input [2:0] D,
 4:    output reg [2:0] Q
 5:  );
 6:
 7:    always @( posedge CLK )
 8:      Q <= D;
 9:
10:  endmodule
```



```
11:
12:
13:  module TOP (
14:    input CLK,
15:    input [2:0] D,
16:    output [2:0] Q
17:  );
18:
19:    DFF UDFF (
20:      .CLK(CLK),
21:      .D(D),
22:      .Q(Q)
23:    );
24:
25:  endmodule
```

The "CLK" port and a connected object in the "UDFF" instantiation statement have the same names. Use .name to connect a net to a port with the same name.

The "D" port and a connected object in the "UDFF" instantiation statement have the same names. Use .name to connect a net to a port with the same name.

The "Q" port and a connected object in the "UDFF" instantiation statement have the same names. Use .name to connect a net to a port with the same name.

How to Fix

Use SystemVerilog **.name** connection instead of named port connection.

Fixed Code

```
 1:  module DFF(
 2:    input CLK,
 3:    input [2:0] D,
 4:    output reg [2:0] Q
 5:  );
 6:
 7:    always @( posedge CLK )
 8:      Q <= D;
 9:
10:  endmodule
11:
12:
13:  module TOP (
14:    input CLK,
15:    input [2:0] D,
16:    output [2:0] Q
17:  );
18:
19:    DFF UDFF (
20:      .CLK,
21:      .D,
22:      .Q
23:    );
24:
25:  endmodule
```


Section 1.4 - Interfaces

ALDEC_SV.1.4.1

Rule Name

When similar groups of ports are passed through port maps of different modules, extract interfaces.

Base Rule

ALINT_SV.1.4.3.3

Phase

Elaboration

Message

The "{ObjectName}" {ObjectType} is connected to {NumberOfInstances} instance(s) of {NumberOfModules} module(s).

The "{ObjectName}" {ObjectType} is connected to the "{PortName}" port of "{InstanceName}" instance.

Problem Description

It is recommended to encapsulate similar groups of ports passed through port maps of different modules into an interface. Such approach helps to avoid the redundant declarations of modules, and so prevents inconvenience.

Level Recommendation 1

Checker Behaviour

The checker verifies ports, nets and variables of the design units depending on the hierarchy levels:

- if there is port/net/variable on the hierarchy level defined by **LEVEL** parameter and this port/net/variable is connected to the number of instantiated modules (the number is set by the **MIN_NUMBER_OF_INSTANTIATED_MODULES** parameter) => violation (main message for port/net/variable declaration + detail for each connection to instance.

Default Configuration

The MIN_NUMBER_OF_INSTANTIATED_MODULES parameter defines the minimum number of instantiated modules to which the checked object should be connected.

MIN_NUMBER_OF_INSTANTIATED_MODULES = 2

The LEVEL parameter defines the hierarchy level at which the objects connected to instances should be checked. Possible values are: 'TOP', 'PART', 'MODULE'.

LEVEL = (TOP, PART)

Example 1 - Some objects connected to several instances

Description

In the following example "CLK" and "RESET" ports, and also "DATA", "ADDRESS", "M_READ", "M_WRITE" wires are connected to "3" instances of "3" different modules => violation.

ConfigurationMIN_NUMBER_OF_INSTANTIATED_MODULES = 3Sample Code

```
 1:  module IG (input wire CLK,
 2:    input wire RESET,
 3:    input wire [7:0] DATA,
 4:    input wire [2:0] ADDRESS,
 5:    input wire M_READ,
 6:    input wire M_WRITE,
 7:    output reg T_RES,
 8:    output reg P_RES);
 9:
10:    // module statements
11:
12:  endmodule
13:
14:  module RAM ( input wire CLK,
15:    input wire RESET,
16:    input wire [7:0] DATA,
17:    input wire [2:0] ADDRESS,
18:    input wire MEM_RD,
19:    input wire MEM_WR,
20:    output reg W_DATA_READY,
21:    output reg R_DATA_READY);
22:
23:    // module statements
24:
25:  endmodule
26:
27:  module PROCESSOR ( input wire CLK,
28:    input wire RESET,
29:    input wire WR_D,
30:    input wire RD_D,
31:    inout wire [7:0] DATA,
32:    output reg [2:0] ADDRESS,
33:    output reg MEM_READ,
34:    output reg MEM_WRITE);
35:
36:    // module statements
37:
38:  endmodule
39:
40:  module TOP ( input wire CLK,
41:    input wire RESET,
42:    output reg T_RES, P_RES);
43:
44:    wire [7:0] DATA;
45:    wire [2:0] ADDRESS;
46:    wire M_READ, M_WRITE;
47:    wire WR_D, RD_D;
48:
49:
50:    IG U_IG1 (.*);
51:
52:    PROCESSOR U_PROC (
```



```
53:       .*, .MEM_READ(M_READ),
54:       .MEM_WRITE(M_WRITE));
55:
56:     RAM U_RAM (
57:       .*, .MEM_RD(M_READ),
58:       .MEM_WR(M_WRITE),
59:       .W_DATA_READY(WR_D),
60:       .R_DATA_READY(RD_D));
61:
62:   endmodule
```

The "CLK" port is connected to 3 instance(s) of 3 module(s).

The "CLK" port is connected to the "CLK" port of "U_IG1" instance.

The "CLK" port is connected to the "CLK" port of "U_PROC" instance.

The "CLK" port is connected to the "CLK" port of "U_RAM" instance.

The "RESET" port is connected to 3 instance(s) of 3 module(s).

The "RESET" port is connected to the "RESET" port of "U_IG1" instance.

The "RESET" port is connected to the "RESET" port of "U_PROC" instance.

The "RESET" port is connected to the "RESET" port of "U_RAM" instance.

The "DATA" net is connected to 3 instance(s) of 3 module(s).

The "DATA" net is connected to the "DATA" port of "U_IG1" instance.

The "DATA" net is connected to the "DATA" port of "U_PROC" instance.

The "DATA" net is connected to the "DATA" port of "U_RAM" instance.

The "ADDRESS" net is connected to 3 instance(s) of 3 module(s).

The "ADDRESS" net is connected to the "ADDRESS" port of "U_IG1" instance.

The "ADDRESS" net is connected to the "ADDRESS" port of "U_PROC" instance.

The "ADDRESS" net is connected to the "ADDRESS" port of "U_RAM" instance.

The "M_READ" net is connected to 3 instance(s) of 3 module(s).

The "M_READ" net is connected to the "M_READ" port of "U_IG1" instance.

The "M_READ" net is connected to the "MEM_READ" port of "U_PROC" instance.

The "M_READ" net is connected to the "MEM_RD" port of "U_RAM" instance.

The "M_WRITE" net is connected to 3 instance(s) of 3 module(s).

The "M_WRITE" net is connected to the "M_WRITE" port of "U_IG1" instance.

The "M_WRITE" net is connected to the "MEM_WRITE" port of "U_PROC" instance.

The "M_WRITE" net is connected to the "MEM_WR" port of "U_RAM" instance.

How to Fix

Create and use interface with corresponding objects declaration.

Fixed Code

```
 1:  interface BUS_IF (input logic clock, reset);
 2:    wire [7:0] DATA;
 3:    wire [2:0] ADDRESS;
 4:    wire M_READ, M_WRITE;
 5:
 6:    modport master (
 7:      input clock, reset, DATA, ADDRESS, M_READ, M_WRITE);
 8:
 9:    modport slave (
10:      input clock, reset,
11:      output ADDRESS, M_READ, M_WRITE,
12:      inout DATA);
13:
14:  endinterface
15:
16:  module IG (BUS_IF.master BUS,
17:    output reg T_RES,
18:    output reg P_RES);
19:
20:    // module statements
21:
22:  endmodule
23:
24:  module RAM ( BUS_IF.master BUS,
25:    output reg W_DATA_READY,
26:    output reg R_DATA_READY);
27:
28:    // module statements
29:
30:  endmodule
31:
32:  module PROCESSOR (
33:    BUS_IF.slave BUS,
34:    input wire WR_D,
35:    input wire RD_D);
36:
37:    // module statements
38:
39:  endmodule
40:
41:  module TOP (
42:    input wire CLK,
43:    input wire RESET,
44:    output reg T_RES, P_RES);
45:
46:    wire WR_D, RD_D;
47:
48:    BUS_IF BUS ( // instance of an interface
49:      .clock(CLK),
50:      .reset(RESET));
51:
```



```
52:    IG U_IG1 (.*, .BUS(BUS));
53:
54:    PROCESSOR U_PROC (.*, .BUS(BUS));
55:
56:    RAM U_RAM (
57:      .*,
58:      .BUS(BUS),
59:      .W_DATA_READY(WR_D),
60:      .R_DATA_READY(RD_D));
61:
62:  endmodule
```

Example 2 - Objects are connected to several instances of same module

Description

In the following example "FF" module instantiated three times so number of instantiated modules is "1" => no violation.

Fixed Code

```
 1:  module FF ( input wire CLK,
 2:    input wire DATA,
 3:    output reg Q);
 4:
 5:    // module statements
 6:
 7:  endmodule
 8:
 9:  module TOP ( input wire CLK,
10:    input wire DATA,
11:    output reg Q_OUT);
12:
13:    reg Q_tmp_1, Q_tmp_2;
14:
15:    FF U_FF_1 (DATA, CLK, Q_tmp_1);
16:
17:    FF U_FF_2 (Q_tmp_1, CLK, Q_tmp_2);
18:
19:    FF U_FF_3 (Q_tmp_2, CLK, Q_OUT);
20:
21:  endmodule
```


ALDEC_SV.1.4.3

Rule Name

Interface methods must be automatic to be synthesizable.

Base Rule

ALINT_SV.2.5.1.6

Phase

Elaboration

Message

The "{SubprogramName}" static {SubprogramType} is declared in the "{UnitName}" {UnitType}.

Problem Description

It is recommended to declare tasks or functions defined in an interface as automatic, and they should not contain static variables. When an automatic task/function is referenced in a module, the unique copy of such task/function is visible only for this module and there is no sharing of task/function items. Such approach prevents mismatches between pre-synthesis and post-synthesis behavior, when several modules reference same task/function.

Level Rule

Checker Behaviour

Checker scans design interfaces:

- if a non-automatic subprogram is detected and it is called in synthesizable description => violation.

Example - Subprogram inside interface

Description

In the following example the "RdWr" task declared in the "intf_bus" interface is static and it is referenced in a module => violation.

Sample Code

`interf_declarations.sv`:

```
1:  interface intf_bus (input logic clock, resetN);
2:    logic input_a;
3:    logic input_b;
4:    logic result;
5:
6:    task static RdWr (input int a, b, output int c);
7:      // statements
8:    endtask
9:
```



```
10:    modport master (import RdWr, input clock, resetN, output result);
11:
12:    // other statements and declarations
13:
14:  endinterface
```

modules.sv:

```
 1:  module top (input logic clock, resetN, output logic result);
 2:    intf_bus bus (clock, resetN);
 3:    master_unit i1 (bus.master);
 4:  endmodule
 5:
 6:  module master_unit (interface i);
 7:    int a_int, b_int, c_int;
 8:    always @(posedge i.clock)
 9:    begin
10:      i.RdWr(a_int, b_int, c_int);
11:    end
12:  endmodule
```

The "RdWr" static task is declared in the "intf_bus" interface.

How to Fix

Change the task type from static to automatic.

Fixed Code

interf_declarations.sv:

```
 1:  interface intf_bus (input logic clock, resetN);
 2:    logic input_a;
 3:    logic input_b;
 4:    logic result;
 5:
 6:    task automatic RdWr (input int a, b, output int c);
 7:      // statements
 8:    endtask
 9:
10:    modport master (import RdWr, input clock, resetN, output result);
11:
12:    // other statements and declarations
13:
14:  endinterface
```

modules.sv:

```
 1:  module top (input logic clock, resetN, output logic result);
 2:    intf_bus bus (clock, resetN);
 3:    master_unit i1 (bus.master);
 4:  endmodule
 5:
 6:  module master_unit (interface i);
 7:    int a_int, b_int, c_int;
 8:    always @(posedge i.clock)
 9:    begin
10:      i.RdWr(a_int, b_int, c_int);
11:    end
12:  endmodule
```


ALDEC_SV.1.4.4

Rule Name

Interface methods should not define static variables to be synthesizable.

Base Rule

ALINT_SV.2.5.1.7

Phase

Elaboration

Message

The "{SubprogramName}" automatic {SubprogramType} declared in the "{UnitName}" {UnitType} contains static variable(s).

The "{VariableName}" static variable is declared here.

Problem Description

When static variables are defined in the automatic task/function inside interface, they can be changed from outside the body of the subprogram. Moreover, initialization of static variables is not synthesizable. In order to avoid mismatches between pre-synthesis and post-synthesis behavior, only automatic variables should be used within automatic subprograms inside interfaces.

Level Rule

Checker Behaviour

The checker verifies automatic subprograms inside interfaces:

- if a static variable is detected inside an automatic subprogram that is called in synthesizable description => violation.

Example - Automatic task with static variable inside interface

Description

In the following example the "RdWr" automatic task contains static variable => violation.

Sample Code

`interf_declarations.sv`:

```
1:  interface intf_bus (input logic clock, resetN);
2:    logic input_a;
3:    logic input_b;
4:    logic result;
5:
6:    task automatic RdWr (input int a, b, output int c);
```



```
 7:      static logic temp_r;
 8:      // statements
 9:    endtask
10:
11:    modport master (import RdWr, input clock, resetN, output result);
12:
13:    // other statements and declarations
14:
15:  endinterface
```

modules.sv:

```
 1:  module top (input logic clock, resetN, output logic result);
 2:    intf_bus bus (clock, resetN);
 3:    master_unit i1 (bus.master);
 4:  endmodule
 5:
 6:  module master_unit (interface i);
 7:    int a_int, b_int, c_int;
 8:    always @(posedge i.clock)
 9:    begin
10:      i.RdWr(a_int, b_int, c_int);
11:    end
12:  endmodule
```

The "RdWr" automatic task declared in the "intf_bus" interface contains static variable(s).

The "temp_r" static variable is declared here.

How to Fix

Change lifetime of the variable from static to automatic.

Fixed Code

interf_declarations.sv:

```
 1:  interface intf_bus (input logic clock, resetN);
 2:    logic input_a;
 3:    logic input_b;
 4:    logic result;
 5:
 6:    task automatic RdWr (input int a, b, output int c);
 7:      logic temp_r;
 8:      // statements
 9:    endtask
10:
11:    modport master (import RdWr, input clock, resetN, output result);
12:
13:    // other statements and declarations
14:
15:  endinterface
```

modules.sv:

```
 1:  module top (input logic clock, resetN, output logic result);
 2:    intf_bus bus (clock, resetN);
 3:    master_unit i1 (bus.master);
 4:  endmodule
 5:
 6:  module master_unit (interface i);
 7:    int a_int, b_int, c_int;
```



```
 8:    always @(posedge i.clock)
 9:    begin
10:      i.RdWr(a_int, b_int, c_int);
11:    end
12:  endmodule
```


ALDEC_SV.1.4.5

Rule Name

Exported methods are not synthesizable.

Base Rule

ALINT_SV.2.5.1.9

Phase

Elaboration

Message-1

A subprogram is used like exported method here. Do not use extern or exported subprograms.

Message-2

A subprogram is used like external method here. Do not use extern or exported subprograms.

Problem Description

Do not use exported tasks and functions within an interface. Such description is not synthesizable, and should be used for abstract models only.

Level Rule

Checker Behaviour

The checker scans interfaces:

- if a subprogram is exported through the **modport** and then it is called in a synthesizable description => violation (message-1);
- if a subprogram is declared like external subprogram (**forkjoin**) => violation (message-2).

Example 1 - Using of exported statement

Description

In the following example the "t" task is exported through the "s" modport => violation.

Sample Code

```
1:  interface f;
2:    reg s0, m0;
3:    modport m (output m0, import task t(output t0));
4:    modport s (output s0, export task t(output t0));
5:  endinterface
6:
7:  module master (f.m i);
8:    always @(*)
```



```
 9:      i.t(i.mO);
10:  endmodule
11:
12:  module slave (f.s i);
13:    task i.t (output tO);
14:      //task statements
15:    endtask
16:
17:    always @(*)
18:      i.t(i.sO);
19:  endmodule
```

A subprogram is used like exported method here. Do not use extern or exported subprograms.

How to Fix

Declare the subprogram inside the interface and then use the **import** statement instead of the **export**.

Fixed Code

```
 1:  interface f;
 2:    task t (output tO);
 3:      //statements
 4:    endtask
 5:
 6:    reg sO, mO;
 7:    modport m (output mO, import task t(output tO));
 8:    modport s (output sO, import task t(output tO));
 9:  endinterface
10:
11:  module master (f.m i);
12:    always @(*)
13:      i.t(i.mO);
14:  endmodule
15:
16:  module slave (f.s i);
17:    always @(*)
18:      i.t(i.sO);
19:  endmodule
```

Example 2 - Using of external subprogram

Description

In the following example the "Func" is used like external function => violation.

Sample Code

```
 1:  function Func(input tO);
 2:    //statements
 3:  endfunction
 4:
 5:  interface f;
 6:    extern function Func(input tO);
 7:
 8:    reg mI, mO;
 9:    modport m (input mI, output mO, import function Func(input tO));
10:  endinterface
11:
12:  module master (f.m i);
13:    always @(*)
```



```
14:      i.mO = i.Func(i.mI);
15:  endmodule
```

A subprogram is used like external method here. Do not use extern or exported subprograms.

How to Fix

Declare the function inside the interface.

Fixed Code

```
 1:  interface f;
 2:    function Func(input tO);
 3:      //statements
 4:    endfunction
 5:
 6:    reg mI, mO;
 7:    modport m (input mI, output mO, import function Func(input tO));
 8:  endinterface
 9:
10:  module master (f.m i);
11:    always @(*)
12:      i.mO = i.Func(i.mI);
13:  endmodule
```


ALDEC_SV.1.4.6

Rule Name

Define modport in interface.

Base Rule

ALINT_SV.1.4.3.10

Phase

Parse

Message

The "{InterfaceName}" interface does not have modports.

Problem Description

It is recommended to declare **modport**(s) inside the interface in order to define the direction of signals at the destination module. Otherwise, the direction of the signals will be set to default. As s result, confusion in identifying the signals direction can occur during the maintain.

Level Rule

Checker Behaviour

The checker verifies **interface** design units:

- if an interface does not have any **modport** definitions => violation.

Example - Interface without modports

Description

In the following example no modports are defined in the "BUS" interface => violation.

Sample Code

```
 1:  interface BUS;
 2:    wire arg1;
 3:    // other signal declarations
 4:  endinterface
 5:
 6:  module master (BUS BUS
 7:    // other port declarations
 8:  );
 9:    // statements
10:  endmodule
```

The "BUS" interface does not have modports.

How to Fix

Define **modport** inside the interface.

Fixed Code

```
 1:  interface BUS;
 2:    wire arg1;
 3:    // other signal declarations
 4:    modport master (input arg1 /*other modport definitions*/);
 5:  endinterface
 6:
 7:  module master (BUS.master BUS
 8:    // other port declarations
 9:  );
10:    // statements
11:  endmodule
```


ALDEC_SV.1.4.7

Rule Name

Specify modport when calling the interface.

Base Rule

ALINT_SV.1.4.3.11

Phase

Elaboration

Message

The modport of "{InterfaceName}" interface is not specified in "{PortName}" port declaration in the "{ModuleName}" module header.

Problem Description

When calling the interface corresponding modport of this interface should be specified in the module header. Since modports define different views of the interface signals, each module sees necessary view on its interface port. Otherwise, all nets in the interface are assumed to have a inout direction, and all variables in the interface are assumed to be of **ref** type that can lead to further problems, since a **ref** port passes values by reference, rather than by copy, and this allows the module to access the variable in the interface, rather than a copy of the variable.

Level Rule

Checker Behaviour

The checker verifies module headers in synthesizable description:

- if there is a port with interface type declared in a module header and a modport is not specified => violation.

Example - Modport is not specified

Description

In the following example modport is not specified in the module header for port with interface type => violation.

Sample Code

```
1:  interface BUS;
2:    wire arg1;
3:    // other signal declarations
4:    modport master (input arg1 /*other modport definitions*/);
5:  endinterface
6:
7:  module master (BUS DATA_BUS
8:    // other port declarations
```



```
 9:  );
10:    // statements
11:  endmodule
```

The modport of "BUS" interface is not specified in "DATA_BUS" port declaration in the "master" module header.

How to Fix

Specify modport for BUS interface in module header.

Fixed Code

```
 1:  interface BUS;
 2:    wire arg1;
 3:    // other signal declarations
 4:    modport master (input arg1 /*other modport definitions*/);
 5:  endinterface
 6:
 7:  module master (BUS.master DATA_BUS
 8:    // other port declarations
 9:  );
10:    // statements
11:  endmodule
```


Section 1.5 - Time units

ALDEC_SV.1.5.1

Rule Name

Prefer module-level 'timeunit'/'timeprecision' declarations to file-level preprocessor directives.

Base Rule

ALINT_SV.1.2.7.12

Phase

Parse

Message

The 'timescale' directive is specified. Use module-level 'timeunit'/'timeprecision' declarations instead.

Problem Description

It is recommended to specify the time unit and precision information within a module using **timeunit/timeprecision** declarations instead of **`timescale** directive. Since the **`timescale** directive affects all modules and files that follow the directive declaration, it cause dependencies of the files compilation order.

Level Recommendation 2

Checker Behaviour

The checker verifies preprocessor directives:

- if **`timescale** directive is specified => violation.

Example - Use of `timescale directive

Description

In the following example the **`timescale** directive is used => violation.

Sample Code

```
 1:  `timescale 1ns/1ps
 2:
 3:  module SUB (
 4:    // port declarations
 5:  );
 6:    // statements
 7:  endmodule
 8:
 9:  module TOP (
10:    // port declarations
11:  );
```



```
12:    // statements
13:  endmodule
```

The 'timescale' directive is specified. Use module-level 'timeunit'/'timeprecision' declarations instead.

How to Fix

Add **timeunit/timeprecision** declarations to each design unit instead of the preprocessor directive.

Fixed Code

```
 1:  module SUB (
 2:    // port declarations
 3:  );
 4:    timeunit 1ns;
 5:    timeprecision 1ps;
 6:
 7:    // statements
 8:
 9:  endmodule
10:
11:  module TOP (
12:    // port declarations
13:  );
14:    timeunit 1ns;
15:    timeprecision 1ps;
16:
17:    // statements
18:
19:  endmodule
```


ALDEC_SV.1.5.3

Rule Name

Prefer 'timeunit'/'timeprecision' declarations in design units instead of '$unit'.

Base Rule

ALINT_SV.1.4.2.9

Phase

Parse

Message

The "{DeclarationName}" {DeclarationType} is specified in the compilation-unit scope.

Problem Description

When time units are specified in the **$unit** compilation scope, then they are applied for all design units only if all files are compiled together. Otherwise, in case of separate compilation, separate **$unit** scope is created for each file and declaration from the **$unit** scope of another file is not visible. Therefore, it is recommended to specify time units directly in the design unit before any declaration and statements to avoid ambiguity in case of separate compilation.

Level Recommendation 2

Checker Behaviour

The checker scans compilation-unit scopes:

- if the **$unit** compilation scope contains time unit declarations => violation.

Example - Time unit in compilation-unit scope

Description

In the following example the **timeunit** statement is used in the compilation-unit scope => violation.

Sample Code

```
 1:  timeunit 1ns;
 2:
 3:  module DFF #(
 4:    parameter DELAY = 5
 5:  )(
 6:    input      CLK,
 7:    input      DATA,
 8:    output reg Q
 9:  );
10:
11:    always @(posedge CLK)
12:      Q = #DELAY DATA;
```



```
13:
14:  endmodule
```

The time units declaration is specified in the compilation-unit scope.

How to Fix

Specify **timeunit** keyword just after the module declaration.

Fixed Code

```
 1:  module DFF #(
 2:    parameter DELAY = 5
 3:  )(
 4:    input      CLK,
 5:    input      DATA,
 6:    output reg Q
 7:  );
 8:    timeunit 1ns;
 9:
10:    always @(posedge CLK)
11:      Q = #DELAY DATA;
12:
13:  endmodule
```


ALDEC_SV.1.5.4

Rule Name

Avoid using default time unit and precision.

Base Rule

ALINT_SV.5.1.2.6

Phase

Parse

Message

The delay has default time unit or precision.

Problem Description

It is recommended to define custom values for the time units and time precision. When default values are used instead of custom ones, then radically different simulation results can occur when simulating the same design on different simulators.

Level Recommendation 1

Checker Behaviour

The checker verifies delays:

- if there is no **timescale** directive in the same or the upper scope, or there is no both **timeunit** and **timeprecision** declarations for a current design unit where a delay is specified => violation.

Example - Timeunits and timeprecisions are not specified

Description

In the following example the "TOP_TB" module has several delays without **timeunit** specifications => violations.

Sample Code

```
 1:  module TOP_TB (  );
 2:    parameter P = 1.5;
 3:    reg clk;
 4:    reg rst;
 5:
 6:    reg [31:0] data_in;
 7:    reg [31:0] data_out;
 8:
 9:    initial
10:    begin
11:      rst = 1'b0;
12:      #10 rst = 1'b1;
13:    end
```



```
14:
15:    always #5
16:    begin
17:      clk = ~clk;
18:    end
19:
20:    always
21:      #(P-1: P: P+1) data_in = data_out;
22:  endmodule
```

The delay has default time unit or precision.

The delay has default time unit or precision.

The delay has default time unit or precision.

How to Fix

Declare **timeunit** and **timeprecision** in the current unit.

Fixed Code

```
 1:  module TOP_TB (  );
 2:    timeunit 1ns;
 3:    timeprecision 100ps;
 4:
 5:    parameter P = 1.5;
 6:    reg clk;
 7:    reg rst;
 8:
 9:    reg [31:0] data_in;
10:    reg [31:0] data_out;
11:
12:    initial
13:    begin
14:      rst = 1'b0;
15:      #10 rst = 1'b1;
16:    end
17:
18:    always #5
19:    begin
20:      clk = ~clk;
21:    end
22:
23:    always
24:      #(P-1: P: P+1) data_in = data_out;
25:  endmodule
```


Chapter 2 - Data Declarations and Uses

Section 2.1 - 2-state Types

ALDEC_SV.2.1.1

Rule Name

In synthesizable code use 2-state types like 'int' only as a counter in 'for' loop.

Base Rule

ALINT_VLOG.1.3.1.1

Phase

Parse

Message-1

The "{ObjectName}" {ObjectType} is of '{TypeName}' type. Do not use the following types in RTL description: {ForbiddenTypesList}.

Message-2

The "{TypeName}" {ObjectKind} is based on the '{BaseTypeName}' data type. Do not use the following types in RTL description: {ForbiddenTypesList}.

Problem Description

It is not recommended to use 2-state data types for synthesizable RTL hardware models. Such types store 0 and 1 values only, but **Z** and **X** values are required at higher levels of modeling to describe unconnected or tri-state logic and uncover design errors. Therefore, 4-state data types should be used in RTL description to avoid mismatches between RTL and gate level simulation. However, 2-state integer types are allowed for loop variables.

Level Rule

Checker Behaviour

The checker scans synthesizable description:

- if 2-state type (**bit**, **byte**, **shortint**, **int**, **longint**) is used for variable, net, port, structure members, subprogram internal variables, or function return type => violation (message-1);
- if user-defined type (except of structure, union, class type, and interface based type) is based on 2-state type => violation (message-2).

Example 1 - Using forbidden type of variable

Description

In the following example the "SIZE" variable is of **int** data type => violation.

Sample Code

```
1:  module TOP (
2:    input CLK
3:    // other declarations
4:  );
5:    int SIZE = 4;
6:    // statements
7:  endmodule
```

The "SIZE" variable is of 'int' type. Do not use the following types in RTL description: "bit", "byte", "shortint", "int", "longint".

How to Fix

Use **integer** type for "SIZE" parameter.

Fixed Code

```
1:  module TOP (
2:    input CLK
3:    // other declarations
4:  );
5:    integer SIZE = 4;
6:    // statements
7:  endmodule
```

Example 2 - User-defined type is based on forbidden one

Description

In the following example the "opcode" type is based on the forbidden data type => violation.

Sample Code

```
1:  module TOP (
2:    input CLK
3:    // other declarations
4:  );
5:
6:    typedef byte opcode;
7:
8:    // statements
9:
10:  endmodule
```

The "opcode" data type is based on the 'byte' data type. Do not use the following types in RTL description: "bit", "byte", "shortint", "int", "longint".

How to Fix

Use **logic** type for "opcode" type.

Fixed Code

```
1:  module TOP (
2:    input CLK
```



```
 3:    // other declarations
 4:  );
 5:
 6:    typedef logic [7:0] opcode;
 7:
 8:    // statements
 9:
10:  endmodule
```

Example 3 - Loop variable

Description

In the following example the "i" loop variable is of **int** type => no violation.

Fixed Code

```
 1:  module REVERSE (
 2:    input         clk,
 3:    input         rst,
 4:    input         enable,
 5:    output reg [31:0] count
 6:  );
 7:
 8:    reg    [31:0] temp;
 9:
10:    always @ (posedge clk or posedge rst)
11:      if (rst)
12:        count = 'b0;
13:      else
14:        for (int i = 0; i < 16; i++)
15:        begin
16:          temp        = count[i];
17:          count[i]    = count[31-i];
18:          count[31-i] = temp;
19:        end
20:
21:  endmodule
```


ALDEC_SV.2.1.2

Rule Name

Avoid using 2-state type variables in 'casex'/'casez' statements due to undefined simulation semantics.

Base Rule

ALINT_SV.1.3.1.10

Phase

Elaboration

Message

The "{ObjectName}" {ObjectDataType} variable is used in 'casex/casez' statement. Avoid using 2-state variables in casex/casez statements due to undefined simulation semantics.

Problem Description

When assigning 'x' or 'z' value to a 2-state variable, then a variable will result in a value of 0 according to LRM. However, there is no standardized requirements how the 2-state variable should be processed within **casex**/**casez** statements and, as a result, the behavior can vary for different tools. Therefore, it is recommended to use 4-state type variables in **casex**/**casez** statements to ensure appropriate simulation results.

Level Recommendation 1

Checker Behaviour

The checker verifies **casex**/**casez** statements:

- if a 2-state variable is detected in a case item or in a case selection expression => violation.

Example - 2-state variable in selection expression

Description

In the following example the "SEL" bit variable is used in a selection expression => violation.

Sample Code

```
 1:  module mux (
 2:    input  bit [1:0]  SEL,
 3:    input  logic      DIN1,
 4:    input  logic      DIN2,
 5:    output logic      DOUT
 6:  );
 7:
 8:    always @(SEL, DIN1, DIN2) begin
 9:      casex (SEL)
10:        2'b0X:   DOUT = DIN1;
11:        2'b1X:   DOUT = DIN2;
12:        default: DOUT = 1'bX;
```



```
13:      endcase
14:    end
15:
16:  endmodule
```

The "SEL" bit variable is used in 'casex/casez' statement. Avoid using 2-state variables in casex/casez statements due to undefined simulation semantics.

How to Fix

Use 4-state variable instead of 2-state one.

Fixed Code

```
 1:  module mux (
 2:    input  logic [1:0] SEL,
 3:    input  logic       DIN1,
 4:    input  logic       DIN2,
 5:    output logic       DOUT
 6:  );
 7:
 8:    always @(SEL, DIN1, DIN2) begin
 9:      casex (SEL)
10:        2'b0X:   DOUT = DIN1;
11:        2'b1X:   DOUT = DIN2;
12:        default: DOUT = 1'bX;
13:      endcase
14:    end
15:
16:  endmodule
```


ALDEC_SV.2.1.3

Rule Name

Be careful with X-assignments to 2-state type variables.

Base Rule

ALINT_SV.5.2.3.9

Phase

Elaboration

Message

Unknown value is assigned to "{ObjectName}" {ObjectDataType} {ObjectType}.

Problem Description

When using 2-state types an unitialized state cannot be described, because 2-state types do not store unknown values. According to LRM unknown or high-impedance value is converted to zero when assigned to 2-state object. The conversion can lead to incorrect simulation results. Therefore, 4-state type objects should be used to model unknown or unconnected states.

Level Rule

Checker Behaviour

The checker verifies each assignment:

- if an unknown value ('X' or 'Z') is assigned in RHS of assignment to a 2-state type object => violation.

Example - Assignment of 'x' to bit variable

Description

In the following example the "DOUT" bit variable is assigned with unknown value => violation.

Sample Code

```
 1:  module mux (
 2:    input  bit [1:0] SEL,
 3:    input  bit       DIN1,
 4:    input  bit       DIN2,
 5:    input  bit       DIN3,
 6:    input  bit       DIN4,
 7:    output bit       DOUT
 8:  );
 9:
10:    always @(SEL, DIN1, DIN2) begin
11:      case (SEL)
12:        2'b00:   DOUT = DIN1;
13:        2'b01:   DOUT = DIN2;
```



```
14:        2'b10:   DOUT = DIN2;
15:        2'b11:   DOUT = DIN2;
16:        default: DOUT = 1'bX;
17:      endcase
18:    end
19:
20:  endmodule
```

Unknown value is assigned to "DOUT" bit port.

How to Fix

Use a 4-state variable instead of a 2-state variable.

Fixed Code

```
 1:  module mux (
 2:    input  logic [1:0] SEL,
 3:    input  logic       DIN1,
 4:    input  logic       DIN2,
 5:    input  logic       DIN3,
 6:    input  logic       DIN4,
 7:    output logic       DOUT
 8:  );
 9:
10:    always @(SEL, DIN1, DIN2) begin
11:      case (SEL)
12:        2'b00:   DOUT = DIN1;
13:        2'b01:   DOUT = DIN2;
14:        2'b10:   DOUT = DIN2;
15:        2'b11:   DOUT = DIN2;
16:        default: DOUT = 1'bX;
17:      endcase
18:    end
19:
20:  endmodule
```


ALDEC_SV.2.1.4

Rule Name

Reset logic is absolutely necessary for flip-flops based on 2-state type variables to avoid simulation vs synthesis mismatches.

Base Rule

ALINT_SV.3.1.1.10

Phase

Synthesis

Message

The "{ObjectName}" variable of the "{ObjectDataType}" data type is used to infer flip-flop without reset logic.

Problem Description

The 2-state variable can store 0 and 1 values and is initialized by 0 for the simulation. But after the logic synthesis flip-flop will not be initialized by a particular value. It can be represented more accurately using the 4-state variable initialized by X. However, if 2-state variable is used to infer flip-flop the reset logic should be necessarily defined to be able to set the flip-flop to a know initial state and avoid mismatches between simulation and synthesis.

Level Rule

Checker Behaviour

The checker verifies sequential processes in synthesizable description:

- if a flip-flop is inferred for a 2-state variable (**int**, **bit**, **byte**, **shortint**, **longint**) and it is described without asynchronous or synchronous control => violation.

Example - No reset defined for flip-flop

Description

In the following example the "q" bit variable is used to infer a flip-flop, but no asynchronous or synchronous control is specified => violation.

Sample Code

```
1:  module test (
2:    input bit clk,
3:    input bit data,
4:    output bit q
5:  );
6:
7:    always @(posedge clk)
8:      q = data;
```



```
 9:
10:  endmodule
```

The "q" variable of the "bit" data type is used to infer flip-flop without reset logic.

How to Fix

Add asynchronous control for the "q" flip-flop.

Fixed Code

```
 1:  module test (
 2:    input bit clk,
 3:    input bit rst,
 4:    input bit data,
 5:    output bit q
 6:  );
 7:
 8:    always @(posedge clk or posedge rst)
 9:      if(rst)
10:        q = 1'b0;
11:      else
12:        q = data;
13:
14:  endmodule
```


Section 2.2 - Constants

ALDEC_SV.2.2.1

Rule Name

Use a fill value ('1,'0,'z,'x) for vector literals instead of a hard coded value.

Base Rule

ALINT_SV.1.2.2.9

Phase

Parse

Message

The "{ConstantLiteral}" constant literal should be replaced with {FillValue} fill literal.

Problem Description

In case when it is required to assign a vector of any width with all bits set to the same value ('0', '1', 'z', 'x') it is recommended to use a fill value (**'<value>**) instead of a hard-coded value. Such approach allows working with very large vector sizes and simplifies code debugging.

Level Recommendation 3

Checker Behaviour

The checker verifies based constant literals in equality comparisons and assignments:

- if each bit of the vector literal is equal to the same value ("1", "0", "X", or "Z") => violation.

Default Configuration

The MAX_ALLOWED_LENGTH parameter defines the maximum number of characters in a literal that should not be scanned.

MAX_ALLOWED_LENGTH = 1

Example 1 - Wide flip-flop with wide reset

Description

In the following example, the "rst" signal is compared with constant literal that consists of **"1"** characters and the "Q" signal is assigned with literal of **"0"** characters => violations.

Sample Code

```
1:  module TOP ( input clk,
```



```
 2:    input      [3:0] rst, D,
 3:    output reg [3:0] Q
 4:  );
 5:
 6:    always @(posedge clk) begin
 7:      if (rst == 4'b1111)
 8:        Q = 4'b0000;
 9:      else
10:        Q = D;
11:    end
12:
13:  endmodule
```

The "4'b1111" constant literal should be replaced with '1 fill literal.

The "4'b0000" constant literal should be replaced with '0 fill literal.

How to Fix

Replace the literals.

Fixed Code

```
 1:  module TOP ( input clk, rst,
 2:    input      [3:0] D,
 3:    output reg [3:0] Q
 4:  );
 5:
 6:    always @(posedge clk) begin
 7:      if (rst == '1)
 8:        Q = '0;
 9:      else
10:        Q = D;
11:    end
12:
13:  endmodule
```

Example 2 - 'h and 'o based literals

Description

In the following example, each bit of "6'h3F" and "7'o177" literals is equal to "1" value => violations.

Sample Code

```
 1:  module TOP ( input clk,
 2:    input      set,
 3:    input      [5:0] D1,
 4:    input      [6:0] D2,
 5:    output reg [5:0] Q1,
 6:    output reg [6:0] Q2
 7:  );
 8:
 9:    always @(posedge clk) begin
10:      if (set) begin
11:        Q1 = 6'h3F;
12:        Q2 = 7'o177;
13:      end
14:      else begin
15:        Q1 = D1;
16:        Q2 = D2;
```



```
17:      end
18:    end
19:
20:  endmodule
```

The "6'h3F" constant literal should be replaced with '1 fill literal.

The "7'o177" constant literal should be replaced with '1 fill literal.

How to Fix

Replace the literals.

Fixed Code

```
 1:  module TOP ( input clk,
 2:    input      set,
 3:    input      [5:0] D1,
 4:    input      [6:0] D2,
 5:    output reg [5:0] Q1,
 6:    output reg [6:0] Q2
 7:  );
 8:
 9:    always @(posedge clk) begin
10:      if (set) begin
11:        Q1 = '1;
12:        Q2 = '1;
13:      end
14:      else begin
15:        Q1 = D1;
16:        Q2 = D2;
17:      end
18:    end
19:
20:  endmodule
```

Example 3 - Unsized literals

Description

In the following example, "'bXX" unsized constant literal extended with "X" characters is used => violation.

Sample Code

```
1:  module TOP ( input en,
2:    input      [7:0] D,
3:    output reg [7:0] Q
4:  );
5:
6:    assign Q = en ? D : 'bXX;
7:
8:  endmodule
```

The "'bXX" constant literal should be replaced with 'X fill literal.

How to Fix

Replace the literal.

Fixed Code


```
1:  module TOP ( input en,
2:    input      [7:0] D,
3:    output reg [7:0] Q
4:  );
5:
6:    assign Q = en ? D : 'X;
7:
8:  endmodule
```

Example 4 - Custom configuration

Description

In the following example, each bit of the "16'hzzzz" constant is equal to "Z" value, but the MAX_ALLOWED_LENGTH parameter is set to 4 characters => no violation.

ConfigurationMAX_ALLOWED_LENGTH = 4Fixed Code

```
1:  module TOP ( input en,
2:    input      [15:0] D,
3:    output reg [15:0] Q
4:  );
5:
6:    assign Q = en ? D : 16'hzzzz;
7:
8:  endmodule
```


ALDEC_SV.2.2.2

Rule Name

Avoid assign of non-constants to run-time constant in synthesizable description.

Base Rule

ALINT_SV.5.2.1.12

Phase

Elaboration

Message

The non-constant value is assigned to "{ObjectName}" const variable that is used in synthesizable description.

The const variable is used here.

Problem Description

A **const** variable in SystemVerilog is a variable that can be initialized in its declaration only, and never assigned another value again. Most synthesis tools ignore these initializations, so they appear as variables that have no drivers. There is also a problem with the initialization order that can produce unexpected results. A **const** constant is treated in the same way as a Verilog **localparam** constant during the synthesis. The main difference in synthesizable code is that a **const** constant can also be declared in automatic functions. It is recommended to use **const** constants in verification description only.

Level Recommendation 1

Checker Behaviour

The checker verifies synthesizable description:

- if a non-constant object is assigned to a **const** variable and then this variable is used in synthesizable description => violation (main message for variable declaration + detail for each usage inside synthesizable description).

Example - Const variable

Description

In the following example the "in" non-constant object is assigned to "c" **const** variable which is used inside **always** construct => violation.

Sample Code

```
1:  module top ( input in, clk, rst
2:    //other ports
3:  );
4:
```



```
 5:    const logic c = in;
 6:    integer d;
 7:
 8:    //other declarations
 9:
10:    always begin
11:      d = c;
12:    end
13:
14:    //other statements
15:
16:  endmodule
```

The non-constant value is assigned to "c" const variable that is used in synthesizable description.

The const variable is used here.

How to Fix

Use net instead of **const** variable.

Fixed Code

```
 1:  module top ( input in, clk, rst
 2:    //other ports
 3:  );
 4:
 5:    wire logic c = in;
 6:    integer d;
 7:
 8:    //other declarations
 9:
10:    always begin
11:      d = c;
12:    end
13:
14:    //other statements
15:
16:  endmodule
```


Section 2.3 - User-defined Types

ALDEC_SV.2.3.1

Rule Name

Do not declare identical 'typedef' in multiple contexts.

Base Rule

ALINT_SV.1.4.2.8

Phase

Elaboration

Message

The "{TypeName}" typedef has duplicating declarations of the same type.

The "{TypeName}" typedef with the same data type is declared here.

Problem Description

Identical data types should not be described using duplicated **typedef** declarations in different contexts. To avoid multiple-definition mistakes, single **typedef** declaration should be defined in the named package, and imported within modules each time it is required.

Level Recommendation 1

Checker Behaviour

Checker verifies type declarations all over the project:

- if multiple **typedef** with identical data type and the same or different names are declared in the project => violation.

Example 1 - Identical types declared in different places

Description

In the following example the "data_packet_t" **typedef** is declared in different contexts with the same data type => violation.

Sample Code

`file1.sv`:

```
1:  module MODULE_1 (
2:    // ports list
3:  );
4:    typedef struct {
```



```
 5:      int a;
 6:      int b;
 7:      logic [1:0] f;
 8:      logic c;
 9:    } data_packet_t;
10:
11:    // other statements
12:  endmodule
```

file2.sv:

```
 1:  module MODULE_2 (
 2:    // ports list
 3:  );
 4:    typedef struct {
 5:      int a, b;
 6:      logic [1:0] c;
 7:      logic d;
 8:    } data_packet_t;
 9:
10:    // other statements
11:  endmodule
```

The "data_packet_t" typedef has duplicating declarations of the same type.

The "data_packet_t" typedef with the same data type is declared here.

How to Fix

Declare the "data_packet_t" type in package and import it in each module.

Fixed Code

package.sv:

```
 1:  package proj_types;
 2:
 3:    typedef struct {
 4:      int a, b;
 5:      logic [1:0] c;
 6:      logic d;
 7:    } data_packet_t;
 8:
 9:  endpackage
```

file1.sv:

```
 1:  module MODULE_1 (
 2:    // ports list
 3:  );
 4:    import proj_types::*;
 5:
 6:    // other statements
 7:  endmodule
```

file2.sv:

```
 1:  module MODULE_2 (
 2:    // ports list
 3:  );
 4:    import proj_types::*;
 5:
 6:    // other statements
```



```
7:  endmodule
```

Example 2 - Different types with the same name

Description

In the following example the "my_type" **typedef** is declared in different contexts, but the data types differ => no violation.

Fixed Code

`file1.sv`:

```
1:  module MODULE_1 (
2:    // ports list
3:  );
4:    typedef logic [4:0] my_type;
5:
6:    // other statements
7:  endmodule
```

`file2.sv`:

```
1:  module MODULE_2 (
2:    // ports list
3:  );
4:    typedef logic [0:4] my_type;
5:
6:    // other statements
7:  endmodule
```

Example 3 - Identical types with different names

Description

In the following example the "data_packet_t" and "struct_packet_t" types are identical and declared inside the "proj_types" package => violation.

Sample Code

`package.sv`:

```
1:  package proj_types;
2:
3:    typedef struct packed {
4:      int a, b;
5:      logic [1:0] c;
6:      logic d;
7:    } data_packet_t;
8:
9:    typedef struct packed {
10:     int a, b;
11:     logic [1:0] d;
12:     logic c;
13:   } packet_t;
14:
15: endpackage
```

`top.sv`:


```
1:  module TOP (
2:    // ports list
3:  );
4:    import proj_types::*;
5:
6:    // other statements
7:  endmodule
```

The "data_packet_t" typedef has duplicating declarations of the same type.

The "packet_t" typedef with the same data type is declared here.

How to Fix

Remove unnecessary declaration.

Fixed Code

`package.sv`:

```
1:  package proj_types;
2:
3:    typedef struct {
4:      int a, b;
5:      logic [1:0] c;
6:      logic d;
7:    } data_packet_t;
8:
9:  endpackage
```

`top.sv`:

```
1:  module TOP (
2:    // ports list
3:  );
4:    import proj_types::*;
5:
6:    // other statements
7:  endmodule
```

Example 4 - Identical basic type of struct members

Description

In the following example **typedef** "A" and "B" has the same data type => violation.

Note, that **typedef** "data_t" and "d_t" has the same data type, but types of struct/union members are checked only by names => no violation.

Sample Code

`package.sv`:

```
1:  package proj_types;
2:
3:    typedef logic A;
4:    typedef logic B;
5:
6:    typedef struct { A a;
7:    } data_t;
```



```
 8:
 9:    typedef struct { B a;
10:    } d_t;
11:
12:  endpackage
```

top.sv:

```
 1:  module TOP (
 2:    // ports list
 3:  );
 4:    import proj_types::*;
 5:
 6:    // other statements
 7:  endmodule
```

The "A" typedef has duplicating declarations of the same type.

The "B" typedef with the same data type is declared here.

How to Fix

Change one of the identical declarations.

Fixed Code

package.sv:

```
 1:  package proj_types;
 2:
 3:    typedef logic A;
 4:    typedef int B;
 5:
 6:    typedef struct { A a;
 7:    } data_t;
 8:
 9:    typedef struct { B a;
10:    } d_t;
11:
12:  endpackage
```

top.sv:

```
 1:  module TOP (
 2:    // ports list
 3:  );
 4:    import proj_types::*;
 5:
 6:    // other statements
 7:  endmodule
```


ALDEC_SV.2.3.2

Rule Name

Never declare two different typedefs with the same name in different contexts.

Base Rule

ALINT_SV.1.4.2.11

Phase

Elaboration

Message

The same "{TypeName}" typedef is declared in different contexts.

The "{TypeName}" typedef is declared in the "{ObjectName}" {ObjectType}.

Problem Description

Different data types should not be declared with the same name in different contexts. To avoid multiple-definition mistakes, single **typedef** declaration should be defined in the named package, and imported within modules each time it is required.

Level Rule

Checker Behaviour

Checker verifies type declarations all over the project:

- if there are multiple **typedefs** with same names, but different data types declared in the project => violation (main message for a project + details for each type declaration).

Example 1 - Different struct types with same names

Description

In the following example there are two "data_packet_t" **typedefs** declared in different contexts with same names, but different data type of the "c" member => violation.

Sample Code

`file1.sv`:

```
1:  module MODULE_1 (
2:    // ports list
3:  );
4:    typedef struct {
5:      int a;
6:      int b;
7:      bit [1:0] c;
```



```
 8:      logic d;
 9:    } my_struct_t;
10:
11:    // other statements
12:  endmodule
```

`file2.sv`:

```
 1:  module MODULE_2 (
 2:    // ports list
 3:  );
 4:    typedef struct {
 5:      int a;
 6:      int b;
 7:      logic [1:0] c;
 8:      logic d;
 9:    } my_struct_t;
10:
11:    // other statements
12:  endmodule
```

The same "my_struct_t" typedef is declared in different contexts.

The "my_struct_t" typedef is declared in the "MODULE_1" module.

The "my_struct_t" typedef is declared in the "MODULE_2" module.

How to Fix

Rename the second **typedef** to "my_second_struct_t".

Fixed Code

`file1.sv`:

```
 1:  module MODULE_1 (
 2:    // ports list
 3:  );
 4:    typedef struct {
 5:      int a;
 6:      int b;
 7:      bit [1:0] c;
 8:      logic d;
 9:    } my_struct_t;
10:
11:    // other statements
12:  endmodule
```

`file2.sv`:

```
 1:  module MODULE_2 (
 2:    // ports list
 3:  );
 4:    typedef struct {
 5:      int a;
 6:      int b;
 7:      logic [1:0] c;
 8:      logic d;
 9:    } my_second_struct_t;
10:
11:    // other statements
12:  endmodule
```


Example 2 - Types with different vector bounds

Description

In the following example the "my_type" **typedef** is declared in different contexts, but the data types differ => violation.

Sample Code

`file1.sv`:

```
1:  module MODULE_1 (
2:    // ports list
3:  );
4:    typedef logic [4:0] my_type;
5:
6:    // other statements
7:  endmodule
```

`file2.sv`:

```
1:  module MODULE_2 (
2:    // ports list
3:  );
4:    typedef logic [0:4] my_type;
5:
6:    // other statements
7:  endmodule
```

The same "my_type" typedef is declared in different contexts.

The "my_type" typedef is declared in the "MODULE_1" module.

The "my_type" typedef is declared in the "MODULE_2" module.

How to Fix

Rename typedefs to "logic_4_downto_0_t" and "logic_0_to_4_t".

Fixed Code

`file1.sv`:

```
1:  module MODULE_1 (
2:    // ports list
3:  );
4:    typedef logic [4:0] logic_4_downto_0_t;
5:
6:    // other statements
7:  endmodule
```

`file2.sv`:

```
1:  module MODULE_2 (
2:    // ports list
3:  );
4:    typedef logic [0:4] logic_0_to_4_t;
5:
6:    // other statements
```



```
7:  endmodule
```

Example 3 - Identical types with different names

Description

In the following example the "data_packet_t" and "struct_packet_t" types are identical and are declared inside the "proj_types" package => no violation.

Fixed Code

`package.sv`:

```
1:  package proj_types;
2:
3:    typedef struct packed {
4:      int a, b;
5:      logic [1:0] c;
6:      logic d;
7:    } data_packet_t;
8:
9:    typedef struct packed {
10:     int a, b;
11:     logic [1:0] d;
12:     logic c;
13:   } struct_packet_t;
14:
15: endpackage
```

`top.sv`:

```
1:  module TOP (
2:    // ports list
3:  );
4:    import proj_types::*;
5:
6:    // other statements
7:  endmodule
```

Example 4 - Identical basic type of struct members

Description

In the following example **typedef** "A" and "B" has the same data type => violation.

Note, that **typedef** "data_t" and "d_t" has the same data type, but types of struct/union members are checked only by names => no violation.

Sample Code

`file1.sv`:

```
1:  module MODULE_1 (
2:    // ports list
3:  );
4:    typedef logic A;
5:    typedef struct { A a;
6:    } data_t;
7:
```



```
8:    // other statements
9:  endmodule
```

file2.sv:

```
1:  module MODULE_2 (
2:    // ports list
3:  );
4:    typedef logic B;
5:    typedef struct { B a;
6:    } data_t;
7:
8:    // other statements
9:  endmodule
```

The same "data_t" typedef is declared in different contexts.

The "data_t" typedef is declared in the "MODULE_1" module.

The "data_t" typedef is declared in the "MODULE_2" module.

How to Fix

Rename "B" typedef to "A".

Fixed Code

file1.sv:

```
1:  module MODULE_1 (
2:    // ports list
3:  );
4:    typedef logic A;
5:    typedef struct { A a;
6:    } data_t;
7:
8:    // other statements
9:  endmodule
```

file2.sv:

```
1:  module MODULE_2 (
2:    // ports list
3:  );
4:    typedef logic A;
5:    typedef struct { A a;
6:    } data_t;
7:
8:    // other statements
9:  endmodule
```


ALDEC_SV.2.3.3

Rule Name

Follow naming conventions for user-defined types.

Base Rule

ALINT_SV.1.1.2.8

Phase

Elaboration

Message-1

The "{ObjectName}" {ObjectType} name matches forbidden regular expression "{REGEXP}" that should not be used for type names.

Message-2

The "{ObjectName}" {ObjectType} name does not match required regular expression "{REGEXP}" that should be used for type names.

Problem Description

It is recommended to use special suffixes in names of user-defined types to enforce clear understanding of object types and scopes. This approach improves code readability.

Level Recommendation 2

Checker Behaviour

The checker verifies **typedef** names depending on the REGEXP.MODE parameter settings:

- if REGEXP.MODE=ALLOW (default) and a name does not match required regular expression => violation (message-2);
- if REGEXP.MODE=DENY and a name matches forbidden regular expression => violation (message-1).

Note
If an empty value is specified for a parameter, then the regular expression-based check is not performed.

Default Configuration

The TYPEDEF parameter defines naming conventions settings for typedefs names that not based on struct, union, enumeration, class type.

The REGEXP parameter field defines regular expression for checked names.

TYPEDEF.REGEXP = '.+_t$'

The TYPEDEF_STRUCT parameter defines naming conventions settings for typedefs names that based on struct type.

The REGEXP parameter field defines regular expression for checked names.

TYPEDEF_STRUCT.REGEXP = '.+_t$'

The TYPEDEF_UNION parameter defines naming conventions settings for typedefs names that based on union type.

The REGEXP parameter field defines regular expression for checked names.

TYPEDEF_UNION.REGEXP = '.+_t$'

The TYPEDEF_ENUM parameter defines naming conventions settings for typedefs names that based on enumeration type.

The REGEXP parameter field defines regular expression for checked names.

TYPEDEF_ENUM.REGEXP = '.+_t$'

The TYPEDEF_CLASS parameter defines naming conventions settings for typedefs names that based on class type.

The REGEXP parameter field defines regular expression for checked names.

TYPEDEF_CLASS.REGEXP = '.+_t$'

The REGEXP parameter defines settings for all regexp-based checks.

The CASE_SENSITIVE parameter filed defines whether regexp-based checks are case sensitive.

REGEXP.CASE_SENSITIVE = NO

The MODE parameter field defines the mode of regexp-based checking. Possible values are: 'allow', 'deny'.

REGEXP.MODE = ALLOW

Example 1 - Incorrect typedef name

Description

The type name does not match the forbidden pattern => violation

Sample Code

```
1:  module TOP ( // ports declaration
2:  );
3:    typedef bit [15:0] word;
4:
5:    //other declarations
6:  endmodule
```


The "word" bit typedef name does not match required regular expression ".+_t$" that should be used for type names.

How to Fix

Add the '_t' suffix to the "word" **typedef**.

Fixed Code

```
1:  module TOP ( // ports declaration
2:  );
3:    typedef bit [15:0] word_t;
4:
5:    //other declarations
6:  endmodule
```

Example 2 - Custom configuration

Description

The type name matches the forbidden pattern => violation

Configuration

REGEXP.MODE = deny

TYPEDEF_STRUCT.REGEXP = '.+_u$'

Sample Code

```
1:  module TOP ( // ports declaration
2:  );
3:    typedef struct {
4:      logic [3:0] a, b;
5:      logic [7:0] opcode;
6:      logic [5:0] address;
7:    } struct_u;
8:
9:    //other declarations
10:  endmodule
```

The "struct_u" struct typedef name matches forbidden regular expression ".+_u$" that should not be used for type names.

How to Fix

Rename "struct_u" to "struct_t".

Fixed Code

```
1:  module TOP ( // ports declaration
2:  );
3:    typedef struct {
4:      logic [3:0] a, b;
5:      logic [7:0] opcode;
6:      logic [5:0] address;
7:    } struct_t;
8:
```



```
 9:    //other declarations
10:  endmodule
```


ALDEC_SV.2.3.4

Rule Name

Use enumeration methods (like .next, .prev) instead of arithmetic operations upon the enumeration variables or literals.

Base Rule

ALINT_SV.1.3.5.12

Phase

Elaboration

Message

The "{EnumVariable}" enumeration variable is detected in expression with arithmetical operation. Use enumeration methods instead of arithmetic operations.

Problem Description

It is recommended to use enumeration methods instead of arithmetic operations to return the value of the next or previous member in the enumerated list. Such approach allows to ensure that the value is always within the list of legal values for the enumerated type.

Level Recommendation 1

Checker Behaviour

The checker verifies expressions with enumeration variables:

- if enumeration variable is used as an operand of arithmetical operation => violation.

Example - Arithmetical operation with enum

Description

In the following example the "State" enumeration variable is incremented by arithmetic operation => violation.

Sample Code

```
 1:  module TOP (
 2:    input logic reset, clock
 3:    //other ports
 4:  );
 5:
 6:    typedef enum logic [2:0] {
 7:      RED = 3'b001,
 8:      YELLOW = 3'b010,
 9:      GREEN = 3'b100} states_t;
10:
11:    states_t State, NextState;
12:
```



```
13:    //other statements
14:
15:    assign NextState = states_t'(State + 1);
16:
17:  endmodule
```

The "State" enumeration variable is detected in expression with arithmetical operation. Use enumeration methods instead of arithmetic operations.

How to Fix

Use the **.next()** method to increment an enumeration variable to the next label in the enumerated list.

Fixed Code

```
 1:  module TOP (
 2:    input logic reset, clock
 3:    //other ports
 4:  );
 5:
 6:    typedef enum logic [2:0] {
 7:      RED = 3'b001,
 8:      YELLOW = 3'b010,
 9:      GREEN = 3'b100} states_t;
10:
11:    states_t State, NextState;
12:
13:    //other statements
14:
15:    always
16:      NextState = State.next(1);
17:
18:  endmodule
```


ALDEC_SV.2.3.5

Rule Name

Do not use class types in RTL description.

Base Rule

ALINT_SV.2.3.1.9

Phase

Elaboration

Message

The "{ObjectName}" variable of "{ClassName}" class is used in synthesizable description.

The class variable is used here.

Problem Description

The variables of class type should be used only in verification code, but not in RTL code. Class type is not supported by the synthesis tools.

Level Rule

Checker Behaviour

The checker verifies synthesizable description:

- if a variable of a class type is used => violation.

Example - Class variable

Description

In the following example the "c" class variable is declared in the "top" module and used inside **always** construct => violation.

Sample Code

```
 1:  class Packet;
 2:    integer value;
 3:
 4:    task set (int i);
 5:      value = i + 1;
 6:    endtask
 7:
 8:    function integer get();
 9:      get = value * value;
10:    endfunction
11:
12:  endclass
```



```
13:
14:  module top ( input clk, rst
15:    //other ports
16:  );
17:
18:    Packet c;
19:    integer d;
20:
21:    //other declarations
22:
23:    always begin
24:      c = new;
25:      c.set(55);
26:      d = c.get();
27:    end
28:
29:    //other statements
30:
31:  endmodule
```

The "c" variable of "Packet" class is used in synthesizable description.

The class variable is used here.

The class variable is used here.

The class variable is used here.

How to Fix

Use **initial** construct instead of **always**.

Fixed Code

```
 1:  class Packet;
 2:    integer value;
 3:
 4:    task set (int i);
 5:      value = i + 1;
 6:    endtask
 7:
 8:    function integer get();
 9:      get = value * value;
10:    endfunction
11:
12:  endclass
13:
14:  module top ( input clk, rst
15:    //other ports
16:  );
17:
18:    Packet c;
19:    integer d;
20:
21:    //other declarations
22:
23:    initial begin
24:      c = new;
25:      c.set(55);
26:      d = c.get();
27:    end
```



```
28:
29:    //other statements
30:
31:  endmodule
```


ALDEC_SV.2.3.6

Rule Name

Prefer named enumeration types to anonymous enumeration variables.

Base Rule

ALINT_SV.1.4.2.12

Phase

Parse

Message

The signal(s) with an anonymous {Type} type is declared here.

Problem Description

It is recommended to use typed enumerations instead of anonymous ones. Anonymous enumerations declared in different modules, even if with the same names and members, are not the same type of enumeration. As a result, problems can occur when passing enumerations through the design blocks.

Level Recommendation 3

Checker Behaviour

The checker verifies object declarations all over the project:

- if an anonymous enumeration is detected => violation.

Example - Anonymous enumeration type

Description

In the following example the "state" and "nextstate" variables are declared with anonymous enumeration type => violation.

Sample Code

```
1:  module TOP (
2:    // ports list
3:  );
4:    enum {IDLE, START, LOAD, FINISH } state, nextstate;
5:
6:    // other statements
7:
8:  endmodule
```

The signal(s) with an anonymous enum type is declared here.

How to Fix

Use **typedef** to declare user-defined type.

Fixed Code

```
 1:  module TOP (
 2:    // ports list
 3:  );
 4:    typedef enum {IDLE, START, LOAD, FINISH } enum_type;
 5:
 6:    enum_type state;
 7:    enum_type nextstate;
 8:
 9:    // other statements
10:
11:  endmodule
```


ALDEC_SV.2.3.7

Rule Name

Use 'var' keyword to emphasize variable declarations with user-defined types.

Base Rule

ALINT_SV.1.2.3.1

Phase

Parse

Message

The "{ListOfVariableNames}" variable(s) of user-defined type is declared without 'var' keyword.

Problem Description

It is recommended to specify **var** keyword when declaring variables created from user-defined types despite it does not affect variables behaviour in simulation or synthesis. Such description improves code readability and makes it easier to maintain.

Level Recommendation 3

Checker Behaviour

The checker verifies declaration of variables with user-defined type:

- if **var** keyword is not specified in variable(s) declaration => violation.

Example 1 - Variables of user-defined type

Description

In the following example there is no **var** keyword in the declaration of the "data_in" and "data_res" variables => violation.

Sample Code

```
 1:  module TOP (
 2:    // ports list
 3:  );
 4:    typedef struct { logic [2:0] code;
 5:      integer num;
 6:    } op_str_t;
 7:
 8:    op_str_t data_in, data_res;
 9:
10:    // other statements
11:
12:  endmodule
```

The "data_in", "data_res" variable(s) of user-defined type is declared without 'var' keyword.

How to Fix

Add **var** keyword to the variables declaration.

Fixed Code

```
 1:  module TOP (
 2:    // ports list
 3:  );
 4:    typedef struct { logic [2:0] code;
 5:      integer num;
 6:    } op_str_t;
 7:
 8:    var op_str_t data_in, data_res;
 9:
10:    // other statements
11:
12:  endmodule
```

Example 2 - Variable of logic type

Description

In the following example the "p_data" variable is declared without **var** keyword, but only variables of user-defined type are checked => no violation.

Fixed Code

```
1:  module TOP (
2:    // ports list
3:  );
4:
5:    logic [1:0] p_data;
6:
7:    // other statements
8:
9:  endmodule
```


ALDEC_SV.2.3.8

Rule Name

Use enumerated type to define FSM states.

Base Rule

ALINT_SV.3.3.3.5

Phase

Synthesis

Message

The "{ObjectName}" {ObjectType} variable is not of an enumerated type.

Problem Description

It is advisable to define enumerated types with proper dimension and values containing all possible FSM states. Such approach allows the synthesis tool to determine the optimal encoding for the state machine.

Level Recommendation 2

Checker Behaviour

The checker verifies current and next state signals of synthesized FSM(s):

- if the signal is not of an enumerated type => violation.

Example - Vector state signal

Description

In the following example FSM state signals are vectors => violation.

Sample Code

```
 1:  module FSM ( input X,
 2:    input clk,
 3:    input rst,
 4:    output reg Z);
 5:
 6:    // fsm states declaration
 7:    parameter [1:0] S0 = 2'b00,
 8:                    S1 = 2'b01,
 9:                    S2 = 2'b10;
10:
11:    // fsm signal declarations
12:    reg [1:0] pre_state, next_state;
13:
14:    always @(posedge clk or posedge rst)
15:      if (rst)
```



```
16:        pre_state <= S0;
17:      else
18:        pre_state <= next_state;
19:
20:    always @*
21:      begin
22:        next_state = pre_state;
23:        Z = 1'b0;
24:        case (pre_state)
25:          S0:
26:            begin
27:              if (X == 1'b0)
28:                next_state = S2;
29:              else if (X == 1'b1)
30:                next_state = S1;
31:              Z = 1'b1;
32:            end
33:          S1:
34:            begin
35:              if (X == 1'b0)
36:                next_state = S0;
37:              Z = 1'b1;
38:            end
39:          S2:
40:            if (X == 1'b1)
41:              next_state = S1;
42:          default: next_state = S0;
43:        endcase;
44:      end
45:
46:  endmodule
```

The "pre_state" current state variable is not of an enumerated type.

The "next_state" next state variable is not of an enumerated type.

How to Fix

Use enumerated type for the current and next state signals.

Fixed Code

```
 1:  module FSM ( input X,
 2:    input clk,
 3:    input rst,
 4:    output reg Z);
 5:
 6:    //fsm enumerated type declaration
 7:    typedef enum logic [1:0] {S0, S1, S2} fsm_states;
 8:
 9:    //fsm signal declarations
10:    fsm_states next_state, pre_state;
11:
12:    always @(posedge clk or posedge rst)
13:      if (rst)
14:        pre_state <= S0;
15:      else
16:        pre_state <= next_state;
17:
18:    always @*
19:      begin
20:        next_state = pre_state;
```



```
21:        Z = 1'b0;
22:        case (pre_state)
23:          S0:
24:            begin
25:              if (X == 1'b0)
26:                next_state = S2;
27:              else if (X == 1'b1)
28:                next_state = S1;
29:              Z = 1'b1;
30:            end
31:          S1:
32:            begin
33:              if (X == 1'b0)
34:                next_state = S0;
35:              Z = 1'b1;
36:            end
37:          S2:
38:            if (X == 1'b1)
39:              next_state = S1;
40:          default: next_state = S0;
41:        endcase;
42:      end
43:
44:  endmodule
```


ALDEC_SV.2.3.9

Rule Name

Typedefs should be defined in packages only.

Base Rule

ALINT_SV.1.4.2.9

Phase

Parse

Message

The "{DeclarationName}" {DeclarationType} is declared in the forbidden scope.

Problem Description

User-defined types declared within the module/interface are synthesizable but they have the reduced visibility. Therefore, it is recommended to define types in named packages instead of modules/interfaces in order to facilitate re-use.

Level Recommendation 2

Checker Behaviour

The checker scans **module** and **interface** design units:

- if the module/interface contains declarations of user-defined types => violation.

Example - Typedef declaration in module

Description

In the following example the "data_packet_t" type is declared in the "top" module => violation.

Sample Code

`TOP.sv`:

```
 1:  module top ( input clk, reset
 2:    // other port declarations
 3:  );
 4:
 5:    typedef struct {int a, b;
 6:      logic [1:0] c;
 7:      logic d;
 8:    } data_packet_t;
 9:
10:    // other statements
11:
12:  endmodule
```


The "data_packet_t" type is declared in the forbidden scope.

How to Fix

Use package for shared declaration.

Fixed Code

`package.sv`:

```
 1:  package declarations;
 2:
 3:    typedef struct {int a, b;
 4:      logic [1:0] c;
 5:      logic d;
 6:    } data_packet_t;
 7:
 8:    // other declarations
 9:
10:  endpackage
```

`TOP.sv`:

```
1:  module top ( input clk, reset
2:    // other port declarations
3:  );
4:
5:    import declarations::data_packet_t;
6:
7:    // other statements
8:
9:  endmodule
```


Section 2.4 - Struct/unions

ALDEC_SV.2.4.1

Rule Name

Prefer named struct/union types to anonymous structure variables.

Base Rule

ALINT_SV.1.4.2.12

Phase

Parse

Message

The signal(s) with an anonymous {Type} type is declared here.

Problem Description

When anonymous structures/unions are declared in different modules, even if with the same names and members, they are not of the same type. Such description can lead to problems when passing structures/unions through the design blocks. Therefore, it is recommended to use typed structures/unions instead of anonymous ones.

Level Recommendation 3

Checker Behaviour

The checker verifies object declarations all over the project:

- if an anonymous structure or union is detected => violation.

Example - Anonymous structure

Description

In the following example the "var_1" and "var_2" variables are declared with an anonymous structure type => violation.

Sample Code

```
1:  module TOP (
2:    // ports list
3:  );
4:    struct {
5:      logic a;
6:      bit [1:0] c;
7:      int d;
8:    } var_1, var_2;
```



```
 9:
10:    // other statements
11:
12:  endmodule
```

The signal(s) with an anonymous struct type is declared here.

How to Fix

Use **typedef** to declare user-defined type.

Fixed Code

```
 1:  module TOP (
 2:    // ports list
 3:  );
 4:    typedef struct {
 5:      logic a;
 6:      bit [1:0] c;
 7:      int d;
 8:    } data_t;
 9:
10:    data_t var_1;
11:    data_t var_2;
12:
13:    // other statements
14:
15:  endmodule
```


ALDEC_SV.2.4.4

Rule Name

Avoid out-of-range references when treating packed structures/unions as vectors.

Base Rule

ALINT_VLOG.1.3.3.4

Phase

Elaboration

Message-1

Non-constant index is used for the "{ObjectName}" {ObjectType}. Possible values {PossibleRange} are out of the {RealRange} {ObjectType} range.

Message-2

Constant index value "{IndexValue}" that is used for the "{ObjectName}" {ObjectType} is out of the {RealRange} {ObjectType} range.

Problem Description

When using packed struct/union indexed names the index value should not exceed the range of referenced object. Such approach prevents unexpected results which depend on simulator and logic synthesis tool.

Level Recommendation 1

Checker Behaviour

The checker verifies indexed names of packed structs and unions:

- if index value is not a non-constant expression:
    - if maximum possible value of index is greater than MSB of referenced object => violation (message-2);
    - if minimum possible value of index is less than LSB of referenced object => violation (message-2);
- if index value is a constant expression and it is out of the possible range => violation (message-1).

Example 1 - Non-constant index

Description

In the following example the "in1" net is used in the "istr" indexed name. The maximum possible value of the "in1" net is greater then the range of "istr" packed struct => violation.

Sample Code


```
 1:  module top (
 2:    input data,
 3:    input [2:0] in1,
 4:    output reg Q
 5:    //other ports
 6:  );
 7:
 8:    struct packed {reg a, b, c, d;} istr;
 9:
10:    always @(*) begin
11:      istr[in1] = data;
12:      Q = istr[1];
13:    end
14:
15:    //others statements
16:
17:  endmodule
```

Non-constant index is used for the "istr" struct. Possible values [7:0] are out of the [3:0] struct range.

How to Fix

Correct bit width of the "in1" net declaration.

Fixed Code

```
 1:  module top (
 2:    input data,
 3:    input [1:0] in1,
 4:    output reg Q
 5:    //other ports
 6:  );
 7:
 8:    struct packed {reg a, b, c, d;} istr;
 9:
10:    always @(*) begin
11:      istr[in1] = data;
12:      Q = istr[1];
13:    end
14:
15:    //others statements
16:
17:  endmodule
```

Example 2 - Greater index

Description

In the following example an index that is out of range is used for "istr" packed struct => violation.

Sample Code

```
1:  module top ( input data, output reg Q
2:    //other ports
3:  );
4:
5:    struct packed {reg a, b;} istr;
6:
7:    always @(*) begin
8:      istr[0] = data;
9:      Q = istr[2];
```



```
10:    end
11:
12:    //others statements
13:
14:  endmodule
```

Constant index value "2" that is used for the "istr" struct is out of the [1:0] struct range.

How to Fix

Use correct index for "istr" struct.

Fixed Code

```
 1:  module top ( input data, output reg Q
 2:    //other ports
 3:  );
 4:
 5:    struct packed {reg a, b;} istr;
 6:
 7:    always @(*) begin
 8:      istr[0] = data;
 9:      Q = istr[1];
10:    end
11:
12:    //others statements
13:
14:  endmodule
```


ALDEC_SV.2.4.5

Rule Name

Avoid unpacked unions in synthesizable description.

Base Rule

ALINT_SV.2.3.1.13

Phase

Elaboration

Message-1

The "{TypeName}" '{UnionOrStruct}' type is not packed. Use only packed unions in synthesizable description.

The "{ObjectName}" {ObjectType} of the '{TypeName}' type is declared here.

Message-2

The "{MemberName}" member of "{TypeName}" '{DataType}' type is not packed.

The "{MemberName}" member has unpacked dimension.

The "{TypeName}" '{UnionOrStruct}' type has unpacked dimension.

The "{TypeName}" '{UnionOrStruct}' type is not packed.

The '{TypeName}' type of "{ObjectName}" needs to be changed.

Problem Description

It is not recommended to use unpacked unions in synthesizable description since such object type is not synthesizable. Unpacked union is an abstract type which is useful for high-level system and transaction level models.

Level Rule

Checker Behaviour

The checker verifies all `typedefs` and anonymous `unions` used in synthesizable description:

- if `union` is not packed => violation (message-1 for type declaration + detail for each object declaration that uses this type);
- if unpacked `union` also contains unpacked member(s) => violation (message-1 for type declaration + message-2 for each member of this type + corresponding detail of described below);
    - detail-1 - for each declared `union` member with unpacked dimension;
    - detail-2 - for object declaration that uses this member type with unpacked dimension;
    - detail-3 - for declaration of `typedef`/anonymous `union`.

- if unpacked `union` contains unpacked members and these members are not of integer vector type (`bit`, `logic`, `reg`), anonymous `struct`/`union` type or `enum` type => violation (message-2 for each member of this type + detail-4 for each declared of `typedef`/anonymous `union` member, type of which cannot be packed).

Example 1 - Unpacked union

Description

In the following example the "UN" unpacked union variable is declared in the "top" module and used inside **always** construct => violation.

Sample Code

```
 1:  typedef union {reg [7:0] arg_4s;
 2:    byte arg_2s; } t_un;
 3:
 4:  module top ( input [7:0] data, output byte Q
 5:    //other ports
 6:  );
 7:    t_un UN;
 8:
 9:    //other declarations
10:
11:    always begin
12:      UN.arg_4s = data;
13:      Q = UN.arg_2s;
14:    end
15:
16:    //others statements
17:
18:  endmodule
```

The "t_un" 'union' type is not packed. Use only packed unions in synthesizable description.

The "UN" variable of the 't_un' type is declared here.

How to Fix

Use **packed** union instead of **unpacked**.

Fixed Code

```
 1:  typedef union packed {reg [7:0] arg_4s;
 2:    byte arg_2s; } t_un;
 3:
 4:  module top ( input [7:0] data, output byte Q
 5:    //other ports
 6:  );
 7:    t_un UN;
 8:
 9:    //other declarations
10:
11:    always begin
12:      UN.arg_4s = data;
13:      Q = UN.arg_2s;
14:    end
15:
16:    //others statements
17:
```



```
18:  endmodule
```

Example 2 - Unpacked union members

Description

In the following example the "un" `union` type and its members are not packed => violation.

Sample Code

```
 1:  module top ( input [7:0] data, output byte Q
 2:    //other ports
 3:  );
 4:
 5:    union {
 6:      int arg [3:0];
 7:    } un;
 8:
 9:    always
10:      Q = un.arg[1];
11:
12:     //others statements
13:
14:  endmodule
```

The 'union' type is not packed. Use only packed unions in synthesizable description.

The "arg" member of 'int' type is not packed.

The "arg" member has unpacked dimension.

The 'int' type of "arg" needs to be changed.

How to Fix

Make all members and top `union` packed. Change type of member from **int** to **bit[31:0]**.

Fixed Code

```
 1:  module top ( input [7:0] data, output byte Q
 2:    //other ports
 3:  );
 4:
 5:    union packed {
 6:      bit [31:0] [3:0] arg;
 7:    } un;
 8:
 9:    always
10:      Q = un.arg[1];
11:
12:     //others statements
13:
14:  endmodule
```


ALDEC_SV.2.4.7

Rule Name

Prefer to use tagged packed unions and define a flag field member in structures with packed unions.

Base Rule

ALINT_SV.2.3.1.14

Phase

Elaboration

Message-1

The "{ObjectName}" variable of "{TypeName}" union packed type is not tagged. Use only tagged packed unions in synthesizable description.

The "{TypeName}" typedef is declared here.

Message-2

No flag field member is defined for the "{ObjectName}" variable of "{TypeName}" struct type. Add flag field member of "{RecommendedFlagFieldBitWidth}" bit width.

The "{TypeName}" typedef is declared here.

Message-3

The bit width of the "{MemberName}" flag field member of "{ObjectName}" variable is "{CurrentFlagFieldBitWidth}" which is {CompareResult} than the recommended bit width set to "{RecommendedFlagFieldBitWidth}".

The "{MemberName}" member is declared in "{TypeName}" typedef.

The "{MemberName}" member is declared here.

Problem Description

Packed structures and unions provide multiple ways to access groups of bits in a vector, and so simplifying the RTL code and the further debugging. A tagged packed union is a type-checked union. It contains an implicit member that stores a tag, which represents how the union was initially accessed. When a value is stored in a tagged union using a tagged expression, the implicit tag automatically stores information about which member the value was written to. It is recommended to use tagged unions or, in case when a union is a member of structure, define a flag field member as the second member of this structure. Such approach helps to indicate that a real value has been stored in the union.

Level Recommendation 1

Checker Behaviour

The checker verifies all **typedefs** and anonymous **structures/unions** used in synthesizable description:

- if **packed union** is not **tagged** => violation (message-1 for variable declaration);
- if **packed union** is not **tagged** and it is a member of **struct** which contains no more than 2 members and the second member is not a **struct**, **union** or non-synthesizable object:
    - if **struct** contains only **packed union** member (flag field member is not defined) => violation (message-2 for variable declaration);
    - if a bit width defined for a flag field member is more or less than bit width required to cover all members of union => violation (message-3 for variable declaration + detail for flag field member);

It is recommended to use this checker in pair with ALDEC_SV.2.4.5.

Example 1 - Packed union

Description

In the following example the "UN" variable of "t_un" **union** type is not tagged => violation.

Sample Code

```
 1:  typedef union packed {reg [7:0] arg_4s;
 2:    byte arg_2s; } t_un;
 3:
 4:  module top ( input [7:0] data, output byte Q
 5:    //other ports
 6:  );
 7:    t_un UN;
 8:
 9:    //other declarations
10:
11:    always begin
12:      UN.arg_4s = data;
13:      Q = UN.arg_2s;
14:    end
15:
16:    //others statements
17:
18:  endmodule
```

The "UN" variable of "t_un" union packed type is not tagged. Use only tagged packed unions in synthesizable description.

The "t_un" typedef is declared here.

How to Fix

Use **tagged packed union** instead of **packed union**.

Fixed Code

```
1:  typedef union tagged packed {reg [7:0] arg_4s;
2:    byte arg_2s; } t_un;
3:
4:  module top ( input [7:0] data, output byte Q
5:    //other ports
6:  );
```



```
 7:    t_un UN;
 8:
 9:    //other declarations
10:
11:    always begin
12:      UN = tagged arg_4s data;
13:      Q = UN.arg_2s;
14:    end
15:
16:    //others statements
17:
18:  endmodule
```

Example 2 - Members of structure type

Description

In the following example the "t_dpt" **struct** type contains **packed union** member, but does not contains field flag member => violation.

Sample Code

```
 1:  module top ( input data, output reg Q
 2:    //other ports
 3:  );
 4:
 5:    typedef union packed {
 6:      logic idle;
 7:      logic start;
 8:      logic done;
 9:    } [1:0] t_st;
10:
11:    typedef struct packed {
12:      t_st arg_r;
13:    } t_dpt;
14:
15:    t_dpt data_tmp;
16:
17:    always @(*) begin
18:      data_tmp.arg_r[1].idle = data;
19:      Q = data_tmp.arg_r[1].done;
20:    end
21:
22:     //others statements
23:
24:  endmodule
```

No flag field member is defined for the "data_tmp" variable of "t_dpt" struct type. Add flag field member of "3" bit width.

The "t_dpt" typedef is declared here.

How to Fix

Add a field flag member with 3 bits width to **struct**.

Fixed Code

```
 1:  module top ( input data, output reg Q
 2:    //other ports
```



```
 3:  );
 4:
 5:    typedef union packed {
 6:      logic idle;
 7:      logic start;
 8:      logic done;
 9:    } [1:0] t_st;
10:
11:    typedef struct packed {
12:      t_st arg_r;
13:      logic [2:0] arg_l;
14:    } t_dpt;
15:
16:    t_dpt data_tmp;
17:
18:    always @(*) begin
19:      data_tmp.arg_r[1].idle = data;
20:      Q = data_tmp.arg_r[1].done;
21:    end
22:
23:     //others statements
24:
25:  endmodule
```

Example 3 - Incorrect flag field member bit width

Description

In the following example the "t_dpt" **struct** type contains flag field member, but its bit width is less than needed => violation.

Sample Code

```
 1:  module top ( input data, output reg Q
 2:    //other ports
 3:  );
 4:
 5:    typedef union packed {
 6:      logic idle;
 7:      logic start;
 8:      logic done;
 9:    } [1:0] t_st;
10:
11:    typedef struct packed {
12:      t_st arg_r;
13:      logic [1:0] arg_l;
14:    } t_dpt;
15:
16:    t_dpt data_tmp;
17:
18:    always @(*) begin
19:      data_tmp.arg_r[1].idle = data;
20:      Q = data_tmp.arg_r[1].done;
21:    end
22:
23:     //others statements
24:
25:  endmodule
```

The bit width of the "arg_l" flag field member of "data_tmp" variable is "2" which is less than the recommended bit width set to "3".

The "arg_l" member is declared in "t_dpt" typedef.

How to Fix

Change the flag field member bit width from 2 to 3.

Fixed Code

```
 1:  module top ( input data, output reg Q
 2:    //other ports
 3:  );
 4:
 5:    typedef union packed {
 6:      logic idle;
 7:      logic start;
 8:      logic done;
 9:    } [1:0] t_st;
10:
11:    typedef struct packed {
12:      t_st arg_r;
13:      logic [2:0] arg_l;
14:    } t_dpt;
15:
16:    t_dpt data_tmp;
17:
18:    always @(*) begin
19:      data_tmp.arg_r[1].idle = data;
20:      Q = data_tmp.arg_r[1].done;
21:    end
22:
23:     //others statements
24:
25:  endmodule
```


Section 2.5 - Arrays

ALDEC_SV.2.5.2

Rule Name

Declare typedefs for identical arrays used more than once or when passing data through function/task ports.

Base Rule

ALINT_SV.1.3.3.8

Phase

Elaboration

Message

It is detected identical data arrays passing through the "{PortName}" port.

The "{ObjectName}" object of the same array is declared here.

The "{ObjectName}" object is referenced here.

Problem Description

It is recommended to define arrays using **typedef** to give each array type a unique name. Such approach makes reusing array types across multiple files easier, and improves the readability and maintainability of the source code.

Level Recommendation 1

Checker Behaviour

The checker verifies instantiations of modules and subprogram calls:

- if referenced object and port are identical arrays of same type => violation (main message for port declaration + detail for object declaration + sub-detail for instances of modules or subprogram calls where object is referenced).

Default Configuration

The SCAN_PACKED_ARRAYS parameter defines whether the arrays without unpacked dimentions should be scanned.

SCAN_PACKED_ARRAYS = NO

Example 1 - Identical array objects referenced through same port

Description

In the following example the "tmp" variable and the "t_out" port are identical array objects, moreover they are passed through the "out" port of the same array type in the "SUB" module => violation.

Sample Code

```
 1:  module TOP(output reg [3:0] t_out [3:0]
 2:    //other ports
 3:  );
 4:    reg [3:0] tmp [3:0];
 5:
 6:    //statements
 7:
 8:    SUB SUB_1 (.out(tmp));
 9:    SUB SUB_2 (.out(t_out));
10:  endmodule
11:
12:  module SUB (output reg [3:0] out [3:0]
13:    //other ports
14:  );
15:    //statements
16:  endmodule
```

It is detected identical data arrays passing through the "out" port.

The "t_out" object of the same array is declared here.

The "t_out" object is referenced here.

The "tmp" object of the same array is declared here.

The "tmp" object is referenced here.

How to Fix

Declare and use **typedef** for violated objects.

Fixed Code

```
 1:  typedef reg [3:0] my_type [3:0];
 2:
 3:  module top(output my_type t_out
 4:    //other ports
 5:  );
 6:    my_type tmp;
 7:
 8:    //statements
 9:
10:    sub sub_1 (.out(tmp));
11:    sub sub_2 (.out(t_out));
12:  endmodule
13:
14:  module sub (output my_type out
15:    //other ports
16:  );
17:    //statements
18:  endmodule
```


Example 2 - Ignoring of packed arrays

Description

In the following example the data from the "out" port with only packed dimensions that belongs to the "TOP" module is passed through the "rez" port of "add" task, but by default configuration such case is not scanned => no violation.

Fixed Code

```
 1:  module TOP(output reg [3:0][3:0] out
 2:    //other ports
 3:  );
 4:    task add (output reg [3:0][3:0] rez
 5:      //other task port declaratons
 6:      );
 7:    //task statements
 8:    endtask
 9:
10:    always_comb
11:      add (.rez(out));
12:
13:  endmodule
```

Example 3 - Custom configuration

Description

In the following example it is configured to scan packed arrays and the data is passed from the "out" port with only packed dimensions through the "rez" port of "add" task => violation.

Configuration

SCAN_PACKED_ARRAYS = YES

Sample Code

```
 1:  module TOP(output reg [3:0][3:0] out
 2:    //other ports
 3:  );
 4:    task add (output reg [3:0][3:0] rez
 5:      //other task port declaratons
 6:      );
 7:    //task statements
 8:    endtask
 9:
10:    always_comb
11:      add (.rez(out));
12:
13:  endmodule
```

It is detected identical data arrays passing through the "rez" port.

The "out" object of the same array is declared here.

The "out" object is referenced here.

How to Fix

Declare and use **typedef** for violated object.

Fixed Code

```
 1:  typedef reg [3:0][3:0] my_t;
 2:
 3:  module TOP(output my_t out
 4:    //other ports
 5:  );
 6:    task add (output my_t rez
 7:      //other task port declaratons
 8:      );
 9:    //task statements
10:    endtask
11:
12:    always_comb
13:      add (.rez(out));
14:
15:  endmodule
```

Example 4 - Parameterized arrays

Description

In the following example the parameterized ports are referenced and their actual bit-widths are same => violation.

Sample Code

```
 1:  module top
 2:    #(parameter T = 4)
 3:    (input arg [T-1 : 0] );
 4:
 5:    //statements
 6:
 7:    sub #( T ) sub (arg);
 8:  endmodule
 9:
10:  module sub
11:    #(parameter T = 2)
12:    (input in [T-1 : 0] );
13:
14:    //statements
15:
16:  endmodule
```

It is detected identical data arrays passing through the "in" port.

The "arg" object of the same array is declared here.

The "arg" object is referenced here.

How to Fix

Declare and use **typedef** like typed prameter, and declare violated objects with this parameter.

Fixed Code

```
 1:  typedef reg my_t [3:0];
 2:
 3:  module top
```



```
 4:    #(parameter type T = my_t)
 5:    (input T arg );
 6:
 7:    //statements
 8:
 9:    sub #( T ) sub (arg);
10:  endmodule
11:
12:  module sub
13:    #(parameter type T = logic)
14:    (input T arg );
15:
16:    //statements
17:
18:  endmodule
```


ALDEC_SV.2.5.4

Rule Name

Avoid dynamic array types in synthesizable description.

Base Rule

ALINT_SV.2.3.1.10

Phase

Elaboration

Message

The "{ObjectName}" object with dynamic array type is used in synthesizable description.

An object with dynamic array type is used here.

Problem Description

Dynamically sized arrays should not be described in synthesizable description. Such objects are intended for use in verification routines and for modeling at very high levels of abstraction, and they are non-synthesizable.

Level Rule

Checker Behaviour

The checker verifies synthesizable description:

- if a dynamic array object is used => violation.

Example - Variable with dynamic array type from package

Description

In the following example the "arg" variable from the "PKG" package has dynamic dimension and it is used in the "TOP" module inside **always** construct => violation.

Sample Code

```
 1:  package PKG;
 2:    reg arg [];
 3:  endpackage
 4:
 5:  import PKG::arg;
 6:
 7:  module TOP (output reg [3:0] res
 8:    //other port declarations
 9:  );
10:    reg [3:0] tmp;
11:
12:    always_comb begin
```



```
13:      arg = new [4];
14:      for ( int i = 0; i < 4; i++ )
15:        tmp = tmp + arg[i];
16:    end
17:
18:    assign res = tmp;
19:
20:    // other statements
21:
22:  endmodule
```

The "arg" object with dynamic array type is used in synthesizable description.

An object with dynamic array type is used here.

An object with dynamic array type is used here.

How to Fix

Use the variable in non-synthesizable **initial** construct.

Fixed Code

```
 1:  package pkg;
 2:    reg arg [];
 3:  endpackage
 4:
 5:  import pkg::arg;
 6:
 7:  module TOP (
 8:    //port declarations
 9:    output reg [3:0] res
10:  );
11:    reg [3:0] tmp;
12:
13:    initial begin
14:      arg = new [4];
15:      for ( int i = 0; i < 4; i++ )
16:        tmp = tmp + arg[i];
17:    end
18:
19:    assign res = tmp;
20:
21:    // other statements
22:
23:  endmodule
```


Section 2.6 - Ports and Variables

ALDEC_SV.2.6.1

Rule Name

Specify port type and direction explicitly.

Base Rule

ALINT_SV.1.2.4.12

Phase

Parse

Message-1

The direction is not specified for the "{PortName}" port.

Message-2

The data type is not specified for the "{PortName}" port.

Problem Description

When a type and a direction is not explicitly specified in the port declaration, then a default value is inferred for it. However, such description worsens the code readability and makes it harder to maintain. So it is recommended to define a type and a direction for each design port.

Level Rule

Checker Behaviour

The checker verifies constructs from the list defined with help of the CONSTRUCTS_TO_CHECK parameter:

- if a direction is omitted in a port declaration => violation (message-1);
- if a data type is omitted in a port declaration => violation (message-2).

Default Configuration

The CONSTRUCTS_TO_CHECK parameter defines the list of constructs that should be verified. Possible values are: 'MODULE', 'FUNCTION', 'TASK'.

CONSTRUCTS_TO_CHECK = (MODULE, FUNCTION, TASK)

Example 1 - Module ports without type

Description

In the following example the "D", "CLK" and "Q" ports are declared without a type definition => violation.

Sample Code

```
1:  module DFF (
2:    input  D,
3:    input  CLK,
4:    output Q
5:  );
6:    // module statements
7:  endmodule
```

The data type is not specified for the "D" port.

The data type is not specified for the "CLK" port.

The data type is not specified for the "Q" port.

How to Fix

Specify the port types explicitly.

Fixed Code

```
1:  module DFF (
2:    input  wire D,
3:    input  wire CLK,
4:    output reg  Q
5:  );
6:    // module statements
7:  endmodule
```

Example 2 - Direction is not specified for function parameters

Description

In the following example the "real2int" function parameters do not have direction specification, but functions are not checked according to the CONSTRUCTS_TO_CHECK parameter => no violation.

ConfigurationCONSTRUCTS_TO_CHECK = "('MODULE')"Fixed Code

```
1:  module TOP (
2:    // ports list
3:  );
4:
5:    function  real2int (
6:      real  i,
7:      logic r
8:    );
9:      // function description
10:   endfunction
11:
12:   // other statement
13:
14: endmodule
```


ALDEC_SV.2.6.2

Rule Name

Use wire type for multi-driven objects and logic type for others.

Base Rule

ALINT_SV.1.3.1.8

Phase

Elaboration

Message-1

The "{ObjectName}" {ObjectType} is not of logic type.

Message-2

The "{ObjectName}" {ObjectType} is driven from multiple places. Use wire type for multi-driven objects.

The "{ObjectName}" {ObjectType} is driven here.

Problem Description

It is recommended to declare any signal with multiple drivers using **wire** data type. Some logic synthesis tools report a syntax error if a signal of logic declaration receives input from multiple **assign** or **always** blocks, or it is connected to multiple module outputs. For other signals, it is recommended to use **logic** data type. Such approach also prevents confusion with terms **reg** and **wire** (their names imply is different from the actual circuits they model).

Level Rule

Checker Behaviour

The checker verifies declarations and assignments in the synthesizable description:

- if object from DECLARATIONS_TO_CHECK list is defined as a **wire** and does not have multiple drivers => violation (message-1 for object declaration).
- if any object (the DECLARATIONS_TO_CHECK parameter is not considered) is of **logic** type and is driven from multiple places => violation (message-2 for object declaration + detail on each assignment).

Note
Only objects of **reg**, **wire**, and **logic** types are scanned.

Default Configuration

The DECLARATIONS_TO_CHECK parameter defines types of objects to be checked. Possible values are: 'INPUT_PORT', 'INOUT_PORT', 'OUTPUT_PORT', 'VARIABLE', 'SIGNAL'.

DECLARATIONS_TO_CHECK = ('INPUT_PORT', 'INOUT_PORT', 'OUTPUT_PORT', 'SIGNAL', 'VARIABLE')

Example 1 - Port data types

Description

In the following example the **wire** and **reg** data types are used for ports each of which is driven only once => violation.

Sample Code

```
1:  module DFF (
2:    input  wire clk, d,
3:    output reg Q
4:  );
5:
6:    always @( posedge clk )
7:      Q = d;
8:
9:  endmodule
```

The "clk" wire is not of logic type.

The "d" wire is not of logic type.

The "Q" reg is not of logic type.

How to Fix

Use **logic** data type instead of **wire** and **reg**.

Fixed Code

```
1:  module DFF (
2:    input  logic clk, d,
3:    output logic Q
4:  );
5:
6:    always @( posedge clk )
7:      Q = d;
8:
9:  endmodule
```

Example 2 - Object with multiple continuous assignments

Description

In the following example "Q_tmp" variable of **logic** type is driven from both the **assign** statement and the "sub" instance => violation.

Sample Code

```
1:  module top (
2:      input logic clk, [3:0] X,
3:      output logic [3:0] Q_out
4:    );
5:
6:    logic [3:0] Q_tmp;
7:
8:    assign Q_tmp = X;
```



```
 9:
10:    sub sub (Q_tmp);
11:
12:    always @(posedge clk)
13:      Q_out = Q_tmp;
14:
15:    //other statements
16:  endmodule
17:
18:  module sub ( output logic [3:0] sig );
19:    //statements
20:  endmodule
```

The "Q_tmp" logic is driven from multiple places. Use wire type for multi-driven objects.

The "Q_tmp" logic is driven here.

The "Q_tmp" logic is driven here.

How to Fix

Declare "Q_tmp" variable as a net.

Fixed Code

```
 1:  module top (
 2:      input logic clk, [3:0] X,
 3:      output logic [3:0] Q_out
 4:    );
 5:
 6:    wire [3:0] Q_tmp;
 7:
 8:    assign Q_tmp = X;
 9:
10:    sub sub (Q_tmp);
11:
12:    always @(posedge clk)
13:      Q_out = Q_tmp;
14:
15:    //other statements
16:  endmodule
17:
18:  module sub ( output logic [3:0] sig );
19:    //statements
20:  endmodule
```

Example 3 - Object with multiple procedural assignments

Description

In the following example "Q_out" port driven form several **always** statements => violation.

Sample Code

```
1:  module top (
2:      input logic clk_x, clk_y, X, Y,
3:      output logic Q_out
4:    );
5:
6:    always @(posedge clk_x)
```



```
 7:      Q_out = X;
 8:
 9:    always @(posedge clk_y)
10:      Q_out = Y;
11:
12:  endmodule
```

The "Q_out" logic is driven from multiple places. Use wire type for multi-driven objects.

The "Q_out" logic is driven here.

The "Q_out" logic is driven here.

How to Fix

Declare two temporary variables, use them in place instead of "Q_out" port. Declare "Q_out" as a net and assign new variables to it.

Fixed Code

```
 1:  module top (
 2:      input logic clk_x, clk_y, X, Y,
 3:      output wire Q_out
 4:    );
 5:
 6:    logic Q_tmp_x, Q_tmp_y;
 7:
 8:    always @(posedge clk_x)
 9:      Q_tmp_x = X;
10:    assign Q_out = Q_tmp_x;
11:
12:    always @(posedge clk_y)
13:      Q_tmp_y = Y;
14:    assign Q_out = Q_tmp_y;
15:
16:  endmodule
```


Chapter 3 - Procedural Blocks

Section 3.1 - Processes

ALDEC_SV.3.1.2

Rule Name

Prefer 'always_comb' to 'always' with @* sensitivity.

Base Rule

ALINT_SV.5.2.2.7

Phase

Parse

Message

The 'always' construct with @* sensitivity list is detected. Use the 'always_comb' construct instead.

Problem Description

It is recommended to use **always_comb** procedural block instead of **always** construct with **@***. The **always @*** is sensitive to signals read directly by this procedural block only (it is not sensitive to signals read in subprograms called by this procedural block). Therefore, to avoid inferring of incomplete sensitivity list, the **always_comb** procedural block should be used.

Level Recommendation 1

Checker Behaviour

The checker scans non-testbench description and verifies combinational **always** process statements:

- if an **always** construct with **@*** sensitivity list is detected => violation.

Example - Incorrect multiplexer description

Description

In the following example the **always** construct has **@*** sensitivity list => violation.

Sample Code

```
1:  module MUX (
2:    input            EN,
3:    input      [1:0] D,
4:    output reg       Q
5:  );
6:
```



```
 7:    always @*
 8:      if (EN)
 9:        Q = D[0];
10:      else
11:        Q = D[1];
12:
13:  endmodule
```

The 'always' construct with @* sensitivity list is detected. Use the 'always_comb' construct instead.

How to Fix

Use **always_comb** construct.

Fixed Code

```
 1:  module MUX (
 2:    input            EN,
 3:    input      [1:0] D,
 4:    output reg       Q
 5:  );
 6:
 7:    always_comb
 8:      if (EN)
 9:        Q = D[0];
10:      else
11:        Q = D[1];
12:
13:  endmodule
```


ALDEC_SV.3.1.3

Rule Name

Use 'always_latch' block for generating latches only.

Base Rule

ALINT_SV.3.2.1.3

Phase

Synthesis

Message

The 'always_latch' block contains output(s) that does not infer latch logic.

The latch is not inferred for "{ObjectName}" {ObjectType}.

Problem Description

The `always_latch` procedural blocks are used for modeling latched-based logic, so the description inside such blocks should generate latches only. In such a case the designer intent to describe latch logic is shown explicitly. Inconsistency between the type of procedural block and the logic described inside it may indicate an error in the description and decreases the code readability. Also the synthesis tools will report a warning for the ambiguous use of the SystemVerilog specialized procedural blocks.

Level Recommendation 1

Checker Behaviour

The checker scans **always_latch** constructs in the synthesizable description:

- if a variable assigned inside the **always_latch** block does not infer a latch => violation (main for process + detail(s) for each violating assignment).

Example - Multiplexer inside 'always_latch' block

Description

In the following example a multiplexer is inferred for the "Q" variable assigned within the **always_latch** block => violation.

Sample Code

```
1:  module TOP (
2:    input  logic EN,
3:    input  logic D,
4:    output logic Q
5:  );
6:
7:    always_latch begin
```



```
 8:      if ( EN )
 9:        Q = 1'b0;
10:      else
11:        Q = D;
12:    end
13:
14:  endmodule
```

The 'always_latch' block contains output(s) that does not infer latch logic.

The latch is not inferred for "Q" port.

How to Fix

Use **always_comb** block instead of **always_latch**.

Fixed Code

```
 1:  module TOP (
 2:    input  logic EN,
 3:    input  logic D,
 4:    output logic Q
 5:  );
 6:
 7:    always_comb begin
 8:      if ( EN )
 9:        Q = 1'b0;
10:      else
11:        Q = D;
12:    end
13:
14:  endmodule
```


Section 3.2 - Tasks/Functions

ALDEC_SV.3.2.1

Rule Name

Prefer void functions to tasks to enforce synthesizability checks.

Base Rule

ALINT_SV.2.5.1.8

Phase

Elaboration

Message

The "{TaskName}" task used in a synthesizable description is detected. Prefer void functions to tasks to enforce synthesizability checks.

Problem Description

It is recommended to define **void function** instead of **task** in a synthesizable description. There are some restrictions for function content (for example, functions cannot contain any event controls) that help to ensure that synthesis results is proper.

Level Recommendation 1

Checker Behaviour

The checker scans a synthesizable description:

- if **task** call is detected => violation

Example 1 - Task is used

Description

In the following example the "convert" task is called in a synthesizable description => violation.

Sample Code

```
 1:  module TOP (
 2:    input      [63:0] I,
 3:    output reg [ 7:0] O0,
 4:    output reg [ 7:0] O1
 5:  );
 6:
 7:    task convert (
 8:      input  [63:0] data_in,
 9:      input         factor,
10:      output [ 7:0] data_out_a,
```



```
11:      output [ 7:0] data_out_b
12:    );
13:      begin: data_assign
14:        data_out_a = { data_in[factor], data_in[factor+3], data_in[factor+6]
    };
15:        data_out_b = !data_out_a;
16:      end
17:    endtask
18:
19:    always @( I )
20:      convert (
21:        .data_in   (I),
22:        .factor    (7),
23:        .data_out_a(O0),
24:        .data_out_b(O1)
25:      );
26:
27:  endmodule
```

The "convert" task used in a synthesizable description is detected. Prefer void functions to tasks to enforce synthesizability checks.

How to Fix

Use **void function** instead of **task**.

Fixed Code

```
 1:  module TOP (
 2:    input      [63:0] I,
 3:    output reg [ 7:0] O0,
 4:    output reg [ 7:0] O1
 5:  );
 6:
 7:    function void convert (
 8:      input  [63:0] data_in,
 9:      input         factor,
10:      output [ 7:0] data_out_a,
11:      output [ 7:0] data_out_b
12:    );
13:      begin: data_assign
14:        data_out_a = { data_in[factor], data_in[factor+3], data_in[factor+6]
    };
15:        data_out_b = !data_out_a;
16:      end
17:    endfunction
18:
19:    always @( I )
20:      convert (
21:        .data_in   (I),
22:        .factor    (7),
23:        .data_out_a(O0),
24:        .data_out_b(O1)
25:      );
26:
27:  endmodule
```

Example 2 - Testbench description is not scanned

Description

In the following example the task is called inside the "TOP_TB" testbench module that is not synthesizable => no violation.

Fixed Code

```
 1:  module TOP_TB (  );
 2:
 3:    wire       clk;
 4:    wire       rst;
 5:    wire [3:0] addr;
 6:
 7:    reg [31:0] data;
 8:
 9:    function read_from_file(
10:      input [3:0] addr
11:    );
12:      // statements
13:    endfunction
14:
15:    task GET_DATA (
16:      input   [3:0] addr,
17:      output [31:0] data
18:    );
19:      @( posedge clk );
20:      if ( rst == 1'b1 )
21:        data = {31{1'b0}};
22:      else
23:        data = read_from_file(addr);
24:    endtask
25:
26:    always begin
27:      GET_DATA( addr, data );
28:    end
29:
30:  endmodule
```


ALDEC_SV.3.2.6

Rule Name

Functions should be defined in packages only.

Base Rule

ALINT_SV.1.4.2.9

Phase

Parse

Message

The "{DeclarationName}" {DeclarationType} is declared in the forbidden scope.

Problem Description

When functions are declared within the module/interface, they are synthesizable but their visibility is reduced. Therefore, it is recommended to define functions in named packages instead of modules/interfaces in order to facilitate re-use.

Level Recommendation 2

Checker Behaviour

The checker scans **module** and **interface** design units:

- if the module/interface contains function declarations => violation.

Example - Function declaration in module

Description

In the following example the "real2int" function is declared in the "top" module => violation.

Sample Code

`TOP.sv`:

```
 1:  module top ( input clk, reset
 2:    // other port declarations
 3:  );
 4:
 5:    function  real2int (
 6:      real  i,
 7:      logic r
 8:    );
 9:      // function description
10:    endfunction
11:
12:    // other statements
```



```
13:
14:  endmodule
```

The "real2int" function is declared in the forbidden scope.

How to Fix

Use package for shared declaration.

Fixed Code

`package.sv`:

```
 1:  package declarations;
 2:
 3:    function  real2int (
 4:      real  i,
 5:      logic r
 6:    );
 7:      // function description
 8:    endfunction
 9:
10:    // other declarations
11:
12:  endpackage
```

`TOP.sv`:

```
 1:  module top ( input clk, reset
 2:    // other port declarations
 3:  );
 4:
 5:    import declarations::real2int;
 6:
 7:    // other statements
 8:
 9:  endmodule
```


Section 3.3 - Assignments

ALDEC_SV.3.3.3

Rule Name

Do not use increment/decrement operators in expressions.

Base Rule

ALINT_SV.2.3.1.4

Phase

Parse

Message

Operator "{Operator}" is used in the expression.

Problem Description

It is not recommended to use increment and decrement operators inside expressions. Such description is confusing and error-prone. Due to possible undefined order of operations in an expression, increment/decrement operator should be used as a separate assignment statement.

Level Recommendation 1

Checker Behaviour

The checker verifies expressions:

- if an increment (++) or decrement (--) is used in expression => violation.

Example - Increment operator

Description

In the following example the increment operator is used inside an expression => violation.

Sample Code

```
1:  module top (input logic en, output logic [3:0] q );
2:
3:    logic [3:0] i;
4:
5:    always_comb
6:      if (en)
7:        q = ++i;
8:
9:  endmodule
```

Operator "++" is used in the expression.

How to Fix

Use increment operator only as a separate blocking assignment.

Fixed Code

```
 1:  module top (input logic en, output logic [3:0] q );
 2:
 3:    logic [3:0] i;
 4:
 5:    always_comb
 6:      if (en)
 7:        begin
 8:          ++i;
 9:          q = i;
10:        end
11:
12:  endmodule
```


ALDEC_SV.3.3.4

Rule Name

Do not use new assignment operators (such as +=, -=) in expressions.

Base Rule

ALINT_SV.2.3.1.4

Phase

Parse

Message

Operator "{Operator}" is used in the expression.

Problem Description

Assignment operators should not be used inside expressions. Such description is confusing and error-prone. Due to possible undefined order of operations in an expression, assignment operator should be used as a separate assignment statement.

Level Recommendation 1

Checker Behaviour

The checker verifies expressions:

- if an assignment is used in expression => violation.

The following assignment operators are checked: '+=', '-=', '*=', '/=', '%=', '&=', '|=', '^=', '>>=', '<<=', '>>>=', '<<<='.

Example - Assignment operator in expression

Description

In the following example an assignment operator ('+=') is used inside an expression => violation.

Sample Code

```
1:  module top (input logic en, output logic [3:0] q );
2:
3:    logic [3:0] i;
4:
5:    always_comb
6:      if (en)
7:        q = (i+=1);
8:
9:  endmodule
```

Operator "+=" is used in the expression.

How to Fix

Use assignment operator as a separate blocking assignment.

Fixed Code

```
 1:  module top (input logic en, output logic [3:0] q );
 2:
 3:    logic [3:0] i;
 4:
 5:    always_comb
 6:      if (en)
 7:        begin
 8:          i+=1;
 9:          q = i;
10:        end
11:
12:  endmodule
```


ALDEC_SV.3.3.5

Rule Name

Do not write same variable in different always_comb, always_ff, always_latch blocks.

Base Rule

ALINT_SV.2.6.1.6

Phase

Elaboration

Message

The "{ObjectName}" object is assigned in different always blocks.

The "{ObjectName}" object is assigned here in the "{BlockName}" {BlockType}.

The "{SubprogramName}" {SubprogramType} is called here in the "{BlockName}" {BlockType}.

Problem Description

The **always_ff**, **always_comb**, or **always_latch** procedural blocks have the restriction that a variable can only be assigned from one procedural block (this is additional checking that a signal declared as a variable only has a single source). Therefore, avoid using same variable in different procedural blocks in order to prevent simulation errors.

Level Rule

Checker Behaviour

The checker scans **always** constructs:

- if the same object is assigned in several **always** blocks and at least one of them is **always_comb**, **always_ff** or **always_latch** => violation (main message for object declaration + detail-1 on each assignment + detail-2 optionally, when task or function is called).

Example 1 - Object is assigned in several always blocks

Description

In the following example the "Q" port is assigned in both simple **always** block and **always_ff** block => violation.

Sample Code

```
1:  module top ( input clk, rst, D, D2,
2:    output reg Q
3:    //other ports
4:  );
5:    //other declarations
6:
```



```
 7:    always @(*)
 8:      Q <= D;
 9:
10:    always_ff @(posedge clk)
11:      begin : ff_block
12:        if (rst)
13:          Q = 1'b0;
14:        else
15:          Q = D2;
16:      end
17:
18:    //others statements
19:
20:  endmodule
```

The "Q" object is assigned in different always blocks.

The "Q" object is assigned here in the always construct.

The "Q" object is assigned here in the "ff_block" always_ff construct.

The "Q" object is assigned here in the "ff_block" always_ff construct.

How to Fix

Declare second port variable and assign it instead in one of blocks.

Fixed Code

```
 1:  module top ( input clk, rst, D, D2,
 2:    output reg Q, Q2
 3:    //other ports
 4:  );
 5:    //other declarations
 6:
 7:    always @(*)
 8:      Q <= D;
 9:
10:    always_ff @(posedge clk)
11:      begin : ff_block
12:        if (rst)
13:          Q2 = 1'b0;
14:        else
15:          Q2 = D2;
16:      end
17:
18:    //others statements
19:
20:  endmodule
```

Example 2 - Object is assigned in subprogram

Description

In the following example the "Q" port is assigned in **always_latch** block and in "latch_task" **task** that is called in other **always_latch** block => violation.

Sample Code


```
 1:  module top ( input en, en2, D, D2,
 2:    output reg Q
 3:    //other ports
 4:  );
 5:    //other declarations
 6:
 7:    task latch_task (input e, d);
 8:      if (e)
 9:        Q = d;
10:    endtask
11:
12:    always_latch
13:      if (en)
14:        Q <= D;
15:
16:    always_latch
17:      latch_task(en2, D2);
18:
19:    //others statements
20:
21:  endmodule
```

The "Q" object is assigned in different always blocks.

The "Q" object is assigned here in the always_latch construct.

The "Q" object is assigned here in the "latch_task" task.

The "latch_task" task is called here in the always_latch construct.

How to Fix

Declare temporary variables and assign them instead.

Fixed Code

```
 1:  module top ( input en, en2, D, D2,
 2:    output reg Q
 3:    //other ports
 4:  );
 5:    //other declarations
 6:
 7:    reg tmp_1, tmp_2;
 8:
 9:    task latch_task (input e, d);
10:      if (e)
11:        tmp_2 = d;
12:    endtask
13:
14:    always_latch
15:      if (en)
16:        tmp_1 <= D;
17:
18:    always_latch
19:      latch_task(en2, D2);
20:
21:    assign Q = tmp_1 || tmp_2;
22:
23:    //others statements
24:
25:  endmodule
```


Example 3 - Object is assigned in several simple always blocks

Description

In the following example the "Q" port is assigned in two simple **always** blocks => no violation.

Fixed Code

```
 1:  module top ( input clk, r, D, D2,
 2:    output reg Q
 3:    //other ports
 4:  );
 5:
 6:    //other declarations
 7:
 8:    always @(D)
 9:      Q = D;
10:
11:    always @(posedge clk)
12:      begin : ff_block
13:        if (r)
14:          Q = 1'b0;
15:        else
16:          Q = D2;
17:      end
18:
19:    //others statements
20:
21:  endmodule
```


Section 3.4 - Expressions

ALDEC_SV.3.4.2

Rule Name

Use only constant expressions within 'inside' operator with wildcards.

Base Rule

ALINT_SV.2.3.1.7

Phase

Elaboration

Message

Non-constant expression in the RHS of 'inside' operator: "{Expression}".

Problem Description

It is recommended to use only constant expressions in the RHS of **inside** operator with wildcard character(s). Some synthesis tools do not support using wildcards and non-constant values in expressions after the **inside** keyword, so description will be not synthesizable.

Level Recommendation 2

Checker Behaviour

The checker verifies values of the **inside** operator in synthesizable description:

- if RHS of the comparison contains at least one wildcard and non-constant value => violation.

Example - Inside operator

Description

In the following example value with wildcard and input ports are used in the RHS of **inside** operator => violation.

Sample Code

```
 1:  module top (
 2:    input  logic [3:0] data, a, b, c,
 3:    output logic pass1, pass2, pass3, pass4);
 4:
 5:    always_comb begin
 6:      pass1 = (data inside {1, 2, 4, 6});
 7:      pass2 = (data inside {[0:12]});
 8:      pass3 = (data inside {4'b11?0, b, c});
 9:      pass4 = (data inside {a, b, c});
10:    end
11:
```



```
12:  endmodule
```

Non-constant expression in the RHS of 'inside' operator: "b".

Non-constant expression in the RHS of 'inside' operator: "c".

How to Fix

Rewrite the expression using '==' and '||' operators.

Fixed Code

```
 1:  module top (
 2:    input  logic [3:0] data, a, b, c, d,
 3:    output logic pass1, pass2, pass3, pass4);
 4:
 5:    always_comb begin
 6:      pass1 = (data inside {1, 2, 4, 6});
 7:      pass2 = (data inside {[0:12]});
 8:      pass3 = (data == 4'b1100 || data == 4'b1110 || data == b || data == c);
 9:      pass4 = (data inside {a, b, c});
10:    end
11:
12:  endmodule
```


Section 3.5 - Type Casting

ALDEC_SV.3.5.1

Rule Name

Avoid casts that narrow the width of the value.

Base Rule

ALINT_SV.1.3.4.6

Phase

Elaboration

Message

The bit width "{ExpBitWidth}" of the "{Expression}" expression is greater than the bit width "{CastBitWidth}" of the "{CastingType}" casting type.

Problem Description

The bit width of the casting type should be equal or wider then the expression to be cast. Otherwise, the data loss can occur due to truncation of upper bits, when an expression is cast to the smaller size than the number of bits in the expression.

Level Rule

Checker Behaviour

The checker verifies casting operators:

- if a bit width of a casting type is narrower than a bit width of an expression to be cast => violation.

Example - Cast operator

Description

In the following example the 8-bit "var_a" byte variable is cast to 1-bit logic type => violation.

Sample Code

```
 1:  module TOP (
 2:    input logic reset, clock
 3:    //other ports
 4:  );
 5:
 6:    byte var_a;
 7:    logic [7:0] var_b;
 8:    //other declarations
 9:
10:    assign var_b = logic'(var_a);
```



```
11:    //other statements
12:
13:  endmodule
```

The bit width "8" of the "var_a" expression is greater than the bit width "1" of the "logic" casting type.

How to Fix

Declare a user-defined 8-bit logic type and use it in a cast operator.

Fixed Code

```
 1:  module TOP (
 2:    input logic reset, clock
 3:    //other ports
 4:  );
 5:
 6:    typedef logic [7:0] logic_7to0_t;
 7:
 8:    byte var_a;
 9:    logic_7to0_t var_b;
10:    //other declarations
11:
12:    assign var_b = logic_7to0_t'(var_a);
13:    //other statements
14:
15:  endmodule
```


ALDEC_SV.3.5.2

Rule Name

Do not use dynamic '$cast' function in RTL description.

Base Rule

ALINT_SV.2.3.1.11

Phase

Parse

Message

The dynamic '$cast' function is used in RTL description.

Problem Description

It recommended to use static compile-time cast operator when describing the RTL code to ensure safe synthesis. The dynamic **$cast** system function might not be supported by the synthesis tools and the description will be non-synthesizable.

Level Rule

Checker Behaviour

The checker verifies synthesizable description:

- if **$cast** function is used => violation.

Example - Use of $cast system function

Description

In the following example dynamic **$cast** function is used to cast the "arg" input to the "out" output of an enumeration type => violation.

Sample Code

```
 1:  typedef enum reg {A, B} en;
 2:
 3:  module TOP (
 4:    input wire arg,
 5:    output var en out
 6:    // other port declarations
 7:  );
 8:    always @(*) begin
 9:      $cast(out, arg);
10:      // other statements
11:    end
12:  endmodule
```

The dynamic '$cast' function is used in RTL description.

How to Fix

Use static cast operator instead of the **$cast** function.

Fixed Code

```
 1:  typedef enum reg {A, B} en;
 2:
 3:  module TOP (
 4:    input wire arg,
 5:    output var en out
 6:    // other port declarations
 7:  );
 8:    always @(*) begin
 9:      out = en'(arg);
10:      // other statements
11:    end
12:  endmodule
```


ALDEC_SV.3.5.3

Rule Name

Avoid casting out-of-range numeric values to enumeration types.

Base Rule

ALINT_SV.1.3.4.7

Phase

Elaboration

Message

The "{Constant}" value is out-of-range for the "{EnumType}" enumeration type.

Problem Description

The out-of-range numeric values should never be assigned to enumerated type variables, because such description can lead to indeterminate behavior. As a result, differences between pre-synthesis simulation of the RTL model and synthesis results can occur.

Level Rule

Checker Behaviour

The checker verifies casting to enumeration types:

- if the casting expression contains only constants and it does not match any of the enumerated labels => violation.

Example - Constant in cast operator expression

Description

In the following example the "0" value that is out of the "state_t" enumeration type range and it is used as an expression for casting operator => violation.

Sample Code

```
 1:  module TOP (
 2:    input logic reset, clock
 3:    //other ports
 4:  );
 5:
 6:    typedef enum logic [2:0] {
 7:      RED = 3'b001,
 8:      YELLOW = 3'b010,
 9:      GREEN = 3'b100} states_t;
10:
11:    states_t State, NextState;
12:
```



```
13:    always @(posedge clock, negedge reset)
14:      if (reset == 0)
15:        NextState = states_t'(0);
16:      //other branches
17:
18:  endmodule
```

The "0" value is out-of-range for the "states_t" enumeration type.

How to Fix

Use enumeration method instead of using casting operator.

Fixed Code

```
 1:  module TOP (
 2:    input logic reset, clock
 3:    //other ports
 4:  );
 5:
 6:    typedef enum logic [2:0] {
 7:      RED = 3'b001,
 8:      YELLOW = 3'b010,
 9:      GREEN = 3'b100} states_t;
10:
11:    states_t State, NextState;
12:
13:    always @(posedge clock, negedge reset)
14:      if (reset == 0)
15:        NextState = State.first;
16:      //other branches
17:
18:  endmodule
```


Section 3.6 - Loops

ALDEC_SV.3.6.1

Rule Name

Prefer locally declared counters in 'for' loops.

Base Rule

ALINT_SV.2.7.1.8

Phase

Parse

Message

The "{LoopVariable}" loop variable is not declared inside the 'for' loop statement.

Problem Description

It is recommended to declare **for** loop variables locally within the loop statements. Under such an approach, variables are visible only to corresponding **for** loop, so values of those variables cannot be used or modified outside this **for** loop statement. Such description is safer for loops implementation.

Level Recommendation 1

Checker Behaviour

The checker scans **for** loop statements in the synthesizable description:

- if a loop variable is declared outside of the **for** loop statement (not as a part of the **for** loop initialization statement) => violation.

Example - Loop variable

Description

In the following example two **for** loop statement use the same variable that is declared within the module => violation.

Sample Code

```
1:  module top (
2:    input SEL, [31:0] OP1, OP2,
3:    output reg [31:0] RES_ADD, RES_SUB
4:  );
5:
6:    int i;
7:
8:    always_comb
```



```
 9:      for (i = 0; i < 32; i = i + 1 )
10:        RES_ADD[i] = OP1[i] + OP2[i];
11:
12:    always_comb
13:      for (i = 0; i < 32; i = i + 1 )
14:        RES_SUB[i] = OP1[i] - OP2[i];
15:
16:  endmodule
```

The "i" loop variable is not declared inside the 'for' loop statement.

The "i" loop variable is not declared inside the 'for' loop statement.

How to Fix

Declare each loop variable within the initialization part of the corresponding loop statement.

Fixed Code

```
 1:  module top (input SEL, [31:0] OP1, OP2, output reg [31:0] RES_ADD, RES_SUB);
 2:
 3:    always_comb
 4:      for (int i = 0; i < 32; i = i + 1 )
 5:        RES_ADD[i] = OP1[i] + OP2[i];
 6:
 7:    always_comb
 8:      for (int i = 0; i < 32; i = i + 1 )
 9:        RES_SUB[i] = OP1[i] - OP2[i];
10:
11:  endmodule
```


Section 3.7 - Conditional Statements

ALDEC_SV.3.7.1

Rule Name

Use SystemVerilog 'case-inside' instead of Verilog 'casex' and 'casez'.

Base Rule

ALINT_SV.5.2.1.11

Phase

Elaboration

Message

The 'casex/casez' statement is detected. Prefer the 'case-inside' statement in SystemVerilog description.

Problem Description

It is recommended to use **inside** operator with **case** statements instead of **casex** or **casez** statements. Such approach prevents problems with propagation of Z and X values, and mismatches between simulation and synthesis functionality.

Level Recommendation 1

Checker Behaviour

The checker scans synthesizable description:

- if **casex** or **casez** statement is detected => violation.

Example - casex statement

Description

In the following example the **casex** statement is used => violation.

Sample Code

```
 1:  module MUX (
 2:    input      CTRL,
 3:    input      RST,
 4:    input      D0,
 5:    input      D1,
 6:    output reg DOUT
 7:  );
 8:
 9:    reg sel, tmp;
10:
11:    always_comb
```



```
12:      case ( CTRL )
13:        1'b0   : sel = RST;
14:        1'b1   : sel = ~RST;
15:        default: sel = 1'bx;
16:      endcase
17:
18:    assign tmp = sel;
19:
20:    always_comb
21:      casex ( tmp )
22:        1'b0:    DOUT = D0 & D1;
23:        1'b1:    DOUT = D0 | D1;
24:        default: DOUT = 1'bx;
25:      endcase
26:
27:  endmodule
```

The 'casex/casez' statement is detected. Prefer the 'case-inside' statement in SystemVerilog description.

How to Fix

Use the **case-inside** statement instead of the **casex** statement.

Fixed Code

```
 1:  module MUX (
 2:    input      CTRL,
 3:    input      RST,
 4:    input      D0,
 5:    input      D1,
 6:    output reg DOUT
 7:  );
 8:
 9:    reg sel, tmp;
10:
11:    always_comb
12:      case ( CTRL )
13:        1'b0   : sel = RST;
14:        1'b1   : sel = ~RST;
15:        default: sel = 1'bx;
16:      endcase
17:
18:    assign tmp = sel;
19:
20:    always_comb
21:      case ( tmp ) inside
22:        1'b0:    DOUT = D0 & D1;
23:        1'b1:    DOUT = D0 | D1;
24:        default: DOUT = 1'bx;
25:      endcase
26:
27:  endmodule
```


ALDEC_SV.3.7.2

Rule Name

Prefer 'unique-case' to full_case + parallel_case pragmas.

Base Rule

ALINT_SV.1.2.7.10

Phase

Parse

Message

Use 'unique-case' statements instead of specifying 'full_case' and 'parallel_case' pragmas.

Problem Description

For synthesis, a SystemVerilog **unique-case** statement is equivalent to specifying both the **full_case** and **parallel_case** pragmas. However, **unique-case** statement performs run-time uniqueness checks for **case** items expressions. Verilog-style **case** statement with **full_case** and **parallel_case** pragmas does not provide such checks. So it is recommended to use **unique-case** statement to prevent mismatches between interpretation of the **case** statement by different tools and ensure that the designer intent is correct.

Level Recommendation 2

Checker Behaviour

The checker verifies **case** statements:

- if combination of the **full_case** and **parallel_case** pragmas are used for **case** statement => violation.

Example - Combination of 'case' specific pragmas

Description

In the following example the **full_case** and **parallel_case** pragmas are used => violation.

Sample Code

```
 1:  module TOP (
 2:    output reg [3:0] DO,
 3:    input      [1:0] DI,
 4:    input            EN
 5:  );
 6:
 7:    always @(DI or EN) begin
 8:      DO = 4'h0;
 9:      (* full_case, parallel_case *)
10:      casex ({EN, DI})
11:        3'b1_00: DO[0] = 1'b1;
12:        3'b1_01: DO[1] = 1'b1;
```



```
13:        3'b1_10: DO[2] = 1'b1;
14:        3'b1_?1: DO[3] = 1'b1;
15:      endcase
16:    end
17:
18:  endmodule
```

Use 'unique-case' statements instead of specifying 'full_case' and 'parallel_case' pragmas.

How to Fix

Use **unique-case** instead of **full_case** + **parallel_case** pragmas.

Fixed Code

```
 1:  module TOP (
 2:    output reg [3:0] DO,
 3:    input      [1:0] DI,
 4:    input            EN
 5:  );
 6:
 7:    always @(DI or EN) begin
 8:      DO = 4'h0;
 9:      unique casex ({EN, DI})
10:        3'b1_00: DO[0] = 1'b1;
11:        3'b1_01: DO[1] = 1'b1;
12:        3'b1_10: DO[2] = 1'b1;
13:        3'b1_?1: DO[3] = 1'b1;
14:      endcase
15:    end
16:
17:  endmodule
```


ALDEC_SV.3.7.3

Rule Name

Prefer 'priority-case' to full_case pragma.

Base Rule

ALINT_SV.1.2.7.10

Phase

Parse

Message

Use 'priority-case' statements instead of specifying 'full_case' pragma.

Problem Description

For synthesis, a SystemVerilog **priority-case** statement is equivalent to specifying the **full_case** pragma. However, **priority-case** checks at run time that at least one **case** item expression matches the **case** selection expression. Verilog **case** statement with **full_case** pragma does not provide such check. So it is recommended to use **priority-case** statement to prevent mismatches between interpretation of the **case** statement by different tools and ensure that the designer intent is correct.

Level Recommendation 2

Checker Behaviour

The checker verifies **case** statements:

- if **case** statement is used with the **full_case** pragma => violation.

Example - Use of 'case' specific pragma ('full_case')

Description

In the following example the **full_case** pragma is used => violation.

Sample Code

```
 1:  module TOP (
 2:    output reg DO,
 3:    input      [2:0] DI,
 4:    input      [1:0] EN
 5:  );
 6:
 7:    always @(DI or EN) begin
 8:      case (EN) // synopsys full_case
 9:        2'b00: DO = DI[0];
10:        2'b01: DO = DI[1];
11:        2'b10: DO = DI[2];
12:      endcase
```



```
13:    end
14:
15:  endmodule
```

Use 'priority-case' statements instead of specifying 'full_case' pragma.

How to Fix

Use **priority-case** statement instead of the **full_case** pragma.

Fixed Code

```
 1:  module TOP (
 2:    output reg DO,
 3:    input      [2:0] DI,
 4:    input      [1:0] EN
 5:  );
 6:
 7:    always @(DI or EN) begin
 8:      priority case (EN)
 9:        2'b00: DO = DI[0];
10:        2'b01: DO = DI[1];
11:        2'b10: DO = DI[2];
12:      endcase
13:    end
14:
15:  endmodule
```


ALDEC_SV.3.7.4

Rule Name

Prefer 'unique0-case' to parallel_case pragma.

Base Rule

ALINT_SV.1.2.7.10

Phase

Parse

Message

Use 'unique0-case' statements instead of specifying 'parallel_case' pragma.

Problem Description

For synthesis, a SystemVerilog **unique0-case** statement is equivalent to specifying **parallel_case** pragma. However, **unique0-case** statement performs run-time uniqueness checks for **case** items expressions. Verilog-style **case** statement with **parallel_case** pragmas does not provide such checks. So it is recommended to use **unique0-case** statement to prevent mismatches between interpretation of the **case** statement by different tools and ensure that the designer intent is correct.

Level Recommendation 2

Checker Behaviour

The checker verifies **case** statements:

- if **case** statement is used with the **parallel_case** pragma => violation.

Example - Use of 'case' specific pragma ('parallel_case')

Description

In the following example the **parallel_case** pragma is used => violation.

Sample Code

```
 1:  module TOP (
 2:    output reg DO,
 3:    input      [2:0] DI,
 4:    input      [3:0] EN
 5:  );
 6:
 7:    always @(DI or EN) begin
 8:      (* parallel_case *)
 9:      casex (EN)
10:        3'b00x: DO = DI[0];
11:        3'b0x1: DO = DI[1];
12:        3'bx11: DO = DI[2];
```



```
13:      endcase
14:    end
15:
16:  endmodule
```

Use 'unique0-case' statements instead of specifying 'parallel_case' pragma.

How to Fix

Use **unique0-case** statement instead of **parallel_case** pragma.

Fixed Code

```
 1:  module TOP (
 2:    output reg DO,
 3:    input      [2:0] DI,
 4:    input      [3:0] EN
 5:  );
 6:
 7:    always @(DI or EN) begin
 8:       unique0 casex (EN)
 9:        3'b00x: DO = DI[0];
10:        3'b0x1: DO = DI[1];
11:        3'bx11: DO = DI[2];
12:      endcase
13:    end
14:
15:  endmodule
```

