Riviera™ PRO

Documentation

Verilog HDL RTL Design Style Checks

Conforming STARC Specification v. 2.0

www.aldec.com

This page is intentionally left blank.

Table of Contents

Chapter 1 Basic Design Constraints

1.1 Naming conventions

1.1.1 Basic naming conventions

STARC_VLOG 1.1.1.1

RULE NAME	**File names should be as follows: "<module name>.v".**	
MESSAGE	**Single file should contain single topmost-level module with the same name.**	
	DETAIL-1	*Name "{TopName}" of the topmost-level module differs from the file "{FileName}" name.*
	DETAIL-2	*Name of the topmost-level module "{TopName}" matches to file name, but this module is described along with another topmost-level modules. Multiple modules can be included in a file only in case when they have a tree hierarchical structure.*
	DETAIL-3	*Topmost-level module "{TopName}" is detected.*
	DETAIL-4	*Topmost-level module "{TopName}" is detected. Multiple modules can be included in a file only in case when they have a tree hierarchical structure.*
PROBLEM DESCRIPTION	Names that make debugging more efficient should be carefully chosen. They help to understand project structure because if different naming conventions are used by different designers, circuits that were divided into sections by multiple designers will be difficult to understand when they are integrated. A single file should contain a single module, but this is unfit for a large design with many files since it becomes difficult to handle. In this case, multiple modules can be included in a file, but modules which have no relation to one another should not be included in the same file. A single file should include modules which have a tree hierarchical structure. The top module name and file name should be the same.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects all topmost-level modules between described in the translation unit hierarchy: – if there is a single topmost-level module in the translation unit and its name differs from the name of the file => violation (detail-2); – if primary module is described along with another topmost-level module(s) in the translation unit: – if one of topmost-level modules has the same name as translation unit => violation (detail-2 + detail-3); – if there is no topmost-level module with the name that is equal to the translation unit name => violation (detail-4). Note-1: translation unit is a source file specified for compilation session (for example: **alog** *ram.v*) and it can include another files (content of such files is considered as part of current translation unit). Note-2: name comparison is case sensitive.	

EXAMPLE-1: [1] file name is the same as top-most module name;
[2] file contains `include directive in the global scope;
[3] included file contains definition of module with another name => violation(detail-2 + detail-3)

```
//file top1.v

`include “top2.v”

module top1;

    ...

endmodule
```

Single file should contain single -level module w ith the same name.

Name of the topmost-level module “top1” matches to file name, but this module is described along w ith another topmost-level modules. Multiple modules can be included in a file only in case w hen they have a tree hierarchical structure.

```
//file top2.v

module top2;

    ...

end
```

Topmost-level module “top2” is detected.

EXAMPLE-2: [1] file contains two modules;
[2] one module is instantiated in another => hierarchical structure presents;
[3] name of the top-most level is the same as file name => no violation.

```
//file top.v

module top ( ... );

    ...

    ff ff ( .clk(clk), .d(d), .q(q) )

endmodule

module ff ( clk, d, q );

    ...

end
```

STARC_VLOG 1.1.1.2

RULE NAME	**Only alphanumeric characters and the underscore '_' should be used, and the first character should be a letter of the alphabet**	
MESSAGE-1	**{ObjectClass} name "{ObjectName}" violates basic naming convention. Only alphanumeric characters and the underscore '_' should be used, and the first character should be a letter of the alphabet.**	
MESSAGE-2	**{ObjectClass} name "{ObjectName}" is an escaped identifier. Try not to use escaped identifiers in pure Verilog designs.**	
MESSAGE-3	**{ObjectClass} name "{ObjectName}" matches forbidden pattern "{RegExp}".**	
MESSAGE-4	**{ObjectClass} name "{ObjectName}" does not match legal pattern "{RegExp}".**	
PROBLEM DESCRIPTION	Object identifiers must facilitate understanding the function of the underlying HDL description especially if multiple designers work on a project. Therefore, it is recommended to use consistent naming conventions. In case of Verilog-HDL, any characters or keywords can be used if an escaped identifier is used. However, it is impossible in VHDL(87) and problems may occur later in the design flow. That is why it is not recommended to use symbols other than alphanumeric characters and the '_'(underscore) or to use escaped identifiers.	
	LEVEL	RULE

CHECKER BEHAVIOR

Checker verifies names of each object and provides built-in checks which activated via DEFAULT_CHECKS parameter:

- if DEFAULT_CHECKS == "1" built-in checks are activated:
  - if the name does not satisfy requirements that only alphanumeric characters and the underscore '_' should be used, and the first character is a letter and it is not an escaped identifier => violation (message-1);
  - if the name is an escaped identifier => violation (message-2);
- if DEFAULT_CHECKS == "0" built-in checks are not performed.

Checker provides also mechanism for regular expression-based checking:

- if REGEXP_MATCH == "deny" and object name matches forbidden pattern => violation (message-3);
- if REGEXP_MATCH == "allow" and object name does not match legal pattern => violation (message-4).

Note-1: if an escaped identifier is a simple identifier and is not a Verilog keyword => no violation.

Note-2: following parameters are supported by the checker:

- parameter DEFAULT_CHECKS defines whether to perform standard checks: "1" means yes (default), "0" means no;
- parameter REGEXP_MATCH controls custom regular expressions checking:
  - "deny" - warnings are issued on matched identifiers;
  - "allow" - warnings are on unmatched identifiers;
  - empty string ("") - do not perform regular expression matching (default);
- REGEXP defines the regular expression for matching against identifiers;
- REGEXP_CASE_SENSITIVE controls the mode of regular expression matching:
  - "1" - case sensitivity (default);
  - "0" - case insensitivity.

Note-3: {ObjectClass} is defined by the following table:

Verilog construction	**{ObjectClass}**
module	module
cell (module marked with `celldefine directive)	cell

RULE NAME	Only alphanumeric characters and the underscore '_' should be used, and the first character should be a letter of the alphabet

Verilog construction	{ObjectClass}
module port (input/output/inout)	port
signal (reg or net of any type, including implicitly declared wires)	signal
parameter	parameter
task	task
function	function
task or function port	port
`define MACRO macro_value	defined macro
named block	block
module instantiation	instance
concatenation	concatenation
constant expression	constant expression
non-constant expression	expression

EXAMPLE-1: [1] module name does not satisfy described condition => violation (message-1)

```
module 1top$;
     ...
endmodule
```

Module name "_top$" violates basic naming convention. Only alphanumeric characters and the underscore '_' should be used, and the first character should be a letter of the alphabet.

EXAMPLE-2: [1] parameter name is an escaped identifier => violation (message-2)

```
parameter [8:0] \~!@#$%^&:-);
```

Parameter name "\~!@#$%^&:-)" is an escaped identifier. Try not to use escaped identifiers in pure Verilog designs.

EXAMPLE-3: [1] parameter name is an escaped identifier that does not contain prohibited symbols but is Verilog-HDL keyword => violation (message-2)

```
reg  \reg ;
```

Parameter name "\reg" is an escaped identifier. Try not to use escaped identifiers in pure Verilog designs.

EXAMPLE-4: [1] parameter name is an escaped identifier but it is a simple identifier => no violation

```
wire [1:0] \non_esc_name ;
```

EXAMPLE-5: [1] parameter REGEXP_MATCH == "deny";
[2] parameter REGEXP = "^denied";
[3] signal name matches forbidden pattern => violation (message-3).

```
reg denied_name;
```

Signal name "denied_name" matches forbidden pattern "^denied".

STARC_VLOG 1.1.1.3

RULE NAME	**Only alphanumeric characters and the underscore '_' should be used, and the first character should be a letter of the alphabet**
MESSAGE-1	**{ObjectClass} name "{ObjectName}" violates basic naming convention. Name belongs to keywords category "{CategoryName}".**
MESSAGE-2	**{ObjectClass} name "{ObjectName}" violates basic naming convention. Lower-case version of the name corresponds to Verilog-HDL keyword "{Keyword}".**
PROBLEM DESCRIPTION	Object identifiers must facilitate understanding the function of the underlying HDL description. Verilog is a case sensitive language. Consequently, the designer is allowed to use reserved language identifiers for object names that differ only by letter case: e.g., INPUT, Reg. However, this is not recommended because it could lead to confusion. Moreover, in addition to Verilog-keywords, VHDL and software keywords should not be used (problems may occur later in the design flow). There are no particular problems with EDIF, SDF and Windows keywords, but for safety they should also be avoided in descriptions. **LEVEL** RULE
CHECKER BEHAVIOR	Checker verifies that name of each object does not belong to one of the restricted sets of keywords: – if name belongs to Verilog set (built-in) => violation (message-2) – if name belongs to VHDL/EDIF/SDF/Windows set => violation (message-1) Note-1: sets of keywords are defined in the configuration file: – additional sets can be defined, for example: – KEYWORD_CATEGORIES = ["EDIF", "SDF", "Windows", "PSL", "SYSTEM_VERILOG"] – each set (except built-in Verilog) can be extended with any keywords necessary for the target design (using configuration file), for example: – KEYWORD_LIST_WINDOWS = ["CON", "AUX", "COM1", "COM2", ...] Note-2: see rule 1.1.1.2 for {ObjectClass} substitution table

EXAMPLE-1: [1] module name belongs to Verilog-keywords category => violation (message-2)

```
module Task;
    ...
endmodule
```

Module name "Task" violates basic naming convention. Low er-case version of the name corresponds to Verilog-HDL keyw ord "task".

EXAMPLE-2: [1] wire name belongs to Windows-keywords category => violation (message-1)

```
wire [8:0] COM1;
```

Signal name "COM1" violates basic naming convention. Name belongs to keyw ords category "Window s".

STARC_VLOG 1.1.1.4

RULE NAME	**Names containing "VDD ", "VSS", "VCC", "GND" or "VREF" must not be used (upper case or lower case or mixed case)**
MESSAGE	**{ObjectClass} name "{ObjectName}" violates basic naming convention. It contains erroneous part(s): {list_of_violated_name_fragments}.**
PROBLEM DESCRIPTION	Names containing "VDD", "VSS", "VCC", "GND", "VREF" must not be used (uppercase or lowercase).
	LEVEL RULE
CHECKER BEHAVIOR	Checker verifies object names: – if there is any fragment from the restricted set => violation Matching algorithm is case-insensitive. Note-1: set of fragments is configurable Note-2: see rule 1.1.1.2 for {ObjectClass} substitution table

EXAMPLE-1: [1] register name is equal to the restricted fragment "GND" => violation

```
reg GND;
```

Signal name "GND" violates basic naming convention. It contains erroneous part(s): ["GND"]

EXAMPLE-2: [1] wire name includes restricted fragment "VCC" => violation

```
wire NetToVcc;
```

Signal name "NetToVcc" violates basic naming convention. It contains erroneous part(s): ["VCC"]

EXAMPLE-3: [1] port name includes two restricted fragments: "VREF" and "VSS" => violation
[2] note: "VREF" is specified twice, but displayed once in the violation message

```
input VREF_vss2Vref;
```

Port name "VREF_vss2Vref" violates basic naming convention. It contains erroneous part(s): ["VREF", "VSS"]

STARC_VLOG 1.1.1.5

RULE NAME	**Do not distinguish names by using upper or lower case English letters (Abc, abc)**	
MESSAGE-1	**{ObjectClass} name “{ObjectName}” has duplicates which differ only by letter case. Such naming style should be avoided.**	
	DETAIL	*{ObjectClass} name “{ObjectName}” differs only by letter case.*
MESSAGE-2	**{ObjectClass} name “{ObjectName}” has duplicates. Avoid naming different objects with the same identifier.**	
	DETAIL	*Duplicate identifier: {ObjectClass} name “{ObjectName}”.*
PROBLEM DESCRIPTION	Semiconductor naming conventions often limit the usage of upper case or lower case letters. Such naming modifications may alter some of the names distinguished only by letter case quite significantly and post-layout verifications becomes very difficult to understand. So all names should not be distinguished only by the cases of the letters used (e.g. abc vs. ABC).	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker search in each scope and all upper scopes for: – all identifiers differ only by case of the letters => violation (message-1) – all equal identifiers => violation (message-2) Note-1: if any identifiers are in parallel scopes => no rule violation. Note-2: details messages are produced for all duplications or differs only by letter case identifiers. Note-3: see 1.1.1.2 for {ObjectClass} description.	

EXAMPLE-1: [1] module name differs only by letter case from task declared in this module i.e. in sub-scope => violation (message-1);

[2] the same module name is equal to the name of signal declared in one of lower level of scope hierarchy => violation (message-2);

[3] name of the first module is equal to the name of signal declared in another module (top2), i.e. in parallel scope => no violation.

```
module top1;

    ...
    task Res;
    ...
            begin : block1

                reg top1;
                ...
            end

    endtask

    task Top1;
        ...
    endtask

    ...

endmodule

module top2 (in1, out1);

    ...
    reg top1;
    ...

endmodule
```

Module name “top1” has duplicates w hich differ only by letter case. Such naming style should be avoided.

Module name “top1” has duplicates. Avoid naming different objects w ith the same identifier.

Duplicate identifier: Signal name “top1”.

Task name “Top1” differs only by letter case.

STARC_VLOG 1.1.1.6

RULE NAME	**Do not use an '_'(underscore) at the end of the primary port name or module name , and do not use '_' consecutively**
MESSAGE	**{ObjectClass} name "{ObjectName}" ends with underscore('_') or contains consecutive underscores.**
PROBLEM DESCRIPTION	In VHDL there is a convention which states that the final character must not be underscore ('_'). Also this character is sometimes used in gate level verification by VITAL. So it is recommended to avoid using underscore at the end of modules and primary port names to avoid problems in mixed-language projects. When two or more underscores are used consecutively it is often difficult to define exact number of underscore characters. Such situation may lead to identifiers mismatching so it is recommended to avoid it too.
	LEVEL: RECOMMENDATION 1
CHECKER BEHAVIOR	Checker collects identifiers of all modules and primary ports. – if identifier ends with underscore or contain two and more underscores consecutively => violation Note-1: see 1.1.1.2 for {ObjectClass} description.

EXAMPLE-1: [1] module name ends with underscore => violation;

[2] name of primary port contains two underscores consecutively => violation

```
module top_ ( in__1 );

    ...

endmodule
```

Port name "in__1" ends w ith underscore('_') or contains consecutive underscores.

Module name "top_" ends w ith underscore('_') or contains consecutive underscores.

EXAMPLE-2: [1] module name contain only one underscore character => no violation;

[2] identifier contain two underscores consecutively, but parameter is not included in list of checking object => no violation;

[3] Input of task violates current restrictions for identifiers, but it is not primary port => no violation.

```
module to_p;

    ...

    parameter para___m = 1'b0;

    ...

    task task1;

        input in__1;
        ...

    endtask

    ...

endmodule
```

STARC_VLOG 1.1.1.7

RULE NAME	**Add an identifying symbol at the end of the name so the polarity of negative logic signals is clearly identified (”_X” , ”_N”, for example).**	
MESSAGE-1	**Declaration of control signal “{SignalName}” intends it to be used as {LogicType} logic signal, but both polarities are used.**	
	DETAIL	*Signal is used as {LogicType} logic.*
MESSAGE-2	**Active low control signal “{SignalName}” violates basic naming convention. Add an identifying symbol (”_X” or ”_N”) at the end of the name so the polarity of negative logic signals is clearly identified.**	
MESSAGE-3	**Active high control signal “{SignalName}” violates basic naming convention. Identifying symbol (”_X” or ”_N”) should be added only at the end of the negative logic signal name to identify its polarity clearly.**	
PROBLEM DESCRIPTION	To make design easier to understand add an identifying symbol (”_X”, ”_N”, for example) at the end of negative logic signals (suffix). If an identifier is added at the beginning of signal name (prefix), like ”X_”, it becomes difficult to distinguish it from identifiers of hierarchies or identifiers of delimiting function. Therefore, to identify negative logic signals, delimit with ”_”(underscore) at the end followed by identifying characters such as ’X’ or ’N’. If adding an identifier of clock system at the end, an identifier of negative logic signal can be used just before it, for example: “SIG_X_CLK1”.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker verifies control signals: – actual polarity (AP) of each control signal X is detected: – for clock signals AP is detected by sensitivity list; – for other controls (asynchronous/synchronous reset, set and enable) AP is detected by conditional branch; – detect the intended polarity (IP) from the control signal declaration (by the regular expression “`\w+_[NX](\w+)?`“) – if AP is not the same in all usage cases: – if parameter CHECK_MIXED_EDGE_SIGNALS == “1” – if AP != IP => violation (message-1 + detail per each improper usage) – else if AP is the same for all usage cases: – if IP is positive and AP != IP => violation (message-2) – if IP is negative and AP != IP => violation (message-3) Note: parameter CHECK_MIXED_EDGE_SIGNALS defines whether to check signals both polarities of which are used: "1" means yes, "0" means no (default value is “0”).	

EXAMPLE-1: [1] clock signal – AP = positive, IP = positive => no violation;

[2] asynchronous reset signal – AP(detected by conditional branch) = negative, IP = positive => violation (message-2);

[3] synchronous set signal – AP(detected by conditional branch) = negative, IP = positive => violation (message-3).

```
module ff (clk,rst,set_n,d,q);

    input clk
    input rst;
    input set_n;
    input d;
    output reg q;
```

Active high control signal “set_n” violates basic naming convention. Identifying symbol (”_X” or ”_N”) should be added only at the end of the negative logic signal name to identify its polarity clearly.

Active low control signal “rst” violates basic naming convention. Add an identifying symbol (”_X” or ”_N”) at the end of the name so the polarity of negative logic signals is clearly identified.

```
    always @( posedge clk, negedge rst)
        if ( !rst )
            q <= 1'b0;
        else if ( set_n )
            q <= 1'b1;
        else
            q <= d;
endmodule
```

EXAMPLE-2: [1] clock signal – IP = positive;

[2] the signal is used in different processes with different polarity;

[3] parameter CHECK_MIXED_EDGE_SIGNALS value is set to “1” => violation(message-1).

```
module ff (clk,d1,d2,q1,q2);

     input clk;
     input d1,d2;
     output reg q1,q2;

    always @( posedge clk )
        q1 <= d1;

    always @( * )
        if ( !clk )
            q2 = d2;
endmodule
```

Declaration of control signal “clk” intends it to be used as positive logic signal, but both polarities are used.

Signal is used as positive logic.

Signal is used as negative logic.

STARC_VLOG 1.1.1.8

RULE NAME	**Instance names should basically be the module names. Instance names that are used more than once should be ”<module name>_<quantity>”.**	
MESSAGE-1	**Instance names should be based on the module name or <module_quantity> if multiple instances exist.**	
	DETAIL-1	*Instance name “{InstanceName}” does not correspond to module name “{ModuleName}”.*
	DETAIL-2	*Module “{ModuleName}” is instantiated multiple times but instantiation names do not correspond to <module_quantity> template.*
MESSAGE-2	**Instance names within 'generate' statement should be based on the module name or <module_quantity> if multiple instances exist.**	
	DETAIL-1	*Instance name “{InstanceName}” does not correspond to module name “{ModuleName}”.*
	DETAIL-2	*Module “{ModuleName}” is instantiated multiple times but instantiation names do not correspond to <module_quantity> template.*
PROBLEM DESCRIPTION	Instance names should be based on the module name. If multiple instances exist name them in such a way: ”<module name>_<quantity>”. Giving meaningful names to instances makes a design hierarchy description clearly structured and facilitate understanding the project. If an instance name that does not conform to this naming convention is used, explicitly describe it in the document.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans modules and checks module instantiation statements: – for modules that are instantiated only once: – if instantiation name differs from the following format: ModuleName => violation (detail-1); – for modules that are instantiated more than once: – if instantiation name differs from the following format: ModuleName_IntegerNumber => violation (detail-2). Note-1: main message is defined by the context of instantiations: – within global scope of currently scanned module => message-1; – within 'generate' block => message-2. Note-2: instantiation names of built-in primitives are not considered. Note-3: UDP and arrays of instances are checked under common conditions (array is checked as a single instance).	

EXAMPLE-1: [1] module is instantiated only once;
[2] instantiation name differs from the module name => violation (message-1 + detail-1).

```
module top;
    ...
    latch latch_inst ( ... );
    ...
endmodule
```

Instance names should be based on the module name or <module_quantity> if multiple instances exist.

Instance name “latch_inst” does not correspond to module name “latch”.

EXAMPLE-2: [1] module is instantiated two times;
[2] instantiation names differ from the format: ModuleName_IntegerNumber => violation (message-1 + detail-2).

```
module top;

    ...

    ff ff_inst1 ( ... );
    ff ff_inst2 ( ... );

    ...

endmodule
```

Instance names should be based on the module name or <module_quantity> if multiple instances exist.

Module “ff” is instantiated multiple times but instantiation names do not correspond to <module_quantity> template.

EXAMPLE-3: [1] module is instantiated within 'generate' statement;
[2] instantiation names differ from the module name => violation (message-2 + detail-1).

```
generate for ( i = 0; i < 8; i = i + 1 )

    begin: gen_ff

        ff ff_inst ( ... );

    end
```

Instance names w ithin 'generate' statement should be based on the module name or <module_quantity> if multiple instances exist.

Instance name “ff” does not correspond to module name “latch”.

STARC_VLOG 1.1.1.9

RULE NAME	**At the top level, module names and port names should consist of 16 or fewer characters and should not be distinguished by upper or lower case alphabet letters**	
MESSAGE-1	**{ObjectClass} name "{ObjectName}" has {NameLength} characters. Length of top level module and port names should be 16 or fewer characters to use design as the IP core.**	
MESSAGE-2	**Name "{ObjectName}" of top level module violates basic naming convention. It should consist only of alphabetical letters and numbers to use design as the IP core.**	
MESSAGE-3	**{ObjectClass} name "{ObjectName}" violates basic naming convention. Top level module and port names should use only upper case or lower case to support systems that are not case sensitive.**	
MESSAGE-4	**Name "{ObjectName}" of top level port violates basic naming convention. It should consist only of alphabetical letters, numbers and underscores to use design as the IP core.**	
MESSAGE-5	**Top level port name "{ObjectName}" violates basic naming convention. Underscores should not be used consecutively and at the beginning/ending of the port name to use design as the IP core.**	
PROBLEM DESCRIPTION	In order to use design as an IP core, set of following rules should be considered when describing top-level unit: – module name should use only alphabetical letters or numbers and be 16 or fewer characters in length – port names should use only alphabetical letters, numbers and underscores (it is prohibited to use underscores at the beginning/ending of the name or to use them consecutively) – to support case-insensitive systems, module and port names should consist of upper/lower case only (cases should not be mixed)	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker verifies each module specified as top-level for elaboration. Object names are checked in the following order: – module name: – if length is greater than 16 => violation (message-1) – else if non-alphabetical letters or underscores are used => violation (message-2) – else if letter cases are mixed => violation (message-3) – each port name: – if length is greater than 16 => violation (message-1) – else if non-alphabetical letters, numbers or underscores are used => violation (message-4) – else if underscores specified at the beginning/end or used consecutively => violation (message-5) – else if letter cases are mixed => violation (message-3)	

EXAMPLE-1: [1] module name contains a digit => violation (message-2);
[2] port name contains consecutive underscores => violation (message-5);

```
module top3( SAMPLE__CLK, ...  );
     ...
endmodule
```

Name "top3" of top level module violates basic naming convention. It should consist only of alphabetical letters and numbers to use design as the IP core.

Top level port name "SAMPLE__CLK" violates basic naming convention. Underscores should not be used consecutively and at the beginning/ending of the port name to use design as the IP core.

EXAMPLE-2: [1] module name contains letters having different cases => violation (message-3);

[2] port name has more than 16 characters in length => violation (message-1);

[3] note: port name also contains underscore at the beginning, but no violation is displayed for it (checker consecutively verifies constraints and if the previous one is broken, the next one is not checked);

```
module TopLevel( _serial_argument_input );
     ...
endmodule
```

Module name "TopLevel" violates basic naming convention. Top level module and port names should use only upper case or lower case to support systems that are not case sensitive.

Port name "_serial_argument_input" has 22 characters. Length of top level module and port names should be 16 or fewer characters to use design as the IP core.

STARC_VLOG 1.1.1.10

RULE NAME	**Do not use the same instance name or cell name as the ASIC library being used**	
MESSAGE-1	**{ObjectClass} name "{ObjectName}" violates basic naming convention. Name corresponds to library name "{LibraryName}".**	
MESSAGE-2	**{ObjectClass} name "{ObjectName}" violates basic naming convention. Name corresponds to attached library "{LibraryName}".**	
PROBLEM DESCRIPTION	Naming conventions are specified by the used semiconductor technology flow. These limitations can be quite different depending on the vendor, but there are two common requirements: names of instances and cells should not correspond to the name of ASIC library being used. Also, it is required to avoid names that are the same as libraries used by the simulator.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies the name of each instance or cell: – if it corresponds to the name of the ASIC library being used => violation (message-1) – list of ASIC libraries is configurable through configuration file parameter LIB_LIST ("GTECH", "LVT", "HVT" – by default) – if it corresponds to the name of attached library => violation (message-2) – attached libraries are specified with the "l" switch (for example, "vlog test.v -l STD -alint" invokes checking of the source file "test.v" with attached library "STD") Note: cell is a module marked with a pair of directives `celldefine - `endcelldefine Note: letter cases are ignored	

EXAMPLE-1: [1] the name of the instance corresponds to the ASIC library "GTECH" => violation (message-1)

Instance name "Gtech" violates basic naming convention. Name corresponds to library name "GTECH".

```
FPU_UNIT Gtech( .SEL( SEL ), ... );
```

EXAMPLE-2: [1] name of the cell corresponds to the attached library "STD" ("-l STD" is specified for compilation) => violation (message-2)

Cell name "std" violates basic naming convention. Name corresponds to library name "STD".

```
`celldefine

    module std( ... );
        ...
    endmodule

`endcelldefine
```

1.1.2 Naming conventions of circuit and port names should be considered by the hierarchy

STARC_VLOG 1.1.2.1

RULE NAME	**Module names and instance names should be between 2 and 32 characters in length**	
MESSAGE-1	**{ObjectClass} name "{ObjectName}" has {NameLength} characters. Module names and instance names should be between {MIN_LENGTH} and {MAX_LENGTH} characters in length.**	
PROBLEM DESCRIPTION	Logic synthesis tools may change module or instance names if they exceed 32 characters. Also there are ASIC vendors limitations which states allowable length between 2 and 32 characters for such identifiers. According to the constraints, module and instance names should be between 2 and 32 characters in length.	
	LEVEL	RULE
MESSAGE-2	**{ObjectClass} name "{ObjectName}" has {NameLength} characters. A length of {MAX_LENGTH_RECOMMEND} or fewer characters is recommended.**	
PROBLEM DESCRIPTION	Long instance names decrease readability when objects (signals, functions etc.) from lower levels of hierarchy are used. Instance names of 16 or fewer characters is recommended.	
	LEVEL	RECOMMENDED 2
MESSAGE-3	**{ObjectClass} name "{ObjectName}" has {NameLength} characters. Instance names with hierarchy should be less than {MAX_LENGTH_HIER} characters.**	
PROBLEM DESCRIPTION	A hierarchy may be flattened by some tool which is used at later stages. It leads to difficulties if instance names are long and hierarchy is deep. Therefore, it is recommended that an instance names including module hierarchy should be 128 or fewer characters.	
	LEVEL	RECOMMENDED 3
CHECKER BEHAVIOR	1) Checker verifies all module and instance names – if length is less than MIN_LENGTH or greater than MAX_LENGTH characters => violation (message-1). 1) Checker verifies all instance names – if length is greater than MAX_LENGTH_RECOMMEND characters => violation (message-2). 3) Checker verifies all hierarchical instance names (identifiers connected via hierarchy separator) – if accumulated hierarchical identifier length is greater than MAX_LENGTH_HIER characters => violation (message-3). Note-1: values of parameters MIN_LENGTH, MAX_LENGTH, MAX_LENGTH_RECOMMEND, MAX_LENGTH_HIER are defined in configuration file. Note-2: see 1.1.1.2 for {ObjectClass} description.	

EXAMPLE-1: [1] module name is only one character in length (shorter than MIN_LENGTH) => violation (message-1);

[2] instance name longer then recommended length => violation (message-2)

```
module t;
    ...
    submod too_long_instance_name(...);

    ...
endmodule
```

Module name "t" has 1 characters. Module names and instance names should be betw een 2 and 32 characters in length.

Instance name "too_long_instance_name" has 22. characters. A length of 16 or few er characters is recommended.

EXAMPLE-2: [1] instance name is longer than MAX_LENGTH => violation (message-1);
[2] note: only one message is produced.

```
module top;

    ...

    submod very_long_and_unreadable_instance_name(...);

    ...

endmodule
```

Instance name “very_long_and_unreadable_instance_name” has 38 characters. Module names and instance names should be betw een 2 and 32 characters in length.

Instance name “very_long_and_unreadable_instance_name” has 38 characters. A length of 16 or few er characters is recommended.

STARC_VLOG 1.1.2.2

RULE NAME	**Output port names and the connected net names should be the same**	
MESSAGE	**There is name inconsistency between port(s) and connected net(s) in the module instantiation statement. Net name of the upper level to which a port name is connected should be the same.**	
	DETAIL	*Net name "{SignalName}" differs from the port name "{PortName}".*
PROBLEM DESCRIPTION	It is best that the net name of the upper level to which an output signal name is connected and the input signal name are the same. The output signal name is used for an input signal name even when the output signal is input to multiple blocks. Such naming conventions makes it easier to trace signal values because output signals are more important for debugging purposes. When using multiple instances of the same module hierarchy identification characters should be added to the net names.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects module instances on each level of project hierarchy: – if module is instantiated two and more times on the same level instantiation statements of the module are not checked – for modules instantiated only once every port name and connected net name are checked – if the names are different => violation Note-1: both ordered and named port connection types are checked Note-2: expressions on port map are ignored	

EXAMPLE-1: [1] module mod1 instantiation statement contains net name which differs from port name (ordered port connection is used, one port is unconnected but it is allowed) => violation;

[2] module mod2 instantiation statement contains net name which differs from port name (named and implicit port connection is used) => violation;

[3] module mod3 instantiation statement contains net name which differs from port name, but the module is instantiated more then once => no violation;

```
module mod1 ( in1, in2, out1, out2);
   ...
endmodule

module mod2 ( a, b ,c );
   ...
endmodule

module mod3 ( x, y );
   ...
endmodule


module top;
    ...
    mod1 inst1 ( in, in2, out1,  );
    mod2 inst2 ( .a(out1), .* );
    mod3 inst3 ( .x(xxx), .y(y) );
    mod3 inst4 ( .* );

endmodule
```

There is name inconsistency betw een port(s) and connected net(s) in the module instantiation statement. Net name of the upper level to w hich a port name is connected should be the same.

Net name "in" differs from the port name "in1".

There is name inconsistency betw een port(s) and connected net(s) in the module instantiation statement. Net name of the upper level to w hich a port name is connected should be the same.

Net name "in" differs from the port name "in1".

1.1.3 Give meaningful names for signals

STARC_VLOG 1.1.3.1

RULE NAME	**Signal names, port names, parameter names, `define names and function names should be between 2 and 40 characters in length**	
MESSAGE-1	**{ObjectClass} name "{ObjectName}" has {NameLength} characters. It should be between {MIN_LENGTH} and {MAX_LENGTH} characters in length.**	
PROBLEM DESCRIPTION	The number of characters in signal names, port names, parameter names and function names is recommended to be between 2 and 40 to make project code easy to read and understand.	
	LEVEL	MANDATORY
MESSAGE-2	**{ObjectClass} name "{ObjectName}" has {NameLength} characters. The length of {MAX_LENGTH_RECOMMEND} or fewer characters is recommended.**	
PROBLEM DESCRIPTION	A tool used at a later stage might convert a signal name which is too long. Although a long signal name is more understandable than a short one, an overly long name makes it unreadable. Therefore, the recommended basis for signal name is up to 24 characters in length.	
	LEVEL	RECOMMEND 2
CHECKER BEHAVIOR	1) Checker verifies signal, port, parameter, function names – if length is less than MIN_LENGTH or greater than MAX_LENGTH characters => violation (message-1) 2) Checker verifies signal and port names – if length is greater than MAX_LENGTH_RECOMMEND characters => violation (message-2) Note-1: values of parameters MIN_LENGTH, MAX_LENGTH, MAX_LENGTH_RECOMMEND are defined in configuration file. Note-2: for signal or port name only one message (message-1 or message-2) is produced. Note-3: see 1.1.1.2 for {ObjectClass} description	

EXAMPLE-1: [1] parameter name is shorter than MIN_LENGTH => violation (message-1).

```
parameter p;
```

Parameter name "p" has 1 characters. It should be between 2 and 40 characters in length.

EXAMPLE-2: [1] port name is longer than recommended => violation (message-2);
[2] signal name is longer than MAX_LENGTH => violation (message 1).

```
module top ( difficult_to_read_such_port_name);
    ...
    reg very_difficult_to_read_so_long_signal_name;
    ...
endmodule
```

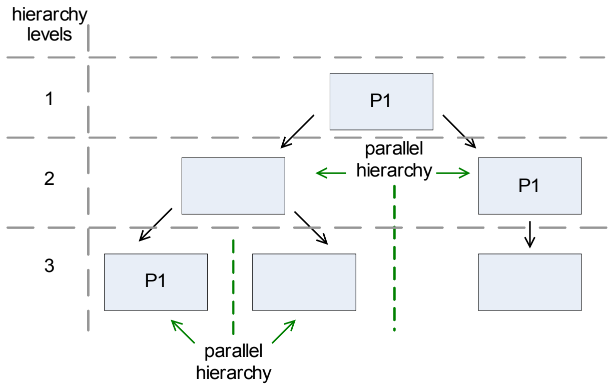
Port name "difficult_to_read_such_port_name" has 32 characters. The length of 24 or fewer characters is recommended.

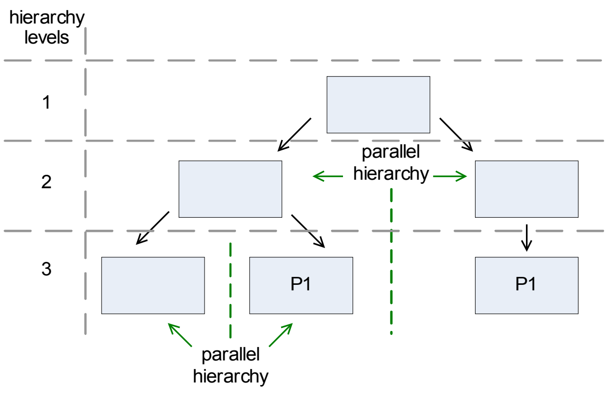
Signal name "very_difficult_to_read_so_long_signal_name" has 42 characters. It should be between 2 and 40 characters in length.

Port name "very_difficult_to_read_so_long_signal_name" has 42 characters. The length of 24 or fewer characters is recommended.

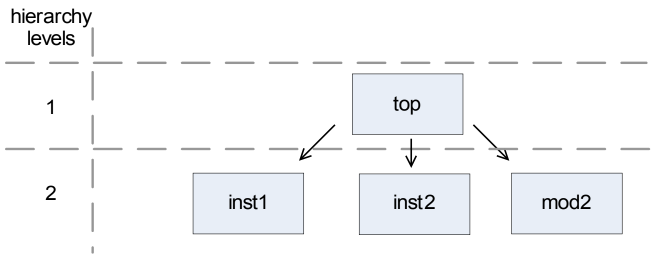
1.1.4 Naming conventions of *include* file, *parameter* and *`define* (different from VHDL)

STARC_VLOG 1.1.4.1

RULE NAME	**Do not use parameters with the same name for different modules**	
MESSAGE	**Module “{ModuleName}” has “{ParamName}” parameter, which name is equal to parameter(s) declared in module(s) from other hierarchy(ies). Current description may lead to confusion with duplicated parameters, because it is recommended to put data to be used as parameters into `include files to make it easy to change parameter values.**	
	DETAIL-1	*Module “{ModuleName}” has parameter with the same name “{ParamName}”.*
	DETAIL-2	*Instance “{InstanceName}” of module “{ModuleName}”.*
PROBLEM DESCRIPTION	Whenever possible, data to be used as parameters should be put into include files, thus making it easy to change parameter values. But care should be taken with parameter names to avoid confusing when including file with parameters declarations. Distinguish parameters used for the overall design from parameters used only under particular hierarchies, and place each one into a separate include file.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker verifies modules from the library: – if there are modules in the design that contain parameter declarations with equal names: – for each hierarchy level (hierarchy begins from top-level – in other words, multiple hierarchies are possible in case of multiple top-levels in the design) and parameter name: – if parameter with the same name exists in any instance (of another module) that is neither a child, nor a parent of the analyzed instance => violation **example #1:** parameters with same name in modules from different levels of parallel hierarchies, bur upper level module also has parameter with the name => no violation  hierarchy levels 1 2 3 P1 parallel hierarchy P1 P1 parallel hierarchy – **example #2:** parameters with same name in modules from 2nd levels in one of parallel hierarchies and from both levels of other, there are no common upper level module which also has parameter with the name => violation	

RULE NAME	Do not use parameters with the same name for different modules
	 hierarchy levels 1 2 3 parallel hierarchy P1 P1 parallel hierarchy Note: displaying of detail message #2 is controlled by "DISPLAY_INSTANCES" rule parameter. "0" means "do not display", "1" means "display" (default value is "0").

EXAMPLE-1: [1] consider structure described at the picture below;
[2] parameters with same names declared within two modules;
[3] the modules instantiated at the same hierarchy level = > violation.
Note: parameter DISPLAY_INSTANCES value is set to "1" => detail messages 2 are displayed.



```
//top module declaration
module top (...);

    ...

    mod1 inst1 (...);
    mod1 inst2 (...);
    mod2 mod2 (...);

endmodule

//sub-modules declaration
module mod1(...);

    parameter p1 = 32;

    ...

endmodule

module mod2(...);

    parameter p1 = 8;

    ...

endmodule
```

Instance "top.inst1" of module "mod1".

Instance "top.inst2" of module "mod1".

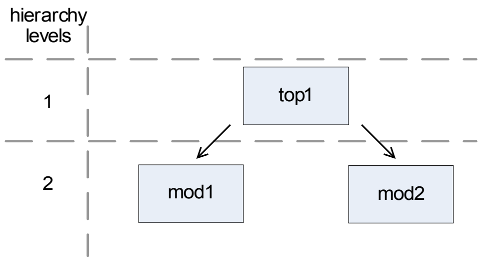
Instance "top.mod2" of module "mod2".

Instance "top". Module "mod1" has "p1" parameter, w hich name is equal to parameter(s) declared in module(s) from other hierarchy(ies). Current description may lead to confusion w ith duplicated parameters, because it is recommended to put data to be used as parameters into `include files to make it easy to change parameter values.

Module "mod2" has parameter w ith the same name "p1".

EXAMPLE-2: [1] consider structure described at the picture below;

[2] parameters with same names declared within both top-level module and instantiated modules => no violation.

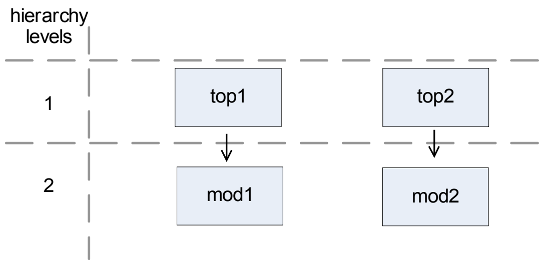


```
//first top module declaration
module top1 (...);

     parameter p1 = 2;

     ...

     mod1 mod1 (...);
     mod2 mod2 (...);

endmodule

//sub-modules declaration
module mod1(...);

     parameter p1 = 32;

     ...

endmodule

module mod2(...);

     parameter p1 = 8;

     ...

endmodule
```

EXAMPLE-3: [1] consider structure described at the picture below;

[2] parameters with same names declared within two modules;

[3] the modules instantiated at the same hierarchy level, but to different top-level modules (belongs to different hierarchies scanned separately) => no violation.



```
//first top module declaration
module top1 (...);

     ...

     mod1 mod1 (...);

endmodule
```

```
//second top module declaration
module top2 (...);

    ...

    mod2 mod2 (...);

endmodule

//sub-modules declaration
module mod1(...);

    parameter p1 = 32;

    ...

endmodule

module mod2(...);

    parameter p1 = 8;

    ...

endmodule
```

STARC_VLOG 1.1.4.2

RULE NAME	**Use `define definitions declared in the same module only (Verilog only)**	
MESSAGE	**Macro "{MacroID}" is used in module "{ModuleName}" {UsageCount} time(s). Use `define definitions declared only in the current module to enable separate generation of each module by the logic synthesis tool.**	
	DETAIL	*Global macro used*
PROBLEM DESCRIPTION	It is recommended not to use macro defined with `define directive on the global position or defined in other modules. When globally defined macro is used, generation of logic circuit for each module separately becomes impossible.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker verifies each 'text_macro_id' statement: – if it is not defined in the current module => violation Note-1: to be defined in the current module means to be defined after the 'module' keyword Note-2: 'text_macro_id' statement refers to the definition created with the `define directive	

EXAMPLE-1: [1] module accesses macro "LENGTH" is not defined in the current module => violation

```
`define LENGTH 8

module tb;
    wire[ `LENGTH - 1: 0 ]
    ...
endmodule
```

Macro "LENGTH" is used in module "tb" 1 time(s). Use `define definitions declared only in the current module to enable separate generation of each module by the logic synthesis tool.

Global macro used

EXAMPLE-2: [1] module accesses macro "COMB" defined outside the module => violation (message-2, because it is defined in the included file "defines.h")

```
`include "defines.h"

module dev8( ... );
    `ifdef COMB
        ...
        `ifdef COMB
    `endif
    ...
endmodule
```

Macro "COMB" is used in module "dev8" 2 time(s). Use `define definitions declared only in the current module to enable separate generation of each module by the logic synthesis tool.

Global macro used

EXAMPLE-3: [1] module accesses macro "COMB" defined in the included file "defines.h" => no violation ("defines.h" is included locally)

```
module dev8( ... );
    `include "defines.h"
    ...
    `ifdef COMB
        ...
        `ifdef COMB
    `endif
    ...
endmodule
```

EXAMPLE-4: [1] module accesses macro "COMB" that is not defined in current file (neither globally nor included) => no violation

```
module dev8( ... );
    `ifdef COMB
        ...
    `endif
endmodule
```

STARC_VLOG 1.1.4.3

RULE NAME	**Fixed values should not be connected directly to output ports**	
MESSAGE-1	**Constant is directly connected to output port "{ObjectName}". It may cause problems during logic equivalence checks.**	
PROBLEM DESCRIPTION	Fixed values should not be connected directly to output ports. After synthesis optimization is applied from the upper levels of the hierarchy, ports that are directly connected to fixed values can become unconnected. Such situation causes problems during the logic equivalence checks.	
	LEVEL	RECOMMENDATION 1
MESSAGE-2	**Constant is assigned to {PortCount} port(s) of lower layer module "{ModuleName}". Some redundant logic may remain after applying synthesis optimization.**	
	DETAIL-1	*Constant is assigned to instance port*
PROBLEM DESCRIPTION	When fixed values are connected to ports of lower hierarchy level, another problem tends to occur there. Some redundant logic might remain after applying of the synthesis optimization => gate count will be increased.	
	LEVEL	REFERENCE
CHECKER BEHAVIOR	1) Checker scans output/inout port drivers (continuous and procedural assignment statements) in synthesizable context: – if constant/parameter is assigned => violation (message-1) 2) Checker verifies list of port connections in the module instantiation statements: – if constant/parameter is connected to the input port => violation (message-2)	

EXAMPLE-1: [1] 'assign' statement connects constant to output port => violation (message-1);
[2] note: constant is member of concatenation;

```
output [1:0] TRAN_A;
...
reg SAMPLE;
...
assign TRAN_A = { 1'b0, SAMPLE };
```

Constant is directly connected to output port "TRAN_A[1]". It may cause problems during logic equivalence checks.

EXAMPLE-2: [1] parameter is connected to the input port of the component => violation (message-2);

```
module circ( input [1:0] SAMPLE, output AL );
    ...
endmodule
...
module top;
    parameter [1:0] STOP_B;
    ...

    circ CIRC_8t( .SAMPLE( STOP_B ), .AL( ... ) )
    ...
endmodule
```

Constant is assigned to 1 port(s) of low er layer module "circ". Some redundant logic may remain after applying synthesis optimization.

Constant is assigned to instance port

STARC_VLOG 1.1.4.4

RULE NAME	**Clarify <value>'b, 'h, 'd, 'o specification for parameters (Verilog only)**	
MESSAGE-1	**Parameter "{ObjectName}" is initialized with value "{ParamValue}" without base specifier. Describe 'b, 'h, 'd or 'o clearly when initializing a parameter.**	
MESSAGE-2	**Parameter "{ObjectName}" is initialized with expression which contains {IllegalConstCount} constant(s) without base specifier. Describe 'b, 'h, 'd or 'o clearly when using constants for parameter initialization.**	
	DETAIL-1	*Base is not specified for constant "{ConstValue}"*
MESSAGE-3	**Parameter is initialized with value "{ParamValue}" without base specifier. Describe 'b, 'h, 'd or 'o clearly when initializing a parameter.**	
MESSAGE-4	**Parameter is initialized with expression which contains {IllegalConstCount} constant(s) without base specifier. Describe 'b, 'h, 'd or 'o clearly when using constants for parameter initialization.**	
	DETAIL-1	*Base is not specified for constant "{ConstValue}"*
PROBLEM DESCRIPTION	It is strongly recommended to describe the base specification clearly for parameter numeric values greater than 8 ('b, 'h', 'o, 'd). If base is not specified, there is a possibility to introduce a mistake (for example, 11 is not 'h11). Moreover, descriptions without base specifiers are harder to maintain.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker verifies statements containing parameter initialization with numeric value: – parameter / localparam – component instantiation – defparam Numeric value can be assigned with: – single constant (violation when base is not specified and value is greater than 8): – message-1 (when parameter name is known) – message-3 (when parameter name is unknown) – expression (operations with another constants or parameters) => each numeric constant is checked (violation when base is not specified and value is greater than 8): – message-2 (when parameter name is known) – message-4 (when parameter name is unknown)	

EXAMPLE-1: [1] numeric value (greater than 8) without base specifier initialize parameter => violation (message-1, because there is no expression, and parameter name is known)

```
parameter PARAM = 32;
```

Parameter "PARAM" is initialized w ith value "32" w ithout base specifier. Describe 'b, 'h, 'd or 'o clearly w hen initializing a parameter.

EXAMPLE-2: [1] instantiated component "generic_divider" is not compiled to the working library yet => port names are unknown;

[2] parameter is overridden at instance "CLK_DIV_10" with a single constant "10" without base specifier => violation (message-3, due to unknown parameter name);

```
generic_divider #(10) CLK_DIV_10( ... );
```

Parameter is initialized w ith value "32" w ithout base specifier. Describe 'b, 'h, 'd or 'o clearly w hen initializing a parameter.

EXAMPLE-3: [1] instantiated component "generic_divider" is not compiled to the working library yet => port names are unknown;

[2] parameter is overridden at instance "CLK_DIV_20" with an expression containing another parameter and numeric value "20" without base specifier => violation (message-4, due to expression and known parameter name, since naming connection is used);

[3] note: there is no warning for declaration of parameter "BASE" – it is initialized with value without base specifier – but it is not greater than 8;

Parameter is initialized w ith expression w hich contains 1 constant(s) w ithout base specifier. Describe 'b, 'h, 'd or 'o clearly w hen using constants for parameter initialization.

```
parameter BASE = 8;
...
generic_divider #( .FACTOR( BASE + 20 ) ) CLK_DIV_20( ... );
```

Base is not specified for constant "20"

STARC_VLOG 1.1.4.5

RULE NAME	**Specify bit width if it is greater than 32 bits**	
MESSAGE-1	**Parameter "{ObjectName}" is initialized with value "{ParamValue}" without width specifier. Specify bit width directly when declaring parameters greater than {CHECK_BIT_WIDTH_GREATER_THAN} bits.**	
MESSAGE-2	**Parameter "{ObjectName}" is initialized with expression which contains {IllegalConstCount} constant(s) without width specifier. Specify bit width directly when declaring parameters greater than {CHECK_BIT_WIDTH_GREATER_THAN} bits.**	
	DETAIL	*Bit width is not specified for constant "{ConstValue}".*
MESSAGE-3	**Parameter is initialized with value "{ParamValue}" without width specifier. Specify bit width directly when declaring parameters greater than {CHECK_BIT_WIDTH_GREATER_THAN} bits.**	
MESSAGE-4	**Parameter is initialized with expression which contains {IllegalConstCount} constant(s) without width specifier. Specify bit width directly when declaring parameters greater than {CHECK_BIT_WIDTH_GREATER_THAN} bits.**	
	DETAIL	*Bit width is not specified for constant "{ConstValue}".*
MESSAGE-5	**Decimal constant greater than 32 bits is truncated by compiler. Specify bit width directly when declaring parameters greater than {CHECK_BIT_WIDTH_GREATER_THAN} bits.**	
PROBLEM DESCRIPTION	Parameters with no width specified have bit width 32. When parameter exceeds 32-bit capacity and bit width is not specified parameter may be trimmed by the compiler. So it is recommended to specify bit width if it is greater than 32.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects all parameter initialization expressions : – parameter or localparam initialization – parameter redefinition in module instantiation statement – parameter redefinition in defparam statement Checker verifies constants in detected expressions: – if constant does not have width specifier <integer_literal>{ 'b |'h | 'd | 'o } and constant width is greater than rule parameter CHECK_BIT_WIDTH_GREATER_THAN => violation – if the expression contains only one constant => message-1 in case of XREF => message-3 – if expression contains operation(s) with constants => message-2 (detail message per each constant) in case of XREF => message-4 Note-1: value of parameter CHECK_BIT_WIDTH_GREATER_THAN is defined in a configuration file. Note-2: decimals longer than 32 bits are trimmed to 32 by VCP, and so cannot be checked with default setting => there is a special warning message-5	

EXAMPLE-1: [1] bit width is 44 and is not specified in parameter declaration statement => violation (message-1);

```
parameter BW = 'h50077766677;
```

Parameter "BW" is initialized w ith value "'h50077766677" w ithout w idth specifier. Specify bit w idth directly w hen declaring parameters greater than 32 bits.

EXAMPLE-2: [1] bit width of decimal parameter is 39 => violation (message-5);

```
parameter BW = 'd500777666777;
```

Decimal constant greater than 32 bits is truncated by compiler. Specify bit w idth directly w hen declaring parameters greater than 32 bits.

EXAMPLE-2: [1] bit width of decimal parameter is 32, but value of rule parameter CHECK_BIT_WIDTH_GREATER_THAN is changed by configuration file to 30 => violation (message-1);

```
parameter BW = 'd4278255360;
```

Parameter "BW" is initialized w ith value "4278255360" w ithout w idth specifier. Specify bit w idth directly w hen declaring parameters greater than 30 bits.

1.1.5 Naming should consider clock systems

STARC_VLOG 1.1.5.1

RULE NAME	**Basically, use “CLK” or “CK” for clock signal names, “RST_X” or “RESET_X” for reset signal names and “EN” for enable signal names. Add identifiers to the end of these basic names.**
MESSAGE-1	**{SignalType} signal “{SignalName}” violates basic naming convention. Use “CLK” or “CK” for clock signal names, “RST” or “RESET” for reset signal names and “EN” for enable signal names. Add up to {IDENT_CHAR_NUMBER} identifying characters to the end of these basic names if multiple {SignalTypeLower} signals exist.**
MESSAGE-2	**{SignalType} signal “{SignalName}” matches forbidden pattern “{RegExp}”.**
MESSAGE-3	**{SignalType} signal “{SignalName}” does not match legal pattern “{RegExp}”.**
PROBLEM DESCRIPTION	In order to improve the readability of a description, a signal name based on the clock system can be used. First, decide the basic signal name for the clock signal, reset signal and enable signal. Then add an identifier to the end of the basic signal name when more than one signals of the same kind exist. It is recommended to use basic signal names of “CLK” or “CK” for a clock signal, “RST_X” or “RESET_X” for a reset signal and “EN” for an enable signal. For example, if multiple clocks exist, add one to three characters to the end of “CLK” or “CK” like “CLK2”, “CLKD” or “CLK_CPU”, etc. Names which suggest a clock system can be given by adding the name of the clock signal source, to the end of the signal name (ex.”_CLKM"). It would be overly verbose to add clock identification to signals for the entire design. However, knowing which clock each signal is dependent upon is important in systems that employ two-phase or three-phase latch based designs or use asynchronous transfer. A clock name should be added when designing such circuits. LEVEL: RECOMMENDATION 3
CHECKER BEHAVIOR	Checker detects clock, reset and enable signals in each process: – identifiers of each detected signal should match the pattern: – clock – `CL?K(_[XN])?(_?[a-zA-Z0-9]{1,IDENT_CHAR_NUMBER})?` – reset – `(RST | RESET)(_[XN])?(_?[a-zA-Z0-9]{1,IDENT_CHAR_NUMBER})?` – enable – `EN (_[XN])?(_?[a-zA-Z0-9]{1,IDENT_CHAR_NUMBER})?` – if identifier does not match corresponding pattern => violation (message-1) Note-1: see 1.1.1.2 for details about messages 2,3 and custom regular expressions. Note-2: parameter IDENT_CHAR_NUMBER configures number of identifying characters allowed to be specified at the end of signal name (default value is 3).

EXAMPLE-1: [1] reset signal does not match the pattern (basic name is inappropriate) => violation (message-1);

[2] enable signal does not match the pattern (number of identifying characters specified after basic name exceeds allowed number) => violation (message-1).

Note: parameter IDENT_CHAR_NUMBER value is default (IDENT_CHAR_NUMBER = 3).

```
always @( posedge clk123 or negedge rs_x_123 )

    if ( rs_clk123 )
        q <= 1'b0;
```

Reset signal “rs_x_123” violates basic naming convention. Use “CLK” or “CK” for clock signal names, “RST” or “RESET” for reset signal names and “EN” for enable signal names. Add up to 3 identifying characters to the end of these basic names if multiple reset signals exist.

```
else if ( enable )
    q <= d;
```

Enable signal “enable” violates basic naming convention. Use “CLK” or “CK” for clock signal names, “RST” or “RESET” for reset signal names and “EN” for enable signal names. Add up to 3 identifying characters to the end of these basic names if multiple enable signals exist.

1.2 Synchronous design

1.2.1 Clock synchronous design

STARC_VLOG 1.2.1.1

RULE NAME	**Designs should use a single clock/single edge as much as possible**	
MESSAGE-1	**Module "{ModuleName}" uses {CLKCount} different clock lines. Designs should use a single clock as much as possible.**	
	DETAIL-1	*Process infers FF(s) with clock signal "{CLKName}" from {FFCount} signal(s)*
MESSAGE-2	**Module "{ModuleName}" uses {CLKCount} different clock lines. Currently allowed clock domains = {CLOCK_DOMAINS_ALLOWED}.**	
	DETAIL-1	*Process infers FF(s) with clock signal "{CLKName}" from {FFCount} signal(s)*
PROBLEM DESCRIPTION	When dealing with really large designs, the circuit operation speed is analyzed using static timing analysis tools instead of logic simulation. If the clock system is complex, such analysis becomes very difficult: system is represented as a few smaller systems operating with a single clock and a single edge. So it is recommended not to use multiple clock lines. In cases, when multiple clocks are necessary, their count should be reduced as much as possible.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker collects clock signals (by extracting them from a flip-flops inferred in the current module): – if not all flip-flops are using the same clock signal => violation: – message-1, if parameter CLOCK_DOMAINS_ALLOWED is equal to 1 (default) – message-2, if parameter CLOCK_DOMAINS_ALLOWED is greater than 1 Note: parameter CLOCK_DOMAINS_ALLOWED can be set up for designs where multiple clocks are required Note: clock edges are checked in rule 1.4.3.6	

EXAMPLE-1: [1] two 'always' processes infer flip-flops having different clock signals => violation (message-1)

```
module trn( ... );
    ...
    always @( posedge CLK_1 ) begin
        REG_A <= OP_STATE;
    end
    ...
    always @( posedge CLK_2 ) begin
        if( RESET )
            REG_B <= 1'b0;
        else
            REG_B <= TS_STATE;
    end
    ...
endmodule
```

Module "trn" uses 2 different clock lines. Designs should use a single clock as much as possible.

Process infers FF(s) w ith clock signal "CLK_1" from 1 signal(s)

Process infers FF(s) w ith clock signal "CLK_2" from 1 signal(s)

EXAMPLE-2: [1] three 'always' processes infer flip-flops having different clock signals whereas parameter CLOCK_DOMAINS_ALLOWED = 2 => violation (message-2);

[2] note: reducing clocks number here to the two eliminates violation;

```
module trn( ... );
    ...
    always @( posedge CLK_1 ) begin
        REG_A <= OP_STATE;
    end
    ...
    always @( posedge CLK_2 ) begin
        if( RESET )
            REG_B <= 1'b0;
        else
            REG_B <= TS_STATE;
    end
    ...
    always @( negedge CLK_3 ) begin
        REG_C <= PS_STATE_0;
        REG_D <= PS_STATE_1;
    end

endmodule
```

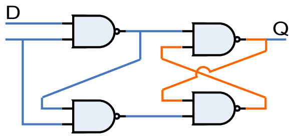
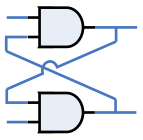
Module "trn" uses 3 different clock lines. Currently allow ed clock domains = 2

Process infers FF(s) w ith clock signal “CLK_1” from 1 signal(s)

Process infers FF(s) w ith clock signal “CLK_2” from 1 signal(s)

Process infers FF(s) w ith clock signal “CLK_3” from 2 signal(s)

STARC_VLOG 1.2.1.2

RULE NAME	**Do not create an RS latch or FF using standard primitive cell such as AND, OR.**	
MESSAGE-1	**Detected combinational description of the circuit that is similar to RS latch. Avoid describing latches with standard primitive cells because timing analysis tools can treat it as feedbacks in combinational circuits.**	
MESSAGE-2	**Detected combinational description of the circuit that is similar to RS latch. Avoid describing latches with standard primitive cells because timing analysis tools can treat it as feedbacks in combinational circuits.**	
	DETAIL-1	*Asynchronous loop propagates through combinational logic.*
	DETAIL-2	*Asynchronous loop propagates through combinational logic line "{LineName}".*
	DETAIL-3	*Asynchronous loop propagates through submodule instance "{InstanceName}" from port "{InputPortName}" to "{OutputPortName}".*
PROBLEM DESCRIPTION	FF or latch can be created by using primitive cells (as D-latch described by primitives shown at the picture), but this could be treated by the timing analysis tool as feedback to a combinational circuit (see also 1.2.1.3). 	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans net list for the presence one of the following feedback types (described at the picture): – if connection and feedback between two gates (either direct or inverter) is present => violation. Note: parameter DETAILED_PROPAGATION_CHAIN configures displaying message (see 1.2.1.3 for information about messages displaying). 	

EXAMPLE-1: [1] D-latch is described using primitives (see picture in "PROBLEM DESCRIPTION" section);
[2] feedback of mentioned type is detected => violation
Note: parameter DETAILED_PROPAGATION_CHAIN is set to 0.

```
module top ( c, d, out );

    input c;
    input d;

    output out;

    wire cd, ccd;
    wire nout;

    nand ( cd, c, d );
    nand ( ccd, c, cd );
    nand ( out, cd, nout );
    nand ( nout, ccd, out );

endmodule
```

Detected combinational description of the circuit that is similar to RS latch. Avoid describing latches w ith standard primitive cells because timing analysis tools can treat it as feedbacks in combinational circuits.

EXAMPLE-2: [1] FF is described using primitives (see picture below);
[2] two feedbacks are detected => two violations
Note: parameter DETAILED_PROPAGATION_CHAIN is set to 1 => message-2 is used.

```
module top ( clk, d, Q );

    input clk;
    input d;

    output Q;

    wire nQ;

    wire cd, ccd;
    wire qcd;
    wire nclk;
    wire rqcd, snqccd;

    nand ( cd, clk, d );
    nand ( ccd, clk, cd );
    nand ( qcd, cd, ~(ccd & qcd) );
    not ( nclk, clk );
    nand ( rqcd, qcd, ~clk );
    nand ( snqccd, ~(ccd & qcd), ~clk );
    nand ( Q, rqcd, nQ );
    nand ( nQ, snqccd, Q );

endmodule
```

Detected combinational description of the circuit that is similar to RS latch. Avoid describing latches w ith standard primitive cells because timing analysis tools can treat it as feedbacks in combinational circuits.

Asynchronous loop propagates through combinational logic line "qcd".

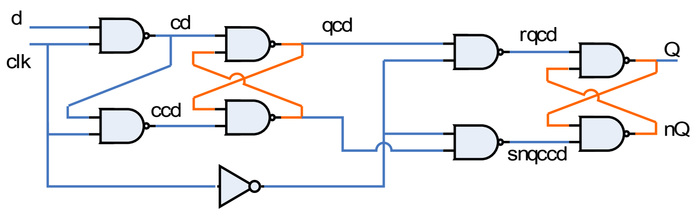
Asynchronous loop propagates through combinational logic.

Asynchronous loop propagates through combinational logic.

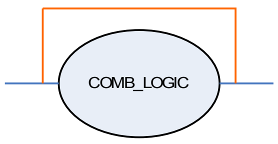
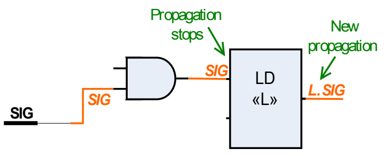
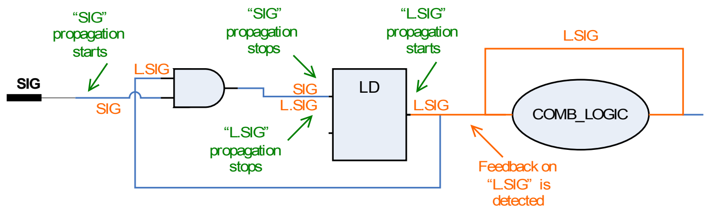
Detected combinational description of the circuit that is similar to RS latch. Avoid describing latches w ith standard primitive cells because timing analysis tools can treat it as feedbacks in combinational circuits.

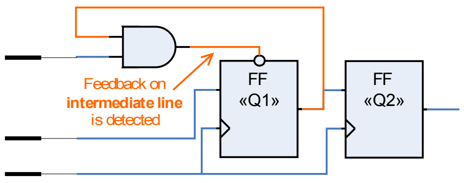
Asynchronous loop propagates through combinational logic line "Q".

Asynchronous loop propagates through combinational logic line "nQ".

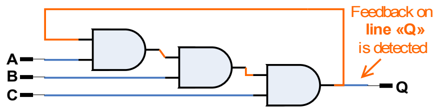


STARC_VLOG 1.2.1.3

RULE NAME	**Do not use feedback in combinational circuits**	
MESSAGE-1	**Combinational feedback is detected on line "{FeedbackLineName}". Do not use feedbacks in combinational circuits to avoid the effect of a feedback loop during the timing analysis**	
	Combinational feedback is detected. Do not use feedbacks in combinational circuits to avoid the effect of a feedback loop during the timing analysis	
	DETAIL-1	*Asynchronous loop propagates through combinational logic*
	DETAIL-2	*Asynchronous loop propagates through combinational logic line "{LineName}"*
	DETAIL-3	*Asynchronous loop propagates through {ObjectType} "{SignalName}" {PortType} input*
	DETAIL-4	*Asynchronous loop propagates through submodule instance "{InstanceName}" from port "{InputPortName}" to "{OutputPortName}"*
PROBLEM DESCRIPTION	Static timing analysis tools are used to analyze the circuit operation speed for large designs. Combinational circuit feedbacks carry into the effect of a feedback loop during timing analysis and should be avoided. 	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans the design hierarchy to detect feedbacks that are propagated through combinational paths: – each signal is propagated through subsequent combinational paths in order to detect the feedback – a combinational path is a path that allows asynchronous propagation of signal (*see Note-1 for feedback signal propagation rules*) – if there is such a path in the design hierarchy (through which propagated signal will asynchronously return to its source) => combinational feedback loop is detected and violation message is displayed (*see Note-2 for message displaying rules*) Note-1: following rules are imposed on propagation of signal through the design hierarchy: – propagation stops on following objects (picture at the right side shows stop of propagation for the "SIG" signal): – synchronous inputs of flip-flops / latches – black boxes  – propagation resumes from outputs of objects that stopped the propagation before (picture below shows that propagation is resumed for signal "SIG" after it has been stopped with data input of the latch "L")  "SIG" propagation starts "SIG" propagation stops "L.SIG" propagation starts L.SIG SIG L.SIG SIG SIG L.SIG LD L.SIG "L.SIG" propagation stops COMB_LOGIC Feedback on "L.SIG" is detected Note-2: violation message is displayed regarding the line for which feedback is detected – feedback is an intermediate line (it is not explicitly specified in the description, but it is implied to be generated by the logic synthesis tool – see the example below) => violation message #2 is displayed (*)	



– feedback is not an intermediate line (see the example below) => violation message #1 is displayed [(*)]



– [(*)] two modes are provided to display details for violation message (depending on state of the parameter):
  - parameter DETAILED_PROPAGATION_CHAIN is equal to “0” (default)
    - violation is reported in short form: no details displayed *(see the example #1)*
  - parameter DETAILED_PROPAGATION_CHAIN is equal to “1”
    - violation is reported in full form: details describe the path through which combinational feedback is propagated *(see the example #2)*
      - when combinational feedback is propagated through an intermediate line, detail #1 is displayed
      - when combinational feedback is propagated through a line that is not intermediate, detail #2 is displayed
      - when combinational feedback is propagated through flip-flops or latches, detail #3 is displayed followed by a list of possible objects *({ObjectType})* and ports corresponding to them *({PortType}):*

{ObjectType}	***{PortType}***
FF latch	asynchronous reset
	asynchronous set

      - when combinational feedback propagates through a submodule instance, detail #4 is displayed

EXAMPLE: [1] violation is reported in the detailed form: feedback propagation path is described (DETAILED_PROPAGATION_CHAIN = 1)

[2] consider the design hierarchy at the picture below

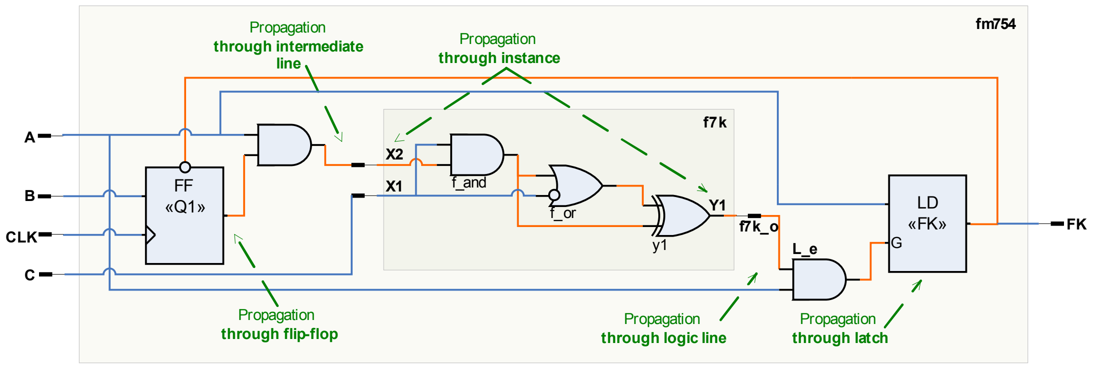
[3] note, that all possible paths of feedback propagation are demonstrated:

– through the submodule instance

– through the flip-flop and latch

– through the combinational logic line

– through the intermediate combinational logic line



```
module fm754( A, B, C, CLK, FK );

    input  A, B, C, CLK;
    output FK;
    reg    FK;

    wire   f7k_o, RST, L_e;
    reg    Q1;

    always @( A or L_e )
        if( L_e )
            FK = A;

    always @( posedge CLK or negedge FK )
        if( !FK )
            Q1 <= 1'b0;
        else
            Q1 <= B;

    f7k CMB_I0( .x2( A & Q1 ), .x1( C ), .y1( f7k_o ) );

    assign L_e = f7k_o & A;

endmodule

module f7k( x1, x2, y1 );

    input  x1, x2;
    output y1;

    wire   f_and, f_or;

    assign f_and = x1 & x2;
    assign f_or  = ~x1 | f_and;
    assign y1    = f_and ^ f_or;

endmodule
```

Instance "fm754". "STARC 1.2.1.3" Combinational feedback is detected. Do not use feedbacks in combinational circuits to avoid the effect of a feedback loop during the timing analysis.

Asynchronous loop propagates through latch "FK" enable input

Asynchronous loop propagates through combinational logic

Asynchronous loop propagates through FF "Q1" asynchronous reset input.

Asynchronous loop propagates through combinational logic

Asynchronous loop propagates through submodule instance "fm754.CMB_I0" from port "x2" to "y1"

Asynchronous loop propagates through combinational logic line "L_e"

1.3 Initial reset

1.3.1 Use asynchronous reset for initial reset

STARC_VLOG 1.3.1.2

RULE NAME	**It is safer to use asynchronous reset for initial reset to a register. - Reset tree synthesis at layout is easy. - Values may not be fixed in a gate-level simulation with synchronous reset.**	
MESSAGE-1	**Global reset "{GlResetSignalName}" is connected to FF "{FFName}" synchronous {ControlType} input. Initial reset should be connected to asynchronous control input to avoid problems with simulation.**	
MESSAGE-2 [INFO]	*None of the specified reset signals was found, auto-detect mode is used. Detected reset signals: {GlResetSignalNameList}.*	
MESSAGE-3 [INFO]	*None of the specified reset signals was found, auto-detect mode is used. No global resets detected.*	
MESSAGE-4 [INFO]	*Specified global reset signal "{SignalName}" could not be found in the design.*	
MESSAGE-5 [INFO]	*Info: No reset signals specified, auto-detect mode is used. Detected reset signals: {GlResetSignalNameList}.*	
MESSAGE-6 [INFO]	*No reset signals specified, auto-detect mode is used. No global resets detected.*	
PROBLEM DESCRIPTION	In some kinds of circuits (for example, state machine which depends on the previous state) using synchronous reset may cause simulation problems. In the beginning of simulation reset is used to clear all registers, but in case of synchronous reset previous state is unknown and may block propagation of the reset to synchronous input, registers are not cleared and simulation result is not valid. So it is recommended to use asynchronous reset for initial reset of a register.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker detects signals that are used as synchronous set/reset for FF(s).: – if signal has **'global_reset'** attribute => violation (message-1). Note: see 1.4.3.4 for details about info messages 2-6.	

EXAMPLE-1: [1] signal top.greset has attribute 'global_reset' (`-alint_grst top.greset`);
[2] the signal is used as synchronous control => violation.

```
module top( clk, greset, data, out1, out2 );

    input  clk;
    input  data;
    input  greset;

    output out1;
    output out2;

    dff_synch DFF1( .D  ( data ), .RESET( greset ), .CLK( clk ), .Q( out1 ) );
    dff_async DFF2( .D  ( data ), .RESET( greset ), .CLK( clk ), .Q( out2 ) );

endmodule


module dff_sync( D, RESET, CLK, Q );
```

```
    input  D;
    input  CLK;
    input  RESET;

    output reg Q;

    always @( posedge CLK )

        if ( RESET )

            Q <= 1'b0;

        else

            Q <= D;

endmodule
```

Instance "top.DFF1" Global reset “top.greset” is connected to FF “Q” synchronous reset input. Initial reset should be connected to asynchronous control input to avoid problems w ith simulation.

```
module dff_async( D, SET, CLK, Q );

    input  D;
    input  CLK;
    input  SET;

    output reg Q;

    always @( posedge CLK, negedge RESET )

        if ( !RESET )

            Q <= 1'b0;

        else

            Q <= D;

endmodule
```

STARC_VLOG 1.3.1.3

RULE NAME	**Do not use asynchronous set/reset pins for anything other than initial reset**
MESSAGE-1	**Global reset "{GlResetSignalName}" is connected to {ObjectType} "{ObjectName}" {PortType} input. Global reset should be connected directly to initial reset pins of flip-flops or latches.**
MESSAGE-2	**Global reset "{GlResetSignalName}" is connected to {ObjectType} {PortType} input. Global reset should be connected directly to initial reset pins of flip-flops or latches.**
MESSAGE-3 [INFO]	*Specified global reset signal "{SignalName}" could not be found in the design.*
MESSAGE-4 [INFO]	*None of the specified reset signals was found, auto-detect mode is used. No global resets detected.*
MESSAGE-5 [INFO]	*None of the specified reset signals was found, auto-detect mode is used. Detected reset signals: {GlResetSignalNameList}.*
MESSAGE-6 [INFO]	*No reset signals specified, auto-detect mode is used. No global resets detected.*
MESSAGE-7 [INFO]	*No reset signals specified, auto-detect mode is used. Detected reset signals: {GlResetSignalNameList}.*
PROBLEM DESCRIPTION	When combinational circuit generates an asynchronous reset signal, there are situations possible in which hazards will occur as result of optimization by logic synthesis tool. Therefore, signals other than initial resets are forbidden for asynchronous control pins of flip-flops.
	LEVEL: RECOMMENDATION 1
CHECKER BEHAVIOR	Checker propagates reset signal(s) through the design hierarchy and restricts its connection to pins other than FF/latch asynchronous control pins: – see the rule 1.4.3.4 (behavior of this checker is almost the same(*)) – *(*) the main difference between these checkers is:* – *1.3.1.4 restricts connection of global clock signal(s) (specified with -**alint_gclk** switch) to pins other than FF clock pins* – *1.2.3.3 restricts connection of global reset signal(s) (specified with -**alint_grst** switch) to pins other than FF asynchronous control pins* Note-1: violation message is displayed per each object with improperly mapped reset (see the table from Note-2 for list of possible objects and their pins) Note-2: following table defines set of strings that are possible for the {ObjectType} token (different from 1.4.3.4):

{ObjectType}	{PortType}
FF	clock
	data
	enable
latch	data
	enable
multiplexer	select
tri-state buffer	Enable

EXAMPLE-1: [1] global reset is specified with "-alint_grst" switch (for example in command line like this: **alint *-alint_grst*** *top.RESET* ***-asim*** *top*);

[2] module 'tristate' infers a tri-state buffer;

[3] module 'top' creates instance of 'tristate' with global reset signal 'RESET' connected to enable input of instantiated tri-state.

```
module top( RESET, D_IN, D_OUT ); // -alint_grst top.RESET

    input  RESET;
    input  D_IN;

    output D_OUT;

    tristate TRISTATE1 ( .in1( D_IN ), .en( RESET ),  .out1( D_OUT ) );

endmodule

module tristate( in1, en, out1 );

    input  in1;
    input  en;

    output out1;

    assign out1 = ( en )? in1 : 1'bz;

endmodule
```

Instance "top.TRISTATE1". Global reset "top.RESET" is connected to tri-state buffer "out1" enable input. Global reset should be connected directly to initial reset pins of flip-flops or latches.

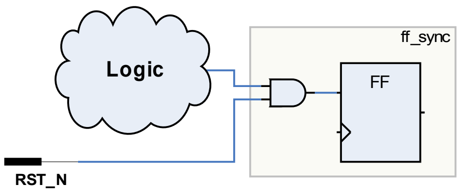
EXAMPLE-2: [1] global resets are auto-detected from signals 'RESET' and 'SET' that are mapped to asynchronous control pins of flip-flop "DATA_OUT" (note, that special message #7 is displayed to indicate the list of reset signals that are auto-detected);

[2] global reset signal 'RESET' is also connected to combinational logic that is mapped to flip-flop 'DATA_OUT' clock pin => violation (message #1 is displayed).

```
module top( CLK, RESET, SET, DATA_IN, DATA_OUT );

    input      CLK, DATA_IN, RESET, SET;
    output reg DATA_OUT;

    and( CLK_gated, CLK, RESET );

    always @( posedge CLK_gated or negedge RESET or negedge SET )
        if( RESET )
            DATA_OUT <= 1'b0;
        else if( SET )
            DATA_OUT <= 1'b1;
        else
            DATA_OUT <= DATA_IN;
endmodule
```

No reset signals specified, auto-detect mode is used. Detected reset signals: RESET, SET.

STARC_VLOG 1.3.1.5

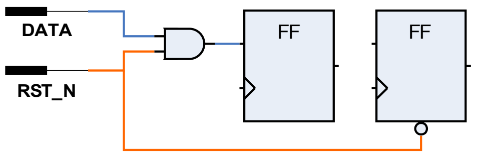
RULE NAME	**Do not use synchronous reset directives for a particular logic synthesis tool**
MESSAGE	**Do not use reset directives that depend on particular logic synthesis tool. Such directives can not guarantee that RTL and post-synthesis results will always match - it is recommended to avoid such directives by using an additional hierarchy with encapsulated control logic.**
PROBLEM DESCRIPTION	In some circuit (like mentioned in 1.3.1.2) the value of the FF becomes unknown state X. Guaranteeing a known reset state can be achieved by creating a separate module containing a FF with an AND gate as seen in the picture. By creating this extra hierarchical module, the RTL description will not generate logic which is put in an indeterminate state upon reset.  If you add the Design Compiler specific directive "//synopsys sync_set_reset" you could assure a value with synchronous reset without using a hierarchical FF. Therefore, if you are not using an additional hierarchy, this directive should be added. However, as this is only effective with Design Compiler, it will not be possible to use this RTL description if other logic synthesis tools are used in the future. Also, because this Synopsys directive is implemented in comment lines, syntax cannot be checked, and it is recognized as a simple comment statement even if only one character is wrong. This method cannot guarantee that a tool always generates a circuit as illustrated above. To ensure that synchronous reset defines value at gate simulation, there is no other way but the use of hierarchy method as above.
LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker catches following directives in the module: – `// synopsys sync_set_reset` – `// synopsys async_set_reset` – `// synopsys async_set_reset_local` – `// synopsys sync_set_reset_local` – `// cadence sync_set_reset` – `// cadence async_set_reset` – `// ambit synthesis set_reset asynchronous` – `// ambit synthesis set_reset synchronous` – `(* synthesis, async_set_reset *)` – `(* synthesis, sync_set_reset *)`

EXAMPLE-1: [1] `synopsys sync_set_reset` directive is detected => violation

```
//synopsys sync_set_reset "set"
always @( posedge CLK )

    if ( SET )

        Q <= 1'b1;

    else

        Q <= DATA;
```

Do not use reset directives that depend on particular logic synthesis tool. Such directives can not guarantee that RTL and post-synthesis results w ill alw ays match - it is recommended to avoid such directives by using an additional hierarchy w ith encapsulated control logic.

STARC_VLOG 1.3.1.6

RULE NAME	**Do not have both asynchronous reset and synchronous reset on the same reset line**	
MESSAGE	**Signal "{SigName}" is used both for synchronous and asynchronous reset line(s). Do not mix synchronous and asynchronous resets in one reset line to avoid problems with logic synthesis and layout.**	
	DETAIL-1	*Connection to asynchronous control pin of flipflop "{FFName}" is detected.*
	DETAIL-2	*Connection to synchronous control pin of flipflop "{FFName}" is detected.*
	DETAIL-3	*Signal "{SigName}" is also used both for synchronous and asynchronous reset line(s) in the same connections.*
PROBLEM DESCRIPTION	If one reset line has both synchronous reset and asynchronous reset, synthesis may not be performed properly. The asynchronous reset line sometimes forms a tree-structure during the layout process. To avoid accidentally inserting a buffer or logical operand during logic synthesis with Design Compiler, *set_ideal_net* directive may be put on this reset line. This directive means that the specified net is excluded from the limitation of the logic synthesis, the timing analysis, and the design rule. Then, if this reset line is also input to a FF's synchronous input, that part will not be synthesized and synthesis will fail as a result. In addition to this, having both asynchronous reset and synchronous reset may cause other problems during logic synthesis and layout, so they should not be mixed. 	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects signals that are used simultaneously as both asynchronous set/reset and synchronous set/reset in the design: – if such signals detected => violation – connection to asynchronous reset pin => detail-1 – connection to synchronous reset pin => detail-2 – if several signals drive simultaneously both asynchronous set/reset and synchronous set/reset pins => only one main message is displayed for all of them (selected by name), and detail-3 per each other driver-signal.	

EXAMPLE-1: [1] signal is used simultaneously as asynchronous reset and synchronous reset => violation (message-1 + detail-1/detail-2).

```
module top( DATA, CLK, RST, Q1, Q2 );

    input        CLK;
    input        RST;
    input  [1:0] DATA;

    output reg Q1;
    output reg Q2;

    always @( negedge CLK ) begin

        if ( !RST )
            Q1 <= 1'b0;
        else
            Q1 <= DATA[0];
    end

    always @( negedge CLK or posedge RST ) begin

        if ( !RST )
```

Signal "RST" is used both for synchronous and asynchronous reset line(s). Do not mix synchronous and asynchronous resets in one reset line to avoid problems w ith logic synthesis and layout.

Connection to synchronous control pin of flipflop "Q1" is detected.

```
            Q2 <= 1'b0;
        else
            Q2 <= DATA[1];
    end

endmodule
```

Connection to asynchronous control pin of flipflop “Q2” is detected.

EXAMPLE-1: [1] two signals drive simultaneously asynchronous reset and synchronous reset pins => violation (message-1 + detail-1/detail-2 + detail-3).

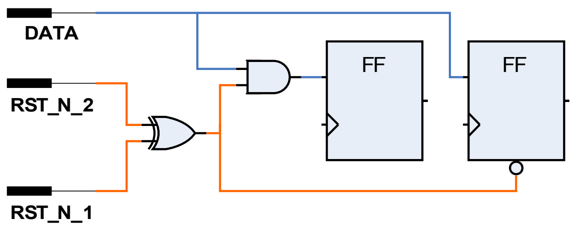
```
module top( DATA, CLK, RST_N_1, RST_N_2, Q1, Q2 );

    input        CLK;
    input        RST_N_1, RST_N_2;
    input  [1:0] DATA;

    output reg Q1;
    output reg Q2;

    wire TMP;

    assign TMP = RST_N_1 ^ RST_N_2;

    always @( negedge CLK ) begin

        if ( !TMP )
            Q1 <= 1'b0;
        else
            Q1 <= DATA[0];

    end

    always @( negedge CLK or negedge TMP ) begin

        if ( !TMP )
            Q2 <= 1'b0;
        else
            Q2 <= DATA[1];

    end

        endmodule
```

Signal “RST_N_1” is used both for synchronous and asynchronous reset line(s). Do not mix synchronous and asynchronous resets in one reset line to avoid problems w ith logic synthesis and layout.

Signal "RST_N_2" is also used both for synchronous and asynchronous reset line(s) in the same connections.

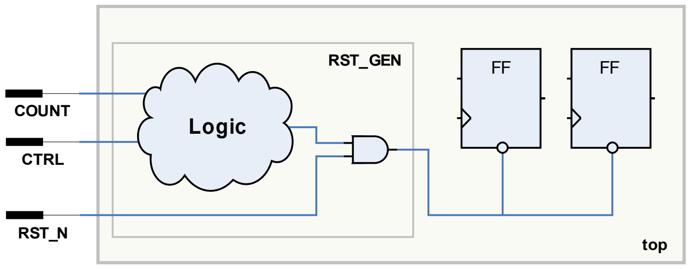
Connection to synchronous control pin of flipflop “Q1” is detected.

Connection to asynchronous control pin of flipflop “Q2” is detected.



1.3.2 Reset line hazards

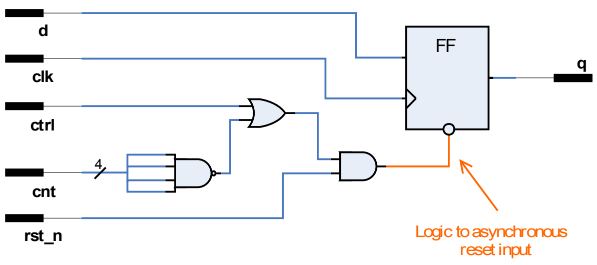
STARC_VLOG 1.3.2.1

RULE NAME	**Do not insert logical operands in a reset line at the local module. In addition, circuits that supply reset lines should be separated into an individual module.** **- Logic order may be replaced by synthesis.** **- Hazards cannot be prevented in the RTL description.**
MESSAGE	**Combinational logic is connected to asynchronous {ControlType} of flip-flop "{FFName}". Do not insert logical operands in a reset line at the local module to avoid problems with optimization of this logic by synthesis tools. It is recommended to separate into individual modules such circuits that supply reset lines.**
PROBLEM DESCRIPTION	An asynchronous reset signal may be supplied as synchronized and additional logic may be inserted in reset line to prevent unstable operation. Also, logic circuits may be inserted so that a system reset can be selected. When logic is needed on a reset line, combine that reset line logic together in the top level (see picture below) as much as possible and directly input the same signal input to all FFs. By creating modules for reset generation, it will also become easier to apply synthesis constraints.  RST_GEN FF FF COUNT Logic CTRL RST_N top
LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker verifies connections to each asynchronous control input of each FF: – if any logic is connected to such control and this logic is defined within the same module => violation Note: following objects are skipped (not treated as logic): – buffers / inverters; – multiplexers; – tri-states; – latches / FFs.

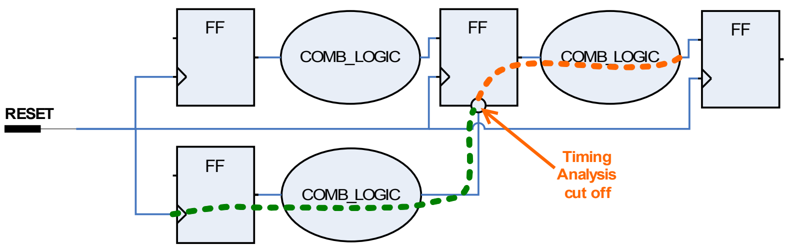
EXAMPLE-1: [1] consider picture below;
[2] logic is connected to asynchronous control input of FF;
[3] the logic is defined within the same module => violation

```
module top (clk, rst_n, ctrl, cnt, d, q);

    input clk, rst_n, ctrl, d;
    input [3:0] cnt;
    output reg q;
    wire [3:0] cnt16;

    assign cnt16 = ~(& cnt) | ctrl;
    assign rcnt_n = rst_n & cnt16;

    always @( posedge clk or negedge rcnt_n )

        if( !rcnt_n )
            q <= 1'b0;
        else
            q <= d;

endmodule
```

Combinational logic is connected to asynchronous reset of flip-flop "q". Do not insert logical operands in a reset line at the local module to avoid problems w ith optimization of this logic by synthesis tools. It is recommended to separate into individual modules such circuits that supply reset lines.



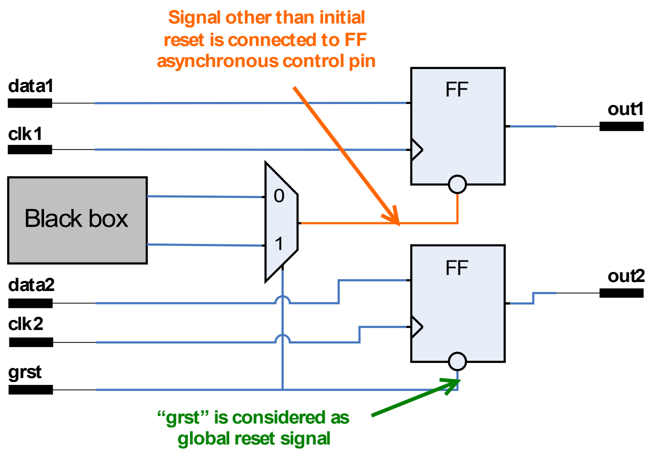
STARC_VLOG 1.3.2.2

RULE NAME	**Do not insert signals other than initial reset to FF asynchronous reset pins**	
MESSAGE-1	**Signal(s) other than global reset(s) is connected to asynchronous control pin(s) of flip-flop "{FFSignalName}". Initial reset should be input for global asynchronous reset to avoid path analysis problems with logic synthesis and static timing tools.**	
	DETAIL	*Asynchronous control signal is other than global reset.*
MESSAGE-2 [INFO]	*Specified global reset signal "{SignalName}" could not be found in the design.*	
MESSAGE-3 [INFO]	*None of the specified reset signals was found, auto-detect mode is used. No global resets detected.*	
MESSAGE-4 [INFO]	*None of the specified reset signals was found, auto-detect mode is used. Detected reset signals: {GlResetSignalNameList}.*	
MESSAGE-5 [INFO]	*No reset signals specified, auto-detect mode is used. No global resets detected.*	
MESSAGE-6 [INFO]	*No reset signals specified, auto-detect mode is used. Detected reset signals: {GlResetSignalNameList}.*	
PROBLEM DESCRIPTION	Signals other than initial reset should not be input for asynchronous reset pins, because it is difficult to analyze the paths which the asynchronous reset and set pass through during the timing analysis (such timing paths are cut off without taking into account – see the picture below for example).  FF COMB_LOGIC FF COMB_LOGIC FF RESET FF COMB_LOGIC Timing Analysis cut off Therefore, avoid such descriptions when using logic synthesis tools or static timing analysis tools to perform timing analysis.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker restricts signal that is connected to asynchronous control pin of flip-flop to be global reset signal: – global reset signal(s) (specified with **-alint_grst** switch or auto-detected) is propagated through the design hierarchy – asynchronous control pin(s) of each flip-flop should be connected to global reset (it means that propagation reached this pin) – if propagation does not reach asynchronous control pin => violation message #1 is displayed (per FF signal assignment) with detail message (per appropriate asynchronous control branch) – *see 1.4.3.4 for details:* – *rules for auto-detection (during the auto-detection, each signal that is connected to asynchronous control pin of flip-flop is considered as reset signal)* – *rules for global reset propagation (global reset propagates through buffers / inverters / combinational logic / multiplexers data / tri-state inputs)* – *rules for displaying info-messages (indicate the list of auto-detected clock or report about reset signals that are specified with -alint_gclk but could not be found)*	

EXAMPLE-1: [1] consider the design hierarchy at the picture below;

[2] signal "grst" is considered as global reset signal because it is directly connected to asynchronous control pin of flip-flop "out2" (and other reset signal(s) is not specified with '-*alint_grst*' command line switch);

[3] output of multiplexer is connected to asynchronous control pin of flip-flop "out1" => violation message is displayed (there is no global reset signal at inputs of the MUX).


Signal other than initial reset is connected to FF asynchronous control pin
data1
clk1
FF
out1
Black box
0
1
data2
clk2
FF
out2
grst
"grst" is considered as global reset signal

```
module top ( clk1, clk2, grst, data1, data2, out1, out2 );

   input  clk1, clk2;
   input  grst;
   input  data1, data2;

   output out1, out2;

   wire   muxout;
   wire   bboxout1;
   wire   bboxout2;


   assign muxout = ( grst ) ? bboxout1 : bboxout2;

   dff      DFF_INSTANCE1 ( .D( data1 ), .CLK( clk1 ), .RESET( grst ), .Q( out1 ) );
   dff      DFF_INSTANCE2 ( .D( data2 ), .CLK( clk2 ), .RESET( muxout ), .Q( out2 ) );

   blackbox BBOX_INSTANCE ( .out1( bboxout1 ), .out2( bboxout2 ) );

endmodule

module dff( D, CLK, RESET, Q );

   input  D;
   input  CLK;
   input  RESET;

   output Q;
   reg    Q;

   always @( posedge CLK or posedge RESET )
      if ( RESET )
         Q <= 1'b0;
      else
         Q <= D;

endmodule

module blackbox( out1, out2 );
    output out1, out2;
endmodule
```

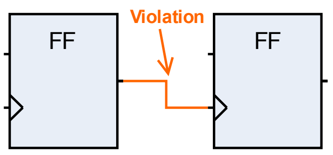
Instance "top.DFF_INSTANCE2". Signal(s) other than global reset(s) is connected to asynchronous control pin(s) of flip-flop "Q". Initial reset should be input for global asynchronous reset to avoid path analysis problems w ith logic synthesis and static timing tools.

Asynchronous control signal is other than global reset.

1.4 Clocks

1.4.3 Gated clocks should be used with special care

STARC_VLOG 1.4.3.2

RULE NAME	**Do not input a FF output pin to other FF clock pins**	
MESSAGE	**The clock pin of flip-flop "{DrivenFFName}" is driven by the output of another flip-flop. CTS tool will not be able to take the clock line balancing into consideration.**	
	DETAIL	*Clock pin is driven with the output of flip-flop "{DriverFFHierName}"*
PROBLEM DESCRIPTION	Clock Tree Synthesis (CTS) tools are used to synthesize and route clock lines after the placement of each FF in the layout. When output pin of some flip-flop is supplied to clock pin of other flip-flop, the CTS tool will not be able to take the clock line balancing into consideration. 	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans the design hierarchy for flip-flops that have output of another flip-flop supplied to their clock pins: – back-propagation is performed from clock pin of each flip-flop to define its driver (*See Note-1 for backward propagation rules*) – clock pin driver is a signal on which backward propagation stops – if the driver is the output of another flip-flop (as shown on image in the “Problem description” section) => violation is reported (main message points at the process that infers the driven flip-flop, whereas detail message points at the assignment of signal that infers the driving flip-flop) Note-1: the following rules are imposed on backward propagation from the signal through the design hierarchy: – backward propagation starts from driven signal, passes through instances and stops on any except of following objects: – inverters – buffers – direct connections (wires) – driver is a signal at which propagation stops	

EXAMPLE-1: [1] two modules infer flip-flops (the first flip-flop has an asynchronous reset, the second one does not have one);

[2] rule violation at the third (top-level) module: output of first flip-flop is mapped to the clock pin of the second one;

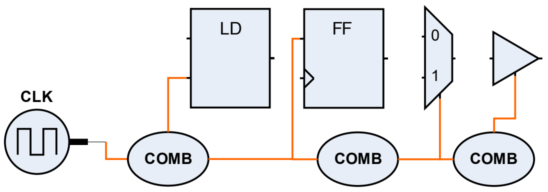
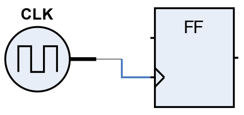
```
module chain( I_A, I_B, I_C, I_D, I_RST, I_CLK, O_A, O_B );
input  I_A, I_B, I_C, I_D, I_RST, I_CLK;
output O_A, O_B;
    M_FF1 U0  ( .I_A( I_A ), .I_CLK ( I_CLK ), .I_RST( I_RST ), .O_A( O_A ) );
    M_FF2 U1  ( .I_A( I_A ), .I_CLK ( O_A ), .O_A( O_B ) );
endmodule

module M_FF1  ( I_A, I_CLK, I_RST, O_A );
input      I_A, I_CLK, I_RST;
output reg O_A;
```

```
    always@( posedge I_CLK or negedge I_RST ) begin
        if ( !I_RST ) begin
            O_A <= 1'b0;
        end
        else begin
            O_A <= I_A; // detail: output pin of flip-flop "test_1_4_3_2.U0.O_A"
                        // is a driver for clock pin of "test_1_4_3_2.U1.O_A"
                        //
        end
    end
endmodule

module M_FF2 ( I_A, I_CLK, O_A );
input      I_A, I_CLK;
output reg O_A;

always@( posedge I_CLK ) begin // main message: clock pin of flip-flop
     O_A <= I_A;               // "test_1_4_3_2.U1.O_A" is driven by
                               // output pin of flip-flop "test_1_4_3_2.U0.O_A"
end
endmodule
```

Clock pin is driven w ith the output of flip-flop "test_1_4_3_2.U0.O_A"

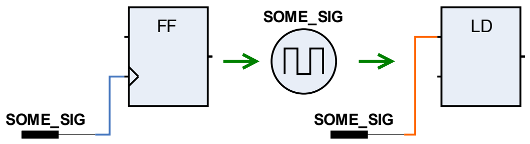
Instance "chain.U1". The clock pin of flip-flop "test_1_4_3_2.U1.O_A" is driven by the output of another flip-flop. CTS tool w ill not be able to take the clock line balancing into consideration.

STARC_VLOG 1.4.3.4

RULE NAME	**Do not supply clock signals to pins other than FF clock input pins (such as D input)**	
MESSAGE-1	**Do not connect clock signal to the pin other than to the FF clock input pin, it may lead to the incorrect timing analysis**	
	DETAIL-1	*Global clock signal(s) "{GlClkSignalNameList}" is connected to the {ObjectType} "{ObjectName}" {PortType} input*
	DETAIL-2	*Global clock signal(s) "{GlClkSignalNameList}" is connected to the multiplexer select input*
MESSAGE-2 [INFO]	*Specified global clock signal "{SignalName}" could not be found in the design.*	
MESSAGE-3 [INFO]	*None of the specified clock signals was found, auto-detect mode is used. No global clocks detected.*	
MESSAGE-4 [INFO]	*None of the specified clock signals was found, auto-detect mode is used. Detected clock signals: {GlCLKSignalNameList}.*	
MESSAGE-5 [INFO]	*No clock signals specified, auto-detect mode is used. No global clocks detected.*	
MESSAGE-6 [INFO]	*No clock signals specified, auto-detect mode is used. Detected clock signals: {GlCLKSignalNameList}.*	
PROBLEM DESCRIPTION	Clock signals should not be connected to input pins other than clock pins of flip-flops. Paths having clock lines connected to latches, enabling logic of multiplexers and tri-states, non-clocking pins of flip-flops are extremely dangerous (consider the picture at the right side): it is impossible to analyze timing correctly for them.  Additionally, logic synthesis tools will not perform any optimizations for such paths.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker propagates clock signal(s) through the design hierarchy and restricts its connection to pins other than FF clock input pins: – correct case is shown on the picture at the right side of the page  – propagation of each clock signal through the design hierarchy (*see Note-1 for clock signals determination and Note-2 for clock signals propagation*) restricts clock signal connection to: – <u>flip-flop</u>: it is not allowed to supply clock signal to pins other than clock pin; – <u>latch</u>: it is not allowed to supply clock signal to any pin; – <u>multiplexer</u>: it is not allowed to supply clock signal to select pin(s); – <u>tri-state</u>: it is not allowed to supply clock signal to enable pin; – violation message is displayed for each case where any of upper options is not true – see Note-3 for list of possible objects (*{ObjectType}*) and ports corresponding to them (*{PortType}*). Note-1: there are two routines provided to detect clock signal(s): – clock signal(s) is specified with **`-alint_gclk`** command line switch (example: **alint *-alint_gclk*** top.clk1 ***-asim*** top); – clock signal(s) is determined automatically (when there is no clock(s) specified with **`-alint_gclk`** switch): – each signal [*] connected to clock pin of any flip-flop in the design hierarchy is	

RULE NAME	**Do not supply clock signals to pins other than FF clock input pins (such as D input)**

considered as a clock signal (picture below demonstrates that “SOME_SIG” considered as a clock because of its connection to the clock pin of flip-flop)



- *$^{(*)}$ following requirements are imposed on such signal:*
  - *it should be the input port of the top-level module*
  - *it should not be connected through the combinational logic (except buffers and inverters)*
- information message-2 is displayed if automatic mode is enabled

Note-2: following rules are used to propagate clock signal(s) through the design hierarchy:

- clock signal is propagated through:
  - combinational logic (logic gates);
  - multiplexer data inputs;
- clock signal is not propagated through:
  - flip-flops;
  - latches;
  - tri-states.

Note-3: following cases are possible *{ObjectType}* and *{PortType}* that are displayed in the violation message:

{ObjectType}	***{PortType}***
FF	data
	enable
	asynchronous set
	asynchronous reset
	synchronous set
	synchronous reset
latch	data
	enable
	asynchronous set
	asynchronous reset
multiplexer	select
tri-state buffer	enable

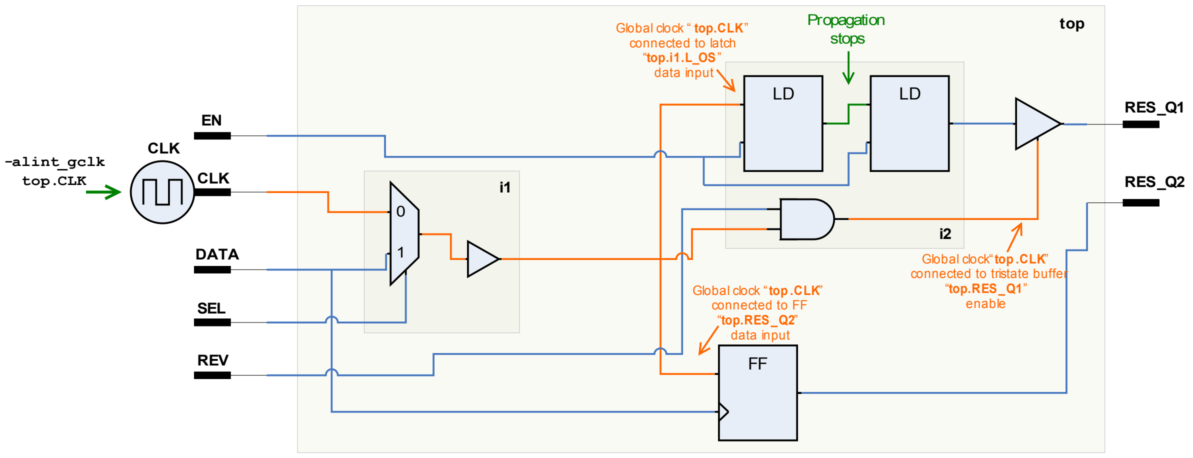
Note-4: this checker displays special messages (#2 – #6, severity = note) when there are some problems with detection of global clock signal(s) in the design:

- `alint_gclk` switch is specified:
  - when any signal from the list of global clocks could not be found in the design (but at least one of specified signals is found) => message #2 is displayed per each signal that could not be found;
  - when all signal(s) from the list of global clocks could not be found in the design:
    - if no clock signals are auto-detected => message #3 is displayed;

RULE NAME	Do not supply clock signals to pins other than FF clock input pins (such as D input)
	– if some clock signals are auto-detected => message #4 is displayed (contains the list of auto-detected clocks); – `alint_gclk` switch is not specified: – if no any clock signals are auto-detected => message #5 is displayed; – if some clock signals are auto-detected => message #6 is displayed (contains the list of auto-detected clocks).

EXAMPLE-1: [1] consider design hierarchy that is represented on the picture below: common violation cases are considered (clock propagation routine is marked with orange color);

[2] clock is specified with "**-alint_gclk**" switch (**alint *-alint_gclk*** top.CLK ***-asim*** top).


-alint_gclk
top.CLK
CLK
EN
CLK
DATA
SEL
REV
i1
0
1
Global clock " top.CLK" connected to latch "top.i1.L_OS" data input
Propagation stops
LD
LD
top
RES_Q1
RES_Q2
i2
Global clock"top.CLK" connected to tristate buffer "top.RES_Q1" enable
Global clock "top.CLK" connected to FF "top.RES_Q2" data input
FF

```
module top ( en, sel, CLK, data, rev, res_q1, res_q2 );

    input en, sel, CLK, data, rev;
    output res_q1, res_q2;

    reg res_q1, res_q2;

    wire buf_to_i1, buf_to_dff;
    wire g_to_en;
    wire ld_to_tri;

    mux_n_buf i1 ( .sel( sel ),
                   .in1( CLK ),
                   .in2( data ),
                   .out1( buf_to_i1 ),
                   .out2( buf_to_dff ) );

    always @( posedge CLK )

        res_q2 <= buf_to_dff;

    ld_n_and i2 ( .L_I( buf_to_dff ),
                  .L_E( en ),
                  .rev( rev ),
                  .L_O( ld_to_tri ),
                  .g_to_en( g_to_en ) );

    always @( g_to_en, ld_to_tri )

        if ( g_to_en )
            res_q1 <= 1'bz;
        else
            res_q1 <= ld_to_tri;
```

Do not connect clock signal to the pin other than to the FF clock input pin, it may lead to the incorrect timing analysis

Global clock signal(s) "top.CLK" is connected to the FF "top.res_q2" data input

Global clock signal(s) "top.CLK" is connected to the tristate buffer "top.res_q1" enable input

Do not connect clock signal to the pin other than to the FF clock input pin, it may lead to the incorrect timing analysis

```
endmodule

module mux_n_buf ( sel, in1, in2, out1, out2 );

    input sel, in1, in2;
    output out1, out2;

    reg mux_out;

    always @( sel, in1, in2 )
        if ( sel == 1'b0 )
            mux_out <= in1;
        else if ( sel == 1'b1 )
            mux_out <= in2;

    buf ( out1, out2, mux_out );

endmodule

module ld_n_and ( L_I, L_E, rev, L_O, g_to_en );

    input L_I, L_E, rev;
    output L_O, g_to_en;

    reg L_O;

    reg L_OS;

    always @( L_E, L_I )
        if ( L_E )
            L_OS <= L_I;

    always @( L_E, L_OS )
        if ( L_E )

            L_O <= L_OS;


    and ( g_to_en, L_I, rev );

endmodule
```

Do not connect clock signal to the pin other than to the FF clock input pin, it may lead to the incorrect timing analysis

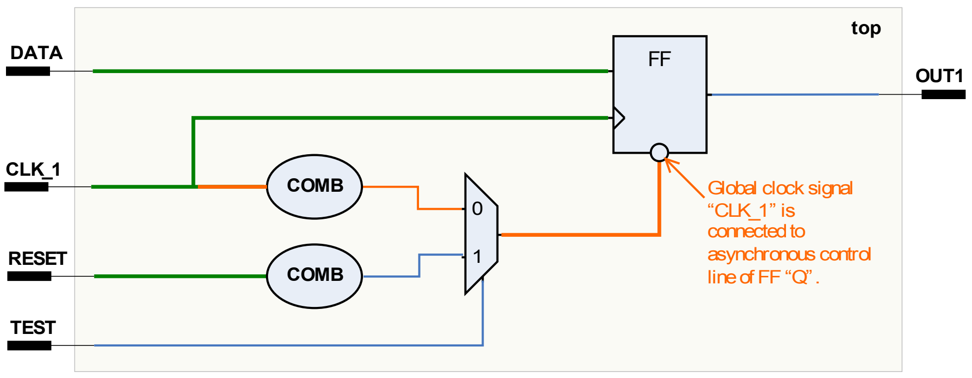
Global clock signal(s) "top.CLK" is connected to the latch "top.i1.L_OS" data input

STARC_VLOG 1.4.3.5

RULE NAME	**Clock signals should not be connected to black boxes, bi-directional pins or reset lines**
MESSAGE-1	**Global clock signal "{GlCLKName}" is connected to black box "{ObjectName}". Do not connect clocks to black-boxes, bi-directional terminals and reset lines to avoid problems with test tools including the BIST insertion.**
MESSAGE-2	**Global clock signal "{GlCLKName}" is connected to bi-directional port "{PortName}" of "{ObjectName}". Bi-directional terminals must be controlled as either inputs or as outputs during a simulation. Do not connect clocks to black-boxes, bi-directional terminals and reset lines to avoid problems with test tools including the BIST insertion.**
MESSAGE-3	**Global clock signal "{GlCLKName}" is connected to asynchronous control line of FF "{ObjectName}". Do not connect clocks to black-boxes, bi-directional terminals and reset lines to avoid problems with test tools including the BIST insertion.**
PROBLEM DESCRIPTION	Clock signals should not be connected to input pins other than clock pins of flip-flops: When clock line is connected to black box, ATPG tool could fail to insert test scan and generate test patterns (*see* 3.3.2.2 *for details*). Also, do not connect clocks to bi-directional terminals. Bi-directional terminals must be controlled as either inputs or as outputs during a simulation. Test tools, including the BIST insertion tool, cannot determine the direction of these bi-directional terminals. Signals other than initial reset (for example – clock) should not be connected to asynchronous reset to avoid problems when using logic synthesis tools or static timing analysis tools(*see* 1.3.2.2 *for details*). LEVEL: Recommendation 3
CHECKER BEHAVIOR	Checker scans the design hierarchy to analyze input lines of: • black-boxes • bi-directional ports (inout ports) • asynchronous set/reset of flip-flops If any of these lines is connected to clock signal(s) (signal specified via **`-alint_gclk`** command line switch) – a respective warning message is issued. Note: refer to 1.4.3.4 for details about **-alint_gclk** switch and clock auto detection.

EXAMPLE 1: [1] consider design hierarchy that is represented on the picture below (clock propagation routine is marked with orange color)

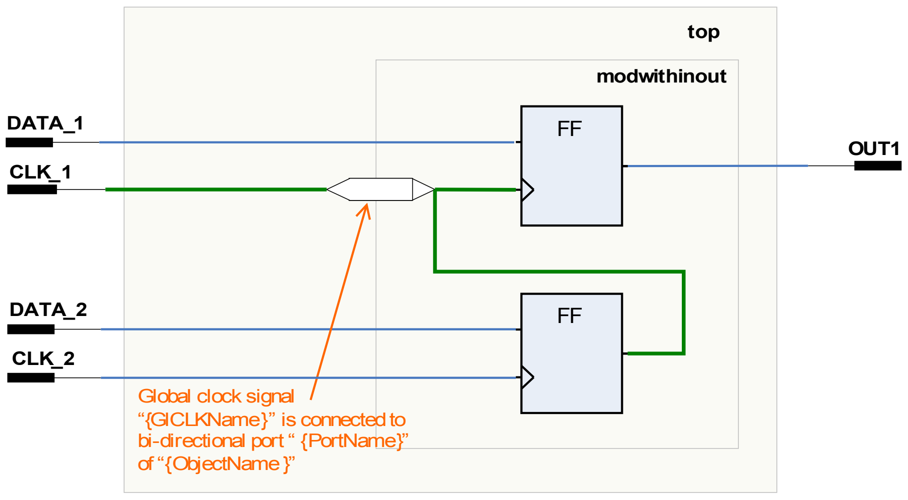
[2] clock "top.CLK_1" is specified with **-alint_gclk** switch (**alint *-alint_gclk*** top.CLK_1 ***-asim*** top)


top
DATA
FF
OUT1
CLK_1
COMB
0
1
RESET
COMB
TEST
Global clock signal "CLK_1" is connected to asynchronous control line of FF "Q".

```
module top( TST, RESET, RESET_EN, DATA, CLK_1, CLK_2, CLK_EN, OUT1 );
    input TST, DATA,
    inout CLK_1,
    input CLK_2, CLK_EN, RESET, RESET_EN;

    output OUT1;

    wire mux_to_res;

    assign mux_to_res = ( TST ) ? ( ( CLK_1 & CLK_EN ) | ( CLK_2 & CLK_EN ) )
                                : ( RESET & RESET_EN );

    dff DFF_INSTANCE ( .D( DATA ), .CLK( CLK_1 ), .RESET( mux_to_res  ), .Q( OUT1 ) );


endmodule

module dff( D, CLK, RESET, Q );
    input  D, CLK, RESET;

    output reg Q;

    always @( posedge CLK or posedge RESET )

            if ( RESET )
                Q <= 1'b0;
            else
                Q <= D;

endmodule
```

Global clock signal "top.CLK_1" is connected to asynchronous control line of FF "Q".

EXAMPLE 2: [1] consider design hierarchy that is represented on the picture below
[2] clock "top.CLK_1" is specified with **-alint_gclk** switch (**alint *-alint_gclk*** top.CLK_1 ***-asim*** top)



top
modwithinout
DATA_1
CLK_1
DATA_2
CLK_2
FF
FF
OUT1
Global clock signal "{GlCLKName}" is connected to bi-directional port " {PortName}" of "{ObjectName}"

```
module top( DATA_1, DATA_2, CLK_1, CLK_2, OUT1 );
    input  DATA_1, DATA_2, CLK_1, CLK_2;

    output OUT1;

    modwithinout
     MOD_INST(.DATA_1(DATA_1),.DATA_2(DATA_2),.CLK_1(CLK_1),.CLK_2(CLK_2),.OUT1(OUT1));

endmodule
```

Global clock signal "top.CLK_1" is connected to bi-directional port "CLK_1" of "MOD_INST"

```
module modwithinout( DATA_1, DATA_2, CLK_1, CLK_2, OUT1 );

    input  DATA_1, DATA_2;
    inout  CLK_1;
    input  CLK_2;

    output OUT1;
    reg    Q1, Q2;

    assign OUT1 = Q1;

    assign CLK_1 = Q2;

    always @( posedge CLK_1 )

        Q1 <= DATA_1;

    always @( posedge CLK_2 )

        Q2 <= DATA_2;

endmodule
```

STARC_VLOG 1.4.3.6

RULE NAME	**Do not use FFs with inverted edges**	
MESSAGE-1	**Module "{ModuleName}" uses clock signal with posedge for {PosedgeCLKFFCount} FF signal(s) and negedge for {NegedgeCLKFFCount} FF signal(s). Inverted edge is: {InvertedEdge}. Do not use FFs with inverted edges.**	
	DETAIL-1	*Process infers flip-flops with inverted edge on a clock signal for {FFCount} FF signal(s)*
MESSAGE-2	**Module "{ModuleName}" uses clock signal "{CLKName}" with posedge for {PosedgeCLKFFCount} FF signal(s) and negedge for {NegedgeCLKFFCount} FF signal(s). Inverted edge is: {InvertedEdge}. Do not use FFs with inverted edges.**	
	DETAIL-1	*Process infers flip-flops with inverted edge on a clock signal for {FFCount} FF signal(s)*
PROBLEM DESCRIPTION	Depending on the ASIC library used, there may be two types of FFs: those that work on a positive clock edge and those that work on a negative clock edge. When the two types of FFs are mixed in a circuit, scan register insertion becomes problematic. It is best not to use FFs that work on an inverted clock.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker collects clock signals (by extracting them from a flip-flops inferred in the current module): – if parameter CHECK_CLOCK_DOMAINS is disabled (OFF) => all clock signals should use the same edge: – if edges are inverted for any of clock signals => violation (message-1) – if parameter CHECK_CLOCK_DOMAINS is enabled (ON) => each clock signal must use the same edge: – if some of clock signals uses inverted edges => violation (message-2) Note: field {InvertedEdge} in the warning message is defined by the less used clock edge (if number of positive and negative edges is equal, negedge is treated as inverted)	

EXAMPLE-1: [1] multiple (2) clocks are used in the module, these clocks have different edges and parameter CHECK_CLOCK_DOMAINS is disabled => violation (message-1);

[2] note: 'posedge' is inverted edge here (it is less used)

```
module sl_dec( ... )
    always @( negedge CLK_2 )
        read_state <= read_finish;

    always @( posedge CLK_1 )
        if( RESET )
            return_slot <= P_INITIAL;
        else
            return_slot <= ssm_sto_slot;

    always @( negedge CLK_2 )
        end_byte <= ds_end_byte;
endmodule
```

Module "sl_dec" uses clock signal w ith posedge for 1 FF signal(s) and negedge for 2 FF signals(s). Inverted edge is: posedge. Do not use FFs w ith inverted edges.

Process infers flip-flops w ith inverted edge on a clock signal for 1 FF signal(s)

EXAMPLE-2: [1] multiple (2) clocks are used in the module, these clocks have different edges and parameter CHECK_CLOCK_DOMAINS is enabled => violation (message-2, only per clock with different edges);

[2] note: 'negedge' is inverted edge here (number of positive and negative edges is equal);

```
module sl_dec( ... )
    always @( posedge CLK_1 )
        if( RESET )
            return_slot <= P_INITIAL;
```

Module "sl_dec" uses clock signal "CLK_2" w ith posedge in 1 FF(s) and negedge in 1 FF(s). Inverted edge is: negedge. Do not use FFs w ith inverted edges.

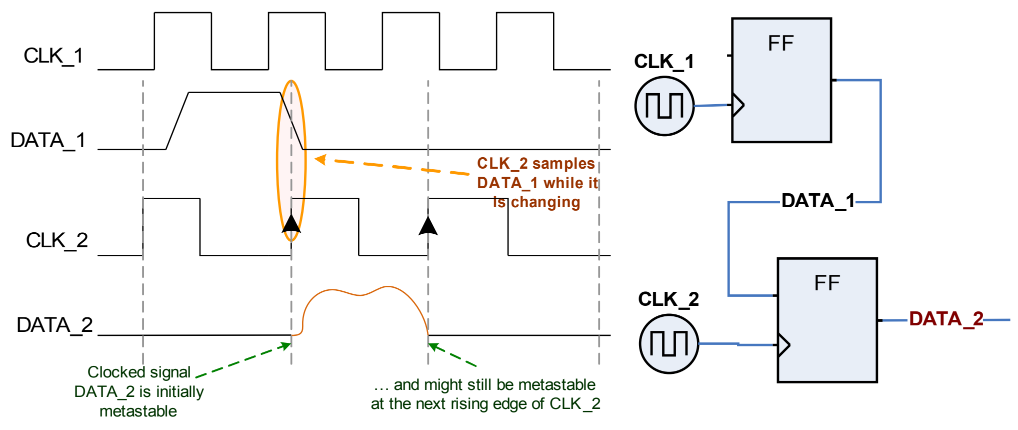
```
        else
            return_slot <= ssm_sto_slot;

    always @( posedge CLK_1 )
        start_byte <= ds_start_byte;

    always @( negedge CLK_2 ) begin
        read_state <= read_finish;
        write_state <= write_setup;
    end

    always @( posedge CLK_2 )
        end_byte <= ds_end_byte;
endmodule
```

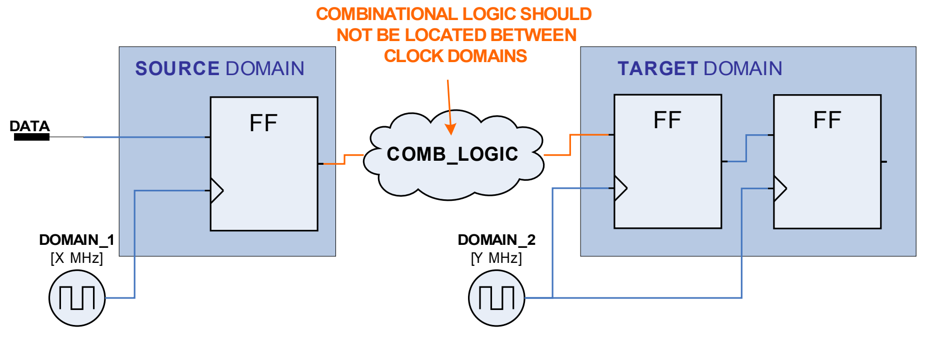
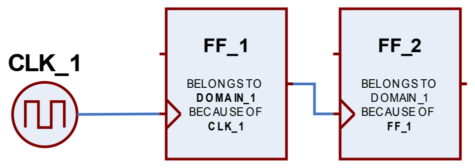
Process infers flip-flops w ith inverted edge on a clock signal for 2 FF signal(s)

1.5 *Handling of asynchronous circuits*

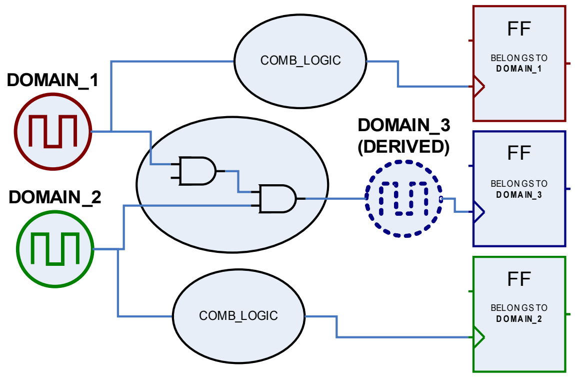
1.5.1 Consider metastable issues in signals between asynchronous clocks

STARC_VLOG 1.5.1.1

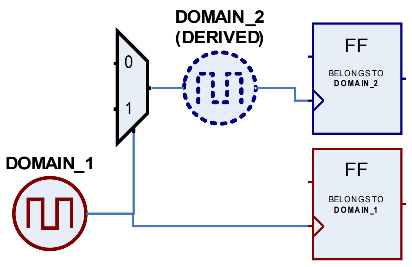
RULE NAME	**To avoid metastable conditions, do not locate combinational logic between asynchronous clock domains**	
MESSAGE	**FF "{FFName}" obtains data from another clock domain through combinational logic. Combinational logic should not be located between asynchronous clock domains, because it significantly increases the risk to propagate incorrect value to downstream logic and cause functional errors.**	
	DETAIL-1	*"{ClockName}" is the origin clock of the source domain.*
	DETAIL-2	*"{ClockName}" is the origin clock of the target domain.*
	DETAIL-3	*Signal "{SignalName}" derived from global clock(s) with circuit "{Circuit}" is the origin clock of the source domain.*
	DETAIL-4	*Signal "{SignalName}" derived from global clock(s) with circuit "{Circuit}" is the origin clock of the target domain.*
PROBLEM DESCRIPTION	Metastability problem should be considered carefully during data transmission between different clock domains. The picture below illustrates that input signal value could be changed during a flip-flop setup/hold time – metastable value (glitch) could be sampled and propagated from a flip-flop to the logic circuits.  CLK_1 DATA_1 CLK_2 DATA_2 CLK_2 samples DATA_1 while it is changing Clocked signal DATA_2 is initially metastable … and might still be metastable at the next rising edge of CLK_2 FF CLK_1 DATA_1 FF CLK_2 DATA_2 It is quite difficult to detect such kind of problems before circuit is fabricated. Therefore, it is highly important to prevent metastability-related issues by enforcing effectiveness-proven methodologies during the RTL design. To understand the rule 1.5.1.1, consider simple synchronizer. A simple synchronizer comprises two flip-flops in series without any combinational circuitry between them. This approach ensures that the first flip-flop exits its metastable state and its output settles before the second flip-flop samples it. For proper work of such synchronization, the signal crossing a clock domain should pass from flip-flop in the original clock domain to the first flip-flop of the synchronizer without passing through any combinational logic. First flip-flop of a synchronizer is sensitive to glitches that combinational logic produces (glitch that occurs at the correct time could meet the setup-and-hold requirements of the first flip-flop in	

RULE NAME	**To avoid metastable conditions, do not locate combinational logic between asynchronous clock domains**	
	the synchronizer, causing the synchronizer to pass a pseudo-valid signal to the rest of the logic in the target clock domain). Therefore, combination logic should not be located between asynchronous clock domains, because it significantly increases the risk to propagate pseudo-valid value to downstream logic.  COMBINATIONAL LOGIC SHOULD NOT BE LOCATED BETWEEN CLOCK DOMAINS SOURCE DOMAIN TARGET DOMAIN DATA FF COMB_LOGIC FF FF DOMAIN_1 [X MHz] DOMAIN_2 [Y MHz]	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans design hierarchy against combinational logic between different asynchronous ***clock domains*** *(*)*: – those flip-flops that have data input driven by combinational logic are validated: – violation is issued if another clock domain drives this combinational logic – ***(*)*** *Clock domains are auto-detected – the following algorithm defines clock domains extracting principles:* – *global clocks are detected (external port(s) that is/are directly connected to FF clock input – see 1.4.3.4 for details)* – *each global clock creates separate clock domain:* – *global clock is propagated through design hierarchy by common principles (through combinational logic and multiplexers – see 1.4.3.4 for details about clock propagation)* – ***note**: feedbacks on clock signal path are not considered (no backward propagation is performed);* – ***note**: latches on clock signal propagation path are transparent* – *each FF which has global clock signal connected to the clock pin is added to the corresponding clock domain;* – ***note**: if the FF clock input is driven by the output of a FF/latch/tristate that belongs to a clock domain "A", this FF is correspondingly added to a clock domain "A";*  CLK_1 FF_1 BELONGS TO DOMAIN_1 BECAUSE OF CLK_1 FF_2 BELONGS TO DOMAIN_1 BECAUSE OF FF_1 – *if two or more clock signals are propagated through the same combinational logic or multiplexer (clock domains crossing) then output of this logic / mux derives new clock signal – resulting in new clock domain for subsequent connections;*	

RULE NAME	**To avoid metastable conditions, do not locate combinational logic between asynchronous clock domains**



– *if clock signal is connected to the select pin of multiplexer then its output derives new clock signal – resulting in new clock domain for subsequent connections;*



– ***note**: after splitting process is over:*
  – *if there is no clock domain detected, then whole design is considered to be in single (**default**) clock domain – CDC-related checks are not performed on whole design;*
  – *if there is at least one clock domain detected and some FF(s) still do not belong to any clock domain, then this/these FF(s) is/are considered to be in **default** domain – CDC-related checks are not performed on this/these FF(s);*

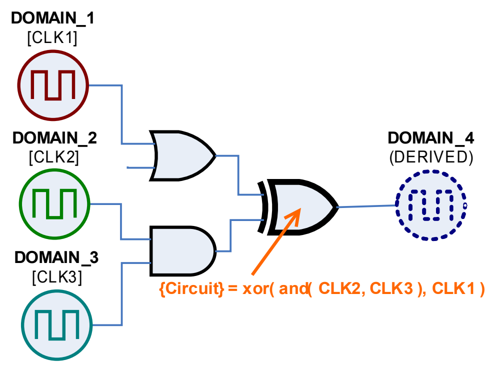
Note-1: source points for violation messages (see "problem description" for explanation of terms "source" and "target"):

– main message points to the target FF;
– detail-1/2 – points to clock declaration of source/target domain (global clock(s));
– detail-3/4 – points to clock *crossing (**)* of source/target domain (derived clock(s));
  – ***(**)***: token {Circuit} indicates elementary circuit that derives new clock signal; the following four alternatives are possible:

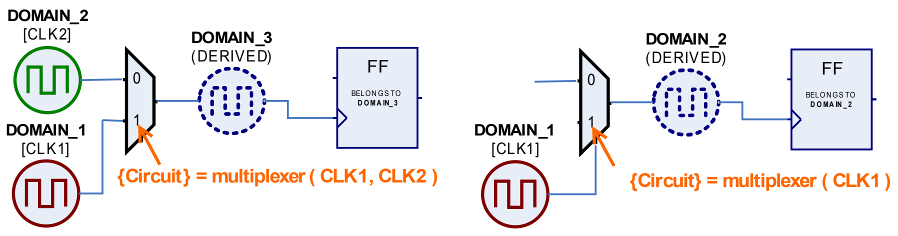
{Circuit}	
1	gate_type (g_clk1 | gate_type (...), ...)
2	multiplexer (g_clk1, ...)
3	arithmetic_type (g_clk1 | arithmetic_type, ...) (adder, subtractor, multiplier, divider, left/right shifter)
4	complex logic (g_clk1, ...)

  – consider examples (numbers correspond to alternatives from the table above):
    1. clock signals are crossed within **combinational logic** (length of such "chain" should not exceed 2, otherwise it will be considered as complex logic – see example-4 below):

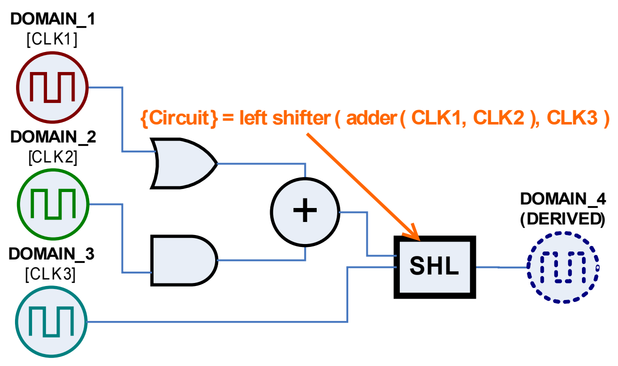
RULE NAME	**To avoid metastable conditions, do not locate combinational logic between asynchronous clock domains**



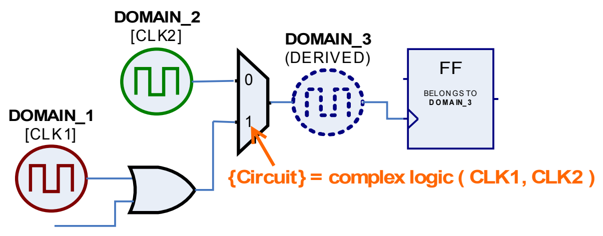
2. clock signals are crossed within **multiplexer**:

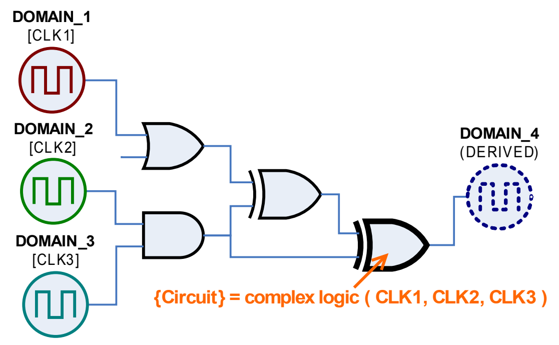


3. clock signals are crossed in **arithmetic circuit** (length of such "chain" should not exceed 2, otherwise it will be considered as complex logic – see example-4 below):



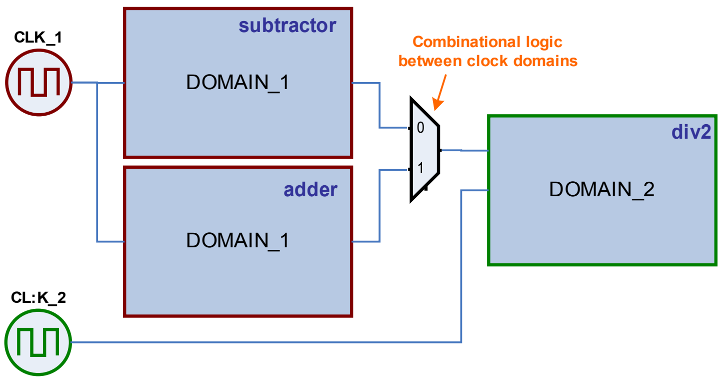
4. all other cases are considered to be **complex logic**:



RULE NAME	To avoid metastable conditions, do not locate combinational logic between asynchronous clock domains
	 DOMAIN_1 [CLK1] DOMAIN_2 [CLK2] DOMAIN_3 [CLK3] DOMAIN_4 (DERIVED) {Circuit} = complex logic (CLK1, CLK2, CLK3)

EXAMPLE-1: [1] consider sample circuit at the picture below – there are two clock domains (driven by global clocks 'CLK_1' and 'CLK_2';

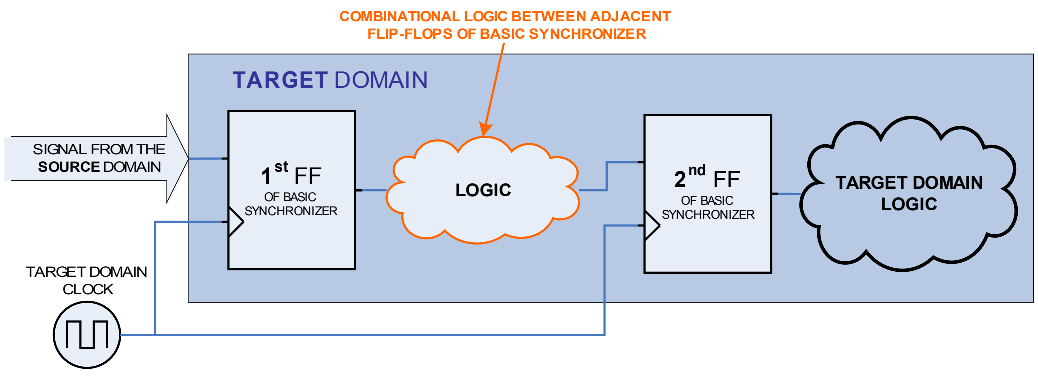
[2] combinational logic (multiplexer) is located between these domains => violation;



```
// alint -alint_gclk add_sub.CLK_1 -alint_gclk add_sub.CLK_2

                                "add_sub.CLK_1" is the origin clock of the source domain.
`define BIT_LENGTH 1
                                "add_sub.CLK_2" is the origin clock of the target domain.
module add_sub( CLK_1, CLK_2, ADD_SUB, ARG_A, ARG_B, RES );

    input                       CLK_1, CLK_2;
    input                       ADD_SUB;
    input    [`BIT_LENGTH-1:0]  ARG_A, ARG_B;

    output   [`BIT_LENGTH-1:0]  RES;

    wire     [`BIT_LENGTH-1:0]  res_add, res_sub, trnsmt;


    adder adder(
        .CLK( CLK_1   ),
        .A  ( ARG_A   ),
        .B  ( ARG_B   ),
        .RES( res_add )
    );

    subtractor subtractor(
        .CLK( CLK_1   ),
        .A  ( ARG_A   ),
        .B  ( ARG_B   ),
        .RES( res_sub )
    );
    div2 div2(
        .CLK( CLK_2  ),
        .ARG( trnsmt ),
        .RES( RES    )
    );

    assign trnsmt = ( ADD_SUB )? res_add : res_sub;

endmodule

module adder( CLK, A, B, RES );

    input                    CLK;
    input  [`BIT_LENGTH-1:0] A, B;
    output [`BIT_LENGTH-1:0] RES;
    reg    [`BIT_LENGTH-1:0] RES;

    always @( posedge CLK )
        RES = A + B;

endmodule

module subtractor( CLK, A, B, RES );

    input                    CLK;
    input  [`BIT_LENGTH-1:0] A, B;
    output [`BIT_LENGTH-1:0] RES;
    reg    [`BIT_LENGTH-1:0] RES;

    always @( posedge CLK )
        RES = A - B;

endmodule

module div2( CLK, ARG, RES );

    input                    CLK;
    input  [`BIT_LENGTH-1:0] ARG;
    output [`BIT_LENGTH-1:0] RES;
    reg    [`BIT_LENGTH-1:0] RES;

    wire   [`BIT_LENGTH-1:0] stable_arg;


    simple_sync simple_sync(
        .CLK     ( CLK        ),
        .DATA_IN ( ARG        ),
        .DATA_OUT( stable_arg )
    );

    always @( posedge CLK )
        RES = stable_arg >> 1;

endmodule

module simple_sync( CLK, DATA_IN, DATA_OUT );

    input                    CLK;
    input  [`BIT_LENGTH-1:0] DATA_IN;
    output [`BIT_LENGTH-1:0] DATA_OUT;
    reg    [`BIT_LENGTH-1:0] DATA_OUT;

    reg    [`BIT_LENGTH-1:0] first_stage_ff;

    always @( posedge CLK )
        first_stage_ff = DATA_IN;

    always @( posedge CLK )
        DATA_OUT = first_stage_ff;

endmodule
```

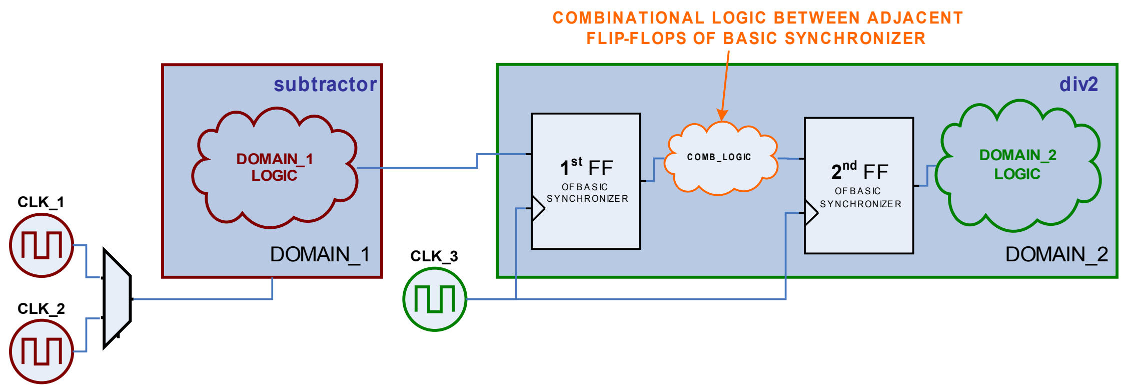
Instance "add_sub.div2.simple_sync". FF "first_stage_ff" obtains data from another clock domain through combinational logic. Combinational logic should not be located between asynchronous clock domains, because it significantly increases the risk to propagate incorrect value to downstream logic and cause functional errors.

STARC_VLOG 1.5.1.2

RULE NAME	**To avoid metastable conditions, do not locate combinational logic between first-stage FF and the next synchronizing FF.**
MESSAGE	**Combinational logic is detected between the adjacent flip-flops of basic synchronizer "{FirstFFName}"-"{SecondFFName}". Glitches produced by combinational logic increase the risk to propagate incorrect value to downstream logic. For proper synchronization, the basic synchronizer cell should contain two pure flip-flops.**
PROBLEM DESCRIPTION	Basic approach to solving a metastable problem (metastability is explained in *1.5.1.1*) is based on simple synchronizer comprising two flip-flops. If combinational logic is located between these two synchronizing flip-flops, the second flip-flop becomes sensitive to glitches produced by combinational logic – a possibility to propagate pseudo-valid value to downstream logic increases significantly and synchronizer could become useless:  COMBINATIONAL LOGIC BETWEEN ADJACENT FLIP-FLOPS OF BASIC SYNCHRONIZER TARGET DOMAIN SIGNAL FROM THE SOURCE DOMAIN 1st FF OF BASIC SYNCHRONIZER LOGIC 2nd FF OF BASIC SYNCHRONIZER TARGET DOMAIN LOGIC TARGET DOMAIN CLOCK Thus, it is not recommended to locate combinational logic between the adjacent flip-flops of basic synchronizer.
LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans design hierarchy against combinational logic between adjacent flip-flops of each basic synchronizer: – each ***target clock domain*** *(*)* is considered: – the first two flip-flops are treated as basic synchronizer; – the violation is issued if there is a circuitry between them (see figure above); – **note**: violation message: – the main violation message points to the second flip-flop; – special detail messages are also displayed in order to indicate origin clocks of source and target domains (see 1.5.1.1 for details); – *(*) see 1.5.1.1 for details about clock domains detection;*

EXAMPLE-1: [1] consider sample circuit at the picture below – there are two clock domains (source domain is driven by derived clock 'multiplexer(CLK_1, CLK_2)', target domain is driven by global clock 'CLK_3');

[2] combinational logic is located between the adjacent flip-flops of simple synchronizer in the target domain => violation;


COMBINATIONAL LOGIC BETWEEN ADJACENT FLIP-FLOPS OF BASIC SYNCHRONIZER
subtractor
DOMAIN_1 LOGIC
DOMAIN_1
div2
1st FF OF BASIC SYNCHRONIZER
COMB_LOGIC
2nd FF OF BASIC SYNCHRONIZER
DOMAIN_2 LOGIC
DOMAIN_2
CLK_1
CLK_2
CLK_3

```
// alint -alint_gclk add_sub.CLK_1 -alint_gclk add_sub.CLK_2 -alint_gclk add_sub.CLK_3

`define BIT_LENGTH 1

module sub( CLK_1, CLK_2, CLK_3, CLK_SEL, ARG_A, ARG_B, RES );

    input                     CLK_1, CLK_2, CLK_SEL;
    input                     CLK_3;
    input  [`BIT_LENGTH-1:0] ARG_A, ARG_B;

    output [`BIT_LENGTH-1:0] RES;

    wire   [`BIT_LENGTH-1:0] res_sub;
    wire                      cur_clk;

    assign cur_clk = ( CLK_SEL )? CLK_1 : CLK_2;

    subtracter subtracter(
        .CLK( cur_clk ),
        .A  ( ARG_A   ),
        .B  ( ARG_B   ),
        .RES( res_sub )
    );

    div2 div2(
        .CLK( CLK_3   ),
        .ARG( res_sub ),
        .RES( RES     )
    );

endmodule


//Subtracter
module subtracter( CLK, A, B, RES );

    input                     CLK;
    input  [`BIT_LENGTH-1:0] A, B;
    output [`BIT_LENGTH-1:0] RES;
    reg    [`BIT_LENGTH-1:0] RES;

    always @( posedge CLK )
        RES = A - B;

endmodule


//Divider
module div2( CLK, ARG, RES );

    input                     CLK;
    input  [`BIT_LENGTH-1:0] ARG;
    output [`BIT_LENGTH-1:0] RES;
    reg    [`BIT_LENGTH-1:0] RES;

    wire   [`BIT_LENGTH-1:0] stable_arg;
```

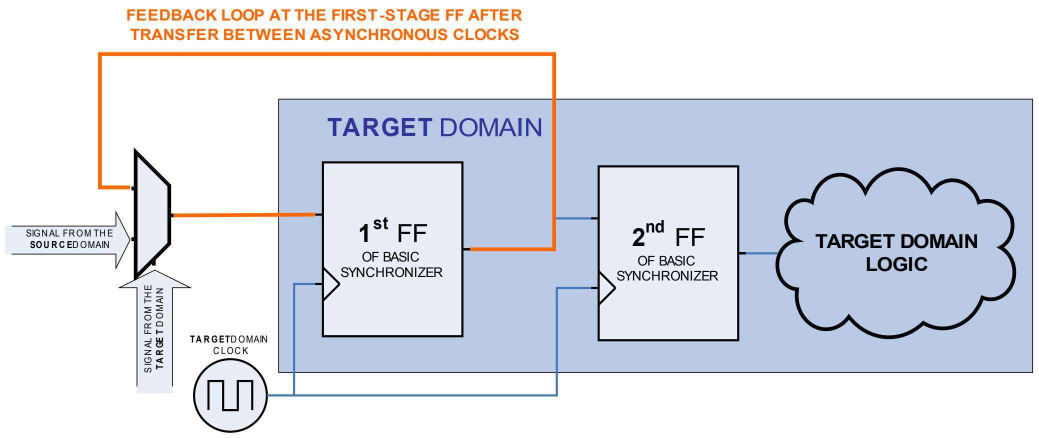
"sub.CLK_3" is the origin clock of the target domain.

Signal “sub.cur_clk” derived from global clock(s) w ith circuit “multiplexer(sub.CLK_1, sub.CLK_2)” is the origin clock of the source domain.

```
    simple_sync simple_sync(
        .CLK      ( CLK        ),
        .DATA_IN  ( ARG        ),
        .DATA_OUT( stable_arg )
    );

    always @( posedge CLK )
        RES = stable_arg >> 1;

endmodule


//Synchronizer
module simple_sync( CLK, DATA_IN, DATA_OUT );

    input                    CLK;
    input  [`BIT_LENGTH-1:0] DATA_IN;
    output [`BIT_LENGTH-1:0] DATA_OUT;
    reg    [`BIT_LENGTH-1:0] DATA_OUT;

    reg    [`BIT_LENGTH-1:0] first_stage_ff;
    reg    [`BIT_LENGTH-1:0] second_stage_ff;
    wire   [`BIT_LENGTH-1:0] custom_gate;

    assign custom_gate = first_stage_ff & (~CLK);

    assign DATA_OUT = second_stage_ff;

    always @( posedge CLK )
        first_stage_ff = DATA_IN;

    always @( posedge CLK )
        second_stage_ff = custom_gate;

endmodule
```

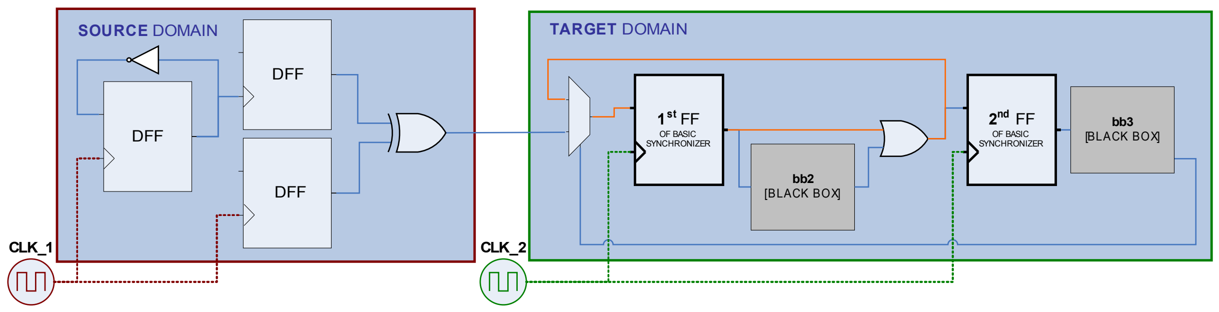
Instance "sub.div2.simple_sync". Combinational logic is detected between the adjacent flip-flops of basic synchronizer "sub.div2.simple_sync.first_stage_ff"-"sub.div2.simple_sync.second_stage_ff". Glitches produced by combinational logic increase the risk to propagate incorrect value to downstream logic. For proper synchronization, the basic synchronizer cell should contain two pure flip-flops.

STARC_VLOG 1.5.1.3

RULE NAME	**Do not have a feedback loop at the first-stage FF after transfers between asynchronous clocks**
MESSAGE	**Feedback loop is detected in the first-stage of basic synchronizer "{FirstFFName}"-"{SecondFFName}". The metastable state on the synchronizer input might affect the latched value and result in circuit malfunction. Even if the placement of logic between the adjacent flip-flops of synchronizer can not be avoided, feedback should not be used.**
PROBLEM DESCRIPTION	 FEEDBACK LOOP AT THE FIRST-STAGE FF AFTER TRANSFER BETWEEN ASYNCHRONOUS CLOCKS TARGET DOMAIN 1st FF OF BASIC SYNCHRONIZER 2nd FF OF BASIC SYNCHRONIZER TARGET DOMAIN LOGIC SIGNAL FROM THE SOURCE DOMAIN SIGNAL FROM THE TARGET DOMAIN TARGET DOMAIN CLOCK Basic approach to solving a metastable problem (metastability is explained in *1.5.1.1*) is based on simple synchronizer containing two flip-flops. If operating frequencies of source and target clock domains are low, it is not an issue if some logic is located between adjacent flip-flops of basic synchronizer. However, there will be problems if there is a feedback to the flip-flop to latch data – the metastable state might affect the latched value and result in circuit malfunction. Thus, even if the placement of some logic between the adjacent flip-flops of synchronizer can not be avoided, feedback should not be used.
	LEVEL: RULE
CHECKER BEHAVIOR	Checker scans design hierarchy against a feedback loop at the first-stage of each basic synchronizer: – each ***target clock domain*** *(*)* is considered: – the first two flip-flops are treated as basic synchronizer; – the violation is issued if there is a feedback at the first of them (see figure above); – **note**: violation message: – the main message points to the line at which the feedback is detected; – special detail messages are also displayed in order to indicate origin clocks of source and target domains (see *1.5.1.1* for details); – additional detail messages could be also displayed in order to indicate the feedback propagation path (depends on value of parameter DETAILED_PROPAGATION_CHAIN – see *1.2.1.3* for details); – *(*) see 1.5.1.1 for details about clock domains detection;*

EXAMPLE-1: [1] consider sample circuit at the picture below – there are two clock domains (source domain is driven by clock 'CLK_1', target domain is driven by clock 'CLK_2';

[2] there is a feedback loop at the first flip-flop of basic synchronizer – right after the transfer between asynchronous clocks => violation;



```
// alint -alint_gclk top.CLK_1 -alint_gclk top.CLK_2 -alint_blackbox bb2 -alint_blackbox bb3

module top( CLK_1, CLK_2, RST, DI1, DI2, DO );

    input   CLK_1, CLK_2, RST;
    input   DI1, DI2;
    output  DO;

    reg     clk_div2, sync_1st, sync_2nd, samp_di1, samp_di2;
    wire    data_src, data_cd1;

    bb2 BlackBox2( sync_1st, proc_bb2 );
    bb3 BlackBox3( sync_2nd, DO, contr_bb3 );

    assign data_src = ( contr_bb3 )? data_cd1 : sync_1st | proc_bb2;
    assign data_cd1 = samp_di1 ^ samp_di2;

    always @( posedge CLK_2 )
        sync_1st = data_src;

    always @( posedge CLK_2 )
        sync_2nd = sync_1st;

    always @( posedge CLK_1 )
        if( RST )
            clk_div2 = 0;
        else
            clk_div2 = ~clk_div2;

    always @( posedge clk_div2 )
        if( RST )
            samp_di1 = 0;
        else
            samp_di1 = DI1;

    always @( posedge CLK_1 )
        if( RST )
            samp_di2 = 0;
        else
            samp_di2 = DI2;

endmodule

module bb2( PIN, POUT ) /* black box */;
    input  PIN;
    output POUT;
endmodule

module bb3( PIN, POUT_1, POUT_2 ) /* black box */;
    input  PIN;
    output POUT_1, POUT_2;
endmodule
```

"top.CLK_1" is the origin clock of the source domain.

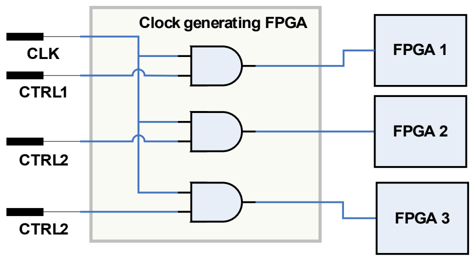
"top.CLK_2" is the origin clock of the target domain.

Instance "top". Feedback loop is detected in the first-stage of basic synchronizer "top.sync_1st"-"top.sync_2nd". The metastable state on the synchronizer input might affect the latched value and result in circuit malfunction. Even if the placement of logic betw een the adjacent flip-flops of synchronizer can not be avoided, feedback should not be used.

1.7 FPGAs

1.7.1 Considerations for using both ASICs and FPGAs

STARC_VLOG 1.7.1.1

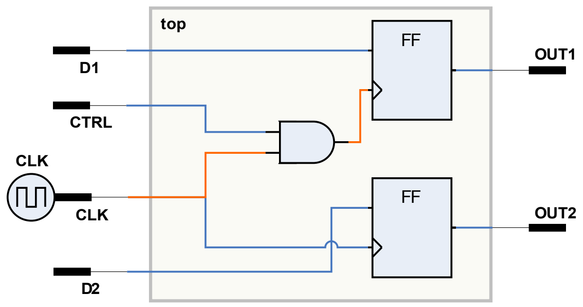
RULE NAME	**Don't use gated clocks in an FPGA**
MESSAGE	**Global clock "{GClkName}" passes through the gate logic. Gated clocks should not be used in FPGAs wherever possible, because it becomes difficult to fine-tune clock delays and skew.**
PROBLEM DESCRIPTION	Using gated clocks is a popular way to decrease circuit power consumption by reducing the clock network power dissipation. This mechanisms is often used for ASIC design, but there are some problems of implementing gated clocks for FPGA design. FPGAs have pre-synthesized clock trees for providing synchronized clock to the finite number of flip-flops and memories across the chip. When additional logic is applied to the clock signal in the RTL, the logic translates into physical gates outside the pre-synthesized balanced clock tree, through which the clock signal passes, thus causing large clock skews between registers of theFPGA. Gated clocks should not be used in FPGAs wherever possible. If you still want to use gated clocks, clock gating logic should be put in a separate FPGA and separate FPGAs should be used for each clock domain (see the picture below).  Clock generating FPGA CLK CTRL1 CTRL2 CTRL2 FPGA 1 FPGA 2 FPGA 3
LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOUR	Checker collects signals that are detected as global clocks (auto-detected or specified with *-alint_gclk* switch): – if the signal from the set is connected directly (*) to some gate – if the output of the gate is connected directly (*) to FF clock pin => violation – *(*) for this rule, signal that is **directly connected** may be connected directly or through buffers/inverters*

EXAMPLE-1: [1] consider the picture below;
[2] global clock CLK (auto-detect mode is used) is directly connected to AND gate;
[3] output of the gate is directly connected to FF clock pin => violation.

```
module top( CLK, CTRL, D1, D2, OUT1, OUT2 );

    input CLK;
    input CTRL, D1, D2;

    output  OUT1, OUT2;

    assign and_clk = CTRL & CLK;

    dff DFF1 ( .CLK( and_clk ), .D( D1   ), .Q( OUT1 ) );
```

Instance "top". Global clock "CLK" passes through the gate logic. Gated clocks should not be used in FPGAs w herever possible, because it becomes difficult to fine-tune clock delays and skew .

```
    dff DFF2 ( .CLK( CLK    ), .D( D2   ), .Q( OUT2 ) );

endmodule

//D flop-flop
module dff( CLK, D, Q );

     input CLK, D;
     output reg Q;

    always @(posedge CLK)
        Q <= D;

endmodule
```



Chapter 2 RTL Description Techniques

2.1 Combinational logic

2.1.1 Use always constructs and function statements correctly

STARC_VLOG 2.1.1.2

RULE NAME	**Describe every case statement expressions in a function statement**	
MESSAGE-1	**Function "{FuncName}" is not defined in all cases. The results of RTL and post-synthesis simulation will not match.**	
	DETAIL-1	*"{Variable}" is not defined in all cases.*
	DETAIL-2	*Function result is assigned with "{Variable}" which is not defined in all cases.*
MESSAGE-2	**"{FuncBit}" bit of function result is not defined in all cases. The results of RTL and post-synthesis simulation will not match.**	
	DETAIL-1	*"{Variable}" is not defined in all cases.*
	DETAIL-2	*Function result is assigned with "{Variable}" which is not defined in all cases.*
MESSAGE-3	**{BitCount} bits of function "{FuncName}" are not defined in all cases. The results of RTL and post-synthesis simulation will not match.**	
	DETAIL-1	*"{Variable}" is not defined in all cases.*
	DETAIL-2	*Function result depends on a value of "{Variable}", which is not defined in all cases.*
PROBLEM DESCRIPTION	The function statement is a sub-program that can return only one value. In hardware it is implemented as a combinational circuit. In function statements, latches are not generated even if the conditions are not completely defined. In such situation, function returns an undefined value during simulation, but synthesis tool will assign the value of 0 or 1. Therefore, it is required to describe full 'case'/'if' statements (value of each signal is defined in each branch) in functions. Otherwise, there is a danger of mismatches between the RTL and post-synthesis simulation results.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans function statements: – if function output is not defined (return value is not assigned) in any possible case => violation – if function result is a scalar or the whole vector violates the rule => message-1 – if function result is a vector and only one bit violates the rule => message-2 – if function result is a vector and only some bits (>1, but not all) violate the rule => message-3	

RULE NAME	**Describe every case statement expressions in a function statement**
	Note-1: intermediate variables are taken into account (if some variable is not completely assigned in some conditional statement, and later is being assigned to function return value, then function is treated as not defined in all cases) Note-2: rule is parameter-dependent (elaboration-time checking is required)

EXAMPLE-1: [1] 'case' statement is described in the 'function' statement;

[2] function output is assigned in all branches but case is not full (not all possible branches are described and 'default' clause is not specified)=> violation (message-1);

```
function [1:0] res;
input [2:0] sel;
    begin
        case ( sel )
            3'b011: res = 2'b00;
            3'b100: res = 2'b01;
            3'b101: res = 2'b10;
        endcase
    end
endfunction
```

Function “res” is not defined in all cases. The results of RTL and post-synthesis simulation w ill not match.

“res” is not defined in all cases.

EXAMPLE-2: [1] 'if' statement is described in the 'function' statement;

[2] intermediate variable is used to assign function output;

[3] intermediate variable is not assigned in all branches of 'if' statement;

[4] only one bit of vector function output is assigned => violation (message-2)

```
function [1:0] res;
input [2:0] sel;
input [2:0] data;
reg c;
reg d;
    begin
        if (sel) c = data[0]
        else d = data[1];
            res = c;
    end
endfunction
```

“res[1]” bit of function result is not defined in all cases. The results of RTL and post-synthesis simulation w ill not match.

Function result depends on a value of “c”, w hich is not defined in all cases.

2.1.2 Define combinational circuits using the *function statement*

STARC_VLOG 2.1.2.1

RULE NAME	**The function statement shouldn't be used for asynchronous reset line logic in an always construct for FF inference**
MESSAGE	**The function statement should not be used for asynchronous reset line logic in an 'always' construct for FF inference.**
PROBLEM DESCRIPTION	Some synthesis tools cannot recognize the type of reset signal (posedge or negedge) if function call is used instead of simple expression in the asynchronous control condition branch.
LEVEL	RULE
CHECKER BEHAVIOR	Checker scans 'always' statements inferring flip-flops with an asynchronous control: – condition of the 'if' statement for asynchronous control is analyzed – if function is called here => violation Note: see checker 2.3.1.6 for asynchronous control description

EXAMPLE-1: [1] function is called in the asynchronous control line => violation;
[2] function is called in the clock enable line => no violation

```
input PRESET_DATA;
...
function invert;
    input arg;
begin
    invert = ~arg;
end
endfunction
...
always @( posedge CLK or negedge RESET ) begin
    if( invert( RESET ) )
        Q <= PRESET_DATA;
    else if( invert( EN ) )
        Q <= DATA;
end
```

The function statement should not be used for asynchronous reset line logic in an 'alw ays' construct for FF inference.

EXAMPLE-2: [1] function is called in the synchronous control line => no violation

```
input PRESET_DATA;
...
function invert;
    input arg;
begin
    invert = ~arg;
end
endfunction
...
always @( posedge CLK ) begin
    if( invert( RESET ) )
        Q <= 1'b0;
    else if( invert( EN ) )
        Q <= DATA;
end
```

STARC_VLOG 2.1.2.2

RULE NAME	**A non-blocking assignment (<=) should not be used in function statements (Verilog only)**	
MESSAGE	**Function contains {NBA_Count} non-blocking assignment statement(s). The results of RTL and post-synthesis simulation may not match.**	
	DETAIL	*Non-blocking assignment inside the function*
PROBLEM DESCRIPTION	Usage of a non-blocking assignment for a function return value must not be performed. Descriptions using such assignments will not result in the expected behavior. During the RTL simulation, change of the return value occurs after whole right-hand-side of non-blocking assignment is evaluated whereas synthesis tool will generate a correct circuit. It causes mismatch between RTL and gate level simulation.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects non-blocking assignments inside the function statements.	

EXAMPLE-1: [1] function statement contains non-blocking assignments => violation

```
function ADD_SUB;
    input   [7:0] ARG_1;
    input   [7:0] ARG_2;
    input         OP_SEL;
    reg           RES;
begin
    if( OP_SEL == 1'b1 )
        RES <= ARG_1 + ARG_2;
    else
        RES <= ARG_1 - ARG_2;

    ADD_SUB = RES;
end
endfunction
```

Function contains 2 non-blocking assignment statement(s). The results of RTL and post-synthesis simulation may not match.

Non-blocking assignment inside the function

Non-blocking assignment inside the function

STARC_VLOG 2.1.2.3

RULE NAME	**All arguments are defined as function statement inputs**	
MESSAGE	**Function "{FunctionName}" uses {ObjectCount} signal(s) not declared as function inputs or local variables. The results of RTL and post-synthesis simulation may not match.**	
	DETAIL	*Signal "{ObjectName}" is not declared as function input or local variable*
PROBLEM DESCRIPTION	Description of the 'function' statement must declare inputs for all input signals used inside the 'function' statement. When using the signal which is not declared as input and a signal of the same name exists somewhere in the module, the global signal is taken. But function does not execute if the global signal value changes. Such description leads to mismatches between the RTL and gate level simulation.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects the signals that have ever been read in the following expressions (inside the function): – right-hand-side of an assignments – conditional expressions ('if', 'for', ternary conditional operator) – another function calls If the signal is read but it is defined neither as function input nor as local variable => violation Note: if signal is not local, but it is 'for' loop variable =>no violation	

EXAMPLE-1: [1] signal is read in the conditional expression of an 'if' statement and it is not defined as function input or local variable => violation

```
function ADD_SUB;
     input   [7:0] ARG_1;
     input   [7:0] ARG_2;
     reg           RES;
begin
     if( OP_SEL == 1'b1 )
         RES <= ARG_1 + ARG_2;
     else
         RES <= ARG_1 - ARG_2;

     ADD_SUB = RES;
end
endfunction
```

Function "ADD_SUB" uses 1 signal(s) not declared as function inputs or local variables. The results of RTL and post-synthesis simulation may not match.

Signal "OP_SEL" is not declared as function input or local variable

EXAMPLE-1: [1] if signal is not local, but it is 'for' loop variable => no violation

```
integer i;
function BW_ADD;
     input   [7:0] ARG_1;
     input   [7:0] ARG_2;
begin
     for ( i = 0; i <=7; i = i + 1 )
         BW_ADD <= ARG_1[i] + ARG_2[i];
end
endfunction
```

STARC_VLOG 2.1.2.4

RULE NAME	**Task statements should not be used (Verilog only)**	
MESSAGE	**Module "{ModuleName}" contains {TaskStatementsCount} 'task' enable statement(s). Usage of 'task' statements in RTL description is not recommended.**	
	DETAIL	*'task' enable statement detected*
PROBLEM DESCRIPTION	It is recommended not to use 'task' statements in the RTL descriptions because of two problems: – multiple outputs can be described in 'task' statements – values can be retained by 'reg' variables	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker detects task enable statements and if any are detected it issues one violation per module.	

EXAMPLE-1: [1] module "d32_ang" contains 2 task enable statements => violation

```
module d32_ang( ... );
     task swap;
          ...
     endtask

     always @( ... ) begin
          swap( SR, DR );
          swap( MR, LR );
     end

endmodule
```

Module "d32_ang" contains 2 'task' enable statement(s). Usage of 'task' statements in RTL description is not recommended.

'task' enable statement detected

'task' enable statement detected

STARC_VLOG 2.1.2.5

RULE NAME	**Clock edge descriptions should not be used in task statements (Verilog only)**	
MESSAGE	**Task "{TaskName}" contains {EventControlStatementCount} event control statement(s). Event control statements should not be used in 'task' statements.**	
	DETAIL	*Event control statement detected*
PROBLEM DESCRIPTION	Clock edge descriptions in 'task' statements cannot be synthesized by logic synthesis tools.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans body of a 'task' statement. If event control statement is present => violation.	

EXAMPLE-1: [1] task "light" contains an event control statement => violation

```
task light;
     output       COLOR;
     input [31:0] TICS;
begin
     repeat( TICS ) @( posedge CLOCK );
     COLOR = OFF;
end
endtask
```

Task "light" contains 1 event control statement(s). Event control statements should not be used in 'task' statements.

Event control statement detected

2.1.3 In a *function statement* , be careful to check arguments and bit width

STARC_VLOG 2.1.3.1

RULE NAME	**Match the argument bit width with the bit width of the function statement input declaration (Verilog only)**	
MESSAGE-1	**Argument bit width does not match to bit width of the function port declaration. Match bit widths exactly to avoid loss or misalign of input signal bit values.**	
	DETAIL-1	*Argument width is "{ArgBitWidth}" while port "{InputName}" width is "{InputBitWidth}"*
MESSAGE-2	**Argument bit width does not match to bit width of the task port declaration. Match bit widths exactly to avoid loss or misalign of input/output signal bit values.**	
	DETAIL-1	*Argument width is "{ArgBitWidth}" while port "{InputName}" width is "{InputBitWidth}"*
PROBLEM DESCRIPTION	When function statement is described, bit width of the input/output arguments must be considered with special care: the bit width of the function input declaration should match to bit width of the input argument. If bit widths doesn't match, value of input signal can be misaligned or lost.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies each function/task call: – each input/output should be mapped with signal/expression having the same bit width, *Note: for decimal constants violation is issued only if they are wider than arguments (narrower decimal constants are allowed);* – violation is issued if bit widths are different; messages are displayed according to the following principles: – message-1 is displayed for functions – message-2 is displayed for tasks Note-1: this rule can be dependent on parameters or hierarchical references => elaboration-time checking is required for such cases Note-2: bit widths of decimal constants is defined by their values (“1” – 1 bit (1'b1); “7” – 3 bits (2'b111)); sign bit is taken into account when unary sign (+ / -) is specified (“1“ or “+1” – 1 bit (1'b1); “-1” – 2 bits (2'11))	

EXAMPLE-1: [1] function "INV" with 1-bit input is described;
[2] function argument is a concatenation that results in 2-bit value => violation (message-1);

```
wire ST_SSM;
wire ST_SSK;
...
function INV;
    input arg;
begin
    INV = ~arg;
end
endfunction
...
assign rev_slot = INV( { ST_SSM, ST_SSK } );
```

Argument bit w idth does not match to bit w idth of the function port declaration. Match bit w idths exactly to avoid loss or misalign of input signal bit values.

Argument w idth is "2" w hile port "arg" w idth is "1"

EXAMPLE-2: [1] task "copy8to16" is described with 8-bit input and 16-bit output;

[2] input is assigned with signal having less bit width (4), output is assigned with signal having greater bit width (32) => violation (message-2, detail per each port);

```
reg [ 3:0] sample_k_bus;
reg [31:0] transn_k_bus;

...
task copy8to16;
     input  [ 7:0] src_arg;
     output [15:0] dest_arg;
begin
     dest_arg = { 8'b00000000, src_arg };
end
endtask
...
always @( * ) begin
     ...
     copy8to16( sample_k_bus, transn_k_bus );
     ...
end
```

Argument bit w idth does not match to bit w idth of the task port declaration. Match bit w idths exactly to avoid loss or misalign of input/output signal bit values.

Argument w idth is "32" w hile port "dest_arg" w idth is "16"

Argument w idth is "4" w hile port "src_arg" w idth is "8"

STARC_VLOG 2.1.3.2

RULE NAME	**Match the return value bit width with the bit width of the assignment destination signal (Verilog only)**
MESSAGE-1	**Function return value bit width "{RetValBitWidth}" is less than bit width "{DestBitWidth}" of the assignment destination. Upper bits of the assignment destination will be filled with zeroes. Match bit widths exactly.**
MESSAGE-2	**Function return value bit width "{RetValBitWidth}" is greater than bit width "{DestBitWidth}" of the assignment destination. Upper bits of the function return value will be truncated. Match bit widths exactly.**
PROBLEM DESCRIPTION	Function return value should match the bit width of assignment destination signal. When bit width of right-hand-side it greater than bit width of assignment destination => upper bits of right-hand-side are truncated. Otherwise, when bit width of right-hand-side is less than bit width of assignment destination => upper bits of destination are filled with zeros. Descriptions with different bit widths may be made inadvertently – they are implicit and readability of the description drops. Concatenations/part-selections should be used to describe filling/truncation explicitly.
LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies each assignment (=, <=, assign) where right-hand-side is pure function call: – bit width of function return value should be the same as bit width of assignment destination – message-1 is displayed when target of assignment has greater bit width than bit width of function return value – message-2 is displayed when target of assignment has less bit width than bit width of function return value Note-1: this rule can be dependent on parameters or hierarchical references => elaboration-time checking is required for such cases Note-2: this checker verifies only cases where right-hand-side is pure function call. This is made intentionally, since checker 2.10.3.3 verifies cases where right-hand-side is not simple function call (for example, an expression with function call as one of its members)

EXAMPLE-1: [1] function "SHL" is called in "true" branch of ternary conditional operator (it is pure call);
[2] bit width of function return value is greater than bit width of assignment destination => violation (message-2)

```
wire [7:0] TS_VAL;
wire [7:0] L_PIPE;
...
function [7:0] SHL;
    input [7:0] arg;
begin
    SHL = arg << 1;
end
endfunction
...
assign L_PIPE[5:0] = ( SEL )? SHL( TS_VAL ) : 6'b000001;
```

Function return value bit w idth "8" is greater than bit w idth "6" of the assignment destination. Upper bits of the function return value w ill be truncated. Match bit w idths exactly.

EXAMPLE-2: [1] function "SHL" is called in a right-hand 'always' statement (it is pure call);
[2] bit width of function return value is less than bit width of assignment destination => violation (message-1);

```
wire [7:0] TS_VAL;
reg  [9:0] L_PIPE;
...
function [7:0] SHL;
    input [7:0] arg;
```

```
begin
     SHL = arg << 1;
end
endfunction
...
always @( TS_VAL )
     L_PIPE = SHL( TS_VAL );
```

Function return value bit w idth "8" is less than bit w idth "10" of the assignment destination. Upper bits of the assignment destination w ill be filled w ith zeroes. Match bit w idths exactly.

EXAMPLE-3: [1] bit width of function return value is greater than bit width of assignment destination, but right-hand-side is not pure function call (function is member of an expression) => no violation (this is case for 2.10.3.3)

```
wire [7:0] TS_VAL;
wire [7:0] L_PIPE;
...
function [7:0] SHL;
     input [7:0] arg;
begin
     SHL = arg << 1;
end
endfunction
...
assign L_PIPE[5:0] = SHL( TS_VAL ) + 8'b01010101;
```

STARC_VLOG 2.1.3.4

RULE NAME	**A function statement should end with a return value assignment**	
MESSAGE	**Function does not end with a return value assignment in every execution path. Return value assignment should be the last statement for any function execution path**	
	DETAIL-1	*Return value assignment is not performed in this execution path*
	DETAIL-2	*Redundant statements after the return value assignment*
PROBLEM DESCRIPTION	The function statement should end by return value assignment to function statement. Statements following the return value in a function statement will not be executed (logic synthesis and simulation ignore these lines)	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans every execution path in a function statement to verify that each execution path is finished by return value assignment (RVA): – lower scopes are not checked if RVA is found in the current scope – if RVA is present at least in one branch of 'case' statement – checkers treats it as if all 'case' branches have RVA (completeness of 'case' in a function is checked by rule 2.1.1.2) – execution path is good if last statement of this execution path has/is RVA	

EXAMPLE-1: [1] function doesn't have RVA in one of execution paths => violation (DETAIL-1);

[2] redundant statements after RVA in another one execution path => violation (DETAIL-2);

```
function FUNC;
     input a;
     input b;
     reg   t;
begin
     if( a ) begin
          FUNC = a | b;
          t = a;
     end
     else begin
          b = ~b;
          a = ~a;
     end
end
endfunction;
```

Function does not end w ith a return value assignment in every execution path. Return value assignment should be the last statement for any function execution path.

Redundant statements after the return value assignment

Return value assignment is not performed in this execution path

EXAMPLE-2: [1] lower level execution path is not full, but there is no violation due to RVA in the upper level execution path;

[2] case statement is treated as if all branches has RVA, because RVA is present in at least one of the branches => no violation;

[3] RVA is not present in the last execution path => violation (DETAIL-1);

```
function FUNC;
     input a;
     input b;
     reg   t;
begin
     if( a ) begin
          if( b ) begin
               t = a;
          end

          FUNC = a | b;
```

Function does not end w ith a return value assignment in every execution path. Return value assignment should be the last statement for any function execution path.

*This execution path "**if**(b)" is not full, but upper level execution path "**if**(a)" contains RVA => no violation*

```
    end
    else
        case( b )
            1'b0: FUNC = ~b;
            1'b1: a = ~a;
        endcase

    if( b ) begin
        t = b;
    end
end
endfunction;
```

One branch contains RVA => case is full => no violation

Return value assignment is not performed in this execution path (*RVA is not performed neither in this execution path "**if**(b)" nor at the upper level execution path "**begin** ... **end**"*)

EXAMPLE-3: [1] function includes execution path that is not full, but the last statement in global execution path is full => no violation

```
function FUNC;
    input a;
    input b;
    reg   t;
begin
    if( a ) begin
        t = a;
        FUNC = a | b;
    end
    else
        t = b;

    if( t )
        FUNC = t;
    else
        FUNC = ~t;
end
endfunction
```

*This execution path "**else**" does not have RVA, but the last statement "**if**(t)" in the upper level execution path "**begin** ... **end**" has RVA => no violation*

STARC_VLOG 2.1.3.5

RULE NAME	**In a function statement, global signal assignment should not be performed**	
MESSAGE	**Function “{FuncName}” assigns values to outer signals. In a function statement, global signal assignment should not be performed.**	
	DETAIL	*Signal “{ObjectName}” is assigned in a function.*
PROBLEM DESCRIPTION	If only global signal is assigned within function statement it makes assignment which has function call in RHS unnecessary because an empty value is returned. However, the description, which invokes the function statement, necessary. If function statement is not invoked global signal, assigned within function body, do not change its value. If an assignment is performed to both a function name and a global signal inside the same function statement, the description becomes complicated and easily leads to mistakes. When executing logic synthesis tools, if multiple outputs are described by one function statement, redundant logic tends to be generated and circuit quality decreases. So do not assign global signals in function statements.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans function statements: – if assignment to external (not local to function) is detected => violation Note: if global signal is for loop variable => no violation	

EXAMPLE-1: [1] global signal is assigned in function statement => violation

```
module top;

    ...

    reg glbl;

    function func1;

        input a;
        input b;

    begin
        if ( a )
            func1 = b;
        else
            glbl = a;
        ...
    end
    endfunction

     ...

endmodule
```

Function “func1” assigns values to outer signals. In a function statement, global signal assignment should not be performed.

Signal “glbl” is assigned in a function.

EXAMPLE-2: [1] global signal is assigned in the function statement but it is for loop variable => no violation

```
module top;

    ...

    integer   glbl;

    function [3:0] bw_or;

        input [3:0] a;
        input [3:0] b;
    begin
        for( glbl = 0; glbl < 4; glbl = glbl + 1 )
            bw_or[glbl] = a[glbl] || b[glbl];

         ...

    end
    endfunction

     ...

endmodule
```

2.1.4 Instructions for equation level descriptions (different from VHDL)

STARC_VLOG 2.1.4.5

RULE NAME	**Logical operator should not be used for vector (Verilog only)**	
MESSAGE	**Logical operator has vector argument(s). Use bit-wise operators for multi-bit arguments and logical operators only for 1-bit arguments.**	
PROBLEM DESCRIPTION	Logical operators (!, &&, ||) treat argument as FALSE if it is 0 or as TRUE in other case and return 1-bit result. Bit-wise operators process arguments bit by bit. Results of calculations are the same for logical and bit-wise operators only if arguments are of 1-bit width. For other arguments results are different. So it is recommended to use logical operators only with 1-bit arguments to avoid mistakes.	
	LEVEL	RECOMMENDED 1
CHECKER BEHAVIOR	Checker detects all logical operators (!, &&, ||): – if width of any argument (signal or constant) is more than 1 bit => violation. Note: only one message is produced per one operator.	

EXAMPLE-1: [1] one argument of logical operator is 2-bit width => violation.

```
reg one_bit_reg;
parameter [1:0] two_bits_param = 2'b01;

assign out_reg = one_bit_reg && two_bits;
```

Logical operator has vector argument(s). Use bit-w ise operators for multi-bit arguments and logical operators only for 1-bit arguments.

STARC_VLOG 2.1.4.6

RULE NAME	**Caution is advised with bit width for reduction operators (Verilog HDL only)**	
MESSAGE-1	**Argument bit width of reduction operator "{ReductOp}" is equal to one bit. Such description is not recommended since no operation will be performed.**	
MESSAGE-2	**Argument bit width "{ArgBitWidth}" of reduction operator "{ReductOp}" is too large. It is recommended to use arguments less than or equal to "{MAX_ARGUMENT_WIDTH}" bits to avoid performing functions with many operational steps.**	
PROBLEM DESCRIPTION	Reduction operators are available only in Verilog HDL. Reduction operators are efficient, however, if the bit widths of the values to be executed are too large, functions with many operational steps are created by a synthesis tool. When bit width is one bit, no operation is performed. Thus, caution is advised with bit width for reduction operators	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker detects reduction operators (&, ~&, |, ~|, ^, ~^): – if bit width of argument is equal to 1 bit => violation (message- 1) – if bit width of argument is greater than MAX_ARGUMENT_WIDTH => violation (message-2) Note: parameter MAX_ARGUMENT_WIDTH is defined in configuration file (default value is 8)	

EXAMPLE-1: [1] argument 'a' of the reduction operator is one bit width => violation (message-1)
[2] bit width of argument 'b' of the reduction operator is greater then recommended => violation (message-2)

```
reg a;
reg [15:0] b;

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

Argument bit w idth of reduction operator "&" is equal to one bit. Such description is not recommended since no operation w ill be performed.

Argument bit w idth "16" of reduction operator "|" is too large. It is recommended to use arguments less than or equal to "8" bits to avoid performing functions w ith many operational steps.

2.1.5 Use a conditional operator *((A)?B:C)* only once (Verilog only)

STARC_VLOG 2.1.5.1

RULE NAME	**Use nesting of a conditional operator (?) only once (Verilog only)**	
MESSAGE-1	**Ternary conditional operator contains {NestedCount} nested conditional operators. It is recommended to use this operator only once to improve readability of the description and decrease possibility of nesting mistakes.**	
PROBLEM DESCRIPTION	The conditional operator can be nested, but since readability of the description decreases and nesting mistakes are more likely, it is recommended that a conditional operation be used only once. When two or more ? exist the nesting relationship is difficult to verify, whether it is an if-if nest or an else-if nest. Therefore, using an if statement within an always construct is recommended.	
	LEVEL	RECOMMENDATION 3
MESSAGE-2	**Ternary conditional operator contains {NestedCount} nested conditional operators. Nesting is limited by {MAX_NESTING_LEVEL} level(s) when it cannot be avoided.**	
PROBLEM DESCRIPTION	When nesting of conditional operators cannot be avoided number of levels should be minimized. Recommended limit of nested operators is 10.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects ternary conditional operators: – if conditional operator contains another (nested) ternary conditional operators => violation (message- 1) – if count (excluding the parent one) of nested ternary operators is greater than value of parameter MAX_NESTING_LEVEL => violation (message-2) Note: parameter MAX_NESTING_LEVEL is defined in configuration file (default value is 10)	

EXAMPLE-1: [1] conditional operator contains nested conditional operator => violation (message- 1)

```
S_OUT = ( SEL_1 - SEL_2 )? A : ( SEL_1 | SEL_2 )? D : B + D;
```

Ternary conditional operator contains 1 nested conditional operators. It is recommended to use this operator only once to improve readability of the description and decrease possibility of nesting mistakes.

EXAMPLE-1: [1] conditional operator contains 4 nested conditional operators;
[2] parameter MAX_NESTING_LEVEL has value 3 => violation (message-2)

```
assign S_OUT = ( SEL_1 - SEL_2 )? A :
     ( SEL_1 + SEL_2 )? C :
          ( SEL_1 | SEL_2 )? B :
               ( SEL_1 & SEL_2 )? D :
                    ( SEL_1 ^ SEL_2 )? E : F;
```

Ternary conditional operator contains 4 nested conditional operators.
Nesting is limited by 3 level(s) w hen it cannot be avoided.

STARC_VLOG 2.1.5.3

RULE NAME	**In the conditional expression of an if statement or the conditional operator (?) the result should not be a vector (Verilog only)**	
MESSAGE	**The result of conditional expression is {CondExpBW} bits wide. The result of conditional expression should be 1 bit.**	
PROBLEM DESCRIPTION	The conditional expression of an if statement or the ternary operator should be judged as being true or false. It is clearly for 1 bit expression result when it is 1 (true) or 0 (false). The vector expression result may easily lead to confusions, so the result of a conditional expression should be 1 bit.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker verifies conditional expression of the if statements and the ternary operator (?): – if expression bit width is greater then 1 => violation	

EXAMPLE-1: [1] result of selection expression has width greater then 1 => violation

```
input [7:0] sel;

    ...

    always @( ... ) begin

        if( sel )

          ...

    end
```

The result of conditional expression is 8 bits w ide. The result of conditional expression should be 1 bit.

EXAMPLE-2: [1] selection expression contains equality operator so result bit width is 1 => no violation

```
input [7:0] sel;
input [7:0] in_a,in_b;
reg   [15:0] res;

    ...

    always @( ... ) begin

        res[15:8] = ( sel == 8'b01010101 )? in_a : in_b;

        ...

    end
```

2.1.6 Specifying the range of an array

STARC_VLOG 2.1.6.1

RULE NAME	**Specification of an array should be [MSB:LSB], if it is one-dimensional**	
MESSAGE	**Module "{ModuleName}" contains {IllegalDeclCount} vector range declaration(s) that does not correspond to [MSB:LSB] style. Such style is recommended to avoid problems with arithmetic operations which are based on [MSB:LSB] assumption.**	
	DETAIL	*Vector "{VecName}" range declaration does not correspond to [MSB:LSB] specification.*
PROBLEM DESCRIPTION	Arithmetic operations are based on the assumption of [MSB:LSB] and if a reverse vector is used, it is necessary to convert it for arithmetic operations otherwise operation result is incorrect. It is recommended to specify [MSB:LSB], even if the array does not perform arithmetic operations.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker scans vector declarations (function return value range, function / task ports are also considered) and checks range declaration [MSB:LSB]: – if (MSB < LSB) => violation Note-1: array of vectors declaration are skipped Note-2: array range may be parameter-dependent => elaboration time checks required	

EXAMPLE-1: [1] within vector declaration MSB > LSB => violation

```
module top;
    reg [0:31] tmp;
    ...
endmodule
```

Module "top" contains 1 vector range declaration(s) that does not correspond to [MSB:LSB] style. Such style is recommended to avoid problems w ith arithmetic operations w hich are based on [MSB:LSB] assumption.

Vector "tmp" range declaration does not correspond to [MSB:LSB] specification.

EXAMPLE-2: [1] within array of vectors declaration MSB > LSB => no violation

```
module top;
    reg [0:5] mem [3:0];
    ...
endmodule
```

STARC_VLOG 2.1.6.2

RULE NAME	**The LSB of an array should be 0**	
MESSAGE	**Least significant bit of vector "{VecName}" is "{LSBValue}". It is recommended to specify "{RECOMMENDED_LSB}" for LSB of vector.**	
PROBLEM DESCRIPTION	The specification of a vector should be [MSB:LSB] and LSB should be 0. By using same style for vector declarations code is simplified, readability is improved and possibility of mistakes decreases (see also 2.1.6.1).	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans vector declarations (function return value range, function / task ports are also considered) and detect LSB in range declaration expression: – if (LSB != RECOMMENDED_LSB) => violation Note-1: array of vectors declaration are skipped Note-2: array range may be parameter-dependent => elaboration time checks required Note-3: parameter RECOMMENDED_LSB value is described in configuration file and may be used to tune vector declarations style (by default RECOMMENDED_LSB == 0).	

EXAMPLE-1: [1] within vector declaration LSB != RECOMMENDED_LSB (RECOMMENDED_LSB == 0) => violation

```
module top;
    reg [15:31] tmp;
    ...
endmodule
```

Least significant bit of vector "tmp" is "15". It is recommended to specify "0" for LSB of vector.

EXAMPLE-2: [1] within array of vectors declaration LSB != RECOMMENDED_LSB (RECOMMENDED_LSB == 0) => no violation

```
module top;
    reg [15:7] mem [3:0];
    ...
endmodule
```

STARC_VLOG 2.1.6.3

RULE NAME	**The index of an array should be simple signal names only**	
MESSAGE	**Operation(s) is detected within index expression of vector "{VectorName}". It is recommended to use simple signal names in vector indexes to avoid generation of redundant logic.**	
PROBLEM DESCRIPTION	By using signal names in a vector index, code is simplified and readability is improved. However, for descriptions using signal names in a vector index, many levels of logic gates is generated by logic synthesis in the case of a complex circuit description. Also, depending on the logic synthesis tool, even if usage of a signal name for a index is supported, highly redundant logic gates may be generated. Therefore, a vector index should simply consist of signal names only, and operations should not be included in it. However, it is not a problem if index consists of an operation with a loop variable (see 2.9.2.1)	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans vector references described in synthesizable context: – if vector index is not a simple signal => violation Note-1: index is simple signal when it does not contain any operators (concatenations are allowed) Note-2: operators are allowed when operands are loop variable and constant (see 2.9.2.1)	

EXAMPLE-1: [1] bit selection in used for function input;
[2] vector index is not a simple signal, it contains arithmetic operation => violation

```
assign out1 = res ( in1 [ sel * 2 : 0] , sel );
```

Operation(s) is detected w ithin index expression of vector "in1". It is recommended to use simple signal names in vector indexes to avoid generation of redundant logic.

EXAMPLE-2: [1] concatenation is used for part selection => no violation

```
assign out1 = in2[ {sel, sel} : 0];
```

STARC_VLOG 2.1.6.4

RULE NAME	**The range of an array index should be appropriately specified**
MESSAGE-1	**Array "{ArrayName}" is referenced by constant index value "{IndexVal}" that is out of array range [{ArrMSB}:{ArrLSB}]. Pay attention to the MSB and LSB values and specify array indexes carefully to avoid unexpected simulation results.**
MESSAGE-2	**Array "{ArrayName}" is referenced by variable index value. Values possible for index ({IndexMin} : {IndexMax}) are out of array range [{ArrMSB}:{ArrLSB}]. Pay attention to the MSB and LSB values and specify array indexes carefully to avoid unexpected simulation results.**
PROBLEM DESCRIPTION	Attention should be paid to the MSB and LSB values of an array, as well as to the range of the signals specified in the array index. If values that exceed the MSB value or index value which are lower than LSB are specified, unexpected results may occur. Many simulators give results of 'x' when a value exceeding MSB is assigned and 'z' when a value less than LSB is assigned. Depending on the simulator, output values will vary and logic synthesis tools will not generate appropriate circuits. LEVEL: RECOMMENDATION 2
CHECKER BEHAVIOR	Checker scans array references within 'always' and 'assign' synthesizable statements: – if any of indexes (constant or variable) used in references does not fit to declared range: – if array index is constant and index > MSB or index < LSB => violation (message-1); – if array index is variable and maximum possible value is greater than MSB or/and minimum possible value is less then LSB => violation (message-2). Note-1: initial assignments are not checked. Note-2: array can be referenced by loop constant index. Loop constant index is an expression of statically countable loops (see 2.9.1.2), that contain only constants, loop variables, unary, binary and ternary operation (in this case index range is counted by values). If expression has non-loop variables, concatenation, it is simple expression (in this case index range is counted by bit-width of operands).

EXAMPLE-1: [1] an index of array reference in constant;
[2] constant value does not lie in the range [MSB:LSB] => violation (message-1).

```
reg [31:0] in1 [10:0];

assign data = in1[ 8'h10 ];
```

Array "in1" is referenced by constant index value "16" that is out of array range [10:0]. Pay attention to the MSB and LSB values and specify array indexes carefully to avoid unexpected simulation results.

EXAMPLE-2: [1] an index of array reference in variable;
[2] maximal index value is grater then MSB (15 > 3) and LSB > 0 => violation (message-2).

```
reg [31:0] in1 [3:2];
reg [ 7:0] sel;

assign data = in1[ sel[3:0] ];
```

Array "in1" is referenced by variable index value. Values possible for index (0 :15) are out of array range [3:2]. Pay attention to the MSB and LSB values and specify array indexes carefully to avoid unexpected simulation results.

STARC_VLOG 2.1.6.5

RULE NAME	**For an array index, 'x' and 'z' should not be used**	
MESSAGE	**Unknown value in the index: {ObjectValue}. Incorrect synthesis results may be generated.**	
PROBLEM DESCRIPTION	An error does not occur during simulation if unknown value is used as index. But error may occur in the logic synthesis tool or an incorrect circuit may be generated. So do not use 'x' or 'z' for an array index in RTL descriptions.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies bit-selection or part-selection index expression which evaluates to a constant: – if value of index contain unknown bits ('x' or 'z') => violation	

EXAMPLE-1: [1] bit selection expression contains unknown ('z') value => violation

```
assign out1 = { in1[1'bz], in2[30:0] };
```

```
Unknown value in the index: 1'bz. Incorrect
synthesis results may be generated.
```

EXAMPLE-2: [1] reference to memory contains unknown 'x' value => violation

```
always @(mem) begin
     out1 = mem[ 3'b1x1 ]
end
```

Unknown value in the index: 3'b1x1. Incorrect synthesis results may be generated.

2.2 always construct description in combinational logic

2.2.1 Avoid the risk of generating latches

STARC_VLOG 2.2.1.1

RULE NAME	**Latches are generated unless all conditions have been described. Care should be taken not to create latches**	
MESSAGE	**Possible latch inference for {LatchSigCount} signal(s)**	
	DETAIL	*Signal "{SignalName}" is not assigned in all cases*
PROBLEM DESCRIPTION	When describing a process for a combinational circuit, each signal should be defined in all execution paths of the process. It usually means that assignment to the signal should be performed in each branch of any conditional statement ('if', 'case') inside the process. Otherwise, logic synthesis tools will recognize that output signal must be retained for certain condition. As a result, latch will be generated to maintain output value.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans 'always' statements in order to detect the latches. Following requirements are imposed on each signal assigned in an 'always' construct: – if signal is assigned in some branch of 'if'/'case' statement, it must be also assigned in all other branches of this statement (i.e. signal assignment should be complete) – if there is no 'else' branch in 'if' statement => all conditions must be covered by existing branches (i.e. 'if' statement should be complete) – if there is no 'default' clause in 'case' statement => all conditions must be covered by existing clauses (i.e. 'case' statement should be complete) Consecutively, latch is inferred if one of the above restrictions is broken. So, violation message is displayed when 'always' block describes latches only and one from the two following cases is true: – 'if'/'case' statement is complete, but signal is not assigned in all branches – the signal is assigned in all branches, but 'if'/'case' statement is incomplete Note-1: when completeness of 'case'/'if' statement depends on parameter => elaboration-time checking is required (warning message will include instance name) Note-2: detail-message for vector can be either single (if latches are inferred by all elements of the vector) or multiple (if latches are inferred only by some bits)	

EXAMPLE-1: [1] 'if' is complete ('else' branch is present) but one of two signals is not assigned in all branches

```
always @( S, A, B ) begin
    if( S == 2'b00 ) begin
        F1 <= A | B;
        F2 <= 1'b0;
    end
    else if( S == 2'b01 ) begin
        F1 <= 1'b0;
        F2 <= 1'b1;
    end
    else if( S == 2'b10 ) begin
        F1 <= 1'b0;
    end
    else begin
        F1 <= 1'b0;
        F2 <= A & B;
    end
end
```

Possible latch inference for 1 signal(s)

Signal "F2" is not assigned in all cases

EXAMPLE-2: [1] 'case' is incomplete ('default' clause is not specified and case clauses don't cover all possible conditions) => violation for each signal;

[2] note, that signal "F2" is not assigned in all branches, whereas signal "F1" is assigned in all branches

```
always @( S, A, B ) begin
     case( S )
          2'b00: begin
               F1 <= A | B;
               F2 <= 1'b0;
          end
          2'b01: begin
               F1 <= 1'b0;
               F2 <= 1'b1;
          end
          2'b10: begin
               F2 <= A & B;
          end
     endcase
end
```

Possible latch inference for 2 signal(s)

Signal "F1" is not assigned in all cases

Signal "F2" is not assigned in all cases

EXAMPLE-3: [1] casex is incomplete (completeness of depends on parameter => elaboration-time checking required);

[2] assigned signal "F" is 2-bit vector (single warning issued for it – due to incomplete 'case' – all bits will infer latches)

```
parameter [1:0] param = 2'b01;

always @( S, A, B ) begin
     casex( S )
          2'bx0: begin
               F[0] <= A | B;
               F[1] <= 0;
          end
          param: begin
               F[0] <= 1;
               F[1] <= A & B;
          end
     endcase
end
```

Possible latch inference for 1 signal(s)

Signal "F" is not assigned in all cases

2.2.2 Define every input signal in an *always construct* in the sensitivity list

STARC_VLOG 2.2.2.1

RULE NAME	**All signals at the right of the conditional expression and the assignment statement in the always construct of the combinational circuit must be defined in the sensitivity list**	
MESSAGE	**The sensitivity list of always statement is incomplete. Differences between RTL and post-synthesis simulation results are possible.**	
	DETAIL	*Signal "{ObjectName}" is read, but is not included in the sensitivity list.*
PROBLEM DESCRIPTION	Logic synthesis tools ignore sensitivity lists in combinational circuits assuming that always construct executes if any of reading signal changes. When any signal read in the process is not included in sensitivity list – differences between RTL and gate level simulation may occur.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies all signals which are read in combinational always block: – signals on the right side of procedural assignment – condition expression of the if-statement – selection expression of the case-statement – input arguments of function/task calls If any of detected signals is not included in sensitivity list of current always block => violation. Checker skips following objects: – initial constructs – internally declared signals – delay expressions – event control statements – always constructs if there is an edge description in sensitivity list – blocks without sensitivity list – incremental statement of loop variable is not checked. – loop variables as index – intermediate variables (variables that are assigned and read in this process). Note: elaboration-time checks are possible when needed.	

EXAMPLE-1: [1] signal is used on the right side of procedural assignment and is not included in sensitivity list => violation

```
always @( in1, in2 ) begin

    out1 = in1 & in2 & in3;

end
```

The sensitivity list of alw ays statement is incomplete. Differences betw een RTL and post-synthesis simulation results are possible.

Signal "in3" is read, but is not included in the sensitivity list.

EXAMPLE-2: [1] signal is used for bit selection and is not included in sensitivity list => violation
[2] loop variable is used for bit selection => no violation

```
always @( in1, in2 ) begin

    for( i = 0; i < 10; i = i + 1 ) begin
        out1 = in1[i] && in2[in3];
        end
    ...

    end
```

The sensitivity list of alw ays statement is incomplete. Differences betw een RTL and post-synthesis simulation results are possible.

Signal “in3” is read, but is not included in the sensitivity list.

STARC_VLOG 2.2.2.2

RULE NAME	**Do not define constants and unnecessary signals in the sensitivity list**	
MESSAGE	**Constant(s) or unnecessary signal(s) detected in the sensitivity list. Differences between RTL and post-synthesis simulation results are possible.**	
	DETAIL-1	*Constant "{ObjectName}" is included in the sensitivity list.*
	DETAIL-2	*Signal "{ObjectName}" is included in the sensitivity list, but is not referenced.*
	DETAIL-3	*Parameter "{ObjectName}" is included in the sensitivity list.*
	DETAIL-4	*Signal "{ObjectName}" is unnecessary because it is used only as a loop variable.*
	DETAIL-5	*Signal "{ObjectName}" is unnecessary because it is used only as an intermediate variable.*
PROBLEM DESCRIPTION	Logic synthesis tools ignore sensitivity lists in combinational circuits assuming that always construct executes if any of reading signal changes. Unnecessary signals described in sensitivity list may cause extra cycles of 'always' block execution and differences between RTL and gate level simulation may occur as a result. So do not define constants and unnecessary signals in the sensitivity list	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker verifies sensitivity lists of combinational always block: – if a constant is included in the sensitivity list => violation (detail-1) – if a parameter is included in the sensitivity list => violation (detail-3) – if a signal is included in the sensitivity list is not referenced within current 'always' block (considering enabled tasks) => violation (detail-2) – if signal which is overridden in a named block is included in the sensitivity list => violation (detail-2) – if a global loop variable is included in the sensitivity list => violation (details-4) – if an intermediate variable is specified in sensitivity list => violation (detail-5) Note: see 2.2.2.1 for context that is skipped.	

EXAMPLE-1: [1] signal is used on the right side of procedural assignment and is not included in sensitivity list => violation

```
always @( in1, in2 ) begin

    out1 = in1 & in2 & in3;

end
```

The sensitivity list of always statement is incomplete. Differences between RTL and post-synthesis simulation results are possible.

Signal "in3" is read, but is not included in the sensitivity list.

EXAMPLE-2: [1] signal is used for bit selection and is not included in the sensitivity list => violation
[2] loop variable is used for bit selection => no violation

```
always @( in1, in2 ) begin

    for( i = 0; i < 10; i = i + 1 ) begin
        out1 = in1[i] && in2[in3];
        end
    ...
end
```

The sensitivity list of always statement is incomplete. Differences between RTL and post-synthesis simulation results are possible.

Signal "in3" is read, but is not included in the sensitivity list.

STARC_VLOG 2.2.2.3

RULE NAME	**Multiple event expressions should not be described with always (at least one event expression is required)**	
MESSAGE-1	**'always' block contains {NumberOfEventControlStatements} event control statements. Such description style is rarely synthesizable. One event control statement is required.**	
	DETAIL-1	*Event control statement detected.*
	DETAIL-2	*Task contains event control statement(s).*
MESSAGE-2	**'always' block does not contain any event control statement. Infinite loop is possible during simulation. One event control statement is required.**	
PROBLEM DESCRIPTION	If multiple event expressions are described in an 'always' construct, logic synthesis might be impossible (except some specific cases). Also, description without any event control statement is hazardous (it has the risk of infinite loop during the simulation)	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans 'always' statements: – if multiple event control statements are detected => violation (message-1) – if no event control statement is detected => violation (message-2)	

EXAMPLE-1: [1] 'always' block contains multiple event expressions => violation (message-1);

```
always @( CLK ) begin
    if( RESET = '1' )
        @( posedge CLK )
            F <= 1'b0;
    else
        F <= Y1 ^ Y2;
end
```

'alw ays' block contains 2 event control statements. Such description style is rarely synthesizable. One event control statement is required.

Event control statement detected

Event control statement detected

EXAMPLE-2: [1] 'always' block doesn't contains event control expressions => violation (message-2)

```
always begin
    if( RESET = '1' )
        F <= 1'b0;
    else
        F <= Y1 ^ Y2;
end
```

'alw ays' block does not contain any event control statement. Infinite loop is possible during simulation. One event control statement is required.

EXAMPLE-3: [1] 'always' block contains multiple event control expressions => violation (message-1)

```
always @( posedge CLK ) begin
        Y <= A & B;
    @( negedge CLK )
        Y <= A | B;
end
```

'alw ays' block contains 2 event control statements. Such description style is rarely synthesizable. One event control statement is required.

Event control statement detected

Event control statement detected

2.2.3 Initial value description in *always constructs* (Verilog only)

STARC_VLOG 2.2.3.1

RULE NAME	**Do not mix blocking assignments (=) and non-blocking assignments (<=) in combinational always construct**	
MESSAGE	**Always construct for combinational circuit contains a mixture of blocking (=) and non-blocking (<=) assignments. It is safer to use only one type of assignments in the same 'always' block.**	
PROBLEM DESCRIPTION	Assignment of a value to the signal with non-blocking assignment takes place after the evaluation of all assignment statements in the always block. When the blocking assignment is performed, the assignment is already completed at the time this line is evaluated, so the assigned values on the lines after that are certainly valid. Mixture of assignment types may easily lead to mistakes. Moreover simulation results may differ depending on Verilog-HDL simulators. It is safer to use only one type of assignments in the same always block.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verify types of assignments in always block: – if both type of assignments (blocking and non-blocking) are used => violation	

EXAMPLE-1: [1] assignments of both types are used => violation

```
always
    begin
        case ( in1 )
            1'b1 : out1 = in2;
            1'b0 : out1 = in3;
            default : out1 <= 1'b0;
        endcase

     ...

    end
```

Alw ays construct for combinational circuit contains a mixture of blocking (=) and non-blocking (<=) assignments. It is safer to use only one type of assignments in the same 'alw ays' block.

STARC_VLOG 2.2.3.2

RULE NAME	**Do not assign over the same signal using a non-blocking assignment for combinational circuits**	
MESSAGE	**Multiple non-blocking assignments over the same signal "{SignalName}" are detected on the same execution path in the 'always' construct for a combinational circuit. Final signal value is undetermined.**	
PROBLEM DESCRIPTION	In non-blocking assignment statements, the left hand side members are assigned after all of the right hand side expressions are evaluated. So, multiple non-blocking assignments over the same signal (in same execution path) will result in undefined value.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans combinational 'always' statements in order to find signals that are assigned with non-blocking assignments: – if signal(*) is assigned multiple times under the same execution path(es) => violation *(*) if separate bits of the same signal are assigned – compacting is performed* Note: 'always' for combinational circuit for this checker is such 'always' statement, where all signals are described without edges in the sensitivity list	

EXAMPLE-1: [1] multiple non-blocking assignments over the same signal "Y1" (in the same execution path) => violation;

[2] multiple non-blocking assignments over the same signal "Y2" (execution path is not the same) => no violation;

```
always @( CLK. SEL, A, B ) begin
     Y1 <= 1'b0;
     if( SEL ) begin
          Y1 <= A | B;
          Y2 <= A & B;
     end
     else begin
          Y1 <= A & B;
          Y2 <= A | B;
     end
end
```

Multiple non-blocking assignments over the same signal "Y1" are detected on the same execution path in the 'alw ays' construct for a combinational circuit. Final signal value is undetermined.

EXAMPLE-2: [1] 2-bit vector is assigned with full assignment and with partial assignment => violation for bit that is assigned multiple times

```
always @( CLK, SEL, A, B ) begin
     Q <= 2'b00;
     case( SEL )
          1'b0: Q[0] <= A | B;
          1'b1: Q[0] <= A & B;
     endcase
end
```

Multiple non-blocking assignments over the same signal "Q[0]" are detected on the same execution path in the 'alw ays' construct for a combinational circuit. Final signal value is undetermined.

EXAMPLE-3: [1] multiple non-blocking assignment over the same signal "Q" in sequential circuit => no violation (this is case for 2.2.3.3)

```
always @( posedge CLK ) begin
     if( RESET )
          Q <= 1'b0;
     else begin
          Q <= DATA1;
          Q <= DATA2;
     end
end
```

STARC_VLOG 2.2.3.3

RULE NAME	**Do not assign over the same signal in an always construct for sequential circuits**	
MESSAGE	**'always' construct for sequential circuit contains multiple assignments to the signal "{SignalName}" in the same execution path. This may lead to malfunctions and is difficult to debug.**	
PROBLEM DESCRIPTION	Descriptions with overwrite assignment to the same signal are prone to causing race problems during simulation. The best practice is to avoid such descriptions whenever it is possible (even when making blocking assignments).	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans body of the 'always' statement for sequential circuit and finds assigned signals (with blocking or nonblocking assignments either): – if signal*(*)* (for which FF or latch is inferred) is assigned more than once in single execution path(es) => violation *(*) if separate bits of the same signal are assigned – compacting is performed* Note: 'always' for sequential circuit for this checker is such 'always' statement, where all signals are described with edges in the sensitivity list	

EXAMPLE-1: [1] multiple assignments (blocking and non-blocking) over the same signal "outx[0]" in sequential circuit (in the same execution path) => violation;

```
always @( posedge clk or negedge rst_n )
if( ~rst_n )
     outx <= 4'h0;
else begin
     outx    <= #1 outx << 1;
     outx[0] <= #1 in0;
end
```

'always' construct for sequential circuit contains multiple assignments to the signal "outx[0]" in the same execution path. This may lead to malfunctions and is difficult to debug.

2.3 *FF inferences*

2.3.1 Unify the description style of FF inferences

STARC_VLOG 2.3.1.1

RULE NAME	**Use non-blocking assignment in FF inferences**	
MESSAGE	**'always' construct infers {FFCount} flip-flop(s) assigned with blocking assignment(s) (=). Use non-blocking assignments (<=) in FF inferences.**	
	DETAIL-1	*FF inference for signal "{ObjectName}", {BACount} blocking assignment(s):*
	DETAIL-2	*Blocking assignment to FF signal "{ObjectName}".*
PROBLEM DESCRIPTION	With standard blocking assignments (=), evaluation timing of the right-hand side and assignment timing of the left-hand side are done at the same time, but in the case of non-blocking assignments, assignment to the left-hand side is performed after evaluation of the right-hand side is completely finished. For that reason, it is possible to avoid race conditions using non-blocking assignment.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects always constructs which infer FFs and all signals for which FFs are inferred : – if signal is assigned with blocking (=) assignment => violation	

EXAMPLE-1: [1] always construct infer FF for one signal but it is assigned with non blocking assignment (in two branches) => violation

```
always @( posedge CLK or negedge RESET )

    begin

    if( RESET )

            Q = 1'b0;

    else

            Q = DATA;

end
```

'alw ays' construct infers 1 flip-flop(s) assigned w ith blocking assignment(s) (=). Use non-blocking assignments (<=) in FF inferences.

Blocking assignment to FF signal "Q".

FF inference for signal "Q", 2 blocking assignment(s):

Blocking assignment to FF signal "Q".

EXAMPLE-2: [1] blocking assignment is used but process does not infer FF => no violation

```
always @(CLK or D)
    if (CLK) Q = D;
```

STARC_VLOG 2.3.1.3

RULE NAME	**Set delay values for FF inference**	
MESSAGE	**Delay is not specified for FF inference. It is recommended to insert delay values into assignment expressions for signals to avoid racing problems.**	
	DETAIL	*FF signal "{SignalName}" is assigned without delay {AssignCount} times.*
PROBLEM DESCRIPTION	There is no particular problem if the design consists of a single clock, but when multiple clocks are present with a gated clock, etc., the racing problem tends to occur. One possible decision is to set delay values only during D input assignment and not to put them in an assignment during an asynchronous reset. In any case, inserting delay values at the time of assignment is to prevent the racing problem during RTL simulation (delay values are ignored during synthesis). Therefore, it is recommended to insert delay values into assignment expressions for FF signals.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker detects 'always' which infers FFs and signals for witch FF is inferred: – if delay for FF signal assignment is not specified (Q <= D) => violation Note: FF signal assignments made under asynchronous control are not checked	

EXAMPLE-1: [1] delay is not specified for FF signal assignment under asynchronous control (reset) => no violation;

[2] delay is not specified for FF signal assignment under synchronous control (set) => violation;

[3] delay is not specified for FF signal assignment expression => violation;

```
always @( posedge CLK or negedge rst) begin

    if( !rst )
        Q1 <= 1'b0;
    else if ( set )
        Q1 <= 1'b1;
        else
            Q1 <= D2;

end
```

Delay is not specified for FF inference. It is recommended to insert delay values into assignment expressions for signals to avoid racing problems.

FF signal "Q1" is assigned w ithout delay 2 times.

EXAMPLE-2: [1] delay is not specified for signal assignment, but no FF is inferred by the description => no violation

```
always @( CLK ) begin

    Q <= DATA;

end
```

STARC_VLOG 2.3.1.4

RULE NAME	**Do not use delay values which infer FFs except in an always constructs**	
MESSAGE-1	**It is recommended not to use delay values otherwise than for a flip-flop signal assignments. RTL and post-synthesis simulation results may not match. However, if delay is necessary here, use parameters to enable modification depending on differences in target technologies.**	
MESSAGE-2	**It is recommended not to use delay values otherwise than for a flip-flop signal assignments. RTL and post-synthesis simulation results may not match.**	
PROBLEM DESCRIPTION	Inserting delay values at the time of assignment prevents the racing problem during RTL simulation. Delay values are ignored during synthesis. Delay values may be specified in FF assignment expressions, but they should not be specified in other description blocks (combinational circuits, etc.). If delay values are specified in a combinational circuit, you risk having the simulation become dependent on those delay values.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans 'always' and 'assign' statements for delays: – if delay is specified for FF signal assignment (Q <= #DELAY D) => no violation – if delay is specified in other part of the description (not for FF signal assignment) – if delay is specified with literal constant => violation (message-1) – if delay is parametrized (specified with parameter) => violation (message-2) Note: FF signal (for this rule) – such signal that is: – assigned within edge-controlled 'always' block and – it is referenced from the multiple description blocks – or it is not an intermediate variable – or it is module output	

EXAMPLE-1: [1] delay is specified for FF signal assignment => no violation

```
always @( negedge CLK ) begin

    Q <= #1 DATA;

end
```

EXAMPLE-2: [1] delay is specified not as intra-assignment, but as common delay;
[2] delay is specified with literal constant => violation (message-1)

```
always @( negedge CLK ) begin

   #1 Q <= DATA;

end
```

It is recommended not to use delay values otherw ise than for a flip-flop signal assignments. RTL and post-synthesis simulation results may not match. How ever, if delay is necessary here, use parameters to enable modification depending on differences in target technologies.

EXAMPLE-3: [1] process is combinational;
[2] delay is specified with parameter => violation (message-2)

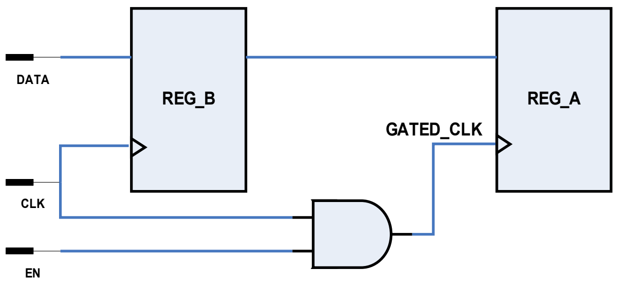
```
parameter DELAY = 1;
```

```
always @( CLK or DATA ) begin
    Q <= #DELAY DATA && CLK;
end
```

It is recommended not to use delay values otherw ise than for a flip-flop signal assignments. RTL and post-synthesis simulation results may not match.

STARC_VLOG 2.3.1.5

RULE NAME	**Specify delay values with integral numbers and do not use negative delay values**
MESSAGE-1	**Specify delay values with integral numbers in flip-flop inferences**
MESSAGE-2	**Negative delay value "{DelayVal}" is detected. Specify delay values with positive numbers in flip-flop inferences.**

PROBLEM DESCRIPTION	In general, it is recommended to insert delay values into assignment expressions for signals in order to prevent the racing problem during the RTL simulation. Consider following section to find out more details about the racing problem:
	`always @( CLK or EN ) begin`
	`    GATED_CLK <= CLK & EN;`
	`end`
	`always @( posedge CLK ) begin`
	`    REG_B <= DATA;`
	`end`
	`always @( posedge GATED_CLK ) begin`
	`    REG_A <= REG_B;`
	`end`
	Gated clock is described here (clock signal CLK and enable signal EN are anded together => gated clock GATED_CLK is generated). Note, that guaranteed relationship is required between the CLK and EN (if EN changes while CLK is active => GATED_CLK pulse => malfunction).
	
	Malfunction can occur either during gate simulation or RTL simulation. Upper drawing displays circuit diagram that is generated from the description above. Think about events order at CLK rise: it is unclear what signal will change first – GATED_CLK or REG_B (if GATED_CLK is first => REG_A will be assigned with old value from REG_B, otherwise – with updated one).
	So, such problem doesn't necessarily occur – it depends on simulator. But, it is strongly recommended to insert delay values into assignment expressions for signals: such style helps to avoid a lot of problems with simulation dependency on some particular tool). As a result, solution to the racing problem will depend on particular device only.
	LEVEL: RULE

CHECKER BEHAVIOR	Checker scans 'always' statements that infer flip-flops and for each detected intra-assignment delay (Q <= #DELAY D) – verifies following: – delay should be specified with integral number (integral numbers are: constants/parameters of integer/reg/time type), if number is not integral => violation with message-1 – delay should be positive (in case of parametrized delay – elaboration-time checking is required) Note-1: if delay is not integral, it will not be checked that it is positive (in other words, only one message will be displayed for non-integral negative delay) Note-2: delay can be specified with integral expression (integral expression is such that consists of integral constants/parameters only)

EXAMPLE-1: [1] delay is specified with positive integral number (parameter) => no violation

```
parameter DELAY = 10;

always @( posedge CLK ) begin
     Q <= #DELAY DATA;
end
```

EXAMPLE-2: [1] delay is specified with integral expression;
[2] expression consist of parameter and constant;
[3] expression is evaluated to negative delay => violation (elaboration-time warning-2);

```
parameter DELAY = 10;

always @( posedge CLK ) begin
     if( RESET )
          Q <= 1'b0;
     else
          Q <= #( DELAY - 20 ) DATA;
end
```

Negative delay value "-10" is detected. Specify delay values w ith positive numbers in flip-flop inferences.

EXAMPLE-3: [1] delay is specified with non-integral constant => violation (message-1)

```
always @( posedge CLK ) begin
     if( RESET )
          Q <= #(7) 1'b0;
     else
          Q <= #(7.7) DATA;
end
```

Specify delay values w ith integral numbers in flip-flop inferences.
(note, if specify delay value "-7.7" here => warning message will be single and same)

EXAMPLE-4: [1] delay is not specified (such case is for rule 2.3.1.3 – "set delay values for FF inferences") => no violation

```
always @( posedge CLK ) begin
     Q <= DATA;
end
```

EXAMPLE-5: [1] delay value is negative in first intra-assignment and non-integral in the second one (but they are specified not in the flip-flop inference) => no violation

```
always @( CLK ) begin
     if( RESET )
          Q <= #(-1) 1'b0;
     else
          Q <= #(7.7) DATA;
end
```

STARC_VLOG 2.3.1.6

RULE NAME	**In FF inference with asynchronous reset, pay attention to the negedge or the posedge of the reset signal**	
MESSAGE-1	**Polarity of asynchronous control signal "{SignalName}" does not match the edge described in the sensitivity list. In FF inference with asynchronous control, pay attention to the negedge or the posedge of the control signal.**	
	DETAIL	*Polarity violation in the condition for asynchronous control*
MESSAGE-2	**Polarity of asynchronous control signal(s) cannot be detected because operator "{OpName}" is used in asynchronous control expression. Define asynchronous control clearly - with signal or its negation - to avoid problems with most of logic synthesis tools.**	
MESSAGE-3	**Polarity of asynchronous control signal(s) cannot be detected because relational operator "{OpName}" is used in asynchronous control expression. Such descriptions are not synthesizable with most of synthesis tools. Use equality operators (=, !=) to compare asynchronous control signal with constant when testing the polarity.**	
MESSAGE-4	**Polarity of asynchronous control signal(s) cannot be detected because comparison is performed with non-constant value. Such descriptions are not synthesizable with most of synthesis tools. Compare asynchronous control signal with constant when testing the polarity.**	
MESSAGE-5	**Polarity of asynchronous control signal(s) cannot be detected because function "{FuncName}" is called on asynchronous reset line. Define asynchronous control clearly - with signal or its negation - to avoid problems with most of logic synthesis tools.**	
MESSAGE-6	**Polarity of asynchronous control signal(s) cannot be detected because sensitivity list contains multiple-bit signal "{SigName}". Such descriptions are not synthesizable. Use single-bit signals to define edge-sensitive 'always' constructs.**	
MESSAGE-7	**Polarity of asynchronous control signal(s) cannot be detected because the asynchronous control logic does not match a standard flip-flop description. Define asynchronous control clearly - with signal or its negation - to avoid problems with most of logic synthesis tools.**	
PROBLEM DESCRIPTION	Asynchronous reset edge should be considered carefully when describing flip-flop inferences. Descriptions where edge in the sensitivity list differs from the polarity in the 'if' conditional branch, can not be synthesized with most logic synthesis tools (it is possible for some tools: unintentional synchronous reset flip-flops will be generated).	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans 'always' statements that infer flip-flops and verifies asynchronous reset controls: – for each asynchronous reset control signal: edge described in the sensitivity list should correspond to polarity in the 'if' branch when it is possible to define polarity – polarity can be detected if following constructs are used in conditions of 'if' branch: – logical / bitwise 'or' (||, |), concatenations({}), logical / bitwise negation (!, ~) Note-1: negation of ORed signals is not allowed (it is not the same that ORing of negated signals) Note-2: vector in an asynchronous condition is the same as ORing of all its bits Note-3: ternary operators are allowed and restrictions for "()?" condition are similar to restrictions for 'if' conditions – comparisons of following format: <signal / concatenation COMP constant>,	

RULE NAME	**In FF inference with asynchronous reset, pay attention to the negedge or the posedge of the reset signal**
	where COMP is one of equality operators (!=, ==, !==, ===) – when it is impossible to define the polarity of asynchronous reset following checks are performed: – if expression, that includes asynchronous control signal, contains an operator that does not belongs to set of allowed operators => violation (message-2) – if expression, that includes asynchronous control signal, contains an comparison that does not belongs to set of allowed comparisons – if comparison contains operator that does not belongs to allowed comparison operators => violation (message-3) – if one side of comparison is not signal / constant / concatenation => violation (message-7) – if asynchronous control signal is compared not with constant => violation (message-6) – if asynchronous control expression contains function call => violation (message-7) (along with 2.1.2.1 that restricts function calls on asynchronous reset lines) – if sensitivity list contains vector / part select => violation (message-6) – in other cases => violation (message-7) – Note-4: when at least one problem is detected it is reported and checking is stopped Note-5: asynchronous control is signal that meets following set of requirements: – 'posedge' or 'negedge' of this signal is specified in the sensitivity list – there is an 'if' branch that includes this signal (signal itself, its logical negation or its comparison with constant of 0/1) and FF signal is assigned inside this branch

EXAMPLE-1: [1] polarity of asynchronous reset control differs from specified in the sensitivity list => violation

```
always @( posedge CLK or negedge RESET )

begin
    if( RESET )
        Q <= 1'b0;
    else
        Q <= DATA;
end
```

Polarity of asynchronous control signal "RESET" does not match the edge described in the sensitivity list. In FF inference w ith asynchronous control, pay attention to the negedge or posedge of the control signal.

Polarity violation in the condition for asynchronous control.

EXAMPLE-2: [1] polarity of asynchronous reset control is parameter-dependent and it differs from specified in the sensitivity list => violation;

[2] elaboration time checking is required => message will include instance name.

```
parameter RESET_POLARITY = 0;

always @( posedge CLK or posedge RESET )

begin
    if( RESET == RESET_POLARITY )
        Q <= 1'b0;
    else
        Q <= DATA;
end
```

Polarity of asynchronous control signal "RESET" does not match the edge described in the sensitivity list. In FF inference w ith asynchronous control, pay attention to the negedge or posedge of the control signal.

Polarity violation in the condition for asynchronous control.

EXAMPLE-3: [1] expression, that includes asynchronous control signal, contains an operator '+' that does not belongs to set of allowed operators => violation (message-2);

[2] expression, that includes asynchronous control signal, contains an operator '-' that does not belongs to set of allowed operators => violation (message-2).

Note: only one warning is expected in the second case.

```
always @( negedge CLK or posedge RESET1 or negedge RESET2 )

begin
    if( RESET1 + RESET2 )
        Q1 <= 1'b0;
    else if( (RESET1 - 1) % 2 )
        Q1 <= 1'b0;
    else
        Q1 <= DATA;
end
```

Polarity of asynchronous control signal(s) cannot be detected because operator “+” is used in asynchronous control expression. Define asynchronous control clearly - w ith signal or its negation - to avoid problems w ith most of logic synthesis tools.

STARC_VLOG 2.3.1.7

RULE NAME	**Do not use both asynchronous set and reset**	
MESSAGE	**{AsynchControlCount} asynchronous set/reset signals for FF "{ObjectName}". Do not use both asynchronous set and reset in FF inferences.**	
	DETAIL-1	*Asynchronous set/reset control*
	DETAIL-2	*Asynchronous reset control*
	DETAIL-3	*Asynchronous set control*

PROBLEM DESCRIPTION

Flip-flops with both asynchronous set and asynchronous reset should not be described. Consider following example:

```
always @( posedge CLK or negedge RESET or negedge SET ) begin
   if( !RESET )
      Q <= 4'b0000;
   else if( !SET )
      Q <= 4'b1111;
   else
      Q <= DATA;
end
```

such description means Q will be realized with asynchronous reset/set flip-flops (while RESET is active – 4'b0000 occurs on Q, while SET is active – 4'b1111). But, consider following description:

```
always @( posedge CLK or negedge RESET or negedge SET ) begin
   if( !RESET )
      Q <= 4'b0000;
   else if( !SET )
      Q <= 4'b1010;
   else
      Q <= DATA;
end
```

it means Q cannot be realized with a simple asynchronous control set/reset flip-flops (while RESET is active – 4'b0000 occurs on Q, while SET is active – 4'b1010). Such description requires logic circuit to be generated in the asynchronous reset line! Moreover, FF may not be inferred correctly (depending on logic synthesis tool).

So, multiple asynchronous controls should not be used.

LEVEL	RULE

CHECKER BEHAVIOR

Checker scans 'always' statements that infer flip-flops and verifies number of asynchronous control signals:

– it is allowed to use only one asynchronous control signal

Note-1: see 2.3.1.6: asynchronous control signal description.

Note-2: if any other signal (asynchronous input) is used as control signal for FF output assignment, it is treated as presence of 2 asynchronous controls: both set and reset.

EXAMPLE-1: [1] flip-flop with asynchronous reset and synchronous set is described => no violation (asynchronous control is single)

```
always @( posedge CLK or negedge RESET ) begin
     if( !RESET )
          Q <= 1'b0;
     else if( SET )
          Q <= 1'b1;
     else
          Q <= DATA;
end
```

EXAMPLE-2: [1] flip-flop with two asynchronous controls => violation with 2 details; [2] first asynchronous control is simple reset line => detail-2; [3] second asynchronous control is asynchronous input => detail-1

```
always @( posedge CLK or negedge RESET or negedge SET ) begin

     if( !RESET )
          Q <= 1'b0;

     else if( !SET )
          Q <= ASYNC_INPUT;

     else
          Q <= DATA;
end
```

2 asynchronous reset/set signals for FF "Q". Do not use both asynchronous set and reset in FF inferences.

Asynchronous reset control

Asynchronous set/reset control

2.3.2 Circuits will vary with non-blocking and blocking assignment statements (Verilog only)

STARC_VLOG 2.3.2.2

RULE NAME	**Do not mix blocking and non-blocking assignments in FF inference always construct**
MESSAGE	**Detected a mixture of non-blocking (<=) and blocking (=) assignments in the 'always' construct for sequential circuit. Such description style may not synthesize.**
PROBLEM DESCRIPTION	Coexistence of blocking assignment statements (=) and non-blocking assignment statements (<=) within a single 'always' construct doesn't cause syntax error and it is allowed by the language specification. But, such description style must be avoided since it risks to cause errors when using logic synthesis tools.
LEVEL	RULE
CHECKER BEHAVIOR	Checker scans 'always' statements for sequential circuits: – if 'always' construct contains mixture of blocking (=) and non-blocking (<=) assignment statements => violation Note: "always for sequential circuit" is 'always' statement where all signals are described with edges in the sensitivity list

EXAMPLE-1: [1] sequential 'always' block is described (all signals in the sensitivity list are specified with edges);
[2] blocking and non-blocking assignments are mixed within the 'always' body => violation

```
always @( posedge CLK or negedge RESET ) begin
    if( !RESET )
        Q = 1'b0;
    else
        Q <= DATA;
end
```

Detected a mixture of non-blocking (<=) and blocking (=) assignments in the 'alw ays' construct for sequential circuit. Such description style may not synthesize.

EXAMPLE-2: [1] combinational 'always' block is described (signals in the sensitivity list are specified without edges);
[2] blocking and non-blocking assignments are mixed within the 'always' body => no violation.

```
always @( A or B or SEL or RESET ) begin
    Y = 0;
    if( !RESET )
        Y <= 1'b0;
    else if( SEL )
        Y <= A && B;
end
```

2.3.3 Do not mix descriptions that have different edges

STARC_VLOG 2.3.3.1

RULE NAME	**Do not use two or more different clock edges within a single always construct**	
MESSAGE-1	**Different edges are used in {EventControlCount} event control statement(s) for signal "{SignalName}" in an 'always' construct for a sequential circuit. Such description style is not synthesizable and should be avoided.**	
	DETAIL	*{EdgeType} {SignalName}*
MESSAGE-2	**Edges are mixed with levels in one event control statement. Do not use such descriptions because they are not synthesizable and should be avoided.**	
MESSAGE-3	**Different edges are used for noncontrol signals within single event control statement. Such description style is not synthesizable and should be avoided.**	
PROBLEM DESCRIPTION	From the syntax point of view, it is not restricted to have both rising and falling clock edges together in a single 'always' construct. But it is prohibited in the real hardware description, since it is non-synthesizable (it is difficult to implement such description in actual hardware: flip-flops that allow data reading for both clock edges do not exist independently).	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects 'always' statements which contain event control statements with edge specifiers (negedge, posedge): – if not the same edges are used for all non-control signals within single event control expression => violation (message-3) Note: 'control' for 'always' process is a signal which is used in the conditional expression inside a process – if edge-sensitive signals are mixed with level-sensitive within single event control expression => violation (message-2) – if same signals are used with different edges in different event control statements => violation (message-1)	

EXAMPLE-1: [1] same signals are used with different edges in different event control statements => violation (message-1)

Different edges are used in 2 event control statements for signal "CLK" in an 'alw ays' construct for a sequential circuit. Such description style is not synthesizable and should be avoided.

```
always @( posedge CLK ) begin
              Y <= A ^ B;
       @( negedge CLK )
              Y <= A & B;
end
```

posedge CLK

negedge CLK

EXAMPLE-2: [1] same signals are used with different edges in different event control statements => violation (message-1)

Different edges are used in 2 event control statements for signal "CLK" in an 'alw ays' construct for a sequential circuit. Such description style is not synthesizable and should be avoided.

```
always @( posedge CLK or negedge RESET ) begin
              Y <= A ^ B;
       @( negedge CLK )
              Y <= A & B;
end
```

posedge CLK

negedge CLK

STARC_VLOG 2.3.3.2

RULE NAME	**Do not use two or more identical clock edges within a single always construct**	
MESSAGE-1	**Identical edges are used in {EventControlCount} event control statements for signal "{SignalName}" in an 'always' construct for a sequential circuit. Such description style may be not synthesizable and should be avoided.**	
	DETAIL	*{EdgeType} {SignalName}*
MESSAGE-2	**Event control statement described in 'always' construct after 'begin' keyword. Logic synthesis may be impossible for some tools. Specify event control statement at the top of an 'always' construct.**	
PROBLEM DESCRIPTION	In actual hardware, it is difficult to implement flip-flops controlled by a multiple clock signals. Some logic synthesis tools will generate implicit state machine circuit controlling edges change order (such practice should not be used: state machine is not defined in the HDL description, but it is generated by logic synthesis tool => such HDL description is not concise). Additionally, event control description should be specified at the top of an 'always' construct (to enable logic synthesis for some tools).	
	LEVEL	RULE
CHECKER BEHAVIOR	1) Checker detects 'always' statements which contain event control statements with edge specifiers (negedge, posedge): – if not same edge is used in all event controls => violation (message-1) 2) Checker detects 'always' statements where is no event control statement at the top of the construct (message-2)	

EXAMPLE-1: [1] two event controls are used (sets are the same, two edges are different) => no violation; [2] no event control statement described at the top of 'always' construct => violation (message-2)

```
always begin
    @( posedge CLK )
        Y <= A ^ B;
    @( negedge CLK )
        Y <= A & B;
end
```

Event control statement described in 'always' construct after 'begin' keyword. Logic synthesis may be impossible for some tools. Specify event control statement at the top of an 'always' construct.

EXAMPLE-2: [1] two event controls are used (sets are same, edges are the same) => violation

Identical edges are used in 2 event control statements for signal "CLK" in an 'always' construct for a sequential circuit. Such description style may be not synthesizable and should be avoided.

```
always @( posedge CLK ) begin
        Y <= A ^ B;
    @( posedge CLK )
        Y <= A & B;
end
```

posedge CLK

posedge CLK

EXAMPLE-3: [1] three event controls are used (sets are the same, edges are not same for all signals) => violation (for signals with same edges)

Identical edges are used in 3 event control statements for signal "CLK" in an 'always' construct for a sequential circuit. Such description style may be not synthesizable and should be avoided.

```
always @( posedge CLK or negedge RESET ) begin

        Y <= A ^ B;

    @( posedge CLK or posedge RESET )

        Y <= A & B;
```

posedge CLK

posedge CLK

```
     @( posedge CLK or posedge RESET )
          Y <= A | B;
end
```

posedge CLK

2.3.4 Do not specify an initial FF value in a description (different from VHDL)

STARC_VLOG 2.3.4.1

RULE NAME	**Do not specify FF initial values explicitly in initial constructs**	
MESSAGE	**Signal "{SignalName}", for which FF is inferred, is assigned {AssignmentsCount} time(s) in {InitialCount} 'initial' construct(s). 'initial' construct is ignored by synthesis tools, use reset signal to initialize FF.**	
	DETAIL-1	*FF output is assigned in the 'initial' construct.*
PROBLEM DESCRIPTION	It is common case to assign FF initial value in an 'initial' construct. This value will function as initial value during the RTL simulation. But logic synthesis ignores 'initial' construct and such "reset" will not be defined.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker collects FF signals from 'always' statements and verifies that these signals are not assigned in an 'initial' constructs Note: FF signal is such signal for which flip-flop is inferred	

EXAMPLE-1: [1] flip-flop is inferred for signal "Q"; [2] signal "Q" is assigned in an 'initial' construct => violation

```
initial
     Q <= 1'b0;

always @( posedge CLK ) begin
     Q <= DATA;
end
```

Signal "Q", for w hich FF is inferred, is assigned 1 time(s) in 1 'initial' construct(s). 'initial' construct is ignored by synthesis tools, use reset signal to initialize FF.

FF output is assigned in the 'initial' construct

EXAMPLE-2: [1] latch is inferred for signal "Q"; [2] signal "Q" is assigned in an 'initial' construct => no violation

```
initial
     Q <= 1'b0;

always @( CLK ) begin
     if( CE )
          Q <= DATA;
end
```

STARC_VLOG 2.3.4.2

RULE NAME	**Logic synthesis ignores initial constructs, so it should not be used**	
MESSAGE	**Module "{ModuleName}" contains {InitialStatementsCount} 'initial' statement(s). An 'initial' statement cannot be used in an RTL description.**	
	DETAIL	*An 'initial' statement detected.*
PROBLEM DESCRIPTION	A value assigned in an initial block is available in the RTL description, but such an initial value cannot be used at the gale level because logic synthesis tools ignore the initial construct.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans each module: – if initial blocks are detected => violation	

EXAMPLE-1: [1] module contains the initial construct => violation

```
module top

    ...
    initial
        tmp = 0;

    always
      ...
        tmp <= tmp + 1;

endmodule
```

Module "top" contains one 'initial' statement. An 'initial' statement cannot be used in an RTL description.

An 'initial' statement detected.

2.3.5 Do not use descriptions which generate FFs having fixed input values

STARC_VLOG 2.3.5.1

RULE NAME	**Do not use descriptions which to generate FFs having fixed input values**	
MESSAGE	**Inferred FF “{ObjectName}” will have constant input. Do not use descriptions which infer FFs with fixed input values.**	
PROBLEM DESCRIPTION	Fixed input values in FF become untestable and fault detection rate drops extremely	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans 'always' statements that infer flip-flops and displays violation if FF signal is assigned with constant/parameter within all execution paths of the process Note: violation is displayed for the first assignment	

EXAMPLE-1: [1] flip-flop is inferred for signal "Q";
[2] signal "Q" is assigned a constant value in all execution paths => violation

```
parameter RESET_LEVEL = 1'b0;

always @( posedge CLK ) begin
    if( RESET )
        Q <= RESET_LEVEL;
    else
        Q <= 1'b1;
end
```

Inferred FF "Q" w ill have a constant input. Do not use descriptions w hich infer FFs w ith fixed input values.

EXAMPLE-2: [1] flip-flop is not inferred for signal "Q";
[2] signal "Q" is assigned with constants in all execution paths => no violation

```
always @( CLK ) begin
    if( RESET )
        Q <= 1'b0;
    else if( SET )
        Q <= 1'b1;
end
```

2.3.6 Do not mix FF inferences with and without asynchronous resets

STARC_VLOG 2.3.6.1

RULE NAME	**Do not mix FF inference with and without asynchronous resets in the same always construct**	
MESSAGE	**Do not mix FF inferences with and without or different asynchronous set/reset signals in the same 'always' construct.**	
	DETAIL-1	*"{SignalName}" is a FF without asynchronous control.*
	DETAIL-2	*"{SignalName}" is a FF with {AsyncControlsCount} asynchronous control(s).*

PROBLEM DESCRIPTION	Descriptions that combine FF inference with and without asynchronous reset are misleading and understanding the behavior becomes difficult during debug. Consider following example:
	`always @( posedge CLK or negedge RESET ) begin`
	   `if( !RESET )`
	      `Q <= 4'b0000;`
	   `else begin`
	      `Q <= DATA;`
	      `K <= DATA;`
	   `end`
	`end`
	In the example FF inference which has an asynchronous reset (for signal Q) and a FF inference without an asynchronous reset (signal K) are described in the same always construct.
	Since some synthesis tools may not perform synthesis correctly, the description should be written in such a way that the always construct is divided as in following description:
	`always @( posedge CLK or negedge RESET ) begin`
	   `if( !RESET )`
	      `Q <= 4'b0000;`
	   `else`
	      `Q <= DATA;`
	`end`
	
	`always @( posedge CLK ) begin`
	      `K <= DATA;`
	`end`	

	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker detects always statements which infer FFs and contain an asynchronous reset signal: – if some FF signal is not assigned in an if/case branch for asynchronous reset signal => violation	

EXAMPLE-1: [1] each of 2 bits of same signal are inferred to be FF, but only one of them has asynchronous reset => violation

```
reg [1:0] Q;

always @( negedge CLK or negedge RESET ) begin
    if( !RESET ) begin
        Q[0] <= 1'b0;
    end
    else begin
        Q[0] <= DATA1;
        Q[1] <= DATA2;
    end
end
```

Do not mix FF inferences w ith and w ithout or different asynchronous set/reset signals in the same 'alw ays' construct.

"Q[0]" is a FF w ith1 asynchronous control(s).

"Q[1]" is a FF w ithout asynchronous control.

EXAMPLE-2: [1] two FFs are inferred by the description, both signals are reset => no violation

```
always @( negedge CLK or negedge RESET ) begin
    if( !RESET ) begin
        Q1 <= 1'b0;
        Q2 <= 1'b0;
    end
    else begin
        Q1 <= DATA1;
        Q2 <= DATA2;
    end
end
```

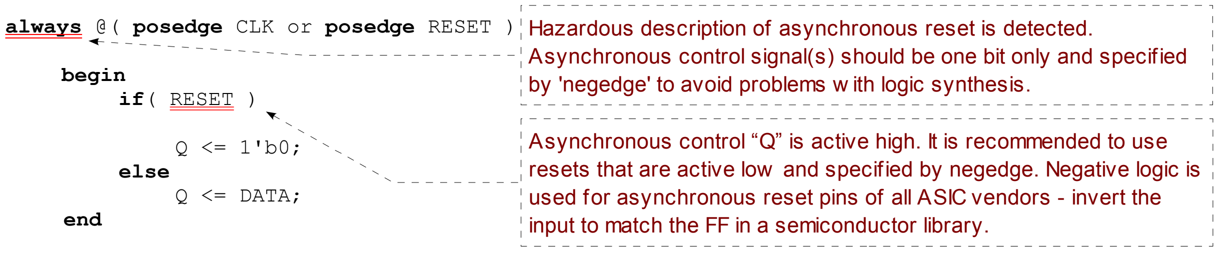
EXAMPLE-3: [1] two FFs are inferred by the description, only one signal is reset, but synchronous reset control is used => no violation

```
always @( negedge CLK ) begin
    if( !RESET ) begin
        Q1 <= 1'b0;
    end
    else begin
        Q1 <= DATA1;
        Q2 <= DATA2;
    end
end
```

STARC_VLOG 2.3.6.2

RULE NAME	**Asynchronous resets are only one bit and active low specified by negedge**	
MESSAGE	**Hazardous description of asynchronous reset is detected. Asynchronous control signal(s) should be one bit only and specified by 'negedge' to avoid problems with logic synthesis.**	
	DETAIL-1	*Asynchronous control "{SigName}" is active high. It is recommended to use resets that are active low and specified by negedge. Negative logic is used for asynchronous reset pins of all ASIC vendors - invert the input to match the FF in a semiconductor library.*
	DETAIL-2	*Asynchronous control signal(s) ({List_Of_Signals}) is used in expression together with synchronous signal(s). It is recommended to use certain signals at asynchronous control lines. Otherwise, be unable to define the kind of reset for output circuit.*
	DETAIL-3	*Asynchronous control signal(s) is used in expression together with another asynchronous control(s). It is recommended to split them between different conditional branches to avoid reset line hazards that tend to occur when logic is described at asynchronous reset lines.*
	DETAIL-4	*"{SigName}" is multiple-bit asynchronous control signal. It is recommended to use one-bit signals for asynchronous control lines to avoid errors at logic synthesis stage.*
PROBLEM DESCRIPTION	All ASIC vendors use negative logic for asynchronous reset pins. So in RTL description, the reset input should be inverted to match the FF in a semiconductor library. There is no problem with logic synthesis even if asynchronous reset is specified by posedge, however inverter cells will be inserted into FF reset line during ASIC layout. It is better if logic cells were not inserted in reset lines and so all asynchronous resets should be integrated by a negedge.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scan 'always' statements which infer flip-flops with asynchronous controls and verify conditional branch inclusive at least one asynchronous control: – if asynchronous control is active high (sensitivity list is checking) => violation (detail-1); – if asynchronous control is used in an expression with another synchronous signals => violation (detail-2); – if asynchronous control is used in an expression with another asynchronous signals => violation (detail-3); – if multiple-bit signal is tested as asynchronous control => violation (detail-4). Note: this checker does not trigger in cases with polarity violation (see 2.3.1.6 for polarity violations)	

EXAMPLE-1: [1] 'always' statement infers flip-flop with asynchronous control;
[2] synchronous control is active high => violation (detail-1).



```
always @( posedge CLK or posedge RESET )
    begin
        if( RESET )
            Q <= 1'b0;
        else
            Q <= DATA;
    end
```

Hazardous description of asynchronous reset is detected. Asynchronous control signal(s) should be one bit only and specified by 'negedge' to avoid problems w ith logic synthesis.

Asynchronous control "Q" is active high. It is recommended to use resets that are active low and specified by negedge. Negative logic is used for asynchronous reset pins of all ASIC vendors - invert the input to match the FF in a semiconductor library.

EXAMPLE-2: [1] 'always' statement infers flip-flop with asynchronous control;

[2] asynchronous control is used in an expression with another synchronous signals => violation (detail-2);

[3] asynchronous control is used in an expression with another asynchronous signals => violation (detail-3).

Hazardous description of asynchronous reset is detected. Asynchronous control signal(s) should be one bit only and specified by 'negedge' to avoid problems w ith logic synthesis.

```
always @( posedge CLK or negedge RST1 or negedge RST2 )
    begin
        if( ~RST2 || RST4 || ~RST1 || RST3 )
            Q1 <= 1'b0;
    else
            Q1 <= DATA;
    end
```

Asynchronous control signal(s) (RST1, RST2) is used in expression together w ith synchronous signal(s). It is recommended to use certain signals at asynchronous control lines. Otherw ise, be unable to define the kind of reset for output circuit.

Asynchronous control signal(s) is used in expression together w ith another asynchronous control(s). It is recommended to split them betw een different conditional branches to avoid reset line hazards that tend to occur w hen logic is described at asynchronous reset lines.

2.4 Latch inferences

2.4.1 Clearly distinguish a latch inference from a combinational circuit

STARC_VLOG 2.4.1.1

RULE NAME	**Clearly distinguish a latch inference from the logic in other combinational circuits**	
MESSAGE	**Possible latch inference for {LatchSigCount} signal(s). Clearly distinguish a latch inference from the other combinational logic.**	
	DETAIL	*Signal "{SignalName}" is not assigned in all cases*
PROBLEM DESCRIPTION	In general, latches are very similar to the typical combinational circuits in a description. It is not easy to distinguish latches from combinational circuit => it is recommended to describe latches separately from other descriptions.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans 'always' statements: – if there is latch inference and some other logic within one block => violation Note-1: see 2.2.1.1: latches inference principles Note-2: elaboration time checking is required for cases where latch inference depends on parameter (main warning will contain instance name)	

EXAMPLE-1: [1] 'always' construct infers a latch for signal "Q";
[2] construct also contains another combinational logic (assignment to a signal "Y1" does not infer a latch) => violation;

```
always @( CLK, DATA, START, G ) begin

    if( G )
        Q <= DATA;

    Y1 <= DATA | START;
end
```

Possible latch inference for 1 signal(s). Clearly distinguish a latch inference from the other combinational logic.

Signal "Q" is not assigned in all cases

EXAMPLE-2: [1] 'always' construct infers latches for all signals being assigned => no violation

```
always @( DATA, START, G ) begin

    if( G )
        Q <= DATA;
    else
        Y1 <= DATA | START;
end
```

STARC_VLOG 2.4.1.2

RULE NAME	**Create latch only blocks and infer latches in these blocks only**	
MESSAGE	**Module "{ModuleName}" contains latchbased description mixed with other combinational logic. It is recommended to place latches in hierarchical modules and keep them separate from other descriptions to avoid the risk of generation an unintentional gated clock circuit.**	
	DETAIL-1	*Process infers latch(es) mixed with other logic. Combinational logic should be described separately.*
	DETAIL-2	*Process infers pure latch(es). If latches are necessary, process could be placed in the separate hierarchical module.*
PROBLEM DESCRIPTION	If complex logic is described in a latch-inferring 'if' statement, an unintentional gated clock circuit may be generated by the logic synthesis tool. To avoid this problem, latches should be placed in hierarchical modules, and kept separate from other descriptions. Moreover, latch inference can not be easily distinguished from combinational logic and there is a risk of unintentional latch generating in this case (see 2.4.1.1). Therefore, it is safer to use latch by instantiating appropriate module.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans modules which infer at least one latch in any process ('always' or 'assign'): – if there are inferences of combinational logic (within same or other processes) => violation: – if process infers latches mixed with another logic => detail-1; – if process infers pure latches => detail-2. Note: latches within instantiated modules are treated as those which do not belong to currently scanned module.	

EXAMPLE-1: [1] module infers latch in the one of processes;
[2] there is an inference of combinational logic in another process => violation (detail-2).

```
module top ( Q1, D1, G, a, b, c  );

    ...

    always @( D1 or G )

        if  ( G )
            Q1  <= D1;

    always @*

        if  ( G )
            c = a;
        else
            c = b;

endmodule
```

Module "top" contains latchbased description mixed w ith other combinational logic. It is recommended to place latches in hierarchical modules and keep them separate from other descriptions to avoid the risk of generation an unintentional gated clock circuit.

Process infers pure latch(es). If latches are necessary, process could be placed in the separate hierarchical module.

EXAMPLE-2: [1] module infers latch in one of process;
[2] the other processes infer sequential logic => no violation.

```
module top (  G, D1, Q1, Q2 );

    ...

    always @( D1 or G )

        if  ( G )
```

```
            Q1 <= D1;

    always @ ( posedge G )

        if ( G )
            Q2 <= D1;

endmodule
```

Sequential process, flip-flop is inferred.

STARC_VLOG 2.4.1.3

RULE NAME	**Do not use latches with an asynchronous set/reset**	
MESSAGE	**{AsynchControlCount} asynchronous control signal(s) for latch "{ObjectName}" is detected. Do not use latches with asynchronous control.**	
	DETAIL-1	*Asynchronous set/reset control*
	DETAIL-2	*Asynchronous reset control*
	DETAIL-3	*Asynchronous set control*
PROBLEM DESCRIPTION	Latches with asynchronous set/reset are are unavailable at the most of ASIC vendors libraries. Additionally, inference of asynchronously controlled latch depends on logic synthesis tool => such descriptions should be avoided as much as possible.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans 'always' statements that infer latches. Violation is issued if asynchronous reset/set signal is present. Following conditions are true for such a signal: – it is used in the condition (ternary condition in the 'assign' construct or 'if'/'case' condition) – constant/parameter is assigned to latch signal under this condition: – violation (detail-1): {'x', 'z', '?'} is assigned – violation (detail-2): '0' is assigned – violation (detail-3): '1' is assigned Note-1: see chapter 2.2.1.1: latches inference principles Note-2: when reset/set detection depends on parameter => elaboration-time checking is required (message will contain instance name)	

EXAMPLE-1: [1] 'always' block describes latch with both asynchronous reset and set => violation;

[2] SET signal is specially omitted in the sensitivity list (synthesis tools ignore sensitivity lists and latch with asynchronous reset/set will be synthesized either with specified SET or omitted);

```
always @( RESET, G, SET ) begin
    if( RESET )
        Q <= 1'b0;
    else if( SET )
        Q <= 1'b1;
    else if ( G )
        Q <= DATA;

end
```

2 asynchronous control signal(s) for latch "Q" is detected. Do not use latches w ith asynchronous control

Asynchronous reset control

Asynchronous set control

EXAMPLE-2: [1] 'assign' describes latch with an asynchronous reset => violation

```
assign Q = ( RESET )? 1'b0 : ( G )? DATA;
```

1 asynchronous control signal(s) for latch "Q" is detected. Do not use latches w ith asynchronous control

Asynchronous reset control

EXAMPLE-3: [1] parameter is equal to 'x' and it is assigned in the asynchronous control branch => violation (elab-time, detail-1 – synthesis tool can map it either to preset or clear terminal of the latch);

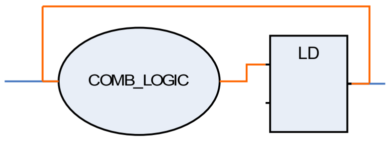
[2] non-constant/parameter is assigned to the latch signal in another asynchronous control branch => no violation (such description will be synthesized using multiplexer-logic)

```
parameter RESET_LEVEL = 1'bx;

always @( RESET, G, SET ) begin
    if( RESET )
        Q <= RESET_LEVEL;
    else if( SET )
        Q <= ASYNC_INP;
    else if ( G )
        Q <= DATA;

end
```

2 asynchronous control signal(s) for latch "Q" is detected. Do not use latches w ith asynchronous control

Asynchronous reset control

STARC_VLOG 2.4.1.4

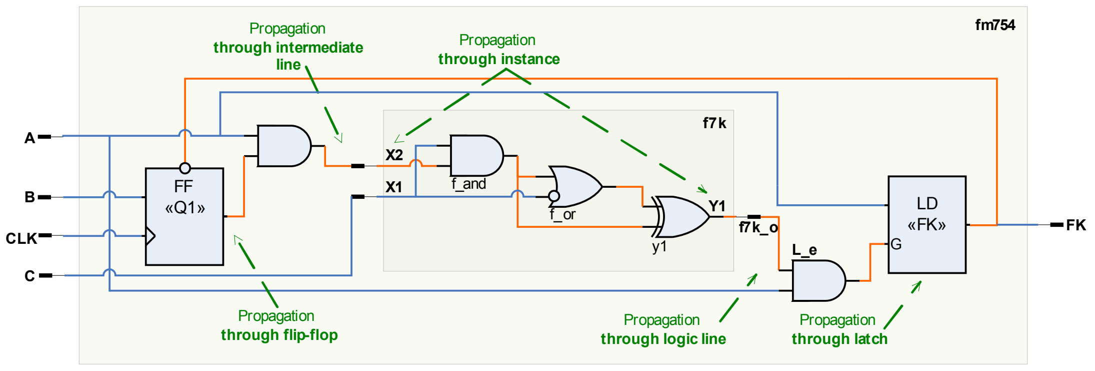
RULE NAME	**Avoid combinational feedback loops which contain latches**	
MESSAGE-1	**Asynchronous feedback containing latch(es) is detected on line "{FeedbackLineName}". Do not use feedback loops to avoid problems with timing analysis tools.**	
	Asynchronous feedback containing latch(es) is detected. Do not use feedback loops to avoid problems with timing analysis tools.	
	DETAIL-1	*Asynchronous loop propagates through combinational logic*
	DETAIL-2	*Asynchronous loop propagates through combinational logic line "{LineName}"*
	DETAIL-3	*Asynchronous loop propagates through {ObjectType} "{SignalName}" {PortType} input*
	DETAIL-4	*Asynchronous loop propagates through submodule instance "{InstanceName}" from port "{InputPortName}" to "{OutputPortName}"*
PROBLEM DESCRIPTION	During the period when gate signal is not active, latch becomes transparent and data from the D input goes through a latch towards to the output. So, if combinational feedback loop contains a latch, it means that asynchronous loop exists.  Static timing analysis tools are used to analyze the circuit operation speed for large designs. Combinational circuit feedbacks carry into the effect of asynchronous feedback loop and create problems with timing analysis. Therefore, circuits like this should be avoided regardless they could be acceptable in terms of their behavior.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans the design hierarchy to detect feedbacks that are propagated through combinational paths and/or latch(es): – see the rule 1.2.1.3 (behavior of this checker is almost the same(*)) – *(*) the difference between these checkers is:* – *2.4.1.4 triggers only on asynchronous loop that has latches in the propagation path* – *1.2.1.3 triggers on any asynchronous loop*	

EXAMPLE: [1] violation is reported in the detailed form: feedback propagation path is described (DETAILED_PROPAGATION_CHAIN = 1);

[2] consider the design hierarchy at the picture below;

[3] note, that all possible paths (see 1.2.1.3) of feedback propagation are demonstrated:

– through the submodule instance
– through the flip-flop and latch
– through the combinational logic line
– through the intermediate combinational logic line



fm754
Propagation through intermediate line
Propagation through instance
f7k
A
B
CLK
C
FF «Q1»
X2
X1
f_and
f_or
y1
Y1
f7k_o
L_e
LD «FK»
G
FK
Propagation through flip-flop
Propagation through logic line
Propagation through latch

```
module fm754( A, B, C, CLK, FK );

    input  A, B, C, CLK;
    output FK;
    reg    FK;

    wire   f7k_o, RST, L_e;
    reg    Q1;

    always @( A or L_e )
        if( L_e )
            FK = A;

    always @( posedge CLK or negedge FK )
        if( !FK )
            Q1 <= 1'b0;
        else
            Q1 <= B;

    f7k CMB_I0( .x2( A & Q1 ), .x1( C ), .y1( f7k_o ) );


    assign L_e = f7k_o & A;

endmodule

module f7k( x1, x2, y1 );

    input  x1, x2;
    output y1;

    wire   f_and, f_or;

    assign f_and = x1 & x2;
    assign f_or  = ~x1 | f_and;
    assign y1    = f_and ^ f_or;

endmodule
```

Instance "fm754". "STARC 2.4.1.4" Asynchronous feedback containing latch(es) is detected. Do not use feedback loops to avoid problems w ith timing analysis tools.

Asynchronous loop propagates through latch "FK" enable input

Asynchronous loop propagates through combinational logic

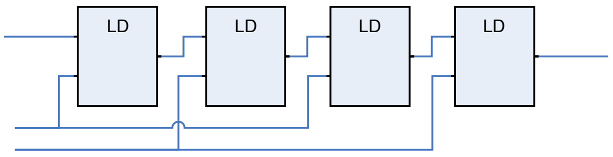
Asynchronous loop propagates through FF "Q1" asynchronous reset input.

Asynchronous loop propagates through combinational logic

Asynchronous loop propagates through submodule instance "fm754.CMB_I0" from port "x2" to "y1".

Asynchronous loop propagates through combinational logic line "L_e"

STARC_VLOG 2.4.1.5

RULE NAME	**In the same phase clock, do not use more than two layers of latches**	
MESSAGE	**Sequence of latches that have the same clock phase with latch "{LatchName}" is detected. Connect latches so that none of the successive ones have clock signals of the same phase to avoid errors in data transfers.**	
	DETAIL	*Latch "{HierLatchName}" has the same clock phase.*
PROBLEM DESCRIPTION	The use of latches should be restricted because it complicates timing analysis. But there are many cases where latches are used to solve the setup/hold time assurance problem (synchronous RAM, transmitting between asynchronous clocks). In latch-based designs, transfers are usually executed serially in clocks of different phases as shown at the picture. But note that more than two latches triggered by the same phase clock should not be connected in series. Because setup and hold time of latches connected in the sequence is not ensured in case of same enable signal. So the reason of latch usage (to solve the setup/hold time assurance problem) is not meet. 	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOUR	Checker scans design hierarchy for latches and verifies lines that are mapped to the gate pin: – if there are two latches that are connected (Q -> D) in series(*) *(*) combinational logic placed between latches is not considered and latches are treated as sequentially placed (series of latches)* – if latch enable signals are the same and have equal polarity => violation.	

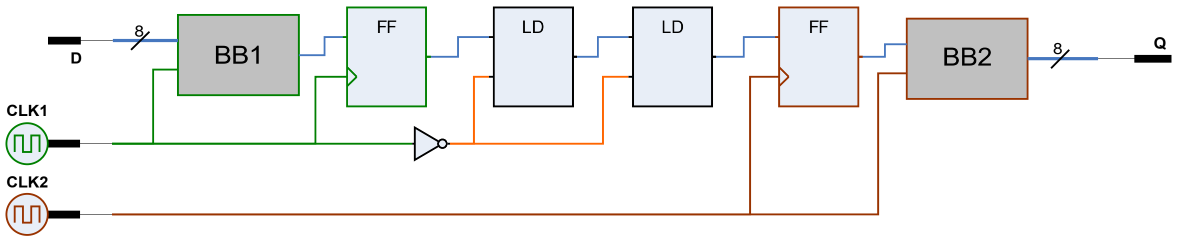
EXAMPLE-1: [1] if there are two latches that are connected in series;
[2] latch enable signals are the same and have equal polarity (~CLK) => violation.

```
module top( CLK1, CLK2, D, Q );

    input CLK1, CLK2;
    input [7:0] D;

    output [7:0] Q;

    assign clk_n = ~ CLK1;

    dff DFF_CLK1 ( .CLK( CLK1 ), .D( bb1_clk1_out ), .Q( ff_clk1_out ) );

    dff DFF_CLK2 ( .CLK( CLK2 ), .D( latch2_out ), .Q( ff_clk2_out ) );

    latch LD1 ( .G ( clk_n ), .D ( ff_clk1_out ), .Q ( latch1_out) );

    latch LD2 ( .G ( clk_n ), .D ( latch1_out ), .Q ( latch2_out) );

    bb1 BB1  ( .CLK( CLK1 ), .D( D ), .Q( bb1_clk1_out ) );

    bb2 BB2   ( .CLK( CLK2 ), .D( ff_clk2_out ), .Q( Q ) );

endmodule

//D flop-flop
module dff( CLK, D, Q );
```

```
     input CLK, D;
     output reg Q;

     always @(posedge CLK)
          Q <= D;

endmodule

//D-latch
module latch( G, D, Q );

     input G, D;
     output reg Q;

     always @(G, D)
          if ( G )
               Q <= D;

endmodule

// BB1 interface definition
module bb1 ( CLK, D, Q);
     input CLK;
     input [7:0] D;
     output Q;

endmodule


// BB2 interface definition
module bb2 ( CLK, D, Q);
     input CLK;
     input D;
     output [7:0] Q;

endmodule
```

Instance "top.LD1". Sequence of latches that have the same clock phase with latch "Q" is detected. Connect latches so that none of the successive ones have clock signals of the same phase to avoid errors in data transfers.

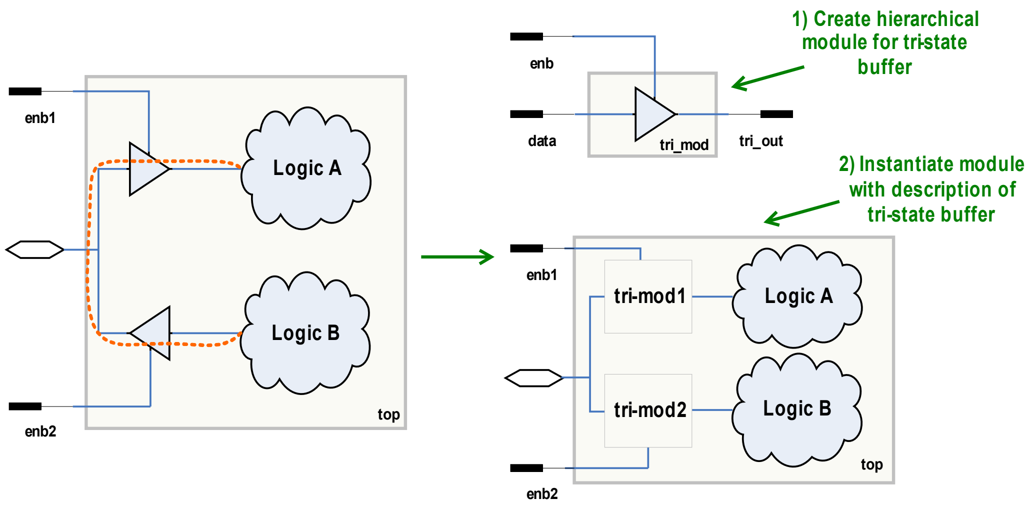
Latch "top.LD2.Q" has the same clock phase.



2.5 *Tri-state buffers*

2.5.1 Create modules for tri-state buffers

STARC_VLOG 2.5.1.1

RULE NAME		**Create modules for tri-state buffers**
MESSAGE-1		**Module "{ModuleName}" contains tristate inference mixed with other logic. Create a simple hierarchical module without any other logic to separate the tri-state descriptions from other code to avoid inconsistencies with timing analysis tools.**
	DETAIL	*Tri-state buffer is inferred.*
MESSAGE-2		**Module "{ModuleName}" contains inferences of tri-state buffers from different signals. Describe different tristates in separate hierarchical modules.**
PROBLEM DESCRIPTION		When tri-state buffers are used in the description, there are cases where paths, which do not require timing analysis, are analyzed. To avoid this problem, the tri-state buffer should be made in a hierarchical module and delays for it should be specified. A tri-state buffer module should be created and then instantiated as shown in the picture below.  enb1 Logic A Logic B enb2 top enb data tri_mod tri_out 1) Create hierarchical module for tri-state buffer 2) Instantiate module with description of tri-state buffer enb1 tri-mod1 Logic A tri-mod2 Logic B top enb2 The following order is recommended for synthesis of design containing tri-state buffers: 1. compile the tri-state buffer block by itself first; 2. before synthesis of the design, specify input and output delays of the tri-state that has become a gate; 3. synthesize the design.
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR		Checker scans each module for tri-state inferences: – if single tri-state is described along with any other non-tri-state logic or/and with combinational logic on tri-state output => violation (message-1); – if multiple tri-state buffers are described (tri-states inferred from signals that are not members of single vector) => violation (message-2); – if multiple tri-state buffers are described along with non-tri-state logic or/and combinational logic on tri-state(s) output => violation (message-1 + message-2).

EXAMPLE-1: [1] module contains tristate inference;
[2] module also contains FF inference;
[3] FF output is connected to tristate input => violation (message-1).

```
module top ( din, enb, clk, dout );

     input din, enb, clk;
     output  dout;

     reg tmp;
     wire wtmp;

     always @ (posedge clk)
          tmp <= din;

     assign wtmp = tmp;

     assign dout = enb ? wtmp : 1'bz;

endmodule
```

Module “top” contains tristate inference mixed w ith other logic. Create a simple hierarchical module w ithout any other logic to separate the tri-state descriptions from other code to avoid inconsistencies w ith timing analysis tools.

Tri-state buffer is inferred.

EXAMPLE-2: [1] module contains tristate inference for different signals => violation (message-2).

```
module top (in1, in2, in3, in4, enb,dout1,dout2);

     input in1,in2,in3,in4,enb;
     output [1:0] dout1;
     output [1:0] dout2;

     assign {dout1, dout2} = enb ? {in1, in2, in3, in4} : 4'bz;

endmodule
```

Module “top” contains inferences of tri-state buffers from different signals. Describe different tristates in separate hierarchical modules.

EXAMPLE-3: [1] module contains tristate inference along with another logic;
[2] combinational logic is connected to the inputs of the tristate => no violation.

```
module top (in1, in2, enb1, enb2, dout);

     input din1,din2,enb1,rnb2;
     output dout;

     assign dout = ( enb1 | enb2 ) ? ( din1 & din2 ) : 1'bz;

endmodule
```

STARC_VLOG 2.5.1.2

RULE NAME	**Do not describe logic in conditional expressions to infer tri-state**	
MESSAGE-1	**Logic is detected at the control input of tri-state "{ThreeStateName}". It is recommended to describe only simple scalar signals in conditional expressions for tri-state inferences to avoid malfunctions during the logic synthesis.**	
MESSAGE-2	**Control input of tristate "{ThreeStateName}" is signal, that is driven by logic. It is recommended to describe only simple scalar signals in conditional expressions for tri-state inferences to avoid malfunctions during the logic synthesis.**	
PROBLEM DESCRIPTION	Logic should not be described at the control of tri-state because there is a risk that the output is not able to be turned on and off with the proper timing due to potential changes to the logic sequence in logic synthesis. If a hazard enters the tri-state buffer selector signal logic, the output will collide with other output drives which could cause malfunctions or increase power consumption. Vector control is treated as logic too, because logic is synthesized to transform vector signal to one-bit tri-state control Therefore, use only simple scalar signal names in the tri-state buffer selection signal.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker scans conditional statements of if / case / ternary constructs that infer tri-state buffers: – if conditional expression contains any logic – expression contains arithmetic / logical operators or it is a vector => violation (message-1) – expression contains simple signal, but it is driven by another logic => violation (message-2) Checker scans built-in primitives (bufif0, bufif1, notif0, notif1): – if parameter which corresponds to control signal contains any logic => violation (same requirements for displaying message-1 or message-2) Note-1: buffers and inverters are not considered as logic Note-2: when two types of violation (with same tri-state) occurs in single case => only one violation (message-1)	

EXAMPLE-1: [1] 'if' construct infer tri-state;
[2] selection expression contains logical operator => violation (message-1)

```
always @ (enb1 or enb2 or in1)

    if ( enb1 & enb2 )
        out1<=1'bz;
    else
        out1<=in1;
```

Logic is detected at the control input of tri-state "out1". It is recommended to describe only simple scalar signals in conditional expressions for tri-state inferences to avoid malfunctions during the logic synthesis.

EXAMPLE-2: [1] 'if' construct infer tri-state;
[2] selection expression contains simple signal, but it is driven by another logic => violation (message-2)

```
always @ (enb1 or enb2 or in1)
    begin

        enb <= enb1 & enb2;

        if ( enb )
            out1<=in1;
        else
            out1<=1'bz;
    end
```

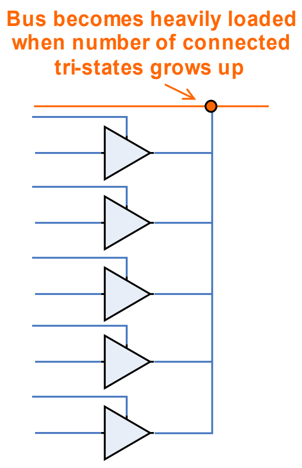
Control input of tristate "out1" is signal, that is driven by logic. It is recommended to describe only simple scalar signals in conditional expressions for tri-state inferences to avoid malfunctions during the logic synthesis.

EXAMPLE-3: [1] built-in primitives bufif0 is used;

[2] parameter which corresponds to control signal is vector, but it is truncated when instantiated => no violation

```
module top (..., enb1, ...);

    ...

    input [1:0] enb1;

    bufif0 Buff1 ( out1, in1, enb1 ) ;

    ...

endmodule
```

STARC_VLOG 2.5.1.4

RULE NAME	**Specify up to five tri-state buffer connectors at most**	
MESSAGE	**{ObjectClass} "{ObjectName}" is driven by {ThreeStateCount} tri-state buffers. Output drivers connected to tri-state buses should be {TRISTATE_DRIVER_COUNT} or less to avoid heavily loading.**	
	DETAIL	*Tri-state driver is detected.*
PROBLEM DESCRIPTION	In the case of a net with many connections and when connecting long distance with bidirectional buses, the current layout tools tend to drastically increase wire area. Even if it seems to be possible using of a shorter wire, the tools significantly increase the wire area. For performance purposes it is highly recommended to design single-direction wires, without a bidirectional bus which usually has a heavy load. Circuit overheat danger also occurs if all tri-states become enabled in the same time. When it is necessary to use a bidirectional bus, output drivers count connected to the tri-state bus should be five or less. This should avoid allowing the bidirectional bus to become heavily loaded.  Bus becomes heavily loaded when number of connected tri-states grows up	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker verifies object's drivers: – if object has multiple tri-state drivers: – if tri-states count is greater then TRISTATE_DRIVER_COUNT => violation. Note-1: parameter TRISTATE_DRIVER_COUNT value is defined in configuration file (default value is 5). Note-2: {ObjectClass} is defined by the following table:	

{ObjectClass}	
module port of 'output' mode	Module output port
module of 'inout' mode	Module inout port
net	Wire

EXAMPLE-1: [1] consider an example shown at the picture below;

[2] all described tri-states considered as directly connected to the bus which spans over hierarchy;

[3] top level module output has 5 tri-state drivers connected from instances and tri-state described at the top level => violation;

[4] second instance contains two tri-state connected to the same output and two tri-state outputs come from first instance => violation.

```
module top ( data1, data2, data3, data4, data5, data6, enb1, enb2, enb3, enb4, enb5, dout );

    input data1, data2, data3, data4, data5, enb1,enb2,enb3,enb4,enb5;
    output dout;
    wire net1;

    tri_mu_connect inst_A ( .data1(data1), .data2(data2),
                            .enb1(enb1), .enb2(enb2), .tri_out(net1));
```

Instance “top”. Module output port “d_out” is driven by 5 tri-state buffers. Output drivers connected to tri-state buses should be 3 or less to avoid heavily loading.

```
    tri_mul_connect_mixed inst_B (.data1(data3), .data2(data4),
                                  .enb1(enb3),.enb2(enb4), .tri_out(d_out), .in_wire(net1));

    assign d_out = enb5 ? data5 : 1'bz;

endmodule
```

Tri-state driver is detected.

Tri-state driver is detected.

Tri-state driver is detected.

Tri-state driver is detected.

```
module tri_mul_connect ( data1, data2, enb1, enb2, tri_out);

    input data1,data2,enb1,enb2;
    output tri_out;

    assign tri_out = enb1 ? data1 : 1'bz;
    assign tri_out = enb2 ? data2 : 1'bz;

endmodule
```

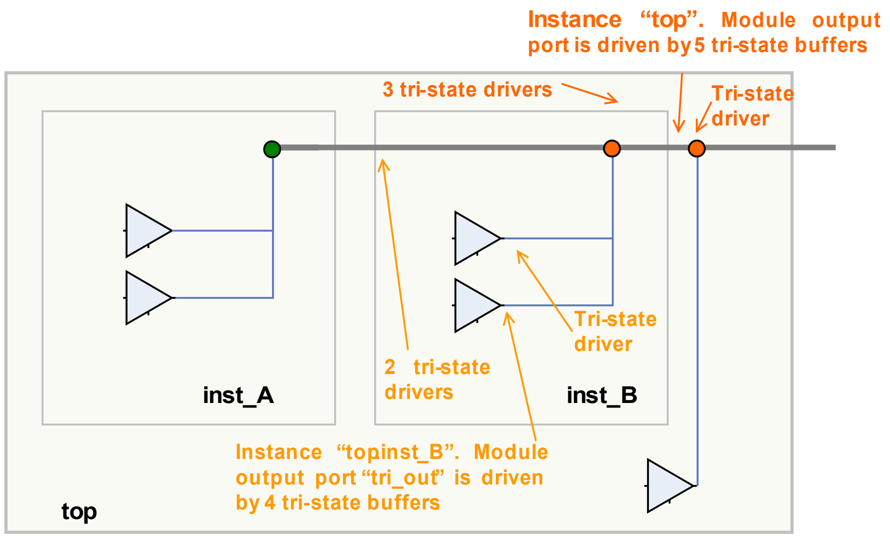
Instance “top”. Module output port “d_out” is driven by 5 tri-state buffers. Output drivers connected to tri-state buses should be 3 or less to avoid heavily loading.

```
module tri_mul_connect_mixed ( data1, data2, in_wire, enb1, enb2, tri_out );

    input data1, data2, enb1, enb2, in_wire;
    output tri_out;

    assign tri_out = enb1 ? data1 : 1'bz;
    assign tri_out = enb2 ? data2 : 1'bz;
    assign tri_out = in_wire;

endmodule
```

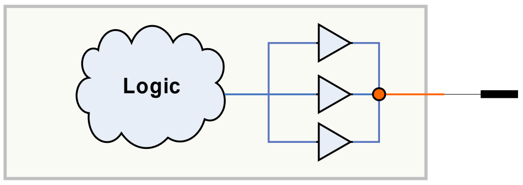
Tri-state driver is detected.

Tri-state driver is detected.

Tri-state driver is detected.


Instance “top”. Module output port is driven by 5 tri-state buffers
3 tri-state drivers
Tri-state driver
Tri-state driver
2 tri-state drivers
inst_A
inst_B
Instance “topinst_B”. Module output port “tri_out” is driven by 4 tri-state buffers
top

STARC_VLOG 2.5.1.5

RULE NAME	**Do not connect two or more outputs other than tri-state buffers even under the same conditions**	
MESSAGE	**{ObjectClass} "{ObjectName}" is driven by {DriverCount} non-tri-state drivers. Two or more output drivers other than tri-state ones should not be used in the RTL description.**	
	DETAIL	*Non-tri-state driver is detected.*
PROBLEM DESCRIPTION	To strengthen the drive capacity of the output, the two cells of BUF, INV or AND, etc. are sometimes connected simultaneously to the same net. Such an adjustment should be done during layout stage, and the descriptions which connect two or more output drivers other than tri-state ones should not be included in the RTL model. 	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies object drivers: – if object has multiple non-tri-state drivers => violation Note: for {ObjectClass} table see 2.5.1.4.	

EXAMPLE-1: [1] consider an example shown at the picture below;

[2] all non-tri-states drivers considered as directly connected to the common bus bus which spans over hierarchy and is analyzed at each hierarchy level;

[3] top level module output has 3 non-tri-state drivers connected from instances and from the top module input => violation;

[4] first instance ("mul_connect") contains two inputs directly connected to the same output => violation;

[5] second instance ("mul_connect_tri") contains two tri-state connected to the same output and multiconnection of two non-tri-state outputs come from first instance => violation.

```
module top ( in1, in2, in3, in4, in5, enb1, enb2, glbl_out );

     input in1, in2, in3, in4, in5, enb1, enb2;
     output glbl_out;

     wire mul_out;

     mul_connect ( .in1(in1), .in2(in2), .mul_out1(net1) );

     mul_connect_tri mul_connect_tri ( .in1(in3), .enb1(enb1), .in2(in4), .enb2(enb2),
                                       .in3(mul_out), .mul_tri_out(glbl_out));

     assign glbl_out = in5;

endmodule
```

Module output port "glbl_out" is driven by 3 non-tri-state drivers. Tw o or more output drivers other than tri-state ones should not be used in the RTL description.

Non-tri-state driver is detected.

Non-tri-state driver is detected.

```
module mul_connect( in1, in2, mul_out);

     input in1,in2;
     output mul_out;

    assign mul_out = in1;
    assign mul_out = in2;

endmodule
```

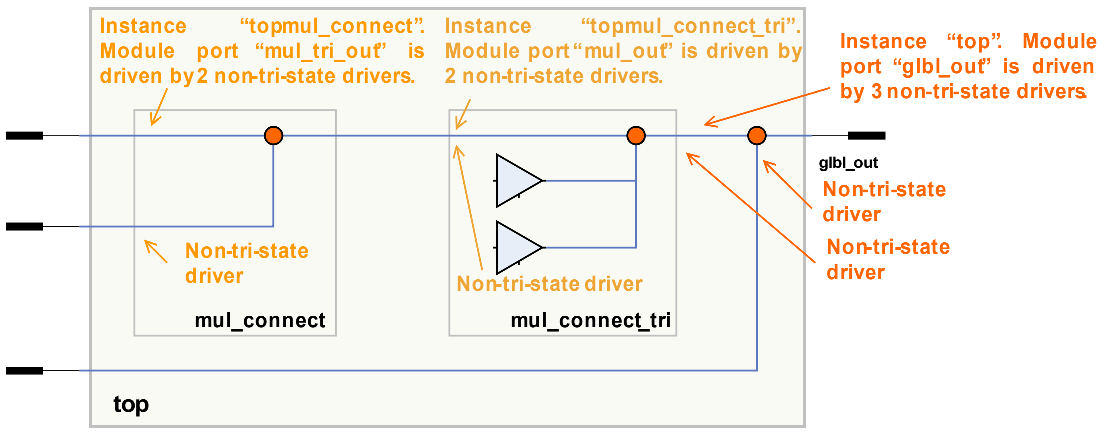
Module output port “mul_out” is driven by 2 non-tri-state drivers. Tw o or more output drivers other than tri-state ones should not be used in the RTL description.

Non-tri-state driver is detected.

```
module mul_connect_tri ( in1, enb1, in2, enb2, in3, mul_tri_out );

     input in1, enb1, in2, enb2, in3;
     output mul_tri_out;

     assign mul_tri_out = enb1?in1:1'bz;
     assign mul_tri_out = enb2?in2:1'bz;

     assign mul_tri_out = in3;

endmodule
```

Module output port “mul_out” is driven by 2 non-tri-state drivers. Tw o or more output drivers other than tri-state ones should not be used in the RTL description.

Non-tri-state driver is detected.


Instance “topmul_connect”. Module port “mul_tri_out” is driven by 2 non-tri-state drivers.
Instance “topmul_connect_tri”. Module port“mul_out” is driven by 2 non-tri-state drivers.
Instance “top”. Module port “glbl_out” is driven by 3 non-tri-state drivers.
glbl_out
Non-tri-state driver
Non-tri-state driver
Non-tri-state driver
Non-tri-state driver
mul_connect
mul_connect_tri
top

STARC_VLOG 2.5.1.6

RULE NAME	**inout should not directly be connected to input/output**	
MESSAGE-1	**Inout port "{ObjectName}" should not be directly connected to input port "{PortName}".**	
MESSAGE-2	**Inout port "{ObjectName}" should not be directly connected to output port "{PortName}".**	
PROBLEM DESCRIPTION	Do not connect a port declared as inout to ports with direct input or output declarations. Paths that do not exist in the RTL description are generated after logic synthesis is performed, and inconsistencies will occur in simulation results in RTL and at the gate level.	
	LEVEL	RULE
MESSAGE--3	**Submodule output port "{OutputPortName}" is driven by input port "{InputPortName}". Avoid meaningless port connections.**	
MESSAGE--4	**Submodule output port "{OutputPortName}" is driven by constant. Avoid meaningless port connections.**	
MESSAGE--5	**Submodule output port "{OutputPortName}" is driven by expression. Avoid meaningless port connections.**	
PROBLEM DESCRIPTION	Connections of an input port, a constant or an expression is meaningless, but could be made by a mistake. Such connections lead to unexpected design behavior and should be avoided. **Note:** this extension is added by Aldec, Inc.	
	LEVEL	RECOMMENDATION 1

CHECKER BEHAVIOR	1) Checker scans description for assignments from input port to inout:
	– if any assignments are detected => violation (message-1)
	2) Checker scans description for assignments from inout port to output port:
	– if any assignments are detected => violation (message-2)
	3) Extension: checker external ports mapped to ports in instance port maps:
	– if input drives output => violation (message-3);
	– if constant is connected to output => violation (message-4);
	– if expression is connected to output => violation (message-5).
	Note-1: checker verifies the following assignment types:
	– continuous assignments (assign)
	– procedural assignments (=, <=)
	Note-2: checker verifies the following assignment structure:
	– direct assignments:
	`inout = input`
	– assignments through temporary signal(s), such as:
	`temp1 = input;`
	`temp2 = temp1;`
	`...`
	`inout = tempN;`	

EXAMPLE-1: [1] input is assigned to the inout throw the temporary signals => violation (message-1)

```
module top (in, io, out);

    input in;
    inout io;
    output out;
    wire w;

    assign io = w;
    assign w = in;
    assign out = in;

    ...

endmodule
```

Inout port “io” should not be directly connected to input port “in”.

EXAMPLE-2: [1] inout is directly assigned to output port => violation (message-2)

```
module top (clk, din, dout);

    input clk;
    inout din;
    output reg dout;

    always @(posedge clk)

        dout = din;

endmodule
```

Inout port “din” should not be directly connected to output port “dout”.

EXAMPLE-3: [1] if input port drives output port => violation (message-3) ;
[2] expression is connected to output port => violation (message-5).

```
module top( I1, I2, O1, O2 );

    input  [3:0] I1, I2;
    output [3:0] O1, O2;

    sub instance_1 (
        .I1( 0  ),
        .I2( I2 ),
        .O1( I1 ),
        .O2( O1 )
    );

    sub instance_2 (
        .I1( 0  ),
        .I2( I2 ),
        .O1( I1 & I2 ),
        .O2( O2 )
    );

endmodule


// Submodule declaration
module sub( I1, I2, O1, O2 );

    input  [3:0] I1, I2;
    output [3:0] O1, O2;

    ...

endmodule
```

Submodule output port “O1” is driven by input port “I1”. Avoid meaningless port connections.

Submodule output port “O1” is driven by expression. Avoid meaningless port connections.

STARC_VLOG 2.5.1.7

RULE NAME	**Tri-state output should not be used in a conditional expression of an if statement**	
MESSAGE	**Signal "{SignalName}" is driven by the output of tri-state. It is not recommended to use tri-state signals for conditional expression of 'if' statements.**	
PROBLEM DESCRIPTION	For tri-state signals, propagation of ’x’ should be considered in order to match simulation results in RTL and gate level. When a signal value that includes ’z’ is used in a conditional expression of if statements, else item is executed. Normally, value ’z’ becomes an ’x’ after passing through a logic gate, but in this case it becomes a fixed value. Therefore, inconsistencies can occur between RTL and gate level with the propagation of 'x'. Avoid using tri-state signals for conditional expression of if statements.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects all tri-states in the module: – if tri-state output is described in the conditional expression of 'if' statements => violation	

EXAMPLE-1: [1] tri-state is inferred by built-in primitive 'notif1';

[2] tri-state output is described in the conditional expression of 'if' statements => violation

```
notif1 Buff1 (out1,in1,enb);

always @ (out1)

    if (out1)
        out2<=1'b0;
    else
        out2<=1'b1;
```

Signal “out1” is driven by the output of tri-state. It is not recommended to use tri-state signals for conditional expression of 'if' statements.

STARC_VLOG 2.5.1.8

RULE NAME	**Tri-state output should not be used in a selection expression of a case statement that is not assigned ’x’ as the default clause**	
MESSAGE-1	**Signal "{ThreeStateSignalName}" is driven by the output of tri-state whereas 'default' clause is not specified. If it is necessary to use tri-state output in 'case' selection expression, assign 'x'-es in the 'default' clause to enable 'z'-value propagation.**	
MESSAGE-2	**Signal "{ThreeStateSignalName}" is driven by the output of tri-state whereas not all signals are assigned with 'x'-es in the 'default' clause. If it is necessary to use tri-state output in 'case' selection expression, assign 'x'-es in the 'default' clause to enable 'z'-value propagation.**	
	DETAIL	*Signal “{SignalName}” is not assigned with 'x'(-es) in 'default' clause.*
PROBLEM DESCRIPTION	For tri-state signals, propagation of ’x’ should be considered in order to match simulation results in RTL and gate level. When a tri-state signal is entered in a selection expression of a case statement that is not assigned 'x' in the default clause, 'x' is not propagated. Avoid using tri-state signals for selection expression of case statements.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects all tri-states in the module and scans description for 'case' statements: – if any signal is assigned under control of tri-state output from 'case' branch – if there is no 'default' clause => violation (message-1) – if 'default' clause is specified, but not all signals are assigned 'x'-es within the clause => violation (message-2)	

EXAMPLE-1: [1] signal is assigned under control of tri-state output from 'case' branch;
[2] there is no 'default' clause at the 'case' statement => violation (message-1)

```
always @(in1 or enb)

    out1 <= enb ? in1 : 1'bz;

always @ (out1)

    case(out1)
        0: out2 <= 1'b0;
        1: out2 <= 1'b1;
    endcase
```

Signal “out1” is driven by the output of a tri-state w hereas 'default' clause is not specified. If it is necessary to use tri-state output in 'case' selection expression, assign 'x'-es in the 'default' clause to enable 'z'-value propagation.

EXAMPLE-2: [1] signal is assigned under control of tri-state output from the 'case' branch;
[2] 'default' clause is specified, but one bit of the signal is assigned with 'x'-es within the clause => violation (message-2)

```
reg [1:0] out2, tmp1, tmp2;

always @(in1 or enb)

    out1 <= enb ? in1 : 1'bz;

always @ (out1)

    case(out1)
        0: out2 <= tmp1;
        1: out2 <= tmp2;
        default: out2[0] <= 1'bx;
    endcase
```

Signal “out1” is driven by the output of tri-state w hereas not all signals are assigned 'x'-es in the 'default' clause. If it is necessary to use tristate output in 'case' selection expression, assign 'x'-es in the 'default' clause to enable 'z'-value propagation.

Signal “out2[1]” is not assigned 'x'(-es) in 'default' clause.

STARC_VLOG 2.5.1.9

RULE NAME	**Tri-state output should not be entered in the selection expression of casex or casez statements**	
MESSAGE	**Selection expression of 'casex(z)' statement contains signal "{ThreeStateSignalName}" that is driven by the output of tri-state. Such description is recognized as don't care and should not be used to avoid difficulties with 'x'-value propagation.**	
PROBLEM DESCRIPTION	When tri-state output is entered in a selection expression of 'casex' or 'casez' statements, 'z' is recognized as don't-care and it is difficult to propagate 'x'. Tri-state output should not be used for selection expression of 'casex' and 'casez' statements.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects all tri-states in the module: – if tri-state output is described in the selection expression of 'casex' or 'casez' statements => violation	

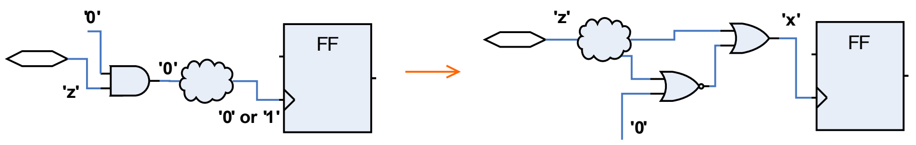
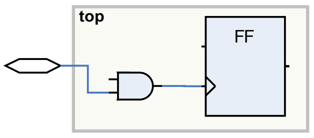
EXAMPLE-1: [1] case statement infer tri-state;
[2] tri-state output is described in the selection expression of 'casez' => violation

```
case ( enb )
    0: out2<=in1;
    1: out2<=1'bz;
    default:out2<=1'bx;
endcase

casez( out1 )
    3'b000: out2<=in1;
    3'b011: out2<=1'b1;
    default: out2<=1'bx;
endcase
```

Selection expression of 'casex(z)' statement contains signal “out2” that is driven by the output of a tri-state. Such description is recognized as don't care and should not be used to avoid difficulties w ith 'x'-value propagation.

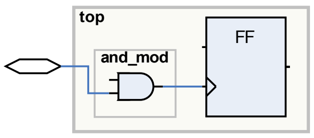
2.5.2 Consider high-impedance propagation in tri-state buses

STARC_VLOG 2.5.2.1

RULE NAME	**Create a block for a tri-state buffer and a block for an input cell connected directly from a bidirectional bus**	
MESSAGE	**Inout port "{PortName}" is directly connected to control line(s). Create a block for an input cell that is connected directly from a bidirectional bus to prevent 'x' propagation.**	
	DETAIL-1	*Inout port is directly connected to {PortType} input of {ObjectType} "{ObjectName}".*
	DETAIL-2	*Inout port is directly connected to select input of multiplexer.*
PROBLEM DESCRIPTION	Bidirectional buses with tri-state buffers may become 'z' momentarily. In that case, RTL code is optimized to block the propagation of 'z' by additional logic. Consequently, the RTL simulation result and the synthesized gate level simulation result may differ. For example (see the picture below), a circuit description using an AND gate to control propagation is added to prevent 'z' propagation in the RTL. As a result of synthesis however, the logic circuit containing the AND gate is optimized, the logic for preventing 'z' propagation is changed to OR, and propagation of unknown value 'x' may occur, without 'z' propagation being prevented. Since the logic connected to the control signal may be changed as a result of optimization, some problems may occur. Therefore, it is necessary to separate the description of this logic from the tri-state bus description, as well as from the description of the tri-state bus readers, by encapsulating this logic into its own separate module. This should prevent risky synthesis optimization.  '0' 'z' '0' '0' or '1' FF 'z' 'x' FF '0'	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker scans 'inout' ports: – if port is connected to: – clock input and asynchronous control inputs of FF; – asynchronous control inputs of latch; – enable input of a tri-state; – select input of MUX; – and if the port is connected: – directly (including the connection through instances, buffers and inverters) => violation – through combinational logic on the same hierarchy level (*) with the port => violation – *(*) to be on the same hierarchy level means to be described in the same module where 'inout' port is declared:* – *logic on the same hierarchy level:* 	

RULE NAME	**Create a block for a tri-state buffer and a block for an input cell connected directly from a bidirectional bus**

– *logic on another hierarchy level:*



Note-1: in case of violation per multiplexer – detail-2 is displayed.
Note-2: possible values for {ObjectType} and {PortType} strings:

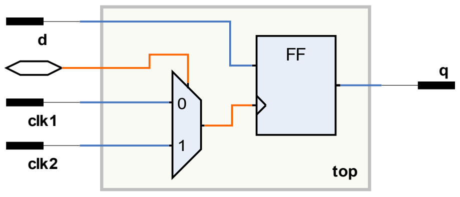
{ObjectType}	***{PortType}***
FF	enable
	asynchronous reset
	asynchronous set
	clock
latch	enable
	asynchronous reset
	asynchronous set
tri-state buffer	enable

EXAMPLE-1: [1] consider the picture below;
[2] 'inout' port is connected directly to select input of MUX => violation.

```
module top (clk1, clk2, sel, d, q);

    input clk1, clk2, d;
    inout sel;
    output reg q;

    wire sel_clk;

    assign sel_clk = sel ? clk1 : clk2;

    always @ ( posedge sel_clk )
        q <= d;

endmodule
```

Instance "top". Inout port "sel" is directly connected to control line(s). Create a block for an input cell that is connected directly from a bidirectional bus to prevent 'x' propagation.

Inout port is directly connected to select input of multiplexer.



2.6 always construct description that takes circuit structure into account

2.6.1 Describe taking the circuit structure into account

STARC_VLOG 2.6.1.2

RULE NAME		**Use intermediate variables when the same logic is used in more than two places**
MESSAGE		**The same selection condition ({SelCondition}) is detected {ExpressionsCount} times in different conditional statements. Use intermediate variables to share the same logic between different constructs to avoid generation of the same logic several times with logic synthesis tools.**
	DETAIL	*Exactly the same selection condition is detected.*
PROBLEM DESCRIPTION		To decrease the number of output signals in one always construct, selection conditions with exactly the same contents are described in two or more always constructs. This logic should be shared using an intermediate variable, except for simple logic. Especially, when using an arithmetic operator with 5 or more bits or a relational operator for conditional expressions – a logic synthesis tool may not share it. By describing it twice in two always constructs, the area will be doubled in size. Therefore, equal logic should be used with caution. It is recommended to assign the shared logic output to an intermediate variable, and then use it in both processes.
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR		Checker verifies conditional expressions from 'if' and ternary conditional statements within current module: – if there are two and more equal conditional expressions within different statements => violation Note-1: context for the rule is following: 'always' statements, continuous assignments and module instantiations. Note-2: commutative operators are considered properly: &&, ||, +, *, ==, !=, ===, !==, &, |, ^ (for example: expressions 'a && b' and 'b && a' are equal).

EXAMPLE-1: [1] there are two equal conditional expressions within different statements => violation.
Note: operator “|” is commutative.

```
always @( * )
     begin
          if ( ctrl1 | ctrl2 )
               ...
     end

assign tmp = ( ctrl2 | ctrl1 ) ? ...
```

The same selection condition (ctrl1 | ctrl2) is detected 2 times in different conditional statements. Use intermediate variables to share the same logic betw een different constructs to avoid generation of the same logic several times w ith logic synthesis tools.

Exactly the same selection condition is detected.

EXAMPLE-2: [1] there are two equal conditional expressions within the same statement => no violation.

```
always @( * )
     begin
          if ( ctrl1 | ctrl2 )
               ...
          else if ( ctrl1 | ctrl2 )
     end
```

STARC_VLOG 2.6.1.3

RULE NAME	**The number of signal outputs from one always construct should be five or less, if possible. The number of outputs should be limited to 15 at most**	
MESSAGE-1	**'always' construct has {OutputsCount} outputs. The number of signal outputs from one 'always' construct should be {OUTPUTS_RECOMMENDED} or less, if possible.**	
MESSAGE-2	**'always' construct has {OutputsCount} outputs. The number of signal outputs from one 'always' construct should be limited to {OUTPUTS_MAX} at most.**	
MESSAGE-3	**Number of 'always' construct outputs exceeds recommended one. The number of signal outputs from one 'always' construct should be {OUTPUTS_RECOMMENDED} or less, if possible.**	
MESSAGE-4	**Number of 'always' construct outputs exceeds maximum recommended one. The number of signal outputs from one 'always' construct should be limited to {OUTPUTS_MAX} at most.**	
PROBLEM DESCRIPTION	It is not recommended to describe too many output signals in a single always construct. If possible, five or less is recommended. If such restriction cannot be kept the number of outputs should be limited to 15 at most. However, it is not problematic to describe any number of (numerous) descriptions within the same always construct without any logic or signals which generate the same logic. RTL descriptions must be checked when unintended simulation results are obtained. If the output signal structure of each always construct is understood, descriptions that consider the circuit structure facilitate checking of the signal flow by proceeding with the debugging while monitoring it.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker counts the number of signals assigned (with blocking or non-blocking assignment type) in the synthesized 'always' construct: – if COUNT_OUTPUT_NUM = "1" – if number is greater than OUTPUTS_RECOMMENDED => violation (message-1); – if number is greater than OUTPUTS_MAX => violation (message-2); – if COUNT_OUTPUT_NUM = "0" – if number is greater than OUTPUTS_RECOMMENDED => violation (message-3); – if number is greater than OUTPUTS_MAX => violation (message-4) Context: – for edge-controlled always, signal is treated as output if it is: – output port of a module – read before assignment – read in another process – for level-controlled always, signal is treated as output if it is: – output port of a module – read in another process and it is an assignment with logic or inside a statement Note-1: when vector is assigned only bits treated as output are counted. Note-2: bits of vector defined as outputs are compacted to slices if they are consecutive and every slice is counted as separate output. Note-3: values of parameters OUTPUTS_RECOMMENDED and OUTPUTS_MAX are defined in configuration file (default values are 5 and 15 respectively).	

EXAMPLE-1: [1] number of outputs from the first 'always' construct exceeds OUTPUTS_RECOMMENDED parameter value;

[2] parameter COUNT_OUTPUT_NUM is set to 1 => violation (message-1);

[3] number of outputs from the second 'always' construct is equal to OUTPUTS_RECOMMENDED parameter value => no violation.

Note: parameter OUTPUTS_RECOMMENDED is set to 2 to simplify the example.

```
module top(clk,d1,d2,d3,in1,in2,in3,in4,q1,q2,q3);

     input clk,d2,d3,in1,in2,in3,in4;
     input [1:0] d1;
     output reg [1:0] q1;
     output reg q2,q3;

     reg tmp1,tmp2,tmp3,sel,en;
```

'alw ays' construct has 3 outputs. The number of signal outputs from one 'alw ays' construct should be 2 or less, if possible.

```
     always @( posedge clk )
          begin
               q1 <= d1;                 //module output
               q2 <= en ? d2 : d3;       //module output
               en <= sel ? tmp1 : tmp2;  //signal is read before assignment
          end

     //green comments mark signals not treated as process outputs
     always @(*)
          begin
               q3 = in1 + in2;    //module output
               sel = in1 | in2;   //signal is read in another process and assigned with logic
               tmp1 = in3;        //signal is read in another process but assigned without logic
               tmp2 = in4;        //signal is read in another process but assigned without logic
               tmp3 = in1 * in2;  //signal is neither a port, nor read in another process
          end

endmodule
```

STARC_VLOG 2.6.1.4

RULE NAME	**The number of lines in an always construct should be up to 20. 2000 lines at most.**	
MESSAGE-1	**'always' statement has “{LinesCount}” lines. Recommended number of lines is {MAX_LINES_RECOMMENDED} or less with the exceptional case of {MAX_LINES_ALLOWED} at most.**	
PROBLEM DESCRIPTION	One always construct should be a single execution unit and should not contain a large number of lines. The logic size to be generated is not in proportion to the number of lines in an always construct. However, the number of lines should be 300 or less with the exceptional case of 2000 at most to increase description readability.	
	LEVEL	RECOMMENDATION 3
MESSAGE-2	**'always' statement has “{LinesCount}” lines. The number of lines should be limited by {MAX_LINES_ALLOWED}.**	
PROBLEM DESCRIPTION	If describing large number of lines in one 'always' construct can not be avoided the number should be 2000 at most.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects number of lines in an 'always' statements by following subtraction: *“SourcePointrer(last token in an 'always' statement) - SourcePointer('always' keyword) + 1”*: – if number of lines is greater than MAX_LINES_RECOMMENDED and less than MAX_LINES_ALLOWED parameter => violation (message-1); – if number of lines is greater than parameter MAX_LINES_ALLOWED=> violation (message-2). Note: values of parameters MAX_LINES_RECOMMENDED and MAX_LINES_ALLOWED are specified in configuration file (300 and 2000 by default).	

EXAMPLE-1: [1] 'always' statement has 7 lines;

[2] the number is greater than MAX_LINES_RECOMMENDED parameter value => violation (message-1).

Note: MAX_LINES_RECOMMENDED parameter value is set to 5 to simplify the example.

```
always @( * ) begin // line #7

    // define AND
    Y1 = A & B;
    // define OR
    Y2 = A | B;
end                 // line #13
```

'alw ays' statement has “7” lines. Recommended number of lines is 5 or less w ith the exceptional case of 2000 at most.

2.6.2 Avoid defining multiple output signals in a single always construct

STARC_VLOG 2.6.2.1

RULE NAME	**Do not describe more than one if or case in one always construct**	
MESSAGE	**Do not describe more than one 'if' statement or 'case' statement in a single 'always' block**	
	DETAIL-1	*'if' statement in the 'always' block*
	DETAIL-2	*'if' statement in the same 'always' block*
	DETAIL-3	*'case' statement in the 'always' block*
	DETAIL-4	*'case' statement in the same 'always' block*

PROBLEM DESCRIPTION

Following example describes a complex 'always' statement:

```
always @( SEL or X_IN or Z_IN) begin
   if( SEL )                                          block (1)
      TMP_STORAGE = X_IN & Z_IN;
   else
      TMP_STORAGE = X_IN | Z_IN;                      block (2)
   LO_B = TMP_STORAGE[31];
   if( TMP_STORAGE[7:0] == 8'b00001111 )              block (3)
      Z_REG = 1'b0;
   else
      Z_REG = 1'b1;                                   block (4)
   if( LO_B == 1'b1 && Z_REG == 1'b0 )
      X_REG = 1'b1;
   else                                               block (5)
      X_REG = 1'b0;
   R_REG = TMP_STORAGE[30:0];
end
```

This description mixes several 'if' constructs and the following is a set of "minor" disadvantages:

– relationships between the signals are not clear

– debugging efficiency is extremely low

– it is very easy to make a mistake

And the following is a possible and very important mistake:

– unnecessary priority circuit can be generated. The reason is:

  – 'always' block contains description of the sequential process. Signal assigned at block (2) and block (3) are used in the conditional expression if the 'if' statement at the block (4) => blocks (2) and (3) have to be executed before block (4). It is evidently, that execution order is unimportant for (2) and (3). But a priority circuit may be generated to maintain such parallel operations described within a sequential block.

Therefore, descriptions using multiple 'if' or 'case' statements must not be used in order to avoid the problems described above. It is much better to write the same code using separate blocks:

```
assign LO_B = TMP_STORAGE[31];

assign R_REG = TMP_STORAGE[30:0];
                                                      block (2)
always @( SEL or X_IN or Z_IN ) begin
   if( SEL )                                          block (5)
      TMP_STORAGE = X_IN & Z_IN;
   else
      TMP_STORAGE = X_IN | Z_IN;                      block (1)
end

always @( TMP_STORAGE ) begin                         block (3)
   if( TMP_STORAGE[7:0] == 8'b00001111 )
```

```
         Z_REG = 1'b0;
      else
         Z_REG = 1'b1;
   end

   always @( LO_B or Z_REG )
      if( LO_B == 1'b1 && Z_REG == 1'b0 )
         X_REG = 1'b1;
      else
         X_REG = 1'b0;
   end
```

block (4)

	LEVEL	RULE
CHECKER BEHAVIOR	Checker restricts usage of more than one 'if'/'case' statement in the single 'always' block: – following descriptions violate the rule: – two or more 'if' statements in the same scope – two or more 'case' statements in the same scope – 'if' and 'case' in the same scope – the following descriptions do not violate the rule: – one 'if' inside an 'if'/'case' branch – one 'case' inside an 'if'/'case' branch Note: in case of violation first 'detail' is displayed without the word "same" (1, 3), whereas all following – with word "same" (2, 4)	

EXAMPLE-1: [1] 'always' block contains multiple (3) if statements in the same scope => violation;

[2] 1st 'if' statement at global scope of the 'always' block contains embedded 'if' => no violation (it is single within scope of the 1st 'if')

Do not describe more than one 'if' statement or 'case' statement in a single 'alw ays' block

'if' statement in the 'alw ays' block

```
always @( SEL or X_IN or Z_IN) begin
    if( SEL )
         TMP_STORAGE = X_IN & Z_IN;
         if( X_IN == 1'b1 )
             INTERNAL = 1'b1;
    else begin
         TMP_STORAGE = X_IN | Z_IN;
         INTERNAL = 1'b1;
    end
    LO_B = TMP_STORAGE[31] ^ INTERNAL;
    if( TMP_STORAGE[7:0] == 8'b00001111 )
         Z_REG = 1'b0;
    else
         Z_REG = 1'b1;
    if( LO_B == 1'b1 && Z_REG == 1'b0 )
         X_REG = 1'b1;
    else
         X_REG = 1'b0;
    R_REG = TMP_STORAGE[30:0];
end
```

'if' statement in the same 'alw ays' block

'if' statement in the same 'alw ays' block

EXAMPLE-2: [1] 'always' block contains both 'if' statement and 'case' statement in the same scope => violation;

[2] 1st 'case' branch contains embedded 'if' => no violation (it is single within scope of the 1st 'case' branch)

Do not describe more than one 'if' statement or 'case' statement in a single 'alw ays' block

'case' statement in the 'alw ays' block

```
always @( SEL or X_IN or Z_IN) begin
    case( SEL ) begin
         1'b1: begin
             TMP_STORAGE = X_IN & Z_IN;
             if( X_IN == 1'b1 )
                  INTERNAL = 1'b1;
             else
```

```
                INTERNAL = 1'b0;
            end
            1'b0: TMP_STORAGE = X_IN | Z_IN;
        endcase

        if( TMP_STORAGE[7:0] == 8'b00001111 )
            Z_REG = 1'b0;
        else
            Z_REG = 1'b1;
end
```

'if' statement in the same 'alw ays' block

STARC_VLOG 2.6.2.2

RULE NAME		**Signals assigned in always construct should not be described on the sensitivity list in the same always construct**
MESSAGE		**Sensitivity list of the 'always' block contains signals which were assigned in the same 'always' block.**
	DETAIL	*Sensitivity list signal "{SignalName}" is assigned.*
PROBLEM DESCRIPTION		If signal is assigned in always block and included in sensitivity list it may mean that the always construct is executed repeatedly and limitless. Such description is hazardous and should be avoided.
	LEVEL	RULE
CHECKER BEHAVIOR		Checker verifies signals assigned in the synthesized always block or mapped to an output/inout of task, called from this always block: – if any of signals is defined in sensitivity list of current always block => violation

EXAMPLE-1: [1] sensitivity list signal is assigned in the always block => violation

```
always @( in1, in2, in3, tmp ) begin

     out1 = in1 & in2 & in3;
     tmp  = in1 ^ in2 ^ in3;

end
```

Sensitivity list of the 'alw ays' block contains signals w hich w ere assigned in the same 'alw ays' block.

Sensitivity list signal "tmp" is assigned.

EXAMPLE-2: [1] signal bit is included in sensitivity list another bit of the same signal is assigned in the always block => no violation

```
always @( in1, in2, tmp[1] ) begin

        tmp[7] =  in1 & in2 ^ tmp[1];

end
```

2.7 if statements

2.7.1 if statements create prioritized circuits

STARC_VLOG 2.7.1.3

RULE NAME	**if statement in combinational circuit ends with else (not with else if)**	
MESSAGE	**When describing combinational circuits using 'always' construct, 'else' item should be used at the end of the 'if' statement to avoid generation of erroneous latches.**	
PROBLEM DESCRIPTION	When combinational circuit is described with an 'always' construct, it is recommended to use 'else' item at end of 'if statement. It helps to avoid generation of latches.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans 'always' statements that describe combinational logic: – each 'if' statement should contain 'else' item at the end Note: for this case, combinational 'always' statements are such 'always' statements that described without edges in the sensitivity list	

EXAMPLE-1: [1] 'always' statement contains 'if' statement without 'else' item (latch is inferred) => violation

```
always @( DATA or G or DATA ) begin
    if( G )
        Q = DATA;
end
```

When describing combinational circuits using 'alw ays' construct, 'else' item should be used at the end of the 'if' statement to avoid generation of erroneous latches.

EXAMPLE-2: [1] combinational 'always' contains 'if'-'else if' statement => violation; [2] 'else if' branch contains embedded 'if' statement without an 'else' item => violation

When describing combinational circuits using 'alw ays' construct, 'else' item should be used at the end of the 'if' statement to avoid generation of erroneous latches.

```
always @( SEL1 or SEL2 or EN or DATA ) begin
    if( SEL1 )
        Q = DATA[0];
    else if( SEL2 )
        if( EN )
            Q = DATA[1];
end
```

When describing combinational circuits using 'alw ays' construct, 'else' item should be used at the end of the 'if' statement to avoid generation of erroneous latches.

2.7.2 Reduce conditional expressions of *if statements* with the same contents

STARC_VLOG 2.7.2.1

RULE NAME	**Reduce conditional expressions of if statement with the same contents**	
MESSAGE	**Branches of the 'if' statement contain {NumberOfBranches} equivalent subexpressions: "{Subexpression}". Reduce the comparison logic to clearly designate comparison priorities.**	
	DETAIL	*Duplicated subexpression: "{Subexpression}"*
PROBLEM DESCRIPTION	Conditional expressions with the same contents generate compare circuits for each conditional expression. Typically, they are optimized during the structuring process of logic synthesis, but size of initially synthesized circuits is quite big and performance drops. Conditional expressions with same contents should be avoided.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans expressions in the conditional branches of 'if' statements: – if similar parts of expressions present => violation Note: as long as possible chains are detected.	

EXAMPLE-1: [1] two branches contain duplicated subexpressions => violation;
[2] order of subexpression items is different.

Branches of the 'if' statement contain 2 equivalent subexpressions: "C==1'b0&&A==1'b0". Reduce the comparison logic to clearly designate comparison priorities.

```
always @( A, B, C ) begin
     if( A == 1'b0 && B ==1'b1 && C == 1'b0 )
          Y = 3'b101;
     else if( C == 1'b0 && A == 1'b0 && B == 1'b0 )
          Y = 3'b110;
     else
          Y = 3'b100;
end
```

Duplicated subexpression: "C==1'b0&&A==1'b0"

EXAMPLE-2: [1] two branches contain duplicated subexpressions => violation;
[2] branches of embedded 'if' contain subexpressions that duplicate subexpressions of the upper-level 'if' => violation (recursive search of duplicated items).

Branches of the 'if' statement contain 3 equivalent subexpressions: "A==10". Reduce the comparison logic to clearly designate comparison priorities.

```
always @( A, B ) begin
     if( (A == 10) && (B != 7)  )
          if( B != 7 )
               Y = A;
          else if( A == 10 )
               Y = B;
     else if( (A < 7) || (A == 10) )
          Y = A ^ B;
end
```

Duplicated subexpression: "A==10"

Duplicated subexpression: "A==10"

Branches of the 'if' statement contain 2 equivalent subexpressions: "B!=7". Reduce the comparison logic to clearly designate comparison priorities.

Duplicated subexpression: "B!=7"

STARC_VLOG 2.7.2.2

RULE NAME	**Avoid describing conditions that will not be executed**	
MESSAGE-1	**'{StatementName}' statement contains branches that will not be executed. Avoid describing such conditions.**	
	DETAIL	*Branch will not be executed.*
PROBLEM DESCRIPTION	If you include lines that will never be executed, you might make mistakes in coding other lines. Condition expressions should be coded carefully and describing conditions that will not be executed should be avoided.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker analyzes 'if', 'case', 'casez' and 'casex' statements: – if there are redundant branches or branches that will never be executed => violation Note-1: if there are nested statements they are checked regarding that conditions of external statement are executable. Note-2: statement is not checked in case if it is nested in the branch that will never be executed. Note-3: {StatementName} may be either “if” or “case”.	

EXAMPLE-1: [1] if statement contains branch that will never be executed => violation.

```
if ( ctrl )

    q = data1;

else if ( ~ctrl )

    q = ~data1;

    else
        q = data2;
```

'if' statement contains branches that w ill not be executed. Avoid describing such conditions.

Branch w ill not be executed.

EXAMPLE-2: [1] first case item contains logical OR of two constants;

[2] second item is the came constant as the result of previous item so this branch will never be executed => violation.

```
reg [1:0] sel;
reg [1:0] data;

always @( sel, data )
    case ( in )
        2'b01 || 2'b10    : q <= data[0];  //result of operation is 'b1
        2'b01             : q <= data[1];  //item is not selected when sel = 2'b01
        default           : q <= 2'bxx;
    endcase
```

'case' statement contains branches that w ill not be executed. Avoid describing such conditions.

Branch w ill not be executed.

STARC_VLOG 2.7.2.3

RULE NAME	**Avoid null condition expressions**	
MESSAGE	**'{StatementName}' statement contains condition expression(s) with null statements. Missing statements in any condition expression of 'if' or 'case' statement may cause mistakes in coding other condition expressions.**	
	DETAIL	*Statement for condition expression is missing.*
PROBLEM DESCRIPTION	You can use an 'always' block to code combinational logic, but if the 'if' or 'case' statement has any null condition expression (a branch without any statement), a latch is most likely to be generated. Thus, you should describe only the necessary condition expressions and avoid null expressions. Even for an 'always' block that causes flip-flops to be inferred, if there is any null condition expression, you might make mistakes in coding other condition expressions. Condition expressions should be coded carefully.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker verifies condition branches of 'if', 'case', 'casex', 'casez' statements: – if there is no statement in this branch => violation.	

EXAMPLE-1: [1] there is no statement in 'else if' branch => violation

```
if ( ctrl1 )
     q = d1;
else if (ctrl2)
     begin
     end
else
     q = d2;
```

'if' statement contains condition expression(s) w ith null statements. Missing statements in any condition expression of 'if' or 'case' statement may cause mistakes in coding other condition expressions.

Statement for condition expression is missing.

EXAMPLE-2: [1] there is no statement in one of the 'case' branches => violation

```
case ( sel )
     2'b00 : q = d1;
     2'b01 : q = d2;
     2'b10 : ;
     2'b11 : q = d3;
     default : q = 1'bx;
```

'case' statement contains condition expression(s) w ith null statements. Missing statements in any condition expression of 'if' or 'case' statement may cause mistakes in coding other condition expressions.

2.7.3 Decrease the number of *if statement* nests

STARC_VLOG 2.7.3.1

RULE NAME	**The number of nests for if-if and else if is best at five or less. The number of nests for if-if and else if should be 10 at most**	
MESSAGE-1	**'if' operator is nested {NestedCount} times. It is recommended to use nesting {RECOMMENDED_NESTING_LEVEL} or less time(s) to avoid generation of complicated prioritized logic and decrease overall circuit size. When nesting cannot be avoided, it should be limited to {MAX_NESTING_LEVEL} time(s) at most.**	
MESSAGE-2	**'if'-'else if'-... conditions chain contains {NestedCount} branches. It is recommended to use {RECOMMENDED_NESTING_LEVEL} or fewer branch(es) to improve readability of the description and decrease possibility of nesting mistakes. When nesting cannot be avoided, it should be limited to {MAX_NESTING_LEVEL} time(s) at most.**	
PROBLEM DESCRIPTION	Within a conditional expression of an if statement, a combinational circuit is predictable. As the number of nests in an if statement increases, prioritized logic is added to this circuit automatically by logic synthesis. Therefore, when an if statement is too deep, the generated prioritized logic becomes complicated and overall circuit size increases significantly. As practice shows, nesting should be limited to seven levels. When deep nesting cannot be avoided number of levels should be 15 at most.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans 'if' statements recursively from the top level to bottom and calculate the number of: – cascaded 'if'-'if' nests (at the same level chain with most depth is chosen) – if calculated number for current 'if' statement is greater than RECOMMENDED_NESTING_LEVEL => violation (message-1) (lower level "if-if" statements are not scanned more) – parallel 'else'-'if' nests (from single level only) – if calculated number for current 'if' statement is greater than RECOMMENDED_NESTING_LEVEL => violation (message-2) Note: values of parameters RECOMMENDED_NESTING_LEVEL and MAX_NESTING_LEVEL are defined in configuration file (default values are 7 and 15 respectively)	

EXAMPLE-1: [1] number of nested 'if' statements is greater then RECOMMENDED_NESTING_LEVEL (value of the parameter is set to 2 to simplify the example) => violation (message-1);

[2] there are two chains of nested statements the longest one is chosen;

```
if( ... )
    begin
        if( ... ) begin   // 2nd nested
            if( ... )     // 3rd nested
        end
        if( ... ) begin   // 2nd nested
            if( ... )     // 3rd nested
                if( ... ) // 4th nested
        end
    end
```

'if' operator is nested 4 times. It is recommended to use nesting 2 or less time(s) to avoid generation of complicated prioritized logic and decrease overall circuit size. When nesting cannot be avoided, it should be limited to 15 time(s) at most.

EXAMPLE-2: [1] number of nested 'else'-'if' statements is greater then RECOMMENDED_NESTING_LEVEL=2 => violation (message-2);

[1] number of nested 'if' statements in one of 'else'-'if' branches is greater then RECOMMENDED_NESTING_LEVEL=2 => violation (message-1);

'if'-'else if'-... conditions chain contains 3 branches. It is recommended to use 2 or fewer branch(es) to improve readability of the description and decrease possibility of nesting mistakes. When nesting cannot be avoided, it should be limited to 15 time(s) at most.

```
if( ... )       // 1st "else-if" nested
else if( ... ) // 2nd "else-if" nested, 1st "if" nested
    if( ... )          // 2nd "if" nested
        if( ... )      // 3rd "if" nested
            if( ... ) // 4th "if" nested
else if( ... ) // 3rd "else-if" nested
```

'if' operator is nested 4 times. It is recommended to use nesting 2 or less time(s) to avoid generation of complicated prioritized logic and decrease overall circuit size. When nesting cannot be avoided, it should be limited to 15 time(s) at most.

STARC_VLOG 2.7.3.4

<table>
<tr><td>RULE NAME</td><td colspan="2"><b>Unify if statements which can be merged.</b></td></tr>
<tr><td rowspan="2">MESSAGE</td><td colspan="2"><b>This scope contains {IfStmtCount} nested 'if-if' statement(s) that could be merged into one plain 'if-else if' statement. Merging is recommended because it improves the performance of generated circuit (it depends on number of 'if' statement nests – the fewer the better).</b></td></tr>
<tr><td>DETAIL</td><td><i>This 'if' statement should be merged with the upper level branch.</i></td></tr>
<tr><td rowspan="2">PROBLEM DESCRIPTION</td><td colspan="2">It is difficult for 'if' statements with deep nesting to precisely comprehend associations with 'else' items. Generally speaking, indents are used to clarify associations with 'if' statements and 'else' items when coding. However, the number of indents increases with 'if' statements that have deep nesting. To improve description try to merge 'if' statements in the way shown below when it is possible. In this case the number of nests for the 'if' statement becomes less, which brings an improvement in comprehension and readability.</td></tr>
<tr><td>LEVEL</td><td>RECOMMENDATION 3</td></tr>
<tr><td>CHECKER BEHAVIOR</td><td colspan="2">

Checker scans 'if' statements recursively from the bottom level to top and verifies 'if' branches which contain only 'if' statement(s) (no other statements):

– if nested 'if' statement(s) exists:

  – NM_NESTED (number of conditional branches inferred by currently scanned nested 'if' statement + 1) is calculated:

```
// NM_NESTED = 1 (current branch) + 2 (1st level nested)
if( A ) begin // currently scanned branch (+ 1 conditional branch)
    ...
    if( B ) // 1st level nested (+ 1 conditional branch)
        if( X ) // 2nd level nested (is not considered)
            ...
        else if( Y )
            ...
    else if( C ) // 1st level nested (+ 1 conditional branch)
        ...
else if( D )
...
end
```

  – current branch of 'if' statement is virtually transformed to equivalent 'if-else if' construction (whole 'if' of upper level will be also transformed) and NM_TRANSFORMED (number of conditional branches inferred by virtually transformed branch) is calculated

  – if (NM_TRANSFORMED <= NM_NESTED) => violation

  consider an example:

```
// 'if' with nested statements - NM_NESTED = 4
//main message
if( A ) begin
    if( B ) begin     // detail-message #1
        statements1;
    end
    else if( C ) begin
        statements2;
    end
    if( D )           // detail-message #2
        statements3;
end
```

</td></tr>
</table>

RULE NAME	Unify if statements which can be merged.

```
// transformed statement - NM_TRANSFORMED = 4
if( A && B && D ) begin
    statements1;
    statements3;
end
else if( A && B && !D ) begin
    statements1;
end
else if( A && !B && C && D ) begin
    statements2;
    statements3;
end
else if( A && !B && C && !D ) begin
    statements2;
end
```

EXAMPLE-1: [1] 'if' branch which contains only 'if' statement is detected;
[2] NM_NESTED = 3; NM_TRANSFORMED = 2 => violation.

```
if ( a )
    if ( b )

        res1 <= c | d;

    else if ( c && !a) //branch will never be executed, and after merge it is case for 2.7.2.2

        res1 <= c & d;
```

This scope contains 1 nested 'if-if' statement(s) that could be merged into one plain 'if-else if' statement. Merging is recommended because it improves the performance of generated circuit (it depends on number of 'if' statement nests – the few er the better).

This 'if' statement should be merged w ith the upper level branch.

EXAMPLE-2: [1] 'if' branch statement which contains only 'if' statement is detected;
[2] but assignment is also detected within current branch => no violation.

```
if ( a )  begin
    if ( c )
        res1 <= a | b;
    res1 <= a & b;
end
```

2.7.4 Always surround multiple statements using block statements (begin-end) (Verilog only)

STARC_VLOG 2.7.4.3

RULE NAME	**Do not use fork-join in RTL descriptions (Verilog only)**	
MESSAGE-1	**'always' statement contains {ForkJoinBlocksCount} 'fork-join' block(s). 'fork-join' blocks cannot be used in RTL descriptions.**	
	DETAIL	*'fork-join' block detected.*
MESSAGE-2	**'task' statement contains {ForkJoinBlocksCount} 'fork-join' block(s). 'fork-join' blocks cannot be used in RTL descriptions.**	
	DETAIL	*'fork-join' block detected.*
PROBLEM DESCRIPTION	There are sequential (begin-end) and parallel (fork-join) blocks in Verilog-HDL. But logic synthesize tools support only sequential blocks, so do not use fork-join statement.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies always blocks and task statements: – if there is fork-join statement => violation	

EXAMPLE-1: [1] always block contains embedded fork-join statements => violation (message-1)

```
always @( in1 or in2 )
    begin : seq_exec

        fork : sequential_fork
            #5 temp1 = 2;


            fork : to_invoke_next_sequential
                #10 temp1 = temp1 + 1;

            join


            begin : sequential_zone_invoke_from_parallel

                @temp1  #1 temp1 = temp1 + 1;
                #10 temp1 = temp1 - 1;

            end

        join

    end
```

'alw ays' statement contains 2 'fork-join' block(s). 'fork-join' blocks cannot be used in RTL descriptions.

'fork-join' block detected.

'fork-join' block detected.

EXAMPLE-2: [1] task contain fork-join statement => violation (message-2)

```
task fork_task;

    ...

   fork
   ...
   join

endtask
```

'task' statement contains 1 'fork-join' block(s). 'fork-join' blocks cannot be used in RTL descriptions.

'fork-join' block detected.

2.8 *case statements*

2.8.1 *case statements* facilitate decoder/encoder description

STARC_VLOG 2.8.1.3

RULE NAME	**Avoid the overlapping of case items**	
MESSAGE	**Case statement contains {TotalOverlapCount} overlapped case clause(s). Avoid the overlapping of case items.**	
	DETAIL	*'{CaseItem}' is overlapped.*
PROBLEM DESCRIPTION	Since the case statement compares from the top, overlapping values are permitted. When there are overlapping values of clauses, the first selection expression is executed first. If there are duplications in the case statements, the logic synthesis tool synthesizes priority circuits with a similar priority as the if statement, but if there are no overlaps in the branch conditions, then a circuit that compares values in parallel is synthesized. Logic synthesis tools automatically judge branch overlaps, but it is also possible to clearly define that there are no overlaps of directives as the comment "//synopsys parallel_case". If coding is limited so that values do not overlap in case statements, "//synopsys parallel_case" can be used to gain advantages in area and speed. However, if using parallel_case when values overlap, the RTL simulation result and the simulation result of logic gates generated by logic synthesis differ. Therefore, it should never be done.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	1) Checker verifies case statements: – if there is a duplication of case items: – if the length of selection expression is 'n' and some of the case items have length 'm': – 'm' < 'n' – if the lowest 'm' bits of case items with length 'n' are duplicated by case items with length 'm' and all the rest bits are '0' => violation – 'm' > 'n' – if the the lowest 'n' bits of case items with length 'm' are duplicated by case items with length 'n' => violation 2) Checker verifies casex statements: – if there is duplication of case items that do not contain don't care conditions ('x' or '?'): – if the length of selection expression is 'n' and some of the case items have length 'm': – 'm' < 'n' – if the lowest 'm' bits of case items with length 'n' are duplicated by case items with length 'm' and all the rest bits are '0' => violation – 'm' > 'n' – if the the lowest 'n' bits of case items with length 'm' are duplicated by case items with length 'n' => violation 3) Checker verifies casez statements: – if there is a duplication or overlapping of case items: – if the length of selection expression is 'n' and some of the case items have length 'm': – 'm' < 'n' – if the lowest 'm' bits of case items with length 'n' are duplicated or	

RULE NAME	Avoid the overlapping of case items
	overlapped by case items with length 'm' and all the rest bits are '0' or 'z' statement) => violation – 'm' > 'n' – if the the lowest 'n' bits of case items with length 'm' are duplicated or overlapped by case items with length 'n' => violation Note: If there are parameters (in case items) then all warnings are shown during the elaboration stage

EXAMPLE-1: [1] case statement contains duplicated items => violation

```
case ( sel )
    4'b0001 : out1 = 2'b00;
    4'b0010 : out1 = 2'b01;
    4'b0010 : out1 = 2'b10;
    4'b1000 : out1 = 2'b11;
    default : out1 = 2'bxx;
endcase
```

Case statement contains 2 overlapped case clause(s). Avoid the overlapping of case items.

'4'b0010' is overlapped.

'4'b0010' is overlapped.

EXAMPLE-2: [1] casez statement contains overlapped items => violation

```
casez ( sel )
    4'b0001 : out1 = 2'b00;
    4'b0010 : out1 = 2'b01;
    4'b0100 : out1 = 2'b10;
    4'b000z : out1 = 2'b11;
    default : out1 = 2'bxx;
endcase
```

Case statement contains 2 overlapped case clause(s). Avoid the overlapping of case items.

'4'b0001' is overlapped.

'4'b000z' is overlapped.

EXAMPLE-3: [1] casez statement contains 'x' in one of the items, but no overlapping in such situation => no violation

```
casez ( sel )
    4'b0001 : out1 = 2'b00;
    4'b0010 : out1 = 2'b01;
    4'b0100 : out1 = 2'b10;
    4'bxxxx : out1 = 2'b11;
    default : out1 = 2'bxx;
endcase
```

EXAMPLE-4: [1] casex statement has overlapped items, that contains don't care value ('?') => no violation

```
casex ( sel )
    4'b0001 : out1 = 2'b00;
    4'b0010 : out1 = 2'b01;
    4'b0100 : out1 = 2'b10;
    4'b???? : out1 = 2'b11;
    default : out1 = 2'bxx;
endcase
```

STARC_VLOG 2.8.1.4

RULE NAME	**Always add default clauses**	
MESSAGE	**Always add 'default' clauses.**	
PROBLEM DESCRIPTION	The default clause of case statement is executed if all comparisons of selection expression and case items fail, so always add default clause to avoid situation when nothing executes.	
	LEVEL	RECOMMEND 1
CHECKER BEHAVIOR	Checker verifies case, casex, casez statements – if there is no default clause => violation.	

EXAMPLE-1: [1] case statement do not contain default clause => violation.

```
case ( sel )  ◄- - - - - - - - - - - -  Alw ays add 'default' clauses.
    4'b0001 : out1 = 2'b00;
    4'b0010 : out1 = 2'b01;
    4'b0100 : out1 = 2'b10;
    4'b1000 : out1 = 2'b11;
endcase
```

EXAMPLE-2: [1] case statement contain default clause => no violation.
Note: such description do not violate current rule but may cause other problems (see 2.8.3.5)

```
case ( sel_expr )
     2'b10 : out1 <= 1'b1;
     default : out1 <= 1'bx;
     2'b01 : out1 <= 1'b0;
endcase
```

STARC_VLOG 2.8.1.5

RULE NAME	**Do not force full_case for case statement directives that depend on a particular logic synthesis tool (Verilog only)**	
MESSAGE	**Do not force 'full_case' directives that depend on particular logic synthesis tool. RTL and post-synthesis simulation results will differ.**	
PROBLEM DESCRIPTION	Design Compiler has a directive called `//synopsys full_case`. If the directive is specified in a case statement, in which not all the case clauses are described, and the default clause is absent – it is assumed that all the case clauses are described. However, when this directive is used, the simulation results for missing cases will differ in the RTL and gate-level models. Therefore, “synopsys full_case” should never be used.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies case, casex, casez statements: – if directive `//synopsys full_case` is present => violation	

EXAMPLE-1: [1] directive `//synopsys full_case` is used => violation

```
case (data)                      //synopsys full_case
    8'b00000001  : code =0;
    8'b00000010  : code =1;
    8'b00000100  : code =2;
    8'b00001000  : code =3;
    8'b00010000  : code =4;
    8'b00100000  : code =5;
    8'b01000000  : code =6;
    8'b10000000  : code =7;
endcase
```

Do not force 'parallel_case' directives that depend on particular logic synthesis tool. RTL and post-synthesis simulation results may differ.

STARC_VLOG 2.8.1.6

RULE NAME	**Pay attention to the selective range of case statement and bit width of each item (Verilog only)**	
MESSAGE-1	**Bit width of the 'case' selection expression is not defined. To avoid simulation and synthesis difficulties, exactly match bit widths of the 'case' selection expression and the 'case' items.**	
MESSAGE-2	**Bit width "{SelExpWidth}" of the 'case' selection expression does not match the bit width "{ItemsBitWidth}" of all 'case' item(s). Match bit widths exactly to avoid simulation and synthesis difficulties.**	
MESSAGE-3	**Bit width "{SelExpWidth}" of the 'case' selection expression does not match the bit width of 'case' item(s). Match bit widths exactly to avoid simulation and synthesis difficulties.**	
	DETAIL-1	*Bit width of 'case' item is "{ItemBitWidth}".*
	DETAIL-2	*Bit width of 'case' item is undefined.*
	DETAIL-3	*Bit width of 'case' item will be defined at elaboration time.*
PROBLEM DESCRIPTION	It is allowed in the syntax of Verilog HDL that some items exceed the range of the signal specified in the selection expression of the case statement. However, this line is ignored by simulation and logic synthesis tools. When item bit width is less then bit width of selection expression, upper bits of items is filled with zeros and compared, but such situation can cause duplication of case items. To avoid difficulty pay attention to the selective range of case statement and bit width of each item.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects case, casex, casez statements and calculates bit width of selection expression: – if bit width of case selection expression cannot be defined (part-selection with variable index is used) => violation (message-1), note: all branches of this case statement are checked for presence of another case statements; Checker collects bit widths of case items: – if all items has equal bit width and it is different from bit width of the selection expression => violation (message-2) – if there are case item(s) with different bit widths => violation (message-3) – if there is at least one case item, bit width of which can be defined only at elaboration-time: – if bit widths of all 'case' items can be defined only at elaboration-time only elaboration-time message is issued – if bit widths of some 'case' items can be defined at compilation-time, whereas another items of this 'case' can be defined only at elaboration-time: – compilation-time violation for elaboration-time items (message-3 + detail-3) – compilation-time violation for compilation-time items (message-3 + detail-1) and elaboration-time check are scheduled for them Note-1: if the width can not be calculated (case item is a part-selection with variable index) detail-2 is used. Note-2: bit widths of decimal constants are defined by their values. Violation is displayed only if constant width is greater than required (narrower decimal constants are allowed).	

EXAMPLE-1: [1] bit width of case selection expression cannot be defined (ternary operator is used and arguments have different bit widths) => violation (message-1)

```
reg [8:0] a;
reg [16:0] b;

case( sel ? a : b )
    16'b01 : tmp = 0;
    8'b01  : tmp = 1;
endcase
```

Bit width of the 'case' selection expression is not defined. To avoid simulation and synthesis difficulties, exactly match bit widths of the 'case' selection expression and the 'case' items.

EXAMPLE-2: [1] bit width of case selection expression is different from bit widths of case items => violation (message-3)

[2] first case item bit width can be defined at the compilation stage => detail-1

[3] second case item bit width can not be defined at the compilation stage (parametrized) => detail-3

```
reg [1:0] sel_expr;
parameter cparam = 1;

case( sel_expr )
    4'b01  : tmp = 1;
    cparam : tmp = 0;
endcase
```

Bit width "2" of the 'case' selection expression does not match the bit width of 'case' item(s). Match bit widths exactly to avoid simulation and synthesis difficulties.

Bit width of 'case' item is "4".

Bit width of 'case' item will be defined at elaboration time.

2.8.2 Divide using if statement, etc. to avoid creating large tables

STARC_VLOG 2.8.2.1

RULE NAME

As a rule of thumb, divide large tables if they have more than 32 I/Os. (input: 20, output: 12)

MESSAGE

'case' statement defines a table with "{InputCount}" inputs and "{OutputCount}" outputs. It is recommended to limit the table size to {RECOMMENDED_INPUT_COUNT} inputs and {RECOMMENDED_OUTPUT_COUNT} outputs.

PROBLEM DESCRIPTION

A table is correspond to each 'case' statement. The size of a 'case' statement table is determined by the number of inputs. When the number of inputs grows, table size grows correspondingly. Large tables cause size increasing and speed decreasing of generated circuit.

However, output count still should be considered. If the number of outputs increases and every logic is not very similar, area may be increased. In the case of a small table, which has five or fewer clauses or four or fewer inputs, the output number is not necessary to consider, but for other cases attention should be paid to the number of outputs.

Therefore, specifying 'case' statements with a large bit width of inputs and outputs causes the synthesis run time to increase and may further worsen the synthesis results. It is recommended that the 'case' statement has up to 32 bits including input and output.

'case' statement with large number of inputs may be simplified by dividing the statement into smaller ones and using 'if' statement. Consider following description example (let's assumes that recommended number of inputs is equal to 4, to simplify the example):

```
// initial description
case ( data )
    8'b00000001  : code =0;
    8'b00000010  : code =1;
    8'b00000100  : code =2;
    8'b00001000  : code =3;
    8'b00010000  : code =4;
    8'b00100000  : code =5;
    8'b01000000  : code =6;
    8'b10000000  : code =7;
endcase
//simplified description
if ( data [7:4] == 4'b0000 )
    case ( data[3:0] )
        4'b0001  : code =0;
        4'b0010  : code =1;
        4'b0100  : code =2;
        4'b1000  : code =3;
    endcase;
else if ( data [3:0] == 4'b0000 )
    case ( data[7:4] )
        4'b0001  : code =4;
        4'b0010  : code =5;
        4'b0100  : code =6;
        4'b1000  : code =7;
    endcase;
```

LEVEL	RECOMMENDATION 3

CHECKER BEHAVIOR

Checker detects case, casex, casez statements:

- if number of case clauses is greater than SMALL_CASE_CLAUSE_COUNT => checker counts NI – number of inputs(*):
  - *(*) number of inputs is defined by number of different signals in case selection expression (each bit in multiple-bit signals is considered as separate input),*

    note: if 'case' items are variable NI is defined taking those signals into account;

RULE NAME	**As a rule of thumb, divide large tables if they have more than 32 I/Os. (input: 20, output: 12)**
	– if NI > SMALL_CASE_INPUT_COUNT => checker counts NO – number of outputs: – *(*) number of outputs defined by number of different signals that are assigned under control of all case clauses (each bit in multiple-bit signals is considered as separate output),* *note: outputs of nested 'case' constructions are also considered;* – if (NI > RECOMMENDED_INPUT_COUNT) or (NO > RECOMMENDED_OUTPUT_COUNT) => violation Note: parameters SMALL_CASE_CLAUSE_COUNT, SMALL_CASE_INPUT_COUNT, RECOMMENDED_INPUT_COUNT, RECOMMENDED_OUTPUT_COUNT values are defined in configuration file (default values are 5, 4, 20, 12 respectively).

EXAMPLE-1: [1] 'case' statement contains 6 clauses;
[2] selection expression bit width is 32 => violation

```
reg [31:0] sel;

    ...

    case ( sel )

        32'b00000001  : dout = din[0];
        32'b00000010  : dout = din[1];
        32'b00000100  : dout = din[2];
        32'b00001000  : dout = din[3];
        32'b00010000  : dout = din[4];
        32'b00100000  : dout = din[5];
        default       : dout = 1b'x;

    endcase
```

Detected 'case' statement defines table w ith “32” inputs and “1” outputs. Recommended is limit table size to 20 inputs and 12 outputs.

EXAMPLE-2: [1] 'case' statement contains 2 inputs and 40 outputs;
[2] there only 4 'case' clauses => no violation.

```
reg [1:0] sel;
reg [31:0] din;
reg [7:0] dout1;
reg [15:0] dout2;

    ...

    case ( sel )

        2'b00   : dout1 = din[7:0];
        2'b01   : dout2[3:0] = din[11:8];
        2'b10   : dout2[3:0] = din[15:12];
        2'b11   : dout2  = din;

    endcase
```

STARC_VLOG 2.8.2.2

RULE NAME	**The number of case items should be up to 200**	
MESSAGE	**Case statement contains {CaseItemsCount} case item expressions. Avoid using case statements with more than {MAX_CASE_ITEM_EXPRESSIONS} case item expressions.**	
PROBLEM DESCRIPTION	With the current logic synthesis tool capacity, area and speed increase at exponential rates when exceeding approximately 200 clauses. Therefore, number of clauses of a case statement should not exceed 200. However, in case of a table where randomness is high such as with a SIN function, dividing a table, (for example with 8 bits of both input and output and 256 clauses by if statement) will result in the decrease of description readability only, so there is no advantage in circuit generation. In this particular situation, a case statement with 256 clauses has to be created, but the number of clauses should not exceed 300.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker detects case, casex, casez statements: – if number of case items expressions is greater than MAX_CASE_ITEM_EXPRESSIONS => violation Note: parameter MAX_CASE_ITEM_EXPRESSIONS value is defined in configuration file	

EXAMPLE-1: [1] number of case items is greater than value specified with MAX_CASE_ITEM_EXPRESSIONS (the value is set to 4 to simplify an example) => violation (message)

```
case (data)
    8'b00000001  : code =0;
    8'b00000010  : code =1;
    8'b00000100  : code =2;
    8'b00001000  : code =3;
    8'b00010000  : code =4;
    8'b00100000  : code =5;
    8'b01000000  : code =6;
    8'b10000000  : code =7;
endcase
```

Case statement contains 8 case item expressions. Avoid using case statements w ith more than 4 case item expressions.

2.8.3 Use *default clauses*

STARC_VLOG 2.8.3.1

RULE NAME	**The don't-care condition is defined by using 'x' as the default clause (only for default clauses, the extensive use of don't-care is recommended)**	
MESSAGE	**It is recommended to specify 'x' to the output of the 'default' clause. "Don't-care" condition allows optimization and size of the circuit will be decreased more than setting a determined value.**	
	DETAIL	*Signal "{SignalName}" is assigned with determined value.*
PROBLEM DESCRIPTION	The 'default' clause is executed if the selection expression specified by the case statement does not match to any item. Constant value may be specified as the default assignment. But if the unknown value 'x' is assigned to the output in the 'default' clause, it is considered as "don't-care" and a synthesis tool selects the best assignment (either '0' or '1') to the output during the optimization. Therefore, setting a non-determined value can usually decrease the size of the generated circuit.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans 'case', 'casex', 'casez' statements: – check signals being assigned in the 'default' clause: – if any signal is assigned with non-'x' value in the 'default' clause => violation – *note:* being assigned with 'x' value means following for this rule: right-hand side of assignment is a constant/parameter containing 'x'-es – *note:* 'default' clause may contain embedded constructions: – embedded 'case': new check is started – other constructions: assignments are considered as being performed in the 'default' clause – *exception:* 'default' clauses that are redundant are skipped (for example, when 'case' is full) – context: synthesizable 'always' constructs Note: elaboration-time checks are performed when right-hand side of assignment cannot be evaluated at compile-time.	

EXAMPLE-1: [1] 'case' statement has signal assignment in the default clause;
[2] signal is assigned with non-'x' value => violation

```
always @(...)
    case (sel)
        4'b0001: res = data1;
        4'b0010: res = data2;
        4'b0100: res = data3;
        4'b1000: res = data4;
        default: res = 1'b0;
    endcase
```

It is recommended to specify 'x' to the output of the 'default' clause. "Don't-care" condition allow s optimization and size of the circuit w ill be decreased more than setting a determined value.

Signal "res" is assigned w ith determined value.

EXAMPLE-2: [1] 'case' statement has signal assignment in the default clause;
[2] signal is assigned with non-'x' value;
[3] 'case' is full and optimization is performed => no violation

```
always @(...)
    case (sel)
        2'b00: res = data1;
        2'b01: res = data2;
        2'b10: res = data3;
        2'b11: res = data4;
        default: res = 1'b0;
    endcase
```

STARC_VLOG 2.8.3.4

RULE NAME	**Do not use the signal to which a don't care condition is assigned for a conditional expression of an 'if' statement**	
MESSAGE-	**Conditional expression in the 'if' statement uses signal “{SignalName}” that is assigned with don't care 'x' condition. It is recommended to avoid such descriptions because after the logic synthesis is completed, it is unknown whether it becomes '0' or '1'.**	
	DETAIL-1	*Signal “{DrivenSignalName}” is driven by another signal “{DriverSignalName}” that is assigned with don't care 'x' condition.*
	DETAIL-2	*Don't care 'x' condition is assigned to signal “{SignalName}”.*
PROBLEM DESCRIPTION	Situation when signal to which a don't care condition is assigned is used in conditional expression of an 'if' statement may lead to the differences in results between simulation of behavioral and RTL models: during simulation of behavioral model 'else' branch is executed or none of the branches are executed (if there is no 'else' branch) and after the logic synthesis is completed, it is undefined whether it becomes '1' or '0' => different conditions may be executed. Consecutively, signals to which 'x' is assigned in 'default' clause of a 'case' statement should not be used in a conditional expression of an 'if' statement.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker detects drivers for signals used in 'if' conditional expression: – if some signal is driven by don't care value => violation: – if driver is explicit => detail-2; – if driver is implicit => detail-1. *Note: signal is driven by don't care value in case of:* – *explicit assignment: 'x'-value is assigned to the signal in the 'default' clause of 'case' / 'casex' /'casez' statement;* – *implicit assignment: the signal is assigned with other signal(s) driven by don't care; following statements are considered as possible don't care drivers:* – *assign, '=', '<=' (only direct assignments propagate don't care value);* – *function call (with only direct assignments within don't care value propagation path).*	

EXAMPLE-1: [1] signal "tmp" is used in the 'if' conditional expression;
[2] the signal is driven by don't care value ('x'-value is assigned to the signal in the 'default' clause of 'case' statement) => violation (detail-2).

```
always @( ... )

    case( sel )
        ...
        default : tmp = 1'bx;
    endcase

always @( tmp )

    if( tmp == 1'b1 )
        ...
    else
        ...
```

Don't care 'x' condition is assigned to signal “tmp”.

Conditional expression in the 'if' uses signal “tmp” that is assigned w ith don't care 'x' condition. It is recommended to avoid such descriptions because after the logic synthesis is completed, it is unknow n w hether it becomes '0' or '1'.

EXAMPLE-2: [1] 'case' statement assigns 'x'-es in the 'default' clause to an output of the "task_case" task;
[2] task "task_case" returns the value to the variable "tmp" that is also an input of the "task_if" task;
[3] task "task_if" uses this input in conditional expression of the 'if' statement => violation (detail-2).

```
task task_case;
    ...
    output t_out1;

    begin
        case( t_sel )
            ...
            default : t_out1 = ( t_in1 == 2'b11 ) ? 1'b1 : 1'bx;
        endcase
    end
endtask
```

Don't care 'x' condition is assigned to signal “tmp”.

```
task task_if;
    input t_in1;
    ...

    begin
        if( t_in1 == 2'b00 )
            ...
    end
endtask

always @( ... )
    begin
        task_case ( ..., tmp );
        task_if   ( tmp, ... )
    end
```

Conditional expression in the 'if' uses signal “tmp” that is assigned w ith don't care 'x' condition. It is recommended to avoid such descriptions because after the logic synthesis is completed, it is unknow n w hether it becomes '0' or '1'.

EXAMPLE-3: [1] signal "tmp" is used in the 'if' conditional expression;

[2] the signal is assigned with another signal (“res”) driven by don't care value=>violation (detail1 + detail-2).

```
always @( sel )

    case( sel )
        ...
        default : res = 1'bx;
    endcase

always @( tmp )

    tmp = res;
    if( tmp == 1'b1 )
        ...
    else
        ...
```

Don't care 'x' condition is assigned to signal “res”.

Signal “tmp” is driven by another signal “res” that is assigned w ith don't care 'x' condition.

Conditional expression in the 'if' uses signal “tmp” that is assigned w ith don't care 'x' condition. It is recommended to avoid such descriptions because after the logic synthesis is completed, it is unknow n w hether it becomes '0' or '1'.

EXAMPLE-4: [1] signal "tmp_1" as used in the conditional expression;

[2] the signal is driven by don't care value ('x'-value is assigned to the signal in the 'default' clause of 'case' statement);

[2] signal "tmp_1" is used not as a simple signal it is an operand of an adding operation => no violation ('x'-es is not propagated).

```
always @( ... )

    case( sel )
        ...
        default : tmp_1 = 1'bx;
    endcase

assign tmp_2 = ~ in1;

always @( tmp )

    if( tmp_1 + tmp_2 == 1'b1 )
        ...
```

STARC_VLOG 2.8.3.5

RULE NAME	**Describe a default clause at the end of a case statement**	
MESSAGE	**Describe a 'default' clause at the end of a 'case' statement.**	
PROBLEM DESCRIPTION	Case items which are defined after default clause are not handled during simulation. That is why it is recommended to describe default clause at the end of a case statement to avoid unexpected simulation results.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies case, casex, casez statements which contain default clause – if default clause is not the last item => violation.	

EXAMPLE-1: [1] case statement contain default clause but it is not the last item => violation.

Describe a 'default' clause at the end of a 'case' statement.

```
case ( sel_expr )
     2'b10  : out1 <= 1'b1;
     default : out1 <= 1'bx;
     2'b01  : out1 <= 1'b0;
endcase
```

STARC_VLOG 2.8.3.6

RULE NAME		**Do not use the signal to which a don't care condition is assigned for selection expression of a case statement which does not assign 'x' in the default clause**
MESSAGE-1		**Selection expression of a 'case' statement uses signal "{SignalName}" that is assigned with don't care 'x' condition. After the logic synthesis is completed, it is unknown whether it becomes '0' or '1'. It is recommended to avoid using such signals in selection expressions of 'case' statements that do not assign 'x'-es in the 'default' clause.**
	DETAIL-1	*Signal "{DrivenSignalName}" is driven by another signal "{DriverSignalName}" that is assigned with don't care 'x' condition.*
	DETAIL-2	*Don't care 'x' condition is assigned to signal "{SignalName}".*
PROBLEM DESCRIPTION		When unknowns are generated because of don't care ('x') usage in RTL description simulation, it is treated as a bug in a design. If signal to which a don't care condition is assigned is used for a selection expression of a 'case' statement it is recommended to add default clause which assigns 'x' values. In such case 'x' value is propagated and it is easy to find bug. If the default clause either assign fixed value or does not exist then it is difficult to find bug. Signals to which 'x' is assigned in the 'default' clause of a 'case' statement should not be used in a selection expression of a 'case' statement if there is no don't care default clause.
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR		Checker detects drivers for signals used in 'case' selection expression: – if some signal is driven by don't care value: – if 'x'-es are not assigned to all signals in the 'default' clause => violation: – if driver is explicit => detail-2; – if driver is implicit => detail-1. Note: see 2.8.3.4 for more information about signals driven by don't care value.

EXAMPLE-1: [1] signal "sel_2" is used in the selection expression of "case_2";
[2] the signal is driven by don't care value from "case_2" => violation (detail-2).

```
always @( ... )

    case( sel_1 ) // case_1
        ...
        default : sel_2 = 1'bx;
    endcase

always @( tmp )

    case( sel_2 ) // case_2
        ...
    default: ... = 8'b10101010;
    ...
```

'x' is assigned to signal "sel_2"

Selection expression of a 'case' uses signal "sel_2" that is assigned with don't care 'x' condition. After the logic synthesis is completed, it is unknown whether it becomes '0' or '1'. It is recommended to avoid using such signals in selection expressions of 'case' statements that do not assign 'x'-es in the 'default' clause.

Note: if this 'default' assign 'x'-es => there will be no violation for 2.8.3.6

EXAMPLE-2: [1] signal "sel" is used in the selection expression of 'case' statement;
[2] the signal is driven by don't care value from function => violation (detail-1 + detail-2).

```
function  res;
    ...
    begin
        case( f_sel )
            ...
            default : res = 1'bx;
        endcase
    end
```

'x' is assigned to signal "res" which is a driver to the signal "tmp"

```
endfunction

always @( ... )
     begin
          sel = res( in1 );
          case( sel )
               ...
          endcase
     end
```

Signal “sel” is driven by another signal “res” that is assigned with don't care 'x' condition.

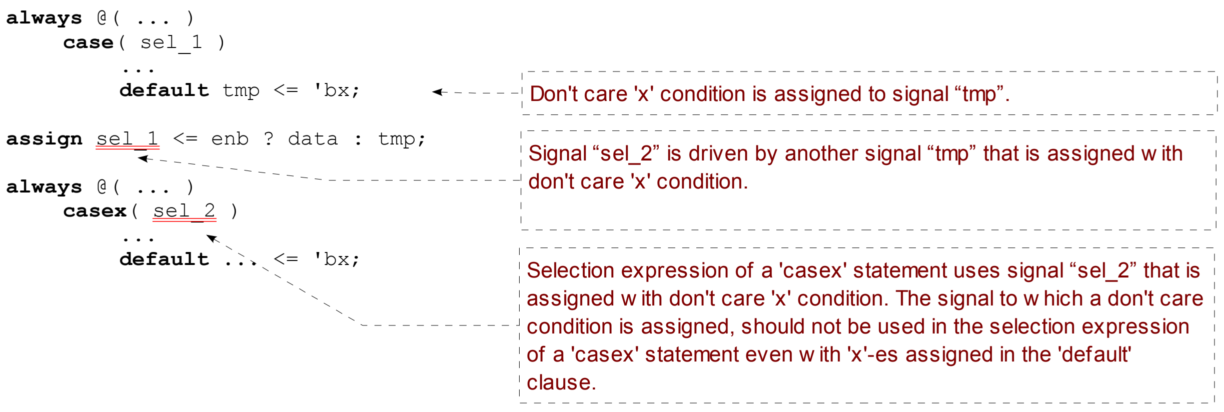
Selection expression of a 'case' uses signal “sel” that is assigned with don't care 'x' condition. After the logic synthesis is completed, it is unknown whether it becomes '0' or '1'. It is recommended to avoid using such signals in selection expressions of 'case' statements that do not assign 'x'-es in the 'default' clause.

STARC_VLOG 2.8.3.7

RULE NAME	**Do not use the signal to which a don't care condition is assigned for selection expression of a casex statement**	
MESSAGE-1	**Selection expression of a 'casex' statement uses signal "{SignalName}" that is assigned with don't care 'x' condition. The signal to which a don't care condition is assigned, should not be used in the selection expression of a 'casex' statement even with 'x'-es assigned in the 'default' clause.**	
	DETAIL-1	*Signal "{DrivenSignalName}" is driven by another signal "{DriverSignalName}" that is assigned with don't care 'x' condition.*
	DETAIL-2	*Don't care 'x' condition is assigned to signal "{SignalName}".*
PROBLEM DESCRIPTION	'x' values are regarded in 'casex' statement as don't care values (either '0' or '1'). Using the signal to which a don't care condition is assigned for selection expression of a 'casex' statement may lead to the differences between RTL and gate-level simulation. And it is also difficult to find bug in such situation even if 'casex' statement has don't care 'default' clause. Signals to which 'x' is assigned in the 'default' clause of a 'case' statement should not be used in a selection expression of a 'casex' statement.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker detects drivers for signals used in 'casex' selection expression: – if some signal is driven by don't care value => violation: – if driver is explicit => detail-2; – if driver is implicit => detail-1. Note: see 2.8.3.4 for more information about signals driven by don't care value.	

EXAMPLE-1: [1] signal "sel_2" is used in the selection expression of 'casex' statement;

[2] signal "sel_2" is driven by signal "tmp" witch is driven by don't care value within default clause of 'case' statement => violation (detail-1 + detail-2).



```
always @( ... )
     case( sel_1 )
          ...
          default tmp <= 'bx;

assign sel_1 <= enb ? data : tmp;

always @( ... )
     casex( sel_2 )
          ...
          default ... <= 'bx;
```

Don't care 'x' condition is assigned to signal "tmp".

Signal "sel_2" is driven by another signal "tmp" that is assigned w ith don't care 'x' condition.

Selection expression of a 'casex' statement uses signal "sel_2" that is assigned w ith don't care 'x' condition. The signal to w hich a don't care condition is assigned, should not be used in the selection expression of a 'casex' statement even w ith 'x'-es assigned in the 'default' clause.

2.8.4 Do not use complex *casex statements* (Verilog only)

STARC_VLOG 2.8.4.3

RULE NAME	**It is best to avoid using casex statements and casez statements (Verilog only)**	
MESSAGE	**Avoid using 'casex' and 'casez' statements.**	
PROBLEM DESCRIPTION	With a casex statement or a casez statement, the value indicated by ’x’ (by 'z' for casez) of the branch conditions is defined as don’t-care. Don’t-care indicates that it does not matter whether the value is ’1’ or ’0’. Using don't-care condition in case items should be performed with great attention because may easily lead to overlapping or duplication of items and as a result to worsening the circuit quality. So It is best to avoid using casex statements and casez statements.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans Verilog module: – if casex or casez statements are present => violation	

EXAMPLE-1: [1] casex statement is used => violation

```
casex (sel)
    4'bx0x0: out1 = a;
    4'b0010: out1 = b;
    4'b0100: out1 = c;
    4'b1000: out1 = d;
    default: out1 = 1'bx;
endcase
```

Avoid using 'casex' and 'casez' statements.

STARC_VLOG 2.8.4.4

RULE NAME	**Do not indiscriminately describe 'x' of casex statement for each bit**	
MESSAGE-1	**Casex statement contains {TotalOverlapCount} overlapped case clause(s). Use 'do not care' condition carefully to avoid the duplication of case items.**	
	DETAIL	*'{CaseItem}' is overlapped.*
MESSAGE-2	**Casex statement contains items that may cause the generation of additional logic that decreases circuit quality. Describe priority logic with triangular form of 'do not care' conditions in case items.**	

PROBLEM DESCRIPTION	With a 'casex' statement or a 'casez' statement, the value indicated by ’x’ of the branch conditions is defined as “don’t-care”. “Don’t-care” indicates that it does not matter whether the value is ’1’ or ’0’. Descriptions in which the “don’t-care” condition differs for each bit invite the duplication of clauses and risks worsening the circuit quality. When using a 'casex' statement, describe so the don’t-care ’x’ specification range has a distinct range. Prioritized logic is often described with the help of don't care value. Consider the example below. If the least significant bit is '1' the first branch will be executed, if the bit is '0' matching continues.
	`casex ( sel )`
	`    4'bxxx1: ...`
	`    4'bxx10: ...`
	`    4'bx100: ...`
	`    4'b1000: ...`
	`endcase`	

	LEVEL	RECOMMENDATION 1

CHECKER BEHAVIOR	Checker detects 'casex' statements:
	– if there are duplications or overlapping of case items that contain don't-care conditions ('x', 'z' or '?') => violation (message-1):
	    – If the length of a case item is not equal to the length of the selection expression – the case item is either truncated of filled with zeroes ('0') to the length of the selection expression. Thus, an unexpected duplication or overlapping may occur.
	    Example:
	`parameter MASK1 = 5'b100x1;`
	`parameter MASK2 = 3'd3;`
	`...`
	`casex ( A[3:0] )`
	`    MASK1   : ...`
	`    4'b0x11 : ...`
	`    MASK2   : ...`
	`    ...`
	`endcase`
	After parameter substitution, items will virtually look like this:
	`casex ( A[3:0] )`
	`    5'b100x1 : ... // will be truncated to 4'b00x1`
	`     4'b0x11 : ...`
	`      3'b011 : ... // will be extended to 4'b0011`
	`    ...`
	`endcase`
	After normalization to 4 bits, items will actually look like this (overlapping occurs):
	`casex ( A[3:0] )`
	`    4'b00x1 : ... // 5'b100x1 truncated to 4 bits`
	`    4'b0x11 : ...`
	`    4'b0011 : ... // 3'b011 extended to 4 bits`	

RULE NAME	Do not indiscriminately describe 'x' of casex statement for each bit

```
        ...
    endcase
```

– if there is no duplication or overlapping, case items are considered as matrix, and it is checked whether it could be transformed to the triangular form, i.e. it may be virtually presented (by replacing columns, if needed) as a structure like this:

```
casex ( A[7:0] )
    8'bxxxxxxx0 :
    8'bxxxxxx01 :
    8'bxxxxx011 :
    ...
    8'bx0111111 :
endcase
```

– if such structure cannot be obtained (if there are columns that have the same number of don't care conditions) => violation (message-2);

Note: If there are some bits that have don't care conditions in all case items than they are cut off and are not regarded.

Note-1: parameter MIN_CASE_SELECTION_WIDTH defines minimal length for case selection expression to activate the mechanism for second case (when there is no overlapping).

Note-2: elaboration-time checks are performed for parameter-dependent cases.

EXAMPLE-1: [1] there is a 'casex' statement within description;

[2] there is an overlapping of 'case' items that contain don't care conditions ('x') => violation (message-1).

```
reg [3:0] sel;

...

casex ( sel )
    4'bxx00: ...
    4'bx001: ...
    4'bx011: ...
    4'bx100: ...
    default: ...
endcase
```

Casex statement contains 2 overlapped case clause(s). Use 'do not care' condition carefully to avoid the duplication of case items.

'4'bxx00' is overlapped.

'4'bx100' is overlapped.

EXAMPLE-2: [1] there is a 'casex' statement within description;

[2] there is no duplication or overlapping of 'case' items;

[3] triangle structure cannot be obtained (fourth and sixth columns have the same number of don't care conditions – two ones) => violation (message-2).

```
reg [5:0] sel;

...

casex ( sel )
    6'bxxxx00: ...
    6'bxxx011: ...
    6'bxx0111: ...
    6'bx0111x: ...
    default  : ...
endcase
```

Casex statement contains items that may cause the generation of additional logic that decreases circuit quality. Describe priority logic w ith triangular form of 'do not care' conditions in case items.

EXAMPLE-3: [1] there is a 'casex' statement within description;
[2] there is no duplication or overlapping of 'case' items;
[3] triangle structure can be obtained (by columns replacing) => violation (message-2).

```
reg [5:0] sel;

...

casex ( sel )
     6'bxxxxx0: ...
     6'bxxx0x1: ...
     6'bxx01x1: ...
     6'b0x11x1: ...
     6'b1x1101: ...
     6'b101111: ...
     default  : ...
endcase

/* could be virtually transformed to
casex ( {sel[4], sel[1], sel[5], sel[3:2], sel[0]} )
     6'bxxxxx0: ...
     6'bxxxx01: ...
     6'bxxx011: ...
     6'bxx0111: ...
     6'bx01111: ...
     6'b011111: ...
     default  : ...
endcase
*/
```

2.8.5 Description relying on parallel_case is prohibited (Verilog only)

STARC_VLOG 2.8.5.1

RULE NAME	**Do not force parallel_case in a case statement directive that depends on a particular logic synthesis tool (Verilog only)**	
MESSAGE	**Do not force 'parallel_case' directives that depend on particular logic synthesis tool. RTL and post-synthesis simulation results may differ.**	
PROBLEM DESCRIPTION	In Verilog-HDL syntax, case statements are processed in order from the top line by a sequential process. If there are no overlaps in the clause described in the case statement, the logic synthesis tool interprets the values to be parallel and generates a circuit with no priority. If there is a duplicated value of clauses, the values are interpreted as a sequential process and a circuit with a priority is generated. If Design Compiler directive `//synopsys parallel_case` is specified, values will be forcibly treated as parallel case even if there are overlapping values. Therefore, there is a possibility of the RTL simulation results and gate level simulation results differing. So it is not recommended to use the directive. If the circuit is to be operated in parallel, a case statement without overlapping items should be used. If it is to be operated in priority, process should be described with help of an if-else-if construction.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker verifies case, casex, casez statements: – if pragma `//synopsys parallel_case` presents => violation	

EXAMPLE-1: [1] pragma `//synopsys parallel_case` is used => violation

```
casex (sel)                   //synopsys parallel_case
    4'bx0x0: out1 = a;
    4'b0010: out1 = b;
    4'b0100: out1 = c;
    4'b1000: out1 = d;
    default: out1 = 1'bx;
endcase
```

Do not force 'parallel_case' directives that depend on particular logic synthesis tool. RTL and post-synthesis simulation results may differ.

STARC_VLOG 2.8.5.2

RULE NAME	**Do not describe fixed values in the selection expression of a case statement**	
MESSAGE	**Selection expression of the case statement should not be constant.**	
PROBLEM DESCRIPTION	Constant selection expression is used with variable case clauses to get priority decoders. But variable case items violates rule 2.8.5.3. If selection expression and case clauses are constants then description does not have sense because same branch is always executed and case statement is optimized by synthesis tool.	
	LEVEL	RECOMMENDED 1
CHECKER BEHAVIOR	Checker verifies case, casex, casez statements – if selection expression is constant (or constant expression) => violation	

EXAMPLE-1: [1] selection expression of case statement is constant (parameter expression) => violation.

```
parameter tmp1 = 2'b10;
parameter tmp2 = 2'b10;

always
    ...
        case ( tmp1 & tmp2 )

            ...

        endcase
```

Selection expression of the case statement should not be constant.

STARC_VLOG 2.8.5.3

RULE NAME	**Do not describe variables (or the expression a + b) in the clause of a case statement**	
MESSAGE	**Case statement contains variable case clauses.**	
	DETAIL	*Case clause expression is not constant: “{CaseClauseExpr}”.*
PROBLEM DESCRIPTION	Variable case clauses may easily lead to items overlapping. In such situation case can not be treated as parallel and circuit with priority is generated by synthesis tool.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies case, casex, casez statements – if case item is variable => violation.	

EXAMPLE-1: [1] case statement contains variable case clauses => violation;

[2] there are two variable case items: signal is used as case item, expression with signal and parameter (result of expression is variable).

```
reg tmp;
parameter tmp_param;

always

    ...

    case ( selection )

        tmp                 : ...
        tmp & tmp_param     : ...

        ...

    endcase
```

Selection expression of the case statement should not be constant.

Case clause expression is not constant: “tmp”.

Case clause expression is not constant: “tmp&tmp_param”.

STARC_VLOG 2.8.5.4

RULE NAME	**Do not describe logical operations and arithmetic operations in the selection expression of a case statement (Verilog only)**	
MESSAGE	**Do not describe logical operations and arithmetic operations in the selection expression of a 'case' statement.**	
PROBLEM DESCRIPTION	Logical or arithmetic operations in 'case' selection expression make it less obvious and take more time to debug and understand. Moreover, such operations may affect the length of the result. For example, logical operation with vectors returns one-bit result, while using logical operation can be simply a misprint. It is safer to assign the expression to some temporary signal with explicitly defined width (bit widths in assignments are checked by another rules), and then use this signal solely as the condition for a 'case' selection expression.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker verifies selection expression of case statements: – if arithmetic and logical operations is detected => violation Note: index (bit- or part-selection) expressions are excluded from checking	

EXAMPLE-1: [1] logical operation is used in selection expression of case statement => violation

```
case( sel1 || sel2 )

    1'b0: out1 = in1;
    1'b1: out1 = in2;

endcase
```

Do not describe logical operations and arithmetic operations in the selection expression of a 'case' statement.

EXAMPLE-2: [1] bit-wise operation is used in selection expression => no violation

```
case( sel1 | sel2 )

    1'b0: out1 = in1;
    1'b1: out1 = in2;

endcase
```

EXAMPLE-3: [1] logical and arithmetic operations are used in selection expression => violation
Note: only one message generated for one case statement

```
case( sel1 && sel2 + sel3 )

    1'b0: out1 = in1;
    1'b1: out1 = in2;

endcase
```

Do not describe logical operations and arithmetic operations in the selection expression of a 'case' statement.

2.8.6 Beware of nesting in which if statements and case statements coexist (2.8.4 in the VHDL version)

STARC_VLOG 2.8.6.1

RULE NAME	**When complex nested if statements and case statements co-exist, it is more advantageous to have fewer conditionals with multiple matches than fewer matches with multiple conditionals**	
MESSAGE	**'case' statement contains multiple 'if' statements with the same structure - possibility of generating a redundant and larger circuit exists. Description of one 'if' statement with multiple conditional branches, containing one 'case' statement per branch, is recommended (state machine description is an exception).**	
PROBLEM DESCRIPTION	With nesting where 'if' statements and case statements coexist, it is more advantageous to have fewer 'if' conditionals with multiple matches in 'case' statements. When starting the description, try to describe using this structure if possible. There is chance that both descriptions (with multiple 'case' statements and with multiple 'if' statements) will infer the same optimal circuit. But when describing one 'case' statement, containing 'if' statement with multiple conditional branches per each item possibility of generating redundant logic increases. However, in cases where coding in this manner might increase the amount of code, it would result in an increased number of input signals. This could result in worse circuit performance. Circuit structure should be kept in mind when creating descriptions. In the case of state machines, following the recommendation lowers readability, thus making debugging difficult. State machines are an exception to this recommendation; following it also lowers the quality of results, especially area.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker detects detect 'case', 'casex', casez' statements: – if statements satisfy following conditions: – every 'case' item contains only one 'if' statement; – 'if' statements have same branch structure; – **note:** if 'case' statement has 'default' clause: – if default clause is optimized => the clause is not verified at all; – default clause is not optimized: – 'default' contains only 'if' statement with the same structure => the clause is considered as item; – otherwise => the clause is not considered; – number of case items is calculated (CASE_ITEM_COUNT) calculate the number 'if' branches (IF_BRANCH_COUNT): – if CASE_ITEM_COUNT belongs to [CASE_ITEM_MIN:CASE_ITEM_MAX] range and IF_BRANCH_NUM belongs to [IF_BRANCH_MIN:IF_BRANCH_MAX] range => violation – **exception:** if 'if' statements has parallel structure (exclusive conditions) => such situations are skipped: `if (A && !B) statement1 else if (B && !A) statement2;` Note: parameters CASE_ITEM_MIN, CASE_ITEM_MAX, IF_BRANCH_MIN, IF_BRANCH_MAX define ranges (of 'case' items number and 'if' branch number) to be verified by the checker. Default parameters values: – CASE_ITEM_MIN = 8; – CASE_ITEM_MAX = 0; – IF_BRANCH_MIN = 1; – IF_BRANCH_MAX = 4.	

RULE NAME	**When complex nested if statements and case statements co-exist, it is more advantageous to have fewer conditionals with multiple matches than fewer matches with multiple conditionals**
	'0' (or negative) value means that bound is not set.

EXAMPLE-1: [1] every 'case' item contains only one 'if' statement;

[2] every 'if' statement has the same branch structure;

[3] CASE_ITEM_COUNT = 4 (belongs to [0:4]), IF_BRANCH_NUM = 3 (belongs to [1:4]) => violation.

Note: parameters values are default ones except CASE_ITEM_MIN, its value is set to 4 to simplify the example.

'case' statement contains multiple 'if' statements w ith the same structure - possibility of generating a redundant and larger circuit exists. Description of one 'if' statement w ith multiple conditional branches, containing one 'case' statement per branch, is recommended (state machine description is an exception).

```
case ( sel )
    2'b00: if (a) statement1 else if (b) statement2 else statement3;
    2'b01: if (a) statement4 else if (b) statement5 else statement6;
    2'b10: if (a) statement7 else if (b) statement8 else statement9;
    2'b11: if (a) statement10 else if (b) statement11 else statement12;
endcase
```

EXAMPLE-2: [1] every 'case' item contains only one 'if' statement;

[2] every 'if' statement has the same branch structure;

[3] 'default' clause contains 'if' statement with same structure, but 'default' is optimized (all possible values of signal `sel` are listed within 'case' items);

[3] CASE_ITEM_COUNT = 4 (does not belong to [0:5]) => no violation.

Note: parameters values are default ones except CASE_ITEM_MIN, its value is set to 5 to simplify the example.

```
reg [1:0] sel;

case ( sel )
    2'b00: if (a) statement1 else statement2;
    2'b01: if (a) statement3 else statement4;
    2'b10: if (a) statement5 else statement6;
    2'b11: if (a) statement7 else statement8;
    default: if (a) statement9 else statement10;
endcase
```

2.9 *for statements*

2.9.1 Do not use for statements other than for simple repeating statements

STARC_VLOG 2.9.1.1

RULE NAME	**Do not use for statements other than for simple repeating statements.**	
MESSAGE	**'for' statement infers cascade logic. Such descriptions are dangerous because logic synthesis tools could create improper balanced tree structure. It is recommended to avoid 'for' statements other than simple repeating statements.**	
PROBLEM DESCRIPTION	'for' statement should be used only for simple repeating. When parity check circuit is described, balanced tree structure is inferred according to the timing constraints. In case when sets of assignment source and target intersect (see the examples below) cascade logic is generated. Balanced tree structure would be inferred depending on the timing constraints of the logic synthesis tool, but if this series is too long, it is uncertain whether the tree structure would be inferred properly.	
	LEVEL	RECOMMENDATION 2

CHECKER BEHAVIOR	Checker scans blocking procedural assignments ('=') within 'for' loops:
	– if in the same procedural assignment the set of right-hand side (RHS) signals (after loop unrolling) intersects with the set of left-hand side (LHS) signals => violation
	Note: if the RHS of an expression is single signal or the signal from the same iteration => no violation:
	`for ( i = 0; i <=3; i = i + 1 )`
	`    out = out;`	

EXAMPLE-1: [1] the set of RHS signals intersects with the set of LHS signals (index from another iteration is used) => violation

```
for ( i = 0; i <= 3; i = i + 1 )
    out[i] = out[i+1];
```

'for' statement infers cascade logic. Such descriptions are dangerous because logic synthesis tools could create improper balanced tree structure. It is recommended to avoid 'for' statements other than simple repeating statements.

EXAMPLE-2: [1] the set of RHS signals intersects with the set of LHS signals (variable itself is used as RHS and LHS) => violation

```
for ( i = 0; i <= 3; i = i + 1 )
    out = out ^ some_sig[i];
```

'for' statement infers cascade logic. Such descriptions are dangerous because logic synthesis tools could create improper balanced tree structure. It is recommended to avoid 'for' statements other than simple repeating statements.

EXAMPLE-3: [1] the set of RHS signals does not intersect with the set of LHS signals => no violation.

```
for ( i = 0; i <= 7; i = i + 2 ) //loop variable increment is 2 => even bits are in set of LHS
    out[i] = out[i+1];            //shift in RHS is 1 => odd bits are in set of RHS
```

STARC_VLOG 2.9.1.2

RULE NAME	**Initial value and conditions of for statement should be constant, in addition do not change the values within a loop variable**	
MESSAGE-1	**Initial value, condition and increment of the 'for' statement should be constant**	
	DETAIL-1	*Loop variable initialization value is not a constant*
	DETAIL-2	*Conditional expression does not contain a comparison operator*
	DETAIL-3	*Loop variable must be compared with a constant*
	DETAIL-4	*Loop step expression does not contain an arithmetic operator*
	DETAIL-5	*Loop step expression must operate with the loop variable and a constant*
MESSAGE-2	**Loop variable is not the same in all parts of the 'for' statement, while it should be the same.**	
MESSAGE-3	**Loop variable is modified within a 'for' statement. It is allowed to modify loop variable only in the incremental part of the 'for' statement.**	

PROBLEM DESCRIPTION	'For' statements define loop iteration by numerically increasing the index variable. It is described by using following form:
	`for( initialization_expression; conditional_expression; loop_step_expression )`
	` begin`
	`    // statement body`
	` end`
	that should be described carefully to synthesize a good circuit. Following rules should be considered when describing 'for' statement:
	– loop variable should be initialized with a constant in the initialization expression
	– loop iteration conditional expression should be comparison with a constant
	– loop step increment should be an expression with arithmetic operator operating with loop variable and constant
	– loop variable should be the same in all parts of 'for' statement
	– loop variable should not be modified within a 'for' statement body
	Using these principles minimize possible problems with synthesis tools, since violation of these principles leads either to synthesis failure or inappropriate results.	

	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies all constraints of this problem: – initial value, condition and increment should be constant – if initialization value is not a constant => violation message-1 and detail-1 – if conditional expression doesn't contains comparison operator => violation message-1 and detail-2 – if conditional expression is comparison, but not of the loop variable and constant => violation message-1 and detail-3 – if loop step expression doesn't contains an arithmetic operator => violation message-1 and detail-4 – if loop step expression contains an arithmetic operator but operates not with loop variable and constant => violation message-1 and detail-5 – if loop variable is not same in all parts of 'for' statement => violation message-2 – loop variable is variable assigned in the initialization expression – if loop variable is modified within 'for' statement body => violation message-3	

EXAMPLE-1: [1] loop variable initialization value is not a constant => violation (message-1, detail-1);

```
for( i = REG_A; i < 100; i = i + 1 ) begin
     ACCUM = ACCUM + i;
end
```

initial value, condition and increment of the 'for' statement should be constant

Loop variable initialization value is not a constant

EXAMPLE-2: [1] conditional expression doesn't contains comparison operator => violation (message-1, detail-2);

```
for( i = 0; i + 100; i = i + 1 ) begin
     ACCUM = ACCUM + i;
end
```

initial value, condition and increment of the 'for' statement should be constant

Conditional expression does not contain a comparison operator

EXAMPLE-3: [1] loop variable is compared not with a constant => violation (message-1, detail-3)

```
for( i = 0; i < REG_A; i = i + 1 ) begin
     ACCUM = ACCUM + i;
end
```

initial value, condition and increment of the 'for' statement should be constant

Loop variable must be compared w ith a constant

EXAMPLE-4: [1] loop step expression doesn't contains an arithmetic operator => violation (message-1, detail-4);
[2] loop variable is modified within the loop body => violation (message-3);
[3] loop variable is compared with parameter in the conditional expression => no violation

```
parameter N = 100;

for( i = 0; i < N; i = i ) begin
     ACCUM = ACCUM + i;
     i = i + 1;
end
```

initial value, condition and increment of the 'for' statement should be constant

Loop step expression doesn't contain an arithmetic operator

Loop variable is modified w ithin a 'for' statement. It is allow ed to modify loop variable only in the incremental part of the 'for' statement.

EXAMPLE-5: [1] loop step expression operates with loop variable and not a constant => violation (message-1, detail-5)

```
for( i = 0; i < 100; i = i + REG_A ) begin
     ACCUM = ACCUM + i;
end
```

Loop step expression must operate w ith the loop variable and a constant

EXAMPLE-6: [1] loop variable is not the same in initialization and loop step expression => violation (message-3)

```
for( i = 0; i < 100; j = i + 1 ) begin
     ACCUM = ACCUM + i;
end
```

Loop variable is not the same in all parts of the 'alw ays' statement, w hile should be the same

2.9.2 Limiting loop-variable operation in *for statements*

STARC_VLOG 2.9.2.1

RULE NAME	**Do not describe any arithmetic operations other than with loop variable and constant**	
MESSAGE	**Argument of an arithmetic operation inside the loop is not a loop variable or a constant.**	
	DETAIL	*{ObjectClass} "{ObjectName}" is not a loop variable or a constant*

PROBLEM DESCRIPTION	Consider following code:
	`for( i = 0; i <= 15; i = i + 1 ) begin`
	`   REG_Z[i] = REG_A + REG_B;`
	`end`
	'for' statement contains an arithmetic operation not with loop variable. Descriptions defined in 'for' statements are copied for the number of specified loops. Therefore, additional circuits are created to execute "REG_A + REG_B" 16 times, as defined by the loop => area is increased. Following code describes arithmetic operation outside 'for' statement:
	`TMP = REG_A + REG_B;`
	`for( i = 0; i <= 15; i = i + 1 ) begin`
	`   REG_Z[i] = TMP;`
	`end`
	Area increase problem (described above) will not occur if executing an arithmetic operations with loop variable and constant values only. In general, common statements should be kept outside 'for' statements as much as possible.	

	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans 'for'-loops and detects an arithmetic operations (+, -, *, /, %, **) within them: – if any argument of arithmetic operation is not a loop variable or constant => violation Note-1: violation is generated for whole vector in case of bit-selections and slices with non-constant indexes Note-2: loop variable is such signal (or signals which comprise a concatenation) that is assigned in the initialization expression of the 'for' statement (and all upper-level 'for' statements in case of their nesting).	

EXAMPLE-1: [1] 'for' loop contains an arithmetic operation with loop variable and parameter => no violation

```
parameter INC = 1;
for( i = 0; i < 8; i = i + 1 ) begin
    RES[i] = i + INC;
end
```

EXAMPLE-2: [1] 'for' loop contains an arithmetic operation with signals (neither loop variable nor constant) => violation;

[2] different classes "{ObjectClass}" are demonstrated (port and signal)

```
input [3:0] INP_A;
reg   [3:0] REG_B;
    ...
    for( i = 0; i < 8; i = i + 1 ) begin

        RES[i] = INP_A * REG_B;

    end
```

Argument of an arithmetic operation inside the loop is not a loop variable or a constant.

Port "INP_A" is not a loop variable or a constant

Signal "REG_B" is not a loop variable or a constant

EXAMPLE-3: [1] embedded 'for' contains an arithmetic operation with upper-level loop variable and constant => no violation;

[2] upper-level 'for' contains an expression with arithmetic and non-arithmetic operators => violation for arguments (that are not loop variables or constants) of arithmetic operators only

```
for( i = 0; i < 8; i = i + 1 ) begin
    RES[i] = A | B | C + i;
    for( j = 0; j < 4; j = j + 1 ) begin
        TMP[j] = i + j - 1;
    end
end
```

Argument of an arithmetic operation inside the a loop is not a loop variable or a constant

Signal "C" is not a loop variable or a constant

STARC_VLOG 2.9.2.2

RULE NAME	**Do not describe any logical or relational operations other than with loop variables and constants**
MESSAGE	**Argument of a {OperationType} operation inside the loop is not a loop variable or a constant. Such descriptions should be avoided because they could lead to decrease of the circuit operation speed.**
DETAIL	*{ObjectClass} "{ObjectName}" is not a loop variable or a constant.*

PROBLEM DESCRIPTION

Consider following code:

```
for( i = 0; i <= 8; i = i + 1 ) begin
     if ( i <= sel )
          q[i] <= data[i];
end
```

'for' statement contains a relational operation not with loop variable. If a relational operator exists in 'for' statement, a comparison circuit is created for loop count in the same way as an arithmetic operation thereby resulting in a degradation of circuit quality. Even if loop count is limited, generated circuits may be lined up in a series according to the loop count in a 'for' statement and operation speed decreases in the circuit in this case. Therefore, it is recommended to avoid describing relational operations and logical operation in 'for' statement.

Description which contains relational operations may be rewritten using 'case' statement to avoid described problems. Considered example may be rewritten in the following way:

```
case ( sel )
     3'b000 : q[0] <= data[0];
     3'b001 : q[1:0] <= data[1:0];
     3'b010 : q[2:0] <= data[2:0];
     3'b011 : q[3:0] <= data[3:0];
     3'b100 : q[4:0] <= data[4:0];
     3'b101 : q[5:0] <= data[5:0];
     3'b110 : q[6:0] <= data[6:0];
     3'b111 : q[7:0] <= data[7:0];
default : q <= 8'bxxxxxxxx;
endcase
```

If logical operation exist within 'for' statement degradation of circuit quality may be avoided by using 'case' statement in the described way. Also such common expression may be kept outside 'for' statement (see 2.9.2.1 for details).

LEVEL	RECOMMENDATION 3

CHECKER BEHAVIOR

Checker scans for loops and detects expressions, where relational ('>', '<', '>=', '<=') or logical ('&&', '||', '!') operation is used::

– if any argument of this operation is not a loop variable or constant => violation.

Note-1: see rule 2.9.2.1 for loop variable definition.

Note-2: see rule 1.1.1.2 for {ObjectClass} substitution table.

Note-3: {OperationType} is defined by the following table:

{OperationType}
logical
relational
logical/relational

EXAMPLE-1: [1] logical operation detected within 'for' loop;
[2] both arguments of the operation are signals => violation.

```
reg [8:0] A;
reg [8:0] B;

for( i = 0; i < 8; i = i + 1 ) begin

     RES[i] = A[i] || B[i];
end
```

Argument of a logical operation inside the loop is not a loop variable or a constant. Such descriptions should be avoided because they could lead to decrease of the circuit operation speed.

Signal "B" is not a loop variable or a constant.

Signal "A" is not a loop variable or a constant.

EXAMPLE-2: [1] relational operation detected within 'for' loop;
[2] arguments of the operation are loop variable and parameter => no violation.

```
parameters max_size = ...;

for( i = 0; i < 4; i = i + 1 ) begin

     if ( i <= max_size )
          RES[i] = A[i];
end
```

STARC_VLOG 2.9.2.3

RULE NAME	**The range of the number of loops is up to 10 if operating logically or relationally other then with loop variables and constants**	
MESSAGE-1	**The range of the number of loops is {LoopNumber}. When 'for' loop description infers comparison circuit, it is recommended to limit the loop variable to up to {LOOP_NUMBER} times to avoid a decrease in circuit quality.**	
MESSAGE-2	**The range of the number of loops could be greater than recommended value {LOOP_NUMBER}. When 'for' loop description infers a comparison circuit, it is recommended to limit the loop variable to avoid a decrease in circuit quality.**	
PROBLEM DESCRIPTION	If a relational operator exists in 'for' statement, a comparison circuit is created for loop count in the thereby resulting in a degradation of circuit quality. The same situation occurs with arithmetic operations. If description using such operations can not be rewritten, limit the loop variable to up to 10 times in order to avoid a decrease in circuit quality (see also 2.9.2.2).	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans 'for' loops for relational or logical operations other than with loop variables and constants (see 2.9.2.2): – if it is possible to calculate the range between initial value and iteration condition: – if the range is greater than LOOP_NUMBER parameter value => violation (message-1); – if the initial value or the iteration condition are expressions with signals, variables => violation (message-2). Note-1: value of LOOP_NUMBER parameter is defined in configuration file (default value is 10). Note-2: this checker triggers concurrently with 2.9.2.2.	

EXAMPLE-1: [1] relational operation detected within 'for' loop;
[2] arguments of the operation are loop variable and signal;
[3] range between initial value and iteration condition is greater than LOOP_NUMBER parameter value (by default) => violation (message-1).

```
reg [15:0] A;

for( i = 0; i < 16; i = i + 1 )
    begin

        if ( i <= B )
            RES[i] = A[i];
    end
```

The range of the number of loops is 16. When 'for' loop description infers comparison circuit, it is recommended to limit the loop variable to up to 10 times to avoid a decrease in circuit quality.

EXAMPLE-2: [1] relational operation detected within 'for' loop;
[2] arguments of the operation are loop variable and signal;
[3] range between initial value and iteration condition is 8 (loop variable step is 2), this number is less than LOOP_NUMBER parameter value (by default) => no violation.

```
reg [15:0] A;

for( i = 0; i < 16; i = i + 2 )
    begin

        if ( i <= B )
            RES[i] = A[i];
        end
```

EXAMPLE-2: [1] logical operation detected within 'for' loop;
[2] both arguments of the operation are signals;
[3] range between initial value and iteration can not be calculated => violation (message-2).
Note: LOOP_NUMBER parameter value is set to 5.

```
reg [8:0] A;
reg [8:0] B;
reg loop_range;

for( i = 0; i < loop_range; i = i + 1 )
     begin

          RES[i] = A[i] || B[i];
     end
```

The range of the number of loops could be greater than recommended value 5. When 'for' loop description infers a comparison circuit, it is recommended to limit the loop variable to avoid a decrease in circuit quality.

STARC_VLOG 2.9.2.4

RULE NAME	**Use for-loop separately from the reset part and the logic part**	
MESSAGE	**Detected use of the asynchronous reset and logic parts in the same for loop statement. Use for loop statement separately from the reset and the logic parts, because such description may not be supported by synthesis tools.**	
	DETAIL	*'for' statement includes asynchronous reset and logic parts.*
PROBLEM DESCRIPTION	Describing flip-flops with asynchronous controls within the same 'for' loop statement is not recommended. Such style is not supported by all synthesis tools and may lead to undesirable circuit to be generated. To fix the problem control and logic part should be put to different 'for' statements.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans 'always' statements those infer flip-flops and include 'for' loops: – if asynchronous set/reset condition is used inside the same 'for' loop statement with synchronous part => violation	

EXAMPLE-1: [1] reset part and logic part are in the same 'for' loop => violation.

```
always @( posedge clk or posedge rst )

    for ( i = 0; i <= 3; i = i + 1 )
        begin
           if ( rst )
               out[i] <= 1'b0;
           else
               out[i] <= data[i];
       end
```

Detected use of the asynchronous reset and logic parts in the same for loop statement. Use for loop statement separately from the reset and the logic parts, because such description may not be supported by synthesis tools.

'for' statement includes asynchronous reset and logic parts.

EXAMPLE-2: [1] separate 'for' loops with reset and logic parts => no violation.

```
always @( posedge clk or posedge rst )
    begin

       if ( rst )
           for ( i = 0; i <= 3; i = i + 1 ) // separate loop
               out[i] <= 1'b0;
       else
           for ( i = 0; i <= 3; i = i + 1 ) // separate loop
               out[i] <= data[i];
    end
```

2.10 Operator description

2.10.1 Order of operators and assignment of 'x'

STARC_VLOG 2.10.1.4

RULE NAME	**Do not compare with 'x' or 'z'**	
MESSAGE	**Comparison operand contains: {ExprValue}. Do not compare with 'x' or 'z'.**	
PROBLEM DESCRIPTION	Comparison operators may return 1 if the expression is true and 0 if the expression is false. But if there are any 'x' or 'z' bits in the operands the the result becomes unknown and expression takes the value 'x'. So do not compare with 'x' or 'z' to get correct comparison results. Moreover, comparison with 'x' is ignored in logic synthesis tools.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans comparison operators ==, !=, <, <=, >, >= and verifies only expressions that contain signals on one of the sides (pure constant expressions are optimized at the compilation or elaboration stage): – if another side contains constants, which includes 'x' or 'z' bits (3'bxxx, 'bz, 2'bx0, etc) => violation	

EXAMPLE-1: [1] signal is compared with value containing one 'z' bit => violation

```
if ( tmp == 2'b1z )
    ...
```

Comparison operand contains:2'b1z. Do not compare w ith 'x' or 'z'.

EXAMPLE-2: [1] loop variable is compared with 32 bits in length unknown value => violation

```
for ( i = 0; i <= 'bx; i = i + 1 )
    ...
```

Comparison operand contains:
32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.
Do not compare w ith 'x' or 'z'.

STARC_VLOG 2.10.1.5

RULE NAME	**Do not assign 'x' except for the default clause of a case statements**
MESSAGE	**An assignment with an 'x' in the right-hand side is detected. The results of RTL and post-synthesis simulation may not match.**
PROBLEM DESCRIPTION	Assigning 'x' is treated as don’t-care, and logic synthesis tools generate a circuit with an unknown value (either 0 or 1). This leads to a simulation result mismatch between RTL and gate-level. This kind of description is risky and should be avoided. The exception is the use of case statements and default clauses. If the unknown value ’x’ is assigned to the output of the default clause, it is regarded as don’t-care and a non-determined value is output due to the optimization result. Setting a non-determined value can usually decrease the size of the circuit more than setting a fixed value in the default clause does. From the simulation point of view, assignment of 'x' in the default clause helps to find missing case clauses and bad case selection expression values ('x' propagation).
LEVEL	RULE
CHECKER BEHAVIOR	Checker scans Verilog assignments: – procedural: blocking (=), non-blocking (<=); – continuous: assign, force statement; – initialization assignments Checker verifies detected assignments: – if right hand side contain only constants and unknown value ('x' or 'z') is used => violation

EXAMPLE-1: [1] unknown value is used in continuous assignment => violation

```
assign {out1,out2} = {in1, {2{1'bx}}, in2}
```

An assignment w ith an 'x' in the right-hand side is detected. The results of RTL and post-synthesis simulation may not match.

EXAMPLE-2: [1] unknown value is used in assignment in case item => violation

```
case (sel)
    2'b01 : out1 = 2'0x;
    2'b10 : out1 = 2'11;
    default : out1 = 2'00;
endcase
```

An assignment w ith an 'x' in the right-hand side is detected. The results of RTL and post-synthesis simulation may not match.

EXAMPLE-3: [1] unknown value is assigned in default clause => no violation

```
case (sel)
    2'b01 : out1 = 2'00;
    2'b10 : out1 = 2'11;
    default : out1 = 2'xx;
endcase
```

STARC_VLOG 2.10.1.6

RULE NAME	**Do not use values including 'x' or 'z'**	
MESSAGE	**Illegal constant: {ExprValue}. Do not use values containing 'x' or 'z'.**	
PROBLEM DESCRIPTION	A value that includes 'x' or 'z' is correct in terms of syntax, but errors will occur when using logic synthesis tools because value is unknown (may be 0 or 1). So do not use such values.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans description for constant operands containing 'x' or 'z' values: – if pure constant assignment => no violation (see 2.10.1.5) – if comparison operators (==, !=, <, <=, >, >=) => no violation (see 2.10.1.4) – if case equality operators (===, !==) => violation – if increment/decrement loop variable => violation – if non-constant assignment (containing signals in RHS) => violation Note-1: expressions in initialization assignments are also checked. Note-2: implicit comparison operators are also considered (it is allowed by Verilog language to specify pure constant expression in the if condition, case selection expression, ternary condition)	

EXAMPLE-1: [1] unknown value is used in the expression assigned in default clause of case statement (second argument is signal) => violation;

[2] pure constant expression (only one constant) which contains unknown values is assigned => no violation;

[3] pure constant expression (constant and parameter) which contains unknown values is assigned => no violation

```
parameter p = 3'b101;
reg tmp;
...
case (sel)
    1'b0: out1 <= 3'bxxx;
    1'b1: out1 <= 3'bzzz & p;
    default: out1 <= tmp + 3'bxxx;
endcase
```

Illegal constant: 3'bxxx. Do not use values containing 'x' or 'z'.

EXAMPLE-2: [1] increment loop variable expression contains unknown value => violation

```
integer i;
for (i = 0; i < 7; i = i + 1'bx)
    ...
```

Illegal constant: 1'bx. Do not use values containing 'x' or 'z'.

EXAMPLE-3: [1] unknown value is used in the expression in initialization assignment => violation

```
reg [1:0] tmp1;
reg [1:0] tmp2 = 2'b1x & tmp1;
```

Illegal constant: 2'b1x. Do not use values containing 'x' or 'z'.

EXAMPLE-4: [1] implicit comparison with one bit unknown value => violation

```
assign  out1 = ( 1'bx ) ? in1 : in2 ;
```

Illegal constant: 1'bx. Do not use values containing 'x' or 'z'.

2.10.3 Match the bit width of the left side and the right side (Verilog only)

STARC_VLOG 2.10.3.1

RULE NAME	**Clearly match the bit widths of the right side and the left side of relational operators (Verilog only)**	
MESSAGE	**Right side bit width "{RHSWidth}" of relational operator "{OpName}" does not match to bit width "{LHSWidth}" of the left side. Use equal bit widths to avoid confusion.**	
PROBLEM DESCRIPTION	When two operands of different bit lengths are used and one or both of the operands is unsigned, the smaller operand is zero filled on the most significant bit side to extend to the size of the larger operand. Such extension may easily lead to mistakes or misunderstanding. Clearly match the bit widths of the right side and the left side of relational operators to avoid any confusions.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects relational operators (<, >, <=, >=): – if argument bit width at the right side do not match to bit width of argument at the left side => violation. Note: bit widths of decimal constants are defined by their values; violation is issued only for decimal constants that are wider than other side of comparison (narrower decimal constants are allowed).	

EXAMPLE-1: [1] argument bit width at the right side do not match to bit width of argument at the left side => violation

```
wire [7:0] W_SLOT_A;
wire [5:0] W_SLOT_B;
wire [7:0] W_SLOT_C;
wire [7:0] W_OR;

assign W_OR = (W_SLOT_A < W_SLOT_B) ? (W_SLOT_A | W_SLOT_B) :  (W_SLOT_A | W_SLOT_C)
```

Right side bit w idth “6” of relational operator “<” does not match to bit w idth “8” of the left side. Use equal bit w idths to avoid confusion.

STARC_VLOG 2.10.3.2

RULE NAME	**Match the bit width of the assignment signal and the operand of logical operators (Verilog only)**	
MESSAGE-1	**Bit-wise expression width "{ExprWidth}" is {CompareResult} than bit width "{DestWidth}" of the assignment destination. Match bit widths exactly when using bit-wise operators.**	
	DETAIL	*Bit width of the argument is "{ArgBitWidth}"*
MESSAGE-2	**Bit-wise expression arguments have different bit width. Match bit widths exactly when using bit-wise operators.**	
	DETAIL	*Bit width of the argument is "{ArgBitWidth}"*
PROBLEM DESCRIPTION	Logical operator bit width should match to bit width of assignment destination. When bit width of right-hand-side it greater than bit width of assignment destination => upper bits of right-hand-side are truncated. Otherwise, when bit width of right-hand-side is less than bit width of assignment destination => upper bits of destination are filled with zeros. Descriptions with different bit widths may be made inadvertently – they are implicit and readability of the description drops. Concatenations/part-selections should be used to describe filling/truncation explicitly.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker verifies each assignment ('=', '<=', 'assign') where right-hand-side is expression with bit-wise operators only: – bit width of each operand should be the same as bit width of assignment destination: – if bit width of whole right-hand-side expression is greater than bit width of assignment target => violation (message-1: {CompareResult} = "greater", detail per each argument with erroneous bit width) – if bit width of whole right-hand-side expression is less than bit width of assignment target => violation (message-1: {CompareResult} = "less", detail per each argument with erroneous bit width) – if bit width of whole right-hand-side expression is equal to bit width of assignment target => each argument of the expression is verified if it has less bit width than assignment destination (message-2 is displayed in case of violation with one detail per each erroneous argument) Note-1: this rule can be dependent on parameters or hierarchical references => elaboration-time checking is required for such cases Note-2: this checker verifies only cases where right-hand-side is an expression with bit-wise operators only. This is made intentionally, since checker 2.10.3.3 verifies cases where right-hand-side is expression of other type (for example, an expression with bit-wise and arithmetical operators)	

EXAMPLE-1: [1] right-hand-side expression contains bit-wise operators only;

[2] bit width of whole expression is greater than assignment destination bit width => violation (message-1);

[3] note: expression includes operands with equal, greater and less bit widths either;

```
wire [7:0] W_SLOT_A;
wire [9:0] W_SLOT_B;
wire [5:0] W_SLOT_C;
wire [7:0] W_OR;

assign W_OR = W_SLOT_A |  W_SLOT_B | W_SLOT_C;
```

Bit-w ise expression w idth "10" is greater than bit w idth "8" of the assignment destination. Match bit w idths exactly w hen using bit-w ise operators.

Bit w idth of the argument is "10"

Bit w idth of the argument is "6"

EXAMPLE-2: [1] right-hand-side expression contains bit-wise operators only;

[2] bit width of whole expression is less than assignment destination bit width => violation (message-1);

```
wire [5:0] W_SLOT_B;
wire [6:0] W_SLOT_C;
wire [7:0] W_OR;

assign W_OR = W_SLOT_B | W_SLOT_C;
```

Bit-w ise expression w idth "7" is less than bit w idth "8" of the assignment destination. Match bit w idths exactly w hen using bit-w ise operators.

Bit w idth of the argument is "6"

Bit w idth of the argument is "7"

EXAMPLE-3: [1] right-hand-side expression contains bit-wise operators only;

[2] bit width of whole expression is equal to assignment destination bit width, but it contains operand with less bit width => violation (message-2);

```
wire [5:0] W_SLOT_B;
wire [7:0] W_SLOT_C;
wire [7:0] W_OR;

assign W_OR = W_SLOT_B | W_SLOT_C;
```

Bit-w ise expression arguments have different bit w idth. Match bit w idths exactly w hen using bit-w ise operators.

Bit w idth of the argument is "6"

EXAMPLE-4: [1] right-hand-side expression contains not only bit-wise operators => no violation of this rule even for operands with erroneous bit width (this is case for 2.10.3.3)

```
wire [7:0] W_SLOT_A;
wire [5:0] W_SLOT_B;
wire [8:0] W_SLOT_C;
wire [7:0] W_OR;

assign W_OR = W_SLOT_B | W_SLOT_C + W_SLOT_A;
```

STARC_VLOG 2.10.3.3

RULE NAME	**The bit width of the right-hand side of an assignment should not be wider than the left-hand side of the assignment (Verilog only)**	
MESSAGE	**Assignment source bit width "{SourceWidth}" is greater than destination bit width "{DestWidth}". Upper bits of the right-hand side will be truncated. Match bit widths exactly to improve the readability of the description.**	
PROBLEM DESCRIPTION	When bit width of right-hand-side it greater than bit width of assignment destination => upper bits of right-hand-side are truncated. Otherwise, when bit width of right-hand-side is less than bit width of assignment destination => upper bits of destination are filled with zeros. Descriptions with different bit widths may be made inadvertently – they are implicit and readability of the description drops. Concatenations/part-selections should be used to describe filling/truncation explicitly.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker verifies each assignment ('=', '<=', 'assign') where right-hand-side expression doesn't belong to the following set: – pure function call (it is case for 2.1.3.2) – signal/expression of integer type (it is case for 2.10.4.3) – expression including bit-wise operators only (it is case for 2.10.3.2) – signal/expression of signed type (it is case for 2.10.6.2) Checker works for all another cases (bit width of right hand side should be the same as bit width of assignment destination): – if bit width of right-hand-side expression is greater than bit width of assignment target => violation Note-1: this rule can be dependent on parameters or hierarchical references => elaboration-time checking is required for such cases	

EXAMPLE-1: [1] right-hand-side doesn't belong to the set (described in the ""checker behavior" section);
[2] right-hand-side is wider than assignment target => violation

```
wire [7:0] W_SLOT_A;
wire [5:0] W_SLOT_B;
wire [8:0] W_SLOT_C;
wire [7:0] W_OR;

assign W_OR = W_SLOT_B | W_SLOT_C + W_SLOT_A;
```

Assignment source bit w idth "9" is greater than destination bit w idth "8". Upper bits of the right-hand side w ill be truncated. Match bit w idths exactly to improve the readability of the description.

EXAMPLE-2: [1] right-hand-side is wider than assignment target;
[2] right-hand-side belongs to the set (described in the "checker behavior" section): it is pure integer expression => no violation (it is case for 2.10.4.3);

```
integer        SLOT_A;
integer        SLOT_B;
reg     [15:0] R_SUM;

assign R_SUM = SLOT_A + SLOT_B;
```

STARC_VLOG 2.10.3.4

RULE NAME	**The bit width of the right-hand side of an assignment should not be narrower than the left-hand side of the assignment (Verilog only)**	
MESSAGE	**Assignment source bit width "{SourceWidth}" is less than destination bit width "{DestWidth}". Upper bits of the right-hand side will be filled with zeroes. Match bit widths exactly to improve the readability of the description.**	
PROBLEM DESCRIPTION	When bit width of a right-hand side is less than bit width of the assignment destination => upper bits of destination are filled with zeros. Descriptions with different bit widths may be made inadvertently – they are implicit and readability of the description drops. Concatenations should be used to describe filling explicitly.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker verifies each assignment ('=', '<=', 'assign') where right-hand-side expression doesn't belong to the following set: – pure function call (it is case for 2.1.3.2) – signal/expression of integer type (it is case for 2.10.4.3) – expression including bit-wise operators only (it is case for 2.10.3.2) – signal/expression of signed type (it is case for 2.10.6.2) Checker works for all another cases: – if bit width of right-hand-side expression is less than bit width of assignment target => violation Note-1: this rule can be dependent on parameters or hierarchical references => elaboration-time checking is required for such cases. Note-2: bit widths of decimal constants are defined by their values; violation is issued only for decimal constants that are wider than RHS of an assignment (narrower decimal constants are allowed).	

EXAMPLE-1: [1] bit width of RHS of continuous assignment is less then LHS => violation

```
input  [3:0] in1;
output [3:0] out2;

assign out1 = in1[3:2];
```

Assignment source bit w idth "2" is less than destination bit w idth "2". Upper bits of the right-hand side w ill be filled w ith zeroes. Match bit w idths exactly to improve the readability of the description.

EXAMPLE-2: [1] bit width of RHS, described in true condition part of ternary operator, is less then LHS => violation

```
input  [3:0] in1;
input  [7:0] in2;
output [7:0] out;

assign out = ( in1 > in2 ) ? in1 : in2;
```

Assignment source bit w idth "4" is less than destination bit w idth "8". Upper bits of the right-hand side w ill be filled w ith zeroes. Match bit w idths exactly to improve the readability of the description.

EXAMPLE-3: [1] bit width of RHS is less then LHS, but pure function call is used => no violation (2.1.3.2)

```
wire  [32:0] a;
wire  [32:0] b;

    function [31:0] func;
        input integer a;
        ...
    endfunction

assign b = func( a );
```

STARC_VLOG 2.10.3.5

RULE NAME	**Specify base format (‘d,’b,’h,’o) for constants and keep them in mind (Verilog only)**
MESSAGE	**Base is not specified for “{ConstValue}”. It is recommended to specify base format ('d, 'b, 'h, 'o) for all the numeric values.**
PROBLEM DESCRIPTION	If constant base format is not clearly specified it may be difficult to see whether it is decimal, hexadecimal, octal or binary number. If the numeric value is between 1 and 7, there is no problem, as value does not change regardless of the numeral system. Yet, as a practice, it is recommended to specify base format (’d, ’b, ’h, ’o) for all the numeric values in Verilog HDL.
	LEVEL: RECOMMENDATION 2
CHECKER BEHAVIOR	1) Checker detects literal constants: – if base format ('d, 'b, 'o, or 'h) is not specified for constant = > violation 2) Checker skips following cases: – cases for checker 1.1.4.8 – comparison in for statement – index expressions – delay expressions

EXAMPLE-1: [1] constant without base specifier is a part of RHS of assignment => violation

```
assign D = ( A ^ 2'b11 ) / 2'16;
```

Base is not specified for “16”. It is recommended to specify base format ('d, 'b, 'h, 'o) for all the numeric values.

STARC_VLOG 2.10.3.6

RULE NAME	**Specify bit width for constants used in conditional expressions (Verilog only)**	
MESSAGE	**Bit width is not specified for the constant: {ObjectValue}.**	
PROBLEM DESCRIPTION	When arguments have different bit widths unexpected comparison results may be obtain. To avoid mistakes in conditional expressions using constants it is recommended to specify their bit widths explicitly.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects all conditional expressions: – conditional expression in if, case, for statements – case statement clauses – conditional expression in ternary operator ?: – arguments of comparison operators (<, <=, >=, >, ==, !=, ===, !==) Checker verifies constants in detected expressions – if bit width is not specified => violation Note: for if conditional expression if integer argument is compared with decimal constant width specifier is not required	

EXAMPLE-1: [1] constant without bit width specified is used in conditional expression of if statement=> violation

```
reg in1;

always @( in1 )
    if ( in1 == 'd1 )
        out1 = 1'b0;
    else
        out1 = 1'b1;
```

Bit w idth is not specified for the constant: 1.

EXAMPLE-2: [1] decimal constant without bit width specified is used in conditional expression of if statement, but compared argument has integer type => no violation

```
integer in1;

always @( in1 )
    if ( in1 == 'd1 )
        out1 = 1'b0;
    else
        out1 = 1'b1;
```

STARC_VLOG 2.10.3.7

RULE NAME	**Match the bit width of the value with the base number part (2'b) (Verilog only)**	
MESSAGE	**Bit width "{ConstantWidth}" of the value "{Value}" does not match the bit width "{BaseWidth}" of base number part. It is recommended to match bit widths clearly for all the constants and parameters unless there is a special reason not to do so.**	
PROBLEM DESCRIPTION	If the value specified in the constant bit width definition part is less than value bit width, upper bits of the constant are truncated and information is lost. If it is less – upper bits of constant are filled with zeros, it is not really a problem but still care should be taken. Therefore, when specifying a constant, it is better to match the value of the bit width definition part(2'b) and the bit width of value properly.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker scans literal constant values defined with base number part: – if constant bit width does not match to base number part => violation Length of each digit in the constant is defined by its base: – hexadecimal: 4 bits per digit – octal: 3 bits per digit Note:for decimal constants violation is issued only when specified constant has greater bit width than base number part	

EXAMPLE-1: [1] constant 'b10' bit width matches to base number part 2 => no violation;
[2] constant 'd10' bit width does not match to base number part 2 => violation;

```
tmp = (in1 ^ 2'b10) + 2'd10
```

Bit w idth “4” of the value “d10” does not match the bit w idth “2” of base number part. It is recommended to match bit w idths clearly for all the constants and parameters unless there is a special reason not to do so.

EXAMPLE-2: [1] constant 'b0' bit width matches to base number part 4 => violation;

```
assign out1 = {4'b0, in1}
```

Bit w idth “1” of the value “b0” does not match the bit w idth “4” of base number part. It is recommended to match bit w idths clearly for all the constants and parameters unless there is a special reason not to do so.

2.10.4 Take note of the different data types between the left and right sides (Verilog only)

STARC_VLOG 2.10.4.1

RULE NAME	**Do not use data types other than reg, wire and integer**	
MESSAGE	**Module "{ModuleName}" uses data types other than reg, wire and integer. It is not allowed in RTL description.**	
	DETAIL	*{ObjectClass} "{ObjectName}" has type: {TypeName}.*
PROBLEM DESCRIPTION	Synthesis of different data types required special libraries and conversion functions. Moreover, it is not supported by all synthesis tools. It is recommended to use only reg, wire and integer data types to avoid unexpected synthesis results.	
	LEVEL	RULE
CHECKER BEHAVIOR	1) Checker verifies signal and function declarations: – if type is other than reg, wire or integer => violation 2) Checker verifies parameters: – if parameter is not a vector (i.e. has real type) => violation Note: vectors of reg/wire, arrays of reg/wire/integer, memories (array of vector of reg) do not violate rule.	

EXAMPLE-1: [1] signal is declared with time type => violation
[2] parameter is declared with real type => violation

```
module top;

    time time_signal;
    parameter real param1 = 10.32;

    ...

endmodule
```

Module "top" uses data types other than reg, w ire and integer. It is not allow ed in RTL description.

Signal "time_signal" has type: time.

Parameter "param1" has type: real.

STARC_VLOG 2.10.4.3

RULE NAME	**Pay attention to bit widths when assigning an integer to reg or wire (Verilog only)**
MESSAGE-1	**Right hand side of the assignment is 32-bit integer that is less than bit width "{DestWidth}" of left hand side. Upper bits of the left hand side will be filled with zeroes. It is recommended to match bit widths exactly.**
MESSAGE-2	**Right hand side of the assignment is 32-bit integer that is greater than bit width "{DestWidth}" of left hand side. Upper bits of the right hand side will be truncated. It is recommended to match bit widths exactly.**
PROBLEM DESCRIPTION	Integers are 32-bit values. Close attention should be paid when assigning them to regs or wires. When bit width of right-hand-side it greater than bit width of assignment destination => upper bits of right-hand-side are truncated. Otherwise, when bit width of right-hand-side is less than bit width of assignment destination => upper bits of destination are filled with zeros. Descriptions with different bit widths may be made inadvertently – they are implicit and readability of the description drops. Concatenations/part-selections should be used to describe filling/truncation explicitly. LEVEL: RECOMMENDATION 1
CHECKER BEHAVIOR	Checker verifies each assignment ('=', '<=', 'assign') where: – right-hand-side expression contains: – signals of integer type – parameters of integer type – literal constants – assignment target is reg or wire Bit width of right hand side should be the same as bit width of assignment destination: – if bit width of right-hand-side expression is less than bit width of assignment target => violation (message-1) – if bit width of right-hand-side expression is greater than bit width of assignment target => violation (message-2) Note-1: this rule can be dependent on parameters or hierarchical references => elaboration-time checking is required for such cases

EXAMPLE-1: [1] right-hand-side is an expression with integer and literal constants members; [2] right-hand-side is less than assignment target => violation (message-1);

```
integer        SLOT_A;
integer        SLOT_B;
reg     [63:0] R_SUM;

always @( SLOT_A, SLOT_B )

     R_SUM = SLOT_A + SLOT_B + 1;
```

Right hand side of the assignment is 32-bit integer that is less than bit w idth "64" of left hand side. Upper bits of the left hand side w ill be filled w ith zeroes. It is recommended to match bit w idths exactly.

EXAMPLE-2: [1] right-hand-side is an expression with integer and literal constant members; [2] right-hand-side is wider than assignment target => violation (message-2);

```
integer        SLOT_A;
integer        SLOT_B;
reg     [15:0] R_SUM;

assign R_SUM = SLOT_A + SLOT_B;
```

Right hand side of the assignment is 32-bit integer that is greater than bit w idth "16" of left hand side. Upper bits of the right hand side w ill be truncated. It is recommended to match bit w idths exactly.

EXAMPLE-3: [1] right-hand-side is an expression with integer and literal constant members; [2] length of right-hand-side is equal to assignment target => no violation;

```
integer        SLOT_A;
integer        SLOT_B;
reg     [31:0] R_SUM;

assign R_SUM = SLOT_A + SLOT_B;
```

STARC_VLOG 2.10.4.5

RULE NAME	**Do not assign negative value to integer**	
MESSAGE	**A negative value is assigned to a variable of 'integer' type: {ExprValue}.**	
PROBLEM DESCRIPTION	Integers are defined as 32-bit values. Assigning negative numbers to integers is possible, but unexpected results may be obtained when assigning with the reg variable. For example, when assigning to variables larger than 32 bits, even if the sign of the top bit of the integer is negative, what is assigned to the upper bits depends on the tool used (it may become a positive value).	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies procedural (blocking (=) and non-blocking (<=)) assignment statements to the signals of an integer type: – if right-hand side of this assignment evaluates to negative value => violation	

EXAMPLE-1: [1] negative value is assigned to the integer signal => violation

```
integer int_var;

always @ ( ... )

    int_var = -2;
```

A negative value is assigned to a variable of 'integer' type: -2.

EXAMPLE-2: [1] expression result is negative
[2] target of assignment has integer type => violation
Note: the violation is detected at the elaboration stage

```
parameter integer p_int = -10;
parameter signed [31:0] p_sig = 5;

integer int_res;

always @ ( ... )

    int_res = p_int + p_sig;

     ...
```

A negative value is assigned to a variable of 'integer' type: -5.

STARC_VLOG 2.10.4.6

RULE NAME	**Do not assign a negative value to signals, which are declared with reg or wire**	
MESSAGE	**A negative value "{ExprValue}" is assigned to a variable of 'reg' or 'wire' type.**	
PROBLEM DESCRIPTION	Negative values can be assigned to reg variables. If assigning a negative value, it is regarded as the variable with a sign and value assigned. However, some logic synthesis tools may not generate a correct circuit, so use of negative values is not recommended.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects procedural and continuous assignments to signals of unsigned reg/wire type: – if right-hand side of this assignment evaluates to negative value => violation	

EXAMPLE-1: [1] negative integer value is assigned to the integer signal of the reg type => violation.

```
reg tmp;

always @ ( ... )

    tmp = -1;
```

A negative value "-1" is assigned to a variable of 'reg' or 'w ire' type.

EXAMPLE-2: [1] target of assignment is a signal of wire type;
[2] leftmost bit of signed constant is '1' => violation.

```
wire tmp;

assign tmp = 3'sb101;
```

A negative value "-3" is assigned to a variable of 'reg' or 'w ire' type.

EXAMPLE-3: [1] target of assignment is a signal of wire type;
[2] leftmost bit of signed constant is '0', because it is not specified implicitly => no violation.

```
wire tmp;

assign tmp = 4'sb101;
```

2.10.5 Do not share resources in speed critical circuits

STARC_VLOG 2.10.5.3

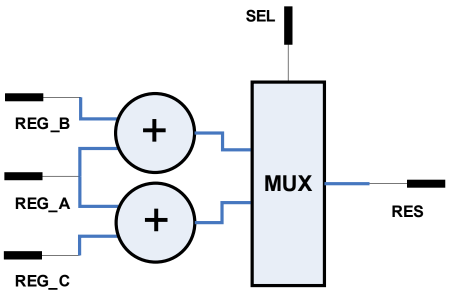
RULE NAME	**Do not describe three or more shared arithmetic operations**	
MESSAGE	**Arithmetic operation "{Operation}" is used in {BranchCount} conditional branches. If resource sharing is expected by a logic synthesis tool, circuit speed may degrade. Do not describe {OperationCount} or more shared arithmetic operations**	
	DETAIL	*Operator may be shared*

PROBLEM DESCRIPTION

Resources sharing uses the same operators for multiple operations to reduce the resources necessary for simultaneous operations. Following is description where operations are not shared:

```
if( SEL == 1'b0 )
   RES = REG_A + REG_B;
else
   RES = REG_A + REG_C;
```

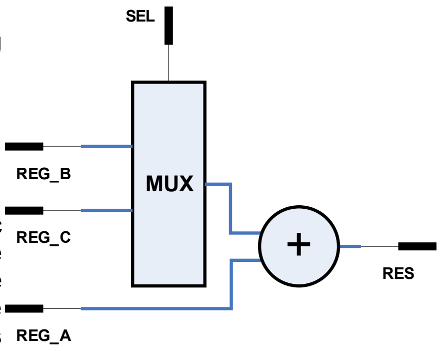
Structure at the right side corresponds to case when each branch operation is executed and operation result is selected by multiplexer.



Note, that conditional branches of the 'if' statement are not executed simultaneously. Consecutively, "+" operator can be shared (operands are placed on the input side of multiplexer and operators are shared by a single element).

Manually, such effect can be reached by following description:

```
if( SEL == 1'b0 )
   TMP = REG_B;
else
   TMP = REG_C;
RES = TMP + REG_A;
```



Of course, logic synthesis tools perform automatic resource sharing with consideration given to timing. The problem is following: resources cannot always be shared properly => it is unsafe for resource timing to be dependent on the sharing. In general, it is recommended not to share resources between more than three arithmetic operations (when automatic sharing is performed for many operators, circuit speed may degrade – a lot of logic is required for it). Moreover, if necessary structure is already known, it should be described clearly.

LEVEL	RECOMMENDATION 1

CHECKER BEHAVIOR

Checker scans branches in the following constructs:

- 'if' statement
- 'case' statement
- ternary operator (?:)

List of used arithmetic operations (+, -, *, /, **, %) is collected for each branch:

- process is recursive: while collecting the list for current branch, arithmetic operations from all nested branches are collected too

When arithmetic operations are collected for all branches, backward process started:

- for each arithmetic operation, all parallel branches with same operation are detected. Violation message is displayed when number of such branches is greater than value of the parameter SHARED_OPERATION_BRANCHES_ALLOWED (it can be changed from the configuration file):
  - parallel branch is such branch that not nested and belongs to the currently scanned construct

		– in case of violation, details are displayed for all (even nested) arithmetic operations (it is impossible to predict which from nested operators will participate in sharing – it depends on synthesis tool) – warning template: {OperationCount} = SHARED_OPERATION_BRANCHES_ALLOWED

EXAMPLE-1: [1] 'if' statement contains more than two arithmetic operators that may be shared => violation (default configuration allows two shared operations);

[2] note, that total count of operations is 4, but warning is issued only since 3 operations in the parallel branches (first branch contains nested 'if' statement with two operations, but they are issued as one operation within first branch);

```
if( SEL == 3'b111 ) then
     if( EN )
          RES <= REG_A * REG_B;
     else
          RES <= REG_A * REG_E;
else if( SEL == 3'b101 )
     RES <= REG_A * REG_C;
else if( SEL == 3'b001 )
     RES <= REG_B * REG_C;
```

Arithmetic operation is used in 3 conditional branches. If resource sharing is expected by a logic synthesis tool, circuit speed may degrade. Do not describe 3 or more shared arithmetic operations

Operator may be shared

Operator may be shared

Operator may be shared

EXAMPLE-2: [1] SHARED_OPERATION_BRANCHES_ALLOWED = 4;

[2] 'case' contains five operations that may be shared => violation;

```
case( SEL )
     3'b111: RES_1 = REG_A - REG_B;
     3'b101: RES_2 = REG_B - REG_C;
     3'b011: RES_1 = REG_C + REG_D;
     3'b001: RES_1 = REG_C - REG_D;
     3'b100: RES_2 = REG_C - REG_D;
     default: begin
          RES_1 = REG_A - REG_D;
          RES_2 = REG_B + REG_C;
     end
endcase
```

Arithmetic operation is used in 5 conditional branches. If resource sharing is expected by a logic synthesis tool, circuit speed may degrade. Do not describe 5 or more shared arithmetic operations

Operator may be shared

Operator may be shared

Operator may be shared

Operator may be shared

EXAMPLE-3: [1] ternary operator contains three operations possible to be shared => violation

```
assign RES = ( SEL == 3'b111 )?
                  REG_A % REG_B:
                  ( SEL == 3'b101 )?
                       REG_B % REG_D:
                       ( SEL == 3'b001 )?
                          REG_B + REG_D:
                          REG_X % REG_Z;
```

Arithmetic operation is used in 3 conditional branches. If resource sharing is expected by a logic synthesis tool, circuit speed may degrade. Do not describe 3 or more shared arithmetic operations

Operator may be shared

Operator may be shared

STARC_VLOG 2.10.5.5

RULE NAME	**Do not describe arithmetic operations with conditional operators (?) in an assignment statement (resource sharing will not be performed)**	
MESSAGE	**Arithmetic operator "{ArithOp}" is used in multiple conditional branches of a ternary operator in an 'assign' statement. Some tools do not perform resource sharing on 'assign' statements. Use 'always' statements with "if-else" descriptions if resource sharing is expected by a logic synthesis tool.**	
PROBLEM DESCRIPTION	Sharing arithmetic operator resources can be supported even when conditional operators are used within assign statements. However, there are tools that do not perform resource sharing on assign statements. Therefore, if resource sharing is expected by a logic synthesis tool, use always statements with if-else descriptions (see 2.10.5.3 for more details about resource sharing).	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects continuous assign statements with the ternary operator and scans the right-hand side for arithmetical operators (+, -, *, /, %, **): – if arithmetic operator is used in different branches of ternary operator => violation	

EXAMPLE-1: [1] division is used in both branches of ternary operator => violation

```
assign ARITH_1 = ( SEL )? A / B  - C : C / A + B;
```

Arithmetic operator "/" is used in multiple conditional branches of a ternary operator in an 'assign' statement. Some tools do not perform resource sharing on 'assign' statements. Use 'alw ays' statements w ith "if-else" descriptions if resource sharing is expected by a logic synthesis tool.

EXAMPLE-2: [1] division is used in both branches of ternary operator, but operator is specified in an always block => no violation

```
always ( SEL, A, B, C )
    ARITH_1 = ( SEL )? A / B  - C : C / A + B;
```

Arithmetic operator "/" is used in multiple conditional branches of a ternary operator in an 'assign' statement. Some tools do not perform resource sharing on 'assign' statements. Use 'alw ays' statements w ith "if-else" descriptions if resource sharing is expected by a logic synthesis tool.

2.10.6 Notes on arithmetic operations

STARC_VLOG 2.10.6.1

RULE NAME	**Carry-out should be considered for bit widths of signals to which operation results will be assigned.**	
MESSAGE	**Bit width "{DestBitWidth}" of assignment destination takes into account {CarryBits}-bit carry-out that is possible for assignment source. When it is necessary to consider the carry-out, it is recommended to describe carry-out logic separately from arithmetic expressions.**	

PROBLEM DESCRIPTION	In arithmetic operations, bit width should be specified correctly keeping carry-out output in mind. Usually it is not recommended to adjust target width considering possible carry-out. But to avoid date lost you should describe carry-out logic separate from an arithmetic operation as shown in the example:
	`reg a,b,c,res;`
	
	`assign res = a + b; //result of arithmetic operation`
	`assign c = a & b;   //carry-out signal`	

	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans assignment expressions which contains arithmetic operator ("+" or "-"): – bit-widths of the assignment source (S_BW) and destination (D_BW) are calculated, then – source bit width that considers carry overflow is also calculated (SO_BW) – if (SO_BW != S_BW), and – if (SO_BW == D_BW) => violation Note: assignments to be checked: – continuous ('assign'); – procedural (blocking/non-blocking); – conditional (?:).	

EXAMPLE-1: [1] arithmetic operation ("+") is detected;
[2] SO_BW = 2, S_BW = 1 => SO_BW != S_BW;
[3] D_BW = 2 => SO_BW == D_BW => violation.

```
reg a,b;
wire [1:0] res;

assign res = a + b;
```

Bit w idth "2" of assignment destination takes into account 1-bit carry-out that is possible for assignment source. When it is necessary to consider the carry-out, it is recommended to describe carry-out logic separately from arithmetic expressions.

EXAMPLE-2: [1] arithmetic operation ("+") is detected within expression;
[2] SO_BW = 3, S_BW = 2 => SO_BW != S_BW;
[3] D_BW = 3 => SO_BW == D_BW => violation.

```
reg a;
reg [1:0] b,c;
reg d,e,f;

wire res1;
wire [2:0] res2;

assign res2 = a * f | c & d + c;

assign res1 = a * b || c ^ d + e;   //result is always 1 bit => no violation
```

Bit w idth "3" of assignment destination takes into account 1-bit carry-out that is possible for assignment source. When it is necessary to consider the carry-out, it is recommended to describe carry-out logic separately from arithmetic expressions.

STARC_VLOG 2.10.6.2

RULE NAME	**Beware on the sign extension and adjust bit widths in signed operations**	
MESSAGE-1	**Signed expression width "{ExprWidth}" is {CompareResult} than bit width "{DestWidth}" of the assignment destination. It is necessary to match exactly the bit widths of signed operands and assignment destination.**	
	DETAIL	*Bit width of the argument is "{ArgBitWidth}"*
MESSAGE-2	**Signed expression arguments have different bit width. Match bit widths exactly when using signed operations.**	
	DETAIL	*Bit width of the argument is "{ArgBitWidth}"*
PROBLEM DESCRIPTION	In case of signed operations, bit width of assignment target should match to bit width of each argument. Sign bit should be expanded to the upper bits and bit widths should be made the same. Signed operations can introduce bit width mistakes. Descriptions with different bit widths may be made inadvertently – they are implicit and readability of the description drops.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker verifies each assignment ('=', '<=', 'assign') where right-hand-side is signed expression with "+" and "-" operators only: – bit widths of each operand should be the same and equal to bit width of assignment destination: – if bit width of whole right-hand-side expression is greater than bit width of assignment target => violation (message-1: {CompareResult} = "greater", detail per each argument with erroneous bit width) – if bit width of whole right-hand-side expression is less than bit width of assignment target => violation (message-1: {CompareResult} = "less", detail per each argument with erroneous bit width) – if bit width of whole right-hand-side expression is equal to bit width of assignment target => each argument of the expression is verified if it has less bit width than assignment destination (message-2 is displayed in case of violation with one detail per each erroneous argument) Note-1: this rule can be dependent on parameters or hierarchical references => elaboration-time checking is required for such cases Note-2: this checker verifies only cases where right-hand-side is a signed expression with "+" or "-" operators only. This is made intentionally, since checker 2.10.3.3 verifies cases where right-hand-side is expression of other type (for example, an unsigned expression or expression with another operators)	

EXAMPLE-1: [1] right-hand-side is an signed expression;
[2] only "+" and "-" operators are used;
[3] right-hand-side expression has greater bit width than assignment destination => violation (message-1, {CompareResult = greater});

```
input signed [7:0] P_1;
input signed [7:0] P_2
output       [7:0] OP_Z;
...

assign OP_Z = P_1 + P_2 - 1;
```

Signed expression w idth "32" is greater than bit w idth "8" of the assignment destination. It is necessary to match exactly the bit w idths of signed operands and assignment destination.

Bit w idth of the argument is "32"

EXAMPLE-2: [1] right-hand-side is an signed expression;
[2] only "-" operators are used;
[3] right-hand-side expression has same bit width as assignment destination, but one of the arguments has less bit width => violation (message-2);

```
input signed [7:0] P_1;
input signed [7:0] P_2
output        [7:0] OP_Z;

reg   signed [3:0] INTRNL;
...
assign OP_Z = P_1 - P_2 - INTRNL;
```

Signed expression arguments have different bit width. Match bit widths exactly when using signed operations.

Bit width of the argument is "4"

STARC_VLOG 2.10.6.3

RULE NAME	**Signed operations and unsigned operations should not be mixed in one statement**	
MESSAGE	**Pay extra attention when operating signed and unsigned signals. It is recommended to distinguish clearly between signed and unsigned arithmetic operations to avoid problems with bit width adjusting.**	
PROBLEM DESCRIPTION	When adjusting the bit widths of the right-hand side and left-hand side of an equation, a signed signal will extend the top bit (sign extension). With unsigned signals, it is not necessary to adjust the bit widths to be the same. When performing operations like this, it will not be possible to distinguish whether it is an operation of signed and unsigned signals, or it is an operation of signed signals with wrong bit widths. Therefore, an extra attention should be payed when operating signed and unsigned signals. And it is not recommended to use mixture of signed and unsigned operation in one statement.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker detects expressions which contain an arithmetic operations (+, -, *, /, %, **): – if operands are not all signed or all unsigned simultaneously => violation	

EXAMPLE-1: [1] arithmetic operation is used;
[2] first argument is of unsigned type;
[3] second argument is negative constant of integer type => violation.

```
reg arg;
reg dest;

always @( ... )

    dest = arg - 1;
```

Pay extra attention w hen operating signed and unsigned signals. It is recommended to distinguish clearly betw een signed and unsigned arithmetic operations to avoid problems w ith bit w idth adjusting.

EXAMPLE-2: [1] arithmetic operation is used within task enable statement;
[2] one of the arguments is of unsigned type, another – of signed => violation.

```
reg arg1;
reg signed arg2;

always @( ... )

    task1 ( arg1 * arg2, ... )
```

Pay extra attention w hen operating signed and unsigned signals. It is recommended to distinguish clearly betw een signed and unsigned arithmetic operations to avoid problems w ith bit w idth adjusting.

EXAMPLE-3: [1] arithmetic operation is used;
[2] first argument is of unsigned type but it is converted to signed by system function;
[3] second argument is of integer type, so it is signed => violation.

```
reg arg1;
integer arg2;
integer dest;

always @( ... )

    dest = $signed(arg1) - arg2;
```

STARC_VLOG 2.10.6.4

RULE NAME	**Do not infer large multipliers by the RTL description but describe the contents of multipliers by logical operation.**
MESSAGE	**Multiplier with {MultBW}-bit output is inferred. It is recommended to use libraries with high performance multipliers or describe them in a logical operation (when multiplier output is greater than {MAX_MULTIPLIER_WIDTH} bits).**
PROBLEM DESCRIPTION	Large scale multiplier can not be obtained from logic synthesis tools. For multiplier with greater than 16 bit output, purchase and use of libraries with high performance multipliers should be considered or multipliers should be described in a logical operation. Also, multipliers generated by a specific tool (such as Module Compiler) can be used. It is recommended to obtain multipliers by using external means or use gate level circuits.
	LEVEL: RECOMMENDATION 1
CHECKER BEHAVIOR	Checker detects multiplier inferences in the source code: – if bit width of operation result is greater then MAX_MULTIPLIER_WIDTH parameter value => violation Note-1: parameter MAX_MULTIPLIER_WIDTH defines maximum bit width which is allow for the result of multiplication operation (default value is “16”). Note-2: see 2.10.6.6 (note-1) for details about context. Note-3: elaboration-time checks are supposed for parameter-dependent expressions.

EXAMPLE-1: [1] two multiplication operations are detected;
[2] result of both operations is 32 bits => two violations.

```
reg [15:0] a,b,c,d;
reg [63:0] res;

assign res = a * b + c * d;
```

Multiplier w ith 32-bit output is inferred. It is recommended to use libraries w ith high performance multipliers or describe them in a logical operation (w hen multiplier output is greater than 16 bits).

Multiplier w ith 32-bit output is inferred. It is recommended to use libraries w ith high performance multipliers or describe them in a logical operation (w hen multiplier output is greater than 16 bits).

EXAMPLE-2: [1] multiplication operation is detected;
[2] bit width of first operand (sum) is 8 bits, result of multiplication is 16 bits in width => no violation.

```
reg [3:0] a,b
reg [8:0] c;
reg [63:0] res;

assign c = (a + b) * c;
```

STARC_VLOG 2.10.6.6 .v1

Note: This rule was actually removed from the Second Version (April 25, 2006) of STARC “RTL Design Style Guide for Verilog HDL”, and a new 2.10.8 section about dividers was added, because the latest implementation technologies and synthesis tools support "divider" elements. We decided to keep this rule renumbered to **2.10.6.6.v1**, because not everybody, especially FPGA designers, will have the possibility to synthesize division as "divider" element. Thus, current 2.10.6.6 rule is the former 2.10.6.7 that became 2.10.6.6 in the Second Version of the Guide. This is the only exception in STARC rules numbering.

RULE NAME	**Do not use division - Exception in the case of division by power of 2**
MESSAGE-1	**Divisor == {ObjectValue}. Do not use division, except in the case of division by the power of 2.**
MESSAGE-2	**Divisor expression is not constant. Do not use division, except in the case of division by the power of 2.**
MESSAGE-3	**Divisor == {ObjectValue}. Do not use modulus, except in the case of division by the power of 2.**
MESSAGE-4	**Divisor expression is not constant. Do not use modulus, except in the case of division by the power of 2.**
PROBLEM DESCRIPTION	Most logic synthesis tools do not support division (/) so do not use division, except in the case of division by the power of 2 (or you may use your own divider circuit implementation). In case of division for the power of 2, such as 2, 4, 8, 16, the operation is just a shift operation and therefore division can be used.
LEVEL	RULE
CHECKER BEHAVIOR	1) Checker detects expressions, where arithmetic division operation (/) is used: – if divisor value is not equal to the power of 2 (divisor is a literal constant, parameter or constant expression) => violation (message-1) – if divisor is a signal or an expression that contains signal => violation (message-2) 2) Checker detects expressions, where modulus operation (%) is used: – if divisor value is not equal to the power of 2 (divisor is a literal constant, parameter or constant expression) => violation (message-3) – if divisor is a signal or an expression that contains signal => violation (message-4) Note: expressions which cause circuit inference (where dividend is a signal) are checked

EXAMPLE-1: [1] arithmetic division operation is used;
[2] divisor value (divisor is a literal constant) is not equal to the power of 2 => violation (message-1)

```
assign out1 = in1 / 3;
```

Divisor == 3. Do not use division, except in the case of division by the pow er of 2.

EXAMPLE-2: [1] modulus operation is used;
[2] divisor is an expression that contains signal => violation (message-4)

```
assign out1 = in1 % (in2 + 2);
```

Divisor expression is not constant. Do not use modulus, except in the case of division by the pow er of 2.

STARC_VLOG 2.10.6.6

Note: This rule actually had the 2.10.6.7 number in the First Version (December 25, 2003) of STARC "RTL Design Style Guide for Verilog HDL". Former 2.10.6.6 rule was removed from the Second Version, and a new 2.10.8 section about dividers was added, because the latest implementation technologies and synthesis tools support "divider" elements. We decided to keep former 2.10.6.6. as 2.10.6.6.v1, because not everybody, especially FPGA designers, will have the possibility to synthesize division as "divider" element. Thus, former 2.10.6.7 rule became 2.10.6.6 in the Second Version of Guide and it is the current rule. The 2.10.6.6.v1 is the only exception in STARC_VLOG rules numbering.

RULE NAME	**Do not describe more than one arithmetic operation in one line (except for carry-in A+B+CIN(1bit))**
MESSAGE	**{ArithOperCount} arithmetic operators are described within the same expression. When more than one operation is performed in one expression, arithmetic operators will be synthesized in a linear fashion. In general, it is not recommended to describe more than one arithmetic operation in one line (except for carry-in A+B+CIN(1bit)).**

PROBLEM DESCRIPTION	Describing more then one arithmetic operation per single expression may cause 2 and more bits to carry around the end. Bit width on the two sides of an assignment is different in this case, and it is not be recommended. Also, when more than one operation is performed in one expression, arithmetic operators will be created in a linear fashion. To clarify the order of operations, this kind of operation should be described as one operation per statement.
	However, there is an exception to this recommendation. Consider an example below:
	`reg [1:0] a;`
	`reg [1:0] b;`
	`assign res = a + b + c;`
	In this case, when the third term is 1 bit, there is no problem because an adder with a carry-in input is generated.
	LEVEL — RECOMMENDATION 3

CHECKER BEHAVIOR	Checker scans expressions which contain arithmetic operator ("+" or "-"):
	– if more then one arithmetic operators is described:
	  – more than 2 operators => violation;
	  – exactly 2 operators:
	    – scan their operands for single-bit
	      – if it is detected => no violation;
	      – otherwise => issue violation.
	Note-1: following objects are considered as separate scopes and are checked separately:
	– concatenation element: { A + B, C + D };
	– function/procedure argument: function_call(A + B, C + D);
	– module instantiation statement FF INST_1(.PORT_A(A + B), .PORT_B(C + D));
	– ternary conditional branches: assign RES = (SEL)? A + B : C + D;
	– always sensitivity list element: always @(A + B, C + D);
	– comma-separated objects: assign A = B + C, E = F + G.
	Note-2: when expression contains operation other then "+" or "-" it is processed from bottom to top and each subexpression is checked separately:
	//operators checked first are marked with green
	//operators checked second are marked with blue
	`c * b * a + e * c - d * (a + b + d)`

EXAMPLE-1: [1] two arithmetic operations ("+") are detected within the first concatenation element;
[2] expression contains one-bit operand => no violation;
[3] one arithmetic operation ("-") is detected within second element => no violation;

```
reg [7:0] a,b,d,e;
reg c;

assign res = { a + b + c, d - e };
```

EXAMPLE-2: [1] two arithmetic operations ("+") are detected in every two branches of conditional operator;
[2] first branch does not contain one-bit operands => violation-1;
[3] second branch does not contain one-bit operands => violation-2
Note: separate violation per every branch.

```
reg [7:0] a,b,c,d,e,f;

assign sum = sel ? a + b + c : d + e + f;
```

2 arithmetic operators are described w ithin the same expression. When more than one operation is performed in one expression, arithmetic operators w ill be synthesized in a linear fashion. In general, it is not recommended to describe more than one arithmetic operation in one line (except for carry-in A+B+CIN(1bit)).

2 arithmetic operators are described w ithin the same expression. When more than one operation is performed in one expression, arithmetic operators w ill be synthesized in a linear fashion. In general, it is not recommended to describe more than one arithmetic operation in one line (except for carry-in A+B+CIN(1bit)).

EXAMPLE-3: [1] one arithmetic operation ("+") is detected within subexpression in parentheses => no violation;
[2] two operations ("+" and "-") are detected within expression on top level and it does not contain one-bit operands => violation.

```
reg [7:0] a,b,c,d,e,f;

assign res = a * b * (c + d) / 2 - e + f;
```

2 arithmetic operators are described w ithin the same expression. When more than one operation is performed in one expression, arithmetic operators w ill be synthesized in a linear fashion. In general, it is not recommended to describe more than one arithmetic operation in one line (except for carry-in A+B+CIN(1bit)).

2.10.7 Take share items out of conditional branches

STARC_VLOG 2.10.7.1

RULE NAME	**Do not use arithmetic operation in the conditional expression of if statements**
MESSAGE-1	**An arithmetic operation is detected in the conditional expression of an 'if' statement. Prefer using intermediate variables to share resources**
MESSAGE-2	**An arithmetic operation is detected in the conditional expression of an ternary operator. Prefer using intermediate variables to share resources**

PROBLEM DESCRIPTION	Arithmetic operations within conditional expressions of 'if' statements or ternary operators are not participating in the automatic resource sharing:
	`always @( INP_A or INP_B or INP_C )`
	`   if( INP_A - INP_B > 4'b0101 )`
	`       RES = 4'b0000;`
	`   else if( INP_A - INP_B - INP_C > 4'b1000 )`
	`       RES = 4'b0110;`
	`   else`
	`       RES = 4'b1111;`
	`end`
	Intermediate variables should be used in such cases (it is recommended to use 'assign' statement):
	`assign TMP =  INP_A - INP_B;`
	`always @( INP_A or INP_B or INP_C )`
	`   if( TMP > 4'b0101 )`
	`       RES = 4'b0000;`
	`   else if( TMP - INP_C > 4'b1000 )`
	`       RES = 4'b0110;`
	`   else`
	`       RES = 4'b1111;`
	`end`

LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans conditional expressions of 'if' statements and ternary operators: – if at least one arithmetic operation detected => violation: – for 'if' statements => message-1 – for ternary operators => message-2 – index expressions (loop variable and constant) are not scanned

EXAMPLE-1: [1] arithmetic operation is used in the conditional expression of 'if' statement => violation

```
always @( A or B or C )
    if( A + B > C )
        RES = 1'b1;
    else
        RES = 1'b0;
```

An arithmetic operation is detected in the conditional expression of an 'if' statement. Prefer using intermediate variables to share resources

EXAMPLE-2: [1] arithmetic operation is used within index expression => no violation (only loop variable and constant are used)

```
for( i = 0; i < 31; i = i + 1 )
    if( REG_A[i+1] > REG_A[i]) begin
        TMP = REG_A[i+1];
        REG_A[i+1] = REG_A[i];
        REG_A[i] = TMP;
    end
```

2.10.8 Division descriptions

STARC_VLOG 2.10.8.1

RULE NAME	**Don't use arithmetic and logical expressions at the right and left sides of the division or modulus operator**
MESSAGE	**Do not use arithmetic and logical expressions at the right and left sides of the division or modulus operator.**
PROBLEM DESCRIPTION	Divider and modulus logic is even slower than multipliers; so care should be taken about the bit width of a divider. If there is a shift operation at the right or left side of the division or modulus operator, the logic synthesis tool can not determine the bit width. Consequently, a 32-bit divider might be inferred. In order to ensure a deterministic divider width, arithmetic and relational operators should not be used at the right and left sides of the division and modulus operators. Use of logical expressions cause the divider width to be deterministic. But divider and modulus logic is complicated and slow. So you should code divider and modulus logic separately from logical operations. It should be taken into account that some logic synthesis tools generate an error if the bit width at the right side of an operator is greater than that of the left side. There are logic synthesis tools that support resource sharing between division and modulus logic. However, resource sharing is not achieved unless the bit widths of the right and left sides of an operator and the variable to which the result is assigned are identical.
LEVEL	RULE

CHECKER BEHAVIOR	1) Checker scans division (/) and modulus (%) operators in a synthesis context:
	– context for search
	  – always processes
	  – assign (continuous, including assignment in net declaration)
	  – synthesize task/function
	  – task/function call arguments
	  – instance port map
	  – index of bit-selection
	– if arguments (both left and right side) contains any of operators mentioned below => violation
	  – forbidden operations:
	    – arithmetic operations (+, -, *, /, %, **, <<, >>, <<<, >>>)
	    – logical operations (!, &&, ||, ==, !=, ===, !==)
	    – relational operations (<, <=, >, >=)
	Note: following constructs are not checked:
	  – initial processes
	  – parameter definitions
	  – parameter redefinition (generic map, defparam)
	  – initialization assignment to reg
	  – conditional expression of for statements
	  – index of part-selection
	  – index of signal declaration
	  – delay and event control statement expression

EXAMPLE-1: [1] arithmetic operation is used in the left side of division operation => violation

```
assign res = ( a + b ) / c;
```

Do not use arithmetic and logical expressions at the right and left sides of the division or modulus operator.

EXAMPLE-2: [1] logic operation is used in the right sides of division and modulus operations in port map of gate instantiation => violation

```
and (a, b / ( c == d ), b % ( c == d ));
```

Do not use arithmetic and logical expressions at the right and left sides of the division or modulus operator.

STARC_VLOG 2.10.8.2

RULE NAME	**Keep the left side of the division or modulus operator within 12 bits**	
MESSAGE	**Left side of the division or modulus operator is {LeftArgWidth} bits wide. Keep the left side of the division or modulus operator within {LEFT_ARGUMENT_WIDTH_MAX} bits.**	
PROBLEM DESCRIPTION	Divider and modulus logic is complicated and slow. Circuit size and complexity increases with bit width of left side argument. So it is recommended to keep the left side of division and modulus operator within 12 bits.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans division (/) and modulus (%) operators in a synthesis context: – context for search (see 2.10.8.1) – if the left argument is greater than LEFT_ARGUMENT_WIDTH_MAX parameter => violation Note-1: LEFT_ARGUMENT_WIDTH_MAX parameter (maximum bit width of the dividend) is defined in the rule configuration (default is 12) Note-2: if argument's width cannot be defined at the compilation stage – it is checked at the elaboration stage	

EXAMPLE-1: [1] the left argument of division operator is greater than LEFT_ARGUMENT_WIDTH_MAX parameter (default 12) => violation

```
reg [15:0] a;
reg [7:0] b;
reg [7:0] c;

assign c = a / b;
```

Left side of the division or modulus operator is 16 bits w ide. Keep the left side of the division or modulus operator w ithin 12 bits.

STARC_VLOG 2.10.8.3

RULE NAME	**Keep the right side of the division or modulus operator within 8 bits (except powers of 2)**	
MESSAGE	**Right side of the division or modulus operator is {RightArgWidth} bits wide. Keep the right side of the division or modulus operator within {RIGHT_ARGUMENT_WIDTH_MAX} bits.**	
PROBLEM DESCRIPTION	It should be taken into account that the area and delay of a divider increase more rapidly with an increase in the bit width of the right side of an operator than that of the left side. So keep the right side of the division or modulus operator within 8 bits. When the right side of the division or modulus operator is a power of two, a shifter is inferred by logic synthesis tools. In this case, there is no limit to the bit width.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans division (/) and modulus (%) operators in a synthesis context: – context for search (see 2.10.8.1) – if the right argument is greater than RIGHT_ARGUMENT_WIDTH_MAX parameter => violation – if the right argument is a power of 2 its width is not checked Note-1: RIGHT_ARGUMENT_WIDTH_MAX parameter (maximum bit width of the divisor) is defined in the rule configuration (default is 8) Note-2: if the argument width cannot be defined at the compilation stage – it is checked at the elaboration stage	

EXAMPLE-1: [1] right argument is greater than RIGHT_ARGUMENT_WIDTH_MAX parameter (changed to 4 in rule configuration) => violation

```
reg [7:0] a;
reg [7:0] b;
reg [7:0] c;

assign c = a % b;
```

Right side of the division or modulus operator is 8 bits w ide. Keep the right side of the division or modulus operator w ithin 4 bits.

EXAMPLE-2: [1] right argument is greater than RIGHT_ARGUMENT_WIDTH_MAX parameter (default 8) but it is parameter with value equal to power of 2 => no violation

Note: elaboration phase is required to judge it is no violation

```
reg [7:0] a;
reg [7:0] c;
parameter [32:0] b = 32;

assign c = a % b;
```

STARC_VLOG 2.10.8.4

RULE NAME	**Minimize the bit widths at the right and left sides of the division and modulus operators**	
MESSAGE	**Minimize the bit widths at the right and left sides of the division and modulus operators. Recommended argument bit widths: left = {LEFT_ARGUMENT_WIDTH}, right = {RIGHT_ARGUMENT_WIDTH}.**	
PROBLEM DESCRIPTION	Divider does not operate at a high frequency even when the left side of the division operator is 12 bits and the right side is 8 bits. So the bit widths of a divider should be minimized even when they are less than it is set in rules 2.10.8.2 and 2.10.8.3. If the division and modulus do not run at a desired frequency, you need to break it into two or three stages (using pipelined units) or use some function library of an arithmetic operation.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans division (/) and modulus (%) operators in a synthesizable context: – context for search (see 2.10.8.1) – if left argument is greater than LEFT_ARGUMENT_WIDTH parameter => violation – if right argument is greater than RIGHT_ARGUMENT_WIDTH parameter => violation – if right argument is a power of 2 its width is not checked Note-1: LEFT_ARGUMENT_WIDTH and RIGHT_ARGUMENT_WIDTH parameters (maximum bit width of the dividend and divisor) are defined in the rule configuration (default are 12 and 8 respectively) Note-2: if argument's width can not be defined at the compilation stage – it is checked at the elaboration stage	

EXAMPLE-1: [1] left argument is greater than LEFT_ARGUMENT_WIDTH parameter (default 12) and right argument is greater than RIGHT_ARGUMENT_WIDTH (default 8) parameter => violation

```
reg [15:0] a;
reg [15:0] b;
reg [15:0] c;

assign c = a % b;
```

Minimize the bit w idths at the right and left sides of the division and modulus operators. Recommended argument bit w idths: left = 12, right = 8.

EXAMPLE-2: [1] left argument is greater than LEFT_ARGUMENT_WIDTH parameter (default 12) => violation

```
reg [15:0] a;
reg [3:0] b;
reg [15:0] c;

assign c = a % b;
```

Minimize the bit w idths at the right and left sides of the division and modulus operators. Recommended argument bit w idths: left = 12, right = 8.

Chapter 3 RTL Design Methodology

3.1 Create function libraries

3.1.3 Standardize description order of module I/O ports

STARC_VLOG 3.1.3.2

RULE NAME	**The port description order should be clock, reset, input, output, inout**	
MESSAGE-1	**Module "{ModuleName}" has not desirable order of port declaration. Following port description order is recommended: {DESIRED_PORT_ORDER}.**	
	DETAIL	*Port "{PortName}" is {PortType}.*
MESSAGE-2 [INFO]	**Rule configuration parameter "DESIRED_PORT_ORDER" has improper value. Default value will be used. See help for details.**	
PROBLEM DESCRIPTION	Defining the port description order according to convention makes it possible to reduce errors when calling the function library from an upper level. The port description should be in order of basic control signals such as clock, reset, and enable, then input, output and inout. There is no particular description order within input, output and inout, but collecting signals according to application whenever possible will contribute to error reduction. It is recommended to describe I/O ports for the module instantiation in the same order as its module declaration.	
	LEVEL	RECOMMENDATION 2

CHECKER BEHAVIOR	Checker scans port list:
	– port declaration order should be the following:
	  – clock signals (FF);
	  – controls:
	    – reset/set signals (asynchronous, synchronous) (FF, latch);
	    – enable signals (FF, latch, tri-state);
	  – other signals of 'input' mode;
	  – signals of 'output' mode;
	  – signals of 'inout' mode;
	– if described order is not hold => violation
	Note-1: port order is described by parameter DESIRED_PORT_ORDER and can be configure (default value is {"clock", "reset", "enable", "input", "output", "inout"}). Only noticed keywords may be used to describe port order and each word may be specified only once (or may be omitted).
	Note-2: if input port is detected with more then one type (clock, set, reset or enable) it should be treated as type which is the earliest in the list.
	Note-3: elaboration-time checks are supposed when port type can not be detected at compilation stage.	

EXAMPLE-1: [1] port order does not corresponds to parameter DESIRED_PORT_ORDER value (port of type 'input' is described at the very beginning) => violation.

Note: there is no any port of type 'inout', but it does not have any affect (such situation is correct).

```
module ff_rst_syncset (
        d,
        clk,
        rst_x,
        set,
        en,
        q);
```

Module “ff_rst_syncset” has not desirable order of port declaration. Follow ing port description order is recommended: {“clock”, “reset”, “enable”, “input”, “output”, “inout”}.

Port “d” is input.

Port “clk” is clock.

Port “rst_x” is reset.

Port “set” is set.

Port “en” is enable.

Port “q” is output.

```
        input clk,rst_x,set,en,d;
        output reg q;

       always @( posedge clk, negedge rst_x )
           begin
               if ( !rst_x )
                   q <= 1'b0;
               else if ( set )
                   q <= 1'b1;
               else if ( en )
                   q <= d;
           end

endmodule
```

EXAMPLE-2: [1] port 'c' is used as enable and clock signal, but clock is earlier in the list so port 'c' is treated as clock;

[2] port order does not corresponds to parameter DESIRED_PORT_ORDER value ({ “reset”, “clock”, “enable”, “input”, “output”, “inout”}) => violation.

Note: no port of 'reset' type, but it does not have any affect (such situation is correct).

```
module top (
        c,
        set,
        d1,
        d2,
        d3,
        q1,
        q2);
```

Module “top” has not desirable order of port declaration. Follow ing port description order is recommended: { “reset”, “clock”, “enable”, “input”, “output”, “inout”}.

Port “c” is clock.

Port “set” is set.

Port “d1” is input.

Port “d2” is input.

Port “d3” is input.

Port “q1” is output.

Port “q2” is output.

```
        input c,set,d1,d2,d3;
        output q1;
        output reg q2;

        assign q1 = c ? d1 : q1;

        assign and_res = d2 & d3;

        always @( posedge c, posedge set)
```

```
         if (set)
              q2 <= 1'b1;
         else
              q2 <= and_res;
endmodule
```

EXAMPLE-3: [1] port order corresponds to parameter DESIRED_PORT_ORDER value ({“clock”, “reset”, “enable”, “input”, “output”, “inout”}) => no violation.

Note: ports of some types are skipped, but it does not have any affect (such situation is correct).

```
module ff( clk, d, q);

     input clk,d;
     output reg q;

    always @( posedge clk )
        q <= d;

endmodule
```

3.1.4 Consider RTL description readability

STARC_VLOG 3.1.4.4

RULE NAME	**Do not describe multiple assignments in one line**	
MESSAGE	**Multiple statements are described in the single line. Describe one statement per line to improve RTL description readability.**	
PROBLEM DESCRIPTION	Describing multiple assignments per line is not recommended from standpoint of readability (especially comma-separated assignments). But inserting more then one statement in the same line also makes description difficult to read and to understand. Therefore, it is better to place only one statement per a single line.	
	LEVEL	RECOMMENDATION 3

CHECKER BEHAVIOR	Checker scans Verilog constructs description:
	– if there are one or more statement or declaration in the single line with another statement:
	– if there are only hierarchically dependent statements (in descending order) => no violations
	– *statements which can hierarchically include nested statements:*
	– *conditional statement (if-else);*
	– *case statement (case, casex, casez);*
	– *looping statements (forever, repeat, while, for);*
	– *procedural timing controls (#, @);*
	– *block statements (begin-end, fork-join);*
	– *structured procedures (initial construct, **always** construct, task, function)*
	– otherwise => violation
	`always @( posedge CLK ) begin  // hierarchically-dependent statements`
	`    Q1 <= DATA_1; Q2 <= DATA_2;// hierarchically-independent`
	`                               // statements`
	`end`
	– **exception:** net/variable declaration assignment does not violate the rule
	`reg a = 1'b1;`	

EXAMPLE-1: [1] hierarchically dependent statements exists;
[2] assignment in 'if' branch is one level lower then 'else' branch (ascending order) => violation.

```
if ( en ) q = d; else q = 1'bz;
```

Multiple statements are described in the single line. Describe one statement per line to improve RTL description readability.

EXAMPLE-2: [1] hierarchically dependent statements exists;
[2] dependency only in descending order => no violation.

```
always @( posedge clk ) q <= #10 d;
```

EXAMPLE-3: [1] multiple declarations per line => no violation;
[2] multiple assignments => violation.

```
reg a,b;

assign a = in1 | in2, b = in1 & in2;
```

Multiple statements are described in the single line. Describe one statement per line to improve RTL description readability.

STARC_VLOG 3.1.4.5

RULE NAME	**The maximum number of characters in one line should be about 110**	
MESSAGE	**Source file contains line entries that exceed the recommended length of {MAX_NUM_OF_CHARACTERS} characters per line.**	
	DETAIL	*Number of characters in this line is "{CharactersCount}".*
PROBLEM DESCRIPTION	The number of characters per line should provide the line to be completely displayed. Such style increases description readability. If the number of lines is too great, wrapping occurs and readability decreases. Therefore, it is better to limit the number of characters in per line to about 110.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans source code in current file for lines which have number of characters greater than parameter MAX_NUM_OF_CHARACTERS: – when first of such lines is detected => violation Note-1: value of MAX_NUM_OF_CHARACTERS parameter is specified in configuration file (default value is 110). Note-2: one warning message is issued per file.	

EXAMPLE-1: [1] there is a line which length is greater than MAX_NUM_OF_CHARACTERS parameter in the file => violation.

Note: MAX_NUM_OF_CHARACTERS parameter value is changed to 50 to simplify the example.

Source file contains line entries that exceed the recommended length of 50 characters per line.

```
assign res = ctrl ? data1 : data2; //very long and important comment
```

Number of characters in this line is "68".

3.2 *Using function libraries*

3.2.2 Define global parameters in separate files (different from VHDL)

STARC_VLOG 3.2.2.4

RULE NAME	**File names specified by `include should be made into relative paths (../include/common.h)**	
MESSAGE-1	**File "{FileName}" is detected in the current directory. Inclusion by simple file name is necessary when included file exists in the directory specified by +incdir +<directory_name>. Otherwise, files specified by `include directive should return to the directory that is one level higher (../one_level_higher/file_to_include). Executions of simulation and logic synthesis are usually performed in separate directories and such style allows to distinguish files that are necessary for particular stage.**	
MESSAGE-2	**Files specified by `include directive should return to the directory that is one level higher (../one_level_higher/file_to_include). Executions of simulation and logic synthesis are usually performed in separate directories files that are necessary for particular stage.**	
PROBLEM DESCRIPTION	Design data require many files in addition to RTL and test bench files. If these files are all saved in the same directory, it will become impossible to distinguish between necessary and unnecessary files. Executions of simulation and logic synthesis are done in separate directories, that is why it is unsafe if simply using <file_name.h> since another file may be invoked. For the file specified by `include directive, its path should be described as a relative path, which returns to the directory that is 1 level higher. In a large scale design, RTL descriptions are stored in predefined locations. With this type of design, there is a limit in specifying the file using a relative path. In this case, options of each tool can be used to specify the file name. Specifying location of including files by +incdir+<directory name> is allowed by most Verilog simulators.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker detects file inclusion compiler directive and scans filename specified: – if name (absolute/relative path) does not return to the directory that is one level higher => violation (message-2) – if filename is simple name (does not have absolute/relative path specified) and file specified by this name exists in the current directory => violation (message-1)	

EXAMPLE-1: [1] simple file name is specified within 'include directive;
[2] file with specified name exists in current directory => violation (message-1).

```
`include "definitions.h"
```

File "definitions.h" is detected in the current directory. Inclusion by simple file name is necessary w hen included file exists in the directory specified by +incdir+<directory_name>. Otherw ise, files specified by `include directive should return to the directory that is one level higher (../one_level_higher/file_to_include). Executions of simulation and logic synthesis are usually performed in separate directories and such style allow s to distinguish files that are necessary for particular stage.

EXAMPLE-2: [1] file name within 'include directive is specified by relative path;
[2] relative path does not return to the directory that is one level higher => violation (message-2).

```
`include "../definitions.h"
```

Files specified by `include directive should return to the directory that is one level higher (../one_level_higher/file_to_include). Executions of simulation and logic synthesis are usually performed in separate directories files that are necessary for particular stage.

STARC_VLOG 3.2.2.5

RULE NAME	**Do not nest text macros**	
MESSAGE	**Nested definition is used for "{MacroName}" text macro. Do not nest text macros, because such description is difficult to read and error-prone.**	
	DETAIL	*Declaration of nested macro "{NestedMacroName}".*
PROBLEM DESCRIPTION	`define definitions are subject to text replacement, and syntax checks are not performed. If nested `define definitions is used (especially in case of more then one nesting level), it is difficult to keep in mind the final result of text substituting and errors may easily occur. It is extremely risky to allow nested `define definitions without some sort of constraint, therefor it is recommended not to nest them.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects text macro definitions (definitions made with `define directive): – if defined macro text contains any other text macros => violation	

EXAMPLE-1: [1] text macro definition contains usage of another macros => violation.

```
`define delay1 10
`define delay2 20
`define delay3 `delay1 + `delay2;
```

Declaration of nested macro "delay1".

Declaration of nested macro "delay1".

Nested definition is used for "{MacroName}" text macro. Do not nest text macros, because such description is difficult to read and error-prone.

3.2.3 *Connect ports by name* for component instantiations

STARC_VLOG 3.2.3.1

RULE NAME	**For component instantiations, connect ports by name connections, not by ordered list**	
MESSAGE	**Ordered port connections are used for component instantiation. Prefer named port connections to avoid port position mistakes.**	
PROBLEM DESCRIPTION	The connection of ports by name clearly describes the correlation between the component port and the connected net name. Such description is easier to understand since it is possible to match upper net names with the port names. The second is the connection of ports by ordered list that describes the net in the port description order of the lower components. It is easer to describe, but incorrect connections may result. So it is recommended to connect ports by names.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker verifies component instantiations: – if order port connection is used => violation	

EXAMPLE-1: [1] ordered port connection is used => violation

```
submod inst1 ( in1, in2, out1 );
```

Ordered port connections are used for component instantiation.
Prefer named port connections to avoid port position mistakes.

STARC_VLOG 3.2.3.2

RULE NAME	**Match the bit width of the component port and the bit width of the net to be connected**	
MESSAGE	**Bit width of {PortCount} lower port(s) does not match the bit width of the connected net(s) in the instantiation of component "{CompName}". Match the bit widths exactly.**	
	DETAIL-1	*Bit width of the net "{NetName}" is "{NetWidth}" while bit width of the port "{PortName}" is "{PortWidth}"*
	DETAIL-2	*Bit width of the connection is "{NetWidth}" while bit width of the port "{PortName}" is "{PortWidth}"*
PROBLEM DESCRIPTION	When bit width of connected net it greater than bit width of component port => upper bits of the net are truncated. Otherwise, when bit width of the net is less than bit width of component port => upper bits of port are filled with zeros. Data can be misaligned or lost. Descriptions with different bit widths may be made inadvertently – they are implicit and readability of the description drops. Concatenations/part-selections should be used to describe filling/truncation explicitly.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans component instantiations (either ordered or named): – bit width of each port should match to bit width of the connected net – if bit widths doesn't match => violation is issued and details are displayed due to following conditions: – detail-1: for simple connections (name of the net can be determined) – detail-2: for complex connections (name of the net cannot be defined: for example, when connection is represented by concatenation, hierarchical reference, constant, etc.) Note-1: this rule can be dependent on elaboration-time references => elaboration-time checking will be performed for such cases Note-2: instantiations of language built-in components (gates and switches) is not verified by this checker Note-3: unconnected ports are skipped by checker (special Design Compiler command exists to eliminate unconnected ports)	

EXAMPLE-1: [1] ordered connection is described;

[2] bit width of component port is less than bit width of simple net connected to it => violation (detail-1);

```
module and3( I1, I2, I3, O );
     input  [7:0] I1, I2, I3;
     output [7:0] O;
     ...
endmodule

module top( ..., R_OUT, ... );
     output [7:0] R_OUT;
     ...
     wire   [4:0] ARG1;
     wire   [7:0] ARG2;
     wire   [7:0] ARG3;
     ...
     and3 INST_000( ARG1, ARG2, ARG3, R_OUT );
     ...
endmodule
```

Bit w idth of 1 low er port(s) does not match to bit w idth of the connected net(s) in the instantiation of component "and3". Match the bit w idths exactly.

Bit w idth of the net "ARG1" is "5" w hile bit w idth of the port "I1" is "8"

EXAMPLE-2: [1] named connection is described;

[2] bit width of one component port is greater than bit width of simple net connected to it => violation (detail-1);

[3] bit width of another one component port is greater than bit width of complex net connected to it => violation (detail-2);

```
module and3( I1, I2, I3, O );
     input  [7:0] I1, I2, I3;
     output [7:0] O;
     ...
endmodule

module top( ..., R_OUT, ... );
     output [7:0] R_OUT;
     ...
     wire   [4:0] ARG1_a;
     wire   [4:0] ARG1_b;
     wire   [8:0] ARG2;
     wire   [7:0] ARG3;
     ...
     and3 INST_000( .I1( { ARG1_a, ARG1_b } ),
                    .I2( ARG2 ), .I3( ARG3 ), .O( R_OUT ) );
     ...
endmodule
```

Bit w idth of 2 low er port(s) does not match to bit w idth of the connected net(s) in the instantiation of component "and3". Match the bit w idths exactly.

Bit w idth of the connection is "10" w hile bit w idth of the port "I1" is "8"

Bit w idth of the net "ARG2" is "9" w hile bit w idth of the port "I1" is "8"

3.2.4 Use # (value) when overwriting parameters from an upper level (different from VHDL)

STARC_VLOG 3.2.4.3

RULE NAME	**Do not use defparam statements**	
MESSAGE	**Do not use 'defparam' statements. Some logic synthesis tools do not support 'defparam'.**	
PROBLEM DESCRIPTION	defparam can be used for rewriting parameters by hierarchical parameter names. If a defparam assignment conflicts with a module instance parameter, the parameter in the module takes the value specified by the defparam. While defparam has such benefit of replacing the parameter values, it should not be used because some logic synthesis tools do not support defparam.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans Verilog description: – if defparam statement is present => violation	

EXAMPLE-1: [1] defparam statement is used => violation

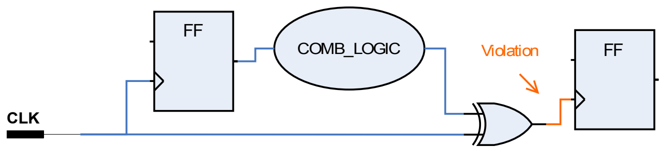
```
module my_module( a, b, c );
    parameter [1:0] param1 = 2'b00;
    defparam top.param1 = 2'b11;

    ...

endmodule

module top( in1, in2, out1 );

    ...

    parameter [1:0] param1 = 2'b00;
    my_module inst ( .a(in1), .b(in2), .c(out1) );
endmodule
```

Do not use 'defparam' statements. Some logic synthesis tools do not support 'defparam'.

3.3 Design for Test (DFT)

3.3.1 Clocks and resets for DFT

STARC_VLOG 3.3.1.1

RULE NAME	**The clocks must be able to be controlled directly from external input ports**
MESSAGE	**Clock input of FF "{FFName}" is not directly controlled from external input port. The clocks must be able to be controlled directly from external input ports, otherwise scan insert tool will exclude FF from the scan.**
PROBLEM DESCRIPTION	When DFT scan chains are inserted, the most important is to consider structure of the circuit in such way, that will enable safe scan shift during the scan test. When clock pin of flip-flop cannot be directly controlled from an external input port, scan chain insertion tool excludes such flip-flop from the scan chain. Such exclusion disables faults detection by ATPG tools for any parts for which scans have not been inserted.  When it is impossible to control the clock signal from an external port, switching circuitry should be used to enable direct control in the test mode. LEVEL: RULE
CHECKER BEHAVIOR	Checker scans the design hierarchy for flip-flops and verifies the signal that is mapped to the clock pin of each flip-flop: – if there is a multiplexer that is directly(*) connected to the clock pin of an FF => no violation (case for 3.3.1.3) (*) *directly or through buffers/inverters* – if there is a situation that is **not covered** by following rules => violation: – OR with two inputs is connected to the clock pin (3.3.5.3, 3.3.5.6); – AND with two inputs is connected to the clock pin (3.3.5.4, 3.3.5.5); – latch output is connected to the clock pin of FF (3.3.5.7).

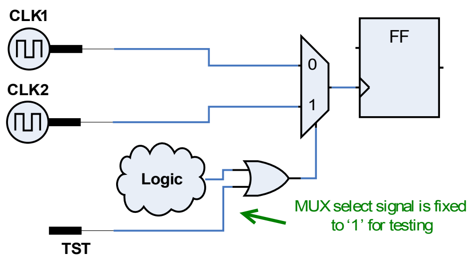
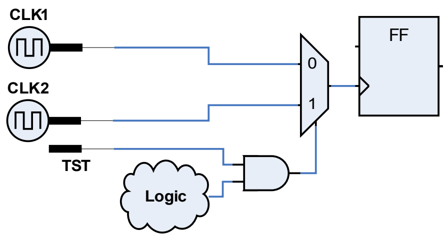
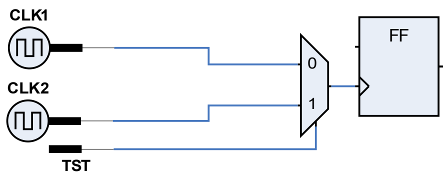
EXAMPLE-1: [1] module 'dff' which infers flip-flop with a clock signal controlled from an external input;
[2] module instantiation statements create two named instances:
- the external signal is mapped to the clock signal in instance 'u_dff_1';
- the output of combinational logic is mapped to the clock signal in instance 'u_dff_2' => violation.

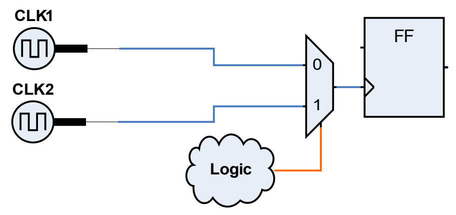
```
module dff( clk, data, q );

    input clk, data;
    output q;

    reg q;

    always @( posedge clk )
        q <= data;

endmodule
```

Clock input of FF "u_dff_2.out" is not directly controlled from external input port. The clocks must be able to be controlled directly from external input ports, otherw ise scan insert tool w ill exclude FF from the scan.

```
module top( clk, gate, data1, data2, out1, out2 );

    input clk, data1, data2;
    output out1, out2;

    wire clk_gate;

    assign clk_gate = clk ^ gate;

    dff u_dff_1 ( .clk( clk ), .data( data1 ), .q( out1 ) );

    dff u_dff_2 ( .clk( clk_gate ), .data( data2 ), .q( out2 ) );

endmodule
```

STARC_VLOG 3.3.1.2

RULE NAME	**When there is a choice between two different clock systems, one clock system must be selected throughout testing**	
MESSAGE	**Selection between clock systems is not controllable from an external port. The single clock system must be selected throughout testing otherwise it is impossible to insert scan.**	
	DETAIL-1	*Alternative clock system: “{HierClockName}”.*
	DETAIL-2	*Alternative clock system.*
PROBLEM DESCRIPTION	When DFT scan chains are inserted, the most important is to consider structure of the circuit in such way, that will enable safe scan shift during the scan test. When clock pin of flip-flop cannot be directly controlled from an external input port, scan chain insertion tool excludes such flip-flop from the scan chain. Such exclusion disables faults detection by ATPG tools for any parts for which scans have not been inserted. But even if the clocks can be controlled from an external input port, if, as shown at the picture, it is possible to switch between two clocks, it will still not be possible to insert a scan. A test signal to control the select signal for the selector must be used so that, during testing, the same clock will be selected throughout the entire process (see the picture).  CLK1 CLK2 FF 0 1 Logic MUX select signal is fixed to ‘1’ for testing TST	
	LEVEL	RULE
CHECKER BEHAVIOUR	Checker scans the design hierarchy for multiplexers that have output connected to the FF clock pin: – if the multiplexer select signal is an output of combinational logic that has at least one external port connected to the gate ('and', 'or', 'nand', 'nor') before select pin => no violation;  – if the select pin is controlled directly from external port => no violation;  – otherwise => violation.	

RULE NAME	**When there is a choice between two different clock systems, one clock system must be selected throughout testing**
	

EXAMPLE-1: [1] multiplexer has output connected to the FF clock pin;

[2] multiplexer select signal is an output of another FF (can not be controlled from external input) => violation.

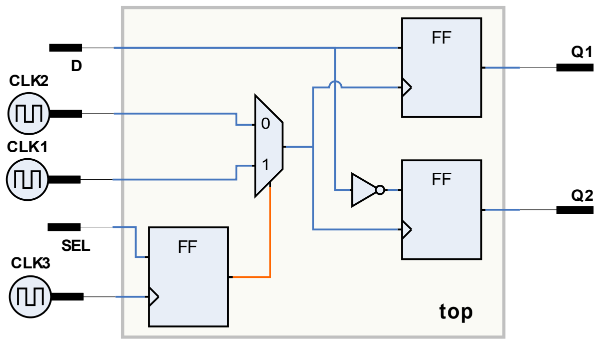
Note: signals 'top.CLK1' and 'top.CLK2' are defined as global clocks with -alint_gclk switch.

```
module top( CLK1, CLK2, CLK3, SEL, D, Q1, Q2 );

    input CLK1,CLK2,CLK3,SEL;
    input D;

    output  Q1, Q2;


    dff DFF1  ( .CLK( mux_out ), .D( ~D ), .Q( Q1 ) );

    dff DFF2  ( .CLK( mux_out ), .D( D  ), .Q( Q2 ) );

    dff DFF3  ( .CLK( CLK3 ), .D( SEL ), .Q( ff_out ) );

    mux2x1 MUX  ( .IN1( CLK1 ), .IN2( CLK2 ), .SEL( ff_out ), .Q( mux_out ));

endmodule

//D flop-flop
module dff( CLK, D, Q );

     input CLK, D;
     output reg Q;

    always @(posedge CLK)
        Q <= D;

endmodule

// Multiplexer
module mux2x1( IN1, IN2, SEL, Q );

     input SEL;
     input IN1,IN2;
     output reg Q;

    always @(*)
        if ( SEL)
             Q = IN1;
        else
             Q = IN2;

endmodule
```

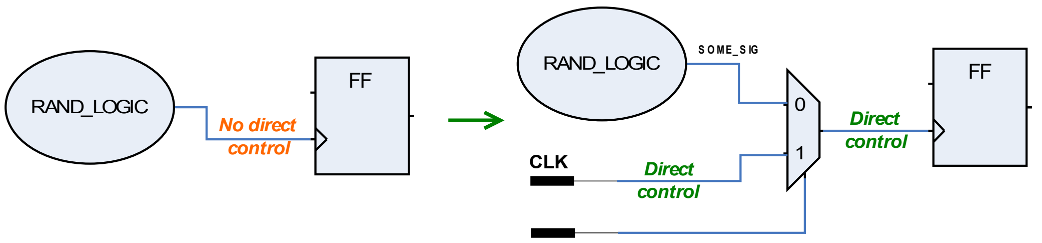
Alternative clock system: “top.CLK2”.

Alternative clock system: “top.CLK1”.

Instance "top". Selection betw een clock systems is not controllable from an external port. The single clock system must be selected throughout testing otherw ise it is impossible to insert scan.


D
CLK2
CLK1
SEL
CLK3
FF
0
1
FF
FF
Q1
Q2
top

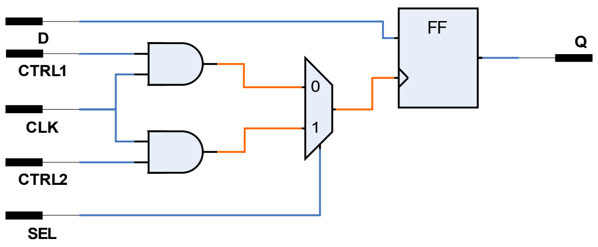
STARC_VLOG 3.3.1.3

RULE NAME	**The output of random logic should not be used as a clock**
MESSAGE	**The output of random logic is used as clock for flip-flop "{FFSigName}". Such circuit structure is unsafe for scan shift during the scan test. It is recommended to insert a selector at the final output of the random logic to make it possible to select an external port.**
PROBLEM DESCRIPTION	 RAND_LOGIC No direct control FF RAND_LOGIC SOME_SIG 0 1 CLK Direct control Direct control FF As it shown on the picture above, if the output of random logic is to be used as a clock, insert a selector at the final output of the random logic to make it possible to select an external clock.
	LEVEL: RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans the design hierarchy for flip-flops and verifies the signal that is mapped to the clock pin of each flip-flop: – if there is a multiplexer that is directly(*) connected to the clock pin of an FF and at least one input is external => no violation *(*) directly or through buffers/inverters* – if there is a situation that is **not covered** by following rules => violation: – OR with two inputs is connected to the clock pin (3.3.5.3, 3.3.5.6); – AND with two inputs is connected to the clock pin (3.3.5.4, 3.3.5.5); – latch output is connected to the clock pin of FF (3.3.5.7).

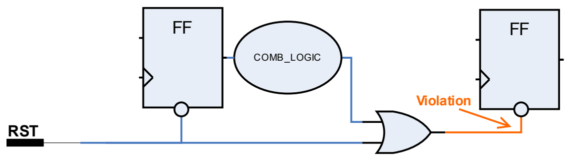
EXAMPLE-1: [1] output of random logic multiplexer is connected to clock pin of flip-flop "Q";
[2] both multiplexers inputs are internally generated signals => violation.

```
module ffg_c( CLK, SEL, CTRL1, CTRL2, DATA, Q );

     input DATA, CTRL1, CTRL2, CLK, SEL;
     output reg Q;

     assign int_clk1 = CLK & CTRL1;
     assign int_clk2 = CLK & CTRL2;

     assign int_clk = SEL ? int_clk2 : int_clk1;

     always @( posedge int_clk )
         Q <= DATA;

endmodule
```

The output of random logic is used as clock for flip-flop "Q". Such circuit structure is unsafe for scan shift during the scan test. It is recommended to insert a selector at the final output of the random logic to make it possible to select an external port.



STARC_VLOG 3.3.1.4

RULE NAME	**The reset for the FFs must be able to be controlled directly from an external input port**	
MESSAGE	**Asynchronous control logic of FF "{FFName}" is not directly controlled from external input ports. Asynchronous controls must be able to be directly controlled from external input ports, otherwise it will become impossible for ATPG tools to detect faults on control lines.**	
	DETAIL	*Asynchronous {ControlType} control is not directly controlled from external input port.*
PROBLEM DESCRIPTION	DFT requires paying special attention to the clock systems when inserting scans (*3.3.1.1*). Along with clocks, the reset lines should be considered with special care: DFT requires that there is no reset could be applied during the scan shift (it would cause the data loss).  ATPG tools will also not be able to detect faults at reset lines that are not directly controlled from an external input ports.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans the design hierarchy for flip-flops and verifies the signal that is mapped to an asynchronous control pin(s) of each flip-flop: – this signal must be directly controlled by an external input port (*) of the design – *(*) see the rule 3.3.1.1* for *details about external input ports* – if signal is not controlled by an external input => violation message is reported (message points on flip-flop signal assignment, detail points on appropriate asynchronous control) – following is the list of possible strings for the {ControlType} token in the detail message: – reset – set – set/reset	

EXAMPLE-1: [1] module 'dff' which infers a flip-flop with the reset signal controlled from an external input;

[2] module instantiation statements create two named instances:

- the external signal is mapped to the reset signal in instance 'u_dff_1';
- the output of combinational logic is mapped to the reset signal in instance 'u_dff_2' => violation.

```
module dff( clk, rst, data, q );
    input clk, data;
    output q;

    reg q;

    always @( posedge clk or posedge rst )
        if ( rst )

            q <= 1'b0;
        else
            q <= data;

endmodule
```

Asynchronous reset control is not directly controlled from an external input port.

Asynchronous control logic of FF "top.u_dff_2" is not directly controlled from external input ports. Asynchronous controls must be able to be directly controlled from external input ports, otherwise it will become impossible for ATPG tools to detect faults on control lines.

```
module top( clk, rst, gate, data1, data2, out1, out2 );
    input clk, rst, data1, data2;
    output out1, out2;

    wire rst_gate;

    assign rst_gate = rst & gate;

    dff u_dff_1 ( .clk( clk ), .rst( rst ), .data( data1 ), .q( out1 ) );

    dff u_dff_2 ( .clk( clk ), .rst( rst_gate ), .data( data2 ), .q( out2 ) );

endmodule
```

3.3.2 Dealing with hardmacros and asynchronous circuits

STARC_VLOG 3.3.2.2

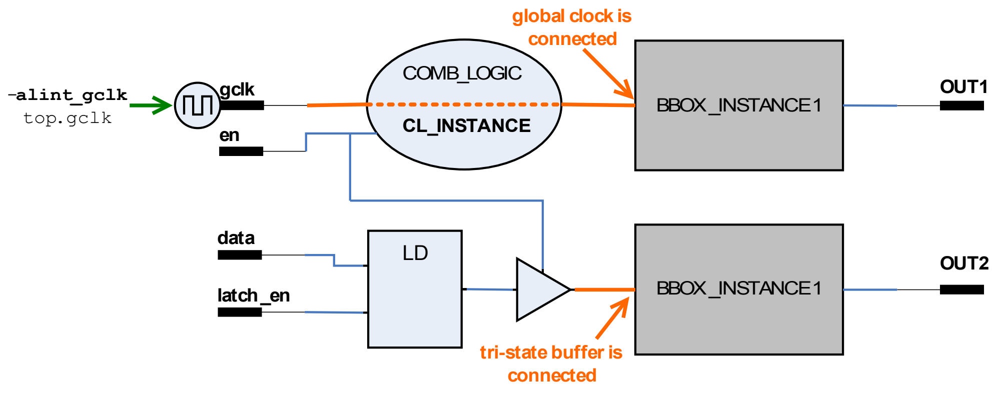
RULE NAME	**Do not connect clock pins, reset pins, or tristate outputs to black boxes**	
MESSAGE-1	**Global {GlControlType} "{GlControlName}" is connected to input of black box "{BboxName}". Do not connect clock pins, reset pins or tri-state outputs to black boxes. Such descriptions could make impossible the insertion of scan chains and test patterns generation with ATPG tools.**	
MESSAGE-2	**Output of tri-state "{TriStateName}" is connected to input of black box "{BboxName}". Do not connect clock pins, reset pins or tri-state outputs to black boxes. Such descriptions could make impossible the insertion of scan chains and test patterns generation with ATPG tools.**	
PROBLEM DESCRIPTION	There are some cases when hard macro library does not exist when LSI design data are generated. When clock/reset line or output of tri-state buffer is connected to black box, ATPG tool could fail to insert test scan and generate test patterns. When black box should be used, it is recommended to generate RTL code wherein only the inputs and outputs are defined or get a library from the vendor.	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans the design hierarchy for ***black boxes*** and verifies lines that are mapped to input pins of each black box: – if input line is *driven by global reset/clock* [(*)] or *directly driven* [(**)] by tri-state => violation is detected and appropriate warning message is displayed: – message #1 if line is driven by global clock / reset – following is set of string that are possible for {GlControlType} token: clock / reset – message #2 if line is driven by tri-state – the token {TriStateName} is not displayed if tri-state line is generated intermediately – [(*)] *see* 1.4.3.4 *for details:* – *rules for auto-detection (during the auto-detection, each signal that is connected to asynchronous control pin of flip-flop is considered as reset signal)* – *rules for global reset propagation (global reset propagates through buffers / inverters / combinational logic / multiplexers data / tri-state inputs)* – *rules for displaying info-messages (indicate the list of auto-detected clock or report about reset signals that are specified with -alint_gclk but could not be found)* – [(**)] *direct driver means that output of tri-state buffer is connected directly or through inverters / buffers.* Note: following types of modules are considered as black boxes: – empty (interface); – compiled without elab-time data (no -alint_elabchecks switch); – specified with -alint_blackbox switch.	

EXAMPLE-1: [1] consider design hierarchy that is represented on the picture below;

[2] there are two violations:

- global clock "gclk" (specified with '*alint_gclk*' attribute) reaches black box "BBOX_INSTANCE1" through combinational logic => rule violation with message #1
- tri-state buffer "sst_to_bbox" is directly connected to black box "BBOX_INSTANCE2" => rule violation with message #2

[3] note, that signal "top.gclk" is set as global clock for this design with command line switch '-alint_gclk'.



```
module top( data, gclk, en, latch_en, out1, out2 );

   input  data, gclk, en, latch_en;
   output out1, out2;

   wire stt_to_bbox, latch_to_stt;
   wire comblogic_to_bbox;

   assign stt_to_bbox = ( en ) ? latch_to_stt : 1'bz;

   comblogic   CL_INSTANCE ( .in1( gclk ), .in2( en       ), .out1( comblogic_to_bbox ) );
   latch     LATCH_INSTANCE ( .D   ( data ), .G   ( latch_en ), .Q    ( latch_to_stt      ) );
```

Instance "top". Global clock "top.gclk" is connected to input of black box "BBOX_INSTANCE1". Do not connect clock pins, reset pins or tri-state outputs to black boxes. Such descriptions could make impossible the insertion of scan chains and test patterns generation w ith ATPG tools.

```
   blackbox BBOX_INSTANCE1 ( .in1( comblogic_to_bbox ), .out1( out1 ) );
   blackbox BBOX_INSTANCE2 ( .in1( stt_to_bbox       ), .out1( out2 ) );

endmodule
```

Instance "top". Output of tri-state "sst_to_bbox" is connected to input of black box "BBOX_INSTANCE2". Do not connect clock pins, reset pins or tri-state outputs to black boxes. Such descriptions could make impossible the insertion of scan chains and test patterns generation w ith ATPG tools.

```
module latch( D, G, Q );

   input  D, G;
   output reg Q;

   always @( G )
   if( G )
      Q = D;

endmodule

module comblogic( in1, in2, out1 );

    input  in1, in2;
    output out1;

    assign out1 = in1 & in2;

endmodule

module blackbox( in1, out1 );
```

```
    input  in1;
    output out1;

endmodule
```

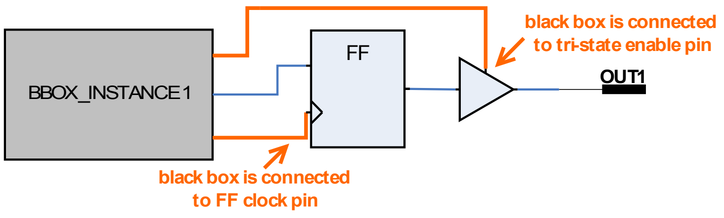
STARC_VLOG 3.3.2.3

RULE NAME	**Do not connect the outputs of a black box to clock pins, reset pins, or tristate-enable pins (Prepare the hard macro library)**	
MESSAGE	**Problem(s) with black box "{BBoxName}" output(s) connection is detected. Do not connect the outputs of black box to clock pins, reset pins or tri-state enable pins. Such descriptions could make impossible the insertion of scan chains and test patterns generation with ATPG tools.**	
	DETAIL	**Output of black box is connected to {ObjectType} "{ObjectName}" {PortType} pin**
PROBLEM DESCRIPTION	Modules that generate control signals should not be left as black boxes. Even if it comes to macro cells such as PLLs, obtain the library for macro cell from the vendor. Note that even there is a clock switching circuit for the test mode for the PLL, when the PLL is left as black box, it maybe impossible to detect whether or not the clock signal could be controlled from an external pin of the LSI. Such situations makes impossible to insert the scan and generate test patterns using the ATPG tool.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans the design hierarchy for black boxes (see 3.3.2.2) and verifies connections of their output lines: – if output line *directly drives* (*) FF clock/reset or tri-state enable pin => violation is detected: – main message #1 is displayed per black box – detail message #2 is displayed per each erroneously mapped output – following table defines set of strings that are possible for {ObjectType}-{PortType} tokens: (see table below) – *(*) direct drive means that output of black box is connected directly or through inverters / buffers.*	

{ObjectType}	*{PortType}*
FF	clock
	asynchronous reset
	asynchronous set
latch	asynchronous reset
	asynchronous set
tri-state buffer	enable

EXAMPLE-1: [1] consider design hierarchy that is represented on the picture below;

[2] violation message is displayed with 2 details: outputs of black box are connected to clock pin of FF and enable pin of tri-state buffer.


BBOX_INSTANCE1
FF
black box is connected to tri-state enable pin
OUT1
black box is connected to FF clock pin

```
module top( out1 );

   output out1;

   wire bbox_to_clk, bbox_to_data;
   wire bbox_to_stt_en, ff_to_stt;


   blackbox BBOX_INSTANCE( .out1(bbox_to_stt_en), .out2(bbox_to_data), .out3(bbox_to_clk) );
   dff      DFF_INSTANCE ( .D( bbox_to_data ), .CLK( bbox_to_clk ), .Q( ff_to_stt ) );

   assign out1 = ( bbox_to_stt_en ) ? ff_to_stt : 1'bz;

endmodule

module dff( D, CLK, Q );

   input  D, CLK;
   output reg Q;

   always @( posedge CLK )
       Q <= D;

endmodule

module blackbox( out1, out2, out3 );

   output out1;
   output out2;
   output out3;

endmodule
```

Instance "top". Problem(s) w ith black box "BBOX_INSTANCE" output(s) connection is detected. Do not connect the outputs of black box to clock pins, reset pins or tri-state enable pins. Such descriptions could make impossible the insertion of scan chains and test patterns generation w ith ATPG tools.

Output of black box is connected to tri-state buffer "out1" enable pin

Output of black box is connected to FF "Q" clock pin

3.3.3 Constraints on the use of flip-flops

STARC_VLOG 3.3.3.1

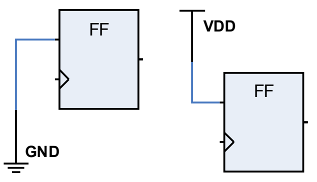
RULE NAME	**A clock must not be connected to the D input of a FF**	
MESSAGE	**Clock signal(s) "{GlClkSignalNameList}" is connected to the FF "{FFName}" data input. Such connection may lead to the risk of generating incorrect test pattern because of the racing problem. Do not connect clock signals to the FF data input.**	
PROBLEM DESCRIPTION	Inputting a clock into the D pin of a FF runs a tremendous risk of an incorrect test pattern generation. (ATPG tools generate test patterns performing the simulation with zero delays => tremendous risk that racing problem will occur). When it is impossible to avoid clock connection to the data input, selector circuitry should be added to switch to external input in the test mode.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans the design hierarchy for flip-flops and verifies the signal that is mapped to the data input of each flip-flop that is detected: – this signal must not be a clock [(*)] for the design – *[(*)] see 1.4.3.4 for definition of **clock signal** in the design: clock(s) could be directly specified (with **`-alint_gclk`** switch) or auto-detected (signal that is connected to clock pin of flip-flop will be considered as clock)* – if this is clock signal => violation message is reported (it points on flip-flop signal assignment)	

EXAMPLE-1: [1] 'clk' signal is auto-detected as a clock signal (it is connected to the clock pin of FF 'out1');
[2] clock signal propagates through 'and' gate ('clk_gate') and through a multiplexer ('mux_out');
[3] signal 'mux_out' is connected to the data input of FF 'out1' => violation;
[4] signal 'clk_gate' is connected to the set control input of FF 'out1' => no violation;

```
module top( clk, gate, sel, data, out1 );

    input clk, data, sel;
    output out1;

    reg out1;

    wire clk_gate;
    reg mux_out;

    assign clk_gate = clk & gate;

    always @( sel, clk_gate, data )
        case ( sel )
            1'b0    : mux_out <= data;
            1'b1    : mux_out <= clk_gate;
            default : mux_out <= 1'bx;
        endcase

    always @( posedge clk or negedge clk_gate )
        if ( clk_gate )
            out1 <= 1'b1;
        else
            out1 <= mux_out;

endmodule
```

Clock signal(s) "clk" is connected to the FF "out1" data input. Such connection may lead to the risk of generating incorrect test pattern because of the racing problem. Do not connect clock signals to the FF data input.

STARC_VLOG 3.3.3.2

RULE NAME	**Do not connect the input of a FF to VDD or GND**	
MESSAGE	**Fixed value is connected to the FF "{FFName}" input port(s).**	
	DETAIL	*{LineType} line is connected to FF {FFInput} port.*
PROBLEM DESCRIPTION	Do not describe flip-flops where the input is fixed to a given voltage level (if input is fixed, ATPG tool will include such flip-flops in the list of undetected paths during fault coverage measuring). Moreover, fixed flip-flops are redundant and not necessary for the synchronous circuits design: it just can form an asynchronous circuit where enable signal is connected to the clock pin. 	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker scans the design hierarchy for flip-flops and verifies signal that is mapped to an input of each flip-flop that is detected: – backward propagation performed from flip-flop data input (see 1.4.3.2 for details regarding the backward propagation) – if backward propagation stops at a constant(*) (VDD is '1', GND is '0') => violation message is reported (message points to 'always' process inferring flip-flop, detail points to flip-flop signal assignment: *{LineType}* could be VDD or GND, see Note-1 regarding possible cases for *{FFInput}* token) – (*) following are the notes regarding constants determination: – direct assignment to the signal or mapping to some input port; – unmapped signals are supplied with GND; – signals without drivers are supplied with GND; Note: possible values for {LineType} are GND and VDD; for {FFInput} are data, clock, enable, asynchronous reset, asynchronous set, synchronous reset, synchronous set.	

EXAMPLE-1: [1] instance of flip-flop has unmapped data input port => violation (note, that unmapped port is treated as GND)

```
module dff( clk, data, q );

    input clk, data;
    output q;

    always @( posedge clk )
        q <= data;

endmodule
```

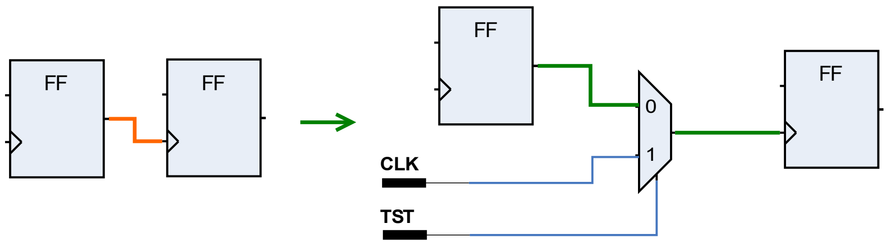
GND line is connected to FF clock port.

Fixed value is connected to the FF "u_dff_1.q" input port(s).

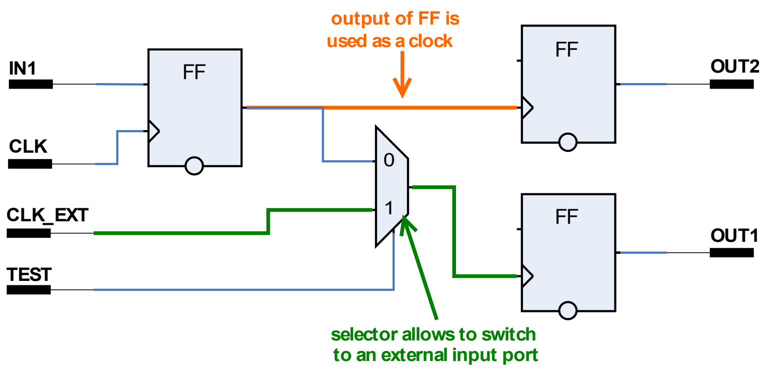
```
module top( data, out1 );

    input data;
    output out1;

    dff u_dff_1 ( .clk(), .data( data1 ), .q( out1 ) );

endmodule
```

3.3.5 DFT in clock lines

STARC_VLOG 3.3.5.2

RULE NAME	**When the output of a FF is used as a clock, switch using a selector**	
MESSAGE	**The clock pin of flip-flop "{DrivenFFName}" is driven by the output of another flip-flop. Such description is not compatible with ATPG tools, because scan insertion could not be performed. For such kind of circuit it is recommended to insert a selector to switch to a clock that is input from the outside during testing.**	
	DETAIL	*Clock pin is driven with the output of flip-flop "{DriverFFHierName}"*
PROBLEM DESCRIPTION	In case when output of flip-flop is used as a clock, the clock pin of that flip-flop could not be controlled directly from an external input port. Such description is not compatible with scan insertion and it makes impossible faults detection with ATPG tools.  During the testing, it is recommended to insert a selector to switch to a clock that is input from the outside (consider the picture above).	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans the design hierarchy for flip-flops and verifies the signal that is mapped to the clock pin of each flip-flop: – if this signal is output of another flip-flop, it should be connected through selector	

EXAMPLE-1: [1] consider design hierarchy that is represented on the picture below;
[2] output of flip-flop is connected to clock pins of two another flip-flops:
- direct connection => violation
- connection through selector => correct


output of FF is used as a clock
IN1
FF
FF
OUT2
CLK
0
CLK_EXT
1
FF
OUT1
TEST
selector allows to switch to an external input port

```
module top( IN1, IN2, IN3, CLK, CLK_EXT, TEST, OUT1, OUT2 );

    input  IN1, IN2, IN3;
    input  CLK, CLK_EXT;
    input  TEST;
    output OUT1, OUT2;

    wire   ff1_to_ff2;
    wire   mux_to_ff3;

    assign mux_to_ff3 = ( TEST ) ? ( ff1_to_ff2 ) : ( CLK_EXT );

    dff DFF_INSTANCE_1 ( .D( IN1 ), .CLK( CLK        ), .Q( ff1_to_ff2 ) );
    dff DFF_INSTANCE_2 ( .D( IN2 ), .CLK( ff1_to_ff2 ), .Q( OUT1       ) );
    dff DFF_INSTANCE_3 ( .D( IN3 ), .CLK( mux_to_ff3 ), .Q( OUT2       ) );

endmodule

module dff( D, CLK, Q );

    input  D, CLK;
    output reg Q;

    always @( posedge CLK )
        Q <= D;

endmodule
```

Instance "top.DFF_INSTANCE2". The clock pin of flip-flop "Q" is driven by the output of another flip-flop. Such description is not compatible w ith ATPG tools, because scan insertion could not be performed. For such kind of circuit it is recommended to insert a selector to sw itch to a clock that is input from the outside during testing.

Clock pin is driven w ith the output of flip-flop "top.DFF_INSTANCE_1.Q"

STARC_VLOG 3.3.5.3

RULE NAME

Circuits that use OR gating of clocks and internally generated signals should be tied to a specific voltage level using an AND gate

MESSAGE

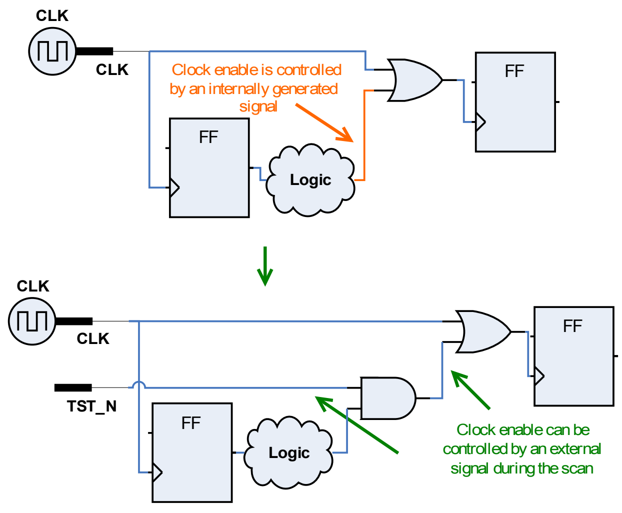
Clock “{GClkName}” passes through the OR gate logic. Internally generated signals, connected to other pins of this OR gate, must be controlled by an external port via the AND gate.

PROBLEM DESCRIPTION

Ability to control clock pins from an external inputs is essential for DFT (see 3.3.1.1). Those clocks that cannot be controlled directly from the outside require switching to an external clock during testing. From this point of view use of gated clocks should be avoided, however use of gated clocks may be unavoidable in efforts to reduce power consumption.

There are two techniques for gated clocks. The first is the method of enabling the clock line through the use of an OR gate, and the second is that of enabling the clock line through the use of an AND gate (see 3.3.5.4). In addition, there are also methods that use latches (see 3.3.5.5) and FFs (see 3.3.5.6) to produce gated clocks.

Consider the picture below. When OR gating is used, when the clock enable signal goes to ’1’ at the time of the scan shift, the clock will no longer be input into the flip-flop, so the scan shift will not be performed properly (the top part of the picture). At the bottom part DFT correct circuit is shown. At the time of the scan shift, the clock enable signal must be tied to ’0’ by the test signal that is input from outside of the LSI. As shown at the picture, the clock enable input of OR gate can be fixed to a specific voltage by inserting an AND gate in the stage prior to the OR gate and then inputting ’0’ into one side by external TST_N signal.


CLK
CLK
Clock enable is controlled by an internally generated signal
FF
FF
Logic
CLK
CLK
TST_N
FF
FF
Logic
Clock enable can be controlled by an external signal during the scan

LEVEL	RECOMMENDATION 1

CHECKER BEHAVIOR

Checker collects collect external signals and verifies whether these signals are directly (*) connected to an OR gate that feeds directly(*) clock pin of an FF:

- if other input of OR gate is external => no violation;
- else if other input is internal:
  - if it is supplied with AND gate with at least one input connected directly(*) to an external port;
  - else if there is no such AND gate => violation.
- *(*) for this rule, signal that is **directly connected** may be connected directly or through buffers/inverters.*

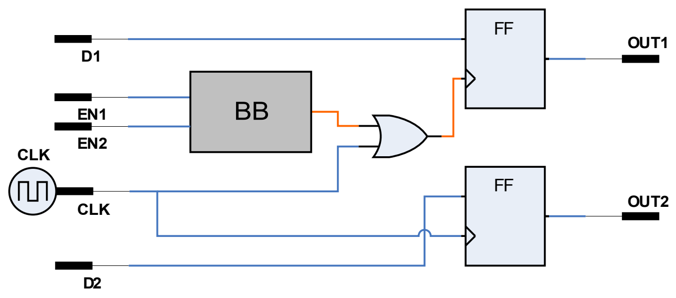
Note-1: OR gates with two inputs only are considered.

Note-2: if an OR gate is driven by an FF then this checker is not regard the situation (for more details see 3.3.5.6).

EXAMPLE-1: [1] global clock signal 'CLK' is directly connected to an OR gate;
[2] other input of OR gate is internal and it is not supplied with AND gate => violation.

```
module top( CLK, EN1, EN2, D1, D2, OUT1, OUT2 );

     input CLK;
     input EN1, EN2, D1, D2;

     output OUT1, OUT2;

     assign or_clk = bb_out | CLK;

     bb BB  ( .IN1( EN1 ), .IN2( EN2 ), .OUT( bb_out ) );

     dff DFF1  ( .CLK( or_clk ), .D( D1   ), .Q( OUT1 ) );

     dff DFF2  ( .CLK( CLK    ), .D( D2   ), .Q( OUT2 ) );

endmodule

//D flop-flop
module dff( CLK, D, Q );

     input CLK, D;

     output reg Q;

     always @(posedge CLK)
          Q <= D;

endmodule

// BB interface definition
module bb  ( input IN1, IN2, output OUT);

endmodule
```

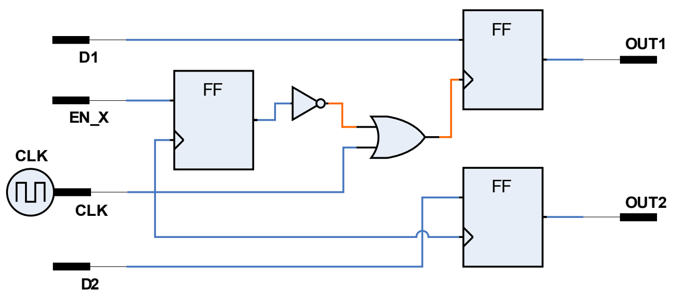
Instance "top". Clock "CLK" passes through the OR gate logic. Internally generated signals, connected to other pins of this OR gate, must be controlled by an external port via the AND gate.



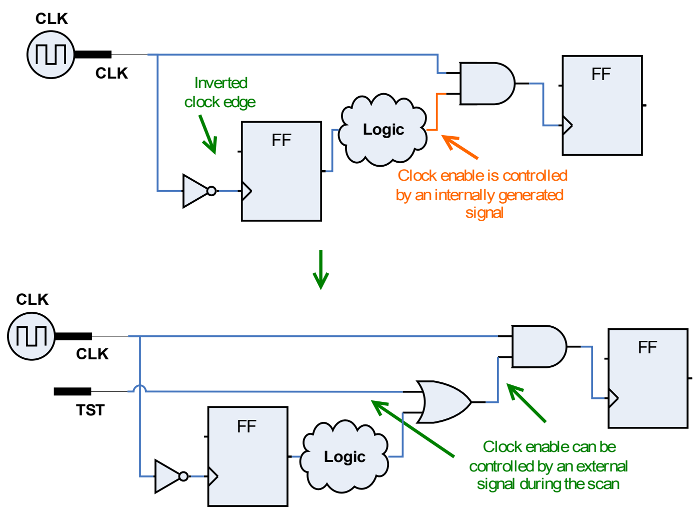
EXAMPLE-2: [1] global clock signal 'CLK' is directly connected to an OR gate;
[2] other input of the gate is driven by FF (inverter is not taken into account) => no violation (case for 3.3.5.6).

```
module top( CLK, EN_X, D1, D2, OUT1, OUT2 );

     input CLK;
     input EN_X, D1, D2;

     output OUT1, OUT2;

     assign or_clk = !en_ff_out | CLK;

     dff DFF1  ( .CLK( CLK    ), .D( EN_X ), .Q( en_ff_out ) );

     dff DFF2  ( .CLK( or_clk ), .D( D1   ), .Q( OUT1 ) );

     dff DFF3  ( .CLK( CLK    ), .D( D2   ), .Q( OUT2 ) );

endmodule
```

```
//D flop-flop
module dff( CLK, D, Q );

     input CLK, D;

     output reg Q;

     always @(posedge CLK)
          Q <= D;

endmodule
```



STARC_VLOG 3.3.5.4

RULE NAME	**Circuits that use AND gating of clocks and internally generated signals should be tied to a specific voltage level using an OR gate**
MESSAGE	**Clock “{GClkName}” passes through the AND gate logic. Internally generated signals, connected to other pins of this AND gate, must be controlled by an external port via the OR gate.**
PROBLEM DESCRIPTION	Ability to control clock pins from an external inputs is essential for DFT (see *3.3.1.1*). Those clocks that cannot be controlled directly from the outside require switching to an external clock during testing. From this point of view use of gated clocks should be avoided, however use of gated clocks may be unavoidable in efforts to reduce power consumption. There are two techniques for gated clocks. The first is the method of enabling the clock line through the use of an OR gate (see *3.3.5.3*), and the second is that of enabling the clock line through the use of an AND gate. In addition, there are also methods that use latches (see *3.3.5.5*) and FFs (see *3.3.5.6*) to produce gated clocks. In AND gating an inverted clock serves as the base. At the picture the enable signal will be the output of a FF using an inverted clock. So that half cycle is available to generate enable signal. When gating is performed on a positive clock signal, a hazard will be produced on the clock line, causing a malfunction. An inversion may also be added after AND gate to obtain design in which everything operates on a inverted clock. Consider the picture below. When AND gating is used, if the clock enable signal goes to ’0’ at the time of the scan shift, the clock will no longer be input into the flip-flop, so the scan shift will not be performed properly (the top part of the picture). At the bottom part DFT correct circuit is shown. At the time of the scan shift, the clock enable signal must be tied to ’1’ by the test signal that is input from outside of the LSI. As shown at the picture, the clock enable input of AND gate can be fixed to a specific voltage by inserting an OR gate in the stage prior to the AND gate and then inputting ’1’ into one side by external TST signal.  CLK CLK Inverted clock edge FF Logic FF Clock enable is controlled by an internally generated signal CLK CLK TST FF Logic FF Clock enable can be controlled by an external signal during the scan
	LEVEL: RECOMMENDATION 1
CHECKER BEHAVIOR	Checker collects collect external signals and verifies whether these signals are directly (*) connected to an AND gate that feeds directly(*) clock pin of an FF: – if other input of AND gate is external => no violation; – else if other input is internal: – if it is supplied with OR gate with at least one input connected directly(*) to an external port; – else if there is no such OR gate => violation. – $^{(*)}$ *for this rule, signal that is* ***directly connected*** *may be connected directly or through*

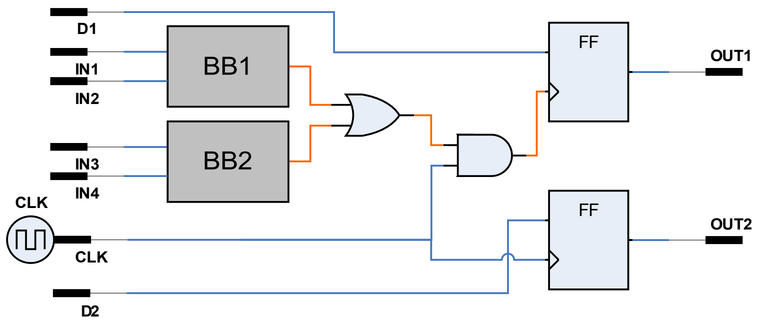
RULE NAME	**Circuits that use AND gating of clocks and internally generated signals should be tied to a specific voltage level using an OR gate**
	buffers/inverters. Note-1: AND gates with two inputs only are considered. Note-2: if an AND gate is driven by a latch then this checker is not regard the situation (for more details see 3.3.5.5).

EXAMPLE-1: [1] CLK signal is directly connected to an AND gate that feeds directly clock pin of an FF;
[2] the other input of AND gate is supplied with OR gate;
[3] both inputs of OR gate are internally generated (they are outputs of black boxes) => violation.

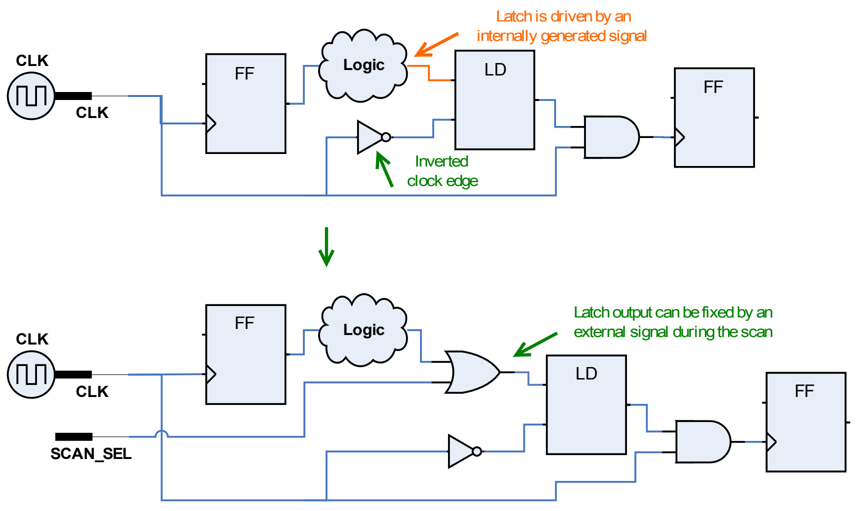
```
module top( CLK, IN1, IN2, IN3, IN4, D1, D2, OUT1, OUT2 );

     input CLK;
     input IN1, IN2, IN3, IN4;
     input D1, D2;

     output OUT1, OUT2;

     assign and_clk = ( bb1_out | bb2_out ) & CLK;

     dff DFF1  ( .CLK( and_clk ), .D( D1 ), .Q( OUT1 ) );

     dff DFF2  ( .CLK( CLK     ), .D( D2 ), .Q( OUT2 ) );

     bb1 BB1   ( .IN1( IN1 ), .IN2( IN2 ), .OUT( bb1_out) );

     bb2 BB2    ( .IN1( IN3 ), .IN2( IN4 ), .OUT( bb2_out) );

endmodule
```

Instance "top". Clock "CLK" passes through the AND gate logic. Internally generated signals, connected to other pins of this AND gate, must be controlled by an external port via the OR gate.

```
//D flop-flop
module dff( CLK, D, Q );

     input CLK, D;
     output reg Q;

     always @(posedge CLK)
          Q <= D;

endmodule


// BB1 interface definition
module bb1 ( input IN1, IN2, output OUT);

endmodule


// BB2 interface definition
module bb2 ( input IN1, IN2, output OUT);

endmodule
```



STARC_VLOG 3.3.5.5

RULE NAME	**Circuits in which gating is performed on clocks and latch outputs should have an OR performed on the scan select and the stage prior to the latches**	
MESSAGE	**Latch (“{LatchName}”) is used for gating clock line and its output is not fixed to '1'. Use OR gate with external signal fixed to high voltage prior to the latch to provide clock to the FFs during scan shift.**	
	DETAIL	*The output of the latch is used for gating with clock “{ClkName}”.*
PROBLEM DESCRIPTION	Ability to control clock pins from an external inputs is essential for DFT (see *3.3.1.1*). Those clocks that cannot be controlled directly from the outside require switching to an external clock during testing. From this point of view use of gated clocks should be avoided, however use of gated clocks may be unavoidable in efforts to reduce power consumption. There are two techniques for gated clocks. The first is the method of enabling the clock line through the use of an OR gate (see *3.3.5.3*), and the second is that of enabling the clock line through the use of an AND gate (see *3.3.5.4*). In addition, there are also methods that use latches and FFs (see *3.3.5.6*) to produce gated clocks. When a latch of an inverted clock is placed prior to the AND gate latch output does not change when the clock is a high pulse. The latch goes to a through state when the pulse goes low after the falling edge, at which time the value of an enable signal propagates to the latch output. Consequently, if the AND with the clock signal is performed after the enable signal passes through the latch (as shown at the picture), then the enable signal will be accepted at the next rising edge of the clock. Such scheme of clock gating provides more stability then scheme described in *3.3.5.4*. Gated clocks using latches such as this normally fix latch output by inserting OR gate prior to the latch (bottom part of the picture). An OR is performed on the SCAN_SEL signal, which is used to switch the circuit to the test mode. When SCAN_SEL is ’1’ (at the time of the scan shift), the output of the OR is always ’1’, so a ’1’ is always input into the AND. As a result, the CLK signal is used to synchronize the FF during the scan shift.  Latch is driven by an internally generated signal CLK CLK FF Logic LD FF Inverted clock edge Latch output can be fixed by an external signal during the scan CLK CLK SCAN_SEL FF Logic LD FF	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker collects collect external signals and verifies whether these signals are directly (*) connected to an AND gate that feeds directly(*) clock pin of an FF: – if a latch is connected to an AND gate: – if its input pin is directly(*) driven by an OR gate with one external input => no violation	

RULE NAME	**Circuits in which gating is performed on clocks and latch outputs should have an OR performed on the scan select and the stage prior to the latches**
	– else if there is no such a gate => violation – $^{(*)}$ *for this rule, signal that is* ***directly connected*** *may be connected directly or through buffers/inverters.* Note-1: AND gates with two inputs only are considered. Note-2: if there is no latch connected to AND gate, then this checker is not regard the situation (for more details see *3.3.5.4*).

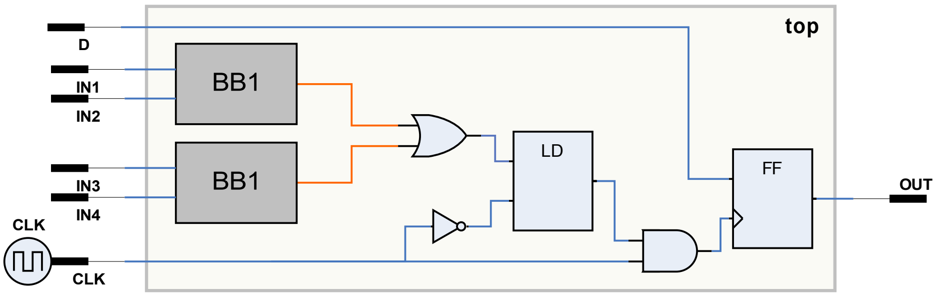
EXAMPLE-1: [1] external signal is directly connected to AND gate which feeds FF clock pin;
[2] latch is connected to the other input of AND gate;
[3] latch input pin is driven by OR gate;
[4] OR gate inputs are both internally generated signals => violation.

```
module top( CLK, IN1, IN2, IN3, IN4, D, OUT );

    input CLK;
    input IN1, IN2, IN3, IN4, D;

    output OUT;

    assign bb_or = bb1_out | bb2_out;

    assign clk_and = latch_out & CLK;

    dff DFF ( .CLK( clk_and ), .D( D ), .Q( OUT ) );

    latch LD ( .G( ~CLK ), .D( bb_or ), .Q( latch_out ) );

    bb1 BB1  ( .IN1( IN1 ), .IN2( IN2 ), .OUT( bb1_out) );

    bb2 BB2   ( .IN1( IN3 ), .IN2( IN4 ), .OUT( bb2_out) );

endmodule


//D flop-flop
module dff( CLK, D, Q );

    input CLK, D;
    output reg Q;

    always @(posedge CLK)
        Q <= D;

endmodule


//D-latch
module latch( G, D, Q );

    input G, D;
    output reg Q;

    always @(G, D)
        if ( G )
            Q <= D;

endmodule

// BB1 interface definition
module bb1 ( input IN1, IN2, output OUT);

endmodule
```

The output of the latch is used for gating w ith clock "CLK".

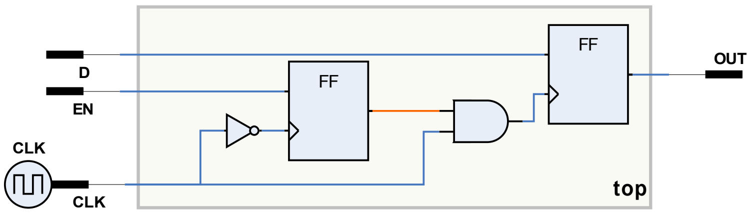
Instance "top.LD" Latch ("Q") is used for gating clock line and its output is not fixed to '1'. Use OR gate w ith external signal fixed to high voltage prior to the latch to provide clock to the FFs during scan shift.

```
// BB2 interface definition
module bb2 ( input IN1, IN2, output OUT);

endmodule
```

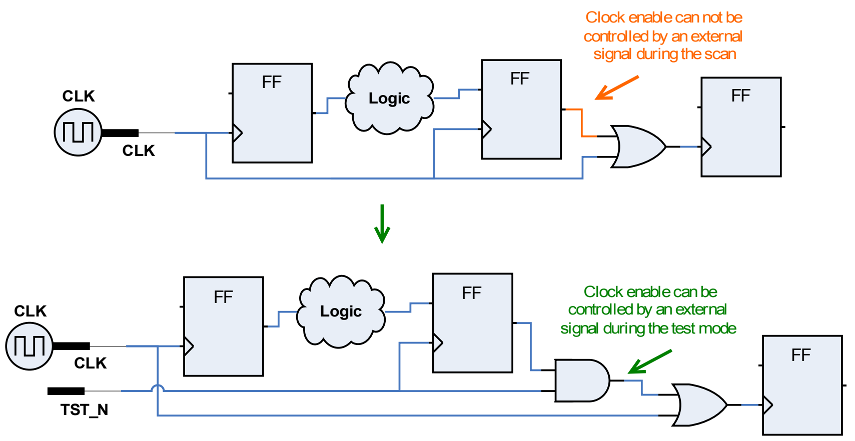


EXAMPLE-2: [1] external signal is directly connected to AND gate which feeds FF clock pin;
[2] there is no latch connected to the other AND gate input => no violation.
Note: this is the case for rule *3.3.5.4*.

```
module top( CLK, EN, D, OUT );

    input CLK, EN;
    input D;

    output OUT;

    assign clk_and = ff_out & CLK;

    dff DFF1 ( .CLK( ~CLK ), .D( EN ), .Q( ff_out ) );

    dff DFF2 ( .CLK( clk_and ), .D( D ), .Q( OUT ) );

endmodule

//D flop-flop
module dff( CLK, D, Q );

    input CLK, D;
    output reg Q;

    always @(posedge CLK)
        Q <= D;

endmodule
```



STARC_VLOG 3.3.5.6

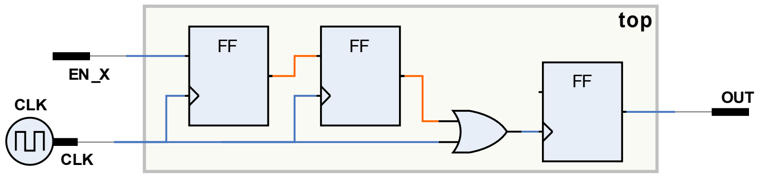
RULE NAME	**When gating a clock and the output of a FF, tie to a specific voltage level by performing AND gating with the stage after the FF**	
MESSAGE	**Detected FF ("{FFName}") using for gating clock without output tied to a specific voltage. Use AND gate with external signal fixed to low voltage after the FF to provide clock to the FFs without stopping during scan shift.**	
	DETAIL	*The output of the FF is used for gating with clock "{ClkName}".*
PROBLEM DESCRIPTION	Ability to control clock pins from an external inputs is essential for DFT (see *3.3.1.1*). Those clocks that cannot be controlled directly from the outside require switching to an external clock during testing. From this point of view use of gated clocks should be avoided, however use of gated clocks may be unavoidable in efforts to reduce power consumption. There are two techniques for gated clocks. The first is the method of enabling the clock line through the use of an OR gate (see *3.3.5.3*), and the second is that of enabling the clock line through the use of an AND gate (see *3.3.5.4*). In addition, there are also methods that use latches (see *3.3.5.5*) and FFs to produce gated clocks. When OR gating is used, a FF is used as the previous stage, rather than a latch (*3.3.5.5*). Clock enable signal must arrive within a half-cycle interval, otherwise the clock pulse width will be different, and the circuit will not function safely. OR gating with FF provides more stability then scheme described in *3.3.5.3*, because enable signal changes only by clock edge. But for DFT purposes when OR gating is performed, an AND gate with one external input should be inserted in the previous stage. So that, during the test a '0' may be input into one side of the AND gate causing the output of the AND go to '0' and as a result a stable clock is supplied to the FF (the bottom part of the picture).  Clock enable can not be controlled by an external signal during the scan CLK CLK FF Logic FF FF CLK CLK TST_N FF Logic FF Clock enable can be controlled by an external signal during the test mode FF	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOUR	Checker collects collect external signals and verifies whether these signals are directly (*) connected to an OR gate that feeds directly(*) clock pin of an FF: – if another input of OR gate is driven by FF => violation; – else if it is driven by an AND gate: – if one of the inputs of AND gate is driven by FF and another is directly(*) connected to external port => no violation – otherwise => violation – else if other input of OR gate is driven by anything else => situation for 3.3.5.3 – *(*) for this rule, signal that is **directly connected** may be connected directly or through buffers/inverters.* Note-1: AND gates with two inputs only are considered.	

EXAMPLE-1: [1] external signal is directly connected to OR gate which feeds FF clock pin;
[2] FF is connected to the other input of OR gate;
[3] FF data input is not driven by OR gate => violation.

```
module top( CLK, EN_X, D, OUT );

     input CLK,EN_X;
     input D;

     output OUT;

     assign or_clk = ff2_out | CLK;

     dff DFF1 ( .CLK( CLK ), .D( EN_X ), .Q( ff1_out ) );

     dff DFF2 ( .CLK( CLK ), .D( ff1_out ), .Q( ff2_out ) );

     dff DFF3 ( .CLK( or_clk ), .D( D ), .Q( OUT ) );

endmodule

//D flop-flop
module dff( CLK, D, Q );

     input CLK, D;

     output reg Q;

     always @(posedge CLK)
          Q <= D;

endmodule
```

The output of the FF is used for gating w ith clock "CLK".

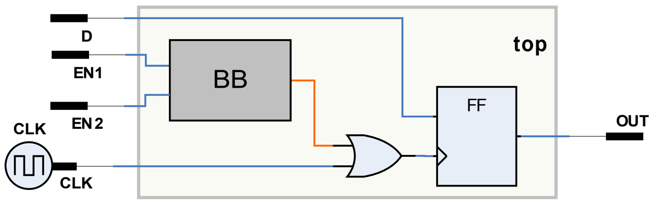
Instance "top.DFF2" Detected FF ("Q") using for gating clock w ithout output tied to a specific voltage. Use AND gate w ith external signal fixed to low voltage after the FF to provide clock to the FFs w ithout stopping during scan shift.



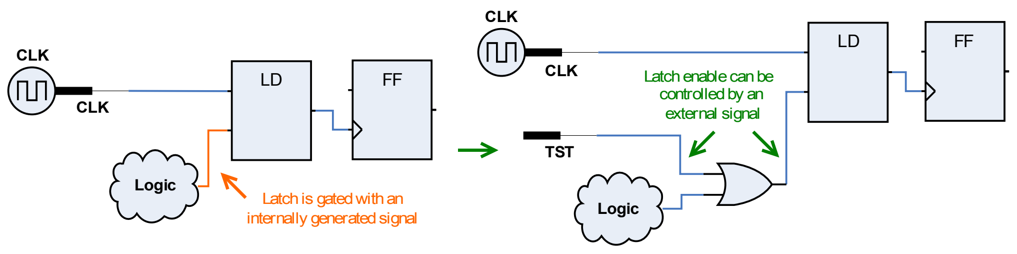
EXAMPLE-2: [1] external signal is directly connected to OR gate which feeds FF clock pin;
[2] FF is not connected to the other input of OR gate => no violation.
Note: this is the case for rule 3.3.5.3.

```
module top( CLK, EN1, EN2, D, OUT );

     input CLK;
     input EN1, EN2, D;

     output OUT;

     assign or_clk = bb_out | CLK;

     dff DFF ( .CLK( or_clk ), .D( D ), .Q( OUT ) );

     bb BB     ( .IN1( EN1 ), .IN2( EN2 ), .OUT( bb_out) );

endmodule

//D flop-flop
module dff( CLK, D, Q );

     input CLK, D;
     output reg Q;

     always @(posedge CLK)
          Q <= D;

endmodule
```

```
// BB interface definition
module bb ( input IN1, IN2, output OUT);

endmodule
```



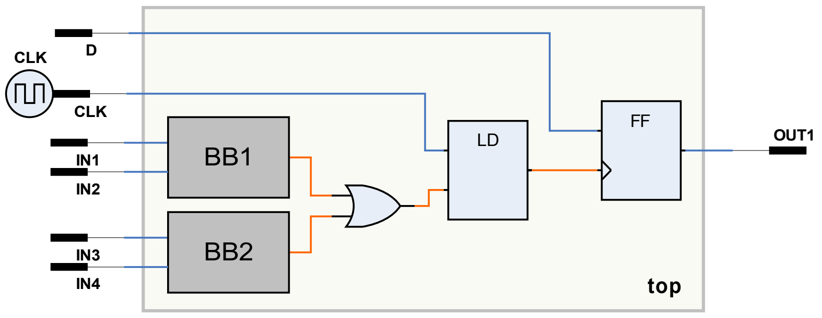
STARC_VLOG 3.3.5.7

RULE NAME	**When the output of the latch is used as a clock, tie to a specific voltage level by performing OR gating with the latch clock input**
MESSAGE	**Clock "{GClkName}" passes through the latch "{LatchName}" but the enable input is not set to a specific voltage. Use OR gating with test signal to tie the clock pin to '1', to provide continual pass of the clock signal to FFs.**
PROBLEM DESCRIPTION	Ability to control clock pins from an external inputs is essential for DFT (see *3.3.1.1*). Those clocks that cannot be controlled directly from the outside require switching to an external clock during testing. From this point of view use of gated clocks should be avoided, however use of gated clocks may be unavoidable in efforts to reduce power consumption. There are two techniques for gated clocks. The first is the method of enabling the clock line through the use of an OR gate (see *3.3.5.3*), and the second is that of enabling the clock line through the use of an AND gate (see *3.3.5.4*). In addition, latches (see *3.3.5.5*) and FFs (see *3.3.5.6*) may be added to gating circuit to produce gated clocks. There are also methods to control the clock lines where a latch is used instead of a gated clock. Such methods are not recommended because they are not desirable in terms of timing analysis. If this method is being used to control the clock lines, signal that is input into the clock pin for the latch to should be able to set to '1' during the test mode. At the picture below the test input and the latch clock input are being tied to a specific value through the use of an OR gate so that the signal always passes through the latch.  CLK CLK LD FF Logic Latch is gated with an internally generated signal CLK CLK LD FF Latch enable can be controlled by an external signal TST Logic LEVEL: RECOMMENDATION 2
CHECKER BEHAVIOUR	Checker collects collect external signals and verifies whether any of these signals is connected directly (*) to data pin of latch: – if latch output is connected to the clock pin of FF: – if the enable input of latch is an external port or it is supplied with OR gate with at least one external signal => no violation; – otherwise => violation – *(*) for this rule, signal that is **directly connected** may be connected directly or through buffers/inverters.* Note: OR gates with two inputs only are considered.

EXAMPLE-1: [1] external signal CLK directly connected to data pin of the latch;
[2] latch output is connected to the clock pin of FF;
[3] latch enable input is is supplied with OR gate with internal signals => violation.

```
module top( CLK, IN1, IN2, IN3, IN4, D, OUT );

    input CLK;
    input IN1, IN2, IN3, IN4, D;

    output OUT;

    assign bb_or = bb1_out | bb2_out;

    dff DFF ( .CLK( latch_out ), .D( D ), .Q( OUT ) );
    latch LD ( .G ( bb_or ), .D ( CLK ), .Q ( latch_out) );

    bb1 BB1  ( .IN1( IN1 ), .IN2( IN2 ), .OUT( bb1_out) );

    bb2 BB2   ( .IN1( IN3 ), .IN2( IN4 ), .OUT( bb2_out) );

endmodule

//D flop-flop
module dff( CLK, D, Q );

    input CLK, D;
    output reg Q;

    always @(posedge CLK)
        Q <= D;

endmodule

//D-latch
module latch( G, D, Q );

    input G, D;
    output reg Q;

    always @(G, D)
        if ( G )
            Q <= D;

endmodule

// BB1 interface definition
module bb1 ( input IN1, IN2, output OUT);

endmodule


// BB2 interface definition
module bb2 ( input IN1, IN2, output OUT);

endmodule
```

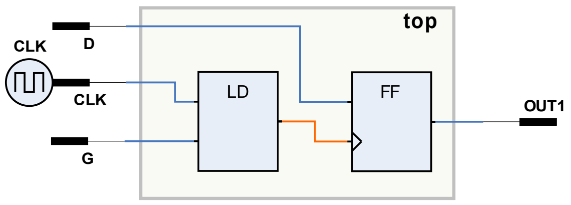
Instance "top". Clock "CLK" passes through the latch "Q" but the enable input is not set to a specific voltage. Use OR gating w ith test signal to tie the clock pin to '1', to provide continual pass of the clock signal to FFs.



EXAMPLE-2: [1] external signal CLK directly connected to data pin of the latch;
[2] latch output is connected to the clock pin of FF;
[3] latch enable input is an external signals => no violation.

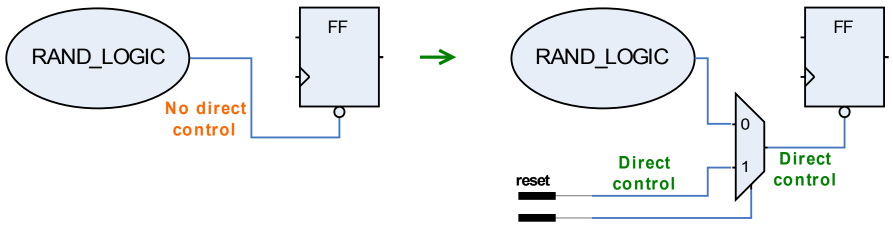
```
module top( CLK, G, D, OUT );

    input CLK, G;
    input D;

    output OUT;

    dff DFF ( .CLK( latch_out ), .D( D ), .Q( OUT ) );

    latch LD ( .G ( G ), .D ( CLK ), .Q ( latch_out) );

endmodule
```

```
//D flop-flop
module dff( CLK, D, Q );

     input CLK, D;
     output reg Q;

     always @(posedge CLK)
          Q <= D;

endmodule

//D-latch
module latch( G, D, Q );

     input G, D;
     output reg Q;

     always @(G, D)
          if ( G )
               Q <= D;

endmodule
```



3.3.6 DFT in reset lines

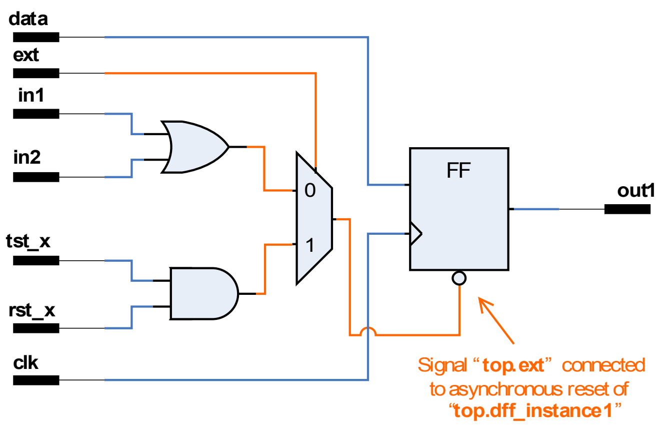
STARC_VLOG 3.3.6.1

RULE NAME	**When the output of random logic is applied to an asynchronous set or reset pin, block the propagation of the random logic output**
MESSAGE	**The output of random logic is used as asynchronous control for flip-flop "{FFSigName}". Such circuit structure is unsafe for scan shift during the scan test. It is recommended to insert a selector at the final output of the random logic to make it possible to select an external port.**
PROBLEM DESCRIPTION	DFT for reset lines requires that they should be structured so that no reset is applied to a FF during the scan shift, otherwise some data will be lost. It is necessary for the reset lines to be controlled directly from external input ports (see *3.3.1.4*). If an output from random logic is connected to an asynchronous set or reset pin or gated with a reset input, you need to ensure the propagation of the random logic output is blocked during testing to make it possible for the ATPG tool to detect faults in the reset lines.  RAND_LOGIC FF No direct control RAND_LOGIC FF 0 1 reset Direct control Direct control
	LEVEL: RULE
CHECKER BEHAVIOR	Checker scans the design hierarchy for flip-flops and verifies the signal that is mapped to the asynchronous control pin of each flip-flop: – this signal must be controlled by an external input port (*) – *(*) for this rule, signal that is* ***controlled by an external input port*** *is a signal that is connected directly or through buffers/inverters/MUXes to any external input port of the design* – if signal is not controlled by an external input port => violation (message points to flip-flop signal assignment).

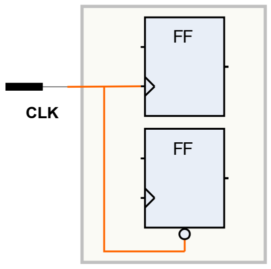
EXAMPLE-1: [1] consider the picture below;
[2] MUX's output signal that is mapped to the asynchronous control of the flip-flop;
[3] data inputs of the MUX are the outputs of combination logic;
[4] MUX control is connected to the external port, but MUX output is still treated as random logic;
[5] propagation of random logic output is not blocked and FF reset cannot be directly controlled from an external input => violation.

```
module top( ext, tst_x, rst_x, in1, in2, clk, data1, out1 );

    input  ext;
    input  tst_x;
    input  rst_x;
    input  in1;
    input  in2;
    input  clk;
    input  data1;

    output out1;

    wire   or_out;
    wire   and_out;
    wire   mux_out;
    wire   ff1_out;

    assign and_out = tst_x & rst_x;

    assign or_out  = in1 | in2;

    assign mux_out = ( ext ) ? and_out : or_out;

    dff_r  DFF_INSTANCE1 ( .DATA( data1 ), .CLK ( clk ), .RES( mux_out ), .Q( out1 ) );

endmodule

module dff_r( DATA, CLK, RES, Q );

    input  DATA;
    input  CLK;
    input  RES;

    output Q;
    reg    Q;

always @( posedge CLK or negedge RES )

    if ( !RES )

        Q <= 1'b0;

    else

        Q <= DATA;

endmodule
```

top.DFF_INSTANCE2. The output of random logic is used as asynchronous control for flip-flop "Q". Such circuit structure is unsafe for scan shift during the scan test. It is recommended to insert a selector at the final output of the random logic to make it possible to select an external port.


data
ext
in1
in2
tst_x
rst_x
clk
0
1
FF
out1
Signal "top.ext" connected to asynchronous reset of "top.dff_instance1"

STARC_VLOG 3.3.6.2

RULE NAME	**Do not mix clock lines and reset lines**	
MESSAGE	**Signal "{SigName}" is used both for clock and reset. Do not mix clock and reset lines to avoid problems with DFT. In such descriptions all lines to which clock signal is connected will be excluded from scanning.**	
	DETAIL-1	*Connection to clock pin of flip-flop "{FFName}" is detected.*
	DETAIL-2	*Connection to asynchronous control pin of flipflop "{FFName}" is detected.*
	DETAIL-3	*Signal "{SigName}" is also used both for clock and reset in the same connections.*
PROBLEM DESCRIPTION	In circuits where clock lines and reset lines are mixed, all FFs to which the CLK signal is connected will be excluded from scanning. Normally, clock lines and reset lines are not mixed in a design. However, there are some rare cases that use circuitry as shown in the picture. But this type of design is risky and should not be used, even when no scan is to be inserted. 	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects signals that are used simultaneously as clock and reset for FF: – if such signals are detected => violation (main message per signal declaration + appropriate detail (detail-2/detail-3) per each connection to FF); – if several signals drive same FF pin (clock/reset) => violation (only one message for all of them + detail-4 per each another driver-signal); *Note:* analysis starts from asynchronous control pin of each FF and backward propagation is performed (propagation rules are the same as for global reset/clock, see 1.4.3.4) to detect "forks" that could be connected to clock pin(s) of other/same FF(s).	

EXAMPLE-1: [1] consider the picture below;

[2] signal top.clk1 is used simultaneously as clock (top.dff_instance1) and reset (top.dff_instance1) => violation (detail-1 + detail-2).

Note: signal is propagated through the MUXes.

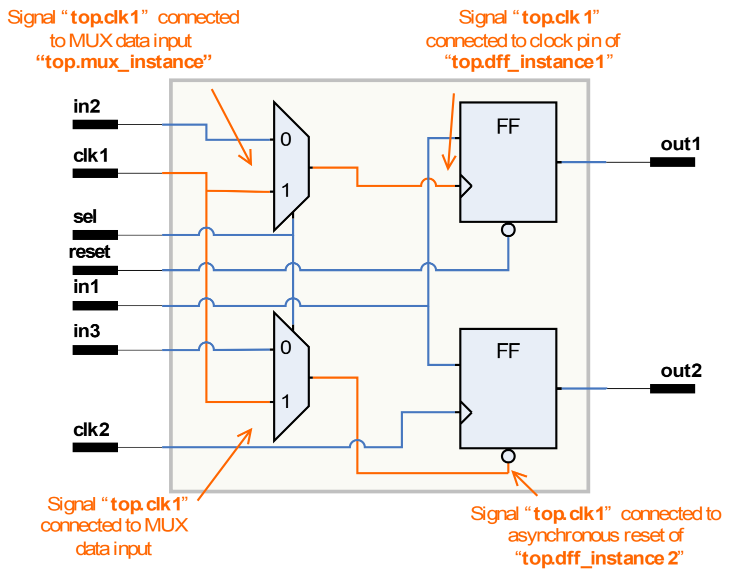
```
module top( in1, in2, in3, clk1, clk2, reset, sel, out1, out2 );

    input  in1;
    input  in2;
    input  in3;
    input  clk1;
    input  clk2;
    input  reset;
    input  sel;

    output out1;
    output out2;

    wire mux_to_ff1;
    wire mux_to_ff2;

    assign mux_to_ff2 = ( sel ) ? clk1 : in3;

    mux MUX_INSTANCE  ( .in1( clk1 ), .in2( in2 ), .sel( sel ), .out1( mux_to_ff1 ) );

    dff DFF_INSTANCE1 ( .D( in1 ), .CLK( mux_to_ff1 ), .RESET( reset ), .Q( out1 ) );
```

Instance "top". Signal "clk1" is used both for clock and reset. Do not mix clock and reset lines to avoid problems w ith DFT. In such descriptions all lines to w hich clock signal is connected w ill be excluded from scanning.

```
    dff DFF_INSTANCE2 ( .D( in1 ), .CLK( clk2 ), .RESET( mux_to_ff2 ), .Q( out2 ) );

endmodule


module dff( D, CLK, RESET, Q );

    input  D;
    input  CLK;
    input  RESET;

    output Q;
    reg    Q;

    always @( posedge CLK or posedge RESET )

        if ( RESET )

            Q <= 1'b0;

        else

            Q <= D;

endmodule


module mux( in1, in2, sel, out1 );

    input  in1;
    input  in2;
    input  sel;

    output out1;

    assign out1 = ( sel ) ? in1 : in2;

endmodule
```

Connection to clock pin of flip-flop “Q” is detected.

Connection to asynchronous control pin of flipflop “Q” is detected.



Signal “top.clk1” connected to MUX data input “top.mux_instance”
Signal “top.clk 1” connected to clock pin of “top.dff_instance1”
in2
clk1
sel
reset
in1
in3
clk2
0
1
FF
out1
FF
out2
Signal “top.clk1” connected to MUX data input
Signal “top.clk1” connected to asynchronous reset of “top.dff_instance 2”

EXAMPLE-2: [1] consider the picture below;

[2] if several signals (in1, in2) drive the same FF pin (clock and reset) => violation (detail-1 + detail-2 + detail-4).

```
module top( in1, in2, clk, data1, data2, out1, out2 );

    input  in1;
    input  in2;
    input  clk;
    input  data1;
    input  data2;

    output out1;
    output out2;

    wire   and_out;

    assign and_out = in1 & in2;

    dff   DFF_INSTANCE2  ( .D( data1 ), .CLK( and_out ), .Q( out1 ) );

    dff_r DFF_INSTANCE1  ( .D( data2 ), .CLK( clk ), .RESET( and_out ), .Q( out2 ) );

endmodule
```

Signal “in1” is used both for clock and reset. Do not mix clock and reset lines to avoid problems w ith DFT. In such descriptions all lines to w hich clock signal is connected w ill be excluded from scanning.

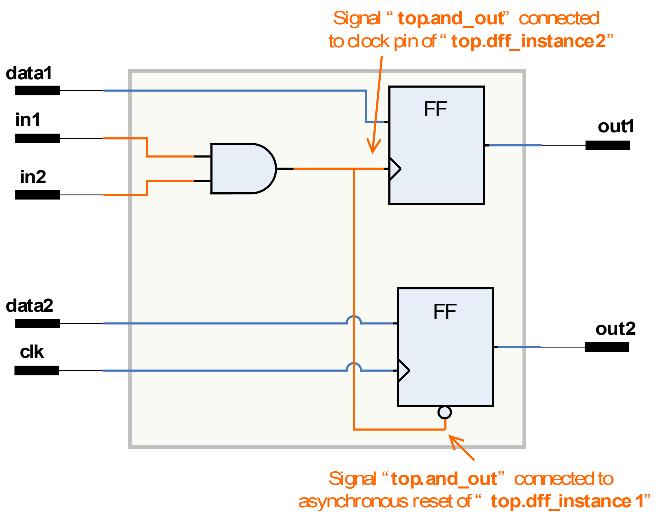
Signal “in2” is also used both for clock and reset in the same connections.

```
module dff( D, CLK, Q );

    input  D;
    input  CLK;

    output Q;
    reg    Q;

    always @( posedge CLK )

        Q <= D;

endmodule
```

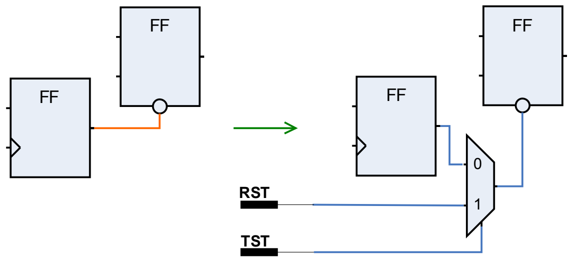
Connection to clock pin of flip-flop “Q” is detected.

```
module dff_r( D, CLK, RESET, Q );

    input  D;
    input  CLK;
    input  RESET;

    output Q;
    reg    Q;

    always @( posedge CLK or posedge RESET )

        if  ( RESET )

             Q <= 1'b0;

        else

             Q <= D;

endmodule
```

Connection to asynchronous control pin of flipflop “Q” is detected.


Signal " top.and_out" connected to clock pin of " top.dff_instance2"
data1
in1
in2
FF
out1
data2
clk
FF
out2
Signal "top.and_out" connected to asynchronous reset of " top.dff_instance 1"

STARC_VLOG 3.3.6.3

RULE NAME	**Do not connect the output of a FF directly to the asynchronous set or reset pin of a FF**	
MESSAGE	**The asynchronous control pin(s) of flip-flop "{DrivenFFName}" is driven by the output of another flip-flop. Insert selector to switch to a reset signal controlled directly from an external port to avoid malfunctions with DFT and ATPG tools.**	
	DETAIL	*Asynchronous {ControlName} pin is driven with the output of flip-flop "{DriverFFHierName}"*
PROBLEM DESCRIPTION	Problems with Automatic Test Pattern Generation (ATPG) tools occur when flip-flop clock pin is driven by output of another flip-flop. If a synchronized reset signal is necessary, prefer to insert the selector that will allow to switch to a reset signal that can be directly controlled from an external port of LSI (see the picture below) 	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans the design hierarchy for flip-flops that have output of another flip-flop supplied to their asynchronous control pins: – backward-propagation is performed from each asynchronous control pin of each flip-flop to define its driver (*See rule 1.4.3.2 for details regarding backward propagation*) – if driver is output of another flip-flop => violation is reported (main message points on process that infers driven flip-flop, whereas detail message points on assignment of signal that infers driver flip-flop) – following is list of possible controls (*{ControlName}*) for the "detail" violation message: – reset – set – set/reset	

EXAMPLE-1: [1] inverted output of first flip-flop is connected to asynchronous set of second flip-flop

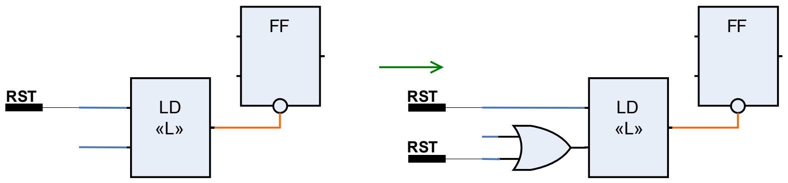
```
module top (CLK, D, SET, RESET, Q, Qn) ;

    input CLK;
    input D;
    input SET;
    input RESET;

    output Q;
    output Qn;

    wire CTRL_from_FF1_out;
    wire WIRE_from_FF1_out;

    dff U1  ( .CLK( CLK ), .D( D ), .SET( SET ), .RESET( RESET ),
              .Q( WIRE_from_FF1_out ), .Qn( CTRL_from_FF1_out ) );
    dff U2  ( .CLK( CLK ), .D(WIRE_from_FF1_out), .SET( CTRL_from_FF1_out ), .RESET( RESET ),
              .Q( Q ), .Qn(Qn) );
```

```
endmodule

module dff( D, CLK, SET, RESET, Q, Qn );

    input  D;
    input  CLK;
    input  SET;
    input  RESET;

    output Q;
    reg    Q;

    output Qn;


    always @( posedge CLK or posedge SET or negedge RESET )

        if( SET )
            Q <= 1'b1;
        else if( !RESET )
            Q <= 1'b0;
        else
            Q <= D;


    assign Qn = ~Q;

endmodule
```

Instance "top.U2". The asynchronous control pin(s) of flip-flop "Q" is driven by the output of another flip-flop. Insert selector to sw itch to a reset signal controlled directly from an external port to avoid malfunctions w ith DFT and ATPG tools.

Asynchronous set pin is driven w ith the output of flip-flop "top.U1.Q"

STARC_VLOG 3.3.6.4

RULE NAME	**Do not connect the output of a latch directly to the asynchronous set or reset pin of a FF**	
MESSAGE	**The asynchronous control pin(s) of flip-flop "{DrivenFFName}" is driven by the output of latch. Add logic enabling to place the latch in a through mode and control reset directly from an external port to avoid malfunctions with DFT and ATPG tools.**	
	DETAIL	*Asynchronous {ControlName} pin is driven with the output of flip-flop "{DriverFFHierName}"*
PROBLEM DESCRIPTION	Latch on a clock line will cause an errors in the DFT and ATPG tools. If such type of circuitry cannot be avoided, add special gate to place the latch in a through mode, as it shown on the image below: 	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker scans the design hierarchy for flip-flops that have output of latch supplied to their asynchronous control pins: – backward-propagation is performed from each asynchronous control pin of each flip-flop to define its driver (*See rule 1.4.3.2 for details regarding backward propagation*) – if driver is output of latch => violation is reported (main message points on process that infers driven flip-flop, whereas detail message points on assignment of signal that infers driver latch) – *see 3.3.6.3 for list of possible controls ({ControlName}) for the "detail" violation message*	

EXAMPLE-1: [1] latch output is directly connected to asynchronous set input of flip-flop => violation;

[2] note, that output of the same latch is connected to reset input of the same flip-flop, but there is no violation because backward propagation stops at logic gate 'or' (ORing of multiple asynchronous controls);

```
module dff_rs_async( D, CLK, RESET, SET, Q, Q1 );

    input  D;
    input  CLK;
    input  RESET;
    input  SET;

    output Q;
    reg    Q;

    output Q1;
    reg    Q1;

    wire   CTRL_from_FF_out;


    // output of LATCH "Q" is line "CTRL_from_FF_out"
    assign CTRL_from_FF_out = Q;

    // output of LATCH "Q" connected to set pin of flip-flop "Q1"
    always @( posedge CLK or posedge CTRL_from_FF_out or negedge RESET )
```

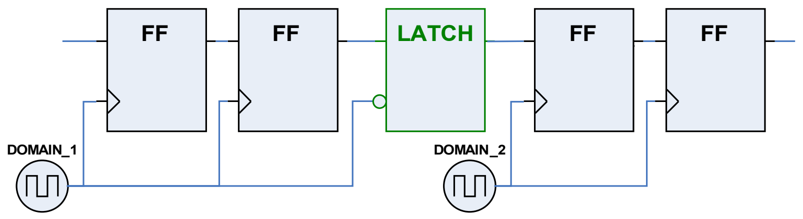
```
        if( CTRL_from_FF_out )
            Q1 <= 1'b1;
        else if( !RESET )
            Q1 <= 1'b0;
        else
            Q1 <= D;

    always @( CLK or SET or RESET )
        if( SET )
            Q <= 1'b1;
        else if( !RESET )
            Q <= 1'b0;
        // LATCH "Q" is inferred
        else if ( CLK )
            Q <= D;

endmodule
```

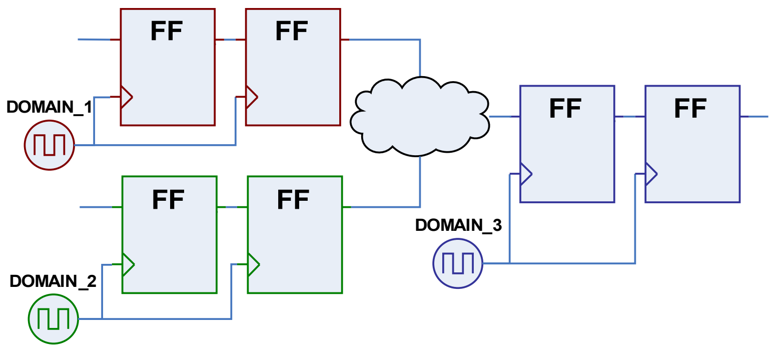
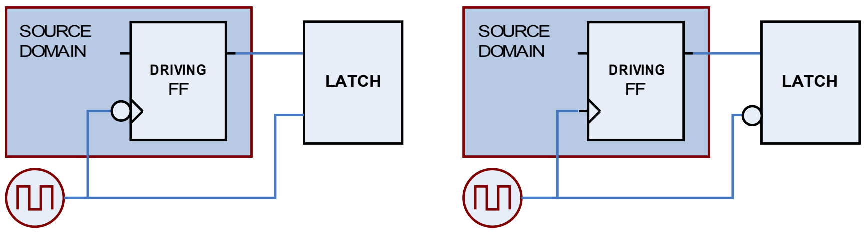
Instance "dff_rs_async". The asynchronous control pin(s) of flip-flop "Q1" is driven by the output of latch. Add logic enabling to place the latch in a through mode and control reset directly from an external port to avoid malfunctions w ith DFT and ATPG tools.

Asynchronous set pin is driven w ith the output of latch "dff_rs_async.Q"

3.3.7 Handling of different clocks

STARC_VLOG 3.3.7.2

RULE NAME		**Insert a latch with an inverted clock when transmitting between asynchronous clocks**
MESSAGE-1		**Data transfer without using a latch is detected. For structuring a single scan chain it is recommended to insert a latch with an inverted clock between the adjacent flip-flops that are in different clock domains.**
	DETAIL-1	*Data is sent by the FF "{FFName}" that belongs to the clock domain "{HierClockName}".*
	DETAIL-2	*Data is accepted by the FF "{FFName}" that belongs to the clock domain "{HierClockName}".*
MESSAGE-2		**Multiple clock domains transfer data through a single latch ("{LatchName}"). For structuring scan chains it is recommended to insert a latch with an inverted clock between the each pair of adjacent flip-flops that are in different clock domains.**
	DETAIL-1	*Data is sent by the FF "{FFName}" that belongs to the clock domain "{HierClockName}".*
	DETAIL-2	*Data is accepted by the FF "{FFName}" that belongs to the clock domain "{HierClockName}".*
MESSAGE-3		**Incorrect enable polarity is used to switch the latch "{LatchName}" to a transparent state. Polarity should be inverted relatively to the driving flip-flop from the source clock domain (for structuring a single scan chain it is recommended to insert a latch with an inverted clock between the adjacent flip-flops that are in different clock domains).**
	DETAIL-1	*Data is sent by the FF "{FFName}" that belongs to the clock domain "{HierClockName}".*
	DETAIL-2	*Data is accepted by the FF "{FFName}" that belongs to the clock domain "{HierClockName}".*
PROBLEM DESCRIPTION		It is important to handle multiple clock domains with care. An attention to clock skew is important: during the scan testing it is easy to meet setup-timing requirements because scan clock frequency is slow, whereas hold-time problems are common. Potential hold-time problems could be avoided by ensuring that a scan chain consists only of flip-flops from the same clock domain. If this is not feasible, latch with an inverted clock should be added between the adjacent flip-flops on a scan chain that are in different clock domains.  The data will be held during the high level of clocking signal and transmitted during the low level of clocking signal (transparent state of the latch). If skew value in the clock line is less than ½ of a clock cycle, then it is not necessary to ensure the hold.
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR		Checker scans interconnections between different asynchronous **clock domains** (see *1.5.1.1* for details about clock domains detection)***:***

RULE NAME	**Insert a latch with an inverted clock when transmitting between asynchronous clocks**
	– violations are issued for the following cases: – there is no latch and data transfer is performed from the single domain (as it is shown at the picture above) => message-1; – there is no latch and data transfer is performed from multiple domains => message-2;  FF FF DOMAIN_1 FF FF DOMAIN_2 FF FF DOMAIN_3 – there is a latch, but its edge is not inverted relatively to the edge of the driving flip-flop from the source domain => message-3 (see correct cases at the figure below);  SOURCE DOMAIN DRIVING FF LATCH SOURCE DOMAIN DRIVING FF LATCH Note: details are similar for all cases; also there are special detail messages to indicate origin clocks of source and target domains (see *1.5.1.1* for details)

EXAMPLE-1: [1] consider sample circuit at the picture below – latch ld_out is properly located between the asynchronous clock domains DOMAIN_1 and DOMAIN_2, but its enable input is active at the same clock phase as flip-flops from the DOMAIN_1 => violation (message-3);

[2] note that clocks are auto-detected.

```
module top ( clk1, clk2, data, out );

    input clk1, clk2, data;
    output reg out;

    reg ff_out, ld_out;

    always @( posedge clk1 )
        ff_out <= data;

    always @( clk1 or ff_out )
```

Data is sent by the FF "top.ff_out" that belongs to the clock domain "clk1".

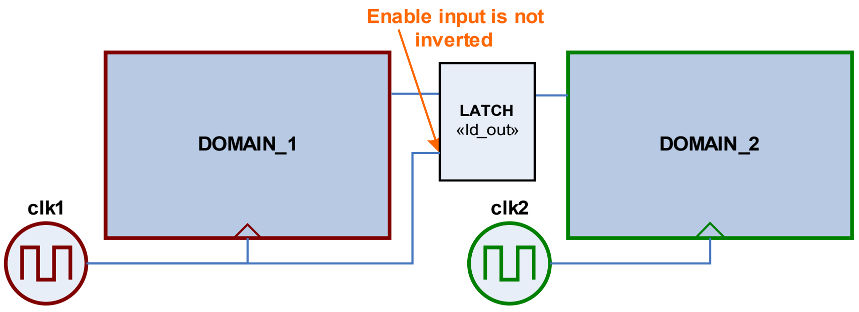
"clk1" is the origin clock of the source domain.

Data is accepted by the FF "top.out" that belongs to the clock domain "clk2".

"clk2" is the origin clock of the target domain.

```
        if ( clk1 )
            ld_out <= ff_out;

    always @( posedge clk2 )
        out <= ld_out;

endmodule
```

Instance "top". Incorrect enable polarity is used to sw itch the latch "ld_out" to a transparent state. Polarity should be inverted relatively to the driving flip-flop from the source clock domain (for structuring a single scan chain it is recommended to insert a latch w ith an inverted clock betw een the adjacent flip-flops that are in different clock domains).


Enable input is not inverted
LATCH
«ld_out»
DOMAIN_1
DOMAIN_2
clk1
clk2

3.3.8 DFT for tri-state circuits

STARC_VLOG 3.3.8.1

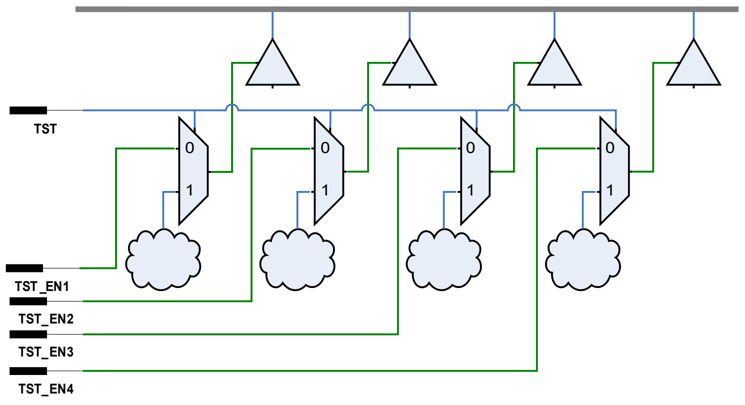
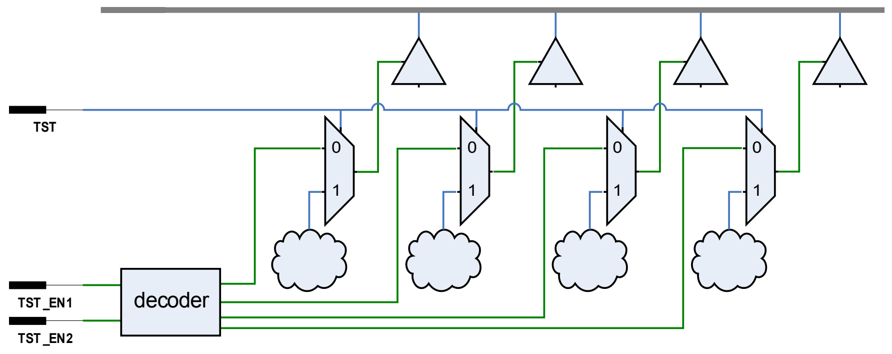
RULE NAME	**Tristate enable signals should be able to be fixed from an external input port**	
MESSAGE	**Tristate enable input is not directly controlled from external input port. The tristates must be able to be controlled directly from external input ports, otherwise ATPG tools will encounter problems.**	
PROBLEM DESCRIPTION	If tristate design takes DFT into consideration, it must enable direct control of tristate buffers from outside of the LSI (such description will be correct for ATPG tools).	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker scans the design hierarchy for tristates and verifies the signal that is mapped to an enable pin of each tristate that is detected: – this signal must be an external (*) signal for the design – (*) ***external*** *signal is such signal that can be directly controlled from an external port (it doesn't pass through any kind of logic except of buffers and inverters)* – if signal is not external => violation message is reported (message points on tristate signal assignment)	

EXAMPLE-1: [1] MUX switches between signals en1 and en2 to control tristate enable signal;
[2] internal signal (MUX out) is connected to the enable signal of tristate => violation;

```
module top( clk, sel, en1, en2, out1 );

    input clk, en1, en2;
    output out1;

    reg int_en, mux_out, ff_out;

    always @( sel, en1, en2 )
        case ( sel )
            1'b0    : int_en <= en1;
            1'b1    : int_en <= en2;
            default : int_en <= 1'bx;
        endcase

    always @( posedge clk )
        ff_out <= mux_out;

    assign out1 = int_en ? ff_out : 1'bz;

endmodule
```

Tristate enable input is not directly controlled from external input port. The tristates must be able to be controlled directly from external input ports, otherwise ATPG tools will encounter problems.

STARC_VLOG 3.3.8.2

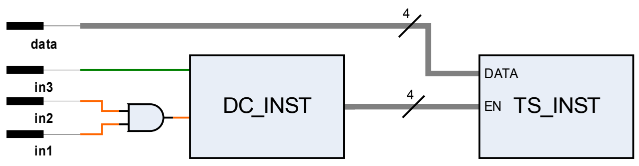
RULE NAME	**Tristate enable signals should be able to be controlled directly from the outside or should be controlled by a decoder that is controlled directly from the outside**
MESSAGE	**Tri-state "{ThreeStateName}" enable pin can not be controlled directly from the outside or can not be controlled by a decoder that is controlled directly from the outside. Such circuit structure is unsafe for scan shift during the scan test. It is recommended to insert a selector at the final output of the random logic to make it possible to select an external port or insert a decoder to reduce number of external pins.**
PROBLEM DESCRIPTION	Tristate design that takes DFT into consideration is, ideally, a design wherein all of the enable signals can be controlled individually from outside of the LSI (see picture below). With such circuitry, there will be no problems with ATPG tools.  In LSIs containing many tristate circuits, it is, as a practical matter, impossible to provide enable signals to all of the tristate elements from outside of the LSI. In such cases, one method that is used widely to decrease a number of required input pins is designs using decoders to control the tristate elements internally. This method is considered at the picture below. 
	LEVEL | RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans the design hierarchy for tri-state and verify the line that is mapped to the enable control pins: – allowed cases: – tri-state enable control pin is driven by an external signal (see 3.3.8.1 for details) – tri-state enable control pin is driven by an ***abstract decoder*** – all inputs of abstract decoder must be connected directly(*) from an external port – *(*): direct connection in this case means that only buffers and inverters are allowed – MUX inputs are not allowed)*

RULE NAME	**Tristate enable signals should be able to be controlled directly from the outside or should be controlled by a decoder that is controlled directly from the outside**
	– tri-state enable control also must be connected directly(**) to an abstract decoder output – *(**): direct connection in this case means that only buffers and inverters are allowed – MUX* inputs are allowed) – *note:* cascade logic is possible for decoders – such cases are considered – all another cases are not allowed and violate the rule. *Note: **abstract decoder** scheme means scheme which consist of MUX that has constants mapped to inputs; so, the following definitions are set:* – *MUX select pins are inputs of abstract decoder* – *MUX output pins are outputs of abstract decoder* – *MUX input pins are mapped nowhere, they are a constants.*

EXAMPLE-1: [1] tri-state enable signals are controlled by the decoder;

[2] one of the decoder inputs is controlled from external port through “and” gate => violation.

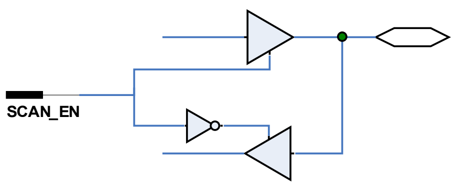
Note: violation is expected per every bit of vector Q, only one message is mentioned here to minimize number of callouts. The only difference between the messages is a name of tri-state enable pin.



```
module top( data, in1, in2, in3, out1 );
    input  [3:0] data;
    input  in1;
    input  in2;
    input  in3;

    output [3:0] out1;

    wire         and_out;
    wire   [3:0] decoder_out;

    assign and_out = in1 & in2;

    decoder2to4 DC_INST ( .in ( {in3, and_out} ), .out ( decoder_out ) );

    tristate   TS_INST ( .EN ( decoder_out ), .DATA( data ), .Q ( out1 ) );

endmodule


module decoder2to4( in, out );

    input  [1:0] in;

    output [3:0] out;
    reg    [3:0] out;


    always @( in )
        case ( in )
            2'b00    : out <= 4'b0001;
            2'b01    : out <= 4'b0010;
            2'b10    : out <= 4'b0100;
            2'b11    : out <= 4'b1000;
            default : out <= 4'b0000;
        endcase

endmodule


module tristate( DATA, EN, Q );

    input   [3:0] DATA;
    input   [3:0] EN;

    output [3:0] Q;

    assign Q = ( EN ) ? DATA : 4'bz;

endmodule
```

top.TS_INST. Tri-state "Q[0]" enable pin can not be controlled directly from the outside or can not be controlled by a decoder that is controlled directly from the outside. Such circuit structure is unsafe for scan shift during the scan test. It is recommended to insert a selector at the final output of the random logic to make it possible to select an external port or insert a decoder to reduce number of external pins.

STARC_VLOG 3.3.8.3

RULE NAME	**External bidirectional pins should be set during the scan shift**	
MESSAGE-1	**Non-tri-state logic is detected on bidirectional bus "{PortName}". Connect tri-states to inout port to set the direction of the port during the scan shift.**	
	DETAIL-1	*Non-tri-state element drives the bidirectional bus.*
	DETAIL2	*Non-tri-state element is driven by the bidirectional bus.*
MESSAGE-2	**Control inputs of tri-states on bidirectional bus "{PortName}" are driven incorrectly. Use a single external port and its inversion for tri-states control inputs to set the inout port to input or output during the scan shift.**	
PROBLEM DESCRIPTION	Bidirectional ports should drive tri-state for output mode and should be driven by tri-state for input mode. To avoid situations when value is written and is read from the port at the same time, tri-state enable signal is used in the way shown at the picture. For DFT there should be a possibility to set the external bidirectional ports of the LSI as inputs during the scan shift, and the mode should be controlled from an external port. 	
	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scan the current module and for each port declared as *inout* checks its actual mode (in case of vector – every bit is checked separately): – if the actual mode is 'unused' or 'input' => no violations – if the actual mode is 'inout': – analyzes all local drivers of the 'inout' port and all lines that are driven by the 'inout' port: – if one or more local drivers is not a tri-state or if 'inout' port drives non-tri-state => violation (message-1 + detail-1 *(per each non-tristate driver object)* / detail-2 *(per each non-tristate driven object)*); – in any other case checker collects: – all enable inputs of tristates that drive 'inout' ports (OUTBUF) – all enable inputs of tristates which are driven (INBUF) by 'inout' port – OUTBUF and INBUF must be driven by the same external port (CONTROL) inversely, i.e. this signal should satisfy a following condition: – if CONTROL drives OUTBUF, !CONTROL should drive INBUF; – if !CONTROL drives OUTBUF, CONTROL should drive INBUF; – otherwise => violation (message-2) and the analysis is stopped (for the currently analyzed 'inout' port).	

EXAMPLE-1: [1] actual mode of *inout* port is 'inout';

[2] local driver is not a tri-state and 'inout' port drives non-tri-state => violation (message-1 + detail-1/detail-2).

```
module top (in, out, io);

    input in;
    output out;
    inout io;

    assign io = in;

    assign out = io;

endmodule
```

Non-tri-state logic is detected on bidirectional bus "io". Connect tri-states to inout port to set the direction of the port during the scan shift.

Non-tri-state element is driven by the bidirectional bus.

Non-tri-state element drives the bidirectional bus.

EXAMPLE-2: [1] actual mode of *inout* port is 'inout';

[2] local driver is a tri-state and 'inout' port drives tri-state;

[3] OUTBUF and INBUF are driven by the same signal (“enb”);

[4] safety condition is not achieved (“enb” drives both INBUF and OUTBUF)=> violation (message-2).

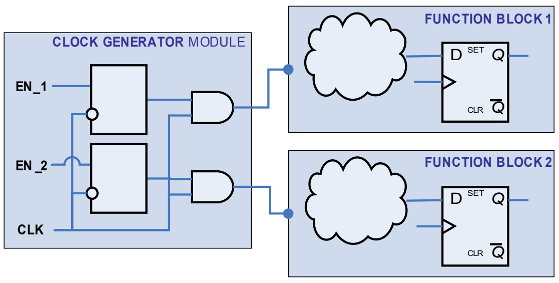
```
module top (in1, enb, out, io);

    input in,enb;
    output out;
    inout io;

    assign io = enb1 ? 1'bz : in;

    assign out = enb1 ? 1'bz : io;

endmodule
```

Control inputs of tri-states on bidirectional bus "io" are driven incorrectly. Use a single external port and its inversion for tri-states control inputs to set the inout port to input or output during the scan shift.

3.4 Low-power design

3.4.1 Low-power design using gated clocks

STARC_VLOG 3.4.1.1

RULE NAME	**In the design of standard ASICs, gated clocks can be used only in the top level.**	
MESSAGE-1	**Global clock "{ClkName}" is gated locally. In the design of standard ASICs, the gated clocks should only be located in the clock generator module at the top level so as to gate the clocks to each functional block.**	
PROBLEM DESCRIPTION	One of the most effective approaches to reduce power dissipation is to use gated clocks or divided clocks. It means clocks could be stopped when they are not needed. Following figure illustrates the most common method of clock gating – through the use of a latch and a gate:  CLOCK GENERATOR MODULE EN_1 EN_2 CLK FUNCTION BLOCK 1 FUNCTION BLOCK 2 D SET Q CLR Q̄ When the CLK is in its low phase, the latch is transparent (propagation of the control input which actually indicates whether to gate the clock or not). So, if the control input is high, the output of the latch is high during the low phase (it remains in this state until the next CLK low phase). In such situation, AND gate is enabled when the posedge CLK arrives. In the design of standard ASICs, the gated clocks should only be located in the clock generator module at the top level so as to gate the clocks to each functional block. When gated clocks are used at local level, there will be many incorrect cells in the clock tree (when the clock tree is generated by synthesis tool). Thus, fixing clock skew will be more difficult. Moreover, when gated clocks are used only in the clock generator module, it becomes easier to insert a DFT circuits.	
	LEVEL	RULE
CHECKER BEHAVIOR	Checker detects each gated clocks in the design: – violation is issued in case if clock is gated locally (correct case – separate module at top level);	

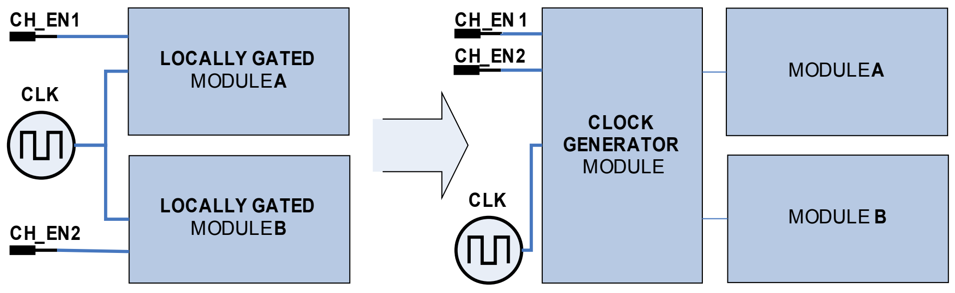
EXAMPLE-1: [1] sample contains two locally gated submodules => violations;

[2] the figure below illustrates an approach to fix this problem – gated clocks are separated from another logic and located separately in the clock generator module.

```
module cmp( CLK, RST, CH_EN1, CH_EN2, ARG_A, ARG_B, RES );

    input        ARG_A, ARG_B;
    input        CLK, RST, CH_EN1, CH_EN2;
    output [1:0] RES;

    wire    wA, wB;
    assign wA = ARG_A & ARG_B;
    assign wB = ARG_A | ARG_B;

    gated_ff GATED_FF_CH_A
    (
        .RST_X( RST    ),
        .CLK  ( CLK    ),
        .DI   ( wA     ),
        .EN   ( CH_EN1 ),
        .DO   ( RES[0] )
    );

    gated_ff GATED_FF_CH_B
    (
        .RST_X( RST    ),
        .CLK  ( CLK    ),
        .DI   ( wB     ),
        .EN   ( CH_EN2 ),
        .DO   ( RES[1] )
    );

endmodule

module gated_ff( DI, RST_X, EN, CLK, DO );

    input  DI;
    input  RST_X, EN, CLK;
    output DO;
    reg    DO;

    reg    d_latch;
    wire   gated_clk;

    always @( CLK, EN ) begin
        if( CLK )
            d_latch = EN;
    end

    assign gated_clk = d_latch & CLK;

    always @( posedge gated_clk or negedge RST_X ) begin
        if( !RST_X )
            DO <= 1'b0;
        else
            DO <= DI;
    end

endmodule
```

Instance "cmp.GATED_FF_CH_A". Global clock "cmp.CLK" is gated locally. In the design of standard ASICs, the gated clocks should only be located in the clock generator module at the top level so as to gate the clocks to each functional block.

Instance "cmp.GATED_FF_CH_B". Global clock "cmp.CLK" is gated locally. In the design of standard ASICs, the gated clocks should only be located in the clock generator module at the top level so as to gate the clocks to each functional block.



3.5 Source codes and design data management

3.5.3 Define necessary information for file headers

STARC_VLOG 3.5.3.3

RULE NAME	**Standardize file headers**
MESSAGE-1	**Token "{TokenRegExp}" ({Token}) is not expected according to the grammar defined by the rule configuration. Possible token(s) to use: {ExpectedTokenRegExpAndToken}.**
MESSAGE-2	**Token "{TokenRegExp}" ({Token}) is not expected. End of file header is reached according to the current grammar defined by the current rule configuration.**
MESSAGE-3	**Unexpected end of file header. Possible token(s): {ExpectedTokenRegExpAndToken} according to the grammar defined by the rule configuration.**
MESSAGE-4	**File header is missed. It is recommended to add standard header for each file to improve readability.**
MESSAGE-5 [ERROR]	**Incorrect grammar description. There is a choice between values “{Values}” and empty set for nonterminal symbol “{SymbolName}”. Use '[ ]' to define optional values.**
MESSAGE-6 [ERROR]	**Incorrect grammar description. Symbol “{SymbolName}” is not defined as neither nonterminal symbol nor token.**
MESSAGE-7 [ERROR]	**Incorrect grammar description. Left recursion is detected in description of nonterminal symbol “{SymbolName}”. Use '{ }' to define repeated values.**
PROBLEM DESCRIPTION	Necessary information should be defined within the file header. Define the file, and include when the file was modified by whom and for what purpose it was created. Following this recommendation makes it possible to improve readability when the description is reused by designers other than the original designer. In the example, the file header, file names, circuit type, function, modifier, originator, and revision are all defined. Any other sections should also be added when necessary. Readability decreases if the file headers’ formats differ among designers, so all designers should follow standardized header format. Example: (see code below)
LEVEL	RECOMMENDATION 1

```
/*-----------------------------------------------------
--                                                   --
-- FILE_NAME : my_circuit.v                          --
--                                                   --
-- TYPE : circuit                                    --
--                                                   --
-- FUNCTION : data decoding with Gray code           --
--                                                   --
-- AUTHOR : First_Name Last_Name                     --
--                                                   --
-- EDIT : Bob                                        --
--                                                   --
-- REV_DATE : 1.0 01/02/08                           --
--            1.1 02/02/08                           --
--                                                   --
-- Some detail text about the function of the        --
-- circuit, data flow, data transformation, etc.     --
--                                                   --
-----------------------------------------------------*/
```

RULE NAME	**Standardize file headers**
CHECKER BEHAVIOR	Checker detect file header(*): – if file header does not satisfy standard defined in the configuration file or grammar description does not specified properly(**) => violation: *(*) **file header** is a continuous sequence of commented lines until first line with code is reached. Considering of empty lines is tune by CONSIDER_END_LINE parameter ("0" means not to consider, "1" – to consider; default value is "0").* *(**)Creation of new grammar should satisfy next rules:* – *there should be the main nonterminal symbol with the rule of its representation;* – *each nonterminal symbol (from the rule for main nonterminal symbol) must have grammatical rules showing how it is made out of simpler constructs;* – *the simplest nonterminal symbol should be represented with terminal one (token), if some of the nonterminal symbol could not be represented with token, then the grammar is not full, and it is impossible to provide correct analysis.* – if token unexpected by the rule is specified => message-1; – if file header ends according to the grammar but comments continue => message-2; – if comments end but file header continues according to the grammar => message-3; – if there is no file header => message-4; – if choice between some object and empty set is specified within configuration => message-5; – if some symbol is not defined within configuration => message-6; – if left recursion is used within configuration => message-7. Note-1: BNF points (according to the standard ISO/IEC 14977 : 1996(E)): – [– start-option-symbol end-option-symbol –] – { – start-repeat-symbol end-repeat-symbol – } – | – definition-separator-symbol – # – instead of second-quote-symbol " to define terminal symbols. Note-2: in should be mentioned that definition separator symbol (|) can not be used inside of option ([ ]) or repeat ({ }) constructs. Note-3: grammar is defined by with two parameters GRAMMAR and TOKENS. Default values of them are following: GRAMMAR = ["content ::= LINE content_item LINE", "content_item ::= empty_line file_name_string [{additional_string}] empty_line author_string [{additional_string}] empty_line type_string [{additional_string}] empty_line function_string [{additional_string}] empty_line edit_string [{additional_string}] empty_line rev_date_string [{additional_string}] empty_line [{additional_string} empty_line]", "file_name_string ::= MARGIN FILE_NAME SEPARATION_CHAR STRING MARGIN", "author_string ::= MARGIN AUTHOR SEPARATION_CHAR STRING MARGIN",

RULE NAME	**Standardize file headers**
	“type_string ::= MARGIN TYPE_NAME SEPARATION_CHAR STRING MARGIN”, “function_string ::= MARGIN FUNCTION_NAME SEPARATION_CHAR STRING MARGIN”, “edit_string ::= MARGIN EDIT_NAME SEPARATION_CHAR STRING MARGIN”, “rev_date_string ::= MARGIN REV_DATE_NAME SEPARATION_CHAR STRING MARGIN”, “empty_line ::= MARGIN MARGIN”, “additional_string::= MARGIN STRING MARGIN”] TOKENS = [“FILE_NAME ::= #FILE NAME#”, “AUTHOR ::= #AUTHOR#”, “TYPE_NAME ::= #TYPE#”, “FUNCTION_NAME ::= #FUNCTION#”, “EDIT_NAME ::= #EDIT#”, “REV_DATE_NAME ::= #REV, DATE#”, “INE ::= #-{3,}#”, “SEPARATION_CHAR ::= #:#”, “MARGIN ::= #--#”] Note-4: IDENTIFIER, END_LINE – is a hard coded tokens, they cannot be changed by user. END_LINE – is processed if parameter CONSIDER_END_LINE is active (is equal to '1').

EXAMPLE-1: [1] default configuration is used;
[2] field FUNCTION is missed from header => violation (message-1).

```
/*-----------------------------------------------------
--                                                   --
-- FILE_NAME : clk_gen.v                             --
--                                                   --
-- AUTHOR  : Robert                                  --
--                                                   --
-- TYPE : CIRCUIT                                    --
--                                                   --
-- EDIT : Bob                                        --
--                                                   --
-- REV_DATE : 1.0 04/01/08                           --
--            1.1 02/02/08                           --
--                                                   --
-- clock frequency is defined by parameter           --
-- CLK_PERIOD                                        --
--                                                   --
-----------------------------------------------------*/

module clk_gen (...);
    ...
endmodule
```

Unexpected token: "EDIT" (EDIT_NAME). Expected token: "FUNCTION" (FUNCTION_NAME). Standardize the format of file header to increase the readability of code.

EXAMPLE-2: [1] default configuration is used;
[2] file header ends according to the grammar but commented lines continue => violation (message-2);
Note: empty line between multi-line and one-line comments is skipped.

```
/*-----------------------------------------------------
--                                                   --
-- FILE_NAME : clk_gen.v                             --
--                                                   --
-- AUTHOR  : Robert                                     --
--                                                      --
-- TYPE : CIRCUIT                                       --
--                                                      --
-- FUNCTION : generate clock signal                     --
--                                                      --
-- EDIT : Bob                                           --
--                                                      --
-- REV_DATE : 1.0 04/01/08                              --
--            1.1 02/02/08                              --
--                                                      --
-- clock frequency is defined by parameter              --
-- CLK_PERIOD                                           --
--                                                      --
-----------------------------------------------------*/

//top module declaration
module clk_gen (...);
    ...
endmodule
```

Unexpected token: "any_ASCII_characters" (IDENTIFIER). End of file header according to the current grammar is reached. Standardize the format of file header to increase the readability of code.

EXAMPLE-3: [1] custom grammar description is used;
[2] header satisfies requirements described with grammar => no violation.

```
#configuration
RULE_CFG STARC_VLOG.3.5.3.3
{
    GRAMMAR = [
            "content ::= LINE content_item LINE",
            "content_item ::= cvs_tag LINE
                                   model_name_block LINE
                                   owner_block LINE
                                   file_description",
            "cvs_tag ::= FIRST_SYMBOL ID_TAG string",
            "model_name_block ::= {FIRST_SYMBOL string}",
            "owner_block ::= {empty_line} {FIRST_SYMBOL string [{empty_line}]}
                                 copyright_item {empty_line}",
            "copyright_item ::= LINE FIRST_SYMBOL [{LINE}] COPYRIGHT company [LINE]
                                         FIRST_SYMBOL [LINE] ALL RIGHTS RESERVED [LINE]
                                    [FIRST_SYMBOL LINE] {empty_line}",
            "company ::= string",
            "file_description ::= filename_str empty_line description_str empty_line",
            "filename_str ::= FIRST_SYMBOL FILE_NAME string",
            "description_str ::= FIRST_SYMBOL DESCRIPTION string",
            "string ::= {IDENTIFIER}",
            "empty_line ::= FIRST_SYMBOL"

    ]

    TOKENS = [
            "ID_TAG ::= #\$Id:#",
            "FILE_NAME ::= #Filename:#",
            "DESCRIPTION ::= #Description:#",
            "COPYRIGHT ::= #Copyright#",
            "ALL ::= #All#",
            "RIGHTS ::= #rights#",
            "RESERVED ::= #reserved.#",
            "LINE ::= #\*{2,}#",
            "FIRST_SYMBOL ::= #\*#"
    ]
}
```

```
/************************************************************************
 * $Id: multiplier.v,v 1.2 2008/01/01 00:00:00 bob Exp $
 ************************************************************************
 * Multiplier - Verilog Behavioural Model
 ************************************************************************
 *
 *
 * This File is owned and controlled by My_vendor and must be used solely
 * for design, simulation, implementation and creation of design files
 * limited to My_vendor devices or technologies.
 *
 *
 *         ******************************
 *         ** Copyright My_Vendor, Inc. **
 *         ** All rights reserved.      **
 *         ******************************
 *
 *
 ***********************************************************************
 * Filename:    multiplier.v
 *
 * Description: The Verilog behavioural model for the multiplier
 *
 ***********************************************************************
 */

`timescale 1ns/10ps

...
```

EXAMPLE-4: [1] custom grammar description is used;
[2] token TOKEN_B is used within grammar description, but it is not defined => error (message-6).

```
#configuration
RULE_CFG STARC_VLOG.3.5.3.3
{
    GRAMMAR = [
            "content ::= LINE content_item LINE",
            "content_item ::= TOKEN_A TOKEN_B TOKEN_C"
    ]

    TOKENS = [
             "TOKEN_A ::= #string_A#",
             "TOKEN_C ::= #string_B#",
             "LINE ::= #\*{2,}#",
    ]
}
```

Incorrect grammar description. Symbol "TOKEN_B" is not defined as neither nonterminal symbol nor token.

3.5.6 Use comments often

STARC_VLOG 3.5.6.3

RULE NAME	**Describe the I/O ports and declarations in one line and always add comments**	
MESSAGE-1	**Detected {ObjectType}s declaration that are not followed by a comment. It is recommended to add comments in the same line per understanding and readability of the code.**	
	DETAIL-1	*{DirectionType} port "{PortName}" does not have corresponding comment in the same line.*
	DETAIL-2	*{SignalVariable} "{ObjectName}" does not have corresponding comment in the same line.*
MESSAGE-2	**Some objects are declared in the single line. It is recommended to write the declaration of each port in a separate line and follow it with the corresponding comments in the same line per each declaration in order to improve understanding and readability of the code.**	
	DETAIL	*{ObjectTypeObjectNameList} are declared in the single line.*
MESSAGE-3	**Detected comment(s) written before declaration of {ObjectType}. It is recommended to declare {ObjectType} followed with comments.**	
	DETAIL	*Comment is written before declaration. Write each declaration in single line followed by the comment.*
PROBLEM DESCRIPTION	The frequent use of comments improves readability of the source code. As a result, it becomes easier to understand and maintain the source code, and this in turn leads to improved reuse efficiency. Recommended volume of comments is generally should to be about 20 to 40% of the source code. Comments indicating the purpose and functionality should be added to operators and statements in the description. Moreover, the I/O port or internal register declarations should be described in one line for each signal or register, and comments should always be added.	
	LEVEL	RECOMMENDATION 2
CHECKER BEHAVIOR	Checker detects port declarations in each module: – if the comment is written before declaration => violation (message-3); – else if ports are declared in separate lines and some declaration is not followed by a comments => violation (message -1 + detail-1); – else if ports are declared within the same line => violation (message-2 + detail). Checker detects signal/variable declarations of any type within module scope: – if the comment is written before declaration => violation (message-3); – else if signals are declared in separate lines and some declaration is not followed by a comments => violation (message -1 + detail-2); – else if signals are declared within the same line => violation (message-2 + detail). Note-1: an empty comment is treated as it is no comment at all Note-2: port declarations are verified both within module declaration and module item.	

EXAMPLE-1: [1] all of the ports are declared within the same line => violation (message-2 + detail).

```
module dff (input clk, rst, d, output q); // module interface declaration

    ...

endmodule
```

Some objects are declared in the single line. It is recommended to w rite the declaration of each port in a separate line and follow it w ith the corresponding comments in the same line per each declaration in order to improve understanding and readability of the code.

EXAMPLE-2: [1] two ports are declared in separate lines and both declaration are not followed by a comments => violation (message -1 + detail-1);

[2] four ports are declared within the same line => violation (message-2 + detail).

```
module  (ctrl, d1, d2, q1, q2);

    input [1:0] ctrl;
    input d1,d2;

    output q1,q2; reg tmp;

    ...

endmodule
```

Detected ports declaration that are not follow ed by a comment. It is recommended to add comments in the same line per understanding and readability of the code.

Input port "ctrl" does have corresponding comment in the same line.

Some objects are declared in the single line. It is recommended to w rite the declaration of each port in a separate line and follow it w ith the corresponding comments in the same line per each declaration in order to improve understanding and readability of the code.

Port(s) "d1, d2, d3, d4" are declared in the single line.

Port(s) "q1, q2", variable(s) "tmp" are declared in the single line.

EXAMPLE-3: [1] comment is written before nets declaration => violation (message-3);

[2] generate loop variable is declared followed by a comment => no violation.

```
module top ( ... );

/*dffs outputs */ net dff1_out, dff2_out;

                        genvar i; //generate loop variable

                        ...
endmodule
```

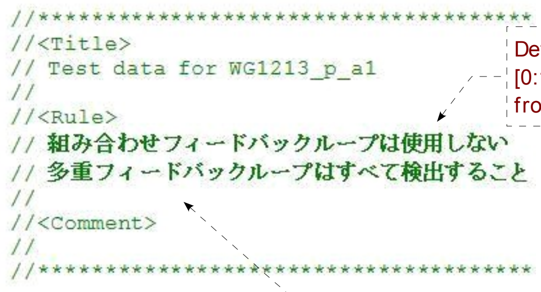
Detected comment(s) w ritten before declaration of port(s). It is recommended to declare port(s) follow ed w ith comments.

Comment is w ritten before declaration. Write each declaration in single line follow ed by the comment.

STARC_VLOG 3.5.6.4

RULE NAME	**Provide the comments in English as much as possible**
MESSAGE	**Detected character in a comment that has ASCII code out of range [0:127]. It is recommended to provide comments using characters from basic ASCII table.**
PROBLEM DESCRIPTION	It is preferable to provide comments in English, if possible, because design resources may be used internationally and English is the most popular language in the area of high technologies. Also, there are many EDA tools that do not support proper operation by source codes using languages other than English. However, if it is hard to provide comments in English, sometimes the comments become inadequate and the number of comments decreases. It is acceptable to write the comments in a language other than English if writing the comments in English is too difficult. Always be sure that detailed comments are added to the greatest extent possible. If possible, it is ideal to create tools that automatically delete comments in languages other than English before using any EDA tool.
	LEVEL — RECOMMENDATION 2
CHECKER BEHAVIOR	Checker scans source file for comments: – if there is at least one character that has ASCII code out of range [0:127] in current comment => violation

EXAMPLE-1: [1] two commented strings have characters which ASCII code out of range [0:127] => two violation



```
//****************************************
//<Title>
// Test data for WG1213_p_a1
//
//<Rule>
// 組み合わせフィードバックループは使用しない
// 多重フィードバックループはすべて検出すること
//
//<Comment>
//
//****************************************
```

Detected character a comment that has ASCII code out of range [0:127]. It is recommended to provide comments using characters from basic ASCII table.

Detected character a comment that has ASCII code out of range [0:127]. It is recommended to provide comments using characters from basic ASCII table.

STARC_VLOG 3.5.6.7

RULE NAME	**Comments should start with “//”**	
MESSAGE	**It is recommended to use single line comments starting with “//”.**	

PROBLEM DESCRIPTION	The frequent use of comments improves readability of the source code. As a result, it becomes easier to understand and maintain the source code, and this in turn leads to improved reuse efficiency. Recommended volume of comments is generally should to be about 20 to 40% of the source code.
	At a minimum, insert comments in all I/O signal and register signal names names (see 3.5.6.3). It is best to comment each meaningful line in 'always' blocks, 'assign' statements, sub-programs and other constructs, adding a comment just next to the line. In all the cases comment should be applied as one line comment and start with “//”.
	`always @( posedge CLK or posedge RST) //FSM clear after reset or next state`
	`    begin`
	`        if (RST == 1'b1)                //reset condition, global reset`
	`            STATE <= FIRST_STATE;       //initial of FSM variable`
	`        else                            //FSM state moves at CLK positive edge`
	`            STATE <= NEXT_STATE;        //moves next state`
	`    end`	

	LEVEL	RECOMMENDATION 3
CHECKER BEHAVIOR	Checker scans source file for comments: – if multi-line comment detected =>violation exception: multi-line comments within file header are skipped (see 3.5.3.3)	

EXAMPLE-1: [1] multi-line comment is used => violation

```
always @ ( C or D )
    /*latch inference*/
    if ( C )
        Q = D;
```

It is recommended to use single line comments starting w ith “//”.

Chapter 4 Verification Techniques

4.1 Test bench description

4.1.2 Use basic test vector descriptions

STARC_VLOG 4.1.2.3

RULE NAME	**The number of lines in fork-join should be a maximum of 5**	
MESSAGE	**'Fork-join' block has 'fork-join' contents to a maximum of {MAX_LINES_RECOMMENDED} concurrent branches to avoid simulation speed slowdown.**	
PROBLEM DESCRIPTION	'fork-join' statement is a syntax that supports simultaneous execution. When 'fork-join' is used effectively, improved flexibility in description is possible and this is convenient. However, 'fork-join' also slows down simulation speed. Moreover, when there are too many assign statements in 'fork-join' the readability declines. So it is recommended to limit the fork-join contents to a maximum of five execution lines.	
	LEVEL	RECOMMENDATION 3

CHECKER BEHAVIOR	Checker counts number of concurrent branches in fork-join blocks:
	– if count of execution lines is greater than MAX_LINES_RECOMMENDED => violation
	Note-1: concurrent line is top-level (executed in parallel) statement in fork-join block:
	`fork`
	`    if(...) begin // execution line #1`
	`        ...`
	`    end`
	`    a = b; c = d; // execution lines #2, #3`
	`join`
	Note-2: value of MAX_LINES_RECOMMENDED parameter is defined in configuration file (5 is default value).	

EXAMPLE-1: [1] 'fork-join' block contains 4 execution lines which is greater than MAX_LINES_RECOMMENDED parameter value => violation.

Note: MAX_LINES_RECOMMENDED parameter value is set to 3 to simplify the example.

'Fork-join' block has 'fork-join' contents to a maximum of 3 concurrent branches to avoid simulation speed slow dow n.

```
always @( ... ) begin
    fork
        case ( sel )            //execution line #1
            1'b0    : ...;
            1'b1    : ...;
            default : ...;
        endcase
        fork                    //execution line #2
            ...
        join
        data1 <= ~tmp1; data2 <= ctrl ? res1 : res2; //execution lines #3 and #4
    join
end
```

4.1.4 Avoid assigning from multiple *initial constructs* (different from VHDL)

STARC_VLOG 4.1.4.1

RULE NAME	**Avoid assigning from multiple initial constructs to the same signal (Verilog only)**	
MESSAGE-1	**Signal "{ObjectName}" is assigned in {AlwaysCount} 'always' constructs. Avoid assignments to the same signal from multiple 'always' or 'initial' constructs.**	
	DETAIL-1	*{AssignmentsCount} assignment(s) to signal "{ObjectName}" in this description block*
MESSAGE-2	**Signal "{ObjectName}" is assigned in {InitialCount} 'initial' constructs. Avoid assignments to the same signal from multiple 'always' or 'initial' constructs.**	
	DETAIL-1	*{AssignmentsCount} assignment(s) to signal "{ObjectName}" in this description block*

PROBLEM DESCRIPTION	It is unsafe to assign value to the same signal from multiple 'initial' constructs at the same time – there is no guarantee what value will be assigned (it depends on simulator being used). Consider following description:
	`initial begin`
	`   SIM = 1;`
	`   #(CLK_PERIOD * 512) SIM = 2;`
	`end`
	`initial begin`
	`   #(CLK_PERIOD * 10);`
	`   #(LOAD_DELAY);`
	`   for( i = 0; i < STIMULUS_COUNT; i = i + 1 )`
	`      #(A2B_DOMAIN) SIM = SIM * 2;`
	`   end`
	`   #(CLK_PERIOD * 77 );`
	`end`
	It is not clear in what order values are assigned to "SIM". Moreover, racing problem tends to occur (there could be multiple assignment simultaneously). Avoid assignments to the same signal either from multiple 'initial' or 'always' constructs.	

	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker collects all signals being assigned in the current module: – it is allowed to assign signal in the single 'initial' or 'always' construct (task statements are taken into account) – if signal is assigned from multiple 'initial' or 'always' constructs => violation (main message (1, 2) – on signal declaration, detail (1) – on each construct that assigns the signal): – message-1: if signal is assigned in multiple 'always' constructs – message-2: if signal is assigned in multiple 'initial' constructs	

EXAMPLE-1: [1] signal "Y" is assigned from the multiple 'always' and 'initial' constructs => violation (message-3)

```
reg Y;
...
always @( A or B or C ) begin
     Y = A & B & C;
     #(PATH_DELAY);
     Y = A | B | C;
end

always begin
```

Signal "Y" is assigned in 2 'alw ays' constructs. Avoid assignments to the same signal from multiple 'alw ays' or 'initial' constructs.

2 assignment(s) to signal "Y" in this description block

```
     #(CLK_PERIOD * 10);
     Y = A | B;
end
```

1 assignment(s) to signal "Y" in this description block

STARC_VLOG 4.1.4.2

RULE NAME	**Define one signal using one description block (Verilog only)**	
MESSAGE	**Description block assigns {SigCount} signals. Defining one signal inside one description block makes the pattern definition easier to comprehend.**	
	DETAIL	*Signal "{SignalName}" is assigned in this description block*
PROBLEM DESCRIPTION	Defining multiple signals inside one description block tends to unsafe (racing problem tends to occur) and hard to comprehend descriptions. It is recommended to define one signal per one description block.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans 'initial' and 'always' description blocks: – if more than one signal are assigned into one block => violation Note-1: different bits of the same vector are treated as the same signal Note-2: if there is any task enabled => checker verifies this tasks assigns same signal as description block	

EXAMPLE-1: [1] two different signals assigned in one 'always' description block => violation

```
always @( A or B or C ) begin
     Y1 = A || B || C;
     Y2 = A && B && C;
end
```

Description block assigns 2 signals. Defining one signal inside one description block makes the pattern definition easier to comprehend.

Signal "Y1" is assigned in this description block

Signal "Y2" is assigned in this description block

EXAMPLE-2: [1] different bits of the same signal are assigned in one 'initial' description block => no violation

```
initial begin
     Y[0] = A || B || C;
     Y[1] = A && B && C;
end
```

EXAMPLE-3: [1] two different signals assigned in one 'initial' description block => violation;
[2] note, that one of signals is assigned as output of the 'task' statement;

```
task sum( input  a, input  b, output res );
     res = a + b;
endtask

initial begin
     Y1 = A + B - C;
     #( CLK_PERIOD * 5 );
     sum( A, B, Y2 );
end
```

Description block assigns 2 signals. Defining one signal inside one description block makes the pattern definition easier to comprehend.

Signal "Y1" is assigned in this description block

Signal "Y2" is assigned in this description block

4.1.8 Descriptions where results do not differ due to simulators (different from VHDL)

STARC_VLOG 4.1.8.1

RULE NAME	**Shift the observation point of a signal from an assignment point**	
MESSAGE	**Edge-sensitive 'always' process contains assignment(s) that is not delayed from the referenced signal observation point. It is recommended to delay assignment(s) of observed signals from the moment of their observation.**	
	DETAIL	*Blocking assignment without intra-assignment delay is detected.*
PROBLEM DESCRIPTION	When signal is shared between the different blocks (simultaneously updated in first one and read in another one), result becomes simulator-dependent, thus undefined.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker detects edge-sensitive 'always' processes (each signal in the sensitivity list has an edge specifier) and scans them for blocking assignment without intra-assignment delay – if RHS of such assignment contains signal => violation Note: other assignments after first violation is detected are not scanned	

EXAMPLE-1: [1] edge-sensitive 'always' process is described;
[2] blocking assignment without intra-assignment delay is specified => violation;
Note: no violation for the second blocking assignment without intra-assignment delay

```
always @( posedge CLK or posedge RESET ) begin

    Q1 = DATA;
    #10;
    Q2 = RESET;

end
```

Edge-sensitive 'alw ays' process contains assignment(s) that is not delayed from the referenced signal observation point. It is recommended to delay assignment(s) of observed signals from the moment of their observation.

Blocking assignment w ithout intra-assignment delay is detected.

EXAMPLE-2: [1] edge-sensitive 'always' process is described;
[2] non-blocking assignment without intra-assignment delay is specified => no violation

```
always @( posedge CLK ) begin

    Q3 <= DATA;

end
```

STARC_VLOG 4.1.8.4

RULE NAME	**Avoid using edges other than clock signals to the greatest extent possible**	
MESSAGE	**Module "{ModuleName}" contains {DogCount} event control statement(s) using {IllegalSigCount} signal(s) other than clock(s): {ListOfClocks}. Avoid using edges other than clock signals to the greatest extent possible.**	
	DETAIL	*Signal "{SignalName}" is not a clock*
PROBLEM DESCRIPTION	Making testbench dependent on the same clock(s) as UUT reduces the risk of race conditions, especially if delays and clock phases are properly ordered. However, Verilog does not define process execution order and simultaneous events on clock and non-clock signals can cause unexpected results. Either some clock-dependent process inside a UUT will react first, or some non-clock-dependent process in a testbench. Thus, wrong value could be written to or read from UUT. It is better to synchronize whole model (testbench + UUT) by the same clock(s) to avoid such race conditions.	
	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker collects list of clock signals (per module) by extracting clocks from: – flip-flop inferences – ports/regs/nets with synthesis attribute “(* synthesis, clock *)” specified Note: since FF inferences are rare in testbench code, clock signal has to be specified somehow. Checker provides support for (* synthesis, clock *) custom non-standard attribute to specify clock signal in testbench. Setting this attribute in Verilog description is up to designer. Checker then verifies all event control statements: – any signal in these controls should be a clock signal – in case of violation, main message is displayed for module and one detail per each non-clock signal	

EXAMPLE-1: [1] all event control expressions are synchronized by clock "SYNC" => no violation;
[2] note: clock "SYNC" is extracted from flip-flop inference;
[3] note: either 'SYNC' or 'negedge/posedge SYNC' are treated as correct clocks;

```
always @( posedge SYNC )
     RES <= #DELAY DATA;
end

initial begin
     TMP = 0;
     ...
     @( SYNC )
          TMP = #DELAY TMP + 1;
     ...
     @( negedge SYNC )
          TMP = #DELAY TMP + REG_SUM;
end
```

EXAMPLE-2: [1] all event control expressions are synchronized by clock "SYNC" => no violation;
[2] note: clock "SYNC" is extracted from port declared with attribute;

```
(* synthesis, clock *) input SYNC;

always @( SYNC )
     RES <= #DELAY DATA;
end

initial begin
     TMP = 0;
```

```
    ...
    @( posedge SYNC )
        TMP = #DELAY TMP + 1;
    ...
    @( negedge SYNC )
        TMP = #DELAY TMP + REG_SUM;
end
```

EXAMPLE-3: [1] module has two clocks: "CLK" and "SYNC" (extracted from the attributes);
[2] 3 event control statements using 2 non-clock signals => violation;

Module "tb" contains3 event control statement(s) using 2 signal(s) other than clock(s): CLK, SYNC. Avoid using edges other than clock signals to the greatest extent possible.

Signal "RESET" is not a clock

Signal "START" is not a clock

Signal "START" is not a clock

```
module tb;
    (* synthesis, clock *) wire SYNC;
    (* synthesis, clock *) wire CLK;

    always @( posedge CLK ) begin
        #DELAY;
        COUNT = #DELAY COUNT + 1;
        ...
    end

    always @( negedge SYNC or posedge RESET or posedge CLK ) begin
        ...
    end

    initial begin
        #DELAY;
        @( posedge START );
        ...
        @( RESET or START or CLK );
        ...
    end
endmodule
```

EXAMPLE-4: [1] module has a clock: "CLK" (extracted from the attribute);
[2] enabled task contains 2 event control statements using 2 non-clock signals => violation;

Module "tb" contains2 event control statement(s) using 2 signal(s) other than clock(s): CLK. Avoid using edges other than clock signals to the greatest extent possible.

Signal "SYNC_1" is not a clock

Signal "SYNC_2" is not a clock

```
module tb;
    (* synthesis, clock *) wire CLK;
                           wire SYNC_1;
                           wire SYNC_2;

    task wait_for_edges;
    begin
        @( posedge SYNC_1 );
        @( negedge SYNC_2 );
    end
    endtask

    always @( posedge CLK ) begin
        ...
        wait_for_edges;
        ...
    end
endmodule
```

4.2 *Task description*

4.2.3 Pay due attention to task I/O arguments (different from VHDL)

STARC_VLOG 4.2.3.2

RULE NAME	**Do not define output arguments when generating test vectors using task**	
MESSAGE	**Task "{TaskName}" has both timing control(s) and assignment(s) to output(s). Such description has risk that necessary values will not be delivered to outputs, because task modifies outputs after the end of its execution only. It is not recommended to define output task arguments when generating test vectors using tasks.**	
	DETAIL	*The first of {AssignmentsCount} assignment(s) to output argument "{OutputName}".*

PROBLEM DESCRIPTION	Verilog language defines following rules for 'task' statements: input values are transferred to the task at start of its execution; output values are returned only after the task is complete. This property of output values makes impossible to generate simulation-time-dependent test vectors on task outputs:
	`task GenSequence;`
	`   input  [1:0] ADDR;`
	`   input  [7:0] DATA;`
	`   output [1:0] A;`
	`   output [7:0] DIN;`
	`begin`
	`   @( posedge CLK );`
	`   A   = ADDR;       //`
	`   DIN = DATA[7:4]; //`
	`   @( posedge CLK );`
	`   A   = ADDR;       //`
	`   DIN = DATA[3:0]; //`
	`end`
	`endtask`
	Example above describes the task that generates simulation-time-dependent test vectors (there are two timing control statement). Changes that will never appear on the task outputs, are marked with red "// -" comment, whereas changes that will appear on the tasks outputs are marked with green "// -" comment.
	Consecutively, in order to avoid the problem described above, task should assigned global signals only or does not contains timing control statement.
	Hint: to generate simulation-time-dependent test vectors with task, do not define outputs, but assign globally defined signals instead:
	`reg [1:0] A;`
	`reg [7:0] DIN;`
	`...`
	`task GenSequence;`
	`   input  [1:0] ADDR;`
	`   input  [7:0] DATA;`
	`begin`
	`   @( posedge CLK );`
	`   A   = ADDR;       //`
	`   DIN = DATA[7:4]; //`
	`   @( posedge CLK );`
	`   A   = ADDR;       //`
	`   DIN = DATA[3:0]; //`
	`end`
	`endtask`	

	LEVEL	RECOMMENDATION 1
CHECKER BEHAVIOR	Checker scans 'task' statements that meet following criteria: – at least one output is defined – at least one timing control statement is specified. Each assigned task output is rule violation. Note-1: intra-assignment delay is not timing control in this context. Note-2: when output argument is vector, assignment to its bit / slice is the same that assignment to whole vector.	

EXAMPLE-1: [1] task "GenSequence" has two event control statements and two outputs are defined;
[2] output "A" is assigned once => violation;
[3] output "DIN" is assigned two times (part select is used) => violation

```
task GenSequence;
     output [1:0] A;
     output [7:0] DIN;
begin
     @( posedge CLK );
     A   = ADDR;
     DIN = DATA[7:4];
     @( posedge CLK );
     DIN = DATA[3:0];
end
endtask
```

Task "GenSequence" has both timing control(s) and assignment(s) to output(s). Such description has risk that necessary values w ill not be delivered to outputs, because task modifies outputs after the end of its execution only. It is not recommended to define output task arguments w hen generating test vectors using tasks.

The first of 1 assignment(s) to output argument "A".

The first of 2 assignment(s) to output argument "DIN".

EXAMPLE-2: [1] task "GenSequence_1" has two event control statements, delay and two outputs are defined;
[2] output "A" is assigned 2 times => violation;
[3] output "DIN" is assigned once => violation

```
integer TMP;
...
task GenSequence_1;
     output [1:0] A;
     output [7:0] DIN;
begin
     A   = ADDR + 1;
     @( posedge CLK );
     TMP = 0;
     #DLY_TRAN;
     TMP = 1;
     @( posedge CLK );
     A   = ADDR + 2;
     DIN = DATA[3:0];
end
endtask
```

Task "GenSequence_1" has both timing control(s) and assignment(s) to output(s). Such description has risk that necessary values w ill not be delivered to outputs, because task modifies outputs after the end of its execution only. It is not recommended to define output task arguments w hen generating test vectors using tasks.

The first of 2 assignment(s) to output argument "A".

The first of 1 assignment(s) to output argument "DIN".

STARC_VLOG 4.2.3.4

RULE NAME	**Do not define as input arguments when regularly observing the signals inside a task**	
MESSAGE	**Task "{TaskName}" has both timing control(s) and reference(s) to input argument(s). Such description has risk that necessary values will not be read from inputs, because inputs are passed to the task only when task is called. It is not recommended to define input task arguments when regularly observing them inside a task.**	
	DETAIL	*The first of {ReferencesCount} reference(s) to input argument "{InputName}".*

PROBLEM DESCRIPTION

Verilog language defines following rules for 'task' statements: input values are transferred to the task at start of its execution only; output values are returned after the task is complete. This property of input values makes impossible to perform simulation-time-dependent observation of task inputs:

```
task GenSequence;
   input  [1:0] ADDR;
   input  [7:0] DATA;
begin
   A   = ADDR;       //
   DIN = DATA;       //
   ...
   @( posedge CLK );
   A   = ADDR;       //
   DIN = DATA;       //
   ...
end
endtask
```

Example above describes the task that observes simulation-time-dependent signals (there are two timing control statement). Green "// +" comment marks input references that got an actual value of input signal, whereas red "// -" comment marks input references that got an obsolete value of input signal. So, it is not always possible to observe the signal values using the task input signal.

Consecutively, in order to avoid the problem described above, task should referenced global signals only or does not contains timing control statement.

Hint: to observe simulation-time-dependent signals in a task, do not define inputs but observe globally defined signals instead:

```
wire [1:0] ADDR;
wire [7:0] DATA;
...
task GenSequence;
begin
   A   = ADDR;       //
   DIN = DATA;       //
   ...
   @( posedge CLK );
   A   = ADDR;       //
   DIN = DATA;       //
   ...
end
endtask
```

LEVEL	RECOMMENDATION 1

CHECKER BEHAVIOR

Checker scans 'task' statements that meet following criteria:

– at least one input is defined

– at least one timing control statement is specified.

Each referenced task input is a rule violation.

Note-1: intra-assignment delay is not timing control in this context.

Note-2: when input argument is vector, reference to its bit / slice is the same that whole vector is

	referenced.

EXAMPLE-1: [1] task "GenSequence_1" has two event control statements and two inputs referenced;
[2] input "ADDR" is referenced once => violation;
[3] input "DATA" is referenced two times => violation

```
task GenSequence_1;
     input  [1:0] ADDR;
     input  [7:0] DATA;
begin
     @( posedge CLK );
     A   = ADDR;
     DIN = DATA;
     @( posedge CLK );
     DIN = DATA;
end
endtask
```

Task "GenSequence_1" has both timing (s) and reference(s) to input argument(s). Such description has risk that necessary values w ill not be read from inputs, because inputs are passed to the task only w hen task is called. It is not recommended to define input task arguments w hen regularly observing them inside a task.

The first of 1 reference(s) to input argument "ADDR".

The first of 2 reference(s) to input argument "DATA".

EXAMPLE-2: [1] task "GenSequence_2" has a delay statement and two inputs referenced;
[2] input "ADDR" is referenced two times => violation;
[3] input "DATA" is referenced once => violation

```
integer TMP;
...
task GenSequence_2;
     input  [1:0] ADDR;
     input  [7:0] DATA;
begin
     A   = ADDR;
     DIN = DATA[3:0];
     TMP = 0;
     #DLY_TRAN;
     TMP = 1;
     A   = ADDR;
end
endtask
```

Task "GenSequence_2" has both timing (s) and reference(s) to input argument(s). Such description has risk that necessary values w ill not be read from inputs, because inputs are passed to the task only w hen task is called. It is not recommended to define input task arguments w hen regularly observing them inside a task.

The first of 2 reference(s) to input argument "ADDR".

The first of 1 reference(s) to input argument "DATA".

NOTES

The Design Verification Company

Aldec, Inc.
Corporate Headquarters
2260 Corporate Circle
Henderson, NV 89074

Tel: +1 702 990 4400
Toll Free: +1 800 487 8743
Fax: +1 702 990 4414