Chapter
1

VHDL Coding Guidelines

1.1 Introduction

Coding of design behaviour and architecture is one of the most important steps in the whole chip design project. It has major impact on logic synthesis and routing results, timing robustness, verifiability, testability and even product support.

The VHDL Coding Guidelines help chip and macro development teams to rapidly understand each other's code. Macro based designs integrate easier, if these common coding styles are followed. This also applies to externally developed softcores. Codes will not need modification if simulator, synthesis tool or technology is exchanged. Code invariance wrt. Synthesis tool is given in case of a similar VHDL synthesis subset. Code invariance wrt. technology is given in case of similar performance and cell set. In addition the given guidelines enable high synthesis quality and simulation performance.

The VHDL Coding Guidelines need continuous adaptation according to new tool properties and new upcoming methodologies. Please participate in this process with your design know-how. Direct your contributions and related questions to the SC Highway Frontend Hotline (hwfe@hl.siemens.de, tel.: 24666). Contribute rules for VHDL coding, that turned out to prevent errors in the downstream flow, or recommendations, that alleviate further design, re-use or maintenance.

The VHDL Coding Guidelines may be passed to sub-contractors or cooperation partners. Ideally their coding works should comply to these guidelines, enabling rapid and safe integration with internally developed modules.

Reading of the VHDL Coding Guidelines is most efficient at the beginning of a chip-design-project. Furthermore "Early Code Review" should be considered in a very early phase of VHDL coding as a training measure.

Up to now every designer is responsible to follow relevant rules. Automated checks are not yet available. However, it is planned, to integrate such a tool into SC Highway, which will then use the rules that are given here.

1.2 Rules by Topics

Each item is marked according to the categories

- ❏ mandatory (m)
- ❏ strongly recommended (r)
- ❏ advisable (a)
- ❏ explanatory (e)

1.2.1 Compilation

1. Configuration must be in a separated file (m)
2. A configuration declaration is needed for each architecture (m).
3. Testbench and DUT should be compiled into the same library (a).
4. Design-internal references must use library work (m).
5. Edit env-files to extend the resource-library list (e).
6. Avoid more package references than needed (a).
7. Use expanded names in binding indications and use clauses only (r).

1.2.2 Partitioning VHDL Code

1. Use standard top-levels xmpl, xmpl_top, and xmpl_chip (r).
2. Keep all objects and subprograms in the nearest possible scope (a).
3. Keep local objects invisible outside a package (a).
4. Transfer stable component declarations into component packages (a).
5. Re-use functionality of standard packages as much as possible (a).
6. A VHDL unit should not exceed 200 lines of code (a)
7. A process or subprogram should not exceed 50 lines of code (a)
8. Component instantiations should not exceed 4 levels of hierarchy (a)
9. An architecture should not contain more than 5 processes (a)
10. Size of VHDL Units(a).
11. VHDL units should take into account later floorplanning (a).
12. Asynchronous parts should be placed into separate entities (a).
13. Structural and behavioural code should not be mixed (r).
14. Define project packages early in the project (a).

1.2.3 Storing VHDL Code

1. Only verified VHDL code allowed for check-in (m).
2. Store each VHDL unit into a separate file (r).
3. Hide old VHDL files (e).
4. Submodules referenced by more than one module should be stored into a separate units directory (a).

1.2.4 Naming Conventions

1. All primary unit names must be unique in the project library (m).
2. Naming conventions for VHDL units (a).
3. Naming conventions for VHDL files (a).
4. Naming conventions for VHDL objects (a).
5. Identifier naming must be based on English language (m).
6. Use meaningful identifier names (a).
7. Signal assignments for renaming purposes should be avoided (a).
8. Choose short instance names (a).
9. Keyword casing should be uniform in all VHDL code (a).
10. Identifier casing should be uniform in all VHDL code (r).
11. Separate identifier components using underscores (a).
12. Trailing and consecutive underscores are not allowed in identifiers (m).
13. Use an intermediate signal for reading from an output port (e).
14. Use standard top-levels xmpl, xmpl_top, and xmpl_chip (r).
15. Length of entity names should not exceed 32 characters (r).

1.2.5 Data-Types

1. Unresolved data types std_ulogic(_vector) are recommended (a).
2. Data types at chip and macro ports: std_(u)logic(_vector) (m).
3. Data types at module ports: std_(u)logic(_vector) recommended (a).
4. Generics must have type integer for synthesis (m).
5. Use complex data types inside architectures (a).
6. Use a small number of self-defined global data types (a).
7. Self-defined conversion functions must assert un-mappable input values (m).
8. Multidimesional arrays can lead to mismatch between TRF file and Verilog netlist (e).

1.2.6 Design Rules

1. Avoid gated clocks unless absolutely necessary (r).
2. Avoid latches unless absolutely necessary (r).
3. Reset is mandatory for sequential processes (m).
4. All outputs of synchronous modules should be registered (r).
5. Use gate instantiation only at few instances (r).
6. Avoid internal three-state busses unless absolutely necessary (r).
7. Component instantiation (e).
8. Avoid snake paths (r).
9. Instantiate the power pads for each power-domain (r).
10. Insert test-enable for scan-test (m).
11. Refer to the rising edge of clock (r).
12. Do not use combinational feedback paths (m).
13. Spacer cells needed between pads (e).

1.2.7 Modeling Details

1. Spend extensive efforts for documentation of interfaces (r).
2. Build interface consistency checks into the model (a).
3. Balance clock to delta accuracy (m).
4. Pull-ups and pull-downs have to be modeled on chip level (r).
5. Strength stripping should be performed on chip level (r).
6. Do not assign the value 'X' (r).
7. Use efficient description for the wrap around of counters (a).
8. Use `case`-statements instead of `if`-statements when the conditions are mutually exclusive (a).
9. Use vector operations instead of loops (a).
10. Use boolean expressions instead of simple `if`-statements (a).
11. Avoid unnecessary repetition of function calls (a).
12. Avoid unnecessary computations within loops (a).
13. Do not use FSM attributes within the VHDL code (m).

1.2.8 Testbenches

1. A testbench has neither ports nor generics (e).

2. Format for pattern text-IO is LSIM-TV (a).
3. Use abstract, compact coding style in testbench and simulation models (a).
4. Stop the simulation from within the testbench (a).
5. Use assertion messages extensively (a).
6. Global signals may be used for testing purposes only (m).

1.2.9 Text-IO

1. Use std.textio only, where really necessary (a).
2. Communication between modules using text-IO is forbidden (m).
3. No characters with a lower rank than space may occur in text-IO (m).
4. Binary file-IO may not be used (a).
5. Use relative pathnames for files accessed through text-IO (m).
6. Check the success of every file operation (r).
7. Store file data to a VHDL variable, if read multiple times (e). Format for pattern text-IO is LSIM-TV (a).

1.2.10 Readability

1. Use a separate library clause for each declared resource library (a).
2. Use a separate use clause for each declared package (a).
3. Indentation should be consistent in all VHDL code (r).
4. Align comments vertically (a).
5. Align declaration lists vertically (a).
6. Align association lists vertically (a).
7. Processes should be labeled (r).
8. Use closing labels for entity, architecture, process, loop, generate, etc. (a).
9. Indicate condition at remote ends of control structures (a).
10. Language for comments and code documentation is English (m).
11. Comments should be related to the lines of code below (a).
12. Comment each process, each subprogram and global aspects (r).
13. Trailing comments are allowed for declarative lists (a).
14. Aliases are allowed in testbenches and behavioural code only (m).
15. Maximum line-length is 80 characters (a).
16. Use one statement per line (a).
17. Group related constructs (a).

18. Group related port or signal declarations (a).
19. Switch constructs should not be nested to more than 4 levels (a).
20. Remove unnecessary code fragments (a).
21. Use named association in association lists (r).
22. Order of items should be the same in package header and body (r).
23. Use predefined attributes to make code generic and more re-usable (a).

1.2.11 Portability

1. Language for modeling is VHDL-87 (m).
2. Check for LRM-compatibility using cv w/o -compat switch (r).
3. VHDL-93 keywords should not be used (r).
4. Verilog keywords should not be used (r).
5. SDF keywords should not be used (r).
6. Carefully use real values with PN-generators (a).
7. Allowable replacement characters are forbidden (m).
8. Tool specific types may not be used (m).
9. Operating system specific features may not be used (r).
10. Never redefine standard operators, subprograms, attributes, and packages (r).
11. Bussed ports of width one are forbidden (r).
12. Design-internal references must use library work (m).
13. Configuration declarations must have the configuration name and the entity name in one line for `vimport` (a).

1.2.12 Arithmetic

1. Bussed arithmetic objects use "downto" as their index orientation (r).
2. Request external VHDL code suppliers to deliver used arith. packages (r).
3. Where to use ieee.std_logic_arith and std_developerskit.synth_regpak (e).
4. Where to use std_logic_signed and std_logic_unsigned (e).
5. Array widths for std_logic_arith operators (e).
6. Use conversion functions around arithmetic operators (e).
7. Comparison of arith. arrays should be based on same type and width (r).
8. Type mark comparison operands (r).

1.2.13 Synthesizability

Note that the restrictions listed below do not apply to pure simulation models and not to testbench models. This section is not a replacement for the "VHDL Compiler Reference Manual" (Synopsys).

1. Run synthesis early in the design process (r).
2. Code for synthesis must match the synthesis subset (m).
3. Store package header and body into same file (m).
4. Default values for ports, signals and variables may not be used (m).
5. Data types must be synthesizable (m).
6. Array ranges must be of type integer (m).
7. Integer type objects must be constrained wrt. to their range (m).
8. Use standard templates for clocked processes (r).
9. Generics must have type integer (m).
10. For soft-core development use generics, don't use constants (r).
11. Place port and generic maps at the component instantiation (m).
12. Instantiated component and entity name must be equal, incl. casing (m).
13. Add an "others" statement to the FSM-switch (m).
14. All conditional assignments should be checked for completeness (r).
15. The sensitivity list for combinational processes must be complete (m).
16. Bit-wise association of arrays in port maps is not possible in presence of generics (m).
17. Use gate instantiation only at few instances (r).
18. Aliases may not be used for synthesis (m).
19. Embedding synthesis scripts as meta comments is not acceptable (r).
20. Use only standard meta comments (r).
21. Requirements for resource sharing (e).

1.2.14 Simulation Performance

1. Remove unnecessary code fragments (a).
2. EMC and transmission line effects need not be modeled (a).
3. Use abstract data types inside modules (a).
4. Avoid using resolved data types (a).
5. Minimize the number of processes, signals and signal assignments (a).
6. Replace signals by variables whenever possible (a).
7. Inhibit execution of statements where not necessary (a).
8. Divide large memories into blocks (a).

9. Avoid unnecessary repetition of function calls (a).
10. Avoid unnecessary computations within loops (a).
11. Avoid declaring constants within subprograms (a).
12. Be aware of the use of the attributes DELAYED, STABLE, QUIET, and TRANSACTION (a).
13. Minimize the number of signals of your sensitivity list (a).
14. Always use `deallocate(access_obj)` to dereference allocated memory (r).

1.2.15 Critical Constructs

1. Do not use anonymous types (r).
2. Do not use guarded signals, guarded assignments, and guarded expressions (r).
3. Do not use disconnect, register, and bus (r).
4. Do not use port modes buffer and linkage (r).
5. Do not use blocks. Use entities instead to describe design hierarchy (a).
6. Use only well tested functions for object-initialization purposes (a).
7. Concurrent procedures may not have any side effects (r).
8. Avoid identifier hiding caused by loop index (r).
9. Global signals may be used for testing purposes only (m).
10. Recursive use of subprograms is forbidden (m).

1.3 Rules by Importance

1.3.1 Mandatory

1. Configuration must be in a separated file (m)
2. A configuration declaration is needed for each architecture (m).
3. Design-internal references must use library work (m).
4. Only verified VHDL code allowed for check-in (m).
5. Store package header and body into same file (m).
6. All primary unit names must be unique in the project library (m).
7. Data types at chip and macro ports: std_(u)logic(_vector) (m).
8. Generics must have type integer for synthesis (m).
9. Trailing and consecutive underscores are not allowed in identifiers (m).
10. Identifier naming must be based on English language (m).
11. Self-defined conversion functions must assert un-mappable input values (m).
12. Reset is mandatory for sequential processes (m).
13. Insert test-enable for scan-test (m).
14. Do not use combinational feedback paths (m).
15. Global signals may be used for testing purposes only (m).
16. Communication between modules using text-IO is forbidden (m).
17. No characters with a lower rank than space may occur in text-IO (m).
18. Use relative pathnames for files accessed through text-IO (m).
19. Balance clock to delta accuracy (m).
20. Do not use FSM attributes within the VHDL code (m).
21. Language for comments and code documentation is English (m).
22. Language for modeling is VHDL-87 (m).
23. Allowable replacement characters are forbidden (m).
24. Tool specific types may not be used (m).
25. Code for synthesis must match the synthesis subset (m).
26. Default values for ports, signals and variables may not be used (m).
27. Data types must be synthesizable (m).
28. Array ranges must be of type integer (m).
29. Integer type objects must be constrained wrt. to their range (m).
30. Place port and generic maps at the component instantiation (m).

31. Instantiated component and entity name must be equal, incl. casing (m).
32. Add an "others" statement to the FSM-switch (m).
33. The sensitivity list for combinational processes must be complete (m).
34. Bit-wise association of arrays in port maps is not possible in presence of generics (m).
35. Aliases may not be used for synthesis (m).
36. Recursive use of subprograms is forbidden (m).

1.3.2 Strongly Recommended

1. Use expanded names in binding indications and use clauses only (r).
2. Identifier casing should be uniform in all VHDL code (r).
3. Use standard top-levels xmpl, xmpl_top, xmpl_chip (r).
4. Length of entity names should not exceed 32 characters (r).
5. Structural and behavioural code should not be mixed (r).
6. Store each VHDL unit into a separate file (r).
7. Avoid gated clocks unless absolutely necessary (r).
8. Avoid latches unless absolutely necessary (r).
9. Use gate instantiation only at few instances (r).
10. Avoid internal three-state busses unless absolutely necessary (r).
11. Avoid snake paths (r).
12. Instantiate the power pads for each power domain (r).
13. All outputs of synchronous modules should be registered (r).
14. Spend extensive efforts for documentation of interfaces (r).
15. Pull-ups and pull-downs have to be modeled on chip level (r).
16. Strength stripping should be performed on chip level (r).
17. Do not assign the value 'X' (r).
18. Check the success of every file operation (r) .
19. Indentation should be consistent in all VHDL code (r).
20. Processes should be labeled (r).
21. Comment each process, each subprogram and global aspects (r).
22. Use named association in association lists (r).
23. Order of items should be the same in package header and body (r).
24. Check for LRM-compatibility using cv w/o -compat switch (r).
25. VHDL-93 keywords should not be used (r).
26. Verilog keywords should not be used (r).

27. SDF keywords should not be used (r).
28. Operating system specific features may not be used (r).
29. Never redefine standard operators, subprograms, attributes, and packages (r).
30. Bussed ports of width one are forbidden (r).
31. Bussed arithmetic objects use "downto" as their index orientation (r).
32. Request external VHDL code suppliers to deliver used arith. packages (r).
33. Comparison of arith. arrays should be based on same type and width (r).
34. Type mark comparison operands (r).
35. Run synthesis early in the design process (r).
36. Use standard templates for clocked processes (r).
37. Refer to the rising edge of clock (r).
38. For soft-core development use generics, don't use constants (r).
39. All conditional assignments should be checked for completeness (r).
40. Embedding synthesis scripts as meta comments is not acceptable (r).
41. Use only standard meta comments (r).
42. Always use `deallocate(access_obj)` to dereference allocated memory (r).
43. Do not use anonymous types (r).
44. Do not use guarded signals, guarded assignments, and guarded expressions (r).
45. Do not use disconnect, register, and bus (r).
46. Do not use port modes buffer and linkage (r).
47. Concurrent procedures may not have any side effects (r).
48. Avoid identifier hiding caused by loop index (r).

1.3.3 Advisable

1. Testbench and DUT should be compiled into the same library (a).
2. Use a separate library clause for each declared resource library (a).
3. Use a separate use clause for each declared package (a).
4. Avoid more package references than needed (a).
5. Keep all objects and subprograms in the nearest possible scope (a).
6. Keep local objects invisible outside a package (a).
7. Transfer stable component declarations into components packages (a).
8. Re-use functionality of standard packages as much as possible (a).
9. A VHDL unit should not exceed 200 lines of code (a)
10. A process or subprogram should not exceed 50 lines of code (a)

11. Component instantiations should not exceed 4 levels of hierarchy (a)
12. An architecture should not contain more than 5 processes (a)
13. Size of VHDL Units(a).
14. VHDL units should take into account later floorplanning (a).
15. Asynchronous parts should be placed into separate entities (a).
16. Define project packages early in the project (a).
17. Naming conventions for VHDL units (a).
18. Submodules referenced by more than one module should be stored into a separate units-directory (a).
19. Naming conventions for VHDL files (a).
20. Naming conventions for VHDL objects (a).
21. Use meaningful identifier names (a).
22. Signal assignments for renaming purposes should be avoided (a).
23. Choose short instance names (a).
24. Keyword casing should be uniform in all VHDL code (a).
25. Separate identifier components using underscores (a).
26. Unresolved data types std_ulogic(_vector) are recommended (a).
27. Data types at module ports: std_(u)logic(_vector) recommended (a).
28. Use complex data types inside architectures (a).
29. Use a small number of self-defined global data types (a).
30. Build interface consistency checks into the model (a).
31. Use efficient description for the wrap around of counters (a).
32. Format for pattern text-IO is LSIM-TV (a).
33. Use abstract, compact coding style in testbench and simulation models (a).
34. Stop the simulation from within the testbench (a).
35. Use assertion messages extensively (a).
36. Use std.textio only, where really necessary (a).
37. Binary file-IO may not be used (a).
38. Align comments vertically (a).
39. Align declaration lists vertically (a).
40. Align association lists vertically (a).
41. Use closing labels for entity, architecture, process, loop, generate, etc. (a).
42. Indicate condition at remote ends of control structures (a).
43. Comments should be related to the lines of code below (a).
44. Trailing comments are allowed for declarative lists (a).
45. Maximum line-length is 80 characters (a).

46. Use one statement per line (a).
47. Group related constructs (a).
48. Group related port or signal declarations (a).
49. Switch constructs should not be nested to more than 4 levels (a).
50. Remove unnecessary code fragments (a).
51. Use predefined attributes to make code generic and more re-usable (a).
52. Configuration declarations must have the configuration name and the entity name in one line for `vimport (a).`
53. Use case-statements instead of if-statements when the conditions are mutually exclusive (a).
54. Use vector operations instead of loops (a).
55. Use boolean expressions instead of simple if-statements (a).
56. Carefully use real values with PN-generators (a).
57. EMC and transmission line effects need not be modeled (a).
58. Minimize the number of processes, signals and signal assignments (a).
59. Replace signals by variables whenever possible (a).
60. Inhibit execution of statements where not necessary (a).
61. Divide large memories into blocks (a).
62. Avoid unnecessary repetition of function calls (a).
63. Avoid declaring constants within subprograms (a).
64. Be aware of the use of the attributes DELAYED, STABLE, QUIET, and TRANSACTION (a)
65. Minimize the number of signals of your sensitivity list (a).
66. Avoid unnecessary computations within loops (a).
67. Do not use blocks. Use entities instead to describe design hierarchy (a).
68. Use only well tested functions for object-initialization purposes (a).

1.3.4 Explanatory

1. Edit env-files to extend the resource-library list (e).
2. Hide old VHDL files (e).
3. Use an intermediate signal for reading from an output port (e).
4. Multidimesional arrays can lead to mismatch between TRF file and Verilog netlist (e).
5. Component instantiation (e).
6. Spacer cells needed between pads (e).
7. A testbench has neither ports nor generics (e).
8. Store file data to a VHDL variable, if read multiple times (e).

9. Where to use ieee.std_logic_arith and std_developerskit.synth_regpak (e).
10. Where to use std_logic_signed and std_logic_unsigned (e).
11. Array widths for std_logic_arith operators (e).
12. Use conversion functions around arithmetic operators (e).
13. Requirements for resource sharing (e).

1.4 Rules with Details

1.4.1 Configuration must be in a separated file (m)

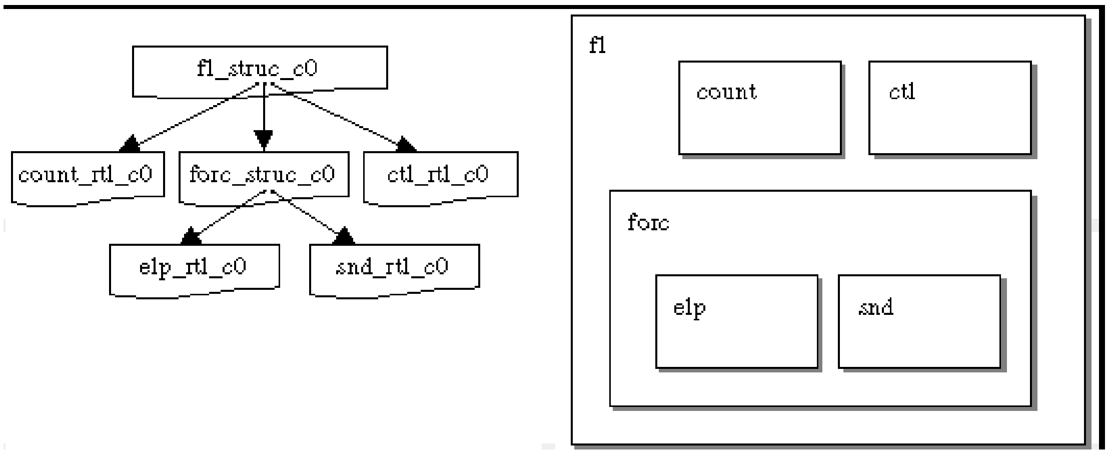
With the SC Highway 2.0 the configurations are not any longer anlayzed by Synopsys' Design Compiler as in previous HW versions but are used by the SSE environment to extract the dependencies only. Therefore it is **necessary** to keep a configuration in a separate file, otherwise the synthesis will fail.

1.4.2 A configuration declaration is needed for each architecture in the design (m)

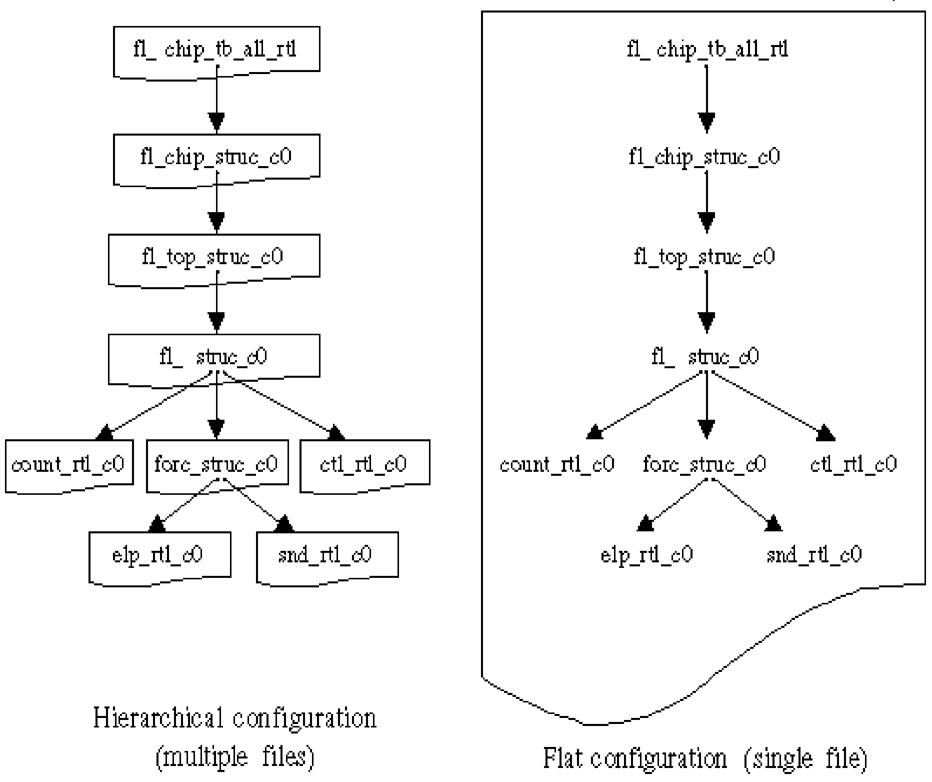
A configuration must be used for each component instantiation. This is a requirement imposed by `crvmake`. Besides that, simulating unconfigured designs is dangerous, as it is difficult to determine, which set of architectures is implied. The rule for binding unconfigured entities is: "most recently analyzed architecture elaborates" (default binding).

Only one level of hierarchy may be configured within one configuration.

```
configuration <configuration-name> of <entity-name> is
  for <architecture-name>
  [
    for <instance-name> | all : <component-name>
      use configuration
        work.<name-of-config-to-be-referenced>
    end for;
    ...
  ]
  end for;
end <configuration-name>;
```



Flat configurations (i.e., one configuration for all levels of the model) are useful as a parts list. They can be derived from hierarchical configurations using, e.g., a Perl script. Flat configurations must not be used for simulation.



Note that configurations from the current project have to be referenced out of the VHDL library work, using the logical name work and no other logical name, even if one exists.

Note that generate loops (as well as the non-recommended blocks) require an additional for ... end for; pair in the configuration:

```
configuration fm_struc_withgen of fm is
  for struc             --<- architecture name
    for gen_adds        --<- generate-loop label
      for accu : add    --<- instance name within
                        --   generate-loop
        use configuration work.add_rtl_c0
      end for;
    end for;
  end for;
end fm_struc_withgen;
```

Configurations must be hierarchical, i.e., instantiated submodules are referenced via their configuration. This means that the statement `...use entity...` may not occur in any configuration. In some situations that is not feasible, since the referenced module resides in a VHDL-library other than work, and has no configuration associated with it:

- ❑ Padlib-, starlib-cells
- ❑ Modules developed externally

In these cases default binding must be used, i.e. omitting the component-configuration for that instantiation. This is acceptable only if the referenced module has not more than one architecture, which is the case for pad-, lunar- and memlib. Make sure to include a general use-statement in the configuration, which allows for Leapfrog-compilation/elaboration without the `-compat` switch:

```
library padlib;
use padlib.all;    --<- make all pads visible

configuration cli_chip_struc_c0 of cli_chip is
  for struc
    for core : cli_core
      use configuration work.cli_core_struc_c0
    end for;
    -- no component-configuration for the pads here!!!
  end for;
end cli_chip_struc_c0;
```

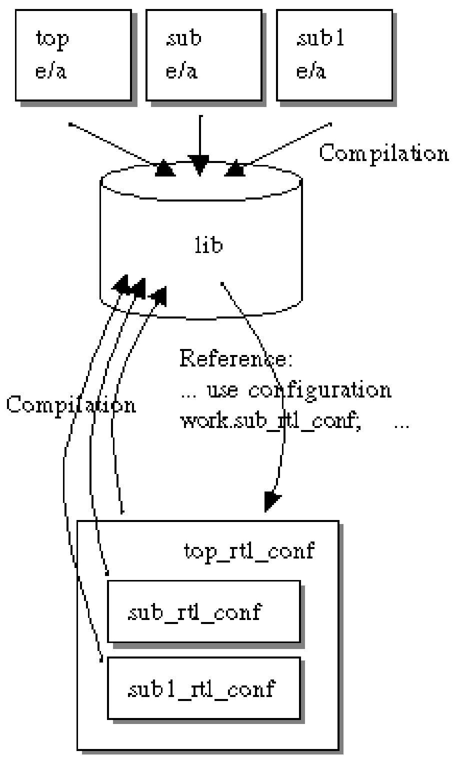
Important: Leapfrog-option -compat is not necessary!!! See chapter 1.4.112 *Check for LRM-compatibility using cv w/o -compat switch (r)* for more information.

1.4.3 Design-internal references must use library work (m)

During chip or core development all VHDL code is compiled into the library `work`. References to submodules (i.e., inside configurations) must use the logical library name work:

```
...
for all: sub use configuration work.sub_rtl_conf;
...                            -- ^^^^
```

This procedure enables compilation of the VHDL code in other environments without prior modification. This is the case, if the code shall be re-used as is. It is then compiled into a resource library, whose logical name can be defined at that point in time.



This reference can be re-used regardless of what the logical library name is (in the given case: "lib"), or will be

1.4.4 Edit env-files to extend the resource-library list (e)

If you are instantiating elements out of the resource libraries, the UNIX location must be defined in startup files:

For Leapfrog the mapping between logical name and UNIX location is done in `$HWPROJECT/env/cds.lib`:

```
    ...
   DEFINE memlib ${HWPROJECT}/resources/memlib/vhdl_rtl/lfobj_2.5
    ...
```

For Synopsys equivalent definitions are stored in `$HWPROJECT/env/.synopsys_vss.setup`:

```
   ...
   MEMLIB : $HWPROJECT/resources/memlib/vhdl_rtl/
vssobj_$SYNOPSYS_VERSION
   ...
```

The paths of standard libraries (ieee, synopsys, ...) are preconfigured in the startup files. Do not switch to the Cadence counterparts (Leapfrog). They are incompatible with the Synopsys libraries, which are used in the SC-Highway.

1.4.5 Testbench and DUT should be compiled into the same library (a)

This policy is chosen due to the logical dependency between the testbench and the device under test (DUT). `crvmake` can only cover this dependency if both units are in the same library.

1.4.6 Avoid more package references than needed (a)

Each use clause of the form

```
use work.mypack.all;
```

introduces a dependency, that is traced by the makefile generated by crvmake. That means, changes to that package cause re-compilation of the actual unit. This behaviour is intended. However, if none of the objects out of mypack are actually used, re-compilation is unnecessary.

Note that only dependences within library work are covered. We still recommend having only relevant references active, even if directed to resource libraries. This makes the code more understandable and reusable. Packaging design sources for re-use is made easier.

1.4.7 Use expanded names in binding indications and use clauses only (r)

Dependences between the VHDL files of a design are extracted by the makefile generator crvmake. It recognizes cross-references at binding indications and use clauses, i.e.

```
configuration cli_chip_struc_c0 of cli_chip is
  for struc
    for core : cli_core
      use configuration work.cli_core_struc_c0   --<-
    end for;
  end for;
end cli_chip_struc_c0;
```

or

```
use work.constants_pack.all;   --<-
```

1.4.8 Keep all objects and subprograms in the nearest possible scope (a)

VHDL objects relevant to one VHDL unit or subprogram should be defined in the nearest scope possible, i.e. in the declarative region of the respective VHDL unit or subprogram:

```
local_proc_example : process
  variable l_v : line;
  procedure local_write_p ( -- <---
    arg : in std_ulogic     -- <---
    ) is                    -- <---
  begin                     -- <---
    <some_manipulation>     -- <---
    write( l_v, arg );      -- <---
  end local_write_p;        -- <---
```

```
begin
  clk <= '0','1' after period_c/2;
  wait for period_c;
  local_write_p( alarm1_s );
  local_write_p( alarm2_s );
end process local_proc_example;
```

In a similar way make sure that types, constants, components and the like are only visible where necessary. VHDL as a language does not enforce object-oriented style. However, it is possible with VHDL, as with most programming languages, to still follow object-orientation and avoid global constructs. Imagine yourself extracting a part of the design as a testcase. With object-oriented code this task is no problem.

The other rule, which must be followed at the same time, is "single source": Project global information may be stored only once. Thus inconsistencies are avoided. An example is the active reset level. Avoid defining this level locally. Rather put the following in one of the project packages:

```
constant reset_active_c : std_ulogic := '0';
```

Do not use explicit constant values, if the value occurs multiple times. Instead define a constant with this value, and reference this.

Declare each object as global as necessary, e.g. signals needed for all architectures of an entity can be declared in the entity, instead of multiple declarations in each of the architectures:

```
entity with_signal is
  port (
        clk   : in  std_ulogic;
        dat_i : in  natural range 0 to 15;
        ctl_i : in  std_ulogic;
        dat_o : out natural range 0 to 31
        );

  signal accu : natural;
end with_signal;
```

The signal accu is available to all architectures of entity with_signal. The same is possible with subprograms.

1.4.9 Keep local objects invisible outside a package (a)

Objects must not necessarily get declared in the package header. If they are used in this package only, it is advisable to declare them in the body

```
package cmfuncs_pack is
  function cm_f ( arg : integer ) --<- visible with
    return natural;               --   the package
end cmfuncs_pack;
```

package body cmfuncs_pack is

```
  constant accuracy :
    integer := 2;  --<- this constant is local to the
                   --   package, i.e. visible in the body
    function cm_shl_f (        --<- this function is local
      arg, shft : integer )    --   to the package, i.e.
      return integer is        --   visible in the body only
    begin
      ... ;
    end cm_shl_f

    function cm_f ( arg : integer ) return natural is
      function cm_add_f (a, b : integer ) --<- this function
        return integer is                 --   is local to
      begin                               --   cm_f
        ...
      end cm_add_f;
    begin
      ...
    end cm_f;

  end cmfuncs_pack;
```

And these are the advantages:

Name space of the unit using the package is not narrowed unnecessarily.

Local subroutines are called from within the package only. Thus the amount of error checking can be less than for a generic subroutine.

1.4.10 Transfer stable component declarations into components packages (a)

Prior to instantiation of a component, the component needs to be declared. This can be done in the architecture declarative region:

```
  architecture struc of test is
    component comp
      port( ... );
    end component;
  begin
    inst: comp
      port map ( ... );
  end test;
```

However, if many architectures make use of this component, the declaration is better moved to a component package. Thus, inconsistencies are avoided. Changes at the component are visible at the same time to all referencing architectures. Then the instantiating architecture reads, as follows:

```
  use work.comp_pack.all;
  entity test ... end test;

  architecture struc of test is
  begin
    inst: comp
      port map ( ... );
  end test;
```

To recall the declaration of the component, you may use the tags-mechanism of emacshw. This mechanism displays the declaration immediately, even if it resides in a different file and directory.

The make utility detects, if a component package has changed. Depending architectures are re-compiled. Grouping the component declarations into several component packages reduces the probability of out-of-date conditions. Group components with similar functionality.

1.4.11 Re-use functionality of standard packages as much as possible (a)

Do not re-implement functionality already contained in standard packages. Use the easy-browsing feature in emacshw menu "VHDL" -> "Browse Packages" to search for a function, procedure or operator, that serves your needs.

1.4.12 Size of VHDL Units (a)

While the predefinition of block sizes the current version of the synthesis tool should be considered, in order to take advantage of its optimization and runtime capabilities. Synthesis times and process sizes get huge, if the blocks are too big. On the other hand avoid tiny blocks, as this would decrease optimization potential in synthesis (assuming that ungrouping is not used as a general style). Complexity must be estimated at partitioning, i.e. before mapping. Take the number of registers times 3 as an estimate.

While partitioning the design make sure that all module-outputs are registered. See "All outputs of synchronous modules should be registered" for more information.

1.4.13 VHDL units should take into account later floorplanning (a)

A project will run more smoothly, if design hierarchy needs not be changed during floorplanning. Therefore it is a good idea to target low connectivity between blocks, and to equalize area at the level of hierarchy, where floorplanning will take place.

1.4.14 Asynchronous parts should be placed into separate entities (a)

Asynchronous units are often subject to re-writing if technology is changed. Furthermore specific synthesis scripts and careful placement is required.

1.4.15 Structural and behavioural code should not be mixed (r)

If you are separating structural (only component instantiations) and behaviourl (RTL) code clearly, SSE will only compile the behavioural descriptions, while on the structural levels only a simple link will be performed. This has two advantages:

A link is much faster than a compile.

If a submodule changes, only the submodule will be recompiled

In addition, time budgeting is much easier, when you have no mix of structural and behavioural code. A graphical tool (`renoirhw) may be used for generation of structural code.`

1.4.16 Define project packages early in the project (a)

Dependences between the units of a VHDL model should be minimized. This is especially important for dependences from package headers. Respecting this rule reduces to a large extent recompilation time and disturbing interference between project members.
Use several dedicated packages for different topics. Don't mix unrelated functionality, this would lead to unnecessary dependences. Use any number of packages below around 30. Handling more packages gets difficult.
Use a separate package for project relevant data types, basic functions, constants, clock periods and other timing constants. Note, that package header and package body have to be stored in a single file, if this package is used for synthesized modules.
Start defining project packages at an early stage in the project. A stable setup is important because typically many VHDL units depend on the project packages.

1.4.17 Submodules referenced by more than one module should be stored into a separate units-directory (a)

Files containing design units are stored under $HWPROJECT/units/<entity.name>/vhdl/{beh,rtl}.

Submodules of <entity.name> may also be placed in this directory. If the subhierarchy of <entity.name> contains modules that are referenced by other modules above or beneath <entity.name> this submodule should be stored in a separate unit-directory. Additionally each entity should be stored into a separate file. Note that in no case design hierarchy is mapped to the directory structure.

Files containing packages are stored in $HWPROJECT/vhdl_packs/{beh,rtl}.

1.4.18 Only verified VHDL code allowed for check-in

Only analyzed and verified VHDL code may be checked in as a Clearcase element. Furthermore, up-to-date comments must be included. This rule must be strictly followed, because corrupt code gets visible to colleagues immediately. Their `crvmake- and smake-runs would then fail.`

By chance this situation may occur. In this case, advise your colleagues to temporarily generate their makefiles using

```
   crvmake ... $HWPROJECT/units/{a,b,c, ...}/vhdl/{rtl,tb}/*.vhd
```

a, b, c, etc. are units which they need at the moment excluding the faulty units that have been checked in unintentionally.

1.4.19 Store each VHDL unit into a separate file (r)

To avoid reanalysis of VHDL units (entity, architecture), which are not affected from a source code change, you should create separate files for the entity and the corresponding architectures. Especially, when you have more than one architecture for an entity, it is necessary to have them separated in different files.

Nevertheless, if VHDL units are not separated in different files, only related VHDL units (entity, corresponding architecture) may be combined in one file.

However with the SC HW version 2.0 all configurations must be located in a separate file.
Part 1.4.1, “Configuration must be in a separated file (m)”

1.4.20 Store package header and body into same file (m)

Packages are stored in `$HWPROJECT/vhdl_packs/{rtl,beh}`. For these packages header and body must be stored into one file due restrictions of Design Compiler.

1.4.21 Hide old VHDL files (e)

VHDL files from experiments and old VHDL files need to be hidden from the makefile generator crvmake. Such experimental code is very often unclean. crvmake might easily fail and the speed for makefile generation might degrade.

To hide the code you may chose to either

Rename the file to ....vhd.old

Remove the file using ct rmname ....vhd

The second solution is the preferred one. The original experimental file can be retrieved using a Clearcase configuration specification like

```
element * CHECKEDOUT
element * /main/LATEST -time yesterday
```

Instead of yesterday any other date in the past may be used.

As temporary workaround, you may select VHDL-files for makefile-generation using commands like:

```
crvmake $HWPROJECT/units/{a,b,c}/vhdl/rtl/*.vhd \
        $HWPROJECT/vhdl_packs/rtl/*.vhd
```

Old or experimental VHDL files are thus hidden from the compilation process.

1.4.22 All primary unit names must be unique in the project library (m)

Entities and configurations are primary VHDL units. Potential naming conflicts must be avoided through project agreements. In case of cooperation with subcontractors, primary unit name usage must be coordinated with them as well. If their units are delivered in precompiled form, name clashes could be resolved by using a separate VHDL library, which offers a new name space. Nevertheless, we recommend strongly not relying on this possibility, because it is working for simulation only. Name space overlap cannot be resolved in synthesis with Design Compiler.

1.4.23 Example for unit and file naming conventions

```
                    VHDL unit name    file name

Entity :            stm4_1            stm4_1-e.vhd
Architecture :      rtl               stm4_1-rtl-a.vhd
Configuration :     stm4_1_rtl_hier   stm4_1-rtl-hier-c.vhd
Testbench entity :  stm4_1_tb         stm4_1_tb-e.vhd
Testbench arch. :   beh               stm4_1_tb-beh-a.vhd
Testbench conf. :   stm4_1_tb_beh_c0  stm4_1_tb-beh-c0-c.vhd
Package :           arith_pack        arith_pack-p.vhd
```

1.4.24 Abbreviations for naming conventions

```
<e.id>      := <entity.identifier>
<a.id>      := <architecture.identifier>
<c.id>      := <configuration.identifier>
<t.id>      := <testbench.identifier>
<e.id.dut>  := <entity.identifier.of.device.under.test>
<p.id>      := <package.identifier>
```

1.4.25 Naming conventions for VHDL units (a)

The following conventions are recommended:

```
   <entity.name>         := <e.id>
   <architecture.name>   := <a.id>
   <configuration.name>  := <e.id>_<a.id>_<c.id>
   <package.name>        := <p.id_pack>
```

(See the definition of "Abbreviations for naming conventions") The same rules apply to testbenches. An additional rule makes sure, that the units are easily identified as testbench constituents:

```
   <e.id> := <e.id.dut>_<t.id>
```

Multiple testbenches for one DUT are created using different `<t.id>s.`

The architecture names need no longer be prefixed using <e-id> because crvmake can now handle equal architecture names, as long as they are related to different entities.

Multiple architectures for one entity are created using different `<a.id>s.`

The architecture name should indicate the abstraction level, e.g.

```
<a.id> := rtl | behavioural | dataflow | structural | verilog
```

Multiple configurations for one entity are created using different `<c.id>s.`

The configuration name may indicate the type of configuration, e.g.

```
<c.id> := c0 | test1 | leaf
```

Configurations are primary units. Their names need to be unique in the library. This is accomplished using the above-mentioned rule. Furthermore it is possible to judge from the configuration name, what entity and what architecture it refers to. Thus, simulation and synthesis jobs can be specified easier.

Should a package be related to a single unit, this can be clarified by using `<p.id> := <e.id>.` See an example for the naming conventions.

1.4.26 Naming conventions for VHDL files (a)

The following file names are recommended for storing VHDL units:

```
Entity :                  <e.id>-e.vhd
Architecture :            <e.id>-<a.id>-a.vhd
Entity & Arch. :          <e.id>-<a.id>-ea.vhd
Configuration :           <e.id>-<a.id>-<c.id>-c.vhd
Entity & Arch. & Conf. :  <e.id>-<a.id>-<c.id>-eac.vhd
Package Header :          <package.name>-h.vhd
Package Body :            <package.name>-b.vhd
Package Header & Body :   <package.name>-p.vhd
```

Note that the identifiers are separated by dashes. Identifiers themselves can contain underscores but not dashes. Thus, the reader can easily judge from the file name, where the identifiers end. SC-Highway tools do not depend on this filename convention. Arbitrary file names can be chosen, but with the disadvantage of poor readability.

Uniqueness of file names may not depend on different casing of the names. We strongly recommend that lowercase file names be generally used.

1.4.27 Naming conventions for VHDL objects (a)

Suffixes indicate the kind of a given identifier.

```
input ports :             *_i                 data_i
output ports :            *_o                 data_o
bidirectional ports :     *_b                 data_b
signals :                 *                   data
variables :               *_v                 data_v
constants :               *_c             load_mem_c
generics :                *_g            bit_width_g
functions :               *_f            gen_table_f
procedures :              *_p         check_timing_p
(sub-)types :             *_t                 stm4_t
enum. literals :          *_e          start_state_e
global signals :          *_global        TestMode_global
clock signals/ports :     *clk*|*clock*     clk155
reset signals/ports :     *res*|*rst*    cpu_reset
inverted logic object :   *_n*                  ena_ni
```

The port suffixes _i, _o, _b may be omitted on the top level of a chip or a core. At this level, the port names should equal the names that are used in the datasheet of the chip or core.

Using these suffixes helps in understanding expressions, without searching the declarations that are involved. Remember that these declarations might be located in other files even.

```
   rtpt_o <= shl_f( iir_rsum, wi_c - wo_g );
```

Thus, it is easy to tell which parameter is a signal, which is a constant, etc. When it comes to a more in-depth analysis of the types, then the declarations must be sought (e.g. using the tags-mechanism in `emacshw).`

1.4.28 Identifier naming must be based on English language (m)

This restriction will help to transfer code worldwide, e.g. for VHDL code of reusable cores or for cooperation of international project teams.

```
   load_memory    --<- preferred
   lade_speicher  --<- do not use
```

1.4.29 Use meaningful identifier names (a)

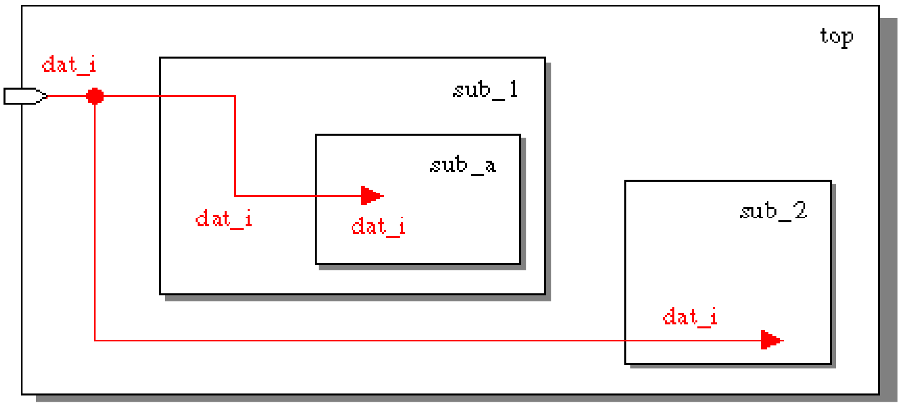
Identifier names should indicate the object's purpose, and not the type of the object:

```
   count_frames   --<- preferred
   integer_count  --<- do not use
```

Less than 15 characters are recommended. Use compound names, preferably separated by underscores ('_'). Start with the most important component and append less important components. Identifiers that are used within a wide scope should be rather long. Those being used in a narrow scope should be rather short.

1.4.30 Signal assignments for renaming purposes should be avoided (a)

The name of a signal should be the same at all places of its occurrence. This makes code more understandable. This rule shall be followed even across hierarchy boundaries as far as possible and especially for clock signals. That does not apply to signals used in a very narrow scope, because one person handles these usually.



The other reason for this rule is the following. Each signal assignment infers a delta for events traveling on this path. That means, for each event on the source a new one is scheduled on the target signal. The behaviour is correct, but consumes computation power for nothing.

1.4.31 Choose short instance names (a)

Pin- and cell-names must be handled in debuggers, waveform-viewers, SDF-files, etc. These names are built relative to the top-module using hierarchy-separators. Short instance names will turn the complete pin- and cell-names into shorter ones that are easier to handle.

```
   counter                               --<- preferred
   counter_to_count_all_incoming_ones    --<- do not use
```

1.4.32 Keyword casing should be uniform in all VHDL code (a)

Casing of keywords should be uniform in all VHDL code. Use uppercase or lowercase. Uppercase keywords are not prescribed, since visual emphasis of keywords can be achieved using color modes in the editor. Pretty printers perform this emphasis in printouts.

1.4.33 Identifier casing should be uniform in all VHDL code (r)

Mixed casing can be used for better readability of identifiers. But it must be consistent in all VHDL code. To achieve consistent casing, dynamic completion in emacshw can be used: Start typing the identifier and then type TAB. emacshw will complete the identifier with the casing, that has already been used in the file.
Entity names must be equal to the design-names used in the synthesis scripts and in the synthesis script names, including case:

```
   entity mux is ...

        mux.rscr    <-- this script gets invoked
        Mux.tscr    <-- this script is not invoked

           Inside one of the scripts:
            current_design = MUX             /* will fail */
```

Lowercase entity names are strongly recommended.

1.4.34 Separate identifier components using underscores (a)

There are two techniques to separate identifier components: Mixing case and using the underscore character ('_') within identifiers. The second technique is preferable. The advantage is that identifier components can be separated even after synthesis and backend steps. Recognizing pin-, net- or cell-names this way is a great help in case of debugging. With mixed casing, this may become impossible due to case unification by the netlist writers.

```
   data_to_mux      --<- preferred
   dataToMux        --<- do not use
```

Do not use underscore to separate numbers in identifiers:

```
   bus2, bus3, bus4       --<- preferred
   bus_2, bus_3, bus_4    --<- do not use
```

1.4.35 Trailing and consecutive underscores are not allowed in identifiers (m)

Such occurrence of underscore is misinterpreted as bus- or hierarchy-delimiter in some netlisters and netlist-readers.

1.4.36 Use an intermediate signal for reading from an output port (e)

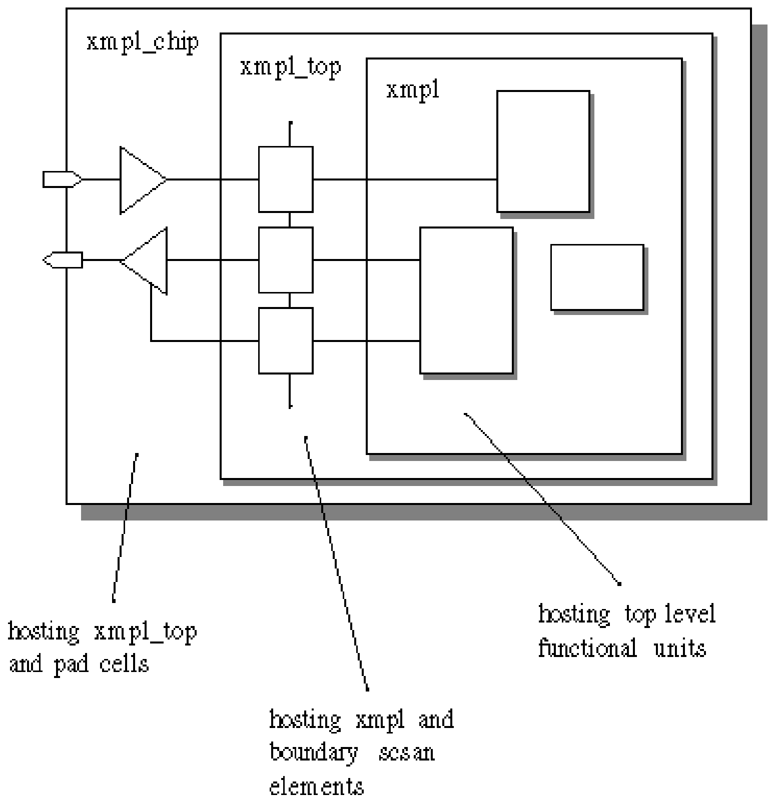
Reading directly from output ports is not possible in VHDL. Use an intermediate signal following this convention:

```
   <port-name>    = <prefix_o>
   <signal-name>  = <prefix>
Example:
   port (
     memdat_i : in  std_ulogic_vector ( 7 downto 0);
     memadr_i : in  std_ulogic_vector ( 9 downto 0);
     memdat_o : out std_ulogic_vector ( 7 downto 0) );
   ...
   architecture xmpl ...
     signal memdat: std_ulogic_vector (7 downto 0);
   begin
     ...
     check_memdat( memdat );  --<- concurrent procedure
                              --   reading output signal
     memdat_o <= memdat;      --<- driving the output port
     ...
   end xmpl;
```

Another solution for reading from output ports would be using port mode buffer. However, port mode buffer is not recommended, see "Do not use port modes buffer and linkage".

1.4.37 Use standard top-levels xmpl, xmpl_top, and xmpl_chip (r)

The three units highest in the design hierarchy must comply with the following naming rules, with xmpl being a placeholder for the actual design-name:


xmpl_chip
xmpl_top
xmpl
hosting xmpl_top and pad cells
hosting xmpl and boundary scsan elements
hosting top level functional units

The suffix _top is fix by BSDA, the boundary scan generator. It cannot be configured. If boundary scan is not applied to the chip, the unit xmpl_top is omitted.

The suffix _chip is a naming convention, which is not strict but recommended for common style in Infineon Technologies designs.

1.4.38 Length of entity names should not exceed 32 characters. (r)

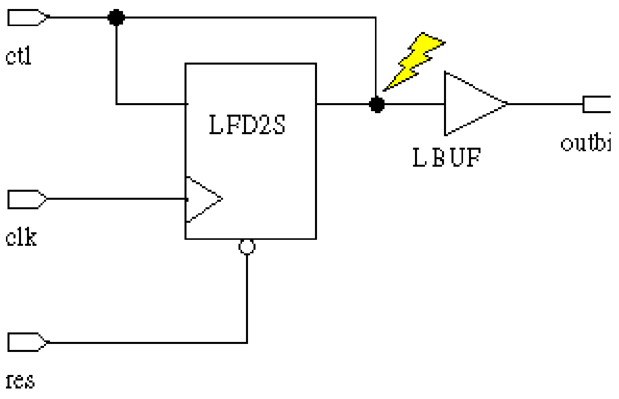
As the netlist reader of Aquarius is not able to handle more than 32 characters for design names, longer names get shorten during the write-backend-info-step in SSE. This causes problems in the links-to-layout flow, as you have to use the new design names for generating the custom-wire-load library, but for resynthesis you have to use the original names, which requires some manual corrections of the automatically generated files. You can avoid these problems by using a maximum length of 32 characters for entity names.

1.4.39 Unresolved data types std_ulogic(_vector) are recommended (a)

The motivation for this recommendation is to avoid short-circuit errors. These may easily occur if a signal of resolved type is assigned twice:

```
  reg: process (clk, res)
  begin -- reg
    if res=reset_active_c then
      outbit <= '0';
    elsif clk'event and clk='1' then
      outbit <= ctl;    --<- ----------------------,
    end if;                                         \
  end process reg;                                  |
  -- ...              --<- lots of code here        |
  outbit <= ctl;      --<- overlooked, that 'outbit' has
                      --   been assigned before
```

The result of such description is a short-circuit in the synthesized netlist as well as simulation behaviour, which is not intended:



Make sure that this situation does not occur unintentionally by checking the file $HWPROJECT/units/test/synthesis/reports/<unit>.chk_dsn for messages like

```
   *** CHECK DESIGN ***
   Warning: In design 'test', input port 'ctl' drives wired logic.
   (port-direction may have been specified incorrectly.) (LINT-6)
   Warning: In design 'test', three-state bus 'ctl' has
   non three-state driver 'outbit_reg/Q'. (LINT-34)
```

This procedure is cumbersome and insecure. Therefore unresolved logic types are recommended, which make all compilers warn about the multiple drivers, e.g. `smake -cve` issues the following message:

```
       outbit <= ctl;
       -------^
   E L35/C8:    multiple sources were detected for
                unresolved signal outbit
```

The package `ieee.std_logic_1164` has defined appropriate types:

```
   type std_ulogic is ( 'U', 'X', '0', '1',...);
   type std_ulogic_vector is array ... of std_ulogic
   function resolved (s: std_ulogic_vector )
     return std_ulogic;
   subtype std_logic is resolved std_ulogic;
```

```
type std_logic_vector is array ... of std_logic
```

`std_ulogic(_vector)` is designed for single driver, whereas `std_logic(_vector)` can handle multiple drivers. For modeling three-state drivers and for connecting bi-directional signals, resolved types may be used without danger. For correct inference of three-state-buffers the assignments must be changed compared to the code above:

```
signal a : std_logic_vector(4 downto 0); --<- use resolved
                                         --   type for 3-state-bus
...
a <= ctl_i when ena = '1' else 'Z';  --<- 3-state-driver
...
a <= mode when enb = '1' else 'Z';   --<- 3-state-driver
```

See "Avoid internal three-state busses unless absolutely necessary" for more information.

One small drawback of using unresolved types is the need for conversion functions at arithmetic operators and subprograms. Related rule: "Use conversion functions around arithmetic operators".

For single bits, however, no conversion function is needed, because `std_logic` is a subtype of std_ulogic:

```
type std_ulogic IS ( 'U', -- Uninitialized
                     'X', -- Forcing Unknown
                     '0', -- Forcing 0
                     '1', -- Forcing 1
                     'Z', -- High Impedance
                     'W', -- Weak Unknown
                     'L', -- Weak 0
                     'H', -- Weak 1
                     '-' -- Don't care
                     );
...
subtype std_logic IS resolved std_ulogic;
^^^^^^^
```

Note, that the situation is different for vectors, std_logic_vector is a separate type and std_ulogic_vector is a separate type:

```
type std_ulogic_vector is
     array ( natural range <> ) of std_ulogic;
...
type std_logic_vector is
     array ( natural range <>)  of std_logic;
^^^^
```

Therefore conversion is necessary for vectors.

The other motivation for rare use of resolved `std_logic(_vector)` is the fact that the simulator's resolution functions consume some computation power.

1.4.40 Data types at chip and macro ports: std_(u)logic(_vector) (m)

All other types (e.g. records and multi-dimensional arrays) cannot be represented accurately in Verilog. They create problems in the Highway flow and if the design shall be turned to a hard macro later. So, this rule shall serve for easier integration and re-use of chip-designs and soft macros into new chip designs. It is therefore applicable to all modules, whose re-use is forecasted. If resolved and unresolved array types are mixed (related rule: "Unresolved data types ... recommended") type conversion is necessary:

```
signal cvm2_unres : std_ulogic_vector(2 downto 0);
signal cvm4_unres : std_ulogic_vector(2 downto 0);
signal cvm2_res   : std_logic_vector(2 downto 0);
signal cvm4_res   : std_logic_vector(2 downto 0);
...
fil_blk1: rtpt18 -- component with unresolved ports
  port map (
    d_i => cvm2_unres,
    d_o => cvm4_unres,
    ...
  );

-- connecting up signals and converting types
-- using type marks

-- to fil_blk1
cvm2_unres <= std_ulogic_vector( cvm2_res   );
-- to fil_blk2
cvm4_res   <= std_logic_vector(  cvm4_unres );

fil_blk2: cvmtv18
  port map (
    d_i => cvm4_res,
    d_o => cvm2_res,
    ...
  );
```

Type marking as an easy form of type conversion is possible between closely related array types, i.e. type that are built on the same base type, std_ulogic in our case. Type marks do not consume simulation performance. They are resolved at compile time.

Note, that type marking is not necessary to connect std_ulogic with std_logic (single bits), because the latter is a subtype of the former:

```
fil_blk1: rtpt18
  port map (
    ...
    flag_bit_o => flag_bit, -- to fil_blk2 ------,
    ...                     --                    \
  );                        --                    |
                            --                    |
fil_blk2: cvmtv18           --                    |
  port map (                --                    |
    ...                     --                    /
    flag_big_i => flag_bit, -- from fil_blk1 <---'
    ...
  );
```

The connecting signal flag_bit can be either of std_logic or std_ulogic type. This does not work with std_(u)logic_vector, because arrays cannot be declared as subtypes.

```
ieee.std_logic_1164 recalled:
   subtype std_logic is resolved std_ulogic;
   type std_logic_vector is array (natural range <>)
     of std_logic;
```

Note that type conversion can also be performed directly in the port map. An explicitly declared function must be used instead of the type mark. Additionally this function must not expect more than one parameter:

```
   fil_blk3: rtpt18
     port map (
       ...
       d_i => to_stdulogicvector( cvm2 ), --<- input dir.
       to_stdlogicvector( d_o ) => cvm4,  --<- output dir.
       ...
     );
```

The functions used stem from the package `ieee.std_logic_1164`. For the input direction the actual parameter gets converted, for the output direction the formal parameter gets converted. This technique cannot be used at outputs, if the code is going to be synthesized. Therefore, we recommend to use separate type marked assignment statements generally, as described above.

Related rule: "Data types at module ports: std_(u)logic(_vector) recommended".

1.4.41 Data types at module ports: std_(u)logic(_vector) recommended (a)

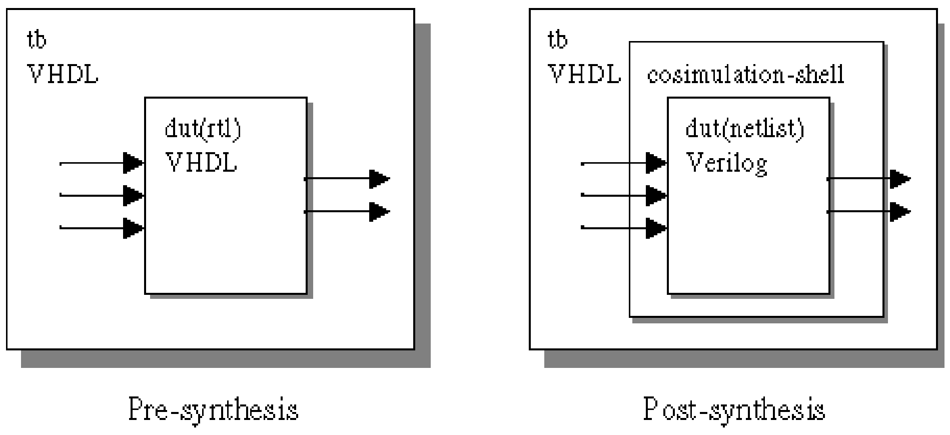
The strict rule "Data types at chip and macro ports: std_(u)logic(_vector)" does not apply to the following modules:

Modules developed and maintained by one designer

Modules developed and maintained by a design team, which is working closely together

At these interfaces, other data types than `std_(u)logic(_vector)` may be used as well. Make sure that these are synthesizable data types, if the module is going to be synthesized (related rule "Data types must be synthesizable"). This is also important if the module architecture is behavioural, but a synthesizable version is planned.

If the module will be synthesized as a sub-block, any of the types described in rule "Data types must be synthesizable" may be used. However, if the synthesized netlist of this module shall be verified against its RTL or behavioural description, then `std_(u)logic(_vector)` is advisable from the beginning. If this type is chosen for ports, then the automatic shell generation facility can be used. This shell allows for simulating the module together with the existing testbench or in the RTL surroundings.



If types other than std_(u)logic(_vector) are used for ports, the co-simulation shell must be created by hand. Conversion functions must be used to connect up the netlist.

Important to note, that this recommendation addresses port types only. Inside architectures, you are advised to make extensive use of abstract data types (related rule: "Use complex data types inside architectures"). This is also possible to a wide extent, even within the synthesis subset (records, integer, enumeration types ...).

1.4.42 Use complex data types inside architectures (a)

Complex or abstract data types are e.g. complex numbers for signal samples, record structures for data frames, arrays of arrays for memories, etc. These types are preferred in architectures for several reasons:

Abstract types make the model more compact and thus easier to understand

Therefore the model is better maintainable and re-usable

The model will simulate faster. E.g. integer operations are carried out on the workstation processor directly.

For circuit models that will be synthesized

Make sure that abstract types match the synthesis subset, e.g. linked lists are not synthesizable.

Use range constraints on integers, otherwise 32 bit will be synthesized

Use integers as long as bit operations do not dominate the functionality. Otherwise make use of the package `ieee.std_logic_arith` and the types `signed and unsigned` therein.

To connect up to the module ports type conversions are required:

```
port (
  ctrl_i : in std_ulogic_vector( wctrl_c-1 downto 0 );
...
signal ctrl : ctrl_record_t;
...
ctrl <=                                  --<- input
  conv_ctrl_vector_to_record_f( ctrl_i ); --<- conversion
...
sig1 <= sigvec( ctrl.select );  --<- inside arch. use
                                --   complex types
```

For closely related array types, type marks may be used.

Related rule: "Data types must be synthesizable".

1.4.43 Use a small number of self-defined global data types (a)

Global types are types being used by more than one designer. Such types must be well maintained. I.e. documentation, conversion functions and other subprograms working on this type need to be provided, typically through packages. This project-wide work can only be done for few such types.

This recommendation does not apply to local types. Types can be local to architectures, processes, functions, etc.:

```
ctrl: process (clk, res)
```

```
      type state_t is                      --<- local to
        ( RESET, WAKEUP, SYNCD, INT );    --   process
      function upd_mem_f ( addr : integer )
        return integer is
        type memstate_t is     --<- local to function
          ( FILLED, INITD );
      begin
        ...
      end upd_mem_f;
   begin
     ...
```

Local types should be used intensively for an increase of readability. For description of states use enumeration types rather than logic types, as `std_ulogic_vector`.

1.4.44 Self-defined conversion functions must assert un-mappable input values (m)

A constant (`conv_unmappable_messages_off_c ) may be used to disable assertions`

```
   function hex_to_int_f (variable c_v : in character)
     return integer is
     variable res_v: integer;
   begin
     case c_v is
       when '0'    => res_v :=  0;
       when '1'    => res_v :=  1;
       ...
       when 'E'    => res_v := 14;
       when 'F'    => res_v := 15;
       when others => res_v :=  0;                 --<-
         assert conv_unmappable_messages_off_c     --<-
           report "hex_to_int_f ERROR: " &         --<-
                  "char. not valid, mapped to 0."  --<-
           severity error;                         --<-
     end case; -- c_v
     return res_v;
   end hex_to_int_f;
```

The assert statement is activated, when the condition `(conv_unmappable_messages_off_c) evaluates to false. Choose false as default value for conv_unmappable_messages_off_c.`

1.4.45 Multidimesional arrays can lead to mismatch between TRF file and Verilog netlist. (e)

Multidimensional arrays lead to escaped names in the Verilog netlist, as there are no multidimensional arrays in Verilog, e.g.:

```
\bus_xy[2][4]
```

If you have set constraints on such an array, this name will also appear in the TRF file. During the write-backend-info-step in SSE the bus naming style in the TRF file is changed from [] to <>. Unfortunately – due to a bug in SSE – the names of the multidimensional arrays are changed, too. This leads to a mismatch between the TRF file and the Verilog netlist. Until this is fixed in a future release of SC Highway, you should not use multidimensional arrays. If you use them, you will only run into this problem, if you are setting constraints on this array. In this case, you will have to correct the TRF file by hand.

See also:

Data types at chip and macro ports: std_(u)logic(_vector)

Data types at module ports: std_(u)logic(_vector) recommended

1.4.46 Avoid gated clocks unless absolutely necessary (r)

Logic within the clock line may be used under two circumstances, only:

Clock gating for power saving.

Static clock multiplexing for fabrication test.

Instances of clock logic must be communicated precisely to the layout engineer for proper clock tree synthesis. Rare usage of gated clocks is recommended due to the fact that handling of these gates in clock tree synthesis is critical wrt. accuracy and involves hand-crafting, which is always error-prone.

At the modeling stage, special care is necessary, since delta races easily infer functional pre-/post-synthesis mismatches. Such mismatches are detected by formal comparison (CVE) or proper simulation pattern. Please refer to the related rule: "Balance clock to delta accuracy".

Outlook: Most probably there will be a technique available in the future to infer gated clocks where the clocked processes have enable-behaviour:

```
enreg: process (clk, res)
begin
  if res=reset_active_c then
    outbit <= '0';
  elsif clk'event and clk='1' then
    if enable then
      outbit <= ctl;
    end if;
  end if;
end process enreg;
```

Such circuit description needs not take into account clock gates. An elaboration switch allows for propagation of the enable into the clock path. Thus, clock gates can be inferred for a power-saving implementation, whereas another implementation utilizes the enable directly. Both implementations are derived from the same VHDL source. However, it must be clarified whether inference works with Starlib, and how to handle the multitude of inferred clock gates in place & route.

1.4.47 Avoid latches unless absolutely necessary (r)

Testability problems are likely to occur with latches. For ATPG, latches need to be switched to transparent mode, in order to allow for scan testing the rest of the circuit. For the latch itself dedicated test-vectors need to be developed, which is not an easy task.

Unintended latch-inference may happen with combinational processes that do not drive their output signals under every possible condition:

```
  xmpl_sig: process( mode0, sig )
  begin
    if mode0 = "00" then
      sig <= sig1 + 1;
    elsif mode0 = "01" then
      sig <= 12;   -- cases mode0 = "10", "11" not covered
    end if;        -- could be covered in an else clause
  end process xmpl_sig;
```

Check the elaboration logfile `$HWPROJECT/units/*/synthesis/logfiles/<unit>_elab.log` to recognize this situation:

```
 Inferred memory devices in process 'xmpl_sig'
     in routine test line 92 in file
            '/home/hwpro/hw_alarm_chip/vob/units/test/...
                   ...vhdl/rtl/test-rtl-a.vhd'.

  ===========================================================
  | Reg.Name | Type  | Width | Bus | AR | AS | SR | SS | ST |
  ===========================================================
  | sig_reg  | Latch |   4   |  Y  | N  | N  | -  | -  | -  |
  ===========================================================

  sig_reg (width 4)
  -----------------
      reset/set: none
```

Furthermore there is a warning issued by ssemake:

```
 Elaboration of /home/hwpro/hw_alarm_chip/vob/project_lib/gdbs/test.db
  finished on erle at Thu Dec 18 19:13:53 MET 1997
       Warning:        Latches are inferred during elaborate  (SSE).
       Check logfile:  logfiles/test_elab.log  (SSE).
```

To remove the latch, you may add an else-clause:

```
  xmpl_sig: process( mode0, sig )
  begin
    if mode0 = "00" then
      sig <= sig1 + 1;
    elsif mode0 = "01" then
      sig <= 12;
    else         --<- else clause
      sig <= 0; --<- prevents latch inference
    end if;
  end process xmpl_sig;
```

or introduce a default assignment:

```
  xmpl_sig: process( mode0, sig )
  begin
    sig <= 0;   --<- default assignment
                --   prevents latch inference
    if mode0 = "00" then
      sig <= sig1 + 1;
    elsif mode0 = "01" then
      sig <= 12;
    end if;
  end process xmpl_sig;
```

1.4.48 Reset is mandatory for sequential processes (m)

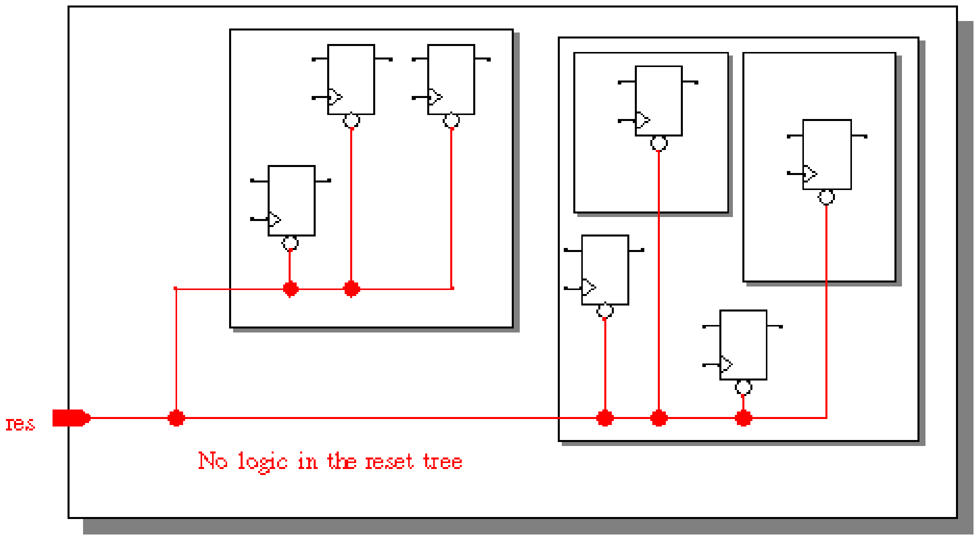
Reset at sequential processes lets synthesis infer reset-able flip-flops into the netlist:

```
  reg: process (clk, res)
  begin -- reg
    if res=reset_active_c then    --<- reset branch
      outbit <= '0';              --<- reset branch
    elsif clk'event and clk='1' then
      outbit <= ctl;
    end if;
  end process reg;
```

This is an important prerequisite for rapid resetting of the whole circuit at fabrication test, for ATPG and for simulation initialization.

There are several advantages for asynchronous reset and several for synchronous. The complete discussion is not unrolled here.

Asynchronous reset is recommended for fabrication test without any logic in the reset tree. Thus the reset signal is not timing-critical. Clock can be disabled on the tester during the reset process. Under these circumstances the reset tree can be buffered in synthesis.

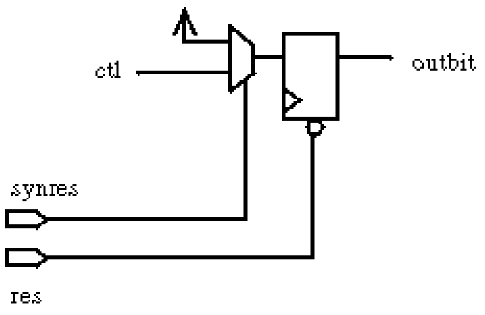


With the respective template given under "Use standard templates for clocked processes" it is granted that the reset tree is connected to the respective reset pins of the flip-flops. Check the elaboration logfile `$HWPROJECT/units/*/synthesis/logfiles/<unit>_elab.log` to make sure that the desired set or reset has been implemented:

```
  Inferred memory devices in process 'reg'
     in routine test line 79 in file
           '/home/hwpro/hw_alarm_chip/vob/units/test/...
              vhdl/rtl/test-rtl-a.vhd'.
  ==========================================================
  | Reg.Name | Type | Width | Bus | AR | AS | SR | SS | ST |
  ==========================================================
  | outb_reg | FF   |   1   |  -  | Y  | N  | N  | N  | N  |
  ==========================================================
                                  ^^^^^^^^^^^^^^^^^^^^^^^
  outb_reg
  --------
      Async-reset: res'
      ^^^^^^^^^^^
```

In-system reset functionality be separated from the above-mentioned reset for fabrication test. It can be combined with the other functional logic:

```
  reg: process (clk, res)
  begin -- reg
    if res=reset_active_c then   --<- async. reset for
      outbit <= '0';             --<- fabrication test
    elsif clk'event and clk='1' then
      if synres=synres_active_c then  --<- sync. reset for
        outbit <= '1';                --<- system operation
      else
        outbit <= ctl;
      end if;
    end if;
  end process reg;
```

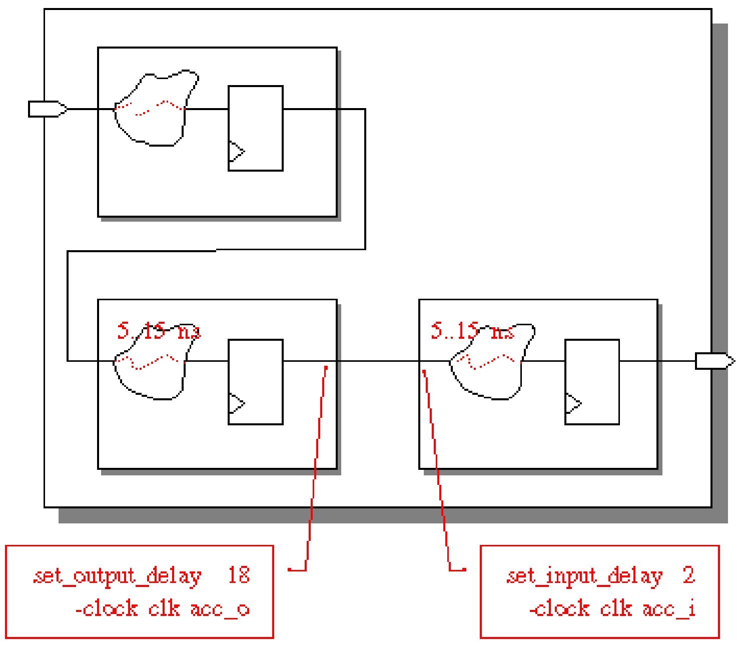


Note that the multiplexer is merged with the other input logic by synthesis, which is not described here.

Refer to rule "Use standard templates for clocked processes" for an example of asynchronous and synchronous reset.

1.4.49 All outputs of synchronous modules should be registered (r)

This makes time budgeting much easier. Timing constraints are to be defined at the unit boundary. Thus the surroundings of the currently synthesized unit are described. A standard constraint set can be applied if the neighbour units have registered outputs: The outer input delay-budget can be modeled quite realistically to 2 ns. The outer output delay-budget is best modeled to `clock-period - 2 ns.`



Realistic modeling of the surroundings of a unit being synthesized is important for the synthesis process. The created logic is adapted to these assumptions. If it turns out that these assumptions were not right, problems will occur, when connecting the units.

Refer to rule "The sensitivity list of clocked processes..." for an example of clocked processes.

1.4.50 Use gate instantiation only at few instances (r)

It is desirable to create circuit descriptions independent from technology, as far as possible. Thus transition to next generation technologies is made easier and re-usability in other projects is improved. However instantiation is necessary in the following cases:

- Pad-instantiation
- High-performance RAM-control
- Clock-gating
- IO-timing adjustment
- Bus-holding
- etc.

There are different ways to instantiate gates and cells:

Option 1: Instantiate gtech gates, technology independent, Synopsys-dependent. This procedure is not preferred due to the dependence on Synopsys.

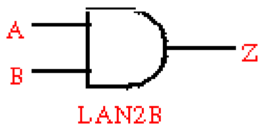
```
library gtech;                  --<- make comp.
use gtech.gtech_components.all; --<- decl.avail.for
                                  --   instantiation
entity inst_xmpl is
  ...
```

```
end inst_xmpl;

architecture struc of inst_xmpl is
  ...
  add: gtech_add_abc port map(  --<- instantiation
           a    => a,
           b    => b,
           c    => c,
           s    => s,
           cout => car);
  ...
end struc;

library gtech;  --<- make entity and arch.
use gtech.all;  --<- available for default binding

configuration inst_xmpl_struc_cdef of inst_xmpl is
  for struc
    -- binding omitted for instance "add"
    -- default binding allowed, because only
    -- one architecture available in library gtech.
  end for;
end inst_xmpl_struc_cdef;
```

Option 2: Bind a specific cell or design to a function or procedure. Thus it is possible to not only force the type of the cell but as well the drive strength. This technique is necessary in extremely rare cases. In general the optimization of Design Compiler much better determines the type and drive strength of the required cells. However, if this technique has been applied, make sure that Design Compiler does not re-arrange the design, or resize the driver strengths. This is best achieved when the cell-references are placed in a separate entity, for which the command compile is then inhibited. link is sufficient.

```
architecture rtl of test is

  function and_f ( A, B : std_ulogic ) --<- *
    return std_ulogic is
    -- pragma map_to_entity LAN2B      --<- *
    -- pragma return_port_name Z       --<- *
    -- * = case sensitive cell and pin names
  begin -- and_f
    return a and b;
  end and_f;

begin  -- rtl

  d <= and_f(x,y);

end rtl;
```

Note that cell/design-name and pin-names must have the same casing as in the Synopsys Design Compiler, i.e. `"LAN2B", "A", "B", "Z" in the given example.`



The technique must be used with extreme care. Inconsistencies are very likely to happen, because the VHDL-function and the cell function are specified in different places (VHDL and library). Netlist versus RTL comparison is absolutely mandatory, e.g. by using formal verification (CVE).

The advantage of this technique is independence from the VHDL-simulation models of the cells. It might therefore be used in models, which must be exchanged with external cooperation partners, who do not have the respective library in-place.

Option 3: Instantiation of starlib- or padlib cells. As with the previous technique, cell-references must not be mixed with other RTL-code. Pin- and cell-name casing must not necessarily match, because the VHDL analyzer resolves the reference.

```
   library starlib;
   use starlib.starlib_components.all;

   entity test is

     port ( a_i : in    std_ulogic;
            b_i : in    std_ulogic;
            z_o : out   std_ulogic );

   end test;

   architecture rtl of test is
   begin  -- rtl

     and : LAN2B port map(
           a => a_i,
           b => b_i,
           z => z_o);

   end rtl;

   library starlib;
   use starlib.all;

   configuration test_rtl_cdef of test is
     for rtl
       -- binding omitted for instance "and"
       -- default binding allowed, because only
-- one architecture available in starlib
     end for;
   end test_rtl_cdef;
```

This is the technique of choice for pad-instantiation.

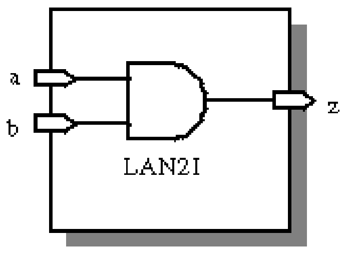
Option 4: Using boolean operators in the VHDL code to force a certain logical structure:

```
   architecture rtl of test is
   begin  -- rtl

     z <= (a or b) xor b;
   end rtl;
```

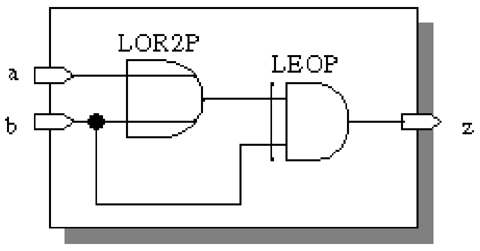
To avoid logic restructuring, the following scripts are required under `$HWPROJECT/units/<unit>/synthesis/scripts` .

```
   <unit>.rscr:
   set_dont_touch find(cell, "*")

   <unit>.iscr:
   remove_attribute find(cell, "*") dont_touch
   compile -incre
```

Without these scripts logic optimization produces one `LAN2I:`



whereas with the scripts the original structure is resembled: `LOR2P, LEOP:`



The drive capabilities are optimized according to the given constraints.

1.4.51 Avoid internal three-state busses unless absolutely necessary (r)

Internal three-state signals are difficult to handle at fabrication test and during constraining for logic synthesis. An alternative is using a multiplexer solution. Depending on the number of channels, this may result in high net areas and problems during routing. Use internal three-state busses only if such routability problems are obvious.

Generate three-state buffers using the following concurrent statement (template):

```
  bus <= ram_dat when ena = '1' else (others => 'Z');
```

Furthermore, instantiate a busholder-cell using option 2 or 3 described in "Use gate instantiation only at few instances".

At the moment 'Z' must be changed temporarily to '0' for processing with CVE. This is no longer necessary for SC-Highway V1.0.1 and higher (new CVE version).

1.4.52 Component instantiation (e)

VHDL provides different ways of instantiating components. In the following, there are some examples (all of them can be compiled for Leapfrog without the -compat switch):

Instance of component out of library WORK:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity inst_xmpl is
  ...
end inst_xmpl;


architecture struc of inst_xmpl is
  ...
  add: add_abc port map(
           a    => a,
           b    => b,
           c    => c,
           s    => s,
           cout => car);
  ...
end struc;


configuration inst_xmpl_struc_conf of inst_xmpl is
  for struc
    for add|all: add_abc
      use configuration work.add_abc_rtl_conf;
    end for;
  end for;
end inst_xmpl_struc_conf;
```

Instance of component out of library XY_LIB (e.g. starlib, padlib, memlib etc.) without configuration:

```
library IEEE;
use IEEE.std_logic_1164.all;
library xy_lib
use xy_lib.components.all;
entity inst_xmpl is
  ...
end inst_xmpl;


architecture struc of inst_xmpl is
  ...
  inst: macro port map(  -- or xy_lib.components.macro port map(
           a    => a,
           b    => b,
           c    => c,
           s    => s,
           cout => car);
  ...
end struc;


library xy_lib;
```

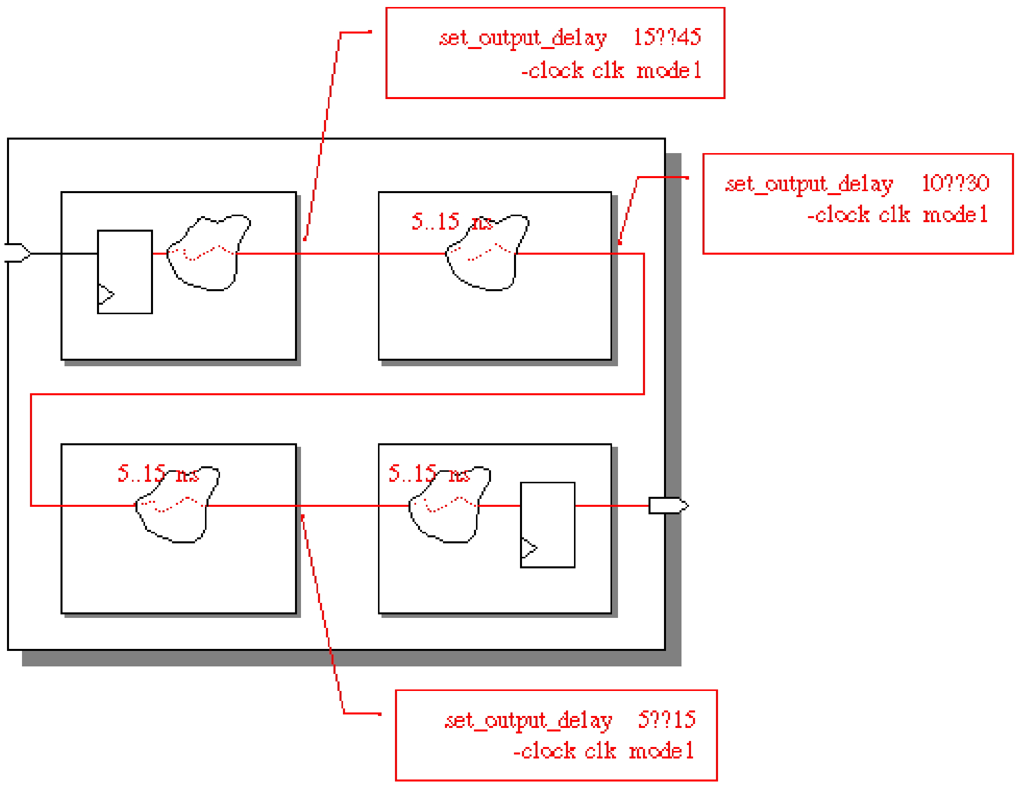
```
use xy_lib.all;

configuration inst_xmpl_struc_conf of inst_xmpl is
  for struc
  end for;
end inst_xmpl_struc_conf;
```

Instance of component out of library XY_LIB (e.g. starlib, padlib, memlib etc.) with configuration:

```
library IEEE;
use IEEE.std_logic_1164.all;
library xy_lib
use xy_lib.components.all;
entity inst_xmpl is
  ...
end inst_xmpl;


architecture struc of inst_xmpl is
  ...
  inst: macro port map(  -- or xy_lib.components.macro port map(
           a    => a,
           b    => b,
           c    => c,
           s    => s,
           cout => car);
  ...
end struc;



configuration inst_xmpl_struc_conf of inst_xmpl is
  for struc
    -- pragma translate_off
    for inst|all: macro
      use configuration xy_lib.inst_rtl_conf;
    end for;
    -- pragma translate_on
  end for;
end inst_xmpl_struc_conf;
```

1.4.53 Avoid snake paths (r)

Snake paths are timing paths that travel through many units in the design hierarchy without being registered. Such paths must be avoided, because they are error prone and difficult to constrain appropriately.



This problem does not occur, if module outputs are registered in general. However, this is probably not feasible for all outputs. If there are only few exceptions (max. 10 in a design), the risk is acceptable.

1.4.54 Instantiate the power pads for each power domain (r)

The power pads should be instantiated in the VHDL code for each power domain. Nevertheless, don’t define ports in advance, because this causes problems in Apollo. In Aquarius there is a bug, where no netlist can be written out without PG ports and so there is an inconsistency compared to the synthesis results, when the PG ports are not defined in the VHDL. For this reason, there exists a script provided by HW LG (Highway Layout Generation) to remove these additional ports.

1.4.55 Insert test-enable for scan-test (m)

The scan-test enable-pin needs to be defined as a port in advance at all units that will be simulated separately as a gate-level-block. Usually the pin name is `test_se`. The same applies to each scan-chain input and output, if these pins cannot be multiplexed to other pins.

1.4.56 Refer to the rising edge of clock (r)

in order to avoid inverters in the clock line (Starlib contains positive edge triggered flip-flops only). Falling-edge-triggered processes should be used as an exception and only if absolutely necessary. An inverter is inferred in the clock line, which must be handled separately in the routing process, which is error-prone.

1.4.57 Do not use combinational feedback loops (m)

Combinational feedback loops are often the reason of problems in tools relying on statical timing informations. Especially designs which are part of an hierarchical design flow or macros should avoid such timing arcs are not characterizable in a later abstraction step.

1.4.58 Spacer cells needed between pads (e)

In the padlib, there are different spacer cells available, which must be placed between pads. There are analog spacer, digital spacer and spacer cells with diodes, which have to be placed between an analog and a digital pad. The safest way to get these cells in the netlist is to instantiate them in the VHDL code, but you have to clarify how much and which spacer cells you need. Nevertheless, it is also possible to add the spacer cells during layout.

1.4.59 Spend extensive efforts for documentation of interfaces (r)

Interface problems are the most frequent source of simulation and even chip failure. Comments can be used for documentation of interfaces. Avoid duplication of comments, put them into the entity, and do not repeat in the architecture. Duplication often leads to inconsistencies or increased maintenance effort.

1.4.60 Build interface consistency checks into the model (a)

Asynchronous interfaces must be equipped with timing checks. Use passive processes for this purpose. A passive process is a process statement where neither the process itself, nor any procedure of which the process is parent, contains a signal assignment statement. Passive processes in the entity statement part can be used for timing or other checks:

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

entity checkit is
  port (
    clk   : in  std_ulogic;
    res   : in  std_ulogic;
    dat_i : in  natural range 0 to 15;
    ctl_i : in  std_ulogic;
    dat_o : out natural range 0 to 31
    );
begin
  check_tim: process (clk, dat_i)
    variable dat_last_value_v  : integer := -1;
    variable dat_last_change_v : time    := 0 ns;
  begin
    -- remember value and time, if change occured
    if dat_i /= dat_last_value_v then
      dat_last_value_v  := dat_i;
      dat_last_change_v := now;
    end if;
    if clk'event and clk='1' then
      assert (now - dat_last_change_v) > 2 ns
        report "Error: Setup violation on dat_i"
        severity error;
    end if;
  end process check_tim;

  -- dynamic range constraint checked:
  -- (range narrowed when ctl_i='1')
  check_range:
    assert ctl_i='0' or dat_i<8
      report "Error: Range violation on dat_i while ctl_i='1'"
      severity error;
```

end checkit;

This technique has the other big advantage that checks are carried out equally for all architectures of this entity.

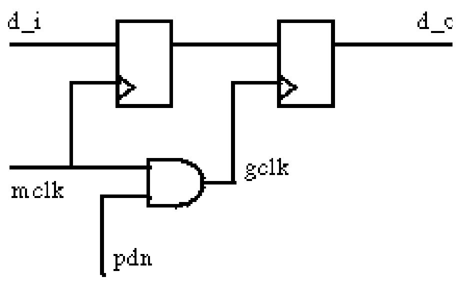
Note that the entity part is analyzed for synthesis by ssemake. Of course, passive processes cannot be synthesized. Therefore, analyze issues the warning

```
SPC_warning: Statements in an entity declaration
  are not supported for synthesis. They are ignored
  in entity checkit on line 4   (VHDL-2001)
```

that can be skipped safely. It is **not** necessary to mask passive processes via -- pragma synthesis_off/on.

1.4.61 Balance clock to delta accuracy (m)

Gated clocks or clock assignments imply a delta-delay in the RTL model. This corrupts the memory function of serially connected register processes (shift-registers).

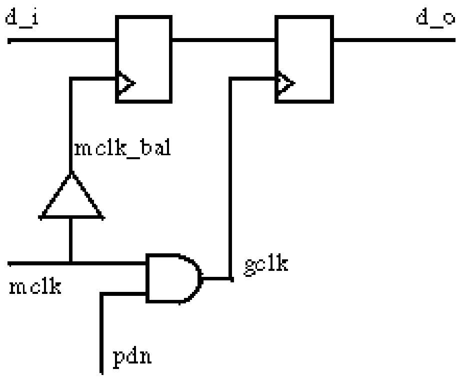


(Note that the and-gate is used here for simplification. In your design, a different gating logic may be appropriate for spike suppression etc.)

If the receiving process is clocked by a gated clock gclk, whereas the sending process runs with the master clock mclk, then the clock-event is recognized before the corresponding data-event. There exists a "hold time violation" at the scale of delta-delays:

```
r1: process (mclk,
  ...
  elsif mclk'event and mclk='1' then
    dat <= d_i;
  end if;
end process r1;

gclk <= mclk and pdn;       --<- this infers on delta!

r2: process (gclk,
  ...
  elsif gclk'event and gclk='1' then
    d_o <= dat;
  end if;
end process r2;
```

"Balancing the clock tree" at delta accuracy restores the desired functionality:



```
mlck_bal <= mclk;    --<- this infers one balancing delta!

r1: process (mclk_bal,
  ...
  elsif mclk_bal'event and mclk_bal='1' then
    dat <= d_i;
  end if;
end process r1;

gclk <= mclk and pdn;    --<- this infers one delta!

r2: process (gclk,
  ...
  elsif gclk'event and gclk='1' then
    d_o <= dat;
  end if;
```

```
   end process r2;
```

The other alternative to cope with these delta races was delaying all registered signals by 1 ns. This would correspond to the actual circuit behaviour. Only the delay value cannot be accurate at all. Hold time violations would be removed that way. This strategy has not been chosen, since CVE (formal verification) does not recognize the delay.

Related rule: "Avoid gated clocks".

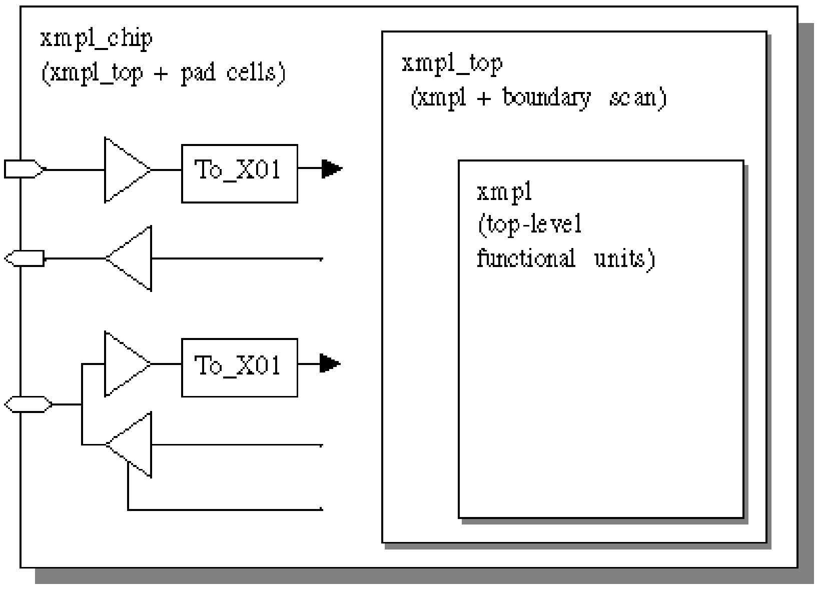
1.4.62 Pull-ups and pull-downs have to be modeled on chip level (r)

Pull-behaviour is necessary for system-simulations, where multiple chips are instantiated on a board. This rule is obsolete if correct IO-cells have been instantiated. Pull-behaviour is modeled in the IO-cells.

1.4.63 Strength stripping should be performed on chip level (r)

The recommended logic type is std_ulogic (see "Data types at chip and macro ports: std_(u)logic(_vector)"). This type is nine-valued, but inside chip- and unit-models only two or three of them are used: '0', '1' and 'Z'.

A testbench or a system simulation model might use weak values as well, which then cause malfunction of the models under development. Therefore these weak values should be translated to 'x', '0' and '1' at the chip level for all inputs. The translation should be performed directly behind the pad. At this point bidirectionals are split to input, output and enable.



1.4.64 Do not assign the value 'X' (r)

The unknown level `'X' is used in standard cell models to warn about setup- and hold-violations. However, in RTL-models timing is not analyzed at that fine scale. Timing-checks are only performed at asynchronous interfaces. Exceptions at these interfaces and other exceptions should be handled using assertions rather than 'X'-assignments:`

```
if sla=3 then
  syna <= "001";
else
  syna <= "XXX";  --<- don't use
  assert false    --<- assertion preferred
    report "unexpected value on signal sla"
    severity warning;
end if;
```

1.4.65 Use efficient description for the wrap around of counters (a)

1. Example: Full-coded counter

```
inefficient1: process (clk)
begin –– inefficient1
  if clk'event and clk = '1' then
    if count = 255 then
      count <= 0;
    else
      count <= count + 1;
    end if;
  end if;
end process inefficient1;
```

Hardware will be synthesized to correspond to the if statement. This extra hardware is not necessary because hardware naturally wraps around from 255 to 0. But for simulation purposes you have to take the wrap around into consideration, else simulation will have a run-time error when the incremented value reaches 256. So use the modulo operator:

```
efficient1: process (clk)
begin –– efficient1
 if clk'event and clk = '1' then
      count <= (count + 1) mod 256;
  end if;
end process efficient1;
```

Extra mod related hardware will not be synthesized because the synthesis algorithms understand the implied wrap around semantics of this model.

Another solution is the use of an array:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned;
entity counter is
  port (clk    : in  std_logic;
        data_o : out std_logic_vector(0 to 7)
       );
```

```
end counter;
---------------------------------------------
...
efficient2: process
begin -- efficient2
  if clk'event and clk = '1' then
    count <= count + '1';
  end if;
end process efficient2;
data_o <= count;
```

The overloaded function for unsigned “+” and the synthesized hardware will both wrap around from FF(hex) to 00(hex).

2. Example: Not full-coded counter

```
inefficient2: process (clk)
begin –– inefficient2
  if clk'event and clk = '1' then
    if count = 8 then
      count <= 0;
    else
      count <= count + 1;
    end if;
  end if;
end process inefficient2;
```

The comparison of count with 8 for the wrap around causes too much hardware, because equality is required. Also there are unused states (9 – 15), which cause uncertainties in the real hardware. So the following description is safer and needs less gates:

```
efficient3: process (clk)
begin –– efficient3
  if clk'event and clk = '1' then
    if count >= 8 then
      count <= 0;
    else
      count <= count + 1;
    end if;
  end if;
end process efficient3;
```

1.4.66 Use case-statements instead of if-statements when the conditions are mutually exclusive (a)

Design Compiler will handle a case statement more efficiently in terms of CPU time than an if … elsif statement because by definition the when conditions are always mutually exclusive.

```
inefficient: process (sel, a, b, c, d)
begin -- inefficient
  if sel = “00” then
    d_out <= a;
  elsif sel = “01” then
    d_out <= b;
  elsif sel = “10” then
    d_out <= c;
 else
    d_out <= d;
  end if;
  end process inefficient;
```

Design Compiler is capable of deducing that the if ... elsif conditions are mutually exclusive but precious CPU time is required for this analysis.

```
efficient: process (sel, a, b, c, d)
begin -- efficient
  case sel is
    when “00” => d_out <= a;
    when “01” => d_out <= b;
    when “10” => d_out <= c;
    when “11” => d_out <= d;
  end case; -- sel
end process efficient;
```

1.4.67 Use vector operations instead of loops (a)

Design Compiler has to unroll the following loop:

```
inefficient: process (scalar_a, vector_b)
begin -- inefficient
  vec_loop: for k in vector_b’range loop
    vector_c(k) <= vector_b(k) and scalar_a;
  end loop vec_loop;
end process inefficient;
```

Instead of such a loop, use vector operations, so Design Compiler works more efficiently:

```
efficient: process (scalar_a, vector_b)
  variable temp_v: std_logic_vector(vector_b’range);
begin –- efficient
  temp_v := (others => scalar_a);
  vector_c <= vector_b and temp_v;
end process efficient;
```

1.4.68 Use boolean expressions instead of simple if-statements (a)

```
if_proc: process (enable, data_i)
begin –- if_proc
  if enable = '1' then
    data_o <= data_i;
  else
    data_o <= '0';
  end if;
end process if_proc;
```

Design Compiler has to explore how to handle the if statement, so use a boolean expression instead of the if-statement:

```
bool_proc: process (enable, data_i)
begin –- bool_proc
  data_o <= enable and data_i;
end process bool_proc;
```

1.4.69 Avoid unnecessary repetition of function calls (a)

```
a <= function1(input1, input2)(31)
b <= function1(input1, input2)(3 downto 0)
```

function1 is called twice with the same inputs. This is inefficient in the simulation domain because the repeated invocations of the function are redundant and waste CPU time; also during synthesis. In other aspect is that resource sharing isn’t possible, because each function call is viewed as being independent. So, use a temporary variable:

```
temp_v := function1(input1, input2);
a <= temp_v(31);
b <= temp_v(3 downto 0);
```

1.4.70 Avoid unnecessary computations within loops (a)

```
loop_bad: for k in 1 to 7 loop
  if k > (a - 1) then
    s(k) <= '1';
  else
    s(k) <= '0';
  end if;
end loop_bad;
```

The computation of (a – 1) is constant for all iterations of the loop. Design Compiler will unroll the for loop and deduce that the computation may be relocated outside of the for loop. So one subtractor and seven comparators will be used by Design Compiler. However, the analysis for this deduction is time consuming, which can be avoided by using a temporary variable:

```
temp_v := a - 1
loop_better: for k in 1 to 7 loop
  if k > temp_v then
    s(k) <= '1';
  else
    s(k) <= '0';
  end if;
end loop_better;
```

However, the best solution is to avoid also the subtractor:

```
loop_best: for k in 1 to 7 loop
  if k + 1 > a then
    s(k) <= '1';
  else
    s(k) <= '0';
  end if;
end loop_best;
```

Remember that when this for loop is unrolled, k is a fixed value on each iteration of the loop. Consequently, k + 1 can be synthesized as the fixed values 2, 3, 4, 5, 6, 7, 8.

1.4.71 Avoid declaring constants within subprograms (a)

For local constants within subprograms memory must be allocated at invocation and must be deallocated when control returns from the subprogram which can be very expensive in terms of simulation performance, in particular if the constant happens to be a table.

1.4.72 Be aware of the use of the attributes DELAYED, STABLE, QUIET, and TRANSACTION (a)

Expressions using these attributes will create additional signals and will degrade performance.

1.4.73 Minimize the number of signals of your sensitivity list (a)

Minimize the number of signals of your sensitivity list so that models are only called when a reevaluation of the outputs is required. But consider that processes which desribe combinational logic will require all inputs in the sensitivity list (The sensitivity list for combinational processes must be complete). On the other hand, synchronous logic requires the clock signal only (Use standard templates for clocked processes).

1.4.74 Always use deallocate(access_obj) to dereference allocated memory (r)

This is especially necessary for pointers declared in subprograms. So, functions must not return the contents of dynamically allocated memory, because this prevents the deallocation of that memory within the function body and thus leads to memory leaks. The following function gives an example for memory leak. If such a function is frequently invoked, this may eventually lead to a simulation failure due to lack of virtual memory.

```
function mem_leak (xyz: integer) return string is
variable buffer_v: line;
begin
  write(buffer_v, xyz);
  return(buffer_v.all);
end mem_leak;
```

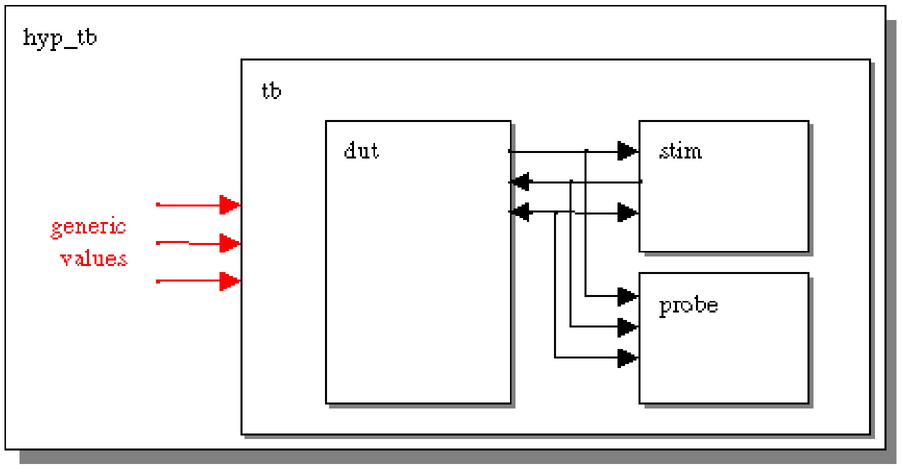
1.4.75 Do not use FSM attributes within the VHDL code (m)

By using the in built Synopsys FSM optimization routines, you can optimize FSMs in a design to achieve maximum performance. The optimization commands for a state machine should be placed in the <unit>.rscr file for that design. This is because there is a problem with the Synopsys VHDL analysis when the FSM attributes (like attribute fsm_state_vector etc.) are in the VHDL code. This causes the make mechanism to start again, every time, although there were no changes to the VHDL source code.

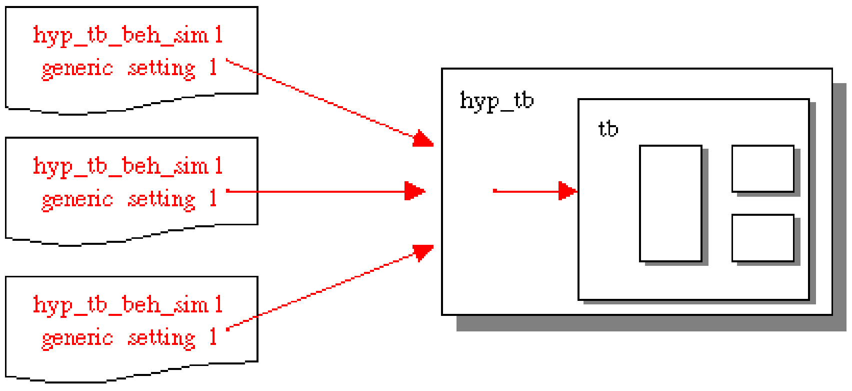
1.4.76 A testbench has neither ports nor generics (e)

Otherwise simulator complains about missing initial values. Another disadvantage of ports and generics is the fact that such testbench is not self-contained. I.e. it would produce different behaviour depending on the applied signals and values.

If a testbench builds on generics, a hyper-testbench is needed to configure the generics. A hyper-testbench is a shell around the main testbench. It consists of an entity without ports, an architecture instantiating the main testbench and the corresponding configuration.



The generic map may be placed in the configuration. Thus multiple configurations with different generic settings can be prepared and precompiled for separate simulations with the same testbench.



1.4.77 Format for pattern text-IO is LSIM-TV (a)

Note that this rule relates to pattern stimuli and responses only. A custom format needs to be defined where abstract data is accessed from VHDL models, e.g. assembler commands. A major disadvantage of using LSIM-TV-pattern is non-reactive stimulus; i.e. stimulus does not depend on the actual output values of the device under test.

LSIM-TV is a tester-oriented format. Input and output values are combined into one line where the @<timeval> prefix indicates the strobe time in multiples of the base unit. Time values must be in ascending order.

```
CODEFILE
UNITS ns
INPUTS a[3], a[2], a[1], a[0], b[3], b[2], b[1], b[0];
```

```
OUTPUTS sum[3], sum[2], sum[1], sum[0];
CODING(ROM)
@0 <00000000>xxxx;
@9 <00000000>xxxx;
@10 <10000000>0000;
@15 <10000000>0000;
@20 <10000000>1000;
@30 <10000000>1000;
@40 <10000000>1000;
@50 <00000001>1000;
@60 <00000010>0001;
@70 <00000000>0010;
@80 <11111110>0000;
END
```

All time units refer to the unit defined in the file header. Possible units are: ps, ns, us, ms.

The values of the inputs are enclosed in sharp brackets <...>; the expected output values at that strobe time are appended. The values are ordered according to the INPUTS and OUTPUTS-statements in the file header.

Utility functions for reading and writing LSIM-TV format files are available through the precompiled resource library lsim_interface. However, do not use them directly. Rather utilize tbgen-hw and wig. tbgen-hw creates a testbench for a unit. This testbench reads LSIM-TV files for stimulus and expected responses. wig instantiates a LSIM-TV writer into an existing testbench. This writer strobes all DUT-interface signals.

Using tbgen-hw and wig guarantees standard-compliant LSIM-TV files (especially wrt. bidir-handling).

1.4.78 Use abstract, compact coding style in testbench and simulation models (a)

Abstract coding styles should be used in testbenches whenever possible, because of flexibility, readability and performance advantages. In testbenches, synthesizability restrictions do not apply, unless the test is carried out on the cycle-based simulator or the hardware emulator.

Note that nowadays the amount of testbench code may even exceed the amount of DUT code.

For abstract coding the following may be used: record types, access types (for pointers), real values, etc.

1.4.79 Stop the simulation from within the testbench (a)

The testbench should stop the simulation by itself, either by ceasing generation of events, or by using an assertion:

```
variable k_v : integer := 0;
...
k_v := k_v + 1;
assert k_v < 86400   --<- activated if expression
                     --   evaluates to false
  report "stopping simulation ..."
    severity failure;
```

These techniques are advantageous, because the simulation duration is "calculated" from within the testbench, and needs not be adjusted in a separate simulator script. For some simulations, the end of the stimulus file is a stopping criterion. Modification of the stimulus file automatically adjusts the simulation duration.

Templates for stopping of simulation are prepared in `emacshw`.

Note that restoring a simulation in Leapfrog is possible even with this technique, if the simulation has been run up to the end. Therefore, short simulation duration should be used during the debugging phase.

1.4.80 Use assertion messages extensively (a)

Use assertions extensively, to monitor significant events. Make sure, that a useful severity level is assigned to the message.

Alternatively use std.textio.output for more compact messages, but only for simulation models (CVE cannot handle text-IO properly up to now). Limit each assertion line to 80 characters:

```
   write( textline, msg );        -- <- writes the message
   writeline( output, textline ); -- <- to stdout
```

Use string concatenation and formatting characters for long messages.

```
   assert error_flag='0'
     report "*** error in function read_byte:" & LF &
            "    undefined symbol in stimulus file"
     severity error;
```

Assertions should be used to

Check parameter and array size consistency unless these are not checked by type definition.

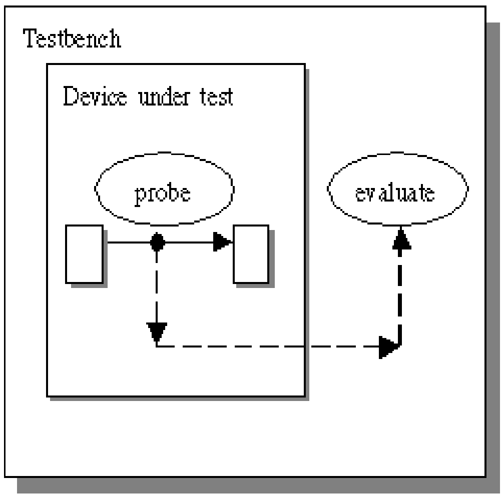
Check whether input values to subprograms or entities are legal.

Give notification of any error condition.

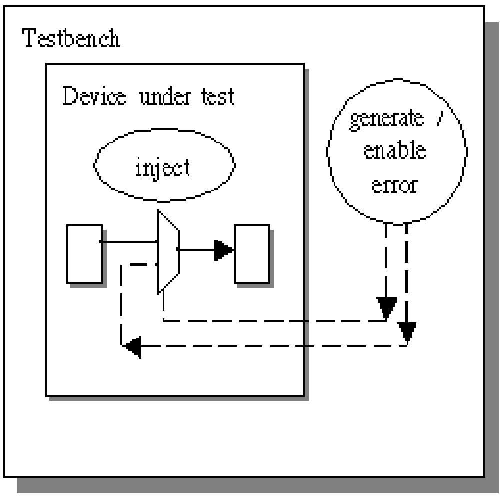
1.4.81 Global signals may be used for testing purposes only (m)

Global signals are declared in a package. They allow for communication between architectures without requiring port connections.

Global signals may be used for testing purposes only, e.g., stimulus injection, and internal probes:



Setting internal probes



Injecting errors

Function of models may not rely on any global signal. Declarations and references of global signals have to be hidden from synthesis via meta comments:

```
package glob_pack is
  signal probe_global       : boolean := FALSE;
  signal errorinject_global : boolean := FALSE;
end glob_pack;
...

architecture ...
  ...

  probe_global <= internal_sig;
  ...

  p1: process (a, b)
  begin -- p1
    z <= (a or b) xor b;
    -- pragma synthesis_off
    if errorinject_global then
      z <= '0';
    end if;
    -- pragma synthesis_on
  end process p1;
  ...
```

Note that **no** meta-comments `-- pragma synthesis_off/on are required for probing via global signals.`

1.4.82 Use std.textio only, where really necessary (a)

Details of text-IO are simulator dependent. Especially avoid std.textio.input. std.textio.endline is excluded from VHDL-87, use l'length=0 instead.

1.4.83 Communication between modules using text-IO is forbidden (m)

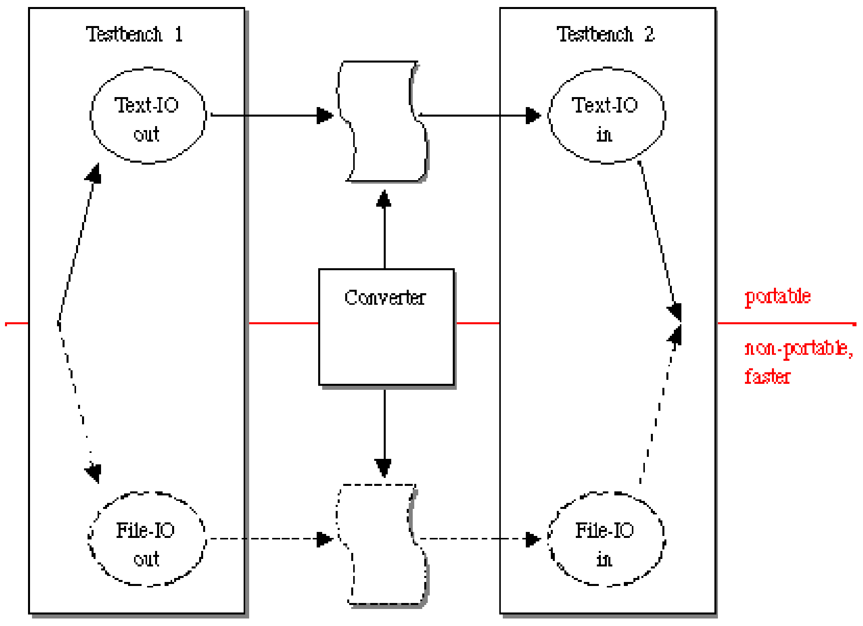
Such techniques may cause non-deterministic behaviour of the model.

1.4.84 No characters with a lower rank than space may occur in text-IO (m)

This would degrade portability over different platforms and over different simulators.

1.4.85 Binary file-IO may not be used (a)

Binary file-IO formats are simulator- and platform-specific and therefore not portable. However, binary file-IO has some performance-advantage. Therefore binary file-IO is not forbidden, but it



A converter consists of a simulation model, which runs the same text-IO and file-IO routines as the testbench.

1.4.86 Use relative pathnames for files accessed through text-IO (m)

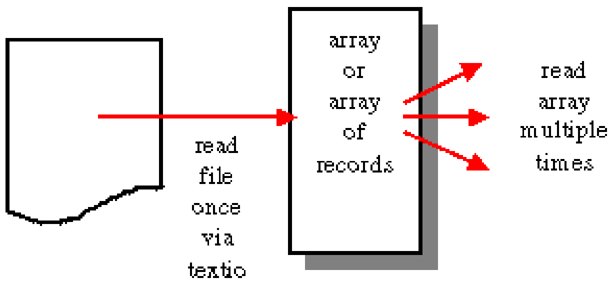
Otherwise models are not portable at all. Note that absolute pathnames are used in the models of memlib at the moment, such that the memlib needs to be generated on a project-by-project basis. This will be changed in the future.

1.4.87 Check the success of every file operation (r)

A readline/writeline operation may crash if the file was not opened successfully. Similarly, a read/ write operation after a failed readline/writeline may cause a failure.

1.4.88 Store file data to a VHDL variable, if read multiple times (e)

In VHDL-87 a file cannot be re-read. There is neither a file-close nor a file-open command. Therefore a file can be read only once. To access the read data multiple times, the data must be stored to a VHDL variable. If the lines contain only one piece of data, an array is appropriate, else you may use an array of records.


read file once via textio
array or array of records
read array multiple times

1.4.89 Use a separate library clause for each declared resource library (a)

Do not make more than one package visible with a use statement, i.e., avoid the following:

```
use ieee, std_developerskit;
```

Instead write:

```
use ieee;
use std_developerskit;
```

In addition it is more readable and partial deactivating is easier using comments.

1.4.90 Use a separate use clause for each declared package (a)

Do not make more than one package visible with a use statement, i.e., avoid the following:

```
use ieee.std_logic_1164, std_developerskit.std_iopak;
```

Instead write:

```
use ieee.std_logic_1164;
use std_developerskit.std_iopak;
```

In addition it is more readable and partial deactivating is easier using comments.

1.4.91 Indentation should be consistent in all VHDL code (r)

Use 2 spaces instead of TAB, being environment dependent. In emacshw automatic TAB-to-spaces conversion is the default. If you use vi, TAB must be used for auto-indentation. In this case all indentation must be done using TAB's. I.e. don't mix TAB's and spaces. The UNIX tool expand can be used to convert TAB's to spaces.

Consistent indentation is important for easy recognition of the code structure. Mixed use of TAB's and spaces may lead to scrambled indentation on printer or other terminals.

1.4.92 Align comments vertically (a)

```
if tst_mode = 0 then
  wait for 10*period_c;
  --  sets time to 08:15 and toggles toggle
  --  for activating alarm
  set_time <= active_set_time;
  wait for period_c;
  ...
```

Aligning comments this way allows for clear recognition of the code structure.

1.4.93 Align declaration lists vertically (a)

Column-wise alignment is recommended. I.e. align not only port or signal names vertically, but also port mode, type and trailing comment. Trailing comments are allowed for declarative statements.

```
port(
    clk        : in  std_ulogic;
    reset      : in  std_ulogic;
    test_se_i  : in  std_ulogic;  --<- scan test enable,
                                  --   active high
    toggle_i   : in  std_ulogic;  --<- toggle enable,
                                  --   active high
    st_o       : out std_ulogic   --<- status bit
);
```

Declare one port per line only. This improves readability and enables straightforward component instantiation. Component instantiation can be done by hand using cut and paste to generate the component declaration from the entity declaration. Alternatively, this can be achieved semi-automatically using `emacshw's feature "component instantiation". It relies on a port list with one port per line.`

1.4.94 Align association lists vertically (a)

```
check_timing_p(
    clk  =>  clk,
    res  =>  rst,
    dat  =>  data_i );
```

This rule applies not only to association lists of procedures, but also to those of functions, port maps, generic maps and array initializations.

1.4.95 Processes should be labeled (r)

Synopsys VHDL Compiler uses process labels to name registers. Therefore, it is easier to identify structures in the netlist, if labels had been used.

1.4.96 Use closing labels for entity, architecture, process, loop, generate, etc. (a)

Use meaningful labels, i.e. not `u1`.

```
protocol_checker: process( pass )  -- <---
begin
  ...
end process protocol_checker;      -- <---
```

1.4.97 Indicate condition at remote ends of control structures (if ... then,

case) (a)

This rule applies to constructs that spread over more than 10 lines.

```
if pass then
  ...
  ...
else -- not pass     -- <---
  ...
  ...
end if; -- pass      -- <---
```

1.4.98 Language for comments and code documentation is English (m)

Many design teams at Infineon Technologies are already international. English is quite natural for them. But this rule also applies to national teams. Their design or parts thereof might get re-used in another country.

1.4.99 Comments should be related to the lines of code below (a)

```
Example:
  ---------------------------------------------------------
  -- output
  ---------------------------------------------------------
  -- address_decoder is an one-from-n decoder.
  -- it is necessary to select the correct register,
  -- corresponding to the address.
  ---------------------------------------------------------
  address_decoder: process (address)
    constant max_regsel : integer :=
                               (2 ** addressbus_width) - 1;
    subtype  rsrange_t is integer range max_regsel downto 0;
    variable regsel_v : std_ulogic_vector(rsrange_t);
  begin
     ...
```

1.4.100 Comment each process, each subprogram and global aspects (r)

Global aspects are not only global signals, but as well

- ❏ Variables, signals or subprograms used in many wide-spread places
- ❏ Types and constants relevant for all of the project
- ❏ File formats used in testbenches

The comments should be placed at the declaration or where the file is written.

Comments should describe what the code does, not how this is achieved.

1.4.101 Trailing comments are allowed for declarative lists (a)

Example:
```
  signal idle_mode : std_ulogic;   -- deactivates accumulator
```

1.4.102 Maximum line-length is 80 characters (a)

The origin of this recommendation lies in the history of terminals. Still many engineers are used to that width of texts. They have adjusted their editors accordingly. The same applies to printouts. Editors and pretty-printers are capable of handling wider code. However, exchangeability and acceptance of your code at the re-user is increased, if 80 columns are used.

Sometimes it is the trailing comments in declarative lists that let the code grow wide. Try to shorten the trailing comments, wrap it to the next line or move them on top of the described declaration:
```
   signal snd_ctl_inv : std_ulogic_vector(snd_range_t); -- sound
                                                        -- control
```

Distribute strings that exceed 80 columns to multiple lines with string concatenation:
```
   library std_developerskit;
   use std_developerskit.std_iopak.all;

   assert mode/=1 or snd_ctl_inv="001"
     report "Error: wrong value of snd_ctl_inv in mode 1" & LF &
            "       snd_ctl_inv=" & to_string(snd_ctl_inv)
     severity error;
```

1.4.103 Use one statement per line (a)

Multiple statements per line may be considered, only, if they are of repetitive type, e.g.:
```
   D0 <= D(0); D1 <= D(1); D2 <= D(2);
```

1.4.104 Group related constructs (a)

Keep this rule in mind while coding concurrent statements. Their order is not crucial for correct functionality. However, concurrent statements and processes with strong interaction should be placed next to each other. With such arrangement, the reader can understand the code more rapidly.

1.4.105 Group related port or signal declarations (a)

Use one declaration per line plus an optional trailing comment. For ordering of ports one of the following methods may be used:

Order according to mode: clock, reset, inputs, bidirectionals, outputs.

Or order ports according to functionality groups, within groups order according to mode, e.g.:

```
port (
  clk : in std_ulogic;  -- system clock
  res : in std_ulogic;  -- async. reset

  test_mode_ni : in  std_ulogic;
  stop_mode_no : out std_ulogic;
  idle_mode_no : out std_ulogic;

  sfr_data_i   : in  std_ulogic_vector (7 downto 0);
  sfr_data_o   : out std_ulogic_vector (7 downto 0);
  sfr_wr_o     : out std_ulogic;
  sfr_rd_o     : out std_ulogic;

  mem_data_i   : in  std_ulogic_vector (7 downto 0);
  mem_addr_o   : out std_ulogic_vector (9 downto 0);
  mem_data_o   : out std_ulogic_vector (7 downto 0);
  mem_wr_no    : out std_ulogic;
  mem_rd_no    : out std_ulogic;
  .... );
```

1.4.106 Switch constructs should not be nested to more than 4 levels (a)

Code exceeding this limit tends to be very difficult to understand. This is especially the case if the switch construct is very long. In such cases it may take a while to find and correlate corresponding if, elsif and else statements.

To come around this problem, use subprograms, which package parts of the functionality. Such subprograms may be local to the process or subprogram. This is the only case, where side effects are acceptable, i.e., manipulation of variable values without passing the variable into and out of the subprogram. The subprograms have to be local to the calling process or subprogram:

```
out_proc: process( clk, rst )
  variable start_v : integer := 12;
  function gen_cnt_f() return integer is --<- local
  begin                                  --<- function
    <LOTS_OF_CODE>                       --<-
    return (start_v + 2);                --<-
  end gen_cnt_f;                         --<-
begin
  if rst=rstactive then             --<-
    cnt <= 0;                       --<- process
  elsif clk'event and clk='1' then  --<- structure
    if mode0='1' then               --<- easily
      cnt <= start_v - 15;          --<- recognized
    else                            --<-
      start_v := start_v + 1;       --<-
      cnt <= gen_cnt_f();           --<-
    end if;                         --<-
  end if;                           --<-
end process out_proc;
```

1.4.107 Remove unnecessary code fragments (a)

For detection use the coverage analysis facility coversim. This tool provides you with a listing of your source code with annotated activity numbers. Of course, it may turn out at this point, that the pieces of code may not be removed, and that the tests need to be enhanced.

Once you have identified code fragments, that are no longer needed, really delete them. Rely on source code versioning, which allows for arbitrary retrieval of previous versions of the code. Using comment signs for deactivating is not appropriate. This may be used for ad-hoc testing only, but may not occur in delivered code.

This rule does not apply to VHDL code referenced from general-purpose resource libraries. These libraries typically contain code, which is not activated in the current application. This code cannot and must not be remove.

In a similar way, softmacros may contain code, which is not relevant in the current application. For security reasons, do not remove the respective code fragments. There might be side effects, which are not obvious.

1.4.108 Use named association in association lists (r)

This rule makes sure that misconnections do not happen. Even if the order of formal parameters is changed in the declaration, the association list is still valid. The rule applies to

Port and generic maps

Function and procedure calls

First exception to this rule: Frequently and locally used, short subprograms with few parameters:

```
execute: process
  file      log_file : text is out log_name;
  variable  textline : line;
  procedure print_p( msg : string ) is --<- subprog. local
  begin                                --<- to this proc.
    write( textline, msg );
    writeline( log_file, textline );
  end print_p;
begin
  ...
  if pline(1 to 4) = ":eot" then
     print_p( ">> Test complete <<" );    --<- *)
  elsif (pline(1 to 7) = ":pct_on") then
     print_p( ">> PC trace on <<" );      --<- *)
  ...

  -- *) positional association is easier in this case
```

Such use of subprograms is very similar to replacement macros. They increase readability and modularity; i.e. changes to the functionality are performed rapidly at one place.

Second exception to this rule: Use positional association for initialization of constant arrays. Named association causes the Leapfrog simulator to crash at this instance.

1.4.109 Order of items should be the same in package header and body (r)

This enhances greatly orientation within the package. Type declarations and constant definitions in the header should precede to the subprograms that are common to header and body. In the same way, package-local subprograms should be placed at the very beginning of the package body:

```
package ordered_pack is
  -- type declarations
  -- constant definitions
  -- subprogram 1 declaration     --<- common
  -- subprogram 2 declaration     --<- part,
  -- ...                          --<- same order
end ordered_pack;

package body ordered_pack is
  -- type declarations, local to the package
  -- constant definitions, local to the package
  -- subprogram definitions, local to the package
  -- subprogram 1 definition      --<-
  -- subprogram 2 definition      --<- common part,
  -- ...                          --<- same order
end ordered_pack;
```

1.4.110 Use predefined attributes to make code generic and more re-usable (a)

Use predefined attributes in order to write generic and re-usable code, especially for array type objects:

```
for k in 1 to rd_line'length loop  --<- independent of
                                   --   actual length
  pure_line(k) := rd_line(k);
  ...
```

Make your VHDL code independent of the direction of vectors by using predefined attributes:

```
function "abs" (arg: signed) return signed is
  constant arg_left : integer := arg'length-1;     --<- *)
  constant xarg     : signed(arg_left downto 0) := arg;
  variable result   : signed(arg_left downto 0);
begin
  result := to_01(xarg, 'x');
  ...
  return result;
end "abs";

-- *) predefined attribute 'length
```

Defining xarg makes sure, that in the algorithm the input vector is 'normalized' to the downto-direction.

1.4.111 Language for modeling is VHDL-87 (m)

In no parts may VHDL code rely on VHDL-93 features. The reason for this policy is laid by the fact that most CAE tools do not support VHDL-93. Remember that we want to use the same VHDL code in various tools:

- ❏ Simulator
- ❏ Synthesis tool
- ❏ Formal Verification tool
- ❏ RTL Floorplaning tool (in the future)

Furthermore, we want to be prepared for eventual tool exchanges by relying on the most common standard. This implies that we find a good balance between using only basic functionality of the language and exploring every niche.

See section Literature for information on how to obtain the correct version of the LRM (Language Reference Manual).

1.4.112 Check for LRM-compatibility using cv w/o -compat switch (r)

Unfortunately VHDL compilers adhere to the VHDL standard to different levels. Synopsys' VHDL Compiler is the tool, which is least strict regarding VHDL rules. In some cases, compilers accept expressions that are not backed by the standard. Other tools then fail on exactly these parts of the code.

Special care is needed while working with the Leapfrog compiler cv and the elaborator ev. It may be operated with the switch -compat, which makes it behave as Synopsys' VHDL Compiler. We urge you not to use this switch, as it only guarantees that these two tools can process the code equally. However, the code is also intended for formal verification with CVE and for the RTL floor-planner (in the future).

Note that the compilation step is not only a preparation of the tool run, but also a check for portability. It seems that the strictest test for portability is performed by smake -cve. Use this compiler extensively. In emacshw, you may consult all available compilers through the Compile menu.

There is one exception, where the use of cv/ev -compat is acceptable:
"Old" VHDL code, where non-standard styles are used. Usually such code is not acceptable. Cleaning the code, or a re-write are recommended. However, all functionality must be verified intensively then. For the intermediate time, the "old" code can be used with -compat.

Note, that -compat is not needed for instantiation of library cells, such as pads!!! If you get the following error-message from ev:

```
ev:*W,1101:component instance is not fully bound ...
```

add a use clause to the configuration:

```
library padlib;               --<- make pad component
use padlib.components.all;    --<- decls. available

entity cli_chip is
  ...
end cli_chip;
```

```
architecture struc of cli_chip is
  ...
begin
   p1 : PGI_ECB1xxEx -- same casing as in padlib!
    port map (
    ...
end struc;

library padlib;    --<- make all pad entities and
use padlib.all;    --<- architectures available for
                   --   default binding

configuration cli_chip_struc_c0 of cli_chip is
  for struc
    for core : cli_core
      use configuration work.cli_core_struc_c0
    end for;
    -- no component-configuration for the pads here!!!
    -- default binding allowed, because only one
    -- architecture exists for every pad entity
  end for;
end cli_chip_struc_c0;
```

For formal verification with CVE it is necessary that the component declaration is provided through the components-package, which resides in the resource-library.

1.4.113 VHDL-93 keywords should not be used (r)

In order to allow the reuse of modules coded according to the VHDL-87 standard in future InWay releases, which will support the IEEE VHDL-93 standard also, the following 93-keywords should be avoided:

```
group   impure   inertial  literal  postponed
pure    reject   rol       ror      shared
sla     sll      sra       srl      unaffected
xnor
```

In additon it should be kept in mind when writing upwards compatible code, that the behaviour of the concatenation operator and the access of discrete slices of arrays have been modified slightly.

So the resulting left bound and the direction of a concatenated arrays are defined differently for VHDL 87 and VHDL 93 under certain cirumstances.

While VHDL 93 requires, that array slices must show the same direction as the original array, the VHDL 87 evaluates opposite directions of slices and arrays to a *null slice*.

1.4.114 Verilog keywords should not be used (r)

```
always      and         assign      attribute
begin       buf         bufif0      bufif1
```

case	casex	casez	cmos
const	deassign	default	defparam
disable	edge	else	end
endattribute	endcase	endfunction	endmodule
endpackage	endprimitive	endspecify	endtable
endtask	event	for	force
forever	fork	function	highz0
highz1	if	initial	inout
input	integer	join	large
macromodule	medium	module	nand
negedge	nmos	nor	not
notif0	notif1	or	output
package	parameter	pmos	posedge
primitive	pull0	pull1	pullup
pulldown	reg	rcmos	real
realtime	reg	release	repeat
rnmos	rpmos	rtran	rtranif0
rtranif1	scalared	signed	small
specify	specparam	strength	strong0
strong1	supply0	supply1	table
task	time	tran	tranif0
tranif1	tri	tri0	tri1
triand	trior	trireg	use
vectored	wait	wand	weak0
weak1	when	while	wire
wor	xor	xnor	

The given list of keywords is merged from all relevant lists. The list varies from tool to tool.

1.4.115 SDF keywords should not be used (r)

SETUPHOLD	RECOVERY	SKEW	WIDTH
SETUP	PERIOD	NOCHANGE	PATHCONSTRAINT
NETDELAY	HOLD	SUM	DIFF
SKEWCONSTRAINT	DEVICE	DESIGN	PORT
INTERCONNECT	COND	IOPATH	DATE
PROGRAM	VERSION	DIVIDER	VOLTAGE
VENDOR	PROCESS	TEMPERATURE	TIMESCALE
CELL	CELLTYPE	CORRELATION	INSTANCE
DELAY	TIMINGCHECK	ABSOLUTE	INCREMENT
PATHPULSE	GLOBALPATHPULSE	DELAYFILE	SDFVERSION

1.4.116 Carefully use real values with PN-generators (a)

The behaviour of pseudo-random number generators based on real type is potentially not reproducible on a different platform.

1.4.117 Allowable replacement characters are forbidden. (m)

Allowable replacement characters are defined in LRM 13.10. It is a rarely used feature. We suspect that different simulators handle these characters differently.

1.4.118 Tool specific types may not be used (m)

Use standard types like boolean, integer, etc. or the ieee standard logic types std_(u)logic(_vector). All VHDL '87 compliant tools support these types.

1.4.119 Operating system specific features may not be used (r)

At the moment, SC-Highway runs under Solaris. However, for the future it cannot be guaranteed that this will always be the (only) operating system. A trend of CAE-tools towards Microsoft Windows is obvious. Therefore make sure that your VHDL model does not rely on operating system specific features like e.g. /dev/null. Furthermore, model exchange with sub-contractors and customers is more seamless if those features are omitted.

1.4.120 Never redefine standard operators, subprograms, attributes, and packages (r)

There is the possibility of overloading in VHDL. An operator or subprogram may be named the same as one that is already existing. The parameter profile must be different. This is a very useful feature because the code reads the same independent from the types being used. Make sure that subprograms of one name implement the same functionality in order to avoid confusion.

If the parameter profile is equal to one of an existing version of the subprogram, a rule of VHDL says that both subprograms get invisible. Thus, an error is issued to give awareness of the dangerous situation. Unfortunately the switch -compat of the Leapfrog simulator enables a search mechanism instead; i.e. the first subprogram encountered will be elaborated. This is a dangerous policy, because it is not clear which subprogram takes effect. We recommend not to use -compat (see the related rule "Check for LRM-compatibility using cv w/o -compat switch"), however there are situations where it cannot be avoided. In such case, make sure that no subprogram gets overridden in the above-described way.

Redefining standard packages or commonly available utility packages is dangerous to the same degree. Re-users of your model might not be aware of this construction and fail while referring to the standard version of the package.

1.4.121 Bussed ports of width one are forbidden (r)

In the past there have been problems in the Cadence tools with such busses. The respective tools are not used in SC-Highway at the moment. Nevertheless avoid this construct for the sake of portability.

1.4.122 Configuration declarations must have the configuration name and the entity name in one line for vimport (a)

A configuration declaration has to look like this:

```
configuration <conf-name> of <ent-name> is  --<- must be one line
  for <arch-name>
  [
    for <inst-name> | all : <comp-name>
      use configuration
        work.<name-of-config-to-be-referenced>
    end for;
    ...
  ]
  end for;
end <conf-name>;
```

If there is a line break in the first line, vimport isn't able to recognize the entity-name.

1.4.123 Bussed arithmetic objects use "downto" as their index orientation (r)

It is common policy to use downto index orientation for vectors that denote arithmetic numbers. Thus the MSB can always be retrieved using the attribute 'high. Furthermore the packages std_logic_arith and synth_regpak rely on this rule.

Use downto-direction for slices as well.

1.4.124 Request external VHDL code suppliers to deliver used arithmetic packages (r)

If the delivered code uses one of the packages std_logic_arith, std_logic_signed or std_logic_unsigned, these must accompany the delivered code. (We must check, whether their contents match their counterparts in SC-Highway, unfortunately some CAE-vendors brought modified versions to their customers). These packages must be referenced out of the library ieee. Proprietary arithmetic packages must not be used prior to confirmation of the project leader at HL.

1.4.125 Where to use ieee.std_logic_arith and std_developerskit.synth_regpak (e)

If bit-wise manipulations are predominant (e.g. rounding bits, quantization, etc.), use ieee.std_logic_arith.signed or unsigned and the operators defined in this package. std_developerskit.synth_regpak should be used only, where std_logic_arith doesn't offer the needed functionality. Make only those objects visible using selective use clauses, e.g.:

```
  use std_developerskit.synth_regpak.regadd;
  use std_developerskit.synth_regpak.regsub;
```

1.4.126 Where to use std_logic_signed and std_logic_unsigned (e)

The arithmetic package being used most is std_logic_arith. This should be reflected in our models as well. However, if one entity uses either signed or unsigned arithmetic only and not a mix of both, then it makes sense to use the package std_logic_signed or std_logic_unsigned respectively. Thus arithmetic signals and variables need not be converted to signed or unsigned. The operators out of these packages work on std_logic_vector's directly. Type marking is still necessary to maintain unresolved vector data types (see the related rule: "Unresolved data types std_ulogic(_vector) are recommended"):

```
variable datin_v  : std_logic_vector(dat_i'range);
variable datout_v : std_logic_vector(dat_o'range);
...
datin_v := std_logic_vector(dat_i);
...
dat_o <= std_ulogic_vector(datout_v);
```

If an entity uses std_logic_(un)signed then a use clause for std_logic_arith is not needed.

Related rule: "Type mark comparison operands".

1.4.127 Take care of array widths for std_logic_arith operators (e)

+/- : both operands must have the same width as the result.
* : width of result = width of operand 1 + width of operand 2.

This holds for std_logic_arith/signed/unsigned regardless of the possible value ranges. The operands can be extended using the following functions:

```
conv_unsigned(arg: unsigned; size: integer)
   return unsigned;
conv_signed  (arg:   signed; size: integer)
   return   signed;
```

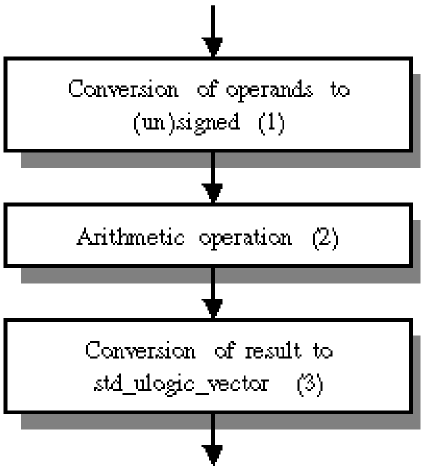
To extend the vector arg, the new size must be larger than its actual width.

1.4.128 Use conversion functions around arithmetic operators (e)

Conversion functions are needed, unless all operands and the target are available as integers. The following three-part structure is proposed:

```
library ieee;
use ieee.std_logic_arith.all;
...
signal op1, op2, res : std_ulogic_vector(wi_c-1 downto 0);
...
arith : process (clk, rst_n)
  variable op1_v, op2_v, res_v :
                integer range 0 to (2**wi_c)-1;
  ...
  elsif clk'event and clk='1' then
    op1_v := conv_integer( unsigned( op1 ));  --<- 1)
```

```
op2_v := conv_integer( unsigned( op2 ));  --<- 1)
res_v := op1 - op2;                       --<- 2)
res   <= std_ulogic_vector(
         conv_unsigned( res_v, wi_c ) );  --<- 3)
 ...
```


Conversion of operands to (un)signed (1)
Arithmetic operation (2)
Conversion of result to std_ulogic_vector (3)

1.4.129 Comparison of arithmetic arrays should be based on same type and width (r)

For comparison use the operators from ieee.std_logic_arith/signed/unsigned. Using same type and width guarantees, that the result is as expected. Comparison operators from ieee.std_logic_1164 are not recommended, as their comparison is based on the lexical value.

Do not rely on VHDL built-in comparison operators. These operators are based on lexical order as well. Such built-in comparison operators get implicitly created for each new type.

Related rule: "Type mark comparison operands".

1.4.130 Type mark comparison operands (r)

This rule must be followed if std_logic_(un)signed is visible. For each declared vector type there is a comparison operator implicitly created. Its declaration is not given. Still the operator is available. For std_logic_vector there is another comparison operator available, which stems from std_logic_(un)signed, e.g.:

```
  use ieee.std_logic_1164.all;
  use ieee.std_logic_signed.all;
  ...
  signal a1 : std_logic_vector(4 downto 0);
  ...
  x <= '1' when a1 = "00000" else '0'; --<-
```

In this situation VHDL states that both operators shall be hidden. The compiler issues an error message to warn about the ambiguity. Unfortunately not all compilers available in the SC-Highway detect this situation correctly. Use the compiler vhdl of CVE to check for this problem (smake -cve).

The solution for the problem is to explicitly state in the code which comparison operator shall be used:

```
  use ieee.std_logic_1164.all;
  use ieee.std_logic_signed.all;
  ...
  signal a1 : std_logic_vector(4 downto 0);
  ...
  x <= '1' when ieee.std_logic_signed."="(a1, "00000") --<-
           else '0';
```

Another solution is the using an unresolved vector (std_ulogic_vector). For this type no additional comparison has been defined:

use ieee.std_logic_1164.all;

use ieee.std_logic_signed.all;

...

signal a1 : std_ulogic_vector(4 downto 0); --<-

...

x <= '1' when (a1 = "00000") else '0'; --<-

Alternatively, type marking to force the use of the same comparison operator:

```
  x <= '1' when (std_ulogic_vector(a1) = "00000") else '0';
```

1.4.131 Run synthesis early in the design process (r)

Early synthesis is recommended as a means to analyze the performance achievable with the chosen RTL-implementation. Only slight performance improvements can be achieved through constraint refinement or modification of the compile strategy. Major performance lacks can only be removed by RTL-code re-arrangements. Such fundamental code-changes should be done as early as possible, in order to come to a stable base for further fine-tuning and chip-/macro-integration.

1.4.132 Code for synthesis must match the synthesis subset (m)

The most important prerequisite for writing synthesizable VHDL code is knowledge of the Synopsys VHDL Compiler. The synthesis subset of VHDL is described in this document: Check out the VHDL Compiler Reference from the Synopsys online-documentation: tooldoc synopsys. A brief overview of the supported VHDL subset is given in appendix C: "VHDL Constructs".

The other source of information is the Synopsys Release Notes. You can access them through Solv-It at www.synopsys.com . For subscription to this service, you'll need the site-ID of Infineon Technologies: It is 611. You can furthermore use this service to specifically query the Synopsys knowledge database.

1.4.133 Default values for ports, signals and variables may not be used (m)

Default values for ports, signals and variables are used in simulation, but not translated by synthesis. Therefore, existence of such default values is very often the reason for simulation mismatches between pre- and post-synthesis simulation. Instead of using default values, activate the reset of storage elements at the beginning of the simulation.

1.4.134 Data types must be synthesizable (m)

The following data types are synthesizable:

- ❑ Enumeration
- ❑ Constrained integer. The constraint makes sure that correct bus-width is inferred.
- ❑ Arrays and arrays of arrays, provided that they are indexed by integers. Ranges must be natural for std_(u)logic_vector.
- ❑ Records

Preferred data types for synthesis are

- ❑ std_(u)logic(_vector) as logic type
- ❑ constrained `integer or (un)signed for arithmetic`
- ❑ `natural for array indices`
- ❑ enumeration types for FSM states and other state descriptions
- ❑ Arrays of `std_(u)logic_vector for memories`
- ❑ Records for complex data-structures and improved readability

1.4.135 Use standard templates for clocked processes (r)

Several template structures let Synopsys Design Compiler infer registers. See the Synopsys online documentation for more information (VHDL Compiler Reference Manual, Chapter 8, "Register Inference", pp. 219 - 271). However, for safe and portable design, it is strongly recommended not to rely on all of these templates, but rather to use the most common ones, i.e.
`for register with asynchronous reset (such template is available in emacshw under "VHDL" -> "Process"):`

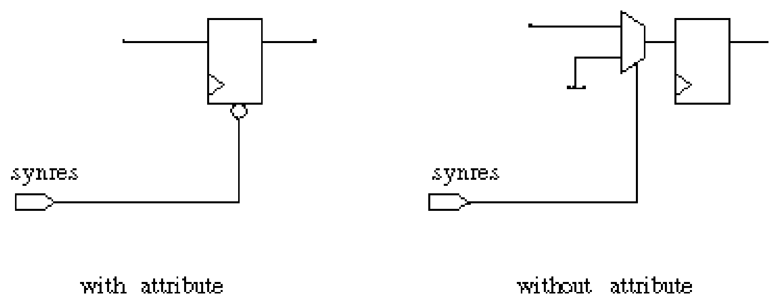
```
async_res: process (clk, rst_n)
begin
  if rst_n='0' then
    <actions to perform at asyn. reset>
  elsif clk'event and clk='1' then
    <actions_to_perform_at_pos_clock_edge>
  end if;
end process async_res;
```

Clock and reset must be in the sensitivity list to avoid simulation mismatches pre- and post-synthesis.

For register with synchronous reset (such template is available in `emacshw under "VHDL"` -> "Process"):

```
library synopsys;              --<- put this in front
use synopsys.attributes.all;   --<- of the entity
...
attribute sync_set_reset_local of  --<- put this in decl.
   sync_res: label is "rst_sync";  --<- region of arch.
...
sync_res: process (clk)         --<- reset need not be
begin                           --   in sensitivity list
  if clk'event and clk='1' then
    if rst_sync='0' then
      <actions_to_perform_at_sync_reset>
    else
      <actions_to_perform_at_pos_clock_edge>
    end if;
  end if;
end process sync_res;
```

The attribute sync_set_reset_local is declared in the package synopsys.attributes. It must be defined to force synthesis to implement the synchronous (re)set using a synchronously (re)settable flip-flop. Otherwise the synchronous (re)set would get merged with the logic in front of the flip-flop's D-input. A flip-flop without synchronous (re)set pin would be used:



Template for latch:

```
latch: process (gate, data) --<- put enable and
begin                       --   data into the sensitivity
  if gate='1' then          --   to avoid simulation
    q <= data;              --   mismatches
  end if;
end process latch;
```

Do not use wait to build registered processes.

1.4.136 Generics must have type integer (m)

This is a restriction of Synopsys Design Compiler. All modules that will be fed to synthesis should follow this rule. We recommend, selecting such generics even if a behavioural architecture is under development, and it is planned to further develop this architecture for synthesis. It may as well be planned to develop an alternative synthesizable architecture later. However, this architecture refers to the same entity. It is efficient to stick to the same entity.

In addition, because of a restriction in SSE it is not allowed to use complex expressions in generic maps. For example something like 2*width_g within a generic map cannot be resolved by the makefile generator of SSE. Therefore, we recommend defining some constants in a global package, which then can be used within the generic maps.
Due to a further limitation the the identifiers of constants intended to be used as generics must be unique throughout all packages in the project. The SSE makefile generator is currently not able to handle constant value on a package base. The value read from the last package is considered during the makefile generation only, independent of the actually referenced packages.

1.4.137 For soft-core development use generics, do not use constants (r)

Generics are the appropriate means for parameterizing re-usable soft-macros. In the past, there have been problems with generic in synthesis, when the soft-macro had to be elaborated separately. In this case, the generic values could not be calculated automatically from the instantiating code. This is now performed by the tool `ssemake, if the soft-macro resides in the library work.`

1.4.138 Place port and generic maps at the component instantiation (m)

Port and generic maps can be overridden in the configuration. However, Synopsys Design Compiler does not recognize such mappings. Therefore always map existing generics at the component instance. Do not rely on default values.

1.4.139 Instantiated component and entity name must be equal, incl. casing (m)

VHDL is case insensitive. Unfortunately, Synopsys Design Compiler does not follow this rule, when linking designs and subdesigns. The component name and the underlying entity name must be equal including case:

```
ctrl: cfl          --<- component name
  port map ( ...

entity Cfl is      --<- this entity name does not match
  ...

entity cfl is      --<- this entity name matches for DC
  ...
```

To minimize eventual problems it is recommended to use lowercase letters for entity and component names generally.

1.4.140 Add an "others" statement to the FSM-switch (m)

The number of enumeration elements need not be `2**N. Adding a when others assignment to the default state will make Design Compiler produce an FSM without undecoded states.`

1.4.141 All conditional assignments should be checked for completeness (r)

Incomplete conditional assignments lack a default statement or an else clause. This may lead to unintended latch inference from non-clocked processes, as being used for combinational logic, e.g. :

```
latch: process (en)
begin
  if en='1' then
    outbit <= pn_gen_ssrg_outbit;
  end if;
end process latch;
```

Always use an else or a default clause:

```
combo: process (en, pn_gen_ssrg_outbit)
begin
  if en='1' then
    outbit <= pn_gen_ssrg_outbit;
  else
    outbit <= '1';
  end if;
end process combo;
```

or a default assignment in front of the conditional structure:

```
combo: process (en, pn_gen_ssrg_outbit)
begin
  outbit <= '1';
  if en='1' then
    outbit <= pn_gen_ssrg_outbit;
  end if;
end process combo;
```

For more information see "Avoid latches unless absolutely necessary".

1.4.142 The sensitivity list for combinational processes must be complete (m)

Synopsys Design Compiler synthesizes combinational logic from processes like the following:

```
combo: process (a, b, c)
begin
  z <= a or (b xor c);
end process combo;
```

The generated logic will react on any event on the inputs a, b or c, and so does the simulated VHDL process. If one of the input signals is omitted in the sensitivity list, the list is called incomplete. Synopsys will generate the same logic, but simulation will produce different results for some stimuli, which may be critical.

Incomplete sensitivity lists may lead to pre-/post-synthesis simulation-mismatches. Unfortunately, these may get unveiled late, when the circuit is already stable. Avoid incomplete sensitivity lists by carefully checking the ssemake transcript for messages like:

```
Elaboration of /home/hwpro/hw_alarm_chip/vob/...
    project_lib/gdbs/test.db
  finished on erle at Sat Dec 27 18:14:57 MET 1997
  Warning:        Sensitivity list is not complete (SSE).
  Check logfile:  logfiles/test_elab.log  (SSE).
```

For simple combinational processes, consider modeling as a concurrent statement:

```
z <= a or (b xor c);
```

Such concurrent statement is triggered, whenever one of its inputs has an event, i.e. this statement has an implicitly complete sensitivity list.

1.4.143 Bit-wise association of arrays in port maps is not possible in presence of generics (m)

This combination causes faulty results in synthesis with Synopsys Design Compiler.

1.4.144 Aliases may not be used in synthesized code (m)

Aliases are not synthesizable. They may therefore be used in testbench or behavioural code only. In these codes, aliases may be used to increase readability:

```
alias carry : std_ulogic := F(3);
```

1.4.145 Embedding scripts as meta comments is not acceptable (r)

Synthesis scripts must be appropriate for Synopsys' Design Compiler. These scripts must be stored as separate files.

Meta comments are tool-specific up to now, leaving the model badly portable and maintainable. The synthesis subset is under standardization, and with it are the meta comments (pragmas). Therefore, wider use might get acceptable in the future.

1.4.146 Use only standard meta comments (r)

Meta comments are tool-specific up to now, leaving the model badly portable and maintainable. The synthesis subset is under standardization, and with it are the meta comments (pragmas). Therefore, wider use might get acceptable in the future.

Try to avoid other meta comments than:

```
-- pragma synthesis_off
-- Makes code invisible for synthesis.
-- pragma synthesis_on

-- pragma translate_off
-- The same as synthesis_off/on, but if the dc-variable
-- hdlin_translate_off_skip_text is set to "true", the
-- code will also be skipped by dc-analyze.
-- pragma translate_on

-- synopsys sde_make_gen_off
-- Makes code invisible for sde_make_gen.
-- synopsys sde_make_gen_on
```

CVE handles `pragma synthesis_off/on` in the right way, but `pragma translate_off/on` has the same function than `synthesis_off/on`, because CVE doesn't know the dc-variables.

Also you should use these meta comments with care. They can cause a mismatch between pre- and postsynthesis simulation.

1.4.147 Requirements for resource sharing (e)

Common resource block must be in

- ❏ same entity
- ❏ same process
- ❏ same if/case statement

Common resource block must never be needed in two places at the same time.

1.4.148 EMC and transmission line effects need not be modeled (a)

These effects are difficult to model in VHDL. The related problems must be solved using more appropriate methods and programs.

1.4.149 Minimize the number of processes, signals and signal assignments (a)

Signals are needed to communicate between processes and ports. Use only as many signals as are needed to cover these tasks. Minimize the number of signals further, by merging processes whenever feasible. Inside the processes, make use of variables.

Note that each concurrent signal assignment is handled as a process with a sensitivity list inside the simulator.

In addition, resource sharing for synthesis works for a single process only.

1.4.150 Replace signals by variables whenever possible (a)

Signals are much more complex objects than variables. For a signal, not only the current value has to be stored, but also the last value, the last event, events scheduled in the future, etc. Variables need not be scheduled into an event queue. Assignments take effect immediately. Therefore simulation speed and memory requirements can be improved a lot by using variables. This can be done easily inside processes to

store intermediate values ("read after write") or even

store values for further processing in the next process activation ("read before write")

If the second option is chosen in a clocked process, then synthesis infers registers to store the value for the next cycle. The following example illustrates this technique:

```
var_regs: process (clk, rst_n)
  variable var_v : integer range 7 downto 0;
begin
  if rst_n='0' then
    var_v := 0;
    dout_1cyc_deld <= 0;
    dout_2cyc_deld <= 0;
  elsif clk'event and clk='1' then
    dout_2cyc_deld <= var_v; --<- read var_v before write
    var_v := din;
    dout_1cyc_deld <= var_v; --<- read var_v after write
  end if;
end process var_regs;
```

The signal dout_1cyc_deld gets the value of din after one cycle of latency. var_v serves as a temporary variable in this case. It is written and read at the same clock edge. No register is generated during synthesis for this read-operation. dout_2cyc_deld gets the din-value after two cycles of latency. var_v stores each value until the process is resumed at the rising clock edge. Then the value of var_v is read.

Note that this modeling style is supported by synthesis.

1.4.151 Inhibit execution of statements where not necessary (a)

Pure simulation models: Suspend processes completely up to wakeup event:

```
stim: process
begin
  get_word_p( dat, dest );
  wait until rdy_n='0';    --<-
  get_word_p( dat, dest );
  wait until rdy_n='0';    --<-
  ...
  wait;    --<- forever (if no further action needed)
end process stim;
```

Synthesizable models: Stick to the register template, create fast escapes. The enable signal may be dynamic (signal) or static (generic, constant):

```
format_frame: process (clk, rst_n)
begin
  if rst_n='0' then
    ...
```

```
      elsif clk'event and clk='1' then
        if formatter_active then          --<- fast escape
          ...    <lots of code>           --   here if
                                          --   inactive
        end if;
      end if;
    end process format_frame;
```

The process is still triggered by clk (and rst_n). However, the fast escape makes sure, that no calculations are done unnecessarily. Do not move the deactivation if ... then to the top of the process. Such process is not recognized by synthesis properly as registered process.

Synthesizable models: Deactivate code segments statically using generate ... if:

```
    rftune_gen: if mode=12 generate
      rftune: process (clk, rst_n)
      begin
        if rst_n='0' then
          ...
        elsif clk'event and clk='1' then
          ...
        end if;
      end process rftune;
    end generate rftune_gen;
```

Deactivation is not solely a technique for speeding simulation. These constructs are useful for synthesis as well, if the spent area is justifiable. Deactivation in general saves dissipated power. There is a relationship between simulation activity (events) and power dissipation.

1.4.152 Divide large memories into blocks (a)

such that each block is activated only if used. This will reduce the simulator process size (RAM-requirement).

1.4.153 Do not use anonymous types (r)

These constructs are not allowed in synthesis. They are furthermore not very common, and tend to make the model more complicated than necessary. These features are not crucial for circuit modeling.

1.4.154 Do not use guarded signals, guarded assignments, and guarded expressions (r)

These constructs are not allowed in synthesis. They are furthermore not very common, and tend to make the model more complicated than necessary. These features are not crucial for circuit modeling.

1.4.155 Do not use disconnect, register, and bus (r)

These constructs are not allowed in synthesis. They are furthermore not very common, and tend to make the model more complicated than necessary. These features are not crucial for circuit modeling.

1.4.156 Do not use port modes buffer and linkage (r)

These port modes have no physical counterpart. Once a buffer port is connected directly to another port, which may be necessary to route signals through hierarchies, this port must be of mode buffer as well. Thus, one buffer port potentially affects many other ports.

Instead, use just the modes in, out and inout. To read the value of an out port, use an intermediate signal (an example is given in "Use an intermediate signal for reading from an output port").

1.4.157 Do not use blocks. Use entities instead to describe design hierarchy (a)

Each block has a name space of its own. It can be used to define signals with the same name as in the host architecture. This makes the model more difficult to understand, to re-use and maintain.

1.4.158 Use only well tested functions for object-initialization purposes (a)

Functions can be used at elaboration, to initialize constants, e.g.:

```
constant propdel_c : real :=
  calc_propdel_f( load_q_c );
```

The function cannot be debugged, since it is carried out at elaboration, i.e. during the last stage of smake. Therefore use only very robust functions for constant initialization. To debug the function, move it temporarily to a concurrent region:

```
-- constant propdel_c : real :=
--   calc_propdel_f( load_q_c );
signal propdel_debug : real;
...
propdel_debug <= calc_propdel_f( load_q_c );
```

Put the breakpoint on the last line and step into the function.

Functions are sometimes used to load a constant data object from a file. However in SC-Highway this is not recommended, since elaboration takes place in $HWPROJECT/project_lib/lfobj. A data file location relative to that directory would be necessary ("Use relative pathnames for files accessed through text-IO").

1.4.159 Concurrent procedures may not have any side effects (r)

Side effects are dangerous, since the model is difficult to understand and to maintain. There is only one exception, where side effects are allowed: Procedures called many times in one process (sequential code) may have side effects. For an example see "Switch constructs must not be nested to more than 4 levels".

1.4.160 Avoid identifier hiding caused by loop index (r)

The index of a loop is implicitly defined. If another object with the same name exists, this object is hidden in the loop-block. Chose a different index name in order to make the code better understandable and to avoid mistakes.

```
hiding: process
  variable i_v    : natural;
  variable accu_v : natural;
begin
  i_v    := 10;
  accu_v := 0;
  bad_loop: for i_v in 5 downto 2 loop
    accu_v := accu_v + i_v;
  end loop bad_loop;
  wait;
end process hiding;
```

The variable i_v is not visible inside the loop. Upon exit of the loop i_v still has the value 10. accu_v has the value 14.

1.4.161 Recursive use of subprograms is forbidden (m)

Not supported in formal verification (CVE) up to now.

1.5 Changes wrt. previous Versions

1.5.1 Changes wrt. VHDL Coding Guidelines v3.2

- ❏ Added rule 1.4.1 : Configuration must be in a separated file (m)
- ❏ Modified rule : VHDL-93 keywords should not be used (r)
- ❏ Modified rule : A configuration declaration is needed for each architecture in the design (m)
- ❏ Removed rule : 1.4.56 process only one clock in RTL units (r)
- ❏ adapt guideline version to flow version scheme (IW2.0)

1.5.2 Changes wrt. VHDL Coding Guidelines v3.1

- ❏ Removed rule: Store each configuration into a separate file (m).
- ❏ Removed rule: Entity and architecture may not be stored within the same file (m).
- ❏ Removed rule: Use only lower-case names for unit directories (e).
- ❏ Removed rule: Memory instantiation (e).
- ❏ New rule: Store each VHDL unit into a separate file (r).
- ❏ Changed rule: Naming conventions for VHDL files (a).
- ❏ Changed rule: Store package header and body into same file (m).
- ❏ Changed Rule: Data types at chip and macro ports: std_(u)logic(_vector) (m).
- ❏ New rule: Spacer cells needed between pads (e).
- ❏ New rule: Avoid declaring constants within subprograms (a).
- ❏ New rule: Be aware of the use of the attributes DELAYED, STABLE, QUIET, and TRANSACTION (a)
- ❏ New rule: Minimize the number of signals of your sensitivity list (a).
- ❏ New rule: Always use `deallocate(access_obj)` to dereference allocated memory (r).
- ❏ New rule: Length of entity names should not exceed 32 characters (r).
- ❏ New rule: Multidimesional arrays can lead to mismatch between TRF file and Verilog netlist (e).
- ❏ New rule: Component instantiations (e).

1.5.3 Changes wrt. VHDL Coding Guidelines v3.0

- ❑ Changed rule: A configuration declaration is needed for each architecture (m).
- ❑ Changed rule: Memory instantiation (e).
- ❑ New rule: Do not use combinational feedback loops (m).

1.5.4 Changes wrt. VHDL Coding Guidelines V2.5

- ❑ Removed rule: Insert a power-pin for each power-domain (m).
- ❑ New rule: Use only lower-case names for unit diretories (e).
- ❑ New rule: Do not use FSM attrbutes within the VHDL code (m).
- ❑ New rule: Entity and architecture may not be stored within the same file (m).
- ❑ Changed rule: Generics must have type integer (m).

1.6 Literature

IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987, to be ordered from Menchini & Associates, P.O.Box 71767, Durham, NC 27722-1767, USA, fax: +1 919-479-1671, email: mench@mench.com, WWW: www.mench.com

- ❑ VHDL BNF as hypertext
- ❑ VHDL Glossary, VHDL International User's Forum
- ❑ Frequently Asked Questions, VHDL International User's Forum
- ❑ Newsgroup comp.lang.vhdl
- ❑ VHDL Modeling Guidelines, European Space Agency, Sept. 1994, Postscript, 118 Kbytes
- ❑ Guidelines for Writing VHDL Models in a Team Environment, J.Bergeron, Bell-Northern Research Ltd. , Postscript, 290 Kbytes
- ❑ VHDL Simulations -Tips for Speeding, V.Madisetti
- ❑ Synopsys Online Documentation - VHDL Compiler Reference Manual
- ❑ VHDL Modeling Guide II (V.Preis et al., Siemens AG, ZT ME 5)
- ❑ Basic Design Rules, Siemens HL TS
- ❑ VHDL Coding Style, (Project SL), Ver. 02, H.Hiller, 18.5.1995, Siemens HL
- ❑ VHDL Coding Style Guidelines, Ver. 0.3, T.Baciglupo, Siemens HL
- ❑ OMI (Open Microprocessor systems Initiative), 1996, Lucent Technologies, Bell Labs Innovations