Verification Continuum™

SpyGlass® lint

Rules Reference Guide

Version S-2021.09, September 2021

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

Contents

Preface

About This Book

The SpyGlass® lint Rules Reference Guide describes the SpyGlass rules that check HDL designs for coding style, language construct usage, simulation performance, and synthesizability.

Contents of This Book

The SpyGlass lint Rules Reference Guide consists of the following chapters:

Chapter	Describes...
Using the Rules in the SpyGlass lint Product	The usage concepts and rule parameters for the SpyGlass lint product
Rules in SpyGlass lint	The SpyGlass Lint Rules

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	*<sig-name>*
Message	Active low signal name '<sig-name>' must end with _X.
Message location	OUT <= IN;
Reworked example with message removed	OUT_X <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
`...` (Horizontal ellipsis)	Other options that you can specify

Using the Rules in the SpyGlass lint Product

It is recommended to use the Guideware (or other) rule goals to run specific rules in this product. These goals may be customized for your specific requirements. It is not recommend to use SpyGlass without selecting specific rules or goals. In case, if you use SpyGlass in such a manner, a default set of rules from the product are run. The **`fast`**, **`fullpolicy`**, **`strict`**, and **`verilint_compat`** parameters influence the rules enabled in this default set.

The SpyGlass® lint product has the following special usage features:

- *SpyGlass lint Rule Parameters*
- *SpyGlass lint Product Reports*
- *verilint Pragmas for SpyGlass lint Product*
- *Reporting Hierarchical Paths*
- *Rule Severity Classes*
- *Same or Similar Rules in Other SpyGlass Products*

SpyGlass lint Rule Parameters

This section provides detailed information on the SpyGlass lint product rule parameters.

You can set these parameters in both Atrenta Console and Tcl by using the following syntax:

```
set_parameter <parameter_name> <parameter_value>
```

For more information on setting the parameters, refer to the *SpyGlass Tcl Interface User Guide* and *Atrenta Console User Guide*.

NOTE: *Unless specified otherwise, all rule parameters are optional.*

allow_clk_in_condition

Specifies whether the *W122* rule traverses a combinational block present inside a sequential block.

By default, the *allow_clk_in_condition* parameter is set to **no**.

If you set this parameter to **yes**, the rule does not traverse a combinational block present inside a sequential block and does not report a violation in such cases.

Used by	*W122*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter allow_clk_in_condition yes`
Usage in goal/ source files	`-allow_clk_in_condition=yes`

NOTE: *The allow_clk_in_condition parameter is supported only for Verilog functionality.*

allviol

Specifies whether all or limited number of the *W257* (Verilog) rule messages should be reported.

By default, the **allviol** rule parameter is unset and the W257 rule reports only a maximum of 500 messages per design and a maximum of 20 messages per module.

You can set the value of the **allviol** rule parameter to **Yes** (to report all messages) or to **No** (to report limited messages). You can also set the value of this parameter to a comma or space-separated list of rules.

NOTE: *The actual number of messages reported for a rule is also governed by the* **set_option lvpr <rule-name>=<num>** *command.*

Used by	*W257*, *W314* (Verilog)
Options	yes, no, comma or space-separated list of rules
Default value	no
Example	
Console/Tcl-based usage	`set_parameter allviol yes`
Usage in goal/ source files	`-allviol="W257, W314"`

The following table lists the rules of SpyGlass products that use the **allviol** rule parameter:

ERC		
checkIOPinConnectedToNet	DisabledAnd	DisabledOr
FloatingInputs	FlopClockConstant	FlopClockUndriven
FlopClockX	FlopDataConstant	FlopDataUndriven
FlopDataX	FlopEConst	FlopSRConst
FlopSREX	LatchDataConstant	LatchDataUndriven
LatchDataX	LatchEnableConstant	LatchEnableUndriven
LatchEnableX	MuxSelConst	OutNotUsed

TristateConst		
Lint		
W257	W314	
MoreLint		
BitDataType-ML	InlineComment-ML	MultiOpInModule-ML
NoArray-ML	RedundantLogicalOp-ML	ResetPreventSRl-ML
UseBusWidth-ML	UseSRLPrim-ML	
OpenMORE		
CombLoop	ConstantComment	HardConst
Indent	NameLength	NoGates
NoTopGates	PortComment	PortGrpComment
SigHierName	SignalComment	TypeComment
VariableComment		
STARC		
STARC-1.1.2.6a	STARC-1.1.2.6b	STARC-1.2.1.3
STARC-1.5.1.1	STARC-1.5.1.2	STARC-1.5.1.5
STARC-1.6.2.2a	STARC-1.6.3.1	STARC-1.6.3.2
STARC-2.3.1.3	STARC-2.7.3.5	STARC-3.1.3.4b
STARC-3.5.6.3b		
STARC2002		
STARC02-1.1.2.6a	STARC02-1.1.2.6b	STARC02-1.2.1.3
STARC02-1.5.1.1	STARC02-1.5.1.2	STARC02-1.5.1.5
STARC02-1.6.2.2	STARC02-1.6.3.1	STARC02-1.6.3.2
STARC02-2.3.1.3	STARC02-2.7.3.5	STARC02-3.1.3.4b
STARC02-3.5.6.3b		
STARC2005		
STARC05-1.1.2.6a	STARC05-1.1.2.6b	STARC05-1.1.3.3e
STARC05-1.2.1.3	STARC05-1.5.1.1	STARC05-1.5.1.2
STARC05-1.6.2.2a	STARC05-1.6.3.1	STARC05-1.6.3.2
STARC05-2.3.1.3	STARC05-3.1.3.4b	STARC05-3.1.4.2

assume_driver_load

Use the **assume_driver_load** parameter in replacement of **checkDriverInModule** parameter.

By default, the **assume_driver_load** parameter is not set and the *W415* rule does not report a violation, if a net connected to output port of an instance is not driven inside that instance.

The **assume_driver_load** parameter performs the rule checking for the *W415* rule based on the following parameter values:

- **load:** Considers the inout and input pins as loads.
- **driver:** Considers inout and output pins as drivers.
- **both or yes:** Considers inout pins as both divers as well as loads. Output pins are considered as drivers. Input pins are considered as loads.

Used by	*W415*
Options	no, yes, both, driver, load
Default value	no
Default Value in GuideWare2.0	yes
Example	
Console/Tcl-based usage	`set_parameter assume_driver_load driver` `set_parameter assume_driver_load both`
Usage in goal/ source files	`-assume_driver_load=both`

avoid_port_signal

Specifies whether the *W498* rule reports a violation for port signals.

By default, the **avoid_port_signal** parameter is set to **no**.

Set this parameter to **yes** to enable the rule to not report a violation for port signals.

Used by	*W498*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter avoid_port_signal yes`
Usage in goal/ source files	`-avoid_port_signal=yes`

avoid_seq_logic

Specifies whether the *W527* rule reports violation for combinational **always** blocks.

By default, the ***avoid_seq_logic*** parameter is set to **no**.

If you set this parameter to **yes**, the rule reports violation for combinational **always** blocks and does not report violation for sequential **always** blocks.

Used by	*W527*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter avoid_seq_logic yes`
Usage in goal/ source files	`-avoid_seq_logic=yes`

avoid_synth_disable

Specifies whether the *W250* rule reports a violation for synthesizable disable constructs.

By default, the **avoid_synth_disable** parameter is set to **no**.

Set this parameter to **yes**, to enable the rule to not report violations for

synthesizable disable constructs for the following cases:

- Tasks
- Blocks
  - Sequential blocks (whose start-end is defined by begin-end)

NOTE: *For Parallel blocks (whose start-end is defined by fork-join), the rule does not waive violations as they are considered as non-synthesizable.*

Used by	*W250*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter` **`avoid_synth_disable`** `yes`
Usage in goal/ source files	**`-avoid_synth_disable`**`=yes`

casesize

Specifies the maximum permissible width of **`case`** selector expression and the maximum number of **`case`** clauses allowed as checked by the *W453* rule.

By default, the W453 rule flags case constructs where the **`case`** selector expression is wider than 16 bits and the number of **`case`** clauses is more than 20.

Specify the **`casesize`** rule parameter as follows:

```
set_parameter casesize '8-16'
```

Please note that the two values specified by the **`casesize`** rule parameter are separated by a hyphen.

Used by	*W453*
Options	<character-string>
Default value	"16-20"
Example	

Console/Tcl-based usage	`set_parameter casesize '8-16'`
Usage in goal/ source files	`-casesize="8-16"`

check_assign_pattern

Use this parameter to check assign pattern statements where the elements are assigned names or indexes.

Used by	*W164a*, *W164a_a*, *W164a_b*, *W164b*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_assign_pattern yes`
Usage in goal/ source files	`-check_assign_pattern=yes`

checkblocking

Specifies to check for blocking assignment statements only while rule checking with the *W415a* and *LINT_MULTIASSIGN_BLOCKING_SIG* rules.

By default, the **`checkblocking`** rule parameter is not set and the W415a rule checks for all kinds of multiple assignment within the same scope.

You can set the value of the **`checkblocking`** rule parameter to **`yes`** (to enable the mode) or to **`no`** (to disable the mode).

Used by	*W415a*, *LINT_MULTIASSIGN_BLOCKING_SIG*
Options	yes, no
Default value	no
Example	

Console/Tcl-based usage	`set_parameter checkblocking yes`
Usage in goal/ source files	`-checkblocking=yes`

checkfullbus

The following table describes the impact of this parameter on the SpyGlass lint product rules:

Rule	Rule behavior	
	When the checkfullbus parameter is set to `no`	**When the checkfullbus parameter is set to `yes`**
W240	Reports a violation if any bit of the input port is unread	Reports a violation only when the entire input port is unread
W120	Reports a violation if any bit of a variable is defined or declared but not used in the current scope of a module or an architecture	Reports a violation only when all bits of a variable are completely unused in the current scope of a module or an architecture
W528	Reports a violation if any bit of a variable is set but not read in the current scope of a module or an architecture	Reports a violation only when all bits of a variable are completely unread in the current scope of a module or an architecture

Used by	*W240*, *W120*, *W528*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter checkfullbus yes`
Usage in goal/ source files	`-checkfullbus=yes`

checkfullrecord

Specifies whether the *W456a*, *W123*, *W528*, *W120*, and *W240* rules should check for all the elements of a VHDL record.

By default, the **checkfullrecord** parameter is set to **no**.

Set this parameter to **yes** to not report violation for a record:

- W456a: if all elements of the record are not read.
- W123: if at least one element of the record is read and set.
- W528: if at least one element of the record is set and read.
- W120: if at least one element of the record is used.
- W240: if at least one element of the record is read.

Used by	*W456a*, *W123*, *W528*, *W120*, *W240*
Options	yes, no, comma separated list of rules
Default value	no
Example	
Console/Tcl-based usage	`set_parameter checkfullrecord yes`
Usage in goal/ source files	`-checkfullrecord=yes`

checkfullstruct

Specifies if the specified rule should report a violation for structs.

By default, this parameter is set to **no.**

Set the **checkfullstruct** parameter to **yes** to not report a violation for structs if at least one element of the struct is read.

Used by	*W240*, *W528*
Options	yes, no
Default value	no
Example	

Console/Tcl-based usage	`set_parameter checkfullstruct yes`
Usage in goal/ source files	`-checkfullstruct=yes`

check_bbox_driver

Specifies if the *W415* rule should consider the black-box as a driver.

By default, this parameter is set to **no** and the rule *W415* does not consider black-box as a driver. Set the **check_bbox_driver** parameter to **yes** to consider the black-box as a valid driver.

Used by	*W415*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_bbox_driver yes`
Usage in goal/ source files	`-check_bbox_driver=yes`

check_case_type

Specifies the case types on which user want to perform rule checking.

By default, the check_case_type parameter is set to all and W551 rule reports violation for all three case types, that is, full_case, unique case and priority case.

Set the value of the parameter to one or more of the following values as shown in the table:

Value	Description
priority	The W551 rule checks only for priority case.
unique	The W551 rule checks only for unique case.
full_case	The W551 rule checks only for full_case case.
all	The W551 rule checks for priority, unique, and full_case cases.

You can assign more than one of the above values to the parameter and rule checking is performed on all case types passed.

Consider the following example:

```
set_parameter check_case_type 'priority,unique'
```

In the above example, the W551 rule detects the presence of default clause in priority cases as well as unique cases.

Used by	*W551*
Options	priority, unique, full_case, all
Default value	all
Example	
Console/Tcl-based usage	`set_parameter check_case_type full_case`
Usage in goal/ source files	`-check_case_type="priority"`

check_complete_design

Specifies whether the *W391* and *W392* should check within the complete design.

By default, **check_complete_design** parameter is set to no and the rules **W391** and **W392** rules checks within the design unit.

You can set the value of **check_complete_design** parameter to **yes** to check within the complete design for violations.

Used by	*W391*, *W392*
Options	yes, no, comma separated list of rules
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_complete_design yes`
Usage in goal/ source files	`-check_complete_design=yes`

check_concat_max_width

Specifies the width to be considered by the *W110*, *W164a*, *W164a_a*, *W164a_b* and *W164b* Verilog rules after adding zero concatenated bits.

If the RHS expression is concatenated with zero bits, by default, no violation is reported when the width of the LHS expression is present between the width of the RHS expression without considering zero concatenated bits and the width of the RHS after adding zero concatenated bits.

When you set the **`check_concat_max_width`** parameter to **`yes`**, the RHS width is considered as the width after adding zero concatenated bits. That is, the violation is reported if the LHS width does not match the RHS width after adding zero concatenated bits.

Used by	*W110*, *W164a*, *W164a_a*, *W164a_b*, *W164b*
Options	yes, no, comma separated list of rules
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_concat_max_width yes`
Usage in goal/ source files	`-check_concat_max_width=yes`

checkconstassign

Specifies whether the *W415* rule should flag cases where the same constant

value is assigned multiple times.

By default, the **checkconstassign** rule parameter is set to **no**, and the W415 rule ignores cases where a net is driven multiple times by the same constant value or supply nets.

Set the value of the **checkconstassign** rule parameter to **yes** to flag cases where a net is driven multiple times by the same constant value or supply nets.

For example, the W415 rule does not report violation in the following case if the **checkconstassign** rule parameter is set to **no**:

```
assign out = 1'b0;
..
assign out = 1'b0;
```

Used by	*W415*
Options	yes, no
Default value	no
Default Value in GuideWare2.0	yes
Example	
Console/Tcl-based usage	`set_parameter checkconstassign yes`
Usage in goal/ source files	`-checkconstassign=yes`

check_const_selector

Specifies if the *W171* and *W226* rules check for the case selector.

By default, the **check_const_selector** parameter is set to **no**.

If you set this parameter to **yes**:

- The W171 rule checks weather case selector is constant or not. The W171 rule does not report any violation if case selector is a constant and case label belongs to one-hot operation, although case label is a variable.

- The W226 rule checks weather all case labels are one-hot. If so, the rule dose not report a violation even if the case selector is a constant.

Used by	*W171*, *W226*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_const_selector yes`
Usage in goal/ source files	`-check_const_selector=yes`

check_counter_assignment

Reports a violation for the counter assignments.

By default, the value of this parameter is set to **no** and the rule does not check counter type assignments. Set the **check_counter_assignment** parameter to **yes** to report violation for counter type of assignments.

When you set the value of the *check_counter_assignment* parameter to **turbo,** an assignment statement is considered as counter type assignment in the following conditions:

- Any constant (including based numbers or parameters) is added to or subtracted from a variable/signal on the RHS of the assignment.
- The variable on RHS matches the variable on the LHS expression, that is, the same variable is used on both sides, LHS and RHS, of assignment with same bit-width.

The following settings can be performed to report the counter type of assignments:

- Set the *check_counter_assignment* parameters to **yes**.
- **Verilog RTL:** Set the *check_static_value* parameter to **yes** to enable rule checking on static expressions and expressions that have a static part.

An assignment expression is considered as counter type assignment if the following conditions are met:

- 1 is added to a variable/signal on the RHS of the assignment
- The variable on RHS matches the variable on the LHS expression, that is, the same variable is used on both sides, LHS and RHS, of assignment with same bit-width

For example, the following assignments are considered as counter type assignments:

```
reg [5:0]a, b;
reg [1:0]c, d;
always @(*)
begin
a = a+1;
b = a+1;
a = 1+a;
c[1] = c[1] + 1;
d[1:0] = 1 + d[1:0];
end
```

The following assignments are **not considered** as counter type assignments (as explained with the in-line comments):

```
reg [5:0]a, b;
reg [1:0]c, d;
always @(*)
begin
a = a+b; //1 is not added to a variable
b[2:0]= a+1; //width of the variable on RHS does not
               match width of LHS

c[1:0]= (c[1]*2)+1; //addition of 1 is with an
                    expression instead of variable

d[1:0]= 2 + d[1:0]; //2 is added instead of 1

c[1:0] = 1 + d[1:0]; //RHS variable 'd' does not match
```

```
                                    with LHS variable 'c'
    end
```

- **VHDL RTL:** Set the *strict* parameter to **yes** to enable rule checking on static expressions and expressions that have a static part.
  The check_counter_assignment parameter is enabled only if the *nocheckoverflow* parameter is set to **no**.

NOTE: *Consider the following points:*

- *The check_counter_assignment parameter is applicable for the W164a (Verilog and VHDL) and W164b (VHDL only) rules.*
- *When check_counter_assignment is set to turbo, counters defined by the same option are not reported by the W164a rule.*

Used by	*W164a*, *W164a_a*, *W164a_b*, *W164b*, *W116*
Options	yes, no, turbo
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_counter_assignment yes`
Usage in goal/ source files	`-check_counter_assignment = yes`

check_counter_assignment_turbo

This parameter is deprecated. The functionality of the check_counter_assignment_turbo parameter is now covered by the turbo option of *check_counter_assignment* parameter.

checkDriverInModule

The **checkDriverInModule** has been deprecated and will be removed in the next major release. Use the *assume_driver_load* parameter in replacement of the **checkDriverInModule** parameter.

checknonblocking

Specifies to check for non-blocking assignment statements only while rule checking with the *W415a* and *LINT_MULTIASSIGN_NONBLOCKING_SIG* rules.

By default, the **checknonblocking** rule parameter is not set and the W415a rule checks for all kinds of multiple assignments within the same scope. It does not distinguish between blocking and nonblocking assignment statements.

You can set the value of the **checknonblocking** rule parameter to **yes** (to enable the mode) or to **no** (to disable the mode).

Used by	*W415a*, *LINT_MULTIASSIGN_NONBLOCKING_SIG*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter checknonblocking yes`
Usage in goal/ source files	`-checknonblocking=yes`

check_genvar

Specifies whether the *W116* rule should report violations for genvar variables.

By default, the **check_genvar** parameter is set to **no** and the W116 does not report a violation when the operands are genvar.

When this parameter is set to **yes**, the W116 rule reports violation for unequal length operands in the bit-wise logical, arithmetic, and ternary operators, although operands are genvar. This parameter is applicable for **Verilog** only.

Used by	*W116*
Options	yes, no
Default value	no
Example	

Console/Tcl-based usage	`set_parameter check_genvar yes`
Usage in goal/ source files	`-check_genvar=yes`

check_implicit_senselist

Specifies whether the *W502* rule reports violation for **`always @*`** and **`always_comb`**.

By default, the *W502* rule does not report violation for **`always @*`** and **`always_comb`**.

Set the value of the **`check_implicit_senselist`** parameter to **`yes`** to report violation for such cases.

Used by	*W502*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_implicit_senselist yes`
Usage in goal/ source files	`-check_implicit_senselist=yes`

check_initialization_assignment

By default, the *W415a*, *LINT_MULTIASSIGN_BLOCKING_SIG*, and *LINT_MULTIASSIGN_NONBLOCKING_SIG* rules report a violation for the following cases:

- when an entire vector is initialized with an initialization value and a subset of the vector is overwritten.
- when an entire packed struct is initialized with an initialization value and a subset of the struct is overwritten.

Set the value of the parameter to **`yes`** to disable the violations for the

above cases.

However, at this parameter value (**yes**), the rule reports a violation when:

- a vector is initialized with an initialization value and the entire vector is assigned again with another value.
- a violation when a packed struct is initialized with an initialization value and the entire packed struct is assigned again with another value.

Used by	*W415a*, *LINT_MULTIASSIGN_BLOCKING_SIG*, *LINT_MULTIASSIGN_NONBLOCKING_SIG*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_initialization_assignment yes`
Usage in goal/ source files	`-check_initialization_assignment=yes`

check_latch

Specifies whether the *W392* rule check for the non-edge triggered cases.

By default, this parameter is set to **no** and the *W392* rule reports the edge triggered cases only.

Set the **check_latch** parameter to yes to enable the *W392* rule to report the non-edge triggered cases also.

Used by	*W392*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_latch yes`
Usage in goal/ source files	`-check_latch=yes`

check_lrm_and_natural_width

Enables the *W164a*,*W164a_a* and *W164b* rules to calculate both LRM and natural width before reporting violation for the assignment statement. If any of these width matches with RHS, then no violation is reported for the assignment.

Used by	*W164a*, *W164a_a*, *W164b*
Options	*yes, no*
Default value	no
Example	
Console/Tcl-based example	`set_parameter check_lrm_and_natural_width yes`
Usage in goal/ source files	`-check_lrm_and_natural_width=yes`

check_natural_width_of_multiplication

Enables the *W164a_a* and *W164a_b* rules to calculate the width of multiplication expressions. Set the parameter to **`yes`**, to calculate the width of the multiplication expressions as per the natural width even if the **`nocheckoverflow`** parameter is set to **`yes`**.

Set the parameter to a comma/ space-separated list of rules to enable (the parameter) for the rules specified in the list.

Used by	*W164a*, *W164a_a*, *W164a_b*, *W164b*
Options	yes, no, comma/ space-separated list of rules
Default value	no
Example	
Console/Tcl-based example	`set_parameter check_natural_width_of_multiplication yes`
Usage in goal/ source files	`-check_natural_width_of_multiplication=yes`

check_natural_width_for_static

Enables the *W164a* rule to not report violations for static LHS depending on the *check_static_value* parameter.

Used by	*W164a*
Options	`yes, no`
Default value	`no`
Example	
Console/Tcl-based example	`set_parameter check_natural_width_for_static yes`
Usage in goal/ source files	`-check_natural_width_for_static=yes`

checkOperatorOverload

Specifies whether the *W116* rule reports inconsistent bit-width mismatch and the *W164a* and *W164b* rules evaluate width of the expression without considering overloaded operators.

This parameter is applicable for **VHDL** rules only.

- For the *W116* rule:

  By default, the **checkOperatorOverload** parameter is set to **yes** and the *W116* rule does not report inconsistent bit-width mismatch for the overloaded operators from non-IEEE packages. Set the value of this parameter to **no** to report violations in such cases.

- For the W164a, W164a_a and W164b rules:

  By default, the **checkOperatorOverload** parameter is set to **yes** and the *W164a*,*W164a_a* and *W164b* rules evaluate the width of the expression considering the overloaded operator. Set this parameter to **no** to evaluate width of the expression without considering overloaded operators.

Used by	*W116*, *W164a*, *W164a_a*, *W164b*
Options	yes, no, comma separated list of rules
Default value	yes
Example	

Console/Tcl-based usage	`set_parameter checkOperatorOverload no`
Usage in goal/ source files	`-checkOperatorOverload=no`

check_parameter

Enables the W362 rule to report violations if one of the operands is a parameter/localparam and the other operand is non-static, even if the value of the check_static_value parameter is set to no.

By default, the value of the parameter is set to no. In this case, the W362 rules does not report violation if one of the operands is a parameter/ localparam and the other operand is non-static.

Set the value of the parameter to yes to report violation for such cases.

Used by	*W362*
Options	yes, no, comma separated list of rules
Default value	yes
Example	
Console/Tcl-based usage	`set_parameter checkOperatorOverload no`
Usage in goal/ source files	`-checkOperatorOverload=no`

check_param_association

Specifies whether the *W146* rule reports violation for parameters in instantiation.

By default, the value of the parameter is set to **no** and the *W146* rule does not report violation for parameter instances.

Set the value of the parameter to **yes** to report such cases.

Used by	*W146*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_param_association yes`
Usage in goal/ source files	`-check_param_association=yes`

check_shifted_only

This parameter has been deprecated and its functionality is covered by the *handle_shift_op* parameter. The following table describes the corresponding values of both the parameters for your reference:

check_shifted_only Parameter Values	**Corresponding handle_shift_op Parameter Values**	**Description**
yes, both, <Rule-Name>	shift_both or <Rule-Name>	Shifted width is considered for the both, left and right shift expressions
left	shift_left	Shifted width is considered only for the left shift expressions
right	shift_right	Shifted width is considered for the right shift expressions only
no	no	Default behavior

check_shifted_width

Specifies whether to consider the natural width of the left operand for a left shift expression.

By default, this parameter is set to **no**. In this case, the rule considers the width of left operand of left shift expression if left operand is constant integer and right operand is non static. Set the value of the parameter to yes to consider the shifted width of the expression.

Used By	*W486*
Options	yes, no, comma separated list of rules
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter check_shifted_width yes`
Usage in goal/ source files	`-check_shifted_width=yes`

check_sign_extend

Checks for width mismatch due to sign extension in signed comparisons.

NOTE: *The check_sign_extend parameter is valid for the Verilog functionality of the W362 rule.*

By default, this parameter is set to **no** and the *W362* rule does not check for width mismatch due to sign extension in signed comparisons.

Set this parameter to **yes** or **W362** to check for width mismatch due to sign extension in signed comparisons.

Used By	*W362*
Options	yes, no, W362
Default Value	no
Example	

Console/Tcl-based usage	`set_parameter check_sign_extend yes`
Usage in goal/ source files	`-check_sign_extend=yes`

check_static_natural_width

Enables the specified rule to check for natural width of a static expression even when the value of the *nocheckoverflow* parameter is set to yes.

Used By	*W164a*, *W164b*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter check_static_natural_width yes`
Usage in goal/ source files	`-check_static_natural_width=yes`

check_static_value

Enables the *W416*, *W164a*, *W164a_a*,*W164a_b*, *W164b*, *W116*, and *W362* rules to report a violation for static expressions and non-static expressions that contain static expressions.

By default, this parameter is set to **no**. In this case, the specified rules do not report a violation for static expressions and non-static expressions that contain static expressions.

In addition, the parameter enables the *W484* rule to report a violation for static expressions and non-static expressions that contain static expressions.

NOTE: *The check_static_value parameter is valid for Verilog rules W416, W164a, W164b, W116, and W362, and VHDL rule W116.*

The following table describes the options supported by the **check_static_value** parameter. You can specify one or more of these as value to the parameter:

Parameter Value	Rule Behavior

only_const	Rules report violations for cases that involve static expressions. For example:
	`set_parameter check_static_value only_const`
	In the above example, all the affected rules report violations for cases that involve static expressions.
	Specify the value followed by the rule names to enable the specified rules to report violations for this parameter value. For example:
	`set_parameter check_static_value only_const,W164a,W116`
	In the above example, the *W164a* and *W116* rules report violations for cases that involve static expressions.

only_expr	Rules report violations for non-static expressions that contain a static part. For example:
	`set_parameter check_static_value only_expr`
	In the above example, all the affected rules report violations for non-static expressions that contain a static part.
	Specify the value followed by the rule names to enable the specified rules to report violations for this parameter value. For example:
	`set_parameter check_static_value only_expr,W164b`
	In the above example, the *W164b* rule reports a violation for non-static expressions that contain a static part.

only_static	Rule will not report violation for expressions, which dose not have any static part.
yes	All the affected rules report violations for all cases of width mismatch, involving static expressions and non-static expressions that contain a static part.
<rule_list>	Specified rules report violations for all cases of width mismatch, involving static expressions and non-static expressions that contain a static part. Set the value of the parameter to the list of rules: `set_parameter check_static_value W164b,W116,W362` This sets the parameter to **`yes`** for the W164b, W116 and W362 rules.
parameter	Rule will report violation for cases that involve parameters/ localparams
sized_based _number	Rule will report violation for cases that involve sized-based numbers.

Parameter Value	Rule Behavior
unsized_based_number	Rule will report violation for cases that involve unsized-based numbers
constant	Rule will report violations for cases that involve constants.
no	None of the rules report violations for static expressions or non-static expressions that contain static expressions.
Setting different values for rules	This parameter can be set to different value for different rules, for example: `set_parameter check_static_value only_expr,W116,W362,only_const,W164b,yes,W484,W164a` This sets the parameter to **`only_expr`** for the W116 and W362 rules, **`only_const`** for the W164b rule and **`yes`** for the W484, and W164a rules.

Used by	*W416*, *W164a*, *W164a_a*, *W164a_b*, *W164b*, *W116*, *W362*, *W484*
Options	yes, no, only_const, only_expr, only_static, parameter, sized_based_number, unsized_based_number, constant comma separated list of rules
Default value	no
Example	
Console/Tcl-based usage	`set_parameter check_static_value W164a, W116`
Usage in goal/ source files	`-check_static_value=only_expr`

Example 1

Consider the following example:

```
module TEST();
wire [7:0] sig1, sig2;

assign sig1 = 8'hFF + 2'b10;  //CASE A
assign sig1 = sig2 + 8'hFF;    //CASE B
```

```
endmodule
```

Assume that you want to specify the parameter only for the *W164a* rule.

In the above example, if the *check_static_value* parameter is set to **yes** or rule name, the *W164a* rule reports a violation in both CASE A and CASE B.

If you want to report a violation for CASE B, that is, for non-static expressions having a static part, set the *check_static_value* parameter as shown below:

```
set_parameter check_static_value only_expr,W164a
```

However, if you want to report a violation for CASE A, that is, for cases involving static expressions, set the *check_static_value* parameter as shown below:

```
set_parameter check_static_value only_const,W164a
```

Example 2

Consider the following parameter specification:

```
set_parameter check_static_value only_expr,parameter
```

In this case, the supported rules will have only only_expr and parameter as the value for check_static_value parameter.

Therefore, the supported rules do not report the violations for all static value options. The rules report only the expressions with the static/ nonstatic part.

checksyncreset

Specifies whether the *W392* rule flags or ignores synchronous resets.

By default, the **checksyncreset** rule parameter is set to **yes** and the *W392* rule considers synchronous resets for rule checking.

You can set the value of the **checksyncreset** rule parameter to **yes** (to consider synchronous resets) or to **no** (to ignore synchronous resets).

Used by	*W392*
Options	yes, no
Default value	yes

Example	
Console/Tcl-based usage	`set_parameter checksyncreset no`
Usage in goal/ source files	`-checksyncreset=no`

check_sequential

Specifies whether the *W71* rule checks sequential block for missing default.

By default, the **check_sequential** rule parameter is set to **no** and the W71 rule does not check for missing default in case construct inside sequential block.

You can set the value of the **check_sequential** rule parameter to **yes** (to see violations of sequential block) or to **no** (to ignore violations of sequential block).

Used by	*W71*
Options	yes, no
Default value	no
Example	
Console/Tcl-based example	`set_parameter check_sequential yes`
Usage in goal/ source files	`-check_sequential=yes`

check_temporary_flop

Specifies whether the *W336* rule reports a violation for temporary flip-flops.

By default, the **check_temporary_flop** parameter is set to **no** and the W336 rule does not report violations for temporary flip-flops.

Set the value of the **check_temporary_flop** parameter to **yes** to report temporary flip-flops.

Used by	*W336*
Options	yes, no
Default value	no
Example	
Console/Tcl-based example	`set_parameter check_temporary_flop yes`
Usage in goal/ source files	`-check_temporary_flop=yes`

check_unsign_overflow

Specifies whether the *W164a*, *W164a_a* and *W164b* rules suppress overflow in unsigned signals due to sign extension.

By default, this parameter is set to **no**. This indicates the rule suppresses the overflow when sign extension is used for unsigned signals in addition or subtraction operation.

If you set this parameter to **yes** or *<rule-name>*, the rule does not suppress the overflow when sign extension is used for unsigned signals in addition or subtraction operation.

NOTE: *Irrespective of the value of this parameter, the rule suppresses the overflow when sign extension is used for signed signals in addition or subtraction operation.*

Used by	*W164a*, *W164a_a*, *W164b*
Options	yes, no, *<rule-name>*
Default value	no
Example	
Console/Tcl-based example	`set_parameter check_unsign_overflow yes`
Usage in goal/ source files	`-check_unsign_overflow=yes`

concat_width_nf

Enables the W164a, W164a_a and W164a_b rules to use new algorithm to calculate the width of concatenations ignoring the self-determined nature of concatenation items.

By default, this parameter is set to no. Set this parameter to yes to enable the rules to calculate the width of concatenations ignoring the self determined nature of the concatenation items.

Used by	*W164a*,*W164a_a*, *W164a_b*
Options	yes, no
Default value	no
Example	
Console/Tcl-based example	`set_parameter concat_width_nf no`
Usage in goal/ source files	`-concat_width_nf=no`

considerInoutAsOutput

Enables the *W122* rule to consider the inout port as output port. Therefore, the signal, which is passed to that port is considered as set.

By default, the value of the *considerInoutAsOutput* parameter is **`no`**. In this case, the *W122* rule considers the inout port as both input and output. Therefore, the signal that is passed to the inout port is considered as both read and set.

NOTE: *This parameter is applicable only for VHDL.*

Set the value of the parameter to **`yes`** to consider the inout port as output port.

Used by	*W122*
Options	yes, no
Default value	no
Example	

Console/Tcl-based example	`set_parameter considerInoutAsOutput yes`
Usage in goal/ source files	`-considerInoutAsOutput=yes`

consider_lrm_for_div

Specifies if the *W164a* and *W164b* rules should consider the width of the division to be the maximum width of the operands.

By default, the **consider_lrm_for_div** parameter is set to **no**.

Set the **consider_lrm_for_div** parameter to **yes** to enable the rules to calculate the width of the division to be the maximum width of the operands (LRM width) when **nocheckoverflow** parameter is set to **no**.

Example:

```
a[2:0] = b[2:0] / c[3:0]; // without parameter on, no violation as width of RHS is 3// but with parameter on,width of RHS is 4 and hence violation.
```

Used by	*W164a*, *W164b*
Options	yes, no
Default value	no
Example	
Console/Tcl-based example	`set_parameter` **`consider_lrm_for_div`** `yes`
Usage in goal/ source files	`-`**`consider_lrm_for_div`**`=yes`

consider_sub_as_add

Specifies if the *W416*, *W164a* and *W164b* rules should calculate the width of a-b/b-1 as a+b/b+1 respectively.

By default, when the **nocheckoverflow** parameter is set to **no**, the *W164a* and *W164b* rules consider the width of expression like b -1 as the width of b when there is no underflow due to -ve operator. However, this

behavior is limited to a simple - ve operation, where this is a part of arithmetic expression other than +/-.

Set the **consider_sub_as_add** parameter to **yes** to enable the rules to calculate the width of a-b / b-1 as a+b/ b+1 respectively.

Used by	*W164a*, *W164b*, *W416*
Options	yes, no
Default value	no
Example	
Console/Tcl-based example	`set_parameter consider_sub_as_add yes`
Usage in goal/ source files	`-consider_sub_as_add=yes`

control_sig_detection_nf

Defines the control signal detection to be done by the *W164a*, *W164a_a* and *W164a_b* rules.

By default, the **control_sig_detection_nf** parameter is set to **no**.

If you set this parameter to yes, the rule treats a signal to be a control signal only if it is used in a conditional operator or control binary operator within the scope of currently processing statement. The rule does not consider the signal to be a control signal even if it is used as a control signal in other places of the module (that is, the statements other than the current statement).

Used by	*W164a*, *W164a_a*, *W164a_b*
Options	yes, no
Default value	no
Example	
Console/Tcl-based example	`set_parameter control_sig_detection_nf yes`
Usage in goal/ source files	`-control_sig_detection_nf=yes`

datapath_or_control

Specifies whether the *W164a*, *W164a_a* and *W164a_b* rules should check the specified type of signals.

By default, the **datapath_or_control** parameter is set to **no** and the rule reports violations for all type of signals.

Set this parameter to **datapath**, **control**, or **all** to specify the type of signals to be checked.

The following are the details of the possible values of this parameter:

- **no**: This is the default value. Existing violation message are reported.
- **all**: The rules checks for all type of signals (both datapath and control signals) at the LHS of assignment, which is the same as the existing behavior. The violation messages mention the type of the signals on the LHS of the assignment (datapath or control).
- **datapath**: Only datapath signals are checked and violation messages mention the type of the signals (datapath signal) at the LHS and type of the assignment. The rules identifies the datapath when a wire declaration is used in the assignment where the signal is used as a datapath.
- **control**: Only control signals are checked and violation messages mention the type of signals (control signal) at the LHS and type of the assignment.

Used by	*W164a*, *W164a_a*
Options	*datapath, control, all, no*
Default value	no
Example	
Console/Tcl-based example	`set_parameter datapath_or_control all`
Usage in goal/ source files	`-datapath_or_control=all`

do_not_run_W71

Specifies whether to run *W71* (Verilog) rule or not. This parameter is

provided to block duplicate violation in case the *W69* and *W71* rules are run in mixed mode (Verilog, VHDL).

By default, the **do_not_run_W71** rule parameter is set to **no** and *W71* (Verilog and VHDL) rule is run.

You can set the value of the **do_not_run_W71** parameter to **yes** to block *W71* (Verilog) rule.

Used by	*W71*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter do_not_run_W71 yes`
Usage in goal/ source files	`-do_not_run_W71=yes`

disable_latch_crossing

Enable the *LINT_abstract01* rule to enable the rule to stop traversal at latches.

By default, the value of the parameter is no. In this case, the rule does not stop the traversal at latches.

Set the value of the parameter to yes to enable the rule to stop traversal at latches.

Used by	*LINT_abstract01*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter disable_latch_crossing yes`
Usage in goal/ source files	`-disable_latch_crossing=yes`

disable_rtl_deadcode

The *disable_rtl_deadcode* parameter specifies whether to report a violation for disabled code in loops and conditional (**if** condition, ternary operator) statements.

NOTE: *The disable_rtl_deadcode parameter is valid for both Verilog and VHDL.*

By default, this parameter is set to **no** and the respective rules report a violation for disabled code in loops and conditional (**if** condition, ternary operator) statements.

Set this parameter to **yes** to disable violations for disabled code in loops and conditional (**if** condition, ternary operator) statements.

Used By	*W116*, *W164a*,*W164a_a*, *W164b*, *W164c*, *W362*, *W484*, *W486*, *W456a*, *W416*
Options	yes, no, *<rule-name>*
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter disable_rtl_deadcode yes`
Usage in goal/ source files	`-disable_rtl_deadcode=yes`

disable_signal_usage_report

Use this parameter to disable the generation of data for the specific rule for the SignalUsageReport report.

By default, the value of this parameter is no. In this case, all rule related data is reported in the SignalUsageReport report.

Used By	*W241*, *W528*
Options	yes, no, *<rule-name>*
Default Value	no
Example	

Console/Tcl-based usage	`set_parameter disable_signal_usage_report {W241}`
Usage in goal/ source files	`-disable_signal_usage_report=W241`

dump_array_bits

Specified whether the *W123* rule should report the multi-dimensional signals/ bits information.

By default, the dump_array_bits parameter is set to **no**.

Set this parameter to **yes**, to enable the W123 rule to include multi-dimensional signals/bits information in the violation message.

This parameter is applicable to VHDL only.

Used By	*W123*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter dump_array_bits yes`
Usage in goal/ source files	`-dump_array_bits=yes`

fast

Specifies to suppress the synthesis of the source RTL description. If this parameter is set, the following rules are not run:

W18	*W287a*	*W287b*	*W323*	*W336*	*W391*	*W392*
W396	*W401*	*W402*	*W402b*	*W415*	*W438*	*W448*
W541						

By default, the **fast** rule parameter is not set. Thus, the source RTL description is synthesized and the above rules are run.

You can set the value of the **fast** rule parameter to **Yes** to (suppress synthesis) or to **No** (to synthesize the source RTL description).

For the *W414* and *W428* rules, the **fast** rule parameter is not set by default and the synthesis version of these rules is run. Set the value of the **fast** parameter to **Yes** to suppress the synthesis and execute the RTL version of the *W414* and *W428* rules.

Used by	*W18*, *W287a*, *W287b*, *W323*, *W336*, *W391*, *W392*, *W396*, *W401*, *W402*, *W402b*, *W414*, *W415*, *W428*, *W438*, *W448*, *W541*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter fast yes`
Usage in goal/ source files	`-fast=yes`

filter_mark_open

Use this parameter to not report violations on those port connections that have been explicitly left open in the RTL.

By default, the value of the parameter is no.

Used by	*W287a*, *W287b*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter filter_mark_open yes`
Usage in goal/ source files	`-filter_mark_open=yes`

force_handle_shift_op

Enables the *W164a*, *W164a_a* and *W164a_b* rules to honor the handle_shift_op parameter even if the nocheckoverflow parameter is set.

By default, the **force_handle_shift_op** parameter is set to **no**.

Set this parameter to **yes** to enable the W164a rule to honor *handle_shift_op* parameter value even if the *nocheckoverflow* parameter is set to **yes**.

Used By	*W164a*,*W164a_a*, *W164a_b*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter force_handle_shift_op yes`
Usage in goal/ source files	`-force_handle_shift_op=yes`

flag_complex_nodes

Enables the *W479* rule to report a violation when step is written as a complex node.

By default, the **flag_complex_nodes** parameter is set to **no**.

Set this parameter to **yes** to enable the *W479* rule to report a violation when step is written as a complex node.

NOTE: *In the above case, the W479 rule reports a violation only when both the flag_complex_nodes and strict parameters are set to yes.*

Used By	*W479*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter flag_complex_nodes yes`
Usage in goal/ source files	`-flag_complex_nodes=yes`

flag_only_instance_ports

Enables the *W446* rule to report violation for module output ports that are connected to instance input pin.

Used By	*W446*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter flag_only_instance_ports yes`
Usage in goal/ source files	`-flag_only_instance_ports=yes`

group_by_module

Specifies whether the *W154* rule groups the violations based on the module and reports them in turbo mode.

The default value of the parameter is set to **`no`**.

Set the parameter value to **`yes`** to enable the rule to group the violations based on the module and report them in turbo mode.

Used By	*W154*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter group_by_module yes`
Usage in goal/ source files	`-group_by_module=yes`

handle_case_select

Enables you to specify the maximum width of case selector that the supported rule should process.

You can specify any integer value as an input to the parameter. By default, the parameter is set to 64. Set the value to -1 to enable the supporting rules to process case statements with any case selector width.

If the width of a case selector is greater than the value specified for the handle_case_select parameter, the rules ignore the case statement and reports a violation.

Used by	*W69*, *W71*
Options	Any integer value
Default value	64
Example	
Console/Tcl-based usage	`set_parameter handle_case_select 23`
Usage in goal/ source files	`-handle_case_select=23`

handle_equivalent_drivers

Enables the *W415* rule to report violations when a net is simultaneously driven by two identical instances of two input gates of type BUFF, NOT, OR, NOR, AND, or, NAND and both instances are driven by same input signals.

By default, the value of the parameter is set to no. In this case, the rule will not report violation for above scenario.

Set the value of the parameter to yes to report violations when a net is simultaneously driven by two identical instances of two input gates of type BUFF, NOT, OR, NOR, AND, or, NAND and both instances are driven by same input signals.

For example, consider the following assignment statements:

```
assign a = b & c;
```

```
assign a = b & c;
```

The W415 rule will report violation for above assignments when the value of the *handle_equivalent_drivers* parameter is set to yes.

Used By	*W415*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter handle_equivalent_drivers yes`
Usage in goal/ source files	`-handle_equivalent_drivers=yes`

handle_large_bus

The **handle_large_bus** parameter specifies whether to check large arrays.

By default, this parameter is set to **no** and the respective rules do not process large arrays.

Set this parameter to **yes** to process large arrays (greater than 50,000) and report violation if not used correctly. If you set this parameter to *<rule-name>*, the **handle_large_bus** parameter performs the check for that particular rule only.

Used By	*W111*, *W120*, *W123*, *W240*, *W241*, *W446*, *W494*, *W495*, *W528*
Options	yes, no, *<rule-name>*
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter handle_large_bus yes`
Usage in goal/ source files	`-handle_large_bus=yes`

handle_large_expr

Specifies whether the *W484* rule should handle large expressions.

By default, if the RHS expression is a binary expression and each term is a

concatenation expression padded with zero bits, the rule checks for overflow while considering zero padded bits. That is, violation is not reported if the actual width of the expression can be accommodated in zero padded bits.

When the **handle_large_expr** parameter is set to **yes**, the *W484* rule limits this checking for expressions containing up to 25 terms and for larger expressions, the LRM width is considered.

Used By	*W484*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter handle_large_expr yes`
Usage in goal/ source files	`-handle_large_expr=yes`

handle_lrm_param_in_shift

Specifies whether the *W164a*, *W164a_a* and *W164a_b* rules should honor *use_lrm_width* parameter in the width calculations of shift expressions.

By default, the **handle_lrm_param_in_shift** parameter is set to **no**.

Set this parameter to **yes** to enable the *W164a* rule to honor the *use_lrm_width* parameter in the width calculations of shift expressions.

Used By	*W164a*, *W164a_a*, *W164a_b*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter handle_lrm_param_in_shift yes`
Usage in goal/ source files	`-handle_lrm_param_in_shift=yes`

handle_shift_op

Specifies whether to consider shifted or non-shifted width of a shift expression.

By default the **handle_shift_op** parameter is set to no. In this case, both the shifted and non-shifted widths of a left or right shift expression are compared and no violation is reported if any of these widths matches the LHS expression. But, the rule does not calculate shifted width, if the RHS of the shift expression is non-static.

NOTE: *This parameter is effective when the nocheckoverflow parameter is set to* ***no****.*

The **handle_shift_op** parameter can be set to **shift_left**, **shift_right**, **shift_both**, **no_shift**, **no**, no_shift_forced, or comma separated list of rule names, to compare shifted and non-shifted widths of a left or right shift expression.

The behavior of these options is as follows:

- **shift_left**: Shifted width is considered for those left-shift expressions that have the RHS of shift expression as either static or non-static. For example:

  ```
  set_parameter handle_shift_op shift_left
  ```

  In this case, the parameter will be set to **shift_left** for all the rules that use this parameter.

  Or

  ```
  set_parameter handle_shift_op shift_left, W164b, W110
  ```

  In this case, the parameter will be set to **shift_left** for the specified rules only.

- **shift_right**: Shifted width is considered for those right-shift expressions that have the RHS of shift expression as static. For example:

  ```
  set_parameter handle_shift_op shift_right
  ```

  Or

  ```
  set_parameter handle_shift_op shift_left,W164a,W164b,W110
  ```

  For the *W164a*,*W164a_a*, *W164a_b*, and *W164b* rules, the **handle_shift_op shift_right** option calculates the shifted width of the right shift operator when both the operands are non-static, where the width is calculated as the width of the LHS operand minus the

maximum value of RHS operand. If this width is less than 1, the width is considered as 1.

- **shift_both**: Shifted width is considered for both left and right shift expressions, as stated above for parameter values **shift_left** and **shift_right**. For example:

  ```
  set_parameter handle_shift_op shift_both
  ```

  Or

  ```
  set_parameter handle_shift_op shift_both,W164b,W110
  ```

- **Comma separated list of rules**: This sets the value of the parameter to **shift_both** for the specified rules. For example:

  ```
  set_parameter handle_shift_op W164b,W110,W164a
  ```

- **no_shift**: Non-shifted width is considered for both left and right shift expressions, that is, the width of the LHS of shift expression is considered. This is applicable only for those shift expressions that have the LHS as non static and the RHS as static. For example:

  ```
  set_parameter handle_shift_op no_shift
  ```

  Or

  ```
  set_parameter handle_shift_op no_shift,W164b,W164a
  ```

- **no**: Default behavior.
- **no_shift_forced**: Checks for the non-shifted width of the shift operator regardless of the staticness of the RHS and LHS expression of the shifted operation.
- This parameter can also be set to different values for different rules, for example:

  ```
  set_parameter handle_shift_op
  shift_left,W110,W164b,shift_both,W164a
  ```

  This sets the parameter to **shift_left** for the W110 and W164b rules and **shift_both** for the W164a rule.

NOTE: *This parameter is applicable for both Verilog and VHDL.*

Used By	*W164a*, *W164a_a*,*W164a_b*, *W164b*, *W110*
Options	*shift_left, shift_right, shift_both, no_shift, no, no_shift_forced, rule name*
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter handle_shift_op shift_left`
Usage in goal/ source files	`-handle_shift_op=shift_left`

handle_static_caselabels

Specifies whether the *W263* rule should ignore static case labels having less width than the case selector.

By default, the **handle_static_caselabels** parameter is set to **no**.

Set this parameter to **yes** to ignore violations for static case labels, which are of less width than the width of case selector when case labels are parameters.

Used By	*W263*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter handle_static_caselabels yes`
Usage in goal/ source files	`-handle_static_caselabels=yes`

handle_zero_padding

Performs leading zero expansion and truncation of RHS of an assignment. This parameter also considers the width specified by the width specifier of sized-base numbers when handling zero padding.

In addition, this parameter performs leading zero expansion and truncation of LHS of an assignment for the *W416*, *W164a*,*W164a_a*, *W164a_b* and *W164b* rules. This is performed only if the LHS of the assignment is static.

If the parameter is set to **yes**, the *W164a_a*, *W164a_b*, *W164a* and *W164b* rules perform zero expansion or truncation if the RHS of the assignment is a concatenation and the first element is padded with zero.

Used By	*W416*,*W110*, *W164a*, *W164a_a*, *W164a_b*, *W164b*, *W362*
Options	yes, no, <rule_name>
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter handle_zero_padding yes`
Usage in goal/ source files	`-handle_zero_padding=yes`

ignore_auto_function_return

Allows the *W499* rule to ignore name of automatic functions to be set in all branches of function body. That is, if this parameter is set while the function is automatic, and all other outputs are assigned in all branches, then no violation is reported.

By default, the value of this parameter is no. In this case, the *W499* rule reports the automatic function name when the return-by-function-name is not specified in all the branches in the function body.

Used By	*W499*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_auto_function_return yes`
Usage in goal/ source files	`-ignore_auto_function_return=yes`

ignore_bitwiseor_assignment

Specifies whether the *W415a*, *LINT_MULTIASSIGN_BLOCKING_SIG*, and *LINT_MULTIASSIGN_NONBLOCKING_SIG* rules report violations for a **bitwise or** assignment inside the **for** loop that is assigned multiple times on the same line.

By default, the value of the **ignore_bitwiseor_assignment** is set to **no** and the rules report a violations for a **bitwise or** assignment inside the **for** loop that is assigned multiple times on the same line.

Set the value of the parameter to **yes** to not to report violations in such cases.

Used by	*W415a*, *LINT_MULTIASSIGN_BLOCKING_SIG*, *LINT_MULTIASSIGN_NONBLOCKING_SIG*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_bitwiseor_assignment yes`
Usage in goal/ source files	`-ignore_bitwiseor_assignment=yes`

ignoreCellName

Specifies the modules that should be ignored by the *W336* rule.

By default, the **ignoreCellName** parameter is not set.

Set this parameter to a comma separated list of PERL regular expressions containing names of the modules that should be ignored by the *W336* rule.

Used By	*W336*
Options	Comma separated list of PERL regular expressions
Default Value	""
Example	

Console/Tcl-based usage	`set_parameter ignoreCellName "module_1,mod_*"`
Usage in goal/ source files	`-ignoreCellName="module_1,mod_*"`

ignore_concat_expr

Specifies whether the *W156* rule should ignore violations for the reverse connections which is an expression that concat operation is involved.

By default, the value of the **ignore_concat_expr** is set to **no**.

Set the value of this parameter to **yes** to not report violations for the reverse connections which is an expression that concat operation is involved.

Used by	*W156*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter` **`ignore_concat_expr`** `yes`
Usage in goal/ source files	**`-ignore_concat_expr`**`=yes`

ignore_cond_having_identifier

Specifies whether the *W159* rule should ignore the constants defined by parameters, localparams or constant integers.

By default, the value of the **ignore_cond_having_identifier** is set to **no** and the *W159* rule considers constants defined by parameters, localparams or constant integers, for rule checking.

Set the value of this parameter to **yes** to not report violation for such constants.

Used by	*W159*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_cond_having_identifier yes`
Usage in goal/ source files	`-ignore_cond_having_identifier=yes`

ignore_const_selector

Specify this parameter to ignore case statements where the case selector is a static expression.

By default, the value of the parameter is no. In this case, the rule considers case statements where the case selector is a static expression.

Set the value of the parameter to yes to ignore such case statements.

Used by	*W171*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_const_selector yes`
Usage in goal/ source files	`-ignore_const_selector=yes`

ignore_counter_with_same_width

Enables the *W164a*, *W164a_a*, and *W164a_b* rules to not report violation for cases where any constant value added/subtracted from different signal on both LHS and RHS with same width is treated as a counter.

Used by	*W164a*, *W164a_a*, *W164a_b*
Options	yes, no, only_by_one

Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_counter_with_same_width yes`
Usage in goal/ source files	`-ignore_counter_with_same_width=yes`

ignore_forloop_indexes

Specifies whether the *W116* rule should ignore the **for** expressions that contain index variables of **for** loops.

By default, the value of the **ignore_forloop_indexes** is set to **no** and the *W116* reports the **for** expressions that contain index variables of **for** loops.

Set the value of this parameter to **yes** to not report violation for such cases.

Used by	*W116*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_forloop_indexes yes`
Usage in goal/ source files	`-ignore_forloop_indexes=yes`

ignore_function_init

Specifies whether the *SigVarInit* rule should ignore violations for the functions that consist initializing values.

By default, the value of the **ignore_function_init** is set to **no** and the *SigVarInit* reports violations for the functions that consist initializing values.

Set the value of this parameter to **yes** to not report violation for such

cases.

Used by	*SigVarInit*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_function_init yes`
Usage in goal/ source files	`-ignore_function_init=yes`

ignore_genvar

Specifies whether the *W121* rule should ignore violations for variable shadowed by **genvar** variables.

By default, the value of the **ignore_genvar** parameter is set to **no** and the *W121* rule reports violations for variables shadowed by **genvar** variables.

Set the value of this parameter to **yes** to not report violation for such cases.

Used by	*W121*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_genvar yes`
Usage in goal/ source files	`-ignore_genvar=yes`

ignore_generatefor_index

Specifies whether the *W226* rule should ignore for **case** statements inside **generte-for** blocks if the case selector is a loop variable of **generate-for** loop.

By default, the value of the **ignore_generatefor_index** parameter is set to **no**.

Set the value of this parameter to **yes** to ignore violations for **case** statements inside **generte-for** blocks, if the case selector is a loop variable of **generate-for** loop.

Used by	*W226*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_generatefor_index yes`
Usage in goal/ source files	`-ignore_generatefor_index=yes`

ignore_greybox_drivers

Enables the *W415* rule to do not consider greybox as a valid driver,

By default, the value of the parameter is no. Default value is no. If the value of the parameter is set to yes, the *W415* rule will not consider greybox as a valid driver.

Used by	*W415*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_generatefor_index yes`
Usage in goal/ source files	`-ignore_generatefor_index=yes`

ignore_hier_scope_var

Specifies whether the *W123* rule should ignore user-defined data types.

By default, the value of the **ignore_hier_scope_var** is set to **no** and the *W123* rule reports all user-defined data types.

Set the value of this parameter to **yes** to not report any violation for user-defined data types. This parameter is applicable for Verilog only.

Used by	*W123*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_hier_scope_var yes`
Usage in goal/ source files	`-ignore_hier_scope_var=yes`

ignore_in_ports

Specifies whether the *W111* considers input and inout ports for rule checking.

By default, the value of the **ignore_in_ports** is set to no and the *W111* rule considers input and inout ports for rule checking.

Set the value of the parameter to **yes** to ignore input and inout ports for rule checking.

Used by	*W111*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_in_ports yes`
Usage in goal/ source files	`-ignore_in_ports=yes`

ignore_inout

Specifies whether the *W210* rule should ignore the inout ports.

By default, the value of the **ignore_inout** is set to **no** and the W210 rule considers inout ports for rule checking.

Set the value of this parameter to **yes** to not report any violation for the inout ports.

Used by	*W210*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_inout yes`
Usage in goal/ source files	`-ignore_inout=yes`

ignore_integer_constant_labels

Use this parameter to ignore violations for constant case labels of integer type.

Used by	*W263*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_integer_constant_labels yes`
Usage in goal/ source files	`-ignore_integer_constant_labels=yes`

ignore_interface_locals

Enables the *W424* rule to not report violations for functions inside an interface that are written to variables/signals in the interface scope.

Enables the *W425* rule to not report violations for functions inside an interface that are read to variables/signals in the interface scope.

By default, the value of the parameter is **no** and the *W424* and the *W425*

rules report violations for functions inside an interface that are written/read to variables/signals in the interface scope respectively.

Set the value of the parameter to **yes** to ignore such violations.

Used by	*W424*, *W425*
Options	**yes, no**
Default value	**no**
Example	
Console/Tcl-based usage	`set_parameter ignore_interface_locals yes`
Usage in goal/source files	`-ignore_interface_locals=yes`

ignore_if_case_statement

Enables the *W415a* rule to ignore violations for signals which are inside if/else or case statement.

By default, the value of the parameter to no. In this case, the rule reports violations for signals which are inside if/else or case statement.

Set the value of the parameter to yes to ignore such violations.

Used by	*W415a*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_if_case_statement yes`
Usage in goal/source files	`-ignore_if_case_statement=yes`

ignore_local_variables

Specifies whether the *W336* rule should ignore variables defined inside the sequential blocks.

By default, the value of the **ignore_local_variables** parameter is

set to **no** and the *W336* rule reports violations for variables (left hand side of the assignment) defined inside the sequential block.

Set the value of this parameter to **yes** to ignore violations for such signals.

Used by	*W336*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_local_variables yes`
Usage in goal/ source files	`-ignore_local_variables=yes`

ignore_multi_assign_in_genforblock

Specifies whether the *W415a* rule should report violations for assign assignments in **generate for** blocks.

By default, the value of the **ignore_multi_assign_in_genforblock** parameter is set to **no** and the *W415a* rule reports violations for assign assignments in **generate for** blocks.

Set the value of this parameter to **yes** to not report violations for assign assignments in **generate for** blocks.

Used by	*W415a*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_multi_assign_in_genforblock yes`
Usage in goal/ source files	`-ignore_multi_assign_in_genforblock=yes`

ignore_mult_and_div

Enables the *W116* rule to ignore violations for multiplication (*) and division (/) operations.

By default, the value of the parameter is set to no. In this case, the W116 rule reports violations for multiplication (*) and division (/) operations.

Set the value of the parameter to yes to ignore violations for such cases.

Used by	*W116*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_mult_and_div yes`
Usage in goal/ source files	`-ignore_mult_and_div=yes`

ignore_nonstatic_counter

Enables the *W116* rule to ignore violations ignore violations for non-static single-bit argument in additions and subtractions.

By default, the value of the parameter is set to **`no`**.

Set the value of the parameter to **`yes`** to ignore violations for non-static single-bit argument in additions and subtractions.

Used by	*W116*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_nonstatic_counter yes`
Usage in goal/ source files	`-ignore_nonstatic_counter=yes`

ignore_parameters

Use this parameter to not report violations for the type parameters.

By default, the *W576* rule reports violation for the type parameters.

Used by	*W576*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_parameters yes`
Usage in goal/ source files	`-ignore_parameters=yes`

ignore_signed_expressions

Enables the W116 rule to report violations on statements where both sides are signed expressions.

By default, the value of the parameter is set to no. In this case, the W116 rule reports violations for the signed expressions.

Set the value of the parameter to yes to ignore violations for such cases.

Used by	*W116*
Options	no, yes
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_signed_expressions yes`
Usage in goal/ source files	`-ignore_signed_expressions=yes`

ignore_unique_and_priority

Enables the *W71* rule to ignore violations for the unique and priority cases. By default, the value of the parameter is set to no.

Set the value of the parameter to yes to ignore violations for the unique,

unique0, and priority cases.

Used by	*W71*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_unique_and_priority yes`
Usage in goal/ source files	`-ignore_unique_and_priority=yes`

ignoreModuleInstance

Specifies whether the *W123* rule flags or ignores the unset variables that are only used in instance port mapping.

By default, the **ignoreModuleInstance** rule parameter is set to **no** and the W123 rule flags the unset variables that are also used in instance port mapping.

Set the value of the **ignoreModuleInstance** rule parameter to **yes** to ignore the unset variables that are only used in instance port mapping.

Used by	*W123*
Options	yes, no
Default value	no
Default Value in GuideWare2.0	yes
Example	
Console/Tcl-based usage	`set_parameter ignoreModuleInstance yes`
Usage in goal/ source files	`-ignoreModuleInstance=yes`

ignore_nonBlockCondition

Specifies whether the *W415a* and *LINT_MULTIASSIGN_NONBLOCKING_SIG*

rules report or ignore non-blocking assignments to a variable inside a conditional construct, if the variable is also assigned in a non-blocking manner before the conditional construct.

By default, the parameter is not set and the rules report conditions.

You can set the value of the **ignore_nonBlockCondition** parameter to **yes** (to ignore such conditions) or to **no** (to flag such conditions).

Used by	*W415a*, *LINT_MULTIASSIGN_NONBLOCKING_SIG*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_nonBlockCondition yes`
Usage in goal/ source files	`-ignore_nonBlockCondition=yes`

ignore_macro_to_nonmacro

Enables the W121 rule to ignore name matching for macro to non-macro names. However, the W121 rule still checks for the macro to macro name matching.

Used by	*W121*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_macro_to_nonmacro yes`
Usage in goal/ source files	`-ignore_macro_to_nonmacro=yes`

ignore_multi_assign_in_forloop

Specifies whether the *W415a*, *LINT_MULTIASSIGN_BLOCKING_SIG*, and *LINT_MULTIASSIGN_NONBLOCKING_SIG* rules check inside a for loop for

multiple assignments on the same line.

When this parameter is set to *Yes,* the rules do not report a violation for a signal in the for loop that is assigned multiple times on the same line.

By default, the value of the **ignore_multi_assign_in_forloop** parameter is set to **no** and the rules report a violation for multiple assignments on the same line.

Consider the following example:

```
reg a, b, c;
  for(i = 0; i < 2; i = i + 1)
    begin
      a = a + 1;
    end
```

In the above example, the rules, in default mode, reports a violation for signal **a** assigned in the for loop. But, when the **ignore_multi_assign_in_forloop** parameter is set to **Yes,** no violation is reported for multiple assignments on the same line in the for loop.

Used by	*W415a*, *LINT_MULTIASSIGN_BLOCKING_SIG*, *LINT_MULTIASSIGN_NONBLOCKING_SIG*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_multi_assign_in_forloop yes`
Usage in goal/ source files	`-ignore_multi_assign_in_forloop=yes`

ignore_macro_to_nonmacro

Enables the W121 rule to ignore name matching for macro to non-macro names. However, the W121 rule still checks for the macro to macro name matching.

Used by	*W121*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_macro_to_nonmacro yes`
Usage in goal/ source files	`-ignore_macro_to_nonmacro=yes`

ignore_nonstatic_leftshift

Specifies whether the*W164a* and *W164b* rules should ignore assignments where the RHS is a left shift operator.

By default, the **`ignore_nonstatic_leftshift`** parameter is set to **`no`**.

Set this parameter to **`yes`** enable the rules to ignore assignments where the RHS is a left shift operator and both the operands are non-static.

Used by	*W164a*, *W164b*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_nonstatic_leftshift yes`
Usage in goal/ source files	`-ignore_nonstatic_leftshift=yes`

ignore_parameter

Specifies whether the *W224* rule should ignore violations for parameters and local parameters.

By default, the **`ignore_parameter`** parameter is set to **`no`**.

Set this parameter to **`yes`** to enable the rule to not report any violations

for parameters and local parameters.

Used by	*W224*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_parameter yes`
Usage in goal/ source files	`-ignore_parameter=yes`

ignore_priority_case

Enables the following rules to ignore violations for the priority cases:

• *W187*	• *W398*	• *W171*
• *W226*	• *W332*	• *W337*
• *W551*		

By default, the value of the parameter is no. In this case, the rule reports violations for priority cases.

Set the value of the parameter to yes to ignore violations for the priority cases.

Used by	*W187 W398 W171 W226 W332 W337 W551*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_priority_case yes`
Usage in goal/ source files	`-ignore_priority_case = yes`

ignore_reinitialization

Specifies whether the *W415a*,*LINT_MULTIASSIGN_BLOCKING_SIG*, and *LINT_MULTIASSIGN_NONBLOCKING_SIG* rules should ignore violations for re-initialization assignments inside the **for** loops.

By default, the **ignore_reinitialization** parameter is not set and the rules report violations for re-initialization when the *check_initialization_assignment* parameter is set to **yes**. Set this parameter to **yes** to ignore violations for re-initialization assignments inside the **for** loops.

Used by	*W415a*, *LINT_MULTIASSIGN_BLOCKING_SIG*, *LINT_MULTIASSIGN_NONBLOCKING_SIG*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_reinitialization yes`
Usage in goal/ source files	`-ignore_reinitialization=yes`

ignore_scope_names

Specifies whether the *W121* rule should ignore rule checking on scope names.

By default, the **ignore_scope_names** parameter is not set and the *W121* rule checks for scope names.

Set this parameter to **yes** to ignore rule checking on scope names.

Used by	*W121*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_scope_names yes`
Usage in goal/ source files	`-ignore_scope_names=yes`

ignore_typedefs

Ignores all type definitions from rule checking.

By default, the value of the *ignore_typedefs* parameter is set to no. In this case, rules mentioned in the table below consider type definitions for rule checking.

Set the value of the parameter to yes to ignore all type definitions.

Used by	*W443*,*W444*,*W467*,*W19*,*W341*,*W342*,*W343*,*W491*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_scope_names yes`
Usage in goal/ source files	`-ignore_scope_names=yes`

ignoreSeqProcess

Specifies whether the *W122* rule checks inside the sequential process block.

By default, the value of the ignoreSeqProcess parameter is set to **`no`** and the *W122* rule ignores the signals read inside clock condition, but reports violation for the signals read inside other conditions in sequential block.

You can set the value of the parameter to **`yes`** to ignore all signals read inside sequential process block.

Used by	*W122*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignoreSeqProcess yes`
Usage in goal/ source files	`-ignoreSeqProcess=yes`

ignore_wildcard_operators

Use this parameter to not report *W467* rule violations reported for RHS side of wild operators, ==? and !=?, as well as inside operator, that is, LHS 'inside' RHS.

By default, the value of the parameter is set to no.

Used by	*W467*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter ignore_wildcard_operators yes`
Usage in goal/ source files	`-ignore_wildcard_operators=yes`

latch_effort_level

Specifies the effort that SpyGlass should make while doing traversal.

By default, the value of the parameter is -1. In this case, the *LINT_abstract01* rule traverses through all the latches during rule checking.

Specify an integer value from -1 to MAX_INT value to specify the effort used by SpyGlass during the traversal.

Used by	*LINT_abstract01*
Options	integer value from -1 to MAX_INT
Default value	-1
Example	
Console/Tcl-based usage	`set_parameter latch_effort_level 5`
Usage in goal/ source files	`-latch_effort_level=5`

limit_task_function_scope

Use this parameter to ignore rule checking if a signal is declared with function/task scope as well as with module scope.

Used by	*W121*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter limit_task_function_scope yes`
Usage in goal/ source files	`-limit_task_function_scope=yes`

modport_compat

Specifies whether the *W497* and *W498* rules should evaluate mod ports that are used as hier scope variable in a port connection.

By default, the **`modport_compat`** parameter is set to **`no`**.

Set the value of the parameter to **`yes`** to enable the rules to not evaluate mod ports that are used as hier scope variables in a port connection.

Used by	*W497*, *W498*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter modport_compat yes`
Usage in goal/ source files	`-modport_compat=yes`

nocheckoverflow

Specifies how the related rules calculate the bit width.

By default, the nocheckoverflow rule parameter is set to no, and the affected rules calculate the natural width of the expression.

You can set the value of nocheckoverflow parameter to yes or rule name(s)

to calculate the width of the expression where the expression is considered to be independent of context, following the self-determined expression rules defined in the LRM.

Used by	**Lint**: *W116*, *W164a*, *W164a_a*,*W164a_b*, *W164b*, *W164c*, *W486*, *W110*, *W263*, *W362*, *W164a_a*, *W164a_b*, *W416* **MoreLint**: AsgnOverflow-ML **STARC**: STARC-2.1.3.1, STARC-2.8.1.6, STARC-2.10.3.1, STARC-2.10.3.2a, STARC-2.10.3.2b, STARC-2.10.3.2c, STARC-3.2.3.2 **STARC2002**: STARC02-2.1.3.1, STARC02-2.8.1.6, STARC02-2.10.3.1, STARC02-2.10.3.2a, STARC02-2.10.3.2b, STARC02-2.10.3.2c, STARC02-3.2.3.2 **STARC2005**: STARC05-2.1.3.1, STARC05-2.8.1.6, STARC05-2.10.3.1, STARC05-2.10.3.2a, STARC05-2.10.3.2b, STARC05-2.10.3.2c, STARC05-3.2.3.2
Options	yes, no, rule name
Default value	no
Default Value in GuideWare2.0	yes
Example	
Console/Tcl-based usage	`set_parameter nocheckoverflow yes` `set_parameter nocheckoverflow W164c`
Usage in goal/ source files	`-nocheckoverflow=yes` `-nocheckoverflow=W164a`

not_used_signal

Specifies a comma-separated name list of signals to be ignored by the *W528* rule.

By default, the value of this parameter is **`nil`**, which means that rule checking by the rule will be done for all the signals.

You can specify full names or regular expressions. The following example matches the signal named **`date_signal`** and all signal names starting with **`bus_`**:

`"data_signal,bus_*"`

Used by	*W528*
Options	comma-separated list of signals
Default value	nil
Example	
Console/Tcl-based usage	`set_parameter not_used_signal "data_signal,bus_*"`
Usage in goal/source files	`-not_used_signal="data_signal,bus_*"`

no_strict

Use this parameter to ignore the rule list to run in the strict mode.

By default, the value of this parameter is no. In this case, the specified rules run in the strict mode.

Set the value of the parameter to yes to ignore the strict mode.

Used by	All Lint Rules
Options	Comma-separated list of rules
Default value	`" "`
Example	
Console/Tcl-based usage	`set_parameter no_strict yes`
Usage in goal/source files	`-no_strict=yes`

process_complete_condop

Enables the rule checking on both the operands of the condop assignment.

By default, the value of this parameter is no.

Set the value of the parameter to yes to enable the rule checking on both operands of the condop assignment.

NOTE: *The process_complete_condop parameter is valid even when the value of the nocheckoverflow parameter is set to yes.*

Used by	*W164a*, *W164a_a*, *W164a_b*, *W164b*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter process_complete_condop yes`
Usage in goal/ source files	`-process_complete_condop=yes`

report_all_connections

Specifies whether the *W156* rule should report a violation for all reverse order buses in the same module.

By default, the **report_all_messages** parameter is set to **no**.

Set this parameter to **yes** to report a violation for all reverse order buses in the same module.

Used by	*W156*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter report_all_connections yes`
Usage in goal/ source files	`-report_all_connections=yes`

report_all_messages

Specifies whether the *W122*, *W456*, and *W502* rules report the bit index information.

By default, the **report_all_messages** parameter is set to **no**.

Set this parameter to **yes** or rule name(s) to report the bit information in

the violation messages of the *W122*, *W456*, and *W502* rules.

NOTE: *If this parameter is set, the violation messages are also modified and old waivers do not work. See the specific rule messages for details.*

Used by	*W122*, *W456*, *W502*
Options	yes, no, comma separated list of rules
Default value	no
Example	
Console/Tcl-based usage	`set_parameter report_all_messages yes`
Usage in goal/ source files	`-report_all_messages=yes`

report_blackbox_inst

Specifies whether the *W110* rule reports violation for port width mismatch for black box instances.

By default, the value of this parameter is set to **no**. In this case, the W110 rule does not report violation for port width mismatch for black box instances.

Set the value of the parameter to **yes** to report violation for port width mismatch for black box instances.

Used by	*W110*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter report_blackbox_inst yes`
Usage in goal/ source files	`-report_blackbox_inst=yes`

report_cast

Used to specify whether the newly introduced message should be reported instead of the previous one.

The W372 rule specifically reports for '$cast' PLI tasks or functions if this parameter is set to yes.

Used by	*W372*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter report_cast yes`
Usage in goal/ source files	`-report_cast=yes`

report_hierarchy

Specifies whether the *W287b* rule should report rtl module name and hierarchy in the violation message.

By default, the parameter is set to **no**.

Set the parameter value to **yes** to report rtl module name and hierarchy in the violation message.

Used by	*W287b*
Options	**yes, no**
Default value	**no**
Example	
Console/Tcl-based usage	`set_parameter report_hierarchy yes`
Usage in goal/ source files	`-report_hierarchy=yes`

report_if_blocks_only

Specifies whether the *W193* rule reports the use of semicolon with the **if**, **else**, and **else-if** statements.

By default, the value of the **report_if_blocks_only** parameter is set to **no**.

Set the value of this parameter to **yes** to report violations only when a semicolon is used with the **if**, **else**, and **else-if** statements.

Used by	*W193*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter report_if_blocks_only yes`
Usage in goal/ source files	`-report_if_blocks_only=yes`

report_inter_nba

Specifies whether the *W280* rule reports the inter-nonblocking assignment delay.

By default, the value of this parameter is set to **no**. In this case, the W280 rule does not report inter-nonblocking assignment delay.

Set the value of the parameter to **yes** to report violation for such cases.

Used by	*W280*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter report_inter_nba yes`
Usage in goal/ source files	`-report_inter_nba=yes`

report_global_param

Enables the *W175* rule to report violations for unused global parameter declarations.

By default, the value of the parameter is no. In this case, the rule does not report violation for unused global parameter.

Set the value of the parameter to yes to report violation for unused global parameter.

Used by	*W175*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter report_global_param yes`
Usage in goal/ source files	`-report_global_param=yes`

reportLibLatch

Specifies whether latches inferred from both .sglib and .lib libraries are reported, as checked by the *W18* rule.

By default, the **reportLibLatch** rule parameter is set to **no**, and the W18 rule does not report library latches.

Set the **reportLibLatch** rule parameter to **yes** to report library latches.

Used by	*W18*
Options	yes, no
Default value	no
Example	

Console/Tcl-based usage	`set_parameter reportLibLatch yes`
Usage in goal/ source files	`-reportLibLatch=yes`

reportconstassign

The *reportconstassign* parameter specifies whether the *W116* rule checks for constants whose width is less than the operand.

NOTE: *The reportconstassign parameter is valid for Verilog only.*

By default, the *reportconstassign* parameter is set to **no** and the *W116* rule does not report a violation for constants whose width is less than the other operand.

If you set this parameter to **yes**, the *W116* rule reports a violation in such cases.

Used By	*W116*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter reportconstassign yes`
Usage in goal/ source files	`-reportconstassign=yes`

report_nonblocking_in_error

Specifies whether the *W164a*, *W164a_a*, *W164a_b* and *W164b* rules should report violations (with **Error** severity) for nonblocking assignments.

By default, the **report_nonblocking_in_error** parameter is set to **no**.

Set the value of the parameter to **yes** to enable the rules to report

violations (with **Error** severity) for nonblocking assignments.

Used By	*W164a*, *W164a_a*, *W164a_b*, *W164b*
Options	**yes, no**
Default Value	**no**
Example	
Console/Tcl-based usage	`set_parameter report_nonblocking_in_error yes`
Usage in goal/ source files	`-report_nonblocking_in_error=yes`

report_only_bitwise

Specifies whether the *W116* rule should report violations only for bit wise operations.

By default, the **report_only_bitwise** parameter is set to **no**.

Set the value of the parameter to **yes** to report violations only for bit wise operations in the *W116* rule.

Used By	*W116*
Options	**yes, no**
Default Value	**no**
Example	
Console/Tcl-based usage	`set_parameter report_only_bitwise yes`
Usage in goal/ source files	`-report_only_bitwise=yes`

report_only_from_one_hierarchy

Specifies whether the *W164a* rule should report only one violation per assignment.

By default, the **report_only_from_one_hierarchy** parameter is set

to **no**.

By default, multiple violations may be reported on the same assignments due to different hierarchies. Set the value of the parameter to yes to report only one violation per assignment.

Used By	*W164a*
Options	**yes, no**
Default Value	**no**
Example	
Console/Tcl-based usage	`set_parameter report_only_from_one_hierarchy yes`
Usage in goal/ source files	`-report_only_from_one_hierarchy=yes`

report_only_overflow

Specifies whether the *W504* rule reports violations only for integers or constant integers where the width of port expression is greater than the width of port.

By default, the **report_only_overflow** parameter is set to **no**.

If you set this parameter to **yes**, the *W504* rule reports violations only for integers or constant integers where the width of port expression is greater than the width of port.

For example:

```
module top ();
blk1 u_blk1 (.a(3));//Violation with report_only_overflow
blk2 u_blk2 (.b(3)); //No violation with report_only_overflow
endmodule
module blk1 (input a);
...
endmodule
module blk2 (input [2:0] b);
...
endmodule
```

However, there are some special cases while calculating width of port expression:

- If port expression contains shift operator, the width of expression will be width of LHS operand. For example:

  `(32 >> 2) - width of 32`

- If port expression contains arithmetic operators (+, -, *, /, %), the width of expression will be maximum width out of LHS and RHS operands. For example:

  `(4+16) - width of 16`

Used By	*W504*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter report_only_overflow yes`
Usage in goal/ source files	`-report_only_overflow=yes`

report_port_net

Specifies if the *W154* rule reports nets/ports declared without a type.

By default, the **`report_port_net`** parameter is set to **`no`**.

If you set this parameter to **`yes`**, the *W154* rule reports nets/ports declared without a type.

Used By	*W154*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter` **`report_port_net`** `yes`
Usage in goal/ source files	`-`**`report_port_net`**`=yes`

report_semicolon

Specifies if the *W193* rule reports the use of extra semicolon.

By default, the **report_semicolon** parameter is set to **no**.

If you set this parameter to **yes**, the *W193* rule reports the use of extra semicolon.

Used By	*W193*
Options	yes, no
Default Value	no
Example	
Console/Tcl-based usage	`set_parameter report_semicolon yes`
Usage in goal/ source files	`-report_semicolon=yes`

report_struct_name_only

By default, rule *W123* gives the violation messages for each violation bit of the struct variables.

Set the value of the parameter to yes to enable the W123 rule to report one violation for a struct variable.

NOTE: *this parameter is applicable only for Verilog.*

Used by	*W123*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter report_struct_name_only yes`
Usage in goal/ source files	`-report_struct_name_only=yes`

reportsimilarassgn

Specifies whether the *W415a*, *LINT_MULTIASSIGN_BLOCKING_SIG*, and *LINT_MULTIASSIGN_NONBLOCKING_SIG* rules should check assignments in which RHS value is the same.

By default, the **reportsimilarassgn** parameter is set to **no** and the rules do not report violation if the RHS value in an assignment is the same as the RHS value in the previous assignment.

Set the **reportsimilarassgn** rule parameter to **yes** to report such cases.

NOTE: *The W415a rule does not report a violation when a mutually exclusive if conditions exist and the conditional expression contains the inside operator, even if the reportsimilarassign parameter is set to yes.*

Used by	*W415a*, *LINT_MULTIASSIGN_BLOCKING_SIG*, *LINT_MULTIASSIGN_NONBLOCKING_SIG*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter reportsimilarassgn yes`
Usage in goal/ source files	`-reportsimilarassgn=yes`

reset_gen_module

Enables you to specify module or cell names for which the *W402b* rule will not report violation if the reset signal is gated by a instance of a given module.

Used by	*W402b*
Options	comma-separated list of reset generator design unit or cell names
Default value	no
Example	

Console/Tcl-based usage	`set_parameter reset_gen_module slave2`
Usage in goal/ source files	`-reset_gen_module=slave2`

set_message_severity

Sets the severity of violation messages reported by the *W210* rule.

By default, the **`set_message_severity`** rule parameter is set to **`no`** and the *W210* rule reports all unconnected ports with message severity as **`Warning`**.

Set the **`set_message_severity`** parameter to **`yes`** if you want to report unconnected input or inout ports with messages severity as **Error**.

Used by	*W210*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter set_message_severity yes`
Usage in goal/ source files	`-set_message_severity=yes`

show_connected_net

The **`show_connected_net`** parameter enables the *W415* rule to report adjacent nets to a fan-in terminal of a multi-driven net and help in preventing hierarchical waiver breaks.

By default, the **`show_connected_net`** parameter is set to **`no`**.

Set the value of the **`show_connected_net`** parameter to **`yes`** to report adjacent nets to a fan-in terminal of a multi-driven net

Used By	*W415*
Options	**`yes, no`**

Default Value	**no**
Example	
Console/Tcl-based usage	`set_parameter show_connected_net yes`
Usage in goal/ source files	`-show_connected_net=yes`

sign_extend_func_names

The *sign_extend_func_names* parameter lists the function names that are used for sign extension. This parameter enables the respective rules to recognize VHDL sign extension functions.

NOTE: *The sign_extend_func_names parameter is valid for VHDL only.*

The respective rules calculate the width of extend functions as per the **const** extension argument specified in the argument list.

Used By	*W116*, *W164a*, *W164a_a*, *W164b*
Options	comma-separated list of function names
Default Value	EXTEND, resize
Example	
Console/Tcl-based usage	`set_parameter sign_extend_func_names EXTEND, resize`
Usage in goal/ source files	`-sign_extend_func_names="EXTEND",“resize”`

simplesense

Specifies for the *W122* rule to flag the signals that are read in an **always** construct and being set in blocking assignment statement but are not present in the sensitivity list.

By default, the **simplesense** rule parameter is not set. Thus, if a signal is assigned some value through a blocking assignment and is later read in same **always** construct, the W122 rule ignores that signal provided its bit index is not a variable.

You can set the value of the **simplesense** rule parameter to **yes** or **1** (to enable the mode) or to **no** or **0** (to disable the mode).

Used by	*W122*
Options	yes, no
Default value	no
Example	
Console/Tcl-based usage	`set_parameter simplesense yes`
Usage in goal/ source files	`-simplesense=yes`

strict

Specifies the strict rule-checking mode.

The effect of the **strict** rule parameter on SpyGlass lint product rules is as follows:

Rule	Language	strict rule parameter	
		Not set	**Set**
W19	Verilog	Does not report truncation of zeroes for binary based numbers	Reports truncation of zeroes for binary based numbers
W69	Verilog	Does not check case constructs in sequential always constructs	Checks case constructs in sequential always constructs
W71	Verilog	Does not report missing **default** clause in fully specified **case** constructs. Does not report violations for unique, unique0 and priority case usage.	Reports missing **default** clause in fully specified **case** constructs also. Reports violations for unique, unique0 and priority case usage if *ignore_unique_and_priority* is set to no.
	VHDL	Does not report missing **default** clause in fully specified **case** constructs	Reports missing **default** clause in fully specified **case** constructs also

Rule	Language	strict rule parameter	
		Not set	**Set**
W116	Verilog	• Ignores Addition (+) and Multiplication (*) operations. • Reports on Subtraction (-), Division (/), or Modulus (%) operations only if the width of the right operand is greater than the width of the left operand. • Ignores ternary operators.	• Checks Addition (+) and Multiplication (*) operations also. • Reports whenever there is a width mismatch irrespective of the relative sizes of the operands. • Checks for ternary operators also.
	VHDL	• Ignores Addition (+) and Multiplication (*) operations. • Reports on Subtraction (-), Division (/), or Modulo (mod), Remainder (rem) operations only if the width of the right operand is greater than the width of the left operand.	• Checks Addition (+) and Multiplication (*) operations also. • Reports whenever there is a width mismatch of the operands.
W122	Verilog	Does not report memories and delay variables that are read in the **always** construct but are not present in the sensitivity list.	Reports memories and delay variables that are read in the **always** construct but are not present in the sensitivity list.
	VHDL	Reports a violation for a signal that is read in the **process** block and initialized to a value outside the **process** block but is not completely present in the sensitivity list.	Does not report a violation for a signal that is read in the **process** block and initialized to a value outside the **process** block.
W156	VHDL	Does not report the reverse connections involving static expressions.	Reports such reverse connections also.
W164a, *W164a_a*,*W164a_b*	Verilog	Assignments in which the RHS expression contains **wire** or **reg** objects are reported.	All assignments are reported. Integer port signals in the RHS expression are also reported.
	VHDL	Does not report violation in case of addition and subtraction expression, if any one operand is a constant and its width is less.	Reports violation case of addition and subtraction expression, if any one operand is a constant and its width is less.

Rule	Language	strict rule parameter	
		Not set	**Set**
W164b	Verilog	Assignments in which the RHS expression contains **`wire`** or **`reg`** objects are reported.	All assignments are reported. Integer port signals in the RHS expression are also reported.
	VHDL	Does not report violation in case of addition and subtraction expression, if any one operand is a constant and its width is less.	Reports violation case of addition and subtraction expression, if any one operand is a constant and its width is less.
W226	VHDL	Does not check inside subprogram descriptions	Check inside subprogram descriptions also
W263	Verilog	Not run even if specified	Run if specified
W287a	Verilog	Does not flag when the net connected to an instance input port is coming out of a black box instance	Flags such instance input ports
W323	Verilog	Ignores busholders as drivers	Considers busholders as drivers
W337	Verilog	Ignores string variables, which are used as case items	Reports violation for string variables, which are used as case items
W341	Verilog	Not run even if specified	Run if specified
W342	Verilog	Not run even if specified	Run if specified
W343	Verilog	Not run even if specified	Run if specified
W362	Verilog	Does not flag when the right or left expression is a constant, a based number, or a parameter	Reports violation for the cases of width mismatch, if the right or left expression is a constant, a based number, or a parameter, when set along with the *check_static_value* parameter. Also, violation will not be reported for width mismatch in condition of "for-loop"
W398	Verilog	Does not report violation for the non-static case labels	Reports violation for the non-static case labels
W415	Verilog, VHDL	Does not consider instances, bus-holders and top-level inout ports as drivers.	Considers instances, bus-holders and top-level inout ports as drivers also.

Rule	Language	strict rule parameter	
		Not set	**Set**
W443	Verilog	Does not check the presence of X value in the **`default`** statement of the **`case`** construct.	Checks the **`default`** statement also.
W444	Verilog	Does not check the presence of Z value in the **`default`** statement of the **`case`** construct.	Checks the **`default`** statement also.
W456	Verilog	Does not report for constants present in sensitivity lists.	Report for constants present in sensitivity lists also.
W456a	Verilog	Does not report for constants present in sensitivity lists.	Report for constants present in sensitivity lists also.
W479	Verilog	Does not report violation if the step variable is not present in the initialization or condition part.	Reports violation if the step variable is not present in the initialization or condition part.
W481a	Verilog	Reports violation only if none of the step variable is present in the condition statement.	Reports violation for all the step variables that are not present in the condition statement.

W484	Verilog	Does not flag the following types of assignments:
		`a = a + 1;`
		`a = b + 1;`
		Where `b` is of same or lesser bit-width than `a`.
		Also, the rule does not flag for the following assignment:
		`assign result = {b[0],b} + {c[0],c};`
			Reports such assignments also.

W490	VHDL	Does not check for constant control expressions in subprogram descriptions.	Checks for constant control expressions inside subprogram descriptions also.
W494	Verilog, VHDL	Reports violation for completely unused ports only.	Reports violation for partially unused ports also.
W494a	VHDL	Reports violation for completely unused ports only.	Reports violation for partially unused ports also.

Rule	Language	strict rule parameter	
		Not set	**Set**
W494b	VHDL	Reports violation for completely unused ports only.	Reports violation for partially unused ports also.
W504	Verilog	Does not report violation for integers used in expressions containing scalar values, such as, **4'h4-2**, **4'h4-i**, where **i** is an integer variable.	Reports violation for integers used in expressions containing scalar values.

You can set the value of the **strict** rule parameter to **Yes** (in the Atrenta Console) or **1** (in the batch mode) to enable the mode or to **No** (in the Atrenta Console) or **0** (in the batch mode) to disable the mode).

You can also set the value of **strict** rule parameter to a comma or space separated list of rules. In such a case, checking will be performed only for the rules specified in the list.

Used by	*W19*, *W69*, *W71*, *W116*, *W122*, *W156*, *W164a*, *W164b*, *W226*, *W263*, *W287a*, *W323*, *W337*, *W341*, *W342*, *W343*, *W362*, *W415*, *W443*, *W444*, *W456*, *W456a*, *W481a*, *W484*, *W490*, *W494*, *W494a*, *W494b*, *W504*
Options	yes, no, comma/space-separated list of rules
Default value	no
Default Value in GuideWare2.0	W343,W342
Example	
Console/Tcl-based usage	`set_parameter strict yes`
Usage in goal/ source files	`-strict=yes`

traverse_function

Specifies whether the inputs should be considered as read when being read inside a function.

By default, the **traverse_function** parameter is set to **no** and the

W123 rule considers the inputs as unread when being read inside the function block.

Set this parameter to **yes** to enable the *W123* rule to consider such inputs as read. This parameter is applicable to **VHDL** only.

Used by	*W123*
Options	yes, no
Default value	no
Example	
Console \ Tcl-based example	`set_parameter traverse_function yes`
Usage in goal/ source files	`-traverse_function=yes`

treat_concat_assign_separately

Use this parameter to report violation for each bucket assignment in unpacked array separately. For a packed array, a violation is reported for the whole array.

Used by	*W164a*, *W164a_a*, *W164a_b*, *W164b*
Options	yes, no
Default value	no
Example	
Console \ Tcl-based example	`set_parameter treat_concat_assign_separately yes`
Usage in goal/ source files	`-treat_concat_assign_separately=yes`

treat_latch_as_combinational

Specifies to treat combinational block inferring latch as combinational block while rule checking with the *W336* and *W414* rules.

By default, the **treat_latch_as_combinational** rule parameter is not set and the W336 and W414 rules treat combinational block inferring

latch as sequential block.

You can set the value of the **treat_latch_as_combinational** rule parameter to **yes** (to enable the mode) or to **no** (to disable the mode).

Used by	*W336*, *W414*
Options	yes, no, comma separated list of rules
Default value	no
Default Value in GuideWare2.0	yes
Example	
Console \ Tcl-based example	`set_parameter treat_latch_as_combinational yes`
Usage in goal/ source files	`-treat_latch_as_combinational=yes`

use_carry_bit

Enables the *W116* (Verilog) rule to get the width after considering the carry bit of addition.

By default, the **use_carry_bit** parameter is set to **no** and the width is taken as maximum of the two operands for a binary expression having plus and minus operators.

Set this parameter to **yes** or **<rule-name>** to get width after considering the carry bit of addition. No violation is reported, even using this parameter, for sub-expressions of a binary expression if all terms have the same width and all operators are either plus or minus.

For *W164a* and *W164b* rules: By default, when the **nocheckoverflow** parameter is set **no**, the width of addition and subtraction is considered as the width of the maximum value that the expression can hold. Set the **use_carry_bit** parameter (with default value as W164a, W164b) to **no** to calculate the width as the LRM, that is the maximum width of the operand.

Example:

```
a[2:0] = b[2:0] + c[2:0]; // by default violation as
```

```
RHS width is 4 // but with parameter off for W164a,
width of RHS is 3 and hence no violation.
```

Used by	*W116*, *W164a*, *W164b*,
Options	yes, no, rule name
Default value	no
Example	
Console \ Tcl-based example	`set_parameter use_carry_bit yes`
Usage in goal/ source files	`-use_carry_bit=yes`

use_lrm_width

Specifies whether the related rules should consider the LRM width, which is 32 bits, for unsized based numbers and integer constants.

By default, the *use_lrm_width* parameter is set to **no**, and the natural width of integer constants is considered, which is **log2(N)+1**.

Set this parameter to **yes** to consider the LRM width, which is 32 bits, for unsized based numbers and integer constants.

Set this parameter to a rule name so that the **use_lrm_width** parameter is applicable to the specified rules only. For the other rules, which are not specified with this parameter, SpyGlass considers the natural width for integer value.

NOTE: *For new width related changes, refer to the New Width Flow Application Note.*

Used by	**Lint**: *W164a*, *W164a_a*, *W164a_b*, *W164b*, *W164c*, *W110*, *W116*, *W263*, *W362*, *W486*, *W416* **STARC**: STARC-2.1.3.1, STARC-2.8.1.6, STARC-2.10.3.1,STARC-2.10.3.2a,STARC-2.10.3.2b, STARC-2.10.3.2c, STARC-3.2.3.2 **STARC2002**: STARC02-2.1.3.1, STARC02-2.8.1.6, STARC02-2.10.3.1,STARC02-2.10.3.2a,STARC02-2.10.3.2b, STARC02-2.10.3.2c, STARC02-3.2.3.2 **STARC2005**: STARC05-2.1.3.1, STARC05-2.8.1.6, STARC05-2.10.3.1,STARC05-2.10.3.2a,STARC05-2.10.3.2b, STARC05-2.10.3.2c, STARC05-3.2.3.2
Options	yes, no, comma-separated rule list
Default value	no
Example	
Console \ Tcl-based example	`set_parameter use_lrm_width yes`
Usage in goal/ source files	`-use_lrm_width=yes`

use_natural_width

Specifies if the *W224* and *W484* rules should calculate the width using the natural width.

By default, the parameter is set to **no**.

Set the parameter to **yes** to calculate the width using the natural width.

In case of the *W484* rule, by default, the rule considers only the LRM width of the RHS in assignments. If this parameter set to **yes,** then rule considers both LRM and natural widths of the RHS in assignments.

Used by	*W224*, *W484*
Options	yes, no
Default value	no
Example	

Console \ Tcl-based example	`set_parameter use_natural_width yes`
Usage in goal/ source files	`-use_natural_width=yes`

use_new_flow

Enables the *LINT_abstract01* rule to use memory caching to improve runtime.

By default, the value of the parameter is set to no.

Set the value of the parameter to yes to enable the *LINT_abstract01* rule to use memory caching.

Used by	*LINT_abstract01*
Options	yes, no
Default value	no
Example	
Console \ Tcl-based example	`set_parameter use_new_flow yes`
Usage in goal/ source files	`-use_new_flow=yes`

verilint_compat

Specifies to run the rules that are normally not required as they check for syntax errors (that are already reported by SpyGlass) or are redundant and to enable processing of the *verilint Pragmas for SpyGlass lint Product*.

By default, the **`verilint_compat`** rule parameter is not set and the following rules are not run:

W159	*W162*	*W163*	*W259*	*W313*	*W316*
W326	*W328*	*W348*	*W464*	*W474*	*W475*
W476	*W477*	*W488*	*W493*	*W546*	

You can set the value of the **`verilint_compat`** rule parameter to **`yes`**

or **1** (to enable the mode) or to **no** or **0** (to disable the mode).

Used by	*W159*, *W162*, *W163*, *W259*, *W313*, *W316*, *W326*, *W328*, *W348*, *W464*, *W474*, *W475*, *W476*, *W477*, *W488*, *W493*, *W546*
Options	yes, no
Default value	no
Example	
Console \ Tcl-based example	`set_parameter verilint_compat yes`
Usage in goal/ source files	`-verilint_compat=yes`

waiver_compat

Enables the waivers in the *W121*, *W143*, *W398*, *W392*, *W402b*, *W415a*, *W546*, *LINT_MULTIASSIGN_BLOCKING_SIG* and *LINT_MULTIASSIGN_NONBLOCKING_SIG* rules to work correctly even if the line numbers of the RTL get changed.

By default, this parameter is set to **no** and the violations are not waived when:

- Rule message has line number string
- Waivers are applied based upon the message string (having line number), and during subsequent runs
- RTL has changed and hence the line number

If you set the value of this parameter to **yes** or **<rule-name>**, it ensures that the *specified* rules do not generate the line number information in the first run itself. Thus waivers work correctly even if the line numbers of the RTL gets changed in the subsequent runs.

Used by	*W121*, *W143*, *W398*, *W392*, *W402b*, *W415a*, *W546*, *LINT_MULTIASSIGN_BLOCKING_SIG*, *LINT_MULTIASSIGN_NONBLOCKING_SIG*
Options	no, yes, comma or space separated list of rules
Default value	no
Example	

Console/Tcl-based usage	`set_parameter waiver_compat yes`
Usage in goal/ source files	`-waiver_compat=yes`

W416_vhdl_only

Specifies whether the *W416* rule should use Verilog or VHDL version.

By default, the value of the parameter is set to no.

Set the value of the parameter to yes to enable the *W416* rule to use only the VHDL version and check VHDL parts of the design.

Used by	*W416*
Options	yes, no
Default value	no
Example	
Console \ Tcl-based example	`set_parameter W416_vhdl_only yes`
Usage in goal/ source files	`-W416_vhdl_only=yes`

SpyGlass lint Product Reports

The SpyGlass lint product generates the following reports in the **spyglass_reports/lint** directory:

Report Name	Description
SignalUsageReport	This report contains the details of the violating bits of multi-dimensional arrays, memory, and vector signals.
W415_Report	This report lists the drivers that are connected to multiple driven signals.
W448_Report	This report contains the details of the reset nets used synchronously and asynchronously.

You can also access the Reports from the *Reports* menu in the Atrenta Console.

SignalUsageReport

The SignalUsageReport is generated by *W86*, *W111*, *W120*, *W123*, *W240*, *W241*, *W497*, *W498*, *W499*, and *W528* rules. This report contains the details of the violating bits of multi-dimensional arrays, memory, and vector signals. You can view the report from the **spyglass_reports/lint** folder.

NOTE: *You can also access the SignalUsageReport report at the consolidated_reports directory.*

A sample format of SignalUsageReport is given below:

```
#########################################################

Module : test(Hierarchy :test)

W123 : Following Bits of signal ‘sig2’ (File Name: case1.v
```

```
Line No. :55) are read but not set

[0][1][1][4:1]  [0][1][2][4:1]  [0][1][3][4:1]
[0][2][3][4:1]  [1][2][3][4:1]

W123 : Following Bits of signal ‘sig1’ (File Name: case1.v
Line No. :25) are read but not set

[0][1][1][4:1]  [0][1][3][4:1]  [0][2][3][4:1]
[1][2][3][4:1]

Module : test(Hierarchy :test)

W528 : Following Bits of signal ‘check’ (File Name: case1.v
Line No. :55) are set but not read
[0][2][4:0]

Module: test(Hierarchy :test)

W111 : Following Bits of signal ‘mem’ (File Name: case1.v
Line No. : 7) are not read
[0][2][4:0]
```

The **SignalUsageReport** displays data for each rule module wise.

W415_Report

The W415_Report is generated by the *W415* rule. This report contains the details of drivers that are connected to multiple driven signals. You can view the report from the **spyglass_reports/lint** folder.

NOTE: *You can also access the W415_Report report at the consolidated_reports directory.*

A sample format of the W415_Report is given below:

```
#################################################################
####################
#
# This file has been generated by SpyGlass:
#     Report Name      : W415_Report
#     Report Created by:
#     Report Created on:
#     Working Directory:
#     SpyGlass Version :
#     Policy Name      : lint(SpyGlass_vM-2017.03)
#     Comment          : W415_Report Detail Message Report
#
#################################################################
####################
W415 Report : Multiple Drivers Detail Report



=========================W415 Multiple Drivers Detail
Report=====================================

SNO   FILE           LINE      Message
```

```
=============================================================
=====================================
2       top.v                       14              Signal
'top.Out1' has multiple simultaneous drivers
                                                    Drivers:
                                                 top.rtlc_I15.Z
(File: top.v Line: 8)
                                                 top.rtlc_I24.Z
(File: top.v Line: 10)
                                                 top.rtlc_I26.Z
(File: top.v Line: 12)
                                                 top.rtlc_I28.Z
(File: top.v Line: 14)


4       top.v                       18              Signal
'top.Out5[1]' has multiple simultaneous drivers
                                                    Drivers:
                                                 top.rtlc_I19.Z
(File: top.v Line: 17)
                                                 top.rtlc_I32.Z
(File: top.v Line: 18)


5       top.v                       16              Signal
'top.Out5[0]' has multiple simultaneous drivers
                                                    Drivers:
                                                 top.rtlc_I17.Z
(File: top.v Line: 15)
                                                 top.rtlc_I30.Z
(File: top.v Line: 16)
```

W448_Report

The W448_Report is generated by the *W448* rule. This report contains the details of the reset nets used synchronously and asynchronously. You can view the report from the **spyglass_reports/lint** folder.

NOTE: *You can also access the W448_Report report at the consolidated_reports directory.*

Use the following option to generate this report:

```
set_option report W448_Report
```

A sample format of the W448_Report is given below:

```
#########################################################
Module: test
*****************************************************
Net Name: rst
---------------------------
  Synchronous usage
---------------------------
out2
---------------------------
  Asynchronous usage
---------------------------
out1
*****************************************************
```

verilint Pragmas for SpyGlass lint Product

You can use the **verilint** pragma to disable checking of a specified part of the RTL description for the specified rule(s). This feature is enabled by the *verilint_compat* rule parameter.

To disable the checking of a code block, specify the following **verilint** pragma at the start of the code block:

```
// verilint <rule-name-list> off
```

Then, at the end of the code block, specify the following **verilint** pragma:

```
// verilint <rule-name-list> on
```

The *<rule-name-list>* is a comma-separated or space-separated list of rules names. These rule names can be the exact rule name (like **W122** or **bothedges**) or can be the numeric portion of the rule name (like **122** for the W122 rule).

For example, you may want to disable rule-checking in the following portions of the design:

```
W110 : from line no 12 to line no 15
W120, W130 : from line no 12 to line no 20
W140 : from line no 12 to line no 25
```

Specify the **verilint** pragmas as follows:

```
line 12: // verilint 110,120,130,140 off
line 15: // verilint W110 on
line 20: // verilint 120 W130 on
line 25: // verilint W140 on
```

You can also add comments with the **verilint** pragma. All text after the **on** or **off** keywords is assumed to be a comment.

Reporting Hierarchical Paths

Some of the rule messages also report the hierarchical path of the scope containing the rule-violating objects.

For Verilog designs, the hierarchical path is the complete hierarchical name of the containing scope.

For VHDL designs, the hierarchical path can be `<entity-name>(<arch-name>)` for rule-violating objects in architectures, `<entity-name>(<arch-name>):<block-name>` for rule-violating objects in blocks in architectures, `<entity-name>(<arch-name>):<component-instance-name>@<entity-name>(<arch-name>)` for rule-violating objects in component instances.

Determining Signals Required in the Sensitivity List

SpyGlass checks all potential cases where a signal is read inside an **always** construct but not present in the sensitivity list.

Signals in Assignment Statements

If a signal is assigned some value through a blocking assignment and is later read in same always construct, that signal is ignored provided its bit index is not a variable.

Signals in for Constructs

SpyGlass ignores the **for** construct loop index variables if they are used afterwards in the construct. Also, a signal whose bit-select is used as the for construct loop index variable, is assumed to be read and the signals used in the corresponding step expressions are ignored.

SpyGlass does not expand nested **for** constructs. Only the last **for** construct is processed.

Only simple **for** constructs as in the following example are expanded:

```
for (i = 0; i < n2; i = i + 1)
  out[i] = in[i];
```

In this example, the value of **n2** should be a constant value.

In case of **for** construct loop variables, only arithmetic expressions with some constant value are computed. All other cases are treated as variable bit indexes and the whole bus is required to be present in the sensitivity list. Consider the following example:

```
for (i = 0; i < n2; i = i + 1)
  out[i] = in[i + n3];
```

The value of **n3** should be some constant value. If **n3** is not a constant, then signal **in** as a whole should be in the sensitivity list. Expressions like **n3 - i** are not be expanded.

Bit-selects of Variables

If any signal's bit-select is a variable, then that signal either as whole should be present in the sensitivity list or it should be with same index

variable expression present in the sensitivity list. Consider the following examples:

```
always @(in or i)
  out = in[i];
```

or

```
always @(in[i] or i)
  out = in[i];
```

Since the bit index is a variable in both examples, the signal **in** as a whole should be present in the sensitivity list or the signal **in** should be present in the sensitivity list with same index variables expression.

Memories

In case of memories, if the word-select is a variable, then the whole bus should be in the sensitivity list or the bus should be present in the sensitivity list with same index variables expression.

Rule Severity Classes

The SpyGlass lint product rule severity labels have been classified under the SpyGlass pre-defined rule severity classes as follows:

Rule Severity Class	Contains the Rule Severity Labels...
WARNING	Warning, Guideline
INFO	Info
DATA	Data

See the *Atrenta Console Reference Guide* for more information about SpyGlass predefined rule severity classes.

Same or Similar Rules in Other SpyGlass Products

The following table lists same rules in the Lint, OpenMORE, and STARC products:

TABLE 1 Same Rules in Standard Products

Rule	Lint	OpenMORE	STARC
VHDL	BothPhase	--	STARC-2.3.3.2
	W526	CaseOverIf	--
	--	PortOrder_C	STARC-3.1.3.2c
	W259	--	STARC-2.5.1.5a
	W415	--	STARC-2.5.1.5b
	--	ExprParen	STARC-2.10.1.2
	W122	NotInSens	STARC-2.2.2.1
	W456	--	STARC-2.2.2.2
	W71	--	STARC-2.8.1.4
	W422	--	STARC-2.3.3.1
	ClockStyle	--	STARC-2.3.1.2c
	W489	--	STARC-2.1.8.9
	W146	NamedAssoc	STARC-3.2.3.1
	--	NoTab	STARC-3.1.4.3
	--	ConsCase	STARC-1.1.1.5

TABLE 1 Same Rules in Standard Products (Continued)

Rule	Lint	OpenMORE	STARC
Verilog+VHDL	--	ClockPhase	STARC-1.2.1.1a
	--	SepClock	STARC-1.4.3.1b
	--	IntClock	STARC-1.2.1.2
	--	GateClockAtTop	STARC-1.4.1.1
	--	GateResetAtTop	STARC-1.3.3.4
	--	PortOrder_A	STARC-3.1.3.2a
	W448	--	STARC-1.3.1.6
	--	PortOrder_B	STARC-3.1.3.2b
	W18	InferLatch	STARC-2.2.1.3
Verilog	W526	CaseOverIf	--
	W415	--	STARC-2.5.1.5a
	--	ExprParen	STARC-2.1.4.1
	W146	NamedAssoc	STARC-3.2.3.1
	W224		STARC-2.1.5.3
	W484	--	STARC-2.10.6.1
	W352	--	STARC-2.9.1.2d
	W468	--	STARC-2.1.6.4
	W456	NotReqSens	--
	W575		STARC-2.10.2.3
	--	NoTab	STARC-3.1.4.3
	W336	NonBlockAssign	STARC-2.3.1.1

The following table lists similar rules in the Lint, OpenMORE, and STARC products:

TABLE 2 Similar Rules in Standard Products

Rule	Lint	OpenMORE	STARC
VHDL	W456a	NotReqSens	--
	--	ReserveName	STARC-1.1.1.3a
	--	Indent	STARC-2.7.3.5
	--	LineLength	STARC-3.1.4.5
Verilog+VHDL	--	IntReset	STARC-1.3.2.2
	--	CombLoop	STARC-1.2.1.3
	--	RegOutputs	STARC-1.6.2.1
Verilog	W238	--	STARC-2.3.2.2
	W496b	--	STARC-2.10.1.5b
	W362	--	STARC-2.10.3.1
	W110	--	STARC-3.2.3.2
	W122	NotInSens	STARC-2.2.2.1
	W164		STARC-2.10.3.2b
	W263		STARC-2.8.1.6
	--	Indent	STARC-2.7.3.5
	--	LineLength	STARC-3.1.4.5
	W336	NonBlockAssign	STARC-2.3.2.1

When these products are run together, you can ignore one of these duplicate rule pairs for rule-checking by using **`set_option ignorerules {rule_names}`** command in the batch mode.

Rules in SpyGlass lint

The SpyGlass lint product provides the following types of rules:

- *Array Rules*
- *Case Rules*
- *Lint_Reset Rules*
- *Lint_Clock Rules*
- *Usage Rules*
- *Lint_Tristate Rules*
- *Assign Rules*
- *Function-Task Rules*
- *Function-Subprogram Rules*
- *Delay Rules*
- *Lint_Latch Rules*
- *Instance Rules*
- *Synthesis Rules*
- *Expression Rules*
- *MultipleDriver Rules*

- *Simulation Rules*
- *Event Rules*
- *Loop Rules*
- *Lint_Elab_Rules*
- *Verilint_Compat Rules*
- *Miscellaneous Rules*

Array Rules

The SpyGlass lint product provides the following array related rules:

Rule	Flags...
W17	Arrays in sensitivity lists that are not completely specified
W86	Arrays where all elements are not set
W111	Arrays where all elements are not read in the process
W488	Arrays that appear in the sensitivity list but all elements of the arrays are not read in the process

W17

Prefer full range of a bus/array in sensitivity list. Avoid bits or slices

Language

Verilog, VHDL

Rule Description

The W17 rule flags those arrays in sensitivity lists that are not completely specified.

While the logic inside a combinational construct (**always** construct or **process** construct) may be sensitive only to certain index or bit-slices of an array, using only index or bit-slices in the construct's sensitivity list can lead to subtle errors if the construct design is subsequently changed.

NOTE: *The W17 rule is also grouped under the Simulation Rules group.*

Message Details

Verilog

The following message appears at the location of an **always** construct's sensitivity list in hierarchy *<hier-path>* that has a bus *<bus-name>* that is not completely specified:

```
Bus '<bus-name>' specified in sensitivity list with incomplete range[Hierarchy: '<hier-path>']
```

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding functions/tasks.

VHDL

The following message appears at the location of a **process** construct's sensitivity list in hierarchy *<hier-path>* that has an array *<array-name>* of type *<sig-type>* but the array is not completely specified:

```
Complete vector <sig-type> '<array-name>' is not used in the sensitivity list. [Hierarchy: '<hier-path>']
```

Where, *<sig-type>* can be **signal**, **port**, or **variable** and *<hier-*

path> is the complete hierarchical name of the containing scope excluding subprograms.

Severity

Warning

Suggested Fix

This is partly a question of taste. If you specify the whole bus and some bits are not read in the block, you may get a warning that the sensitivity list is over-specified. In such cases, it is not possible to satisfy both requirements simultaneously, so you must decide which requirement you want to ignore. From an implementation point of view, it is safest to ignore this rule.

W86

Not all elements of an array are set

Language

VHDL

Rule Description

The W86 rule flags arrays where all elements are not set.

Designs where all elements of arrays are not set, may not fully exercise all paths in to the array and thus may mask possible errors in index or bit-slice selection.

The W86 rule will not check for memory elements having non-static index.

To see the details of the bits that are not set, refer to the *SignalUsageReport* report in the *spyglass_reports/lint* directory.

Message Details

Arrays used in Process Sensitivity Lists

The following message appears at the location of a process' sensitivity list that has an array *<array-name>* of type *<array-type>* and all array elements are not set in the process scope (hierarchy *<hier-path>*):

`Not all the elements of <array-type> '<array-name>' (Bits: <offending-bits>) are set. [Hierarchy: '<hier-path>']`

Where, the *<array-type>* can be **signal**, **port**, or **variable** and *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Arrays not used in Process Sensitivity Lists

The following message appears at the location of an array declaration *<array-name>* of type *<array-type>* that is not used in any process sensitivity list and all array elements are not set in the scope *<hier-path>*:

`Not all the elements of <array-type> '<array-name>' (Bits: <offending-bits>) are set [Hierarchy: '<hier-path>']`

Where, the *<array-type>* can be **signal**, **port**, or **variable** and

<hier-path> is the complete hierarchical name of the containing scope excluding subprograms.

Severity

Warning

Suggested Fix

This is an informational rule. No action may be required if all locations (required) can actually be set.

W111

Not all elements of an array are read

Language

Verilog, VHDL

Rule Description

The W111 rule reports a violation for arrays where all elements of the array are not read.

The W111 rule checks for input/inout ports, internal signals, and variables.

An array with unused elements indicates that the model does not fully exercise all paths out of the array and thus may mask possible errors in index or bit-slice selection.

NOTE: *For Verilog, inside the nested* **`for`** *loops, signals of user-defined type like structures, interfaces, etc. are considered fully* **`set`** *if used on the left-hand side of an expression and fully* **`read`** *if used on the right-hand side of an expression.*

By default, the W111 rule reports violation for array declared as input/inout port, internal signal or variable but not read. Set the value of the *ignore_in_ports* parameter to **`yes`** to ignore input and inout ports for rule checking.

By default, this rule does not process large arrays. Set the *handle_large_bus* parameter to **`yes`** to process large arrays (greater than 50,000) and report violation if not used correctly.

To see the details of the bits that are not read, refer to the *SignalUsageReport* report in the *spyglass_reports/lint* directory.

NOTE: *The W111 rule only checks static indexes.*

The W111 rule is also grouped under the *Usage Rules* group.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

Verilog

Message 1

The following message appears at the location of an array *<array-name>* in hierarchy *<hier-path>* that is not read completely:

```
Not all elements of array '<array-name>' are 'read' [Hierarchy: '<hier-path>']
```

Where, *<array-type>* can be **signal**, **port**, or **variable**. The *<hier-path>* is the complete hierarchical name of the containing scope excluding functions/tasks.

Message 2

The following message is displayed when the signal size is greater than 50,000 and the *handle_large_bus* parameter is disabled:

```
Signal '<signal-name>' size too big thus not processed, use 'set_parameter handle_large_bus yes' for enabling handling of these signals
```

If the **handle_large_bus** parameter is not enabled, the violation for signals of size greater than 50,000 is missed.

VHDL

The following message appears at the location of an array *<array-name>* of type *<array-type>* in hierarchy *<hier-path>* that is not read completely. The message also displays the bits which are not read in the array:

```
Not all the elements of <array-type> '<array-name>' '(<offending-bits>)' are read. [Hierarchy: '<hier-path>']
```

Where, *<array-type>* can be **signal**, **port**, or **variable**. The *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Severity

Warning

Suggested Fix

Consider this rule primarily informational.

Verilog Examples

Consider the following example:

```
module top (out,in);
    output [1:0] out;
    input [2:0] in;

  assign out=in[1:0];

endmodule
```

In the above example, the *W111* rule reports a violation because all bits of the array **in** are not read. The following message is reported by this example:

```
Not all elements of array 'in[2]' are 'read' [Hierarchy:
':top']
```

VHDL Examples

Consider the following example where all bits of array **a** are not read in architecture **arc_e_l** of entity **e_l**:

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;

entity e_l is
  port(a: in bit_vector(4 downto 0);
  b: out bit_vector(3 downto 0));
end e_l;

architecture arc_e_l of e_l is
  begin
    b <= a(3 downto 0);
  end arc_e_l;
```

In the above example, the *W111* rule reports a violation as all bits of array a are not read in the architecture **arc_e_l** of entity **e_l**. The following message is reported by this example:

```
Not all the elements of in port 'a' (Bits: 4) are read.
[Hierarchy: ':e_l(arc_e_l):']
```

W488

A bus variable appears in the sensitivity list but not all bits of the bus are read in the contained block (Verilog)
An array signal appears in the sensitivity list but not all bits of the array are read in the process (VHDL)

Language

Verilog, VHDL

Rule Description

Verilog

The W488 rule flags the bus variables that appear in the sensitivity list but not all bits of that bus are read in the contained block.

While such designs are allowed, they may impact simulation performance. Many simulators re-evaluate the block on each change in each bit, even though the evaluation is redundant.

NOTE: *The W488 rule supports generate-if and generate-for block.*

NOTE: *The W488 rule is switched off by default. You can enable this rule by either specifying the* `set_goal_option addrules W488` *command or by setting the verilint_compat rule parameter to* `yes`*.*

VHDL

The W488 rule flags those arrays that appear in the sensitivity list but all elements of the arrays are not read in the process.

While such designs are allowed, they may impact simulation performance. Many simulators re-evaluate the process on each change in each element of an array in the sensitivity list, even though the evaluation is redundant.

NOTE: *The W488 rule also grouped under the Simulation Rules group.*

Message Details

Verilog

The following message appears at the location of the contained block's sensitivity list that contains a bus/memory variable, *<var-name>*, and all the bits/words of that variable are not read in the contained block:

`Bus/Memory '<var-name>' (or some of its bits/words) which is`

not read in always block is not required in the sensitivity list[Hierarchy: '<hier-path>']

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

VHDL

The following message appears at the location of a process' sensitivity list that has an array *<array-name>* of type *<array-type>* and all array elements are not read in the process scope (hierarchy *<hier-path>*):

Array <array-type> '<array-name>' is in the sensitivity list but not all its bits are read inside the process [Hierarchy: '<hier-path>']

The *<array-type>* can be **signal**, **port**, or **variable** and *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Severity

Warning

Suggested Fix

Verilog

There is a trade-off in fixing this rule versus *W122*, *W456* and *W456a* rules. It is recommended to clear W122, W456 and W456a rules and waive or ignore the W488 rule.

VHDL

Remove variables from the sensitivity list if they are not required. This will help improve simulation performance without effecting synthesis behavior.

Case Rules

The SpyGlass lint product provides the following case construct related rules:

Rule	Flags...
W69	case constructs that do not have all possible clauses described and also do not have the default clause
W71	case constructs that do not contain a default clause
W171	Expressions used as case clause labels
W187	case constructs where the default clause is not the last clause
W226	case constructs where the selector is a constant or a static expression
W263	case clause labels whose widths do not match the width of the corresponding case construct selector
W332	case constructs that do not have all possible clauses described and have a default clause
W337	Illegal case construct labels
W398	Duplicate choices in CASE construct
W453	case constructs with large selector bit-width and more number of case clauses
W551	case constructs with the default clause and full_case pragma applied or priority/unique case constructs with default clause

W69

Ensure that a case statement specifies all possible cases and has a default clause

When to Use

Use this rule to identify case constructs that does not describes all possible cases and does not have a default clause.

Rule Description

The W69 rule reports violation for case constructs that do not have all possible clauses described and also do not have the default clause.

NOTE: *The W69 rule supports generate-if and generate-for block.*

NOTE: *By default, the W69 rule is switched off, and will be deprecated in a future SpyGlass release. You can use the W71 rule as a replacement of the W69 rule.*

Rule Exceptions

The W69 rule ignores case constructs with **`full_case`** pragma specified.

Language

Verilog

Default Weight

5

Parameter(s)

- *handle_case_select*: Use this parameter to specify the maximum width of case selector that the supported rule should process.
- *strict*: The default value is no. Set the value of the parameter to yes to check the case constructs in sequential always constructs. You can also specify a comma-separated list of rule as an input to the parameter.

Constraint(s)

None

Messages and Suggested Fix

Message 1

The following message appears at the first line of a case construct in a combinational always construct that does not have all possible clauses described and also does not have a default clause:

```
[WARNING] Case statement is missing cases and has no default in
Combinational block [Hierarchy: '<hier-path>']
```

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Potential Issues

A violation is reported when a **CASE** construct in a combinational always construct does not describes all possible clauses, including the **default** clause.

Consequences of Not Fixing

A latch is inferred, if the case construct does not describe all possible states and also does not have a default clause. While you may have intended to infer a latch, it is advisable to examine such cases to avoid creation of unexpected logic.

How to Debug and Fix

For more information on debugging and fixing the violation, click *How to Debug and Fix*.

Message 2

The following message appears at the first line of a case construct in a sequential always construct that does not have all possible clauses described and also does not have a default clause when the **strict** rule parameter has been set:

```
[WARNING] Case statement is missing cases and has no default in
Sequential block [Hierarchy: '<hier-path>']
```

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Potential Issues

A violation is reported when a **CASE** construct in a combinational always construct does not describes all possible clauses, including the **default**

clause when the strict parameter is set.

Consequences of Not Fixing

A latch may be inferred, if the case construct does not describe all possible states and also does not have a default clause. While you may have intended to infer a latch, it is advisable to examine such cases to avoid creation of unexpected logic.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the case statement with missing default is declared.

To fix the violation, you can perform any of the following tasks:

- add case clauses (or a default clause) for those cases that are not covered and specify the appropriate behavior in that default clause.
- add pragma **`full_case`** to the case statement, implying you know that the remaining cases can never happen.

Example Code and/or Schematic

Example 1

Consider the following example where the case construct does not have all possible cases described (case **`in3=2'b10`** is not described) and also does not have a default clause:

```
...
always@(in1 or in2 or in3 or mem)
begin
  case(in3)
    2'b00 : out[0] = in1 || mem[0];
    2'b01 : out[1] = in1 && mem[1];
    2'b11 : out[2] = in2 && mem[3];
  endcase
end
...
```

For this example, SpyGlass generates the following message:

```
Case statement is missing cases and has no default [Hierarchy:
‘<hier-path>’]
```

Example 2

Consider the following example:

```
module mod(sel,o1,o2,o4);

input [1:0] sel;
output [3:0] o1, o2,o4;
reg [3:0] o1, o2,o4
reg [3:0] mem[0:3];

always@(sel)
  begin
    case(sel)  //violation(Case statement is missing cases
and has no default in combinational block)
      2'b00 : o1 = mem[0];
      2'b01 : o2 = mem[1];
      2'b11 : o4 = mem[3];
    endcase
  end

endmodule
```

In the above example, the W69 rule reports a violation as the **`case`** statement has missing cases and a missing **`default`** clause.

Default Severity Label

Warning

Rule Group

Case, Lint_Elab_Rules

Reports and Related Files

No related reports or files.

W71

Ensure that a case statement or a selected signal assignment has a default or OTHERS clause

When to Use

Use this rule to identify case constructs without the default or **OTHERS** clause.

Description

The W71 rule reports violation for case constructs without the default clause (Verilog designs) and case constructs and selected signal statements without the OTHERS clause.

A default (or OTHERS) clause should always be specified in a case construct to handle unexpected situations even if the construct covers all potential situations.

NOTE: *The W71 rule supports generate-if and generate-for block.*

Rule Exceptions

For **Verilog** designs, the W71 rule does not report for missing default clause in the following cases:

- If the target signals in the case construct are assigned using a blocking or nonblocking assignment statement before the case statement
- Case constructs that are inside always construct that infer a flip-flop (Set check_sequential rule parameter to report such cases).
- Case constructs with associated full_case pragma and unique / priority cases.
- Fully-specified case constructs.
- Case statements with static case select (including generate case).

For **VHDL** designs, the W71 rule has the following exceptions:

- The syntax is reported if all case constructs are not covered.
- When selector is a port and all the case constructs are covered, then the others clause is not required. Hence, no violation is reported in such a case.

- When case selector is a variable then the others clause must be used.

Language

Verilog, VHDL

Default Weight

5

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameter(s)

- *handle_case_select*: Use this parameter to specify the maximum width of case selector that the supported rule should process.
- *strict*: The default value is no. Set the value of the parameter to yes to report fully-specified case constructs without the default clause.
- *check_sequential*: The default value is no. Set the value of the parameter to yes to enable rule checking in sequential block.
- *do_not_run_W71*: The default value is no. Set the value of the parameter to yes to block the W71 rule. This parameter is provided to avoid duplicate violation in case the *W69* and *W71* rules are run in mixed mode.
- *ignore_unique_and_priority*: Default value is no. Set the value of the parameter to yes to ignore violations for the unique, unique0, and priority cases.

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message appears at the location where a case construct is defined without a default clause:

```
[WARNING] Case statement does not have a default clause and is
not preceded by assignment of target signal <block-type>
```

`[Hierarchy: '<hier-path>']`

Where, *<block-type>* can be either a sequential or a combinational block and the *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Potential Issues

A violation is reported when a **case** construct is defined without a **default** clause.

Consequences of Not Fixing

If all possible cases of the case construct selector are covered, this is not directly an error. However, if the case construct selector has an undefined value (**X** or **Z**) and there is no default clause, then the design simulation may produce unexpected results.

If the width of the case construct selector changes as the design evolves, then what had once been a fully covered case construct may become only partially covered and can lead to inferred latches. Hence, it is recommended to always describe a default clause even if all possible cases of the case construct selector are described.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window is displayed. The HDL Viewer window highlights the line where the case statement with missing default is declared.

To fix the violation, add a default clause to specify default behavior. If you are specifying simulation X behavior, bracket this behavior in **translate_off** and **translate_on** pragmas.

VHDL

The following message appears at the location where a construct of *<type>* is defined without an OTHERS clause:

`[WARNING] Others clause not found in '<type>' statement`

Where, *<type>* can be a **CASE** (for case constructs) or **selected signal assignment** (for Selected Signal Assignments).

Potential Issues

A violation is reported when a **CASE** construct or a selected signal assignment is defined without an **OTHERS** clause.

Consequences of Not Fixing

If all possible cases of the case construct selector are covered, this is not directly an error. However, if the case construct selector has an undefined value (**X** or **Z**) and there is no default clause, then the design simulation may produce unexpected results.

If the width of the case construct selector changes as the design evolves, then what had once been a fully covered case construct may become only partially covered and can lead to inferred latches. Hence, it is recommended to always describe a OTHERS clause even if all possible cases of the case construct selector are described.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window is displayed. The HDL Viewer window highlights the line where the case statement with missing **OTHERS** clause is declared

To fix the violation, add an OTHERS clause to specify default behavior. If you are specifying simulation X behavior, bracket this behavior in **translate_off** and **translate_on** pragmas.

Example Code and/or Schematic

Verilog

Consider the following example where the case construct has all possible clauses described but does not have the default clause:

```
...
always@(in1 or in2 or mem)
  begin
    case(in2)
      2'b00 : out1 = in1 || mem[0];
      2'b01 : out1 = in1 && mem[1];
      2'b10 : out1 = in1 && mem[2];
      2'b11 : out1 = in1 && mem[3];
    endcase
  end
...
```

For this example, SpyGlass generates the W71 rule message, if you analyze the example using the **strict** rule parameter.

VHDL

Example of CASE Construct

Consider the following example where the CASE construct does not have an OTHERS clause defined:

```
entity ent is
  port(a: in bit_vector(1 downto 0);
       b: out bit_vector(1 downto 0));
end ent;

architecture arc of ent is
  begin
    process(a)
      begin
        case a is
          when "00" | "11" => b <= a;
          when "01" | "10" => b <= not a;
        end case;
     end process;
end arc;
```

For this example, SpyGlass generates the following message:

```
Others clause not found in 'CASE' statement
```

Example of Selected Signal Assignment Statement

Consider the following example where the select assignment statement does not have an OTHERS clause:

```
entity ent is
  port(a: in bit_vector(1 downto 0);
       b: out bit_vector(1 downto 0));
end ent;

architecture arc of ent is
  begin
    with a select
      b <= a when "00" | "11", not a when "01" | "10";
end arc;
```

For this example, SpyGlass generates the following message:

```
Others clause not found in 'selected signal assignment'
statement
```

Default Severity Label

Warning

Rule Group

Case, Lint_Elab_Rules

Reports and Related Files

No related reports or files.

W171

Case label is non-constant.

Language

Verilog

Rule Description

The W171 rule flags expressions used as case clause labels.

Typical state machine descriptions require constant case clause labels as in the following example:

```
case (x)
  1: ...
  2: ...
 25: ...
...
endcase
```

Some designers may represent complex **if** conditions as a case construct as in the following example:

```
case (sel)
  expr1: ...
  expr2: ...
  ...
endcase
```

The W171 rule flags such case constructs.

The W171 rule flags non-static case clause labels provided they are not assigned a constant value elsewhere in the design.

By default, the *W171* rule reports violation for priority cases. Set the value of the *ignore_priority_case* parameter to yes to ignore violations for the priority cases.

By default, the *check_const_selector* parameter is set to **no** and the rule gives violation when case label is non-constant. If you set the **check_const_selector** parameter to **yes**, this rule checks weather case selector is constant or not. The rule does not report any violation if case selector is a constant and case label belongs to one-hot operation,

although case label is a variable.

By default, the value of the *ignore_const_selector* parameter is no and the W171 rule considers case statements where the case selector is a static expression. Set the value of the parameter to yes to ignore such case statements.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location where a non-constant case clause label is encountered:

`Case-label is not constant`

Rule Severity

Warning

Suggested Fix

Generally nothing to fix, unless you find a case which does not match expectations.

Examples

Consider the following example where one clause of the case construct has a non-constant label (**`sel`**):

```
module test;
  reg [1:0] addr_int, data_out, data_in;
  reg clk, sel;

  always @(posedge clk)
    case (1)
      0       : data_out <= data_in;
   (sel)      : data_out <= data_in;
      2       : if (~sel) data_out <= data_in;
      3       : data_out <= 2'bz;
      default : data_out <= 2'bx;
```

```
    endcase
endmodule
```

For this example, SpyGlass generates the W171 rule message.

W187

The 'default' or 'others' clause should be the last clause in a case statement

Language

Verilog, VHDL

Rule Description

The W187 rule flags those case constructs where the default clause (Verilog) or others clause (VHDL) is not the last clause in the construct.

While the default clause (or others clause) can be placed anywhere in the case construct, it is recommended to place it as the last clause for better understanding.

By default, the *W187* rule reports violation for priority cases. Set the value of the *ignore_priority_case* parameter to yes to ignore violations for the priority cases.

Message Details

Verilog

The following message appears at the first line of a case construct where the default clause is not the last clause in the construct:

```
Default clause should be last clause in a case stmt
```

VHDL

The following message appears at the first line of a case construct where the others clause is not the last clause in the construct:

```
OTHERS clause should be last clause in a case stmt
```

Rule Severity

Warning

Suggested Fix

Move the default clause (Verilog) or others clause (VHDL) to the end of the case statement.

Verilog Examples

Consider the following example where the default clause of the case construct is not the last clause:

```
module test;
  reg [1:0] addr_int, data_out, data_in;
  reg clk, clk1;
  reg [1:0]sel;
  reg [3:0] a_1, b_1;

  always @(posedge clk or posedge clk1)
    case (sel)
      default : data_out <= 2'bx;
      0       : data_out <= data_in;
      1       : data_out <= data_in;
      2       : if (a_1 == b_1) data_out <= data_in;
      3       : if (a_1 != b_1) data_out <= 2'b11;
    endcase
endmodule
```

For this example, SpyGlass generates the W187 rule message.

W226

Case select expression is constant

Language

Verilog, VHDL

Rule Description

Verilog

The W226 rule flags case constructs where the case construct selector is a constant or a static expression.

Typical state machine descriptions require non-constant case construct selector as in the following example:

```
case (x)
  1: ...
  2: ...
 25: ...
...
endcase
```

Some designers may represent complex **if** conditions as a case construct as in the following example:

```
case (1)
  expr1: ...
  expr2: ...
  ...
endcase
```

The above case construct uses a constant case construct selector (**1**) and is equivalent to the following **if** construct:

```
if (expr1)
  else if (expr2)
    else ...
```

The W226 rule flags such case constructs.

By default, the *check_const_selector* parameter is set to **no**. When you set the parameter to **yes**, the *W226* rule checks weather all case labels are

one-hot. If so, the rule dose not report a violation even if the case selector is a constant.

By default, the *W226* rule reports violation for priority cases. Set the value of the *ignore_priority_case* parameter to yes to ignore violations for the priority cases.

VHDL

The W226 rule flags case constructs or selected signal assignments where the selector is a constant or a static expression.

A constant control expression or sub-expression evaluates to a TRUE/FALSE situation and thus, the construct is always executed or never executed.

By default, the W226 rule does not check inside function bodies unless the *strict* rule parameter is set.

By default, the value of the *ignore_generatefor_index* parameter is set to **no**. Set the value of this parameter to **yes** to ignore violations for **case** statements inside **generte-for** blocks, if the case selector is a loop variable of **generate-for** loop.

Message Details

Verilog

The following message appears at the start of a case construct where the selector is a constant or static expression <*expr*>:

```
Case select expression '<expr>' is constant
```

VHDL

The following message appears at the location of a statement <*stmt*> where the selector is a constant or static expression:

```
The <stmt> expression is constant
```

Where <*stmt*> can be CASE construct or selected signal assignment statement.

Rule Severity

Warning

Suggested Fix

Generally nothing to fix, unless you find a case which does not match expectations.

Examples

Verilog

Consider the following case construct that has a constant select expression:

```
case (1)
  0       : cc_data_out <=cc_data_in;
  (sel)   : cc_data_out <=cc_data_in;
  2       : if (~sel) cc_data_out <= cc_data_in;
  3       : cc_data_out <= 2'bz;
  default : cc_data_out <= 2'bx;
endcase
```

For this example, SpyGlass generates the W226 rule message.

VHDL

Consider the following example:

```
case bit_vector'("11") is
     when "00" | "01" => q <= d;
     when others => q <= not d;
end case;
```

In the above example, the *W226* rule reports a violation because the case construct selector is a constant **`bit_vector'("11")`**.The following message is reported by this example:

```
The case expression is constant
```

W263

Reports a case expression width that does not match case select expression width

When to Use

A violation is reported when a case expression width does not match the select expression width.

Description

The *W263* rule reports the case clause labels whose widths do not match the width of the corresponding case construct selector.

This rule calculates the width of case expressions on the basis of the following conditions:

- If the *nocheckoverflow* rule parameter is set to **yes** or **W263**, width is calculated as per LRM. However, for constants, natural width is considered.
- If the *nocheckoverflow* rule parameter is set to **no**, width is calculated according to the following methods:
  - For plus/minus operator, value-based width is considered.
  - For multiplication operator, width is sum of operand widths.
  - For division operator, LHS width is considered.
  - For concat operator, width is calculated as sum of all the operands.

Rule Exceptions

The *W263* rule does not report a violation in the following scenarios:

- When case select is a static expression
- When case expression is a constant (integer/macro) or unsized based number, and its width is smaller than the case select width

Language

Verilog

Default Weight

5

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameter(s)

- *ignore_integer_constant_labels*: Default value is no. Set the value of the parameter to yes to ignore violations for constant case labels of integer type.
- *nocheckoverflow*: Default value is **no**. This indicates the *W263* rule does not check the bit-width as per LRM. Set this parameter to **yes** or rule name to check the bit-width as per LRM.
- *strict*: The *W263* rule is switched off by default. You can enable this rule by specifying the **set_goal_option addrules W263** command.
- *use_lrm_width*: Default value is no and the *W263* rule considers the natural width of integer constants and unsized based numbers. Set this parameter to **yes** to consider LRM width of integer constants and unsized based numbers, which is 32.
- *handle_static_caselabels*: Default value is **no**. Set this parameter to **yes** to ignore violations for static case labels, which are of less width than the width of case selector.

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location of a case clause label whose width (*<case-label-width>*) is not same as the case construct selector width (*<selector-width>*).

```
[WARNING] Case label (<case-label-name>) width (<case-label-
width>) does not match selector (<selector-name>) width
(<selector-width>) [Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Potential Issues

Violation may arise when the case label width does not match the selector width.

Consequences of Not Fixing

When the case clause labels are of width smaller than the width of the case construct selector, it might lead to unwanted target in case expressions.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line of case label. Scroll up the HDL Viewer window to view the case selector and the width mismatch. Alternatively, open the HDL in editor and search for case selector.

Modify the case expressions to explicitly match the case selector in all bits. This makes the behavior of the RTL code easier to understand and does not rely on undefined behavior in handling unmatched bits.

Example Code and/or Schematic

Example 1

Consider the following example where the **sel** clause of the case construct is of width 4 bits and the case clause labels have the same or different bit-widths.

```
module test(in, out, sel, data);
  input [3:0] in, data, sel;
  output [3:0]out;
  reg [3:0]out;

  parameter [3:0]S = 23;
  parameter [2:0]P = 10;
  parameter [15:0]Q = 16'h1101;
  parameter R = 2;

  always @(in or sel)
    begin
      case (sel)
        S : out = 4'b0000;
        P : out = 4'b0000;
        Q : out = 4'b0001;
        R : out = 4'b0011;
```

```
        in: out = 4'b0011;
        4'b1010 : out = 4'b0011;
        3'b110 : out = 4'b0011;
        default out = 0;
      endcase
    end
endmodule
```

The case clause labels (**S**, **in**, and **4'b1010**) have the same width (4 bits) as the case construct selector **sel**.

The case clause labels **P** (3 bits), **Q** (16 bits), **R** (32 bits being an integer), and **3'b110** (3 bits) do not have the same bit-width as the case construct selector **sel**.

Example 2

Consider the following example:

```
module mod(sel,o1,o2,o3,o4,clk);
input clk;
input [1:0] sel;
output [3:0] o1, o2, o3, o4;
reg [3:0] o1, o2, o3, o4;
reg [3:0] mem[0:3];

always@(sel)
  begin
    case(sel)
      2'b00 : o1 = mem[0];
      2'b01 : o2 = mem[1] ;
      1'b1 : o4 = mem[3];
      default : o4 = mem[2];
    endcase
  end

endmodule
```

In the above example, the *W263* rule reports a violation as the case label width does not match the selector width.

Example 3

Consider the following example:

```
case (sel)
(a[2:0]+b[2:0]): out=1'b1;
```

In the above example, width of **a[2:0]+b[2:0]** is 4.

Example 4

Consider the following example:

```
case (sel)
(a[2:0]*b[2:0]): out=1'b1;
(a1[2:0]*b1[2]): out=1'b0;
```

In the above example, the **a[2:0]*b[2:0]** is 6. Also, the width of **a1[2:0]*b1[2]** is 3.

Example 5

Consider the following example:

```
case (sel)
(a[2:0]/b[2]): out=1'b0;
```

In the above example, width of **a[2:0]/b[2]** is 3.

Example 6

Consider the following example:

```
case(sel)
{1'b0,a[2:0]}: out=1'b1;
{1'b1,a[2:0]}: out=1'b0;
```

In the above example, width is 3 for the first expression, ignoring all the leading 0s. The width is 4 bits for the second expression.

Example 7

Consider the following example:

```
module top(input clk);
wire     [7:0] pState;
wire     in1,in2;
reg      out;
```

```
parameter   ASTATE = 4;
parameter   BSTATE = 512;

    always @ (posedge clk) begin
        case (pState)
        ASTATE : out = in1;
        BSTATE : out = in2;
        endcase
    end
endmodule
```

In the above example, the W263 rule does not report a violation when the case label width is less than the selector width, since the case labels ASTATE and BSTATE are parameters.

Default Severity Label

Warning

Rule Group

Lint_Elab_Rules, Case

Reports and Related Files

None

W332

Not all cases are covered - default clause will be used

Language

Verilog

Rule Description

The W332 rule flags case constructs that do not have all possible clauses described and have a default clause.

The W332 rule ignores case constructs that have Synopsys full_case pragma specified.

While such case constructs are allowed, the W332 rule informs that the default clause will almost certainly be exercised.

By default, the *W332* rule reports violation for priority cases. Set the value of the *ignore_priority_case* parameter to yes to ignore violations for the priority cases.

Message Details

The following message appears at the first line of a case construct that does not have all possible clauses described and has a default clause:

```
Not all cases are covered in case statement: default case may be used
```

Rule Severity

Warning

Suggested Fix

No fix required, but check default case carefully to make sure you cover all default possibilities.

W337

Reports illegal case construct labels

When to Use

Use this rule to report violation for illegal **case** construct labels.

Description

The *W337* rule reports illegal case construct labels, which includes the following types:

- Labels of type **real**

  The case constructs are evaluated by comparing the case construct selector value against each case clause label in turn. As real values cannot be accurately compared, it is recommended not to use real values as case labels.

- Labels containing **X**, **Z**, or **?**

  The case labels containing **X** have meaning only for the casex constructs. The case labels containing **Z** and **?** have meaning for the casex and casez constructs.

- **time** and **event** variables as labels.

By default, the W337 rule does not report violations for string variables, which are used as case items. Set the value of the *strict* parameter to yes to report violations for such cases.

By default, the *W337* rule reports violation for priority cases. Set the value of the *ignore_priority_case* parameter to yes to ignore violations for the priority cases.

Language

Verilog

Parameters

strict: The default value is no. Set the value of the strict parameter to yes to report violations for string variables, which are used as case items.

Constraints

None

Messages and Suggested Fix

The following message appears at the location of a rule-violating case construct label.

```
[WARNING] Illegal value as case item
```

Potential Issues

Violation may arise when an illegal value is used as a case construct label.

Consequences of Not Fixing

Real values cannot be accurately compared.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location of the illegal case item.

Use only integer or reg values as case selectors and in case expressions. If you need to use **X**, **Z**, or **?** in a label, use the appropriate case type.

Example Code and/or Schematic

```
module top;
   reg source;
   reg dest;
   reg clk;
   reg sel;
   reg out;


always @(posedge clk)
   case (sel)
      out          :    dest <=source;
      1            :    dest <=source;
      8'b11110000    :    dest <=source;
      8'b1111xxxx    :    dest <=source;
      8'b1111zzzz    :    dest <=source;
      8'b1111????    :    dest <=source;
```

```
    0.1             :    dest <=source;
  default           :    dest <=source;
endcase

endmodule
```

In the above example, the W337 rule reports a violation as illegal case construct labels, such as, **x**, **z**, and **?** are used in the design. The rule will also flag a violation as a **real** value (**0.1**) is used as a **case** label.

Default Severity Label

Warning

Rule Group

Case

Reports and Related Files

None

W398

Reports a case choice when it is covered more than once in a case statement

When to Use

Use this rule to identify duplicate **`case`** constructs.

Rule Description

The *W398* rule reports duplicate choices in case construct. It is most likely that this error occurs while using don't-care values in the case labels. It is the only way that this condition can occur in VHDL.

By default, the *W398* rule reports violation for priority cases. Set the value of the *ignore_priority_case* parameter to yes to ignore violations for the priority cases.

When creating waivers, rule ignores line numbers.

Consider the following Verilog example:

```
casex (choice)
   100: ...
   10?: ...// violation
   default: ...
endcase;
```

Consider the following VHDL example:

```
case (choice)
   when "100" =>  ...
   when "10-" =>  ...-- violation
   when others => ...
end case;
```

Rule Exceptions

The *W398* rule does not process case constructs when the **`parallel_case`** pragma is specified.

Language

Verilog, VHDL

Parameters

- *strict*: The strict parameter is applicable only for the Verilog rule. Default value for this parameter is **no**. In this case, the rule does not report violation for the non-static case labels. Set the value of the parameter to **yes** or <*rule-name*> to report violations for non-static case labels.
- *waiver_compat*: Default value is no. If you set the value of this parameter to **yes** or **<rule-name>**, it ensures that the rule does not generate the line number information in the first run itself. Thus waivers work correctly even if the line numbers of the RTL gets changed in the subsequent runs.

Constraints

None

Messages and Suggested Fix

Verilog

The following message appears at the location where a duplicate choice <*duplicate_case*> of an already existing case label <*existing-case-label*> in **case** construct is encountered.

```
[WARNING] Case '<duplicate_case>' covered more than once at
'<existing-case-label>'
```

Potential Issues

Violation may arise when a **case** choice is covered more than once in a **case** statement.

Consequences of Not Fixing

Repeated clauses in case construct are redundant and may result in an error for some applications. All such redundant choices are not covered by simulator and synthesis tool.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the duplicate case label is used.

To find the initial usage of the overlapping case label, scroll up the HDL Viewer window or search backward for the second expression given in the

violation message.

To fix this problem, carefully examine offending case items and modify the coding to avoid overlap.

VHDL

The following message appears at the location of a statement <*stmt*> when a clause value <*valuec*> overlaps the value of another clause <*valuee*> found earlier at a line <*num*>.

`[WARNING] Choice <valuec> overlaps with another entry <valuee> (line no <num>) in <stmt> statement`

Where <*stmt*> can be a **`case`** construct or a selected signal assignment statement.

Potential Issues

Violation may arise when a clause value overlaps with another clause value in a case construct or in a signal assignment statement.

Consequences of Not Fixing

Repeated clauses in a case construct or a signal assignment statement is redundant and may result in an error for some applications. All such redundant choices are not covered by simulator and synthesis tool.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the duplicate case label is used.

To find the initial usage of the overlapping case label, scroll up the HDL Viewer window or search backward for the second expression given in the violation message.

To fix this problem, carefully examine offending case items and modify the coding to avoid overlap.

Example Code and/or Schematic

Verilog Example

Consider the following example where the choice **`2'b00`** is repeated:

```
module test(a, b, c, o1);
  input [3:0]c;
  input [3:0]a;
```

```
  input [1:0]b;
  output [3:0]o1;
  reg [3:0]o1;
 always@(c or b)
  begin
    case(b)
      2'b00 : o1 = a || c;
      2'b01 : o1 = a & c;
      2'b00 : o1 = a | c;     //violation by default
          c : o1 = a ^ b;
      2'b11 : o1 = a ^ c;
        c : o1 = a || b;    //violation when strict is set
    default : o1 = 0;
    endcase
  end
endmodule
```

In the above example, the *W398* rule reports a violation for the case label **2'b00** because it is repeated within the same case block. When the strict parameter is set, the W398 rule reports a violation for the case label *c* because it is a non-static case label.

VHDL Example

Example of a CASE Construct

In the following example, the clauses **0-0** and **-00** (at line 13) overlap:

```
12: case sig3 is
13:   when "-00" => sig1 <= sig2;
14:   when "0-0" => sig1 <= sig2 + 10;
15:   when "000" => sig1 <= sig2 + 20;
16:   when "010" => sig1 <= sig2 + 20;
17:   when "011" => sig1 <= sig2 + 20;
18:   when "100" => sig1 <= sig2 + 20;
19:   when "101" => sig1 <= sig2 + 20;
20:   when "110" => sig1 <= sig2 + 20;
21:   when "111" => sig1 <= sig2 + 20;
22:   when others => null;
23: end case;
```

Example of a Select Statement

In the following example, the clauses **---** and **ZX1** overlap:

```
with a select
  b <= 0 when "ZX1",
       6 when "ZX0",
       8 when "---",
       unaffected when others;
```

Default Severity Label

Warning

Rule Group

Case

Reports and Related Files

None

W453

Large case constructs should not be used

Language

Verilog

Rule Description

The W453 rule flags case constructs with large selector bit-width and large number of case clauses.

By default, the W453 rule flags case constructs with the case select condition bit-width greater than 16 bits and the number of case clauses greater than 20. Use the *casesize* rule parameter to change the default values.

The W453 rule flags case constructs where both conditions are violated (select bit-width *and* number of clauses). Violation of one condition is not a rule-violation.

When case select expression has larger width, number of case clauses becomes larger to cover all the cases. Also, this is a readability problem for user.

NOTE: *The W453 rule supports generate-for and generate-case blocks.*

Message Details

The following message appears when a case construct with a large condition bit-width *<bit-width>* and more number of case clauses *<num-of-clauses>*, is encountered:

```
case construct is too wide with condition bit-width '<bit-width>' bits, and '<num-of-clauses>' clauses. [Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical path of the containing scope.

Severity

Warning

Suggested Fix

In the case of large case selector width, try to break the case statement into several case statements with small case selector width.

W551

Ensure that a case statement marked full_case or a priority/unique case statement does not have a default clause.

When to Use

Use this rule to identify the case statements that are marked either **full_case** or **priority/unique_case** and have a **default** clause.

Description

The W551 rule reports violation for the following constructs with a default clause:

- case constructs with full_case pragma
- priority/unique case constructs

The W551 rule checks for priority cases inside always blocks, initial blocks, tasks and functions in all scopes like generate block, packages, global scope, and interfaces.

The rule reports violation when a priority modifier is used with **case**, **casex**, or **casez** statements with **default** clause as one of its case selection item.

By default, the *W551* rule reports violation for priority cases. Set the value of the *ignore_priority_case* parameter to yes to ignore violations for the priority cases.

Language

Verilog

Default Weight

5

Parameter(s)

check_case_type: Default value is all. Therefore, the W551 rule checks for priority, unique, and full_case cases. Set the value of the parameter to either priority or unique or full_case to check for priority, unique and full_case cases, respectively. You can also specify multiple values for the parameter so as to check for the specified cases.

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the first line of a case construct with the default clause and full_case pragma or priority/unique case construct with a default clause:

```
[WARNING] Case statement marked <type> has a default clause
```

Where, <*type*> can be **full_case** or **priority_case** or **unique_case**.

Potential Issues

The W551 rule may report a violation because of the following reasons:

- Full_case pragma or the default clause is redundant.
- In case of priority/unique case the default clause is redundant.

Consequences of Not Fixing

This rule points to the need of intent review. If the designer meant to specify full_case then there should not be a reason for a default clause. A review at the RTL coding stage can help you uncover subtle design issues.

How to Debug and Fix

Use either the full_case pragma directive or the default clause in a case construct. For priority/unique case constructs, default clause is redundant.

To fix the violation, remove either the full_case pragma or the default clause.

Example Code and/or Schematic

Example 1

Consider the following example:

```
module test(inp, outp, sel);
  input [3:0] inp;
  input [1:0] sel;
  output outp;
  reg outp;

  always @(sel or inp)
```

```
    case(sel)
//synopsys full_case
      2'bx0 : outp = inp[0];
      2'b01 : outp = inp[1];
      2'b10 : outp = inp[2];
      default: outp = inp[3];
    endcase

endmodule
```

In the above example, the W551 rule reports a violation as the case statement specified full_case has a default statement.

Example 2

Consider the following example:

```
module test;
wire a, b, c;
logic x;
always @ (a, b, c)
begin
  priority casez (c)
  a: x = 1'b0;
  b: x = 1'b1;
  default: x = x;
  endcase
end
endmodule
```

In the above example, the rule reports a violation as the default clause is used with a priority case.

Default Severity Label

Warning

Rule Group

Case

Reports and Related Files

No related reports or files.

Lint_Reset Rules

The SpyGlass lint product provides the following reset related rules:

Rule	Flags...
W392	Reset/Set signals that have been used in both negative and positive polarity in the same architecture
W395	process/always blocks that use multiple asynchronous set/reset signals
W396	Processes that have a clock signal but no asynchronous reset signal
W402	Reset signals that are internally generates at level other than the top-level of the design
W402a	Synchronous reset signals that are not inputs to the module
W402b	Asynchronous reset signals that are not inputs to the module
W448	Signals that are used both as synchronous and asynchronous reset/set in a design
W501	Reset ports in component instantiations that are connected to static names (generic or constant)

W392

Reports reset or set signals used with both positive and negative polarities within the same design unit

When to Use

Use this rule to identify reset or set signals used with both positive and negative polarities.

Description

The *W392* rule reports reset or set signals that have been used in both negative and positive polarities in the same module.

The *W392* rule not check sync reset/set, when flops have asynchronous reset.

Consider the following example:

```
module test ( M4 , M5 , clk , rst , syncRst , in ) ;
input in, clk , rst , syncRst;
output reg M4 , M5 ;

always @ (posedge clk or negedge rst) begin
if (!rst) begin
M4 <= 1'b0;
end
else if ( syncRst ) begin
M4 <= in ;
end
else begin
M4 <= 1'b0;
end
```

```
 end

always @ (posedge clk or negedge rst) begin
if (!rst) begin
M5 <= 1'b0;
end
else if ( !syncRst) begin
M5 <= in ;
end
else begin
M5 <= 1'b0;
end
end

endmodule
```

In the above example, the M4 and M5 flops have asynchronous reset in the same polarity. Therefore, the rule does not check synchronous reset/set behavior.

Rule Exceptions

The *W392* rule fails to run if you set the *fast* rule parameter to **yes** and SpyGlass lint product is run. Also, this rule does not report a violation for set/reset pin driven by constant value that can be propagated.

Language

Verilog, VHDL

Parameters

- *checksyncreset*: Default value is **yes**. This indicates the *W392* rule considers synchronous resets for rule checking. The rule tries to find the synchronous resets from RTL **always** blocks. The rule also honors the *reset -sync* constraint and looks for the out-of-phase flip-flops. Set this parameter to **no** to ignore synchronous resets.
- *check_complete_design*: Default value is **no**. This indicates the *W392* rule checks within the design unit. Set this parameter to **yes** to check within the complete design.
- *fast*: Default value is **no**. Set this parameter to **yes** to suppress synthesis. Hence, the *W392* rule will be switched off.
- *waiver_compat*: Default value is no. If you set the value of this parameter to **yes** or **<rule-name>**, it ensures that the rule does not generate the line number information in the first run itself. Thus waivers work correctly even if the line numbers of the RTL gets changed in the subsequent runs.
- *check_latch*: Default value is **no** and the rule reports the edge triggered cases only. Set the **check_latch** parameter to **yes** to enable the rule to report non-edge triggered cases also.

Constraint(s)

- *set_case_analysis*: The *W392* rule supports the constant value propagation. Therefore, this rule does not traverse on a blocked path. For black boxes, this rule traverses further only if the *assume_path* constraint is specified in the SGDC file.

Tcl Attributes

- **is_reset_used_with_both_polarity**: This Tcl attribute returns those reset or set signals that are used as both positive and negative polarity in the same design unit.

  For example:

```
sg_shell> set_pref dq_design_view_type flat
sg_shell> set net_iter [get_nets * -
filter"is_reset_used_with_both_polarity == true"]
```

For more details, refer to the **`is_reset_used_with_both_polarity`** attribute in the *Base Attributes* section of the *SpyGlass Tcl Shell Interface User Guide.*

Messages and Suggested Fix

Message 1

The following message appears at the location where a different polarity of a reset signal <*rst-name*> is used when another polarity has already been used in the same module <*module-name*> in a file <*file-name*>, line <*num*>.

```
[WARNING] Reset '<rst-name>' also used with different polarity
in module '<module-name>'(file: '<file-name>' line: '<num>')
```

Potential Issues

Violation may arise when two different IP blocks, each operating perfectly fine with a given polarity of reset/set, are connected together at a SoC level.

Also, in some cases, the designer might intend to have mutually exclusive operations (usually different modes of design).

Consequences of Not Fixing

When both polarities of reset/set signal are used, one logic block always remain in a reset/set state. If a reset/set signal is high, the logic that operates on positive reset/set polarity would be reset/set. However, when the reset/set signal is de-asserted, some other portion of logic would be reset/set. The usage leads to mutually exclusive blocks of logic.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location of first usage of the reset.

Also, the corresponding Incremental schematic displays the following information:

- Negative polarity Reset Path from Reset Source to Flip-Flop Reset Pin.
- Positive polarity Reset Path from Reset Source to Flip-Flop Reset Pin.
- Flip-Flop with Negative polarity Reset.
- Flip-Flop with Positive polarity Reset.

The violation message also shows the file and line number of reset used with different polarity. User can trace the RTL code with this information to see the other usage of the reset.

NOTE: *If the Incremental schematic is not supported for the violation, it means the violation is related to synchronous resets.*

In majority of cases, such behavior is unintentional. To fix this problem, review the design and change the reset logic so that externally reset is active only in one state.

Message 2

The following message appears if the *check_latch* parameter is set as the rule will check for latches:

```
[WARNING] Reset <rst-name> used with different polarity in the design at latches <latch1-name> and <latch2-name>
```

Potential Issues

Violation may arise when two different IP blocks, each operating perfectly fine with a given polarity of reset/set, are connected together at a SoC level.

Also, in some cases, the designer might intend to have mutually exclusive operations (usually different modes of design).

Consequences of Not Fixing

When both polarities of reset/set signal are used, one logic block always remain in a reset/set state. If a reset/set signal is high, the logic that operates on positive reset/set polarity would be reset/set. However, when the reset/set signal is de-asserted, some other portion of logic would be reset/set. The usage leads to mutually exclusive blocks of logic.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location of first usage of the reset.

Also, the corresponding Incremental schematic displays the following information:

- Negative polarity Reset Path from Reset Source to Latch Reset Pin.
- Positive polarity Reset Path from Reset Source to Latch Reset Pin.
- Latch with Negative polarity Reset.
- Latch with Positive polarity Reset.

The violation message also shows the file and line number of reset used with different polarity. User can trace the RTL code with this information to see the other usage of the reset.

Example Code and/or Schematic

Verilog

Example 1:

Consider the following example:

```
module test3( Q1, Q2, DataIn, C_SCLK1, C_SRST1 );
input [2:0] DataIn;
input C_SCLK1;
input  C_SRST1;
output [2:0]  Q1, Q2;
reg clk;
reg [2:0] Q1,Q2,Q3;
always @( posedge C_SCLK1 or posedge  C_SRST1 )
begin
if (C_SRST1)                    //VIOLATION
  Q1 = 2'b11;
else
  Q1 = DataIn[1:0];
end
always @( posedge C_SCLK1 or negedge C_SRST1 )begin
if (!C_SRST1)
  Q2 = 2'b11;
else
  Q2 = DataIn[1:0];
end
endmodule
```

In the above example, W392 rule reports a violation as both polarities of reset signal, **C_SRST**, is used in the same module.

Example 2:

Consider the following example, where the *check_latch* parameter has been set to **yes**:

```
module test (y1, y2, data1, data2, enable, preset, clear);
```

```
  input data1, data2, enable, preset, clear;
  output y1, y2;
  reg y1, y2;

  always @(enable or clear or preset or data1)
    begin: forset1_PROC
    if (clear)
      y1 = 0;
    else
      if (preset) //high active "preset"
        y1 = 1;
      else
        if (enable)
          y1 = data1;
    end

  always @(enable or clear or preset or data2)
    begin: forset2_PROC
    if (clear)
      y2 = 0;
    else
      if (!preset) //low active "preset"
        y2 = 1;
      else
        if (enable)
          y2 = data2;
    end
endmodule
```

For the above example, the rule reports the following violation message:

```
Reset 'test.preset' used with different polarity in the design
at latches 'test.y2' and 'test.y1'
```

VHDL

Consider the following example where both polarities of the **`rst`** reset signal have been used in same architecture.

```
library IEEE;
use ieee.std_logic_1164.all;

entity ent is
  port (d, clk, rst: in bit; q1,q2: out bit);
end ent;

architecture behav of ent is
  begin
    process(clk,rst)
      begin
        if (rst = '1') then
          q1 <= '0';
        elsif clk'event and clk ='1' then
          q1 <= d;
        end if;
    end process;
    process(clk,rst)
      begin
        if rst = '0' then
          q2 <= '0';
        elsif clk'event and clk ='1' then
          q2 <= d;
        end if;
    end process;
end behav;
```

The schematic for this example is given below:

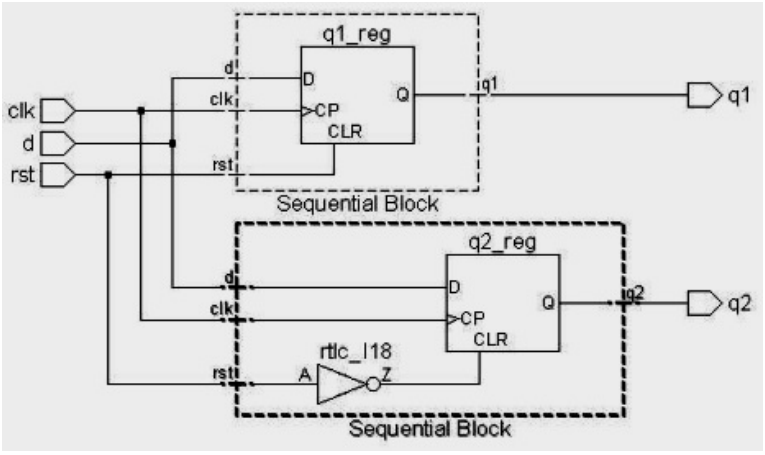


FIGURE 1. Incremental schematic

Default Severity Label

Warning

Rule Group

Lint_Reset

Reports and Related Files

None

W395

Multiple asynchronous resets or sets in a process or always may not be synthesizable

Language

Verilog, VHDL

Rule Description

The *W395* rule reports if more than one asynchronous reset or set signals exist in the same **process** or **always** block.

Some synthesis tools may be able to handle **process** or **always** blocks with multiple asynchronous resets or sets but such designs are unlikely to be portable.

NOTE: *The rule supports generate-if and generate-for blocks.*

Message Details

Verilog

The following message appears at the start of an **always** block that uses multiple asynchronous set or reset signals (*<rst-list>*):

```
Multiple set/reset signals <rst-list> used in the always block may not be synthesizable [Hierarchy: '<hier-path>']
```

VHDL

The following message appears at the start of a **process** block that uses multiple asynchronous set or reset signals (*<rst-list>*):

```
Multiple set/reset signals <rst-list> used in the process block may not be synthesizable
```

Severity

Warning

Suggested Fix

Precompute a single condition under which the **process** or **always** block should be reset or set, and use that condition as a reset or set inside the

process or **always** block.

Examples

VHDL

Consider the following example where the **process** block uses two reset signals (**rst1** and **rst2**):

```
process(clk1,rst1,rst2)
  begin
    if (rst1 = '1') then
      t <= '0';
    elsif (rst2 = '1') then
      t <= '0';
    elsif (clk1'event and clk1 = '1') then
      t <= not in1;
    end if;
end process;
```

For this example, SpyGlass generates the following message:

```
Multiple set/reset signals 'rst1', 'rst2' used in the process
block may not be synthesizable
```

Verilog

Consider the following example where the **always** block uses two reset signals (**rst1** and **rst2**):

```
always @(posedge clk, posedge rst1, posedge rst2)
    begin
        if (rst1)
            t <= 1'b0;
        else if (rst2)
            t <= 1'b0;
        else if(clk)
            t <= in1 & in2;
    end
```

For this example, SpyGlass generates the following message:

```
Multiple set/reset signals 'rst1', 'rst2' used in the always
block may not be synthesizable
```

W396

A process statement has clock signal, but no asynchronous reset signal

Language

VHDL

Rule Description

The W396 rule flags processes that have a clock signal but no asynchronous reset signal.

If a process statement has a clock signal then an asynchronous reset signal must be given so that the corresponding sequential circuit starts from a known state.

NOTE: *You can enable the W396 rule by either specifying the* **`set_goal_option addrules W396`** *command. However, this rule will not run if you set the fast rule parameter to* **`yes`** *and* **`lint`** *product is run.*

Message Details

Following message appears at the start of a process that has a clock signal but no asynchronous reset signal:

```
Process has a clock signal, but not an asynchronous reset signal
```

Severity

Warning

Suggested Fix

Use this rule to check processes which are not asynchronously reset. If you are using only synchronous reset, ignore the rule.

Examples

Consider the following example where the process has a clock signal **`clk`** but no asynchronous reset signal:

```
entity test is
```

```
  port(in1: in bit;
       clk: in bit;
         q: out bit);
end test;

architecture behav of test is
  begin
    process(clk)
       begin
          if(clk'event and clk = '1') then
             q <= in1;
           end if;
    end process;
end behav;
```

For this example, SpyGlass generates the W396 rule message.

W402

If you internally generate reset signals, do so in a single module instantiated at the top-level of the design

Language

VHDL

Rule Description

The W402 rule flags reset signals that are internally generated at level other than the top-level of the design.

Internal reset generation creates significant problems for SpyGlass DFT methodologies. If you must gate resets, you should localize gating to one module at the top-level of the design and provide a method to disable gating in test mode.

The W402 rule reports violation if any one or more condition described below occurs:

- Top-level module has asynchronous resets and they are not ports
- Asynchronous resets are generated in more than one instance
- All asynchronous resets are generated in an instance, but all output ports of that instance are not reset ports.

NOTE: *You can enable the W402 rule by either specifying the* **`set_goal_option addrules W402`** *command. However, this rule will not run if you set the fast rule parameter to* **`yes`** *and* **`lint`** *product is run.*

Message Details

Following message appears at the location where a reset signal *<rst-name>* is generated at a level other than the top-level of the design:

```
Gated reset <rst-name> is not generated in single module at top level
```

Severity

Warning

Suggested Fix

Consider gating the generated reset with an externally controllable test mode signal.

W402a

Synchronous reset signal is not an input to the module

Language

Verilog

Rule Description

The W402a rule flags synchronous reset signals that are not inputs to the module.

A signal appears to be used as a synchronous reset and is generated inside the same module. This can cause difficulties in verification where it may be desirable to start all logic in a reset state.

The W402a rule flags gated, buffered, or internally generated synchronous reset signals.

NOTE: *Synchronous reset detection in the SpyGlass lint product is heuristic. Hence, this rule may report some false errors.*

Message Details

The following message appears at the location where a reset signal <*rst-name*> is used in the module <*module-name*> that is not an input to the module:

```
Reset '<rst-name>' is not an input to the module '<module-name>'
```

Severity

Warning

Suggested Fix

If the error is false, consider waiving it.

Examples

Consider the following example where synchronous reset signal **rst** is not an input to the module:

```
module test(in, out, clk);
```

```
  input [3:0] in;
  input clk;
  output [3:0] out;
  reg [3:0] out;
  reg rst;

  always@(posedge clk)
    begin
      if(rst) out <= 0;
      else out <= in;
    end
endmodule
```

For this example, SpyGlass generates the following message:

```
Reset 'rst' is not an input to the module 'test'
```

W402b

Asynchronous set/reset signal is not an input to the module

Language

Verilog

Rule Description

The W402b rule flags asynchronous set/reset signals that are not inputs to the module.

The W402b rule flags gated, buffered, or internally generated asynchronous set/reset signals.

Such set/reset signals may cause problems for SpyGlass DFT tools.

NOTE: *You can enable the W402b rule by specifying the* **`set_goal_option addrules W402b`** *command. However, this rule will not run if you set the fast rule parameter to* **`yes`** *and SpyGlass* **`lint`** *product is run.*

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameters

waiver_compat: Default value is no. If you set the value of this parameter to **`yes`** or **`<rule-name>`**, it ensures that the rule does not generate the line number information in the first run itself. Thus waivers work correctly even if the line numbers of the RTL gets changed in the subsequent runs.

Message Details

The following message appears at the location where the output of a flip-flop *<flop-name>* is first assigned when its asynchronous *<pin-type>* signal *<pin-name>* is not an input to the module *<module-name>*:

`<pin-type> '<pin-name>' to flop '<flop-name>' (line <num>, file <file-name>) is gated or internally generated (in module <module-name>)`

where *<pin-type>* is **`set`** or **`reset`** and *<pin-name>* is the name of set/reset.

Rule Severity

Warning

Suggested Fix

Consider gating the generated set/reset with an externally controllable test mode signal.

Examples

Example 1

Consider the following example where asynchronous reset signal **rst** is not an input to the module:

```
module test(in, out, clk);
  input [3:0] in;
  input clk;
  output [3:0] out;
  reg [3:0] out;
  reg rst;

  always @(posedge clk or posedge rst)
    begin
      if(rst) out <= 0;
      else if(!rst) out <= 1;
      else out <= in;
    end
endmodule
```

For this example, SpyGlass generates the following message:

```
'test.rst' to FF 'out[3]' (line 23, file test.v) is gated or
internally generated (in module test)
```

Example 2

Consider the following example:

```
module top(input a1,a2,a3,a4,d,clk , output reg o1 , output
reg o2);
wire rst1,rst2;
slave1 inst1(a1,a2,rst1);
```

```
slave2 inst2(a3,a4,act_rst);
assign act_rst = rst1;

always @(posedge clk or posedge act_rst)
    if(act_rst) o1 <= 1'b0;
    else o1 <= d;
endmodule

module slave1(input x, input y, output z);
assign z = x & y;
endmodule

module slave2(input x, input y, output z);
assign z = x & y;
endmodule
```

For the above example, by default, W402b rule reports violation for reset "act_rst" as it is internally generated.

To ignore violations for reser, act, rst, set the value of the reset_gen_module parameter as shown below:

set_parameter reset_gen_module slave2

For this example, the violation is ignored because reset, act_rst, is gated by instance of module, slave2.

W448

Reset/set is used both synchronously and asynchronously

Language

Verilog, VHDL

Rule Description

The *W448* rule reports signals that are used both as synchronous reset/set and asynchronous reset in a design.

Using a signal both as a synchronous and asynchronous reset/set can be confusing and difficult to debug. An exception to this is the generation of an "asynchronous assert, synchronous deassert" reset, but such cases should be localized.

Synchronous reset/set function is triggered with respect to a clock edge/phase. The reset/set signal can transition well in advance meeting the required set up and hold times. However, the resetting effect of a synchronous reset/set is synchronized to the clock edge.

The asynchronous reset/set takes place immediately after the assertion of the reset/set signal. Here, no clock edge/phase is required for resetting function.

If a single signal is used both synchronously and asynchronously then the cycle partitioning intent should be reviewed. In such cases, the integrity of scan and capture operations may get hampered during testing phase. However, there are some rare cases when the requirement is to use a reset/set signal both synchronously and asynchronously.

Use the following option to generate the *W448_Report*, which lists the reset nets that are used synchronously and asynchronously:

```
-report=W448_Report
```

NOTE: *In Lint, synchronous reset/set detection is heuristic and subject to false errors.*

NOTE: *You can enable the W448 rule by specifying the* **`set_goal_option addrules W448`** *command. However, this rule will not run if you set the fast rule parameter to* **`yes`** *and SpyGlass* **`lint`** *product is run.*

Design Impact

Functionality (bug escape) and testability

Message Details

Following message appears at the location where a reset signal *<rst-name>* is declared that is used both as a synchronous reset and asynchronous reset in the same design:

```
Reset '<rst-name>' used both synchronously and asynchronously
```

Rule Severity

Warning

Suggested Fix

Check each case to ensure you understand the implications of using both edge-based and state-based reset. Try to prefer usage based on just one style and localize variants as much as possible

Examples (Verilog)

Consider the following example where signal **`rst`** is used both as synchronous reset as well as asynchronous reset:

```
module test(in, out1, out2, clk, rst);
  input in;
  input clk, rst;
  output out1, out2;
  reg out1, out2;
  always @(posedge clk or posedge rst)
  begin
    if(rst) out1 <= 0;
    else out1 <= in;
  end

  always@(posedge clk)
  begin
    if(rst) out2 <= 0;
    else out2 <= in;
  end
endmodule
```

For this example, SpyGlass generates the following message:

Reset 'rst' is used as both synchronous and asynchronous reset

The schematic for the above example is shown below:

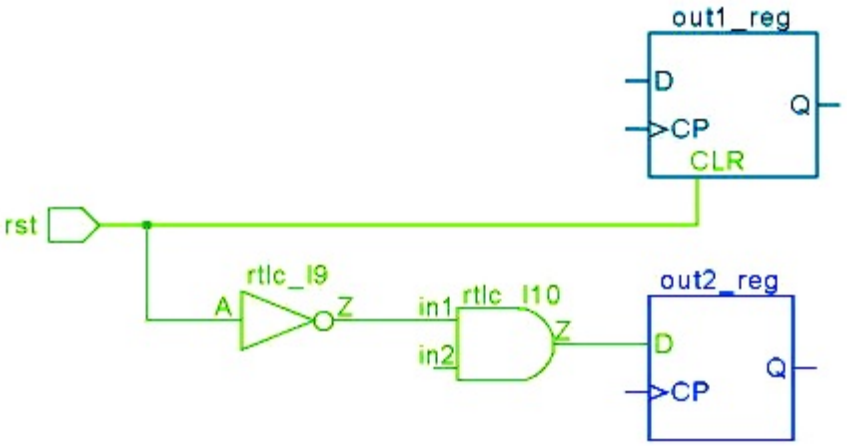


FIGURE 2. Incremental schematic

Examples (VHDL)

Consider the following example where the signal **reset** is used both as synchronous and asynchronous resets in the same architecture:

```
entity test is
  port(d: in Bit;
       clk: in Bit;
     reset: in Bit;
        q1: out Bit;
        q2: out Bit);
end test;

architecture structure of test is
  begin
    P1: process(clk, reset)
      begin
        if (reset = '1') then
           q1 <= '0';
```

```
        elsif (clk'event and clk = '1') then
           q1 <= d;
         end if;
    end process P1;

    P2: process(clk)
      begin
        if (clk'event and clk = '0') then
          if (reset = '1') then
             q2 <= '0';
          else
             q2 <= d;
          end if;
         end if;
    end process P2;
end structure;
```

For this example, SpyGlass generates the following message:

```
Reset 'reset' is used as both synchronous and asynchronous
reset
```

The schematic for the above example is shown below:

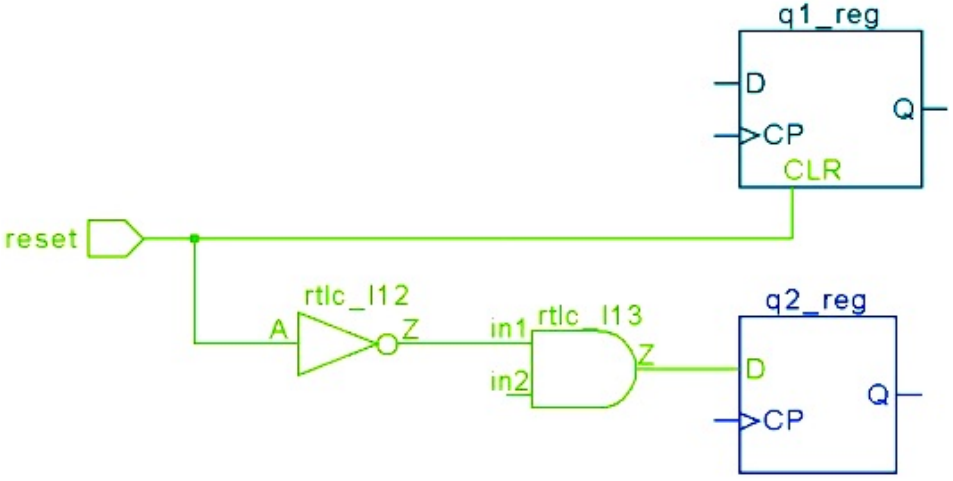


FIGURE 3. Incremental schematic

W501

A connection to a reset port should not be a static name

Language

VHDL

Rule Description

The W501 rule flags reset ports in component instantiations that are connected to static names (generic or constant).

Message Details

The following message appears at the location where the reset port *<rst-name>* is connected to a generic or constant in a component instantiation:

`Connection to <reset> port '<rst-name>' is a static name (<type>)`

Where *<type>* can be **`generic`** or **`constant`**.

Severity

Warning

Suggested Fix

Always connect a real signal. Tie that signal off if you really want to disable the reset.

Examples

Consider the following example where the reset port **`rst`** of instantiation **`INST1`** is tied to a constant value **`0`**:

```
...
INST1: test port map (  d => input(0),
                      clk => clk,
                      rst => '0',
                        q => s1);

INST2: test port map (d => s1,
```

```
                    clk => clk,
                    rst => input(1),
                      q => output);
...
```

For this example, SpyGlass generates the following message:

```
Connection to reset port 'rst' is a static name (constant)
```

Also, the reset port **rst** of instantiation **INST2** is connected to signal **input(1)** which is not a constant or generic. Hence, SpyGlass does not flag a message here.

Lint_Clock Rules

The SpyGlass lint product provides the following clock related rules:

Rule	Flags...
W391	Modules where both edges of a clock are used to describe sequential elements
W401	Clock signals that are not input to the module where they are used
W422	Event control descriptions with more than one clock
W500	Clock ports of component instances that use bus, indexed-name, sliced-name, expressions, or concatenation

W391

Reports modules driven by both edges of a clock

When to Use

Use this rule to identify modules driven by both edges of a clock.

Description

The *W391* rule reports modules where both edges of a clock are used to describe sequential elements.

Rule Exceptions

The *W391* rule fails to run if you set the *fast* rule parameter to **yes** and SpyGlass **lint** product is run.

Also, no rule checking is done when a clock is driven by a constant value, which is propagated through an assignment or by using the SGDC file.

Language

Verilog, VHDL

Parameters

- *check_complete_design*: Default value is **no**. This indicates the *W391* rule checks within the design unit. Set this parameter to **yes** to check within the complete design.

Constraints

- *set_case_analysis*: The *W391* rule supports the constant value propagation. Therefore, this rule does not traverse on a blocked path.
- *assume_path*: A path through a black box is treated as a blocked path if there is no *assume_path* constraint specified in the SGDC file for that black box.

Tcl Attributes

- **is_clock_used_with_both_edges**: This Tcl attribute returns the clock signal that is driven on both the edges.

  For example:

```
sg_shell> set_pref dq_design_view_type flat
sg_shell> set net_iter [get_nets * -filter
"is_clock_used_with_both_edges == true"]
```

For more details, refer to the **is_clock_used_with_both_edges** attribute in the *Base Attributes* section of the *SpyGlass Tcl Shell Interface User Guide*.

Messages and Suggested Fix

For a design unit <*module-name*> where different edges of a clock signal <*clk-name*> have been used to define sequential elements <*hier-flop1-name*> and <*hier-flop2-name*>, respectively, the following message appears at the location where the output net of a second flip-flop <*hier-flop2-name*> is first set.

```
[WARNING] Design unit '<module-name>' uses both edges of clock
'<clk-name>' (1st at flop '<hier-flop1-name>', 2nd at flop
'<hier-flop2-name>')
```

Potential Issues

Violation may arise when a module uses both edges of a clock.

Consequences of Not Fixing

As a result of using both the edges, the behavior of that module gets dependent on the duty cycle of the clock.

How to Debug and Fix

Double-click the violation message to view the Incremental Schematic corresponding to the message. The Incremental Schematic displays the following information:

- Negative Edge Clock Path from Clock Source to Flip-Flop Clock Pin.
- Positive Edge Clock Path from Clock Source to Flip-Flop Clock Pin.
- Flip-Flop with Negative Edge Clock.
- Flip-Flop with Positive Edge Clock.

To fix the problem, avoid using both edges of a clock in general. If absolutely necessary, localize such cases as much as possible, that is, use different edges for different modules.

Example Code and/or Schematic

Verilog Example

Consider the following example where a flip-flop **out1** is described using the posedge of clock **clk** when a flip-flop **out2** has already been described using the negedge of the same clock **clk** in the same module **test**.

```
module test (out1, out2, in1, in2, clk);
  input  in1, in2, clk;
  output out1, out2;

  reg out1, out2;

  always @(posedge clk)
    out1 = in1;

  always @(negedge clk)
    out2 = in2;
endmodule
```

VHDL Example

Consider the following example where both edges of the **clk** clock signal are used to describe sequential elements:

```
architecture rtl of test is
  begin
    P1 : process(clk, reset1)
      begin
        if ( reset1 = '1' ) then
          q1 <= '0';
        elsif( clk'event and clk = '1' ) then
          q1 <= d;
        end if;
    end process P1;

    P2 : process(clk)
      begin
        if( clk'event and clk = '0') then
```

```
            if ( reset2 = '1' ) then
              q2 <= '0';
            else
              q2 <= d;
            end if;
          end if;
      end process P2;
end rtl;
```

Default Severity Label

Warning

Rule Group

Lint_Clock

Reports and Related Files

None

W401

Clock signal is not an input to the design unit

Language

Verilog, VHDL

Rule Description

The W401 rule flags clock signals that are not inputs to the modules where the clock signals are used to describe flip-flops.

The W401 rule flags clock signals that are gated, buffered, or internally generated in the module where they are used.

Using internally generated clocks may cause problems in timing budgeting and in SpyGlass DFT tools.

The W401 rule does not flag a violation for clock pin driven by a constant value, which can be propagated through assignment or using the SGDC file.

NOTE: *You can enable the W401 rule by specifying the* **`set_goal_option addrules W401`** *command. However, this rule will not run, if you set the value of the fast rule parameter to* **`yes`** *and SpyGlass* **`lint`** *product is run.*

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location where a rule-violating clock *<clk-name>* is used in design unit *<du-name>*:

```
Clock '<clk-name>' is not an input to design unit '<du-name>'
```

Rule Severity

Warning

Suggested Fix

Localize clock generation and gating to a single module if possible, so that

timing and test issues can be managed carefully with respect to that one module rather than in many locations in the design.

Examples (Verilog)

Consider the following example where the **DFF_clk** of flip-flop **out1** is internally generated:

```
module test(in1, in2, in3, out1);
  input in1, in2, in3;
  output out1;

  my_clock  mod1(in1, in2, in3, out1);
endmodule

module my_clock(in1, in2, clock, out1);
  input in1, in2, clock;
  output out1;

  reg DFF_clk, out1;

  always@(posedge clock)
    begin
      DFF_clk <= in1;
    end

  always@(posedge DFF_clk)
    begin
      out1 <= in2;
    end
endmodule
```

For this example, SpyGlass generates the following message:

```
Clock 'test.mod1.DFF_clk' is not an input to design unit
'my_clock'
```

Examples (VHDL)

Consider the following example where the clock signal **intClk** of flip-flop **output** is internally generated (process **FF_CLK**) in architecture

struct_CLK_Source of entity **ent_CLK_Source**:

```
entity ent_CLK_Source is
  port (data: in Bit;
       clk_in: in Bit;
        reset: in Bit;
       output: out Bit);
end ent_CLK_Source;

architecture struct_CLK_Source of ent_CLK_Source is
  signal intClk: Bit;
  begin
    FF_CLK: process(clk_in)
      begin
        if(clk_in'event and clk_in = '1') then
          intClk <= data;
        end if;
    end process FF_CLK;

    FF: process(intClk, reset)
      begin
        if (reset = '1') then
          output <= '0';
        elsif (intClk'event and intClk = '1') then
          output <= data;
        end if;
    end process FF;
end struct_CLK_Source;

entity ent_Test is
  port(d: in Bit;
      clk: in Bit;
      rst: in Bit;
  dataOut: out Bit);
end ent_Test;

architecture struct_Test of ent_Test is
  component CLK_Source
```

```
    port (data: in Bit;
          clk_in: in Bit;
           reset: in Bit;
          output: out Bit);
  end component;

  begin
    inst1: ent_CLK_Source
            port map (data => d,
                      clk_in => clk,
                       reset => rst,
                      output => dataOut);
end struct_Test;
```

For this example, SpyGlass generates the following message at the location of instantiation **inst1** of entity **ent_CLK_Source**:

```
Clock 'ent_Test.inst1.intClk' is not an input to design unit
'ent_CLK_Source(struct_CLK_Source)'
```

W422

Unsynthesizable block or process: event control has more than one clock

When to Use

Use this rule to identify the potentially unsynthesizable block or process.

Description

The W422 rule reports violation for always constructs or process constructs having event control descriptions with more than one clock.

For Verilog, the W422 rule flags potentially un-synthesizable block as shown below:

```
always @(posedge clk1 or posedge clk2) ...
```

For VHDL designs, SpyGlass recognizes the following types of constructs as clocks:

`clk'event and clk = '1'`
`not clk'stable and clk = '1'`
`wait until clk = '1'`
`wait until clk'event and clk = '1'`
`falling_edge(clk) or rising_edge(clk)`

Rule Exceptions

The W422 rule reports violation for potentially un-synthesizable block. It is possible that the block is synthesizable by some synthesis tool.

Default Weight

5

Language

Verilog, VHDL

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message appears at the location of an event control description using more than one clock:

```
[WARNING] Block might be un-synthesizable by some tool: event control has more than one clock
```

Potential Issues

Violation may arise when an event control uses more than one clock.

Consequences of Not Fixing

The violating block may not be synthesizable by some synthesis tool as an event control has more than one clock.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the start line of the **`always`** block in which more than one clock event control is used.

If you want to switch on edges of both clock signals, precompute a combined clock signal and switch on the appropriate edge of that combined signal.

VHDL

The following message appears at the first line of a process construct that has two clocks *<clk1-name>* and *<clk2-name>*:

```
[WARNING] Multiple clock signals <clk1-name>, <clk2-name> used in the process may not be synthesizable
```

Potential Issues

Violation may arise when multiple clock signals are used in a process construct.

Consequences of Not Fixing

Many synthesis tools support more than one clock inside a process, but this is not considered good coding style because the trigger conditions for the

process as a whole can be confusing.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the start line of the **process** block in which more than one clock event control is used.

Unless you know for certain that your synthesis tool can handle multiple clocks in a process, break these cases into multiple processes, each with a single clock.

Example Code and/or Schematic

Verilog

The following Verilog code shows the example of an event control construct, which is not synthesizable:

```
module mod(in1, in2, clk1, clk2, out1);
input in1, in2;
input clk1, clk2;
output out1;
reg out1;

  always@(posedge clk1 or posedge clk2)
    out1 = in1 ^ in2;

endmodule
```

The W422 rule reports a violation message for the above example.

VHDL

Consider the following example where two clocks clk1 and clk2 are used in the same process:

```
library IEEE;
use ieee.std_logic_1164.all;

entity multclk is
port(
      d,clk1,clk2 : in std_logic;
      q1      : out std_logic
    );
```

```
end multclk;

architecture arc of multclk is

begin
  process(clk1, clk2)
    begin
      if (clk1'event and clk1 = '1') then
        q1 <= '0';
      end if;
      if (clk2'event and clk2 = '1') then
        q1 <= d;
      end if;
  end process;
```

For this example, SpyGlass generates the following message:

```
Multiple clock signals clk1, clk2 used in the process may not
be synthesizable
```

Default Severity Label

Warning

Rule Group

Lint_Clock

Reports and Related Files

No related reports or files.

W500

A connection to a clock port is not a simple name

Language

VHDL

Rule Description

The W500 rule flags clock ports of component instances that use bus, indexed-name, sliced-name, selected-name, expression, or concatenation.

It is recommended to use simple names as clocks.

Message Details

Following message appears at the location where the connection to clock port *<clk-name>* of an instantiation is not a simple name:

`Connection to <clock> port '<clk-name>' is not a simple name`

Severity

Warning

Examples (VHDL)

In the following examples, the connection to clock port **clk** is not a simple name:

```
-- Bit-select connected to clock port
INST: test port map
      (d => s1, clk => input(0), q => output);

-- Expressions connected to clock port
INST: test port map
       (d => input(0), clk => (c and d), q => output);

-- Constants connected to clock port
INST: test port map
       (d=> input(0), clk=>('0' XOR '1'), q=> output);
```

Usage Rules

The following usage rules have been deprecated:

W557a	*W557b*	*W558*

The SpyGlass lint product provides the following usage related rules:

Rule	Flags...
W34	Macros that are defined but not used
W88	Memories where all their elements are not set in the design
W111	Arrays where all elements are not read in the process
W120	Variables that are declared but not used
W121	Object names that are not unique within the current scope
W123	Signal/ variable that has been read out but is never set
W143	Macros redefinitions in the same file
W154	Implicit net declarations
W175	Generics that are never used
W188	Assignments to input ports
W215	Bit-selects of integer or time variables
W216	Part-selects of integer or time variables
W240	Input ports that are never read
W241	Output ports that are not completely set
W333	UDPs (user-defined primitives) that are never instantiated
W423	Ports that are re-declared with a different range in the same module
W468	Variables used as array index that are narrower than the array width
W493	Use of shared variables with global scope
W494	Inout ports that are never used
W494a	Input ports that are never read
W494b	Output ports that are never set
W495	Inout ports that are read but never set
W497	Bus signals that are not completely set in the design
W498	Bus signals that are not completely read in the design

Rule	Flags...
W528	Signals or variables that are set but never read
W529	Preprocessor conditional directives
W557	Runs the *W557a* and *W557b* rules
W557a	This rule has been deprecated
W557b	This rule has been deprecated
W558	This rule has been deprecated

W34

Macro defined but never used

Language

Verilog

Rule Description

The W34 rule flags macros that are defined but are not used.

You may define a macro but may not use it in the design. Such a case is not an error but generally a mistake.

NOTE: *By default, the* `W34` *rule is switched off and will be deprecated in a future SpyGlass release. The problem reported by this rule can be handled by most of the commercial compilers.*

Message Details

The following message appears at the location where an unused macro *<macro-name>* is defined in the file, included through the include directive:

```
Macro '<macro-name>' is never used
```

Rule Severity

Info

Suggested Fix

Check all such cases. If you see a macro reported that you think should be used, suspect a possible typing error.

W88

All elements of a memory are not set

Language

Verilog

Rule Description

The W88 rule flags memories where all their elements are not set in the design.

The W88 rule does not check for memory elements having non-static index.

When testing logic, this indicates that the model does not fully exercise all paths into the memory and thus possible errors in address selection may be masked.

Message Details

The following message appears at the location where a memory *<mem-name>* is declared but all its elements are not set in the design:

`All elements of memory '<mem-name>' are not set [Hierarchy: '<hier-path>']`

Where, *<hier-path>* is the complete hierarchical path of the containing process.

Rule Severity

Warning

Examples

Consider the following example where all elements of memory **mem** are not set in the design:

```
module test(in1, in2, out1);
  input [3:0] in1, in2;
  output [3:0] out1;

  reg [3:0] out1;
```

```
  reg [7:0] mem [0:3];

  always@(in1)
    begin
      out1 = in1 & in2;
      mem[0] = {in1,out1};
      mem[2] = {in1,in2};
    end
endmodule
```

For this example, SpyGlass generates the following message:

```
All elements of memory 'mem' are not set [Hierarchy: ‘test’]
```

W120

A variable has been defined but is not used (Verilog)
A signal/variable has been declared but is not used (VHDL)

Language

Verilog, VHDL

Rule Description

Verilog

The *W120* rule reports the variables that are declared but are not used in scope of the module where they are defined.

The rule checks all non-port variables (**`wire`**, **`reg`**, **`integer`**, **`time`** variables, etc.) and requires that the variable must be either set, read, or both in the current scope. For vector variables, the rule requires that each bit must be either set, read, or both in the current scope. Therefore, the rule reports any bits of vector variables that are neither set nor read in the current scope.

While such constructs are allowed, the rule helps you clean up your design.

NOTE: *For Verilog, inside the nested* **`for`** *loops, signals of user-defined type like structures, interfaces, etc. are considered fully* **`set`** *if used on the left-hand side of an expression and fully* **`read`** *if used on the right-hand side of an expression.*

By default, the rule reports a violation if any bit of a variable is defined but not used in the current scope of a module. If you set the *checkfullbus* parameter to **`yes`**, the rule reports a violation only when all bits of a variable are completely unused in the current scope of a module.

By default, this rule does not process large arrays. Set the *handle_large_bus* parameter to **`yes`** to process large arrays (greater than 50,000) and report violation if not used correctly.

NOTE: *This rule does not check the same variable used in different generate blocks.*

VHDL

The *W120* rule reports signals or variables that are never used in the design.

The rule requires that the signal or variable must be either set, read, or both. For vector signals or variables, the rule requires that each bit must

be either set, read, or both. Therefore, the rule reports any bits of vector signals or variables that are neither set nor read in the current scope.

While such constructs are allowed, the rule helps you clean up your design.

For multidimensional variables, only the total count of unused bits is reported in the violation message. To see the details of the violating bits, please refer to the *SignalUsageReport*. You can access the report from the *spyglass_reports/lint* directory.

By default, the rule reports a violation if any bit of a variable is declared but not used in the current scope of an architecture. If you set the *checkfullbus* parameter to **yes**, the rule reports a violation only when all bits of a variable are completely unused in the current scope of an architecture.

By default, value *checkfullrecord* is set to **no**. Set this parameter to **yes** to not report violation for a record if at least one element of the record is used.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

Verilog

Message 1

The following message appears at the location where a variable *<var-name>* is declared but never used in the design:

```
Variable '<var-name>'<no_of_bits> declared but not used
[Hierarchy: ‘<hier-path>’]
```

Where,

- *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.
- *<no_of_bits>* is the number of unused bits of multidimensional array.

Message 2

The following message is displayed when the signal size is greater than

50,000 and the *handle_large_bus* parameter is disabled:

```
Signal '<signal-name>' size too big thus not processed, use
'set_parameter handle_large_bus yes' for enabling handling of
these signals
```

If the **handle_large_bus** parameter is not enabled, the violation for signals of size greater than 50,000 is missed.

VHDL

The following message appears at the location where a signal *<sig-name>* is declared that is never used in the design:

```
The signal '<sig-name>'<bits> is not used [Hierarchy: ‘<hier-
path>’]
```

The following message appears at the location where a variable *<var-name>* is declared that is never used in the design:

```
The variable '<var-name>'<bits> is not used [Hierarchy: ‘<hier-
path>’]
```

Where, *<bits>* refers to unused bits of vector variable/signal . For multidimensional array, *<bits>* is the total number of unused bits.

Also, the following message is generated, when the *W120* rule flags a violation for vector or multidimensional signal:

```
Please refer to 'SignalUsageReport.rpt' for details of
violating bits
```

In the Console, click on the above violation message to view the *SignalUsageReport*.

Rule Severity

Warning

Suggested Fix

Check all such cases. If you see a variable reported that you think should be used, suspect a possible typing error.

Examples (Verilog)

Example 1

Consider the following example:

```
module W120_mod2(in1, clk, out1);
  input [2:0] in1;
  input clk;
  output [2:0] out1;
  reg [2:0] out1;

  wire [4:0] net;

  always@(posedge clk)
    #20 out1 <= in1 && net[2:0];

endmodule
```

In this example, not all bits of the net are used in the **always** block. Therefore, by default, the *W120* rule reports a violation. However, if you set the *checkfullbus* parameter to **yes**, the rule does not report any violation.

Example 2

Consider the following example:

```
module W120_top(i1, clock, o1);
 inout [2:0] i1;
 input clock;
 output [2:0] o1;

  W120_mod my_mod(i1, clock, o1);

endmodule

module W120_mod(in1, clk, out1);
 input [2:0] in1;
 input clk;
 output [2:0] out1;
 reg [2:0] out1;
```

```
 wire [2:0] net;

 always@(posedge clk)
  #20 out1 <= in1;

endmodule
```

In this example, all bits of the net are completely unused in the **always** block. Therefore, by default, the *W120* rule reports a violation. The rule will also report a violation even if you set the *checkfullbus* parameter to **yes**.

Examples (VHDL)

Example 1

Consider the following example:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity entW120 is
  port(a: in std_logic;
       b: in std_logic;
       z: out std_logic
  );
end entW120;

architecture behav of entW120 is
signal sig1: std_logic_vector(2 DOWNTO 0);
begin
  process(a, b)
  begin
    z <= a or sig1(0);
  end process;
end behav;
```

In this example, not all bits of the **sig1** signal are used in the **process** block. Therefore, by default, the *W120* rule reports a violation. However, if you set the *checkfullbus* parameter to **yes**, the rule does not report any

violation.

W121

A variable name collides with and may shadow another variable

Language

Verilog

Rule Description

The W121 rule flags object names that are not unique within the current scope.

The W121 rule flags an object name that is same as the name of another object of different object type located within the same scope.

The W121 rule checks for name case variants within a specific scope, that is, the boundaries of a module, a UDP, a function, a task, or a named block. If a scope is contained in another scope (say, a named **`always`** construct in a module), the names in the containing scope (module) are also checked against the names in the contained scope (named **`always`** construct).

Names are not checked across parallel scopes. The W121 rule supports generate-if, generate-for, and generate-case blocks.

NOTE: *When creating waivers, rule ignores line numbers.*

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameters

- *waiver_compat*: Default value is no. If you set the value of this parameter to **`yes`** or **`<rule-name>`**, it ensures that the rule does not generate the line number information in the first run itself. Thus waivers work correctly even if the line numbers of the RTL gets changed in the subsequent runs.
- *ignore_genvar*: Default value is **`no`** and the *W121* rule reports violations for variables shadowed by genvar variables. Set the value of this parameter to **`yes`** to not report violation for such cases.

- *ignore_scope_names*: Default value is **no** and the *W121* rule checks for scope names. Set this parameter to **yes** to ignore rule checking on scope names.
- *ignore_macro_to_nonmacro*: Default value is no. Set the value of the parameter to yes to ignore name matching for macro to non-macro names.
- *limit_task_function_scope*: Default value is no. Set the value of the parameter to yes to ignore rule checking if a signal is declared with function/task scope as well as with module scope.

Message Details

The following message appears at the location where an object name <*name*> is declared that is same as the name of another object of different object type located within the same scope:

```
Name '<name>' is not unique - may shadow another variable
(Previously used at line no. <line-num> in <file-name>)
```

Where <*line-num*> is the line number where the variable of similar name has been defined earlier and <*file-name*> is the file in which the variable of similar name has been defined earlier.

Rule Severity

Warning

Suggested Fix

In general, this is considered as loose coding style. It is best to ensure that variables within a given scope have unique names. Take cases reported by this rule and change names so the rule is not violated.

Examples

The following example shows some cases of name clashes:

```
`define M1 1
module test (in1, in2, clk, rst, out1, out2);
  input in1, in2, clk, rst;
  output out1, out2;
  reg out1, out2;
```

```
  wire test;
  assign test = rst ? in1 : in2;
  always @( clk)
    begin : P2
      out1 = P1(in1 , rst);
      M1( in1, in2, out2);
    end
  always@(posedge clk)
    begin: block1
      if (rst)
        begin: block1
          reg out1;
          out1 = P1(in2, rst);
          M1( in1, `M1, out2);
        end
      else
        begin
          out1 = 0;
          out2 = 1;
        end
    end
  task M1;
    input in, sel;
    output out;
    out = in & sel;
endtask
  function P1;
    input in, sel;
    P1 = in & sel;
  endfunction
endmodule
```

The wire `test` has a name clash with the module `test`. Thus, SpyGlass generates the following message:

```
Name 'test' is not unique - may shadow another variable
(Previously used at line no. 2 in top.v)
```

There are two blocks named **`block1`** in the same always construct. Thus, SpyGlass generates the following message:

```
Name 'block1' is not unique - may shadow another variable
(Previously used at line no. 14 in top.v)
```

The register **out1** declared in the **always** construct has a name clash with a register with same name declared outside the construct. Thus, SpyGlass generates the following message:

```
Name 'out1' is not unique - may shadow another variable
(Previously used at line no. 10 in top.v)
```

The task **M1** has a name clash with parameter **M1**. Thus, SpyGlass generates the following message:

```
Name 'M1' is not unique - may shadow another variable
(Previously used at line no. 19 in top.v)
```

W123

Identifies the signals and variables that are read but not set

When to Use

Use this rule to identify signals and variables that are read but not set.

Description

Verilog

The *W123* rule reports violations for the variables that are read but not set in the design.

The rule determines the non-port value within a module which is read but never assigned a value. The rule also checks for all the **reg**, **wire**, and **integer** variables that are declared inside the module.

The W123 rule also considers simple non-static conditions, in addition to static conditions, involving FOR-loop index. When evaluating rule will only processes through true branches.

NOTE: *Inside the nested* ***for*** *loops, signals of user-defined type like structures, interfaces, etc. are considered fully* ***set*** *if used on the left-hand side of an expression and fully* ***read*** *if used on the right-hand side of an expression.*

For multidimensional variables, only the total count of violating bits that are read but not set, is reported in the violation message. To see the details of the violating bits, refer to the *SignalUsageReport* report in the *spyglass_reports/lint* directory.

A gate of a CMOS transistor needs to be at a defined state and electrical level of logic 1 or 0, for the transistor to function as an on-off switch. If the gate of a transistor is left floating, then the corresponding transistor can get into a **limbo** state of partially on.

In the context of a logic gate, such a partially-on transistor can lead to indeterminate logic value and analog electrical state on the output of that gate.

For an inverter, if the input is left floating, both the **n** and **p** transistors can be partially on creating a steady state path from VDD to ground.

In such cases, the output of a transistor would be in a non-digital state, that is, somewhere between logic 1 and 0. This state would then propagate

to all the input pins of gates in the fan-out cone of the inverter. Those gates, in turn would produce a non-digital output and would propagate further. Thus potentially large number of nodes in a chip would be in the indeterminate non-digital state.

Also, the steady state path created between VDD and ground through the inverter transistors leads to an excessive current draw from VDD. When many nodes are in this indeterminate state, a large amount of current is drawn from VDD leading to a burn-out or latch-up of the chip.

VHDL

The *W123* rule reports violations for signals or variables that are read but not set in the design. The rule evaluates only the static conditions.

For multidimensional variables, only the total count of offending bits that are read but not set, is reported in the violation message. To see the details of the violating bits, refer to the **`SignalUsageReport`** report in the report from the *spyglass_reports/lint* directory.

Rule Exceptions

For VHDL, following are the exceptions to the W123 rule:

- The rule does not report violations for variables read inside dead code (conditional or selected signal assignment).
- The rule does not consider assignments inside a conditional statement when the condition is false. In VHDL, only the static conditions are evaluated. In Verilog, in addition to static conditions, simple non-static conditions involving FOR-loop index are also evaluated.

NOTE: *The rule does not evaluate the complex IF-conditions involving more than one loop variable. The rule checks both the assignments inside the* **`if`** *and the* **`else`** *constructs.*

Language

Verilog, VHDL

Default Weight

10

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameter(s)

- *ignoreModuleInstance*: Default value is **no**. Set this parameter to **yes** to ignore the unset variables that are used only in the instance port mapping.
- *handle_large_bus*: Default value is **no** and this rule does not process large arrays. Set the parameter to **yes** to process large arrays (greater than 50,000) and report violation if not used correctly.
- *traverse_function*: Default value is **no.** Set this parameter to **yes** to consider the inputs as read when being read inside a function. This parameter is applicable to **VHDL** only.
- *checkfullrecord*: Default value is **no.** Set this parameter to **yes** to not report violation for a VHDL record if at least one element of the record is read and set.
- *ignore_hier_scope_var*: Default value is **no** and the *W123* rule reports all user-defined data types. Set the value of this parameter to **yes** to not report any violation for the user-defined data types. This parameter is applicable for Verilog only.
- *dump_array_bits*: Default value is **no.** Set this parameter to **yes**, to enable the W123 rule to include multi-dimensional signals/bits information in the violation message. This parameter is applicable to VHDL only.
- *report_struct_name_only*: By default, the value of the parameter is no. Set the value of the parameter to yes to enable the W123 rule to report one violation for a struct variable.

Constraint(s)

None

Messages and Suggested Fix

Verilog

Message 1

The following message is displayed when the <*var-name*> variable that is never set in the design, is read for the first time:

```
[WARNING] Variable '<var-name>'<no_of_bits> read but never set
```

```
[Hierarchy: '<hier-path>']
```

Where,

- *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.
- *<no_of_bits>* is the number of offending bits of multidimensional array, which are read but not set.

Potential Issues

A violation is reported when a variable is read but never set.

Consequences of Not Fixing

It is a poor design practice to set values across hierarchy boundaries by using hierarchical names. Reading uninitialized value may result in undriven nets post synthesis.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the line in which the variable is read. To check whether the variable is set, search the variable name in the source file and check all the usages of the variable in the corresponding module or architecture. To view the list of signal bits that are read but not set, open the *SignalUsageReport* section from the **`Tools -> Report`** menu or double-click the violation message referring to the report in Atrenta Console

To resolve the violation, review the RTL to determine why the variable is not set. Also, check for the typing error in the variable name.

Message 2

The following message is displayed when the signal size is greater than 50,000 and the *handle_large_bus* parameter is disabled:

```
[INFO] Signal '<signal-name>' size too big thus not processed,
use 'set_parameter handle_large_bus yes' for enabling handling
of these signals
```

Potential Issues

If the **`handle_large_bus`** parameter is not enabled, the violation for signals of size greater than 50,000 is missed.

Consequences of Not Fixing

None

How to Debug and Fix

None

Message 3

The following message is displayed when the *report_struct_name_only* parameter is set to yes:

```
[INFO] Variable '<variable_name> (total bits
<violation_bit_count>)' read but never set.[Hierarchy:
'<hierarchy>']"
```

Potential Issues

The violation message is reported when the *report_struct_name_only* parameter is set to yes

Consequences of Not Fixing

None

How to Debug and Fix

None

VHDL

Message 1

The following message is displayed for the *<sig-name>* signal that is read but not set in the design:

```
[WARNING] The signal '<sig-name>' <bits> is read but not set
[Hierarchy: ‘<hier-path>’]
```

Potential Issues

A violation is reported when a signal is read but never set.

Consequences of Not Fixing

Reading uninitialized value may result in undriven nets post synthesis.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the line in which the signal is read. To check whether the signal is set, search the signal name in the source file and check all the usages of the signal in the corresponding module or architecture. To view the list of signal bits that are read but not set, open the *SignalUsageReport* section from the **`Tools -> Report`** menu or double-click the violation message

referring to the report in Atrenta Console.

To resolve the violation, review the RTL to determine why the signal is not set. Also, check for the typing error in the signal name.

Message 2

The following message is displayed at the location of the declaration of a variable <*var-name*> that is read but not set in the design:

```
[WARNING] The variable '<var-name>' <bits> is read but not set
[Hierarchy: '<hier-path>']
```

Where, <*bits*> are the vector bits, which are read but not set.

Potential Issues

A violation is reported when a variable is read but never set.

Consequences of Not Fixing

Reading uninitialized value may result in undriven nets post synthesis.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the line in which the variable is read. To check whether the variable is set, search the variable name in the source file and check all the usages of the variable in the corresponding module or architecture. To view the list of variable bits that are read but not set, open the *SignalUsageReport* section from the **Tools -> Report** menu or double-click the violation message referring to the report in Atrenta Console.

To resolve the violation, review the RTL to determine why the variable is not set. Also, check for the typing error in the variable name.

Message 3

The following message is generated, when the *W123* rule reports a violation for vector or multidimensional signal:

[INFO] Please refer to 'SignalUsageReport.rpt' for details of violating bits

Potential Issues

A violation is reported when a vector or a multi-dimensional signal is read but not set in the design.

Consequences of Not Fixing

Reading an uninitialized value may result in undriven nets post synthesis.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the line in which the signal is read. To check whether the signal is set, search the signal name in the source file and check all the usages of the signal in the corresponding module or architecture. To view the list of signal bits that are read but not set, open the *SignalUsageReport* section from the **Tools -> Report** menu or double-click the violation message referring to the report in Atrenta Console.

To resolve the violation, review the RTL to determine why the signal is not set. Also, check for the typing error in the signal name.

Example Code and/or Schematic

Example 1

In the following example code, SpyGlass reports a violation for the *net[2:0]* variable, which is read but never set:

```
module test(in1, clk, out1);
  input [2:0] in1;
  input clk;
  output [2:0] out1;
  reg [2:0] out1;

  wire [2:0] net;

  always@(posedge clk)
    out1 <= in1 && net;

endmodule
```

Example 2

Consider the following example:

```
always @(posedge clk) begin
  for (k = 0; k < 3; k = k + 1)
  begin
   if (k > 3)
    sig2[k] <= sig1[k];
   else
```

```
    sig2[k] <= 1'b0;
  end
end
```

The W123 rule does not report a violation for the above example as a signal is assigned inside the **IF** conditional statement, which is false.

Example 3

Consider the following **VHDL** example:

```
architecture rtl of test is
signal match : std_logic;
signal input1 : std_logic;

function fnout(
  input2   : in std_logic;
  match1 : in std_logic
  )
return std_logic is
variable match2 : std_logic;
begin
  match2 := match1; -- match used inside the function
return match2;
end fnout;

begin

y <= fnout(input1,match);

end architecture;
```

In the above example, as the *traverse_function* parameter is set to **yes,** the **input1** signal is not being read inside the function block, so it is not considered as read while the **match** signal is being read inside the function block, so it is considered as read.

Default Severity Label

Warning, Info

Rule Group

Usage

Reports and Related Files

SignalUsageReport

W143

Macro has been redefined

Language

Verilog

Rule Description

The W143 rule flags macros redefinitions.

Redefining a macro in the same file in which it was originally defined, is likely to cause confusion and limit readability.

The W143 rule also flags a violation if a macro is defined in a file, and then the same file is included in a separate file where the macro is redefined. Consider an example in which a macro, **M1**, is defined in a file, **F1**, and **F1** is included in a separate file, **F2**. If you redefine the macro, **M1**, in **F2**, this rule will report violation. These apparent errors can be ignored.

Parameters

waiver_compat: Default value is no. If you set the value of this parameter to **yes** or **<rule-name>**, it ensures that the rule does not generate the line number information in the first run itself. Thus waivers work correctly even if the line numbers of the RTL gets changed in the subsequent runs.

Message Details

The following message appears, at the location where a macro *<macro-name>* is redefined, providing details of the line number *<line-num>* and the file name *<file-name>* in which the macro was previously defined:

```
Redefining macro '<macro-name>'. Previously defined at line
'<line-num>' in file '<file-name>'
```

Rule Severity

Warning

Suggested Fix

If the macro is redefined with a different value, rewrite the code to use a different macro in the second case. Redefining the same macro with a different value can make the code error-prone.

W154

Do not declare nets implicitly

Language

Verilog

Rule Description

The W154 rule flags implicit net declarations.

While implicit net declarations may be convenient for handling unconnected (don't care) outputs, they can also result through misspelling of net names which are intended to be connected.

Running the W154 rule ensures that all implicit declarations are flagged. If you require all nets to be declared, the W154 rule effectively becomes a spell-check for net connections; any message found represents a true error.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameter(s)

- *report_port_net*: Default value is no. Set the value of the parameter to yes to report nets/ports declared without a type.
- *group_by_module*: Default value is no. Set the value of the parameter to **`yes`** to enable the rule to group the violations based on the module and report them in turbo mode.

Message Details

The following message appears at the location where a net *<net-name>* is used that is not explicitly declared:

```
Declare net '<net-name>' explicitly
```

Rule Severity

Guideline

Suggested Fix

Explicitly declare all nets found by this rule (after checking they are not a result of spelling errors).

W175

A parameter / generic has been defined but is not used

Language

Verilog, VHDL

Rule Description

The *W175* rule reports parameters and generics that are never used in the design.

While such constructs are allowed, the rule helps you clean up your design.

NOTE: *For Verilog designs, the rule also reports a violation for unused local parameters.*

By default, the *W175* rule does not report violation for unused global parameter. Set the value of the *report_global_param* parameter to yes to report violation for unused global parameter.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

Verilog

The following message appears at the location where a parameter *<param-name>* is declared that is never used in the design:

```
Parameter '<param-name>' declared but not used
```

VHDL

The following message appears at the location where a generic *<gen-name>* is declared that is never used in the design:

```
The generic '<gen-name>' is not used
```

Rule Severity

Warning

Suggested Fix

Confirm that parameter/generic is really redundant. Remove it if possible to reduce clutter in the design and to reduce warning messages from SpyGlass.

W188

Do not write to input ports

Language

Verilog

Rule Description

The W188 rule flags assignments to input ports.

Writing to an input port creates a wire-or inside the module. Also it creates a possibility for the port to behave as an inout port, which may drive the external logic to a conflict, where a user of the module assumed the port to always behave as an input port.

Message Details

The following message appears at the location where an input port *<port-name>* is assigned a value:

```
input port '<port-name>' should not be assigned any value
```

The following message appears at the location where an input port *<port-name>* is instantiated:

```
input port '<port-name>' should not be connected to an instance output port
```

Rule Severity

Warning

Suggested Fix

Buffer or gate the internal feedback to ensure it cannot drive back out through the input port.

W215

Reports inappropriate bit-selects of integer or time variables

When to Use

Use this rule to identify inappropriate bit-selects of **`integer`** or **`time`** variables.

Description

The *W215* rule reports bit-selects of integer or time variables.

NOTE: *The W215 rule supports generate-if, generate-for, and generate-case blocks.*

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a bit-select of a variable <*var-name*> of type <*var-type*> is used:

```
[WARNING] Inappropriate bit select for <var-type> variable: "<var-name>"
```

Where <*var-type*> can be **`integer`** or **`time`**.

Potential Issues

A violation is reported when a bit-select is used for an integer or time variable.

Consequences of Not Fixing

Bit-selects are not meaningful for integer variables and time variables.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where a bit-select is used for a variable whose data type is integer or time. You can confirm the data type of the variable by scrolling up the HDL to the most recent definition of the signal.

To fix the violation, remove the bit-select. If you want to select a bit from an integer variable, either mask and shift, or assign the integer to a **`reg`** and then do bit-select on the **`reg`**.

Example Code and/or Schematic

Consider the following example:

```
module W215_mod2(in1, sel, out1);
  input in1, sel;
  output out1;
  reg out1;

  integer i;
  initial
    i = 10;

  always@(in1 or sel)
    if(sel == 1'b1)
      out1 <= in1;
    else
      out1 <= i[0];

endmodule
```

In the above example, the *W215* rule reports inappropriate bit-selects for the **`int`** variable.

Default Severity Label

Warning

Rule Group

Usage

Reports and Related Files

None

W216

Reports inappropriate range select for integer or time variable

When to Use

Use this rule to identify inappropriate range selects for integer or time variables.

NOTE: *The W216 rule supports generate-if, generate-for, and generate-case blocks.*

Description

The *W216* rule reports range selects of integer or time variables.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where range select of a variable <*var-name*> of a type <*var-type*> is used.

```
[WARNING] Inappropriate range select for <var-type> variable: "<var-name>"
```

Where <*var-type*> can be **integer** or **time**.

Potential Issues

A violation is reported when a range select is used for an integer or time variable.

Consequences of Not Fixing

Range selects are not meaningful for integer variables and time variables.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where integer or time variable is used with selected range.

Remove the range select. If you want to select a range from an integer variable, either mask and shift, or assign the integer to a **`reg`** and then do range select on the **`reg`**.

Example Code and/or Schematic

Consider the following example:

```
module W216_mod2(in1, sel, out1);
  input [1:0] in1;
  input sel;
  output [1:0] out1;
  reg [1:0] out1;

  integer i;

  initial
    i = 10;

  always@(in1 or sel)
    if(sel == 1'b1)
      out1 = in1;
    else
      out1 = i[1:0];

endmodule
```

In the above example, the *W216* rule reports inappropriate range select for the *`int_part_sel`* variable.

Default Severity Label

Warning

Rule Group

Usage

Reports and Related Files

None

W240

An input has been declared but is not read

Language

Verilog, VHDL

Rule Description

The *W240* rule reports input ports that are never read in the module.

While such ports are allowed, the rule helps you clean up your design.

NOTE: *For Verilog, inside the nested* **`for`** *loops, signals of user-defined type like structures, interfaces, etc. are considered fully* **`set`** *if used on the left-hand side of an expression and fully* **`read`** *if used on the right-hand side of an expression.*

By default, the rule reports a violation if any bit of the input port is unread. If you set the *checkfullbus* parameter to **`yes`**, the rule reports a violation only when the entire input port is unread.

By default, this rule does not process large arrays. Set the *handle_large_bus* parameter to **`yes`** to process large arrays (greater than 50,000) and report violation if not used correctly.

By default, the value of *checkfullrecord* is set to **`no`**. Set this parameter to **`yes`** to not report violation for a VHDL record if at least one element of the record is read.

By default, the value of *checkfullstruct* is set to **`no`**. Set this parameter to **`yes`** to not report a violation for structs if at least one element of the struct is read.

To see the bits of the input that are not read, refer to the *SignalUsageReport* report in the *spyglass_reports/lint* directory.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

Verilog

Message 1

The following message appears at the location where an input port *<port-name>* is declared but never read in the module:

```
Input '<port-name>' declared but not read. [Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Message 2

The following message is displayed when the signal size is greater than 50,000 and the *handle_large_bus* parameter is disabled:

```
Signal '<signal-name>' size too big thus not processed, use 'set_parameter handle_large_bus yes' for enabling handling of these signals
```

If the **handle_large_bus** parameter is not enabled, the violation for signals of size greater than 50,000 is missed.

VHDL

The following message appears at the location where an input port *<port-name>* is declared that is never read in the process. For multi-bit ports the rule also displays the offending bits in the message.

```
Not all the elements of in port '<port-name>' '(<offending-bits>)' are read [Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical path of the containing process.

Rule Severity

Warning

Suggested Fix

In most cases, you should remove the input. This will force a change in design units instantiating this module. In some special cases, for example scan-enable, an unconnected input appears as a place holder for later logic

insertion. These cases should be waived.

Examples

VHDL

Consider the following example:

```
entity test is
  port(a: in bit_vector(0 to 5) ;
       b: in bit;
       c: in bit;
       z: out bit
       );
end test;

architecture behav of test is
  begin
    process(a, b)
      begin
        z <= a(3) or b;
    end process;
end behav;
```

For this example, SpyGlass generates the following messages:

```
Not all the elements of in port 'a' (Bits: 0:2 4:5) are read
[Hierarchy: ‘test(behav):’]
```

```
Not all the elements of in port 'c' are read [Hierarchy:
‘test(behav):’]
```

The first message indicates that the bits of vector **a** at the 0, 1, 2, 4, and 5 position numbers of the port were not read and the second message indicates that none of the bits of vector **c** were read.

Verilog

Consider the following example:

```
module ffd (z,a,b,c);
    input b,c;
    output reg z;
    input [5:0] a;
```

```
  always @(a,b)
    z<=a[3] | b;
endmodule
```

In the above example, the *W240* rule reports a violation because the input *C* is never read in the module. The following message is reported by this example:

```
Input 'c' declared but not read.[Hierarchy: ':ffd']
```

W241

Output is never set

Language

Verilog, VHDL

Rule Description

The W241 rule flags output ports that are not set (completely set for VHDL design) in the module.

While such ports (for example a Qbar output from a flip-flop) are allowed, un-set outputs can confuse other users.

NOTE: *For Verilog, inside the nested* ***for*** *loops, signals of user-defined type like structures, interfaces, etc. are considered fully* ***set*** *if used on the left-hand side of an expression and fully* ***read*** *if used on the right-hand side of an expression.*

By default, this rule does not process large arrays. Set the *handle_large_bus* parameter to **yes** to process large arrays (greater than 50,000) and report violation if not used correctly.

To see the details of the output that is not set, refer to the *SignalUsageReport* report in the *spyglass_reports/lint* directory.

By default, all rule related data is reported in the *SignalUsageReport* report. Set the value of the *disable_signal_usage_report* parameter to rule-name to disable data generation in the *SignalUsageReport* report for the specified rule.

Message Details

Verilog

Message 1

The following message appears at the location where an output port *<port-name>* is declared that is not set in the module:

```
Output '<port-name>' is never set. [Hierarchy: ‘<hier-path>’]
```

The *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Message 2

The following message is displayed when the signal size is greater than 50,000 and the *handle_large_bus* parameter is disabled:

```
Signal '<signal-name>' size too big thus not processed, use
'set_parameter handle_large_bus yes' for enabling handling of
these signals
```

If the **handle_large_bus** parameter is not enabled, the violation for signals of size greater than 50,000 is missed.

VHDL

The following message appears at the location where an output port *<port-name>* is declared that is not completely set in the process:

```
Not all the elements of out port '<port-name>' '<offending-
bits>' are set [Hierarchy: ‘<hier-path>’]
```

Where, *<offending-bits>* refers to the offending bits for multi-bit ports.

The *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Rule Severity

Warning

Suggested Fix

In most cases, you should remove the output. This will force a change in design units that are instantiating this module. In some special cases, an undriven output appears as a place holder for later logic insertion. These cases should be waived.

Examples

Verilog

Consider the following example:

```
module test1(out1,out2,in1,in2 );
     input  [13:0] in1;
     input  [13:0] in2;
```

```
     output [15:0] out1;
     output  out2;

     assign out1[13:0]= in1 | in2;

endmodule
```

In the above example, the *W241* rule reports two violations because the output *out2* and *out1[15:14]* are never set in the module. The following messages are reported by this example:

```
Output 'out1[15:14]' is never set.[Hierarchy: ':test1']

Output 'out2' is never set.[Hierarchy: ':test1']
```

VHDL

Consider the following example:

```
entity test1 is
         port( in1: in bit_vector(13 downto 0);
         in2: in bit_vector(13 downto 0);
         out1: out bit_vector(15 downto 0);
         out2: out bit);
end test1;

architecture behav of test1 is
begin
         process(in1, in2)
         begin
                  out1(13 downto 0) <= in1 or in2;
         end process;
end behav;
```

In the above example, the *W241* rule reports two violations because the output *out2* and *out1* (bits: 15:14) are never set in the architecture behave of entity **test1** . The following messages are reported by this example:

```
Not all the elements of out port 'out1' (Bits: 15:14) are set.
[Hierarchy: ':test1(behav):']

Not all the elements of out port 'out2' are set. [Hierarchy:
':test1(behav):']
```

W333

Unused UDP

Language

Verilog

Rule Description

The W333 rule flags UDPs (user-defined primitives) that are never instantiated in the design.

While such descriptions are allowed, the W333 rule helps you clean up your design.

Message Details

The following message appears at the first line of a UDP declaration *<udp-name>* that is never instantiated in the design:

```
UDP '<udp-name>' has not been used in the design
```

Rule Severity

Warning

Suggested Fix

Check all such cases to make sure you expect the UDP not to be instantiated in the design. To exclude such cases from analysis, use the **`set_option top`** command.

W423

A port with a range is redeclared with a different range

Language

Verilog

Rule Description

The W423 rule flags ports that are re-declared with a different range in the same module.

While such description is not necessarily an error, it may not be the design intent.

Message Details

The following message appears at the location where a port *<port-name>* is re-declared with a different range:

```
Port '<port-name>'(Range <previous-range>) is redeclared with different range <latest-range>. [Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

This rule also highlights previous declaration with different range.

Rule Severity

Warning

Suggested Fix

In some cases this may be a neat way of compactly managing a calculation, but it can be confusing. Recommend in general you avoid this style. If necessary, declare and assign to a temporary bus of the appropriate width.

Examples

In the following example, the input port **in1** is being re-declared as a wire with a different range:

```
input [15:0] in1;
```

```
wire [20:0] in1;
```

While such description is not necessarily an error, it may not be the design intent. In the above example, the top five bits of wire **in1** will not be driven by the input port **in1**.

W468

Index variable is too short

Language

Verilog

Rule Description

The W468 rule flags variables used as array index that are narrower than the array width.

When a variable used to index a bit-select or array has a width narrower than the width of the bus or the size of the array, some elements of the bus or array will not be accessible.

Message Details

The following message appears at the location where the value of variable or signal *<name>* used to index an array or a bus signal bit-select *<vector-name>* that is narrower than the bit-width of the array or bus, is encountered:

```
Variable/Signal '<vector-name>' is indexed by '<name>' which can not index the full range of this vector. [Hierarchy: '<hier-path>']
```

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Rule Severity

Warning

Suggested Fix

If possible, use an index large enough to span the full range of the vector.

If you intend to access only a lower subset of bits, make this explicit by using a concat of the correct width with upper bits set to zero, for example.

W493

A variable is not declared in the local scope, that is, it assumes global scope

Language

Verilog, VHDL

Rule Description

The W493 rule flags variables that are read but not declared in the local scope.

Using variables in this manner causes the variable value to be read from a more global scope, if available or for the variable to be implicitly declared, and possibly not set.

This design practice is not recommended and may result in unexpected values.

NOTE: *The W493 rule supports generate-if and generate-for blocks.*

NOTE: *(Verilog) The W493 rule is switched off by default. You can enable this rule by specifying the* **`set_goal_option addrules W493`** *command.*

Message Details

Verilog

The following message appears at the location where a variable *<var-name>* is read that is not declared in the local scope:

```
Temporary variable '<var-name>' is not declared in local scope
```

VHDL

The following message appears at the location where a shared variable *<var-name>* is used:

```
Shared Variable '<var-name>' used. Avoid using shared variables
```

Rule Severity

Fatal (Verilog) / Warning (VHDL)

Suggested Fix

It is always best to avoid coding which depends on or creates side-effects. Pass values from other contexts explicitly as parameters/arguments.

W494

Inout port is not used

Language

Verilog, VHDL

Rule Description

The W494 rule flags inout ports that are never read or set in the design.

While such ports are allowed, the W494 rule helps you clean up your design.

NOTE: *For Verilog, inside the nested* ***for*** *loops, signals of user-defined type like structures, interfaces, etc. are considered fully* ***set*** *if used on the left-hand side of an expression and fully* ***read*** *if used on the right-hand side of an expression.*

By default, the rule reports violation for the completely unused inout ports. Set the value of the *strict* parameter to **yes** to also report violation for partially unused ports.

By default, this rule does not process large arrays. Set the *handle_large_bus* parameter to **yes** to process large arrays (greater than 50,000) and report violation if not used correctly.

Message Details

The following message appears at the location where an inout port <*port-name*> is declared but never read or set in the design:

Verilog

```
Unused inout '<port-name>'. [Hierarchy: ‘<hier-path>’]
```

Where, <*hier-path*> is the complete hierarchical name of the containing scope.

VHDL

```
The inout port '<port-name>' is not used
```

Rule Severity

Warning

Suggested Fix

Check to make sure you intended to ignore the output value.

W494a

Input port is not used

Language

VHDL

Rule Description

The W494a rule flags input ports that are never read in the design.

While such ports are allowed, the W494a rule helps you clean up your design.

By default, the rule reports violation for the completely unused input ports. Set the value of the *strict* parameter to yes to also report violation for partially unused ports.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location where an input port *<port-name>* is declared that is never set in the design:

```
The in port '<port-name>' is not used
```

Severity

Warning

Suggested Fix

Determine if the input port is really not required. If not, consider removing the port.

W494b

Output port is not used

Language

VHDL

Rule Description

The W494b rule flags output ports that are never set in the design.

While such ports are allowed, the W494b rule helps you clean up your design.

By default, the rule reports violation for the completely unused output ports. Set the value of the *strict* parameter to **yes** to also report violation for partially unused ports.

Message Details

The following message appears at the location where an output port *<port-name>* is declared that is never set in the design:

```
The out port '<port-name>' is not used
```

Severity

Warning

Suggested Fix

Determine if the output port is really not required. If not, consider removing the port.

W495

Inout port is never set

Language

Verilog, VHDL

Rule Description

The W495 rule flags inout ports that are read in the design but never set.

While such ports are allowed, the W495 rule helps you clean up your design so that you can redefine such inout ports as input ports.

It is an error in CMOS to let an input float, unless that input has a **pull-up** or **pull-down**. Since an inout has an input mode, this condition is an error.

NOTE: *For Verilog, inside the nested* ***for*** *loops, signals of user-defined type like structures, interfaces, etc. are considered fully* ***set*** *if used on the left-hand side of an expression and fully* ***read*** *if used on the right-hand side of an expression.*

By default, this rule does not process large arrays. Set the *handle_large_bus* parameter to **yes** to process large arrays (greater than 50,000) and report violation if not used correctly.

Message Details

The following message appears at the location where an inout port <*port-name*> is declared that is read in the design but is never set:

Verilog

`Undriven inout '<port-name>'. [Hierarchy: ‘<hier-path>’]`

Where, <*hier-path*> is the complete hierarchical name of the containing scope excluding subprograms.

VHDL

`The inout port '<port-name>' is not set`

Rule Severity

Warning

Suggested Fix

If you don’t care about the input value, tie it high or low, but do not let it float.

W497

Not all bits of a bus are set

Language

Verilog

Rule Description

The W497 rule flags multi-bit signals that are not completely set in the design.

The W497 rule does not check multi-bit input and inout ports.

When some bits of a multi-bit signal are not set in the design, these bits may achieve unknown values and may infer latches or flip-flops during synthesis.

For all multi-bit signals (single-dimension vectors and multidimensional arrays), the total count of unset bits is reported in the violation message. To see the details of the violating bits, please refer to the *SignalUsageReport*. You can access the report from the *spyglass_reports/lint* directory.

Parameter(s)

- *modport_compat*: Default value is **no**. Set this parameter to **yes** to not evaluate mod ports that are used as hier scope variables in a port connection.

Message Details

The following message appears at the location of a multi-bit signal declaration *<sig-name>* when all bits of the signal are not set in the design:

```
Not all bits of bus '<sig-name>'<no-of_bits> are set
[Hierarchy: ‘<hier-path>’]
```

Where,

- *<hier-path>* is the complete hierarchical path of the signal.
- *<no_of_bits>* is the number of unset bits of vector or multidimensional signals.

Also, the following message is generated, when the W497 rule flags a

violation for vector or multidimensional signal:

Please refer to 'SignalUsageReport.rpt' for details of violating bits

In the Console, click on the above violation message to view the *SignalUsageReport*.

Rule Severity

Warning

Suggested Fix

If the bits are not required, use a narrower bus, or tie the unused bits to a fixed value.

W498

Not all bits of a bus are read

Language

Verilog

Rule Description

The W498 rule flags multi-bit signals that are not completely read in the design.

When some bits of a multi-bit signal are not read in the design, it may not be an error but should be examined closely as you would always expect all bits to be read.

For all multibit signals (single-dimension vectors and multidimensional arrays), the total count of unread bits is reported in the violation message. To see the details of the violating bits, please refer to the *SignalUsageReport*. You can access the report from the *spyglass_reports/lint* directory.

Parameter(s)

- *modport_compat*: Default value is **no**. Set this parameter to **yes** to not evaluate mod ports that are used as hier scope variables in a port connection.
- *avoid_port_signal*: Default value is **no**. Set this parameter to **yes** to enable the rule to not report a violation for port signals.

Message Details

The following message appears at the location of a multi-bit signal declaration *<sig-name>* when all bits of the signal are not read in the design:

```
Not all bits of bus '<sig-name>'<no_of_bits> are read
[Hierarchy: ‘<hier-path>’]
```

Where,

- *<hier-path>* is the complete hierarchical path of the signal.
- *<no_of_bits>* is the number of unread bits of vector or multidimensional signals.

Also, the following message is generated, when the W498 rule flags a violation for vector or multidimensional signal:

Please refer to 'SignalUsageReport.rpt' for details of violating bits

In the Console, click on the above violation message to view the *SignalUsageReport*.

Rule Severity

Warning

Suggested Fix

No fix necessarily required, but such cases should be examined for possible errors.

W528

A signal or variable is set but never read

Language

Verilog, VHDL

Rule Description

The *W528* rule reports variables that are set in the design but are never read. In addition, the rule reports vector variables if all bits are not read in a scope of a process.

However, in case of indirect addressing with parameters (Verilog designs), the vector variable is assumed to be completely read if the complete range of the vector variable can be accessed. See *Examples* below for details.

While setting unused variables is allowed, the rule helps you clean up your design of unused variables, thereby reducing clutter in the design description or possibly preventing a potential name confusion error.

NOTE: *For Verilog, inside the nested* ***for*** *loops, signals of user-defined type like structures, interfaces, etc. are considered fully* ***set*** *if used on the left-hand side of an expression and fully* ***read*** *if used on the right-hand side of an expression.*

To avoid violations for certain signals by this rule, provide a comma-separated list of those signals in the *not_used_signal* parameter. You can specify full signal names or regular expressions in the parameter. For example, if you specify the list, "**data_signal,bus_***", in the parameter, the rule does not report a violation for the **data_signal** signal and all the signals with names starting from **bus_**.

For multidimensional variables, only the total count of offending bits (set but not read) is reported in the violation message. To see the details of the violating bits, please refer to the *SignalUsageReport*. You can access the report from the *spyglass_reports/lint* directory.

By default, the rule reports a violation if any bit of a variable is set but not read in the current scope of a module or an architecture. If you set the *checkfullbus* parameter to **yes**, the rule reports a violation only when all bits of a variable are completely unread in the current scope of a module or an architecture.

By default, this rule does not process large arrays. Set the *handle_large_bus*

parameter to **yes** to process large arrays (greater than 50,000) and report violation if not used correctly.

By default, value *checkfullrecord* is set to **no**. Set this parameter to **yes** to not report violation for a record if at least one element of the record is set and read.

By default, all rule related data is reported in the *SignalUsageReport* report. Set the value of the *disable_signal_usage_report* parameter to rule-name to disable data generation in the *SignalUsageReport* report for the specified rule.

By default, the value of *checkfullstruct* is set to **no**. Set this parameter to **yes** to not report a violation for structs if at least one element of the struct is read.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

Verilog

Message 1

The following message appears at the location where a variable *<var-name>* is set but is never read in the design:

```
Variable '<var-name>'<no_of_bits> set but not read. [Hierarchy: '<hier-path>']
```

Where,

- *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.
- *<no_of_bits>* is the number of bits of multidimensional array, which are set but not read.

Message 2

The following message is displayed when the signal size is greater than 50,000 and the *handle_large_bus* parameter is disabled:

```
Signal '<signal-name>' size too big thus not processed, use
```

'set_parameter handle_large_bus yes' for enabling handling of these signals

If the **handle_large_bus** parameter is not enabled, the violation for signals of size greater than 50,000 is missed.

VHDL

The following message appears at the location of the declaration of a signal *<sig-name>* that is set but not read in the design:

The signal '<sig-name>' <bits> is set but not read

The following message appears at the location of the declaration of a variable *<var-name>* that is set but not read in the design:

The variable '<var-name>' <bits> is set but not read

Where, *<bits>* are the vector bits that are set but not read. For multidimensional array, *<bits>* is the total number of bits that are set but not read.

In addition, the following message is generated, when the *W528* rule flags a violation for vector or multidimensional signal:

Please refer to 'SignalUsageReport.rpt' for details of violating bits

In the Console, click the above violation message to view the *SignalUsageReport*.

Rule Severity

Warning

Suggested Fix

Check your logic. If the declaration and set are redundant, remove both to reduce clutter in rule-check reports.

Examples (Verilog)

Example 1

Consider the following example where the signal **bigbus** is completely set in the first **assign** statement:

```
module test1 (a, b);
```

```
  input [13:0] a;
  output [1:0] b;
  wire [255:0] bigbus;

assign bigbus =
  {{16{4'h0}},{16{4'h0}},{16{4'h0}},{16{4'h0}}};
assign b={bigbus[a[13:7]],bigbus[a[6:0]]};
endmodule
```

However, the second **assign** statement that assigns to signal **b** using indirect addressing, only 128 bits (2**(13-7+1) or 2**(6-0+1)) of signal bigbus can be addressed. Therefore, the *W528* rule reports a violation message.

Now, consider the following example:

```
module test2 (a,b);
  input [15:0] a;
  output [1:0] b;
  wire [255:0] bigbus;

assign bigbus =
  {{16{4'h0}}, {16{4'h0}}, {16{4'h0}}, {16{4'h0}}};
assign b={bigbus[a[15:8]], bigbus[a[7:0]]};
endmodule
```

The signal **bigbus** is completely set in the first **assign** statement. In addition, the second **assign** statement that assigns to signal **b** using indirect addressing, all 256 bits (2**(15-8+1) or 2**(7-0+1)) of signal bigbus can be addressed. Therefore, the *W528* rule does not report a violation message.

Example 2

Consider the following example:

```
module mod(in1, clk, out1);
  input [1:0] in1;
  input clk;
  output [1:0] out1;
  reg [1:0] out1;
  reg [1:0] set;

  assign set = 2'b10;

  always @(posedge clk)
    out1[0] = in1[0] & set[0];

endmodule
```

In this example, the *W528* rule, by default, reports a violation for the **set** signal. If you set the *checkfullbus* parameter to **yes** or *not_used_signal* parameter to **set**, the rule does not report any violation.

Examples (VHDL)

Example 1

Consider the following example:

```
library ieee;
use ieee.std_logic_1164.all;

entity test is
  port (
    a :       in  bit;
    b :       in  bit;
    y :       out bit
  );
end test;

architecture test_arc of test  is
   signal sig : std_logic_vector (7 downto 0);
```

```
   signal sig1 : std_logic_vector (7 downto 0);
begin
   sig <= ("11001010");
    y <= a and b;
    sig1(1) <= sig(0);
end test_arc;
```

In this example, the *W528* rule, by default, reports a violation for **sig** and **sig1** signals. If you set the *checkfullbus* parameter to **yes**, the rule does not report a violation for the **sig** signal.

Example 2

Consider the following example:

```
library ieee;
use ieee.std_logic_1164.all;

entity top is

end entity;

architecture behav of top is
type HADDR_RANGE_ARRAY is array (natural range <>) of
std_logic_vector (7 downto 0);
SIGNAL master_haddr : HADDR_RANGE_ARRAY (7 downto 0);
SIGNAL master_haddr1 : HADDR_RANGE_ARRAY (7 downto 0);
SIGNAL master_out : HADDR_RANGE_ARRAY (7 downto 0);
begin
      master_haddr <=
("11001010","11001010","11001010","11001010","11001010","110
01010","11001010","11001010");
blk: process
begin
test: for i in 0 to 5 loop
         master_out (i) <=master_haddr (i) and
master_haddr1(i);
end loop test;
end process blk;
end behav;
```

In this example, the *W528* rule, by default, reports a violation for **master_addr** and **master_out** signals. If you set the *not_used_signal* parameter to **master_out**, the rule does not report a violation for the **master_out** signal. If you set the *not_used_signal* parameter to **master_***, the rule does not report any violation for signals with names starting from **master_**.

W529

`ifdef is not supported by all tools

Language

Verilog

Rule Description

The W529 rule flags **'ifdef** compiler directives used in the design.

Some synthesis tools do not support preprocessor conditional directives.

Message Details

The following message appears at the location where a **'ifdef** compiler directive is encountered:

```
Compiler directive 'ifdef is not supported by all synthesis tools
```

Rule Severity

Warning

Suggested Fix

Avoid using these directives in synthesizable logic.

W557

Range value and part-selects of parameters should be avoided.

The W557 rule runs the *W557a* and *W557b* rules.

W557a

This rule has been deprecated

The W557a rule has been deprecated. Range values for parameter are synthesizable by almost all synthesis tools.

W557b

This rule has been deprecated

The W557b rule has been deprecated. Part-Select on parameter value are synthesizable by almost all synthesis tools.

W558

This rule has been deprecated

The W558 rule has been deprecated. Bit-Select on parameter value are synthesizable by almost all synthesis tools.

Lint_Tristate Rules

The SpyGlass lint product provides the following miscellaneous rules:

Rule	Flags...
W438	Tristate descriptions that are not at the top-level of the design
W541	Inferred tristate nets

W438

Ensure that a tristate is not used below top-level of design

When to Use

Use this rule to identify the tristate descriptions that are used below top-level of the design.

Description

The W438 rule flags tristate descriptions that are not at the top-level of the design. Tristate busing should be restricted to the top-level of the design.

In most of the current SoC designs, tristate signals are not widely used. If tristate signals are used, their usage is usually for muxing of the top-level buses or for some specialized fabric. Such fabrics are usually implemented at the top-level of SoC, and most of the designers prefer to manage the hierarchy accordingly for bus-holder cells, which in some technologies require a weak resistor. Bus holder cells ensure that the tristate bus does not float for a long time.

Rule Exceptions

You can enable the W438 rule by specifying the **`set_goal_option addrules W438`** command. However, this rule will not run if you set the *fast* rule parameter to **`yes`** and SpyGlass **`lint`** product is run.

Language

Verilog

Default Weight

5

Parameter(s)

fast: The default value is no. Set the value of the parameter to yes to suppress synthesis of the source RTL description.

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a tristate net *<sig-name>* is used at a level other than the top-level:

```
[WARNING] Tristate function '<sig-name>' used below top level
of design
```

Potential Issues

Violation may arise when a tristate function is used below top level of design.

Consequences of Not Fixing

For local sub-modules and lower level hierarchy blocks, use of tristates should be avoided for testability, timing, and reliability reasons.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the tristate value, **z**, is used in the design.

To fix the violation, use muxing at lower levels, rather than tristates.

Example Code and/or Schematic

Consider the following example where tristate descriptions are not at the top-level:

```
module test(a1, b1, x, y, ena, out1, clk);
  input [3:0] x, y;
  input clk;
  input [1:0] ena;
  output [3:0] a1, b1, out1;

  reg [3:0] out1;

  always @(posedge clk)
    begin
      out1 <=  x | y;
    end

  state1 ONE (a1, b1, x, y, ena);
endmodule
```

```
module state1(a, b, x, y, ena);
  input [1:0] ena;
  input [3:0] x, y;
  output [3:0] a, b;

  assign a = (ena == 2'b01) ? x : 4'bz;
  assign b = (ena == 2'b10) ? 4'bz : y;
endmodule
```

For this example, SpyGlass generates the following violation messages:

```
Tristate function 'a[3:0]' used below top level of design
```

Tristate function 'b[3:0]' used below top level of design

Default Severity Label

Warning

Rule Group

Lint_Tristate

Rule Severity

Warning

W541

A tristate is inferred

Language

Verilog

Rule Description

The W541 rule flags inferred tristate nets.

NOTE: *You can enable the W541 rule by specifying the* **`set_goal_option addrules W541`** *command. However, this rule will not run if you set the fast rule parameter to* **`yes`** *and SpyGlass* **`lint`** *product is run.*

Message Details

The following message appears at the location where a tristate signal *<sig-name>* is inferred:

```
A tristate '<sig-name>' is inferred
```

Rule Severity

Info

Suggested Fix

Use muxing wherever possible, rather than tristates.

Examples

A tristate net **`sig`** is inferred for the following assignment:

```
assign sig = enable ? in1 : 1'bz;
```

Assign Rules

The SpyGlass lint product provides the following assignment related rules:

Rule	Flags...
W19	Usage of constants where the constant is wider than the usage context
W164	Assignments in which LHS width does not match with the RHS width of an expression
W164c	Assignments in which the LHS width is greater than the (implied) width of the RHS expression
W257	AFTER clauses
W280	Intra-assignment delays specified with nonblocking assignments
W306	**integer** type to **real** type conversions
W307	Unsigned type (**reg** type) to real type conversions
W308	**real** type to **integer** type conversions
W309	Unsigned type (**reg** type) to **integer** type conversions
W310	**integer** type to unsigned type (**reg** type) conversions
W311	**real** type to unsigned type (**reg** type) conversions
W312	**real** type to single-bit type conversions
W314	Multi-bit **reg** types to single-bit conversions
W317	Assignments to supply nets
W336	Blocking assignment used in sequential **always** constructs
W397	Value assignments to input ports
W414	nonblocking assignments used in combinational **always** constructs
W446	Output ports that are read in the module where they are set
W474	Variables that are assigned but not deassigned
W475	Variables that are deassigned without being assigned
W476	Variables that are forced but are not released
W477	Variables that are released without being forced

Rule	Flags...
W484	Possible loss of carry or borrow bits during assignments using addition and subtraction arithmetic operators
W505	Signals or variables that are being assigned values using both blocking and nonblocking assignments

W19

Reports the truncation of extra bits

When to Use

Detects truncation of extra bits.

Rule Description

Reports a violation when you have specified a constant value wider than the specified width of the constant.

For example, constant **`4'b10110`** is truncated to **`4'b0110`** and a message is flagged.

Rule Checking for Binary Based Numbers

The W19 rule reports violation for the following cases:

- If significant bits are truncated.
- For loss of non-significant bits, if you set the value of the *strict* parameter to **`yes`**.

Rule Checking for Decimal Based Numbers

The W19 rule reports a violation if significant bits are truncated.

Rule Checking for Hexadecimal and Octal Based Numbers

The W19 rule reports violation for the following cases:

- If significant bits are truncated.
- If based word contains more bits than specified size and there is truncation of non-significant bits. However, the W19 rule does not report a violation if based word can fit well within the specified size, as this is a valid design practice.

Rule Exceptions

The W19 rule does not check for unused macro definitions and unused parameters.

Language

Verilog

Default Weight

5

Parameter(s)

- *strict*: The default value of the parameter is **no** and the rule W19 does not report truncation of zeroes for binary based numbers. If you set the the value of this parameter to **yes** or the rule name, the rule also reports truncation of zeroes for binary based numbers.
- *ignore_typedefs*: By default, the W443 rule considers all type definitions for rule checking. Set the value of the *ignore_typedefs* parameter to yes to ignore all type definitions for rule checking.

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a constant *<const-name>* is being truncated as it is wider than the usage context:

```
[WARNING] Constant <const-name> will be truncated
```

Potential Issues

A violation is reported when you have specified a constant value wider than the specified width of the constant.

Consequences of Not Fixing

When constant value is wider than the width of the constant, it results in truncation of extra bits. This in turn leads to data loss and unexpected code behavior.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where truncation of extra bit on constant number is violated. Search for the based number that is described in the violation message in the highlighted line.

To resolve the violation, determine the width specification and the constant value. Fix the width specification, if incorrect.

Example Code and/or Schematic

Example 1

Consider the following example:

```
module operator(clk1,out1);

  input clk1;
  output out1;
  reg out1;

  always @(posedge clk1) begin
    out1 = 1'b101; //(Constant 1'b101 will be truncated)
    if(out1 == 2'b0101)//Constant 2'b0101 will be truncated )
      begin
      end
  end

endmodule
```

In the above example, the *W19* rule generates the truncation message for the constants, **`1'b101`** and **`2'b0101`**, as the constants are wider than the usage context.

Example 2

Consider the following example:

```
2'b111
```

In the above example, the rule reports a violation.

Example 3

Consider the following example:

```
2'b011
```

In the above example, the rule reports a violation for this expression, if you have set the **`strict`** parameter to **`yes`**.

Example 4

Consider the following example:

```
2'd15, 1'd3
```

In the above example, the W19 rule reports a violation.

Example 5

Consider the following example:

```
2'd03, 4'd0014
```

In the above example, the rule does not report a violation for this expression.

Example 6

Consider the following example:

```
3'hF, 2'o7
```

In the above example, the rule reports a violation.

Example 7

Consider the following example:

```
3'h01, 3'o01
```

In the above example, the rule reports a violation.

Example 8

Consider the following example:

```
3'h1, 2'o1
```

In the above example, the rule does not report a violation for this expression.

Default Severity Label

Warning

Rule Group

Assign

Reports and Related Files

None

W164

Language

Verilog, VHDL

Rule Description

The W164 rule runs the *W164a* and *W164b* rules.

W164c

LHS width is greater than RHS width of assignment (Extension)

Language

Verilog

Rule Description

The *W164c* rule flags assignments in which the LHS width is greater than the (implied) width of the RHS expression.

The *W164c* rule is same as the *W164b* rule except the following two conditions:

- The W164c rule checks for all assignments (including those assignments containing no wire or reg objects or static expressions) and the W164b rule checks all assignments only when the *strict* rule parameter is set and *check_static_value* is set.
- The *W164c* rule, unlike *W164b* rule, does not report a violation, if the RHS is an unsigned unbased constant.

NOTE: *The W164c rule is switched off by default and can be run only when specially selected in Atrenta Console or is specified using the* **`set_goal_option addrules W164c`** *command in the batch mode.*

By default, the *W164c* rule considers the natural width of integer constants, which is log2(N)+1. Set the value of the *use_lrm_width* rule parameter to **`yes`** to consider the LRM width of integer constants, which is 32 bits.

NOTE: *For new width related changes, refer to New Width Flow Application Note.*

When a signed signal is divided by another signed signal, the *W164c* rule calculates the width as follows:

```
<width of numerator> + 1
```

Consider the following example:

```
wire signed   [2:0] a_3_sign, b_3_sign, c_sign_div;
assign c_sign_div = a_3_sign / b_3_sign;
```

The *W164c* rule reports a violation for the above assignment when the RHS width is 4. This calculation is valid only for simple division expressions.

By default, the *W164c* rule reports violation for disabled code in loops and

conditional (if condition, ternary operator) statements. Set the value of the *disable_rtl_deadcode* parameter to yes to disable violations for disabled code in loops and conditional (if condition, ternary operator) statements.

By default, the width is calculated considering the best fit width of an expression. That is the width in which maximum value of an expression can be accommodated.

When you set the value of the *nocheckoverflow* parameter to **yes** then width is calculated according to the LRM and the natural width is considered for constants.

The behavior of the rule is explained by the cases below.

- For Constant Expressions
  - For constant integer expressions, the width is calculated based on the value of expression:

    ```
    out[5:0] = 2 + 15 ;     //RHS Width = 5 (Value = 17),
                              violation
    out[10:0] = 100 << 4 ; //RHS Width = 11, NoViolation
    out[6:0] = 100 >> 2 ;  //RHS Width = 5
    ```

  - If constant is not a part of sub-expression then the specified width is considered:

    ```
    out[11:0] = 12'b0 ;     //RHS Width = 12, No violation
    ```

  - If the width of a based number is not specified then the natural width is considered:

    ```
    out[11:0] = 'h344;      //RHS Width = 10, violation
    ```

    For above cases, behavior of rule remains same when the *nocheckoverflow* parameter is set to **yes**.

- For Arithmetic Operators
  - For addition operator, the width is calculated as the width in which maximum value of expression is accommodated. Consider the following example:

    ```
    out[3:0] = in1[1:0] + in2[1:0] + in3[1:0]; //RHS Width
    4, No violation
    Max value RHS 3 + 3 + 3 = 9
    RHS Width = 4
    ```

```
out[2:0] = in1[1:0] + (3/3) ;   //RHS Width 3, No
violation
Max Value RHS = 3 + 1 = 4
RHS Width = 3

out[6:0] = in1[5:0] + 5'b1010;  //RHS Width 7, No
violation
out[4:0] = 15 + 3'b111;         //RHS width 5, No
violation
```

- For the subtraction operator, the width is calculated in a similar way as it is done for addition:

```
out[4:0] = in1[3:0] - in1[1:0]; //RHS Width 5, No
violation
out[4:0] = in1[3:0] - 4'b1010; //RHS Width 5, No
violation
```

- For the multiplication operator, the width is calculated as follows:

  When both operands are variable, then the RHS width is the sum of the width of both operands:

```
out[5:0] = in1[1:0] * in2[1:0] * in3[1:0]; //RHS Width
6, No violation
out[4:0] = in1[2:0] * in1[1:0];          //RHS Width 5,
No violation
```

  When one operand is static and other is variable, then the RHS width is calculated considering maximum value of the expression:

```
out[7:0] = in1[3:0] * 4'b1;    //RHS Width 4, Max
value = 15*1 = 15, Violation
out[7:0] = in1[3:0] * 4'b10;   //RHS Width 5, Max
value = 15*2 = 30, Violation
out[7:0] = in1[3:0] * 4'b1010; //RHS Width 8, Max
value = 15*10 = 150, No violation
out[6:0] = in1[3:0] * 4;       //RHS Width 6, Max
value = 15*4 = 60, violation
```

- For the division operator, the width of the RHS is assumed as the width of left operand:

```
out[2:0] = in1[2:0]/in1[1:0]; //RHS Width 3, No
```

```
violation
out[2:0] = in1[1:0]/in1[2:0]; //RHS Width 2, Violation
out[3:0] = in1[3:0]/4'b10;    //RHS Width 4, No
violation
out[3:0] = 4/in1[3:0];         //RHS Width 3,violation
```

- When the *nocheckoverflow* parameter is set to **yes**, the width is assumed to be the same as the width of term having maximum width. Consider the following example:

```
out[3:0] = in1[1:0] + in2[1:0] + in3[1:0]; //RHS Width
2, violation
out[6:0] = in1[5:0] + 5'b1010; //RHS Width 6, Violation
out[4:0] = 15 + 3'b111;        //RHS width 4, Violation
out[4:0] = in1[3:0] - in1[1:0];//RHS Width 4, Violation
out[4:0] = in1[3:0] - 4'b1010; //RHS Width 4, Violation
out[4:0] = in1[2:0] * in1[1:0];//RHS Width 3, Violation
out[7:0] = in1[3:0] * 4'b10;   //RHS Width 4, Violation
out[2:0] = in1[1:0]/in1[2:0];  //RHS Width 3, No
violation
out[3:0] = 4/in1[3:0];         //RHS Width 4, No
violation
```

- For Shift Operator
  - For the right shift, if left operand width is matching the LHS width then no violation is reported. Consider the following example:

```
out[3:0] = in1[3:0] >> in ;       //RHS Width 4, No
violation
out[5:0] = in1[4:0] >> 2 ;        //RHS Width 5,
Violation
out[31:0] = 4 >> in ;             //RHS Width 32, No
violation
out[4:0] = in1[4:0] >> in2[1:0] ; //RHS Width 5, No
violation
```

```
out[5:0] = in1[4:0] >> in2[1:0] ; //RHS Width 5,
violation
```

- For left shift, if shifted or left operand width is matching LHS width then no violation is reported. Consider the following example:

```
out[5:0] = 4'b0001 << in ;  //RHS Width 4, Violation
out[31:0] = 4 << in ;             //RHS Width 32, No
violation
out[5:0] = in1[4:0] << 1 ;        //RHS Width 6, No
violation
out[6:0] = in1[4:0] << 2 ;        //RHS Width 7, No
violation
out[4:0] = in1[4:0] << 2 ;        //RHS Width 5, No
violation
out[7:0] = in1[4:0] << in2[2:0] ; //RHS Width 5,
Violation
```

- When parameter *nocheckoverflow* is set to **yes** then width of left operand is considered for both left shift and right shift operations.

- For Self-Determined Expression
  - For self-determined expressions, the width is calculated as per the LRM:

```
wire a,b,c;

assign a = {b+c};    //LHS: 1, RHS: 1, No violation

assign out[2:0] = {1'b1,b+c};  //LHS: 3, RHS: 2,
Violation
```

  - Behavior of the rule remains the same when the *nocheckoverflow* parameter is set to **yes**

- For Conditional Operator

  The width of conditional operator is calculated as follows:

  - A violation is reported if there is a width mismatch either in left expression or in right expression. Consider the following example:

```
out[2:0] = in1[2] ? in2[0] : in3[2:0] ;  //RHS Width 1,
Violation

out[2:0] = in1[2] ? in2[2:0] : in3[1:0];//RHS Width 2,
```

```
Violation

out[0]   = in1[2] ? in2[0] : in3[2] ; //RHS Width 1, No
violation
```

- ❒ Behavior remains same when the *nocheckoverflow* parameter is set to **yes.**

- For Power Operator

  The width of power operator is calculated as follows:

  - ❒ If the RHS expression of the power operator is static, then the width of the expression is calculated as per the following formula:

    ```
    Expression width = LHS expression width * RHS expression
    value
    ```

    Consider the following example:

    ```
    out[23:0] = in1[1:0] ** 12;  //RHS Width 24, No
    violation
    ```

  - ❒ If the RHS expression of power operator is non static then the LHS expression width is reported. Consider the following example:

    ```
    out[1:0] = in1[1:0] ** in2[3:0];  //RHS Width 2, No
    violation
    ```

  - ❒ When the *nocheckoverflow* parameter is set to **yes**, the LHS expression width is reported. Consider the following example:

    ```
    out[23:0] = in1[1:0] ** 12; //RHS Width 2, violation

    out[1:0]  = in1[1:0] ** in2[3:0];   //RHS width 2, No
    violation
    ```

- For Unbased Unsized Constants

  - ❒ No violation is reported if the RHS is unbased unsized constant:

    ```
    assign out[3:0] = 3;    //No-Violation
    assign out[2:0] = 2'd3; //RHS Width 2, Violation
    ```

  - ❒ Behavior remains same when the *nocheckoverflow* parameter is set to **yes.**

- For concatenation operator, when the RHS expression is concatenated with zero bits:

- No violation is reported when the width of the LHS expression lies between the original width of the RHS expression and the width after adding zero concatenated bits. Here, the original width is the width without considering the zero concatenation.
- When the *nocheckoverflow* parameter is set to yes, a violation is reported when LHS width is greater than the RHS width after adding zero concatenated bits.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location where the width <*widthl*> of LHS of an assignment is greater than the width <*widthr*> of the RHS:

```
LHS width '<widthl>' is greater than RHS width '<widthr>'
[Hierarchy: ‘<hier-path>’]
```

Where, <*hier-path*> is the complete hierarchical path of the signal.

Rule Severity

Warning

Suggested Fix

It is more readable to explicitly extend as necessary, rather than relying on default behavior. In case of counters specified as integers with range, fix may not be required as you may have put the check to avoid overflow. This is the normal practice to use integer range for counters.

W257

Synthesis tools ignore delays

Language

Verilog, VHDL

Rule Description

The W257 rule flags delay/after clauses in the design.

For Verilog designs, the W257 rule reports a maximum of 500 rule messages per design and a maximum of 20 rule messages per module. Set the *allviol* rule parameter to **yes** or *<rule-name>* to report all violation messages.

As delays are ignored by synthesis tools, the pre- and post-synthesis simulations of designs with delays may not match.

NOTE: *The W257 rule also grouped under the Synthesis Rules group.*

Message Details

Verilog

The following message appears at the location where a **delay** statement is encountered:

```
Delays will be ignored in synthesis
```

VHDL

The following message appears at the location where an after clause is encountered:

```
AFTER clause will be ignored in synthesis
```

Severity

Warning

Suggested Fix

If possible, avoid using delays. Test the detailed timing in post-synthesis timing analysis, and not in simulation. Where essential to avoid race conditions, use only unit delays. Restrict the use of non-unit delays to

testbenches only.

W280

A delay has been specified in a nonblocking assignment

Language

Verilog

Rule Description

The W280 rule flags intra-assignment delays specified with nonblocking assignments.

Such description is unlikely to correspond to the physical implementation. However, such description may be required to use a unit delay to avoid race conditions in simulation.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameters

report_inter_nba: Default value is **no**. In this case, the W280 rule does not report inter-nonblocking assignment delay. Set the value of the parameter to **yes** to report such cases.

Message Details

The following message appears at the location where a nonblocking assignment is used to describe an intra-assignment delay:

`Intra-assignment delay used in a nonblocking assignment`

Rule Severity

Warning

Examples

Consider the following example, where an intra-assignment delay in a nonblocking assignment mimics a gate delay:

```
a <= #10 b;
```

W306

Converting integer to real

Language

Verilog

Rule Description

The W306 rule flags **`integer`** type to **`real`** type conversions.

Type conversion may indicate an unintended error. Such type of conversions may lead to loss in accuracy.

Message Details

The following message appears at the location where an **`integer`** type is being converted to a **`real`** type:

`Converting integer to real`

Rule Severity

Warning

Suggested Fix

Check to see if the conversion was intended. If yes, waive messages on those cases.

W307

Converting unsigned (reg type) to real

Language

Verilog

Rule Description

The W307 rule flags unsigned type (**`reg`** type) to **`real`** type conversions.

Any type conversion may indicate an unintended error. Such type of conversions may lead to loss in accuracy. Since a reg is effectively an unsigned integer, you may unexpectedly get a negative integer where you expected a positive value.

Message Details

The following message appears at the location where a **`reg`** type is being converted to a **`real`** type:

`Converting reg to real`

Rule Severity

Warning

Suggested Fix

Check to see if the conversion was intended. If yes, waive messages on those cases.

W308

Converting real to integer

Language

Verilog

Rule Description

The W308 rule flags **`real`** type to **`integer`** type conversions.

Any type conversion may indicate an unintended error. In this case there may be a loss of significant data (the fractional part of the float).

Message Details

The following message appears at the location where a **`real`** type is being converted to an **`integer`** type:

```
Converting real to integer
```

Rule Severity

Warning

Suggested Fix

Check to see if the conversion was intended. If yes, waive messages on those cases.

W309

Converting unsigned (reg type) to integer

Language

Verilog

Rule Description

The W309 rule flags unsigned type (**reg** type) to **integer** type conversions.

Type conversion may indicate an unintended error. Since a reg is effectively an unsigned integer, you may unexpectedly get a negative integer where you expected a positive value. There could also be a loss of significant data if the **reg** type is wider than the **integer** type (32 bits).

Message Details

The following message appears at the location where a **reg** type is being converted to an **integer** type:

`Converting reg to integer`

Rule Severity

Warning

Suggested Fix

Check to see if the conversion was intended. If yes, waive messages on those cases.

W310

Converting integer to unsigned (reg type)

Language

Verilog

Rule Description

The W310 rule flags **integer** type to unsigned type (**reg** type) conversions.

Type conversion may indicate an unintended error. Since a reg is effectively an unsigned integer, what may have been a signed value will become unsigned. This would in turn affect the outcome of arithmetic operations (For example, a negative number becomes bigger than a positive number).

Message Details

The following message appears at the location where an **integer** type is being converted to an unsigned type (**reg** type):

`Converting integer to reg`

Rule Severity

Warning

Suggested Fix

Check to see if the conversion was intended. If yes, waive messages on those cases.

W311

Converting real to unsigned (reg type)

Language

Verilog

Rule Description

The W311 rule flags **`real`** type to unsigned type (**`reg`** type) conversions.

Type conversion may indicate an unintended error. In this case, there may be a loss of significant data (the fractional part of the float).

Message Details

The following message appears at the location where a **`real`** type is being converted to an unsigned type (**`reg`** type):

```
Converting real to reg
```

Rule Severity

Warning

Suggested Fix

Check to see if the conversion was intended. If yes, waive messages on those cases.

W312

Converting real to single bit

Language

Verilog

Rule Description

The W312 rule flags violation when a **`real`** type node is assigned to single-bit node.

For example, consider the following scenario:

```
reg data;
real realval;
data = realval;
```

Any type conversion may indicate an unintended error, which may cause loss of significant data.

Message Details

The following message appears at the location where a **`real`** type is being converted in to a single-bit type:

```
Converting real to single bit
```

Rule Severity

Warning

Suggested Fix

```
Check to see whether the conversion was intended. Waive
violations on such cases.
```

W314

Converting multi-bit reg type to single bit

Language

Verilog

Rule Description

The W314 rule flags violation when multi-bit **reg** type node is assigned to a single-bit. For example, consider the following scenario:

```
output out1;
reg [3:0] count;
out1 = count
```

Any type conversion may indicate an unintended error. In this case, there may be loss of significant data.

By default, the W314 rule reports only one violation for every violating line, even if a module is instantiated more than once with different parameter values. Set the value of the *allviol* parameter to **yes** or *<rule-name>* to report different violations for every violating line resulting from a module, which is instantiated more than once with different parameter values.

NOTE: *The W314 rule supports generate-if, generate-for, and generate-case blocks.*

Message Details

The following message appears at the location where a multi-bit **reg** type is being converted in to a single-bit type:

`Converting multi-bit reg type to single bit [Hierarchy: '<hier-path>']`

Where, *<hier-path>* is the complete hierarchical path.

Rule Severity

Warning

Suggested Fix

Check to see whether the conversion was intended. Waive violations on such cases.

W317

Reports assignment to a supply net

When to Use

Use this rule to identify assignment to a supply net.

Rule Description

The *W317* rule reports assignments to supply nets.

Language

Verilog

Parameters

None

Constraints

None

Messages and Suggested Fix

The following message appears at the location where a supply net is being assigned.

`[WARNING] Assigning to a supply net is an error`

Potential Issues

Violation may arise when an assignment to a supply net is encountered in the design.

Consequences of Not Fixing

Assigning to a supply net, such as **`supply0`** and **`supply1`**, nets is an error as supply nets cannot be driven.

How to Debug and Fix

Select the message that you want to debug.

In the HDL window, it takes you to the line where assignment to a supply net is made.

To fix the violation, remove the assignment.

Example Code and/or Schematic

Consider the following example:

```
module W317_top1(i1, i2, o1);
  input i1, i2;
  output o1;
  reg o1;

supply1 vdd;

  assign vdd = i2;

endmodule
```

In the above example, the rule reports a violation for a supply net, **vdd**.

Default Severity Label

Warning

Rule Group

Assign

Reports and Related Files

None

W336

Blocking assignment should not be used in a sequential block (may lead to shoot through)

When to Use

Use this rule to identify blocking assignment in a sequential block.

Description

The *W336* rule reports a blocking assignment used in sequential always constructs. Hence, it is best to review these violations during RTL creation phase.

Apart from Verilog style blocking assignments, rule also reports violation for SystemVerilog style of blocking assignments.

Rule Exceptions

The *W336* rule fails to run if you set the value of the *fast* parameter to **yes** and the SpyGlass lint product is run, simultaneously.

Language

Verilog

Parameters

- *treat_latch_as_combinational*: Default value is **no**. This indicates the *W336* rule treats combinational block inferring latch as a sequential block. Set this parameter to **yes** to treat combinational block inferring latch as a combinational block.
- *check_temporary_flop*: Default value is **no** and the W336 rule does not report violation for temporary flip-flops. Set the value of this parameter to **yes** to report violation for temporary flip-flops.
- *ignore_local_variables*: Default value is **no** and the *W336* rule reports violations for variables (left hand side of the assignment) defined inside the sequential block. Set the value of this parameter to **yes** to ignore violations for such cases.
- *ignoreCellName*: Default value is not set. Set this parameter to a comma separated list of PERL regular expressions containing the module names that should be ignored by the *W336* rule.

Constraints

None

Messages and Suggested Fix

```
[WARNING] Blocking assignment '<assignment>' used inside a
<Latch|FlipFlop> inferred sequential block.
```

Potential Issues

Violation may arise when a blocking assignment is used inside a sequential block.

Consequences of Not Fixing

When a blocking assignment is used in a sequential block, inherent sequence of operation is implied in simulation. However, the synthesized hardware may behave in a concurrent fashion. Therefore, there is no assurance that the gate-level simulations match RTL level simulations. Also, the intent of the designer may not be fully captured in RTL.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the blocking assignment is used to infer flip-flop or latch in a sequential **`always`** block.

View the violation message to check the type of the sequential cell (flip-flop or latch) inferred from the block.

Use only nonblocking assignments in sequential blocks.

Example Code and/or Schematic

Example 1

Consider the following example:

```
module test3(clk, reset, d, q);
input clk, reset, d;
output q;
reg q;

always @(posedge clk or negedge reset)
begin
```

```
   if (!reset)
     q = 1'b0;
   else
     q = d;
end

endmodule
```

In the above example, the W336 rule reports a violation as a blocking assignment is used inside a flip-flop inferred sequential block.

Example 2

Consider the following example of SystemVerilog style blocking assignments:

```
always @ (posedge clk)
  sig++;

always @ (posedge clk)
  sig2 += 1;
```

In the above example, the rule reports a violation as the flip-flop is inferred in the design.

Example 3

Consider the following example:

```
module mod(in1, in2, clk, sel, out1);
   input [2:0] in1, in2;
   input clk, sel;
   output [2:0] out1;
   reg [2:0] out1;

   always @(posedge clk)
    if(sel)
      out1 = in1;
    else
      out1 = in2;

endmodule
```

In the above example, the W336 rule reports a violation as blocking

assignments are used in the design.

Example 4

Consider the following example:

```
module test(in1,en,out1);
input  in1;
input  en;
output out1;
reg    out1;

always begin
@(en)
   begin
   if(en)
      out1 = in1;
   end
end
endmodule
```

In the above example, the W336 rule does not report a violation when the value of the *treat_latch_as_combinational* parameter is set to **yes**.

Default Severity Label

Warning

Rule Group

Assign

Reports and Related Files

None

W397

Destination of an assignment is an IN port

Language

VHDL

Rule Description

The W397 statement flags signal assignments to input ports.

As you cannot drive an input, such assignments are design errors. An input bit should be driven only by an input port.

Message Details

The following message appears at the location where an input port *<port-name>* is assigned a value:

`Cannot assign to IN port '<port-name>'`

Severity

Fatal

Suggested Fix

Remove the assignment to the input port. If an OR or WIRE-OR function is necessary, create that function explicitly.

W414

Reports nonblocking assignment in a combinational block

When to Use

Use this rule to identify nonblocking assignments in a combinational block.

Description

The *W414* rule reports violations:

- When a nonblocking assignment is followed by a blocking assignment in a combinational block
- When a nonblocking assignment (signal used on LSH of assignment) is used in a clock path, unless the block is a latch

For combinational blocks, the rule reports violations only for the following cases:

- A nonblocking assignment is followed by the blocking assignments. For example, consider the following cases:

  Case 1

  ```
  out1 <= in1 & in2;
  out2 = in3 & in4;
  ```

  Case 2

  ```
  out1 <= in1 & in2;
  out2 < = in3 & in4;
  ```

  Case 3

  ```
  out1 = in1 & in2;
  out2 <= in1 & in2;
  ```

- Signal that is assigned using the nonblocking assignments is used in the clock path of a flip-flop. For example, consider the following case:

  Case 1

  ```
  always @(in1 or in2)
    clk <= in1;
  always @(posedge clk) //used as a clock
    out1 <= in1 & in2;
  ```

When the parameter *fast* is set to **yes**, the RTL version of the W414 rule is run.

The RTL version of the W114 rule reports violation for those non-blocking assignments that are followed by a blocking assignment in a combinational block. For example, consider the following cases:

Case 1

```
out1 <= in1 & in2;
out2 = in3 & in4;
```

Case 2

```
out1 <= in1 & in2;
out2 <= in3 & in4;
```

Case 3

```
out1 = in1 & in2;
out2 <= in1 & in2;
```

Language

Verilog

Parameters

- *treat_latch_as_combinational*: This indicates that the W414 rule treats combinational block inferring latch as a sequential block. Set this parameter to **yes** to treat the combinational block inferring latch as a combinational block. When the RTL version of the rule is run, all the latches are combinational except always_latch. Set the *treat_latch_as_combinational* parameter to **yes** to treat **always_latch** as a combinational block for the RTL version.
- *fast*: The default value is no. Set the value of the parameter to **yes** to run the RTL version of the W414 rule.

Constraints

None

Messages and Suggested Fix

The following message appears at the location where a nonblocking

assignment is used in a combinational **always** construct:

```
[WARNING] nonblocking assignment should not be used in a
combinational block
```

Potential Issues

Violation may arise when a nonblocking assignment is used in a combinational block.

Consequences of Not Fixing

Not fixing the violation may result in unexpected code behavior.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the nonblocking assignment is used for the specific signal. This assignment is specified inside a combinatorial **always** block.

You can confirm whether the **always** block is combinatorial using one of the following ways:

- Visual inspection of the block
- Scrolling up the HDL Viewer window till you reach the **always** block
- If the **always** block is a long block, view the HDL in an editor, and search backward for the nearest always keyword

If the **always** block is not combinational and is a latch, set the value of the *treat_latch_as_combinational* parameter as **yes**.

To fix this problem, use only blocking assignments in combinational blocks.

Example Code and/or Schematic

Consider the following example:

```
module test3(set, reset, p, q);
input set, reset;
output p, q;
reg p, q;

always @ (posedge p)
  q <= set;
always @(set or reset)
begin
```

```
  p <= set & reset;
  q <= set | reset;
end

endmodule
```

In the above example, the *W414* rule reports a violation as a nonblocking assignment is used in a combinational **always** block, when the *fast* parameter is not set.

Default Severity Label

Warning

Rule Group

Assign

Reports and Related Files

None

W446

Output port signal is being read (within the module)

Language

Verilog

Rule Description

The W446 rule flags output ports that are read in the module where they are set.

Such models could lead to an unintended feedback path from the instantiating module in the post-synthesis simulation while this issue is not apparent in the pre-synthesis simulation. Such models are also not recommended for some test tools that need to handle inout ports specially (by attaching bus-holders, for example).

NOTE: *Inside the nested* ***for*** *loops, signals of user-defined type like structures, interfaces, etc. are considered fully* ***set*** *if used on the left-hand side of an expression and fully* ***read*** *if used on the right-hand side of an expression.*

By default, this rule does not process large arrays. Set the *handle_large_bus* parameter to **yes** to process large arrays (greater than 50,000) and report violation if not used correctly.

The rule reports violation for all the scenarios by default. Set the value of the *flag_only_instance_ports* parameter to yes to report scenarios where output port is connected to the input instance pin.

Message Details

The following message appears at the location where an output port *<port-name>* is declared in a module when the port is being both set and read in the module:

```
Output port '<port-name>' is being read inside module.
[Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the hierarchical path of the containing scope.

Rule Severity

Warning

Suggested Fix

If the port really should be an inout port, change the declaration to inout. Otherwise, buffer the signal before the port to guard against possible feedback.

Examples

Consider the following example:

```
module top (out,feedback,in1, in2);
    input in1,in2;
    output reg out;
    output reg feedback;

    assign out=in1 & in2 ;
    assign feedback=out | in2;

endmodule
```

In the above example, the *W446* rule reports a violation because the output *out* is read in the design instead of setting it. The following message is reported by this example:

```
Output port 'out' is being read inside module.[Hierarchy:
':top']
```

W474

Variable assigned but not deassigned

Language

Verilog

Rule Description

The W474 rule flags variables that are assigned but not deassigned.

This rule flags an **assign** statement for register data types inside an **always** block, for which a **deassign** statement does not exist.

The procedural continuous assignment statement allows an expression to be driven continuously on registers or nets. It overrides the previous procedural assignments to a register.

The **deassign** statement ends a procedural assignment to register.

The value of register remains the same until the register is assigned a new value.

NOTE: *The W474 rule is switched off by default. You can enable this rule either by specifying the* **set_goal_option addrules W474** *command or by setting the verilint_compat rule parameter to* **yes**.

Message Details

The following message appears when the W474 rule encounters a variable *<var-name>* that is assigned but not deassigned:

```
Variable '<var-name>' assigned but not deassigned
```

Severity

Warning

Suggested Fix

Use deassign statement for the variable assigned.

Examples

Consider the following example where the variable '**q**' is assigned but not

deassigned:

```
module test5(inp,outp);
  input  inp;
  output outp;
  reg outp;
  reg q;
  always
    begin
      if(inp)
        deassign outp;
      else if (!inp)
        assign outp = 0;
      else
        assign outp = 1;
    end

  always
    begin
      if(outp)
        assign q = 1;
    end
endmodule
```

For this example, SpyGlass generates the following message:

```
Variable 'q' assigned but not deassigned
```

W475

Variable deassigned but not assigned

Language

Verilog

Rule Description

The W475 rule flags the deassignment statement for a variable that does not have a corresponding **`assign`** statement.

If a variable is deassigned, it should have a corresponding **`assign`** in the **`always`** block.

The W475 rule reports message for registers which are not assigned through procedural continuous assignment statement but are deassigned.

NOTE: *The W475 rule is switched off by default. You can enable this rule either by specifying the* **`set_goal_option addrules W475`** *command or by setting the verilint_compat rule parameter to* **`yes`**.

Message Details

The following message appears when a variable *<var-name>* is encountered that is deassigned without being assigned:

```
Variable '<var-name>' deassigned but not assigned
```

Severity

Warning

Suggested Fix

Use **`assign`** statement for the variable deassigned.

W476

Variable forced but not released

Language

Verilog

Rule Description

The W476 rule flags variables that are forced but are not released.

The **force** construct is a form of procedural continuous assignment like **assign**. It applies to registers, nets, a constant bit-select of a vector, a part select of a vector net, or a concatenation. It does not apply on a memory word (arrays) or a bit select or part select of a vector register.

NOTE: *The W476 rule is switched off by default. You can enable this rule either by specifying the* **set_goal_option addrules W476** *command or by setting the verilint_compat rule parameter to* **yes**.

Message Details

The following message appears when a variable <*var-name*> that is forced but not released, is encountered:

```
Variable '<var-name>' forced but not released
```

Severity

Warning

Suggested Fix

Use **release** statement for all the variables that are forced.

Examples

Consider the following examples, where

```
module test(a,b);
  input [3:0] a;
  output [3:0] b;
  reg [3:0] b;
```

```
  wire out1, out2 ,out3;

  always @(a or b)
    if(a)
      force {out1, out2, out3} = {1'b1,1'b0,1'b1};
    else
      release out1;
endmodule
```

W477

Variable released but not forced

Language

Verilog

Rule Description

The W477 rule flags the **release** statement for variables for which there is no corresponding **force** statement.

The W477 rule flags registers or nets that are not forced through procedural continuous assignment statement but are released.

NOTE: *The W477 rule is switched off by default. You can enable this rule either by specifying the* **set_goal_option addrules W477** *command or by setting the verilint_compat rule parameter to* **yes**.

Message Details

The following message appears at a location where a variable *<var-name>* is encountered that is released without being forced:

```
Variable '<var-name>' released but not forced
```

Severity

Warning

Suggested Fix

Use **force** statement for all the variables that are released.

W484

Possible loss of carry or borrow due to addition or subtraction

Language

Verilog

Rule Description

The *W484* rule reports possible loss of carry or borrow bits for assignments that are using addition and subtraction arithmetic operators.

The rule reports assignments where the result of an addition or subtraction operation is being assigned to a bus of the same width as the operands of the addition or subtraction operation. In such cases, the carry or borrow bit may be lost.

By default, the rule considers only the LRM width of the RHS in assignments. Set the *use_natural_width* parameter to consider both LRM and natural widths of the RHS in assignments.

NOTE: *For new width related changes, refer to New Width Flow Application Note.*

By default, the rule reports a violation for disabled code in loops and conditional (if condition, ternary operator) statements. Set the *disable_rtl_deadcode* parameter to **yes** to disable violations for disabled code in loops and conditional (**if** condition, ternary operator) statements.

Consider the following example:

```
if(0)
  a[3:0] = b[3:0] + c[3:0];
```

In the above example, the *W484* rule does not report a violation when the *disable_rtl_deadcode* parameter is set to **yes**, because the **if** statement in the above example is a disabled conditional statement.

By default, the rule does not report a violation for the following cases:

- Where same variable is assigned to itself:

```
reg [2:0] a;
a = a + 1;
```

- Where bit-width of RHS is less than or equal to the width of the LHS:

```
reg [2:0] a;
reg [2:0] b;
reg [1:0] c;
a = b + 1;
a = c + 1;
```

Set the *strict* parameter to report a violation for such cases.

By default, rule checking is disabled for RHS expressions that are static or for RHS expressions that are non-static but contain a static part.

Consider the following example:

```
wire [2:0] a, b, c;
assign a = 3'b101 + 3'b110; //RHS is static
assign a = b + c + 1;  //RHS is non-static with static part
assign a = b + 5;       //RHS is non-static with static part
```

To report violations in the above example, use the *check_static_value* parameter as follows:

- Set this parameter to **only_const** to enable rule checking for RHS expression, which is static.
- Set this parameter to **only_expr** to enable rule checking for RHS expression, which is a non-static expression that contains a static part.
- Set this parameter to **yes** to enable rule checking for static expression and non-static expression that contains a static part.

NOTE: *Cases of the type, a = b + c + 1, are considered as complex counters. Set either the strict or check_static_value parameter to report violations for such cases.*

By default, violations are not reported for the counter type of cases. Set the *strict* parameter to report violations for such cases. Consider the following example:

```
wire [2:0] a, b, c;
assign b = c + 1; //Will flag with strict set to yes
assign a = b + c + 5; //Not a counter, will not be flagged
                      //with strict set to yes
```

For static expressions, the rule reports a violation if carry or borrow bit is lost. For example:

```
wire [1:0] out;
assign out = 2'b10 + 1'b1;
assign out = 2'b10 + 2'b11;
```

The same logic is also applicable for static expressions inside concatenation. For example:

```
wire [2:0] out1;
wire c;
assign out1 = {2'b10+2'b01,c};
assign out1 = {2'b10+2'b10,c};
```

When a static value is added or subtracted from a non-static value, SpyGlass adds the maximum non-static value to the static value, and based on this value, SpyGlass calculates the width of the expression.

For example, consider the following code snippet:

```
wire [11:0] b;
wire [3:0] c;
 assign b = 12'hFF0 + c;
  assign b = 12'hFF1 + c;
```

In the above example, the maximum value of the first expression is 4095 and there is no overflow. Therefore, the rule does not report a violation for the first expression. In addition, the maximum value of the second expression is 4096 and there is an overflow. Therefore, the rule reports a violation for the second expression.

By default, the rule does not report any violation inside concat operator, if guard bit is added to carry bit to accommodate overflow. For example, no violation is reported in the following case:

```
wire [2:0] b,c;
wire [3:0] result;
wire [5:0] out

assign result = {b[0],b} + {c[0],c};
// LSB added to carry
assign result = {b[2],b} + {c[2],c};
// MSB added to carry
assign out = {{2{b[2]}},b[2],b} + {{2{c[2]}},c[2],c};
// MSB added to carry
assign out = {{3{b[2]}},b[2],b[2:1]} + {{3{c[2]}},c[2],c[2:1]};
// MSB added to carry
```

Set the *strict* parameter to report such cases.

Consider a case in which RHS contains an addition operator with at least one concatenation expression. The rule does not report a violation if the number of continued zeros at the end of concatenation expression is greater than or equal to the width of other operand. For example, no violation is reported in the following case:

```
wire [6:0] a;
wire [3:0] b;
wire [2:0] c;
assign a = {b, 3'b0} + c;
assign a = {3'b111} + {b, 3'b0};
```

NOTE: *The W484 rule supports the generate-if, generate-for, and generate-case blocks.*

By default, if the RHS expression is a binary expression and each term is a concatenation expression padded with zero bits, the rule checks for overflow while considering zero padded bits. That is, violation is not reported if the actual width of the expression can be accommodated in zero padded bits.

When the *handle_large_expr* parameter is set to **yes**, the rule limits this checking for expressions containing up to 25 terms and for larger

expressions, the LRM width is considered. For example:

```
module top();
wire [5:0] a1;
reg a,b,c,d,e,f,g,h,i,j;
assign a1[4:0] =   {4'b0,a} +  ({4'b0, b} + ({4'b0, c} +
({4'b0, d} + ({4'b0, e} + ({4'b0, f} + ({4'b0, g} + {4'b0,
h}))))))  ;    //8 terms, no violation
assign a1[4:0] =   {4'b0,a} +  ({4'b0, b} + ({4'b0, c} +
({4'b0, d} + ({4'b0, e} + ({4'b0, f} + ({4'b0, g} + {4'b0,
h}))))))  + {4'b0,a} + ({4'b0, b} + ({4'b0, c} + ({4'b0, d} +
({4'b0, e} + ({4'b0, f} + ({4'b0, g} + {4'b0, h}))))))  +
{4'b0,a} + ({4'b0, b} + ({4'b0, c} + ({4'b0, d} + ({4'b0, e}
+ ({4'b0, f} + ({4'b0, g} + {4'b0, h}))))))  + {4'b0, b};
// 25 terms, no violation
assign a1[4:0] =   {4'b0,a} + ({4'b0, b} + ({4'b0, c} +
({4'b0, d} + ({4'b0, e} + ({4'b0, f} + ({4'b0, g} + {4'b0,
h}))))))  + {4'b0,a} + ({4'b0, b} + ({4'b0, c} + ({4'b0, d}
+({4'b0, e} + ({4'b0, f} + ({4'b0, g} + {4'b0, h}))))))  +
{4'b0,a} + ({4'b0, b} + ({4'b0, c} + ({4'b0, d} + ({4'b0,
e} + ({4'b0, f} + ({4'b0, g} + {4'b0, h}))))))  + {4'b0, b}
+ {4'b0, b};     //26 terms, no violation by default,
violation is reported when parameter handle_large_expr is
set

endmodule
```

In the above example, by default the rule will not report violation for any of the assignment.

When the *handle_large_expr* parameter is set then the rule will report violation for the last statement as LRM width is considered for this because it has more than 25 terms.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location where the width <*widthl*> of LHS expression <*lexpr*> is not wider than the width <*widthr*> of the

RHS expression <*rexpr*>:

```
Possible assignment overflow: lhs width <widthl> (Expr:
'<lexpr>') should be greater than rhs width <widthr> (Expr:
'<rexpr>') to accommodate carry/borrow bit [Hierarchy: ‘<hier-
path>’]
```

Where, <*hier-path*> is the complete hierarchical path.

Rule Severity

Warning

Suggested Fix

Make sure you intend to discard the carry or borrow bit.

W505

Ensure that the signals or variables have consistent value.

When to Use

Use this rule to identify the signals or variables that are assigned values using both blocking and nonblocking assignments

Description

Verilog

The W505 rule flags signals or variables that are assigned values using both blocking and nonblocking assignments.

When creating waivers, rule ignores line numbers.

VHDL

The W505 rule flags signals or variables that are assigned values both with and without **`after`** clause (or with inconsistent **`after`** clauses).

Rule Exceptions

For VHDL, the W505 rule does not report signal assignments in asynchronous reset block.

Language

Verilog, VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message appears at the location where a signal or variable

<*name*> is being assigned in a block (nonblocking) mode when it has already been assigned in a nonblocking (blocking) mode:

```
[WARNING] Variable/Signal '<name>' is being assigned in both blocking and nonblocking manner
```

Potential Issues

A violation is reported arise when a variable or a signal is assigned in both blocking and nonblocking manner.

Consequences of Not Fixing

Synthesis semantics require that the same variable or signal should not be assigned in both blocking mode and nonblocking mode. Otherwise, the pre- and post-synthesis simulation behavior of the RTL and gate-level design may differ.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line, where a signal or a variable is again assigned a value again using the blocking/nonblocking assignment and the previous assignment to that signal was done through nonblocking/blocking assignment.

To fix the violation, ensure that all assignments to the same signal or variable are made in a consistent manner.

VHDL

Message 1

The following message appears at the location where a signal <*sig-name*> is assigned value using an **AFTER** clause when the same signal has already been assigned value at line <*line-num*> without using an **AFTER** clause:

```
[WARNING] Signal '<sig-name>' is used with 'after clause' while at <line-num> it was used without 'after clause'
```

Potential Issues

A violation is reported when a signal is first used without an AFTER clause and is subsequently assigned a value using the AFTER clause.

Consequences of Not Fixing

Assigning a value to the same signal/variable both with and without AFTER clauses (or with inconsistent AFTER clauses) is not synthesizable. As delays

are ignored during synthesis, the pre- and post-synthesis simulation

of the design containing AFTER constructs may differ.

How to Debug and Fix

For more information on debugging and fixing the violation, click *How to Debug and Fix*.

Message 2

The following message appears at the location where a signal <*sig-name*> is assigned value without using an **AFTER** clause when the same signal has already been assigned value at line <*line-num*> using an **AFTER** clause:

```
[WARNING] Signal '<sig-name>' is used without 'after clause'
while at <line-num> it was used with 'after clause'
```

Potential Issues

A violation is reported when a signal is first used with an AFTER clause and is subsequently assigned a value without using the AFTER clause.

Consequences of Not Fixing

Assigning a value to the same signal/variable both with and without AFTER clauses (or with inconsistent AFTER clauses) is not synthesizable. As delays are ignored during synthesis, the pre- and post-synthesis simulation

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line, where a signal is assigned a value again without the **after** clause and the previous assignment to that signal was done with the **AFTER** clause.

To fix the violation, ensure that all assignments to the same signal or variable are made in a consistent manner.

Example Code and/or Schematic

Example 1 (Verilog)

Consider the following example:

```
reg [3:0] reg_a;
reg_a <= 1 ;
reg_a =  2 ;
```

In the above example, **reg_a** is assigned using both blocking and nonblocking assignments. Hence, a violation is reported in this case.

Example 2 (VHDL)

Consider the following example:

```
signal z : std_logic;
signal a : std_logic;
signal b : std_logic;
signal c : std_logic;
z <= a or b;
z <= b and c after 10 ns;
```

In the above example, **z** is assigned with and without **after** clause. Hence, a violation is reported in this case.

Example 3

Consider the following example:

```
entity ent_test1 is
  port(a: in bit;
       b: in bit;
       c: in bit;
       z: out bit
  );
end ent_test1;

architecture behav of ent_test1 is
begin
  process(a, b, c)
  begin
    z <= a or b;
    z <= b and c after 10 ns;
  end process;
end behav;
```

In the above example, the W505 rule reports a violation as the signal, **z**, is used first with **after** clause and then without **AFTER** clause.

Default Severity Label

Warning

Rule Group

Synthesis, Assign

Reports and Related Files

No related reports or files.

Function-Task Rules

The SpyGlass lint product provides the following function and task related rules:

Rule	Flags...
W190	Tasks or procedures that are declared but not used in the design
W191	Functions that are declared but not used in the design
W243	Recursive task calls
W345	Presence of an event control in a task or procedure body may not be synthesizable
W346	Task descriptions with multiple event control statements
W372	User-defined PLI functions that are not registered in the Verilog Lint ruledeck file
W373	User-defined PLI tasks that are not registered in the Verilog Lint ruledeck file
W424	Functions that set global variables
W425	Functions that read global variables
W426	Tasks that set global variables
W427	Tasks that read global signals
W428	Task calls in combinational **`always`** constructs
W429	Task calls in sequential **`always`** constructs
W489	Functions where the last statement in the function description is not assigning to the function return value
W499	Functions where all bits of the function return value are not assigned in the function description

W190

Task or procedure declared but not used

Language

Verilog, VHDL

Rule Description

The W190 rule flags tasks or procedures that are declared but not called in the design.

While describing unused tasks is allowed, the W190 rule helps you clean up your design of unused tasks, thus reducing clutter in the design description or possibly preventing a potential name confusion error.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

Verilog

The following message appears at the location where a task *<task-name>* is declared that is not called anywhere:

```
Task '<task-name>' declared but not used
```

VHDL

The following message appears at the location of a procedure declaration *<proc-name>* that is not used in the design:

```
The procedure '<proc-name>' is not used
```

Rule Severity

Warning

Suggested Fix

If the task or procedure is confirmed to be redundant, remove it if possible. This will reduce clutter in the design and also reduce warning messages from SpyGlass.

W191

Function declared but not used

Language

Verilog, VHDL

Rule Description

The W191 rule flags functions that are declared but not used in the design.

While unused functions are allowed, the W191 rule helps you clean up your design of unused functions, thus reducing clutter in the design description or possibly preventing a potential name confusion error.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location of a function declaration <*func-name*> that is not used in the design:

Verilog

```
Function '<func-name>' declared but not used
```

VHDL

```
The function '<func-name>' is not used
```

Rule Severity

Warning

Suggested Fix

If the function is confirmed to be redundant, remove it if possible. This will reduce clutter in the design and also reduce warning messages from SpyGlass.

W242

This rule has been deprecated (Verilog)
A function is calling itself; that is, it is recursive (VHDL)

Language

Verilog, VHDL

Rule Description

Verilog

The W242 rule has been deprecated. The recursive functions are now synthesizable. In addition, a synthesis error SYNTH_5369 is flagged if a function is unrollable.

VHDL

The W242 rule flags recursive functions.

Recursive functions have no physical equivalent and may not be synthesized.

Message Details

VHDL

The following message appears at the location of a function declaration *<func-name>* that is recursive:

```
The function '<func-name>' is recursive thus may not be
synthesizable;
```

Rule Severity

Warning (VHDL)

Suggested Fix

Find some other way to implement the function. In general, whatever can be written recursively can be replaced by a loop.

W243

Recursive task enable

Language

Verilog

Rule Description

The W243 rule flags the task statement block, which has a call to task with the same name.

The W243 rule also reports a message, if a task call is recursive through more than one task. For example, if '**task_a**' calls task '**task_b**' and '**task_b**' calls '**task_a**' then this will be reported as recursive call.

Message Details

The following message appears when a task block having calls to task(s) *<task-name-list>* with the same name or having recursive task calls is encountered:

`Recursive task calls in the design:Order of task Calls: <task-name-list>`

Where *<task-name-list>* is the list of tasks, separated by dot, in the order of invocation.

Severity

Warning

Suggested Fix

In case of recursive task enable, there is always a chance that two tasks call each other infinitely. Avoid such situation by using loop constructs or some alternate method of writing the code.

Examples

Consider the following example where a task **t1** calls task **t1**:

```
module taskloop (in1, in2, out1);
  input  in1, in2;
```

```
  output out1;
  input  in1, in2;
  output out1;
  reg out1;

  always @(in1 or in2)
    begin
      t1 (out1, in1, in2);
    end

  task t1;
  output o2;
  input i1;
  input i2;

    begin
      t1 ( o2, i1 , i2 );
    end
  endtask
endmodule
```

For this example, SpyGlass generates the following message:

```
Recursive task calls in the design:Order of task Calls: t1.t1
```

W345

Presence of an event control in a task or procedure body may not be synthesizable

Language

Verilog, VHDL

Rule Description

The W345 rule detects presence of an event control in a task or a procedure body. The rule reports violation at the first event control statement in a task. It also reports the total number of event control statements present in the task.

The presence of an event control within a task or procedure is unsynthesizable by some synthesis tools.

Message Details

Verilog

The following message appears at the first event control statement in a task:

```
Event control statement should not be used in task body, total event usage '<num>' in task '<task_name>'
```

Where,

- *<num>* is the total number of event control statements present in a task.
- *<task_name>* is the name of the violating task.

VHDL

The following message appears at the location where a clock edge is used in a sub-program:

```
Clock in sub-program body may not be synthesizable
```

Rule Severity

Warning

Suggested Fix

No fix required if this occurs in a testbench. If this is targeted for synthesis, rewrite the task or procedure to avoid the event control.

W346

Task may be unsynthesizable because it contains multiple event controls

Language

Verilog

Rule Description

The W346 rule flags task descriptions with multiple event control statements.

Task descriptions with multiple event control statements are not synthesizable.

Message Details

The following message appears at the first line of a task description that has multiple event control statements:

`Task containing multiple event controls may be unsynthesizable`

Rule Severity

Warning

W372

A PLI function ($something) not recognized

Language

Verilog

Rule Description

The W372 rule flags user-defined PLI functions that are not registered in the Verilog Lint ruledeck file.

Besides the standard set of PLI functions that ships with each Verilog simulator, you can define your own PLI functions. Then, you need to register each user-defined PLI function to the Verilog Lint ruledeck by adding a line like the following sample in the PERL ruledeck file:

```
$dummyPLIfunc = "" if !defined $dummyPLIfunc;
```

Also, the W372 rule specifically reports for '$cast' PLI tasks or functions if the *report_cast* parameter is set to yes

Message Details

The following message appears at the location where an unregistered user-defined PLI function *<func-name>* is encountered:

```
PLI function (<func-name>) not recognized
```

The following message is displayed when the *report_cast* parameter is set to yes:

```
Non-synthesizable system function "<$cast>" is ignored, a 0
return value is assumed. The signal <signal-name> is unassigned
```

Rule Severity

Warning

Suggested Fix

Ensure that all the PLI functions to be used are registered in SpyGlass (this is locally configurable). If they are, check to make sure you have not mis-typed the name.

W373

A PLI task ($something) is used but not recognized

Language

Verilog

Rule Description

The W373 rule flags user-defined PLI tasks that are not registered in the Verilog Lint ruledeck file.

Besides the standard set of PLI tasks that ships with each Verilog simulator, you can define your own PLI tasks. Then, you need to register each user-defined PLI task to the Verilog Lint ruledeck by adding a line like the following sample in the PERL ruledeck file:

```
$dummyPLItask = "" if !defined $dummyPLItask;
```

Message Details

The following message appears at the location where an unregistered user-defined PLI task *<task-name>* is encountered:

```
PLI Task (<task-name>) not recognized
```

Rule Severity

Warning

Suggested Fix

Ensure that all the PLI tasks to be used are registered in SpyGlass (this is locally configurable). If they are, check to make sure you have not mistyped the name.

W424

Ensure that a function or a sub-program does not sets a global signal/variable

When to Use

Use this rule to identify a function or a sub-program, which is writing to a signal/variable outside its scope.

Description

The W424 rule reports violation for signals or variables that are not passed as parameters to a sub-program but are modified in the sub-program. The rule reports a violation only once for each global variable in a function or a sub-program.

Rule Exceptions

In VHDL, using global signal/variable inside a pure function is not allowed. SpyGlass generates syntax error, STX_472, in such cases.

Default Weight

5

Language

Verilog, VHDL

Parameter(s)

- *ignore_interface_locals*: The default value is no. Set the value of the parameter to yes to not report violations for functions inside an interface that are written to variables/signals in the interface scope.

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message appears at the location where a variable <*var-name*> that is not passed as a parameter, is modified in the function:

```
[WARNING] Function should not set a global variable: <var-name>
```

Potential Issues

Violation may arise when a function sets a global variable.

Consequences of Not Fixing

A global variable can potentially be modified from any part of the design. Therefore, it has unlimited potential for creating mutual dependencies, which increases complexity. This can lead to unexpected design behavior when the code is changed.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where a global variable is set inside the function body. This means this variable is not a function argument and is defined outside the function body, but it is set inside the function.

To fix the violation, it is advisable to only read from and write to internal variables.

VHDL

Message 1

The following message appears at the location where a signal *<sig-name>* that is not passed as a parameter, is modified in the sub-program *<subp_name>*:

```
[WARNING] Signal '<sig-name>' being modified in sub-program
'<subp-name>' is outside its scope
```

Potential Issues

Violation may arise when a signal modified in a sub-program is outside the scope of the sub-program.

Consequences of Not Fixing

A global signal can potentially be modified from any part of the design. Therefore, it has unlimited potential for creating mutual dependencies, which increases complexity. This can lead to unexpected design behavior when the code is changed.

How to Debug and Fix

Double-click the violation message. The HDL window, highlights the location of the declaration of the signal (global signal), which is modified inside the violating sub-program.

To fix the violation, it is advisable to only read from and write to internal variables.

Message 2

The following message appears at the location where a variable *<var-name>* that is not passed as a parameter, is modified in the sub-program *<subp_name>*:

[WARNING] Variable '<var-name>' being modified in sub-program '<subp-name>' is outside its scope

Potential Issues

Violation may arise when a variable modified in a sub-program is outside the scope of the sub-program.

Consequences of Not Fixing

A global variable can potentially be modified from any part of the design. Therefore, it has unlimited potential for creating mutual dependencies, which increases complexity. This can lead to unexpected design behavior when the code is changed.

How to Debug and Fix

Double-click the violation message. The HDL window, highlights the location of the declaration of the variable (global variable), which is modified inside the function specified in the violation message.

To fix the violation, it is advisable to only read from and write to internal variables.

Example Code and/or Schematic

Consider the following example where the function **factorial** sets the values of a global variable **g1**:

```
module func(in1, out1, out2, rst, clk);
  input clk, rst;
  input [3:0] in1;
  output [31:0] out1, out2;

  reg [31:0] out1, out2;
  integer g1;
```

```
  always @(in1 or rst)
    if (rst)
      out1 = 1;
    else
      out1 = factorial(in1);

  always @(posedge clk)
    out2 <= out1;

  function [31:0] factorial;
    input [3:0] operand;
    reg [4:0] index;
    begin
      factorial = operand ? 1 : 0;
      g1 = 1;
      for (index=2; index<=15; index=index+1)
        begin
          if (index <= operand)
            factorial = index * factorial;
        end
    end
  endfunction
endmodule
```

For this example, SpyGlass generates the following violation message:

Function should not set a global variable: g1

Default Severity Label

Warning

Rule Group

Function-task

Reports and Related Files

No related reports or files.

W425

Ensure that a function or a sub-program does not uses a global signal/variable

When to Use

Use this rule to identify a function or a subprogram, which is reading from a signal/variable outside its scope.

Description

The W425 rule reports violation for signals or variable that are not passed as parameters to a function or sub-program but are read in the functions or sub-programs. The rule reports a violation only once for each global variable in a function or a sub-program.

Rule Exceptions

In VHDL, using global signal/variable inside a pure function is not allowed. SpyGlass generates syntax error, STX_472, in such cases.

Default Weight

5

Language

Verilog, VHDL

Parameter(s)

- *ignore_interface_locals*: The default value is no. Set the value of the parameter to yes to not report violations for functions inside an interface that are read to variables/signals in the interface scope.

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message appears at the location where a variable *<var-name>* that is not passed as a parameter to a function, is read in the function:

```
[WARNING] Function uses a global variable: <var-name>
```

Potential Issues

Violation may arise when a function uses a global variable.

Consequences of Not Fixing

A global variable can potentially be modified from any part of the design. Therefore, it has unlimited potential for creating mutual dependencies, which increases complexity. This can lead to unexpected design behavior when the code is changed.

Also, usage of such functions under **`always @*`** creates simulation-synthesis mismatch because **`always @*`** is not sensitive to global variables used inside function.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where a global variable is read inside the function body. This means this variable is not a function argument and is defined outside the function body, but read inside the function.

To fix the violation, it is advisable to read only from the internal variables and task arguments.

VHDL

The following message appears at the location where a global variable named <*name*> that is not passed as a parameter to function <*func-name*>, is read in the sub-program:

```
[WARNING] Global shared variable '<name>' is read in function
'<func-name>'
```

Potential Issues

Violation may arise when a global shared variable is read in a function.

Consequences of Not Fixing

A global variable can potentially be modified from any part of the design. Therefore, it has unlimited potential for creating mutual dependencies, which increases complexity. This can lead to unexpected design behavior when the code is changed.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location

where a global variable is read inside the function body. This means this variable is not a function argument and is defined outside the function body, but read inside the function.

To fix the violation, it is advisable to only read from and write to internal variables.

Example Code and/or Schematic

In the following example, function **func1** directly reads signal **in1**:

```
module test (clk1, in1, in2, cntr, out1);
  input clk1, in1, in2;
  input [1:0] cntr;
  output out1;

  assign out1 = func1(cntr, in2, clk1);

  function func1;
    input clk, in_f;
    input [1:0] cntr_f;

    if (clk == 1'b0)
      func1 = in1 + cntr_f[1];
    else
      func1 = cntr_f[0];
  endfunction
endmodule
```

For this example, SpyGlass generates the following message:

Function should not use a global variable: in1

Default Severity Label

Warning

Rule Group

Function-task

Reports and Related Files

No related reports or files.

W426

Ensure that the task does not sets a global variable

When to Use

Use this rule to identify the task, which is writing to a variable outside its scope.

Description

The W426 rule reports violation for tasks that set global variables.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a variable *<var-name>* that is not passed as a parameter, is set in the task:

```
[WARNING] Global variable '<var-name>' should not be 'set' in task
```

Potential Issues

Violation may arise when a global variable is set in a task.

Consequences of Not Fixing

A global variable can potentially be modified from any part of the design. Therefore, it has unlimited potential for creating mutual dependencies, which increases complexity. This can lead to unexpected design behavior when the code is changed.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where a global variable is set inside the task. This means this variable is not a task argument and is defined outside the task, but it is set inside the task.

To fix the violation, it is advisable to only read from and write to internal variables.

Example Code and/or Schematic

In the following example, the global signal **b** is set in task **proc_a**:

```
module test(in, out, clk);
  input [3:0] in;
  input clk;
  output [3:0] out;

  reg [3:0] out, a, b, c;

  always @(posedge clk)
    begin
      a = in - 1;
      proc_a(a, c);
      out = a;
    end

  task proc_a;
    input [3:0] a;
    output [3:0] c;
    reg [3:0] c;

    begin
      b = a + 1;
      if (a==0)
        disable proc_a;
      c = b + 2;
    end
  endtask
endmodule
```

For this example, SpyGlass generates the following message:

```
Global variable 'b' should not be 'set' in task
```

Default Severity Label

Warning

Rule Group

Function-task

Reports and Related Files

No related reports or files.

W427

Ensure that a task does not uses a global variable

When to Use

Use this rule to identify a task, which is reading a variable outside its scope.

Description

The W427 rule reports violation for those variables that are not passed as parameters to a task but are read in the task.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a variable *<var-name>* that is not passed as a parameter to a task, is read in the task:

```
[WARNING] Global variable '<var-name>' should not be 'read' in task
```

Potential Issues

Violation may arise when a global variable is read in a task.

Consequences of Not Fixing

A global variable can potentially be modified from any part of the design. Therefore, it has unlimited potential for creating mutual dependencies, which increases complexity. This can lead to unexpected design behavior when the code is changed.

Also, usage of such tasks under **always @*** create simulation-synthesis mismatch because **always @*** is not sensitive to global variables used inside tasks.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where a global variable is read inside the task. This means this variable is not an argument to the task and is defined outside the task, but it is read inside the task.

To fix the violation, it is advisable to only read from and write to internal variables.

Example Code and/or Schematic

In the following example, the global signal **b** is read in task **proc_a**:

```
module test(in, out, clk);
  input [3:0] in;
  input clk;
  output [3:0] out;

  reg [3:0] out, a, b, c;

  always @(posedge clk)
    begin
      a = in - 1;
      proc_a(a, c);
      out = a;
    end

  task proc_a;
    input [3:0] a;
    output [3:0] c;
    reg [3:0] c;

    begin
      b = a + 1;
      if (a==0)
        disable proc_a;
```

```
      c = b + 2;
    end
  endtask
endmodule
```

For this example, SpyGlass generates the following message:

```
Global variable 'b' should not be 'read' in task
```

Default Severity Label

Warning

Rule Group

Function-task

Reports and Related Files

No related reports or files.

W428

Ensure that a task is not called inside a combinational block

When to Use

Use this rule to identify the tasks that are called inside a combinational block.

Rule Description

The W428 rule reports violation for task calls in combinational always constructs unless the construct describes a latch.

As tasks may contain event controls, calling a task inside a combinational block may turn that block, unexpectedly, into a sequential block. Ensure that the correct behavior is being modeled.

When the parameter *fast* is set to **yes**, the RTL version of the rule W428 is run. The always block inferring latch is considered as combinational block. The RTL version of this rule reports inside the latch inferred always block, except always_latch.

Rule Exceptions

You can enable the W428 rule either by specifying the **set_goal_option addrules W428** command.

Language

Verilog, VHDL

Default Weight

5

Parameter(s)

fast: The default value is no. Set the value of the parameter to yes to suppress synthesis of the source RTL description.

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a task call is encountered in a combinational **always** construct:

```
[WARNING] Task called in a combinational block
```

Potential Issues

Violation may arise when a task is called in a combinational block.

Consequences of Not Fixing

Since tasks may contain event controls, a task called inside a combinational **always** construct may turn that construct, unexpectedly, into a sequential **always** construct.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where a task call has been made inside a combinational always construct. User can check the nature of the always block using one of the following ways:

- Slightly scrolling up the HDL window, till he reaches the **always** block.
- If the **always** block is a very long block, user might view the HDL in an editor, and, search backwards for the nearest **always** block.

To fix the violation, it is advisable to use functions, rather than tasks, inside combinational blocks.

Example Code and/or Schematic

Consider the following example:

```
module test(in, out, clk, reset);
  input   in;
  input clk, reset;
  output  out;
  reg     out;

  always@(posedge clk)
  begin
       proc_d(in);// No Violation,Task called from sequential
block
```

```
  end

  always@(clk)
  begin
       proc_d(in);  // Violation, Task called from
combinational block

  end

  task proc_d;
  input   in;
  begin
  end
  endtask

endmodule
```

In the above example, the W428 rule reports a violation as a task is called from a combinational **`always`** block, when the *fast* parameter is not set.

Default Severity Label

Warning

Reports and Related Files

No related reports or files.

W429

Task called in a sequential block

Language

Verilog

Rule Description

The W429 rule flags task calls in sequential **always** constructs.

Since tasks may contain event controls, a task called inside a sequential **always** construct may unexpectedly create an implicit state machine (which may not be synthesizable if the task uses different edges or clocks from the block in which it is instanced). Therefore while it is not an error to call a task inside a sequential **always** block, it should be handled with an extra caution.

Message Details

The following message appears at the location where a task call is encountered in a sequential **always** construct:

`Task called in a sequential block`

Rule Severity

Warning

Suggested Fix

Nothing to fix if this is done right, but does suggest need for careful review.

Examples

Consider the following example where a sequential **always** construct has a task call **proc_a**:

```
module test(in, out, clk);
  input [3:0] in;
  input clk;
  output [3:0] out;
```

```
  reg [3:0] out;

  reg [3:0] r1, r2, r3;

  always @(posedge clk)
    begin
      r1 = in - 1;
      proc_a(r1, r3);
      out = r3;
    end

  task proc_a;
//  <task-body>
  endtask
endmodule
```

For this example, SpyGlass generates the W429 rule message.

W489

The last statement in a function does not assign to the function (Verilog)
The last statement in a function does not assign to the function (VHDL)

Language

Verilog, VHDL

Rule Description

The W489 rule flags functions where the last statement in the function description is not assigning to the function return value.

Statements after the last function return value assignment are effectively redundant. Hence, any functionality described after the last function return value assignment is lost.

The W489 rule does not report a violation if there is statement following the return value assignment inside the following constructs:

- if-else
- while loop
- for loop
- repeat
- case
- assign
- Other looping statements

NOTE: *Use the W499 rule to check if all bits of a function are set inside that function.*

Message Details

Verilog

The following message appears at the location of the last function return value assignment statement in a function description when other statements exist after this last assignment to the function return value:

```
The last statement in a function does not assign a return value
```

VHDL

The following message appears at the location of a function *<func-name>* that can return without executing a return statement:

```
The function <func-name> may return without executing a return statement
```

Rule Severity

Guideline (Verilog) / Warning (VHDL)

Suggested Fix

Check the function logic again. You should either remove the statements after the last assignment to the function value, or determine what function value should be assigned after those statements.

W499

Ensure that all bits of a function are set.

When to Use

Use this rule to identify the undefined bits of a function.

Rule Description

The W499 rule reports violation for functions returning a multi-bit value where all bits of the function return value are not assigned in the function description.

Language

Verilog

Default Weight

5

Parameter(s)

ignore_auto_function_return: The default value is no. Set the value of the parameter to yes to ignore name of automatic functions to be set in all branches of function body.

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the first line of a function description where all bit of the function return value are not assigned in the function description:

```
[WARNING] Not all bits of the function '<function_name>' value are set in the function body or in all branches of the function [Hierarchy: '<hier_path>']
```

Potential Issues

The W499 rule reports a violation when different bits in different parts of the function are not set or some bits in the conditional branch are not set.

Consequences of Not Fixing

This can be very error prone if not fixed. A user of the function will expect all bits of a function to be set.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the function, where, not all bits of the function are set.

Review the code and confirm the reason for not setting all bits of the function.

To fix the violation, make sure you define all bits in all possible branches in the function.

Default Severity Label

Warning

Rule Group

Function-task

Reports and Related Files

SignalUsageReport

Function-Subprogram Rules

The following function-subprogram rules have been deprecated:

W242 (Verilog)

The SpyGlass lint product provides the following function and subprogram rules:

Rule	Flags...
W190	Tasks or procedures that are declared but not used in the design
W191	Functions that are declared but not used in the design
W242	(Verilog) This rule has been deprecated. (VHDL) A function is calling itself, that is, it is recursive.
W345	Presence of an event control in a task or procedure body may not be synthesizable
W416	Functions where the range of the return type is not same as the function return value
W424	Functions that set global variables
W425	Functions that read global variables
W489	Functions where the last statement in the function description is not assigning to the function return value

W416

Width of return type and return value of a function should be same (Verilog). Reports functions in which the range of the return type and return value of a function are not same (VHDL)

When to Use

Use this rule to identify the functions in which the range of the return type and return value of a function is not the same.

Description

Verilog

The *W416* rule reports functions in which the width of the return type is not same as the function return value. This rule reports a violation for the following cases:

- There is a width mismatch between the return type and the return value of the function.
- There are order mismatches between the declared ranges in the return type and the return value - (reversed ranges).
- The rule will check for the order mismatches with the following limitations:
  - There should not be a width mismatch between the return type and the return value of the function.
  - The number of ranges in the return type and the return value should be same.
  - LHS and RHS of ranges should be static.

VHDL

The *W416* rule reports functions in which the range of the return type is not same as the function return value. This rule reports violation for the following cases:

- The left values and right values of the ranges of the return type and return value are different even if the widths are same.
- The range types of the return type and return value are different (**`to`** in one range and **`downto`** in the other).

Language

Verilog, VHDL

Parameters

Verilog

- *nocheckoverflow*: Default value is **no**. Set the value of the parameter to yes or rule name to calculate the width as per the LRM. See *Width Calculation for Verilog* and *Width Calculation for VHDL* to know more about calculating width.
- *check_static_value*: Default value is no. Set the value of the parameter to **yes** to report violation for cases with width mismatch, involving static expressions and non-static expressions having a static part.

  Other possible values:

  - **only_const** - Reports a violation for cases involving static expressions.
  - **only_expr** - Reports a violation for non-static expressions having a static part.
  - **only_static** - Does not report a violation for expression that does not have any static part.
  - **parameter** - Reports a violation when the expression is a parameter.
  - **sized_based_number** - Reports a violation when the expression is a sized based number.
  - **unsized_based_number** - Reports a violation when the expression is a unsized based number.
  - **constant** - Reports a violation when the expression is a constant.
- *disable_rtl_deadcode*: Default value is no. Set the value of the parameter to yes to disable violations for disabled code in loops and conditional (if condition, ternary operator) statements.
- *handle_zero_padding*: Default value is no. Set the value of the parameter to yes or rule name to perform leading zero expansion and truncation of RHS of an assignment.

- *consider_sub_as_add*: By default, when the **nocheckoverflow** parameter is set to **no**, the rule considers the width of expression like b -1 as the width of b when there is no underflow due to -ve operator. However, this behavior is limited to a simple - ve operation, where this is a part of arithmetic expression other than +/-. Set the value of the parameter to **yes** to enable the rule to calculate the width of a-b / b-1 as a+b/ b+1 respectively.
- *use_lrm_width*: Default value is **no**. Set this parameter to **yes** to consider the LRM width of integer constants, which is 32 bits.
- *W416_vhdl_only*: Default value is **no**. Set this parameter to **yes** to enable the rule to use only the VHDL version and check VHDL parts of the design.

Constraints

None

Messages and Suggested Fix

Verilog

Message 1: Width mismatch

```
[WARNING] Return type width '<width>' is <less | greater> than
return value width '<width>' in function  'function_name'.
[Hierarchy: '<hierarchy>']
```

Potential Issues

A violation is reported when the return type width is lesser or greater than the width of the return value of a function.

Consequences of Not Fixing

Not fixing the violation may lead to unexpected rule behavior.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the RTL line where the function value is returned.

Message 2: Range mismatch

```
[WARNING] Order mismatch between return type range(s)
'<range_list>' and return value ranges(s) '<range_list>' in
function '<function_name>'. [Hierarchy: '<hierarchy>']
```

Potential Issues

A violation is reported when there is an order mismatch between the return type and the return value in a function.

Consequences of Not Fixing

The order mismatch between expected return value of a function and the actual return value of a function can lead to two different interpretations during verification and test development.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the RTL line where the function value is returned.

To fix the violation, it is recommended to follow consistent order declarations throughout the design. To fix this problem, match the order of return value with that of return type of the function.

VHDL

The following message appears at the location of the return statement of a function <*func-name*> where the range of the return type (<*rettype-range*>) is different from the range of the return value (<*retvalue-range*>):

`[WARNING] Range Mismatch between return type (<rettype-range>) and return value (<retvalue-range>) in function '<func-name>'`

Where <*rettype-range*> and <*retvalue-range*> are range specifications in <*num*> **to** <*num*> format or <*num*> **downto** <*num*> format.

Potential Issues

Violation may arise when there is a range mismatch between the return type and the return value in a function.

Consequences of Not Fixing

The range mismatch between expected return value of a function and the actual return value of a function can lead to two different interpretations during verification and test development.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location of the return statement of a function where there is mismatch in range of

return type of the function and the range of return value of the function.

To fix the violation, it is recommended to follow consistent range declarations throughout the design. To fix this problem, match the range of return value with that of return type of the function.

Example Code and/or Schematic

Verilog

Example 1

Consider the following example:

```
module test (input in, output out);
function logic [3:0] test_func ();
logic [2:0] tmp;
return tmp; // violation - width mismatch
endfunction

assign out = test_func();
endmodule
```

Example 2

Consider the following example:

```
module test (input in, output out);
function logic [3:0] test_func ();
logic [2:0] tmp;
test_func = tmp; // violation - width mismatch
endfunction

assign out = test_func();
endmodule
```

Example 3

Consider the following example:

```
module test (input in, output out);
function logic [3:0] test_func (); // no-violation - widths are same, range are in same order
logic [4:1] tmp;
assign test_func = tmp;
endfunction

assign out = test_func();
endmodule
```

Example 4

Consider the following example:

```
module test (input in, output out);
function logic [2:0] test_func (); // no-violation - same width & same range
logic [2:0] tmp;
return tmp;
endfunction

assign out = test_func();
endmodule
```

Example 5

Consider the following example:

```
module test (input in, output out);
function logic [2:0] test_func (); // no-violation - same width and same range.
reg [2:0] tmp;
return tmp;
```

```
endfunction

assign out = test_func();
endmodule
```

Example 6

Consider the following example:

```
module test (input in, output out);
function logic [0:2] test_func ();
logic [2:0] tmp;
return tmp; // violation - order mismatch - ranges are in
reverse order
endfunction

assign out = test_func();
endmodule
```

Example 7

Consider the following example:

```
module test (input in, output out);
function logic [3:0] test_func ();
logic [3:0] [2:0] tmp;
return tmp; // violation - width mismatch
endfunction

assign out = test_func();
endmodule
```

Example 8

Consider the following example:

```
module test (input in, output out);
```

```
function logic [11:0] test_func (); // no-violation - widths are same
logic [3:0] [2:0] tmp;
return tmp;
endfunction

assign out = test_func();
endmodule
```

Example 9

Consider the following example:

```
module test (input in, output out);
function logic [3:0] [2:0] test_func (); // no-violation - ranges are identical
logic [3:0] [2:0] tmp;
return tmp;
endfunction

assign out = test_func();
endmodule
```

Example 10

Consider the following example:

```
module test (input in, output out);
function logic [2:0] [3:0] test_func (); // no-violation - widths are same
logic [3:0] [2:0] tmp; // rule will not flag for 'swapped ranges'
return tmp;
endfunction

assign out = test_func();
endmodule
```

Example 11

Consider the following example:

```
module test (input in, output out);
function logic [2:0] test_func ();
return 10; // violation - width mismatch
endfunction

assign out = test_func();
endmodule
```

Example 12

Consider the following example:

```
module test (input in, output out);
function logic test_func ();
logic [2:0] tmp;
return tmp; // violation - width mismatch
endfunction

assign out = test_func();
endmodule
```

Example 13

Consider the following example:

```
module test (input in, output out);
typedef logic [2:0] my_type;
function my_type test_func (); // no-violation, same width and same range
logic [2:0] tmp;
return tmp;
endfunction
```

```
assign out = test_func();
endmodule
```

Example 14

Consider the following example:

```
module test (input in, output out);
function logic [3:0] test_func (); // unused function
(disable_rtl_deadcode=no)
logic [3:0] [2:0] tmp;
return tmp; // violation by default - width mismatch
endfunction
endmodule
```

Example 15

Consider the following example:

```
module test (input in, output out);
function logic [3:0] test_func ();
logic [4:0] tmp;
logic [3:0] tmp_2;
if (0) // dead (disable_rtl_deadcode=no)
return tmp; // violation by default - width mismatch
else
return tmp_2;
endfunction

assign out = test_func();
endmodule
```

Example 16

Consider the following example:

```
module test (input in, output out);
```

```
function logic [3:0] test_func ();
logic tmp [2:0];
return tmp; // syntax error - STX_VE_310
endfunction

assign out = test_func();
endmodule
```

Example 17

Consider the following example:

```
module test (input in, output out);
function logic [3:0] test_func ();
logic [2:0] tmp;
logic [4:0] tmp_2;
return tmp + tmp_2; // violation - width mismatch
endfunction

assign out = test_func();
endmodule
```

Example 18

Consider the following example:

```
module test (input in, output out);
function logic [5:0] test_func ();
logic [2:0] tmp;
logic [4:0] tmp_2;
return tmp + tmp_2; // violation when nocheckoverflow yes
endfunction

assign out = test_func();
```

```
endmodule
```

Example 19

Consider the following example:

```
module test (input in, output out);
function logic [4:0] test_func ();
logic [2:0] tmp;
logic [4:0] tmp_2;
return tmp + tmp_2; // violation when nocheckoverflow no
endfunction

assign out = test_func();
endmodule
```

Example 20

Consider the following example:

```
module test (input in, output out);
function logic [2:0] test_func (); // no-violation
return '0;
endfunction

assign out = test_func();
endmodule
```

Example 21

Consider the following example:

```
module test (input in, output out);
function logic [3:0] test_func ();
reg [3:0] a = 4'b1010;
reg [2:0] b = 3'b110;
return {a,b}; // violation - width mismatch
```

```
endfunction

assign out = test_func();
endmodule
```

Example 22

Consider the following example:

```
module test (input in, output out);
function logic [6:0] test_func (); // no-violation - widths are same
reg [3:0] a = 4'b1010;
reg [2:0] b = 3'b110;
return {a,b}; // a concatenation
endfunction

assign out = test_func();
endmodule
```

Example 23

Consider the following example:

```
module test (input in, output out);
typedef struct {logic [2:0] a; logic [3:0] b;} ST;
typedef struct {logic [4:0] a; logic [5:0] b;} ST_2;
function ST test_func ();
ST_2 tmp;
return tmp; // syntax error - STX_VE_310
endfunction

ST x;
assign x = test_func();
endmodule
```

Example 24

Consider the following example:

```
module test (input in, output out);
function bit [3:0] test_func (); // no-violation - rule will not flag for signed unsigned mismatches
bit signed [3:0] tmp;
return tmp;
endfunction
assign out = test_func();
endmodule
```

Example 25

Consider the following example:

```
module test (input in, output out);
function logic signed [4:0] test_func ();
logic signed [2:0] tmp;
logic [4:0] tmp_2;
return tmp + tmp_2; // violation - signed unsigned mismatch
endfunction

assign out = test_func();
endmodule
```

Example 26

Consider the following example:

```
interface inf;
function logic [2:0] my_func; // no-violation
logic [3:1] tmp;
return tmp;
```

```
endfunction
endinterface

module test (input in, output out_1, out_2);
inf inst();
assign out_2 = inst.my_func();
endmodule
```

Example 27

Consider the following example:

```
module test (input in, output out_1, out_2);
parameter param = 1;
if (param == 1)
begin
function logic [2:0] test_func ();
logic [3:0] tmp;
return tmp; // violation - width mismatch
endfunction
assign out = test_func();
end
endmodule
```

VHDL

Example 1

Consider the following example:

```
...
subtype S1 is MH (0 to 7);
subtype S2 is MH (1 to 8);
...
function foo (a: S1) return S1 is
```

```
variable EN: S2;
...
return EN;
...
```

In the above example, left values and right values of the range of the return type S1 (0,7) are different from those of the range of the return value **S2** (1,8) while the actual width is same (8 bits). Therefore, the W416 rule reports a violation.

Example 2

Consider the following example:

```
...
subtype S1 is MH (0 to 7);
subtype S2 is MH (7 downto 0);
...
function foo (a: S1) return S1 is
variable EN: S2;
...
return EN;
...
```

In the above example, the W416 rule reports a violation as the range types of the return type and return value are different (**to** in one range and **downto** in the other).

Default Severity Label

Warning

Rule Group

Function-Subprogram

Reports and Related Files

None

Delay Rules

The SpyGlass lint product provides the following delay related rules:

Rule	Flags...
W126	Non-integer delay values
W127	Delay values containing X or Z
W128	Negative delays
W129	Non-constant delay values

W126

Do not use non-integer delays

Language

Verilog

Rule Description

The W126 rule flags non-integer delay values.

It is recommended to control delay precision using the **`'timescale`** directive.

A floating-point delay value will at best be truncated to the current timescale and will not track changes in timescale values.

NOTE: *The W126 rule does not flag negative integer delay values. See the W128 rule to catch negative delay values.*

Message Details

The following message appears at the location where a non-integer delay value is encountered:

`Do not use non-integer delay value <value>`

where, *<value>* is a delay expression.

Rule Severity

Warning

Suggested Fix

Change all delay values to integer values and use **`'timescale`** to control delay accuracy.

W127

Delay values should not contain X (unknown value) or Z (high-impedance state)

Language

Verilog

Rule Description

The W127 rule flags delay values containing X (unknown value) or Z (high-impedance state).

Though not a syntax error, such descriptions are meaningless as they describe delay for an indeterminate period of time.

Message Details

The following message appears at the location where a delay value containing X (unknown value) or Z (high-impedance state) is encountered:

`Delay value <value> should not contain X or Z`

where, *<value>* is a delay expression.

Rule Severity

Warning

Suggested Fix:

Change delay value to an integer value.

Examples:

Consider the following example:

```
out1 = # (8'h1x) in1;
out2 = # 5 in2;
out3 = #(4'b10z1) in3;
```

In the above example, the *W127* rule reports two violations because the delay values contain X (unknown value) or Z (high-impedance state).

The following messages are reported by this example:

```
Delay value '8'h1x' should not contain X or Z
Delay value '4'b10z1' should not contain X or Z
```

W128

Avoid using negative delays

Language

Verilog, VHDL

Rule Description

The W128 rule flags negative delay values.

A negative delay involves a step backward in time which is at least confusing (and therefore error-prone) and may cause efficiency problems in some simulators. At most, negative delays should be used only to model special behavior in cell libraries.

Message Details

The following message appears at the location where a negative delay value is encountered:

`Negative delay <value> specified`

where, *<value>* is a delay expression.

Rule Severity

Warning

Suggested Fix

Try to re-code the logic and use either zero or positive delays only.

W129

Variable delay values should be avoided

Language

Verilog

Rule Description

The W129 rule flags non-constant delay values.

While it is not a fundamental restriction, debugging in the presence of variable delays dramatically increases complexity and the opportunity for error.

Message Details

The following message appears at the location where a non-constant delay value is encountered:

`Variable delay values <value> should be avoided`

where, *<value>* is a delay expression.

Rule Severity

Warning

Suggested Fix

If possible, avoid using variable delay values.

Examples

Consider the following example:

```
module top (out,in);
    input in;
    output reg out;
    integer i=0;
    always @(in)
  begin
    out = #i in;
  end
```

```
endmodule
```

In the above example, the *W129* rule reports a violation because the delay value **i** is not a constant. The following message is reported by this example:

```
Variable delay value 'i' should be avoided
```

Lint_Latch Rules

The SpyGlass lint product provides the following latch related rules:

Rule	Flags...
W18	Latches inferred in the design

W18

Do not infer latches

Language

Verilog, VHDL

Rule Description

The W18 rule flags latches in the design.

Except where explicitly intended, latch inference is usually caused by an oversight — for example, failure to define a default value for a variable defined in a conditional statement — and may lead to incorrect behavior. Further, inferred latches in a register-based design cause problem for SpyGlass DFT tools.

By default, the W18 rule does not report latches inferred from both .sglib and .lib libraries. Set the *reportLibLatch* parameter to **`yes`** to report library latches.

NOTE: *You can enable the W18 rule by specifying the* **`set_goal_option addrules W18`** *command. However, this rule will not run, if you set the fast rule parameter to* **`yes`** *and SpyGlass lint product is run.*

Message Details

The following message appears at the location where a latch is inferred for signal <*sig-name*>:

```
Latch inferred for signal '<sig-name>' in module '<module-name>'
```

Severity

Info

Suggested Fix

Check the inference to make sure it is what you intended. If not, prevent latch inferences by providing an explicit **`else`** clause at the end of the **`if`** statement, or **`default`** clause at the end of the case statement, to prevent inferring the latch.

Examples

In the following example, a latch is inferred for signal **q**:

```
process (reset, d)
  begin
    if (reset = '1') then
      q <= d;
    end if;
end process;
```

For this example, SpyGlass generates the following message:

```
Latch inferred for signal 'q'
```

Instance Rules

The SpyGlass lint product provides the following module instance related rules:

Rule	Flags...
W107	Bus connections to primitive gates
W110	Width mismatch between a module port and the net connected to the port in a module instance
W110a	Use same port index bounds in component instantiation and entity declaration.
W146	Module instances where the port association is by position
W156	Reverse connected buses
W210	Module/Interface instances with unconnected ports
W287a	Module instances where nets connected to input ports are not driven
W287b	Module instances where the output ports are not connected
W287c	Module instances where the inout ports are not connected or connected net is hanging
W504	Port expression that uses integers

W107

Do not make bus connections to primitive gates (and, or, xor, nand, nor, xnor)

Language

Verilog

Rule Description

The W107 rule flags bus connections to primitive gates.

Verilog does not provide a method to check that a connection made to the inputs of a primitive gate (**and**, **or**, etc.) is of an appropriate width.

If a wider bus is attached, the function simply expands to include the additional bits. This expansion can lead to unexpected behavior if the bus width changes as the design evolves. Thus it is safer to explicitly connect, bit-by-bit, those bits, which should be gated.

Message Details

The following message appears at the location where connection of a bus *<net-name>* to a primitive gate is encountered:

`Do not make multibit bus (<net-name>) connections to gate ports. [Hierarchy: '<hier-path>']`

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Rule Severity

Warning

Suggested Fix

Make connections bit-by-bit. For example, use **`and(b[0],b[1],b[2]..)`** rather than **`and(b[0:3])`**.

Examples

Consider the following example that has multi-bit ports connected to primitive gates:

```
module test(in1, in2, in3, in4, in5, in6,
              out1, out2, out3);
  input [3:0] in1, in2, in3, in4, in5, in6;
  output [3:0] out1, out2, out3;

  and(out1, in1, in2);
  or(out2, in3, in4);
  xor(out3, in5, in6);
endmodule
```

SpyGlass generates the W107 rule message for each multi-bit connection (input and output) to primitive gates. Thus, nine messages are generated for the above example.

W110

Identifies a module instance port connection that has incompatible width as compared to the port definition

When to Use

Use this rule to identify the width mismatch between a module port and the net connected to the port of that module instance.

Description

The *W110* rule reports violations for the width mismatch between a module port and the net connected to the port of that module instance.

In addition, the *W110* rule checks the correct branch of a ternary operator and reports a violation when the port connection is a ternary operator and the condition of the ternary operator is evaluated.

NOTE: *The W110 rule supports generate-if, generate-for, and generate-case blocks.*

Calculation of width

The *W110* rule calculates the width of port expressions on the basis of the following conditions:

- When the *nocheckoverflow* parameter is set to **`yes`** or **`W110`**, the width is calculated as per LRM. For the constants, natural width is considered.
- When the **`nocheckoverflow`** parameter is set to **`no`**, the width is calculated according to following methods:
  - For the plus or the minus operator, the value based width is considered.
  - For the multiplication operator, the width is calculated as the sum of operand widths.
  - For the division operator, the LHS width is considered.
  - For the concat operator, the width is calculated as the sum of all operands.

Language

Verilog

Default Weight

10

Parameter(s)

- *check_concat_max_width*: Default value is no. Set the value of the parameter to yes to enable the rule to consider leading zeros to calculate its width.
- *nocheckoverflow*: Default value is **no**. Set this parameter to **yes** or rule name to check the bit-width as per LRM. Other possible value is the rule name.
- *report_blackbox_inst*: Default value is **no**. In this case, the rule does not report violation for port width mismatch for black box instances. Set the value of the parameter to **yes** to report violation for port width mismatch for black box instances.
- *use_lrm_width*: Default value is **no**. Set this parameter to **yes** to consider the LRM width of integer constants, which is 32 bits.
- *handle_shift_op*: Default value is **no**. In this case, no violation is reported if the shifted or non-shifted width of a shift expression (at the port connection of a module instance) matches the width of the corresponding module port definition. But the rule does not calculate the shifted width, if the RHS of the shift expression is non-static. Set this parameter to **shift_left**, **shift_right**, **shift_both**, **no_shift**, **no_shift_forced**, or comma separated list of rule names, to compare shifted or non shifted widths for left and right shift expressions.
- *handle_zero_padding*: Default value is no. Set the value of the parameter to yes or rule name to perform leading zero expansion and truncation of RHS of an assignment.

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for the *<inst-name>* module instance when there is a bit-width mismatch between a net of *<width2>* bits, which is connected to the *<port-name>* port of *<width1>* bits:

```
[WARNING] Incompatible width for port '<port-name>'(width
<width1>) on instance '<inst-name>'(actual width <width2>)
```

```
[Hierarchy: '<hier-path>']
```

Where, *<hier-path>* is the complete hierarchical path of the containing process.

Potential Issues

Width mismatch between module port and net connected to port or when the wrong signal is connected to the terminal.

Consequences of Not Fixing

A mismatched width connection causes the following results:

- Connection of a bus which is wider than the port: Excess bits are ignored for the input bus and floated for the output bus.
- Connection of a bus which is narrower than the port: Missing bits are driven unknown for the input bus and ignored for the output bus.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the module or the interface instance where a port connection has incompatible width as compared to the port definition.

The violation message shows the following information:

- Name of the port which has incompatible width in its port connection
- Name of the Module/Interface
- Name of the Module/Interface Instance
- Width of the port in the Module/Interface definition
- Width of the Connected port expression

To resolve the violation, ensure the connection of the correct number of bits across the hierarchy boundary, then explicitly ignore the bits that are not needed.

Example Code and/or Schematic

Example 1

Consider the following example:

```
inst IN1 (.in1(a[2:0]+b[2:0]));

// Width of expression, a[2:0]+b[2:0], will be 4 (7+7=14, 4
bits wide)
```

Example 2

Consider the following example:

```
inst IN1 (.in1(a[2:0]*b[2:0]));
// Width of expression, a[2:0]*b[2:0], will be 6
inst IN2 (.in1(a1[2:0]*b1[2]));
// Width of expression, a[2:0]*b1[2], will be 3
```

Example 3

Consider the following example:

```
inst IN1 (.in1(a[2:0]/b[2]));
// Width of expression, a[2:0]/b[2], will be 3
```

Example 4

Consider the following example:

```
inst IN1 (.in1({1'b0,a[2:0]}));
// Width is 3 bits (leading 0 ignored)

inst IN2 (.in1({1'b1,a[2:0]}));
// Width is 4 bits (1+3 bits)
```

Example 5

Consider the following example code:

```
module top(in1, in2, out1, out2, out3);
  input in1, in2;
  output out1, out2;
  output [1:0] out3;

  lower L1 (.i1(in1),.i2(in2),.o1(out1),.o2(out2));
  OR2 rtlc_I4 (.Z(out3), .A(in1), .B(in2));
endmodule

module lower(i1, i2, o1, o2);
  input [2:0] i1, i2;
  output o1, o2;

  reg o1, o2;
```

```
  always@(i1 or i2)
    begin
      o1 <= ~i1;
      o2 <= &i2;
    end
endmodule
```

In the instance **L1** of module **lower**, there is a bit-width mismatch between ports **i1**, **i2** (3 bits wide) and the connected nets **in1**, **in2** (single-bit). Therefore, SpyGlass generates the following messages:

```
Incompatible width for port 'i1'(width 3) on instance
'top.L1'(actual width 1) [Hierarchy: ‘top’]
```

```
Incompatible width for port 'i2'(width 3) on instance
'top.L1'(actual width 1)[Hierarchy: ‘top’]
```

In addition, in the instance **rtlc_14** of ASIC cell **OR2**, there is a bit-width mismatch between port **Z** (2-bits) and the connected net **out3** (single-bit). Therefore, SpyGlass generates the following message:

```
Incompatible width for port 'Z'(width 1) on instance
'top.rtlc_I4'(actual width 2)[Hierarchy: ‘top’]
```

Default Severity Label

Warning

Rule Group

Instance

Reports and Related Files

None

W110a

Use same port index bounds in component instantiation and entity declaration.

Language

VHDL

Rule Description

The *W110a* rule checks the mismatch, if any, in between the port index bound of component instantiation and entity declaration.

In VHDL, if there is any port width mismatch during component instantiation, it is a syntax error.

However, if a port is declared with the integer range and the range is configured through generic, it is not a syntax error. In such a case, there are chances that port range may overflow or underflow. Therefore, the purpose of this rule is to identify situations where the port index during component instantiation does not match with the entity declaration.

The rule also supports the following cases:

- Typed/sub-typed and aliased port.
- Positive and natural range data type, apart from integer ports.

Message Details

The *W110a* rule reports the following violation message at component instantiation:

```
Incompatible index for port '<port_name>'(Range:'<port_range>')
in component instantiation '<instance_name>' port
'<en_port_name>' (Range:'<en_port_range>')[Hierarchy: ‘<hier-
path>’]
```

Arguments

- Component instance port name, <port_name>
- Component instance port range, <port_range>
- Component instance name, <instance_name>
- Entity port name, <en_port_name>

- Entity port range, <en_port_range>
- Hierarchical path of the design unit where the component is instantiated, <hier_path>

Rule Severity

Warning

Examples

Syntax Errors

The following examples describes the cases in which there is a port width mismatch during component instantiation and the *W110a* rule considers this as syntax error:

Example 1

Consider the following example:

```
ain1 : in std_logic_vector (0 to 2)
ain : in std_logic_vector (0 to 7)
```

Example 2

Consider the following example:

```
ain1 : in integer range (-16) to (15)
ain : in integer range (-8) to (7)
```

In both the ***Example 1*** and ***Example 2***, component instantiation signal **`in`** is assigned with **`ain`**. Since there is a port width mismatch for the above examples and the range is not configured through generic, the rule considers this as a syntax error. However, if the range is configured through generic, the rule reports a **`warning`** message in this case.

Violating Cases

The following examples describe the cases where the rule reports a **warning** message because there is a port width mismatch and the range is configured through generic:

Example 1

Consider the following example:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity mult is
 generic (
  A: integer :=-1;
  B: integer :=1
 );
 port (ain : in integer range (A) to (B);
       z: out integer range (A) to (B)
       );
end mult;

architecture rtl of mult is
 begin
   process(ain) begin
   z <= ain  ;
 end process;
 end rtl;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity top is
 port (ain1 : in integer range (-16) to (15);
       cout1: out integer range -(16) to (15)
       );
end top;
```

```
architecture structure of top is
 component mult
 generic (
  A: integer :=-1;
  B: integer :=1
 );
 port (ain : in integer range (A) to (B);
       z: out integer range (A) to (B)
      );
 end component;
-- begin
 begin
  u1: mult
     generic map(
       A=> -8,
       B=> 7
     )
     port map(ain=>ain1,z=>cout1);
 end structure;
```

Example 2

Consider the following example:

```
ain1 : in integer range (-8) to (7)
ain : in integer range (-16) to (15)
```

In the above example, the component instantiation signal **in** is assigned with **ain**.

Example 3

Consider the following example:

```
ain1 : in integer range (2) downto (0)
ain : in integer range (15) downto (13)
```

In the above example, the component instantiation signal **in** is assigned with **ain**.

Non-Violating Cases

The following examples describe the cases where the rule does not report any violation because there is no port width mismatch:

Example 1

Consider the following example:

```
ain1 : in integer range (0) to (4)
ain : in integer range (4) downto (0)
```

Example 2

Consider the following example:

```
ain1 : in std_logic_vector (4 downto 0)
ain : in std_logic_vector (0 to 4)
```

Example 3

Consider the following example:

```
ain1 : in integer range (2) downto (0)
ain : in integer range (3) downto (1)
```

In ***Example 1***, ***Example 2***, and ***Example 3***, the component instantiation signal **`in`** is assigned with **`ain`**.

W146

Use named-association rather than positional association to connect to an instance

Language

Verilog, VHDL

Rule Description

The W146 rule flags module or component instantiations where the port association is by position.

Using positional association significantly increases the possibility of mis-connects, some of which may prove very subtle and difficult to find. Using named association completely eliminates this possibility. Also, other users do not have to cross-reference to the definition of the master of an instance to understand signal connections.

By default, the W146 rule does not report violation for parameters in instantiation, which are positional association. Set the value of the *check_param_association* parameter to **yes** to check for parameters in instantiation.

Message Details

Verilog

The following message appears at the location of a module instantiation where the port association is by position:

```
Explicit named association is recommended in instance references
```

VHDL

The following message appears at the location of a component instantiation where the port association is by position:

```
Explicit named association is recommended in instance port/generic map
```

Rule Severity

Guideline

Suggested Fix

In Verilog designs, if this is a top-level testbench, there is nothing to fix. Otherwise you can only read to and write from this design unit through global variables, which is unsynthesizable. May sometimes be OK for simulation monitors (though would still be considered by many to be poor coding style).

For VHDL designs, rewrite all instances using named-association connections.

W156

Do not connect buses in reverse order

Language

Verilog, VHDL

Rule Description

Verilog

The W156 rule flags reverse connected buses.

Making reversed connections (for example, **15:0** connected to **0:15**) is legal but bad design practice and may represent an error.

One exception (which can be handled with a waiver) is in making a **big-endian/little-endian** choice in connecting to a processor bus port.

VHDL

The W156 rule flags reverse connected buses.

Making reversed connections (for example, **15** downto **0** connected to **0** to **15**) is legal but bad design practice and may represent an error.

One exception (which can be handled with a waiver) is in making a **big-endian/little-endian** choice in connecting to a processor bus port.

For a constant declaration that has its subtype indication as an unconstrained array, the direction of the index constraint is the same direction of the index subtype definition used in the declaration of the unconstrained array. For example,

```
type MYBIT_VEC is array (natural range <>) of BIT;
...
constant myconst : MYBIT_VEC := "1011";
```

Here, the direction of **myconst** is same as that of **natural** that is **to**.

By default, the W156 rule does not report the reverse connections involving static expressions. Set the *strict* rule parameter to report such reverse connections also.

NOTE: *The W156 rule is also grouped under the Miscellaneous Rules group.*

Parameter(s)

- *report_all_connections*: Default value is **no.** Set the value of this parameter to **yes** to report a violation for all reverse order buses in the same module.
- *ignore_concat_expr*: Default value is **no.** Set the value of this parameter to **yes** to not report violations for the reverse connections which is an expression that concat operation is involved.

Message Details

Verilog

The following message appears at the location of a module instantiation where the port association of bus *<sig-name>* is in reverse order:

```
Bus net '<sig-name>' is connected in reverse. [Hierarchy: '<hier-path>']
```

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

VHDL

The following message appears at the location where the port association of bus *<sig-name>* is in reverse order:

```
<sig-name> contains bus connection in reverse order
```

Rule Severity

Warning

Suggested Fix

Where possible, connect in the same order as defined. In what should be localized **big-endian/little-endian** hookup cases, use a waiver.

Examples (Verilog)

In the following example, the module instance **lower1** has the bit-slice **sel[0:1]** is connected in reverse order than the port definition in the parent module **lower**:

```
module test(i1, i2, o1, o2);
  input [3:0] i1, i2;
  output [0:3] o1, o2;

  wire [0:5] sel;

  lower lower1(i1, i2, sel[0:1], o1, o2);
endmodule

module lower(in1, in2, sel, out1, out2);
  input [3:0] in1, in2;
  inout [1:0] sel;
  output [0:3] out1, out2;

  reg [0:3] out1, out2;

  always@(sel)
    case(sel)
      2'b00: out1 <= in1;
      2'b01: out1 <= in2;
      2'b10: out2 <= in1;
      2'b11: out2 <= in2;
    endcase
endmodule
```

For this example, SpyGlass generates the following message:

```
Bus net 'sel' is connected in reverse. [Hierarchy: ‘:test’]
```

W210

Number of connections made to an instance does not match number of ports on master

Language

Verilog, VHDL

Rule Description

The W210 rule flags a violation for module or interface instances with unconnected ports.

The W210 rule flags message only when number of terminals are less than the number of ports. The W210 rule ignores ports that are intentionally kept open (by passing extra comma [Verilog] or by connecting them to `open` [VHDL]).

If any of the unconnected ports are inputs or inouts, this is an error. It may or may not be an error if the unconnected ports are outputs.

NOTE: *The W201 rule does not report violations for constructs in the dead-code segments of a generate block.*

Parameters

- *set_message_severity*: The default value is **no** and the *W210* rule reports all unconnected ports with severity as **Warning**. Set this parameter to **yes** to report unconnected input or inout ports with messages severity as **Error**.
- *ignore_inout*: Default value is **no.** Set the value of this parameter to **yes** to not report any violation for inout port.

Message Details

Verilog

The following message appears at the location of instance <*inst-name*> of master module <*master-du-name*> where the port, <*port-name*>, is not connected:

```
[WARNING] Instance <inst-name>(master: <master-du-name>) has
too few ports - <port-type> port '<port-name>' not connected
```

where, <*port-type*> can be **input** or **inout**.

NOTE: *If the set_message_severity parameter is set to yes, the severity of this message will be Error.*

VHDL

The following message appears at the location of a component instantiation <*inst-name*>, of master module <*master-du-name*>, which has no port corresponding to the formal port <*port-name*>:

```
[WARNING] Port Map has no actual corresponding to formal <port-type> '<port-name>' (or some of its bits) in instance '<inst-name>' (master: <master-du-name>)
```

where, <*port-type*> can be **input** or **inout**.

NOTE: *If the set_message_severity parameter is set to yes, the severity of this message will be Error.*

Examples

Consider the following example code:

```
module top(input in, mis, output q);
  test u2(mis);
endmodule
module test(input in1, input in2);
endmodule
```

For the above example, the default violation message is:

```
[WARNING] Instance u2(Master: test) has too few ports - Input port 'in2' not connected
```

When the *set_message_severity* parameter is set to yes, the same message is generated with the message severity as Error.

```
[ERROR] Instance u2(Master: test) has too few ports - Input port 'in2' not connected
```

Rule Severity

Warning/Error

Suggested Fix

Make sure that all inputs and inouts are connected (at least tied off

appropriately if they are don’t care and that all outputs you expect to be used are connected.

W287a

Some inputs to instance are not driven or unconnected

Language

Verilog, VHDL

Rule Description

The W287a rule flags module or gate instances where nets connected to input ports are not driven and the instance input which are not connected.

The W287a rule (Verilog) also checks pins/ports driven by inout pins/ports.

For Verilog designs, the W287a rule does not flag instance input ports where the connected net is coming out of a black box instance unless the *strict* rule parameter is set.

It is an error in CMOS to let an input float, unless that input has a pull-up or pull-down.

In case, a bit-select or a part-select of a multi-bit signal is connected to an input of a module or gate instance, the W287a rule requires that all connected bits must be driven and other bits need not be driven.

NOTE: *For VHDL, you can enable the W287a rule by specifying the* **`set_goal_option addrules W287a`** *command. However, this rule will not run, if you set the fast rule parameter to* **`yes`** *and SpyGlass lint product is run.*

By default, the rule reports violation for port connections that have been explicitly left open in the RTL. Use the *filter_mark_open* parameter to not report violations on those port connections that have been explicitly left open in the RTL.

Turbo Mode Support

The Turbo mode support is available for this rule. In this mode, rule will not report violation for dead codes. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

Verilog

The following message appears at the location of a module instance where a signal <*sig-name*> of instance <*instance-name*> connected to an

input port is never driven:

```
[WARNING] Input '<sig-name>' of instance '<instance-name>' is
undriven. [Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Similarly, the following message appears where a signal *<sig-name>* of instance *<instance-name>* is not connected to an input port:

```
[WARNING] Input '<sig-name>' of instance'<instance-name>' is
unconnected. [Hierarchy: ‘<hier-path>’]
```

VHDL

The following message appears at the location of a component instantiation where bits *<bit-list>* of signal *<sig-name>* of instance *<instance-name>* connected to an input port are never driven:

```
[WARNING] Input Signal '<sig-name>' of instance '<instance-
name>' not driven. [Elaborated Module Name: <module-name>]
```

Where, *<module-name>* is the module name.

Rule Severity

Warning

Suggested Fix

If you don’t care about the input value, tie it high or low, but do not let it float.

Examples

Consider the following example where wire **w2** connected to second input port of instance **a1** of AND gate cell is not driven:

```
module test(in1, in2, clk, out1);
  input in1, in2, clk;
  output out1;

  wire w1, w2;
```

```
  assign w1 = in1 & in2;

  and a1(out1, w1, w2);
endmodule
```

For this example, SpyGlass generates the following message:

```
Undriven instance input 'w2'. [Hierarchy: ‘:test’]
```

W287b

Output port to an instance is not connected

Language

Verilog, VHDL

Rule Description

The W287b flags module or gate instances where the output ports are not connected.

While such descriptions are allowed, they are generally design mistakes.

NOTE: *For VHDL, you can enable the W287b rule by specifying the* **`set_goal_option addrules W287b`** *command. However, this rule will not run, if you set the fast rule parameter to* **`yes`** *and SpyGlass lint product is run.*

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameter(s)

- *report_hierarchy*: The default value of the parameter is no. Set the value of the parameter to yes to report rtl module name and hierarchy in the violation message.
- *filter_mark_open*: The default value is no. Set the value of the parameter to yes to not report violations on those port connections that have been explicitly left open in the RTL.

Message Details

Verilog

The following message appears at the location of a module instance where the output port <*port-name*> is not connected:

```
[Warning] Instance output port'<port-name>' is not connected
```

VHDL

The following message appears at the location of a component instantiation where the signal connected to an output port of the component is never

used in the design:

```
[Warning] Instance output '<sig-name>' not used. [Elaborated
Module Name: <module-name>]
```

Where, **<module-name>** is the name of the module.

The following message appears when the **report_hierarchy** parameter set to **yes**:

```
[Warning] Instance output '<output name>' of module '<rtl
module name>' not used. [Hierarchy: <instance hierarchy>]
```

NOTE: *The above violation is not applicable for turbo mode.*

Rule Severity

Warning

Suggested Fix

Check carefully to make sure that you intended to ignore this output.

Example

Consider the following example:

```
entity low1 is
  port(
    in1: in bit_vector(1 downto 0);
                out1: out bit
                  );
end low1;

architecture behav of low1 is
begin
  process(in1)
  begin
    out1 <= in1(1) and in1(0);
  end process;
end behav;

entity low2 is
  port(
```

```
   in2: in bit_vector(1 downto 0);
   out2: out bit
      );
end low2;

architecture behav of low2 is
begin
  process(in2)
  begin
    out2 <= in2(0) or in2(1);
  end process;
end behav;

entity top is
  port(
    inp1 : in bit_vector(1 downto 0);
                inp2 : in bit_vector(1 downto 0);
                outp : out bit_vector(1 downto 0)
      );
end top;

architecture struct of top is
component  low1 is
  port(
    in1: in bit_vector(1 downto 0);
    out1: out bit
      );
end component;

component low2 is  port(
   in2: in bit_vector(1 downto 0);
   out2: out bit
      );
end component;
signal sig1, sig2 : bit;
begin
    par_and : low1 port map (in1 => inp1, out1 => sig1);
    par_or  : low2 port map (in2 => inp2, out2 => sig2);
```

```
end struct;
```

W287c

Inout port of an instance is not connected or connected net is hanging

Language

Verilog, VHDL

Rule Description

The W287c flags module or gate instances where inout ports are not connected or connected net is not used anywhere in the module.

While such descriptions are allowed, they are generally design mistakes. When in the input mode, an unconnected inout port is equivalent to an undriven input which is a design error in CMOS unless it is a pull-up or pull-down.

Message Details

Verilog

The following message appears at the location of a module instance where the inout port, *<port-name>*, is not connected:

`Unconnected instance inout '<port-name>'[Hierarchy: ‘<hier-path>’]`

The following message appears at the location of the module instance where the net connected to the port *<port-name>* is hanging:

`Net connected to instance inout '<port-name>' is hanging [Hierarchy: ‘<hier-path>’]`

VHDL

The following message appears at the location of component instantiation where port, *<port-name>*, is not connected:

`Unconnected instance inout '<port-name>'[Elaborated Module Name: <module-name>]`

The following message appears at the location of component instantiation where a net connected to the port, *<port-name>*, is hanging:

`Net connected to instance inout '<port-name>' is hanging [Elaborated Module Name: <module-name>]`

where, <*module-name*> is the name of the module.

Rule Severity

Warning

Suggested Fix

Tie the input high or low if it will not be used.

W504

Integer is used in port expression

Language

Verilog

Rule Description

The *W504* rule reports the port expression that uses integers.

The rule reports a violation if during instantiating a module, a variable of type integer or a constant integer is connected to the module's port.

By default, the rule does not report violation for integers used in expressions containing scalar values, such as, 4'h4-2, 4'h4-i, where **i** is an integer variable.

Set the value of the *strict* parameter to **yes** to report such violations.

The W504 rule does not report violation if integer is used in module instantiation inside SystemVerilog module because integer ports are synthesizable in SystemVerilog.

You can run this rule by specifying the following option in the project file:

```
set_goal_option rules W504
```

If you set the *report_only_overflow* parameter to **yes**, the *W504* rule reports violations only for integers or constant integers where the width of port expression is greater than the width of port. By default, this parameter is set to **no**.

Message Details

The following message appears when the integer <*num*> is encountered in the port expression of the port <*port-name*>:

```
Integer <num> is used in port expression of port <port-name>
```

Severity

Warning

Suggested Fix

Do not use integers in port expression. Pass integer to lower level module by parameters.

Examples

Example 1

In the following example, the *W504* rule reports violations because integer variable **i** and constant integer **2** are used in the expression of port **a**:

```
module m;
        integer i;
       mm1 u1(.a(i));
       mm1 u2(.a(2));
endmodule

module mm1(input a);
endmodule
```

Example 2

In the following example, the *W504* rule reports violations, under the *strict* parameter, for integers used in expressions containing scalar values:

```
module top(output O);
integer i;

test    ins1 (.A(4'h4-2),.Q(O));  //Violation under strict
test   ins2 (.A(4'h3),.Q(O));   //No violation
test    ins4 (.A(4'h4-i),.Q(O));  //Violation under strict

endmodule

module test (input A,Q);
endmodule
```

Synthesis Rules

The SpyGlass lint product provides the following synthesis related rules:

Rule	Flags...
AllocExpr	Allocator expressions
ArrayEnumIndex	Arrays with enumeration type as index
No related reports or files.AssertStmt	ASSERT constructs
badimplicitSM1	Sequential descriptions where the clock and reset cannot be inferred
badimplicitSM2	Sequential descriptions where the states are updated on different clock edges
badimplicitSM4	Sequential descriptions where event control expressions use with more than one clock edge
BlockHeader	Port or generics used in the block header statements
bothedges	Multiple edges of a variable used in the control list
BothPhase	Processes that are driven by both edge of a clock
ClockStyle	Un-synthesizable clocking styles
DisconnSpec	Disconnection specification constructs
EntityStmt	Statements in entity description
ExponOp	Non-static left operands of the exponentiation operator
ForLoopWait	WAIT statements used in FOR-LOOP constructs
IncompleteType	Incomplete Types
infiniteloop	**`while`** or **`forever`** loops without event control to break the loop
InitPorts	Default initial value settings for output and inout ports
IntGeneric	Non-integer types used in generic declarations
LinkagePort	Ports of type LINKAGE
LoopBound	LOOP constructs with locally non-static bounds
mixedsenselist	(Verilog) Mixed edge and non-edge conditions in sensitivity list of an **`always`** construct (VHDL) Edge and level conditions specified together in **`if`** statement

Rule	Flags...
MultiDimArr	Multi-dimensional arrays
MultipleWait	Multiple WAIT constructs of the same clock expression
NoTimeOut	Timeouts in WAIT constructs
PhysicalTypes	declarations of un-synthesizable physical constructs
PortType	Ports of unconstrained types
PreDefAttr	Un-synthesizable pre-defined attributes
readclock	Sequential descriptions where the clock signal is read inside the always construct
ResFunction	Resolutions functions
ResetSynthCheck	All synthesis issues related to reset
SigVarInit	Initial value assignment to signals and variables
SynthIfStmt	IF, IF-ELSIF, and IF-ELSIF-ELSE constructs that have un-synthesizable constructs
UserDefAttr	User-defined attributes
W43	**wait** statements used in the design
W182c	**time** variable declarations
W182g	**tri0** declarations
W182h	**tri1** declarations
W182k	**trireg** declarations
W182n	Switches (**pmos**, **nmos**, and **cmos**)
W213	PLI functions
W218	Event expressions that check for edge on a multi-bit signal
W239	Hierarchical name references
W250	**disable** statements
W257	Delays
W293	Functions that return real values
W294	Un-synthesizable constructs
W295	**event** variables
W339	Runs *W339a* rule.
W339a	Identity operators — identity equal (===) and identity not equal (!==) operators

Rule	Flags...
W430	**`initial`** constructs
W442	Runs *W442a*, *W442b*, *W442c*, and *W442f* rules
W442a	Un-synthesizable asynchronous reset sequences
W442b	Complex reset sequences
W442c	Reset sequences where the reset signal is being modified by operators other than logical inverse (**!**) and bit-wise inverse (**~**) operators
W442f	Reset sequences where reset signal is being compared using an operator other than the binary equal (**==**) operator
W464	Unsupported synthesis directives
W496a	Comparisons to tristate signals in control expressions
W496b	Comparisons to tristate signals in `case` construct control expressions
W503	**`event`** variables that are never triggered
W505	Signals or variables that are being assigned values using both blocking and nonblocking assignments
WhileInSubProg	WHILE constructs used in sub-program descriptions

AllocExpr

Identifies the allocator expressions which are not synthesizable

When to Use

Use this rule to identify the allocator expressions which are not synthesizable by some synthesis tools.

Description

The *AllocExpr* rule reports violation for the allocator expressions which are not properly synthesizable by some synthesis tools.

Default Weight

5

Language

VHDL

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for an allocator expression which is not synthesizable by a synthesis tool:

[Warning] `Allocator expression may not be synthesizable`

Potential Issues

A violation is reported when an unsynthesizable allocator expression is encountered in a design.

Consequences of Not Fixing

Using the unsynthesizable allocator expression could lead to unsynthesizable code.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the violating allocator expression is used. This expression is un-synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. However, avoid this style in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
begin

process
  variable MOD1PTR,MOD2PTR:PTR;
begin
  MOD1PTR := new MODULE;

  MOD2PTR := new MODULE'(25, 10 ns, 4, 9);
end process;
```

In the above example, the AllocExpr rule reports a violation as **MOD1PTR** and **MOD2PTR** are allocator expressions.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

ArrayEnumIndex

When to Use

Use this rule to identify the arrays which are not synthesizable by some synthesis tools.

Description

The *ArrayEnumIndex* rule reports violations for the arrays defined using enumeration type as index.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed where an array is declared with enumeration type as its index as it is not synthesizable by a synthesis tool:

```
[WARNING] Array defined using enumeration type as index may not be synthesizable
```

Potential Issues

Violation may arise when an array is declared with enumeration type as its index.

Consequences of Not Fixing

Arrays defined using enumeration type as index are not synthesizable by some synthesis tools

How to Debug and Fix

Double-click the violation message. The HDL window highlights the line where an array is declared with an enumeration type as its index. This is un-synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. However, avoid this style in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
architecture arc of ent is
type my_type is(low1,mid1,high1);
type my_array is array(my_type(low1) to my_type(high1)) of bit;
```

In the above example, the ArrayEnumIndex rule reports a violation as the array, **my_array**, is declared using enumeration type as index.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

AssertStmt

Assertion statements have no significance in synthesis

Language

VHDL

Rule Description

The AssertStmt rule flags **`assert`** statements.

Assertions are for functional verification and have no meaning in synthesis. The **`assert`** statements are ignored by some synthesis tools.

NOTE: *Synopsys recommends use of* **`translate_off`** *and* **`translate_on`** *directives against* **`synthesis_off`** *and* **`synthesis_on`** *directives. If* **`synthesis_off`** *and* **`synthesis_on`** *directives are used within an expression, they have the potential to create incorrect logic by synopsys tools.*

Message Details

The following message appears at the location where an ASSERT statement is encountered:

```
Assertion statement may be ignored by synthesis tools
```

Severity

Warning

Suggested Fix

To avoid messages, enclose these statements in **`synthesis_off`** and **`synthesis_on`** pragmas.

badimplicitSM1

Identifies the sequential logic in a non-synthesizable modelling style where clock and reset cannot be inferred

When to Use

Use this rule to identify the sequential descriptions in a module which are not synthesizable by some synthesis tools.

Description

The *badimplicitSM1* rule reports violation for the sequential descriptions in a module where clock and reset cannot be distinguished and the module is not synthesizable by some synthesis tools.

Default Weight

5

Language

Verilog

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed at the last line of a module which violates the rule:

```
[ERROR] The 'if-else if' statement chain conditional expression
should check for all the asynchronous
reset/set signals first
```

Potential Issues

Violation may arise when uncommon sequential logic is used, which is unsynthesizable.

Consequences of Not Fixing

The module is not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the start of the **`if-else-if`** block. Scroll down the HDL Viewer window to search for the **`always`** block. To confirm the non-synthesizable modeling style for a sequential logic, perform one of the following steps:

- Inspect the block visually
- Scroll up the HDL Viewer window to search for the **`always`** construct
- Search the HDL for the **`if-else`** block inside the **`always`** construct
- Browse through the assignments and conditions used in the **`if-else`** block

Since the implicit modeling style is uncommon, it is most likely you made a mistake in modeling a conventional synchronous block. For a standard synchronous logic, it is neither required nor a good practice to use signals other than those present in the sensitivity list to set/reset the circuit.

Example Code and/or Schematic

In the following example code, SpyGlass generates a violation as it cannot distinguish between the clock and the reset signals:

```
module bism1(set,reset,in1,in2,out1);

input in1,in2,reset,set;
output out1;
reg clk,out1;
always @(posedge clk or negedge set)
  if(reset)
    out1 = 0;
  else if(!set)
    out1 = 1;
  else if(in2)
    out1 = in2;
  else
    out1 = in1;
endmodule
```

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

badimplicitSM2

Identifies the implicit sequential logic in a non-synthesizable modeling style where states are not updated on the same clock phase

When to Use

Use this rule to identify the implicit sequential descriptions in a non-synthesizable module which are not synthesizable by some synthesis tools.

Description

The *badimplicitSM2* rule reports violations for the sequential descriptions in a module whose states are updated on different clock edges and it is not synthesizable by some synthesis tools.

The implicit sequential logics are synthesizable only if all transitions within a block switch on the same edge of the same clock. Finite-state Machines (FSMs) are not synthesizable if you attempt to switch on both edges of the same clock or same edges of different clocks.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for the implicit sequential logic which is not synthesizable by a synthesis tool :

```
[ERROR] Unsynthesizable implicit sequential logic: states can only be updated on same clock phase
```

Potential Issues

A register which is updated on both edges of a clock in a given sequential block leads to ambiguous synthesis results.

Consequences of Not Fixing

Following are the consequences of not fixing the violation:

- The synthesis tool can get confused about which edge to use for updating the register.
- RTL and gate-level simulation results may not match.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the line where switching on the second edge of the same clock is done. To confirm the implicit sequential logic using both clock phases, perform one of the following steps:

- Inspect the block visually
- Scroll up the HDL Viewer window to search for the **`always`** construct
- Search the HDL for the edge of clock used in the **`if`** statement

To resolve the violation, break the sequential logic into multiple sequential logic blocks, each of which can independently meet the requirement. If the register depends on both edges of the clock, describe the sequential nature separately and use the combinational logic to generate the final output.

Example Code and/or Schematic

In the following example code, SpyGlass generates a violation as the state depends on more than one edge of the **`clk`** clock:

```
module test(out1,out2);
output out1,out2;
reg out1,out2,a,c,clk;
always
  begin
    @(posedge clk) out1 <= c;
    @(negedge clk) out2 <= a;
  end
endmodule
```

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

badimplicitSM4

Identifies the non-synthesizable implicit sequential logic where event control expressions have multiple edges

When to Use

Use this rule to identify the sequential descriptions which are not synthesizable by some synthesis tools.

Description

The *badimplicitSM4* rule reports violations for sequential descriptions where event control expressions use more than one edge. These descriptions are not synthesizable by some synthesis tools.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for an event control expression which uses another edge:

```
[ERROR] Unsynthesizable implicit sequential logic: event control expression may not have more than one edge
```

Potential Issues

A violation is reported A register which is updated on both edges of a clock in a given sequential block leads to ambiguous synthesis results.

Consequences of Not Fixing

Following are the consequences of not fixing the violation:

- The synthesis tool can get confused about which edge to use for updating the register.
- RTL and gate-level simulation results may not match.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the line in which multiple event control expressions are used in a condition. Scroll down the HDL Viewer window to search for the **`always`** block. To confirm the implicit sequential logic where control expressions do not have more than one edge, perform one of the following steps:

- Inspect the block visually
- Scroll up the HDL Viewer window to search for the **`always`** construct
- Search the HDL for the edges of clock used in the **`if`** statement

To resolve the violation, break the sequential logic into multiple sequential logic blocks, each of which can independently meet the requirement.

Example Code and/or Schematic

In the following example code, SpyGlass reports a violation as the event control expressions use multiple edges:

```
always
  begin
    @(posedge a or negedge a) out1 <= in1;
    @(negedge a) out2 <= in1;
  end
endmodule
```

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

None

BlockHeader

Identifies ports and generics in the block statement header which are not synthesizable

When to Use

Use this rule to identify ports and generics in the block statement header which are not synthesizable by some synthesis tools.

Description

The *BlockHeader* rule reports violations for ports and generics used in the block header statements.

Default Weight

10

Language

VHDL

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for a port or a generic in a block statement which is not synthesizable by a synthesis tool:

`[WARNING] PORT/GENERIC in the block statement header may be unsynthesizable`

Potential Issues

A violation is reported when ports or generics are used in the block statement.

Consequences of Not Fixing

Ports and generics used in a block statement are not synthesizable by some synthesis tools.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the beginning of the block in which a port, a generic, or both are defined. The block contains the definition of the port, the generic, or both. If the block is too long, view the HDL in an editor and search forward for the port or generic definition.

No fix is required in testbenches or simulation models. Avoid this style in the code targeted for synthesis. If the register depends on both edges of the clock, describe the sequential nature separately and use the combinational logic to generate the final output.

Example Code and/or Schematic

In the following example code, the rule reports a violation for the non-synthesizable port used in a block statement:

```
entity entBlockHeader1 is
 port(in1    : in bit;
   in2    : in bit;
   pcarry : in bit;
   sum    : out bit;
   scarry : out bit);
end entBlockHeader1;
architecture behavioral of entBlockHeader1 is
begin
  b1 : block \\port may be unsynthesizable
  port (a, b, c : in bit;
    d, e : out bit
    );
  port map (a=>in1, b=>in2, c=>pcarry, d=>sum,   e=>scarry);
    begin
      d <= (a xor b xor c) ;
      e <= ((a and b) or (a and c) or (b and c));
end block;
process (in1, in2)
  begin
    sum <= in1 ;
    sum <= in2 ;
end process ;
```

```
end behavioral;
```

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

bothedges

Identifies the variable whose both the edges are used in an event control list

When to Use

Use this rule to identify the variable which is not synthesizable by some synthesis tools.

Description

The *bothedges* rule reports a violation for a variable whose both the edges are used in a sensitivity list, hence making the variable not synthesizable by some synthesis tools.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for an event control list that contains both the edges of the <*var-name*> variable:

```
[ERROR] Both edges of the same variable ( <var-name> ) are not allowed in the event control list
```

Potential Issues

A violation is reported when both edges of the same variable are used in an event control list.

Consequences of Not Fixing

Synthesis tools do not allow both edges of the same variable in an event

control list.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the sensitivity list of the **always** block. Check the sensitivity list. It displays both the **posedge** and the **negedge** edges of the signal mentioned in the message.

To resolve the violation, replicate the block, one switching on the positive edge and the other on the negative edge.

Example Code and/or Schematic

In the following example code, SpyGlass generates a violation as both edges of the same variable are being used in a control list, hence making the variable non-synthesizable:

```
module test(q);

output q;
reg q,d,reset;

always @(posedge reset or negedge reset)
  begin
    if (reset != 0)
      q =  d;
  end
```

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

None

BothPhase

Identifies the processes that are driven by both the edges of a clock

When to Use

Use this rule to identify the processes that are driven by both the edges of a clock.

Description

The *BothPhase* rule reports violations for processes that are driven by both the edges of a clock.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed at the first line of the *<process-name>* process that is driven by both the edges of the *<clk-name>* clock:

```
[WARNING] Clock '<clk-name>' driving a process ('<process-name>') on two different edges may not be synthesizable
```

Potential Issues

A violation is reported when a process is driven by both the edges of a clock.

Consequences of Not Fixing

The processes that are driven by both the edges of a clock are not

synthesizable by some synthesis tools.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the process, which is driven by both the edges of a clock.

No fix is required in testbenches or simulation models. In the code targeted for synthesis, split it into two processes, one driven on the positive edge and the other on the negative edge.

Example Code and/or Schematic

The rule checks for the following type of clock declarations which use both the edges:

Sample Code 1

```
if( clk'event and clk = '1' ) then --positive edge
  q1 <= data;
end if;
if( clk'event and clk = '0' ) then --negative edge
  q2 <= data;
end if;
```

Sample Code 2

```
if( not clk'stable and clk='1' ) then  --positive edge
  q1 <= data;
end if;
if( not clk'stable and clk='0' ) then  --negative edge
  q2 <= data;
end if;
```

Sample Code 3

```
wait until clk='1';  --positive edge
q1 <= data;
wait until clk='0';  --negative edge
q2 <= data;
```

Sample Code 4

```
wait until clk'event and clk='1';  --positive edge
q1 <= data;
wait until clk'event and clk='0';  --negative edge
q2 <= data;
```

Sample Code 5

```
wait until rising_edge(clk);  --positive edge
q1 <= data;
wait until falling_edge(clk); --negative edge
q2 <= data;
```

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

ClockStyle

A clocking style is used which may not be synthesizable

Language

VHDL

Rule Description

The ClockStyle rule flags un-synthesizable clocking styles used in the design.

The following table lists the clocking styles may not be synthesizable by some synthesis tools:

TABLE 1 Un-synthesizable Clocking Styles

`if (clk'event) then` `elsif(clk'event) then`
`if(not clk'stable) then` `elsif(not clk'stable) then`
`if (clk'stable) then` `elsif(clk'stable) then`
`if (clk'stable and clk = '1') then` `elsif(clk'stable and clk = '1') then`
`if (clk'stable and clk = '0') then` `elsif(clk'stable and clk = '0') then`
`wait until clk'event;`
`wait until not clk'stable;`
`wait until clk'stable;`
`wait until clk'stable and clk = '1';`
`wait until clk'stable and clk = '0';`
`if (not clk'event and clk = '0') then`
`if (clk'event and not clk'stable) then`
`if (clk'event and clk'stable)`

TABLE 1 Un-synthesizable Clocking Styles (Continued)

`wait until not clk'event and clk = '0';`
`wait until clk'event and not clk'stable;`
`wait until ((clk'event and clk = '1') xor (cond = 1 ))`
`if ((clk'event and clk'stable) and clk= '0') then`
If clock is being used as multi-bit signal. For example: `if (clk'event and clk = "00") then)`

The ClockStyle rule checks for clocking styles based on **`IF`**, **`ELSIF`**, and **`WAIT`** constructs only.

NOTE: *Gate between clock expression and other conditions should not be other than the 'and'.*

Such descriptions may not be synthesizable by some synthesis tools.

Message Details

The following message appears at the location where an un-synthesizable clocking style is encountered:

`This clocking style may not be synthesizable`

Severity

Warning

Suggested Fix

No fix is required in testbenches or simulation models. In code targeted for synthesis, recode to meet synthesizability guidelines.

DisconnSpec

Identifies the disconnection specification constructs which are not synthesizable

When to Use

Use this rule to identify the disconnection specification constructs which are not synthesizable by some synthesis tools.

Description

The *DisconnSpec* rule reports violations for the disconnection specification constructs which are not synthesizable by some synthesis tools.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for a disconnection specification construct which is not synthesizable by a synthesis tool:

```
[Warning] Disconnection specification may not be synthesizable
```

Potential Issues

A violation is reported when a disconnection specification construct is used in the design.

Consequences of Not Fixing

The disconnect specification is used to model the disable time for a signal driver in a guarded assignment. Hence, it is ignored during synthesis.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the disconnection specification construct used in the design.

No fix is required in testbenches or simulation models. Avoid this style in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
--library IEEE;
--use ieee.std_logic_1164.all;
--entity e is
--end e;

architecture arc of e is
 signal a : std_logic register;
 signal b : std_logic bus;
 disconnect a :std_logic after 50 ns;
 disconnect b :std_logic after 20 ns;
--begin
--process
--begin
--  if(b = '0') then
--     a <= '1' after 8 ns;
--  else
--     a <= null after 10 ns;
--  end if;
--  wait on b;
--end process;
--end arc;
```

In the above example, the DisconnSpec rule reports a violation as the disconnection specification constructs are used in the design.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

EntityStmt

Statements in entity block may be ignored by some synthesis tools

Language

VHDL

Rule Description

The EntityStmt rule flags statements in entity declarations.

Statement part of entity declarations is ignored by some synthesis tools.

NOTE: *Synopsys recommends use of* **translate_off** *and* **translate_on** *directives against* **synthesis_off** *and* **synthesis_on** *directives. If* **synthesis_off** *and* **synthesis_on** *directives are used within an expression, they have the potential to create incorrect logic by synopsys tools.*

Message Details

The following message appears at the location where a statement is encountered in an entity declaration:

```
Statement inside an entity may be ignored by some synthesis tools
```

Severity

Warning

Suggested Fix

Avoid this style in synthesizable code or enclose these statements in **synthesis_off** and **synthesis_on** pragmas.

ExponOp

This rule has been deprecated.

This rule is now removed. Built-In rules *SYNTH_5338* and *SYNTH_5240* can be used instead.

There is no need to add *SYNTH_5338* and *SYNTH_5240* in run script as these run by-default.

ForLoopWait

Identifies the WAIT statements used in FOR-loop constructs which are not synthesizable

When to Use

Use this rule to identify the WAIT statements which are not synthesizable by some synthesis tools.

Description

The *ForLoopWait* rule reports violations for the WAIT statements used in FOR-loop constructs.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for a WAIT statement in a FOR-loop construct which is not synthesizable by a synthesis tool:

```
[WARNING] WAIT statement inside a FOR-loop statement may be unsynthesizable
```

Potential Issues

A violation is reported when WAIT statement is used inside a FOR-loop construct.

Consequences of Not Fixing

The WAIT statements that are used inside a FOR-loop are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location of the for loop which has offending wait statement inside the for loop.

If the for loop is small, you can scroll down to see the usage of the **wait** statement. If the for loop is too long you might view the HDL in an editor and search forward for wait statement inside the for loop to find the actual location of the **wait** statement. This is un-synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. Put event control outside loops in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
--entity test is
--end test;

--architecture test of test is
--  signal sig1 : bit;
--begin

  p1: process
  begin
    for i in 0 to 10 loop
      wait on sig1;
    end loop;
  end process p1;

--end test;
```

In the above example, the ForLoopWait rule reports a violation as a **wait** statement is used inside the **for** loop.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

IncompleteType

Identifies the incomplete type declarations which are not synthesizable

When to Use

Use this rule to identify the incomplete type declarations which are not synthesizable by some synthesis tools.

Description

The *IncompleteType* rule reports violations for the incomplete type declarations which are not synthesizable by some synthesis tools.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for an incomplete type declaration which is not synthesizable by a synthesis tool:

```
[WARNING] Incomplete type declaration may not be synthesizable
```

Potential Issues

A violation is reported when an incomplete type declaration is used in a design.

Consequences of Not Fixing

Incomplete type declarations are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending incomplete type is declared. This is un-synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. Avoid this style in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
--entity e is
--end e;

architecture arc of e is
type COMP;
type NET;

--type COMP_PTR is access COMP;
--type NET_PTR is access NET;

--constant MODMAX:integer:= 100;
--constant NETMAX: integer := 2500;

--type COMP_LIST is array(1 to MODMAX) of COMP_PTR;
--type NET_LIST is array(1 to NETMAX) of  NET_PTR;

--type COMPLIST_PTR is access COMP_LIST;
--type NETLIST_PTR is access NET_LIST;

--type COMP is
--  record|
--    COMP_NAME : STRING(1 to 10);
--    NETS      : NETLIST_PTR;
--  end record;

--type NET is
--  record
--    NET_NAME   :  STRING(1 to 10);
```

```
--    COMPONENTS :  COMPLIST_PTR;
--  end record;

--begin
--end arc;
```

In the above example, the IncompleteType rule reports a violation as an incomplete type declaration is used in the design.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

infiniteloop

While/forever loop has no break control

Language

Verilog

Rule Description

The inifiniteloop rule flags **`while`** or **`forever`** loops without event control to break the loop.

Such descriptions may not be synthesizable by some synthesis tools.

NOTE: *The offending construct is now synthesizable by almost all standard synthesis tools. Therefore, the infiniteloop rule has been switched off and will be removed in the next release of SpyGlass.*

NOTE: *The infiniteloop rule is switched off and will be removed in a future release. Use the SpyGlass SYNTH_5230 Built-In rule that reports the while unrolling loops.*

Message Details

The following message appears at the location of a **`while`** or **`forever`** loop's terminating condition when the condition has no event control to break the loop:

```
while/forever loop has no event control to break loop
```

Rule Severity

Error

Suggested Fix

Fix may not be necessary if this is expected to be free-running, but you should check each such case.

Examples

In the following example, the terminating condition of the **`while`** loop is always false, resulting in an infinite loop:

```
`define cond 1'b1
```

```
module test(in1, in2, sel, out1);
  input in1, in2, sel;
  output out1;
  reg out1;

  always@(sel or in1)
    while(`cond)
      case(sel)
        1'b0: out1 <= in1;
        1'b1: out1 <= in2;
      endcase
endmodule
```

For this example, SpyGlass generates the infiniteloop rule message.

InitPorts

Default initial value of in/out/inout port may be ignored by some synthesis tools

Language

VHDL

Rule Description

The InitPorts rule flags default initial value settings for input, output, and inout ports.

Such descriptions may not be synthesizable by some synthesis tools.

NOTE: *Synopsys recommends use of* **`translate_off`**/**`translate_on`** *directives against* **`synthesis_off`**/**`synthesis_on`** *directives. If* **`synthesis_off`**/**`synthesis_on`** *directives are used within an expression, they have the potential to create incorrect logic by synopsys tools.*

Message Details

The following message appears at the location where an input, output, or inout port *<port-name>* is declared with default initial value:

```
Default initial value of in/out/inout port '<port-name>' may be ignored by some synthesis tools
```

Severity

Warning

Suggested Fix:

If you want to set initial state of ports, use reset connections. If you need initial values for simulation, enclose those statements in the **`synthesis_off`**/**`synthesis_on`** pragmas. Ideally, try to avoid using such statements in synthesizable code since they may lead you to believe that the logic is functioning correctly when it actually does not reset correctly in the implementation.

IntGeneric

Identifies the non-integer type used in the declaration of a generic which is not synthesizable

When to Use

Use this rule to identify the non-integer types used in generic declarations which are not synthesizable by some synthesis tools.

Description

The *IntGeneric* rule reports violations for the non-integer types used in generic declarations which are not synthesizable by some synthesis tools.

Rule Exceptions

Enumeration type generics are not reported by the rule as the synthesis process supports them.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for the <*type-name*> non-integer type, which is used in the declaration of the <*gen-name*> generic and is not synthesizable by a synthesis tool:

```
[WARNING] Non-integer type '<type-name>' used in declaration of generic '<gen-name>' may be unsynthesizable
```

Potential Issues

A violation is reported when a non-integer type is used in a generic

declaration.

Consequences of Not Fixing

Non-integer types used in a generic declaration may not be synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window takes you to the location where the offending non-integer generic is declared. This is un-synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. Use only integer generics in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
entity test is
  generic
    gen1 : real := 10.0);
end test;
```

In the above example, the IntGeneric rule reports a violation for **`gen1`** as it is a non-integer generic declaration.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

LinkagePort

Identifies the linkage ports which are not synthesizable

When to Use

Use this rule to identify the linkage ports which are not synthesizable by some synthesis tools.

Description

The *LinkagePort* rule reports violations for ports which are of the **`linkage`** type and not synthesizable by some synthesis tools.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for a port of the **`linkage`** type which is not synthesizable by a synthesis tool:

```
[WARNING] Linkage port may not be synthesizable
```

Potential Issues

A violation is reported when a port of **`linkage`** type is used in the design.

Consequences of Not Fixing

Ports of **`linkage`** type are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending port of type **`linkage`** is declared. This is un-

synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. Avoid this style in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
entity e is
port(
  a:  linkage bit;
  b : out bit
);
end  e;
```

In the above example, the LinkagePort rule reports a violation for the port a, as it is of type **linkage**.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

LoopBound

Identifies the for loop range bounds that are not locally or globally static

When to Use

Use this rule to identify the For loop constructs which are not synthesizable by some synthesis tools.

Description

The *LoopBound* rule reports violations for the **`for`** loop constructs that have locally or globally non-static bounds and are therefore not synthesizable by some synthesis tools.

Language

VHDL

Default Weight

10

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed at the first line of a FOR-loop construct that has a locally or a globally non-static range bound:

```
[WARNING] For loop range bounds should either be locally static or globally static
```

Potential Issues

A violation is reported when a for loop construct has locally or globally non-static bounds.

Consequences of Not Fixing

The synthesis process cannot be deterministic and can possibly hang if the

loop bounds are not static. In such cases, there is also a possibility of hanging of the simulation.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the for construct, which has locally or globally non-static bounds.

To resolve the violation, use a static lower and a static upper bound. If this is not possible, implement the loop as a multi-cycle logic.

To avoid hanging of the simulation, check the **`for`** loops at the RTL coding stage.

Example Code and/or Schematic

Example 1

In the following example code, SpyGlass reports a violation as **`dataIn`** is non-static:

```
signal dataIn : integer ;
for i in 1 to dataIn loop
  temp <= temp * i ;
end loop ;
```

Example 2

In the following example code, no violation is reported by SpyGlass:

```
for i in 1 to 3 loop
  temp <= temp * i ;
end loop ;
```

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

mixedsenselist

Mixed conditions in sensitivity list may not be synthesizable (Verilog)
Edge and level conditions are mixed in if statement (VHDL)

Language

Verilog, VHDL

Rule Description

Verilog

The mixedsenselist rule flags mixed edge and non-edge conditions in the sensitivity list of an **`always`** construct.

Such conditions in sensitivity list may not be synthesizable by some synthesis tools.

VHDL

The mixedsenselist rule flags if edge and level conditions are specified together in an **`if`** statement.

Message Details

Verilog

The following message appears at the location of an **`always`** construct where mixed edge or non-edge condition is used in the sensitivity list:

```
Mixed conditions in sensitivity list may not be synthesizable
```

VHDL

The following message appears if edge and level conditions are specified together in an **`if`** statement:

```
Edge and level conditions are mixed in 'if' statement
```

Rule Severity

Error (Verilog) / Warning (VHDL)

Suggested Fix

Decide if you really want to trigger the block on any change in the mixed signal. Check if your requirement can be met by mapping to either a positive-edge change or a negative-edge change. If both are required, consider duplicating the block, one triggering on each edge.

Examples (Verilog)

In the following example, the **always** construct has both an edge specification and a non-edge specification:

```
always @(posedge clock or reset)
  q = d;
```

For this example, SpyGlass generates the mixedsenselist rule message.

Examples (VHDL)

In the following example, both edge and level conditions are specified together in the **if** statement:

```
elsif ((clk'event and clk='1')and (d1='1') and
  (d2='0')) then
```

Here, clock edge **clk** is mixed with **d1** and **d2** level checking. Hence, SpyGlass generates the mixedsenselist rule message.

MultiDimArr

This rule has been deprecated.

Since multi-dimensional arrays are supported by SpyGlass and are synthesizable, this rule is no longer valid and is removed.

MultipleWait

Identifies multiple wait statements having the same clock expression which are not synthesizable

When to Use

Use this rule to identify multiple **wait** constructs which are not synthesizable by some synthesis tools.

Description

The *MultipleWait* rule reports violations for multiple **wait** constructs having the same clock expression which are not synthesizable by some synthesis tools.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for a **wait** construct at the <*num*> line when another **wait** construct is found for the same clock expression:

```
[WARNING] Multiple wait statements (first one found at line no.
<num>) having same clock expression may not be synthesizable
```

Potential Issues

A violation is reported when multiple **wait** statements have the same clock expression.

Consequences of Not Fixing

Some synthesis tools support multiple **wait** statements in a process though it is not a generally accepted synthesizability practice. But even then, your formal verification tool may not support the coding style.

How to Debug and Fix

Double-click the violation message. The HDL window, highlights the location of the wait statement which has same clock expression as the previously declared wait statement (line number of the wait statement is also mentioned in the violation) inside the process block. This is un-synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. Ensure that your synthesis flow supports this style in the code targeted for synthesis. Also, use a waiver to avoid the violation if your downstream tool supports the coding style.

Example Code and/or Schematic

Consider the following example:

```
begin
  latch_behavior: process is
  begin
    wait until clk='1';
      q1<=d ;
    wait until clk='1';    --VIOLATION
       q2<=d ;
    wait until clk='1';
      q3<=d ;
end process latch_behavior;
```

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

NoTimeOut

Identifies the timeout expression in a wait statement, which is not synthesizable

When to Use

Use this rule to identify the timeout expressions which are not synthesizable by some synthesis tools.

Description

The *NoTimeOut* rule reports violations for the timeout expression in a wait construct, which is not synthesizable by some synthesis tools.

The rule reports wait constructs which are in the following format:

`WAIT for <time-expression>;`

where *<time-expression>* is the wait time.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for a timeout expression in a wait construct, which is not synthesizable by a synthesis tool:

`[WARNING] Timeout expression in wait statement may not be synthesizable`

Potential Issues

A violation is reported when a timeout expression is used in a **`wait`**

statement.

Consequences of Not Fixing

The timeout expressions in a wait construct are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the location where the offending timeout expression in wait statement is used. This is un-synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. Avoid this style in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
entity test is
end test;

architecture test of test is
  signal sig1, sig2 : bit;
begin  -- test

  p1: process
    begin  -- process p1
    sig1 <= sig2;
    wait for 10 ns;
  end process p1;

end test;
```

In the above example, the NoTimeOut rule reports a violation as a timeout expression is used inside a wait statement.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

PhysicalTypes

Identifies the physical constructs which are not synthesizable

When to Use

Use this rule to identify the physical constructs which are not synthesizable by some synthesis tools.

Description

The *PhysicalTypes* rule reports violations for declaration of the Physical, Floating Point, File, or Access type physical constructs which are not synthesizable by some synthesis tools.

Language

VHDL

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed where an un-synthesizable physical construct <*construct-name*> is declared:

[Warning] `<construct-name> declaration may be ignored by some synthesis tools`

Potential Issues

A violation is reported when a physical construct is used in a design.

Consequences of Not Fixing

The Physical Type, Floating Point Type, File Type, and Access Type constructs are not synthesizable by most of the synthesis tools.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the line where a Physical, Floating, File, or Access type object is

declared. The message specifies the type of the offending object.

No fix is required in testbenches or simulation models. Enclose these statements in the **synthesis_off**, **synthesis_on** pragmas in the code targeted for synthesis.

To create correct logic, Synopsys recommends to using the **translate_off** or the **translate_on** directive instead of the **synthesis_off** or the **synthesis_on** directive.

Example Code and/or Schematic

In the following example code, SpyGlass reports a violation for the physical and the access type physical constructs which are non-synthesizable:

```
end ent;
architecture arc of ent is
  type current is range 0 to 1E9
  units
    nA;
  end units;  \\physical type declaration
  type MODULE is
    record
      SIZE    : integer range 20 to 200;
      i       : current;
    end record;
  type PTR is access MODULE;\\access type declaration
begin
  process
    variable MOD1PTR,MOD2PTR:PTR;
  begin
    MOD1PTR := new MODULE;
    MOD2PTR := new MODULE'(25, 10 nA);
  end process;
end arc;
```

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

PortType

Identifies ports of unconstrained types which are not synthesizable

When to Use

Use this rule to identify ports which are not synthesizable by some synthesis tools.

Description

The *PortType* rule reports violations for ports of unconstrained types.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed where the <*port-name*> port of the unconstrained type, which is not synthesizable by a synthesis tool, is declared:

`[WARNING] Port <port-name> of unconstrained type may not be synthesizable`

Potential Issues

A violation is reported when a port of unconstrained type is used in the design.

Consequences of Not Fixing

Ports of unconstrained types are not synthesizable by some synthesis tools

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where unconstrained port is used. This is un-synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. Avoid this style in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
package testpack is
  type unconsIntArr is array (integer range <>) of integer;
end testpack;

use work.testpack.all;
entity test is
  port (
    p1 : in unconsIntArr);
end test;
```

In the above example, the PortType rule reports a violation for the port, **p1**, as it is of unconstrained type.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

PreDefAttr

Identifies the pre-defined attributes which are not synthesizable

When to Use

Use this rule to identify the pre-defined attributes which are not synthesizable by some synthesis tools.

Description

The *PreDefAttr* rule reports violation for the following pre-defined attributes:

VAL	SUCC	PRED	LEFTOF
RIGHTOF	DELAYED	QUIET	TRANSACTION
ACTIVE	LAST_EVENT	LAST_ACTIVE	

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for a pre-defined attribute <*attr-name*> which is not synthesizable by a synthesis tool:

```
[WARNING] Use of pre-defined attribute '<attr-name>' may not be synthesizable
```

Potential Issues

A violation is reported when one of the above mentioned pre-defined

attribute is used in the design.

Consequences of Not Fixing

The above mentioned pre-defined attributes are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending pre-defined attribute is used. This is un-synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. Avoid these attributes in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
entity test is
end test;

architecture test of test is
  signal sig1 : integer;
begin  -- test

  sig1 <= integer'VAL(0);

end test;
```

In the above example, the PreDefAttr rule reports a violation as the predefined attribute, **VAL**, is used in the design.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

readclock

Unsynthesizable implicit sequential logic: clock read inside always block.

Language

Verilog

Rule Description

The readclock rule flags sequential descriptions where the clock signal is read inside the always construct.

Message Details

The following message appears at the location where the clock signal *<clk-name>* is read in an always construct:

```
Unsynthesizable implicit sequential logic: cannot read clock
'<clk-name>' inside always block
```

Suggested Fix

First determine if you really need to test the value. The fact that the block has already triggered on positive edge implies the clock value, unless you are looking for simulation X states. In that case, consider putting the clock test inside **translate_off** and **translate_on** pragmas so the construct will be ignored in synthesis.

Examples

Example 1

In the following example, the clock signal **clk** is read inside the always construct:

```
always@ (posedge clk)
  if(clk == 1'b1)
    out1 <= in1 & in2;
```

For this example, SpyGlass generates the *readclock* rule message.

Example 2

The *readclock* rule does not flag if different bit-select of clock signal is read

inside **always** construct.

In following example, **bit-select clk[1]** is read inside **always** construct but **bit-select clk[0]** is provided as clock. So the *readclock* rule should not report a violation.

However, if **bit-select clk[0]** is read instead of **clk[1]** inside **always** construct, then the *readclock* rule should report a violation.

```
always_ff @(posedge clk[0], negedge rst)
begin
if (!rst) begin
out <= 1'b0;
end
else if (clk[1]) begin
if (in1) begin
out <= in2;
end
end
end
```

Rule Severity

Warning

ResFunction

Identifies the resolution functions which are not synthesizable

When to Use

Use this rule to identify the resolution functions which are not synthesizable by some synthesis tools.

Description

The *ResFunction* rule reports violations for the resolution functions which are not synthesizable by some synthesis tools.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for the <*func-name*> resolution function which is not synthesizable by a synthesis tool:

`[WARNING] Resolution function <func-name> may not be synthesizable`

Potential Issues

A violation is reported when a resolution function is used in the design.

Consequences of Not Fixing

Resolution functions are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the

location where the offending resolution function is used. This is un-synthesizable by several commercially available synthesis tools.

To resolve the violation, use the **std_logic** type instead of a resolution function.

Example Code and/or Schematic

Consider the following example:

```
entity test is
end test;

architecture test of test is

  function res_func (
    inputs : bit_vector)
    return bit is
  begin
    return '0';
  end res_func;

  signal sig1 : res_func bit;
begin
end test;
```

In the above example, the ResFunction rule reports a violation as the resolution function is used in the design.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

ResetSynthCheck

This rule group checks all synthesis issues related to reset

Rule Description

The ResetSynthCheck rule group checks all synthesis issues related to reset. This rule runs *W442*, *badimplicitSM1*, *badimplicitSM2*, and *badimplicitSM4* rules.

SigVarInit

Identifies the initial values of signals and variables which are not synthesizable

When to Use

Use this rule to identify the initial values which are not synthesizable by some synthesis tools.

Description

The *SigVarInit* rule reports violations for the assignment of initial values to signals and variables, which are not synthesizable by some synthesis tools.

Language

VHDL

Default Weight

5

Parameter(s)

- *ignore_function_init*: Default value is **no**. Set the value of this parameter to **yes**, to not report violations for the functions that consist initializing values.

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for the <*name*> signal or the <*name*> variable whose initial value is not synthesizable by a synthesis tool:

```
[WARNING] Default initial value of '<name>' may be ignored by some synthesis tools
```

Potential Issues

A violation is reported when the initial value is assigned to a signal or variable.

Consequences of Not Fixing

Assignment of initial values to signals and variables is not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the signals or variables have been declared with some initial values. This is un-synthesizable by several commercially available synthesis tools.

If you want to set an initial state, setup the reset connections. If you need initial values for the simulation, enclose such initial value statements in the **synthesis_off/synthesis_on** pragmas. Try to avoid the statements in the synthesizable code since they make you feel that the logic is functioning correctly, whereas in the implementation the logic does not function correctly.

Use the following SpyGlass commands to interpret the VHDL design code enclosed within the **translate_off/translate_on** and the **synthesis_off/synthesis_on** pragmas:

- **set_option hdlin_translate_off_skip_text yes**

  For details on the usage, refer to the *VHDL-specific Options* section in the *Atrenta Console Reference Guide*.

- **set_option hdlin_synthesis_off_skip_text yes**

  For details on the usage, refer to the *VHDL-specific Options* section in the *Atrenta Console Reference Guide*.

Example Code and/or Schematic

Consider the following example:

```
architecture test of test is
  signal sig1 : integer;
begin

  p1: process (sig1)
    variable var1 : integer := 0;  --VIOLATION

  begin
  end process p1;
end test;
```

In the above example, the SigVarInt rule reports a violation as the

variable, **var1**, is assigned an initial value.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

- *Atrenta Console User Guide*: For details on how SpyGlass reads and interprets the Synopsys **translate_off/translate_on** and **synthesis_off/synthesis_on** synthesis pragmas.
- *Atrenta Console Reference Guide*: For details on the display of the inactive code.

SynthIfStmt

Identifies the IF statements which are not synthesizable

When to Use

Use this rule to identify the IF statements which are not synthesizable by some synthesis tools.

Description

The *SynthIfStmt* rule reports violations for the **`if`**, **`if-elsif`**, or **`if-elsif-else`** constructs which are not synthesizable by some synthesis tools.

The rule reports the following constructs, where:

- the **`if`** statement uses one of the following non-synthesizable attributes:

ASCENDING	IMAGE	VALUE
POS	SUCC	PRED
LEFTOF	RIGHTOF	DELAYED
QUIET	TRANSACTION	LAST_EVENT
LAST_ACTIVE	LAST_VALUE	DRIVING
DRIVING_VALUE	SIMPLE_NAME	INSTANCE_NAME
PATH_NAME		

- an asynchronous or synchronous process is followed by an **`else`** or an **`elsif`** statement

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed at the first line of an **if**, **if-elsif**, or **if-elsif-else** construct which is not synthesizable by a synthesis tool:

```
[WARNING] The IF-statement does not conform with any
synthesizable description style
```

Potential Issues

A violation is reported when an IF statement does not confirm with any synthesizable description style.

Consequences of Not Fixing

Such **If** and **If-else-If** statements are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending **if**, **if-elsif** or **if-elsif-else** constructs are used. This is un-synthesizable by several commercially available synthesis tools.

No fix is required in testbenches or simulation models. Recode the code targeted for synthesis to meet the synthesizability guidelines.

Example Code and/or Schematic

Example 1

Consider the following example:

```
PROC_1: PROCESS(clk, rst)
  begin
    if(rst = '0') then q <= '0';
      if(clk'event and clk = '1') then q <= d;
      else q <= '1';
    end if;
end process;
```

In the above example, SpyGlass reports a violation as the process is

followed by the **else** statement.

Example 2

Consider the following example:

```
PROC_2: PROCESS(clk, rst)
  begin
    if(clk'event and clk = '1') then
      if(rst = '0') then q <= '0';
      else q <= d;
      end if;
    else q <= '1';
    end if;
end process;
```

In the above example, SpyGlass reports a violation as the process is followed by the **else** statement.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

UserDefAttr

Identifies the user-defined attributes which are not synthesizable

When to Use

Use this rule to identify the user-defined attributes which are not synthesizable by some synthesis tools.

Description

The *UserDefAttr* rule reports violation for the user-defined attributes which are not synthesizable by some synthesis tools.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for a user-defined attribute:

```
[WARNING] Use of user-defined attribute may not be
synthesizable
```

Potential Issues

A violation is reported when a user-defined attribute is used in the design.

Consequences of Not Fixing

User-defined attributes are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where offending user-defined attribute is used. This is un-synthesizable by

several commercially available synthesis tools.

No fix is required in testbenches or simulation models. Avoid this style in the code targeted for synthesis.

Example Code and/or Schematic

Consider the following example:

```
architecture arc of nand_gate is
component nand_comp
  port(in1,in2: in bit; out1 : out bit);
end component;

type DOUBLE_INT is
    record
     X,Y : INTEGER;
    end record;

attribute PLACEMENT: DOUBLE_INT;
attribute SIZE:DOUBLE_INT;
attribute PLACEMENT of N1: label is (50,45);
attribute SIZE of N1:label is (2,4);
signal PERIMETER:INTEGER;
signal A,B,Z:bit;
begin
  N1: NAND_COMP port map(A,B,Z);
  PERIMETER <= 2 * (N1'SIZE.X + N1'SIZE.Y);
end arc;
```

In the above example, the UserDefAttr rule reports a violation as a user-defined attribute, **PERIMETER**, is used in the design.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

W43

Reports unsynthesizable wait statements

When to Use

Use this rule to identify the wait statements that are not synthesizable.

Description

The W43 rule reports violation for wait statements used in the design.

Following forms of wait statement may not be synthesizable by some synthesis tools:

```
wait;
wait on <sensitivity list>;
wait for <time-expression>;
```

The wait statement of the form, **`wait until`** *<Boolean-expression>*, can be used to infer a clock and hence, it is synthesizable.

The W43 rule checks only for unsynthesizable wait statements.

Language

VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a wait statement is encountered:

```
[WARNING] <stmt> statement may not be synthesizable
```

Where, *<stmt>* refers to a form of wait statement. It can have values such as **Timeout expression in wait** or **Sensitivity list in wait**.

Potential Issues

Violation may arise when an unsynthesizable **wait** statement is encountered in the design.

Consequences of Not Fixing

Some synthesis tools may not be capable of translating wait statements into efficient logic.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the **wait** statement is used.

No fix is required in testbenches or simulation models. In code targeted for synthesis, recode to meet synthesizability guidelines.

Example Code and/or Schematic

Consider the following example in which a wait statement is used in a while loop:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity test2 is
port(
  in1: in std_logic_vector(2 downto 0);
  out1: out std_logic_vector(2 downto 0)
  );
end test2;

architecture behav of test2 is
signal sig1: std_logic_vector(2 downto 0);
begin
  process
  begin
    while(sig1 = "111") loop
      out1 <= in1;
```

```
        sig1 <= "10X";
        wait on in1;
    end loop;
    end process;
end behav;
```

SpyGlass flags the following violation for the above case:

```
Sensitivity list in wait statement may not be synthesizable
```

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

W182c

Identifies the time declarations which are not synthesizable

When to Use

Use this rule to identify the time declarations which are not synthesizable by some synthesis tools.

Description

The *W182c* rule reports the ''**time**'' variable declarations which are not synthesizable by some synthesis tools.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for the **time** declarations which are not synthesizable by a synthesis tool:

```
[ERROR] 'time' declaration are not synthesizable
```

Potential Issues

A violation is reported when a time variable is declared.

Consequences of Not Fixing

The **time** declarations have no physical equivalent. Therefore, such declarations are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location

where the offending **time** data type is used. This is un-synthesizable by several commercially available synthesis tools.

To resolve the violation, remove the **time** declarations from the logic intended for synthesis. Use these in testbenches only.

Example Code and/or Schematic

Consider the following example:

```
module test(in1, in2);
input in1, in2;
time t;
endmodule
```

In the above example, the W182c rule reports a violation as a **time** data type is used.

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

None

W182g

Identifies the tri0 net declarations which are not synthesizable

When to Use

Use this rule to identify the **tri0** net declarations which are not synthesizable by some synthesis tools.

Description

The *W182g* rule reports violations for the **tri0** net declarations used in the design.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for a **tri0** declaration which is not synthesizable by a synthesis tool:

[Error] `'tri0' net types may not be synthesizable`

Potential Issues

A violation is reported when a `tri0` net is used in the design.

Consequences of Not Fixing

The **tri0** and **tri1** net declarations represent connections with resistive pull-down or pull-up. These nets are useful in developing simulation models, but they do not have an unambiguous physical counterpart because mapping depends upon the target technology. For example, some

technologies may not support tristate operations.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending "tri0" is declared. This is un-synthesizable by several commercially available synthesis tools.

To resolve the violation, replace the **tri** declarations with the standard signal declarations and instantiate pull-up or pull-down cells from the target technology library, as required.

Example Code and/or Schematic

Consider the following example:

```
module test (y,s0,d1,d0);
input s0, d1, d0;
output y;

tri0 y;

assign y = s0 ? d0 : d1;
endmodule
```

In the above example, the W182g rule reports a violation as a **tri0** net is used in the design.

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

None

W182h

Reports tri1 net declarations that are not synthesizable

When to Use

Use this rule to detect tri1 net declarations that are not synthesizable

Description

The *W182h* rule reports the **tri1** net declarations even if the net is not used in the design.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a **tri1** declaration is encountered.

[ERROR] `'tri1' net types may not be synthesizable`

Potential Issues

A violation is reported when a **tri1** net is used in the design.

Consequences of Not Fixing

The **tri0** and **tri1** declarations represent connections with resistive pull-down or pull-up. These nets may not be synthesizable by some synthesis tools and are useful only in developing simulation models. They do not have an unambiguous physical counterpart because mapping depends upon the target technology. For example, some technologies may not

support tristate operations.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending "tri1" has been declared. This is un-synthesizable by several commercially available synthesis tools

Replace the **tri** declarations with standard signal declarations and instantiate pull-up or pull-down cells from the target technology library, as required.

Example Code and/or Schematic

Consider the following example:

```
module test(y,s0,d1,d0);
input s0, d1, d0;
output y;
tri1 y;

  assign y = s0 ? d0 : d1;
endmodule
```

In the above example, the W182h rule reports a violation as the **tri1** net is used.

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

None

W182k

Reports trireg declarations that are not synthesizable

When to Use

Detect trireg declarations that are not synthesizable.

Description

The *W182k* rule reports **trireg** register declarations.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a **trireg** declaration is encountered.

```
[ERROR] 'trireg' net types are not synthesizable
```

Potential Issues

A violation is reported when a **trireg** net is used in the design.

Consequences of Not Fixing

The **trireg** declarations represent charge storage that does not have a clear physical interpretation in synthesis. Such descriptions are useful only in developing simulation models.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending **trireg** is declared. This is un-synthesizable by

several commercially available synthesis tools.

Do not try to infer charge storage elements. Use library memory models if appropriate.

Example Code and/or Schematic

Consider the following example:

```
module test(y,s0,d1,d0);
input s0, d1, d0;
output y;

trireg y;

assign y = s0 ? d0 : d1;
endmodule
```

In the above example, the W182k rule reports a violation as a **trireg** net, **y**, is used in the design.

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

None

W182n

Reports MOS switches, such as cmos, pmos, and nmos, that are not synthesizable

When to Use

Detects MOS switches that are not synthesizable.

Description

The *W182n* rule reports MOS switches, such as CMOS, PMOS, NMOS, used in the design. These switches should be used for modeling only and are not synthesizable by most synthesis tools.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a switch declaration <*switch-name*> is encountered.

`[ERROR] Switch '<switch-name>' is not synthesizable`

Potential Issues

A violation is reported when a MOS switch is used in the design.

Consequences of Not Fixing

Except for custom or analog design, transistor-level design is generally discouraged because behavior and timing are difficult to predict under all possible circumstances.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending switches (CMOS/PMOS/NMOS) are declared. This is un-synthesizable by several commercially available synthesis tools.

Do not try to infer switch elements. Use predefined and characterized library cells that have well-defined characteristics and are silicon-proven.

Example Code and/or Schematic

Consider the following example:

```
module test(out, in1, in2);
input in1,in2;
output out;
wire out1,out2;
wire n;
cmos (out,in1,in2,n);
pmos (out1,in1,in2);
nmos (out2,in1,in2);
endmodule
```

In the above example, the W182n rule reports a violation as MOS switches are used in the design.

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

None

W213

Reports PLI tasks or functions that are not synthesizable

When to Use

Use this rule to identify PLI tasks and functions used in the design.

Description

The *W213* rule reports PLI tasks and functions used in the design.

Language

Verilog

Default Weight

5

Parameter(s)

ignore_pli_tasks_and_functions: By default, the **`ignore_pli_tasks_and_functions`** parameter is not set. Set this parameter to a comma-separated list of PLI tasks or functions that should be ignored by the *W213* rule.

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a PLI task or a function <*name*> is encountered.

```
[WARNING] PLI Task/Function '<name>' is not synthesizable
```

Potential Issues

A violation is reported when a PLI task or function is used in the design.

Consequences of Not Fixing

The PLI tasks or functions, such as **`$display`**, have no physical meaning and therefore are not synthesizable.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending PLI Task/Function is used. This is un-synthesizable by several commercially available synthesis tools.

No fix is required if these functions appear in testbenches or simulation models. If you need to include these functions in RTL for debug purposes, surround them with the **translate_off** and **translate_on** pragmas for your target synthesis tool.

Example Code and/or Schematic

Consider the following example:

```
module test (in1, clk);

input in1, clk;
always @ (clk)
  $display ("Value of in1 %b\n", in1);

endmodule
```

In the above example, the W213 rule reports a violation as the PLI task, **$display**, is used in the design.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

W218

Reports multi-bit signals used in sensitivity list

When to Use

Use this rule to identify multi-bit signals used in the sensitivity list.

Description

The *W218* rule reports event expressions that check for an edge on a multi-bit signal.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where an edge specification on a multi-bit signal is encountered.

```
[WARNING] Edge specification should not be used for a multibit expression: '<multi-bit-expressions>'
```

Potential Issues

A violation is reported when an edge specification is used on a multi-bit expression.

Consequences of Not Fixing

Edge specifications for multi-bit expression is semantically incorrect. In such cases, only the changes on least significant bit are important. Also the direction of an edge on a multi-bit signal is not uniquely defined.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window shows the sensitivity list, where the multi-bit signal is used. The sensitivity list confirms that more than one bit of the signal is participating.

To fix the violation, use the edge expression on the appropriate bit of the multi-bit signal.

Example Code and/or Schematic

Consider the following example:

```
module test1(in1,clk,out1);
input [2:0] in1, clk;
output [2:0] out1;

reg [2:0] out1;

always @(posedge clk)
out1 = in1;

endmodule
```

In the above example, the *W218* rule reports a violation as the event expression checks for the edge on a multi-bit signal.

Default Severity Label

Warning

Rule Group

Synthesis, Event

Reports and Related Files

None

W239

Reports hierarchical references that are not synthesizable

When to Use

Use this rule to identify hierarchical references used in the design.

Rule Description

The *W239* rule reports hierarchical references. Hierarchical references, such as **`top.abx.clk`** , are a useful way to probe design behavior during debug.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a hierarchical reference is encountered.

```
[WARNING] Hierarchical references may not be synthesizable
```

Potential Issues

A violation is reported when a hierarchical reference is used in the design.

Consequences of Not Fixing

Synthesis tools, in general, do not create connections corresponding to these references.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the

hierarchical references used in the design.

If you are using the reference for simulation, enclose it in code that can be disabled. This enables you to verify that functional behavior of the synthesis logic does not depend on this reference.

If you actually need to make connections in the synthesis logic, do so through the conventional approach, that is, add ports to lower level modules and make connections through those ports.

Example Code and/or Schematic

Consider the following example:

```
module top(output [3:0] w2);
assign w2 = temp.w1;

endmodule
module temp();
wire [3:0] w1;
endmodule
```

In the above example, the W239 rule reports a violation as a hierarchical reference, **`temp.w1`**, is used in the design.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

W250

Reports disable statements that are not synthesizable

When to Use

Use this rule to identify **disable** statements used in the design.

Description

The *W250* rule reports the **disable** statements.

Language

Verilog

Default Weight

5

Parameter(s)

avoid_synth_disable: Default value is **no**. Set this parameter to **yes**, to enable the rule to not report violations for synthesizable disable constructs for the following cases:

- Tasks
- Blocks
  - Sequential blocks (whose start-end is defined by begin-end)

NOTE: *For Parallel blocks (whose start-end is defined by fork-join), the rule does not waive violations as they are considered as non-synthesizable.*

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a **disable** statement is encountered.

```
[WARNING] Disable statement may not be synthesizable
```

Potential Issues

A violation is reported when a **disable** statement is used in the design.

Consequences of Not Fixing

The **disable** statements are useful in behavioral modeling but have no physical counterpart because they represent an arbitrary jump out of loop. Therefore, such descriptions are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending **disable** statement is used. This is un-synthesizable by several commercially available synthesis tools.

No fix required if this is in a simulation mode or testbench. If this is intended to be synthesizable, you should rewrite the code to avoid the **disable** statement.

Example Code and/or Schematic

Consider the following example:

```
module case_250 (clk1,in,cntr,out);

  input clk1,in;
  input [1:0] cntr;
  output out;

  initial begin
   f_190;
  end

  initial begin
   #10 disable f_190;
  end

task f_190;
reg clk, f_191;
reg[1:0] cntr_f;
  if ( clk == 1'b0) f_191 = cntr_f[1];
  else  f_191 = cntr_f[0];
endtask

endmodule
```

In the above example, the rule reports a violation as a disable statement is used in the design.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

W293

Reports functions that return real values

When to Use

Use this rule to identify functions that return **`real`** values.

Description

The *W293* rule reports functions that return real values.

Language

Verilog, VHDL

Parameters

None

Constraints

None

Messages and Suggested Fix

Verilog

The following message appears at the start of a function description that returns a real value.

```
[WARNING] Function returns a real value which is not synthesizable
```

Potential Issues

Violation may arise when a function returns a real value.

Consequences of Not Fixing

Objects with real values have no physical equivalent and therefore are not synthesizable.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line of function definition, which returns a real value.

VHDL

The following message appears at the start of a subprogram <*subp-name*> description that returns a real value:

`[WARNING] The <subp-name> returns a real value which may not be synthesizable`

Where <*subp-name*> can be **`function`** or **`procedure`**.

Potential Issues

Violation may arise when a subprogram returns a real value.

Consequences of Not Fixing

Objects with real values have no physical equivalent and therefore are not synthesizable.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line of function definition, which returns a real value.

No fix required if this occurs in a testbench. If this is targeted for synthesizable RTL, you will need to explicitly implement floating point logic.

Example Code and/or Schematic

Consider the following example:

```
entity ent is
  port (
         in1 : in real;
         out1: out real
       );
end ent;

architecture arc of ent is
function myFunc (number: real) return real is
begin
  return number * 3.142857;
end myFunc;
begin
process
begin
```

```
  out1 <= myFunc(in1);
end process;
end arc;
```

In the above example, the *W293* rule reports a violation as the function, **`myFunc`**, returns a real value, which is unsynthesizable.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

W294

Reports real variables that are unsynthesizable

When to Use

Use this rule to identify real variables, which are unsynthesizable.

Rule Description

The *W294* rule reports **`real`** variables used in the design.

Language

Verilog

Parameters

None

Constraints

None

Messages and Suggested Fix

The following message appears at the location where a variable <*var-name*> of type **`real`** is encountered.

```
[WARNING] Real variable '<var-name>' is not synthesizable
```

Potential Issues

Violation may arise when a design has a real variable.

Consequences of Not Fixing

Objects with real values have no physical equivalent and therefore may not be synthesizable by some synthesis tools. Real variables should only be used in testbenches and simulation models.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending real variable is declared. This is un-synthesizable by several commercially available synthesis tools.

If targeting synthesizable logic, use integer or **`reg`** variables to fix the

violation.

Example Code and/or Schematic

Consider the following example:

```
module test(in1, in2, z);
input in1;
input in2;
output z;
real r = 0.025;   //VIOLATION
assign z = r + in1 + in2;
endmodule
```

In the above example, the W294 rule reports a violation as a real variable, **r**, is used in the design.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

None

W295

Reports event variables that are not synthesizable

When to Use

Use this rule to identify **`event`** variables used in the design.

Rule Description

The *W295* rule reports **`event`** variables.

Language

Verilog

Parameters

None

Constraints

None

Messages and Suggested Fix

The following message appears at the location where an **`event`** variable is declared.

```
[ERROR] Event variable may not be synthesizable
```

Potential Issues

Violation may arise when an event variable is encountered in a design.

Consequences of Not Fixing

Event variables have no physical equivalent and therefore are not be synthesizable by some synthesis tools. However, **`event`** constructs may appear in testbenches or system-level models.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line of event signal declaration.

If you intend to make the code synthesizable, trigger on signals.

Example Code and/or Schematic

Example of Event Variable Declared and Used

In the following example, an **event** variable **test** is declared and used.

```
module test (clk1, in, cntr, out);
  input clk1, in;
  input [1:0] cntr;
  output out;

  event test;
  reg a_1, a_2;

  always @(clk1 or test)
    a_2 = in;

  always @(test)
    a_1 <= in;

  assign out = a_1 + a_2;
endmodule
```

Example of Event Variable Declared But Not Used

In the following example, an **event** variable **in2** is declared but is not used.

```
module test(in1, out1);
  input in1;
  event in2;
  output out1;

  reg out1;

  always @(in1)
    begin
      #10 out1 = in1;
    end
endmodule
```

Example of Event Variable Declared and Triggered

In the following example, an **event** variable **in2** is declared and triggered.

```
module test(in1, out1);
  input in1;
  event in2;
  output out1;

  reg out1;

  always @(in1)
    begin
      #10 out1 = in1;
      ->in2;
    end
endmodule
```

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

None

W339

Identity operators and non-constant divisors are not synthesizable.

Language

Verilog

Rule Description

The W339 rule runs the *W339a* rule.

W339a

Case equal operator (===) and case not equal (!==) operators may not be synthesizable

Language

Verilog

Rule Description

The W339a rule flags case equality operators — case equal (**===**) and case not equal (**!==**) operators.

The case equality operators compare non-physical values (**X**) as well as real values, and are frequently used in simulation to check for unknown states. However, they have no physical equivalent. Some synthesis tools may be able to handle these operators but reduce them to their non-case equivalents (for example **===** gets changed to **==**). This may lead to mismatches between pre and post synthesis behavior.

Message Details

The following message appears at the location where a case equality operator is encountered:

```
Operator '<operator-name>' should be avoided in synthesis logic
```

Rule Severity

Warning

Suggested Fix

First, make sure that your synthesis logic does not depend on the simulation checks. Then, bracket the simulation checks inside **translate_off** / **translate_on** pragmas so that they are ignored in synthesis.

Examples

Example of Case Equal Operator

Consider the following example that uses case equal (**===**) operator:

```
module MEM(clk, address, enable, data);
  input [7:0] data;
  input enable, clk;
  output [7:0] address;

  reg [7:0] address;

  always @(posedge clk)
    begin
      if (enable === 1)
        address <= data;
      else
        address <= 8'h00;
    end
endmodule
```

SpyGlass generates the W339a rule message for every use of case equality operators.

Example of Case Not Equal Operator

In the following example, the case not equal operator (**!==**) is used:

```
module test(in1, in2, clk, out1);
  input [1:0] in1, in2;
  input clk;
  output [1:0]out1;

  reg [1:0]out1;

  wire [1:0] count;

  assign count = (in1 !== 1'b1)? 2'b10 : 2'b01;

  always @(posedge clk)
    begin
      out1 <= (in2 !== count) ? in1 : in2;
    end
endmodule
```

SpyGlass generates the W339a rule message for every use of case equality

operators.

W430

The "initial" statement is not synthesizable

When to Use

Use this rule to identify the unsynthesizable **initial** statements.

Description

The W430 rule reports violation for **initial** constructs used in the design.

Default Weight

5

Language

Verilog

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where an **initial** construct is encountered:

[WARNING] Initial statement is not synthesizable

Potential Issues

Violation may arise when an **initial** construct is used in the design.

Consequences of Not Fixing

The **initial** constructs have no physical equivalent. Therefore, such constructs are unsynthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where offending initial construct are used. This is un-synthesizable by

several commercially available synthesis tools.

No fix required if the statement is used inside a testbench. In general you should not use initial blocks in code targeted for synthesis. Use physically realizable reset logic to initialize code instead. If you must include an initial block in synthesizable code, bracket it in **translate_off** and **translate_on** pragmas to disable the code in synthesis.

Example Code and/or Schematic

Consider the following example:

```
module W430_mod1(in1,clk,out1,out2);
  input in1,clk;
  output reg out1,out2;

  initial
  begin
    out1 = 1'b0;
    out2 = 1'b0;
  end

endmodule
```

In the above example, the W430 rule reports a violation as an initial statement is used in the design.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

W442

This rule group checks all synthesis issues related to reset

Rule Description

The W442 rule group checks all synthesis issues related to reset. This rule runs *W442a*, *W442b*, *W442c*, and *W442f* rules.

W442a

Ensure that for unsynthesizable reset sequence, first statement in the block must be an if statement

When to Use

Use this rule to identify the asynchronous **reset** sequence where the first statement is not an **if** statement.

Description

The W442a rule reports violation for asynchronous reset sequences where the first statement is not an **if** statement.

The first statement after the sensitivity list on the **always** construct must be an **if** statement using the reset signal.

Rule Exceptions

This is not a requirement inside testbenches or other simulation only code.

Default Weight

5

Language

Verilog

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location of an **always** construct where the first statement is not an **if** statement using a reset signal:

```
[ERROR] Asynchronous reset/set always block has missing 'if' statement at the top level
```

Potential Issues

Violation may arise when an asynchronous reset/set always block has missing **if** statement at the top level.

Consequences of Not Fixing

In general, synthesis tools expect that the first statement inside an asynchronously reset block is an **if** statement. Else, the RTL may become unsynthesizable. Therefore, if code is meant for synthesis, this style should not be used.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the start line of the **always** block in which the first statement is not an **if** statement.
To fix the violation, reorganize the block into one or more blocks, such that any asynchronous reset block meets the requirement.

Example Code and/or Schematic

Consider the following example where the first statement in the **always** construct describing an asynchronous reset sequence, is not an **if** statement:

```
module DFF(D, clk, R, Q);
  output Q;
  input D, clk, R;
  reg Q;

  reg d1, d2;
  always@(posedge clk or posedge R)

    begin
      d2 = d1;
      if(R) Q = 0;
      else Q = D;
    end
endmodule
```

In the above example, the W442a rule reports a violation as the asynchronously reset/set **always** block has a missing **if** statement.

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

W442b

Ensure that for unsynthesizable reset sequence, reset condition is not too complex

When to Use

Use this rule to identify unsynthesizable and complex reset sequence.

Description

The W442b rule reports violation for complex reset sequences.

The rule also reports violation for reset signal when it is compared either with any other signal or variable.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a complex reset sequence is encountered:

```
[ERROR] In asynchronous reset/set always block, comparison is being made to non-constant expression ( <expr> ) in reset/set condition
```

Potential Issues

Violation may arise when a reset signal is compared with any other signal or variable or a non-constant expression.

Consequences of Not Fixing

Only certain forms of asynchronous reset descriptions are recognized by

the synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the asynchronous reset signal is compared with a non-constant expression.

Scroll through the HDL to search for the value of the signal used in comparison of the asynchronous reset signal.

To fix the violation, precompute the reset condition and then use that value as the actual reset.

Example Code and/or Schematic

Example 1

Consider the following example:

```
module mod(in1, clk, reset, set, out1);
  input [1:0] in1;
  input clk, reset, set;
  output [1:0] out1;
  reg [1:0] out1;

  always @(posedge clk or posedge reset)
    begin
      if(reset == !set)
        out1 <= 2'b00;
      else
        out1 <= in1;
    end
endmodule
```

In the above example, the W442b rule reports a violation as an asynchronous **`reset`** signal is compared to a non-constant expression, **`!set`**, in an **`always`** block.‘

Example 2

Consider the following example:

```
always @(posedge reset or negedge clk)
 begin
```

```
    if (reset != b)
      q = 1'b0;
    else
      q = d;
 end
```

In the above example, the W442b rule report violations as the asynchronous **reset** signal is compared with the other signal **b**.

To fix the violation, precompute the reset condition and then use the resultant value as the actual reset as shown below:

```
assign actual_reset = (reset != b);
always @(posedge actual_reset or negedge clk)
 begin
  if (actual_reset)
    q = 1'b0;
  else
    q = d;
 end
```

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

W442c

Ensure that the unsynthesizable reset sequence are modified only by ! or ~ in the if condition

When to Use

Use this rule to identify the unsynthesizable reset sequence that are modified by anything other than **!** or **~** in the **if** condition.

Description

The W442c rule reports violation for reset sequences that are modified by operators other than logical inverse (**!**) and bit-wise inverse (**~**) operators.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a reset signal is being modified by an operator other than logical inverse (**!**) and bit-wise inverse (**~**) operators:

```
[ERROR] Asynchronous reset/set always block may have the reset/set condition only as a simple identifier or its negation (! or ~)
```

Potential Issues

Violation may arise when a reset signal is being modified by an operator other than logical inverse (**!**) and bit-wise inverse (**~**) operators.

Consequences of Not Fixing

You can modify the reset signals using only logical inverse (**!**) and bit-wise inverse (**~**) operators in the **if** statement in the reset sequences. Modification by all other types of operators is not synthesizable.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the asynchronous reset signal is modified by anything other than **!** or **~**. Scroll through the HDL for the usage of asynchronous reset signal.

To fix the violation, precompute the value you want to use in the condition before using it.

Example Code and/or Schematic

Consider the following example:

```
module test(reset,q);
output q;
input reset;
reg q,clk,d;

always @(posedge reset or negedge clk)
  begin
    if (&reset)
      q = 1'b0;
    else
      q = d;
  end
endmodule
```

In the above example, the W442c rule reports a violation as an asynchronous **reset**/**set** always block may have the **reset**/**set** condition only as a simple identifier or its negation (**!** or **~**).

To fix the above violation, precompute the value you want to use in the condition and then use that value as shown below

```
...
assign actual_reset = &reset;
always @(posedge actual_reset or negedge clk)
  begin
```

```
    if (actual_reset)
      q = 1'b0;
    else
      q = d;
  end
...
```

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

No related reports and files.

W442f

Ensure that the unsynthesizable reset sequence is compared using only == and != binary operator in the if condition

When to Use

Use this rule to identify the reset sequences where reset signal is being compared using an operator other than the binary equal (**==**) and not equals (**!=**) operator.

Description

The *W442f* rule reports violation for reset sequences where reset signal is being compared using an operator other than the binary equal (**==**) and not equals (**!=**) operator.

Language

Verilog

Default Weight

5

Rule Exceptions

The *W442f* rule does not report a violation for **"==="** or **"!=="** used inside if condition, because these operators are treated as **"=="** or **"!="**, respectively.

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a reset sequence uses an operator other than the binary equal (**==**) or not equals (**!=**) operator in the **if** statement:

```
[ERROR] Only '==' and '!=' binary operators are allowed in
```

validation of the asynchronous reset/set condition

Potential Issues

Violation may arise when a reset sequence uses an operator other than the binary equal (**==**) or not equals (**!=**) operator in the **if** statement.

Consequences of Not Fixing

Reset signal that are compared using an operator other than the binary equal (**==**) and not equals (**!=**) operator are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the binary operator other than **==** or **!=** is used in the asynchronous reset signal expression. Scroll through the HDL for the type of binary expression used inside the asynchronous reset expression.
To fix the violation, either precompute the reset condition and use the resultant value as the actual reset or use a nested **if** condition.

Example Code and/or Schematic

Consider the following example:

```
module test(q,reset,set);
input reset,set;
output q;
reg q,clk;
always @(posedge clk or negedge reset)
  begin
    if (reset & set)
      q = 0;
  end
endmodule
```

In the above example, the W442f rule reports a violation as only '**==**' and '**!=**' binary operators are allowed in validation of the asynchronous reset/set condition.
To fix the violation, you can use any one of the following methods:

- Precompute the reset condition and use that value as the actual reset. For example:

```
assign actual_reset = reset & set;
  always @(posedge actual_reset or negedge clk)
    begin
      if (actual_reset)
        q = 1'b0;
      else
        q = d;
    end
```

- Use a nested **if** condition. For example:

```
always @(posedge reset or negedge clk)
 begin
  if (reset )
   if ( set )
    q = 1'b0;
   else
    q = d;
  else
   q = d;
 end
```

Default Severity Label

Error

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

W464

Ensure that the unrecognized synthesis directive is not used in the design

When to Use

Use this rule to identify the unsupported pragma or synthesis directives used in the design.

Rule Description

Verilog

For Verilog designs, the W464 rule reports all pragma directives of specified prefix in the design.

By default, SpyGlass flags pragma directives of prefix type **synthesis** if the value **synthesis** is also passed using the SpyGlass **set_option pragma <values>** command while invoking SpyGlass. Thus, the following pragma directives are flagged:

```
//synthesis atrenta
//synthesis translate_off
//synthesis synthesis_on
```

To specify a different prefix type, first replace the default prefix value (**synthesis**) given in the W464 rule registration of the Lint-Verilog ruledeck file (**verilint.pl**) with your prefix type:

```
“prefix notmatch synthesis”
```

For example, to flag all pragma directives of prefix type **synopsys**, modify the value as follows:

```
“prefix notmatch synopsys”
```

Then, specify the SpyGlass **set_option pragma <values>** command with the same value (**synopsys**) while invoking SpyGlass.

VHDL

For VHDL designs, all Synopsys synthesis directives except the following are reported:

dc_script_begin	dc_script_end	translate_off
translate_on	synthesis_off	synthesis_on
resolution_method	built_in	map_to_entity
return_port_name	label	label_applies_to
map_to_operator	infer_mux	coverage_off
coverage_on		

Rule Exceptions

The W464 rule does not report a violation, if you do not specify the **`set_option pragma <values>`** command with the same value that exists in the Lint-Verilog ruledeck file.

Language

Verilog, VHDL

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message appears at the location where a pragma directive <*directive*> of the specified prefix type is encountered:

```
[WARNING] Synthesis directive '<directive>' is not recognized
```

Potential Issues

Violation may arise when a pragma directive of the specified prefix type is encountered.

Consequences of Not Fixing

If **synthesis_off** and **synthesis_on** directives are used within an expression, they have the potential to create incorrect logic by the Synopsys tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending unrecognized synthesis directive are used.

To fix the violation, Synopsys recommends use of **translate_off** and **translate_on** directives against **synthesis_off** and **synthesis_on** directives.

VHDL

The following message appears at the location where an unsupported synthesis directive is encountered:

```
[WARNING] Synthesis directive '<directive>' is not recognized
```

Potential Issues

Violation may arise when an unsupported synthesis directive is encountered.

Consequences of Not Fixing

If **synthesis_off** and **synthesis_on** directives are used within an expression, they have the potential to create incorrect logic by the Synopsys tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending unrecognized synthesis directive is used.

To fix the violation, Synopsys recommends use of **translate_off** and **translate_on** directives against **synthesis_off** and **synthesis_on** directives.

Example Code and/or Schematic

Verilog

Consider the following example:

```
module W464_mod1(inp, outp, sel);
  input [3:0] inp;
```

```
  input [1:0] sel;
  output outp;
  reg outp;

  always @(sel)
  begin
  //synthesis translate on          //VIOLATION
    case(sel)
      2'b00 : outp = inp[0];
      2'b01 : outp = inp[1];
      2'b10 : outp = inp[2];
      2'b11 : outp = inp[3];
    endcase
  end

endmodule
```

In the above example, when you specify **`set_option pragma synthesis`** in the project file, the W464 rule reports the following violation because an unrecognized synthesis directive is present in the design.

```
Synthesis directive 'synthesis' is not recognized
```

VHDL

Consider the following example:

```
library ieee;
use IEEE.std_logic_1164.all;

entity test1 is
port (i : in std_logic;
      o : out std_logic);
end;

-- synopsys synth_off
ARCHITECTURE arc_test1 OF test1 IS
BEGIN
   o <= i;
END arc_test1;
```

The W464 rule reports the following violation message for the above example:

```
Synthesis directive 'synopsys synth_off' is not recognized
```

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

No related reports and files.

W496a

Reports comparison to a tristate in a condition expression

When to Use

Use this rule to identify the comparisons to a tristate value in the condition expressions.

Description

The W496a rule reports violation for comparisons to tristate signals in **`if`** statement conditions and conditional statements.

NOTE: *The W496a rule supports generate-if, generate-for, and generate-case blocks.*

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a comparison (using operator <*op-name*>) with tristate value <*value*> is encountered in a control expression:

`[WARNING] Comparison (<op-name>) to tristate value (<value>) is treated as false in synthesis`

Potential Issues

Violation may arise when a tristate value is compared in a conditional expression.

Consequences of Not Fixing

Comparison to a tristate value does not have a physical equivalent in

synthesis and always defaults to a false value.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where a comparison with a tristate value is encountered.

To fix the violation, avoid making such comparisons in synthesizable code. If the check is being made for simulation reasons, bracket it in **translate_off** and **translate_on** statements.

Example Code and/or Schematic

Example 1

Consider the following example of an if Statement Condition:

```
always @ (en1 or in1 or in2)
  if (en1 == 2'b1z)
    out <= in1;
  else
  out <= in2;
```

In the above example, the signal, **en1**, is being compared with a tristate value, **2'b1z**, in the **if** statement condition. Also, the W496a rule reports the following violation message for this example:

```
Comparison (==) to tristate value (1z) is treated as false in synthesis
```

Example 2

Consider the following example of a conditional statement.

```
always @ (en1 or in1 or in2)
  out <= (en1 == 1'bz) ? in1 : in2;
```

In the above example, the conditional statement condition, **en1**, is being compared with a tristate value, **1'bz**. Also, the W496a rule reports the following violation message for this example:

```
Comparison (==) to tristate value (z) is treated as false in synthesis
```

Example 3

Consider the following example:

```
module test(out);
output out;
reg en1,out;
always @ (en1)
if (en1 == 2'b1z)
out = 2'b10;
else
out = 2'b11;
endmodule
```

In the above example, the W496a rule reports a violation message as the comparison to tristate value, **1z**, is treated as false in synthesis.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

W496b

Reports comparison to a tristate in a case statement

When to Use

Use this rule to identify the comparisons to a tristate value in a case statement.

Description

The W496b rule reports violation for comparisons to tristate signals in case construct control expressions.

NOTE: *The W496b rule supports generate-case block.*

Rule Exceptions

The W496b rule ignores **casex** and **casez** constructs.

Language

Verilog, VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a comparison with tristate value <*value*> is encountered in a case construct control expression <*expr*>:

```
[WARNING] Case comparison of expression: "<expr>" to tristate
value: '<value>' is treated as false in synthesis
```

Potential Issues

Comparison to a tristate value does not have a physical equivalent in

synthesis and always defaults to a false value.

Consequences of Not Fixing

Comparison to a tristate value does not have a physical equivalent in synthesis. Synthesis tools typically default the result of the comparison to a false value.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where a tristate value is used in a case label.

To fix the violation, avoid making such comparisons in synthesizable code. If the check is being made for simulation reasons, bracket it in **`translate_off`** and **`translate_on`** statements.

Example Code and/or Schematic

Example 1

In the following example, case construct selector **`sel`** is being compared with a tristate value **`2'bzz`**:

```
case(sel)
  2'b00 : outp = inp[0];
  2'b01 : outp = inp[1];
  2'bzz : outp = inp[2];
default : outp = inp[3];
endcase
```

For this example, SpyGlass generates the following message:

```
Case comparison of expression: "sel" to tristate value: 'zz' is
treated as false in synthesis
```

Example 2

Consider the following example:

```
module top(out,in1,in2);
output out;
reg out,sel;
input in1,in2;

     always @(sel)
     case (sel)
```

```
        1'b0 : out = in1;
        1'bz : out = in2;
        endcase

    endmodule
```

In the above example, the W496b rule reports a violation message as the case comparison of expression to tristate value, **z**, is treated as false in synthesis.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

W503

An event variable is never triggered

Language

Verilog

Rule Description

The W503 rule flags **`event`** variables that are never triggered.

Event declarations that are never triggered are redundant.

NOTE: *The W503 rule also grouped under the Event Rules group.*

Message Details

The following message appears at the location of an event declaration *<event-name>* that is never triggered:

```
event '<event-name>' is declared but not used
```

Rule Severity

Warning

Suggested Fix

This may be a left-over from earlier debug. Suggest you delete the declaration to reduce the number of spurious errors in rule-checking.

WhileInSubProg

Reports unsynthesizable While statements used inside subprograms

When to Use

Use this rule to identify the unsynthesizable **`while`** statements used in subprograms

Description

The WhileInSubProg rule flags the **`while`** constructs used in sub-program descriptions.

Default Weight

5

Language

VHDL

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a **`while`** statement *<construct-name>* is encountered in a sub-program description:

```
[WARNING] <construct-name> statement inside a subprogram body may be unsynthesizable
```

Potential Issues

A violation is reported when a while statement is used inside a sub-program.

Consequences of Not Fixing

Such descriptions are not synthesizable by some synthesis tools.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location where the offending **while** statement has been used inside subprogram.

No fix is required in testbenches or simulation models. In code targeted for synthesis, use **for**-loops if possible.

Synopsys recommends use of **translate_off/translate_on** directives against **synthesis_off/synthesis_on** directives. If **synthesis_off/synthesis_on** directives are used within an expression, they have the potential to create

Example Code and/or Schematic

Consider the following example:

```
entity e is
end e;
architecture arc of e is
function func (signal k : integer) return integer is

  variable v1,v2,v3 : integer:=7;
begin
  while v2 < 7 loop
  while v1 < 10 loop
     v1 := v1 + 1;
     v2 := k + 1;
  end loop;
  end loop;

return v2;
end func;
signal s1,s2 : integer;
begin
  s1 <= func(s2);
end arc;
```

In the above example, the *WhileInSubProg* rule reports a violation as a while statement is used inside a subprog.

Default Severity Label

Warning

Rule Group

Synthesis

Reports and Related Files

No related reports or files.

Expression Rules

The SpyGlass lint product provides the following expression related rules:

Rule	Flags...
W116	(Verilog) Unequal length operands in bitwise logical/ arithmetic/ ternary operator (VHDL) Unequal length operands in bitwise logical/ arithmetic/ relational operator
W159	Control expressions that evaluate to a constant
W180	Constant value specifications where the specified constant value is narrower than the specified width
W224	Multi-bit expressions found where single-bit expressions were expected
W289	**`real`** operands used in logical comparisons
W292	Logical comparison operations on real type operands
W341	Assignments to constants where the size of the assigned value is narrower than the constant and the high-order bit/byte are zero
W342	Assignments to constants where the size of the assigned value is narrower than the constant and the high-order bit/byte are X
W343	Assignments to constants where the size of the assigned value is narrower than the constant and the high-order bit/byte are Z
W362	Unequal widths in arithmetic comparison operations
W443	Based numbers that contain the unknown character (**`X`**)
W444	All occurrences of the high impedance character (Z) or ? in the design
W467	Based numbers that contain the don't care character (?)
W486	Shift operation overflows
W490	Constant control expressions or sub-expressions
W491	`case` clause condition constants that are ?-extended
W561	Zero-width based numbers
W563	Unary reduction operations on single-bit expressions
W575	Logical **`not`** operators used on vector signals
W576	Logical operations performed on vector signals

W116

Identifies the unequal length operands in the bit-wise logical, arithmetic, and ternary operators

When to Use

Use this rule to identify bit-width mismatch between operands of the bit-wise logical, arithmetic, and ternary operators.

Description

Verilog

The *W116* rule flags bit-width mismatch between operands of bit-wise logical, arithmetic, or ternary operators.

The *W116* rule does not report a violation when two static expressions are present on both the LHS and the RHS.

Following is the list of operators covered under the W116 rule:

Arithmetic Operators	
Subtraction (-)	Addition (+)
Multiplication (*)	Division (/)
Modulus (%)	
Bit-wise Operators	
bit-wise xor (^)	bit-wise negation (~)
bit-wise and (&)	bit-wis e or (|)
Ternary Operator (? :)	

NOTE: *If the count of operators is more than 500 in an expression, then the expression is ignored for rule checking.*

Width Calculation for Verilog

If you set the value of the *nocheckoverflow* parameter to **`yes`** or **`W116`**, the W116 rule checks the bit-width as per the LRM, as shown in the following table:

Operator	LRM width	Normal width
+, -	Max (LHS width, RHS width)	Max (LHS width, RHS width)
*	Max (LHS width, RHS width)	LHS width + RHS width
/	Max (LHS width, RHS width)	LHS width
%	Max (LHS width, RHS width)	RHS (non-static) - RHS width RHS (static) - width of (RHS Value -1)

For constants, the natural width is considered.

NOTE: *For new width related changes, refer to the New Width Flow Application Note.*

NOTE: *The W116 rule does not check for expressions in signal or variable indexes, such as bit-select expression.*

NOTE: *This rule does not report a violation for integer variable expressions.*

VHDL

The W116 rule flags bit-width mismatch between LHS and RHS expressions of bit-wise logical, arithmetic, or relational operations.

The W116 rule also checks the range in case of range-constrained integers, instead of calculating the bit-width.

Following is the list of operators covered under the W116 rule:

Arithmetic Operators	
Subtraction (-)	Addition (+)
Multiplication (*)	Division (/)
Modulo (mod)	Remainder (rem)
Logical Operators	
logical and (and)	logical or (or)
logical nand (nand)	logical nor (nor)
logical xor (xor)	logical xnor (xnor)
Relational Operators	
Equals (=)	Not Equals (/=)
Greater than (>)	Greater than or equal to (>=)
Less than (<)	Less than or equal to (<=)

When using integer constants, width of the constant is the natural width of the initial value, not the default width (VHDL LRM states it to be 32 bits). For example,

```
constant c1 : integer := 9;
```

Here, the width of the constant will be **log2(9) + 1 = 4**, not **32**.

For VHDL constant arrays, such as Array = {1,0,3,6,0,2,3}, when non-static index is passed for constant arrays like Array[x], it's width is considered to be the maximum width among all elements of the array. For example, here the width of the Array[x] is 3, which is the width of the largest element (6).

NOTE: *For new width related changes, refer to New Width Flow Application Note.*

Width Calculation for VHDL

If you set the value of the *nocheckoverflow* parameter to **yes** or *W116*, the *W116* rule calculates the width according to the LRM (numeric_std lib) as shown in the following examples.

- For addition and subtraction, width is calculated based on the following rules:
  - If both the operands are variables, then max width is considered.
  - If one operand is a variable and other static, then the variable width is considered.
- For multiplication, width is calculated based on the following rules:
  - If both the operands are variable, then the RHS width is the sum of both the variables.
  - If one operand is a variable and other static, then the RHS width is **2*(Variable width)**.
- For Division width is calculated based on the following rules:
  - If both operands are variable, then the left operand width is considered as the expression width.
  - If one operand is a variable and the other static then the width of the expression is considered.

Verilog Width-Mismatch Examples:

check_genvar: When the **check_genvar** parameter is set to **yes**:

```
genvar i;
```

```
for (i = 0; i < 3 ; i++)
```

assign b[3:0] = c[2:0] + i; //violation would be given when this parameter is set to yes

ignore_forloop_index: When the **ignore_forloop_index** parameter is set to **yes**:

```
for (int i = 0; i < 3; i++)
```

b[3:0] = c[2:0] + i; //no violation would be given when this parameter is set to yes

VHDL Width-Mismatch Examples:

strict: When the **strict** parameter is set to **yes**:

```
a(3 downto 0) + b (2 downto 0) ; --violation would be given when strict is set to yes

a(3 downto 0) * b (2 downto 0) ; --violation would be given when strict is set to yes

a(3 downto 0) - b (4 downto 0) ; --violation by default

a(3 downto 0) - b (2 downto 0) ; --violation would be given when strict is set to yes

a(3 downto 0) / b (4 downto 0) ; --violation by default

a(3 downto 0) / b (2 downto 0) ; --violation would be given when strict is set to yes

a(3 downto 0) mod b (4 downto 0) ; --violation by default

a(3 downto 0) mod b (2 downto 0) ; --violation would be given when strict is set to yes

a(3 downto 0) rem b (4 downto 0) ; --violation by default

a(3 downto 0) rem b (2 downto 0) ; --violation would be given when strict is set to yes
```

nocheckveroverflow: When the **nocheckveroverflow** parameter is set to **yes:**

- For addition and subtraction, the width will be calculated based on the following rules:

- If both the operands are variable, then the max width will be taken:

```
signal a: unsigned(2 downto 0);
signal b: unsigned(2 downto 0);
if((a + b) /= b) then   --No violation when, nocheckoverflow=yes
output <= input;
end if;
```

- If one operand is a variable and the other static, then the variable width will be taken:

```
if(a  /= b + 2) then    --No violation when, nocheckoverflow=yes
output <= input;
end if;
```

- For multiplication, the width is calculated as follows:
  - If both the operands are variables, then the RHS width will be the sum of both variables:

```
signal a: unsigned(2 downto 0);
signal b: unsigned(3 downto 0);
if((a * b) /= b) then  --violate a*b = 7, b = 4
b <= a*b;
end if;
```

  - If one operand is a variable and other static, then the RHS width will be 2*(Variable width):

```
if(a*2  /= b) then  --violate a*2 = 6, b = 4
output <= input;
end if;
```

- For division, the width is calculated as follows:
  - If both the operands are variables, then the expression width is the left operand width:

```
signal a: unsigned(2 downto 0);
signal b: unsigned(4 downto 0);
```

```
if((a / b) /= b) then  -- Two violations are reported
output <= input;
end if;
```

- If one operand is a variable and the other static, then the RHS width will be the variable width:

```
if(a  /= b/5) then --violation (a=3, (b/5) = 5)
output <= input
if(a  /= 5/b) then --violation (a=3, (b/5) = 5)
output <= input;
```

Customizing Violation Messages

You can customize the violation messages by using the SpyGlass's overload feature. Following message handles are available for Verilog and VHDL:

- **Verilog**: For logical and ternary operators, use the LINT_W116_ERR message handle to overload the messages. For rest of the operators, use LINT_W116_WRN.
- **VHDL**: For logical and relational operators, use the LINT_W116_ERR message handle to overload the message. For rest of the operators, use LINT_W116_WRN.

To overload the rule, create a lint-policy-overload.pl file and modify the message details as needed. For example, to overload the message severity for logical and ternary operators, in the rule, add the following line to the lint-policy-overload .pl file:

```
spyOverload("LANGUAGE" => "Verilog", "RULE" => "W116", "SEVERITY" => "ERROR", "MESSAGELABEL" => "LINT_W116_ERR")
```

To know more about the rule overloading feature, refer to *SpyGlass Policy Customization Guide.*

Rule Exceptions

For **Verilog**, the rule does not report violations for expressions where the width of a constant expression, a constant integer, or a base number is less than the width of the other operand.

For **VHDL**, following are the exceptions to the W116 rule:

- The W116 rule does not check in the function and procedure body as size of the arguments may depend on the actual passed in the function or procedure call.
- The W116 rule does not report violation for expressions where the width of decimal literal or character literal or based literal is less than the width of the other operand.
- The W116 rule does not report violation for relational operators when one side of relational operator is incremental or decremental and the size of its object is equal to size of object on other side of relation.

Language

Verilog, VHDL

Default Weight

10

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameter(s)

- *check_static_value*: Default value is no. Set the value of the parameter to **`yes`** or **`<rule_list>`** to report violation for cases with width mismatch, involving static expressions and non-static expressions having a static part. Other possible values are **`only_const`** and **`only_expr`**. For an expression having one operand constant and other non constant, a violation is reported only when the width of the constant operand is greater.
- *disable_rtl_deadcode*: The default value is no. Set the value of the parameter to yes to disable violations for disabled code in loops and conditional (if condition, ternary operator) statements.
- *reportconstassign*: The default value is no. Set the value of the parameter to yes to enable the W116 rule to check for constants whose width is less than the operand.
- *sign_extend_func_names*: The default value is ''EXTEND'', “resize”. Set the value of the parameter comma-separated list of function names to

enable the W116 rule to recognize VHDL sign extension functions and calculate width of extend functions as per the **const** extension argument specified in the argument list.

- *strict*: Default value is **no**. Therefore, the rule behavior is as follows:
  - ❒ Ignores addition (**+**) and multiplication (*****) operations
  - ❒ Reports violation on subtraction (**-**), division (**/**), or modulus (**%**) operations only if the width of the right operand is greater than the width of the left operand.

  Set this parameter to **yes** to check for the addition and multiplication operations and to report subtraction, division, or modulus operations when there is a width mismatch between operands (both **A > B** and **B > A** for operations **A-B**, **A/B**, and **A%B**). You can also set this parameter to check for ternary operators
- *use_lrm_width*: Default value is **no**. Set this parameter to **yes** to consider the LRM width of integer constants, which is 32 bits. The rule does not check the bit-width as per LRM by default.

  On setting the **use_lrm_width** parameter to **yes** or **W116**, the *W116* rule does not report violations for operations between integer types when the widths are not greater than 32. If one operand is of the integer type and the other is not, then the width of that expression is considered as 32 (provided that expression's maximum value is not greater than (2^32)-1).
- *nocheckoverflow*: Default value is **no**. Set the value of the parameter to yes or rule name to calculate the width as per the LRM. See *Width Calculation for Verilog* and *Width Calculation for VHDL* to know more about calculating width.
- *checkOperatorOverload*: Default value is **yes**. Set the value of this parameter to **no** to report inconsistent bit-width mismatch for the overloaded operators from non-IEEE packages. This parameter is applicable for **VHDL** only.
- *use_carry_bit*: Default value is **no** and the width is taken as maximum of the two operands for a binary expression having plus and minus operators. Set this parameter to **yes** or **<rule-name>** to get width after considering the carry bit of addition. No violation is reported, even using this parameter, for sub-expressions of a binary expression if all terms have the same width and all operators are either plus or minus.

This parameter is applicable for **Verilog** only. Also, refer to the *Example 10*.

- *check_genvar*: Default value is **no** and the rule does not report violation when the operands are genvar. When this parameter is set to **yes**, the *W116* rule reports a violation for unequal length operands in the bit-wise logical, arithmetic, and ternary operators, although operands are genvar. This parameter is applicable for **Verilog** only.
- *ignore_forloop_indexes*: Default value is **no**. Set the value of this parameter to **yes** to ignore the **for** expressions that contain index variables of **for** loops.
- *check_counter_assignment*: Default value is **no**. Set this parameter to **yes** to report a violation for the counter type of assignments. You can also set the value of the parameter to turbo.
- *ignore_mult_and_div*: Default value is no. Set the value of the parameter to yes to ignore violations for multiplication (*) and division (/) operations.
- *ignore_signed_expressions*: Default value is no. Set the value of the parameter to yes to ignore violation messages for statements where both sides are signed expressions.
- *report_only_bitwise*: Default value is no. Set the value of the parameter to yes to report violations only for bit wise operations in the *W116* rule.
- *ignore_nonstatic_counter*: Default value is no. Set the value of the parameter to yes to ignore violations for non-static single-bit argument in additions and subtractions.

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message appears at the location of an operation of operator *<opr-name>* where there is a bit-width mismatch between left expression *<exprl>* of bit-width *<bit-widthl>* and right expression *<exprr>* of bit-width *<bit-widthr>*:

```
[WARNING] For operator (<opr-name>), left expression: "<exprl>"
width <bit-widthl> should match right expression: "<exprr>"
width <bit-widthr>. [Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical path of the containing scope.

Potential Issues

A violation is reported when there is a bit-width mismatch between the left expression of bit-width and right-expression of bit-width.

Consequences of Not Fixing

While working with expressions of different bit-widths may be the intended behavior, it is also a potentially error-prone design practice. For example, the addition of two words of unequal widths may indicate that you forgot to update the width of one of the buses.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the line, where, width mismatch in operators is detected.

To resolve the violation, check each case for potential messages, especially in the bitwise operators. Review the RTL code mentioned in message, this code may cause some unintended behavior. Make all arguments in such comparisons of equal width such as by explicitly extending narrower operators in a concatenation, to see cases where upper bits are zeroed.

VHDL

The following message appears at the location of a statement where there is a bit-width mismatch between left expression *<exprl>* of bit-width *<bit-widthl>* and right expression *<exprr>* of bit-width *<bit-widthr>*:

```
[WARNING] Left expression: '<exprl>' (width <bit-widthl>) does
not match right expression: '<exprr>' (width <bit-widthr>)
[Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical path of the containing scope.

Potential Issues

Violation can arise when there is a bit-width mismatch between operands of the bit-wise logical, arithmetic, and ternary operators.

Consequences of Not Fixing

While working with expressions of different bit-widths may be the intended behavior, it is also a potentially error-prone design practice. For example, the addition of two words of unequal widths may indicate that you forgot to update the width of one of the buses.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the line, where, width mismatch in operators is detected.

To resolve the violation, check each case for potential messages, especially in the bitwise operators. Review the RTL code mentioned in message, this code may cause some unintended behavior. Make all arguments in such comparisons of equal width such as by explicitly extending narrower operators in a concatenation, to see cases where upper bits are zeroed.

Examples Code and/or Schematic

Example 1

Consider the following example:

```
signal a: unsigned(2 downto 0);
signal b: unsigned(2 downto 0);
 if((a + b) /= b) then
    output <= input;
  end if;
```

In the above example, since both the operands are variables, max width is considered. Also, the **W116** rule does not flag a violation, if the value of the *nocheckoverflow* parameter is set to **yes**.

Example 2

Consider the following example:

```
if(a  /= b + 2) then
  output <= input;
end if;
```

In the above example, since one operand is a variable and other is static, the variable width is considered. Also, the **W116** rule does not flag a violation in this case, if the value of the *nocheckoverflow* parameter is set to **yes**.

Example 3

Consider the following example:

```
signal a: unsigned(2 downto 0);
signal b: unsigned(3 downto 0);
  if((a * b) /= b) then
    b <= a*b;
  end if;
```

In the above example, since both the operands are variables, the RHS width is the sum of both the variables. Also, the **W116** rule flags a violation in this case as the width of (**a*b**) is **7** and the width of **b** is **4**.

Example 4

Consider the following example:

```
if(a*2  /= b) then
  output <= input;
end if;
```

In the above example, since one operand is a variable and other static, the RHS width is **2*(Variable width)**. Also, the **W116** rule flags a violation in this case as the width of (**a*2**) is **6** and the width of **b** is **4**.

Example 5

Consider the following example:

```
signal a: unsigned(2 downto 0);
signal b: unsigned(4 downto 0);
  if((a / b) /= b) then
    output <= input;
  end if;
```

In the above example, since both the operands are variables, the left operand width is considered as the expression width. Also, the **W116** rule flags two violations for the **If** condition.

Example 6

Consider the following example:

```
if(a  /= b/5) then
  output <= input;
if(a  /= 5/b) then
```

```
  output <= input;]
```

In the above example, since one operand is a variable and the other static, the width of the expression is considered as the variable width. Also, the W116 rule flags a violation as the width of **a** is **3** and **(b/5)** is **5**.

Example 7

Consider the following example:

```
assign out = base [4:0] - (1023>>6);
assign out = base [4:0] - 10 ;
assign out = base [4:0] - 1023;
```

In the above example, when the **check_static_value** parameter is set, the rule reports violation only for the third expression. No violations are reported for the first two expressions because the width of a constant expression, a constant integer, or a base number is less than the width of the other operand.

Example 8

Consider the following example:

```
if(a = b + 1)
```

In the above example, the W116 rule does not report violation, if size of **a** is equal to size of **b**, or size of **a** is equal to size of **(b + 1)**.

Example 9

(Verilog)

Consider the following example code in which the arithmetic operation involves operands of different bit-widths — register **a** (4 bits) and register **data** (16 bits):

```
module test(in, out, clk);
  input [3:0] in;
  input clk;
  output [3:0] out;

  reg [3:0] out, a, b;
  reg [15:0] data;

  always @(a)
    begin
```

```
      b = a + data;
    end

  always @(posedge clk)
    begin
      out <= b + 2;
    end
endmodule
```

Here, SpyGlass generates the following violation:

For operator (+), left expression: "a" width 4 should match right expression: "data" width 16.

Example 10

(Verilog)

Consider the following example code:

```
module test ( a, b, c, o1);
    input a,b,c;
    output o1;
    wire d, e, f, g, h, i;

    assign o1 = a & (b + c); //Violation, when parameter
    use_carry_bit is set
    assign o1 = a+b+c-d-e+f;  //No violation
endmodule
```

For the above example, SpyGlass reports the following violation when the **`use_carry_bit`** parameter is set:

```
For operator (&), left expression: "a" width 1 should match
right expression: "(b + c)" width 2. [Hierarchy: ':test']
```

Default Severity Label

Warning

Rule Group

Expression

Reports and Related Files

No related reports or files.

W159

Condition contains a constant expression

Language

Verilog

Rule Description

The W159 rule flags control expressions that evaluate to a constant.

The W159 rule flags constant control expressions used in **`if`** statement conditions, loop conditions, and conditional statements.

NOTE: *The W159 rule is switched off by default. You can enable this rule by either specifying the* **`set_goal_option addrules W159`** *command or by setting the verilint_compat rule parameter to* **`yes`**.

Constant control expression or sub-expression may result in an always ON or always OFF logic inference. A condition with a constant expression is usually a mistake, except where the condition represents a hard-wire configuration parameter.

By default, the value of the *ignore_cond_having_identifier* is set to **`no`** and this rule considers constants defined by parameters, localparams or constant integers, for rule checking. Set the value of this parameter to **`yes`** to not report violation for such constants.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location where the control expression *<expr>* evaluates to a constant:

`Constant expression <expr> in condition [Hierarchy: '<hier-path>']`

Where, *<hier-path>* is the complete hierarchical path of the containing scope.

Rule Severity

Warning

Suggested Fix

There may not need to be a fix as long as you are aware that portion of your logic may be permanently disabled/optimized out as a result

Examples

Following examples show constant control expressions where SpyGlass generates the W159 rule message:

```
if(1'b1)
  ...
`define cond 1'b1
...
while(`cond)
...
out1 <= 1'b0 ? in1: in2;
```

W180

Zero extension of extra bits

Language

Verilog

Rule Description

The W180 rule flags constant value specifications where the specified constant value is narrower than the specified width.

When the specified constant value has fewer bits in value specification as compared to size specification, it requires value extension by logic 0.

NOTE: *The W180 rule is switched off by default. You can enable this rule by specifying the* **`set_goal_option addrules W180`** *command.*

NOTE: *The W180 rule has been deprecated. The functionality of this rule is covered by the W341 rule.*

Message Details

The following message appears at the location where the constant value <*value*> is zero-extended to match its size specification:

```
Constant value <value> will be '0'-extended to match size specification
```

Rule Severity

Warning

Suggested Fix

Generally nothing to fix, unless you find a case where zero-extension is not acceptable.

Examples

In the following examples, constants need 0-extension:

```
wire cnst = 2'b1;

reg [7:0] reg1;
```

```
reg1 <= 8'b0x;

`define cnst 2'b1
```

W224

Multi-bit expression found when one-bit expression expected

Language

Verilog

Rule Description

The W224 rule flags multi-bit expressions where single-bit expressions were expected.

A condition expression should evaluate to a single-bit value. While the method for evaluating a multi-bit value as a truth value is well defined, this form is generally less readable and is known to lead to errors of interpretation.

In case of complex expressions, the W224 rule will report violation only if the final value evaluated (for a condition **expr**) is not a single-bit value. For example, this rule will report violation for the following expression

```
...
input C;
reg [3:0] H;
...
H = ( H -  C ) ? 4'd14 : 4'd2;
...
```

Parameters

- *use_natural_width*: Use this parameter to calculate the width using natural width.
- *ignore_parameter*: By default, the parameter is set to no. Set this parameter to **yes** to enable the rule to not report any violations for parameters and local parameters.

Message Details

The following message appears at the location of use of a multi-bit expression <*expr*> where a single-bit expression was expected:

`Multi-bit expression '<expr>' found when one-bit expression expected [Hierarchy: ‘<hier-path>’]`

Where, <*hier-path*> is the complete hierarchical path.

Rule Severity

Warning

Suggested Fix

Select the appropriate bit and test that bit only.

Examples

In the following example, the constant **a_def** is of 16 bits (being an integer) and is used in the **if** statement that expects a scalar signal:

```
`define a_def 1
...
always @(posedge clk)
  if (`a_def)
    q <= d;
```

For this example, SpyGlass generates the following message:

```
Multi-bit expression '1' found when one-bit is expected
```

W289

Reports real operands that are used in logical comparisons

When to Use

Use this rule to identify cases where a real operand is used in a logical comparison.

Rule Description

The *W289* rule reports real operands used in logical comparisons.

NOTE: *The W289 rule supports generate-if, generate-for, and generate-case blocks.*

Language

Verilog

Parameters

None

Constraints

None

Messages and Suggested Fix

The following message appears at the location where a real operand <*oprd-name*> is used with a logical comparison operator <*op-name*>.

```
[WARNING] A real operand: '<oprd-name>' should not be used with logical comparison operator '<op-name>'
```

Potential Issues

Violation may arise when a real operand is used with a logical comparison operator.

Consequences of Not Fixing

It is unlikely that conditions, such as **`RealValue == 1`**, would ever be true, given rounding errors in floating point values.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the logical comparison statement where a real variable is used as one of the

operand.

You should allow some margin for rounding. For example, **`RealValue > 0.99 && RealValue < 1.01`**

Example Code and/or Schematic

Consider the following example:

```
module mod(i1, i2, clk, res, o1);
  input  i1, i2;
  input clk, res;
  output  o1;
  reg o1;

  real a, b;

  always @(posedge clk or negedge res)
    if (a!=b)
      o1 <= a + b;
    else
      o1 = a-b;
endmodule
```

In the above example, the *W289* rule reports a violation as the real operand, **a**, is used with logical comparison operator, '!='.

Default Severity Label

Warning

Rule Group

Expression

Reports and Related Files

None

W292

Reports the comparison of real operands

When to Use

Use this rule to identify logical comparison operations on real type operands.

Description

The *W292* rule reports logical comparison operations on real type operands.

Language

VHDL

Parameters

None

Constraints

None

Messages and Suggested Fix

The following message appears at the location where a logical comparison of real operands is encountered.

`[WARNING] Logical Comparison of Real operands is not recommended`

Potential Issues

Violation may arise when real operands are used in logical comparisons.

Consequences of Not Fixing

It is unlikely that conditions, such as **`RealValue == 1`**, would ever be true, given rounding errors in floating point values.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where real signals are used in comparison.

You should allow some margin for rounding. For example, **`RealValue >`**

```
0.99 and RealValue < 1.01
```

Example Code and/or Schematic

Consider the following example:

```
entity test1 is
  port(
       in1, in2 : in bit;
       out1, out2 : out bit
       );
end test1;

architecture behav of test1 is
signal  b1, b2, b3: real;
signal sig1: bit;
begin
  process(in1)
  begin
    if(b1 = 1.0)  then
        out1 <= in1;
    else
        out1 <= not in1;
    end if;
  end process;
  process(in2)
  begin
    if(b2 = b3)  then
       out2 <= in2;
    else
       out2 <= not in2;
    end if;
  end process;
end behav;
```

In the above example, the W292 rule reports a violation as the logical comparison of real operands is not recommended.

Default Severity Label

Warning

Rule Group

Expression

Reports and Related Files

None

W341

Constant will be 0-extended

Language

Verilog

Rule Description

The W341 rule flags constant assignments where the size specification of the assigned value is wider than the value specification and the high-order bit/byte is zero. As a result, the constants will be 0-extended.

No rule checking is done for unused macro definitions and unused parameters.

By default, the W443 rule considers all type definitions for rule checking. Set the value of the *ignore_typedefs* parameter to yes to ignore all type definitions for rule checking.

NOTE: *The W341 rule is switched off by default. You can enable this rule by specifying the* **`set_goal_option addrules W341`** *command.*

Message Details

The following message appears at the location of the constant assignment where the constant <*const-name*> is extended by zeros to match the size specification:

```
Constant <const-name> is extended by 0; value has fewer bits than size specification
```

Rule Severity

Warning

Suggested Fix

It is best to use the correct width and value specification, rather than depending on default extension.

Examples

Consider the following examples:

Specified Value	Is Equivalent to...	Whether Message
8'b0x	8'b0000000x	Yes as the value is 0-extended
9'h0x	9'b00000xxxx	Yes as the value is 0-extended
8'h0x	8'b0000xxxx	No as the value is not 0-extended

W342

Reports constant assignments that are X-extended

When to Use

Use this rule to identify constant assignments that are X-extended.

Description

The *W342* rule reports constant assignments where the size specification of the assigned value is wider than the value specification and the high-order bit/byte is X (unknown). As a result, the constant would be X-extended.

Prerequisites

The *W342* rule is switched off by default. You can enable this rule in the following ways:

- Adding this rule in the list of rules run by **`set_option addrules`**. For example, **`set_goal_option addrules W342.`**
- When this rule is not mentioned in the **`addrules`** section, then switch on the **`strict`** and **`fullpolicy`** parameters.

Rule Exceptions

Rule checking is not done for unused macro definitions and unused parameters.

Language

Verilog

Parameters

- *strict*: The default value of this parameter is no. Set the value of this parameter to yes to switch on the rule and report constant assignments that are X-extended. You can also specify a comma-separated list of rules as an input to this parameter.\
- *ignore_typedefs*: By default, the W443 rule considers all type definitions for rule checking. Set the value of the *ignore_typedefs* parameter to yes to ignore all type definitions for rule checking.

Constraints

None

Messages and Suggested Fix

The following message appears at the location of constant assignment where a constant <*const-name*> is extended by Xs to match the size specification.

```
[WARNING] Constant <const-name> will be X-extended, value has fewer bits than size specification
```

Potential Issues

Violation may arise when the constant value has fewer bits than size specification.

Consequences of Not Fixing

If the constant value has fewer bits than the size specification, then the constant is X-extended.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the line where the constant that is X-extended.

To fix the violation, it is best to use the correct width and value specification, rather than depending on default extension.

Example Code and/or Schematic

Consider the following examples:

Specified Value	Is Equivalent to...	Whether Message
8'bx0	8'bxxxxxxx0	Yes as the value is X-extended
9'hx0	9'bxxxxx0000	Yes as the value is X-extended
8'hx0	8'bxxxx0000	No as the value is not X-extended

Default Severity Label

Warning

Rule Group

Expression

Reports and Related Files

None

W343

Reports constant assignments that are Z-extended

When to Use

Use this rule to identify constant assignments that are Z-extended.

Description

The *W343* rule reports constant assignments where the size specification of the assigned value is wider than the value specification and the high-order bit/byte is Z. As a result, the constant will be Z-extended.

Prerequisites

The *W343* rule is switched off by default. You can enable this rule in the following ways:

- Adding this rule in the list of rules run by **`set_option addrules`**. For example, **`set_goal_option addrules W343.`**
- When this rule is not mentioned in the **`addrules`** section, then switch on the **`strict`** and **`fullpolicy`** parameters.

Rule Exceptions

Rule checking is not done for unused macro definitions and unused parameters.

Language

Verilog

Parameters

- *strict*: The default value of this parameter is no. Set the value of this parameter to yes to switch on the rule and report constant assignments that are Z-extended. You can also specify a comma-separated list of rules as an input to this parameter.
- *ignore_typedefs*: By default, the W443 rule considers all type definitions for rule checking. Set the value of the *ignore_typedefs* parameter to yes to ignore all type definitions for rule checking.

Constraints

None

Messages and Suggested Fix

The following message appears at the location of constant assignment where a constant <*const-name*> is extended by Zs to match the size specification.

```
[WARNING] Constant <const-name> will be Z-extended, value has fewer bits than size specification
```

Potential Issues

Violation may arise when the size specification of the assigned value is wider than the value specification and the high-order bit/byte is Z.

Consequences of Not Fixing

Not fixing the violation can make the constant Z-extended.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location of the offending constant that is Z-extended.

To fix the violation, it is best to use the correct width and value specification, rather than depending on default extension.

Example Code and/or Schematic

Consider the following examples:

Specified Value	Is Equivalent to...	Whether Message
8'bz0	8'bzzzzzzz0	Yes as the value is Z-extended
9'hz0	9'bzzzzz0000	Yes as the value is Z-extended
8'hz0	8'bzzzz0000	No as the value is not Z-extended

Default Severity Label

Warning

Rule Group

Expression

Reports and Related Files

None

W362

Reports an arithmetic comparison operator with unequal length

When to Use

Use this rule to identify arithmetic comparison operator with unequal length.

Description

The *W362* rule reports arithmetic comparison operations with operands of unequal widths. Following is the list of operators covered under this rule:

Relational Operators	
Greater than (>)	Greater than or equal to (>=)
Less than (<)	Less than or equal to (<=)
Equality Operators	
logical Equality(==)	logical inequality(!=)
case equality(===)	case inequality(!==)

NOTE: *If the count of operators is more than 500 in an expression, then the expression is ignored for rule checking.*

NOTE: *The W362 rule does not report violation for counter cases, for example, data == data+1.*

If the *nocheckoverflow* parameter is set to **`yes`** or **`W362`**, the width of the expression is calculated as per the LRM. However, for constants, normal width is considered.

Operator	LRM Width	Normal Width
+, -	Max (lhswidth, rhswidth)	Max (lhswidth, rhswidth) + 1
*	Max (, rhswidth)	lhswidth + rhswidth
/	Max (lhswidth, rhswidth)	lhswidth
%	Max (lhswidth, rhswidth)	RHS (non-static) - RHS width RHS (static) - width of (RHS Value -1)

Handling of Unary Negation:

In case of unary negation, the width is calculated as follows:

- **Variable and based numbers:** If an operand is a variable or a based number, the width is incremented by one. For example, width of **-a[3:0]** is 5, and width of **-3'b101** is 4, etc. This is applicable only when the *nocheckoverflow* parameter is set to **no**.
- **Constant and unsized based number:** In case of constants, the natural width is calculated first and is then incremented by one. For example, width of **-9** is 5. An unsized based number is treated similarly. For example, width of **-'b101** is 4. However, if you set the *use_lrm_width* parameter to **yes**, the width is considered as 32 bit.

Width-Mismatch Example:

handle_zero_padding: When the **handle_zero_padding** parameter is set to **yes**:

```
a[3:0] == 12'h00A //no violation when this parameter is on as leading zeroes can be truncated to match the LHS width
```

Rule Exceptions

The *W362* rule does not report a violation if any one operand is of integer data type. See the *Example 5* for details.

Language

Verilog

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameters

- *check_parameter*: Default value is no. Set the value of the parameter is yes to report violations if one of the operands is a parameter/localparam and the other operand is non-static, even if the value of the check_static_value parameter is set to no

- *check_sign_extend*: Default value is no. Set the value of the parameter to yes to check for width mismatch due to sign extension in signed comparisons.
- *handle_zero_padding*: Default value is no. Set the value of the parameter to yes to perform leading zero expansion and truncation of RHS of an assignment.
- *disable_rtl_deadcode*: The default value is **no**. Set the value of the parameter to yes to disable violations for disabled code in loops and conditional (if condition, ternary operator) statements.
- *use_lrm_width*: Default value is **no**. This indicates the *W362* rule considers the natural width of integer constants. Set this parameter to **yes** to consider the LRM width, which is 32 bits.
- *nocheckoverflow*: Default value is **no**. This indicates the *W362* rule does not check the bit-width as per LRM. Set this parameter to yes or rule name to check the bit-width as per LRM.
- *check_static_value* and *strict*: By default, the *W362* rule does not report violation when the right or left expression is a constant, including parameter, sized or unsized based number, and unsized integer. Setting the **check_static_value** parameter to **yes** changes this behavior, and setting the *strict* parameter in addition further alters the behavior. The following table summarizes these variations in behavior:

Type of left or right expression	**check_staic_value set to no**	***check_staic_value* set to yes**	
		***strict* set to no**	***strict* set to yes**
Parameter	Does not report	Reports if the width of constant is larger	Reports any width mismatch
Sized based number (8'h15)	Does not report	Reports if the width of constant is larger	Reports any width mismatch
Unsized based number ('h15)	Does not report	Reports if the width of constant is larger	Reports if the width of constant is larger
Unsized integer (18)	Does not report	Does not report	Reports if the width of constant is larger

See the *Example 6* for details.

Also, when set to **yes**, the **check_static_value** parameter checks for width mismatch involving static expressions and non-static expressions that contain a static part. Refer to the *check_static_value* section for more details.

NOTE: *When the strict parameter is set, the W362 rule does not report violation for width mismatch in the for loop condition.*

Constraints

None

Messages and Suggested Fix

The following message appears at the location where a width <*widthl*> of a left expression <*exprl*> does not match a width <*widthr*> of a right expression <*exprr*> in an operation of an arithmetic operator <*op-name*>.

[WARNING] For operator (<op-name>), left expression: "<exprl>" width <widthl> should match right expression: "<exprr>" width <widthr> [Hierarchy: '<hier-path>']

Where, <*hier-path*> is the complete hierarchical path.

Potential Issues

A violation is reported when an arithmetic operation has operands of unequal length.

Consequences of Not Fixing

For some range of values of the wider operand in arithmetic comparison operations, the comparison operation evaluates to a constant, independent of the value of the narrower operand. This may result in an unexpected behavior.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the width mismatch is found for comparison operator.

This is not a major issue. However, you can avoid possible problems by explicitly comparing sub-expressions of equal width. This also enhances the code readability.

Example Code and/or Schematic

Example 1

Consider the following example:

```
module top(q,clk, d,reset);
input  clk,d,reset;
output  q;
reg q;
reg [3:0] a;
reg [7:0] b;
reg [1:0]data;

always @(posedge clk)
begin
if (data == reset)    //violation
    a <=17;
else
    b =  8'hbb;
end
endmodule
```

For the above example, the *W362* rule reports the following violation message:

```
For operator (==), left expression: "data" width 2 should match
right expression: "reset" width 1 [Hierarchy: ':top']
```

Example 2

Consider the following example in which the **`if`** statement condition expression is an arithmetic comparison operation involving expressions of unequal bit-width.

```
module test (clk);
  input clk;

  reg [3:0] a;
  reg [7:0] b;
  reg [2:0] data;

  always @(posedge clk)
```

```
    begin
      if (data <= (a[1] + b[2]))
        a <= 17;
      else
        b <= 8'hbb;
    end
endmodule
```

Example 3

In the following example, the *check_sign_extend* parameter is set to **yes**.

```
reg signed [9:0] data;
always@(*)
if (data == 'sh300) ;
```

In this example, SpyGlass reports the following violation message because the width of the left expression does not match the width of the right expression:

```
For operator (==), left expression: "data" width 10 should
match right expression: "'sh300" width 32 [Hierarchy: ':top ']
```

Example 4

Consider the following example:

```
module top(input clk);
reg [6:0]a;
reg [2:0] data;
always @(posedge clk)
begin
  for(data =1;data <= 8'hAA; data++)
        a <= 17;
end
endmodule
```

For the above example, when the **strict** parameter is set to **yes,** the *W362* rule does not report violation for width mismatch in the **for** loop condition. Whereas, by default, the rule reports violation for such cases.

Example 5

Consider the following example:

```
module top(input clk);
```

```
reg [3:0] a, b, data;
integer i1;
int i2;

always @(posedge clk)
begin
if (data == i1)
   a <= 17;
else if(data == i2)
   b =  8'hbb;
end

endmodule
```

In the above example, the *W362* rule does not report violations because operands **i1** and **i2** are of integer/int data type.

Example 6

Consider the following example:

```
module top(input clk);
reg [3:0] a, b, data;
integer i1;
int i2;
parameter P1 = 119;
parameter P2 = 2;

always @(posedge clk)
begin
if (data == P1)      //violation with check_static_value
   a <= 17;
else if(data == P2) //violation with check_static_value and strict
   b =  8'hbb;
if (data == 10'd119)  //violation with check_static_value
   a <= 17;
else if(data == 2'b10)//violation with check_static_value and strict
   b =  8'hbb;
if (data == 'd119)     //violation with check_static_value
   a <= 17;
```

```
else if(data == 'b10)
   b =  8'hbb;
if (data == 119)  //violation with check_static_value and strict
   a <= 17;
else if(data == 2)
   b =  8'hbb;
end

endmodule
```

In the above example, violations are reported for different cases (mentioned above in red) when the *check_static_value* and/or *strict* parameters are set.

Default Severity Label

Warning

Rule Group

Expression, Lint_Elab_Rules

Reports and Related Files

None

W443

'X' value used

Language

Verilog, VHDL

Rule Description

The W443 rule flags based numbers that contain the unknown value character (**X**).

The unknown value character (**X**) has no physical counterpart and may lead to a mismatch between pre- and post-synthesis simulation.

By default, the W443 rule does not check the presence of **X** value in the **default** statement of the case construct in Verilog designs. Use the *strict* rule parameter to check in the **default** statement also.

For Verilog, no rule checking is done for unused macro definitions and unused parameters.

By default, the W443 rule considers all type definitions for rule checking. Set the value of the *ignore_typedefs* parameter to yes to ignore all type definitions for rule checking.

NOTE: *The W443 rule ignores case select item value within* **case** *constructs.*

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

Verilog

The following message appears at the location where an unknown value character (**X**) is encountered in a based number *<num>*:

```
Based number <num> contains an X - has no meaning in synthesis
```

VHDL

The following message appears at the location where a forcing unknown character (**X**) is encountered in a bit string:

```
'X' state used - has no meaning in synthesis
```

Rule Severity

Warning

Suggested Fix

Use of **X** states is often valuable in functional debug, so the best way to handle these cases is to bracket them in **translate_off**, **translate_on** pragmas, making them visible in simulation but invisible in synthesis. You should check carefully to ensure that implemented behavior is implicitly dependent on **X** behavior.

Examples

Verilog

Consider the following example:

```
module top(out, en1);
    input en1;
    output reg [1:0] out;

  always @ (en1)begin
        if (en1)
             out<=2'bxx;
        else
            out<=2'b01;
  end

endmodule
```

In the above example, the *W443* rule reports a violation because the based numbers that contain the unknown value character (X) are assigned to **out** in the module. The following message is reported by this example:

```
Based number 2'bxx contains an X - has no meaning in synthesis
```

VHDL

Consider the following example:

```
entity top is
port(
    en1: in std_logic;
    out1: out std_logic_vector(1 downto 0)
    );
end top;

architecture behav of top is begin
        process (en1) begin
                if (en1 = '1') then
                     out1<="XX";
                else
                    out1<="01";
                end if;
        end process;
end behav;
```

In the above example, the *W443* rule reports a violation because the based numbers that contain the unknown value character (X) are assigned to *out1* in the architecture behave of the entity **top**. The following message is reported by this example:

```
'X' state used - has no meaning in synthesis
```

W444

'Z' or '?' value used

Language

Verilog, VHDL

Rule Description

The W444 rule flags all occurrences of the high impedance character ('**Z**') in the design.

Using '**Z**' in the design creates several issues, such as:

- If '**Z**' is used in an assignment statement, tristates are inferred
- If '**Z**' is used in a comparison expression, the condition is always considered false in synthesis and may lead to a mismatch between pre- and post-synthesis simulation

The W444 rule also flags the usage of '**?**', which is Verilog HDL alternative for the z character.

By default, the W444 rule does not check for the presence of '**Z**' value in the **`default`** statement of the `case` construct in Verilog designs. Use the *strict* rule parameter to check the **`default`** statement also.

For Verilog, no rule checking is done for unused macro definitions and unused parameters.

By default, the W443 rule considers all type definitions for rule checking. Set the value of the *ignore_typedefs* parameter to yes to ignore all type definitions for rule checking.

NOTE: *The W444 rule ignores case select item value within* **`case`** *constructs.*

Message Details

Verilog

The following message appears at the location where a high impedance character ('**Z**') or '**?**' is encountered in a tristate value <*value*>:

`Tristate value <value> specified`

VHDL

The following message appears at the location where a high impedance character ('**Z**') is encountered in a bit string:

```
Tristate value specified
```

Rule Severity

Warning

Suggested Fix

For assignments, confirm that you intended to infer a tristate. Avoid using '**Z**' values in comparisons, except for simulations tests, which you should bracket in **translate_off**, **translate_on** pragmas to avoid inferring spurious logic.

Examples

Verilog

Consider the following example:

```
module top(out, in1, en1);
    input in1;
    input [1:0] en1;
    output out;
    reg out;

  always @ (en1 or in1)
    begin
        if (en1 == 2'b1z)
            out <= in1;
        else
            out <= 1'b?;
        end

endmodule
```

In the above example, the *W444* rule reports two violations because of the high impedance characters ('Z') and ('?') in the design.

The following messages are reported by this example:

```
Tristate value '2'b1z' specified
Tristate value '1'b?' specified
```

VHDL

Consider the following example:

```
entity test1 is
port(
    clk: in std_logic;
    out1: out std_logic_vector(2 downto 0)
    );
end test1;

architecture behav of test1 is
begin
    process(clk)
    begin
        if (clk'event and clk = '1') then
             out1 <= "10Z";
        end if;
    end process;
end behav;
```

In the above example, the *W444* rule reports a violation because of the high impedance character ('Z') in the design. The following message is reported by this example:

```
Tristate value specified
```

W467

Use of don't-care except in case labels may lead to simulation / synthesis mismatch

When to Use

Use this rule to identify the usage of don't care character in the design.

Description

The W467 rule reports violation for based numbers that contain the don't care character.

Rule Exceptions

The W467 rule does not flag a violation for a parameter, generic, or constants that are assigned a don't-care value and are used only as case-label.

Also, for Verilog, no rule checking is done for unused macro definitions.

Language

Verilog, VHDL

Default Weight

5

Parameter(s)

- *ignore_typedefs*: By default, the W443 rule considers all type definitions for rule checking. Set the value of the *ignore_typedefs* parameter to yes to ignore all type definitions for rule checking.
- *ignore_wildcard_operators*: Default value is no. Set the value of the parameter to yes to ignore violations given in RHS side of wild operators ==? and !=? as well as inside operator LHS 'inside' RHS.

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message appears at the location where a don't care character (**?**) is encountered in a based number <*num*>:

```
[WARNING] Based number <num> contains a don't care (?) - might
lead to simulation/synthesis mismatch
```

Potential Issues

Violation may arise when a based number contains a **don't care** value.

Consequences of Not Fixing

There is no physical counterpart for the don't-care value. In simulation, these values are typically mapped to '**Z**' which causes a tristate to be inferred. However, this behavior should be avoided as it may result in inferring spurious logic.

How to Debug and Fix

For more information on debugging and fixing the violation, click *How to Debug and Fix*

VHDL

The following message appears at the location where a don't care character (**-**) is encountered in a bit string:

```
[WARNING] Don't-care (-) used - might lead to simulation/
synthesis mismatch
```

Potential Issues

Violation may arise when a **don't care** character (**-**) is encountered in a bit string.

Consequences of Not Fixing

There is no physical counterpart for the don't-care value. In simulation, these values are typically mapped to '**Z**' which causes a tristate to be inferred. However, this behavior should be avoided as it may result in inferring spurious logic.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the **don't** care value, **?**, is used in the design other than in the

case label.

To fix the violation, test both 0 and 1 values in a comparison. Also, in an assignment, choose either 0 or 1.

Example Code and/or Schematic

Consider the following example:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity test1 is
port(
      in1: in std_logic_vector(2 downto 0);
      out1: out std_logic_vector(2 downto 0)
     );
end test1;

architecture behav of test1 is
signal sig1: std_logic_vector(2 downto 0);
begin
  process(in1)
  begin
    sig1 <= "-0-";
    out1 <= sig1 and in1;
  end process;
end behav;
```

In the above example, the W467 rule reports a violation as a **don't-care** character (**-**) is used in signal assignment.

Default Severity Label

Warning

Rule Group

Expression Rules

Reports and Related Files

No related reports or files.

W486

Reports shift overflow operations

When to Use

Use this rule to identify shift overflow operations.

Description

The W486 rule reports violation for shift overflow operations.

Width Calculation

The W486 rule calculates the width of left operand of shift operator on the basis of the following conditions:

- If the *nocheckoverflow* rule parameter is set to **yes** or **W486**, width is calculated as per LRM. However, for constants, natural width is considered.
- If the *nocheckoverflow* is set to **no**, width is calculated according to the following methods:
  - For plus/minus operator, value based width is considered. For example:

    ```
    a[2:0]+b[2:0])>>1
    // Width of expression, a[2:0]+b[2:0], will be 4
    (7+7=14, 4 bits wide)
    ```

  - For multiplication operator, width will be sum of operand widths. For example:

    ```
    (a[2:0]*b[2:0])<<1;
    // Width of expression, a[2:0]*b[2:0], will be 6
    (a1[2:0]*b1[2])<<2;
    // Width of expression, a1[2:0]*b1[2], will be 3
    ```

  - For division operator, LHS width will be considered. For example:

    ```
    (a[2:0]/b[2])<<2;
    // Width of expression, a[2:0]/b[2], will be 3
    ```

  - For concat operator, width will be calculated as sum of all the operands. For example:

    ```
    1'b0,a[2:0]}<<1;
    ```

```
// Width concat exp=3 bits (leading 0 ignored)
{1'b1,a[2:0]}<<2;
// Width concat exp=4 bits (1+3 bits)
```

- For left_shift expression whose LHS is a constant and the RHS is a variable, the W486 rule calculates the width as shown below:

  Consider the following example expression:

  ```
  a[3:0] = 1 << b[3:0];
  ```

  For the above expression, the W486 rule calculates the width as shown in the following table:

check_shifted _width	use_lrm_width	Width of the above expression	Violation
Yes	Yes	32, unless value exceeds	Yes
Yes	No	16 (1 shifted by 15)	Yes
No	Yes	32 (LRM width of 1)	Yes
No	No	1 (natural width of 1)	No

NOTE: *For new width related changes, refer to New Width Flow Application Note.*

NOTE: *The W486 rule supports generate-if, generate-for, and generate-case blocks.*

Language

Verilog

Default Weight

5

Parameter(s)

- *disable_rtl_deadcode*: The default value is no. Set the value of the parameter to yes to disable violations for disabled code in loops and conditional (if condition, ternary operator) statements.
- *nocheckoverflow*: The default value is **no**. Set the value of the parameter to **yes** or rule name to check the bit-width as per the LRM.

- *use_lrm_width*: The default value is **no**. Set the value of the parameter to **yes** to consider the LRM width of integer constants, which is 32 bits.
- *check_shifted_width*: The default value is no. In this case, the rule considers the width of left operand of left shift expression if left operand is constant integer and right operand is non static. Set the value of the parameter to yes to consider the shifted width of the expression.

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location of a left-shift operation where the width <*widthr*> of RHS expression <*rexpr*> is greater than the width <*widthl*> of the LHS expression <*lexpr*>:

```
[WARNING] Rhs width '<widthr>' with shift (Expr: '<rexpr>') is
more than lhs width '<widthl>' (Expr: '<lexpr>'), this may
cause overflow [Hierarchy: ‘<hier-path>’]
```

Where, <*hier-path*> is the complete hierarchical path.

Potential Issues

Violation may arise when the width of the RHS expression is greater than the width of the LHS expression in a left-shift operation.

Consequences of Not Fixing

When an expression containing a left-shift is assigned to a bus with a width less than required to hold some of the most-significant shifted bits, such bits are truncated and this can lead to arithmetic functional error.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the width mismatch due to shift overflow is found.

To fix the violation, make sure you intend to discard the overflow bits.

Example Code and/or Schematic

Example 1

Consider the following example:

```
wire [2:0] net1, net2;
assign net1 = net2 << 1;
```

In the above example, the W486 rule reports a violation as the expression containing a left-shift is assigned to a bus with a width less than required to hold some of the most-significant shifted bits.

Example 2

Consider the following examples for based numbers for which the W486 rule does not report a violation:

```
wire [4:0] w1;
assign w1 = 5'b1 << 2;
assign w1 = 3'b1 << 2;
```

Example 3

Consider the following example:

```
module mod(in1, clk, sel, out1);
  input [2:0] in1;
  input clk, sel;
  output [2:0] out1;
  reg [2:0] out1;

  always @(posedge clk)
    if(sel)
      out1 = in1;
    else
      out1 = in1 << 1;    //violation(Rhs width (Expr: '(in1
<< 1)') is more than lhs Lhs width (Expr: 'out1'), this may
cause overflow)

endmodule
```

In the above example, the W486 rule reports a violation as the RHS width of the expression, **`(in1 << 1)`**, is more than the LHS width of the expression, **`(out1)`**. This condition may cause an overflow.

Example 4

Consider the following example:

```
module top ();
```

```
wire [4:0]a;
reg [1:0] b;
assign a[3:0] = 1 << b; //Violation by default
assign a[3:0] = 'b11 << b; //Violation by default
assign a[3:0] = 'd13 << b; //Violation by default

endmodule
```

In the above example, for all three assignments by default, the width is considered as 32 bits for the RHS and a violation is reported.

When the *check_shifted_width* parameter is set to **yes**, the natural width of left operand of the RHS expression is considered and no violation is reported for any of the assignments.

Default Severity Label

Warning

Rule Group

Expression, Lint_Elab_Rules

Reports and Related Files

No related reports or files.

W490

A control expression / sub-expression is a constant

Language

Verilog, VHDL

Rule Description

The W490 rule flags constant control expressions or sub-expressions.

Constant control expressions or sub-expressions may result in an always ON or an always OFF logic inference. Such descriptions are not recommended.

Constant control expressions or sub-expressions should only occur in hard-wired configuration settings. Use this rule to find and check all such cases.

By default, the W490 rule does not flag such constructs in subprogram descriptions. For VHDL designs, set the *strict* parameter to check for constant control expressions inside subprogram descriptions also.

NOTE: *The W490 rule supports generate-for block.*

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location of a constant control expression or sub-expression <*expr*>:

```
Control expression/sub-expression '<expr>' is a constant
```

Rule Severity

Warning

Suggested Fix

Fix may not be required as long as you are aware that portion of your logic may be permanently disabled/optimized out as a result.

Examples

Consider the following example:

```
module test (in, in1, out, out1, out2, clk);
  input [3:0] in,in1;
  input clk;
  output out, out1, out2;

  reg out, out1, out2;
  wire data, node;
  parameter par = 4'b1010;

  assign data = 2 ? in : in1;
  assign data = par ? in : in1;

  always @(clk)
    begin
      if(3)
        out = node ? in1 : in;
      else
        out = in1;
      while (par < 3)
        out1 = in1 & in;
      while (par[2])
        out2 = in1 & in;
   end
endmodule
```

In the first conditional assignment statement, the select condition (**2**) is a constant. Thus, SpyGlass generates the following message:

```
Control expression/sub-expression '2' is a constant
```

In the second conditional assignment statement, the select condition (**par**) is a parameter. Thus, SpyGlass generates the following message:

```
Control expression/sub-expression 'par' is a constant
```

In the first **always** construct, the **if** statement condition (**3**) is a constant. Thus, SpyGlass generates the following message:

```
Control expression/sub-expression '3' is a constant
```

In the **always** construct, the first **while** statement condition (**par < 3**) evaluates to a constant. Thus, SpyGlass generates the following message:

```
Control expression/sub-expression '(par < 3)' is a constant
```

In the **always** construct, the second **while** statement condition (**par[2]**) is a constant. Thus, SpyGlass generates the following message:

Control expression/sub-expression 'par[2]' is a constant

W491

Reports case expression with a width greater than the specified value

When to Use

Use this rule to identify case expression that have a width greater than the specified value and have a don't care (?) value in the high order bit/byte.

Rule Description

The W491 rule reports violation for case clause condition constants where the size specification of the constant is wider than the value specification and the high-order bit/byte is don’t care (**?**). As a result, the constant is **?**-extended.

Rule Exceptions

No rule checking is done for unused macro definitions and unused parameters.

Default Weight

5

Language

Verilog

Parameter(s)

ignore_typedefs: By default, the W443 rule considers all type definitions for rule checking. Set the value of the *ignore_typedefs* parameter to yes to ignore all type definitions for rule checking.

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where the case clause expression constant <*constant*> is **?**-extended:

```
[WARNING] Constant <constant> will be ?-extended
```

Potential Issues

Double-click the violation message. The HDL window highlights the line where constant is ?-extended. Review the code, use the correct width and value specification rather than depending on the auto extension by simulator.

Consequences of Not Fixing

Due to extension, more cases may match the expression than intended.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location of the offending constant that is ?-extended.

To fix the violation, it is best to use the correct width and value specification, rather than depending on default extension. This is particularly important here because the implication of extension is that more cases may match than you intended.

Example Code and/or Schematic

Consider the following examples:

Specified Value	Equivalent Value	Violation Reported
8'b?0	8'b???????0	Yes as the value is ?-extended
9'h?0	9'b?????0000	Yes as the value is ?-extended
8'h?0	8'b????0000	No as the value is not ?-extended

Default Severity Label

Warning

Rule Group

Expression

Reports and Related Files

No related reports or files.

W561

A zero-width-based number may be evaluated as 32-bit number

Language

Verilog

Rule Description

The W561 rule flags zero-width based numbers.

A zero-width based-number (like **0'd0**) is treated as a zero and hence, may result in unwanted results.

It is possible to create a zero width-based number either by omitting the width or by using a zero-value macro to specify the width. Consider the following example in which a zero-value macro is used to specify the width:

```
‘define X 0
... a = ‘X’h3
```

In the above example, the based number will be evaluated as 32 bit number, which may not be what you intended.

Message Details

The following message appears at the location where a zero-width based number <*num*> is encountered:

Zero-width based number '<num>' may be evaluated as 32 bit number

Rule Severity

Warning

Suggested Fix

Check each such case to make sure that the behavior is what you intended.

W563

Reduction of a single-bit expression is redundant

Language

Verilog

Rule Description

The W563 rule flags unary reduction operations on single-bit expressions.

Unary reduction operators (unary **and** reduction operator (**&**), unary **or** reduction operator (**|**), and unary **xor** reduction operator (**^**)) reduce a vector to a single-bit. However, using the unary reduction operators on single-bit expressions is redundant.

NOTE: *The W563 rule supports generate-if, generate-for, and generate-case blocks.*

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location where a unary reduction operator *<op-name>* is used on a single-bit expression *<expr>*:

```
Use of unary reduction operator "<op-name>" on a 1-bit
expression "<expr>" is unnecessary.[Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical path.

Rule Severity

Warning

Suggested Fix

Examine the logic to make sure behavior is what you intended.

W575

Logical NOT operating on a vector

Language

Verilog

Rule Description

The W575 rule flags logical **NOT** operators used on vector signals.

Logical **NOT** evaluates its operand as a Boolean (**zero = false**, **non-zero = true**) and returns the logical inverse.

If you intent to perform a bit-wise inversion, use the bit-wise inverse (**~**) operator.

NOTE: *The W575 rule supports generate-if, generate-for, and generate-case blocks.*

Message Details

The following message appears at the location where a logical **NOT** operation is used on a multi-bit signal *<sig-name>* of width *<num>*:

`Logical NOT(!) operator used on a multibit(<num>) value: <sig-name>`

Rule Severity

Warning

Suggested Fix

Examine the logic to make sure behavior is what you intended.

W576

Logical operation on a vector

Language

Verilog

Rule Description

The W576 rule flags logical operations performed on vector signals.

It is particularly easy to confuse with the usage of operators, **&** and **&&**. The operations using these two operators (for example **a & b** and **a && b**) will in general return very different results. The operation using the **&** operator is a bit-wise **and** of two vectors which could evaluate to a vector with the number of bits set. However, the operation using the **&&** operator evaluates its arguments as Boolean (**zero = false**, **non-zero = true**) and returns the logical **and**, which will always be **1** or **0**.

NOTE: *The W576 rule supports generate-if, generate-for, and generate-case blocks.*

By default, the rule reports violation for the type parameters. Set the value of the *ignore_parameters* parameter to yes to not report violations for the type parameters.

Message Details

The following message appears at the location where a logical operator *<operator-name>* is used on a multi-bit signal *<sig-name>* of width *<num>*:

```
Logical operator (<operator-name>) used on a multibit (<num>) value: <sig-name>
```

Rule Severity

Warning

Suggested Fix

You should examine each of these cases to make sure you are using the correct operator.

MultipleDriver Rules

The SpyGlass lint product provides the following multiple driver related rules:

Rule	Flags...
W259	Signals that have multiple drivers but no associated resolution function
W323	Non-tristate inout nets that are driven in more than one **always** construct or module instance
W415	Non-tristate nets that are driven in more than one **always** construct or module instance
W415a	Signals that are multiply assigned in the same **always** construct
W552	Flip-flop outputs whose different bit-selects are driven in different sequential **always** constructs
W553	Nets whose different bit-selects are driven in different combinational **always** constructs

W259

Signal has multiple drivers

Language

VHDL

Rule Description

The W259 rule flags signals that have multiple drivers but no associated resolution function.

A signal is to be said to be driven by multiple drivers, if more than one concurrent statements (process, instance, conditional, and selected signal assignment statements etc.) contain signal transform for the same signal.

By default, only simple types like **`std_logic`** and **`std_logic`** vector have defined resolution functions. Signals with multiple drivers but no resolution function are driven to tristate or to value x.

The W259 rule currently does not handle multi-dimensional arrays, aggregate types, and record types.

NOTE: *The W259 rule supports generate-if and generate-for blocks.*

NOTE: *The W415 rule flags those data type that have built-in resolution function (for example,* **`std_logic`***). However, if the data type has no built-in resolution function, then the W259 rule is applicable to check for multiple drivers.*

Message Details

The following message appears at the location where a signal *<sig-name>* with multiple drivers but no resolution function is used:

```
Non-resolved signal '<sig-name>' has multiple drivers
[Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical path.

Rule Severity

Warning

Suggested Fix

Determine first if you really intended a **`WIRE-OR`**. If so, provide a

resolution function to resolve the connection.

W323

Multiply driven inout net

Language

Verilog

Rule Description

The W323 rule flags non-tristate inout nets that are driven by multiple drivers.

The W323 rule considers a net to be tristate if all the drivers in its immediate fan-in are tristate elements.

The W323 rule ignores nets driven from black boxes.

By default, the W323 rule ignores bus-holders unless the *strict* rule parameter is set.

NOTE: *You can enable the W323 rule by specifying the* **`set_goal_option addrules W323`** *command. However, this rule will not run if you set the fast rule parameter to* **`yes`** *and SpyGlass lint product is run.*

Message Details

The following message appears at the location where a non-tristate inout net *<net-name>* is found to be driven in more than one module instance:

```
Inout '<net-name>' is written in more than one place
```

Rule Severity

Warning

Suggested Fix

Most probably, there is an error in your logic. Ensure that only one driver drives the net, unless you actually intended a tristate net.

W415

Reports variable/signals that do not infer a tristate and have multiple simultaneous drivers

When to Use

Use this rule to identify variable/signals that do not infer a tristate and have multiple simultaneous drivers.

Rule Description

The *W415* rule reports variables/signals that do not infer tristate nets and have multiple simultaneous drivers.

The W415 rule does not report violation for tristate nets whose immediate fan-in drivers are tristate elements. However, the rule reports a violation when an enable signal for two tristate elements is same as both the elements get enabled at the same point of time.

This rule reports variables/signals that are driven in more than one sequential always/process constructs, instances, and/or concurrent statements. Such cases are errors unless the variable infers a tristate net.

Rule Exceptions

Following are the expectations to the W415 rule:

- The *W415* rule fails to run if you set the *fast* rule parameter to **`yes`** and SpyGlass lint product is run.
- By default, this rule does not report any violation if the multiple driven signals/variables are not loaded or they have hanging driver input.
- In VHDL, the *W415* rule reports those data types that have built-in resolution function, such as **`std_logic`**. However, if the data type has no built-in resolution function, the *W259* rule is applicable to check for multiple drivers.

Language

Verilog, VHDL

Parameters

- *strict*: By default, the *W415* rule ignores instances, bus-holders and top-level inout ports as drivers. Set this parameter to consider instances,

bus-holders and top-level inout ports as drivers. For all other modules, inout ports are always checked for rule violation irrespective of the value of this parameter.

NOTE: *A violation is reported even without the strict parameter, if two or more output terminals are connected together with an inout port. No violation is reported (even if the strict parameter is set), if an inout port is connected with the non-enabled tristate (disabled or the tristate having unknown value) driver.*

- *assume_driver_load*: Default value is **no**. This indicates the *W415* rule does not flag any violation, if the multiple driven signal/variable is not loaded or have hanging driver input. Set the value of the parameter to **yes** to report violations for such cases.
- *checkconstassign*: Default value is **no**. This indicates the *W415* rule ignores nets that are driven multiple times by the same constant value. Set this parameter to **yes** to report such cases.
- *check_bbox_driver*: Default value is **no**. Set this parameter to **yes** to consider the black-box as a valid driver.
- *ignore_greybox_drivers*: Default value is **no**. Set the value of the parameter to **yes** to consider greybox as a valid driver, if this parameter is set to 'yes' then rule will not consider greybox as a valid driver.
- *handle_equivalent_drivers*: The default value is no. Set the value of the parameter to yes to report violations when a net is simultaneously driven by two identical instances of two input gates of type BUFF, NOT, OR, NOR, AND, or, NAND and both instances are driven by same input signals.
- *show_connected_net*: The default value is no. Set the value of the parameter to yes to report adjacent nets to a fan-in terminal of a multi-driven net and help in preventing hierarchical waiver breaks.

Constraints

None

Tcl Attributes

- **is_multiple_driver**: This Tcl attribute returns the nets that have multiple drivers.

  For example:

```
sg_shell> set_pref dq_design_view_type flat
sg_shell> set net_iter [get_nets * -filter
"is_multiple_driver == true"]
```

For more details, refer to the **is_multiple_driver** attribute in the *Base Attributes* section of the *SpyGlass Tcl Shell Interface User Guide*.

Messages and Suggested Fix

Message 1

```
[WARNING]Signal '<sig-name>' has multiple simultaneous drivers
```

Potential Issues

The violation is reported when a signal has multiple simultaneous drivers.

Consequences of Not Fixing

A net that is not a tristate net should have a unique driver. If such nets are driven by more than one net, there is a possibility of an electrical conflict. For example, consider a net driven by outputs of two different AND gates. If one output is 0 and other output is 1, the driven net is in a non-digital state. Such cases lead to excessive current flows from one AND gate to another AND gate causing chip electrical failures, causing chip malfunction. The effect of a non-digital net can also propagate to its fan-out points. For such fan-out points, the net behaves as if it was floating resulting in further chip electrical and functional failures.

How to Debug and Fix

View the Incremental Schematic for the violation message. The Incremental Schematic shows the multiple driven net and a corresponding load for that net.

Most probably, there is an error in your logic. To fix the violation, ensure that variable/signal does not have multiple drivers, unless you actually intended tristate logic.

Message 2

```
[INFO] Please refer 'W415_Report.rpt' for driver details
```

Potential Issues

The information message is reported when the W415_Report report is generated.

Consequences of Not Fixing

This is an information message. There are no consequences of not fixing the message.

How to Debug and Fix

This is an information message. There is no debug or fix for the message.

Example Code and/or Schematic

Example 1

Consider the following example:

```
library ieee;
use ieee.std_logic_1164.all;
entity test is
port(
in1,in2 : in std_logic;
output  : out std_logic
);
end test;
architecture rtl of test is
begin
output <= in1 ;
output <= in2 ;
end rtl;
```

In the above example, the *W415* rule reports a violation as the signal, **`output`**, has multiple simultaneous drivers, **`in1`** and **`in2`**.

Default Severity Label

Warning

Rule Group

MultipleDriver

Reports and Related Files

None

W415a

Signal may be multiply assigned (beside initialization) in the same scope

Language

Verilog

Rule Description

The *W415a* rule reports signals that are assigned multiple times within the same block.

The rule reports a violation when an assignment overrides previous assignment other than the initialization statement within the same block.

If a signal is multiply assigned in parallel conditional blocks and if those conditions can be true simultaneously, the second assignment may override the first. The rule reports such assignments.

NOTE: *The rule supports the multi-line functionality.*

By default, the rule checks for all kinds of multiple assignments (both blocking and nonblocking) within the same scope. Use the *checknonblocking* rule parameter to restrict rule checking to nonblocking assignment statements. Use the *checkblocking* rule parameter to restrict rule checking to blocking assignment statements.

The rule only reports for assignments at the RTL description level irrespective of the logic inferred after synthesis.

NOTE: *The rule does not report a violation if a mutuallyexclusive* **`if`** *conditions exists and the conditional expression contains the* **`inside`** *operator.*

By default, the *ignore_reinitialization* parameter not set and the *W415a* rule reports violations for re-initialization when the *check_initialization_assignment* parameter is set to **`yes`**. Set the **`ignore_reinitialization`** parameter to **`yes`** to ignore violations for re-initialization assignments inside the **`for`** loops.

By default, the rule reports violations for signals which are inside if/else or case statement. Set the value of the *ignore_if_case_statement* parameter to yes to ignore violations for such signals.

By default, the rule reports a violation for the following cases:

- when an entire vector is initialized with an initialization value and a subset of the vector is overwritten.
- when an entire packed struct is initialized with an initialization value and a subset of the struct is overwritten.

Set the value of the *check_initialization_assignment* parameter to **yes** to disable the violations for the above cases.

However, at this parameter value (**yes**), the rule reports a violation when:

- a vector is initialized with an initialization value and the entire vector is assigned again with another value.
- a violation when a packed struct is initialized with an initialization value and the entire packed struct is assigned again with another value.

NOTE: *The rule does not report any violation for* **assign** *assignments in* **generate for** *blocks if the ignore_multi_assign_in_genforblock parameter is set to* **yes**.

Example 1

In the following example, the *check_initialization_assignment* parameter is set to **yes**:

```
reg [3:0] b;
    always @(*)
    begin
        if(inp)
        begin
            b = a;
            b[2] = 1'b0;
        end
    end
```

In this case, the vector **b** is initialized with an initialization value and a subset of the vector is overwritten. Therefore, the *W415a* rule does not report a violation in this case.

Example 2

In the following example, the *check_initialization_assignment* parameter is set to **yes**:

```
reg [3:0] b;
    always @(*)
    begin
```

```
        if(inp)
        begin
            b = a;
            b[3:2] = 1'b0;
        end
    end
```

In this case, the vector **b** is initialized with an initialization value and a subset of the vector is overwritten. Therefore, the *W415a* rule does not report a violation in this case.

Example 3

In the following example, the *check_initialization_assignment* parameter is set to **yes**:

```
reg [3:0] b;
    always @(*)
    begin
        if(inp)
        begin
            b = a;
            b = 1'b0;
        end
    end
```

In this case, the vector **b** is initialized with an initialization value, and the entire vector is assigned again with another value. Therefore, the *W415a* rule reports a violation in this case.

Example 4

In the following example, the *check_initialization_assignment* parameter is set to **yes**:

```
reg [3:0] b;
    always @(*)
    begin
        if(inp)
        begin
            b = a;
            b[3:0] = 1'b0;
        end
```

```
    end
```

In this case, the vector **b** is initialized with an initialization value, and the entire vector is assigned again with another value. Therefore, the *W415a* rule reports a violation in this case.

By default, the rule reports assignment to a variable inside a conditional construct if the variable is also being assigned non-static value in a nonblocking manner before the conditional construct as in the following examples:

Example 1

Consider the following example:

```
always @ (clk)
begin
  if (clk)
  begin
     out1<= in1; //Assignment of non-static value
     if(en)
       out1<= in2; // Violation
  end
end
```

Example 2

Consider the following example:

```
always @ (clk)
begin
  out1<= in1; //Assignment of non-static value
  if (clk)
  begin
    out1<= in1; //Violation
  end
end
```

In the above example, the rule reports a violation as the variable is assigned a non-static value.

The *W415a* rule does not report assignment to a variable inside a conditional construct if the variable is also being assigned static value in a nonblocking manner before the conditional construct as in the following examples:

Example 1

```
always @ (clk)
begin
  if (clk)
  begin
     out1<= 1'b0; //Assignment of static value
     if(en)
       out1<= in2; // No Violation
  end
end
```

Example 2

```
always @ (clk)
begin
  out1<= 1'b0; //Assignment of static value
  if (clk)
  begin
    out1<= in1; // No Violation
  end
end
```

If you set the *ignore_nonBlockCondition* parameter to **yes**, then no violation is reported for case mentioned above. For example,

```
...
a <= b;
if (C1)
  begin
    a <= b1; // ignore_nonBlockCondition set to yes
  end
...
```

The *W415a* rule does not report violation if two **if** conditions are mutually exclusive as only one path will be executed. For example:

```
if(sel)
  out = in1;
if(!sel)
 out = in2; //Only one condition will be true in above          //case

if(sel[1])
```

```
  out = in1;
if(!sel[1])
 out = in2;

if(sel==0)
  out=in1;
if(sel==1)
 out=in2;
```

If the conditions contain binary expression then simple conditions are checked. Otherwise, no checking is done. For example:

```
if(sel==2)
  out=in1;
if(sel>1)
  out=in2; //The conditions can be true simultaneously
```

If the left expression of binary expressions is complex or right expression of binary expression is not constant, no checking is done. For example:

```
if(sel==0)
  out = in1;
if(sel>a)
  out = in2; //No Condition checking as right
             //expression is not constant
if(sel+sel1>a)
  out = in3; //No Condition checking as left expression
              //is not simple expression
```

By default, the *W415a* rule does not report violation if the RHS value in an assignment is same as the RHS value in the previous assignment. Set the *reportsimilarassgn* rule parameter to **yes** to flag such cases. For example, the *W415a* rule will not report violation in the following case with the **reportsimilarassgn** rule parameter is set to **no**:

```
if(sel)
  q =a+b;
if(sel0)
  q = b+a;
if(sel1)
  q = 1;
```

```
if(sel2)
  q = 1;
```

NOTE: *Only simple expressions are checked for checking similar assignments. Complex expressions such as* **a+b+c** *and* **a+c+b** *are not checked.*

By default, the W415a rule reports violation for a signal assigned multiple times on the same line inside a for-loop irrespective of the expression on RHS of the assignment. Set the *ignore_multi_assign_in_forloop* parameter to **Yes** to skip rule checking inside a for-loop for multiple assignments on the same line.

In the following examples, the W415a rule reports violations for signal *a*. When the *ignore_multi_assign_in_forloop* parameter is set to **Yes**, no violations are reported.

Example 1

```
reg a, b, c;
  for(i = 0; i < 2; i = i + 1)
    begin
     a = a + 1;
    end
```

Example 2

```
reg a, b, c;
  for(i = 0; i < 2; i = i + 1)
    begin
     a = b + c;
    end
```

By default the *waiver_compat* parameter is set to **no**. If you set the value of this parameter to **yes** or **<rule-name>**, it ensures that the rule does not generate the line number information in the first run itself. Thus waivers work correctly even if the line numbers of the RTL gets changed in the subsequent runs.

By default the value of *ignore_bitwiseor_assignment* parameter is set to **no** and the rule reports a violation for a bitwise **or** assignment inside the **for** loop that is assigned multiple times on the same line. Set the value of the parameter to **yes** to not to report violation for such cases.

For example:

```
reg a, b, c;
for(i = 0; i < 2; i = i + 1)
begin
    a |= a + 1;
end
```

In the above snippet, by default the *W145a* rule reports violation for signal **a** assigned inside the **for** loop. But when the *ignore_bitwiseor_assignment* parameter is set to **yes**, no violation is reported for the **bitwise or** assignment inside the **for** loop.

The *W415a* rule checks a vector when it is initialized with an initialization value and the vector is assigned again with another value within a "for" loop.

For example, consider the following snippet:

```
module test (a, b, clk, y);
input [3:0] a, b;
input clk;
output [3:0] y;
reg [3:0] y;
integer i;

always @(clk) begin
    y <= 3'b000;
    for (i=0; i<=3; i=i+1)
        y[i] <= a[i] & b[i]; // warning
end
endmodule
```

For the above example, the rule reports the following violation message:

```
Signal y[0:3] is being assigned multiple times ( previous
assignment at line 9 ) in same always block [Hierarchy:
':test']
```

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

Message 1

The following message appears at the location where a signal *<sig-name>* is assigned another value after it has already been assigned a value at line number *<line-num>* in the same always construct:

```
Signal <sig-name> is being assigned multiple times (previous assignment at line "<line-num>") in same always block [Hierarchy: '<hier-path>']
```

Message 2

The following message appears at the location where a signal *<sig-name>* is assigned multiple values in the same for-loop and in the same always construct:

```
Signal <sig-name> is being assigned multiple times ( assignment within same for-loop ) in same always block [Hierarchy: '<hier-path>']
```

Rule Severity

Warning

Suggested Fix

Aside from overwriting an initial value, multiple overwrites of a value within a block is at least confusing. In some cases, it may indicate an error. Therefore, it is recommended to rewrite the code to remove multiple overwrites.

Examples

Consider the following examples where signal **out** is being assigned multiple times in the same **always** construct:

Example of Simple Multiple Assignments

Consider the following example:

```
always @(in1 or in2)
  begin
    out = in1;
```

```
    out = in2;
  end
```

Here, the previous assignment to signal **out** is overwritten by the second assignment. For this example, SpyGlass generates the following message:

```
Signal 'out' is being assigned multiple times in same always
block
```

Example of Conditional Multiple Assignments

Consider the following example:

```
always @(in1 or in2 or a or b)
  begin
    if (a)
      out = in1;
    if (b)
      out = in2;
  end
```

Here, the previous assignment to signal **out** is overwritten by the second assignment. Also, there is potential case of overwriting to signal **out**. For this example, SpyGlass generates the following message:

```
Signal 'out' is being assigned multiple times in same always block
```

Example of Multiple Assignments at Different Level

Consider the following example:

```
...
a = b;
if (C1)
  begin
    a = b1; //first assignment
    if (C12)
      a = b12;  //second assignment
  end
...
```

Since the assignments are at different level in nested conditional blocks, SpyGlass does not flag a violation.

Example of Multiple Assignments in Different Branches of case Construct

Consider the following example:

```
always @(in1 or in2)
  begin
    case (sel)
      1'b0 : out = in1;
```

```
      1'b1 : out = in2;
      default : out = 1'b0;
  end
```

Since, there is no overwriting to signal **out**, SpyGlass does not flag a violation.

Example of Multiple Assignments in Different Branches of If Statement

Consider the following example:

```
always @(in1 or in2 or c)
  begin
    if (c)
      out = in1;
    else
      out = in1;
  end
```

Since, there is no overwriting to signal **out**, SpyGlass does not flag a violation.

Example of Multiple Assignments in Same For-Loop and Same Always construct

Consider the following example:

```
reg [0:5] out2;
always @(in1)
begin
  for(i = 0; i < 2; i = i + 1)
  begin
    out2[3:4] <= in1[i];
  end
end
```

In the above example, the W415a rule reports a violation for the signal, **out2[3:4]**, because it is assigned multiple times within the same for loop and same always construct.

W552

Different bits of a bus are driven in different sequential blocks

Language

Verilog

Rule Description

The W552 rule flags bus outputs whose different bit-selects are driven in different sequential **always** constructs.

While such descriptions are allowed, they can be confusing and could lead to non-uniform handling of the bus in some conditions. Such descriptions also create a possibility of (simulation) race conditions on the bus.

For multidimensional arrays, the W552 rule flags if different bits of packed dimensions are driven in different sequential **always** constructs. For example, the W552 reports a violation in the following case:

```
reg [1:0][3:0]out;
always@(posedge clk1)
  out[1][1] = in1[1];
always@(negedge clk1)
  out[1][0] = in1[0];
always@(negedge clk2)
  out[0] = in1;
```

NOTE: *The W552 rule supports generate-if, generate-for, and generate-case blocks.*

NOTE: *The W552 rule does not check for unpacked arrays.*

NOTE: *The rule supports the multi-line functionality.*

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location where a bus output *<bus-name>* is declared whose different bit-selects are driven in different sequential always constructs:

```
Bus '<bus-name>' is driven inside more than one sequential
block. [Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical name of the containing scope excluding subprograms.

Rule Severity

Warning

Suggested Fix

Wherever possible, drive all bits of a bus from the same block.

W553

Different bits of a bus are driven in different combinational blocks

Language

Verilog

Rule Description

The W553 rule flags bus whose different bit-selects are driven in different combinational always constructs.

While such descriptions are allowed, they can be confusing and could lead to non-uniform handling of the bus in some conditions. Such descriptions also create a possibility of (simulation) race conditions on the bus.

NOTE: *The W553 rule supports generate-if, generate-for, and generate-case blocks.*

NOTE: *The rule supports the multi-line functionality.*

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

The following message appears at the location where a bus <*bus-name*> is first assigned whose different bit-selects are driven in different combinational always constructs:

```
net/bus '<bus-name>' is driven inside more than one
combinational block. [Hierarchy: ‘<hier-path>’]
```

Where, <*hier-path*> is the complete hierarchical name of the containing scope excluding subprograms.

Rule Severity

Warning

Suggested Fix

Wherever possible, drive all bits of a bus from the same block.

Simulation Rules

The SpyGlass lint product provides the following simulation related rules:

Rule	Flags...
W17	Arrays in sensitivity lists that are not completely specified
W122	Signal that is read in a combinational process/ always block, but is not included in the sensitivity list
W167	Modules that do not have any port interface
W456	Signals that are in the sensitivity list of a combinational **`always`** construct but are not completely read in the construct
W456a	Signals that are in the sensitivity list of a combinational process block but are not read in the process block
W488	Bus signals that are in the sensitivity list of an **`always`** construct but are not completely read in the construct
W502	variable that is present in the sensitivity list and is modified in the always block
W526	IF-ELSE constructs that should be changed to `case` constructs to improve performance

W122

A signal is read inside a combinational always block but is not included in the sensitivity list (Verilog)
A signal is read inside a combinational process but is not included in the sensitivity list (VHDL)

When to Use

Use this rule to identify the signals that are read inside a combinational **`always`** block or combinational process but are not included in the sensitivity list.

Description

Verilog

The *W122* rule reports violations for the signals that are read in a combinational always construct but are not present in the sensitivity list of the construct.

The rule also reports violations for the signals that are declared inside a generate block.

Refer to the *Determining Signals Required in the Sensitivity List* section for details on how SpyGlass determines the signals required in the sensitivity list.

The rule does not check the for loop indices because they do not represent real signals.

The rule reports violations for global signals or variables that are read in the tasks or functions called from the always blocks that have wildcard (**@***) sensitivity list. The wildcard infers sensitivity to only those signals that are directly referenced in the always block. It does not infer sensitivity to signals that are externally referenced in a function or a task that is called from the always block. Therefore, the wildcard is sensitive only to those signals that are passed to a function or a task.

NOTE: *The rule supports the multi-line functionality, generate-if block, and generate-for block.*

VHDL

The *W122* rule reports violations for the signals, which are read in a combinational process but are not present in the sensitivity list of the process.

NOTE: *The rule supports the multi-line functionality.*

Rule Exceptions

- The rule does not report violations for signals which are assigned constant values outside the always block using continuous assignment statement and are missing from the sensitivity list. The rule also does not report violations for the variables which are assigned values in tasks and then are read in the same **always** construct. These variables are connected to the output of a task.
- The rule will not report any violation for delay variable used inside the always blocks, which are not read inside the sensitivity list.

Language

Verilog, VHDL

Default Weight

10

Parameter(s)

Verilog

- *allow_clk_in_condition*: Default value is **no**. If you set this parameter to **yes**, the rule does not traverse a combinational block present inside a sequential block and does not report a violation in such cases.
- *simplesense*: Default value is **no**. Set the value of this parameter to **yes** to enable the rule to report the signals that are read in an **always** construct and being set in blocking assignment statement but are not present in the sensitivity list.
- *strict*: Default value is **no**. Set this parameter to **yes** (in the Atrenta Console) or **1** (in the batch mode) to report memories and delay variables that are read in the **always** construct but are not present in the sensitivity list.
- *report_all_messages*: Default value is **no**. Set this parameter to **yes** or rule name(s) to report the bit information in the violation messages.

VHDL

- *considerInoutAsOutput*: Default value is no. Set the value of the parameter to yes to consider the inout port as output port.

- *strict*: Default value is **no**. This indicates the *W122* rule reports a violation for a signal that is read in the **process** block and initialized to a value outside the **process** block but is not completely present in the sensitivity list. If you set this parameter to **yes**, the *W122* rule does not report a violation in such cases.

NOTE: *The W122 rule does not report a violation for a signal that is assigned a value outside the* **process** *block by using the concurrent assignment statement.*

- *ignoreSeqProcess*: Default value is **no**. Set this parameter to **yes** to ignore rule checking inside the sequential process blocks.

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message is displayed for the <*name*> signal or the <*name*> variable which is read in the **always** construct but is not present in the sensitivity list:

[WARNING] The signal/variable '<name>' (or some of its bits) read in the block is not in the sensitivity list [Hierarchy: '<hier-path>']

If the *report_all_messages* parameter is set, then the violation message is modified, as follows:

[WARNING] Signal/variable '<name>', that is read in this block, is not present in sensitivity list[Hier: <hier-path>]

Where, <*hier-path*> is the complete hierarchical path of the signal.

Potential Issues

A violation is reported for a signal or a variable, which is read in the **always** block but is not present in the sensitivity list.

Consequences of Not Fixing

Signals missing from the sensitivity list can lead to a mismatch between the pre- and post-synthesis simulations. The simulation only evaluates the changes in the combinational logic when the signals in the sensitivity list

change before synthesis. The synthesis process generates a logic which reads all the required values, whether they are present in the sensitivity list or not. Effectively, the missing signals are added to the sensitivity list. Therefore, the post-synthesis simulation results are different from the pre-synthesis simulation results.

How to Debug and Fix

To debug the violation, double-click the message. The HDL Viewer window highlights the line in which the specified signal or variable is read in the denominational **always** block. This signal is missing in the sensitivity list of the denominational **always** block. Confirm the same in one of the following ways:

- Scroll up the HDL window till you reach the **always** block and then check the sensitivity list.
- When the **always** block is very long, view the HDL in an editor and search backwards for the nearest **always** block, and then check the sensitivity list.
  To ensure that the pre-synthesis simulation results match the post-synthesis simulation results, add the missing signals to the sensitivity list of the combinational **always** construct.

VHDL

The following message is displayed for the *<construct-name>* construct of the *<construct-type>* type which is read in the process block but is not present in the sensitivity list:

`[WARNING] The <construct-type> '<construct-name>' (or some of its bits) read in the process is not in the process sensitivity list. [Hierarchy: '<hier-path>']`

Where, *<construct-type>* is a **port**, **signal**, or **variable** and *<hier-path>* is the complete hierarchical path of the containing process.

Potential Issues

A violation is reported for a construct, which is read in the process block but is not present in the sensitivity list.

Consequences of Not Fixing

Signals missing from the sensitivity list can lead to a mismatch between the pre- and post-synthesis simulations. The simulation only evaluates the

changes in the combinational logic when the signals in the sensitivity list change before synthesis. The synthesis process generates a logic which reads all the required values, whether they are present in the sensitivity list or not. Effectively, the missing signals are added to the sensitivity list. Therefore, the post-synthesis simulation results are different from the pre-synthesis simulation results.

How to Debug and Fix

To debug the violation, double-click the violation message. The HDL Viewer window highlights the line, where, the specified signal is read in the denominational **process** block. This signal is missing in the sensitivity list of the denominational **process** block. You can confirm the same in any one of the following ways:

- Scroll up the HDL window till you reach the process and check the sensitivity list.
- If the **process** block is a very long block, view the HDL in an editor, and, search backwards for the nearest **process** block and then check the sensitivity list.

To resolve the violation, add the reported signals to the sensitivity list.

Example Code and/or Schematic

Verilog

Example 1

Consider the following example:

```
module mod_NotInSensOM_Test1 ( in1, in2, out1 );
input in1, in2;
output out1;
reg out1;

  always @(in1)
  begin
    out1 = in1&in2;
  end

endmodule
```

The *W122* rule reports a violation for the above example because the **in2**

variable, which is read in the combinational **always** block, is not present in the sensitivity list.

Example 2

Consider the following example:

```
module chip (input wire [ 7:0] a, b,
  input wire [15:0] max_prod,
  input wire [ 8:0] max_sum,
  input wire error,
  output logic [ 8:0] sum_out,
  output logic [15:0] mult_out);

  function [8:0] mult (input [7:0] m, n);
    mult = 0;
    if (!error) // error is an external signal
      mult = m * n;
      if (mult > max_prod)
      //max_prod is an external signal
        mult = max_prod;
  endfunction
  task sum (input [7:0] p, q, output [8:0] sum_out);
    sum_out = 0;
    if (!error) // error is an external signal
      sum_out = p + q;
      if (sum_out > max_sum)
      // max_sum is an external signal
        sum_out = max_sum;
  endtask

  always @* begin
    // @* will only be sensitive to a and b
    sum(a, b, sum_out);
    // @* will not be sensitive to max_prod,
    // max_sum or error
    mult_out = mult(a, b);
  end
endmodule
```

In the above example, the *W122* rule reports a violation because the global signals or variables, which are read in the tasks or functions called from the **always** blocks, have wildcard (**@***) sensitivity list.

Example 3

Consider the following example:

```
module test (clk,in,in1,out);
input clk,in,in1;
output out;
reg out;

always@(posedge clk)
begin
if (clk)
    out = in ^ in1;
end
endmodule
```

In the above example, the *W122* rule, by default, reports violations for signals **in** and **in1**. If you set the *allow_clk_in_condition* parameter to **yes**, the rule does not report any violation.

VHDL

Example 1

Consider the following example where the signal **rst** is read in the process block **test** (in architecture **arc** of entity **ent**) but is not present in the sensitivity list of the process block:

```
test: process(clk)
  begin
    if (rst = '0') then
        out1 <= "0000";
    elsif (clk'event and clk = '1') then
        out1 <= in1;
    end if;
end process test;
```

For this example, SpyGlass generates the following message:

```
The in port 'rst' (or some of its bits) read in the process is
```

```
not in the process sensitivity list. [Hierarchy:
':ent(arc):test:']
```

To resolve the violation, rewrite the design to include the signal **rst** in the sensitivity list as follows:

```
test: process(rst, clk)
```

Example 2

Consider the following example:

```
entity test is
port (d1 : in std_logic_vector (2 downto 0);
      d2  : in std_logic;
      o1  : out std_logic_vector (2 downto 0));
end test;
architecture behav of test is
begin
 process(d1(2),d2)
  begin
      for i in 1 to 1 loop
      o1(1) <= d1(i+1) and d2;
    end loop;
   end process;
end behav;
```

In the above example, the signal **d1** is read in the process block and is also present in the sensitivity list. However, note that the *W122* rule evaluates only single-level binary operations, such as (**d1(i+1)**). Any level more than one for binary operations are not evaluated (for example, **d1(1+i+1)**).

Example 3

In the following example, the *W122* rule reports a violation for the **SIG1** signal because it is initialized to a value outside the process block:

```
entity ent24 is
 port(A : in BIT_VECTOR(0 to 3); Z : out BIT_VECTOR(0 to 3));
end ent24;
architecture arch24 of ent24 is
 signal SIG : BIT_VECTOR(0 to 3);
```

```
 signal C : BIT_VECTOR(0 to 3);
 signal SIG1 : BIT_VECTOR(0 to 3) := "0010";
 begin
  SIG <= (OTHERS => '0');
  process (A(0 to 1),SIG(0 to 1),SIG1(0 to 1))
   begin
    for NUM in 0 to 3 loop
       C(NUM) <= SIG(NUM);
       Z(NUM) <= SIG1(NUM);
    end loop;
   end process;
end arch24;
```

The *W122* rule does not report a violation for the **SIG** signal because it is assigned a value outside the **process** block by using the concurrent assignment statement.

Default Severity Label

Warning

Rule Group

Simulation

Reports and Related Files

No related reports or files.

W167

Module has no input or output ports

Language

Verilog, VHDL

Rule Description

The W167 flags modules that do not have any port interface (no input or output ports).

Message Details

Verilog

The following message appears at the location where a module *<module-name>* is defined without any inputs or outputs:

```
Module <module-name> has no ports
```

VHDL

The following message appears at the location where a design unit is defined without any inputs or outputs:

```
Module has no input/output ports
```

Severity

Warning

Suggested Fix

If this is a top-level testbench, there is nothing to fix. Otherwise you can only read to and write from this design unit through global variables, which is unsynthesizable. It may sometimes be OK for simulation monitors (though would still be considered by many to be poor coding style).

W456

A signal is included in the sensitivity list of a combinational always block but not all of its bits are read in that block (Verilog)
A signal is included in the sensitivity list of a process but not all of its bits are read in that block (VHDL)

Language

Verilog, VHDL

Rule Description

Verilog

The W456 rule flags signals that are in the sensitivity list of a combinational always construct but are not completely read in the construct.

See *Determining Signals Required in the Sensitivity List* for details of how SpyGlass determines which signals are required in the sensitivity list.

While such signals do not affect the functionality, they slow down the design simulation. Also, the logic related to a signal in the sensitivity list may have been accidentally removed or not added, which results in wrong functionality. Many simulators re-evaluate the always construct on each change in a redundant signal, even though the evaluation itself is redundant.

By default, the W456 rule does not report messages for constants present in sensitivity lists. Set the *strict* rule parameter to flag constants also.

By default, the W456 rule does not report bit information with the messages. Set the *report_all_messages* parameter to **yes** or rule name(s) to report the bit information in the violation messages.

NOTE: *The W456 rule supports generate-if and generate-for block.*

VHDL

The W456 rule flags signals that are in the sensitivity list of a process construct but are not completely read in the construct.

While such signals do not affect the functionality, they slow down the design simulation. Also, the logic related to a signal in the sensitivity list may have been accidentally removed or not added, which results in wrong functionality. Many simulators re-evaluate the process construct on each change in a redundant signal, even though the evaluation itself is redundant.

In combinational process constructs, any signal (as a whole or some of its bits) that is not read in the process construct but included in the sensitivity list, is flagged.

In sequential process constructs, signals other than the clock and the reset signals are not required in the sensitivity list even if they are read in that process construct and hence, are flagged.

Message Details

Verilog

The following message appears at the location of the sensitivity list of a combinational always construct where signal/variable *<name>* is present in the sensitivity list of the always construct but is not completely read in the always construct:

[WARNING] Either all or some of the bits of the signal/variable '<name>' are not required in sensitivity list, since they are not read in always block [Hierarchy: '<hier-path>']

If the *report_all_messages* parameter is set, then the violation message is modified, as follows:

[WARNING] signal/variable '<name>' is not required in sensitivity list, since it is not read in always block [Hierarchy: '<hier-path>']

Where, *<hier-path>* is the complete hierarchical path of the containing scope.

VHDL

The following message appears at the location of the sensitivity list of a combinational process construct where a signal or input port *<name>* is present in the sensitivity list of the process construct but is not completely read in the process construct:

[WARNING] The <type> '<name>' may not be required in sensitivity list as not all of its bits are read in process [Hierarchy: '<hier-path>']

Where, *<type>* can be **signal** or **inport** and *<hier-path>* is the complete hierarchical path of the containing scope.

The following message appears at the location of the sensitivity list of a sequential process construct where a signal or input port *<name>* is present but is not required in the sensitivity list of the process construct:

```
The <type> '<name>' is not required in sensitivity list of the sequential process [Hierarchy: '<hier-path>']
```

Where, *<type>* can be **signal** or **in port** and *<hier-path>* is the complete hierarchical path of the containing scope.

Rule Severity

Warning

VHDL Examples

Consider the following example of a sequential process construct:

```
test: process(clk, rst, in1)
  begin
    if (rst = '0') then
        out1 <= "0000";
    elsif (clk'event and clk = '1') then
        out1 <= in1;
    end if;
end process test;
```

While signal **in1** is being read in the sequential process construct, it is not the clock signal or the reset signal and hence, is not required in the sensitivity list.

Suggested Fix

Verilog

This is partly a question of taste. If you specify the whole bus and some bits are not read in the block, you may get a warning that the sensitivity list is over-specified. In these cases, it is not possible to satisfy both requirements simultaneously, so you must decide which requirement you want to ignore. From an implementation point of view, it is safest to ignore this rule.

VHDL

Remove variables from the sensitivity list if they are not required. This will

help improve simulation performance. It will have no effect on synthesis behavior.

W456a

A signal is included in the sensitivity list of a combinational always block but none of its bits are read in that block (Verilog)
A signal is included in the sensitivity list of a combinational process block but none of its bits are read in that block (VHDL)

Language

Verilog, VHDL

Rule Description

Verilog

The W456a rule flags signals that are in the sensitivity list of a combinational always construct but are never read in the construct.

See *Determining Signals Required in the Sensitivity List* for details of how SpyGlass determines which signals are required in the sensitivity list.

NOTE: *The W456a rule is compatible with Design Compiler behavior. Therefore, this rule will report violation only when none of the bits of signal are read in a combinational always block.*

While such signals do not affect the functionality, they slow down the design simulation as simulators will re-evaluate the block on each change in the variable. Also, the logic related to a signal in the sensitivity list may have been accidentally removed or not added, which results in wrong functionality.

By default, the W456a rule does not report messages for constants present in sensitivity list. Set the *strict* rule parameter to flag constants also.

By default, the rule checking is performed for all branches of *if-elseif-else* statement irrespective of the evaluated condition. Set the *disable_rtl_deadcode* parameter to **yes** to stop rule checking for disabled *if-elseif-else* branch.

NOTE: *The W456a rule supports generate-if and generate-for block.*

VHDL

The W456a rule flags signals that are in the sensitivity list of a process construct but are never read in the construct.

While such signals do not affect the functionality, they slow down the design simulation as simulators will re-evaluate the block on each change

in the variable. Also, the logic related to a signal in the sensitivity list may have been accidentally removed or not added, which results in wrong functionality.

NOTE: *The W456a rule is compatible with Design Compiler behavior. Therefore, this rule will report violation only when the whole bus is present in the sensitivity list of a process and none of its bits are read in that process.*

For sequential process constructs, signals other than the clock and the reset signals are not required in the sensitivity list even if they are read in that process construct and hence, are flagged.

By default, the *W456a* reports violations for all the records present in the process sensitivity list and whose elements are not read in the process block. Set the *checkfullrecord* parameter to **yes** to not to report violation for a record if any of its element bits is read.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Message Details

Verilog

The following message appears at the location of the sensitivity list of a combinational **always** construct where the signal or variable *<name>* is present in the sensitivity list of the always construct but is never read in that always construct:

```
The signal/variable '<name>' is not required in sensitivity
list as none of its bits is read in always block [Hierarchy:
‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical path of the containing scope.

VHDL

The following message appears at the location of the sensitivity list of a combinational process construct where an object *<name>* of type *<type>* is present in the sensitivity list of the process construct but is never read in that process construct:

```
The <type> '<name>' is not required in sensitivity list as none
of its bits is read in process [Hierarchy: ‘<hier-path>’]
```

Where, <*type*> can be **in port** or **signal** and <*hier-path*> is the complete hierarchical path of the containing process.

The following message appears at the location of the sensitivity list of a sequential process construct where a signal or input port <*name*> is present but is not required in the sensitivity list of the process construct:

`The <type> '<name>' is not required in sensitivity list of the sequential process [Hierarchy: '<hier-path>']`

Where, <*type*> can be **signal** or **in port** and <*hier-path*> is the complete hierarchical path of the containing scope.

Rule Severity

Warning

Suggested Fix

Remove these signals from the sensitivity list.

Examples

Verilog

Consider the following example:

```
module top( in1, in2, out1 );
    input in1, in2;
    output out1;
    reg out1;

    always @(in1 or in2)
    begin
        out1 = ~ in2;
    end

endmodule
```

In the above example, the *W456a* rule reports a violation because the *in1* is included in the sensitivity list of the combinational **always** block but none of it's bits are read in that block. The following message is reported by this example:

```
Either all or some of the bits of signal/variable 'in1' are not
required in sensitivity list, since they are not read in always
block [Hierarchy: ':top:']
```

VHDL

Consider the following example:

```
entity top is
port(in1: in bit;
     in2: in bit;
     out1: out bit
    );
end top;

architecture behav of top is
begin
     process(in1, in2)
     begin
          out1 <= not in2;
     end process;

end behav;
```

In the above example, the *W456a* rule reports a violation because the *in1* is included in the sensitivity list of the combinational **process** block but none of it's bits are read in that block. The following message is reported by this example:

```
The in port 'in1' may not be required in sensitivity list as
not all of its bits are read in process [Hierarchy:
':top(behav)::']
```

W502

Ensure that a variable in the sensitivity list is not modified inside the always block

When to Use

Use this rule to identify the signals in the sensitivity list that are modified inside the same always block.

Description

The W502 rule reports violation for the signal/variable that is present in the sensitivity list and is modified in the same always block.

NOTE: *The W502 rule supports generate-if and generate-for block.*

Language

Verilog

Default Weight

5

Parameter(s)

- *check_implicit_senselist*: The default value is **no**. Set the value of the parameter to **yes** to report violation for always @* and always_comb.
- *report_all_messages*: Default value is **no**. Set this parameter to **yes** or rule name(s) to report the bit information in the violation messages.

Constraint(s)

None

Messages and Suggested Fix

The following message appears when a signal/variable *<var-name>* is encountered that is present in the sensitivity list and is being modified in that same the always block:

```
[WARNING] The signal/variable '<var-name>' is modified inside always block [Hierarchy: ‘<hier-path>’]
```

Where, *<hier-path>* is the complete hierarchical path of the containing

scope.

Potential Issues

Violation may arise when a variable in the sensitivity list is modified inside the **always** block.

Consequences of Not Fixing

Updating a signal from sensitivity in the same always block may result in erroneous simulation results.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the statement where the signal mentioned in the violation message is modified (set) inside the always block. The same signal is also present in the sensitivity list of the same always block, which can be verified using one of the following ways:

- Scroll up the HDL window, till you reach the **always** block. Verify the presence of the signal in the sensitivity list of this **always** block.
- If the **always** block is a long block, view the HDL in an editor, and, search backwards for the nearest **always** block. Verify the presence of the signal in the sensitivity list of this **always** block.

To fix the violation, ensure that the signal is not present in the sensitivity list, or the signal is not set in the always block.

Example Code and/or Schematic

Consider the following example:

```
module test(out1,inp1);
output out1;
input inp1;
reg out1;

always@(inp1 or out1)
  begin
    out1 <= inp1;
  end
```

In the above example, signal **out1** read in the sensitivity list is getting modified inside the same **always** block.

Default Severity Label

Warning

Rule Group

Simulation, Lint_Elab_Rules

Reports and Related Files

No related reports or files.

W526

Use case statements rather than if/else, where feasible, if performance is important

Language

Verilog, VHDL

Rule Description

Verilog

The W526 rule flags those if-else constructs that should be changed to case constructs to improve simulation performance.

The W526 rule assumes the following:

- If the condition expressions of a nested if-else construct test the same signal **n** times, the nesting level of the construct is assumed to be **n**.
- A signal is assumed to be tested if the signal and only the signal appears on the LHS of a comparison operation (**==**, **<=**, **>=**, **!=** etc.).

The W526 rule flags an if-else construct only if the nesting level is more than 3, and the same value is being tested in each branch. You can change the nesting level (default limit of 3) to any required value by changing the rule definition given in the PERL ruledeck file.

A chain of if-else statements on the same test signal is inferred as a priority encoder. The same logic build using case statement is inferred as a multiplexer, which is likely to simulate faster than a priority encoder.

Using a case construct instead of if-else construct is meaningful only if the same value is being tested in each branch of the if-else construct.

VHDL

The W526 rule flags those if-then-else constructs that should be changed to case constructs to improve simulation performance.

A chain of if-then-else statements on the same test signal is inferred as a priority encoder. The same logic build using case statement is inferred as a multiplexer, which is likely to simulate faster than a priority encoder.

Using a case construct instead of if-then-else construct is meaningful only if the same value is being tested in each branch of the if-then-else construct.

The W526 rule flags an if-then-else construct only if the nesting level is more

than 3, and the same value is being tested in each branch.

Message Details

Verilog

The following message appears at the first line of a rule-violating if-else construct:

```
Case statement is recommended over deeply nested 'if'statement
```

VHDL

The following message appears at the first line of an if-then-else construct for condition clause <*name*> if the number of branches is more than 3:

```
Deeply nested if statement (condition '<name>') could be
rewritten as a case statement
```

Rule Severity

Guideline

Suggested Fix

Generally it is best to recode deeply nested if statements as case statements unless you actually need a priority encoder or the logic is not in a performance-critical part of the circuit.

Examples (Verilog)

Consider the following if-else construct that has four levels of nesting as the same signal **sel** is being compared four times in the conditional expressions:

```
if (sel == 4'h0 )
    out = (in1 + int1) - P1;
else if (sel == 4'h1 )
    out = in2 + (int2 - P2);
else if (sel == 4'h2 )
    out = ((in3 + int1) - `M1) + (in2 + (int2 - P2));
else if (sel == 4'h3 )
    out = (data1 + (int2 - `M2));
else
    out = int2;
```

For this example, SpyGlass generates the W526 rule message.

Examples (VHDL)

Consider the following example if-then-else construct that has more than 3 branches and the same value (**x**) is being tested in each branch:

```
if ( x = "000" ) then y <= a and b;
elsif ( x = "001" ) then y <= a or b;
elsif ( x = "010" ) then y <= a nand b;
elsif ( x = "011" ) then y <= a xor b;
else y <= '0';
end if;
```

For this example, SpyGlass generates the following message:

```
Deeply nested if statement (condition 'x') could be rewritten
as a case statement
```

To fix this violation, you can rewrite the above example using the case construct as follows:

```
case (x) is
  when "000" => y <= a and b;
  when "001" => y <= a or b;
  when "010" => y <= a nand b;
  when "011" => y <= a xor b;
  default   => y <= '0';
end case;
```

Event Rules

The SpyGlass lint product provides the following event related rules:

Rule	Flags...
W218	Event expressions that check for edge on a multi-bit signal
W238	**always** construct where both blocking assignments and nonblocking assignments are used
W245	Reduction OR operator (|) or logical OR operator (||) used in the sensitivity list of an **always** construct
W253	Data event variables specified with an edge in a timing check system task call
W254	Reference event variables specified without an edge in a timing check system task call
W256	Notifiers that are not single-bit registers
W326	**event** variables used with edges
W421	**always** constructs that have neither a sensitivity list nor an event control statement
W503	**event** variables that are never triggered

W238

Mixing combinational and sequential styles

Language

Verilog

Rule Description

The W238 rule flags always construct where both blocking assignments and nonblocking assignments are used.

In general, you should use blocking assignments in combinational always constructs and non-blocking assignments in sequential always constructs. Mixing the two assignment types in the same always construct can lead to mismatches between pre- and post-synthesis simulation results. The task calls are considered as blocking assignments, so the violations are reported accordingly.

Message Details

The following message appears at the location where a blocking (non-blocking) assignment is encountered in an always construct after a non-blocking (blocking) assignment has already been found:

`Mixing combinational and sequential styles`

For a non-blocking or blocking assignment, the previous respective blocking or non-blocking assignment is also highlighted.

Rule Severity

Warning

Suggested Fix

Change all non-blocking assignments (**<=**) in combinational blocks to blocking (**=**) assignments. Change all blocking assignments in sequential blocks to non-blocking assignments.

Examples

In the following example, the task calls are considered as blocking assignments, so the violations are reported:

```
module top(in, in2, in3, en, out, clk);
output out;
  input in, en, clk, in2, in3;
  reg out;
  wire a, b;

always_ff @ (posedge clk)
begin
out <= a;   //non blocking
convert(a, out);    //blocking
out <= a;   //non blocking
convert(a, out);    //blocking
end

task convert;
input  temp_in1;
output  temp_out1;
begin
    temp_out1 <= temp_in1;
end
endtask

endmodule
```

W245

Probably intended "or", not "|" or "||" in sensitivity list

Language

Verilog

Rule Description

The W245 rule flags bit-wise **or** operator (|) or logical **or** operator (||) used in the sensitivity list of an always construct.

As the sensitivity list of an always construct should be sensitive to the events in the control signals, it is recommended to use the or operator.

The expression **@(a || b)** is generally not equivalent to **@(a or b)**. The first expression evaluates to the value **a||b** and triggers the event when that value changes. This is generally not the design intent. The second expression triggers when either signal **a** or signal **b** changes which is generally the design intent.

NOTE: *The W245 rule supports generate-if, generate-for, and generate-case blocks.*

Message Details

The following messages appear at the location of the sensitivity list of an always construct where the bit-wise **or** operator (|) is used:

```
Probably intended "or" instead of "|" in sensitivity list
```

The following message appears at the location of the sensitivity list of an always construct where the logical **or** operator (||) is used:

```
Probably intended "or" instead of "||" in sensitivity list
```

Rule Severity

Warning

Suggested Fix

If you meant the first case, precompute **a || b**, then use that value in the sensitivity list. Only use **or** in the sensitivity list.

Examples

Consider the following example:

```
always @( clk )
begin
...
end

always @( clk | clk1)
begin
...
end
```

In the above snippet, the *W245* rule reports a violation because the bit-wise or operator (|) or logical or operator (||) is used in the sensitivity list of the **always** construct that is highlighted in red.

```
Probably intended "or" instead of "||" in sensitivity list
```

W253

Data event has an edge

Language

Verilog

Rule Description

The W253 rule flags data event variables that have been specified with an edge in a timing check system task call.

For example, the **$setup** task data variable should not be an edge.

Message Details

The following message appears at the location where the data event variable <*var-name*> is used with an edge specification in a timing check system task call:

```
Timing check data event: '<var-name>' should not use edge specification
```

Rule Severity

Warning

Suggested Fix

Replace the edge specification with a legal data variable.

Example

The following example of timing check system task call violates the W253 rule as the data variable (**out1**) has been specified with an edge specification:

```
$setup(posedge out1, posedge clk, 5, notifier);
```

W254

Reference event does not have an edge

Language

Verilog

Rule Description

The W254 rule flags reference event variables that have been specified without an edge in a timing check system task call.

For example, the **$setup** task reference variable must be an edge.

Message Details

The following message appears at the location where the reference event variable *<var-name>* is used without an edge specification in a timing check system task call:

`Timing check reference event: '<var-name>' should use edge specification`

Rule Severity

Warning

Suggested Fix

Replace the reference specification with a legal edge variable.

Examples

The following example of timing check system task call violates the W254 rule as the reference event (**clk**) does not use an edge specification:

```
$setup(out2, clk, 2, notifier);
```

W256

A notifier must be a one-bit register

Language

Verilog

Rule Description

The W256 rule flags notifiers that are not single-bit registers.

A notifier is the last argument in a timing check system task (**$setup**, **$hold**, etc.) and must be a single-bit register.

NOTE: *By default, the W256 rule is switched off. The functionality of this rule will now be covered by STX_VE_1366 rule. For more details on the STX_VE_1366 rule, refer to SpyGlass® Built-In Rules Reference Guide.*

Message Details

The following message appears at the location where a notifier *<noti-name>* that is not a single-bit register, is used:

```
notifier '<noti-name>' must be a single-bit register
```

Rule Severity

Warning

Suggested Fix

Replace the notifier specification with a legal single-bit register.

Examples

The following example of timing check system task call violates the W256 rule as the notifier is not a single-bit register:

```
...
wire [1:0]notifier = 2'b10;
...
$setup(out2, clk, 2, notifier);
```

W326

Event variable appearing in a posedge/negedge expression

Language

Verilog

Rule Description

The W326 rule flags **event** variables used with edges.

Events do not have edges and therefore, should not appear in edge-based expressions.

NOTE: *The W326 rule is switched off by default. You can enable this rule by either specifying the* **set_goal_option addrules W326** *command or by setting the verilint_compat rule parameter to* **yes**.

Message Details

The following message appears at the location where an **event** variable is used in an edge-based expression (**posedge** or **negedge** expressions):

```
Event variable should not appear in posedge/negedge expressions
```

Rule Severity

Fatal

Suggested Fix

If you intend to make the code synthesizable, trigger on edges of signals.

Examples

Consider the following example where an **event** variable is used with an edge-based expression:

```
module test (clk1, in, cntr, out);
  input clk1, in;
  input [1:0] cntr;
  output out;
```

```
  event test;
  reg a_1, a_2;

  always @(clk1 or test)
    a_2 = in;

  always @(posedge test)
    a_1 <= in;

  assign out = a_1 + a_2 ;
endmodule
```

For this example, SpyGlass generates the following message:

```
Event variable should not appear in posedge/negedge expressions
```

W421

Reports “always” or “process” constructs that do not have an event control

When to Use

Use this rule to identify **always** or **process** constructs that do not have an event control.

Description

Verilog

The *W421* rule reports always constructs that neither have a sensitivity list nor an event control statement.

VHDL

The *W421* rule reports process constructs that neither have a sensitivity list nor a wait statement.

Language

Verilog, VHDL

Parameters

None

Constraints

None

Messages and Suggested Fix

Verilog

The following message appears for an always construct that has neither a sensitivity list nor an event control statement.

```
[WARNING] No event control (@) in always block
```

Potential Issues

Violation may arise when an always block does not have an event control.

Consequences of Not Fixing

With no associated event statement, an always block remains active and no other block can start execution. This can result in the simulator hanging on the design.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the **always** block with no event control is declared.

Unless the **always** block is the very top process in the analysis and you are depending on an internal task or procedure to terminate simulation, specify an event control statement for the corresponding violating **always** block.

VHDL

The following message appears for a process construct that has neither a sensitivity list nor a wait statement.

```
[WARNING] No wait statement or sensitivity list in process
```

Potential Issues

Violation may arise when a process construct does not have a **wait** statement.

Consequences of Not Fixing

With no associated event/wait statement, a process construct is always active and no other block can start execution. This can result in the simulator hanging on the design.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the **process** block with no event control is declared.

Unless the **process** block is the very top process in the analysis and you are depending on an internal task or procedure to terminate simulation, specify a wait statement for corresponding violating process construct.

Example Code and/or Schematic

Consider the following example:

```
entity ent_test is
  port(a: in bit;
       b: in bit;
```

```
        z: out bit
  );
end ent_test;

architecture behav of ent_test is
begin
  process
  begin
    z <= a or b;
  end process;
end behav;
```

In the above example, the *W421* rule reports a violation as the **process** statement is declared without an event control.

Default Severity Label

Warning

Rule Group

Event

Reports and Related Files

None

Loop Rules

The following loop rules have been deprecated:

W478

The SpyGlass lint product provides the following loop rules:

Rule	Flags...
W66	**`repeat`** constructs with non-static control expressions
W352	**`for`** constructs with condition expression that evaluate to a constant
W478	This rule has been deprecated
W479	**`for`** constructs where the control expression does not set the value of the variable used in the step expression or always sets it to a constant value
W480	**`for`** constructs where the loop index variable evaluates to a non-integer
W481a	**`for`** constructs where the variable used in the step expression is not used in the condition expression
W481b	**`for`** constructs where the variable used in the initialization expression is not same as the variable used in the step expression

W66

Ensure that a repeat construct has a static control expression

When to Use

Use this rule to identify **repeat** constructs with non-static control expressions.

Rule Description

The W66 rule reports violation for **repeat** constructs with non-static control expressions.

NOTE: *The W66 rule supports generate-if, generate-for, and generate-case blocks.*

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a non-static **repeat** construct control expression is encountered:

`[WARNING] Unsynthesizable repeat loop because repeat expression is not constant.[Hierarchy: ‘<hier-path>’]`

Potential Issues

The number of times a **repeat** construct is executed should be a static constant value.

Consequences of Not Fixing

A variable value for **repeat** construct execution is not synthesizable using

simple logic.

If a variable is used in the control part of a loop, the synthesis tool will not have a deterministic way to synthesize the logic. Hence, this will not assure gate level integrity, though RTL may simulate as expected.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the **`repeat`** statement whose **`condition`** part is not constant.

Verify the variable/signal causing this non-static **`repeat`** construct control expression.

To fix the violation, you may need to recode this as multi-cycle logic. This is because there is no easy way to implement a variable bound loop in synthesizable logic.

Example Code and/or Schematic

Consider the following example:

```
module mod(i1, en, o1);
  input i1, en;
  output o1;
  reg o1;

  always@(en or i1)
    repeat(en)
      o1 = i1;

endmodule
```

In the above example, the W66 rule reports a violation as the **`repeat`** expression is not constant, which causes an unsynthesizable **`repeat`** loop.

Default Severity Label

Warning

Rule Group

Loop

Reports and Related Files

No related reports or files.

W352

Reports “for” constructs with condition expression

When to Use

Use this rule to identify **for** constructs with condition expression.

Description

The *W352* rule reports **for** constructs with condition expression that evaluates to a constant.

NOTE: *The W352 rule supports generate-for block.*

Language

Verilog

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location where a **for** construct with constant control expression is encountered.

```
[WARNING] The 'for' condition is constant - the loop will
either never execute or never terminate
```

Potential Issues

Violation may arise when a **for** construct is used in a condition expression that evaluates to a constant.

Consequences of Not Fixing

When the **for** construct has a constant condition expression, the construct either exits immediately without doing anything or never exits.

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the

line where a **for** loop with a constant control expression is specified.

To fix the violation, rewrite the condition.

Example Code and/or Schematic

Example 1

Consider the following example;

```
module mod(out, in, clk, reset);

  output [3:0] out;
  input [3:0] in;
  input clk;
  input reset;
  reg [3:0] out;
  integer i;

  always @ (posedge clk or negedge reset)
    for (i = 0; 4; i = i + 1)
      if ( !reset )
        out[i] = 1'b0;
      else
        out[i] = in[i];


endmodule
```

In the above example, the W352 rule reports a violation as the **for** condition is constant. The **for** loop will either never execute or never terminate.

The conditional expression of the following **for** loop is a constant:

```
for (i=0 ; 2; i=2)
...
```

Example 2

Consider the following example:

```
`define ITER 5
module W352_mod1(in1, in2, out1, out2, out3, out4);
```

```
input [5 : 0] in1, in2;
output [5 : 0] out1, out2, out3, out4;
reg [5 : 0] out1, out2, out3, out4;
parameter COND = 5;
integer i;

always@(in1 or in2)
   for(i = 0 ; `ITER; i = i + 1)
     out1[i] = in1[i] | in2[i];

always@(in1 or in2)
   for(i = 0 ; COND; i = i + 1)
     out2[i] = in1[i] | in2[i];

always@(in1 or in2)
   for(i = 0 ; 5; i = i + 1)
     out3[i] = in1[i] | in2[i];

always@(in1 or in2)
   for(i = 0 ; i <= 5; i = i + 1)
     out4[i] = in1[i] | in2[i];

endmodule
```

In the above example, the W352 rule reports violation for the first three **`for`** loops as the conditional expression for all of them is constant. These **`for`** loop will either never execute or never terminate.

However, the W352 rule does not flag a violation for the last **`for`** loop as the condition expression does not evaluate to a constant.

Default Severity Label

Warning

Rule Group

Loop

Reports and Related Files

None

W478

This rule has been deprecated

The W478 rule has been deprecated. The functionality of this rule is covered by the SYNTH_5233 Built-In rule.

W479

Checks if loop step statement variables are not properly incremented or decremented

Language

Verilog

Rule Description

The W479 rule reports a violation message if the step variable is not incremented or decremented in the step part of the for statement.

For example, in the following case the rule reports violation:

```
for ( i = 0 ; i < 10 ; i = 1 )
```

Parameter(s)

- *strict*: By default, the W479 rule does not report a violation if the step variable is not present in the initialization or condition part. Set the *strict* parameter to **yes** to report violation in such a case.

  For example, in the following case the rule reports violation when the **strict** parameter is set:

  for (i = 0 ; i < 10 ; k = 1)

- *flag_complex_nodes*: By default, this parameter is set to **no**. Set this parameter to **yes** to report a violation when step is written as a complex node.

NOTE: *In the above case, the W479 rule reports a violation only when both the flag_complex_nodes and strict parameters are set to yes.*

Message Details

The following message appears at the location of a **for** construct where the variable *<var-name>* used in the step expression is not incremented or decremented:

```
step variable '<var-name>' not assigned properly
```

Rule Severity

Warning

Suggested Fix

Correct the **for** loop.

Examples

The following example violates the W479 rule as the step expression (**i=2**) does not increment or decrement the value of step variable:

```
for ( i = 0 ; i < 10 ; i = 2 )
  ...
```

W480

Ensure that the loop index is of integer type

When to Use

Use this rule identify the for constructs in which the loop index is not of integer type.

Description

The W480 rule reports violation for those **`for`** constructs in which the loop index variable evaluates to a non-integer.

Language

Verilog

Default Weight

5

Parameter(s)

None

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location of a **`for`** construct where the loop index variable <*var-name*> evaluates to a non-integer value:

[wARNING] Loop index '<var-name>' is not of type integer

Potential Issues

Violation may arise for a **`for`** construct where the loop index variable evaluates to a non-integer value.

Consequences of Not Fixing

You can also use **`reg`** as a loop index. However, using **`reg`** may cause logic to be inferred for that value, which is wasted if the value is required only as a loop index. This may or may not be an issue, depending on whether your

synthesis tools can trim the unused logic.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the line, where non integer signal is used as for loop index. Review the loop to see if really an integer value is required.

To fix the violation, prefer using an integer index wherever possible.

Example Code and/or Schematic

Example 1

Consider the following example:

```
time i1;
realtime i2;
integer i3;
for(i1=0 ; i1<5 ; i1 = i1+1 )
for(i2=0 ; i2<5 ; i2 = i2+1 )
for(i3=0 ; i3<5 ; i3 = i3+1 )
```

In the above example, the W480 rule flags those **`for`** constructs in which the loop index is of type **`time`** and **`realtime`**.

Example 2

Consider the following example:

```
reg [2:0] q;
...
for (q=0 ; q<10; q=q+1)
```

In the above example, the W480 rule reports a violation as the loop index variable, **`q`**, is not an integer.

Default Severity Label

Warning

Rule Group

Loop

Reports and Related Files

No related reports or files.

W481a

Ensure that a for loop uses the same step variable as used in the condition

When to Use

Use this rule to identify the **for** constructs where the variable used in the step expression is not used in the condition expression.

Description

The W481a rule reports violation for the **for** constructs where the variable used in the step expression is not used in the condition expression.

For example, it is meaningless to define a **for** construct as follows:

```
for (i = 0; j < 4; i = i + 1)
```

Here, the variable **i** used in the step expression is not used in the condition expression (**j < 4**).

Default Weight

5

Language

Verilog

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Parameter(s)

strict: The default value is **no**. In this case, the rule reports violation only if none of the step variable is present in the condition statement. Set the value of the parameter to **yes** to report violation for all the step variables that are not present in the condition statement.

Constraint(s)

None

Messages and Suggested Fix

The following message appears at the location of a **for** construct where the variable <*var-name*> used in the step expression is not used in the condition expression:

```
[WARNING] Possibly unsynthesizable loop: step variable '<var-name>' is not used in condition
```

Potential Issues

Violation may arise when a variable used in the step expression in a **for** construct is not used in the condition expression

Consequences of Not Fixing

Synthesis tools may not unroll loops where different variables are used in step and condition expressions as the upper bound is not in general determinable. Also this is arguably poor coding even in logic not intended for simulation.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location of the "for loop" in which step variable is not same as that of the variable used in the condition.

To fix the violation, correct the **for** loop.

Example Code and/or Schematic

Example 1

Consider the following example:

```
for (a=0,b=0; a<10 && b<10 ; a++,b++)
```

In the above example, the W481a rule does not report any violation, even if the value of the *strict* parameter is set to yes. This is because the variables, a and b, are used in step as well as condition expressions.

Example 2

Consider the following example:

```
for (a=0,b=0; a<10; a++,b++)
```

In the above example, by default, the W481a rule does not report any violation.

However, if you set the value of the *strict* parameter to yes, the W481a rule reports violation for variable b, as it is used in the expression variable but is not used in the condition expression.

Example 3

Consider the following example:

```
for (a=0,b=0,c=0,d=0; a<10; b++,c++,d++)
```

In the above example, the W481a rule reports a violation for variables b, c, and d, by default as well as when the *strict* parameter is set to yes. This is because the variables, b, c, and d, are used in the step expression but not in the condition expression.

Default Severity Label

Warning

Rule Group

Loop

Reports and Related Files

No related reports or files

W481b

Ensure that a for loop uses the same initialization variable as used in the condition

When to Use

Use this rule to identify the **for** constructs where the variable used in the initialization expression is not used in the condition expression

Description

The W481b rule reports violation for the **for** constructs where the variable used in the initialization expression is not same as the variable used in the step expression.

For example, it is meaningless to describe a **for** construct as follows:

```
for (i = 0; k < 10; k = k+1)
```

Here, the variable **i** used in the initialization expression is not same as the variable **k** used in the step expression.

In SystemVerilog, there can be multiple initialization/step expressions inside a single **for** construct. In such types of **for** constructs, violation will be reported if a variable used in any of the step expression is not present in any of the initialization expression.

Language

Verilog

Default Weight

5

Messages and Suggested Fix

The following message appears at the location of a **for** construct where the variable *<var1-name>* used in the initialization expression is not same as the variable *<var2-name>* used in the step expression:

```
[WARNING] Unsynthesizable loop: Init variable '<var1-name>' is
not same as step variable '<var2-name>'
```

Potential Issues

Violation may arise when a variable used in the initialization expression in a **for** construct is not same as the variable used in the step expression

Consequences of Not Fixing

Synthesis tools cannot unroll loops where different variables are used in initialization and condition expressions as the upper bound is not in general determinable. Also this is arguably poor coding even in logic not intended for simulation.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the location of the for loop in which initialization expression is not same as that of the variable used in the step expression.

To fix the violation, correct the **for** loop.

Example Code and/or Schematic

Example 1

Consider the following example:

```
for (i = 0,j=0; j < 4; k++,l++)
```

In the above example, the W481b rule reports a violation for k and l as both the variables are used in the step expression but are not used in the initialization expression.

Example 2

Consider the following example:

```
for (i = 0,j=0; j < 4; k++,i++)
```

In the above examples, the W481b rule reports a violation for k as it is used in the step expression but is not used in the initialization expression.

Example 3

Consider the following example:

Consider the following example where the W481a rule does not report any violation:

```
for (i = 0,k=0; j < 4; k++,i++)
```

In the above example, the W481b rule does not report any as the variables, i and k, are used in both initialization and step expressions.

Default Severity Label

Warning

Rule Group

Loop

Reports and Related Files

No related reports or files.

Lint_Elab_Rules

The SpyGlass lint product provides the following function and subprogram rules:

Rule	Flags...
W17	Arrays in sensitivity lists that are not completely specified
W69	case constructs that do not have all possible clauses described and also do not have the default clause
W71	case constructs that do not contain a default clause
W86	Arrays where all elements are not set
W88	Memories where all their elements are not set in the design
W107	Bus connections to primitive gates
W110	Width mismatch between a module port and the net connected to the port in a module instance
W110a	Use same port index bounds in component instantiation and entity declaration
W111	Arrays where all elements are not read in the process
W116	(Verilog) Unequal length operands in bitwise logical/ arithmetic/ ternary operator (VHDL) Unequal length operands in bitwise logical/ arithmetic/ relational operator
W120	Variables that are declared but not used
W122	Signal that is read in a combinational process/ always block, but is not included in the sensitivity list
W123	Signal/ variable that has been read out but is never set
W156	Reverse connected buses
W162	Constant integer assignments to signals when the width of the signal is wider than the width of the constant integer
W163	Cases where a constant integer value is assigned to a vector of smaller size
W164a	Assignments in which the LHS width is less than the (implied) width of the RHS expression
W164b	Assignments in which the LHS width is greater than the (implied) width of the RHS expression

Rule	Flags...
W164c	Assignments in which the LHS width is greater than the (implied) width of the RHS expression
W240	Input ports that are never read
W241	Output ports that are not completely set
W259	Signals that have multiple drivers but no associated resolution function
W263	Case clause labels whose widths do not match the width of the corresponding case construct selector
W287a	Module instances where nets connected to input ports are not driven
W316	Integer conversions where the left expression is wider than the right expression
W328	Constant conversions where the left expression is narrower than the right expression but the extra bits in the right expressions are all zeros
W362	Unequal widths in arithmetic comparison operations
W423	Ports that are re-declared with a different range in the same module
W446	Output ports that are read in the module where they are set
W453	Case constructs with large selector bit-width and more number of case clauses
W456	Signals that are in the sensitivity list of a combinational always construct but are not completely read in the construct
W456a	Signals that are in the sensitivity list of a combinational process block but are not read in the process block
W468	Variables used as array index that are narrower than the array width
W484	Possible loss of carry or borrow bits during assignments using addition and subtraction arithmetic operators
W486	Shift operation overflows
W488	(Verilog) Bus signals that are in the sensitivity list of an always construct but are not completely read in the construct (VHDL) Arrays that appear in the sensitivity list but all elements of the arrays are not read in the process
W494	Inout ports that are never used
W495	Inout ports that are read but never set
W497	Bus signals that are not completely set in the design
W498	Bus signals that are not completely read in the design
W502	Variable that is present in the sensitivity list and is modified in the always block

Rule	Flags...
W504	Port expression that uses integers
W528	Signals or variables that are set but never read
W552	Flip-flop outputs whose different bit-selects are driven in different sequential always constructs
W553	Flip-flop outputs whose different bit-selects are driven in different sequential always constructs

W162

Extension of bits in constant integer conversion

Language

Verilog

Rule Description

The W162 rule flags constant integer assignments to signals when the width of the signal is wider than the width of the constant integer.

When assigning a constant integer value to a LHS operand, the width specification for RHS should match LHS operand width. If the signal is wider than the constant integer, the extra bits are padded with zeros. If the signal is narrower than the constant integer, the extra high-order non-zero bits are discarded.

The W162 rule also checks for those non-constant assignments where extension is due to the constant present in the expression.

```
reg [4:0] a;
reg [1:0] b;
a = 10 + b;
```

NOTE: *The W162 rule supports generate-if, generate-for, and generate-case blocks.*

NOTE: *The W162 rule is switched off by default. You can enable this rule by specifying the* **`set_goal_option addrules W162`** *command.*

Message Details

The following message appears at the location where a constant integer is assigned to a signal that is wider than the constant integer:

`Extension of bits in constant integer conversion is erroneous [Hierarchy: '<hier-path>']`

Where, *<hier-path>* is the complete hierarchical path of the signal.

Rule Severity

Warning

Suggested Fix

No fix is required in general, as long as your are OK with zero-extension.

W163

Truncation of bits in constant integer conversion

Language

Verilog

Rule Description

The W163 rule flags constant integer assignments to signals when the width of the signal is narrower than the width of the constant integer.

When assigning a constant integer value to a LHS operand, the width specification for RHS should match LHS operand width. If the signal is wider than the constant integer, the extra bits are padded with zeros. If the signal is narrower than the constant integer, the extra high-order non-zero bits are discarded.

NOTE: *The W163 rule is switched off by default. You can enable this rule by specifying the* **`set_goal_option addrules W163`** *command.*

Message Details

The following message appears at the location where a constant integer (width *<widthr>*) is assigned to a signal (width *<widthl>*) that is narrower than the constant integer:

`Significant bits of constant will be truncated (lhs width <widthl>, rhs width <widthr>)[Hierarchy: '<hier-path>']`

Where, *<hier-path>* is the complete hierarchical path of the signal.

Rule Severity

Warning

Suggested Fix

Adjust the constant so you assign a smaller constant which will not be truncated.

W164a

Identifies assignments in which the LHS width is less than the RHS width

When to Use

Use this rule to identify assignments in which the LHS width is less than the RHS width.

Description

Verilog

The rule reports violations for assignments in which the width of LHS is less than the (implied) width of the RHS expression.

If there is a signal that used in control path and datapath, the should consider such a signals as datapath, not control path. That is, if a signal is not a control signal, then it is considered as a datapath.

The W164a rule considers all signals as a datapath signal except if the signal is used in a non-control construct of a RHS part of an assignment in the module. Here, a non-control construct refers to a construct, such as, exclude condition of a conditional operation and operands of comparison operator. The W164a rule considers all other signals as a datapath signal.

NOTE: *To enable detailed hierarchy, use the enable_detailed_hierarchy option on SpyGlass For more information, see SpyGlass Explorer User Guide.*

The W164a rule checks if the width of the **`$signed`** negation matches the LHS width on increasing the RHS width by 1. The rule considers both the scenarios and reports violations for such width mismatch scenarios.

The rule determines the assignment type of a binary expression as per the following table:

Condition	Assignment Type of a Binary Expression
LHS < RHS	Expression type of RHS
LHS > RHS	Expression type of LHS
LHS = RHS	The expression type is same as that of the LHS or the RHS type, which can be one of the following: reg, wire, real, realtime, time, integer, static, constant integer, Boolean

Width Calculation for Verilog

By default, the width is calculated considering the best fit width of an expression. That is the width in which maximum value of an expression can be accommodated. When you set the value of the *nocheckoverflow* parameter to **yes** then the width is calculated according to the LRM and natural width is considered for constants.

The behavior of the rule is explained by the following cases.

- For Constant Expressions (with the value of the *nocheckoverflow* parameter set to **no**):
  - For Constant integer expressions, the width is calculated based on the value of the expression:

```
out[3:0] = 2 + 15 ;    //RHS Width = 5 (Value = 17)
out[6:0] = 100 << 4 ; //RHS Width = 11
out[4:0] = 100 >> 2 ; //RHS Width = 5
```

  - If constant is not a part of the sub-expression then the specified width is considered:

```
out[11:0] = 12'b0 ;   //RHS Width = 12
```

  - If the width of a based number is not specified then the natural width is considered:

```
out[8:0] = 'h344;      //RHS Width = 10
```

    For above cases, behavior of the rule remains the same when *nocheckoverflow* parameter is set to **yes**.

- For Arithmetic Operators
  - The width addition operator (with the value of the *nocheckoverflow* parameter set to **no**), is calculated as a width where maximum value of an expression is accommodated. Consider the following example:

```
out[1:0] = in1[1:0] + in2[1:0] + in3[1:0]; //RHS
Width 4
Max value RHS 3 + 3 + 3 = 9
RHS Width = 4


out[2:0] = in1[1:0] + (3/3) ;   //RHS Width 3
Max Value RHS = 3 + 1 = 4
RHS Width = 3
```

```
out[5:0] = in1[5:0] + 5'b1010;  //RHS Width 7
out[4:0] = 15 + 3'b111;      //RHS width 5
```

- For the subtraction operator (with the value of the *nocheckoverflow* parameter set to **no**), the width of the expression is calculated based on the minimum and maximum values of the expression if it is unsigned:

  ```
  out[3:0] = in1[3:0] - in1[1:0]; //Min value is -3 (width is 3), Max value is 12 (width is 4), Therefore, RHS Width 4
  out[3:0] = in1[3:0] - 4'b1010; //Min value is -10 (width 5), Max value is 5 (width is 3), Therefore, RHS Width 5
  ```

  If the expression is of type 1-b, then the width is considered as 1+b.

- For the subtraction operator (with the value of the *nocheckoverflow* parameter set to **no**), the width of the expression is calculated in a similar way as it is done for addition, if the expression is signed.

- For the multiplication operator (with the value of the *nocheckoverflow* parameter set to **no**), the width is calculated as follows:

  When both operands are variable, then the RHS width is the sum of the width of both operands:

  ```
  out[6:0] = in1[1:0] * in2[1:0] * in3[1:0]; //RHS Width 6
  ```

  ```
  out[2:0] = in1[2:0] * in1[1:0];          //RHS Width 5
  ```

  When one operand is static and other is variable, then the RHS width is calculated considering maximum value of the expression:

  ```
  out[3:0] = in1[3:0] * 4'b1;   //RHS Width 4, Max value = 15*1 = 15
  out[3:0] = in1[3:0] * 4'b10;    //RHS Width 5, Max value = 15*2 = 30
  out[7:0] = in1[3:0] * 4'b1010; //RHS Width 8, Max value = 15*10 = 150
  out[3:0] = in1[3:0] * 4;        //RHS Width 6, Max value = 15*4 = 60
  ```

- For the division operator (with the value of the *nocheckoverflow* parameter set to **no**), the width of the RHS is assumed as the width of left operand:

  ```
  out[2:0] = in1[2:0]/in1[1:0]; //RHS Width 3
  out[1:0] = in1[1:0]/in1[2:0]; //RHS Width 2
  ```

```
out[3:0] = in1[3:0]/4'b10;    //RHS Width 4
out[2:0] = 4/in1[3:0];       //RHS Width 3
```

- When the *nocheckoverflow* parameter is set to **yes**, the width is assumed to be the same as the width of term having maximum width. Consider the following example:

```
out[1:0] = in1[1:0] + in2[1:0] + in3[1:0]; //RHS Width = 2
out[5:0] = in1[5:0] + 5'b1010;  //RHS Width 6
out[2:0] = 15 + 3'b111;          //RHS width 4
out[3:0] = in1[3:0] - in1[1:0]; //RHS Width 4
out[2:0] = in1[3:0] - 4'b1010;  //RHS Width 4
out[2:0] = in1[2:0] * in1[1:0]; //RHS Width 3
out[3:0] = in1[3:0] * 4'b10;   //RHS Width 4
out[1:0] = in1[1:0]/in1[2:0];    //RHS Width 3
out[2:0] = 4/in1[3:0];           //RHS Width 4
```

- For Shift Operators
  - For the right shift, if shifted or left operand width is matching LHS width then no violation is reported. Consider the following example:

```
out[3:0] = in1[3:0] >> in ;      //RHS Width 4
out[2:0] = in1[4:0] >> 2 ; //RHS Width 5
out[31:0] = 4 >> in ;   //RHS Width 32
out[4:0] = in1[4:0] >> in2[1:0] ; //RHS Width 5
out[3:0] = in1[4:0] >> in2[1:0] ; //RHS Width 5
```

  - For the left shift (with the value of the *nocheckoverflow* parameter set to **no**), if shifted or left operand width is matching the LHS width, no violation is reported. Consider the following example:

```
out[2:0] = 4'b0001 << in ;         //RHS Width 4
out[2:0] = 4 << in ;               //RHS Width 32
out[4:0] = in1[3:0] << 1 ;      //RHS Width 5
out[6:0] = in1[4:0] << 2 ;      //RHS Width 7
out[3:0] = in1[4:0] << in2[2:0] ; //RHS Width 5
```

  - When the *nocheckoverflow* parameter is set to **yes**, the width of the left operand is considered for both left shift and right shift operations.

- For Self-Determined Expressions
  - The width of a self-determined expression (with the value of the *nocheckoverflow* parameter set to **no**), is calculated according to the LRM. Consider the following example:

```
wire a,b,c;
assign a = {b+c};     //LHS: 1, RHS: 1
assign b = {1'b1,b+c};  //LHS: 1, RHS: 2
```

  - The behavior of the rule remains the same when *nocheckoverflow* parameter is set to **yes.**
- For Conditional Operators

  The width of a conditional operator (with the value of the *nocheckoverflow* parameter set to **no**), is calculated as follows:
  - A violation is reported if there is width mismatch either in the left expression or in the right expression. Consider the following example:

```
out[0] = in1[2] ? in2[0] : in3[2:0] ;   //RHS Width 3
out[1:0] = in1[2] ? in2[2:0] : in3[1:0];//RHS Width 3
out[0] = in1[2] ? in2[0] : in3[2] ;   //RHS Width 1
```

  - The behavior of the rule remains same when *nocheckoverflow* parameter is set to **yes.**
- For Power Operators

  The width of a power operator (with the value of the *nocheckoverflow* parameter set to **no**), is calculated as follows:
  - If the RHS expression of the power operator is static, then the width of the expression is calculated as per the following formula:

    `Expression width = LHS expression width * RHS expression value`

    Consider the following example:

```
out[1:0] = in1[1:0] ** 12;     //RHS Width 24
```

  - If the RHS expression of a power operator is non-static then the LHS expression width is reported:

```
out[1:0] = in1[1:0] ** in2[3:0];  //RHS Width 2
```

- When the *nocheckoverflow* parameter is set to **yes** then the LHS expression width is reported:

  ```
  out[1:0] = in1[1:0] ** 12;        //RHS Width 2
  out[1:0] = in1[1:0] ** in2[3:0];  //RHS width 2
  ```

- For concatenation operator (with the value of the *nocheckoverflow* parameter set to **no**), when the RHS expression is concatenated with zero bits:
  - No violation is reported when the width of the LHS expression lies between the original width of the RHS expression and the width after adding zero concatenated bits. Here, the original width is the width without considering the zero concatenation.
  - When the *nocheckoverflow* parameter is set to yes, a violation is reported when the LHS width is less than the RHS width after adding zero concatenated bits. When the *handle_zero_padding* parameter is set to yes, the rule performs zero expansion or truncation if the RHS of the assignment is a concatenation and the first element is padded with zero.

NOTE: *The rule does not report violation if the RHS is numeric constant zero or any expression multiplied by a numeric constant zero or a based number zero. For example, violation will not be reported if the RHS is either of 0, 4 * 0, 4 * 1'b0, w1 * 4'b0, or (w1+w2) * 0.*

While comparing width of assign statements inside the **for** loops, width of the **for** loop variable is limited to the minimum width, which is enough to occupy values of the **for** loops range (from initial value to condition value). This limiting happens when only the following conditions are true, otherwise the actual loop variable width is considered for comparisons:

- Loop initial value and condition value should be static expressions
- Loop variable condition should be a simple one with one comparison operator
  - **i >= <static expr>**
  - **<static_expr> < j**

  This does not work for the following examples:
  - **(i < <static expr>) && (j < <static expr>)**

- **(i < <static expr>) || (j < <static expr>)**

- Same loop variable/variable expression with same bit selection as used in for-loop condition should be present in RHS of assignment statement

  Example 1

  ```
  for (integer j=31; j>=0; j=j-1)
  wr_domain_sel[4:0] = j;  // No violation because width (width sufficient to occupy loop range) of loop variable j is 5
  ```

  Example 2

  ```
  reg j [31:0] j;
  for (j[15:0]=31; j[15:0]>=0; j[15:0]=j[15:0]-1)
  wr_domain_sel[4:0] = j[15:0]; // No Violation because width (width sufficient to occupy loop range) of loop variable j[15:0] is 5
  ```

  Will not work for following examples

  ```
  reg [31:0]
    for (j[7:0]=31; j[7:0]>=0; j[7:0]=j[7:0]-1)
    wr_domain_sel[4:0] = j[15:0]; // Violation because j[15:0] is not a loop variable and so its width (actual width) is 16
  ```

Width-Mismatch Examples:

use_lrm_width: When the **use_lrm_width** parameter is set to **yes**:

Applicable for *W164a* and *W164a_a* and *W164a_b* rules:

assign a[3:0] = 16; // with use_lrm_width parameter is set to yes, width of 16 will be considered as 32 and the violation width would be 32.

For *W164b* rule:

assign a[7:0] = 16; // with use_lrm_width parameter is set to yes, width of 16 will be considered as 32 and no violation given to rule W164b.

disable_rtl_deadcode: When the **disable_rtl_deadcode** parameter is set to **yes**:

assign a[3:0] = 0 ? b[4:0] : c[3:0] ; // with this parameter is on no

violation will be given for rule W164a and W164a_a

assign a[3:0] = 0 ? b[2:0] : c[3:0] ; // with this parameter is on no violation will be given for rule W164b

assign a[3:0] = 0 ? (b[3:0] + c[3:0]) : c[3:0]; //with this parameter is on no violation will be given for rule W164a_b

check_counter_assignment: When the **check_counter_assignment** parameter is set to **yes**:

```
assign a = a + 1; // by default no violation, and violation will be given when this is set to yes, applicable to W164a and W164a_b
```

```
assign a = a + 5; // by default violation and will not be given when this is set to turbo, applicable for W164a, W164a_a and W164a_b
```

handle_zero_padding: When the **handle_zero_padding** parameter is set to **yes:**

For W164 rules' group:

assign a[3:0] = 12'h00A; // no violation when this parameter is on as leading zeroes can be truncated to match the LHS width

assign a[3:0] = {1'b0, b[3:0]}; // no violation when this parameter is on as leading zeroes can be truncated to match the LHS width

assign a[5:0] = {1'b0, b[3:0]}; // no violation when this parameter is on as leading zeroes can be padded to match the LHS width

assign a[5:0] = 12'h00A << in; // no violation when this parameter is on as leading zeroes can be padded to match the LHS width

```
assign a[15:0] = 12'h00A ; // there will be a violation for this assignment statement as rule W164b will be considering the given width of a size-based number
```

process_complete_condop: When the **process_complete_condop** parameter is set to **yes:**

```
assign a[3:0] = cond ? b[4:0] : c[4:0] ; // two violations will be given for rule W164a when this parameter is on
```

check_concat_max_width: When the **check_concat_max_width** parameter is set to **yes:**

```
assign a[3:0] = {1'b0, b[3:0]} ; //violation with this parameter is set to yes for rule W164a and W164a_a
```

report_only_from_one_hierarchy: When the **report_only_from_one_hierarchy** parameter is set to **yes:**

```
module test();
sub #(1) ();
sub #(2) ();
endmodule
module sub();
parameter P =1;
wire [2:0] b;
wire a = b; //there would be 2 violations given for this assign statement where violation width is same with different hierarchies
// with this parameter is on, only one violation would be given
endmodule
```

VHDL

The *W164a* rule reports violations for assignments in which the width of LHS is less than the (implied) width of the RHS expression.

The *W164a* rule does not report violations for assignments in which the width of the LHS lies between the zero padded and non-zero padded width, provided the RHS of the assignment is an arithmetic expression on the two zero padded concatenation expressions.

For VHDL designs, the rule does not check the function or procedure bodies as the size of the arguments depends on the actual values passed.

In case of all integer types where ranges are specified, the rule works in a different manner. Consider the following example:

```
signal A1 : integer range -100 to 100 ;
signal A2 : integer range -100 to 100 ;
signal A3 : integer range -199 to 199 ;
```

```
signal A4 : integer range -200 to 200 ;
A3 <= A1 + A2 ; -- violation

A4 <= A1 + A2 ; -- No violation
```

In the first assignment, the range of the RHS expression **A1+A2** is from **-200** to **200** but the range of the LHS expression **A3** is from **-199** to **199**, which is less than the range of the RHS expression. Hence, the rule reports a violation.

In the second assignment, the range of the RHS expression **A1+A2** is from **-200** to **200**, which is equal to the range of the LHS expression **A4**. Hence, the rule does not report a violation.

NOTE: *If any operand in the RHS has a non-integer type of range, then the bit-width is compared as per the package, not the range.*

Assignment Types

The rule handles the different assignment types as follows:

- *<Integer_range>* **<=** *<Integer_range>* OP *<Integer_range>*

  The RHS range is evaluated and then compared with the LHS range.

- *<Non_integer_range>* **<=** *<Non_integer_range>* OP *<Non_integer_range>*

  The bit-width for the RHS and the LHS is calculated as per the package.

- *<Non_integer_range>* <= *<Integer_range>* OP *<Integer_range>*

  The RHS range is evaluated and converted to bit-width, which is then compared with the bit-width of the LHS.

- *<Integer_range>* **<=** *<Non_integer_range>* OP *<Non_integer_range>*

- *<Non_integer_range>* <= *<Integer_range>* OP *<Non_integer_range>*

- *<Integer_range>* <= *<Integer_range>* OP *<Non_integer_range>*

  The bit-width of the RHS is calculated from the package and then compared with the bit-width of the LHS.

An integer type without the range specified is treated as *<Non_integer_range>* and bit-width is calculated instead of range.

Width Calculation for VHDL

Set the *nocheckoverflow* parameter to **yes** or *W164a* to calculate the width according to the LRM (numeric_std lib) as per the following cases:

- For the addition and subtraction operators, the width is calculated based on the following rules:
  - If both the operands are variables, then the max width is taken.
  - If one operand is a variable and the other is a static, then the width of the variable is taken.
- For the multiplication operator, the width is calculated based on the following rules:
  - If both the operands are variables, then the RHS width is the sum of both the variables.
  - If one operand is a variable and the other is a static, then the RHS width is 2*(Variable width).
- For the division operator, the width is calculated based on the following rules:
  - If both the operands are variables, then the RHS width is the width of the left operand.
  - If one operand is a variable and the other is a static, then the RHS width is the width of the variable.
- For concatenation operator, no violation is reported when the RHS expression is concatenated with zero bits and the width of the LHS expression lies between original width of the RHS expression and width after adding zero concatenated bits. Here, original width is the width without considering the zero concatenation.

NOTE: *For constant arrays, the declared width is considered when the nocheckoverflow parameter is set to yes. Whereas by default, the maximum width among the initialized elements is considered. For more, refer to the VHDL Example 11.*

NOTE: *For new width related changes, refer to New Width Flow Application Note.*

NOTE: *The W164a rule supports generate-if and generate-for blocks.*

Language

Verilog, VHDL

Parameter(s)

- *check_assign_pattern*: Default value is no. Set the value of the parameter to yes to check assign pattern statements where the elements are assigned names or indexes.
- *check_lrm_and_natural_width*: Default value is no. Set the value of the parameter to yes to check for both LRM and natural widths before reporting violations.
- *check_static_value*: Default value is no. Set the value of the parameter to **yes** or **<rule_list>** to report violation for all cases with width mismatch, involving static expressions and non-static expressions having a static part. You can also set the value of the parameter to **only_static** to ignore violations for expressions, which do not have a static part. Other possible values are **only_const** and **only_expr**.
- *check_static_natural_width*: Default value is no. Set the value of the parameter to yes to enable the specified rule to check for natural width of a static expression even when the value of the *nocheckoverflow* parameter is set to yes.
- *disable_rtl_deadcode*: The default value is no. Set the value of the parameter to yes to disable violations for disabled code in loops and conditional (if condition, ternary operator) statements.
- *sign_extend_func_names*: The default value is ''EXTEND'', “resize”. Set the value of the parameter to any comma-separated list of function names to enable the W164a rule to recognize VHDL sign extension functions and calculate width of extend functions as per the **const** extension argument specified in the argument list.
- *strict*: Default value is **no.** For **Verilog**, by default, the rule reports violation if the RHS expression contains **wire** or **reg.** Set this parameter to **yes** to report all assignments. Also, the rule reports integer port signals in the RHS expression, in this case.

  For **VHDL**,

  - Set the value of the parameter to **yes** to report violation for addition and subtraction expression, if any one operand is a constant and its width is lesser.
  - Set the value of the parameter to **no** to report violations for the following scenarios:

- Maximum range of the RHS is greater than the maximum range of the LHS.
- Minimum range of the RHS is lesser than the minimum range of the LHS.
- RHS value exceeds the maximum LHS range or if it is lesser than the minimum LHS range.

- *use_lrm_width*: Default value is **no**. Set this parameter to **yes** to consider the LRM width of integer constants, which is 32 bits.

  On setting the **use_lrm_width** parameter to **yes** or **W164a**, the *W164a* rule considers the RHS width as 32 if the RHS is of an integer type (where the width is not greater than 32).

- *use_carry_bit*: By default, when the **nocheckoverflow** parameter is set **no**, the width of addition and subtraction is considered as the width of the maximum value that the expression can hold. Set the value of this parameter (with default value as W164a) to **no** to calculate the width as the LRM, that is the maximum width of the operand.

- *nocheckoverflow*: Default value is **no**. Set this parameter to **yes** or rule name to check the bit-width as per LRM. Other possible value is the rule name.

- *checkOperatorOverload*: Default value is **yes**. Set this parameter to **no** to evaluate width of the expression without considering overloaded operators. This parameter is applicable for **VHDL** only.

- *check_counter_assignment*: Default value is **no**. Set this parameter to **yes** to report a violation for the counter type of assignments. You can also set the value of this parameter to turbo. This parameter is applicable for both Verilog and VHDL in the *W164a* rule. Note that when check_counter_assignment is set to turbo, counters defined by the same option are not reported by the W164a rule.

NOTE: *For details about the cases that are considered as counter cases in Verilog, see the check_counter_assignment parameter section.*

- *check_unsign_overflow*: Default value is **no**. This indicates the rule suppresses the overflow when sign extension is used for unsigned signals in addition or subtraction operation. If you set this parameter to **yes** or **W164a**, the rule does not suppress the overflow and reports a violation when sign extension is used for unsigned signals in addition or subtraction operation.

- *check_natural_width_of_multiplication*: Default value is **no**. Set the parameter to **yes**, to calculate the width of the multiplication expressions as per the natural width even if the **nocheckoverflow** parameter is set to **yes**. Set the parameter to a comma/ space-separated list of rules to enable (the parameter) for the rules specified in the list.
- *check_concat_max_width*: Default value is **no.** In this case, no violation is reported when the width of the LHS expression is present between the width of the RHS expression without considering zero concatenated bits and the width of the RHS after adding zero concatenated bits.
  If you set this parameter to **yes**, the RHS width is considered as the width after adding zero concatenated bits. That is, the violation is reported if the LHS width does not match the RHS width after adding zero concatenated bits. This parameter is applicable for **Verilog** only.
- *concat_width_nf*: Default value is no. Set the value of the parameter to yes to specify if the W164a rule should use new algorithm to calculate the width of concatenations ignoring the self-determined nature of concatenation items.
- *handle_shift_op*: Default value is **no**. In this case, no violation is reported if the shifted or non-shifted width of a shift expression matches the LHS width of an assignment. But the rule does not calculate the shifted width, if the RHS of the shift expression is non-static. Set this parameter to **shift_left**, **shift_right**, **shift_both**, **no_shift**, **no_shift_forced**, or comma-separated list of rule names, to compare shifted or non shifted widths for left and right shift expressions.
- *datapath_or_control*: Default value is **no**. In this case, the rule reports violations for all type of signals. Set this parameter to **datapath**, **control**, or **all** to specify the type of signals to be checked.
- *force_handle_shift_op*: Default value is **no**. Set this parameter to **yes** to enable the rule to honor *handle_shift_op* parameter value even if the *nocheckoverflow* parameter is set to **yes**.
- *control_sig_detection_nf*: Default value is **no**. If you set this parameter to **yes**, the rule treats a signal to be a control signal only if it is used in a conditional operator or control binary operator within the scope of currently processing statement. The rule does not consider the signal to

be a control signal even if it is used as a control signal in other places of the module (that is, the statements other than the current statement).

- *handle_lrm_param_in_shift*: Default value is **no**. Set this parameter to **yes** to enable the *W164a* rule to honor the *use_lrm_width* parameter in the width calculations of shift expressions.
- *handle_zero_padding*: Default value is **no**. Set the value of the parameter to **yes** or **rule name** to perform leading zero expansion and truncation of RHS and LHS of an assignment. This is performed only if the RHS and LHS of the assignment are static. In addition, if the parameter is set to **yes**, the rule performs zero expansion or truncation if the RHS of the assignment is a concatenation and the first element is padded with zero.
- *process_complete_condop*: Default value is no. Set the value of the parameter to yes to enable the rule checking on both operands of the condop assignment.
- *treat_concat_assign_separately*: Default value is no. Set the value of the parameter to yes to use this parameter to report violation for each bucket assignment in unpacked array separately. For a packed array, a violation is reported for the whole array.
- *check_natural_width_for_static*: Default value is no. By default, the rule reports violations for static LHS depending on the *check_static_value* parameter. Set the value of the parameter to yes to enable the rule to not report violations for such cases.
  Example:
  ```
  abc[199:49] = 200'b0; // no violation as value 0 can be accommodated in 151 bit LHS.
  def[2:0] = 4'1000;    // violation as value 8 can't be accommodated in 3 bit LHS
  ```
- *report_only_from_one_hierarchy*: Default value is no. By default, multiple violations may be reported on same assignments due to different hierarchies. Set the value of the parameter to yes to report only one violation per assignment.
- *consider_sub_as_add*: By default, when the **nocheckoverflow** parameter is set to **no**, the *W164a* rule considers the width of expression like b -1 as the width of b when there is no underflow due to -ve operator. However, this behavior is limited to a simple - ve

operation, where this is a part of arithmetic expression other than +/-. Set the value of the parameter to **yes** to enable the rule to calculate the width of a-b / b-1 as a+b/ b+1 respectively.

- *consider_lrm_for_div*: Default value is **no**. Set the value of the parameter to **yes** to enable the rule to calculate the width of the division to be the maximum width of the operands (LRM width) when *nocheckoverflow* parameter is set to **no.**
- *ignore_nonstatic_leftshift*: Default value is **no**. Set this parameter to **yes** enable the rule to ignore assignments where the RHS is a left shift operator and both the operands are non-static.
- *report_nonblocking_in_error*: Default value is **no**. Set the value of the parameter to **yes** to report a violation (with **Error** severity) for nonblocking assignments.
- *ignore_counter_with_same_width*: Default value is no. Set the value of the parameter to **yes** to not report violations for cases where any constant value added/subtracted from different signal on both LHS and RHS with the same width is treated as a counter.

  Example 1:

  a = b + 1 where width of both a and b are same.

  Example 2:

```
logic [5:0]a;
logic [5:0]a_nxt;
a_nxt = a + 3; // no violation
```

  Example 3:

```
logic [6:0]a;
logic [5:0]a_nxt;
a_nxt = a + 3;   // violation as width of a & a_nxt are different
```

  Example 4:

  If **ignore_counter_with_same_width** is set to **only_by_one,** then the rule will suppress violations for cases where there is increment/ decrement only by 1 (constant's natural width should be 1).

```
wire [2:0] a,b;
```

```
assign a = b + 1; // no violation
assign a = b + 1'b1; // no violation
assign a = b + 1'd1; // no violation
```

Example 5:

```
logic [24:0] a, b, c;
assign a = {2'b01, c[22:0]};
assign b = ~a + 25'h1; // no violation
```

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message is displayed for the <*lexpr*> LHS expression of an assignment of width <*widthl*>, which is less than the <*rexpr*> RHS expression of width <*widthr*>:

[WARNING] `LHS: '<lexpr>' width <widthl> is less than RHS: <rexpr> width <widthr> in assignment [Hierarchy: '<hier-path>']`

Where, <*hier-path*> is the complete hierarchical path of the signal.

If the *datapath_or_control* parameter is set **datapath**, **control**, or **all**, the following message is reported:

[WARNING] `LHS <control | datapath> signal: '<LHS-expression>' width <LHS-width> is less than RHS: '<RHS-expression>' width <RHS-width> in assignment [Hierarchy: '<hier-path>']`

Potential Issues

A violation is reported when the LHS width is less than the RHS width.

Consequences of Not Fixing

For more information on consequences of not fixing the violation, click *Consequences of Not Fixing*.

How to Debug and Fix

For more information on how to debug and fix the violation, click *How to Debug and Fix*.

VHDL

Message 1

The following message is displayed for the <*lexpr*> LHS expression of an assignment of width <*widthl*>, which is less than the <*rexpr*> RHS expression of width <*widthr*>:

[WARNING] `LHS: '<exprl>' (width <widthl>) is less than RHS: '<exprr>' (width <widthr>) in assignment [Hierarchy: ‘<hier-path>’]`

Where, <*hier-path*> is the complete hierarchical path of the signal.

Potential Issues

A violation is reported when the LHS width is less than the RHS width in an assignment.

Consequences of Not Fixing

For more information on consequences of not fixing the violation, click *Consequences of Not Fixing*.

How to Debug and Fix

For more information on how to debug and fix the violation, click *How to Debug and Fix*.

Message 2

The following message is displayed for the <*rrange*> range of the <*exprr*> RHS expression, which is not within the <*lrange*> range of LHS expression <*exprl*>:

[WARNING] `Range of RHS '<exprr>' (<rrange>) is not within the range of LHS '<exprl' (<lrange>)]`

Potential Issues

A violation is reported when the range of the RHS expression is less than the range of the LHS expression.

Consequences of Not Fixing

This may result in overflow and loss of data.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the line where the width mismatch is detected.

To resolve the violation, explicitly truncate as necessary since it is more readable, instead of relying on the default behavior. Examine the logic to ensure that the truncation does not affect the behavior of the design.

In case of counters specified as integers with range, a fix is not required since the user puts the check for avoiding overflow. It is considered a normal practice to use integer ranges for counters.

Example Code and/or Schematic

Verilog

Example 1

Consider the following example code in which the *strict* parameter reports a violation:

```
wire a,b;
assign a = b + 4 ; //Violation reported with strict
```

Example 2

Consider the following example in which no violation is reported:

```
A [12:0] = { 3'b000, b[9:0] } ;
```

In this case, the LHS width is 13. This LHS width lies between original RHS width 10, which is calculated without considering leading zeros and RHS width 13, which is calculated after considering leading zeros. Therefore, no violation is reported.

Example 3

Consider another example, in which no violation is reported:

```
A [11:0] = { 3'b000, b[9:0] } ;
```

In this case, the LHS width is 12. This LHS width lies between original RHS width 10, which is calculated without considering leading zeros and RHS

width 13, which is calculated after considering leading zeros. Therefore, no violation is reported.

Example 4

Consider another example, in which a violation is reported:

```
A [7:0] = { 3'b000,b[9:0] };
```

In this case, the LHS width is 8. This LHS width does not lie between original RHS width 10, which is calculated without considering leading zeros and RHS width 13, which is calculated after considering leading zeros. Therefore, a violation is reported.

Example 5

Consider the following example, in which violations are reported for a concatenation operator:

```
module test3(d,clk,rst,q1, q2, q3, q4, q5);
input d,clk,rst;
output reg [4:0]q1;
output reg [9:0]q2;
output reg [7:0]q3;
output reg [12:0]q4;
output reg [4:0]q5;
wire [4:0]datain;

always @(posedge clk)
begin
   if(rst)
    begin
      q1 = {5'b00000, datain};
    q2 = {5'b00000, datain};
    q3 = {5'b00000, datain};
      q4 = {5'b00000, datain};
      q5 = {5'b00100, datain};

     q2 = {5'b00000, 3'b000, datain};
    q3 = {5'b00000, 3'b000, datain};

      q2 = {5'b00010, 3'b000, datain};
     q2 = {5'b00000, 3'b010, datain};
```

```
    end
end
endmodule
```

The *W164a* rule reports the following violations for the above example:

```
LHS: 'q5' width 5 is less than RHS: '{5'b00100, datain}'
width 10 in assignment [Hierarchy: ':test3']

LHS: 'q2' width 10 is less than RHS: '{5'b00010, 3'b000,
datain}' width 13 in assignment [Hierarchy: ':test3']
```

Example 6

Consider the following example:

```
assign out2 [8:0] = {3'b001,in1,8'b00001111};
```

In the above example, the rule preserves all leading zeros in case of **out2** assignment, which is **3'b001**. Therefore, the width of the assignment is calculated as **12** bits.

Example 7

Consider the following example:

```
a[3:0] <= b[5:2] + 1; //violation because RHS variable 'b'
does not match with LHS variable 'a'
a[3:0] <= a[3:0] + 1; // no violation
```

Example 8

The following example demonstrates width calculation for the *W164a* rule when the *handle_shift_op* parameter is set to **shift_both**:

```
wire [3:0] w1, w2, w3, w4;
assign w1[2:0] = w2 << w3;                 //RHS Width is 19
assign w1 = 3'b110  << w2;                 //RHS Width is 18
assign w1[2:0] = w2 << (w3 + w4);          //RHS Width is 19
assign w1[2:0] = w2 << (w3 + w4 + 5'd0);   //RHS Width is
34
assign w1[2:0] = w2 << (w3 - w4);          //RHS Width is 19
assign w1[2:0] = w2 << {w3[0],w4[0]};      //RHS Width is 7
assign w1[2:0] = w2 << {w3[0],1'b0};       //RHS Width is 6
```

```
assign w1 = 10'h001 << w3[1:0];          //RHS Width is 10
assign w1 = {1'b0, w2[2:0]} << w3[0];    //RHS Width is 4
```

NOTE: *The example applies to force_handle_shift_op parameter.*

Example 9

The following example demonstrates width calculation for the *W164a* rule when the *handle_shift_op* parameter is set to **shift_both**:

```
module m1();
wire [40:0]a, b, c;

assign a[3:0] = b[3:0] << 1;         //RHS width = 5
assign a[3:0] = b[3:0] << c[1];      //RHS width = 5
assign a[3:0] = 11 << c[0];          //RHS width = 5
assign a[31:0] = 11 << c[0];         //RHS width = 5

assign a[3:0] = b[3:0] >> 1;         //RHS width = 3
assign a[2:0] = b[3:0] >> c[1];      //RHS width = 4
assign a[2:0] = 11 >> c[1];          //RHS width = 4
endmodule
```

Example 10

The following command reports the counter type of assignments for the *W164a* rule when the *check_static_value* and *check_counter_assignment* parameters are set to **yes**:

```
assign w1[3:0] = w1[3:0] + 1;
```

Example 11

Consider the following example in which the *check_unsign_overflow* parameter is set to **yes**:

```
wire [5:0] a,b;
wire [6:0] c;
assign c = {a[5],a} + {b[5],b};
```

The *W164a* rule reports a violation for the unsigned expression.

Example 12

Consider the following example in which the *W164a* rule reports violations for all three assignments when the *check_static_value* parameter is set to

yes:

```
wire [4:0] AAA, BBB, CCC;
assign CCC = 6'b100111;
assign CCC = 120;
assign CCC = (AAA + 1) - BBB;
```

Example 13

Consider the following example:

```
module top;
   for (genvar i = 0 ; i < 4; i = i +1)
      begin : ABC
    mid m1 ();
    end
endmodule


interface mid;
    wire a;
    wire [1:0] b ,c ;
    assign a = b + c;
endinterface
```

When the SpyGlass option, **enable_detailed_hierarchy** is not set, the rule reports the following violation message:

```
LHS: 'a' width 1 is less than RHS: '(b + c )' width 3 in
assignment [Hierarchy: ':top']
```

Set the value of the SpyGlass option, **enable_detailed_hierarchy** to yes to report detailed hierarchy in the violation message, as shown below:

```
LHS: 'a' width 1 is less than RHS: '(b + c)' width 3 in
assignment [Hierarchy: ':top:\ABC[0].m1@mid ']
```

VHDL

Example 1

Consider the following example code in which the **strict** parameter reports a violation:

```
a (3 downto 0) <= b ( 3 downto 0) + 4;
a (3 downto 0) <= 4 + b (3 downto 0);
```

Example 2

Consider the following example:

signal SIG1, SIG3	:Unsigned (3 downto 0)
signal SIG2	:Unsigned (2 downto 0)
signal SIG4	:Unsigned (4 downto 0)
SIG3 < =SIG1+SIG2 //no violation (left width 4 right shift 4)	

In the above example, since both the operands are variables, then the max width is taken. Also, the W164a rule does not report a violation in this case as the LHS width is equal to the RHS width, which is 4.

Example 3

Consider the following example:

```
ibrary ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity test is
  port (
    sig1 :          unsigned( 3 downto 0);
    sig4 :          unsigned( 4 downto 0);
    sig3 :         out unsigned( 3 downto 0)
  );
end test;

architecture test_arc of test  is
begin
        sig3 <= sig1 + 17;  --No violation if parameter
nocheckoverflow is set then width(left4,right4)
```

```
        sig3 <= sig4 + 17; --Violation if parameter
nocheckoverflow is set then width(left4,right5)

end test_arc;
```

Example 4

Consider the following example:

```
signal SIG1          :Unsigned (3 downto 0)
signal SIG2          :Unsigned (2 downto 0)
signal SIG3          :Unsigned (1 downto 0)
SIG1 < =SIG2*SIG3 //violation (left width 4
              right width 5)
```

In the above example, the rule reports violation for the expression as there is a mismatch between LHS and RHS since LHS is 4 and RHS is 5.

Example 5

Consider the following example:

```
signal SIG1          :Unsigned (3 downto 0)
signal SIG2          :Unsigned (2 downto 0)
signal SIG3          :Unsigned (1 downto 0)
SIG1 < =SIG2*SIG3 //violation (left width 4
              right width 5)
```

In the above example, since both the operands are variables, the RHS width is the sum of both the variables. Therefore, the LHS width is 4 and RHS width is 5. As a result, the rule reports a violation in this case.

Example 6

Consider the following example:

```
SIG1 <= SIG3 * 17 //no violation (width is 4 on
                 both sides)
SIG1 <= SIG2 * 17 //violation (left 4, right 6)
```

In the above example, one operand is a variable and the other variable is static. Therefore, the RHS width is 2*(Variable width), that is, LHS width is 4 and the RHS width is 6. As a result, the rule reports a violation in this case.

Example 7

Consider the following example:

```
signal SIG1          :Unsigned (3 downto 0)
signal SIG2          :Unsigned (3 downto 0)
signal SIG3          :Unsigned (1 downto 0)
signal SIG4          :Unsigned (4 downto 0)
SIG1 < =SIG2/SIG3 //No violation (left width 4
                   right width 4)
```

In the above example, both the operands are variables. Therefore, the RHS width is the left operand width, that is, the width of both LHS and RHS is 4. As a result, the rule does not report a violation in this case.

Example 8

Consider the following example:

```
SIG1 <= SIG2 / 17 //no violation (width is 4 on
                   both sides)
SIG1 <= SIG4 / 17 //violation (left 4, right 5)
SIG1 <=17 / SIG2 //no violation (left 4, right 4)
```

In the above example, one operand is a variable and the other variable is static. Therefore, the RHS width is variable width. The rule reports a violation for the second expression as LHS width is 4 and the RHS width is 5. Also, the rule does not report a violation for first and third expression as the LHS and RHS width for both the expressions is 4.

Example 9

Consider the following example:

```
a(3 downto 0)  <=  b(5 downto 2)  +  1;
```

In the above example, the W164a rule does not report a violation as it is a case of counter, where addition is with 1.

Example 10

The following command reports the counter type of assignments for the *W164a* rule when the *check_counter_assignment* and *strict* parameters are set to **yes**:

```
w1(3 downto 0) <= w2(3 downto 0) + 1;
```

Example 11

Consider the following example:

```
architecture rtl of rgx_pbe_gammacmp_lut is
   type aa_type is array(3 downto 0) of std_logic_vector(11
   downto 0);
   constant const_gamma_table : aa_type := (
      X"1ff",
      X"ff",
      X"2ff",
      X"00f"
   );

signal t : std_logic_vector( 9 downto 0);
signal a : std_logic_vector( 13 downto 0);
begin
  t <= const_gamma_table(to_integer(unsigned(a)));
end rtl;
```

In the above example, by default, no violation is reported because the width of RHS and LHS is the same (10), as for RHS, maximum width among the initialized elements of the array (that is the width of **x"2ff"**) is considered. When the *nocheckoverflow* parameter is set to **yes**, for RHS, the declared width of an element of array is considered (12), hence a violation is reported.

Default Severity Label

Warning

Rule Group

Lint_Elab_Rules

Reports and Related Files

None

W164a_a

LHS width is less than RHS width of assignment (Hard Mismatch)

When to Use

Use this rule to identify assignments in which the LHS width is less than the RHS width (hard mismatch).

Description

A hard mismatch happens when the width of the RHS of the assignment is greater than the width of the LHS of the assignment. In the tool, the assignment statements that are violated by the *W164a* rule irrespective of the value of the *nocheckoverflow* parameter are categorized as hard mismatches. In other words, in the assignment, if both the LRM width and the natural width of the RHS of the assignment is greater than the width of the LHS, it is considered as a hard mismatch. But for multiplication operators, the violation is categorized as a hard mismatch if the natural width is greater than the LHS width and not the LRM width. For more details, see the following examples.

Example 1:

```
module test();
wire [3:0] a;
wire [7:0]b;
assign a = b; //both natural and LRM width is 8.
endmodule
```

In the above example, the violation for a = b is treated as a hard mismatch as assigning an 8-bit variable to a 4-bit variable results in a data loss of the most significant bits of variable b.

Example 2:

```
module test();
wire [3:0] a;
wire [4:0]b,c;
assign a = b + c; //both natural and LRM width is 5
endmodule
```

In the above example, the violation for a = b+c is treated as a hard mismatch as assigning a 5-bit addition to a 4-bit variable results in a data loss of the most significant bits of the addition of a+b. Violation for rule W164a is reported even if *nocheckoverflow* is set to **yes**.

Example 3:

```
module test();
wire [3:0] a, b, c;
assign a = b + c; //natural width is 5 and LRM width is 4.
endmodule
```

In the above example, the addition of two 4-bit variables needs 5 bits to accommodate the carry bit of the addition. When *nocheckoverflow* is set to **yes**, carry bit is discarded and no violation is reported when it is assigned to a 4-bit variable. This violation is not treated as a hard mismatch. Data loss is for missing the carry bit of the addition and this violation is covered in the *W164a_b* rule.

Example 4:

```
module test();
wire [3:0] a, b, c;
assign a = b * c; //natural width is 8 and LRM width s 4
and violation when check_natural_width_of_multiplication
is set to yes
endmodule
```

In the above example, the multiplication of two 4-bit variables needs 8 bits to accommodate the maximum value of a*b. When *nocheckoverflow* is set to **yes**, according to LRM, the width would be the maximum width of its operands and in this example it is 4. By default this violation is categorized in rule *W164a_b*, but if you set **check_natural_width_of_multiplication**, then we consider this multiplication violation as a hard mismatch because unlike in the addition operator where carry bit is discarded, this potentially discards multiple bits that results in a major data loss.

Example 5:

```
module test();
```

```
wire [3:0] a, b;
wire  [4:0] c ;
assign a = c / b; //natural width is 5 and LRM width is 4.
endmodule
```

Example 6:

```
module test();
wire  [3:0] a, b;
assign a = b << 2; //natural width 6, LRM width is 4.
```

For a shift operator, the parameter value *handle_shift_op* is honored and the violation is considered as hard mismatch if the width calculated according to parameter value *handle_shift_op* is greater than the width of LHS of the assignment. For this rule, a violation is reported if the *handle_shift_op* parameter value is **shift_left** or **shift_both**.

The *W164a_a* rule reports violations for assignments in which the width of the LHS is less than the (implied) width of the RHS expression. This mismatch may result in truncation of some bits in the assignment.

The assignment type of a binary expression is determined in following order:

- If LHS or RHS is reg then expression type is reg
- If LHS or RHS is wire then expression type is wire
- If LHS or RHS is real then expression type is real
- If LHS or RHS is real-time then expression type is real-time
- If LHS or RHS is time then expression type is time
- If LHS or RHS is integer then expression type is integer
- If LHS or RHS is static then expression type is static
- If LHS or RHS is constant integer then expression type is constant integer
- If LHS or RHS is boolean then expression type is boolean

Width Calculation

By default, the width is calculated considering the best fit width of an

expression. That is the width in which maximum value of an expression can be accommodated. When you set the value of the parameter to **yes** then the width is calculated according to the LRM *nocheckoverflow* and natural width is considered for constants.

The behavior of the rule is explained by the following cases.

- For Constant Expressions (with the value of the *nocheckoverflow* parameter set to **no**):
  - For Constant integer expressions, the width is calculated based on the value of the expression:

```
out[3:0] = 2 + 15 ;   //Violation, RHS Width = 5 (Value
= 17))
out[6:0] = 100 << 4 ; //Violation, RHS Width = 11
out[4:0] = 100 >> 2 ; //RHS Width = 5
```

  - If constant is not a part of the sub-expression then the specified width is considered:

```
out[11:0] = 12'b0 ;  //RHS Width = 12
```

  - If the width of a based number is not specified then the natural width is considered:

```
out[8:0] = 'h344;      //RHS Width = 10
```

    For above cases, behavior of the rule remains the same when *nocheckoverflow* parameter is set to **yes**.

- For Arithmetic Operators
  - The width addition operator (with the value of the *nocheckoverflow* parameter set to **no**), is calculated as a width where maximum value of an expression is accommodated. Consider the following example:

```
out[2:0] = in1[1:0] + (3/3) ;
Max value RHS 3 + 3 + 3 = 9
RHS Width = 4


out[2:0] = in1[1:0] + (3/3) ;  //RHS Width 3
Max Value RHS = 3 + 1 = 4
RHS Width = 3

out[5:0] = in1[5:0] + 7'b1110010;  //RHS Width 8
out[4:0] = 15 + 3'b111;      //RHS width 5
```

- For the subtraction operator (with the value of the *nocheckoverflow* parameter set to **no**), the width is calculated in a similar way as it is done for addition:

```
out[3:0] = in1[3:0] - in1[1:0]; //RHS Width 5
out[4:0] = 15 + 3'b111  //RHS Width 5
```

- For the multiplication operator (with the value of the *nocheckoverflow* parameter set to **no** and the *check_natural_width_of_multiplication* is **yes**), the width is calculated as follows:

  When both operands are variable, then the RHS width is the sum of the width of both operands:

```
out[6:0] = in1[1:0] * in2[1:0] * in3[1:0]; //RHS Width 6
out[2:0] = in1[2:0] * in1[1:0];           //RHS Width 5
```

  When one operand is static and other is variable, then the RHS width is calculated considering maximum value of the expression:

```
out[3:0] = in1[3:0] * 4'b1;   //RHS Width 4, Max value = 15*1 = 15
out[3:0] = in1[3:0] * 4'b10;    //RHS Width 5, Max
value = 15*2 = 30
out[7:0] = in1[3:0] * 4'b1010; //RHS Width 8, Max value = 15*10 =
150
out[3:0] = in1[3:0] * 4;        //RHS Width 6, Max
value = 15*4 = 60
```

- For the division operator (with the value of the *nocheckoverflow* parameter set to **no**), the width of the RHS is assumed as the width of left operand:

```
out[2:0] = in1[2:0]/in1[1:0]; //RHS Width 3
out[1:0] = in1[1:0]/in1[2:0]; //RHS Width 2
out[3:0] = in1[3:0]/4'b10;    //RHS Width 4
out[2:0] = 4/in1[3:0];       //RHS Width 3
```

- When the *nocheckoverflow* parameter is set to **yes**, the width is assumed to be the same as the width of term having maximum width. Consider the following example:

```
out[1:0] = in1[1:0] + in2[1:0] + in3[1:0]; //RHS Width = 2
out[5:0] = in1[5:0] + 5'b1010;  //RHS Width 6
out[2:0] = 15 + 3'b111;          //RHS width 4
out[3:0] = in1[3:0] - in1[1:0]; //RHS Width 4
```

```
out[2:0] = in1[3:0] - 4'b1010;  //RHS Width 4
out[2:0] = in1[2:0] * in1[1:0]; //RHS Width 3
out[3:0] = in1[3:0] * 4'b10;   //RHS Width 4
out[1:0] = in1[1:0]/in1[2:0];   //RHS Width 3
out[2:0] = 4/in1[3:0];          //RHS Width 4
```

- For Shift Operators:
  - For the right shift (with the value of the *nocheckoverflow* parameter set to **no**), if left operand width is matching the LHS width then no violation is reported. Consider the following example:

```
out[3:0] = in1[3:0] >> in ;     //RHS Width 4
out[2:0] = in1[4:0] >> 2 ;         //RHS Width 5
out[31:0] = 4 >> in ;          //RHS Width 32
out[4:0] = in1[4:0] >> in2[1:0] ; //RHS Width 5
out[3:0] = in1[4:0] >> in2[1:0] ; //RHS Width 5
```

  - For the left shift (with the value of the *nocheckoverflow* parameter set to **no**), if shifted or left operand width is matching the LHS width, no violation is reported. Consider the following example:

```
out[2:0] = 4'b0001 << in ;         //RHS Width 4
out[2:0] = 4 << in ;               //RHS Width 32
out[4:0] = in1[3:0] << 1 ;     //RHS Width 5
out[6:0] = in1[4:0] << 2 ;     //RHS Width 7
out[3:0] = in1[4:0] << in2[2:0] ; //RHS Width 5
```

  - When the *nocheckoverflow* parameter is set to **yes**, the width of the left operand is considered for both left shift and right shift operations.
- For Self-Determined Expressions
  - The width of a self-determined expression (with the value of the *nocheckoverflow* parameter set to **no**), is calculated according to the LRM. Consider the following example:

```
wire a,b,c;
assign a = {b+c};     //LHS: 1, RHS: 1
assign b = {1'b1,b+c};  //LHS: 1, RHS: 2
```

  - The behavior of the rule remains the same when *nocheckoverflow* parameter is set to **yes.**

- For Conditional Operators

  The width of a conditional operator (with the value of the *nocheckoverflow* parameter set to **no**), is calculated as follows:

  - A violation is reported if there is width mismatch either in the left expression or in the right expression. Consider the following example:

    ```
    out[0] = in1[2] ? in2[0] : in3[2:0] ;   //RHS Width 3
    out[1:0] = in1[2] ? in2[2:0] : in3[1:0];//RHS Width 3
    out[0] = in1[2] ? in2[0] : in3[2] ;   //RHS Width 1
    ```

  - The behavior of the rule remains same when *nocheckoverflow* parameter is set to **yes.**

- For Power Operators

  The width of a power operator (with the value of the *nocheckoverflow* parameter set to **no**), is calculated as follows:

  - If the RHS expression of the power operator is static, then the width of the expression is calculated as per the following formula:

    ```
    Expression width = LHS expression width * RHS expression value
    ```

    Consider the following example:

    ```
    out[1:0] = in1[1:0] ** 12;     //RHS Width 24
    ```

  - If the RHS expression of a power operator is non-static then the LHS expression width is reported:

    ```
    out[1:0] = in1[1:0] ** in2[3:0];  //RHS Width 2
    ```

  - When the *nocheckoverflow* parameter is set to **yes** then the LHS expression width is reported:

    ```
    out[1:0] = in1[1:0] ** 12;       //RHS Width 2
    out[1:0] = in1[1:0] ** in2[3:0];   //RHS width 2
    ```

Parameter(s)

- *check_assign_pattern*: Default value is no. Set the value of the parameter to yes to check assign pattern statements where the elements are assigned names or indexes.

- *check_static_value*: Default value is no. Set the value of the parameter to **yes** or **<rule_list>** to report violation for all cases with width mismatch, involving static expressions and non-static expressions having a static part. You can also set the value of the parameter to **only_static** to ignore violations for expressions, which do not have a static part. Other possible values are **only_const** and **only_expr**.
- *disable_rtl_deadcode*: The default value is no. Set the value of the parameter to yes to disable violations for disabled code in loops and conditional (if condition, ternary operator) statements.
- *sign_extend_func_names*: The default value is ''EXTEND''. Set the value of the parameter comma-separated list of function names to enable the W164a rule to recognize VHDL sign extension functions and calculate width of extend functions as per the const extension argument specified in the argument list.
- *checkOperatorOverload*: Default value is yes. Set this parameter to no to evaluate width of the expression without considering overloaded operators. This parameter is applicable for VHDL only.
- *check_counter_assignment*: Default value is no. Set this parameter to yes to report a violation for the counter type of assignments. You can also set the value of this parameter to turbo. This parameter is applicable for both Verilog and VHDL in the W164a rule.

  For details about the cases that are considered as counter cases in Verilog, see the check_counter_assignment parameter section.
- *check_unsign_overflow*: Default value is no. This indicates the rule suppresses the overflow when sign extension is used for unsigned signals in addition or subtraction operation. If you set this parameter to yes or W164a, the rule does not suppress the overflow and reports a violation when sign extension is used for unsigned signals in addition or subtraction operation.
- *handle_shift_op*: Default value is **no**. In this case, no violation is reported if the shifted or non-shifted width of a shift expression matches the LHS width of an assignment. But the rule does not calculate the shifted width, if the RHS of the shift expression is non-static. Set this parameter to **shift_left**, **shift_right**, **shift_both**, **no_shift**, **no_shift_forced**, or comma-separated list of rule names, to compare shifted or non shifted widths for left and right shift expressions.

- *handle_lrm_param_in_shift*: Default value is **no**. Set this parameter to **yes** to enable the *W164a* rule to honor the *use_lrm_width* parameter in the width calculations of shift expressions.
- *check_concat_max_width*: Default value is **no.** In this case, no violation is reported when the width of the LHS expression is present between the width of the RHS expression without considering zero concatenated bits and the width of the RHS after adding zero concatenated bits.
  If you set this parameter to **yes**, the RHS width is considered as the width after adding zero concatenated bits. That is, the violation is reported if the LHS width does not match the RHS width after adding zero concatenated bits. This parameter is applicable for **Verilog** only.
- *datapath_or_control*: Default value is **no**. In this case, and the rule reports violations for all type of signals. Set this parameter to **datapath**, **control**, or **all** to specify the type of signals to be checked.
- *control_sig_detection_nf*: Default value is **no**. If you set this parameter to **yes**, the rule treats a signal to be a control signal only if it is used in a conditional operator or control binary operator within the scope of currently processing statement. The rule does not consider the signal to be a control signal even if it is used as a control signal in other places of the module (that is, the statements other than the current statement).
- *handle_zero_padding*: Default value is no. Set the value of the parameter to yes to perform leading zero expansion and truncation of RHS of an assignment.
- *force_handle_shift_op*: Default value is **no**. Set this parameter to **yes** to enable the rule to honor *handle_shift_op* parameter value even if the *nocheckoverflow* parameter is set to **yes**.
- *concat_width_nf*: Default value is no. Set the value of the parameter to yes to specify if the W164a rule should use new algorithm to calculate the width of concatenations ignoring the self-determined nature of concatenation items.
- *nocheckoverflow*: Default value is no. Set this parameter to yes or rule name to check the bit-width as per LRM. Other possible value is the rule name.
- *treat_concat_assign_separately*: Default value is no. Set the value of the parameter to yes to use this parameter to report violation for each bucket assignment in unpacked array separately. For a packed array, a violation is reported for the whole array.

- *process_complete_condop*: Default value is no. Set the value of the parameter to yes to enable the rule checking on both operands of the condop assignment.
- *check_lrm_and_natural_width*: Default value is no. Set the value of the parameter to yes to calculate both lrm and natural width before reporting violation for the assignment statement. If any of these width matches with RHS, then no violation is reported for the assignment.
- *strict*: Default value is **no** and the rule reports violation if the RHS expression contains **wire** or **reg.**Set this parameter to **yes** to report all assignments. Also, the rule reports integer port signals in the RHS expression, in this case.
- *use_lrm_width*: Default value is **no**. Set this parameter to **yes** to consider the LRM width of integer constants, which is 32 bits.
- *check_natural_width_of_multiplication*: Default value is **no**. Set the parameter to **yes**, to calculate the width of the multiplication expressions as per the natural width even if the **nocheckoverflow** parameter is set to **yes**. Set the parameter to a comma/ space-separated list of rules to enable (the parameter) for the rules specified in the list.
- *report_nonblocking_in_error*: Default value is **no**. Set the value of the parameter to **yes** to report a violation (with **Error** severity) for nonblocking assignments.
- *ignore_counter_with_same_width*: Default value is no. Set the value of the parameter to **yes** to not report violations for cases where any constant value added/subtracted from different signal on both LHS and RHS with the same width is treated as a counter.

  Example 1:

  a = b + 1 where width of both a and b are same.

  Example 2:

```
logic [5:0]a;
logic [5:0]a_nxt;
a_nxt = a + 3; // no violation
```

  Example 3:

```
logic [6:0]a;
```

```
logic [5:0]a_nxt;
a_nxt = a + 3;   // violation as width of a & a_nxt are different
```

Example 4:

If **ignore_counter_with_same_width** is set to **only_by_one,** then the rule will suppress violations for cases where there is increment/decrement only by 1 (constant's natural width should be 1).

```
wire [2:0] a,b;
assign a = b + 1; // no violation
assign a = b + 1'b1; // no violation
assign a = b + 1'd1; // no violation
```

Example 5:

```
logic [24:0] a, b, c;
assign a = {2'b01, c[22:0]};
assign b = ~a + 25'h1; // no violation
```

Constraint(s)

None

Messages and Suggested Fix

The following message is displayed for the <*lexpr*> LHS expression of an assignment of width <*widthl*>, which is less than the <*rexpr*> RHS expression of width <*widthr*>:

[WARNING] `LHS: '<lexpr>' width <widthl> is less than RHS: <rexpr> width <widthr> in assignment [Hierarchy: '<hier-path>']`

Where, <*hier-path*> is the complete hierarchical path of the signal.

If the *datapath_or_control* parameter is set **datapath**, **control**, or **all**, the following message is reported:

[WARNING] `LHS <control | datapath> signal: '<LHS-expression>' width <LHS-width> is less than RHS: '<RHS-expression>' width <RHS-width> in assignment [Hierarchy: '<hier-path>']`

Potential Issues

A violation is reported when the LHS width is definitely less than the RHS width.

Consequences of Not Fixing

This may result in overflow and loss of data.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the line where the width mismatch is detected.

To resolve the violation, explicitly truncate as necessary since it is more readable, instead of relying on the default behavior. Examine the logic to ensure that the truncation does not affect the behavior of the design.

In case of counters specified as integers with range, a fix is not required since the user puts the check for avoiding overflow. It is considered a normal practice to use integer ranges for counters.

Example Code and/or Schematic

Consider the following example:

```
wire [4:0] AAA, BBB, CCC;
assign CCC = 6'b100111;
assign CCC = 120;
assign CCC = (AAA + 100) - BBB;
```

For the above example, by default, no violation is reported. However, when you set the value of the check_static_value parameter to yes, the W164a_a rule reports a violation.

Default Severity Label

Warning

Reports and Related Files

None

W164a_b

LHS width is less than RHS width of assignment (Soft Mismatch)

When to Use

Use this rule to identify assignments in which the LHS width is less than the RHS width (soft mismatch).

Description

A soft mismatch happens when the width of the RHS of the assignment is greater than the width of the LHS of the assignment for some of the values in the RHS of the assignment. In the tool, the assignment statements that are violated by the *W164a* rule when the *nocheckoverflow* parameter is set **no**, but are not violated when the *nocheckoverflow* is set to **yes**, are categorized as soft mismatches. In other words, a violation is categorized as a soft mismatch, if the LHS of the assignment matches with LRM width but not with the natural width of the RHS of the assignment. For more details, see the following examples.

Example 1:

```
module test();

wire [3:0] a, b, c;

assign a = b + c; //natural width is 5 and LRM width is 4.

endmodule
```

In the above example, the addition of two 4-bit variables needs 5 bits to accommodate the carry bit of the addition. When *nocheckoverflow* is set to **yes**, carry bit is discarded and no violation is reported when it is assigned to a 4-bit variable. This violation is not treated as a soft mismatch. Data loss is for missing the carry bit of the addition.

NOTE: *Any other violations that are not reported by the W164a_a rule but reported by the W164a rule are considered as soft mismatches.*

The *W164a_b* rule reports violations for assignments in which the width of LHS is less than the (implied) width of the RHS expression. This mismatch

may result in truncation of some bits in the assignment.

The assignment type of a binary expression is determined in following order:

- If LHS or RHS is reg then expression type is reg
- If LHS or RHS is wire then expression type is wire
- If LHS or RHS is real then expression type is real
- If LHS or RHS is real-time then expression type is real-time
- If LHS or RHS is time then expression type is time
- If LHS or RHS is integer then expression type is integer
- If LHS or RHS is static then expression type is static
- If LHS or RHS is constant integer then expression type is constant integer
- If LHS or RHS is boolean then expression type is boolean

Width Calculation

By default, the width is calculated considering the best fit width of an expression. That is the width in which maximum value of an expression can be accommodated. When you set the value of the *nocheckoverflow* parameter to **yes** then the width is calculated according to the LRM and natural width is considered for constants.

The behavior of the rule is explained by the following cases.

- For Arithmetic Operators
  - The width addition operator (with the value of the *nocheckoverflow* parameter set to **no**), is calculated as a width where maximum value of an expression is accommodated. Consider the following example:

```
out[1:0] = in1[1:0] + in2[1:0] + in3[1:0]; //RHS
Width 4
Max value RHS 3 + 3 + 3 = 9
RHS Width = 4


out[2:0] = in1[1:0] + (3/3) ;   //RHS Width 3
Max Value RHS = 3 + 1 = 4
RHS Width = 3
```

```
out[5:0] = in1[5:0] + 5'b1010;  //RHS Width 7
out[4:0] = 15 + 3'b111;      //RHS width 5
```

- For the subtraction operator (with the value of the *nocheckoverflow* parameter set to **no**), the width is calculated in a similar way as it is done for addition:

```
out[3:0] = in1[3:0] - in1[1:0]; //RHS Width 5
out[4:0] = in1[3:0] - 4'b1010;  //RHS Width 5
```

- For the division operator (with the value of the *nocheckoverflow* parameter set to **no**), the width of the RHS is assumed as the width of left operand:

```
out[2:0] = in1[2:0]/in1[1:0]; //RHS Width 3
out[1:0] = in1[1:0]/in1[2:0]; //RHS Width 2
out[3:0] = in1[3:0]/4'b10;   //RHS Width 4
out[2:0] = 4/in1[3:0];      //RHS Width 3
```

- When the *nocheckoverflow* parameter is set to **yes**, the width is assumed to be the same as the width of term having maximum width. Consider the following example:

```
out[1:0] = in1[1:0] + in2[1:0] + in3[1:0]; //RHS Width = 2
out[5:0] = in1[5:0] + 5'b1010;  //RHS Width 6
out[2:0] = 15 + 3'b111;          //RHS width 4
out[3:0] = in1[3:0] - in1[1:0]; //RHS Width 4
out[2:0] = in1[3:0] - 4'b1010;  //RHS Width 4
out[2:0] = in1[2:0] * in1[1:0]; //RHS Width 3
out[3:0] = in1[3:0] * 4'b10;   //RHS Width 4
out[1:0] = in1[1:0]/in1[2:0];   //RHS Width 3
out[2:0] = 4/in1[3:0];          //RHS Width 4
```

- For Self-Determined Expressions
  - The width of a self-determined expression (with the value of the *nocheckoverflow* parameter set to **no**), is calculated according to the LRM. Consider the following example:

```
wire a,b,c;
assign a = {b+c};     //LHS: 1, RHS: 1
assign b = {1'b1,b+c};  //LHS: 1, RHS: 2
```

- ❒ The behavior of the rule remains the same when *nocheckoverflow* parameter is set to **yes.**

- For Conditional Operators

  The width of a conditional operator (with the value of the *nocheckoverflow* parameter set to **no**), is calculated as follows:

  - ❒ A violation is reported if there is width mismatch either in the left expression or in the right expression. Consider the following example:

    ```
    out[0] = in1[2] ? in2[0] : in3[2:0] ;   //RHS Width 3
    out[1:0] = in1[2] ? in2[2:0] : in3[1:0];//RHS Width 3
    out[0] = in1[2] ? in2[0] : in3[2] ;   //RHS Width 1
    ```

  - ❒ The behavior of the rule remains same when *nocheckoverflow* parameter is set to **yes.**

Parameter(s)

- *check_assign_pattern*: Default value is no. Set the value of the parameter to yes to check assign pattern statements where the elements are assigned names or indexes.
- *strict*: Default value is **no** and the rule reports violation if the RHS expression contains **wire** or **reg.**Set this parameter to **yes** to report all assignments. Also, the rule reports integer port signals in the RHS expression, in this case.
- *check_static_value*: Default value is no. Set the value of the parameter to **yes** or **<rule_list>** to report violation for all cases with width mismatch, involving static expressions and non-static expressions having a static part. You can also set the value of the parameter to **only_static** to ignore violations for expressions, which do not have a static part. Other possible values are **only_const** and **only_expr**.
- *check_counter_assignment*: Default value is **no**. Set this parameter to **yes** to report a violation for the counter type of assignments.
- *use_lrm_width*: Default value is **no**. Set this parameter to **yes** to consider the LRM width of integer constants, which is 32 bits.
- *handle_shift_op*: Default value is **no**. In this case, no violation is reported if the shifted or non-shifted width of a shift expression matches the LHS width of an assignment. But the rule does not calculate the shifted

width, if the RHS of the shift expression is non-static. Set this parameter to **shift_left**, **shift_right**, **shift_both**, **no_shift**, **no_shift_forced**, or comma-separated list of rule names, to compare shifted or non shifted widths for left and right shift expressions.

- *handle_lrm_param_in_shift*: Default value is **no**. Set this parameter to **yes** to enable the *W164a* rule to honor the *use_lrm_width* parameter in the width calculations of shift expressions.
- *nocheckoverflow*: Default value is no. Set this parameter to yes or rule name to check the bit-width as per LRM. Other possible value is the rule name.
- *check_concat_max_width*: Default value is **no.** In this case, no violation is reported when the width of the LHS expression is present between the width of the RHS expression without considering zero concatenated bits and the width of the RHS after adding zero concatenated bits.
  If you set this parameter to **yes**, the RHS width is considered as the width after adding zero concatenated bits. That is, the violation is reported if the LHS width does not match the RHS width after adding zero concatenated bits. This parameter is applicable for **Verilog** only.
- *datapath_or_control*: Default value is **no**. In this case, and the rule reports violations for all type of signals. Set this parameter to **datapath**, **control**, or **all** to specify the type of signals to be checked.
- *control_sig_detection_nf*: Default value is **no**. If you set this parameter to **yes**, the rule treats a signal to be a control signal only if it is used in a conditional operator or control binary operator within the scope of currently processing statement. The rule does not consider the signal to be a control signal even if it is used as a control signal in other places of the module (that is, the statements other than the current statement).
- *handle_zero_padding*: Default value is no. Set the value of the parameter to yes to perform leading zero expansion and truncation of RHS of an assignment.
- *force_handle_shift_op*: Default value is **no**. Set this parameter to **yes** to enable the rule to honor *handle_shift_op* parameter value even if the *nocheckoverflow* parameter is set to **yes**.
- *concat_width_nf*: Default value is no. Set the value of the parameter to yes to specify if the W164a rule should use new algorithm to calculate the width of concatenations ignoring the self-determined nature of concatenation items.

- *treat_concat_assign_separately*: Default value is no. Set the value of the parameter to yes to use this parameter to report violation for each bucket assignment in unpacked array separately. For a packed array, a violation is reported for the whole array.
- *process_complete_condop*: Default value is no. Set the value of the parameter to yes to enable the rule checking on both operands of the condop assignment.
- *check_concat_max_width*: Default value is no. Set the value of the parameter to yes to calculate both lrm and natural width before reporting violation for the assignment statement. If any of these width matches with RHS, then no violation is reported for the assignment.
- *check_natural_width_of_multiplication*: Default value is **no**. Set the parameter to **yes**, to calculate the width of the multiplication expressions as per the natural width even if the **nocheckoverflow** parameter is set to **yes**. Set the parameter to a comma/ space-separated list of rules to enable (the parameter) for the rules specified in the list.
- *report_nonblocking_in_error*: Default value is **no**. Set the value of the parameter to **yes** to report a violation (with **Error** severity) for nonblocking assignments.
- *ignore_counter_with_same_width*: Default value is no. Set the value of the parameter to **yes** to not report violations for cases where any constant value added/subtracted from different signal on both LHS and RHS with the same width is treated as a counter.

  Example 1:

  a = b + 1 where width of both a and b are same.

  Example 2:

```
logic [5:0]a;
logic [5:0]a_nxt;
a_nxt = a + 3; // no violation
```

  Example 3:

```
logic [6:0]a;
logic [5:0]a_nxt;
a_nxt = a + 3;   // violation as width of a & a_nxt
```

```
are different
```

Example 4:

If **ignore_counter_with_same_width** is set to **only_by_one,** then the rule will suppress violations for cases where there is increment/decrement only by 1 (constant's natural width should be 1).

```
wire [2:0] a,b;
assign a = b + 1; // no violation
assign a = b + 1'b1; // no violation
assign a = b + 1'd1; // no violation
```

Example 5:

```
logic [24:0] a, b, c;
assign a = {2'b01, c[22:0]};
assign b = ~a + 25'h1; // no violation
```

Messages and Suggested Fix

The following message is displayed for the <*lexpr*> LHS expression of an assignment of width <*widthl*>, which is less than the <*rexpr*> RHS expression of width <*widthr*>:

[WARNING] `LHS: '<lexpr>' width <widthl> is less than RHS: <rexpr> width <widthr> in assignment [Hierarchy: '<hier-path>']`

Where, <*hier-path*> is the complete hierarchical path of the signal.

If the *datapath_or_control* parameter is set **datapath**, **control**, or **all**, the following message is reported:

[WARNING] `LHS <control | datapath> signal: '<LHS-expression>' width <LHS-width> is less than RHS: '<RHS-expression>' width <RHS-width> in assignment [Hierarchy: '<hier-path>']`

Potential Issues

A violation is reported when the LHS width may be less than the RHS width.

Consequences of Not Fixing

This may result in overflow and loss of data.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the line where the width mismatch is detected.

To resolve the violation, explicitly truncate as necessary since it is more readable, instead of relying on the default behavior. Examine the logic to ensure that the truncation does not affect the behavior of the design.

In case of counters specified as integers with range, a fix is not required since the user puts the check for avoiding overflow. It is considered a normal practice to use integer ranges for counters.

Example Code and/or Schematic

Example 1

Consider the following example:

```
wire a,b;
assign a = b + 1 ; //Violation with strict
```

Example 2

Consider the following example:

```
wire [4:0] AAA, BBB, CCC;

assign CCC = (AAA + 1) - BBB;
```

In the above example, by default, no violation is reported. However, when the value of the check_static_value parameter is set to yes, the rule reports a violation because the RHS expression is a static expression.

Example 3

Consider the following examples for both Verilog and VHDL:

```
assign w1[3:0] = w2[3:0] + 1;   //Verilog

w1(3 downto 0) <= w2(3 downto 0) - 1; --VHDL
```

When the value of the check_counter_assignment parameter is set to yes, the rule reports a violation.

Also, when the value of the check_counter_assignment parameter is set to turbo, the rule reports violation for the following example:

```
assign w1[3:0] = w1[3:0] + 1;  //Verilog
```

```
assign w1[3:0] = w1[3:0] + 2;  //Verilog
```

Default Severity Label

Warning

Reports and Related Files

None

W164b

Identifies assignments in which the LHS width is greater than the RHS width

When to Use

Use this rule to identify assignments in which the LHS width is greater than the RHS width.

Description

Verilog

The W164b rule flags assignments in which the LHS width is greater than the (implied) width of the RHS expression.

The rule determines the assignment type of a binary expression as per the following table:

Condition	Assignment Type of a Binary Expression
LHS < RHS	Expression type of RHS
LHS > RHS	Expression type of LHS
LHS = RHS	The expression type is same as that of the LHS or the RHS type. The assignment type is determined from the following types in the left to right order: reg, wire, real, realtime, time, integer, static, constant integer, Boolean

Width Calculation for Verilog

By default, the width is calculated considering the best fit width of an expression. That is the width in which maximum value of an expression can be accommodated.

When you set the value of the *nocheckoverflow* parameter to **yes** then width is calculated according to the LRM and the natural width is considered for constants. The *nocheckoverflow* parameter handles the width calculations separately for LHS and RHS and it does not have an effect on the comparison of the calculated widths.

The behavior of the rule is explained by the cases below.

- For Constant Expressions
  - For constant integer expressions, the width is calculated based on the value of expression:

    ```
    out[5:0] = 2 + 15 ;     //RHS Width = 5 (Value = 17)
    out[10:0] = 100 << 4 ; //RHS Width = 11
    out[6:0] = 100 >> 2 ;  //RHS Width = 5
    ```

  - If constant is not a part of sub-expression then the specified width is considered:

    ```
    out[11:0] = 12'b0 ;   //RHS Width = 12
    ```

  - If the width of a based number is not specified then the natural width is considered:

    ```
    out[11:0] = 'h344;      //RHS Width = 10
    ```

    For above cases, behavior of rule remains same when the *nocheckoverflow* parameter is set to **yes**.

- For Arithmetic Operators
  - For addition operator, the width is calculated as the width in which maximum value of expression is accommodated. Consider the following example:

    ```
    out[3:0] = in1[1:0] + in2[1:0] + in3[1:0]; //RHS Width 4
    Max value RHS 3 + 3 + 3 = 9
    RHS Width = 4

    out[2:0] = in1[1:0] + (3/3) ;   //RHS Width 3
    Max Value RHS = 3 + 1 = 4
    RHS Width = 3

    out[6:0] = in1[5:0] + 5'b1010;  //RHS Width 7
    out[4:0] = 15 + 3'b111;       //RHS width 5
    ```

  - For the subtraction operator (with the value of the *nocheckoverflow* parameter set to **no**), the width of the expression is calculated based on the minimum and maximum values of the expression if it is unsigned:

    ```
    out[3:0] = in1[3:0] - in1[1:0]; //Min value is -3 (width is 3), Max value is 12 (width is 4), Therefore, RHS Width 4
    ```

```
out[5:0] = in1[3:0] - 4'b1010; //Min value is -10 (width 5), Max
value is 5 (width is 3), Therefore, RHS Width 5
```

If the expression is of type 1-b, then the width is considered as 1+b.

- For the subtraction operator (with the value of the *nocheckoverflow* parameter set to **no**), the width of the expression is calculated in a similar way as it is done for addition, if the expression is signed.
- For the multiplication operator, the width is calculated as follows:

  When both operands are variable, then the RHS width is the sum of the width of both operands:

```
out[5:0] = in1[1:0] * in2[1:0] * in3[1:0]; //RHS Width 6
out[4:0] = in1[2:0] * in1[1:0];        //RHS Width 5
```

  When one operand is static and other is variable, then the RHS width is calculated considering maximum value of the expression:

```
out[7:0] = in1[3:0] * 4'b1;     //RHS Width 4, Max
value = 15*1 = 15
out[7:0] = in1[3:0] * 4'b10;    //RHS Width 5, Max
value = 15*2 = 30
out[7:0] = in1[3:0] * 4'b1010; //RHS Width 8, Max value = 15*10 =
150
out[6:0] = in1[3:0] * 4;        //RHS Width 6, Max
value = 15*4 = 60
```

- For the division operator, the width of the RHS is assumed as the width of left operand:

```
out[2:0] = in1[2:0]/in1[1:0]; //RHS Width 3

out[2:0] = in1[1:0]/in1[2:0]; //RHS Width 2

out[3:0] = in1[3:0]/4'b10;   //RHS Width 4

out[3:0] = 4/in1[3:0];         //RHS Width 3
```

- When the *nocheckoverflow* parameter is set to **yes**, the width is assumed to be the same as the width of term having maximum width. Consider the following example:

```
out[3:0] = in1[1:0] + in2[1:0] + in3[1:0]; //RHS
Width 2

out[6:0] = in1[5:0] + 5'b1010;  //RHS Width 6
```

```
out[4:0] = 15 + 3'b111;          //RHS width 4
out[4:0] = in1[3:0] - in1[1:0]; //RHS Width 4
out[4:0] = in1[3:0] - 4'b1010;  //RHS Width 4
out[4:0] = in1[2:0] * in1[1:0]; //RHS Width 3
out[7:0] = in1[3:0] * 4'b10;    //RHS Width 4
out[2:0] = in1[1:0]/in1[2:0];  //RHS Width 3
out[3:0] = 4/in1[3:0];       //RHS Width 4
```

- For Shift Operator
  - For the right shift, if left operand width is matching the LHS width then no violation is reported. Consider the following example:

```
out[3:0] = in1[3:0] >> in ;     //RHS Width 4
out[5:0] = in1[4:0] >> 2 ;         //RHS Width 5
out[31:0] = 4 >> in ;         //RHS Width 32
out[4:0] = in1[4:0] >> in2[1:0] ; //RHS Width 5
out[5:0] = in1[4:0] >> in2[1:0] ; //RHS Width 5
```

  - For left shift, if shifted or left operand width is matching LHS width then no violation is reported. Consider the following example:

```
out[5:0] = 4'b0001 << in ;         //RHS Width 4
out[31:0] = 4 << in ;         //RHS Width 32
out[5:0] = in1[4:0] << 1 ;     //RHS Width 6
out[6:0] = in1[4:0] << 2 ;     //RHS Width 7
out[4:0] = in1[4:0] << 2 ;     //RHS Width 5
out[7:0] = in1[4:0] << in2[2:0] ; //RHS Width 5
```

  - When parameter *nocheckoverflow* is set to **yes** then width of left operand is considered for both left shift and right shift operations.
- For Self-Determined Expression
  - For self-determined expressions, the width is calculated as per the LRM:

```
wire a,b,c;
assign a = {b+c};          //LHS: 1, RHS: 1
assign out[2:0] = {1'b1,b+c};  //LHS: 3, RHS: 2
```

- Behavior of the rule remains the same when the *nocheckoverflow* parameter is set to **yes**

- For Conditional Operator

  The width of conditional operator is calculated as follows:

  - A violation is reported if there is a width mismatch either in left expression or in right expression. Consider the following example:

    ```
    out[2:0] = in1[2] ? in2[0] : in3[2:0] ;  //RHS Width 1
    out[2:0] = in1[2] ? in2[2:0] : in3[1:0];//RHS Width 2
    out[0]   = in1[2] ? in2[0] : in3[2] ; //RHS Width 1
    ```

  - Behavior remains same when the *nocheckoverflow* parameter is set to **yes.**

- For Power Operator

  The width of power operator is calculated as follows:

  - If the RHS expression of the power operator is static, then the width of the expression is calculated as per the following formula:

    ```
    Expression width = LHS expression width * RHS expression value
    ```

    Consider the following example:

    ```
    out[23:0] = in1[1:0] ** 12;  //RHS Width 24
    ```

  - If the RHS expression of power operator is non static then the LHS expression width is reported. Consider the following example:

    ```
    out[1:0]  = in1[1:0] ** in2[3:0];  //RHS Width 2
    ```

  - When the *nocheckoverflow* parameter is set to **yes**, the LHS expression width is reported. Consider the following example:

    ```
    out[23:0] = in1[1:0] ** 12;          //RHS Width 2
    out[1:0]  = in1[1:0] ** in2[3:0];   //RHS width 2
    ```

- For concatenation operator, when the RHS expression is concatenated with zero bits:

  - No violation is reported when the width of the LHS expression lies between the original width of the RHS expression and the width after adding zero concatenated bits. Here, the original width is the width without considering the zero concatenation.

- When the *nocheckoverflow* parameter is set to yes, a violation is reported when the LHS width is less than the RHS width after adding zero concatenated bits. When the *handle_zero_padding* parameter is set to yes, the rule performs zero expansion or truncation if the RHS of the assignment is a concatenation and the first element is padded with zero.

NOTE: *The rule does not report violation if the RHS is numeric constant zero or any expression multiplied by a numeric constant zero or a based number zero. For example, violation will not be reported if the RHS is either of 0, 4 * 0, 4 * 1'b0, w1 * 4'b0, or (w1+w2) * 0.*

VHDL

The *W164b* rule flags assignments in which the LHS width is greater than the (implied) width of the RHS expression.

The *W164b* rule does not report violations for assignments in which the width of the LHS lies between the zero padded and non-zero padded width, provided the RHS of the assignment is an arithmetic expression on the two zero padded concatenation expressions.

For VHDL designs, the W164b rule does not check function or procedure bodies as the size of the arguments may depend on the actual passed in the function or procedure call.

In case of all integer types with range specified, the W164b rule checking is in different manner. Consider the following example:

```
signal A1 : integer range -100 to 100 ;
signal A2 : integer range -100 to 100 ;
signal A3 : integer range -201 to 201 ;
signal A4 : integer range -200 to 200 ;
A3 <= A1 + A2 ;
A4 <= A1 + A2;
```

In the first assignment, RHS expression **A1+A2** can have values from **-200** to **200** but range of the LHS expression **A3** is **-201** to **201** which is greater than the range of RHS expression. Hence, the W164b rule flags a message.

However, in the second assignment, range of LHS expression **A4** is **-200** to **200**, which is equal to range of RHS expression **A1+A2**. Hence, the W164b rule does not flag this assignment.

NOTE: *If any operand of RHS has non-integer range type, bit-width will be compared as per the package, not the range.*

Assignment Types

The W164b rule handles different assignment types as follows:

- *<int-range>* **<=** *<int-range>* OP *<int-range>*

  Range will be evaluated for RHS and then compared with the range of LHS.

- *<non-int-range>* **<=** *<non-int-range>* OP *<non-int-range>*

  The bit-width for RHS and LHS will be calculated as per the package.

- *<non-int-range>* **<=** *<int-range>* OP *<int-range>*

  The range will be evaluated for RHS, which will then be converted in bit-width and the bit-width will then be compared with bit-width of LHS.

- *<int-range>* **<=** *<non-int-range>* OP *<non-int-range>*

- *<non-int-range>* **<=** *<int-range>* OP *<non-int-range>*

- *<int-range>* **<=** *<int-range>* OP *<non-int-range>*

  The bit-width of RHS will be calculated from package and then compared with the bit-width of LHS.

An integer type without the range specified will be treated as *<non-int-range>* and bit-width will be calculated instead of range.

Width Calculation for VHDL

Set the *nocheckoverflow* parameter to **yes** or *W164b* to calculate the width according to the LRM (numeric_std lib) as per the following cases:

- For the addition and subtraction operators, the width is calculated based on the following rules:
  - ❒ If both the operands are variables, then the max width is taken.
  - ❒ If one operand is a variable and the other is a static, then the width of the variable is taken.
- For the multiplication operator, the width is calculated based on the following rules:
  - ❒ If both the operands are variables, then the RHS width is the sum of both the variables.

- If one operand is a variable and the other is a static, then the RHS width is 2*(Variable width).

- For the division operator, the width is calculated based on the following rules:
  - If both the operands are variables, then the RHS width is the width of the left operand.
  - If one operand is a variable and the other is a static, then the RHS width is the width of the variable.
- For concatenation operator, no violation is reported when the RHS expression is concatenated with zero bits and the width of the LHS expression lies between original width of the RHS expression and width after adding zero concatenated bits. Here, original width is the width without considering the zero concatenation.

NOTE: *For constant arrays, the declared width is considered when the nocheckoverflow parameter is set to yes. Whereas by default, the maximum width among the initialized elements is considered. For more, refer to the VHDL Example 9.*

NOTE: *The W164b rule supports generate-if and generate-for blocks.*

Language

Verilog, VHDL

Default Weight

10

Parameter(s)

- *check_assign_pattern*: Default value is no. Set the value of the parameter to yes to check assign pattern statements where the elements are assigned names or indexes.
- *check_lrm_and_natural_width*: Default value is no. Set the value of the parameter to yes to check for both LRM and natural widths before reporting violations.
- *check_natural_width_of_multiplication*: Default value is **no**. Set the parameter to **yes**, to calculate the width of the multiplication expressions as per the natural width even if the **nocheckoverflow** parameter is set to **yes**. Set the parameter to a comma/ space-separated list of rules to enable (the parameter) for the rules specified in the list.

- *check_static_value*: Default value is no. Set the value of the parameter to **yes** or **<rule_list>** to report violation for all cases with width mismatch, involving static expressions and non-static expressions having a static part. Other possible values are **only_const** and **only_expr**.
- *disable_rtl_deadcode*: The default value is no. Set the value of the parameter to yes to disable violations for disabled code in loops and conditional (if condition, ternary operator) statements.
- *sign_extend_func_names*: The default value is ''EXTEND'', “resize”. Set the value of the parameter to any comma-separated list of function names to enable the W164b rule to recognize VHDL sign extension functions and calculate width of extend functions as per the **const** extension argument specified in the argument list.
- *strict*: Default value is **no**.

  For **Verilog**, by default, the rule reports violation if the RHS expression contains **wire** or **reg.**Set this parameter to **yes** to report all assignments. Also, the rule reports integer port signals in the RHS expression, in this case.

  For **VHDL**,

  - Set the value of the parameter to **yes** to report violations for the following scenarios:
    - Addition and subtraction expression, if any one operand is a constant and its width is lesser.
    - Maximum range of the RHS is lesser than the maximum range of the LHS and minimum range of the RHS is equal to or greater than the minimum range of the LHS.
    - Minimum range of the RHS is greater than the minimum range of the LHS and maximum range of the RHS is equal to or lesser than the maximum range of the LHS.
- *use_lrm_width*: Default value is **no**. Set this parameter to **yes** to consider the LRM width of integer constants, which is 32 bits.

  On setting the **use_lrm_width** parameter to **yes** or **W164b**, the *W164b* rule does not report a violation if the RHS is of an integer type with width lesser than 32 and LHS is of an integer type with the width being 32.

- *use_carry_bit*: By default, when the **nocheckoverflow** parameter is set **no**, the width of addition and subtraction is considered as the width of the maximum value that the expression can hold. Set the value of this parameter (with default value as W164b) to **no** to calculate the width as the LRM, that is the maximum width of the operand.
- *nocheckoverflow*: Default value is **no**. Set this parameter to **yes** or rule name to check the bit-width as per LRM. Other possible value is the rule name.
- *checkOperatorOverload*: Default value is **yes**. Set this parameter to **no** to evaluate width of the expression without considering overloaded operators. This parameter is applicable for **VHDL** only.
- *check_unsign_overflow*: Default value is **no**. This indicates the rule suppresses the overflow when sign extension is used for unsigned signals in addition or subtraction operation. If you set this parameter to **yes** or **W164b**, the rule does not suppress the overflow and does not report a violation when sign extension is used for unsigned signals in addition or subtraction operation.
- *check_concat_max_width*: Default value is **no**. In this case, no violation is reported when the width of the LHS expression is present between the width of the RHS expression without considering zero concatenated bits and the width of the RHS after adding zero concatenated bits.

  If you set this parameter to **yes**, the RHS width is considered as the width after adding zero concatenated bits. That is, the violation is reported if the LHS width does not match the RHS width after adding zero concatenated bits. This parameter is applicable for **Verilog** only.
- *handle_shift_op*: Default value is **no**. In this case, no violation is reported if the shifted or non-shifted width of a shift expression matches the LHS width of an assignment. But the rule does not calculate the shifted width, if the RHS of the shift expression is non-static. Set this parameter to **shift_left**, **shift_right**, **shift_both**, **no_shift**, or comma separated list of rule names, to compare shifted or non shifted widths for left and right shift expressions.
- *handle_zero_padding*: Default value is no. Set the value of the parameter to yes or rule name to perform leading zero expansion and truncation of RHS and LHS of an assignment. This is performed only if the RHS and LHS of the assignment are static. In addition, if the parameter is set to

yes, the rule performs zero expansion or truncation if the RHS of the assignment is a concatenation and the first element is padded with zero.

- *check_counter_assignment*: Default value is **no**. In this case, no violation is reported for counter cases. Set this parameter to **yes** to report violations for counter cases. You can also set the value of the parameter to turbo. In the *W164b* rule, this parameter is applicable for **VHDL** only.
- *consider_sub_as_add*: By default, when the **nocheckoverflow** parameter is set to **no**, the *W164b* rule considers the width of expression like b -1 as the width of b when there is no underflow due to -ve operator. However, this behavior is limited to a simple - ve operation, where this is a part of arithmetic expression other than +/-. Set the value of the parameter to **yes** to enable the rule to calculate the width of a-b / b-1 as a+b/ b+1 respectively.
- *consider_lrm_for_div*: Default value is **no**. Set the value of the parameter to **yes** to enable the rule to calculate the width of the division to be the maximum width of the operands (LRM width) when *nocheckoverflow* parameter is set to **no.**
- *ignore_nonstatic_leftshift*: Default value is no. Set this parameter to **yes** enable the rule to ignore assignments where the RHS is a left shift operator and both the operands are non-static.
- *process_complete_condop*: Default value is no. Set the value of the parameter to yes to enable the rule checking on both operands of the condop assignment.
- *treat_concat_assign_separately*: Default value is no. Set the value of the parameter to yes to use this parameter to report violation for each bucket assignment in unpacked array separately. For a packed array, a violation is reported for the whole array.
- *report_nonblocking_in_error*: Default value is **no**. Set the value of the parameter to **yes** to report a violation (with **Error** severity) for nonblocking assignments.

Turbo Mode Support

The Turbo mode support is available for this rule. For more information, see the *SpyGlass Lint Turbo Structural User Guide*.

Constraint(s)

None

Messages and Suggested Fix

Verilog

The following message appears at the location where the LHS expression <*lexpr*> of width <*widthl*> of an assignment is greater than the RHS expression <*rexpr*> of width <*widthr*>:

```
[WARNING] LHS: '<lexpr>' width <widthl> is greater than RHS:
<rexpr> width <widthr> in assignment [Hierarchy: ‘<hier-path>’]
```

Where, <*hier-path*> is the complete hierarchical path of the signal.

Potential Issues

A violation is reported when the LHS width is greater than the RHS width.

Consequences of Not Fixing

For more information on consequences of not fixing the violation, click *Consequences of Not Fixing*.

How to Debug and Fix

For more information on how to debug and fix the violation, click *How to Debug and Fix*.

VHDL

Message 1

The following message appears at the location where the width <*widthl*> of LHS expression <*exprl*> of an assignment is greater than the width <*widthr*> of the RHS expression <*exprr*>:

```
[WARNING] LHS: '<exprl>' (width <widthl>) is greater than RHS:
'<exprr>' (width <widthr>) in assignment [Hierarchy: ‘<hier-
path>’]
```

Where, <*hier-path*> is the complete hierarchical path of the signal.

Potential Issues

A violation is reported when the LHS width is greater than the RHS width in an assignment.

Consequences of Not Fixing

For more information on consequences of not fixing the violation, click *Consequences of Not Fixing*.

How to Debug and Fix

For more information on how to debug and fix the violation, click *How to Debug and Fix*.

Message 2

The following message appears at the location where the range of LHS expression *<exprl>* is wider than the range of RHS expression *<exprr>* :

```
[WARNING] Range of LHS '<exprl>' (<lrange>) is wider than the range of RHS '<exprr>' (<rrange>)
```

Where, *<rrange>* and *<lrange>* refer to the range of RHS and LHS expressions, respectively.

Potential Issues

A violation is reported when the range of the LHS expression is wider than the range of the RHS expression.

Consequences of Not Fixing

Bit-width mismatch between the LHS and RHS of a signal assignment is allowed but may lead to inadvertent errors.

How to Debug and Fix

Double-click the violation message. The HDL window highlights the line where the width mismatch is detected.

To resolve the violation, explicitly extend as necessary since it is more readable, instead of relying on the default behavior. Examine the logic to ensure that the truncation does not affect the behavior of the design.

In case of counters specified as integers with range, fix may not be required as user may have put the check to avoid overflow this is the normal practice to use integer range for counters.

Example Code and/or Schematic

Verilog

Example 1

Consider the following example code in which the *strict* parameter reports a violation:

```
wire a,b;
assign a = b + 4 ; //Violation reported with strict
```

Example 2

Consider the following example, in which no violation is reported:

```
A [12:0] = { 3'b000, b[9:0] } ;
```

In this case, the LHS width is 13. This LHS width lies between original RHS width 10, which is calculated without considering leading zeros and RHS width 13, which is calculated after considering leading zeros. Therefore, no violation is reported.

Example 3

Consider another example, in which no violation is reported:

```
A [11:0] = { 3'b000, b[9:0] } ;
```

In this case, the LHS width is 12. This LHS width lies between original RHS width 10, which is calculated without considering leading zeros and RHS width 13, which is calculated after considering leading zeros. Therefore, no violation is reported.

Example 4

Consider another example, in which a violation is reported:

```
A [14:0] = { 3'b000,b[9:0] } ;
```

In this case, the LHS width is 15. This LHS width does not lie between original RHS width 10, which is calculated without considering leading zeros and RHS width 13, which is calculated after considering leading zeros. Therefore, a violation is reported.

Example 5

Consider the following example, in which violations are reported for a concatenation operator:

```
module test3(d,clk,rst,q1, q2, q3, q4, q5);
input d,clk,rst;
output reg [4:0]q1;
output reg [9:0]q2;
output reg [7:0]q3;
output reg [12:0]q4;
```

```
output reg [4:0]q5;
wire [4:0]datain;

always @(posedge clk)
begin
   if(rst)
    begin
      q1 = {5'b00000, datain};
    q2 = {5'b00000, datain};
    q3 = {5'b00000, datain};
      q4 = {5'b00000, datain};
      q5 = {5'b00100, datain};

     q2 = {5'b00000, 3'b000, datain};
    q3 = {5'b00000, 3'b000, datain};

     q2 = {5'b00010, 3'b000, datain};
    q2 = {5'b00000, 3'b010, datain};
    end
end
endmodule
```

The *W164b* rule reports the following violations for the above example:

```
LHS: 'q4' width 13 is greater than RHS: '{5'b00000,
datain}' width 5 in assignment [Hierarchy: ':test3']
```

Example 6

Consider the following example:

```
assign out2 [8:0] = {3'b001,in1,8'b00001111};
```

In the above example, the rule preserves all leading zeros in case of **out2** assignment, which is **3'b001**. Therefore, the width of the assignment is calculated as **12** bits.

Example 7

The following example demonstrates width calculation for the *W164b* rule when the *handle_shift_op* parameter is set to **shift_both**:

```
wire [7:0] w1;
wire [1:0] w2, w3, w4;
```

```
assign w1 = w2 << w3;                       //RHS Width is 5
assign w1 = 3'b110  << w2;                  //RHS Width is 6
assign w1 = w2 << (w3 + w4);                //RHS Width is 5
assign w1 = w2[0] << (w3 + w4 + 5'd0);      //RHS Width is 7
assign w1 = w2 << (w3 - w4);                //RHS Width is 5
assign w1 = w2 << {w3[0],w4[0]};            //RHS Width is 5
assign w1 = w2 << {w3[0],1'b0};             //RHS Width is 4
assign w1 = 3'h001 << w3;                   //RHS Width is 4
assign w1 = {1'b0, w2[0]} << w3[0];         //RHS Width is 2
```

Example 8

The following example demonstrates width calculation for the *W164b* rule when the *handle_shift_op* parameter is set to **shift_both**:

```
module m1();
wire [40:0]a, b, c;

assign a[3:0] = b[3:0] << 1;          //RHS width = 5
assign a[3:0] = b[3:0] << c[1];       //RHS width = 5
assign a[3:0] = 11 << c[0];           //RHS width = 5
assign a[31:0] = 11 << c[0];          //RHS width = 5

assign a[3:0] = b[3:0] >> 1;          //RHS width = 3
assign a[2:0] = b[3:0] >> c[1];       //RHS width = 4
assign a[2:0] = 11 >> c[1];           //RHS width = 4
endmodule
```

Example 9

Consider the following example in which the *check_unsign_overflow* parameter is set to **yes**:

```
wire [5:0] a,b;
wire [8:0] d;
assign d = {{2{a[5]}},a} + {{2{b[5]}},b};
```

The *W164b* rule does not report a violation for the unsigned expression.

Example 10

Consider the following example in which the *W164b* rule reports violations for all three assignments when the *check_static_value* parameter is set to **yes**:

```
wire [4:0] AAA, BBB;
wire [7:0] CCC;
assign CCC = 6'b100111;
assign CCC = 120;
assign CCC = (AAA + 1) - BBB;
```

VHDL

Example 1

Consider the following example code in which the **strict** parameter reports a violation:

```
a (5 downto 0) <= b ( 2 downto 0) + 4;
a (5 downto 0) <= 4 + b (2 downto 0);
```

Example 2

Consider the following example:

```
signal SIG1, SIG3      :Unsigned (3 downto 0)
signal SIG2            :Unsigned (2 downto 0)
signal SIG4            :Unsigned (4 downto 0)
SIG3 < =SIG1+SIG2 //no violation (left width 4
                  right shift 4)
```

In the above example, since both the operands are variables, then the max width is taken. Also, the W164b rule does not report a violation in this case as the LHS width is equal to the RHS width, which is 4.

Example 3

Consider the following example:

```
SIG3 <= SIG1 + 17 //no violation (width is 4 on
                  both sides)
SIG3 <= SIG4 - 17 //violation (left 4, right 5)
```

In the above example , the rule reports violation for the second expression as LHS is 4 and the RHS is 5.

Example 4

Consider the following example:

```
signal SIG1          :Unsigned (3 downto 0)
signal SIG2          :Unsigned (2 downto 0)
signal SIG3          :Unsigned (1 downto 0)
signal SIG4          :Unsigned (4 downto 0)
signal SIG5          :Unsigned (9 downto 0)
SIG4 < =SIG2*SIG3 //no violation (left width 5
                    right width 5)
```

In the above example, the rule does not report violation. Also, as both the operands are variables, the RHS width is the sum of both the variables.

Example 5

Consider the following example:

```
signal SIG1          :Unsigned (3 downto 0)
signal SIG2          :Unsigned (2 downto 0)
signal SIG3          :Unsigned (1 downto 0)
SIG1 < =SIG2*SIG3 //violation (left width 4
                    right width 5)
```

In the above example, since both the operands are variables, the RHS width is the sum of both the variables. Therefore, the LHS width is 4 and RHS width is 5. As a result, the rule reports a violation in this case.

Example 6

Consider the following example:

```
SIG1 <= SIG3 * 17 //violation (left 5, right 4)
SIG5 <= SIG2 * SIG3//violation (left 10, right 5)
```

In the above example, one operand is a variable and the other variable is static. Therefore, the RHS width is **`2*(Variable width)`**.

Example 7

Consider the following example:

```
signal SIG1          :Unsigned (3 downto 0)
signal SIG2          :Unsigned (3 downto 0)
signal SIG3          :Unsigned (1 downto 0)
signal SIG4          :Unsigned (4 downto 0)
SIG1 < =SIG2/SIG3 //No violation (left width 4
                    right width 4)
```

In the above example, both the operands are variables. Therefore, the RHS width is the left operand width, that is, the width of both LHS and RHS is 4. As a result, the rule does not report a violation in this case.

Example 8

Consider the following example:

```
SIG1 <= SIG2 / 17 //no violation (width is 4 on
                    both sides)
SIG1 <= SIG4 / 17 //violation (left 4, right 5)
SIG1 <=17 / SIG2 //no violation (left 4, right 4)
```

In the above example, one operand is a variable and the other variable is static. Therefore, the RHS width is variable width. The rule reports a violation for the second expression as LHS width is 4 and the RHS width is 5. Also, the rule does not report a violation for first and third expression as the LHS and RHS width for both the expressions is 4.

Example 9

Consider the following example:

```
architecture rtl of rgx_pbe_gammacmp_lut is
   type aa_type is array(3 downto 0) of std_logic_vector(11
   downto 0);
   constant const_gamma_table : aa_type := (
      X"1ff",
      X"ff",
      X"2ff",
```

```
      X"00f"
   );

   signal t : std_logic_vector( 11 downto 0);
   signal a : std_logic_vector( 13 downto 0);
begin
   t <= const_gamma_table(to_integer(unsigned(a)));
end rtl;
```

In the above example, by default, a violation is reported because the LHS width is 12 and RHS width is 10, which is maximum width among the initialized elements of the array (that is the width of **x"2ff"**). When the *nocheckoverflow* parameter is set to **yes**, for RHS, the declared width of an element of array is considered (12), hence no violation is reported.

Default Severity Label

Warning

Rule Group

Lint_Elab_Rules

Reports and Related Files

None

W316

Reports extension of extra bits in integer conversion

When to Use

Use this rule to identify extension of extra bits in integer conversion.

Description

The *W316* rule reports integer conversions where the left expression is wider than the right expression. When assigning a value to an integer variable, if the width of the value is less than the width of an integer (32 bits), the value is extended. Constant expressions are also checked.

This rule reports the following types of assignments assuming that **n** is an integer variable, **basedNum** is a constant, and **m** and **k** are variables. All of these have bit-widths less than 32 bits:

```
n = basedNum;
n = basedNum + basedNum;
n = basedNum + m;
n = m;
n = m + k;
```

When an integer variable (left expression) is assigned a value (right expression), such value should be an integer or integer equivalent (that is, 32 bits).

NOTE: *The W316 rule supports generate-if, generate-for, and generate-case blocks.*

Prerequisites

The *W316* rule is switched off by default. You can enable this rule by specifying the **set_goal_option addrules W316** command.

Language

Verilog

Parameters

- *verilint_compat*: Default value is **no**. This indicates the *W316* rule is switched off. Set this parameter to **yes** to run this rule.

Constraints

None

Messages and Suggested Fix

The following message appears at the location where a width <*widthl*> of left expression is more than a width <*widthr*> of right expression in an integer conversion expression.

```
[WARNING] LHS expression width: <widthl> is greater than RHS
expression width: <widthr> [Hierarchy: '<hier-path>']
```

Where, <*hier-path*> is the complete hierarchical path.

Potential Issues

Violation may arise when the LHS expression width is greater than the RHS expression width.

Consequences of Not Fixing

<what are the consequences of not fixing the violation>

How to Debug and Fix

Double-click the violation message. The HDL Viewer window highlights the line where the integer conversions have the left expression wider than the right expression. View the violation message to check the width of the RHS of the assignment statement.

In simple integer arithmetic in a testbench, this message can be waived or ignored. In synthesizable logic, it is best to assign a value explicitly sized to 32 bits to avoid confusion.

Example Code and/or Schematic

Consider the following example:

```
module test(in1,in2,out);
input [7:0] in1;
input [7:0] in2;
output [7:0] out;
reg [7:0] out;

integer n,m,l;
```

```
always @(in1 or in2)
begin
out = in1 & in2;
n = m[31:16] + l[15:0];
end

endmodule
```

In the above example, the *W316* rule reports a violation as the LHS expression width is greater than the RHS expression width.

Default Severity Label

Warning

Rule Group

Verilint_Compat, Lint_Elab_Rules

Reports and Related Files

None

W328

Truncation in constant conversion, without loss of data

Language

Verilog

Rule Description

The W328 rule flags constant assignments where the left expression is narrower than the right expression but the extra bits in the right expressions are all zeros.

The W328 rule flags cases where a value with defined width is being assigned to a vector with smaller width. However the truncated bits are all zeros and no data is lost. While these cases are not directly error conditions, they can become error conditions if the constant changes.

NOTE: *The W328 rule is switched off by default. You can enable this rule by specifying the* **`set_goal_option addrules W328`** *command.*

Message Details

The following message appears at the location of constant assignment where the left expression width *<widthl>* is narrower than the right expression width *<widthr>* but the truncated bits are all zeros:

```
Non-significant bits of constant are truncated during
assignment (lhs width <widthl>, rhs width <widthr>)[Hierarchy:
'<hier-path>']
```

Where, *<hier-path>* is the complete hierarchical path of the containing scope.

Rule Severity

Warning

Suggested Fix

Consider adjusting the constant so truncation will not occur.

Verilint_Compat Rules

The SpyGlass lint product provides the following verilint compatible rules:

Rule	Flags...
W162	Constant integer assignments to signals when the width of the signal is wider than the width of the constant integer
W163	Constant integer assignments to signals when the width of the signal is narrower than the width of the constant integer
W313	integer type to single-bit type conversions
W316	Integer conversions where the left expression is wider than the right expression
W326	event variables used with edges
W328	Constant conversions where the left expression is narrower than the right expression but the extra bits in the right expressions are all zeros
W348	Concatenation expressions where the width of an integer expression is unspecified
W474	Variables that are assigned but not deassigned
W475	Variables that are deassigned without being assigned
W476	Variables that are forced but are not released
W477	Variables that are released without being forced
W488	(Verilog) Bus signals that are in the sensitivity list of an always construct but are not completely read in the construct (VHDL) Arrays that appear in the sensitivity list but all elements of the arrays are not read in the process
W493	Use of shared variables with global scope
W546	Duplicate design unit previously declared in a file at a specified line number

W313

Converting integer to single bit

Language

Verilog

Rule Description

The W313 rule flags violation on assignment statements where an **`integer`** type node is assigned to a single-bit node.

For example, consider the following scenario:

```
reg data;
integer intval;
data = intval; //violation
```

Any type conversion may indicate an unintended error. In this case there may be a loss of significant data.

NOTE: *The W313 rule is switched off by default. You can enable this rule by specifying the* **`set_goal_option addrules W313`** *command.*

Message Details

The following message appears at the location where an **`integer`** type is being converted in to a single-bit type:

`Converting integer to single bit`

Rule Severity

Warning

Suggested Fix

Check to see whether the conversion was intended. Waive violations on such cases.

W348

Unspecified width for integer expression in a concatenation

Language

Verilog

Rule Description

The W348 rule flags concatenation expressions where the width of an integer expression is unspecified.

Concatenations define objects of a specific width. Therefore, each item within the concatenation list must have a defined width. An integer value by default has a width of 32 bits that may probably be larger than you intended and therefore triggers a violation.

The W348 rule does not report violation for unsized based number used in concatenation. Since, such type of error is a syntax error, it is reported by SpyGlass **`STX_VE_384`** rule.

Message Details

The following message appears at the location of a concatenation expression where one of the integer expression *<int-expr>* in the concatenation list does not have a specified width:

`Undefined width for integer expression '<int-expr>' in concatenation`

Rule Severity

Warning

Suggested Fix

Provide an explicit width for constant values, making it clear what width is intended.

For example,**`{sig1, sig2, 3, sig3}`** triggers a violation by the W348 rule. To avoid this violation, replace it with **`{sig1, sig2, 2'b11, sig3}`**.

If you are using an expression like **`3+x`** in concatenation, precompute the

expression and then use as many bits as required in the concatenation. Consider the following example that uses the expression, **3+x**, in concatenation:

```
...{sig1, sig2, 3+x, sig3}...
```

The above code can be replaced with the following:

```
tmp = 3+x;
...{sig1, sig2, tmp[1:0], sig3}...
```

Examples

The following concatenation example is an error because item **17** (being an integer) is considered not to have a defined width:

```
out1 = {in1, in2, 17, in3}
```

Miscellaneous Rules

The SpyGlass lint product provides the following Miscellaneous rules:

Rule	Flags...
W156	Reverse connected buses in instance port maps, signal or variable assignments, and block port maps
W189	Nested **`translate_off`** comments
W192	Empty BEGIN-END blocks
W193	Empty statements (isolated semicolons)
W208	Nested **`translate_on`** comments
W350	Control characters found in strings
W351	Control characters found in comment lines
W433	Multiple top-level modules
W527	Dangling ELSE statements
W546	Duplicate design unit
W701	Included files that are not required for analysis
LINT_abstract01	Generates relevant base policy constraints for block abstractions
LINT_blksgdc01	Migrates relevant top-level lint constraints to block boundaries
LINT_sca_validation	Reports unconstrained port of abstracted block driven by a constant value from top-level

W189

Nested Synopsys translate_off comments

Language

Verilog

Rule Description

The W189 rule flags nested **translate_off** comments.

At the beginning of a Verilog file, translation is enabled for synthesizable code. If required, later in the file a block can be treated as non translatable by placing a **translate_off** comment at the beginning of the block and a **translate_on** comment at the end of the block. Hence a nested **translate_off** comment may indicate a possible missing of corresponding **translate_on** comment.

Message Details

The following message appears when a nested **translate_off** comment is encountered:

`For a translate_off, a translate_on should appear, before next translate_off`

This rule also highlights the previous nested **translate_off** comment.

Severity

Warning

Suggested Fix

Make sure that for every **translate_off**, a **translate_on** appears, before the next **translate_off**.

Examples

Consider the following example:

```
module test1(outp);
    output reg outp;
```

```
    reg [3:0] inp;
    reg [1:0] sel;

//synopsys translate_off
    initial inp=3'b000;
//synopsys translate_off
    initial begin
    sel=2'b00;
end
//synopsys translate_on
  always @(sel,inp)
      case(sel)
        2'b00 :outp = inp[0];
        2'b01 :outp = inp[1];
        2'b10 :outp = inp[2];
        2'b11 :outp = inp[3];
        default:
           outp = 1;
       endcase

endmodule
```

In the above example, the *W189* rule reports a violation because the second *synopsys translate_off* appears before *synopsys translate_on*. The following message is reported by this example:

```
For a translate_off, a translate_on should appear, before next
translate_off
```

W192

Empty block

Language

Verilog

Rule Description

The W192 rule flags empty **begin-end** blocks.

A **begin-end** block containing no logic may create clutter and reduce readability. It may also represent an unintended edit.

Message Details

The following message appears at the start of an empty **begin-end** block:

```
Block does not contain any statement
```

Rule Severity

Warning

Suggested Fix

Check to make sure you really intend the block to be empty.

W193

Empty statement

Language

Verilog

Rule Description

The W193 rule flags empty statements (isolated semicolons).

An empty statement may create clutter and reduce readability. It may also represent an unintended edit or an unintended comment.

You can set the value of the *report_if_blocks_only* parameter to **yes** to report violations only when a semicolon is used with the **if**, **else**, and **else-if** statements.

Message Details

Message 1

The following message appears at the location where an isolated semicolon is encountered:

`Empty statement`

Message 2

The following message appears for an extra semicolon used, if the *report_semicolon* parameter is set to **yes**:

`Extra semicolon used`

Rule Severity

Warning

Suggested Fix

Check to make sure you really intend the statement to be empty.

W208

Nested Synopsys translate_on comments

Language

Verilog

Rule Description

The W208 rule flags the nested **translate_on** comments.

At the beginning of Verilog file , translation is enabled for synthesizable code. If required, later in the file a block can be treated as non translatable by placing a **translate_off** comment at the beginning of the block and a **translate_on** comment at the end of the block. Hence a nested **translate_on** comment may indicate a possible missing of corresponding **translate_off** comment.

Message Details

The following message appears when a nested **translate_on** comment is encountered:

`No associated translate_off found`

Severity

Warning

Suggested Fix

Make sure that no **translate_on** comment is followed by a next **translate_on**, unless a **translate_off** appears before this next **translate_on**.

W350

A control character appears inside a string

Language

Verilog

Rule Description

The W350 rule flags control characters found in strings.

Some EDA tools may not be able to handle control characters in strings.

Message Details

The following message appears at the line where a control character is encountered in a string <*string*> after substring <*substring*>:

`Control character found after substring <substring> in the string <string>`

Rule Severity

Warning

Suggested Fix

If your editor has a method to display control characters, use that to locate the problem. Otherwise, delete and re-enter the string.

W351

A control character appears inside a comment

Language

Verilog, VHDL

Rule Description

The W351 rule flags control characters found in comment lines.

Some EDA tools may not be able to handle control characters in comment line.

Message Details

The following message appears at the line where a control character is encountered in a comment line:

`Control character in comment`

Rule Severity

Warning

Suggested Fix

If your editor has a method to display control characters, use that to locate the problem. Otherwise, delete and re-enter the string.

W433

More than one top-level design unit

Language

Verilog

Rule Description

The W433 rule flags multiple top-level modules in the design.

Designs containing more than one top-level module may be a mistake. This can occur due to the following reasons:

- The design is being analyzed together with library elements which are referenced directly (rather than through **`set_option y <directory-path>`** or **`set_option v <file-name>`** commands), and all the library elements are not used in the design. As a result, all the unused elements become top-level units.
- The design is missing some intermediate-level components, due to which the lower-level components in the hierarchy do not get linked. As a result, the lower level components become top-level units.
- The design contains certain modules that are specified as stop-levels but no top-level is specified. As a result, units appearing under those stop modules and not elsewhere in the design become top-level units.

Message Details

The following message appears at the first line of the first top-level module of a design that has more than one top-level module:

`Design contains more than one top level module`

Rule Severity

Info

Suggested Fix

Make reference to libraries through the **`set_option y <directory-path>`** or

set_option v <file-name> commands options wherever possible. This will ensure only needed library elements will be included in the analysis. If you still see multiple top-level units, use the **set_option top <name>** command to select the one you want to analyze and exclude others.

W527

Dangling else in sequence of if conditions. Make sure nesting is correct

Language

Verilog

Rule Description

The W527 rule flags dangling **else** statements (that is, **else** statements following a sequence of **if** statements).

It is recommended to use **begin-end** blocks to force the correct order of evaluation.

Parameters

avoid_seq_logic: Default value is **no**. If you set the value of this parameter to **yes**, the *W527* rule reports violation for combinational **always** blocks and does not report violation for sequential **always** blocks.

Message Details

The following message appears at the location of an **if** statement that is inferred not to have a corresponding **else** statement:

```
Potential dangling 'else' statement of 'if' statement
(<condition expression>) - check your logic
```

Rule Severity

Warning

Suggested Fix

Use **begin..end** to force the correct evaluation. This removes any ambiguity.

Examples

Consider the following example:

```
if (x)
  if (y) do_B
else do_C
```

Looking at the structure of the construct, it may appear that **do_C** block will be executed only if **x** is false irrespective of the value of **y**.

However, the Verilog semantics associate the **else** statement with the immediately preceding **if** statement rather than the outermost **if** statement. Thus, the above construct will be evaluated as follows:

```
if (x)
  if (y) do_B
  else do_C
```

The **do_C** block is evaluated only if **x** is true and **y** is false.

W546

Duplicate design unit

Language

Verilog

Rule Description

The W546 rule flags duplicate modules.

This error may occur as a result of including more files than required in the analysis when you have multiple revisions of a design within one directory.

NOTE: *The W546 rule is switched off by default. You can enable this rule by either specifying the* **`set_goal_option addrules W546`** *command or by setting the verilint_compat rule parameter to* **`yes`**.

Parameters

waiver_compat: Default value is no. If you set the value of this parameter to **`yes`** or **`<rule-name>`**, it ensures that the rule does not generate the line number information in the first run itself. Thus waivers work correctly even if the line numbers of the RTL gets changed in the subsequent runs.

Message Details

The following message appears at the location where a duplicate module name *<module-name>* is encountered when a module with the same name has been encountered in file *<file-name>* at line *<num>*:

```
Duplicate design unit '<module-name>', previously declared in file '<file-name>' at line no '<num>'
```

Rule Severity

Fatal

Suggested Fix

Determine which source file contains the correct module description and exclude any other such files from the analysis.

W701

Included file is not used

Language

Verilog

Rule Description

The W701 rule flags included files that are not required for analysis.

The W701 rule checks for the presence of the following constructs only in the included file:

- Macros
- Parameters
- Task declarations
- Function declarations
- Module definitions
- User-defined primitives (UDPs)
- Timescale directive
- Interface definitions
- Signal definitions

It is recommended to remove **`include`** statements for unwanted files from the design to save the time wasted in analyzing such files.

Message Details

The following message appears at the location where an unused file *<file-name>* is included:

`File: <file-name> is 'included but not used`

Rule Severity

Warning

Suggested Fix

Consider removing the include statement

LINT_abstract01

Generates relevant base policy constraints for block abstraction

When to Use

Use this rule to generate an abstracted model of a block.

For information on using the SoC abstraction flow, refer to the *SpyGlass SoC Methodology Guide*.

Rule Description

This rule generates an abstracted model of a block, which is used during SoC-level validation and verification.

Prerequisite

To enable this rule, run the **`block_abstract`** goal.

If you are running this rule using -**`rules`** option in a goal file, specify the following command in the project file:

```
set_option block_abstract yes
```

Language

Verilog, VHDL

Default Weight

2

Parameter(s)

- *disable_latch_crossing*: Default value is no. Set the value of the parameter to yes to enable the rule to stop traversal at latches.
- *latch_effort_level*: Default value is -1. Set the value of the parameter to integer value from -1 to MAX_INT value to Specifies the effort that spyglass should make while doing traversal
- *use_new_flow*: Default value is no. Set the value of the parameter to yes to enable the LINT_abstract01 rule to use memory caching to improve runtime.

Constraint(s)

None

Messages and Suggested Fix

`[INFO] Abstract view for design <du-name> successfully created`

Where, *<du-name>* is the design unit name.

Potential Issues

Not Applicable

Consequences of Not Fixing

Not Applicable

How to Debug and Fix

Not Applicable

Example Code and/or Schematic

Consider the following block RTL specified for generating an Abstract View:

// Block RTL

```
module top (output op1, op2, input in1,
            in2, in3, rst, clk, tm);
    reg   ff1, ff2, ff3;
    wire  d1, clkn;
    assign clkn = !clk;
    assign d1   = in1 & in2;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            ff1 <= 1'b0;
            ff2 <= 1'b0;
        end
        else begin
            ff1 <= d1;
            ff2 <= ff1;
        end
    end
 always @(posedge clkn or posedge rst) begin
        if (rst) begin
            ff3 <= 1'b0;
        end
        else begin
            ff3 <= !(tm & in2);
        end
    end
    assign op1 = ff2 & ff3;
    assign op2 = ff3 & in3;
endmodule
```

// Block SGDC (blockA.sgdc)

```
current_design top
set_case_analysis -name tm -value 1
```

Following is the schematic of the above design:

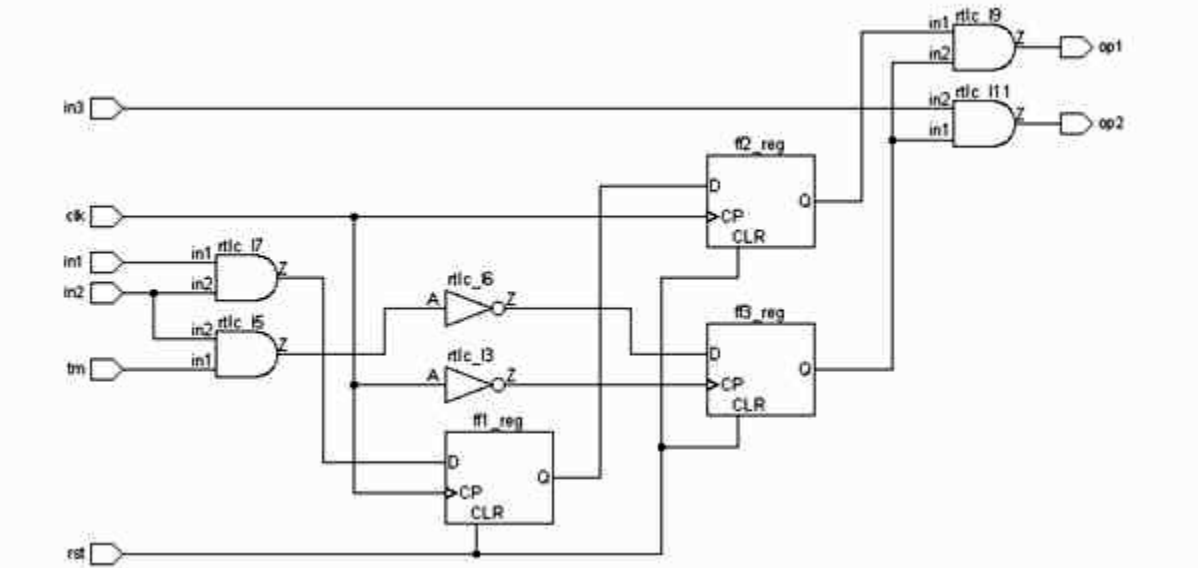


FIGURE 4. Incremental schematic

When you specify the block RTL and block SGDC files to SpyGlass and run the **`lint_abstract`** goal, the following SGDC file is generated representing the Abstract View:

```
current_design "blockA"

abstract_port -ports "op1" -connected_inst "\blockA.ff2_reg
" -inst_master "RTL_FDC" -inst_pin "Q" -path_logic combo
-path_polarity buf -mode set_case_analysis -scope base

abstract_port -ports "op2" -connected_inst "\blockA.ff3_reg
" -inst_master "RTL_FDC" -inst_pin "Q" -path_logic combo
-path_polarity buf -mode set_case_analysis -scope base

abstract_port -ports "in1" -connected_inst "\blockA.ff1_reg
" -inst_master "RTL_FDC" -inst_pin "D" -path_logic combo
-path_polarity buf -mode set_case_analysis -scope base
```

```
abstract_port -ports "in2" -connected_inst "\blockA.ff3_reg
" -inst_master "RTL_FDC" -inst_pin "D" -path_logic combo
-path_polarity inv -mode set_case_analysis -scope base

abstract_port -ports "in2" -connected_inst "\blockA.ff1_reg
" -inst_master "RTL_FDC" -inst_pin "D" -path_logic combo
-path_polarity buf -mode set_case_analysis -scope base

abstract_port -ports "in3" -related_ports "op2" -path_logic
combo -path_polarity buf -mode set_case_analysis -scope base

abstract_port -ports "rst" -connected_inst "\blockA.ff2_reg
" -inst_master "RTL_FDC" -inst_pin "CLR" -path_logic buf
-path_polarity buf -mode set_case_analysis -scope base

abstract_port -ports "clk" -connected_inst "\blockA.ff3_reg
" -inst_master "RTL_FDC" -inst_pin "CP" -path_logic inv
-path_polarity inv -mode set_case_analysis -scope base

abstract_port -ports "clk" -connected_inst "\blockA.ff2_reg
" -inst_master "RTL_FDC" -inst_pin "CP" -path_logic buf
-path_polarity buf -mode set_case_analysis -scope base
```

For detailed explanation, refer to *Example - Generating an Abstract View in SpyGlass lint* section of the *SpyGlass SoC Methodology Guide*.

Default Severity Label

Info

Rule Group

Miscellaneous rules

Reports and Related Files

The LINT_abstract01 generates the **<block-name>_abstract.sgdc** file in the spyglass_reports/abstract_view/lint/ directory. This file contains

specifications of the *abstract_port* and *set_case_analysis* constraints generated for a block.

LINT_blksgdc01

Migrates relevant top-level lint constraints to block boundaries

When to Use

Use this rule to generate a block-level SGDC file from top level *set_case_analysis* constraint or due to supply net.

Rule Description

The *LINT_blksgdc01* rule generates block-level **set_case_analysis** constraints by migrating top-level constraints to block boundary.

The rule generates block-level constraints based on the following:

- If any **set_case_analysis** constraint is specified on the top-level port.
- If input of a block that you want to abstract is connected to a supply net.

Prerequisite

Top-level SGDC file

Language

Verilog, VHDL

Default Weight

2

Parameter(s)

None

Constraint(s)

set_case_analysis

Messages and Suggested Fix

The following information message is reported when an SGDC file is generated using the top-level constraint migration:

```
[INFO] SGDC file generated for instance '<inst-name>' (block:
'<block-name>') using top level constraints migration
```

Potential Issues

None

Consequences of Not Fixing

None

How to Debug and Fix

None

Example Code and/or Schematic

In the schematic below, block **BLK** is instantiated inside the TOP. From the top-level SGDC, the *set_case_analysis* constraint is set on the port **in** to **0**.

- The block **BLK** should be abstracted with the following **set_case_analysis** constraints:

```
set_case_analysis -name in -value 1
set_case_analysis -name in1 -value 0
set_case_analysis -name in2 -value 0
```

- In the top-down flow, the block-level SGDC having above **set_case_analysis** constraints is created itself.

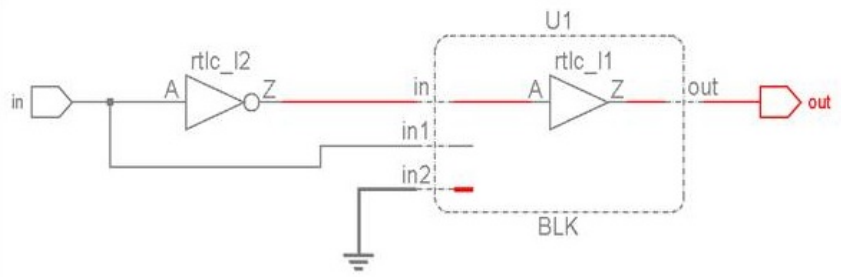


FIGURE 5. Incremental schematic

The following are the steps of the top-down flow for the above example:

1. Create the **top.sgdc** constraint file and specify the **set_case_anlysis** constraint on the top-level port, if required.
2. In the **top.sgdc** file, specify the block that you want to abstract, as follows:

```
sgdc -export blk
```

3. The *LINT_blksgdc01* rule creates the block-level SGDC, having *set_case_analysis* constraints mentioned *above*.
4. This block-level SGDC is passed to the block abstraction.

Default Severity Label

Info

Rule Group

Miscellaneous rules

Reports and Related Files

None

LINT_MULTIASSIGN_BLOCKING_SIG

Signal may be multiply assigned in blocking manner (beside initialization) in the same scope.

When to Use

Use this rule to identify signals that are assigned multiple times, in blocking manner in the same scope.

Rule Description

The rule is equivalent to the W415a rule and is used with the parameter *checkblocking*`=yes`.

Parameter(s)

The rule considers the following parameters (same as W415a).

- *waiver_compat*
- *reportsimilarassgn*
- *ignore_reinitialization*
- *check_initialization_assignment*
- *ignore_bitwiseor_assignment*
- *ignore_multi_assign_in_forloop*

Constraint(s)

None

Messages and Suggested Fix

```
[WARNING] Signal <signal_name> is being assigned multiple times
(previous assignment at line <line_no>) in same always block
[Hierarchy:'<hier_name>']
```

Potential Issues

The violation message is reported when a signal is assigned multiple times in the same block.

Consequences of Not Fixing

Not fixing the violation may result in unexpected rule behavior.

How to Debug and Fix

To fix the violation, ensure that the signal is not assigned multiple times in the same block.

Example Code and/or Schematic

```
module test (a, b, c, d);
input a, b;
output c;
output d;
reg c;
reg d;
always @(a or b)
begin
c < = a;
c < = b;
end
always (@a or b)
begin
d = b;
d = a;
end
endmodule
```

Default Severity Label

Warning

Reports and Related Files

None

LINT_MULTIASSIGN_NONBLOCKING_SIG

Signal may be multiply assigned in non-blocking manner (beside initialization) in the same scope.

When to Use

Use this rule to identify signals that are assigned multiple times, in a non-blocking manner in the same scope.

Rule Description

The rule is equivalent to the W415a rule and is used with the parameter *checknonblocking*`=yes`.

Parameter(s)

The rule considers the following parameters (same as W415a).

- *waiver_compat*
- *reportsimilarassgn*
- *ignore_reinitialization*
- *check_initialization_assignment*
- *ignore_bitwiseor_assignment*
- *ignore_multi_assign_in_forloop*
- *ignore_nonBlockCondition*

Constraint(s)

None

Messages and Suggested Fix

```
[WARNING] Signal <signal_name> is being assigned multiple times
(previous assignment at line <line_no>) in same always block
[Hierarchy:'<hier_name>']
```

Potential Issues

The violation message is reported when a signal is assigned multiple times in the same block.

Consequences of Not Fixing

Not fixing the violation may result in unexpected rule behavior.

How to Debug and Fix

To fix the violation, ensure that the signal is not assigned multiple times in the same block.

Example Code and/or Schematic

```
output d;
reg c;
reg d;
always @(a or b)
begin
c <= c
if (a)
c <= c+1;
else
c <= c-1;
end
always @(a or b)
begin
c < = c;
c < = a; //c<=a ? c + 1 : c-1;
end
endmodule
```

Default Severity Label

Warning

Reports and Related Files

None

LINT_sca_validation

Reports unconstrained port of abstracted block driven by a constant value from top-level

When to Use

Use this rule to check the top-level design for propagation of constant value to block port.

Rule Description

This rule reports a violation for the following cases:

- If constant value is propagated to the block port from top-level and no *set_case_analysis* is specified on the block port.
- If set_case_analysis constraint is specified on block port but constant value is not propagated from the top-level.
- If there is any value mismatch between top-level port and block level port.

Prerequisites

Following are the prerequisites of running this rule:

- Specify the following command in the project file:

  ```
  set_option sgdc_validate yes
  ```

- Specify the *set_case_analysis* constraint.

Language

Verilog, VHDL

Default Weight

5

Parameter(s)

None

Constraint(s)

set_case_analysis: Specifies the case analysis conditions.

Messages and Suggested Fix

Message 1

```
[WARNING] Simulated value '<value>' reaches to port '<port>' of
block instance '<inst_name>' (block: '<block_name>') however no
set_case_analysis is specified in block level constraint file
```

Arguments

- Simulated value, <value>
- Port name, <port>
- Instance name, <inst_name>
- Block name, <block_name>

Potential Issues

A violation message appears, if a constant value propagates from the top-level, but the abstracted block port is not constrained with the *set_case_analysis* constraint.

Consequences of Not Fixing

If you do not fix this violation, the design may not operate in the desired mode.

How to Debug and Fix

To fix this violation, perform the following steps:

1. Analyze the top-level design for propagation of a constant value to the block port.
2. Specify the **set_case_analysis** constraint on the block port.

Message 2

```
[WARNING] Simulated value does not reach to port '<port>' of
block instance '<inst_name>' (block: '<block_name>' ) where as
set_case_analysis defined in block-level constraint file
```

Arguments

- Simulated value, <value>
- Port name, <port>
- Instance name, <inst_name>
- Block name, <block_name>

Potential Issues

This violation appears, if a block port is constrained with the *set_case_analysis* constraint, but no constant value propagates from the top-level.

Consequences of Not Fixing

If you do not fix this violation, the reported ports can block or enable propagation of unexpected signals across the abstracted block.

How to Debug and Fix

To fix this violation, perform the following steps:

1. Analyze the top-level design for propagation of a constant value to the block port.
2. Specify the **set_case_analysis** constraint, if a valid constant value does not reach the block port.

Message 3

```
[WARNING] Simulated value at port '<port>' of instance
'<inst_name>' (block: '<block_name>') is '<sim_value>' but
specified value in block level constraint file is '<value>'
```

Arguments

- Port name, <port>
- Instance name, <inst_name>
- Block name, <block_name>
- Simulated value, <sim_value>
- Value in constraint file, <value>

Potential Issues

This violation appears if block-level ports are constrained to values that do not match with constant values propagated from the top-level.

Consequences of Not Fixing

If you do not fix this violation, the following issues may arise depending upon different situations:

- If the specified value at the block-level port is incorrect, block-level lint verification is inaccurate.

- If the specified value at the block-level port is correct but constant propagation at the top-level is incorrect, it indicates a logical issue at the top-level because of which incorrect value is propagated at the block-level.

How to Debug and Fix

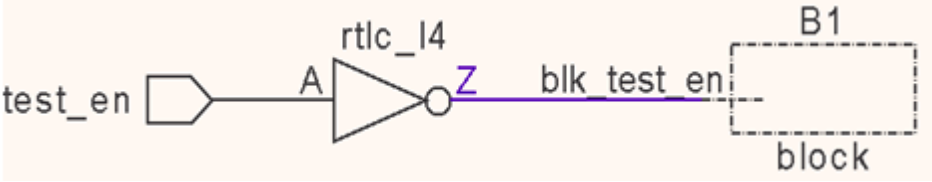
To fix this violation, perform the following steps:

1. Check the value specification of the *set_case_analysis* constraint on a block port.
2. Analyze the top-level design for propagation of a constant value to the block port.

Example Code and/or Schematic

Example 1

Consider the following schematic of a violation reported by this rule:




```
// SGDC file for the block module

current_design block
set_case_analysis -name block.blk_test_en -value 0
```

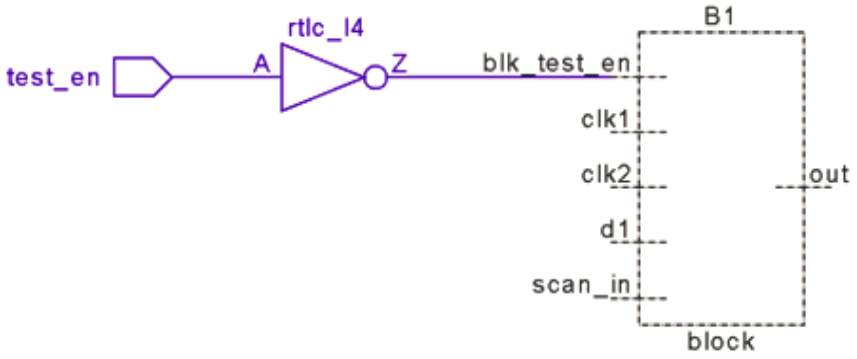
FIGURE 6. Incremental schematic

In the top-level SGDC file, the *set_case_analysis* constraint is not defined for the **`test_en`** signal.

In this case, the *LINT_sca_validation* rule reports a violation because at the top-level, no constant value propagates at the net connected to the **`blk_test_en`** block pin, whereas the constant value is specified in the block-level SGDC file.

Example 2

Consider the following schematic of a violation reported by this rule:




```
// Top-level SGDC file
current_design top
set_case_analysis -name top.test_en -value 0

// Block-level SGDC file
current_design block
set_case_analysis -name block.blk_test_en -value 0
```

FIGURE 7. Incremental schematic

In the above example, the *LINT_sca_validation* rule reports a violation because at the top-level, the constant value **1** is propagated at the net connected to the **blk_test_en** block pin, whereas in the block-level SGDC file, the value specified is **0**.

To fix this violation, modify the *set_case_analysis* constraint specification of the block-level SGDC file to the following:

```
set_case_analysis -name block.blk_test_en -value 1
```

Default Severity Label

Warning

Rule Group

Miscellaneous rules

Reports and Related Files

No related reports or files.

Appendix: SGDC Constraints

SpyGlass Design Constraints (SGDC) provides additional design information that is not apparent in an RTL.

In addition, you can restrict SpyGlass analysis to certain objects in a design by specifying these objects by using SGDC commands.

The following table lists the SGDC commands used by SpyGlass lint product:

Lint		
abstract_port	*assume_path*	*set_case_analysis*

List of Topics