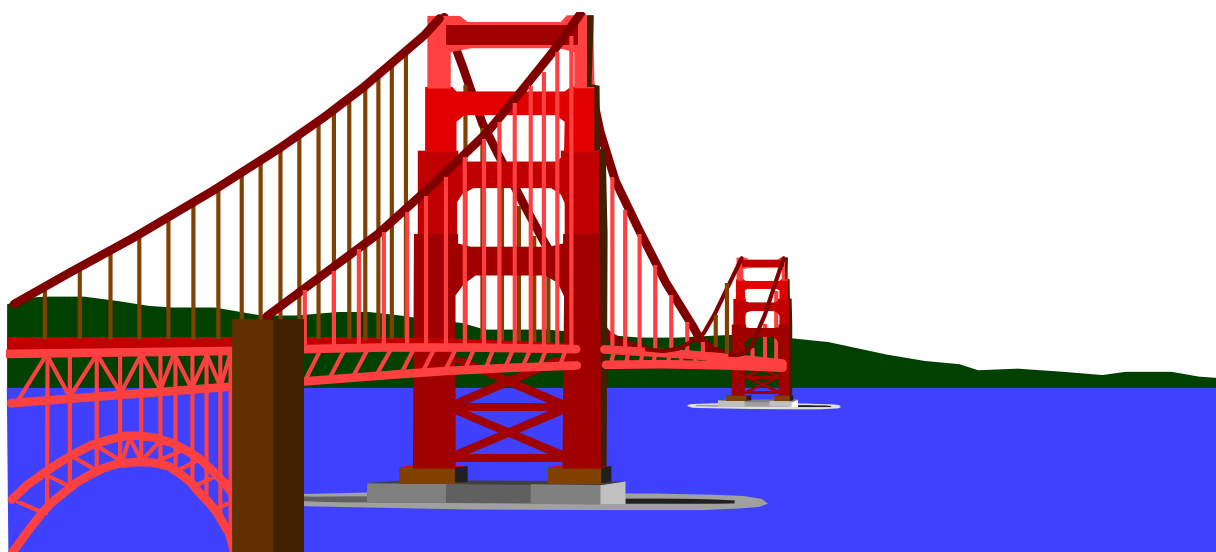


Design Style Guide

Hardware design Guideline with Verilog-RTL

Rev4.02 02/05/17



Editor : Toshiba Corporation
hd Lab, inc.

hdLab

Foreword

As design circuit size gets larger, efficient reuse of design property has become extremely important. The design property, which is also called IP (Intellectual Property), is essential technology to realize System-on-a-Chip.

The organization called VSI (Virtual Socket Interface) is now trying to commonize IP to enable us to structure a system by purchasing IP from multiple vendors that a large-scale system can easily be integrated into a single chip. Therefore, a circuit, which is larger than conventional ones, can be designed within the similar timeframe.

Software IP is described in RTL (Register Transfer Level) by using mainly HDL (Hardware Description Language).

By having IP described in RTL, changes can easily be made to the HDL description. Also, in the parameterized descriptions, a circuit can easily be changed according to the object of an application.

These software IP are realized in gate level circuits at the end by using logic synthesis tools and by specifying design constraints and technology.

The design constraint includes specification of timing condition, area, and operation environment etc. as the required circuit performance. If the constraint given at the time of synthesis is not appropriate, the circuit may be synthesized incorrectly and therefore it becomes unlikely to meet the target performance. If description is not appropriate, it will also become hard to meet the target performance.

This Design Style Guide defines the design style to proceed with HDL design based on the IP reuse method.

Commonizing design style enables design methods such as description style, synthesis and verification, which differ in each designer, to be commonized. Therefore, designers other than the original author can modify descriptions or improve readability to facilitate understanding.

The Design Style Guide subjects to the RTL description by Verilog-HDL, verification by simulator, which supports Verilog-HDL, and synthesis by the Design Compiler of Synopsys.

Verilog-HDL syntax regulations and operations of Design Compiler are not included. For these, please refer to the manual of each tool.

The Design Style Guide consists of the following sections.

Section 1: Basic Design Constraints

introduces basic design constraints, which should be considered when starting design, that are naming conventions, design style, clock design, considerations for synchronous design and asynchronous design, and the way of thinking for hierarchy design.

Section 2: RTL Description Techniques

introduces the basic RTL description style necessary to create RTL description by Verilog-HDL, including the description style of combinatorial circuits and sequential circuits. How to use *always construct*, *function statement*, *if statement* and *case statement* etc., are also explained.

Section 3: RTL Design Methodology

on the assumption of circuit partitioned design, introduces the creation method of the function library, IP parameterization, design for test, low power consumption design and design data management.

By following this design method, IP reusability will be improved.

Section 4: Verification Techniques

introduces the techniques to execute simulation; parameterization to describe test bench, task usage and procedure of verification.

Section 5: Logic Synthesis by DesignCompiler

introduces commands, script examples and synthesis methods to synthesize circuit using Synopsys DesignCompiler.

Creating script is important to execute optimization, which satisfy required performance of circuit. Script examples include speed and area optimization example, combinational circuit, 2 phase clock and hierarchy synthesis.

All rights reserved by hd Lab, inc. No part of this publication may be reproduced without the prior written permission of hd Lab, inc.

Remarks	Contents
mandatory	Items, which should always be followed
recommend 1	Items, which should be followed, but will not cause problem if appropriate counter measures are taken.
recommend 2	Recommended design style itmes regarding issues on circuit quality, readability and the usage of various tools.
recommend 3	Recommended styles, but strict observation is not required. Cautions are needed, however.
reference	Reference items of design know-how, which are better to know

The remarks used in this document are categorized as follows.

Design Compiler, PowerCompiler, DesignWare, ModuleCompiler, VCS and VSS are the registered trademarks of Synopsys, Inc.

Verilog-XL, NC-Verilog, LeapFrog, CTGen and PBopt are the registered trademarks of Cadence.

ModelSim and SSTVerocity are the registered trademarks of Mentor Graphics Corp.

Table of Contents

Chapter1 Basic Design Constraints	1-1
1.1. Naming conventions	1-2
1.1.1. Basic naming conventions	1-2
1.1.2. Naming conventions of circuit and pin names should be considered by the hierarchy	1-6
1.1.3. Give meaningful names for signals	1-11
1.1.4. Naming conventions of include file, parameter and define (differ from VHDL)	1-13
1.1.5. Give register output names that suggest clocks or registers	1-16
1.2. Synchronous design	1-18
1.2.1. Clock synchronous design	1-18
1.3. Initial reset	1-20
1.3.1. Use asynchronous reset for initial reset	1-20
1.3.2. Reset line hazards	1-25
1.3.3. Be careful about external noise on an initial reset signal	1-27
1.4. Clocks	1-30
1.4.1. Modularizing clock generation circuits	1-30
1.4.2. Use clock tree synthesis for clock balancing	1-32
1.4.3. Gated clocks should be used with special care	1-34
1.4.4. Multiple clock systems	1-36
1.5. Handling of asynchronous circuits	1-38
1.5.1. Consider meta stable in signals between asynchronous clocks	1-38
1.5.2. Use memory in transfers between asynchronous same-period clocks	1-44
1.5.3. Guaranteeing the setup/hold margin for synchronous RAM	1-45
1.6. Hierarchical design	1-47
1.6.1. Consider limitations based on hierarchical scale	1-47
1.6.2. Make basic block FF output & combinational circuit input	1-50
1.6.3. Follow sub-block (the levels below basic clocks) constraints	1-52
1.6.4. Do not insert gates into upper levels of basic blocks	1-54
1.6.5. Separate the data path section from the controller	1-57
1.6.6. Designate buffer outputs in upper levels with 200,000 or more gates	1-59
Chapter2 RTL Description Techniques	2-1
2.1. Combinational logic	2-2
2.1.1. Use the always construct and function statements correctly (Verilog only)	2-2
2.1.2. Define combinational circuits using the function statement (Verilog only)	2-6
2.1.3. In a function statement , be careful to check arguments and bit width	2-8
2.1.4. Instructions for equation level descriptions (differs from VHDL)	2-10
2.1.5. Use conditional operator ((A)?B:C) only once (Verilog only)	2-13
2.1.6. Specifying the range of an array	2-15
2.2. always construct description in combinational logic	2-17
2.2.1. Avoid the risk of generating latches	2-17

2.2.2.	Define every input signal in an always construct in the sensitivity list	2-19
2.2.3.	Initial value description in always constructs (Verilog only)	2-21
2.3.	FF inferences	2-25
2.3.1.	Unify the description style of FF inferences	2-25
2.3.2.	Circuits will vary with non-blocking and blocking assignment statements (Verilog only)	2-30
2.3.3.	Do not mix descriptions that have different edges	2-32
2.3.4.	Do not specify an initial FF value in a description (differs from VHDL)	2-34
2.3.5.	Do not describe to generate FFs having fixed input values	2-35
2.3.6.	Do not mix FF inference with asynchronous resets and without	2-36
2.4.	Latch inference	2-39
2.4.1.	Clearly distinguish a latch inference from a combinational circuit	2-39
2.5.	Tri-state buffers	2-42
2.5.1.	Make a block for a tri-state buffer	2-42
2.5.2.	Consider high-impedance propagation in tri-state bus	2-47
2.6.	always construct description that takes circuit structure into account	2-48
2.6.1.	Describe taking the circuit structure into account	2-48
2.6.2.	Avoid defining multiple output signals in a single always construct	2-51
2.7.	if statements	2-54
2.7.1.	if statements create prioritized circuits	2-54
2.7.2.	Reduce conditional expression of if statement with the same contents	2-56
2.7.3.	Decrease the number of if statement nests	2-58
2.7.4.	Always surround multiple statements using block statements (begin-end) (Verilog only)	2-60
2.8.	case statements	2-62
2.8.1.	case statements facilitate decoder/encoder description	2-62
2.8.2.	Divide using if statement, etc. to avoid creating large tables	2-65
2.8.3.	Use default clauses	2-67
2.8.4.	Do not use complex casex statements (Verilog only)	2-71
2.8.5.	Description relying on parallel_case is prohibited (Verilog only)	2-73
2.8.6.	Beware of nesting that if statements and case statements coexist (2.8.4 in the VHDL version)	2-75
2.9.	for statements	2-77
2.9.1.	Do not use for statement for other than simple repeating statements	2-77
2.9.2.	Limiting loop-variable operation in for statements	2-79
2.10.	Operator description	2-81
2.10.1.	Order of operators and assignment of 'x'	2-81
2.10.2.	Efficient description using logical operation expressions (Verilog only)	2-85
2.10.3.	Match the bit width of the left side and the right side (Verilog only)	2-86
2.10.4.	Take note of the different data types between the left and right sides (Verilog only) ..	2-90
2.10.5.	Do not share resource in speed critical circuits	2-92
2.10.6.	Notes on arithmetic operations	2-94
2.10.7.	Take share items out of conditional branches	2-97
2.11.	State machine description	2-98
2.11.1.	Use Mealy type and Moore type descriptions properly	2-98
2.11.2.	Isolate state machine circuits	2-101

2.11.3. Separate FF inference and case statements	2-103
2.11.4. Consider the state allocation	2-104
 Chapter3 RTL Design Methodology	 3-1
3.1. Create function libraries	3-2
3.1.1. Create and utilize function libraries	3-2
3.1.2. Describe a style that takes reuse into account	3-3
3.1.3. Unify description order of module I/O ports	3-5
3.1.4. Consider RTL description readability	3-7
3.1.5. Parameterize the array range of module I/O	3-9
3.1.6. Parameter description using 'ifdef (Verilog only)	3-11
3.2. Using function libraries	3-13
3.2.1. Manage libraries in a common directory (differ from VHDL)	3-13
3.2.2. Define global parameters in separate files (differ from VHDL)	3-14
3.2.3. Connect ports by name for component instantiations	3-17
3.2.4. Use # (value) when overwriting parameters from an upper level (differ from VHDL)	3-18
3.2.5. The function library of an arithmetic operation improves data path design performance	3-20
3.3. Design for Test (DFT)	3-22
3.3.1. Clocks and Resets for DFT	3-25
3.3.2. Dealing With Hard Macros and Asynchronous Circuits	3-28
3.3.3. Constraints on the Use of Flip-Flops	3-30
3.3.4. Cautions When Using Latches	3-32
3.3.5. DFT in Clock Lines	3-34
3.3.6. DFT in Reset Lines	3-41
3.3.7. Handling of Different Clocks	3-45
3.3.8. DFT for Tristate Circuits	3-47
3.3.9. Handling Hard macros and Asynchronous Circuits, and Test Strategies for Large-scale ASICs	3-50
3.4. Low Power-Consumption Design	3-55
3.4.1. Low Power-Consumption Design Using Gated Clocks	3-55
3.4.2. Low power-consumption design hints by the logic circuits	3-58
3.4.3. Low power-consumption design hints by the clock tree	3-60
3.5. Source codes and design data management	3-61
3.5.1. Create a directory for each objective	3-61
3.5.2. File suffix names	3-64
3.5.3. Define necessary information for file header	3-65
3.5.4. Managing the version of a file	3-66
3.5.5. Periodically back up files	3-68
3.5.6. Use comments often	3-69
3.5.7. Using CVS when managing the versions	3-73
3.5.8. Using CVS basic commands checkout and commit	3-74
3.5.9. Managing versions using CVS (as example)	3-76
3.5.10. Check modified contents using the CVS history	3-78

Chapter4 Verification Techniques	3-1
4.1. Test bench description	3-2
4.1.1. Hierarchizing the test bench	3-2
4.1.2. Use basic test vector descriptions	3-3
4.1.3. Note input signal timing	3-5
4.1.4. Avoid assigning from multiple initial constructs (differ from VHDL)	3-7
4.1.5. Describe on a clock edge basis	3-8
4.1.6. Set the cycle using parameters	3-10
4.1.7. Define a signal for each clock in a multiple clocks design	3-11
4.1.8. Description where the results do not different due to the simulators (differ from VHDL)	3-12
4.1.9. Simulation between asynchronous clocks	3-19
4.1.10. Use handshakes when the process cycle count is unclear	3-21
4.1.11. Output simulation results to a file	3-24
4.1.12. Use PLI (Verilog only) (reference: There may be cases where PLI use is not be permitted due to use flows)	3-25
4.2. Task description	3-27
4.2.1. Describe using tasks to provide structure	3-27
4.2.2. Describe function operations using tasks	3-28
4.2.3. Pay due attention to task I/O arguments (differ from VHDL)	3-32
4.3. Verification Process	3-34
4.3.1. Making verification flow	3-34
4.3.2. Create test specifications	3-36
4.3.3. Create a simulation checklist	3-41
4.3.4. Clarify simulation results	3-46
4.3.5. Compare the expected value files with the simulation results	3-48
4.3.6. Simulating while comparing with the expected values	3-49
4.3.7. Efficient verification using a behavior model	3-52
4.3.8. Verification methods for large-scale design	3-55
4.3.9. Separate the function verification patterns and the fault detection patterns	3-57
4.3.10. Verification method using random function	3-58
4.3.11. Efficient debugging by embedding descriptions using 'ifdef (Verilog only)	3-60
4.4. Gate level simulation	3-61
4.4.1. Gate level simulation problems	3-61
4.4.2. Inconsistencies can occur between RTL and at gate level with..... the propagation of X	3-64
4.4.3. Beware of malfunctions caused by the timing	3-67
4.4.4. Other causes of mismatches between RTL and gate level	3-68
4.5. Static timing analysis	3-70
4.5.1. Key points of static timing analysis	3-70
4.5.2. Circuit design according to static timing analysis	3-73
 Chapter5 Logic Synthesis by DesignCompiler	 4-1
5.1. dc_shell basic commands	4-2
5.1.1. Command script example	4-2
5.1.2. read command (reading design)	4-4

5.1.3.	current_design command (design specification)	4-6
5.1.4.	set_operating_conditions command (specify operating conditions)	4-6
5.1.5.	set_wire_load_model command (wire load model specification)	4-8
5.1.6.	The create_clock command (clock definition)	4-10
5.1.7.	set_input_delay, set_output_delay (input, output delay setting)	4-12
5.1.8.	The set_driving_cell command (drive strength setting of input port connection)	4-14
5.1.9.	The set_load command (set output port load)	4-16
5.1.10.	The set_max_area command (set area constraints)	4-17
5.1.11.	The compile command (logic synthesis and circuit optimization)	4-18
5.1.12.	The report command group (synthesis results report)	4-19
5.1.12.1.	report_timing	4-19
5.1.12.2.	report_constraint	4-21
5.1.12.3.	report_reference	4-23
5.1.12.4.	report_area	4-24
5.1.13.	The write command (saving a design, generating a netlist)	4-24
5.1.14.	The check_design command	4-25
5.2.	dc_shell advanced commands	4-26
5.2.1.	Setting the hold time guarantee	4-26
5.2.2.	Hierarchical synthesis (uniquify, set_dont_touch, ungroup, compile -inc)	4-30
5.2.2.1.	uniquify	4-30
5.2.2.2.	set_dont_touch	4-31
5.2.2.3.	ungroup	4-31
5.2.2.4.	Hierarchical synthesis (compile -incremental_mapping)	4-32
5.2.2.5.	set_clock_transition	4-33
5.2.2.6.	remove_unconnected_ports	4-34
5.2.3.	Flatten (set_flatten)	4-35
5.2.4.	Structuring (set_structure)	4-36
5.2.5.	The group_path command	4-37
5.2.6.	set_false_path	4-39
5.2.7.	set_multicycle_path	4-40
5.2.9.	fanout, capacitance, transition	4-41
5.2.8.	write_script, write_sdf	4-41
5.2.10.	Variable of Design Compiler better to be specified	4-43
5.3.	Basic principles of Synthesis	4-45
5.4.	Script examples	4-48
5.4.1.	Area optimization	4-48
5.4.2.	Speed optimization	4-49
5.4.3.	Circuit synthesis consisting of only combinatorial circuit	4-54
5.4.4.	Multiple clock optimization	4-55
5.4.5.	Hierarchical optimization and its concept	4-59
5.5.	characterize optimization	4-63
5.6.	Circuit synthesis including operators	4-66
5.7.	2001.08 performance and comparison with past versions	4-69
5.7.1.	Improving performance in execution speed	4-69
5.7.2.	Improving speed performance	4-70
5.7.3.	Performance degradation by if statements and case statements	4-71

5.7.4. New functions in 1998.03 through 2001.08	4-74
5.7.4.1. Operating environment MAX, MIN support (useful)	4-74
5.7.4.2. Time budgeter (useful but rarely used)	4-74
5.7.4.3. Selecting implementation (cla, rpl) when compile -inc is specified (caution)	4-76
5.7.4.4. Taking TNS (Total Negative Slug) and subsequent delay paths into account (caution)	4-76
5.7.4.5. The new boolean optimization function (useful but not effective)	4-76
5.7.4.6. Multi-bit cell support (unnecessary)	4-77
5.7.4.7. set_simple_compile_mode (from 1999.05, updated in 2000.11)	4-78
5.7.4.8. compile -top (from 1999.05)	4-78
5.7.4.9. propagate_constraints (1999.05)	4-79
5.7.4.10. ACS compile (2000.05 or later)	4-79
5.7.4.11. case_analysis (2000.11 or later)	4-80
5.7.4.12. clock gating (2000.11)	4-81
5.7.4.13. set_isolate_ports (2001.08)	4-81

A. Appendix

A-1. Index

Chapter 1 Basic Design Constraints

This chapter introduces naming conventions and synchronous design issues that should be kept in mind during the design process, as well as considerations and cautions relating to asynchronous design, clocks, and hierarchical design.

Contents

- 1.1 Naming conventions
- 1.2 Synchronous design
- 1.3 Initial reset
- 1.4 Clocks
- 1.5 Asynchronous circuits
- 1.6 Hierarchical design

1.1. Naming conventions

1.1.1. Basic naming conventions

[1]	File names should be as follows: "<module name>.v"	recommend 2
[2]	Only alphanumeric characters and the underscore '_' should be used, and the first character should be a letter of the alphabet	mandatory
[3]	Key words in Verilog-HDL(IEEE1364), VHDL(IEEE1076.X) may not be used	mandatory
[4]	Names containing "NC_0", "NC_1", "_TSB", "VDD", "VSS", "VCC" or "GND" (uppercase or lowercase) must not be used	mandatory
[5]	Do not distinguish names by using upper or lower case English letters (Abc, abc)	mandatory
[6]	Do not use an '_' (underscore) at the end of the primary <i>port name</i> or <i>module name</i> , and do not use '_' consecutively	recommend 1
[7]	Add an identifying symbol at the end of the name so the polarity of negative logic signals is clearly identified ("_X", "_N", for example)	recommend 2
[8]	<i>Instance names</i> should basically be the <i>module names</i> . <i>Instance names</i> that are used more than once should be "<module name>_<quantity>"	recommend 3
[9]	At the top level, <i>module names</i> and <i>port names</i> should consist of 16 or fewer characters and should not be distinguished by upper or lower case English letters	recommend 1
[10]	Do not use the same <i>instance name</i> or <i>cell name</i> as the ASIC library being used	mandatory
[11]	All names should be within 40 characters in length	mandatory

Explanation

Verilog-HDL Keywords (lower case)

```

always,  and,      assign,  begin,   buf,      bufif0,  bufif1,
case,    casex,    casez,   cmos,    deassign, default, defparam,
disable, edge,     else,     end,      endattribute, endcase,
endmodule, endfunction, endprimitive, endspecify, endtable,
endtask, event,    for,      force,   forever, fork,    function,
highz0,  highz1,  if,       ifnone,  initial, inout,   input,
integer, join,     large,    macromodule, medium, module,
nand,    negedge,  nmos,     nor,     not,      notif0,  notif1,
or,      output,   parameter, pmos,    posedge, primitive, pull0,
pull1,   pulldown, pullup,  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,    trile,   triand,  trior,
triereg, unsigned, vectored, wait,    wand,    weak0,   weak1,
while,   wire,     wor,     xor,     xnor

```

VHDL Keywords (not exist in Verilog-HDL)

abs,	access,	after,	alias,	all,	architecture,	array,
assert,	attribute,	block,	body,	buffer,	bus,	component,
configuration,	constant,	disconnect,	downto,	elsif,	entity,	
exit,	file,	generate,	generic,	group,	guarded,	impure,
in,	inertial,	is,	lable,	library,	linkage,	literal,
loop,	map,	mod,	new,	next,	null,	of, on,
open,	others,	out,	package,	port,	postpond,	procedure,
process,	pure,	range,	record,	register,	reject,	rem,
report,	return,	rol,	ror,	select,	severity,	signal,
shared,	sla,	sll,	sra,	srl,	subtype,	then,
to,	transport,	type,	unaffected,		units,	until,
use,	variable,	when,	with			

Examples of negative logic *signal* names

SIGA_X, reset_X, EN2B_X, enb_X

Examples of *instance* names

```
Hard_rend Hard_rend( .CLK(CLK), .RST_X(RST_X), ...
USBintf    USBintf_0( .CLK(CLK), .RST_X(RST_X), ...
USBintf    USBintf_1( .CLK(CLK), .RST_X(RST_X), ...
```

Example 1-1 Basic naming conventions

Identifiers such as *module names*, *instance names*, *signal names* and *port names* must facilitate understanding of the HDL description functions. Names that make debugging more efficient should be carefully chosen because if different naming conventions are used by different designers, circuits that were divided into sections by multiple designers will be difficult to understand when they are integrated together. This is why consistent naming conventions are important. The script codes will be more concise if a uniform naming convention is used throughout the entire design, in order to automatically analyze or modify the circuit structure further using scripts in the design tools.

A single file should contain a single module, but this is unfit for a large design with many files since it becomes difficult to handle them. In this case, multiple modules can be included in a file, but at least modules, which have no relation one another, should not be included in one file. A single file should include modules, which have a tree hierarchical structure. The top *module name* and *file name* should be the same.

In the case of Verilog-HDL, the characters `!`, `@`, `#`, `$`, `\`, `{`, `}` can also be used if an Escape identifier is used. However, since this cannot be used with VHDL(87) and there is a risk of problems occurring in the system at a subsequent stage, it is recommended that no characters other than alphanumeric characters and the `'` (underscore) be used.^[2]

Since keywords defined in the Verilog-HDL language specifications are in lower case, a designer would have the advantage of names being easily distinguishable within the code, provided that identifiers such as designer-defined I/O *signal names* were defined in upper case letters. Since Verilog-HDL is case-sensitive, it is possible to define an identifier in upper case letters in situations such as this with the same spelling as the keyword, although it is not recommended because it could lead to confusion (INPUT, TASK for example). Not only just the Verilog-HDL keywords, but also software and VHDL keywords that are used at later stages should not be used.^[3]

The use of EDIF, SDF and Windows keywords will not cause problems. Nevertheless, if need be, refer to the following section to help avoid any possible confusion.

Avoid using as SDF, EDIF keywords:

ABSOLUTE, cell, celltype, edif, DELAY, HOLD, IOPATH, NET, VIEW, SETUP

Avoid using as a Windows keyword:

CON, AUX, COM1, COM2, COM3, LPT1, LPT2, PRN, NUL

When a naming convention is specified by a semiconductor technology flow in use, the convention is prioritized. Specify keywords, enabled characters, and the limitation of characters in accordance with the conventions. As the semiconductor naming convention often limits the usage of upper case or lower case letters, modification is easily achieved by logic synthesis, etc., as long as name is not distinguished solely by the difference of upper case or lower case. Since there are some systems that do not distinguish between upper and lower case letters, all names that would become identical to another name if the case of the letters alone is changed must be avoided (Abc, abc for example).^[5] The use of upper and lower case letters in *comment statements* is acceptable.

In VHDL there is a convention, which states that the final character must not be '_'. This character is sometimes used in gate level verification by VITAL, so please refrain from using '_' at the end of top level *module names* and *port names*. In addition, it is also forbidden to use '_' consecutively.

Add an identifying symbol ("_X", "_N", for example) at the end of negative logic signals.^[7] If an identifier is added at the top of signals, like "X_", it becomes difficult to distinguish it from identifiers of hierarchies or identifiers delimiting *function*. Therefore, to identifying negative logic signals, delimit with "_" at the end followed by identifying characters such as 'X' and 'N'. If adding an identifier of clock system at the end, an identifier of negative logic signal can be just before it. Describe the identifier, which is used, in the document. As a rule, the same name should be used for the *module* and the file. This makes it easy to generate script when using a simulator or a logic synthesis tool.

Instance names should be based on the *module name* and "<module names>_<quantity>" if multiple *instances* exist.^[8] If an *instance name* is used that does not conform to this naming convention (all the rules of user definition), describe it in the document.

In order to be used as the IP core, *module names* must use only alphabetical letters or numbers and be 16 or fewer characters in length in the top level. *Port names* should be 16 or fewer characters in length, and use alphabetical letters, numbers, or '_' (names beginning or ending with '_' and names that use '_' consecutively are prohibited). To support systems that are not case sensitive, names should use only upper case or lower case and cases should not be mixed.^[9]

Refer to the following sections for the rules of file naming, *signal* naming, pin naming and *module* naming, etc .

- “1.1.2. Naming conventions of circuit and pin names should be considered by the hierarchy
- “1.1.3. Give meaningful names for signals”
- “3.1.3. Unify the description order of *module* I/O ports”
- “3.5.2. File suffix names”

RTqualify checks all the items of 1.1.1.

Characters used and the limitation on the number of character may be changed by variable in rule file.

- 1111a(W2) File name "<file_name>" does not match module name "<module_name>".
- 1111b(W3) Multiple unrelated modules exist in file "<file_name>".
- 1112a(E) Cell name is not of format specified by cell_naming_style.
- 1112b(E) Top layer module name is not of format specified by top_module_naming_style.
- 1112c(E) Module name is not of format specified by module_naming_style.
- 1112d(E) Function name is not of format specified by function_naming_style.
- 1112e(E) Instance name not of format specified by instance_naming_style.
- 1112f(E) Signal name is not of format specified by signal_naming_style.
- 1113a(E) Word same as reserved word in Verilog-HDL, including capital letters, is used
- 1113b(E) <category_file_name> reserved word "<*_name>" used as a [signal name | port name | instance name | cell name | function name | task name | parameter name].
- 1114a(E) "<*_name>", which includes <category_file_name> reserved word "<reserve_word>", used as a [signal name | port name | instance name | cell name | function name | task name | parameter name].
- 1114b(E) "<*_name>" is formatted in a way that is specified for reserved words.
- 1115a(E) Identical identifiers (except for different capitalization) used in same module.
- 1115b(W2) Identical identifiers (except for different capitalization) used in different module.
- 1116 (W1) Two or more adjacent *e_*f characters used in [module_name | port_name].
- 1117a(W1) No ID character for low-active signal. (Not to be checked by default setting)
- 1117b(W1) No ID character for low-active clock. (Not to be checked by default setting)
- 1117c(W1) No ID character for low-active set/reset signal. (Not to be checked by default setting)

Limited check is applied to 1117 as positive and negative logics can not be distinguished.

- 1118a(W3) Instance name "<instance_name>" neither same as the cell name "<cell_name>", nor includes the cell name.
- 1118b(W3) Instance name "<instance_name>" not of specified format.
- 1119a(W1) Top layer module name longer than <n=12> characters.

1.1.1.[10] can be checked by 1113b by registering the library name of ASIC vendor in category_file.

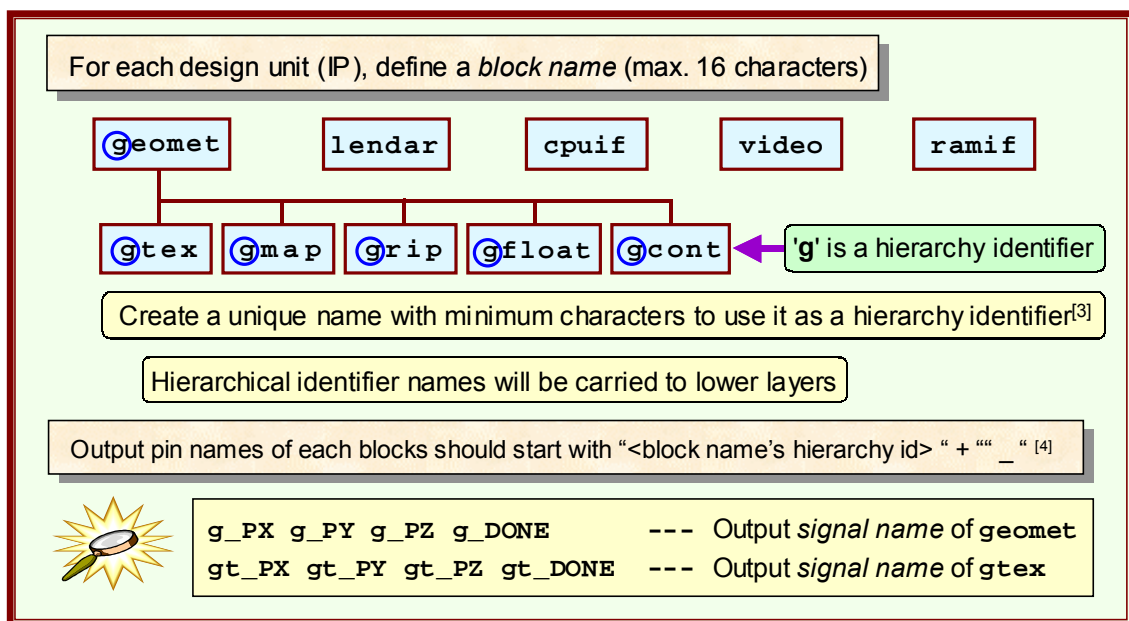
Naming conventions are explained in RMM:5.2.1. There is no limitation for the number of characters in RMM and negative logic has _n at the end.

1.1. Naming Conventions

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

- | | |
|--|-------------|
| [1] <i>Module names</i> and <i>instance names</i> should be between 2 and 32 characters in length
- The length of 16 or fewer characters is recommended (recommend 2) | mandatory |
| [2] Add a hierarchy identification character to the beginning of the top level block name | recommend 3 |
| [3] Add the hierarchy identification character of the upper level to the hierarchy identification character of the sub-block | recommend 3 |
| [4] The first string of the output pin name for each block should be "<hierarchy identification character>" + "_" | recommend 3 |
| [5] Naming conventions for input <i>signal names</i> and output <i>signal names</i> for each block should be different from those for internal <i>signal names</i> | reference |
| [6] Output pin names and the connected net names should be the same
- Upper level net names and the input pin names should be the same (recommend 2) | recommend 2 |
| [7] Do not use <i>signal names</i> that have sub-block <i>instance names</i> at the beginning | recommend 2 |

Example Code



Example 1-2 Naming following the convention

Explanation

Module name and *instance name* should be between 2 and 32 characters in length. Some ASIC vendors have a limitation of up to 32 characters that changing name by logical synthesis or others will be necessary if the number of characters exceed 32 in this case. *Instance names* should not be long because readability decreases when confirming a *signal* value of the 3, 4 or lower levels by a simulator or confirming timing analysis report. *Instance name* of 16 or fewer characters is recommended.

Level created by a designer may be ungrouped by tools, which is used at later stages. If it occurs, the *instance* names of levels between the top and the ungrouped level are added to the top of the *signal name* and *instance name* of the ungrouped one. It becomes problematic at later stages if level is deep and *instance name* is long. Therefore, *instance name* including level should be 512 or fewer characters.

A description can be made more readable and debugging efficiency improved by naming identifiers such as circuit names, *signal names*, and *instance names* in the HDL description with a uniform naming convention that follows the hierarchical structure. When a design calls for a circuit to be divided, if same *module names* exist, the circuit name (*module name*) may invite confusion in data management. Care must therefore be given to the *instance name (module name)* of each level of hierarchy.

The beginning of the *instance name* (level) placed in the top level will be the hierarchy identification character.^[2] Also, the hierarchy identification character of the top level will be added to the beginning of the *instance name* in the level under that.^[3]

Example 1-2 illustrates the use of a unique letter (g, l, c, v, r) as the hierarchy identification character for the first character of the *instance names* (5 *instances* in this case) placed at the top level. The sub-block name under the *geomet* block adds an identification character (gt, gm, gr, gf, gn, gc) at the beginning that includes the *geomet* level identification character (g). By adding a hierarchy identification character to the beginning of each hierarchy name in this manner, it becomes possible to prevent the same name from appearing.

The number of identification characters would be too large if the hierarchy identification characters of the levels under each sub-block consecutively inherit the hierarchy identification characters of its upper level.

(Example : *geomet*(g) -> *gtex*(gt) -> *gtlport*(gtl) -> *gtlcount*(gtlc))

It is not essential that all-upper level hierarchy identification characters be included in the hierarchy identification characters of a sub-block, but it is necessary to use at least the upper hierarchy identification characters at the beginning as well as be unique within designs.

Using the hierarchy identification characters for each level of hierarchy in the output pin names facilitates debugging.^[4] The initial character of the block should be added at the beginning. This is to make it easier to comprehend which block a signal is output from when debugging. Defining circuit names, *signal names*, and *instance names* that conform to the naming conventions makes it easier to examine a circuit during debugging and improves the readability of the code.

Be sure to follow these naming conventions during the entire design process. However, we do not recommend is the use of an instance name as the beginning of a signal name.^[7] If ungrouping *gtex* block from a upper level when the signal name in *gtex* block is *gtex_pt_x*, the instance name will become *gtex/gtex_pt_x* that is verbose. Therefore, we recommend to name the signal name of lower level as the hierarchy identification character(*gt_*, in this case).

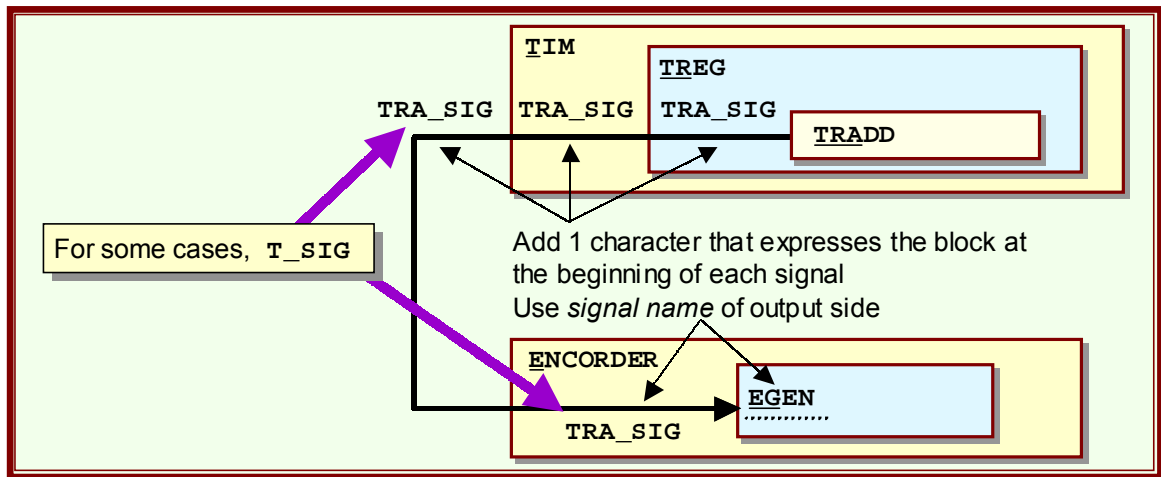


Figure 1-1 Naming convention according to hierarchical structure

Figure 1-1 illustrates the naming conventions for *module names* and *signal names* in circuits that have a hierarchical structure. The initial character in the string of the sub-blocks under the upper level must be a unique character within the same level. *Module names* under the sub-blocks add to their beginning one character that indicates a *module* in a top level, making it easier to understand the hierarchical structure by the *module name* alone.

Use output side *signal names* for *signal names* in the hierarchical structure. By doing so, it allows you to know which *module* that identifier is generated from. As illustrated in Figure 1-1, when the output *signals* from the lower level directly end up as an upper level output, the lower level output *signal name* is used as is as a general rule.

It would be useful to have a different naming convention for the input/output *signal names* for a block.^[5] One example is to use only upper case characters for the block's I/O *signal names* and use lower case for the internal *signal names*. This will simplify distinguishing between external *signals* and internal *signals*. But these I/O *signal names* are often converted to all upper case or lower case after synthesis and layout using an ASIC vendor's tools. Therefore, some suggest that either all upper case, or all lower case should be used. However, it is not easy to read names if all upper or lower case characters are used. As long as you follow the rules described in the item 5 of "1.1.1. Basic naming conventions"; "Do not distinguish names by using upper or lower case English letters", it will help improve readability to use both upper case and lower case characters.

It is best that the net name of the upper level to which an output *signal* is connected and input *signal name* are the same.^[6] The output *signal name* is used for an input *signal name* even when the output *signal* is input to multiple blocks because, during debugging, where signals are output from is more important than at what level they are used as input *signals*.

During the initial stage of a collaborative design by large numbers of designers, the output *signal name* of the blocks is sometimes unknown. In such a case, an input *signal* cannot follow the rules. At the very least, the output *signal name* and net name should be

the same. Top level output *signals* may lead to confusion if a hierarchy name is added, so hierarchy identification characters should not be added. This rule does not apply to *modules* used more than once as a design library or to the small sub-blocks of each person in charge.

When working with hierarchy on a larger scale, avoid using a *module* for multiple purposes because when logic synthesis is executed to the newly added *module name*, the content of each *module* will end up being different. It is confusing to have a *module name* that is different from the one at RTL description.

If a *module* is used more than once, the upper net name should basically consist of "<hierarchy identification character>" + "<number added to instance name>" + "_" + "<output signal name (except for hierarchy identification character)>".

```
module GEOMETOR ( ...
...
GFLOAT GFLOAT_0 ( .GF_DATA(GF0_DATA), GF_CARRY(GF0_CARRY),
GFLOAT GFLOAT_1 ( .GF_DATA(GF1_DATA), GF_CARRY(GF1_CARRY)
```

Example 1-3 The upper net name when using multiple instances

With regard to two-way bus signals, the upper net name and lower I/O *port* name should be the same and hierarchy identification character should not be added because multiple outputs exist.

RTqualify checks all the items of 1.1.2. All the naming conventions can be defined by regular expression. However, checks are based on limited specifications for 1122, 1123 and 1124.

- 1121a(E) Module name longer than <n=20> characters.
- 1121b(W3) Module name longer than <n=8> characters.
- 1121c(E) Module name shorter than <n=2> characters.
- 1121d(E) Cell name longer than <n=20> characters.
- 1121e(W3) Cell name longer than <n=8> characters.
- 1121f(E) Cell name shorter than <n=2> characters.
- 1121g(E) Instance name longer than <n=20> characters.
- 1121h(W3) Instance name longer than <n=8> characters.
- 1121i(E) Instance name shorter than <n=2> characters.
- 1122 (W3) No layer ID character added to layer name.
- 1123a(W3) Sublayer did not inherit layer ID character.
- 1123b(W3) Layer ID character not unique within a single layer.
- 1124 (W3) Layer ID character not used in output port name.

A large number of warning messages may be output for 1123 and 1124.
If not necessary, exclude them by ignore_message.

- 1125a(E) Not of format specified by `input_port_naming_style`.
 - 1125b(E) Not of format specified by `output_port_naming_style`.
 - 1125c(E) Not of format specified by `inout_port_naming_style`.
 - 1125d(E) Not of format specified by `top_input_port_naming_style`.
 - 1125e(E) Not of format specified by `top_output_port_naming_style`.
 - 1125f(E) Not of format specified by `top_inout_port_naming_style`.
 - 1126a(W2) Output port name "<name>" does not match net name "<name>" to which it is connected.
 - 1126b(W2) Net name "<net_name>" of higher net does not match input port name "<port_name>".
- 1126 will not be checked if same module is use for two or more.

With RTqualify, all the numeric values and `naming_style` can be modified by setting file.

Naming conventions are explained in RMM:5.2.1. RMM recommends to use lower case character for all the names.

1.1.3. Give meaningful names for signals

- [1] Naming conventions for *internal signals* of blocks should be different from those for input and output *signals* reference
- [2] Give meaningful and comprehensive names for *internal signals* of hierarchy reference
- [3] *Signal names, port names, parameter names*, define names and *function names* should be between 2 and 40 characters in length
- The length of 24 or fewer characters is recommended (recommend 2) mandatory

Example Code

_x, _N should be added at the end of *signal names* when using negative logic

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

➔

```
if ( !RESET_X )
  Q <= 1'b0;
```

Explicit naming of positive logic

```
if ( PLUS_or_MINUS )
  Q <= Q + 1'b1;
else
  Q <= Q - 1'b1;
```

➔

```
if ( PLUS )
  Q <= Q + 1'b1;
else
  Q <= Q - 1'b1;
```

Internal signals

```
DataMemWrite      - Memory write signal
DataMemAdr[31:0]  - Memory address
pos_GT_cos        - Comparison result
Counter_Load_Val  - Load signal to register
Hsync_conter_clear - Reset signal of counter
```

Parameter names

```
parameter P_Slot_length      = 4'd10;
parameter P_Timegrad_length  = 3'd3;
parameter P_StvRiseStartPoint= 0;
parameter P_StvFallStartPoint= 8'd111;
```

Example 1-4 Giving meaningful names

Explanation

Giving meaningful names to *signal names* and *port names* improves the readability of a description and enhances the efficiency of debugging.^[2] If the internal *signal name* of DataMemWrite in the Example 1-4 was DMW, it would be difficult for others to understand. Therefore, give meaningful name mixing upper and lower case letters.

If following “1.1.1.[5] Do not distinguish names by using upper or lower case English letters” strictly, only either one of upper or lower case letters might be used for all the *signal*

1.1. Naming Conventions

names, *port names* and *module names*. However, readability decreases if all of those names are in either one of upper or lower case letters. The item of 1.1.1.5 can currently be checked by RTL check tool. It is preferable to mix upper and lower case letters after checking by this kind of tools. Understandable name is preferable for increased readability than too simplified short name as long as the number of character is up to about 24(Max 40).

Naming conventions should be specified for *port name*, internal *signal name* and *parameter name* to distinguish between each other.^[1] For example, all the *port names* are in upper case letters and internal *signal* alone bases on lower case letters. Another example would be that hierarchy identification character is added to *port names* and only *parameters* are in upper case letters. In any case, consider naming conventions, which are easy-to-understand and unified.

The number of characters for *signal name*, *port name*, *parameter name* and *function name* should be between 2 and 40. A tool used at a later stage might convert a too long *signal name*. Although it is true that a long *signal name* is more understandable than a short one, a long name makes a number of characters in a line too many and readability decreases. Therefore, up to 24 characters in length is a base of a *signal name*.

RTqualify cannot judge whether a name is meaningful or not. 1.1.3.[3] to check the number of character is performed for each of signal name, port name, parameter name and define name and function name separately.

Naming conventions are explained in RMM:5.2.1. RMM recommends to use lower case character for all the names.

1.1.4. Naming conventions of include file, parameter and define (differ from VHDL)

- | | |
|---|-------------|
| [1] Use either ".h", ".vh" or ".inc" for RTL description and ".inc", ".ht" or ".tsk" for test bench as the <i>include</i> file (Verilog only) | recommend 2 |
| [2] <i>Parameter names</i> should have different naming convention | recommend 3 |
| [3] Do not use parameters with same name for different modules | recommend 3 |
| [4] Use <i>define statements</i> , declared in the same module only (Verilog only) | recommend 1 |
| [5] Add hierarchy ID characters to parameters, which are used only for within a level of hierarchy (reference) | reference |
| [6] Do not propagate parameters through ports | recommend 1 |
| [7] Parameterize the bit width of ports required for circuits that will be reused | recommend 3 |
| [8] Clarify <value> 'b, 'h, 'd, 'o specification for parameters | recommend 1 |
| [9] Specify the bit width if it is greater than 32 bits (Verilog only) | mandatory |

Example Code

Parameters used in the overall design (include file common.h)

```
parameter P_MaxDataPacketNum = 11'd1521
parameter P_MinDataPacketNum = 11'd61;
parameter P_Max_streage      = 14'd3853;
```

Parameters used in each layer(include file geomet.h)

```
parameter P_Geomet_Datalength = 8;
parameter P_Geomet_PxDefalut  = 10'd323;
```

Example 1-5 Naming parameters

Explanation

Whenever possible, put data to be used as parameters into *include* files and make it easy to change the parameter value. Distinguish parameters used for the overall design from parameters used only under particular hierarchies^[3], and place each one into a separate *include* file. In Verilog-HDL descriptions, use a relative path name for *include* files.

Even if there is an *include* file in the same directory, refer to the next higher level using "include ../RTL/compara.v". This should be done to prevent trouble from occurring when executing an EDA tool in another directory.

You should add the special identification character "P_" in front of a *parameter* to distinguish it from other *signals*. There are two methods to declare a *parameter*; using a *parameter statement* and using 'define. 'define is regarded as a compile directive and therefore becomes available inside *modules* other than the one in which 'define is declared.

1.1. Naming Conventions

In RTL description, 'define on global position and 'define in other *modules* should not be used. When generating a logic circuit with a logic synthesis tool, each *module* may be generated separately. In this case, it becomes impossible to generate a logic circuit. Of course, it is OK to call a file that defines 'define from an *include statement*.

There are cases when 'define may be used outside the *module* inadvertently, so please pay extra attention when using 'define. (Should be checked with an RTL check tool.)

We recommend that *parameters*, which are used in overall design, be defined with a *parameter statement* in the *include* files that are called by each *module*. To avoid unnecessary confusion, we recommend that you avoid using 'define as a parameter for use in overall design.

It may be preferable to add a hierarchy ID character, as defined in 1.1.2, to parameters that are used in each level of hierarchy as much as possible so that they are distinguished from other parameters.^[5] However, since the identifier "P_" is added to the top of a parameter, it may become complicated if another hierarchy ID character is added.

Avoid specifying directly to lower level ports those parameter values that have been specified in an upper level.^[6] Constant values provided to lower levels are implemented as connections to either the power supply or the ground. Such connections may cause errors as layout rules. In such cases, ungroup them and remove the port using commands.

In a *parameter*, describe 'b, 'h, 'd and 'o clearly when defining any numeric value greater than 8. In particular, when a value greater than 10 is specified, there is a possibility that a designer may mistake it for a hexadecimal number. For example, 12 is not 'h12.

Specify bit width in a *parameter* as much as possible. However, since one parameter value may be assigned to multiple signals with different bit width, it is not necessary to indicate it to all.^[8] Please note that *parameters* with no bit width specified have a bit width of 32. Specify the bit width when declaring *parameters* greater than 32 bits.^[9]

When using *constants*, use a *parameter* as much as possible so that check and modification may be easily done. However, if parameterizing all the *constants* such as 0,1, numeric value for which all the bits are 1, and clauses in *case statement*, readability will be decreased. In particular, parameterizing all the clauses in a *case statement* loses bit image (except for a state machine description) and therefore quality may be decreased. As for this type of clause, a constant value should be described as is, except when describing complete parameterization.

It is not always necessary to parameterize all of the constants. However, it is better to parameterize those which are used in many locations and where modification, such as bit width, is expected.

RTqualify checks the following items.

- 1141 (W1) Extension of include file "<file_name>" not <extension>.
- 1142a(W3) Not of format specified by parameter_naming_style.
- 1143 will be supported in the next version.
- 1144a(E) Different values are defined by 'define' in multiple places.
- 1144b(W1) Character string defined by 'define' used in another module.
- 1145 (N) Layer ID character not used in parameter name.
- 1146a(W1) A[parameter | constant] is used when connecting to an input port in a lower layer.
- 1146b(W1) [parameter | constant] is directly output to output port.
- 1147 to be checked by 3151.
- 1148 (W1) 'd, 'b, 'o, or 'h not specified for a parameter.
- 1149 (E) Bit width not specified for [constants | parameters] of more than 32 bits.

Note: the item 1.1.4 is the convention only for the Verilog-HDL version.

In the VHDL version, the item similar to this, "1.1.4 parameter (constant) naming conventions" is provided.

1.1.5. Give register output names that suggest clocks or registers

- | | |
|--|-------------|
| [1] Give register output <i>signal names</i> that suggest the clock system or register | recommend 3 |
| [2] Basically, use “CLK” or “CK” for clock signal names, “RST_X” or “RESET_X” for reset signal names and “EN” for enable signal names. Add identifier to the end of these basic names. | recommend 3 |
| [3] Names that suggest the clock | reference |
| [4] Names that suggest the register | reference |

Example Code

Example description that suggests the clock

```
always @(posedge CLK1 or negedge RESET_X) begin
    if (!RESET_X)
        GT_Preamble_CLK1 <= 0;
    else if (RE_EtherEnable_CK5)
        GT_Preamble_CLK1 <= GT_Seripara8bit_CLK1;
end
```

Example description that suggests the register

```
always @(posedge CLK1 or negedge RESET_X) begin
    if (!RESET_X)
        GT_Preamble_REG <= 0;
    else if (RE_EtherEnable_CK5)
        GT_Preamble_REG <= GT_Seripara8bit_REG;
end
```

Example 1-6 Naming that suggest the clock or the register

Explanation

In order to improve the readability of a description, a signal name based on the clock system or signal names, which explicitly identify that a signal is a register output signal, can be given to output signals of the register inference description.

First, decide the basic signal name for clock signal, reset signal and enable signal. Then add identifier to the end of the basic signal name when more than one signals of a same kind exist.

It is recommended to use basic signal names of “CLK” or “CK” for clock signal, “RST_X” or “RESET_X” for reset signal and “EN” for enable signal.

For example, if multiple clocks exist, add 1 to 3 characters to the end of “CLK” or “CK” like “CLK1”, “CLKM” or “CLK_CPU” etc.

Names, which suggest the clock system, can be given by adding the name of original clock, which supplies the signal, to the end of signal name (ex.”_CK5”).

It would be overly verbose to add clock identification to signals for the entire design. However, knowing which clock each signal is dependent upon is important in systems that

employ two-phase or three-phase latch based designs or use asynchronous transfer. The clock name should be added when designing such circuits.

To clearly distinguish between the *signal name* to be a register (FF, D latch) and the *signal name* of combinational logic, one option is to add "_REG" (_reg, if the *signal name* is in lower case letters) at the end.

However, in the logic synthesis tool Design Compiler, the *instance name* of a register is "<signal name>_reg" when a gate circuit is generated. If the *signal name* of register is GT_Preamble_REG, as in Example 1-6, the register name to deal by logic gate becomes GT_Preamble_REG_reg, which can lead to confusion.

RTqualify checks the following items.

- 1151a(W3) Clock signal name not added to register output signal.
(not to be checked by default setting)
- 1151b(W3) No ID character for a register output signal.
(not to be checked by default setting)
- 1152a(W3) Clock signal name not of format specified by clock_naming_style.
- 1152b(W3) Reset signal name not of format specified by reset_naming_style.

General naming conventions are explained in RMM: 5.2.1. RMM describes that "_r" should be used for "_REG", "_a" for asynchronous signal and "_z" for tri-state signal.

1.2. Synchronous design

1.2.1. Clock synchronous design

- | | |
|--|-------------|
| [1] Designs should use a single clock/single edge as much as possible | recommend 1 |
| [2] Do not create a RS latch or FF using primitive cells such as AND, OR | mandatory |
| [3] Do not use feedback in combinational circuits | mandatory |

Explanation

Use the synchronous design method in HDL and logic synthesis tools. Using asynchronous clocks makes adding precise design constraints difficult on logic synthesis. Utilize a single clock with a single edge in your design wherever possible.

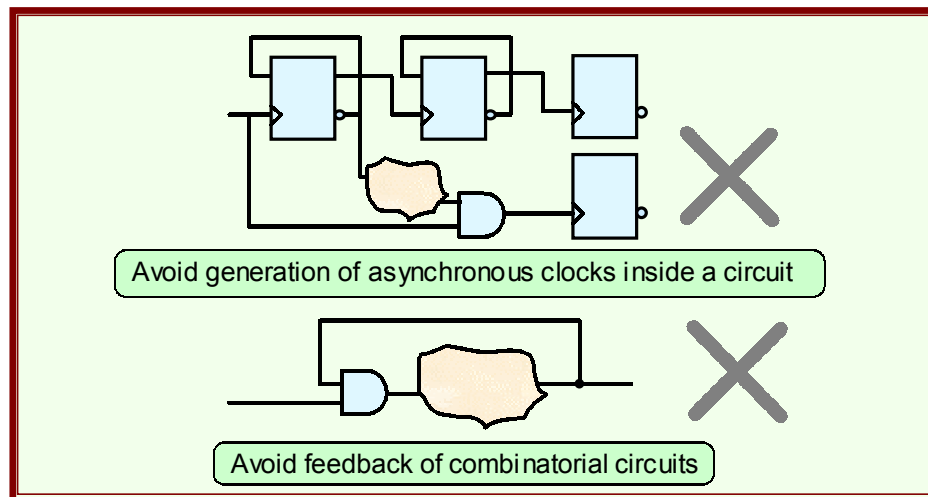


Figure 1-2 Asynchronous circuit and feedback of combinational circuits (bad example)

As designs grow larger, the circuit operating speed is analyzed using static timing analysis tools (Design Compiler, PrimeTime, BuildGates, etc.) instead of logic simulation. In such situations, analysis becomes difficult if the clock system is complex.^[1] In reality, there are not so many systems that operate with single clock and single edge.

If using multiple clocks, try to keep the number of clocks to as few as possible.

FF or latches can be created by using primitive cells, but this could be treated by the timing analysis tool as feedback to a combinational circuit.^[2]

If combinational circuit feedback cannot be avoided, use the `set_disable_timing` setting to avoid the effect of a feedback loop during timing analysis.^[3]

Refer to “4.5. Static Timing Analysis” for the setting method.

Circuit designs such as the example in Figure 1-2 above should be avoided, but if your design requires internally generated clocks, specify `set_create_clock` to the output of the FF that generates the clock.

RTqualify checks the following items.

- | | |
|---------|---|
| 1211(N) | Clock system indication. Judge clock system according to this system list |
| 1213(E) | Asynchronous loop. |

Verilint Warning

W408 : Combinational circuit loop is detected

W506 : Description, which may become combinational circuit, is detected

Synchronous design is explained in RMM 3.2.1.

1.3. Initial reset

1.3.1. Use asynchronous reset for initial reset

- | | |
|--|-------------|
| [1] Circuits which can not be reset by synchronous reset may be optimized | reference |
| [2] It is safer to use asynchronous reset for initial reset to a register
- Unlike ordinary paths, initial reset is not critical in timing
- Reset tree synthesis at layout is easy
- Values may not be fixed in a gate-level simulation with synchronous reset | recommend 3 |
| [3] Do not use asynchronous set/reset pins for anything other than initial reset | recommend 1 |
| [4] When using synchronous reset circuits, establish a new hierarchy for the register with synchronous reset | reference |
| [5] Do not use synchronous reset directives for a particular logic synthesis tool | recommend 3 |
| [6] Do not have both asynchronous reset and synchronous reset on the same reset line | mandatory |
| [7] Do not use a FF with both asynchronous set and asynchronous reset | recommend 1 |

Example Code

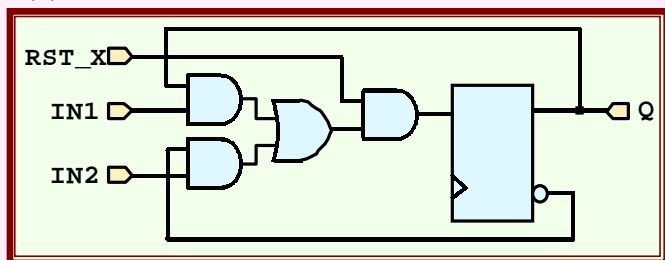
Error example of synchronous reset

```

always @( posedge CLK )
  if(RST_X == 1'b0)
    Q <= 1'b0;
  else if(Q == 1'd1)
    Q <= IN1;
  else
    Q <= IN2;
  
```

- With circuit (a), value is determined at initial state (Q is unknown)
- When circuit (b) is generated, a value is not determined at initial state

(a)



becomes...

(b)

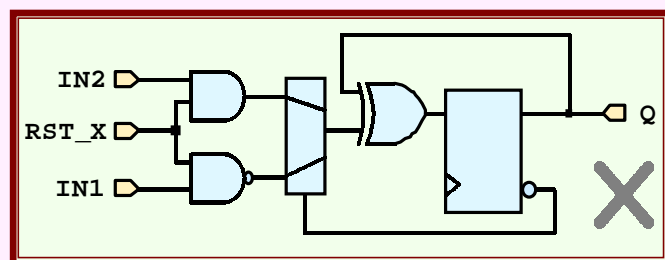


Figure 1-3 Synthesis by synchronous reset description

Explanation

Several different circuits can be optimized with the synchronous reset FF inference illustrated in Figure 1-3. For example, (a) is a circuit that resets after selecting the input signal and (b) is a circuit that resets the input signal and then selects the signal.

In logic simulation, the initial FF state is (X) (unknown), but because circuit (a) connects the reset signal to the gate directly before the FF data input, the data input is defined since an AND operation is performed with the reset signal even if the FF's output is 'X'. In circuit (b), however, the output signal from the FF that is input to the selector is 'X' and since the EXOR gate output becomes 'X' regardless of the input signal value, the FF data input is always 'X'. As a result, the value for the FF is not defined by the synchronous reset signal.

This will not cause any problems as long as a circuit (a) is always optimized from the synchronous reset description, but there is no guarantee that such a circuit is always optimized.^[1] Therefore, it is safer to use asynchronous resets in the initial reset.^[2]

Example 1-7 shows an example in which a synchronous reset FF description has been changed into an asynchronous reset FF description. The *always construct* is activated by the rising edge clock and the active low reset signal.

Asynchronous description example of initial reset

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

Example 1-7 Asynchronous description example of initial reset (mandatory)

In addition to the above-mentioned reason, asynchronous reset is more realistic for initial reset to create a reset line at layout(1.4.2. Use clock tree synthesis for clock balancing) and since some systems originally accept only asynchronous reset as described in “1.3.3. Be careful about external noise on an initial reset signal”.

Initial reset should be input for asynchronous reset, but other signals must not be input to the asynchronous set and reset pins^[3] because it is difficult to analyze the paths which the asynchronous set and reset pass through during the timing analysis. In other words, when using logic synthesis tools or static timing tools to perform an analysis, the timing path is cut without taking in account the timing from the register B reset input, as shown in Figure 1-4, to the Q output of register B.

Timing problems like this may occur if sets/resets other than initial reset are used, so we recommend that asynchronous set and reset not be used for the purposes other than initial reset.

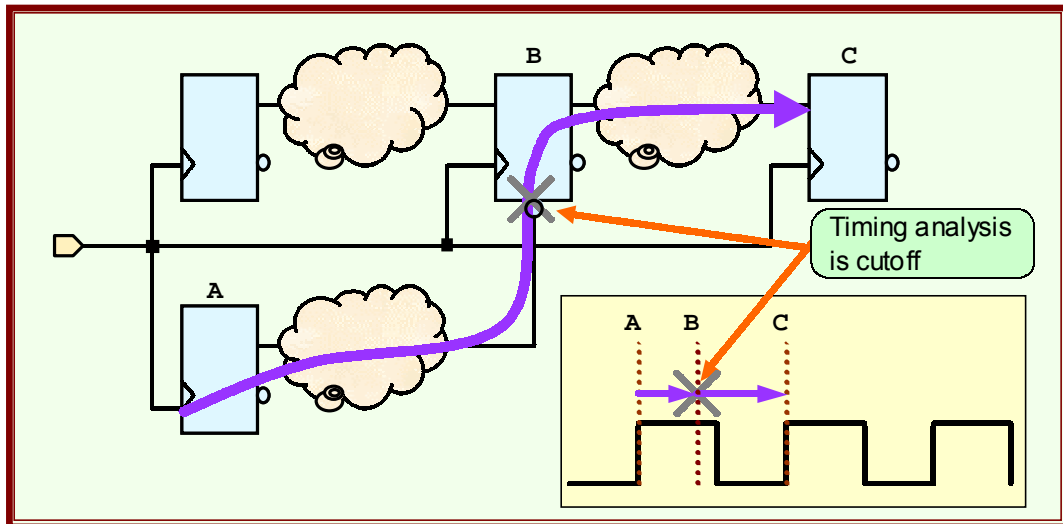


Figure 1-4 Timing analysis of asynchronous reset path

When using a circuit structure like the one in Figure 1-4 with a logic synthesis tool (Design Compiler), the timing of the path arriving at the B asynchronous reset from A is never analyzed as long as it is manually specified. Therefore, it is possible that no error report will be displayed even if this path has a delay of 1000 ns, and this will only be discovered after the layout has been completed.

Any resets other than initial resets should be synchronous resets, but there are situations in which logic that yields synchronous resets in a synthesized result similar to initial resets is separated from the FF and the FF cannot be initialized. As shown in Figure 1-5, for example, there are cases in which the logic that combines the reset signal and FF output signal is separated from the FF data input, and logic in which the data input FF is in unknown state 'X' cannot be initialized.

To avoid these cases by using a synchronous reset circuit, prepare a block consisted of only FFs with synchronous reset.^[4] Logic synthesis tools have a command called ungroup (Design Compiler) which damages this type of block. If the block is damaged by this command, values may not be fixed. Take care not to specify ungroup to this type of block.

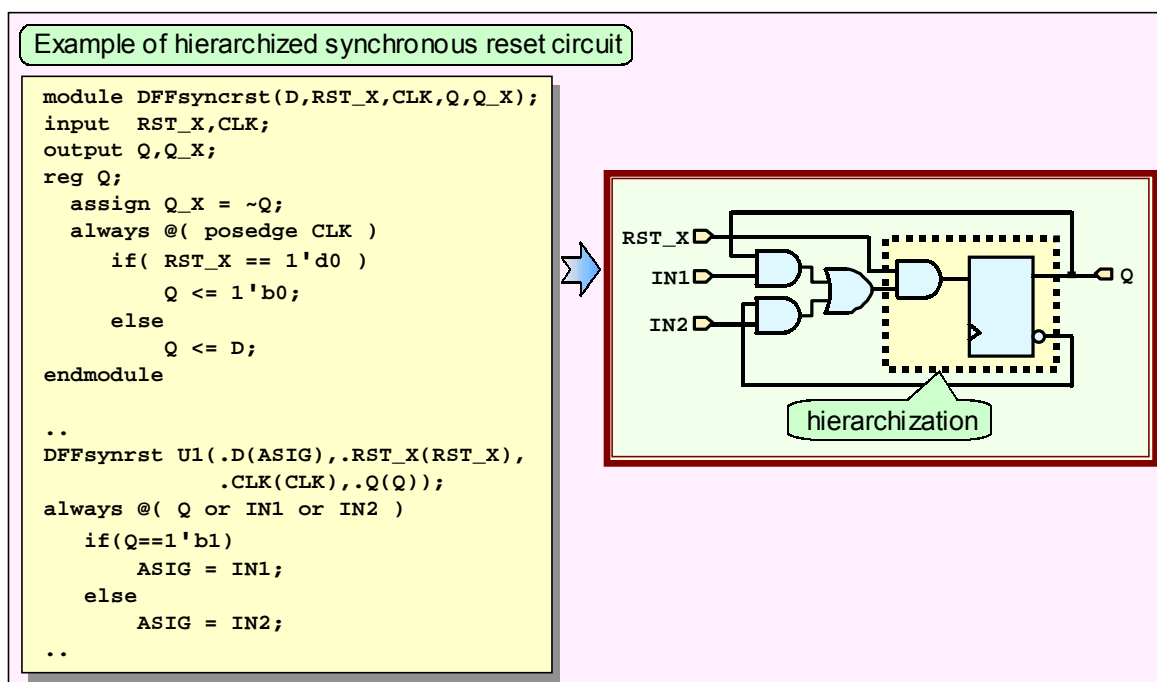


Figure 1-5 Synthesis of hierarchized synchronous reset circuit

If you add the Design Compiler specific directive `"/synopsys sync_set_reset"` you could assure a value with synchronous reset without using a FF level. Therefore, if you are not using a level, this property should be added. However, as this is only effective with Design Compiler, it will not be possible to use this RTL description as is if other logic synthesis tool is used in the future.^[5] Also, because this Synopsys attribute is realized in comment lines, syntax cannot be checked and it is recognized as a simple *comment statement* even if only a character is wrong. This method cannot guarantee that a tool always generate a circuit illustrated above. To secure that synchronous reset defines value at gate simulation, there is no other way but the hierarchization method as above.

If one reset line has both synchronous reset and asynchronous reset, synthesis may not be performed properly.^[6] The asynchronous reset line sometimes creates a tree during the layout process. To avoid inserting a buffer or logical operand by mistake during logic synthesis with Design Compiler, `set_dont_touch_network` (the setting with which this line is not modified during synthesis) may be put on this reset line.

Then, if this reset line is also input to a FF synchronous input, that part will not be synthesized and synthesis will fail as a result. Other than this, having both asynchronous reset and synchronous reset may cause problems during logic synthesis and layout, and they should not be mixed.

If you do not use any asynchronous reset other than an initial reset, you will only need either asynchronous reset FF or asynchronous set FF.

1.3. Initial reset

Do not use FF with both asynchronous set and reset.^[7]

```
always @(posedge CLK or negedge RST_X or negedge SET_X)
  if(!SET_X)
    QOUT <= #DLY 1'b1;
  else if(!RST_X)
    QOUT <= #DLY 1'b0;
  else
    QOUT <= #DLY DIN;
```

When RST_X is '0' and SET_X is '0' in this case, QOUT output becomes '1' as set is prioritized. If only SET_X changes to '1' here, this always construct does not operate that QOUT output remains as '1'. However, in actual FF gate behavior, it becomes '0' that RTL and gate level simulation results will not match. When RST_X and SET_X are both active, or actual FF is prioritize set or reset, RTL and gate level simulation results may not match that is hazardous.

RTqualify checks except for 1.3.1.[4]. Tools cannot recognize 1.3.1.[4].

- 1312(W3) Neither an asynchronous set nor a reset for an F/F in description.
- 1.3.1.[3] Checked by 1321 and 1322.
- 1315(W1) //synopsys sync_set_reset used in a synchronous reset.
- 1316(W2) An asynchronous reset or an asynchronous set is connected to a F/F data input path.
- 1317(W1) An F/F is used that has both an asynchronous reset and an asynchronous set.

Verilint Warning

- W396 : No asynchronous reset for flip-flop.
- W392 : The polarity of asynchronous rest is wrong.
- W395 : More than one asynchronous reset are detected.

Initial reset is explained in RMM3.2.4.

1.3.2. Reset line hazards

- | | |
|--|-------------|
| [1] Do not insert logical operands (AND, OR, XOR) in a reset line at the local level, and create circuits that supply resets as separate modules
- Logic order may be replaced by synthesis
- Hazards cannot be prevented in the RTL description | recommend 1 |
| [2] Do not insert signals other than initial reset to FF asynchronous reset pins | recommend 1 |
| [3] Optimize by keeping in mind the timing of the <i>signal</i> that outputs the reset line | reference |

Example Code

```

reg[4:0] count;
wire REN_X, EN_X, count32_x, ctl_x;

assign count32_x = ~(& count) | ctl_x;
assign REN_X = EN_X | count32_x;

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

```

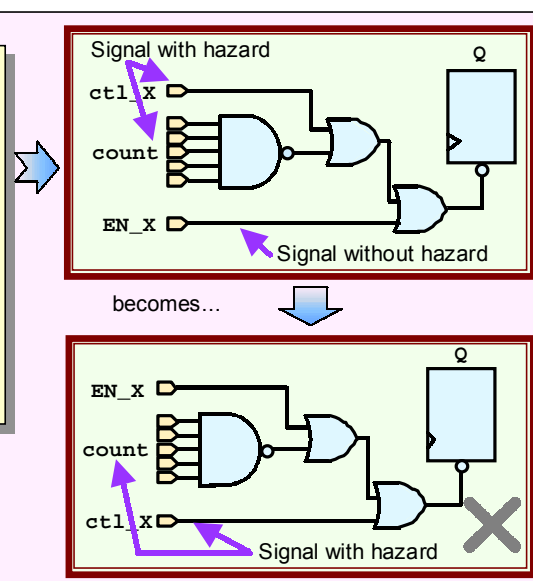


Figure 1-6 Example of hazard on the reset line

When a combinational circuit generates an asynchronous reset signal, as illustrated in the figure above, even if enable logic is inserted before the FF reset signal in an RTL description to avoid hazards, there are situations in which the enable signal would be separated from the FF as the result of optimization, and signals with a hazard may drive the reset input.^[3]

In this case, depending on the timing of the combinational circuit input signal, a hazard could develop in the reset circuit and the FF could be reset with unexpected timing.

It would be difficult to discover a problem such as this because hazard is prevented in the RTL description. As a rule, it is therefore forbidden to directly input signals other than the initial reset signal to the asynchronous reset pins.^[2] If, in creating such a circuit, it should be necessary to perform anyone of these, you should hierarchize this part in the same way as synchronous resets explained in “1.3.1. Use asynchronous reset for initial reset”.

As explained in “1.3.3. Be careful about external noise on initial an reset signal”, an asynchronous reset signal may be supplied as synchronized. Also, logic circuits may be inserted so that a system reset can be selected. When logic is needed on a reset line, always combine that reset line logic together in the top level as much as possible and the same signal should be directly input to all FFs.^[1]

RTqualify checks the following items.

1321(W1) Logic on an asynchronous set/reset line.

1322(W3) Output of an internal F/F used on an asynchronous set/reset line.

1.3.3. Be careful about external noise on an initial reset signal

- [1] There is danger of malfunction unless attention is paid to reset lines on the circuit board reference
- [2] An initial reset may have to be synchronized or else a noise elimination circuit may be needed recommend 3
- [3] In some systems, an initial reset signal is asserted before the clock
- The asynchronous method is preferred in this case, but you should add a noise elimination circuit reference

Explanation

As explained in “1.3.1 Use asynchronous reset for initial reset” an asynchronous reset is preferred for the initial reset, but operation tends to become unstable when slowly sloped waveforms or waveforms with a lot of noise are directly input from outside the LSI. Also, if the asynchronous mode is used for the reset, it may violate Setup, Hold since the rise point of the clock and the rise or fall of the asynchronous reset signal may occur at the same time. In this situation we would recommend synchronizing then distributing the waveforms as illustrated below.

If there is a significant concern about the influence of noise, you should use three or five FF stages, and reset should be only executed when all FF outputs are in the RESET state.

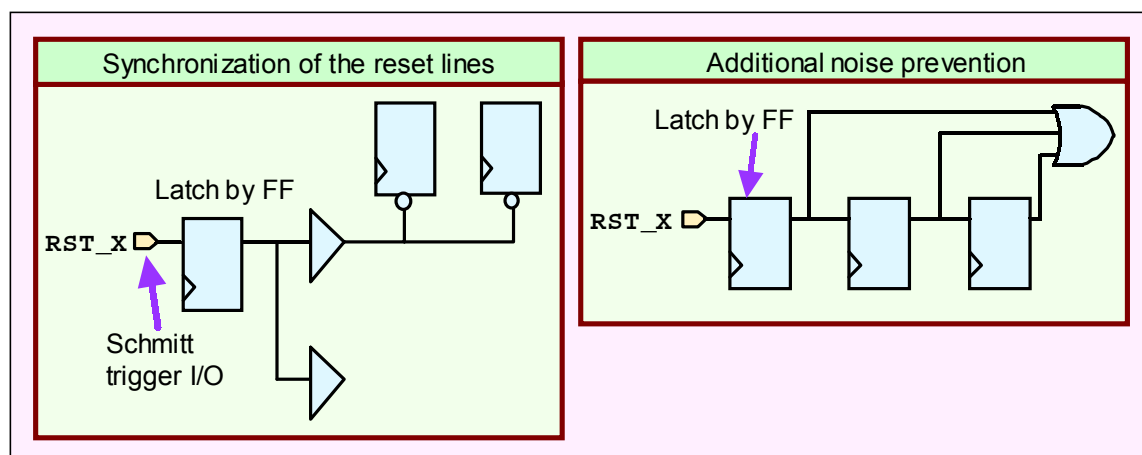


Figure 1-7 Synchronization of the initial reset signal

In some systems, the initial reset signal is asserted before the clock. Some portable home electrical appliances automatically turn on the initial reset at the time the power is turned ON. By loading the resistor and condenser on the reset pin, the voltage rise of the reset line, as shown in Figure 1-8, slows down in comparison to the voltage rise of power.

However, a clock signal generated by a PLL also will be slower in comparison to the voltage rise of power. In most cases the rise of the clock signal becomes even slower than the reset input. Thus, if the reset signal is synchronized by a FF, as shown in Figure 1-7, the system would have no reset.

Synchronization is not possible in this type of system, and the reset signal from outside the LSI, which should be supplied to FF asynchronous reset input, is input directly.

1.3. Initial reset

In this case, asynchronous reset signals are preferred, but it would be better to add some noise elimination circuitry.

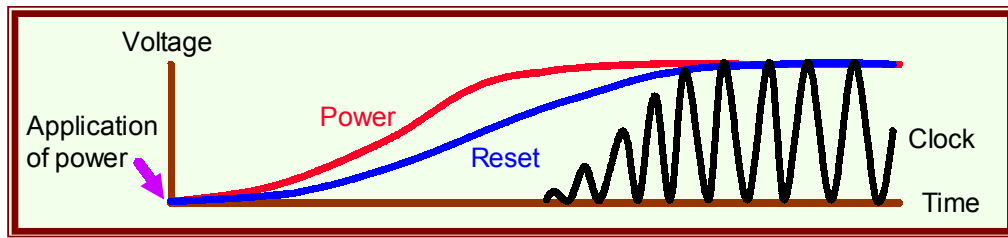


Figure 1-8 External power and initial reset input

As countermeasures, the use of a Schmitt trigger I/O, a VDD and GND for the I/O pin next to the reset pin to reduce noise (Figure 1-9), or the addition of a DLY element to prevent hazards as shown in Figure 1-10, maybe applied.

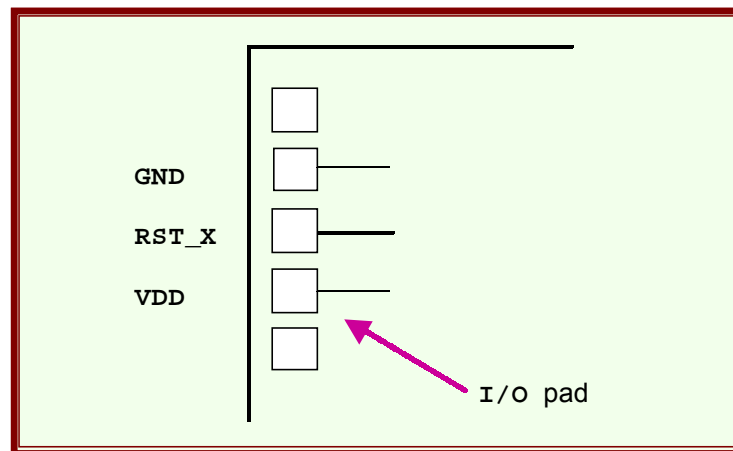


Figure 1-9 Use power pins beside reset pin to eliminate noise

In Figure 1-10, the value at B becomes '0' after the value at A becomes '0'.

If A again becomes 1 during the time span of the DLY cell (delay element), the OR output will be '1' as it is since B becomes '0' after A becomes '1'. Therefore, hazards can be prevented to a certain extent. Yet, in ASIC design, a DLY cell (or BUFFER) and OR gates are assigned in distant positions and may end up with values different from the assumed delay value. Take extra care when inserting such a circuit.

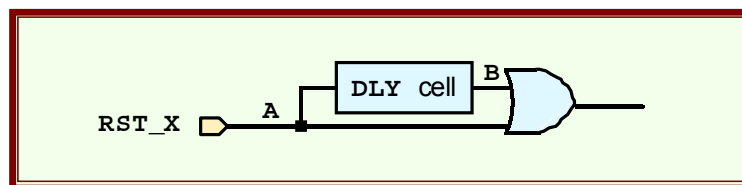


Figure 1-10 Noise prevention for reset line by DLY element

In addition, one countermeasure against asynchronous reset noise and malfunction due to Setup, Hold is to first create steady clock stop states and then meanwhile input an asynchronous reset between them.

However, no counter measure is perfect, so please give careful consideration thoroughly to how the reset line should be laid out on the board.^[1]

RTqualify outputs the following messages at top level mode (--top)

1332(W3) A signal from an external port is inputted directly into an asynchronous set/reset line.

1.4. Clocks

1.4.1. Modularizing clock generation circuits

- [1] Modularize circuits that supply clocks separately
- Put gated clocks and multiplication clocks together in a single level of hierarchy at the top level
- [2] Do not subject clocks to synthesis

recommend 1

reference

Explanation

Combine circuits such as gated clocks and multiplication clocks that supply clocks to the internal sub-circuits together in the same level as much as possible.

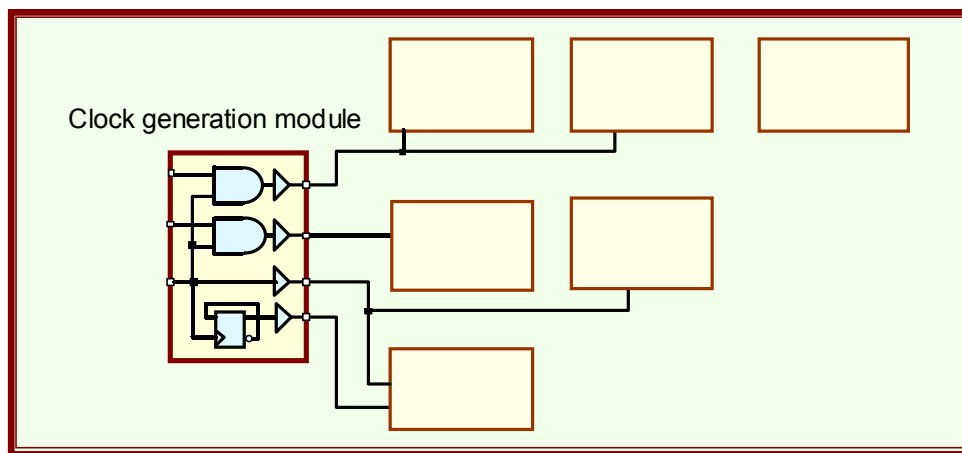


Figure 1-11 Creating a clock generation module

Creating a clock generation module and supplying clocks from a single clock generation module offer the following advantages:

- * Easy to manage clock distribution

Redundant circuits may be generated when the necessary clocks are created at each level of hierarchy. By centrally controlling clock generation, it is no longer necessary to consider clock generation at each level of hierarchy. Also, it becomes easier to use the clock tree optimization function during layout.

- * Easy to apply clock constraints to each level of hierarchy

Applying clock constraints to internal clocks generated from within a circuit is difficult. Defining clock constraints is facilitated by supplying clocks only from outside a level.

- * Easy to implement clock controls during test design

With scan design, it is necessary to enable controlling the clock signals from external pins during scan operation. Scan design is facilitated by putting clocks into separate levels.

RTqualify check s the following items.During top level mode (-top) and other than in clock generation modules,

1411a(W1) There is a gated clock.

1411b(W1) There is an inverted clock.

Clock modulation is explained in RMM:5.4.5.

1.4.2. Use clock tree synthesis for clock balancing

- [1] For clock lines, do not use primitive cells other than dummy buffers on a clock tree
- [2] Clock should be balanced in accordance with the number of cells to be connected to each clock tree

reference

reference

Explanation

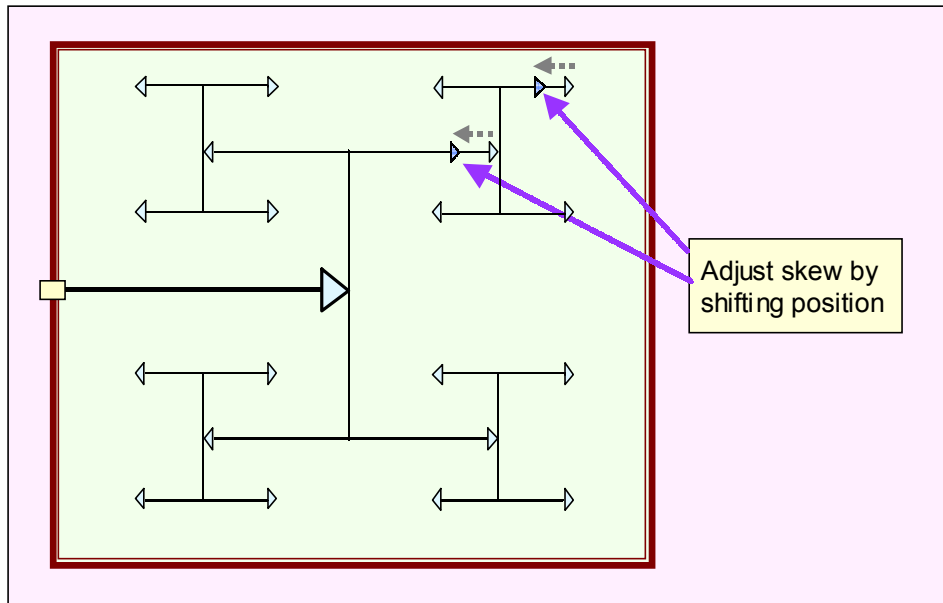


Figure 1-12 H-type clock tree optimization

In LSI design, the Clock Tree Synthesis (CTS) tool is used to synthesize the clock after the placement of each FF in the layout. Figure 1-12 provides an overview of the CTS. The clock lines extend from the center in an H-shaped tree structure. Wires from each terminal buffer are also arranged in H-shaped structures, supplying the clock to the local FF from the final terminal buffer.

In these clock lines, the arrival time will vary depending on the number of FFs ultimately supplied, and the length of the interconnect lines. The CTS tool adjust at this point by shifting the position of any given buffer forward to delay the arrival time.

To use the CTS, add a CTS dummy buffer to the circuit. The CTS dummy buffer is inserted either into the clock generator *module* during the RTL description stage, or after the gate-level net list has been generated. Any buffers or delay cells in the clock lines aside from CTS dummy buffers are not recommended. In particular, do not use buffers, etc., in lines after the CTS dummy buffer.

The CTS tool generates the clock tree after the CTS dummy buffer, and adjusts the clock *signal* arrival time. However, it does not adjust the arrival time between different clocks. To adjust the arrival time between different clocks, adjust by inserting delay cells (such as buffers) in the stage prior to CTS. If there is too many clock lines, this adjustment process becomes laborious.

The CTS tool capabilities have been enhanced recently, making it possible to adjust the delay values even when gated clocks are present in the clock lines. Even if gated clocks are present, the placement of a CTS dummy buffer in the stage prior to the gated clock will make clock adjustments easier during layout. However, when this approach is used, a single clock line will be encumbered by a large capacitance, reducing the degree to which power consumption in the clock line can be reduced. (See “3.4.Low power consumption design”) Although gated clocks are used to reduce power consumption, they are unable to produce as much of an effect as anticipated. Because of this, there is also the approach of fabricating a clock tree for each gated clock. Of course, asynchronous clocks having frequency dividers or differing periods will use different lines.

During the design of the LSI, consideration should be given to the number of these clock lines. Additionally, the designer should be aware of the approximate number of FFs connected to each clock line in order to consider the balance of the clock lines.^[2]

In parts that the interface with the CPU bus, the PCI bus interface, or the interface with external memory, it may be necessary to fine-tune the timing of the *signals* input to the LSI from the outside or output by the LSI to the outside. For these LSI external interface parts that require fine-tuning of the timing, it is recommended that the clock lines be divided in advance.

As is introduced in “3.3.Design for Test(DFT)”, scan registers are inserted into the LSI using the scan register insertion tool. These scan registers generally structure scan chains for each clock line. As circuits become larger and individual clock lines are connected to more FFs, scan paths may become too long, and too many test vectors are output from the automatic test pattern generator (ATPG) tool. Because of this, the designer may have to divide the clock lines. When it comes to how the clock lines should be divided, the answer is dependent on how the circuit is divided into blocks in the layout, and thus the clock lines often cannot be divided in the RTL design stage. However, it is recommended that the problem of power consumption (discussed above) and scan line insertion be considered to some degree when working on the clock lines.

Additionally, it is not possible to insert scan registers for each clock line if the number of clock lines is too large. When this is the case, it may be necessary to switch the clocks when in test mode to use the same clock lines. Moreover, because there is also the issue of detecting interferences between different clock lines (“3.3.7. Handling of different clocks”), the connections of *signal* lines between the various clock lines should also be considered.

RTqualify checks

1421(N) There is a buffer cell in a clock line.
It is up to designer to decide whether this cell is the buffer for CTS.

1.4.3. Gated clocks should be used with special care

- | | |
|---|-------------|
| [1] Avoid reversals on the same clock line, using gated clocks and using FFs with different edges | recommend 2 |
| [2] Do not input a FF output pin to other FF clock pins | recommend 1 |
| [3] Using gated clocks is an effective method for achieving low power consumption | reference |
| [4] Do not supply clock signals to pins other than FF clock input pins (such as D input) | recommend 1 |
| [5] Clock signals should not be connected to black boxes, bi-directional pins or reset lines | recommend 3 |
| [6] Do not use FFs with inverted edges | recommend 1 |

Explanation

As explained in “1.4.2. Use of clock tree optimizing for clock balancing”, the recent CTS tools (Clock Tree Synthesis tools) are now able to take clock tree balancing into consideration even when there are gated clocks or inverter clocks. However, as described above, great care must be taken when designing the clock lines.^[1] As a result, gated clocks or inverted clocks should be gathered in the clock generator module in the top level, and clocks should not be generated in the local levels. If you wish to consider moving to gated clocks on at the very detailed level in order to reduce power consumption that little bit more, use the “generation of gated clock circuits using EDA tools” explained in “3.4.1. Low power consumption design”.^[3]

The connection of the output of one FF to the clock pin of another FF should also be limited to the clock generator module located in the top-most level.^[2] When the output of an FF becomes another clock line, the CTS tool cannot take the clock line balancing into consideration.

Do not connect clock signals to anything except for the clock pins of FFs.^[4] If the clock line passes through a logical gate to arrive at the D input of the FF, the logic synthesis tool cannot perform optimization for that part. Additionally, such a path is extremely dangerous because the timing cannot be analyzed correctly. Use caution when fabricating a circuit wherein an external clock circuit is latched by a clock signal within the LSI (used, for example, finely dividing clock *signals*).

Clocks input from outside of the LSI may be connected to clock generator circuits (PLLs and DLLs). If there is no library provided for these clock generator circuits (PLLs or DLLs) so that they are “black boxes” errors will result when performing simulations and logic synthesis, so make sure that the libraries are present.

Be sure also to avoid outputting clocks to bi-directional terminals. Bi-directional terminals must be controlled as either inputs or as outputs during a simulation. Tools related to testing and the BIST insertion tool cannot determine the direction of these bi-directional terminals, which may lead to problems.^[5]

Depending on the ASIC library used, there may be two types of FFs: those that work on a positive clock edge and those that work on an inverted clock edge. When the two types of FFs are mixed in a circuit, the scan register insertion becomes problematic. It is best not to use FFs that work on an inverted clock.^[6] However, it is not a problem if latches that work on inverted logic are used in only a single stage (See “1.5.3.Ensure synchronous RAM setup hold margin”, “2.4.Latch description” and “3.3.7.Protection Between Different Clock Lines”).

RTqualify checks the followings.

- 1431a(W3) There is a gated clock.
- 1431b(W3) There is a inverted clock.
- 1431c(W2) There is description of a falling edge in clock line "<clock_name>".
- 1432a(W1) Output of F/F connected to a clock.
- 1435 (W2) A clock signal is used other than for an F/F clock.

Gated clock is explained in RMM:5.4.3 and 5.4.4.

1.4.4. Multiple clock systems

- [1] Divide up hierarchical blocks in each clock whenever possible
- [2] When inputting multiple clocks in the same block, provide an integral multiple period as a clock constraint

reference

reference

Explanation

Define clock generation circuits in different levels when clocks consist of two or more systems. See “1.4.1.Modularize clock generation circuits” for the benefits of placing clocks in different levels.

When creating sub-blocks, create a sub-block for each clock system as often as possible^[1] to avoid the racing problem (see “2.3.1.Unify the description style of FF inferences”) during simulation and to facilitate clock synthesis during layout. Even if creating a sub-block for each clock is difficult, avoid inputting multiple clocks to the same block whenever possible (Figure 3-13).

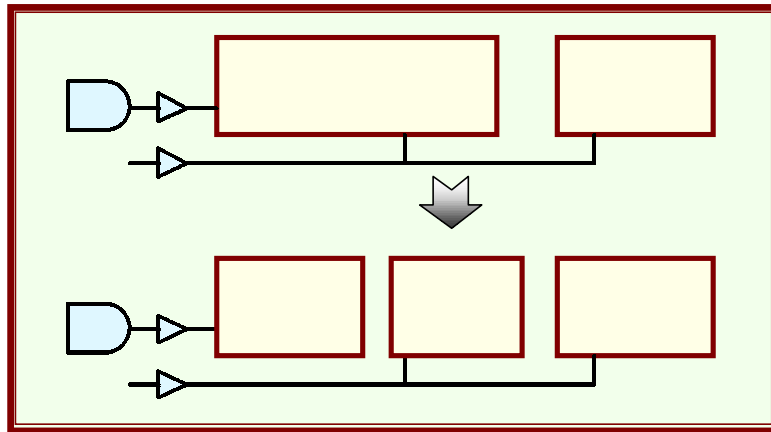


Figure 1-13 Creating block for each clock

If the speed of an external interface operation is stable and its operation frequency is lower than an internal clock when retrieving data from external interface *signals* such as a CPU interface, all the *signals* should be synchronized by the internal clock before a circuit to retrieve the data should be created.

The Figure 1.14 shows an example in which a typical CPU interface is created by full synchronization. In this case, the rise of the WR_X *signal* is detected and the values for ADR and DATA at that time are retrieved. This type of synchronization circuit can be applied only when CPU interface speed is one half or less than that of the internal clock. As the falling edge is used to write instead of the rising edge of the WR_X signal, it is valid only when CPU interface speed does not change.

The CPU interface may take the WR_X signal as clock and retrieve ADR and DATA as shown in the Figure below. In this case, pay attention to meta stable measures. In such CPU interfaces, since there are few circuits, which operate by WR_Xclock, the circuits become difficult to understand if they have FFs, which operate by an internal clock system as a separate block. It is not recommended that blocks be divided for each clock in a circuit such as this.

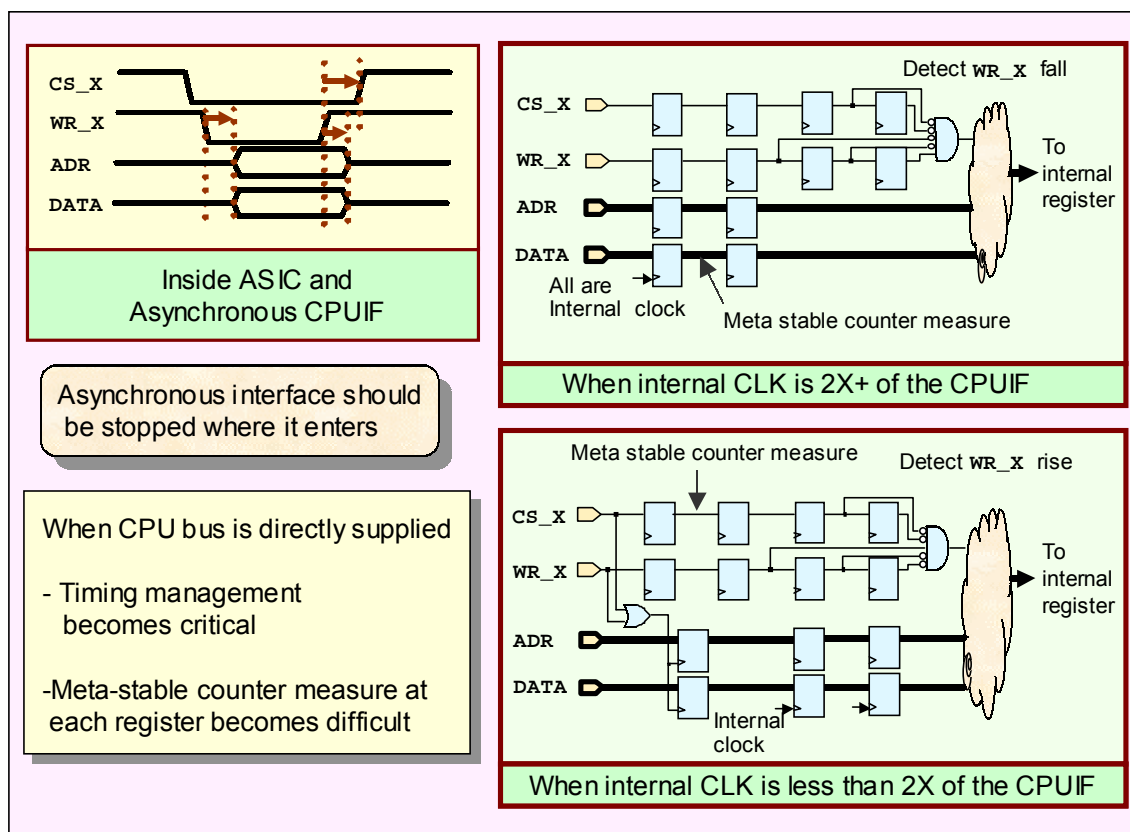


Figure 1-14 Creating CPU interface circuit

When synthesizing circuits that input multiple clocks to the same block, provide a multiple cycle relationship as much as possible.^[2] In the case of clocks that have a 12ns and a 6ns cycle, the logic synthesis tool optimizes with the basic cycle set to 12ns, which is the least common multiple. The basic cycle is 60ns for clocks with cycles of 12ns and 15ns, and is 84ns for clocks with cycles of 12ns and 14ns. As the least common multiple increases, optimization and timing analysis takes more time to complete. (See “5.4.4 Multiple clock synthesis” for more information.)

In case LCM cycle becomes large when giving actual clock cycle, modify circuit division or give clock cycle so that LCM becomes small even if the cycle at the synthesis becomes different from the actual clock cycle.

RTqualify checks

1442 (N) Multiple clock signals are inputted into a single module.

This is explained in RMM 5.4.5.

1.5. Handling of asynchronous circuits

1.5.1. Consider meta stable in signals between asynchronous clocks

- | | |
|---|-------------|
| [1] To avoid meta stable conditions, do not locate logic between asynchronous clocks | mandatory |
| [2] To avoid meta stable conditions, do not locate logic between FF for data retrieval and the next FF | recommend 1 |
| [3] Do not have a feedback loop at the first-stage FF after transfers between asynchronous clocks | mandatory |
| [4] Meta stable measures should be taken even for input from outside LSIs that are susceptible to noise | recommend 2 |
| [5] To avoid erroneous input data, latch the clock <i>signals</i> and use them as enable signals | reference |

Explanation

There is a difficult problem termed “meta stable” when transmitting data between asynchronous clocks. In order to solve meta stable problems, one must first understand the operating principles of flip-flops (FFs).

* The Operating Principles of FFs

FFs have specified setup times and hold times. In synchronous design, the design calls for the value of the input signal to remain unchanging for a specific period of time before and after the rising edge of the clock (termed the “setup time” and the “hold time”)(Figure 1-15). However, in asynchronous transmission, there is no guarantee that these specifications will be fulfilled. A portion of the data will violate the specifications for setup time and hold time. The meta stable problem may occur when the setup time or hold time specifications are violated.

Figure 1-16 shows the circuit structure of a MOS LSI FF. (1) and (3) in the butterfly shape are MOS switches. With the MOS switch as its entrance, the structure then has an inverter loop in its next stage. This loop, in its steady state, either has a '1' on the left side and a '0' on the right side, or conversely, is stable with a '0' on the left side and a '1' on the right side.

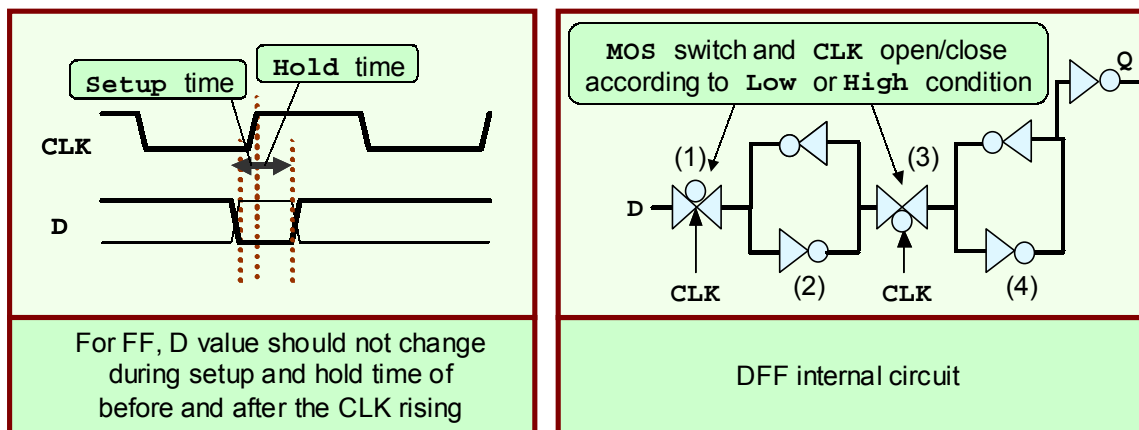


Figure 1-15 setup/hold time

Figure 1-16 DFF Internal circuit

Let us assume that the left side in the left-hand loop (2) is stabilized at '0' and the right side is stabilized at '1'. Given this, let us assume that the switch at (1) is opened. Actually, over the interval where the CLK *signal* is low, the left-hand switch is open. The inverter in the loop is only capable of driving a very low current. When a '1' signal arrives at the D input, the value that is looped by the inverter is simply inverted.

Next let us consider what happens when the CLK signal transitions from low to high (i.e., at the rising edge). (Figure 1-17) At this time, the left-hand switch (1) is closed and the right-hand switch (3) is opened. The left-hand loop (2) maintains the value that it had immediately before the switch was closed, and that value is copied to the right-hand loop (4). Because of this, the output of the FF will vary according to the value of the D input as it existed immediately before the rising edge of the CLK *signal*.

Conversely, next let us consider what happens when the CLK signal goes from high to low (i.e., the falling edge). At this time, the right-hand switch (3) closes and it continues to maintain its previous value. Consequently, the output from the FF does not change. The left-hand switch (1) opens, and, in order to prepare for the next rising edge of the CLK *signal*, the D input value is continuously copied into the left-hand loop (2).

In an actual LSI, the FFs function as described above. Cells termed "latches" (or "D latches") do not have the right-hand loop of the FF structure. Consequently, latches require less area than FFs.

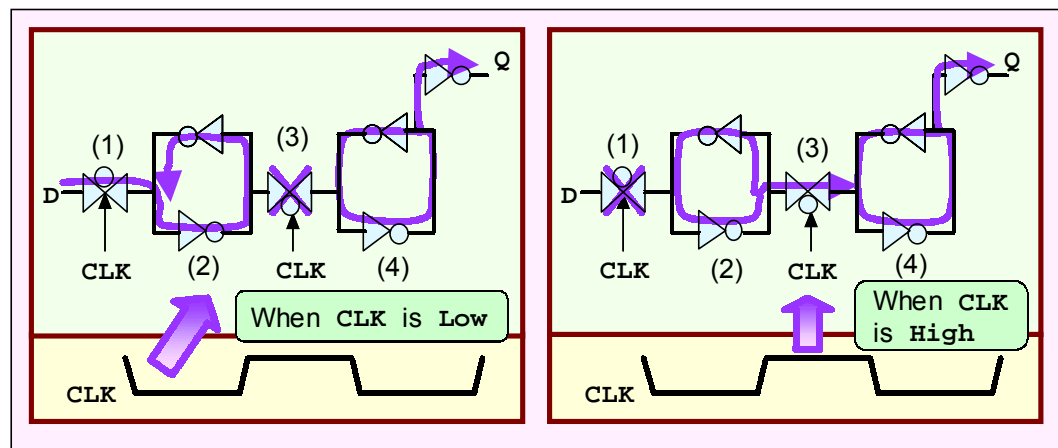


Figure 1-17 How DFF works

* The Meta stable State

If in this structure the value of the D input changes from '0' to '1' immediately prior to the CLK *signal* changing from low to high (the rising edge), then the left-hand MOS switch (1) closes as the D input is changing to '1' and thus the '1' value is sent to the left-hand loop (2) for only an instant. If the period of time over which this instantaneous '1' is present is shorter than the delay time of the loop (2), then the loop (2) will begin to oscillate. The oscillation of (2) is propagated directly to the right-hand loop (4), and the oscillating value is output from the FF. (Figure 1-18).

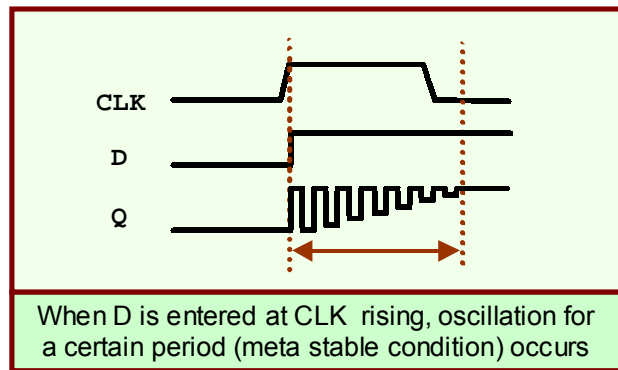


Figure 1-18 Meta stable

This oscillating state is propagated to the logic circuits, and thus the logic circuit may not function properly. This is the meta stable problem.

In the design of large logic circuits, it is not possible to perform adequate investigations for all of the circuits. When the meta stable problem is found in circuits that have already been fabricated, it is difficult to trace back the causes, presenting a major impediment to circuit debugging. There is a need for a simple, easily-to-understood approach to the meta stable problem.

* Approaches to the Meta stable Problem^{[1] [2]}

Figure 1-19 (a) shows an approach to solving the meta stable problem. Not only are FFs placed between the asynchronous clocks CLK1 and CLK2, but after transmission to CLK2, one more FF is placed, rather than placing logic in the interval.

The oscillation from the meta stable state does not continue perpetually. The delay times of the inverters in the FF loops have some variability, so the unstable oscillations converge to either a '1' or a '0' constant value. Consequently, as long as there is convergence in the meta stable state by the rising edge of the next CLK, there will be no influence on the circuitry at the right-hand FF or beyond.

There is no precise data on the time required for the meta stable state to reach convergence. Because it is difficult for LSI manufacturers to measure this time period, the times are not publicly disclosed. As a result, no specific times can be provided; however, these times are estimated to be about 10ns for a 0.18μm design rule, and 12ns for a 0.25μm design rule. If the operating frequency at the 0.18μm rule is more than 100 MHz, then the delay time that is allowable between FFs is less than 10ns and, from the perspective of safety, another stage of FFs may be required.

In logic circuits wherein the operating speed is high, the lower-stage FF accepts the data without the oscillations from the higher-stage FF converging. While one may think that if this is repeated, then the oscillation will not converge regardless of the number of stages of FF that are used, this is not the case. The meta stable state is a probabilistic phenomenon. When the oscillation from a previous stage arrives at the FF of the later stage, then even if we assume that the oscillation does not converge, the probability that the lower-stage FF will oscillate is low. Even if there is no convergence, the amount of oscillation will decline. Additionally, it does not always take the same amount of time for an oscillation to converge. While most oscillations converge, when there is no convergence, the probability that the oscillation will propagate is

extremely low. Consequently, if several stages of FFs are added, the probability of oscillation in the output of the FFs becomes asymptotically close to '0', so there is no problem in actual application. Regardless of how high the operating frequency, if some number of FF levels is used, then the oscillation is sure to converge.

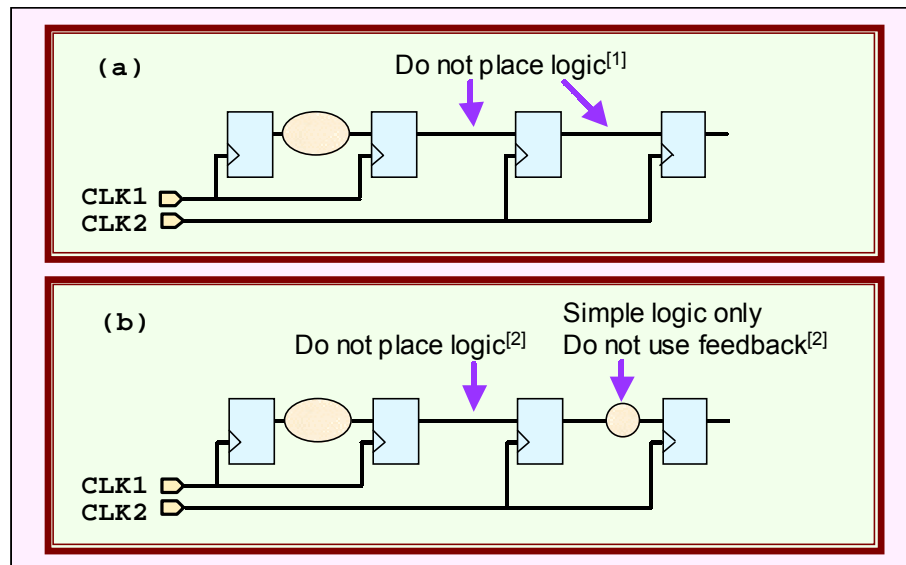


Figure 1-19 Meta stable measure counter measure

* After Asynchronous Transfer, Do Not Have a Feedback Loop in the First-Stage FF^[3]

If, conversely, the CLK1 and CLK2 operating frequencies are not high, then there is no problem if there is some degree of logic before the next-stage FF, such as shown in Figure 1-19 (b). However, even when this is the case, there will be problems when there is feedback to the FF in order to latch data. In FFs wherein there is feedback, the value that is latched by the FF may be ungrouped by the input of a meta stable state, depending on the circuit structure. As a result, the circuit will malfunction. Because of this, even if the placement of some amount of logic is unavoidable, be absolutely sure not to have feedback.

It becomes difficult to confirm whether or not a circuit is safe when a logic circuit is inserted after asynchronous transfer. Even if the designer has assembled the circuit while paying careful attention to the meta stable problem, if the checks by the other designers are difficult, then the result will be much extra work and expense. The meta stable problem is extremely difficult, and the checks are difficult as well. In order to eliminate this problem, it is best to be thoroughly entrenched in the habit of not placing logic prior to the next-level FF, even when the operating frequency is low.

* Take Meta stable counter measures in Outside Inputs of Noise-sensitive LSIs^[4]

The meta counter measures must be taken not just when data is exchanged between asynchronous clocks, but also when *signals* are input into the LSI from the outside. Inputs from elements that turn on and off asynchronously (such as switches) and *signals* that are supplied from locations that are physically distant on the circuit board are sensitive to noise, and this noise may cause a meta stable state, resulting in malfunctions. For this type of input *signal*, it is necessary to handle the meta stable state in the same way it was handled between asynchronous clocks. It is be-

1.5. Asynchronous circuits

coming more common to have power supply voltages of 3.3V or 2.5V even on the circuit boards, and recently circuit boards operating at about 1.9V have begun to appear. If the power supply voltage is low, then the lower the voltage, the more sensitive the device is to noise. Moreover, the finer the design rule for the LSI, the faster the I/O operating speed will be, making the LSI extremely sensitive to noise.

* The Approach to Sending Data Between Asynchronous Clocks^[5]

Data transmission that takes only the meta stable problem into account can do nothing but simply transmit and enable *signal* asynchronously. In cases such as when bus values are transmitted, considerably ingenuity is required. In Figure 1-20, the data is latched at CLK1, and this signal is passed to CLK2 using asynchronous transmission. In such a case, if the rising edges of CLK1 and CLK2 are close together and if the rising edge of CLK1 is applied during the setup time for CLK2, then the problem will not be one of a meta stable state, but rather the problem that will occur is one wherein the value will not be read correctly.

Because the data *signal* uses a bus (8 bits wide), the individual signals will arrive at minutely different times. When these signals are latched, some bits might accept the value after the value has changed, while some bits might accept the value before the value has changed. The result is the danger that, if, for example, the value "36" should be received, a totally different value may be read in.

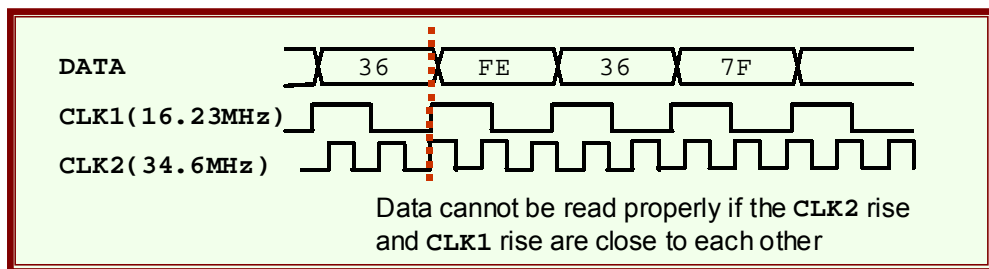


Figure 1-20 Unreadable data between asynchronous clocks

One method by which to safely read data from a bus of some width is the method where CLK1 is latched by CLK2. As is shown in Figure 1-22, the signal wherein CLK1 is latched by CLK2 is used as the select signal. When this is done, then if the CLK 2 clock period is less than 1/3 of the CLK1 clock period, then it is possible to receive the data signal in a safe place without being constrained by setup/hold time violations. However, at the point in time wherein CLK1 is latched by CLK2, there is the potential for the meta stable problem to occur, and thus it is necessary to have a 1-stage shift by CLK2, such as shown in Figure 1-22. This method does not substantially increase the size of the circuit, and the circuit can be created easily; it is really quite simple. See "1.5.2. The Use of Memory in Transmission Between Asynchronous Clocks With the Same Period".

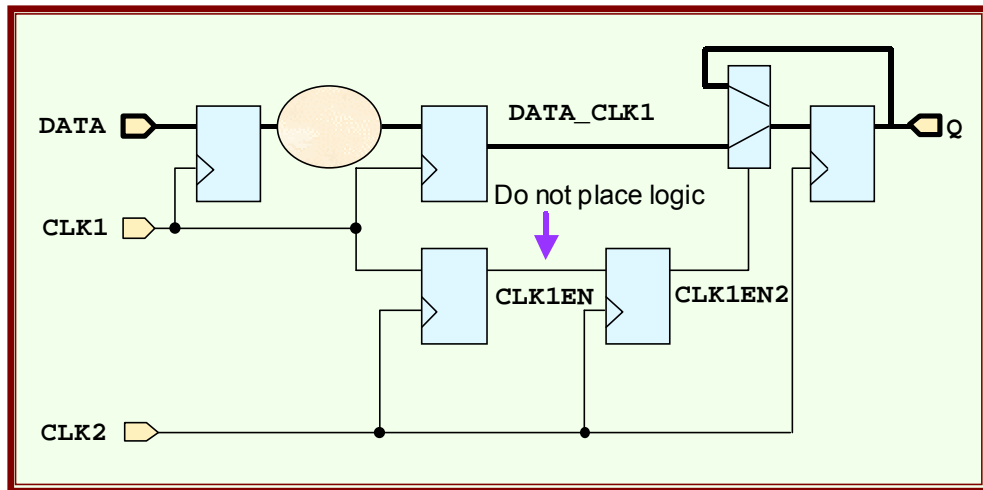


Figure 1-21 Simple data transfer between asynchronous clocks

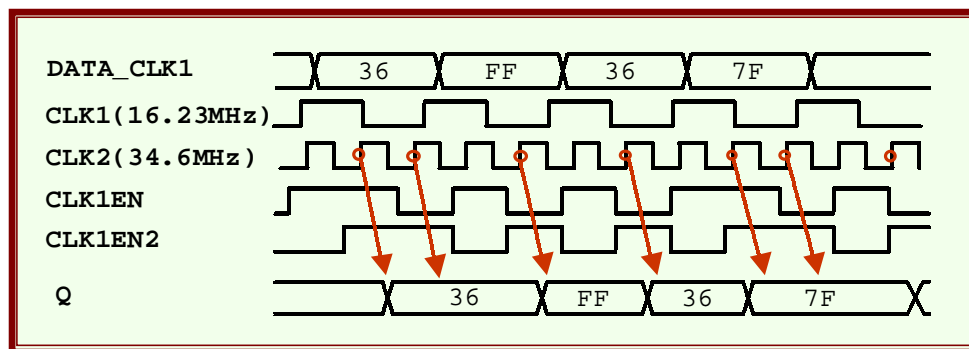


Figure 1-22 Timing diagram

RTqualify will cover the analysis between different clocks and check against meta stable in its next version.

1.5.2. Use memory in transfers between asynchronous same-period clocks

- [1] Use FIFO for transfers between asynchronous clocks with the same clock period

- Clocks have the same clock period

reference

- [2] Use frame memory for transfers in asynchronous with different periods

reference

Explanation

Use FIFOs for data transfers between asynchronous clocks with the same clock period. Data are input to the FIFO according to the address of the input side free run counter. At the output side of FIFO, use a different free run counter that indicates a different address value than the address value at the free run counter at the input side to read out data.^[1]

If there are six FIFO stages, shifting the three-stage counter value will avoid the same address from being simultaneously accessed. In this case, neither the input side nor the output side clocks will function erroneously even if a shift of two cycles or less occurs. However, a reference enable signal is required for data exchanges during the asynchronous period that used this FIFO.

Even if this enable *signal* is not present, if there is a pause in the data being input, there is also a method that uses this pause interval to forcibly retrieve an enable *signal*. If no enable *signal* exists and there is no pause in the data, there is no way to safely transfer the correct data. Using an initial reset as the start *signal* for a free run counter in a design can be hazardous.

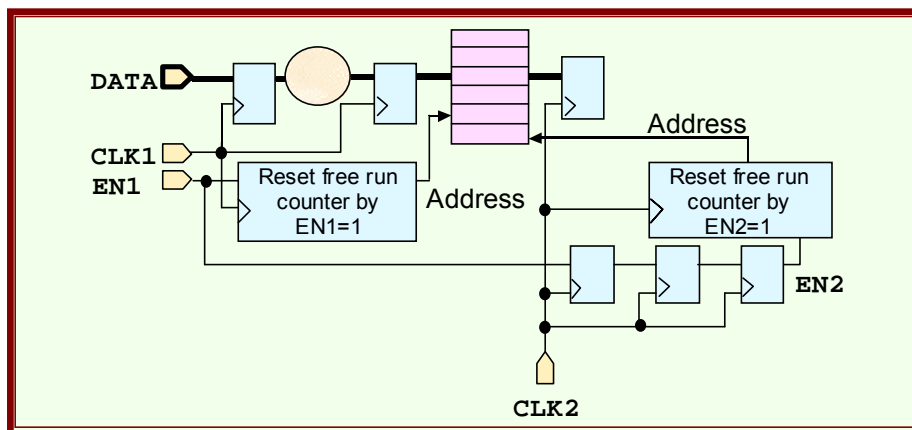


Figure 1-23 Transfer between asynchronous clocks with the same clock period

Large frame memory (dual port RAMs) is required for transfers between clocks that are asynchronous but which do not have the same clock period. Similar to data transfer during an asynchronous period using FIFO, the reference enable *signal* determines the input side and output side address values. If data are output from the input side consecutively and the duration of each frame is long, then there must be sufficient memory for that frame. Therefore, it is necessary to carefully examine the architecture to make sure that it does not grow too large.^[2]

The check for this item is not available in RTqualify.

1.5.3. Guaranteeing the setup/hold margin for synchronous RAM

[1] Guarantee the setup/hold margin	mandatory
[2] Synchronous RAM has a long hold time, so some measures are necessary	recommend 3
[3] Margin can be guaranteed by using a latch or by using buffer, inverter or delay cells	reference
[4] Allocate RAM to the top level if possible	reference
[5] There is also a method for avoiding modification of I/O for each ASIC vendor by creating a general-purpose RAM module and fixing the I/O	reference

Explanation

FFs operating by clock edge have setup time and hold time. (Refer to “1.5.1 Consider meta stable in signals between asynchronous clocks”. Besides FFs, cells operating by clock edge have setup time and hold time. Designers should follow this rule when coding RTL to generate logic circuit by logic synthesis tool.

The RAM that is used inside LSIs and FPGAs is mostly synchronous RAM. Because synchronous RAM operates on the rising edge of the clock just as other circuits do, it can be used following the same approaches as for other logic circuits. However, synchronous RAM has an extremely long hold time, and thus one must consider ways to insure the hold time.^[2]

Hold time violations that occur in synchronous RAM are problems that can be solved by the execution of commands to insure the hold times using the logic synthesis tools. However, in some cases the hold time for synchronous RAM may exceed 1ns, and in some cases a large number of buffers will be inserted in order to insure the hold time (See Figure 1-24). At 0.18um, 0.13um, or the like, the delay for one stage worth of buffer is only about 10 or 20ps. If we assume that the RAM hold assurance time is 1ns, then 50 or more buffers would be added. If you do not mind it if the circuit gets large, then it is a simple matter to use the logic synthesis tools to ensure the hold time, even though many buffers will be inserted.

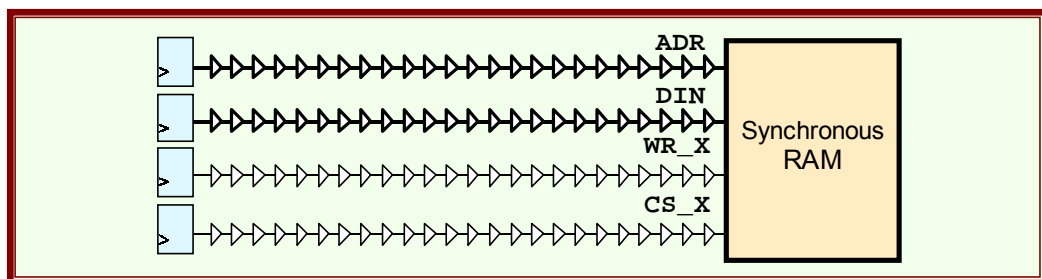
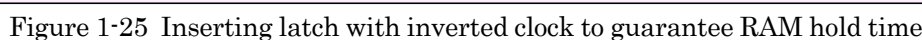


Figure 1-24 Large amount of buffers inserted to guarantee hold time of synchronous RAM

If an increase in the size of the circuit is considered to be a problem, there is also the method where a latch on the inverted clock is used. As is shown in Figure 1-25, when a latch on the inverted clock is used before the synchronous RAM, there will be no changes to the value of this *signal* when the clock is high. If the hold time value of the synchronous

RAM is less than half of the clock period, then this will solve any problems with hold time. Latches that use the inverted clock have open gates during the interval over which the clock is low. Even if we assume that there is a large number of logic gates prior to the latch, the *signal* passes straight through the latch, and so the delay on this path can get by with only taking a latch delay from one cycle time. In other words, by inserting the latch, it is possible to solve the hold time assurance problem without being particularly aware of the delay time before the *signal* arrives at the synchronous RAM.



A BIST circuit is inserted into the RAM for automatic testing. (Figure 1-26) Additionally when laying out the LSI, the positioning of the RAM should be decided first. If the RAM is positioned in the topmost level instead of in a deep position in the local levels, these operations are simplified.^[4] However, in levels that use RTL description, it may require a large amount of work to move the RAM to the topmost level. For example, for an IP for which the design has already been completed, the design management would not want to move to the top level those parts of the IP wherein RAM is used. Thus this should be interpreted as “layout the RAM in as high a level as possible”.

When RAM is used within the IP, the RAM cell names, the I/O pin names, and the specifications will vary depending on the ASIC library used. If possible, it is convenient to prepare a standard RAM interface and reuse the RAM by adopting a method where the RAM is called up from the ASIC library.^[5]



Designing with memories is explained in RMM:5.7.

1.6. Hierarchical design

1.6.1. Consider limitations based on hierarchical scale

- | | | |
|-----|--|-------------|
| [1] | Limit the gate size of a single level to 10,000 gates or fewer for safety's sake
- Keep the size of a single level as small as possible as the operating frequency grows higher
- Limit the gate size of a level to 20,000 gates or fewer even when the operating frequency is low (mandatory) | recommend 2 |
| [2] | Consider hierarchies on a scale of 2,000~10,000 gates to be the standard, and do not place logic gates in the upper levels (except for CTS buffers and I/O cells) | recommend 1 |
| [3] | Lower-levels' size of the basic blocks (on a 2,000~10,000 gate scale) are optional | reference |
| [4] | Assign clock generation <i>modules</i> , reset generation <i>modules</i> , RAM, I/O cells, Setup/Hold guarantee buffers, and the top level of RTL description only to the top level | mandatory |

Explanation

When an ASIC design is made, these are segmented into each block by taking into account the functionality and the distribution to each designer. When divided into four blocks (geometo, lendar, cpuif, video), as illustrated in Figure 1-27, if the geometo block is to a certain extent rather large, then it is split up further into blocks such as gtxc, gclip, and gmap. In this case, it will end up having hierarchical structures such as TOP (top level), geometo (upper levels), gtxc (lower levels).

These hierarchical structures have two important meanings in ASIC design. One is the segmentation of a hierarchical structure that is easy to verify or easy to reuse. The other is the allocation of levels that are easy to handle using logic synthesis and layout tools.

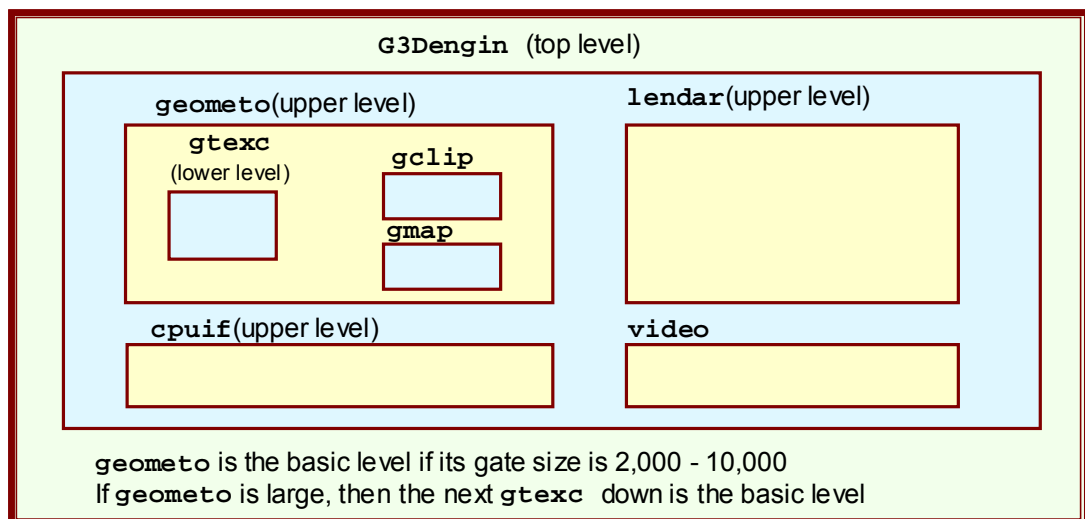


Figure 1- 27 Hierarchical structure of 3D chip

When considering the hierarchy in terms of logic synthesis and layout, it is better to have a clear conception of what the basic block (basic level of hierarchical design) will be. The

1.6. Hierarchical design

basic block is constrained by size and structure. The size of the each basic block should fundamentally be between 2,000 and 10,000 gates.^[1] If the size of the four TOP (upper-level levels) blocks (geometo, lendar, cpuif, video) is between 2,000 and 10,000 gates, as illustrated in Figure 1-27, then these four blocks will constitute the basic blocks.

If any of the geometo, lendar, cpuif or video blocks is larger than 20,000 gates, then the next lower levels (gtexc, gclip, gmap) would constitute the basic blocks.

- * Logic synthesis is usually performed from these basic blocks.

- * The hierarchical structure of the basic blocks is maintained by logic synthesis and is passed on to the layout tool.

- * Levels below the basic blocks are referred to as sub-blocks.

Sub-blocks will frequently ungroup hierarchies when used with a logic synthesis tool, so levels are not taken into account when transferred over to the layout tool.

With the exception of I/O buffers, CTS buffers, PLLs, RAM, and ROM, no other logic gates may be placed in the top level (TOP) beside these four basic blocks.^[3] If a description is executed in the top level that could somehow generate logic gates, it would be difficult for logic synthesis tools to optimize those gates.^[4]

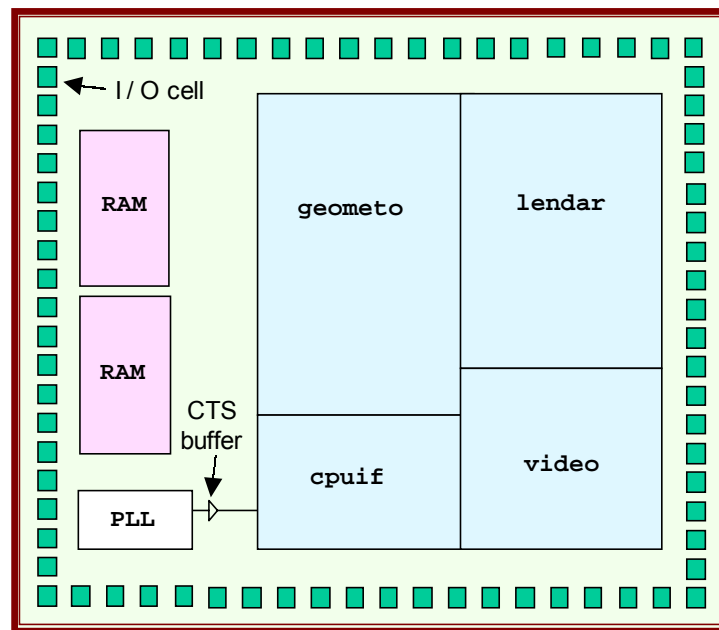


Figure 1-28 Top level diagram

Logic synthesis is performed at the basic block unit. If there is an RTL description in the top level that has logic, it may be difficult to synthesize. It will be problematic to increase operating speed even if logic synthesis using hierarchical compile was performed. If there are logic gates that do not belong to any level, it will be difficult to place cells in their proper locations during the layout phase, the circuit speed will slow down, and wiring efficiency will decrease.

Logic synthesis performance will not be adversely affected if the basic blocks have up to about 20,000 gates, thanks in part to recent improvements in the performance of logic synthesis tools. The faster the operating speed required by circuit be, the smaller the

basic block should be made; otherwise, required performance cannot be met. Using a smaller basic block is also advantageous to layout tools for faster operating speed. Therefore, you may wish to keep in mind that the faster the operating speed becomes, the smaller the basic blocks should be.

It is not easy to make broad generalizations, but the relationship between the operating speed and size of the basic blocks typically will depend on which ASIC vendor technology is being employed or what type of circuit structure is being employed. Therefore, on a scale of 0.35um, caution is urged for frequencies above about 50MHz, and unless careful consideration is given to the size and structure of these basic blocks, problems could occur during logic synthesis or layout when the frequency exceeds 100MHz. It should be considered vital that the basic block size not surpass 10,000 gates when exceeding 50MHz at 0.35um or 5,000 gates when exceeding 100MHz.

The limitation of basic block size is one of constraints to be observed. However, the circuit structure constraint noted in “1.6.2. Make basic block FF output & combinational circuit input” is even more important.

Size constraints will be meaningless if the circuit structure constraint is not adhered to.

RTqualify warns that

1611(W2) There are <number_of_line> lines that generate logic.

Comment, declaration statement and blank line are not counted for the number of line judgment.

1612(W1) No combinational circuits or F/Fs should exist on top layer or layers based thereon.

Chip level is explained in RMM:5.6.9.

1.6.2. Make basic block FF output & combinational circuit input

- | | |
|--|-------------|
| [1] Make all basic blocks combinational circuit input and FF output | recommend 3 |
| [2] When the above is impossible, have the timing path cover no more than two blocks | recommend 1 |
| [3] The above restrictions do not apply to smaller levels below the basic blocks (2,000 – 10,000 gates in scale) | reference |

Explanation

Output of basic block should be FF output whenever possible.^[1] By adhering to this rule, you can gain advantages during the synthesis phase such as:

- *Drive capacity and output arrival time from the outputs of the basic blocks will be clearer.
- *Input delay attributes (set_input_delay) provided during block synthesis and the values for input drive capacity (set_driving_cell) are given with greater consistency.

As a result, the quality of an optimized circuit can be enhanced.

Also, the timing path can be kept within a single block when such a circuit structure is employed. As a result, it is possible to prevent excessive loss of speed during layout since the wiring length is restricted. Consider this to be a mandatory circuit structure for designs that are tight in terms of speed.

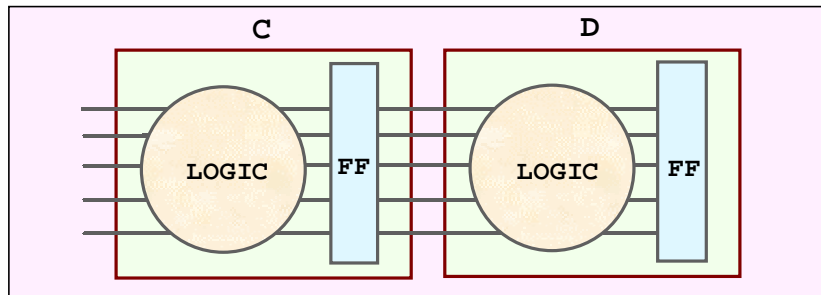


Figure 1-29 Basic block structure

Nevertheless, it is difficult in the actual design to make all basic blocks combinational circuit input and FF output. In certain situations, FF input combinational circuit output will result. Even in these cases, however, you should observe the restrictions in “Figure 1-30 Make paths in two levels if possible”.^[2] This is because it will become very difficult to achieve the speed improvement on this path in the layout if there is a path that passes through the basic block as combinational logic.

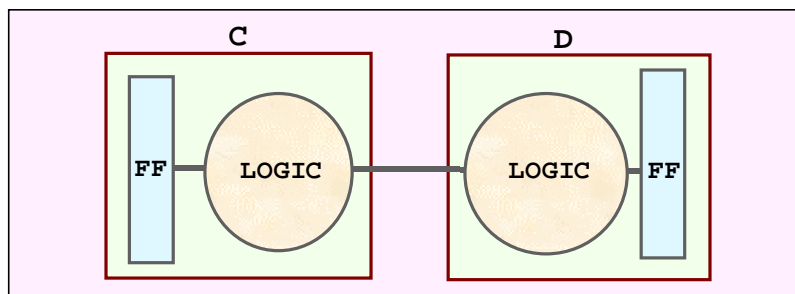


Figure 1-30 Make paths two-modules paths if possible

RTqualify outputs the warning message,
when the number of line is more than the last figure set by the variable of
basic_block_line_number(default=1000),

1621(W3) Signal "<name>" outputted from this module as a combinational circuit.

If passing as combinational circuit at a module, which is larger than the forward figure set by
the variable of basic_block_line_number(default=500), the following message is output.

1622(W1) There is a path that passes through a combinational circuit.

Register output is explained in RMM:5.6.1.

1.6.3. Follow sub-block (the levels below basic clocks) constraints

- | | | |
|-----|---|---------------|
| [1] | Limit paths that are critical in terms of speed to within two sub-blocks inside each basic block, if possible | reference |
| [2] | Stay within three sub-blocks whenever possible if there are any problems in terms of speed | recommended 3 |

Explanation

Item “1.6.2. Make basic block FF output & combinational circuit input” is not applicable to sub-blocks (levels below the basic blocks). However, it is not desirable to span a large number of sub-blocks for designs that are tight in terms of speed.. It is advisable to avoid as much as possible the use of designs in which timing paths span multiple blocks.^[1] Try to keep the timing path to within no more than three sub-blocks, especially if there are any speed problems.^[2]

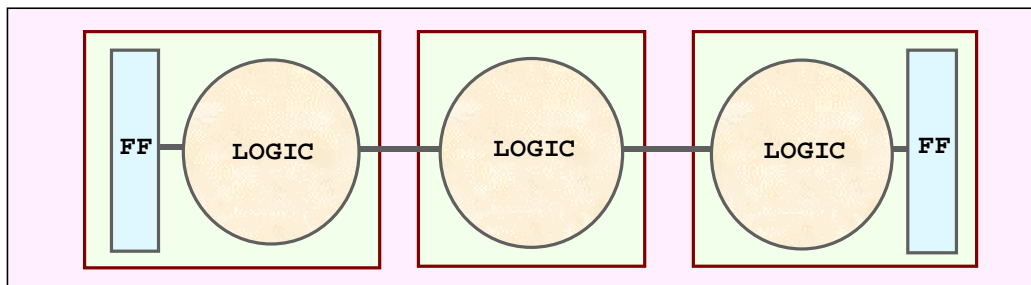


Figure 1-31 Keep paths within three levels even inside subblock

Optimizing combinational circuits that span across levels presents the following problems:

- * Timing optimization goals are difficult to distribute

If the timing target between FF in Figure 1-31 had been set to 10ns or less, for example, the timing goal must be distributed so that the total of each combinational circuit is 10ns or less. It is difficult to distribute a precise timing that will optimize each one to the same degree while keeping the structure of each combinational circuit in mind.

- * Optimization is restricted by levels

Logic synthesis tools retain the block I/O ports so long as the level is not destroyed. When optimizing the logic before and after a level, even if flattening and structuring makes it possible to improve the timing and area of a circuit, the progress of optimization process is limited by the level in order to preserve the I/O ports.

- * Timing paths are difficult to trace

It is not easy to modify the RTL description in order to improve the timing path when the blocks in each level of hierarchy are designed by different designers. When the timing path is kept within a single level, then it is relatively easy to trace and improve the timing path.

* Floor plan considerations are required

When executing the floor plan, it is necessary to construct the physical levels so that the timing path is kept within a single cluster. When physical levels are segmented by combinational circuits, there are cases in which the wiring will become longer and the timing will deteriorate.

When the operating speed is somewhat fast (100Mz at 0.13um), the above restrictions must be strictly adhered to.

RTqualify changes variable and uses 1611 and 1612 to check the inside of sub block.

Configuring combinational logic inside a single module is explained in RMM:5.6.2.

1.6.4. Do not insert gates into upper levels of basic blocks

- | | | |
|-----|--|-------------|
| [1] | The upper levels of basic blocks should contain only the connections of each block | mandatory |
| [2] | Make paths with severe speed constraints similar, even in the sub-blocks | recommend 3 |
| [3] | If the scale of a basic block is about 10,000 gates, the number of I/O ports should be specified to no more than 200 | recommend 2 |

Explanation

Do not place logic (except for I/O cells, CTS buffers) in the upper levels of the basic blocks.^[1] This is because a timing path that spans across three blocks, as illustrated in Figure 1-32, will result, even if only one AND gate is placed there. This would be in violation of the constraint in “1.6.2 Make basic block FF output & combinational circuit input”, which states that “the timing paths of basic blocks must span across no more than two blocks”. If such a path exists, executing a floor plan in the layout will become more difficult and a very long timing path may result. There may also be *instances* in which logic cells that do not exist in any basic block cannot be placed in the appropriate positions in the layout. This will cause a floor plan to become problematic and layouts that take speed into account to become impossible.

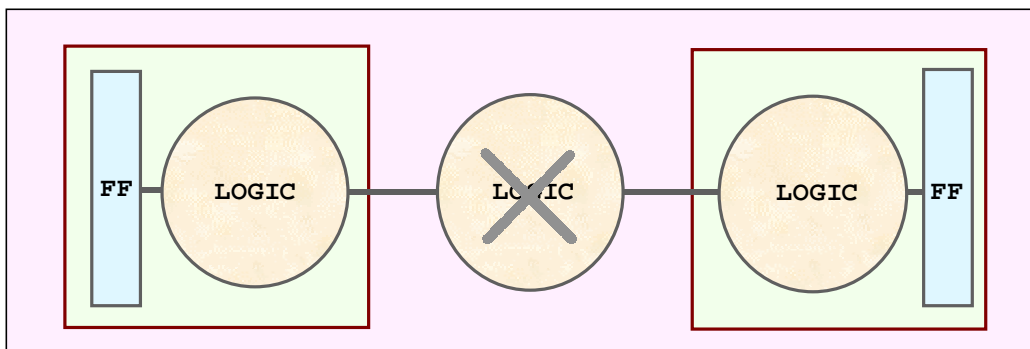


Figure 1-32 Inserting combinational circuit into an upper level

When a timing path passes across multiple blocks, it becomes difficult to distribute the optimum timing constraint necessary for properly optimizing the timing path. Also, when synthesizing upper levels in which multiple basic blocks exist (50,000 – 200,000 gates), synthesis will take a long time to run and it will become difficult to use synthesis methods that improve the operating speed. Thus, combinational circuits that exist in the core do not belong to any basic block, so it will not be possible to use a method that executes synthesis in basic block units.

Selectors for binding busses are sometimes placed in the upper levels of designs in which busses exist. Also, selectors used for sharing I/O pins with test I/O are placed in the top level (Figure 1-33). You should consider it a prerequisite to at least hierarchize such logic.

If hierarchized, it will become possible to implement synthesis in smaller units on this block (level of hierarchical design). Also during layout, it becomes possible to place these

in each level of hierarchy and to implement an efficient layout. Thus, by placing small selectors as levels in a level higher than the basic blocks, paths that span across two or three blocks may exist. Bus signals in particular have a certain degree of width at 16 bits and 32 bits, so the wiring area will expand.

If possible, try to include such selectors inside some of the blocks in your design. However, even if a selector to bundle the bus is included inside the ctrl block, the output from alu will pass through the ctrl block and therefore is in violation of the fact that “the timing path span should be no more than two blocks” stipulated in “1.6.2. Make basic block FF output & combinational circuit input”. Naturally, an unnecessarily long path such as alu->ctrl->ram ends up being created in the layout.

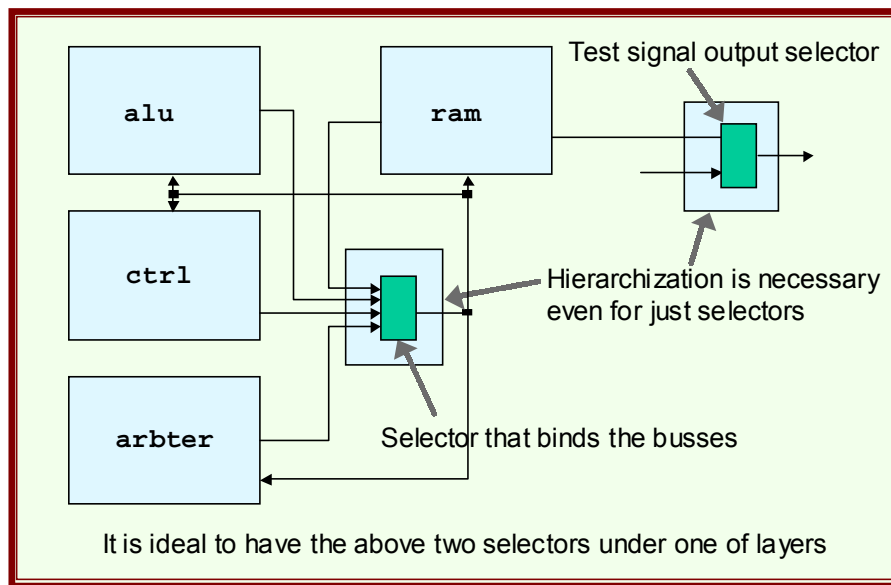


Figure 1-33 Inserting selectors in an upper level

If possible, it is best not to use a bus structure but rather to exchange *signals* directly, such as from alu to arbter or from ram to ctrl. It may appear to be disadvantageous since the number of ports for each *module* will increase, but this method offers a number of advantages in current layout tools.

However, since the number of I/O ports for each block increases in a design where *signals* are supplied from alu to all the other blocks, and from each of ctrl, ram and arbter to all the other blocks, there is no other alternative but to choose a bus structure. Whether it is necessary to assume a bus structure or not is determined at the time of system design. At that point, you should try to consider how you could avoid using a bus structure.

When the number of ports for each basic block increases, the number of paths, which pass through breaks in the levels during layout, increases and layout (floor plan) for deciding the basic block assignment may become difficult. To increase the degree of freedom at the time of layout, you will want to carefully consider the number of I/O ports of each basic block. 200 or fewer number of I/O ports is preferable, if the number of basic blocks is about 10,000.

RTqualify uses 1611 and 1612 to check 1641 and 1642.

1643(W1) Sum of number of input ports and number of output ports more than <n=200>.

Elimination of glue logic at the top level is explained in RMM:5.6.8.

1.6.5. Separate the data path section from the controller

- [1] Description styles are different for the data path section and controller
- [2] Different synthesis methods can be chosen for the data path section and the controller

reference

reference

Explanation

The data path section utilizes special algorithms tailored to the application for processing *signals* that correspond to conditions given to control *signals* or data. The controller determines conditions for extracting the various control *signals* included in external protocols and for the control *signal* obtained from the data.

The description for the data path section consists primarily of FFs and operators. The description for the controller mainly consists of control syntax including encoders/decoders such as *if statements* or *case statements*, and state machine descriptions, such as those described in “2.11. State machine descriptions”.^[1]

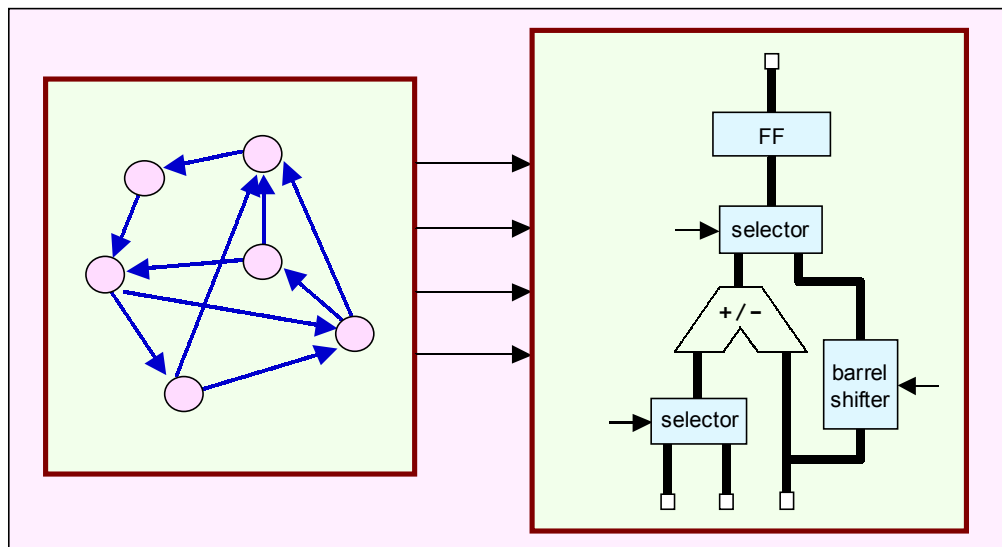


Figure 1-34 Separation of data path part and control part

Since descriptions for the data path block mainly consist of operators, most logic synthesis tools perform processes such as allocating resources corresponding to the operators, sharing resources, and optimizing operational expressions. See “5.6. Circuit synthesis including operators” for more information.

Because control blocks mainly consist of state machine descriptions, logic operational expressions, and FF descriptions, most logic synthesis tools perform logic optimization processes such as structuring, flattening, or optimizing the state machine. See “2.11. State machine descriptions” for more information.^[2]

Separating the data path part from the control part is said to be useful in establishing policies for RTL description and for logic synthesis. However, the paths that cross data paths from the control part often are critical in terms of timing, and thus the control part and the data path part must have no hierarchical relationships to each other.

1.6. Hierarchical design

Ideally, it would probably be best for the data path part and the control part to exist as sub levels of a basic block. Doing so will cause the paths from the control parts to the data path parts to terminate within the same basic block. However, it is difficult to provide both levels in the basic block when the size of the circuits in the data path part is large. Additionally, the relationship between control parts and data path parts is often not a 1-to-1 relationship, but rather multiple control parts are related to multiple data path parts.

It is not easy to create the ideal hierarchy structure. It is better to think in terms of the basic blocks, even more so than separating the data path parts from the control parts.

The rules described in this section should be considered an additional policy to the rules described in “1.6.2. Make basic block FF output & combinational circuit input” and to be implemented when possible.

The check for this item is not available in RTqualify.

1.6.6. Designate buffer outputs in upper levels with 200,000 or more gates

- | | | |
|-----|--|-------------|
| [1] | ASIC I/O cells should be inserted only in the top level or the I/O cell level | recommend 3 |
| [2] | In large-scale ASICs, it is not possible to synthesize everything from the top level | reference |
| [3] | When a high drive capacity buffer is required for the output of a level with 200,000 to 800,000 gates, create a separate level containing only buffers | reference |

Explanation

When performing LSI design, the designer must notate the I/O cells using RTL description. While there is a method wherein the I/O cells are inserted by the logic synthesis tools, in the design of large scale LSIs, there is a large number of types of I/O cells (in comparison to FPGA), and because the I/O cells will be changed one after another, this approach is not particularly practical. It is necessary to check what type of I/O cells is being used.

I/O cells should either be notated as being laid out directly in the highest level, or should be notated within the block of I/O cells that are laid out in the highest level.^[1]

The recent increases in speed with which the logic synthesis tools run have made it possible to perform logic synthesis on a relatively large scale. Even logic synthesis of circuits containing as many as 500,000 gates and beginning at the uppermost level can take as little as four to five hours, or, if on a slow system, the logic synthesis can be completed within two days. However, when one takes into account the execution time and performance, 500,000 to 800,000 gates is probably about the limit for performing logic synthesis. In designs in the two to three million-gate range, only timing analysis can be done at the highest level.^[2] In large designs in the two million to three million gate range, the creation of the higher-level levels for the basic blocks is extremely important. Is not particularly desirable for a circuit design to have some levels with 800,000 gates and other levels with 2000 to 3000 ,gates so that there are 200 or 300 levels at the very top.

There are some cases wherein it is difficult to completely fulfill the operating speed requirements using the logic synthesis tool within only the basic blocks. Even if the logic synthesis started with the basic blocks, fine-tuning the timing using circuits in the scope of 200,000 to 800,000 gates is a wise policy. The top level in the design of large LSI circuits should have levels with no more than 200,000 to 800,000 gates.

In the design of large-scale LSIs, it is best to only have levels with 200,000 to 800,000 gates in the top level. Special caution is required regarding the inputs and outputs of the 200,000 to 800,000 gate blocks because logic synthesis using hierarchical compile to fine-tune the final circuit is not performed in the top level. The interfaces between these blocks require, of course, long interconnects in the layout. When the interconnect lines are long, it is necessary to provide buffers with strong drive capabilities.

Buffers that have strong drive capabilities can be produced by the logic synthesis tool, a delicate balance between the added capacitance and delay values in the external ports is required, and it is difficult to derive the drive capability that would be hoped for. If one

1.6. Hierarchical design

wishes to place a buffer that has a strong drive capability, then it is probably best to specify the buffer cell one's self.

However, when it comes to this buffer, problems may arise in the form that is called directly from the RTL description. If the cell is frozen through the use of the “set_dont_touch” command in logic synthesis for the buffer that has a powerful drive capability called by RTL description, then this cell itself will not be deleted. However, the output net for this cell may be cut off and the output of the cell may pass through an indirect route.

To avoid such circumstances, prepare the levels for buffers with high drive capabilities such as shown in the Figure 1-35, and use a method for calling those levels.^[3] When this is the case, in so far as “set_dont_touch” is specified for the level, then the output network will never be cut off in this way.

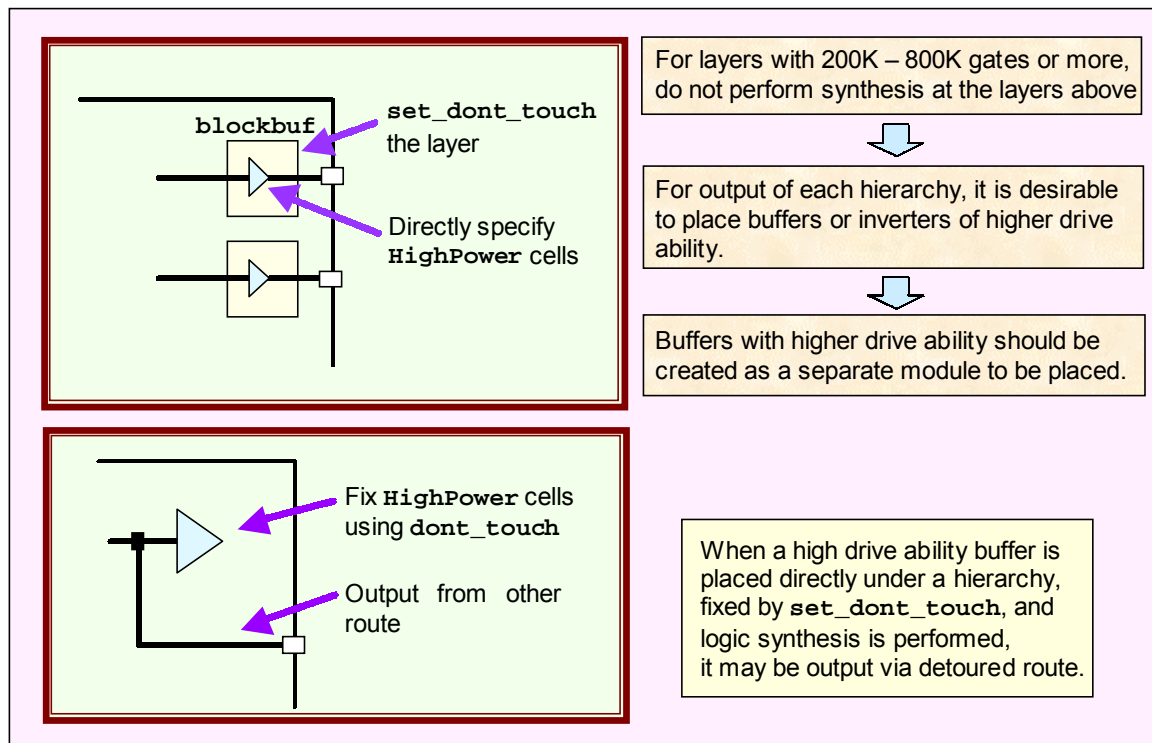


Figure 1-35 Hierarchization and placement of buffers

Adding buffers in output port means not only securing the usage of high drive capacity cell but also closing timing analysis within the level. If the output of cell, which is the last level connected to output port, is used not only for output port but also for internal signal, the delay value inside the level changes depending on additional capacity to be connected to output port. Therefore, buffer should be inserted just before output port to facilitate timing analysis of a large scale design.

With the latest version of logic synthesis tool, the buffer just before output port can be inserted by the command of tool. Please refer to "5.7.4.[13] set_isolate_ports" for details. Buffer insertion just before output port is not necessary if logic synthesis tool generates particular buffers.

The check for this item is not available in RTqualify.

Chapter 2 RTL Description Techniques

This chapter introduces the basic style of RTL description using Verilog-HDL, as well as items to take note of when coding and some example codes. Performance of the optimized circuits and the readability of the source code can be improved by following the description style and cautionary items introduced in this chapter.

Contents

- 2.1 Combinational logic
- 2.2 *always construct* description of combinational logic
- 2.3 FF inference
- 2.4 Latch inference
- 2.5 Tri-state buffer
- 2.6 *always construct* description that takes circuit structure into account
- 2.7 *if statements*
- 2.8 *case statements*
- 2.9 *for statements*
- 2.10 Operator description
- 2.11 State machine description

2.1. Combinational logic

2.1.1. Use the *always construct* and *function statements* correctly (Verilog only)

- [1] For the effective reuse of design resources, standardize the use of *always construct* or *function statement+assign statement* (Verilog only)
- [2] Describe every case of conditions in a *function statement*
- [3] Use the syntax checker tool to avoid mistakes
- [4] For *always construct*, add a comment for each *reg* variable (Verilog only)
 - Make clear whether they are for combinational circuits or sequential circuits

reference

mandatory

reference

reference

Example Code

2 to 4 decoder by function

```

module DEC2TO4 ( AIN, EN, OUT )
input  [1:0] AIN;
input    EN;
output [3:0] OUT;

function [2:0] DEC;
input [1:0] AIN;
input  EN;
begin
  if ( EN )
    case ( AIN )
      2'h0: DEC = 4'b0001;
      2'h1: DEC = 4'b0010;
      2'h2: DEC = 4'b0100;
      2'h3: DEC = 4'b1000;
      default: DEC = 4'bxxxx;
    endcase
  else
    DEC = 4'b0000;
  end
endfunction
assign OUT = DEC ( AIN, EN );
endmodule

```

Beware of errors in the return value bit width: it must be [3:0]

Can be used even if there is no argument EN declared

2 to 4 decoder by *always construct*

```

module DEC2TO4 ( AIN, EN, DEC )
input  [1:0] AIN;
input    EN;
output [3:0] DEC;

reg for decode circuit(combinational circuit)
[3:0] DEC;

always @ ( AIN or EN )
  if ( EN )
    case ( AIN )
      2'h0: DEC = 4'b0001;
      2'h1: DEC = 4'b0010;
      2'h2: DEC = 4'b0100;
      2'h3: DEC = 4'b1000;
      default: DEC = 4'bxxxx;
    endcase
  else
    DEC = 4'b0000;
  end
endmodule

```

Beware of lacks in sensitivity list

Latches are generated unless all conditions are covered

Example 2-1 Decoder description using *function statement* and *always construct*

Explanation

Generally speaking, there are two description styles for combinational circuits. One is a method of describing using *function* and *assign statements*, while the other uses *always constructs*.^[1] A *function statement* defines the functions (sub-programs) that can return only one return value. By calling these, they are implemented as a combinational circuit. In the case of *function* in the Example 2-1, calls and passing arguments are done by the *assign statement* and the results calculated with a *function statement* are automatically ascribed to an output signal.

In the case of *always constructs*, *sequential process statements* with the same time stamp after the *always construct* are executed when signals defined in the sensitivity list change. Execution will return to the beginning when the end of the *always construct* is reached and will repeat.

	<i>function statement</i> + <i>assign statement</i>	<i>always construct</i>
Merits	<ul style="list-style-type: none"> • Does not generate latch • Distinction between combinational and sequential circuits is clear 	<ul style="list-style-type: none"> • Simulation is slightly faster • Not much description required (no extra statements needed)
Demerits	<ul style="list-style-type: none"> • May cause erroneous bit width of arguments or return values • No syntax error results even if arguments are not declared • Only one output (Multiple output possible by Bundling them) 	<ul style="list-style-type: none"> • May generate latch • Lacks in sensitivity list will cause operation mismatch between the RTL and the gate level after optimization • Circuit gets large if multiple outputs are described together

Table 2-1 Merits/Demerits of using *function statement* and *always constructs*

Since *function statements* cannot be used in sequential circuits, it is possible to clearly indicate that a circuit is a combinational circuit. Also, there is no hazard of generating latches as with *always constructs*. However, since settings of argument and return value are necessary, the description amount grows little that it will be troublesome to use. Even if not arguments are declared in the *function statement*, there may be difficult situations in which there are no syntax errors and a problem cannot be discovered as a bug if *signals* in a module are used directly.

In the case of *function statements*, latches will not be generated even if not all of the conditions are described internally. In this case, however, it behaves like latch during simulation, but logic gates generated by logic synthesis tool have the result of don't care, so that it is uncertain whether the value takes 0 or 1. As a result this description can be very hazardous since the RTL simulation and gate simulation results will no longer match.^[2]

always constructs are easier to describe than *function statement*, but there is no essential difference between latch inferences and combinational descriptions. Therefore There is a greater likelihood of latches being generated by coding mistake. Also, unless all input *signals* in the *always construct* are defined in the sensitivity list, there is a danger of simulation mismatches occurring between the RTL and post-optimization gate circuits.^[4]

The issues of *function statement* bit widths and of *always construct* sensitivity list are simple but easy to be misunderstood. In a design by RTL description, there are also other problems, which the RTL simulation cannot detect. The design can be secured by gate level simulation. However, recently it has become difficult to perform gate level simulation for every test pattern as design scale has become large. Therefore, coding the unquestionable RTL and checking design by RTL simulations alone are essential To eliminate the problem of having results that differ in the RTL simulation and gate level simulation, please follow the style guide rules to design and then finally check with the RTL check tool so that the RTL description is without any problems.^[3]

If the results differ in the RTL description and gate level simulation, please refer to the following items;

- “1.3.1. Use asynchronous reset for initial reset”
- “1.4.3. Gated clocks should be used with special care”
- “2.2.2. Define every input *signal* in the *always construct* in the sensitivity list”
- “2.2.3. Initial value description in *always constructs*”
- “2.3.1. Unify the description style of the FF inference”
- “2.8.5. Description relying on *parallel_case* is prohibited”
- “4.1.3. Take note of input signal timing”
- “4.1.8. Description where the results do not differ due to the simulators”

In RTqualify,

each check for items of 2.1.1, 2.1.2 and 2.1.3 for *function statements* and *task statements* is performed more into details than the items written in Style guide.

2111 (N)	Combinational circuit descriptions mix <i>always</i> and <i>function</i> . (not to be checked by default setting)
2112a(W1)	No assignment is made to return value in a <i>function declaration</i> . When there is <i>function statement</i> without any meaning or 2.1.3.5 Assigned to global signal in <i>function statement</i> ,
2112b(W1)	Assignment of return value for <i>function</i> is not made for all cases. Hazardous because the results of RTL and gate level simulation will not match.
2114 (N)	<i>Register declaration</i> has no comment. (not to be checked by default setting)
2123a(E)	'<=' is used in a <i>function declaration</i> . Hazardous because this is unclear syntax that the results of RTL and gate level simulation will not match.
2123a(W1)	A signal without an input declaration "<name>" is used in a <i>function declaration</i> .
2124b (W1)	A signal without an <i>input declaration</i> "<name>" is used in a <i>task declaration</i> .
2125(W2)	<i>A task is used</i> .
2126(E)	There is clock edge description in a task declaration.
2131a(E)	An argument bit width differs from function input declaration bit width.
2131b(E)	An argument bit width differs from task input and output declaration bit widths.
2132(E)	Bit width of return value is different from assigned bit width.
2134a(W1)	Assignment statement in end of function declaration is not a return value assignment.
2135a(W1)	A value is assigned to signal "<name>" within module in a function declaration.
2135b(W1)	A value is assigned to signal "<name>" within module in a task declaration.

Verilint Warning

function statement related

W19 : The width of signal to be assigned by *assign statement* is larger than *function* bit width.(Truncation of extra bits)

W163 : *function* bit width is larger than the width of signal to be assigned by *assign statement*.(Truncation of bits in constant integer conversion)

W425 : Task sets a global variable.

W18 : Variable possibly not assigned in all paths.

W489, W499 : Last function statement does not assign to the *function*.

W554 : Assignment made by (\leq) in *function* . Change to \rightarrow ($=$).

2.1.2. Define combinational circuits using the *function statement* (Verilog only)

[1] The <i>function statement</i> should not be used for asynchronous reset line logic in an <i>always construct</i> for FF inference	mandatory
[2] A non-blocking assignment (<code><=</code>) should not be used in <i>function statements</i> (Verilog only)	mandatory
[3] All arguments are defined as <i>function statement</i> inputs	mandatory
[4] <i>task statements</i> should not be used (Verilog only)	recommend 1
[5] Clock edge descriptions should not be used in <i>task statements</i> (Verilog only)	mandatory

Example Code

```

module MUX(Y, A, B, C, D, S, Y);
input      A, B, C, D;
input[1:0] S;
output     Y;

function FUNC_MUX;
input A, B, C, D;
input[1:0] S;
begin
  case (S)
    2'b00 : FUNC_MUX = A;
    2'b01 : FUNC_MUX = B;
    2'b10 : FUNC_MUX = C;
    2'b11 : FUNC_MUX = D;
    default : FUNC_MUX = 1'bx;
  endcase
end
endfunction

assign Y = FUNC_MUX(A, B, C, D, S);

endmodule

```

Do not use
non-blocking
assignment

Example 2-2a Description using *function statement*

```

always @(posedge CLK or negedge RST_X)
  if(bar(RST_X))
    DOUT <= #P_STB 1'b0;
  else
    DOUT <= #P_STB DIN;

function bar;
input RST_X;
  bar = ~RST_X;
endfunction

```

Example 2-2b Description using asynchronous reset in *function statement*

Explanation

For more software-oriented descriptions, it is possible to define sub-programs and then call them from various locations in the HDL description. Sub-programs can contain function and task. Function can be considered as an additional operator. Return values are restricted to a single value. Tasks are used in *sequential process statements* and can have multiple return values.

When function is used for conditional expression to infer reset in if statement (Example 2-2b), it will not be synthesized correctly. function statement should not be used for asynchronous reset line.^[1] It is safer to consider *function statement* only for combinational circuit, and to use a call from continuous assignment.

In terms of syntax, the use of a non-blocking assignment (\leq) for a return value of a function is permitted. However, it will not result in the expected behavior because, in simulation, the change of output (return value of the *function*) occurs one moment late. But the synthesis generates a correct circuit in the expected behavior such that there will be a mismatch between RTL and synthesized logic.^[2]

Timing-related descriptions cannot be defined inside a *function statement*. *function statement* is called from either inside of *always construct* or from a *continuous assignment*, but in the RTL description it is often called from a *continuous assignment*.

When describing the function statement, declare inputs for all input signals used inside the function statement.^[3] When there is a signal without input declared and a signal of same name exists, the signal is directly read.

In this case, this function itself does not operate even if global signal value changes. Logic synthesis tool does not recognize the input with declaration even if global signal is used and generate circuit. Therefore, simulation results may differ between RTL and gate level that is problematic.

Multiple outputs can be described in a *task statement*. If multiple outputs are described inside sub-programs, it may cause trouble, which is hazardous. Also, because value can be retained by reg variable, it is recommended not to use task statements in RTL.^[4] Moreover, clock edge description in task statement can not perform logic synthesis.^[5]

Usage of *function statement* is explained in RMM: 5.2.13.

Note: 2.1.2. is consulted only for Verilog. 2.1.8 in the VHDL version describes about *function statement*.

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

[1] Match the argument bit width with the bit width of the <i>function statement</i> input declaration (Verilog only)	mandatory
[2] Match the return value bit width with the bit width of the assignment destination <i>signal</i> (Verilog only)	mandatory
[3] When returning values to multiple signals, use concatenation (Verilog only)	reference
[4] A <i>function statement</i> should end with a return value assignment	recommend 1
[5] In a <i>function statement</i> , global <i>signal</i> assignment should not be performed (Verilog only)	mandatory

Example Code

```

module DEC2TO4 ( AIN, EN, OUT, F );
input  [1:0] AIN;
input      EN;
output [3:0] OUT;
output      F;

function [3:0] DEC;
input [1:0] AIN;
begin
  if ( EN )
    case ( AIN )
      2'h0: DEC = 4'b0001;
      2'h1: DEC = 4'b0010;
      2'h2: DEC = 4'b0100;
      2'h3: DEC = 4'b1000;
      default: DEC = 4'bxxxx;
    endcase
  else
    DEC = 4'b0000;
  end
endfunction

assign {F, OUT} = DEC ( AIN );

endmodule

```

Use signals in the module since **EN** is not declared as **function** input

Bit widths of the left and right hand sides do not match. '0' is concatenated and assigned to the upper bits of the right side

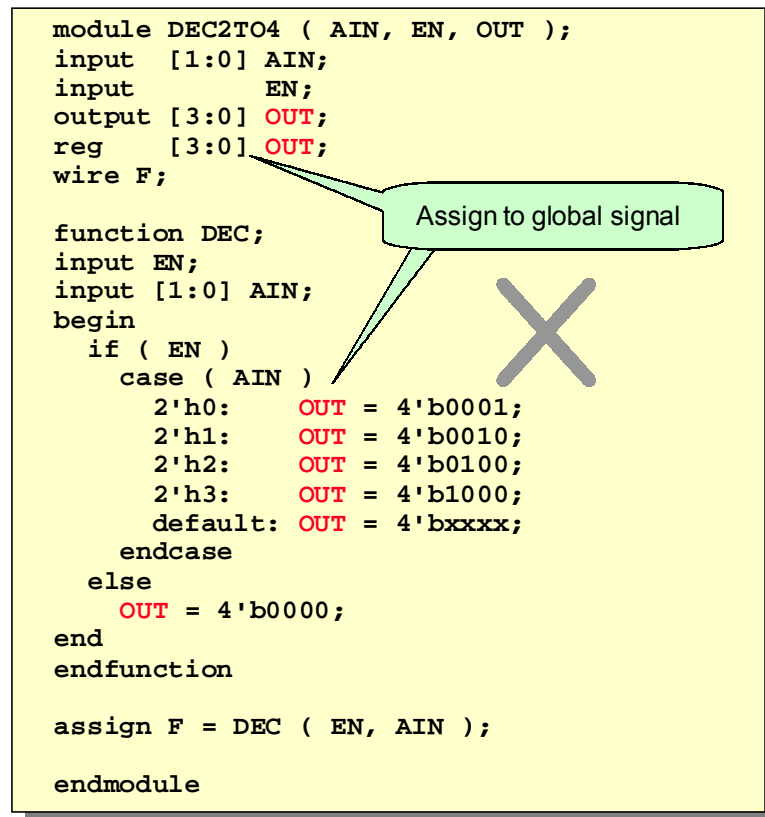
Example 2-3 function description (bad example)

Explanation

When describing the function statement, the bit width of function input declaration should match all input signal as well as all input signals are declared in function declaration part. If bit width does not match, some bit value of input signals may lost or misaligned.^[1]

Return values from the *function statement* are returned by assigning values to the *function name*. Make the return value and the bit width of the assignment destination *signal* match.^[2] If they do not match, "0" will be inserted in the upper bit when the bit width of the right hand is narrower than that of the left hand; the upper bit is truncated if the bit width of the right hand is wider than that of the left hand. Also, concatenation can be used on the left hand if values are to be returned to multiple *signals*.^[3]

The *function statement* should end by return value assignment to *function statement*.^[4] Some lines described after the return value in a *function statement* will not be executed. Logic synthesis and simulation ignore these lines. However, it might be confusing due to the fact that the described line becomes invalid.



```

module DEC2TO4 ( AIN, EN, OUT );
input  [1:0] AIN;
input      EN;
output [3:0] OUT;
reg  [3:0] OUT;
wire F;

function DEC;
input EN;
input [1:0] AIN;
begin
  if ( EN )
    case ( AIN )
      2'h0:  OUT = 4'b0001;
      2'h1:  OUT = 4'b0010;
      2'h2:  OUT = 4'b0100;
      2'h3:  OUT = 4'b1000;
      default: OUT = 4'bxxxx;
    endcase
  else
    OUT = 4'b0000;
  end
endfunction

assign F = DEC ( EN, AIN );

endmodule

```

Example 2-4 Assignment made to global signals inside function

Please avoid assignment to a global signal inside *function* and *task statements*.^[5] The description in Example 2-4 is a modified version of Example 2-3. The assignment in 2-4 is performed to the global signal OUT, not the function name DEC. In this function, value is assigned to global signal that calling by *assign statement* may seem unnecessary. However, the description, which calls the function statement, is necessary.

```
assign F =DEC(EN,AIN);
```

If *function statement* is not called, OUT value in the Example 2-4 becomes 4'bxxxx. If an *assign statement* is used to call, an empty value has to be returned like in the example. Moreover, if assignment is performed to both a function name and a global signal inside the same function statement, the description becomes complicated and easily leads to mistakes.

Also, when executing logic synthesis tools, if multiple outputs are described by one *function statement*, redundant logic tends to be generated since both relate, so that circuit quality may decrease.

Note : 2.1.3. is consulted only for Verilog. 2.1.8 in the VHDL version explains *function statement*.

2.1.4. Instructions for equation level descriptions (differs from VHDL)

- | | |
|---|-------------|
| [1] Operators having the same precedence are evaluated from the left (Verilog only) | reference |
| [2] Use bit-wise operators instead of equation levels, even in single-bit operator expressions (Verilog only) | reference |
| [3] Descriptions of equation levels have advantages in terms of area and speed (especially in selectors, encoders, and descriptions with large bit width) | reference |
| [4] Readability deteriorates if complex logic is described with equation levels | reference |
| [5] Logical operator should not be used for vector (Verilog only) | recommend 1 |
| [6] Be careful about bit width for reduction operators (Verilog only) | recommend 3 |

Example Code

```

module MUX (A, B, C, D, EN, S, Y);
input      A, B, C, D, EN;
input[1:0] S;
output Y;
wire W;

assign W = (A & ~S[1] & ~S[0]) | (B & ~S[1] & S[0]) |
           (C & S[1] & ~S[0]) | (D & S[1] & S[0]);

assign Y = W & EN;

endmodule

```

Example 2-5 Multiplexer description using logic operators

Explanation

When a *signal* in the right-hand side expression changes, the right side expression is automatically evaluated and *continuous assignments* substitute into the left-hand side after the specified delay has elapsed. Descriptions of *continuous assignments* generate combinational circuits by logic synthesis tools.

Several bit-wise operators can be used by Verilog-HDL: bit-wise negation (~), bit-wise AND (&), bit-wise OR (|), and bit-wise XOR (^). Each operator is evaluated as precedence. Since the precedence of '&' is higher than '|' used in the Example 2-5, '|' is evaluated after evaluating '&' (parentheses are not required). However, use parentheses to improve the readability of expressions that have different operators. Expressions will be evaluated from the left if they have the same precedence.^[1]

If an operand is single-bit (scalar), the operation results will not differ even if the logic operators, which are logical negation (!), logical AND (&&), or logical OR (||), or bit-wise operators, are used. However, when an operand is a vector, the logic operator will interpret results to be FALSE (1'b0) when all bits are 0, or other values to be TRUE (1'b1), then will evaluate the operation, so the operation results will become single-bit.^[2] Note that the

operation results of logic operators and bit-wise operators are different when the operand is a vector. See “2.10.1. Order of operators and assignment of X” for more information.

If “5.2.3.Flatten (set_flatten)” is not executed by logic synthesis or cannot be executed, Equation level descriptions are more beneficial from an area or speed standpoint than if or *case statement* descriptions. The difference is quite clear in simple descriptions such as for selectors and encoders. This difference is more noticeable as the logic gets larger and the bit width gets wider. In circuits with little margin with respect to area and speed, it would be recommended to describe Equation levels.^[3]

However, Equation level descriptions deteriorate the readability of the RTL descriptions and make debugging more difficult.^[4] In normal designs, it would not be advisable to forcibly describe just describing Equation levels. There would not be much of a difference as long as the description style defined in this Design Style Guide is followed, even if description is by *if statements* and *case statements*.

It is thought to be better to describe Equation levels for simple logic consisting only of ‘&’ or ‘|’. However, it would not be recommended to forcibly use Equation levels to describe easy descriptions that should be described using *case statements* and *if statements*.


In order to create designs with better performance without sacrificing readability, we would recommend creating a function library as explained in “3.1.Create function libraries”. Especially in the design of data path parts, area and speed after logic synthesis will be improved when using selector and multiple arithmetic operations as simple function library without FFs.

When using a function library, readability is not compromised since one simple name corresponds to one logic. Moreover, results after logic synthesis will also become better if equation levels are used in the description inside the function library. Please refer to “3.1.5. Parameterize the array range of module I/O” for the descriptions of selectors with variable bit widths.

Verilog-HDL use logic operators (&&, ||, !) and bit-wise operators(&, |, ^, ~).

Refer to “2.10.1.Order of operators and assignment of X”. A logic operator judges whether the operand is 0(FALSE) or other value(TRUE) and then returns 1 bit (0 or 1), while a bit-wise operator calculates by every bit.

A logic operator is to perform logical operation of each conditional expression of an *if statement*. A logic operator is used when the right-hand side and left-hand side are both (0,1) 1 bit. Be careful if using a logic operator to a value with bit width, as it may be problematic.

<pre>if(A == 4'h5 && !Enable_X) Q <= 1'b1; else Q <= 1'b0;</pre>		<pre>if(A == 4'h5 & !Enable_X) Q <= 1'b1; else Q <= 1'b0;</pre>
--	---	---

Example 2-6 Usage of logic operator and bit-wise operator

2.1. Combinational Logic

In cases where logical operation is 1 bit to 1 bit, the result will remain the same regardless of whether a logic operator is used or bit operator is used. Description style is better when only a bit-wise operator is used since it is unlikely to cause mistakes. However, if a designer is familiar with C language, a logic operator may be easy to handle in the conditional expression of an *if statement*.

The use of logic operators is not prohibited, but note that the right hand and left hand should both be 1 bit.

```
reg [4:0] A;
assign ALL_AND = & A;      assign ALL_AND = A[0] & A[1] & A[2] & A[3] & A[4];
assign ALL_OR  = | A;      assign ALL_OR  = A[0] | A[1] | A[2] | A[3] | A[4];
assign ALL_EX  = ^ A;      assign ALL_EX  = A[0] ^ A[1] ^ A[2] ^ A[3] ^ A[4];
```

Example 2-7 Usage of reduction operator

A reduction operator is a useful operator unique to Verilog-HDL, which VHDL can not use. However, if the bit widths of the values to be executed are too large, functions with many levels will be performed. Pay attention to the bit width.^[6] When bit width is 1 bit, no operation will be performed.

RTqualify checks the following

- | | |
|----------|---|
| 2143(N) | A logical operator is used on a 1-bit signal.
(not to be checked by default setting) |
| 2145(W1) | Logical operator '['&&', ' ']' performed to multi-bit signal |
| 2146(W3) | Checks "reduction operator to one bit signal is meaningless" |

Checker cannot check other items.

Verilint Warning

W563 : Reduction of a single bit expression is redundant.

W575 : Logical NOT operating on a vector

W576 : Multibit operand in a logical expression

Note: 2.1.4 is consulted only for Verilog Point of view differs in logic description between Verilog HDL and VHDL

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

- | | |
|---|-------------|
| [1] Use nesting of <i>conditional operator</i> (?) only once (Verilog only)
- Use of (?) should be limited to 10 times at most (MANDATORY) | recommend 3 |
| [2] Where a conditional expression is an unknown value 'x', the assignment values are compared for each bit of the <i>branch statement</i>
- Where the conditional expression is an unknown value 'x', a value will be more precise using <i>if statement</i> (Verilog only) | reference |
| [3] The result should not be a vector in the conditional expression of <i>if statement</i> and conditional expression (?) (Verilog only) | recommend 2 |

Example Code

```

module MUX (A, B, S, Y);
input[3:0] A, B;
input S;
output[3:0] Y;
wire[3:0] Y;

assign Y = (S==1'b0) ? A : B;

endmodule

```

Example 2-8 Multiplexer description using a assignment with the *conditional operation*

Explanation

The above example shows a multiplexer described using a conditional operator. In the assignment statement with conditional operator as above, A is assigned when S is the logic value 1'b0, but B is assigned in all other cases. The conditional operator can be nested, but since the readability of the description decreases and it becomes likely to make more mistakes in nesting, it is recommended that conditional operation be used only once.^[1]

```

assign Y = (S==1'b0)? (M==1'b0)? A : B : C;
assign Y = (S==1'b0)? A :(M==1'b0)? B : C;

```

When 2 or more ? exist, as in the above sequence, the nesting relationship (whether it is if-else if or else if-else) is difficult to verify. Therefore, it is recommended to describe in *always construct* using *if statement*.

```

always @(S or M or A or B or C) begin
    if(S==1'b0)
        if(M==1'b0) begin
            Y = A;
        end
    else begin
        Y = B;
    end
else
    Y = C;
end
(continue)

```

```
(continue)
always @(S or M or A or B or C) begin
    if(S==1'b0)
        Y = A;
    else if(M==1'b0)
        Y = B;
    else
        Y = C;
end
```

In the assignment statement with the conditional operator, if conditional expression becomes unknown 'x', the values in 2 branch statements are compared bit by bit.^[2] If they are same, the value is output, or the unknown value 'x' is output if they are different. For example, when S is unknown 'x' in Example 2-8, if A = 4'b1100, B = 4'b1010, output value Y becomes 4'b1xx0.

In *if statement*, value specified by an *else item* is assigned when the conditional expression contains the unknown value 'x'. Assignment with the conditional operator operate closer to the actual hardware than *assignment* using *if statements*. However, if conditional operator '?' is used more than once, the nesting relationships may become erroneous, so we recommend limiting the use of one *assign statement* to one time only. If nesting cannot be avoided, it should be limited to 10 times at most.

The *conditional expression* of *if statement* should be judged as being true or false. The result of a conditional expression in description should be 1 bit.

Not

```
reg [3:0] AVECTOR;
if(AVECTOR)
```

```
if(AVECTOR != 4'h0)
```

Should be ^[3]

RTqualify checks the following

- 2151a(W2) Incorrect bit width for conditional operation '?':
A= b? c[2:0]:d[3:0];
 - 2151b(W3) More than <n=1> conditional operators '?' are used.
 - 2151c(E) More than <n=3> conditional operators '?' are used.
 - 2153 (W2) Result of a conditional expression not single bit.
- The message of 2151 changes by changing the value of variable question_nesting_limit= 1,10.

Verilint Warning

W224 : Multi-bit expression when one bit expression is.

? is explained in RMM:5.5.7.

Note : 2.1.5 is consulted only for Verilog. 2.1.5 in the VHDL version, when-else is allowed up to 4 times.

2.1.6. Specifying the range of an array

- | | |
|---|-------------|
| [1] Specification of an array should be [MSB:LSB], if it is one-dimensional | recommend 2 |
| [2] The LSB of an array should be 0 | recommend 3 |
| [3] The index of an array should be simple <i>signal names</i> only | recommend 3 |
| [4] The range of an array index should be appropriately specified | recommend 2 |
| [5] For an array index, 'x' and 'z' should not be used | mandatory |

Example Code

<pre> module SEL4TO1(A,SEL,DOUT); input [3:0] A; input [1:0] SEL; output DOUT; assign DOUT = A[SEL]; endmodule </pre>	=	<pre> module SEL4TO1(A,SEL,DOUT); input [3:0] A; input [1:0] SEL; output DOUT; reg DOUT; always @(A or SEL) begin case (SEL) 2'b00: DOUT = A[0]; 2'b01: DOUT = A[1]; 2'b10: DOUT = A[2]; 2'b11: DOUT = A[3]; default : DOUT = 1'bx; endcase end endmodule </pre>
---	---	---

Only a simple *signal name* should be described in array index

Example 2-9 A signal used in an array index (selector description)

Explanation

The specification of array, which is one-dimensional, should be [MSB:LSB].^[1] Arithmetic operation is based on the assumption of [MSB:LSB] and if a reverse array is used, it is necessary to convert for arithmetic operation. It is recommended to specify [MSB:LSB], in which LSB is 0, even if the array does not perform arithmetic operations.^[2]

By using *signal names* in an array index, code is simplified and readability will be improved. In the example code of 4to1 selector in the Example 2-9, the left hand side code using signal name for array index is simple and easy to understand compared to the right hand side code using case statement. However, for description using *signal names* in an array index, many levels of logic gate would be generated by logic synthesis in case of intricate circuit.

Also, depending on logic synthesis tool, even if usage of *signal name* for array index is supported, highly redundant logic gates may be generated if circuit is intricate. An array index should simply consist of *signal names* only, and operations should not be included in it.^[3] However, as introduced in “2.9.2. Limiting loop-variable operation in *for statements*”, it will not be problematic if it consists of an operation with a loop variable

2.1. Combinational Logic

Pay attention to the MSB and LSB values of an array and the array range of the *signal names* specified in the index.^[4]

If values that exceed the MSB value or which are lower than LSB are assigned, you sometimes end up with unexpected results (Many simulators give results of 'x' when a value exceeding MSB is assigned and 'z' when a value less than LSB is assigned). Depending on the simulator, output values will vary and logic synthesis tools will not generate appropriate circuits.

Do not use 'x' and 'z' for an array index.^[5] For example, an error does not occur during simulation if describing

```
assign DOUT = A[1'bx];
```

but may occur with logic synthesis tool or incorrect circuit may be generated.

To simplify the MSB and LSB values of *signal names* specified in an index, set to [MSB:LSB], and make the LSB 0.

RTqualify checks the following.

- 2161a(W3) A bit width declaration is a [<declared bit specification>].
- 2162 (W3) LSB is non-zero.
- 2163 (W3) An array index is not a simple signal name.
- 2164a(W3) Range of indexes that can be assumed is too narrow.
- 2164b(W2) Range of indexes that can be assumed is too broad.
- 2164c(E) Non-constant or non-signal name used in array index.

It is also possible to test based on [LSB:MSB] by using variable `left_value_of_array`. The default is `left_value_of_array=high` (error when the left side is smaller).

2.2. *always construct* description in combinational logic

2.2.1. Avoid the risk of generating latches

- | | |
|--|-----------|
| [1] Latches are generated unless all conditions have been described
- Care should be taken not to create latches | mandatory |
| [2] Pay attention to warning messages in RTL check tool and logic synthesis tool, and verify the presence or absence of latch generation | reference |

Example Code

```

always @(S or AIN or BIN) begin
  if (S==2'b00)begin
    Y = AIN & BIN;
    Z = 0;
  end
  else if (S==2'b01) begin
    Y = 0;
    Z = AIN | BIN;
  end
  else if (S==2'b10) begin
    Y = 0;
  end
  else begin
    Y = 1;
    Z = 0;
  end
end
end

```

Since assignment expression to **Z** is not defined when **s==2'b10**, a latch, which holds output, is added to **Z** when **s** is **2'b10**

Example 2-10 Generated latch


Explanation

If the assignments to individual *signals* are not defined in all branch conditions, as in the Example 2-10, circuits following conditions, which have not been defined (or, defined conditions), will generate latches that are connected to an enable signal. In other words, a logic synthesis tool will recognize the output *signal* as a circuit like latch because condition has not been defined. As a result, latches are generated in order to retain the output value.^[1]

There is also a risk that the generated unexpected latches may cease to function properly due to setup time and hold time violations. There are also concerns that unexpected latches could become obstacles in clock-tree synthesis or a factor in lowering the fault detection rate.

To generate a combinational circuit from the above example code, describe the Z output with an S="10" condition and specify the assignment of output values for every *signal* in all the branch conditions.

Latches are also generated if the *else item* is not defined. Inferences of latches and combinational circuit using an *always construct* are quite similar and cannot be distinguished. The unintentional creation of latches can be confirmed with the warning messages of logic synthesis tools, but in consideration of greater safety, you should verify this with a RTL check tool.^[2]



```

always @ (S or AIN or BIN) begin
  Y = 0;
  Z = 0;
  if (S==2'b00) begin
    Y = AIN & BIN;
  end
  else if (S==2'b01) begin
    Z = AIN | BIN;
  end
  else begin
    Y = 1;
  end
end

always @ (S or AIN or BIN) begin
  if (S==2'b00) begin
    Y = AIN & BIN;
    Z = 0;
  end
  else if (S==2'b01) begin
    Y = 0;
    Z = AIN | BIN;
  end
  else begin
    Y = 1;
    Z = 0;
  end
end

```

Example 2-11 Description modified so that latches are not generated

If the branch conditions are complex and it is difficult to define output values for all branch conditions, or if the output values differ only during limited conditions, the following can be done to facilitate description: define the initial output value, or describe the assignment expressions only for conditions that differ from the initial value.

The description on the left side of Example 2-11 has a redefinition of Example 2-10. Since Y and Z often turn out to be 0, 0 is set to the initial value and assignment expressions are described only during different conditions.

Latches are not generated here because a value is always assigned as the initial value. As explained in “2.2.3. Initial value description in *always constructs*”, this description tends to make racing cause malfunctions, so caution is recommended.

As described in the right-hand part of Example 2-11, it is probably best to write the description in such a way that the values of all the output signals are determined in each condition.

Essentially, it is not desirable to describe too many output *signals* in a single *always construct*. Refer to “2.6.2. Avoid defining multiple output *signals* in a single *always construct*”.

In RTqualify, the following message is output.

2211(E) Possible generation of an unintended latch.

It is difficult to judge whether this description is created to infer latch or it is a combinational logic description mistake; still, both cases violate the Style Guide rules.

Verilint Warning

W410 : Not every case described in *always construct*. A latch is inferred

Erroneously inferred latch is explained in RMM:5.5.2.

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

- | | |
|--|-------------|
| [1] Define all the signals, which are in conditional expression and in the right-hand side of <i>assignment statements</i> in the <i>always constructs</i> , in the sensitivity list | mandatory |
| [2] Do not define constants and unnecessary signals in the sensitivity list | recommend 2 |
| [3] Multiple event expressions should not be described with <i>always constructs</i> (an event expression is necessary for one) | mandatory |
| [4] The sensitivity list should be verified with a RTL check tool rather than a logic synthesis tool | reference |

Example Code

Description (a) : Incomplete sensitivity list

```
always @(A or B) begin
    Y = A & B & C;
end
```

Description (b) : Complete sensitivity list (OK)

```
always @(A or B or C) begin
    Y = A & B & C;
end
```

Example 2-12 Definition of a sensitivity list

Explanation

Define all input *signals* that are used in the *always construct* in sensitivity lists.^[1] *Signals* such as the following should be defined in the sensitivity lists: operands on the right-hand side of assignment in the *always construct*, conditional expressions of *if statements* or selection expression of *case statements*, arguments of *functions* or *tasks*.

Logic synthesis tools ignore the sensitivity lists for *always constructs* of combinational circuits: circuits are synthesized as a combinational logic, which is driven by all the events of all the signal inputs on *always construct*. Three simple input AND gates are synthesized in description (a) above. In description (a) however, signal C which is used in the assignment, is not defined in the sensitivity lists. Therefore, no assignments are made even if C changes, and mismatches with the gate level simulation results occur. Define all *signals* in the sensitivity lists as in description (b) above.

Do not describe multiple event expressions "@()" in *always construct*.^[3]

```
always begin
    @(posedge CLK)...;
    @(posedge CLK)...;
end
always begin
    @( A)...;
    @( B)...;
end
```

2.2. *always construct* description in combinational logic

Logic synthesis with this kind of description might be possible but not in many cases. Also, description without any event expression "@" has risk of unlimited loop at simulator that is hazardous.

The issue of sensitivity list is hazardous since it will cause a mismatch in results between the RTL and gate level simulations. Some logic synthesis tools may not output a warning message properly. Always use RTL check tool. ^[4]

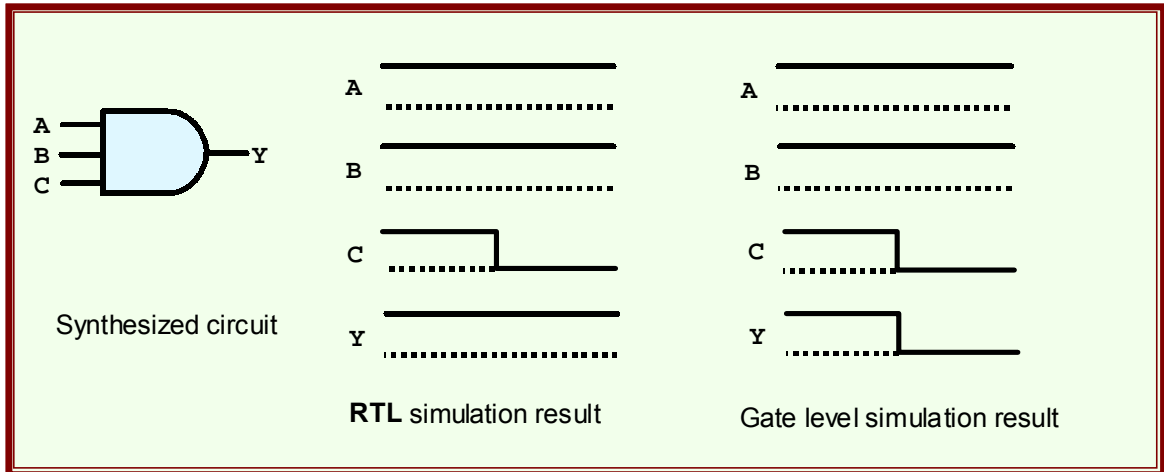


Figure 2-1 Mismatch of RTL and gate level simulation results

RTqualify checks the following items.

- 2221(E) A signal not included in sensitivity list is used.
- 2222a(W2) A constant is included in sensitivity list.
- 2222b(W2) An unneeded signal is included in a sensitivity list.
- 2222c(W1) More than one event expression in an always construct.


```
always @(A or B) begin
    if....
    @(C or D)....
    ....
end
```
- 2222d(E) There is an always construct without a sensitivity list.

Regarding sensitivity list, please be careful with the following, other than above checks.

- 2622(E) A signal to which an assignment is made in an always construct is included in sensitivity list.

Verilint Warning

- W122 : Signal name not described in sensitivity list exists
- W456 : Included in sensitivity list, but not used.
- W502 : Variable included in sensitivity list is internally modified
- W17 : Range (rather than full vector) in the sensitivity list.

Sensitivity list is explained in RMM:5.5.4.

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

- | | | |
|-----|--|-----------|
| [1] | Do not mix blocking assignments (=) and non-blocking assignments (<=) in combinational <i>always construct</i> | mandatory |
| [2] | Do not assign over the same signal using a non-blocking assignment for combinational circuit | mandatory |
| [3] | Do not assign over the same signal in an <i>always construct</i> for sequential circuits | mandatory |

Example Code

```

reg [7:0] ReadEnable;
always @(PCA or RDb) begin
  ReadEnable <= 8'd0;
  if (RDb == 1'b0)
    case (PCA)
      3'b000: ReadEnable[0] <= 1'b1;
      3'b001: ReadEnable[1] <= 1'b1;
      3'b010: ReadEnable[2] <= 1'b1;
      3'b011: ReadEnable[3] <= 1'b1;
      3'b100: ReadEnable[4] <= 1'b1;
      3'b101: ReadEnable[5] <= 1'b1;
      3'b110: ReadEnable[6] <= 1'b1;
      3'b111: ReadEnable[7] <= 1'b1;
    endcase
end

```

First assign initial value of 0s to all bits

Assign 1 for each bit
This cause a problem
in the case of non-blocking
assignments



```

reg [7:0] ReadEnable;
always @(PCA or RDb) begin
  ReadEnable = 8'd0;
  if (RDb == 1'b0)
    case (PCA)
      3'b000: ReadEnable[0] = 1'b1;
      3'b001: ReadEnable[1] = 1'b1;
      3'b010: ReadEnable[2] = 1'b1;
      3'b011: ReadEnable[3] = 1'b1;
      3'b100: ReadEnable[4] = 1'b1;
      3'b101: ReadEnable[5] = 1'b1;
      3'b110: ReadEnable[6] = 1'b1;
      3'b111: ReadEnable[7] = 1'b1;
      default: ReadEnable = 8'bxxxxxxxx;
    endcase
end

```

First assign initial value of 0s to all bits

Assign 1 for each bit
OK for blocking assignments

Example 2-13 Description of overwrite (initial value) assignments

Explanation

It is safer to use only blocking or non-blocking assignments in the *always constructs* for combinational circuit and avoid mixed use.^[1]

In the description in the upper portion of Example 2-13, '0' is assigned first as the initial value for the "ReadEnable" output *signal*. After this, only specified bits are set to '1' by the selection expression of *case statements*. In *non-blocking assignment statements*, the left-

2.2. *always construct* description in combinational logic

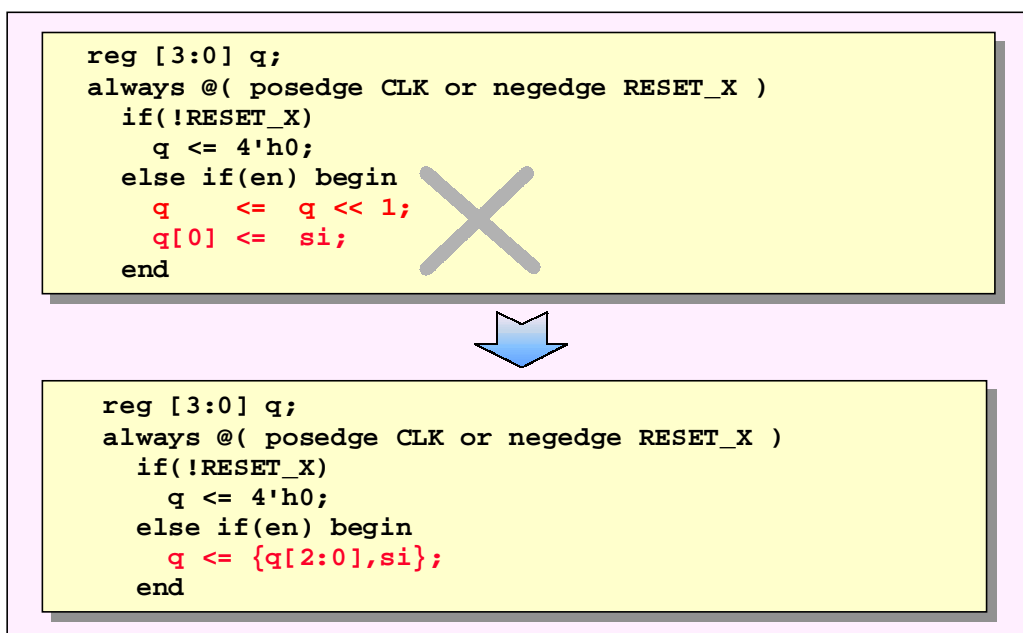
hand side members are assigned after all of the right-hand side expressions are evaluated. Because the assignment of a value to the "ReadEnable" signal takes place after the evaluation of all assignment statements has ended, the value after assignment is not defined.

In simulations, it appears that this assignment event is processed in the described order. However, there is no clear definition in the Verilog-HDL syntax regarding the order of assignments. Among the compiler type Verilog-HDL simulators, it often happens that simulation speed is improved by the fact that this sequence is independent of the description order. Even when the sequence is in description order on some Verilog HDL simulators, a hazard occurs because evaluation timing and assignment timing are different, resulting in factors that drive mistake even in another *always construct* referring these signals. The description at the bottom of Example 2-13 is one that has used a blocking assignment. When the blocking assignment is assigned, the assignment is already completed at the time this line is evaluated, so the assigned values on the lines after that are certainly valid.

Also, there is no risk of the hazard becoming problematic because the process does not migrate outside of the *always construct* in the interval between initial value assignment and reassignment. To avoid trouble such as this, it would be safer to use blocking assignments.

However, as introduced in "2.3.2.Circuits will vary with non-blocking and blocking assignment", non-blocking assignments (\leq) must be used in the *always construct* of FFs. Some designers point out that confusion is likely if blocking assignments are used only in the *always constructs* of combinational circuits.

When non-blocking assignments (\leq) are used as the description style even in the *always constructs* of combinational circuits, descriptions that assign an initial value like the one in Example 2-13 are not supposed to be used.^[2] Descriptions with overwrite assign to same signal are prone to cause problems, so it is best to avoid them whenever possible, even when making blocking assignments.

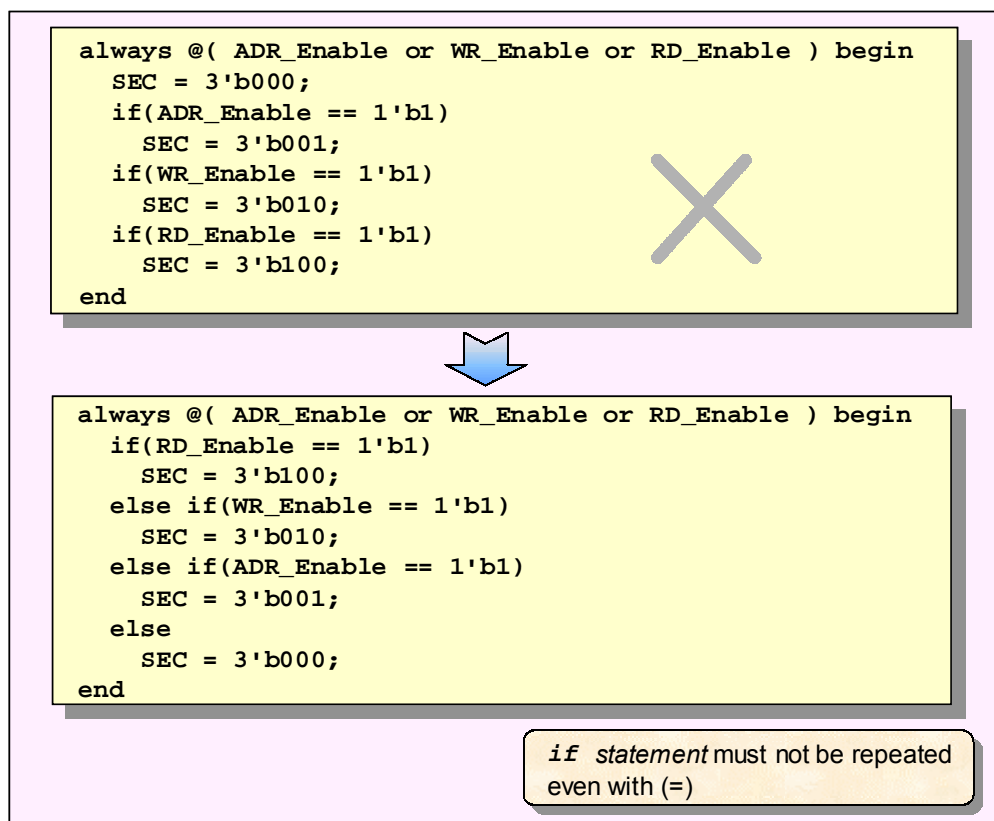


Example 2-14 Overwrite assignment in a sequential circuit

Non-blocking assignments must always be used in the *always constructs* of sequential circuits. Initial value descriptions must not be made in this case too.^[3] The upper diagram in Example 2-14 is a description of a serial-parallel conversion that uses a shift operator (<<). The portion in the upper part of Example 2-14 where 0 is assigned to bit [0] of *signal* q by the << operator is then overwritten and assigned q[0]<= si;" in the next line. In this sort of description, q[0] <=si; is not always assigned after the assignment operation q <= q<< 1; is assigned. Therefore, malfunctions may occur. This description should be described in a single line, as in the lower part of Example 2-14.

Initial value assignment or overwrite descriptions are also generated by using multiple *if statements* and *case statements* in addition to this. In the upper diagram in Example 2-15, the *if statement* of if(WR_Enable) and finally the *if statement* of ig(RD_Enable) are described after the line for the ig(ADR_Enable) *if statement*. In this case, if it were assumed that a non-blocking assignment (<=) was used instead of a blocking assignment (=), the ADR_Enable would be 1 and even if it is assumed that 3'b001 had been assigned to SEC in this description, it would be overwritten after that as long as RD_Enable is 1 and SEC ends up being 3b'100. There is also a possibility of malfunctions occurring in this situation since 3b'100 is not necessarily assigned after 3b'001. In this manner, it will more easily become a source of difficulty if you line up *if statements* in an *if statement* description without using an *else item*. Try to write the description so that *else branches* are always added and connected together as a single *if statement*, as illustrated in the lower diagram in Example 2-15.

Try to avoid writing the upper description of Example 2-15, even when not only non-blocking assignment(<=) but also blocking assignment(=) is used.



Example 2-15 Overwrite assignment description resulting from lining up *if statements*

2.2. *always construct* description in combinational logic

An initial value assignment problem will not occur if the upper diagram in Example 2-15 is assigned with a blocking assignment. However, for a designer who is familiar with the *if-else if* style of description, this will easily lead to mistakes and be difficult to debug.

Also, for logic circuits, NAND gates, NOR gates and INV gates operate in parallel. Take this circuit behavior into consideration to some degree in RTL description as well, and choose an *if-else* nesting structure or describe parallel behavior with the appropriate *case statement*. As in the upper diagram of Example 2-15, if a description is made in which syntax is executed in the order of description, a long path may be generated linearly so operating speed may decrease.

RTqualify checks the following items for initial value assignment.

- | | |
|----------|---|
| 2231(E) | Assignment description in an <i>always construct</i> uses a mixture of '=' and '<=' symbols. |
| 2232 (E) | Assignments are made in multiple places for same signal in an <i>always construct</i> for a combinational circuit.
Output only when using ('<=') |
| 2233 (E) | Assignments are made in multiple places for same signal in an <i>always construct</i> that generates an F/F. |

The description example of Example 2-15 is checked by “2621(W2) There is more than one *if statement* or case statement in a single *always construct*. ”.

Verilint Warning

W336: Use of (=) is prohibited in *always* block to infer sequential circuit.

W335 : Non blocking delay assignment in combinational *always* block.

W414 : Non-blocking assignment in combinational block.

W562 : Variable is assigned in both blocking and non-blocking assignment.

Note : 2.2.3 is consulted only for Verilog. 2.2.3 in the VHDL version includes some explanation but it is not as important as in the case of Verilog HDL.

2.3. FF inferences

2.3.1. Unify the description style of FF inferences

[1] Use <i>non-blocking assignment</i> in FF inferences	mandatory
[2] Do not use unsynthesizable FF inference styles	mandatory
[3] Set delay values for FF inference	recommend 3
[4] Do not use delay values other than in <i>always constructs</i> that infer FFs	recommend 1
[5] Specify delay values with integral numbers and do not use minus delay values	mandatory
[6] In FF inference with asynchronous reset, pay attention to the negedge or the posedge of the reset signal	mandatory
[7] Do not use both asynchronous set and reset	mandatory

Example Code

```
parameter DELAY = 1;

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

Example 2-16 FF inference

Explanation

Use *non-blocking assignment* (\leq) to FFs output.^[1] With normal *blocking assignment* ($=$), evaluation timing of the right-hand side and assignment timing to the left-hand side are at the same time, but in the case of *non-blocking assignment*, *assignment* to the left side is performed after the evaluation of the right-hand side is completely finished. For that reason, it is possible to avoid racing states at the same time in *non-blocking assignment statements*.

The example code shown here postulates a 1-bit FF, but the description style does not change even if there are multiple bits.

Create gated clock

```
always @( CLK or EN ) begin
    GATECLK <= CLK & EN;
end
```

Register A

```
always @( posedge CLK ) begin
    A <= AIN;
end
```

Register K

```
always @( posedge GATECLK ) begin
    K <= A;
end
```

Example 2-17 Gated clock description

Example 2-17 illustrates a description that creates a gated clock. The gated clock signal GATECLK is bit-wise AND output between the clock signal CLK and enable signal.

In a gated clock, the timing relationship between clock and enable signals needs to be guaranteed. If enable signal changes while a clock is active, a pulse is generated and FF may malfunction as a result.

In gate simulation, timing problems may cause a malfunction. If a gated clock is used in a clock line, malfunction of the simulation result can occur not only in the gate simulation but also during RTL simulation. Figure 2-2 displays the description in Example 2-17 in a circuit diagram. Delay values have not been fully specified in Example 2-17. In this description, (the right-hand side expression of) the line, where gated clock is described, and (the right-hand side expression of) the line of FF inference ($A \leq AIN$) are evaluated by CLK rising edge.

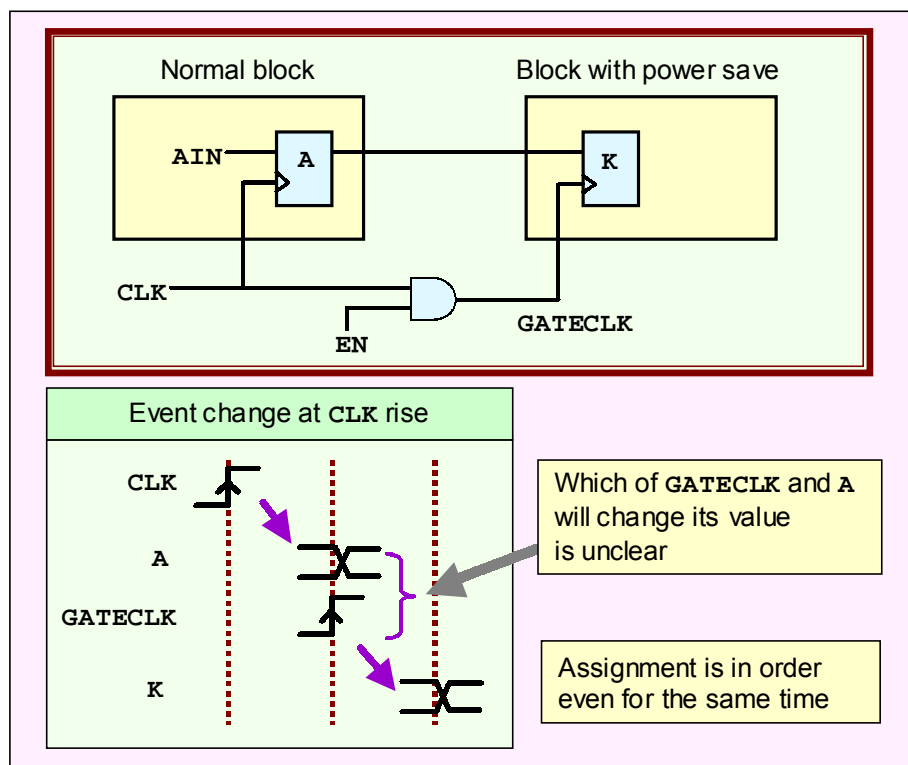


Figure 2-2 Racing state by gated clock

After evaluation, the value, which is previously evaluated, will be assigned to GATECLK and A at the same time by using non-blocking assignment(<=) (see “2.3.2. Circuits will vary with non-blocking and blocking assignment”). What it means by the simultaneous assignment to GATECLK and A is that it cannot be determined which one of these signal changes first. The previous assignment to register A might have been executed before the change in GATECLK.

This phenomenon does not necessarily occur. If the gated clock in Example 2-17 is described with an *assign statement*, there will no longer be any problems on most of the simulators. However, a solution to this problem is by any means not guaranteed with every tool. It is believed that the clock line description may change even further afterward depending on the actual device.

We strongly recommend inserting delay values into assignment expressions for signals ($Q <= \#DELAY D$).^[3] There is no particular problem if the design consists of a single clock, but when multiple clocks are present with a gated clock, etc., the racing problem tends to occur.


With regard to gated clocks, refer to “1.4.3. Gated clocks should be used with special care”.

It is one of possible concepts that delay values should only be set during D input assignment and do not put them in an assignment during an asynchronous reset. In any case, inserting delay values at the time of assignment is to prevent the racing problem during RTL simulation. Delay values are ignored during synthesis. Also be careful not to rely too much on delay values that result in using problematic descriptions (descriptions that are dependent upon asynchronous operation).

Delay values are specified in FF assignment expression, but they should not be specified in other parts (combinational circuits, etc.).^[4] If delay values are specified in a combinational circuit, you risk having the simulation become dependent on those delay values. You may choose to parameterize delay values (#DELAY), but do not describe directly delay values. Do this because it makes it possible for you to change delay values based on differences in technologies. This will also facilitate simulations in RTL that take delays into consideration.


(a) Erroneous asynchronous reset specification

```
always @(posedge CLK or negedge RESET_X) begin
    if(RESET_X==1'b1)
        Q <= #DELAY 1'b0;
    else
        Q <= #DELAY D;
end
```



(b) Description with 2 asynchronous inputs

```
always @(posedge CLK or negedge RESET_X or negedge LD_X) begin
    if(!RESET_X)
        Q <= #DELAY 8'h00;
    else if(!LD_X)
        Q <= #DELAY 8'h28;
    else
        Q <= #DELAY D;
end
```



Example 2-18 Erroneous reset description

There are many variations in FF inference besides the synthesizable descriptions shown here. There are descriptions that use *assign* and *deassign statements* in the reset operation as well as those that use UDP (User Designed Primitive). However, these should not be used since none of them can be synthesized with logic synthesis tools.

Pay close attention to the asynchronous reset edge when you are describing FF inferences.^[6] In Example 2-18(a), although the RESET_X signal is defined by *negedge* in the sensitivity list, RESET_X is compared with the '1' period in an *if statement*. This type of description will generate unintentional synchronous reset FFs in a logic synthesis tool.

Example 2-18(b) is an example in which two asynchronous sets and asynchronous resets are described.

When the value is 8'hff (8'h28 in the Example) in *if(!LD_X)* statements, the value '1' is set on LD_X *negedge* and value '0' is set on RESET_X *negedge* for each bit. In this case, FF with both of asynchronous set and reset is generated (Refer to "1.3.1.[7] Do not use a FF with both asynchronous set and asynchronous reset"). When the value is 2'b28 (Example 2-18(b)), it would set the same value '0' on LD_X *negedge* and RESET *gedge* for some bit. This type of description is hazardous because a logic circuit is generated in an asynchronous reset line. Also, FF may not be inferred correctly depending on logic synthesis tool that attention should be paid.^[7]

Be sure that the clock signal edge in an *always construct* is unified to the *posedge* or *negedge* and try not to mix them in the same design. (if possible, unify to *posedge*). If these are mixed together, you will end up inserting an inverter in the local position on the clock line, which could become an obstacle during the execution of "1.4.2. Use clock tree synthesis for clock balancing" or else you could have the potential for ATPG being unable to execute.

RTqualify checks all the times.

2311(E)	A '=' symbol is used in an always construct that generates an F/F.
2312(E)	It is possible that the F/F will not be generated.
2313(W3)	No delay is used in an always construct that generates an F/F.
2314(W1)	A delay is used other than in an always construct that generates an F/F.
2316(E)	Polarities of asynchronous reset signals do not match.
2317(E)	More than one asynchronous set/reset signal.

2313 outputs many warning messages if delay value is not described for FF assignment. If delay value is not set in cases such as single phase design, delete message by `ignore_message`.

Verilint Warning

W336 : Use of (=) is prohibited in *always* block to infer sequential circuit.

W280, 257: Delay value used (OK if this message is output)

W384 : Unintended clock or FF with set/reset is used.

W390 : More than one clock exist in module.

W391 : Wrong clock polarity.

W392 : Wrong reset polarity.

W395 : Multiple resets in the always block.

FF inference is explained in RMM:5.5.1.

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

- [1] Circuits generated by *non-blocking assignments* (\leq) and *blocking assignments* ($=$) may be different in FF inferences
- [2] Do not mix *blocking* and *non-blocking* assignments in FF inference *always construct* (Verilog only)

reference

mandatory

Example Code

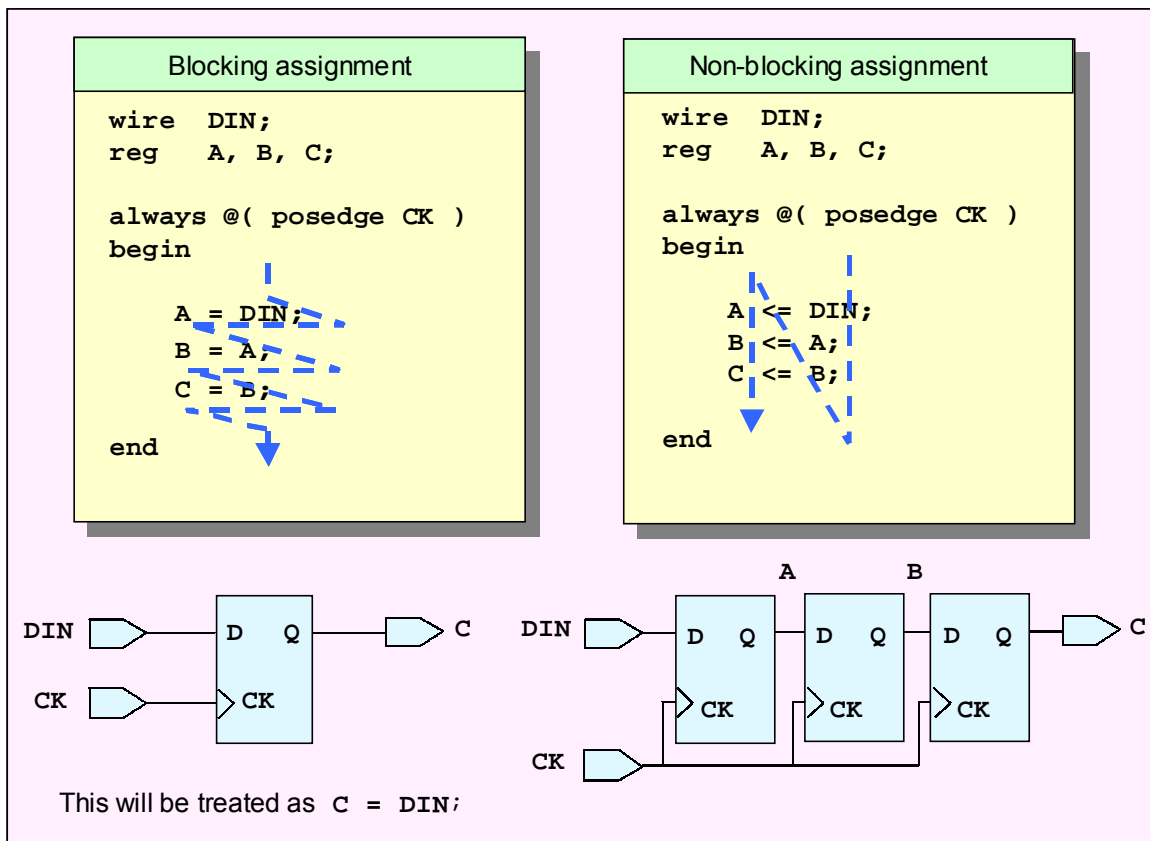


Figure 2-3 Differences in circuit between non-blocking and blocking assignment statements

Explanation

Keep in mind that circuit operation and synthesized circuits may differ with *non-blocking assignment statements* and *blocking assignment statements*.^[1]

In the *non-blocking assignment statement* in Fig. 2-3, the assignment of the left side signal is made after the right-hand value is defined at the same time. Therefore, the B and A values are evaluated and the assignment is made to the left side at the point where end is reached. The A and B values are also assigned to B and C respectively and are configured as a shift register.

In the *blocking assignment statement* in Fig. 2-3, evaluation of the right side and assignment to the left side are performed simultaneously. Since the assignment to A is performed simultaneously with the evaluation of the B value in the first assignment expression, the B value is already A when the next *assignment statement* is executed. As a result, values

B and C both have the same value A, and the circuit has FFs that are placed in parallel.

Fundamentally, you should use *non-blocking assignment statements* in FF inference descriptions. Also, you should keep “2.6.2.[1]. Do not describe more than one *if* or *case* in one a single *always construct*” in mind to avoid the differences between such *non-blocking* and *blocking assignment statements*.

It is also important to note that coexistence of *blocking assignment statements* and *non-blocking assignment statements* within a single *always construct* will not cause a syntax error in the language specifications, but that errors will occur when using a logic synthesis tool.^[2]

RTqualify checks the following

2311(E) A '=' symbol is used in an *always construct* that generates an F/F.

Blocking and *non-blocking assignments* are explained in RMM:5.5.5.

Note: 2.3.2. is consulted only for Verilog. 2.3.2 in the VHDL version describes about signal and variable.

2.3.3. Do not mix descriptions that have different edges

- | | |
|--|-----------|
| [1] Do not use two or more different clock edges within a single <i>always construct</i> | mandatory |
| [2] Do not use two or more identical clock edges within a single <i>always construct</i> | mandatory |

Example Code

Example code (a) : Synthesizable description

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

Example code (b) :
Unsynthesizable description

```
always begin
    @(posedge CLK);
    CNT <= CNT + 1;
    @(negedge CLK);
    CNT <= CNT + 2;
end
```

Example code (b) :
Synthesizable but inappropriate description

```
always begin
    @(posedge CLK);
    CNT <= CNT + 1;
    @(posedge CLK);
    CNT <= CNT + 2;
end
```

Example 2-19 Unsynthesizable description (bad example)

Explanation

Having both rising and falling clock edges coexist in a single *always construct* does not constitute a syntax error, but it is prohibited since it is unsynthesizable.^[1] Similarly, describing multiple different clock signals within a single *always construct* is prohibited even if the edges are the same.^[2]

This is all quite difficult to implement with the actual hardware. FFs that enable data reading for each clock edge do not exist independently. Similarly, it is also difficult to implement FF controlled by multiple clock signals. These descriptions will cause errors when using a logic synthesis tool. In addition, if the @(posedge CLK) description is not at the top of an *always construct*, logic synthesis becomes impossible.

Moreover, depending on the logic synthesis tool, it is possible to describe multiple clock edges in a single *always construct* for the same edge of a single clock. In this case, the logic synthesis tool generates state machine circuits that control the edge change order. However, this state machine circuit becomes an implicit state machine circuit that is not defined in the HDL description, so it can no longer be claimed to be a concise RTL description. Logic synthesis would be possible, but such a usage should be avoided.

RTqualify checks the following

- | | |
|---------|--|
| 2331(E) | More than one different clock edge in an <i>always construct</i> . |
| 2332(E) | More than one identical clock edge in an <i>always construct</i> . |

Verilint Warning

W421: Clock edge described after *always begin*.

W394: 2 or more clock edges described in an *always construct*.

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

[1] Do not specify FF initial values explicitly in *initial constructs* (Verilog only)

mandatory

[2] Logic synthesis ignores *initial constructs*, so it should not be used

mandatory

Example Code

```
initial
  CNT = 0;

always @(posedge CLK)
  CNT <= CNT + 1;
```



Example 2-20 a description that set the initial value (bad example)

Explanation

It is possible for *reg* signals used as an *always construct* output to assign the initial value in an *initial construct*. This value functions as the initial value in the RTL description, but since the reset signal is not defined, the initial value is not set by the reset signal when actually synthesizing at the gate level. Be aware that logic synthesis tools will ignore this *initial construct*.^{[1] [2]}

RTqualify checks the following:

2342(E) An initial statement cannot be used in RTL description.

Verilint Warning

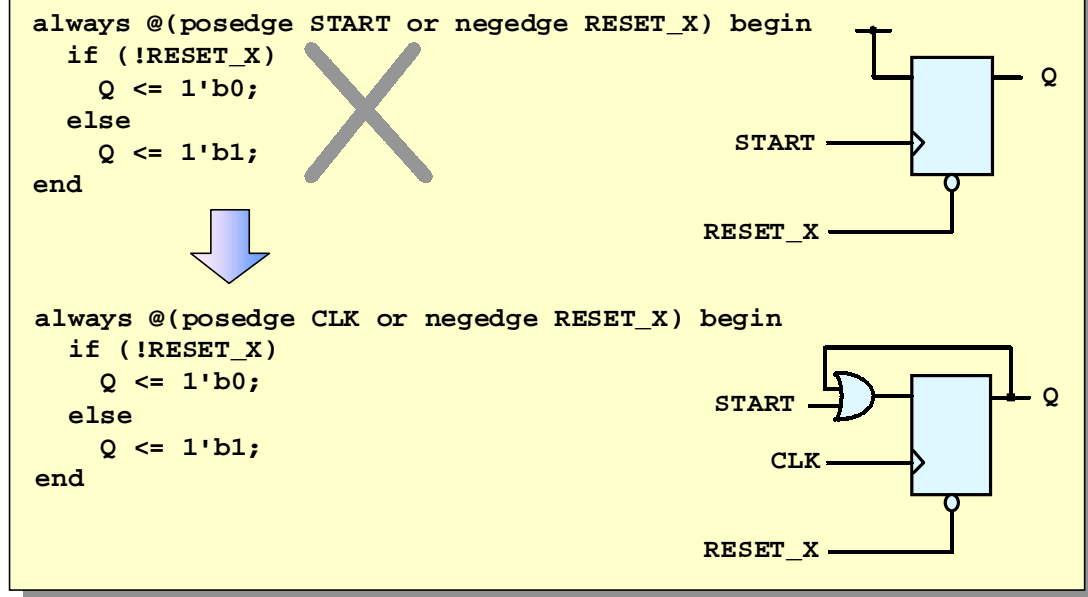
W599, W430: *initial construct* used.

2.3.5. Do not describe to generate FFs having fixed input values

[1] Do not describe to generate FFs having fixed input values

recommend 1

Example Code



Example 2-21 FF circuit with fixed input value

Explanation

A FF having a fixed input value is generated from the description in the upper portion of Example 2-21. In this case, '0' is output when the reset signal is asynchronously input, and '1' is output when the START signal rises. Therefore, the FF data input is fixed at the power supply, since the typical value '1' is output following the rise of the START signal

When FF input values are fixed, the fixed inputs become untestable and the fault detection rate drops. When implementing a scan design and converting to scan FF, a scan may not be executed properly, so such descriptions are not recommended.^[1]

As in the lower part of Example 2-21, be sure to construct a synchronous type of circuit and ensure that the clock signal is input to the clock pin of the FF.

Other than the example shown in Example 2-21, there are situations where for certain control signals, those that had been switched due to the conditions of an external input will no longer need to be switched, leaving only a FF. If logic exists in a lower level and a fixed value is input from an upper level, the input value of the FF may also end up being fixed as the result of optimization with logic synthesis tools.

In a situation like this, it may be difficult to eliminate it completely, although you should be careful to try to do so as much as possible.

RTqualify checks the following:

2351(W1) An F/F with a non-changing input will be generated.

2.3.6. Do not mix FF inference with asynchronous resets and without

[1] Do not mix FF inference with asynchronous resets and without in the same *always construct*

recommend 1

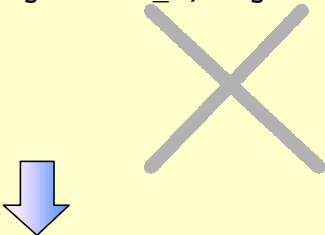
[2] Asynchronous reset is only one bit which is low active with *negedge*

recommend 3

Example Code

Example code(a) : FF with asynchronous reset and
FF without asynchronous reset coexist

```
always @(posedge CLK or negedge RESET_X) begin
    if(RESET_X==1'b0)
        A <= 1'b0;
    else begin
        A <= D;
        B <= A;
    end
end
```



Example code(b) : Separate FF with asynchronous reset and
FF without asynchronous reset

```
always @(posedge CLK or negedge RESET_X) begin
    if(RESET_X==1'b0)
        A <= 1'b0;
    else
        A <= D;
end

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

Example 2-22 Description where FF with asynchronous reset and FF without asynchronous reset coexist (bad example)

Explanation

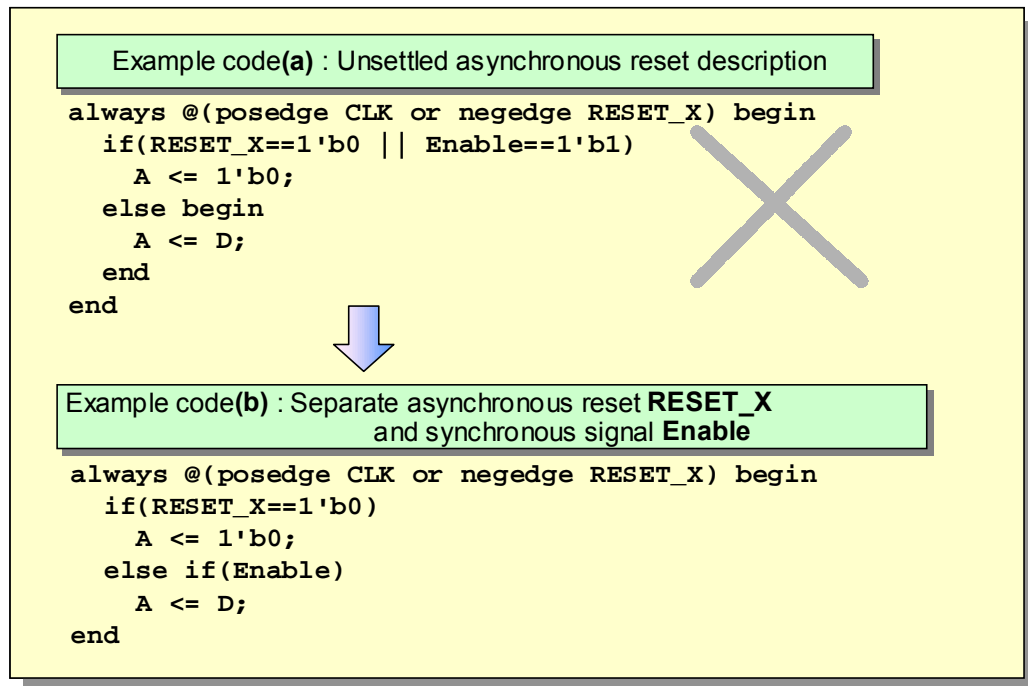
In the example code (a) in Example 2-22 above, a FF inference which has an asynchronous reset (signal A) and a FF inference without an asynchronous reset (signal B) are mixed together. Many of these types of description occur as a result of missing asynchronous reset descriptions. Therefore, you should always be sure to check whether all of the signals in an asynchronous reset description have been reset.^[1]

Also, when inferring FFs without asynchronous resets, you should try not to write descriptions that mix them together with FFs with asynchronous resets in the same *always construct*. The description in which a FF with an asynchronous reset and a FF without an asynchronous reset are mixed together is misleading, and capturing the behavior will become difficult during debug.

Since some synthesis tools may not perform synthesis correctly, your description should also be written in such a way that the *always construct* is divided, as in example code (b). Among signals, which are event defined by posedge or negedge, in sensitivity list, logic synthesis tools recognize only those, which appear in a condition expression of if statement on the first line of always construct, as asynchronous reset (or asynchronous set) signals.

In the example code (a) in Example 2-23, the signal *Enable* is not described with *negedge*, but it exists in the condition expressions of the *if statement* on the first line of the *always construct*.

If such a signal is present, logic synthesis tools cannot verify whether it is an asynchronous or a synchronous reset, so an error occurs. And even if we presume that the logic synthesis tool had not found an error, it would still be unable to determine what kind of circuit is output.



Example 2-23 Description where FF with asynchronous reset is not settled

Two asynchronous reset signals should not be described by adding the *posedge Enable* in the sensitivity list of the example code (a) in Example 2-23. As explained in “1.3.2 Reset line hazards”, that can be very hazardous if logic is present. Also, this description will result in an error in some logic synthesis tools.

Asynchronous resets should be designated as low active resets and specified by *negedge*.^[2] Inverted logic is used for asynchronous reset pins of all ASIC vendors. In RTL description, the input should be inverted to match the FF in a semiconductor library. Logic synthesis tools will generate a circuit correctly even if the description is made by positive logic with *posedge RESET*. However, inverter cells will end up being inserted into the FF asynchronous line.

Although “1.4.2. Use clock tree synthesis for clock balancing” explains that “clock line is automatically generated during ASIC layout”, it has recently been the case that reset lines, with the same mechanism as clock tree synthesis, are generated more often during layout. All asynchronous resets should be integrated in a *negedge*, and it would be better if logic cells were not inserted in reset lines.

As explained in “1.3.2. Reset line hazards” when you need to insert logic cells (AND, OR, INVERTER) or FFs in asynchronous reset lines, be sure that you construct a reset generation module in the top level and that it is only done in that level.

RTqualify checks the following:


- | | |
|----------|---|
| 2361(W1) | Describe with asynchronous set/reset signals is mixed with description without. |
| 2362(W3) | Asynchronous set/reset should be described using negedge. |

2.4. Latch inference

2.4.1. Clearly distinguish a latch inference from a combinational circuit

[1] Clearly distinguish a latch inference from the logic in other combinational circuits	recommend 1
[2] Create latch only blocks and infer latch in these blocks only	recommend 3
[3] Do not use latches with an asynchronous set/reset	recommend 1
[4] Avoid conditional feedback loops that contain latches	mandatory
[5] Do not use two level latches in the same phase clock	recommend 1

Example Code

Latch inference	Latch inference with asynchronous reset
<pre> module LAT(G, DATA, Q); input G, DATA; output Q; reg Q; always @(G or DATA) if(G) Q <= DATA; endmodule </pre>	<pre> module LATR(G, DATA, RESET_X, Q); input G, DATA, RESET_X; output Q; reg Q; always @(G or DATA or RESET_X) if(!RESET_X) Q <= 0; else if(G) Q <= DATA; endmodule </pre> 

Example 2-24 Latch inference

Explanation

Since latches are inferred using *if statements*, they closely resemble typical combinational circuits in description. Latch inferences allow data to pass through when the enable signal is turned ON and do not specify anything when it is turned OFF. As a result, the previous state is preserved.

Latch inferences are hazardous since they are not easy to distinguish from combinational circuits.^[1] Moreover, if complex logic is described in a latch inference's *if statement*, the logic synthesis tool may end up generating an unintentional gated clock circuit. To avoid this problem, latches should be hierarchized, and you should keep them separate from other descriptions.^[2]

If no branch that is appropriate to the conditions exists when a combinational circuit is described using *if statements*, logic synthesis tools will interpret that to mean the previous data is to be retained. Therefore, whenever the *if statement* branches are incomplete, you need to be aware that latches are often generated unintentionally.

2.4. Latch inference

Typically, there are not very many situations in which latches that have asynchronous set/resets are available in ASIC vendor libraries. And the latch inferences of a logic synthesis tool will vary, depending upon the tool, which is not exactly reliable, so you should avoid a latch inference, which has an asynchronous set/reset.^[5] (such as the description in right portion on Example 2-24)

D input data goes through a latch toward the output for the period that the gate signal is inactive. If there is a combinational feedback loop that contains a latch, it will mean that asynchronous loops exist. Also, if there is more than one latch in the same phase, it will mean that there are paths that pass through multiple latches. Circuits like this may be acceptable in terms of their behavior, but they create problems in timing analysis and should not be used.^[4]

Feedback loops that contain latches create paths whose timing is difficult to analyze, much like combinational circuits. In fact, even if it were assumed that they were false paths and merely feedback, a timing analysis tool would not understand them.

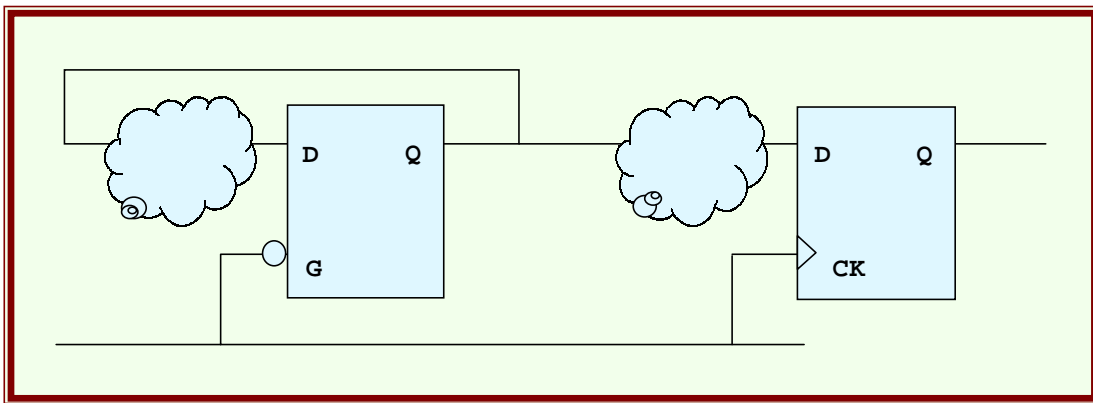


Figure 2-4 Feedback containing latch

The use of latches should be restricted because it complicates timing analysis. But there are many cases where latches are used to secure setup as explained in “1.5.3.Guaranteeing setup/hold and margin for synchronous RAM” and “3.3.7.Handling of different clocks”. Usage rules like these present no problem.(Figure 2-5)

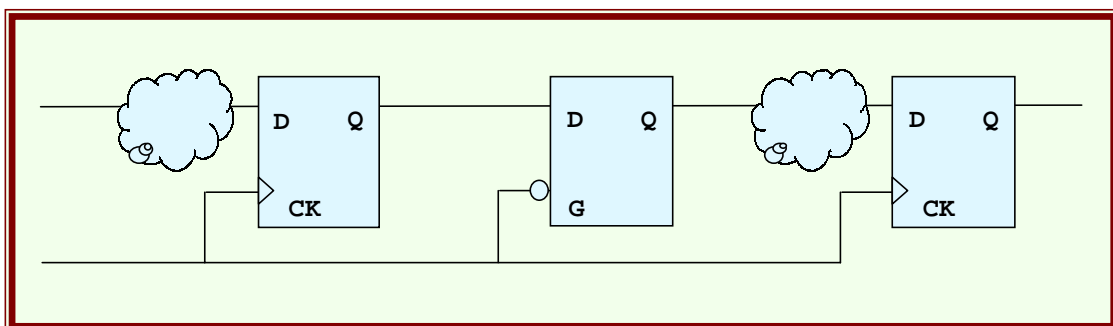


Figure 2-5 Use of latches

In latch-based designs, transfers are usually executed serially in clocks of different phases like the Figure 2-6.^[5]

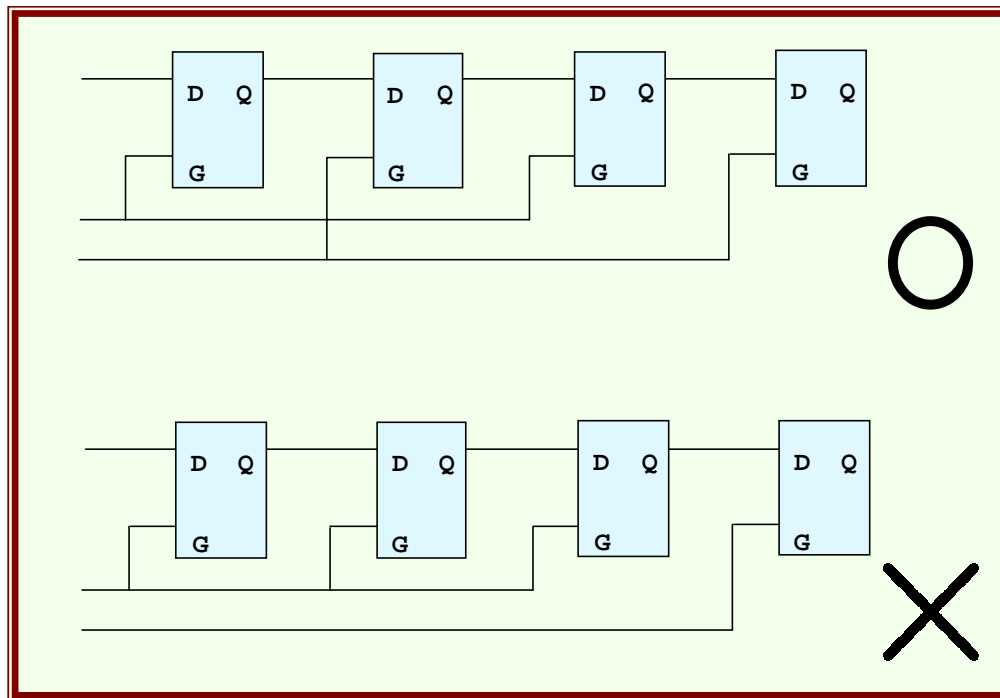


Figure 2-6 Latch-based design

RTqualify checks the following items

- | | |
|----------|---|
| 2411(N) | A simple latch will be generated. (A separate message for complex one is provided as there is a possibility of mistake in combinational logic.) |
| 2412(W3) | There is a latch that is not layered. |
| 2413(W1) | A latch will be generated with an asynchronous reset. |

Latch inference is explained in RMM:5.5.2.

Verilint Warning

W410: Latch inferred.

2.5. Tri-state buffers

2.5.1. Make a block for a tri-state buffer

[1] Make a block for a tri-state buffer	recommend 3
[2] Do not describe logic in conditional expression to infer tri-state	recommend 2
[3] Infer a tri-state buffer by assigning high impedance 'z'	reference
[4] Specify up to five tri-state buffer connectors at most	recommend 2
[5] Do not connect 2 or more outputs other than tri-state buffer even under the same conditions	mandatory
[6] <i>inout</i> should not directly be connected to input/output	mandatory
[7] Tri-state output should not be used in a <i>conditional expression</i> of an <i>if statement</i>	recommend 2
[8] Tri-state output should not be used in a <i>selection expression</i> of a <i>case statement</i> that is not assigned 'x' as default clause	recommend 2
[9] Tri-state output should not be entered in the <i>selection expression</i> of <i>casex</i> or <i>casez statements</i>	recommend 2

Example Code

```

module TRISTATE_1(EN, DATA, Y);
input  EN, DATA;
output Y;
wire   Y;
assign Y = EN ? DATA : 1'bz;
endmodule

```

continuous assignment used

```

module TRISTATE_2(EN, DATA, Y);
input  EN, DATA;
output Y;
reg    Y;
always @(EN or DATA) begin
    if(EN)
        Y = DATA;
    else
        Y = 1'bz;
end
endmodule

```

always construct used

Example 2-25 Tri-state buffer code

Explanation

When tri-state buffers are used, there are cases where paths, which do not require timing analysis, are analyzed. In Figure 2-7, it is necessary to analyze the timing from FF in A to the DBUS pin and the timing from the DBUS pin to B. However, there is no need to analyze the path from A to B.

If `set_input_delay` and `set_output_delay` are specified to the DBUS port in this instance, the path from A to B is no longer analyzed. However, if the tri-state buffer is not directly connected to the I/O port, or if timing analysis is performed from an even higher level, it would then be subject to timing analysis.

If constructing a bidirectional bus with the tri-state buffer in an ASIC in this manner, it would be problematic that paths might end up being analyzed unnecessarily.

There are situations in which `set_output_delay` must be set in the logic synthesis phase in order to prevent this phenomenon. For that reason, the tri-state buffer should be made on a block.^[1] As shown in the example, create a tri-state buffer module, then instantiate this module from the description of Example 2-25. By separating the tri-state code from other codes in this manner and by synthesizing in the following order at logic synthesis, a synthesis that has preserved the timing constraints before and after the tri-state becomes possible.

1. Compile the tri-state buffer block by itself first.
2. Before synthesis of the whole, specify `set_output_delay` to the input and `set_input_delay` to the output of the tri-state that has become a gate.
3. Synthesize the whole

Position it as a simple level without any logic since there may also be situations in which it is necessary to recognize the tri-state buffer in a layout other than this. However, wherever an I/O is to become a tri-state in the top level, module will be instantiated in the top level without change.

Only *signal names* are described in the tri-state buffer selection signal (EN in the description of Example 2-25), whereas logic is not.^[2] This is done because there is risk that the output will not be able to be turned on and off with the proper timing due to potential changes to the logic sequence in logic synthesis.

If a hazard enters the tri-state buffer selector signal logic, the output will collide with other output drives which could cause malfunctions or increase power consumption. Therefore, you should be especially careful about this.

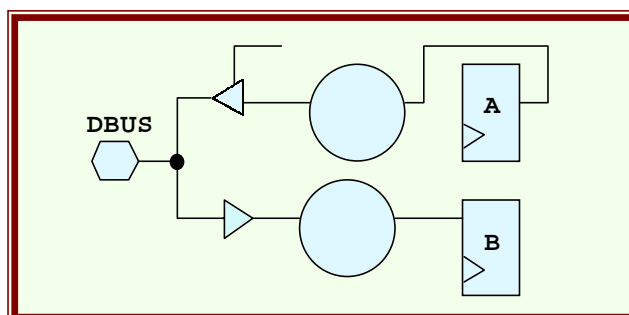


Figure 2-7 Circuit including bidirectional bus

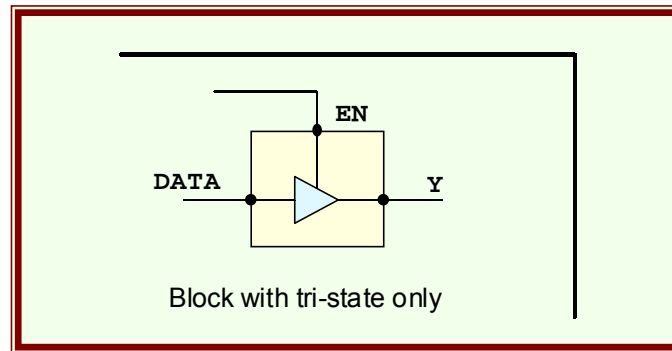
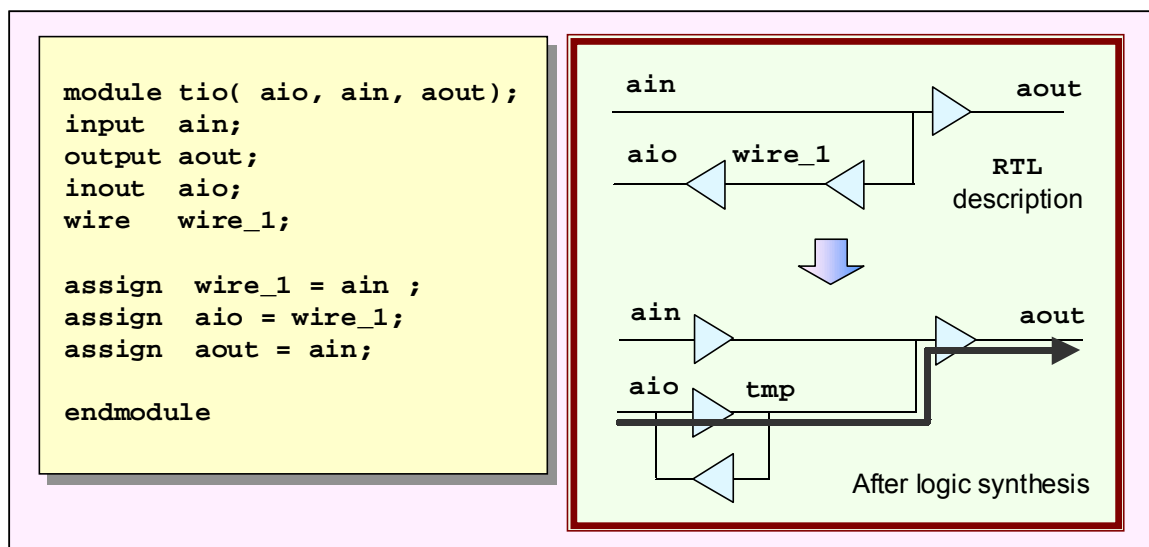


Figure 2-8 Create block with tri-state only

Be sure that you do not connect a signal using an *inout* declaration to signals with direct *input* or *output* declarations.^[6] Paths that do not exist in the RTL description are generated after logic synthesis is performed, and inconsistencies will occur in simulation results in RTL and at the gate level.

Figure 2-9 *inout* should not be directly connected to *input/output*

Except for cases where CPU exists within ASIC, it is better not to use bidirectional because of undesirable problems in design arising from the fact that bidirectional bus makes timing analysis difficult, as seen above, where wires will become long at layout, which decrease speed, and may cause inconsistency of the RTL and gate level simulations.

When using bidirectional from necessity, output drivers connected to tri-state bus should be 5 or less and avoid allowing the bidirectional bus to become heavily loaded.^[4]

In the case of net, where there are many connection numbers and when connecting long distance such as bidirectional bus, the current layout tools tend to drastically increase wire area.

It may seem that short wire is possible, but layout tools may not connect as we assume. It will be highly advantageous to design in a single direction without a bidirectional bus with a heavy load for speed improvement (not to excessively decrease speed quality). Not only the number of output drivers, but also the number on input side determines the load of bidirectional bus.

The number at input side cannot be inferred from the RTL description alone. Therefore, hierarchize the first gate on the input side as seen in “2.5.2. Consider high-impedance propagation in tri-state bus”.

To strengthen the drive capacity of output, two cells of BUF, INV and AND etc. are sometimes connected simultaneously to the same net. Such an adjustment should be done on layout, and the descriptions to connect two or more of output drivers other than tri-state should not be made in the RTL description.^[5]

For tri-state signals, propagation of 'x' should be considered in order to match simulation results in RTL and gate-level. When a signal value that includes 'z' is used in a conditional expression of *if statements*, *else item* will be executed. Normally, a value 'z' will become 'x' after passing through a logic gate, but in this case it becomes a fixed value. Refer to “4.4.2 Inconsistencies can occur between RTL and gate level with the propagation of 'x'” regarding this problem.

```
assign TZ[3:0] = (ENB == 1'b1) ? DB[3:0] : 4'bzzzz;
always @(TZ or DA or DC)
    if (TZ == 4'hf)
        IFO = DA;
    else // When TZ is 'z', 'x' does not propagate.
        IFO = DC;
```

Example 2-26 Tri-state output should not be entered in a *conditional expression* of *if statements* and a *selection expression* of *case statements*

Similarly, when it is entered in a selection expression of a case statement that is not assigned 'x' in default clause, 'x' will not propagate. Avoid using tri-state signals for conditional expression of if statements or selection expression of case statements.^{[7] [8]}

```
assign TZ[3:0] = (ENB == 1'b1) ? DB[3:0] : 4'bzzzz;
always @(TZ or DD)
    case TZ
        4'h0: CASEN = DD[0];
        4'h1: CASEN = DD[1];
        4'h2: CASEN = DD[2];
        4'h4: CASEN = DD[3];
        default: CASEN = 1'b0; // When TZ is 'z', 'x' does not propagate.
        // default: CASEN = 1'bx; // 'x' propagate.
    endcase
```

Example 2-27 Tri-state output should not be entered in a *selection expression* of *case statements* that is not assigned 'x' in default clause

When it is entered in a selection expression of casex or casez statements, 'z' will be recognized as don't-care and it will be difficult to propagate 'x'. Tri-state output should not be used for selection expression of casex and casez statements. The issue of casex and casez statements is explained in “4.4.2. Inconsistencies can occur between RTL and gate level with the propagation of 'x'”.^[9]

2.5. Tri-state buffers

```

assign TZ[3:0] = (ENB == 1'b1) ? DB[3:0] : 4'bzzzz;
always @(TZ or DD)
  casex TZ // 'x' and 'z' are don't care
    4'h0: CASEN = DD[0]; // This clause is selected when TZ is 4'bzzzz.
    4'h1: CASEN = DD[1];
    4'h2: CASEN = DD[2];
    4'h4: CASEN = DD[3];
    default: CASEN = 1'bx; // When TZ is 'z', 'x' does not propagate
  endcase

```

Example 2-28 Tri-state output should not be entered in a selection expression of *casez* or *casex* statements

RTqualify checks the following

- | | |
|-----------|---|
| 2511a(N) | A tri-state buffer will be generated. |
| 2511b(W3) | There is a tri-state buffer that is not layered. |
| 2512 (W3) | Tri-state control conditions are not simple. |
| 2514 (W2) | More than <n=5> tri-state buffer outputs connected in a single net. |
| 2515 (E) | There is more than one assignment for a single output using
<i>always constructs</i> or <i>assign statements</i> . |

Verilint Warning

- W490: Tri-state selection signal is not a *variable name*
- W438: Precise examination required: Tri-state is not in a top level
(it is OK that Tri-state is not in a top level module)

2.5.2. Consider high-impedance propagation in tri-state bus

- | | |
|--|-------------|
| [1] Not only a block for a tri-state buffer but also a block for input cell connected directly from bidirectional bus should be made | recommend 2 |
| [2] 'z' becomes 'x' when passing through gates, so circuits that mask using gates may be added as a gate level verification means to prevent 'x' propagation | reference |
| [3] Mask circuit may be optimized after logic synthesis, so 'x' propagation may not be prevented | reference |
| [4] To do RTL simulation, remove bus holder temporarily | reference |

Explanation

Bidirectional bus with tri-state buffer may become 'z' momentarily. When this occurs, the RTL simulation result and the result of simulating the synthesized circuit generated by the logic synthesis tool may differ.

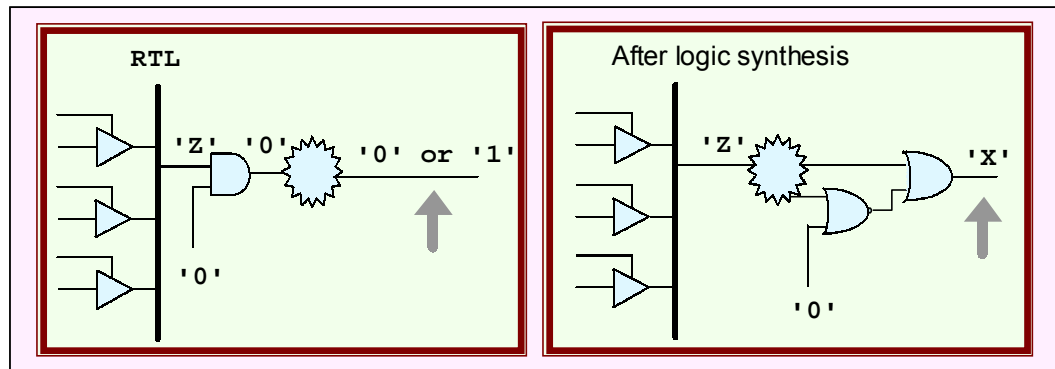


Figure 2-10 X propagation with tri-state buffer

As illustrated in Figure 2-10 for example, a circuit that controls propagation using an AND gate is added to prevent 'z' propagation in the RTL.^[1] As a result of synthesis however, the logic circuit containing the AND gate is optimized, the logic for preventing 'z' propagation changed to OR, and propagation as unknown value 'x' may result without 'z' propagation being prevented.^[3] Since the logic connected to the control signal may be changed as a result of optimization, some problems may occur. Therefore, it will be necessary to make a block for the input description from the bidirectional bus as well as output to the tri-state bus, and to fix these logics.

In reality, bus holder is connected to tri-state bus and the previous value is retained in case of no buffer outputs. However, when executing simulation, remove this bus holder function to check.^[4] If any logic exists in selector input of tri-state buffer, a hazard may occur.

This hazard is not always correctly observed by simulation. Therefore, even if the tri-state bus value is reversed by this hazard, simulation may not be able to detect it.


It is very risky if one designs on the assumption of the value that is held by the bus holder.

2.6. *always construct* description that takes circuit structure into account

2.6.1. Describe taking the circuit structure into account

- | | |
|---|-------------|
| [1] Describe with the structure of the circuit to be generated in mind | recommend 2 |
| [2] Use <i>always constructs</i> as single execution units
(The number of outputs should be limited to 15 at most) | recommend 1 |
| [3] The number of signal output from one <i>always construct</i> should be five or less, if possible | recommend 3 |
| [4] The number of lines in an <i>always construct</i> should be up to 200
- 2000 lines at most (mandatory) | recommend 3 |
| [5] Use intermediate variables when a same logic is used in more than two places | recommend 2 |

Example Code



```

always @(A or B or C or ZIN or YIN) begin
    if(A==1 && B==0) begin
        DOUT = 1;
        EOUT = ZIN;
        FOUT = 0;
        GOUT = 0; end
    else if(A==0 && B==0) begin
        DOUT = 0;
        EOUT = YIN;
        FOUT = 0;
        GOUT = 0; end
    else if(A==0 && B==1) begin
        DOUT = 0;
        EOUT = YIN;
        FOUT = 1;
        GOUT = 0; end
    else if(C==1) begin
        DOUT = 1;
        EOUT = ZIN;
        FOUT = 1;
        GOUT = 1; end
    else begin
        DOUT = 1;
        EOUT = ZIN;
        FOUT = 1;
        GOUT = 0; end
end

always @(A or ZIN or YIN) begin
    if(A==1) begin
        DOUT = 1;
        EOUT = ZIN; end
    else begin
        DOUT = 0;
        EOUT = YIN;
    end
end

always @(B) begin
    if(B==1)
        FOUT = 1;
    else
        FOUT = 0;
end

always @(A or B or C) begin
    if(A==1 && B==1 && C==1)
        GOUT = 1;
    else
        GOUT = 0;
end

```

Example 2-29 Code that is circuit structure oriented

Explanation

Since signals defined in the sensitivity list change, the sequential process statements after the *always construct* are executed and *always constructs* return to the beginning after

2.6. *always construct* description that takes circuit structure into account

the end of the *always construct* is reached and then wait for next signal change.

In the code on the left side of Example 2-29, four output signals (DOUT, EOUT, FOUT, GOUT) are defined for five input signals (A, B, C, ZIN, YIN). Then, assignments are made to all signals according to the conditional expression of each if statement. However, if the assignment conditions to FOUT are associated with signal B for example, FOUT is 1 when B is 1 and FOUT is 0 when B is 0. FOUT is not associated with other input signals.

Associated outputs and inputs should be combined into a single *always construct* with the circuit structure kept in mind, resulting in the code on the right side of the above Example 2-29. Dividing the code makes it easier to comprehend the nature of the input signal and output signal associations.^[1]

One *always construct* should be a single execution unit and should not contain a large number of lines.^[2] The logic size to be generated is not in proportion to the number of lines in *always construct*. However, the number of lines should be 200 or less with the exceptional case of 2000 at most.^[4]

It is not recommended to describe too many output signals in an *always construct*. If possible, five or less is favorable.^[3] However, as explained in “2.3.2.Circuits will vary with non-blocking and blocking assignment”, *always construct* without any logic or signals which generate same logic will not be problematic no matter how many are described in a same *always construct*.

Not describing too many output signals in one *always construct* means that selection condition with exactly the same contents are described in two or more *always constructs*. This logic should be shared using intermediate variable except for a simple one. Especially, when using arithmetic operator with 5 or more bits and relational operator for conditional expressions, logic synthesis tool does not share. If describing in two places, then the area will be double in size. Pay attention to it.

Making codes that consider the circuit structure have the following advantages:^[1]

- * Improved accuracy in estimating the performance of optimized circuits

Logic synthesis tools are not perfect tools. Even if codes were to have identical functions, performance (timing, area) of the optimized circuits varies depending on the code method used. When the structure of the circuit to be optimized is assumed beforehand, describing with the structure of that circuit kept in mind decreases the time required to get a circuit that satisfies the design constraints.

- * Improved debugging efficiency

RTL descriptions must be checked when unintended simulation results are obtained. If the output signal structure of each *always construct* is understood, for example, descriptions that consider the circuit structure facilitate checking of the signal flow by proceeding with the debugging while monitoring it.

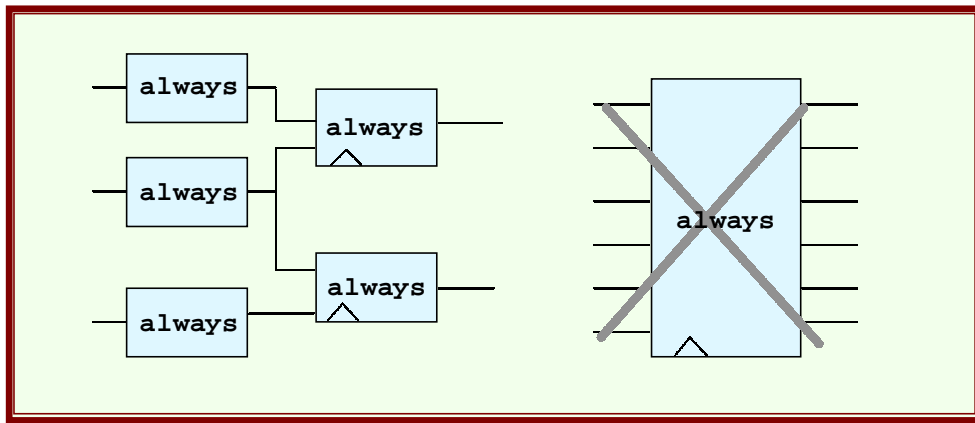
2.6. *always construct* description that takes circuit structure into account

Figure 2-11 Keep the circuit structure in mind

RTqualify checks the following

- | | |
|-----------|--|
| 2613(W3) | Assignments are made to more than <N=5> signals in a single <i>always construct</i> .
(can be set by signal_number_in_an_always=5) |
| 2614a(W3) | There are more than <N=200> lines in an <i>always construct</i> . |
| 2614b(W2) | There are more than <N=800> lines in an <i>always construct</i> . |
| 2614c(E) | There are more than <N=2000> lines in an <i>always construct</i> . (can be set by variable always_linelimit=W3 limitation, W2 limitation and E limitation) |
| 2615(W2) | Same logical expression, arithmetic operations, or relational operations exist in multiple places. |

For the check of intermediate variable with the current version, message is output only in case of arithmetic operation and comparison operators.

2.6.2. Avoid defining multiple output signals in a single *always construct*[1] Do not describe more than one *if* or *case* in one *always construct*

mandatory

[2] Signals assigned in *always construct* should not be described on sensitivity list in the same *always construct*

mandatory

Example Code

```

always @(A or AIN or BIN)
begin
    if(A == 1'b1)
        S_TMP = AIN + BIN;
    else
        S_TMP = AIN - BIN;
    C = S_TMP[15];
    if(S_TMP[14:0] == 15'b0 )
        Z = 1'b1;
    else
        Z = 1'b0;
    if( C == 1 && Z == 1'b0 )
        COUT = 1'b1;
    else
        COUT = 1'b0;
    SOUT = S_TMP[14:0];
end

always @(A or AIN or BIN) begin
    if(A==1'b1)
        S_TMP = AIN + BIN;
    else
        S_TMP = AIN - BIN;
    end
    always @(S_TMP) begin
        if(S_TMP[14:0] == 15'b0)
            Z = 1'b1;
        else
            Z = 1'b0;
        end
    end
    assign C = S_TMP[15];

    always @(C or Z) begin
        if(C==1'b1 && Z==1'b0)
            COUT = 1'b1;
        else
            COUT = 1'b0;
        end
    end
    assign SOUT = S_TMP[14:0];

```

Example 2-30 Description of multiple output signals partitioned by *always construct*

Explanation

The description in Example 2-30 divides complex *always constructs*. The description on the left side defines assignment expressions for four signals (*S_TMP*, *Z*, *COUT*, *SOUT*) as output signals. *if statements* are lined up and use in the sequential block.

As shown above on the right, inputs and outputs are divided into those with high relationships. This division makes the relationships of the input signals and the output signals clear and, additionally, increases the debugging efficiency and makes it more difficult to make mistakes.

Defining multiple *assignment statements* in a single *always construct* may have the following disadvantages.^[1]

Unnecessary priority circuit may be generated

It is sequential process inside *always construct* and logic circuit is generated according to the sequential process. In the description of Example 2-30, the signal assigned at (a) and (b) are used for the conditional expression of *if statement* at (c). Therefore,

2.6. *always construct* description that takes circuit structure into account

(a) and (b) have to be executed before (c). However, it does not matter which one of (a) and (b) is operated first. When describing this kind of parallel operation by sequential process in *always construct*, priority circuit may be generated to maintain sequential process in some cases. This kind of unnecessary priority circuit is eliminated if description size is small, but may not be eliminated when the size is large that circuit operation speed is deteriorated..

Descriptions become complex, debugging efficiency decreases

As descriptions become more complex, readability decreases and more effort is required to understand the specifications. For example, since descriptions that have nothing to do with the sensitivity list like C and Z are also included in the description on the left hand side of Example 2-30, operation becomes complex and the debugging efficiency decreases.

Do not describe signals, which are assigned in *always construct*, in sensitivity list.[2]

```
always @( AIN or BIN or CIN) begin
    tmp = AIN & BIN;
    if(tmp == CIN)
        QOUT = 1'b0;
    else
        QOUT = 1'b1;
end
```

In the above description, tmp is assigned in *always construct* and used in conditional expression of *if statement*. A signal assigned by blocking(=) does not need to be described in sensitivity list because tmp value changes before the *if statement* in next line is executed that there is no problem.

If this *always construct* is assigned by non-blocking(<=), it is hazardous description. In such a case, the value of tmp signal does not change before the *if statement* in next line is executed. In this kind of *always construct*, tmp signal is necessary in sensitivity list.

Even if assigned by blocking(=), it is hazardous description when the *assignment statement* to tmp signal is described in later lines like the following.

```
always @( AIN or BIN or CIN or tmp) begin
    if(tmp == CIN)
        QOUT = 1'b0;
    else
        QOUT = 1'b1;
    tmp = AIN & BIN;
end
```

If tmp signal is not described in sensitivity list, QOUT value becomes not to operate unless AIN, BIN or CIN changes. This is extremely hazardous description since there is a high possibility that tmp signal is not described accidentally in sensitivity list.

Also, even if described correctly, simulation speed is deteriorated since *always construct* is repeatedly executed. Moreover, it may become description where *always construct* is executed repeatedly and unlimitedly that is extremely hazardous.

RTqualify check the followings

- | | |
|----------|--|
| 2621 (E) | There is more than one if statement or case statement in a single <i>always construct</i> . |
| 2622 (E) | A signal to which an assignment is made in an <i>always construct</i> is included in sensitivity list. |

2.7. if statements

2.7.1. if statements create prioritized circuits

- [1] Bring signals with critical timing or signals with many changes to the beginning of the conditional branch
 - Timing acceleration is possible
 - Advantageous in reducing power consumption
- [2] Avoid unnecessary priorities
- [3] *if statement* in combinational circuit ends with *else* (not with *else if*)

recommend 3

recommend 2

recommend 1

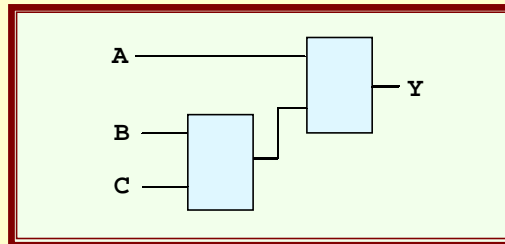
Example Code

```

module DECODE(A, B, C, S1, S2,Y);
input A, B, C, D;
input S1, S2;
output Y;
reg Y;

always @(A or B or C or S1 or S2) begin
  if (S1 == 1'b1)
    Y = A;
  else if (S2 == 1'b1)
    Y = B;
  else
    Y = C;
end
endmodule

```



Example 2-31 Selector description using *always construct* and *if statement*

if statements defines branch according to *conditional expression*. *conditional expression* of *if statements* evaluate according to the sequence, but if either of the conditional expressions is True, then the statement that corresponds to that *if statement* is executed and subsequent evaluations not executed.

There is no need to be defined as *block statements* if each statement is just one statement. When executing multiple statements, it is necessary to define description blocks by begin-end pair. *else items* can handle default conditions that could not satisfy any *conditional expression* of *if statements* at all and can also be omitted.

The Example 2-31 shows a selector description that uses *if statements*. In this example, first S1 is evaluated, and then S2 is evaluated. As a result, a value will be assigned to Y. The first conditional branch of an *if statement* will be placed in a position close to the output. Therefore, it is best to bring signals with critical timing constraints to the first conditional expression of *if statement* as often as possible.^[1]

Also, the conditional branches of the *if statement* are compared one by one until the result is TRUE, so if it is possible to change the branching sequence of signals with many changes, bringing them to the beginning of the *if statements* would be advantageous with respect to the simulation execution speed and the power consumption of circuit after logic synthesis.

Description of *if statements* adds priorities. Too many nesting levels lead to speed decrease. Be careful not to add unnecessary priorities^[2], and use logic expressions such as '&' and '|' where you don't need to describe priorities. However, logic expressions that includes complex use of '&' and '|', instead of simple logic of just '&' or '|', will decrease its readability. Use *case statements* for such complex logics. When using *case statements*, refer to the notes described in “2.8 *case statements*”.

When describing combinational circuit using *always* construct, *else* item should be used at the end of *if* statement to avoid generation of erroneous latch.^[3]

RTqualify checks the following


2713(W3) There is no *else* statement in an *if* statement described in an *always* construct for a combinational circuit.

if statement and *case statement* are explained in RMM:5.5.7.

2.7.2. Reduce *conditional expression* of *if statement* with the same contents[1] Reduce *conditional expression* of *if statement* with the same contents

recommend 1

Example Code



```

always @(A or B or C )
  if( A==1'b0 && B== 1'b0 && C== 1'b0 )
    Y = 3'b010;
  else if(A==1'b0 && B==1'b0 && C==1'b1)
    Y = 3'b101;
  else if(A==1'b0 && B==1'b1 && C==1'b0)
    Y = 3'b100;
  else if(A==1'b0 && B==1'b1 && C==1'b1)
    Y = 3'b000;
  else if(A==1'b1 && B==1'b0 && C==1'b0)
    Y = 3'b001;
  else if(A==1'b1 && B==1'b0 && C==1'b1)
    Y = 3'b110;
  else if(A==1'b1 && B==1'b1 && C==1'b0)
    Y = 3'b111;
  else
    Y = 3'b011;

always @(A or B or C)
  if(A== 1'b0 )
    if(B== 1'b0 )
      if(C== 1'b0 )
        Y = 3'b010;
      else
        Y = 3'b101;
    else
      if(C== 1'b0 )
        Y = 3'b100;
      else
        Y = 3'b000;
  else
    if(B==1'b0)
      if(C==1'b0)
        Y = 3'b001;
      else
        Y = 3'b110;
    else
      if(C==1'b0)
        Y = 3'b111;
      else
        Y = 3'b011;

```

Example 2-32 Reducing the description by removing the same *conditional expression*

Explanation

Conditional expressions with the same contents generate compare circuits for each conditional expression. They are optimized through structuring process of logic synthesis, but since the size of the first synthesized circuits becomes quite large, performance may get worse. Avoid as much as possible conditional expressions with the same contents.[1]

In the description on the left side of the Example 2-32, conditional expressions for signals A, B, and C are written in parallel within the *if statement* conditional expressions. *if statements* are synthesized as circuits that have priorities, so circuits that compare signals in order from the first conditional expression are generated.

The description on the right side uses *if statements* to clearly designate comparisons of signals that have priorities. There are seven levels of prioritized logics in *if statement* on the left side but only three levels on the right side.

Therefore, from the standpoint of the number of conditional expressions with the same contents and the number of *if statement* levels, the description on the right side is better than the left side.

In reality, conditional expressions are not overlapped in the description on the left side and processed in parallel so that the logic gates generated by Design Compiler will be same on the right and the left. However, if the number of *if statement* nesting becomes large and

describing with values overlapped, the speed slows as priority logic is generated.

For a safe coding, it should be noted that one if-else if nesting and if-if nesting generate one prioritized logic, and pay attention to avoid the description that has many nesting levels.

The description on the left side of the Example 2-32 may seem to have better readability compared to the one on the right. If describing in the format where values are aligned as the description on the left, description by *case statement* to avoid prioritized logic. *if statement* is to describe priority and *case statement* is to describe parallel operation.

2.7.3. Decrease the number of *if statement* nests

- | | |
|---|-------------|
| [1] The number of nest for <i>if-if</i> and <i>else if</i> is best at 5 or less | recommend 3 |
| [2] Associating <i>if statements</i> that have deep nesting with <i>else items</i> is difficult | reference |
| [3] The number of nest for <i>if-if</i> and <i>else if</i> should be 10 at most | recommend 1 |
| [4] Unify <i>if statements</i> that are mergeable | recommend 3 |
| [5] Use of tabs (indenting) will reduce mistakes in <i>if statement</i> nesting | reference |

Example Code

```

always @( posedge CK or negedge RST_X )
  if( !RST_X )
    Y <= 3'b000;
  else if( A==1'b0 ) begin
    if( B==1'b1 ) begin
      if( C==2'b01 )
        Y <= DIN;
      end
    end
  else
    Y <= 3'b010;

```

↓

```

always @( posedge CK or negedge RST_X )
  if( !RST_X )
    Y <= 3'b000;
  else if( A==1'b0 ) begin
    if( B==1'b1 && C==2'b01 )
      Y <= DIN;
    end
  else
    Y <= 3'b010;

```

Example 2-33 Deeply nested *if statement* description

Explanation

It is difficult for *if statements* with deep nesting to precisely comprehend associations with *else items*.^[2] Generally speaking, indents are used to clarify associations with *if statements* and *else items* when coding.^[5] However, the number of indents increases with *if statements* that have deep nesting.

The above Example 2-33 shows the conditional expressions for two *if statements*, $B==1'b1$ and $C==2'b01$, being combined into one. In this case, although the number of items in the *conditional expression* of *if statement* increases, the number of nests for *if statement* becomes one less which brings about a minor improvement in the comprehension and readability.^[4]

Likewise, *if statement* of $A==1'b0$ can also be merged. However, this *if statement* includes *else item*, which results in the number of nests not changing. Also as a result of merge there are two *else if* created ((*else if*($A==1'b0 \ \&\& \ B==1'b1 \ \&\& \ C==2'b01$) and *else*

`if(A==1'b1))`, and there are same selector as the conditional expression of the previous *if statement*, description becomes rather complicated. Therefore, in Example 2-33, it is recommended not to merge *if statement* of `A==1'b0`.

Besides the case in the Example 2-33, when *if statement* exists in *else item* the number of nests can be decreased by coding as *else if item* instead of *else item*. When considering the performance of the generated circuits, the fewer *if statement* nests there are the better. However, there is no problem if the items in the conditional expression of *if statement* are a little more numerous.

Within conditional expression of *if statement*, combinational circuit is predictable. As the number of nests in *if statement* increases, prioritized logic is added to this circuit.

The prioritized logic structure cannot be specified by the RTL description but is automatically generated by logic synthesis. Therefore, when *if statement* is too deep, prioritized logic becomes complicated and circuit size may become large.

The criterion number of nests is 5 times except for *if statement* of clock event. The number of nest should be 10 at most.^{[1] [3]}

RTqualify checks the following items.

- 2731a(W3) *if statements or case statements are nested more than <N=5> deep.*
- 2733a(W2) *if or case statement nests is more than <N=10>.*
- 2733a(W2) *if statements or case statements are nested more than <N=10> deep.*
- 2733b(E) *if statements or case statements are nested more than <N=30> deep.*
- 2734 (W3) *There are if statements that can be merged.*

If statement and case statement are explained in RMM:5.5.7.

2.7.4. Always surround multiple statements using *block statements* (begin-end) (Verilog only)

[1] It is not necessary to surround a single statement by *block statements* (Verilog only)

reference

[2] Be sure to remember to attach begin-end (Verilog-only)

reference

[3] Do not use fork-join in RTL descriptions (Verilog only)

mandatory

Example Code

Example code	Simulator interpretation
<pre> always @(A or B or C or S) begin if (S==1'b0) Y = 1'b0; else if (A==1'b1) Y = 1'b1; if (B==1'b1) Z = 1'b0; else begin Y = C; Z = 1'b1; end end </pre>	<pre> always @(A or B or C or S) begin if (S==1'b0) Y = 1'b0; else if (A==1'b1) Y = 1'b1; if (B==1'b1) Z = 1'b0; else begin Y = C; Z = 1'b1; end end </pre>

Block statement was not described

Example 2-31 *Block statement that was not written*

Explanation

In the case of Verilog-HDL, multiple statements are combined using *block statements*. *Block statements* consist of statements such as sequential blocks by begin-end and parallel blocks by fork-join. Logic synthesis tools only support sequential blocks, so use begin-end.^[3]

If there is one statement for each *if statement* conditional branch, *block statements* are not necessary, but *block statements* must always be used when defining multiple statements.^[1] If the *block statements* are mistakenly left out, the *if-else* associations that the designer intended and the *if-else* associations interpreted by the simulator may differ.^[2] In the Example 2-34, since the *block statement* definition was forgotten, even though the second *if statement* was intended to be nested in the first *if statement*, the simulator interpreted it as a separate *if statement*. It is important to note that no warning or error message in particular is displayed in such descriptions.

The coding amount will increase in order to avoid such mistakes, but one way to resolve this is to be sure that statements are surrounded by begin-end even if the statement is just one.

RTqualify checks the following items

2741a and 2741b are not to be checked by default setting

2742(W3) Use of else in an *if statement* might be incorrect. Enclose between begin and end lines.

2743(E) fork-join cannot be used in RTL description.

Verilint Warning

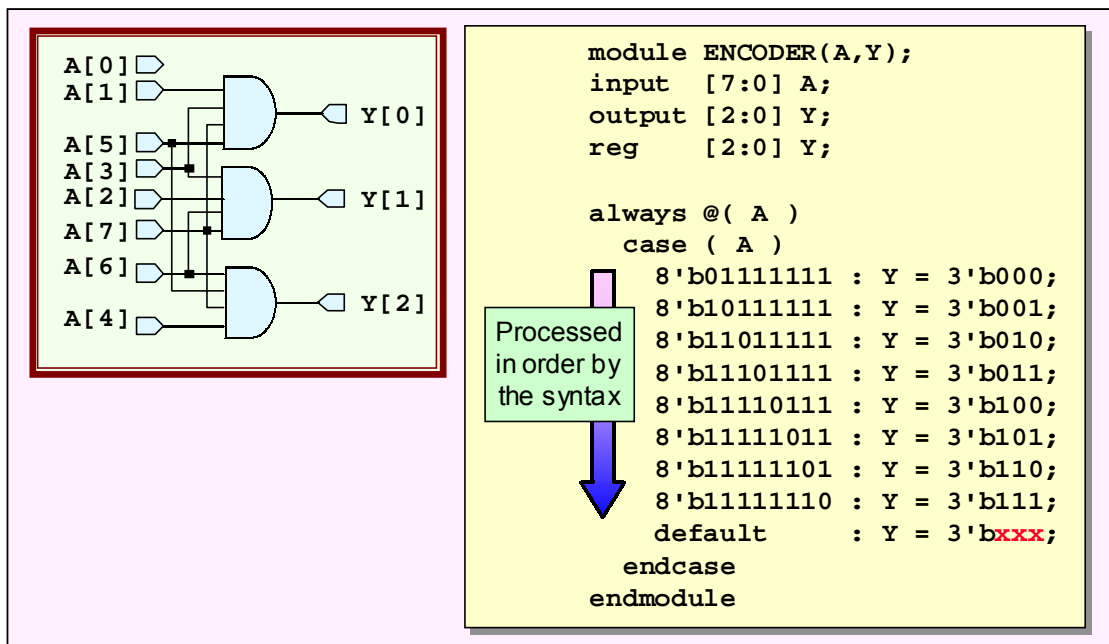
W527: Dangling else. -> use begin-end

2.8. case statements

2.8.1. case statements facilitate decoder/encoder description

- | | |
|--|-------------|
| [1] <i>case statements</i> are suitable for describing decoders/encoders with many branches | reference |
| [2] Described properly, <i>case statements</i> create faster circuits than <i>if statements</i> do | reference |
| [3] Avoid the overlapping of <i>case items</i> | recommend 1 |
| [4] Always add <i>default clauses</i> | recommend 1 |
| [5] Do not force <code>full_case</code> for <i>case statement</i> directives that depend on a particular logic synthesis tool (Verilog only) | mandatory |
| [6] Pay attention to the selective range of <i>case statement</i> and bit width of each item (Verilog only) | recommend 2 |

Example Code



Example 2-35 Multiplexer description using *case statement*

Explanation

Using *case statements* makes it possible to easily describe many combinational circuits such as decoders/encoders and multiplexers.^[1] It is possible to describe with *case clause* overlapped. However, when describing *case clauses*, it is recommended to describe the circuits, which operate in parallel. The description, in which values overlap, with priority should be described in *if statement* and consider using *if statement* and *case statement* properly depending on the situation.

Refer to “2.7.2. Reduce conditional expression of if statement with the same contents” and “2.7.3. Decrease the number of *if statement* nests” for *if statement* description.

case statements compare clause values in the order described in the clause. If one selection expression matches a selection expression in parentheses, the statement that corresponds to the clause is executed. The *default clause* is executed if all comparisons fail, so make sure that *default clause* is defined otherwise nothing will be executed.

case syntax consists of the following statements: *case statement*, *casex statement*, *casez statement*. The casex statement is used to recognize unknown value 'x' and high-impedance as don't care. The casez statement is used to recognize only high-impedance 'z' as don't care.

Example 2-35 shows an example *case statement* description. First, signal A, which judges selections, is defined in parentheses of the selection expression of case statements. Then, an 8-bit value is defined as an item, so it will have the same bit width as 8-bit signal A. Next, the item is compared from the top.

If it matches, then that branch is selected and an assignment is made to signal Y. If there is no matching item, then the *default clause* is executed and unknown value 'x' is assigned to Y in the example description.

Design Compiler has a directive called `//synopsys full_case`. If specifying this in *case statement*, in which not all the selection expressions are described, and not using *default clause*, it is assumed that all the selection expressions are described. However, when this directive is used, simulator result of a case not described will differ in the RTL and gate level. Therefore, `full_case` should never be used.^[5] Refer to “2.8.3. Use *default clause*”.

Since the *case statement* compares from the top, overlapping values are permitted. The first selection expression is executed first when there are overlapping value of clauses.

If there are duplications in the *case statements*, the logic synthesis tool synthesizes priority circuits with a similar priority as the *if statement*, but if there are no overlaps in the branch conditions, then a circuit that compares values in parallel is synthesized. Logic synthesis tools automatically judge branch overlaps, but it is also possible to clearly define that there are no overlaps of compile directive as the comment `//synopsys parallel_case`.

case statements are sequentially processed in Verilog-HDL syntax, but we recommend describing so that values do not overlap.^[3] If a limitation is to be observed so that the *case statement* value of clauses do not overlap, defining the comment `//synopsys parallel_case` may be a little advantageous with respect to speed and area. This is because the Design Compiler may not properly recognize `parallel_case`; there are cases that may be advantageous to the speed and area.

However, if using `parallel_case` when values overlap, the RTL simulation result and simulation result of logic gates generated by logic synthesis will differ. Therefore, it should never be done. Refer to “2.8.5. Description relying on `parallel_case` is prohibited”.

In the syntax of Verilog-HDL, error does not occur even if some items exceed the range of signal specified in the selection expression of *case statement*. In addition, even if signal specified in the selection expression of *case statement* and bit width of each time differ, error does not occur. This line is ignored by simulation and logic synthesis tool. Pay attention that description is not erroneous.^[6]

RTqualify checks the following items:

- | | |
|-----------|--|
| 2813 (W2) | Redundancy in branches of <i>case statement</i> . |
| 2814a(E) | There is no <i>default</i> in this <i>case statement</i> , and not all of possible values that can be assumed from <i>selection expression</i> are described.
(<i>always construct</i> of combinational logic) |
| 2814b(W1) | There is no <i>default</i> in this <i>case statement</i> , and not all of possible values that can be assumed from <i>selection expression</i> are described.
(<i>always construct</i> of combinational logic with initial value assignment) |
| 2814c(W3) | There is no <i>default</i> in this <i>case statement</i> , and not all of possible values that can be assumed from <i>selection expression</i> are described.
(<i>always construct</i> to infer FF) |
| 2816a(E) | <i>Items</i> for comparison in <i>case statement</i> exceed range of <i>selection expression</i> . |
| 2816b(W1) | Bit width of <i>comparison items</i> in <i>case statement</i> does not match bit width of <i>selection expressions</i> . |

2.8.1.[1] and 2.8.1.[2] cannot be checked. 2.8.1.[5] is checked by 2833.

Verilint Warning

W398 : *case* covered more than once or overlapped

W264 : Not all possible cases covered

W332 : Precise examination required : Not all possible cases covered, but *default* case exists

W69 : *case statement* without *default clause*, but all

if statement and *case statement* are explained in RMM : 5.5.7.

2.8.2. Divide using *if statement*, etc. to avoid creating large tables

- [1] Division criterion is within 24 I/O (input: 16, output: 8)
- [2] Item count should be within 100
- [3] Do not put the truth table in *case statements* as is
 - Create truth table that takes the HDL description into account
 - Divide in advance *case statements* that are easily divisible

recommend 3

recommend 3

reference

Example Code

```

case ( A )
32'b00000001000000001111011011100110: DOUT = 5'b00000;
32'b00000001000000001111011111110110: DOUT = 5'b00010;
32'b00000001000000001101011011100110: DOUT = 5'b00011;
32'b00000001000000001111011011110110: DOUT = 5'b00100;
32'b00000010000000001110011011100111: DOUT = 5'b00101;
32'b00000010000000001111011011100110: DOUT = 5'b00110;

```



```

if(A[25:24]==2'b01) begin case ( A[18:0] ) .....
else if(A[25:24]==2'b01) begin case ( A[18:0] ) .....

```

Example 2-36 *case statement* simplification

Explanation

When a large table is created as a branch condition of a *case statement*, the logic synthesis tool will create circuits that examine whether it matches with each defined branch condition. Therefore, specifying branch conditions with large bit width will cause the synthesis time to increase and may further worsen the synthesis results.

It is recommended that the *case statement* have up to 24 bits including input and output.^[1] The size of a *case statement* table is determined by the number of inputs. However, output should still be considered. As the number of output increases, if every logic is not so similar, area may be increased as much as creating multiple tables.

In the case of a small table, which has five or fewer clauses or four or fewer inputs, the output number is not necessarily considered but for other cases, attention should be paid to the number of outputs.

The number of clauses of a *case statement* should not exceed 100.^[2] However, in the case of a table, where randomness is high such as SIN function, dividing a table, for example with 8 bits of both input and output and 256 clauses by *if statement* will result only in the decrease of description readability so there is no merit in circuit generation. In this case, *case statement* with 256 clauses has to be created, but the number of clauses should not exceed 300.

2.8. case statements

With the current logic synthesis tool capacity, area and speed increase at exponential rates when exceeding about 200 clauses.

If common values are used for branch conditions, the common values should be first compared by *if statement* and then branched off by *case statement*.^[3] Doing this makes it possible to shorten the amount of time required for synthesis and improves circuit performance.

In the above Example 2-36, the upper bit value is compared by an *if statement*, then the lower bit value is branched by *case statements*. This makes it possible to reduce the size of the initial circuit structure at logic synthesis.

RTqualify checks the following

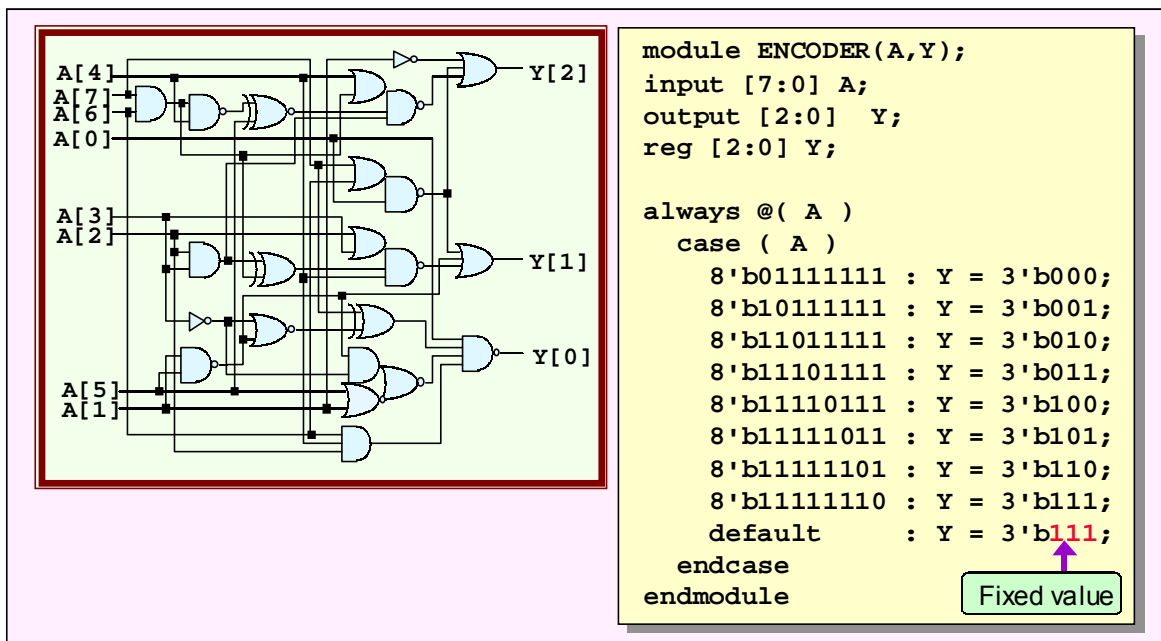
- | | |
|-----------|--|
| 2821a(W3) | Bit widths of inputs and outputs for case statement total more than <N=24> bits. |
| 2821b(W3) | There are more than <N=100> items for comparison in case statement. |
| 2821c(W2) | More than <N=300> items for comparison in case statement. |
| 2821d(E) | More than <N=100> items for comparison in case statement. |

RTqualify cannot check “2.8.2.[3] Do not put the truth table in *case statements* as is”.

2.8.3. Use *default clauses*

- | | |
|--|-------------|
| [1] The <i>don't-care</i> condition is defined by using 'x' as the <i>default clauses</i> (only for <i>default clauses</i> , the extensive use of don't care is recommended) | recommend 3 |
| [2] When 'x' is used as <i>default clauses</i> , 'x' is propagated by the simulation even if 'x' is not <i>don't-care</i> | reference |
| [3] Can suppress latch generation (Verilog only)
- Use <i>default clause</i> , do not use //synopsys full_case | reference |
| [4] Do not use the signal to which a <i>don't-care</i> condition is assigned for a conditional expression of <i>if statement</i> | recommend 1 |
| [5] Describe a <i>default clause</i> at the end of a <i>case statement</i> (Verilog only) | mandatory |
| [6] Do not use the signal to which <i>don't-care</i> condition is assigned for selection expression of <i>case statement</i> that do not assign 'x' in default clause | recommend 1 |
| [7] Do not use the signal, to which <i>don't-care</i> condition is assigned, for selection expression of <i>casex statement</i> | recommend 1 |

Example Code

Example 2-37 Description of *case statement* (unlike the Example 2-35, default clause is a fixed value)

Explanation

The *default clause* is executed if the selection expression specified by the *case statement* does not match. The Example 2-37 illustrates an example in which a fixed value is specified to the default.

If the unknown value 'x' is specified to the output of the *default clause*, it is regarded as don't care and non-determined value is output due to the optimization result. Setting a non-determined value can usually decrease the size of the circuit more than setting a fixed value in the *default clause* does.^[1]

2.8. case statements

If there is no *default clause* and a value other than a branch condition appears, a latch is generated to retain the output value. However, this latch is not generated if all branch conditional expression values are covered.^[3]

If there is no *default clause* and there is no need to generate a latch, it is possible to synthesize using //synopsys full_case to direct that all branch conditions are defined. In this case, the output signal is retained since the *default clause* is not included in the RTL description, but it becomes *don't care* at gate levels and what kind of value will be output is unknown. It will result in the difference between the RTL simulation and gate level simulation. Therefore, the use of full_case is prohibited. Refer to “2.8.1.case statements facilitate decoder/encoder description” for full_case.

Description that uses *don't care 'x'* in *default clauses* may generate different results in RTL and gate-level simulations.^[2] However, for *case statements*, gate count may increase significantly and the quality will be low if *don't care* is not used in *default clauses*. When a *don't care 'x'* is used in a *default clause* of a *case statement*, consider that line as something that should not be executed in simulation. Normally, when unknowns are generated because of *don't care 'x'* in RTL, it is treated as a bug. So, the code should be corrected to avoid this kind of description.

Depending on how they are described, *case statements* may increase the circuit size. Avoid unnecessary use of *case statements*, and instead use *if statements* for priority logic, or use '&' or '|' to express parallel logic. Use *case statements* only when it is not possible to describe a logic with a simple '&' or '|' Keeping that in mind, *case statements* with fixed value in *default clause* will be less necessary.

It is necessary to use *default clause* in *case statement* in *always construct* of combinational circuit. However, in the *always construct* to generate FF, *default clause* is not necessarily used. In the *always construct* to generate FF, the previous status of undescribed conditions are retained so that FF values one clock before are retained. However, as the description practice, it is better to describe explicitly as follows:

```
default : DOUT <= DOUT;
```

Signals to which 'x' of *don't care* is assigned in *default clause* in *case statement* should not be used in conditional expression of *if statement*.^[4]

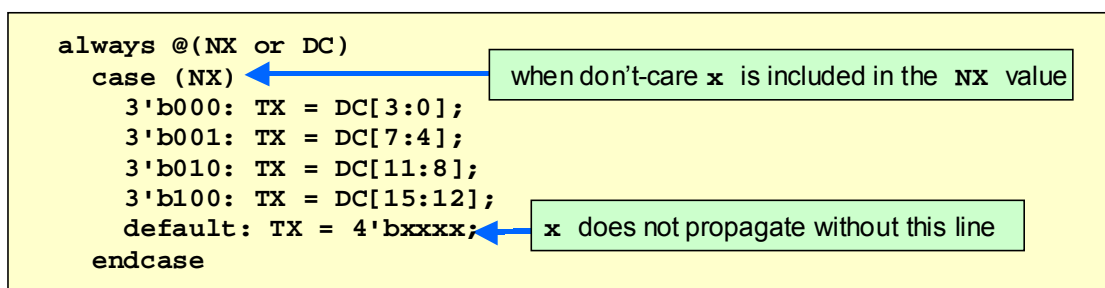
In the above case, if the *default clause* is executed, *don't care 'x'* is assigned to Y. If this Y

```
always @(A)
  case (Y)
    ...
    default: Y = 4'bxxxx;
  endcase
always @(posedge CLK or negedge RST_X)
  if(!RST_X)
    DOUT <= #DLY 1'b0;
  else if(Y== 4'b0000)
    DOUT <= #DLY 1'b1;
  else
    DOUT <= #DLY 1'b0;
```

signal is used for a conditional expression of *if statement* in the following stage, an *else item* is executed and DOUT becomes '0'. After the logic synthesis of *don't care 'x'* it is unknown whether it becomes '0' or '1'.

If all the bits become '0', DOUT becomes '1' that simulation result would differ in RTL and gate levels. Please refer to "4.4.2. Inconsistencies can occur in RTL and gate level with the propagation of X". Normally, the line, to which *don't care 'x'* is assigned, is not executed in RTL simulation. If done, it is a bug. However, there is a chance to overlook erroneous execution.

Do not use signals, to which don't care conditions are assigned, for selection expressions. Because 'x' becomes not to propagate and simulation results may differ between RTL and gate level.^[6] The signal, to which don't care condition is assigned, should not be used for the selection expression of *casex statement* even if 'x' is assigned in default clause.^[7] There is no problem using for *casez statement* as 'x' propagates. The difference between *casex statement* and *casez statement* is explained in "4.4.2 Inconsistencies can occur in RTL and gate level with the propagation of X".



Example 2-38 Do not use signals with *don't-care* conditions assigned for selection expressions of *case statements* without 'x' in default

RTqualify performs the following checks

- | | |
|-----------|---|
| 2831a(W3) | In this case statement, even though all values that can be assumed from <i>selection expression</i> are described, a fixed value is assigned by default. |
| 2831b(W3) | In this case statement, not all of values that can be assumed by <i>selection formula</i> are covered, and a default value is provided. Additionally, value assigned by default is a fixed value. |
| 2832 (W3) | In this case statement, all of values that can be assumed by <i>selection formula</i> are covered, and no default value is provided. |
| 2833 (E) | In this case statement, <code>synopsys full_case</code> is specified even though not all of values that can be assumed by <i>selection formula</i> are covered. |
| 2834 (W3) | A conditional <i>formula</i> uses a signal for which x is assigned in another statement such as a case statement. |
| 2835 (W1) | default is found other than at end of a <i>case statement</i> . |

Verilint Warning

W264 : Not all possible cases covered

W332 : Not all possible cases covered, but default case exists

W551 : Both of full_case and default specified.

W69 : *Case statement* without *default clause*, but all

W71 : *Case statement* without *default clause*

W187 : *Default clause* is not the last clause in a case

W453 : Case too wide (8 or more bits) to be checked if all cases are

2.8.4. Do not use complex *casex statements* (Verilog only)

[1] Complex <i>casex statements</i> decrease circuit quality (Verilog only)	reference
[2] Readability declines with complex <i>casex statements</i> (Verilog only)	reference
[3] It is best to avoid using <i>casex statements</i> and <i>casez statements</i> (Verilog only)	recommend 3
[4] Do not indiscriminately describe 'x' of <i>casex statement</i> for each bit (Verilog only)	recommend 1

Example Code

```

module PENCODE2 ( A , Y );
input [7:0] A;
output [2:0] Y;
reg [2:0] Y;

always @( A )
  casex ( A )
    8'bxxxxxxx0 : Y = 3'b111;
    8'bxxxxxxx01 : Y = 3'b110;
    8'bxxxxxx011 : Y = 3'b101;
    8'bxxxxx0111 : Y = 3'b100;
    8'bxxxx01111 : Y = 3'b011;
    8'bxxx011111 : Y = 3'b010;
    8'bx0111111 : Y = 3'b001;
    default : Y = 3'b000;
  endcase

endmodule

```

don't care condition described with x and z

Example 2-39 Better description that uses *casex* (prioritized logic)

Explanation

The description of Example 2-39 is an example of a priority encoder that uses *casex statement*.

With *casex statement* or *casez statement*, the value indicated by 'x' of the branch conditions is defined as *don't-care*. *don't-care* indicates that it does not matter whether the value is '1' or '0'. Since only the least significant bit of the first value in the Example 2-39 is '0', if the least significant bit is '0', the first branch condition will be executed regardless of the other bit values.


```

input[3:0] A,B,C;
wire[11:0] D;
reg [3:0] Y;
assign D = {A,B,C};
always @( posedge CLK )
  casex ( D )
    12'b0001xxxxxxxx: Y <= 4'b0001;
    12'b0010xxxx0101: Y <= 4'b0010;
    12'b0010xxxx0100: Y <= 4'b0010;
    12'b0010xxxx0110: Y <= 4'b0010;
    12'b0010xxxx0111: Y <= 4'b0010;
    12'b01000000xxxx: Y <= 4'b0011;
    12'b01001111xxxx: Y <= 4'b0100;
    12'b0100xxxxxxxx: Y <= 4'b0101;
    12'b1000011x0000: Y <= 4'b0110;
    12'b10000100xxxx: Y <= 4'b0111;
    12'b10001000xxxx: Y <= 4'b0111;
    12'bxxxxxxxx1010: Y <= 4'b1001;
    12'bxxxxxxxx1000: Y <= 4'b1010;
    12'bxxxxxxxx0000: Y <= 4'b1100;

```

Example 2-40 Description using *casex* (bad example)

When using *casex statement*, describe so the *don't-care* 'x' specification range has a distinct range as mentioned in Example 2-39. Descriptions in which the *don't-care* condition differs for each bit invite the duplication of clause and risks worsening the circuit quality.^{[1] [2] [4]} Example 2-40 is a bad example of using *casex* statement.

RTqualify checks the following

- 2841a(W3) A don't care input is used in a *casex* statement or a *casez* statement.
- 2841b(W1) *don't-care* inputs are used in a variety of different positions.

2.8.5. Description relying on parallel_case is prohibited (Verilog only)

- | | |
|---|-------------|
| [1] Do not force parallel_case in a <i>case statement</i> directives that depends on a particular logic synthesis tool (Verilog only) | recommend 1 |
| [2] Do not describe fixed values in the selection expression of <i>case statement</i> (Verilog only) | recommend 1 |
| [3] Do not describe variables (or the expression: a+b) in the clause of <i>case statement</i> (Verilog only) | mandatory |
| [4] Do not describe logical operations and arithmetic operations in the selection expression of <i>case statement</i> (Verilog only) | recommend 2 |

Example Code

```

module SEL9C(A,B,Z);
input [8:0] A,B;
output Z;
reg Z;
always @( A or B )
  casex ( A ) // synopsys parallel_case
    9'bxxxxxxxxl : Z = B[0];
    9'bxxxxxxxxlx : Z = B[1];
    9'bxxxxxxxxlxx : Z = B[2];
    9'bxxxxxxxxlxxx : Z = B[3];
    9'bxxxxxlxxxx : Z = B[4];
    9'bxxxclxxxxx : Z = B[5];
    9'bxclxxxxxxx : Z = B[6];
    9'bxlxxxxxxx : Z = B[7];
    9'blxxxxxxx : Z = B[8];
    default : Z = 1'bx;
  endcase
endmodule

module SEL9D(A,B,Z);
input [8:0] A,B;
output Z;
reg Z;
always @( A or B )
  casex (1'b1) // synopsys parallel_case
    A[0] : Z = B[0];
    A[1] : Z = B[1];
    A[2] : Z = B[2];
    A[3] : Z = B[3];
    A[4] : Z = B[4];
    A[5] : Z = B[5];
    A[6] : Z = B[6];
    A[7] : Z = B[7];
    A[8] : Z = B[8];
    default:Z = 1'bx;
  endcase
endmodule

module SEL9A(A,B,Z);
input [8:0] A,B;
output Z;
reg Z;
always @( A or B )
  case ( A )
    9'b000000001 : Z = B[0];
    9'b000000010 : Z = B[1];
    9'b000000100 : Z = B[2];
    9'b000001000 : Z = B[3];
    9'b000010000 : Z = B[4];
    9'b000100000 : Z = B[5];
    9'b001000000 : Z = B[6];
    9'b010000000 : Z = B[7];
    9'b100000000 : Z = B[8];
    default : Z = 1'bx;
  endcase
endmodule

module SEL9B(A,B,Z);
input [8:0] A,B;
output Z;
reg Z;
always @( A or B )
  if(A[0]) Z = B[0];
  else if(A[1]) Z = B[1];
  else if(A[2]) Z = B[2];
  else if(A[3]) Z = B[3];
  else if(A[4]) Z = B[4];
  else if(A[5]) Z = B[5];
  else if(A[6]) Z = B[6];
  else if(A[7]) Z = B[7];
  else Z = B[8];
endmodule

```

Example 2-41 Order description forcibly paralleled by //synopsys parallel_case

Explanation

In Verilog-HDL syntax, *case statements* are processed in order from the top line by a sequential process. If there are no overlaps in the clause described in the *case statement*, the logic synthesis tool interprets the values to be parallel and generates a circuit with no priority. If there is duplicated value of clauses, the values are interpreted as a sequential process and a circuit with a priority is generated. However, in the sense of comprehensive descriptions, consider it a good idea to distinguish parallel processes for *case statements*,

2.8. case statements

and sequential processes for *if statements*.

It is easy to make mistakes when describing *sequential statements* by *case statements*, so be careful to describe without having any overlapping values.

There is a directive in the Design Compiler called `//synopsys parallel_case`. If this directive is specified, values will be forcibly treated as parallel case even if there are overlapping values. Therefore, there is a possibility of the RTL simulation results and Gate simulation results differing.^[1]

For example, in the top part of the example (SEL9C, SEL9D), if bit3 and bit0 of A are '1' the top line is executed in the RTL simulation, becoming `Z=B[0];`. However, the logic synthesis tool judges this to be parallel case, so it judges that the `Z=B[0];` line and `Z=B[3];` line were executed simultaneously and generates a circuit that outputs `B[0] | B[3]` logic.

With large designs, there is usually not enough time to execute gate simulation using all test patterns. In such a situation, there is a possibility of the RTL simulation results not matching the gate simulation results that there will be no choice but to run the Gate simulation.

The possibility of the results differing from those of the Gate simulation must be removed from the RTL description.

In the lower part of the above example (SEL9A), if bit3 and bit0 of A are '1' the default clause is executed and the value becomes `Z='x'` in the case of the RTL simulation.

The logic synthesis tool judges this to be *don't-care* and generates a circuit, and then the Gate simulation results will become either '0' or '1'. Since the results at least will not become 'x', the RTL simulation results and gate simulation results will differ. However, if the RTL simulation results are 'x', the expected value will become 'x', so there will be no mismatch problem. When the RTL simulation results become 'x', the selection expression that causes this 'x' result should generally be removed from the circuit.

If the circuit in the upper part of the example 2-41 is to be operated in parallel, *case statement* in the lower left part(SEL9A) should be used, if to be operated in prioritized, *if statement* in the lower right part(SEL9B) should be used as described.

RTqualify checks the following

- | | |
|-----------|---|
| 2851 (E) | Although there are redundant comparison conditions in a <i>case statement</i> , <code>synopsys parallel_case</code> is specified. |
| 2853 (W1) | A constant is specified in a <i>case statement</i> selection expression. |
| 2854 (E) | A signal is specified in a <i>case statement</i> comparator. |
| 2855 (W2) | Logical operator or arithmetic operator in <i>case statement selection expression</i> . |

Whether same values used (regardless of `parallel_case`) in *case statement* is checked by 2.8.1.3.

Verilint Warning

W398 : Case covered more than once

W226 : Case-select expression is constant

W225 : Case expression is not constant

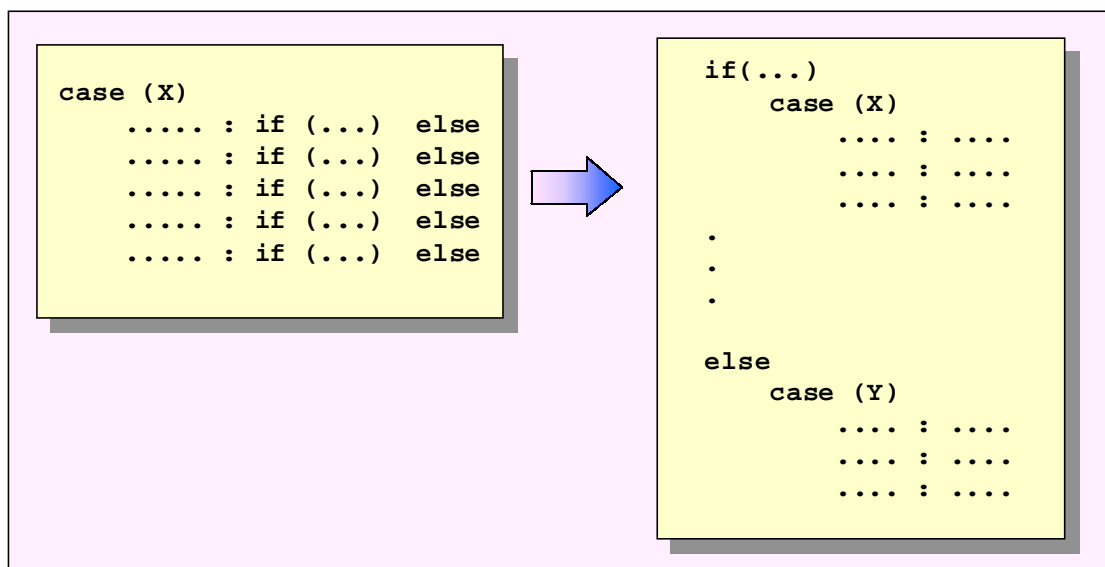
2.8.6. Beware of nesting that *if statements* and *case statements* coexist (2.8.4 in the VHDL version)

- [1] In the case of nesting that *if statements* and *case statements* coexist, it is better to have a large table in a small condition than to have a small condition in a large table

recommend 3

Explanation

With nesting where *if statements* and *case statements* coexist, it is more advantageous to have large tables in small conditions than it is to have small conditions in large tables. Before starting the description, try to describe using this structure if possible. However, if having a large table in a small condition increases the description amount and the number of input, circuit performance will be poor. Keep circuit structure in mind for description.



Example 2-42 Nesting style of *if statements* and *case statements*

Figure 2-12 describes a circuit with a RD (read signal), WR (write signal) and ADR (address signal). As you can see, comparing the circuit on the right hand and left hand, the circuit on the right hand, where large tables exist in small conditions, is better. Designers would not employ the circuit on the left hand when designing on circuit drawing. However, when designing by the RTL description, they often describe the circuit on the left hand.

Even though making the RTL description as on the left hand of Figure 2-12 when generating a circuit, logic synthesis tool may output the circuit on the right hand of Figure 2-12 at circuit generation considering the order of assignment. However, it depends on the performance of the logic synthesis tool that the circuit on the right hand is not always output.

When the circuit size is simple like Figure 2-12, there is high possibility that it is changed to the circuit on the right hand. However, in case of nest with three or four levels, current logic synthesis tool does not change the order.

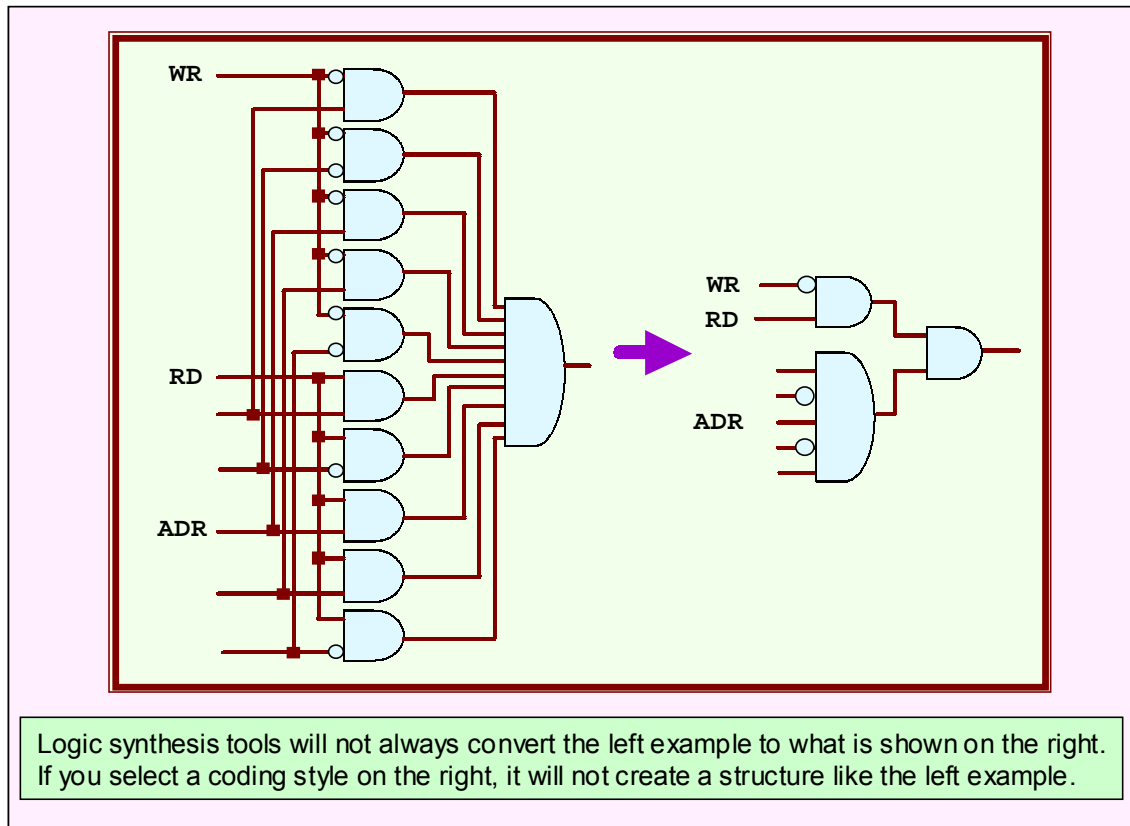


Figure 2-12 Address enable circuit

Describing to have large tables in small conditions, the possibility of generating a redundant and large circuit like the left hand of Figure 2-12 decreases. The state machine description explained in “2.11.State machine description” often becomes description, where are small tables in large condition. In the case of state machine descriptions, readability decreases too much when describing as large tables in small conditions so that debug will be difficult. The state machine description is the exception of this item so the performance of the logic circuit, especially area, tends to be decreased. In such cases, the size of state machine should not be made too large in order to avoid a decrease in performance.

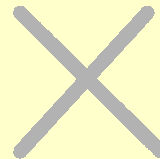
2.9. for statements

2.9.1. Do not use *for statement* for other than simple repeating statements

- | | |
|---|-------------|
| [1] Do not use <i>for statements</i> for other than simple repeating statements (that do not generate priorities) | recommend 2 |
| [2] Initial value and <i>conditions</i> of <i>for statement</i> should be constant, and do not change the values within loop variable | mandatory |

Example Code

```
integer I;
always @(INA) begin : loop
    F = INA[0] ^ INA[1];
    for (I = 2; I <= 7; I = I + 1)
        F = F ^ INA[I];
    end
```



Example 2-43 Parity check circuit that uses *for statements*

Loops statements like *for statements*, control the iteration of statements that are executed multiple times. Verilog-HDL has four loop statements: the *for statement*, the *while statement*, the *repeat statement*, and the *forever statement*. However, only the *for statements* and *while statements* can be used for optimization.

for statements define loop iteration by numerically increasing the index variable.

First, variable *I* is declared as an integer, which is the index variable that indicates the number of *for statement* iterations. It is okay for the index variable to be the reg variable. The *for statement* iteration *conditions* are described in the order variable *I*, initial value, iteration conditions, value of increase.

In the Example code, exclusive logic XOR is executed on the signal *INA* from the lower bits, then a parity check circuit is created. In this description, XOR is executed in order from the lower bits. A balanced tree structure is created depending on the timing constraints.

The Example code in Example 2-43 is not erroneous; instead think of it as a hazardous description.

This description creates a *xor* permutation by repeatedly using variable *F*.

In this case, *xor* is created by permutation and the operating speed of the circuit is compromised. A balanced tree structure would be created depending on the timing constraints of the logic synthesis tool, but if this permutation gets too long, it is uncertain whether the balance tree can be created properly.

When using *for statements*, restrict the generated circuits to those that are mutually exclusive.

RTqualify checks the following items.

2911(E)	Same signal appears on both sides of an assignment in <i>for</i> statement.
2912a(W1)	A constant is used in a <i>for</i> statement conditional expression.
2912b(W1)	For statement initialization value not a constant.
2912c(W1)	A <i>for</i> statement conditional expression is other than a comparison to a constant.
2912d(E)	Increment in a <i>for</i> statement loop variable is not a constant.
2912e(E)	Loop variable is modified within <i>for</i> statement.
2912f(E)	"<=" is used in a <i>for</i> statement.
2912g(W2)	A <i>for</i> statement loop variable is used outside of <i>for</i> statement.

Verilint Warning

W478 : Wrong loop *initialization statement*

W479 : Wrong loop step statement

W480 : Loop index not an integer

W481 : Loop variable not same in all parts

W482 : Wrong loop condition

for-loop is explained in RMM:5.2.14.

2.9.2. Limiting loop-variable operation in *for statements*

[1]	Do not describe any arithmetic operations other than with loop variable and constant	mandatory
[2]	Do not describe any logic or relation operations other than with loop variable and constant	recommend 3
[3]	The number of loops range is up to 10 if operating logically or relationally other than with loop variable and constant	recommend 1
[4]	Use for loop separately from reset part and logic part	mandatory

Example Code

Example code(a) : Arithmetic or of relational operations of other than loop variable and constant

```
for (I = 0; I <= 7; I = I + 1) begin
  if (I >= A - 1)
    S[I] = 1'b1;
  else
    S[I] = 1'b0;
end
```



Example code(b) : The example that arithmetical operation is described outside *for statement*

```
TMP = A - 1;
for (I = 0; I <= 7; I = I + 1) begin
  if (I >= TMP)
    S[I] = 1'b1;
  else
    S[I] = 1'b0;
end
```

Example 2-44 Operation of loop variable in a *for statement*

Explanation

Be sure to avoid using a loop variable in a *for statement* for arithmetical operation with values other than constant value.^[1] The code (a) in Example 2-44 includes relational operation of $I \geq A-1$ and arithmetical operation in conditional expression of *if statement* in *for statement*. Descriptions defined in *for statements* are copied for the number of specified loops. Therefore, in the description (a) of Example 2-44, subtraction circuits that execute $A-1$ as the loop count (8 times) and a comparison circuit that compares I with $(A-1)$ are created. As a result, it will invite area increase.

```
always @(posedge CLK or negedge RESET_X) begin
  if (RESET_X == 1'b0)
    S[7:0] <= 8'h00;
  else begin
    for (I = 0; I <= 7; I = I + 1)
      S[I] <= S[I+1];
    S[7] <= DIN;
  end
end
```

Example 2-45 The operation of loop variable and constant value

2.9. for statements

The loop variable is not generated when executing operation with constant values as Example 2-45 so that this kind of problem would not occur. It does not matter how many loops are in the register description.

Structuring process optimizes common expressions, but if the common expression is large, structuring will be difficult and the circuit structure will be caused a poor result.

As the code (b) of Example 2-44, therefore, keep common statements outside *for statements* as much as possible. In the code (b) of Example 2-44, relational operation (\geq) of loop variable I and TMP is also performed in *for statement*. If relational operator exists in *for statement*, a comparison circuit is created for loop count in the same way as arithmetical operation so that circuit quality may be poor.

In this case, limit the loop variable up to 10 times and pay attention to avoid the decrease in circuit quality.^[3]

```
assign TMP = A - 1;
always @(TMP) begin
  case (TMP)
    3'b000: S = 8'b11111111;
    3'b001: S = 8'b11111110;
    3'b010: S = 8'b11111100;
    3'b011: S = 8'b11111000;
    3'b100: S = 8'b11110000;
    3'b101: S = 8'b11100000;
    3'b110: S = 8'b11000000;
    3'b111: S = 8'b10000000;
    default: S = 8'bxxxxxxx;
  endcase
end
```

Example 2-46 Description after Example 2-44 (b) improved by *case statement*

Even if loop count is limited, generated circuits may be lined in series according as the loop count in *for statement*. Note that the operation speed decreases in the circuit in this case. It is better to avoid describing relational operation and logical operation in *for statement*.^[2]

In the code (a) of Example 2-44, it is safer to describe using not *for statement* but *case statement* like Example 2-46.

It is generally true that it is better not to use *for statement*. However, it is necessary when creating design library with array range free. It is recommended that *for statement* is used only for descriptions, which keeps reuse strongly in mind, or when creating design library.

RTqualify checks the following

- | | |
|-----------|--|
| 2921 (E) | Arithmetic operators aside from loop variables and constants are used in a <i>for statement</i> . |
| 2922 (W3) | Relational operators aside from loop variables and constants are used in a <i>for statement</i> . |
| 2923 (W1) | There are more than <N=10> loops in the <i>for</i> loop. |
| 2924 (E) | Reset part of an <i>always</i> construct that generates a F/F is described as a <i>for statement</i> . |

2.10. Operator description

2.10.1. Order of operators and assignment of 'x'

[1]	Describe with the operation precedence in mind	recommend 3
[2]	Use parentheses for logical operation to make the description more readable even if the precedences are clear	recommend 2
[3]	Use parentheses for two or more arithmetical operations	recommend 2
[4]	Do not compare with 'x' and 'z'	recommend 1
[5]	Do not assign 'x' except for the <i>default clause</i> of <i>case statements</i>	mandatory
[6]	Do not use values including 'x' or 'z'	mandatory
[7]	Do not use RAM output signals for a conditional expression of <i>if statements</i>	recommend 1
[8]	Do not use RAM output signals for the selection expression of <i>case statements</i> that is not assigned 'x' in default clause	recommend 1

Example Code

<code>Y = A + B + C + D;</code>	Operation sequence is from the left to right
<code>Y = (A + B) + (C + D);</code>	Specify the operation sequence using parentheses
<code>Y = (A B C) & D;</code>	Parentheses are required since the & priority is high
<code>Y = (A == B) ? C : D;</code>	Parentheses are not necessary, but using them increases readability

Example 2-47 Logical operation description

Explanation

When describing operation expressions, be sure to take the operation precedences into account.^[1] Figure 2-13 illustrates a list of the operators that can be used in Verilog-HDL and their associated precedences. Evaluation is performed in order from operations with high precedence. If precedences are the same, then operations are processed from the left to the right. Parentheses are used when changing the operator precedences. Use parentheses as much as possible to make the description more readable even if the operation sequence is clear.^{[2] [3]}

For simulation description, it is meaningful to consider the propagation of 'x' and compare 'x' and 'z'.

```
if(A === 3'bxxx )
    Q <= 1'bx;
```

However, "===" and "!==" cannot be used in RTL description since logic synthesis tools do not support these. "==" and "!=" are supported by logic synthesis tools. However, comparison will not be correct if those are used to compare with 'x'.^[4]

```
if(A == 3'bxxx)
    Q <= 1'bx;
else
    Q <= 1'b0;
```

In this case, the conditional expression of *if statement* will never be true no matter what value comes in A, so that Q<=1'bx; will not be executed. Comparison with 'x' is ignored in logic synthesis tools, however, try not to compare with 'x' unnecessarily.

A value that includes 'x', such as 3'b1x1 is correct in terms of syntax, but errors will occur when using logic synthesis tools such that it should not be used.^[6]

```
if(A==2'b11)
    Q <= 1'b1;
else if( A==2'b10)
    Q <= 1'bx;
else
    Q <= 1'b0;
```

As explained in “2.8.3. Use *default clauses*”, assigning 'x' will be treated as *don't-care* and logic synthesis tool will generate circuit as unknown value.

This will lead to a simulation result mismatch between RTL and gate-level as explained in “4.4.2. Inconsistencies can occur in RTL and gate level with the propagation of X”, this kind of description is risky and should be avoided.

```
if(A==1'b1)
    Q <= 1'b1;
else if( A==1'b0)
    Q <= 1'b0;
else
    Q <= 1'bx;
```

In above description, *else item* will be executed only when value of A is 'x' or 'z'. Therefore, this line will not be executed by logic synthesis tool and it will be ignored.

Some describe all conditions at condition expressions of *else if items* and assign 'x' for output signal in *statement* of *else item* to propagate 'x' in simulation, but this is not a correct concept.

will not match. Refer to “2.5 Tri-state Buffers”, “2.8.3. Use *default clauses*” and “4.4.2. Inconsistencies can occur in RTL and at gate level with the propagation of ‘x’, regarding this issue.

RTqualify checks the following

21012(W3) 3 or more clause exist in one expression without brackets.

21013(W2) 2 or more arithmetic operator exist in one line without brackets.

21014(W2) x and z are used for relational calculations

21015(E) x is used for assignment

21016(E) x or z is used for some bits of constant.

210111 cannot be checked by RTqualify..

Verilint Warning

W576: Precise examination required: logic operator (&& | |) used for those with bit widths.

W575: NG : Logical negation (!) is used for vectors

2.10.2. Efficient description using logical operation expressions (Verilog only)

- [1] Is possible to handle logical operation results (TRUE, FALSE) in the same manner as logic values, and use them to simplify the description (Verilog only)
- [2] Results of logical operation will be 1 bit (Verilog only)
- [3] Do not perform logical negation to vector (Verilog only)

reference

reference

recommend 1

Example Code

Normal description

```
reg[7:0] A, B;
reg S;
if (S == 1'b1)
if (S != 1'b1)
Y = (A < B) ? 1'b1 : 1'b0;
if ( A == 8'b00000000)
if ( A == 8'b11111111)
```



Simplified description

```
if (S)
if (!S)
Y = (A < B);
if (!A)
if (&A)
```

Example 2-49 Simplified logical operation

Explanation

The logical operation results return 1'b1 if TRUE or 1'b0 if FALSE. Since they are handled in the same manner as logical values, using this specification makes it possible to simplify the description.

The Example 2-49 shows such simplified description. Since the *if statement* condition branch judges logical values, it is possible to directly specify logical values in conditional expression of *if statement*. It is also possible to directly describe reduction operators or Boolean operation expressions in the conditional expression of *if statement* and perform the condition branch from the operation results.

As explained in “2.1.4. Instructions for equation level descriptions”, logical operation should be used only to 1-bit signals to avoid confusion. If logical negation is performed to vector signal, it is recognized as FALSE when all the bits of signals used for the operation are "0"(all 0). That the result will be TRUE. In cases other than "all 0", it is recognized as TRUE that the result will be FALSE.

With understanding such Verilog-HDL language specifications, it is possible to simplify the descriptions, but take note that readability will decrease if they are oversimplified.

RTqualify checks the following

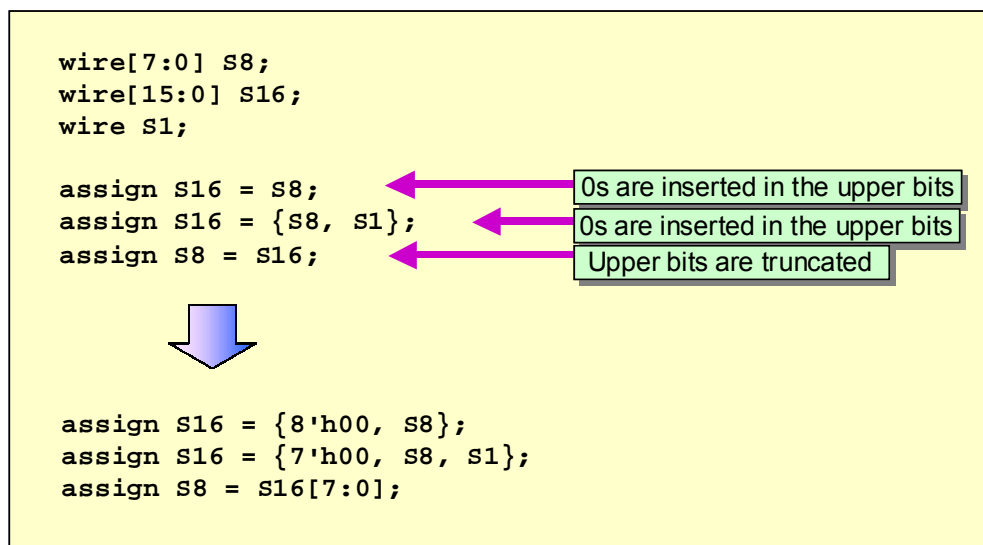
21023(W2) A logical NOT operator is used with multibit data.

2.10.3. Match the bit width of the left side and the right side (Verilog only)

- | | | |
|-----|---|-------------|
| [1] | Clearly match the bit widths between the right side and the left side of relational operator (Verilog only) | recommend 2 |
| [2] | Match the bit width of assignment signal and operand of logical operator (Verilog only) | recommend 1 |
| [3] | The bit width of the right-hand side of assignment should not be wider than the left-hand side assignment (Verilog only) | recommend 1 |
| [4] | The bit width of the right-hand side of assignment should not be narrower than the left-hand side assignment (Verilog only) | recommend 2 |
| [5] | Specify base format ('d','b','h','o') for constant and these base formats should be kept in mind (Verilog only) | recommend 2 |
| [6] | Specify bit width for constants used in conditional expressions and such (Verilog only) | mandatory |
| [7] | Match the bit width of the value with the base number part (2'b) (Verilog only) | recommend 3 |

Example Code

With Verilog-HDL, all data types are fundamentally bit-based. Assignments are allowed even if the data types and bit widths are different. Unless attention is paid to the data types and bit widths, it is very difficult to find mistakes.



Example 2-50 Assignment expressions with different left and right bit widths and example correction

Explanation

The left-hand and right-hand bit widths in the assignment expression of Example 2-50 differ, but all assignments are allowed. These assignment expressions may or may not display warning messages depending on the tool used.

If the bit width of the right-hand side is narrower, 0s are inserted into the upper bits of the right side; then the value of right side is assigned to left side after the bit width justified to the left-hand side. Conversely, if the bit width of the right-hand side is wider, the upper

bits of the right-hand side are truncated.

There will be no problem to mismatch width of the left-hand side and the right-hand side, if these specifications are thoroughly understood. However, descriptions with different bit widths may be made inadvertently without knowing it. Match definitely the bit widths of the left-hand and right-hand sides of the assignments in order to improve the readability of the description.^{[1] [2] [3] [4]}

Concatenations ({}) are used to insert 0s to the upper bits when the bit width of right-hand side operand is narrower. Also, in order to truncate bit width when the bit width of right-hand side operand is wider, part selection is used and sub-vectors are defined.

Always describe the bit width clearly when assigning constant values to signals. If the bit width is not described clearly, the bit width will be treated as 32 bits, and either post-optimization circuits may become larger or erroneous values may be assigned.

```
reg [4:0] A;
reg [3:0] B;

always @(A or B) begin
    if(A == B)
        A_EQ_B = 1'b1;
    else
        A_EQ_B = 1'b0;
```

Example 2-51 Example of comparison operation

The comparison operation in Verilog-HDL is judged by the value of the signal.

In Example 2-51, if A=5'b00101, B=4'b0101, result is true. However, if A=5'b10101, B=4'b0101, the result is false. When bit widths are different like this, if the value of upper bits of the difference is 0 and values of other bits are equal, it is regarded as equivalent. It is confusing if bit widths differ, therefore bit widths should be the same.

When describing value in Verilog-HDL, specify base format ('d', 'b', 'h', 'o').^[5] If not clearly specifying which number and

```
assign A = 15;
```

then, it may be difficult to see whether it is 15 of a decimal number or 15 of a hexadecimal number. If the numeric value is between 1 and 7, there is no problem, as value does not change regardless of the numeral system. Yet, as a practice, it is recommended to specify base format ('d', 'b', 'h', 'o') for all the numeric values.

Since Verilog-HDL is vague in bit width, bit width should be clearly described as much as possible.^[6] When using parameter, it is sometimes advantageous not to specify bit width so that any bit width is allowed. Yet, bit width should be specified for all the constants and parameters unless there is a special reason.^[7] Then, the bit width of the value should be matched with the base number part (4'b, 12'h). Because it will not be an error even if they do not match, and that will lead to confusion.

2.10. Operator description

Particularly when bit width is above 5, definitely recommend describing bit width on base number.

Up to 4 bits, need one digit by hexadecimal only, but above 5 bits, it becomes two digits display, therefore, more careful attention is necessary. Also, a logic synthesis tool with more than 5 bits might cause hierarchy of arithmetic operator and related operator, and bit width gives certain effect to the circuit ability.

Always specify bit width for constants used in conditional expressions and where it is not assigned directly to signal.^[6]

```
parameter C = 15;
if( B+1 > ~C)
```

In the above case, the constant C is 32 bits width. The value of negation bit operation is 32'hffffff0. It is very problematic since it often becomes unexpected operation.

When specifying constant, match the value of bit width definition part(2'b) and the bit width of value.

```
A = 3'b1111111;
A = 3'b11;
```

If the bit width definition part and value do not match like the above, attention should be paid as it easily leads mistakes.

When assigning 0 to multi bit signal, it may be trouble to describe the value for the bit width.

```
A = 32'h0;
```

There is no problem with the above description as 0 is extended to upper bit. The descriptions like this has no problem. However, try to match as much as possible.

Warning messages are not output by simulation or logic synthesis tools for almost all the bit width issues. Always check for them using RTL check tools.

RTqualify checks the following items

- | | |
|-----------|---|
| 21031(W2) | There are differing bit widths for a relational operation. |
| 21032(W1) | Not all bit widths used by a bit operation are same. |
| 21033(W1) | Most significant side is truncated because assigned value bit width is more than bit width of location to which assignment is made. |
| 21034(W2) | Most significant bits may not be defined because assigned value bit width is less than bit width of location to which assignment is made. |
| 21035(W2) | Base ('d', 'b', 'o', or 'h') is not specified for a constant. |
| 21036(W3) | Bit width (<number_of_bit>) not specified for a constant. |
| 21037(W3) | There is a discrepancy in this constant between bit width of value and part wherein bit width is specified. |

Verilint Warning

W19 : Insufficient left side bit width. Upper bits will be truncated.

W328 :Bit width of assigned value (constant) is larger than the variable

W163 : (No bit width specified)Bits truncated by constant assignment exist

W164 : (No bit width specified) No bit was truncated by W163

W162 : (No bit width specified) Variable larger than 32 bits was assigned in variable

W180 : Insufficient right side bit width (insert 0s)

2.10.4. Take note of the different data types between the left and right sides (Verilog only)

[1]	Do not use data types other than reg, wire and integer (Verilog only)	mandatory
[2]	Integers are defined as 32-bit values	reference
[3]	Pay attention to bit widths when assigning integer to reg or wire (Verilog only)	recommend 1
[4]	Pay attention when comparing integer variables and reg/wire variables	reference
[5]	Do not assign negative value to integer	mandatory
[6]	Do not assign negative value to signals, which are declared with reg or wire	recommend 2

Example Code

```
integer INT;
wire[63:0] L;
wire[7:0] S1, S2;
wire W;
reg R;

assign L = INT;
assign S1 = INT;
assign W = S1 == S2;
assign W = R;
```

- 0s are inserted in the upper 32 bits
- Lower 8 bits of the integer are assigned
- Assign the relational operation result
- Assignment from **reg** to **wire** is possible (the reverse care is not possible)

Example 2-52 Assignments among differing data types

Explanation

Verilog-HDL can assign among different data types. This is impossible among nearly all data types in the case of VHDL. The big difference is that VHDL must use type conversion function when assigning among different data types while Verilog can assign unlimitedly. Use only reg, wire or integer data types.^{[1] [3] [4]}

Integers are defined as 32-bit values.^[2] Assigning negative numbers to integers is possible, but attention should be paid when assigning with the *reg* variable. For example, when assigning to variables larger than 32 bits, even if the sign of the top bit of the integer is negative, what is assigned to the upper bits depends on the tool used (It may become positive value).^[5] If the sign should be kept in mind while making assignments, then make the assignments after checking the integer sign bit and deciding the values to be assigned to the upper bits.

Negative value can be assigned to *reg* variables. If assigning negative value, it is regarded as the variable with sign and value assigned. However, some logic synthesis tools may not generate correct circuit so it is recommended not to use negative values.^[6]

Also, if the compare operation expression results are TRUE, 1'b1 is returned, but if they are FALSE, 1'b0 is returned. Therefore, it is possible to assign the operation results described in the Example 2-52 as a signal. However, take note that descriptions will be very difficult to understand when reading these descriptions if these specifications are not clearly understood.

RTqualify checks the following

- | | |
|-----------|---|
| 21044(E) | A constant for which no bit width is specified is used in a relation operation. |
| 21045(E) | A negative value is assigned to an integer variable. |
| 21046(W2) | A negative value is assigned to a signal. |

Verilint Warning

W316 : 0s are assigned to the upper bits in the conversion from integers

2.10.5. Do not share resource in speed critical circuits

- | | |
|--|-------------|
| [1] Take notice that arithmetic operators (+, -, *) and relational operators (<, >, =) with large bit width generate large circuits
- Take notice specially of relational operators, which are 8-bits or more | reference |
| [2] Logic synthesis tools can share resources, which are arithmetic operators and relational operators | reference |
| [3] Do not describe 3 or more shared arithmetic operations | recommend 1 |
| [4] It is hazardous to depend too much on resource sharing
Resource sharing is not perfect | recommend 3 |
| [5] Do not describe arithmetic operations with conditional operators (?) in <i>assign statement</i> (resource sharing will not be performed) | recommend 2 |
| [6] When performing resource sharing, always check generated circuits | recommend 3 |

Explanation

Resource sharing shares resources to reduce the resources necessary for performing operations when it has been detected that operations are not performed simultaneously. For example, as in Figure 2-14 below, in structures that follow the description, each branch operation is executed, then the structure becomes like that in the left side of the figure, to select the operation results using the multiplexer.

However, it is possible to share operators to a single one by placing them in the input side of a multiplexer since each conditional branch of the *if statements* are not executed simultaneously. In the right side of the following figure, the operators are shared to a single one. This makes it possible to reduce the resources needed for the operation.^[2]

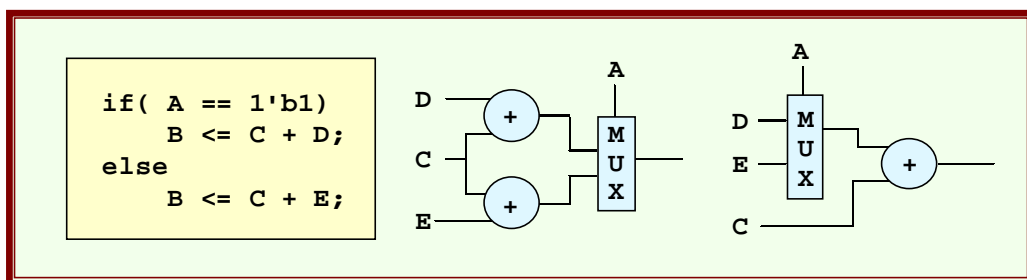


Figure 2-14 Resource sharing

However, the following precautions are necessary regarding resource sharing of the arithmetic operators or relational operators.^[1] With logic synthesis tools, resource sharing is executed out of the timing. For example, if the timing of select signal A in the circuit of the Example 2-14 is slow, the circuit on the right side of Figure 2-14 will have a timing disadvantage since the multiplexer is on the input side. In this case, resource sharing is not performed to satisfy the timing.

Logic synthesis tools automatically judge resource sharing and determine the structure, but since resources cannot always be shared properly, it is hazardous for resource timing

according to this timing to be dependent on resource sharing.^[4] As a standard, try not to share resources between more than three arithmetic operations.^[3] If the appropriate structure is already known, then define it clearly.

Resource sharing of arithmetic operation will not be performed when a conditional operator (?) is used in *assign statement* as described below. If you intend to perform resource sharing, use if-else description.^[5]

```
assign B = (A==1'b1) ? C + D: C + E;
```

The result of resource sharing by Design Compiler may become poor when it is performed in speed. Design Compiler selects circuits, which a large amount of operators to improve speed even a little.^[6] If specifying `hlo_resource_allocation=area_only`, resource sharing is performed only in area that is safe. If resource sharing is not used at all with the Design Compiler, it is better to use `hlo_resource_allocation=none`. `simple_compile_mode` is available from Design Compiler 1999.05 version. Since resource sharing is performed only in area in the case of `simple_compile_mode`, resource sharing is relatively safe.

RTqualify checks the following:

- 21052(W3) Resources will be shared if there are more than <n> arithmetic operations in an *always construct*.
- 21054(W1) Resources may not be shared correctly if there are more than <n> arithmetic operations in an *always construct*.
- 21055(E) Resource sharing may be extremely time consuming if there are more than <n> arithmetic operations in an *always construct*.
- 21057(W2) Resource sharing is not possible for this operation.

2615 acts for 21051 to check.

2.10.6. Notes on arithmetic operations

- | | |
|---|-------------|
| [1] Carry-out should be considered for bit width of signals to which operation result will be assigned | recommend 3 |
| [2] Beware of the sign extension and adjust bit widths in signed operations | recommend 1 |
| [3] Signed operations and unsigned operations should not be mixed in one statement | recommend 3 |
| [4] Use <i>continuous assignments</i> to clearly define data path structures | recommend 3 |
| [5] Do not infer large multiplier by the RTL description but describe the contents of multiplier by logical operation | recommend 1 |
| [6] Do not use division | mandatory |
| [7] Do not describe more than one arithmetic operation in one line (except for carry-in A+B+CIN(1bit)) | recommend 3 |

Example Code

```

reg [15:0] D1, D2, D3, D4, D5, D6, D7;
wire[16:0] T1, T2, T3;
wire[17:0] U1, U3;
wire[18:0] U2;
reg [18:0] DATA;

assign T1 = {D1[15], D1} + {D7[15], D7};
assign T2 = {D2[15], D2} + {D6[15], D6};
assign T3 = {D3[15], D3} + {D5[15], D5};
assign U1 = {T1[16], T1} - {T2[16], T2};
assign U2 = {U1[17], U1} + {T3[16], T3[16], T3};
assign U3 = {T3[16], T3} + {D4[15], D4, 1'b0};
always @(posedge clk)
    DATA <= {U3[17], U3} + {U2[18], U2[18:1]};

```

Example 2-53 Description of data path circuit

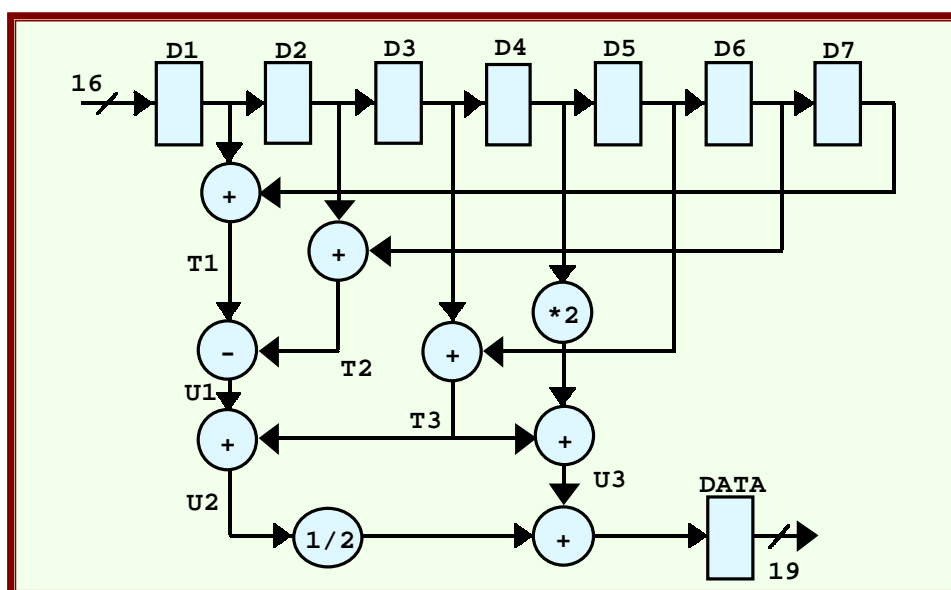


Figure 2-15 Structure of data path circuit

In arithmetical operation, bit width should be specified correctly keeping carry out output in mind.^[1]

With Verilog-HDL, this becomes an unsigned operation if no particular sign is specified, but since signed and unsigned are the same in the case of add (+) and subtract (-), signed operation is also possible. In the case of signed operations, it is necessary for the bit width of the assigned variable and both sides of the expression to match.^[2]

Expand the sign bit to the upper bits and make the bit widths same. The signed operation is easy to lead bit widths mistakes. Therefore, it is recommended to check by RTL check tool.

Pay extra attention when operating signed and unsigned signals.^[3] When adjusting bit widths of right hand side and left hand side of an equation, signed signal will extend the top bit (sign extension), while with unsigned signals, it is not necessary to adjust the bit widths to be the same. When performing operations like this, it won't be possible to distinguish whether that is an operation of signed and unsigned signals, or it is an operation of signed signals with wrong bit widths. In a case like this, unsigned signals should be handled as signed signals by assigning 0 for upper bits.

One *always construct* is subject to resource sharing. In order to describe structures that are not dependent on resource sharing, define the operational expression using a *continuous assignment*. With data path structures, descriptions that use *continuous assignments* have improved readability.^[4]

When performing arithmetic operations, beware of the carry out signal. For example, when adding 8-bit A and 8-bit B, the output becomes 9-bit when carry out is included.

Excellent circuit of a large multiplier cannot be generated by logic synthesis tool. The contents of the multiplier, the result of which would become 16 bits or more, should be described in logical operation.^[5] Some ASIC vendors may have high performance multiplier as library. Also, multiplier generated by exclusive tool (Module Compiler) can be used. It is recommended to use logical operation or gate level circuit, which are obtained from outside.

Most logic synthesis tools do not support division (/) so obtain a divider on your own.^[6] In case of division for the power of 2, such as 2, 4, 8, 16, the operation is just a shift operation and therefore division can be used.

```
reg [15:0] A;
wire [11:0] Y;
assign Y = A / 16;
```

However, this type of division easily leads to confusion.

Therefore, it is better to describe by specifying bit position as follows.

```
assign Y = A[15:4];
```

It is better not to describe more than one arithmetic operation in one line.


```
reg [7:0] DIN0,DIN1,DIN2,DIN3;
reg [9:0] DOUT;
assign DOUT = DIN0 + DIN1 + DIN2 + DIN3;
```

The above description may cause 2 bits to carry around end. Bit width on two sides of an assignment will be different, and it cannot be recommended. Also, when more than one operation is performed in one line, arithmetic operators will be linearly created. In order to clarify the order of operations, this kind of operation should be described as one operation per line, as in the following example. ^[7]

```
reg [7:0] DIN0,DIN1,DIN2,DIN3;
reg [8:0] DIN0plusDIN1,DIN2plusDIN3;
reg [9:0] DOUT;
assign DIN0plusDIN1 = DIN0 + DIN1;
assign DIN2plusDIN3 = DIN2 + DIN3;
assign DOUT = DIN0plusDIN1 + DIN2plusDIN3;
```

However, even if you have two arithmetic operations in one line as in the following example when three items are 1 bit;

```
assign Q = A + B + CIN;
```

adder circuit with carry input will be generated and there will be no problem in this case.

RTqualify checks the following:

21066(E) '/' a '%' cannot be used in RTL description. (Excluded when divided by power of 2)

21067(W3) There is more than one arithmetic operation on a single line.

2.10.6.[5] is checked by 3252

3252 There is a multiplier using a '*' operator where output exceeds <N=15> bits.

Verilint Warning

W484 : Carry out signal not output.

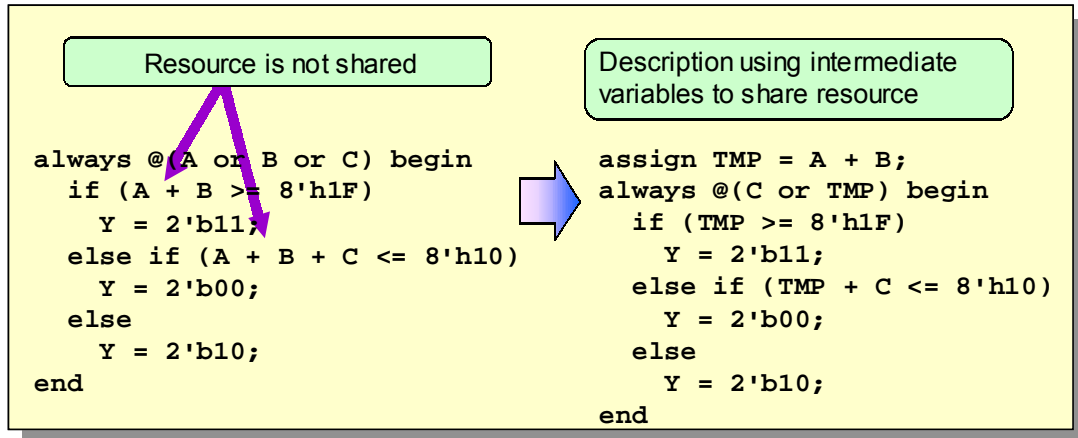
No warning related to resource sharing is output.

Resource sharing is explained in RMM:5.6.6.

2.10.7. Take share items out of conditional branches

- | | |
|---|-------------|
| [1] Do not use arithmetic operation in the conditional expression of <i>if statements</i> | recommend 1 |
| [2] Clearly describe common operational expressions | recommend 2 |

Example Code



Example 2-54 Sharing by intermediate variables

Explanation

Operation statements in the conditional expressions of *if statement* are not subject to resource sharing, even if they are in the *always construct*.^[1] Automatic resource sharing is therefore not performed, so use intermediate variables, define common items, and share them. However, as mentioned previously, it is hazardous to heavily depend on resource sharing. Use an *assign statement* to clearly define these as the Example 2-54.

As previously mentioned in “2.10.6. Notes on arithmetic operations”, if the data path operation structure is known in advance, then clearly describe the structure.^[2] With resource sharing, a structure may not be a designer’s intention.

RTqualify checks the following:

21071(W2) There is an arithmetic operation in conditional expression in an *if statement*.

2.11. State machine description

2.11.1. Use Mealy type and Moore type descriptions properly

- | | |
|--|-------------|
| [1] Use the Moore type in principle
- Either Mealy or Moore type descriptions are okay if all output signals are FF | recommend 3 |
| [2] Minimize the bit change in state transitions (fundamentally gray code) | recommend 3 |
| [3] Define state values by parameters | recommend 3 |
| [4] Keep the number of states to 40 or less | recommend 2 |
| [5] If there are more than 40 states, divide the states and describe using a separate module | recommend 3 |

Explanation

There are two types of state machine circuits: Mealy type and Moore type. The Mealy type has branch depending on the input signal conditions in one state. In contrast, there is no branch in one state with the Moore type. With the Mealy type, two or more output conditions can be described in a single state, so the number of states decreases. This decrease results since there are paths in which timing paths exist where combinational logic passes from the input signals to the output. If such paths exist, delay propagation may be a problem when connected to other blocks.

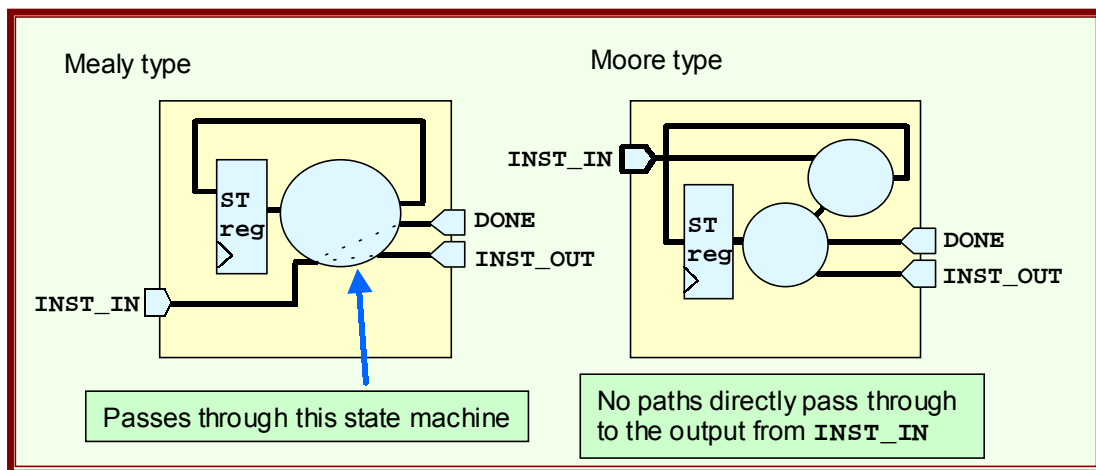


Figure 2-16 Mealy type vs. Moore type

We recommend using Moore type state machine in principle.^[1] The number of states increases for the Moore type, but there are no longer any timing paths passing through state machine, so this can be called a safe state machine from the perspective of timing. Figures 2-17 and 2-18 illustrate the Mealy type and Moore type states, respectively, of the same logic.

Normally, state machine output is output as combinational logic. Therefore, this contradicts “1.6.2. Make basic block FF output & combinational circuit input”.

However, if the state machine output is specified as FF output as the Figure 2-19, output signal will be delayed by one clock and output will change. This is okay for systems in which it is okay for one clock delayed by FF latch, but it will have to be output as combinational logic for systems in which it is not allowed.

If it is okay for the output to be FF, there is no need to describe it in the Moore type.

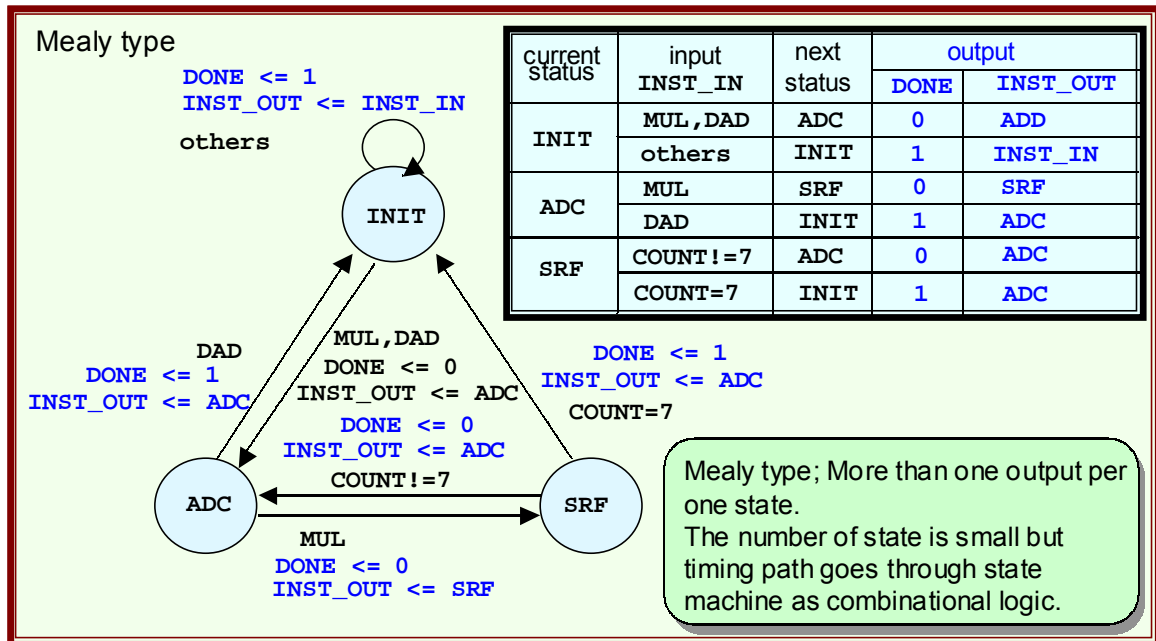


Figure 2-17 Mealy type state machine

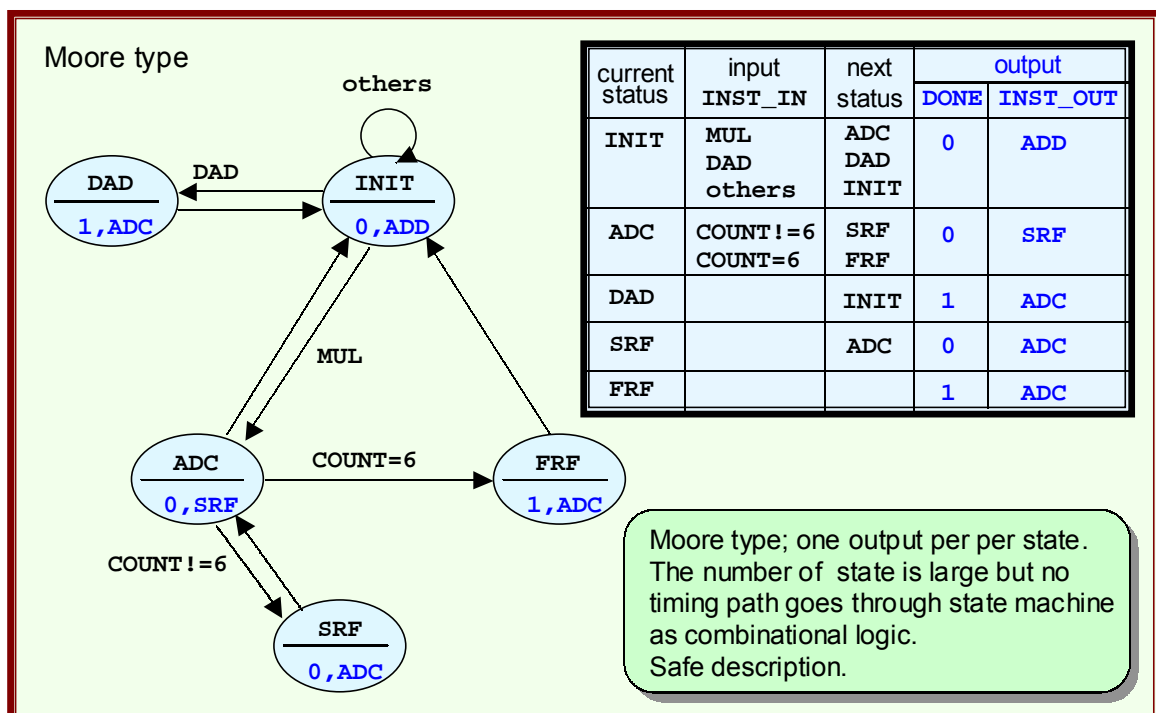


Figure 2-18 Moore type state machine

2.11. State machine description

It is better to have state machine modules (blocks) as sub blocks, which are smaller than basic blocks. Make its upper level as basic level as much as possible. It is preferable to have the combinational circuit FF output level structure in the basic level.

However, in reality, data path circuit size, which receives state machine, becomes large so that it may be difficult to put both the state machine block and data path block under the basic level. Even though not ideal, pay attention to the relationship of blocks between the state machine and data path part. As for the relationship between the state machine and data path part, refer to “1.6.5. Separate the data path section from the controller”.

In the state machine description, separate the descriptions of the combinational circuit block, which implements the state transitions, and the register block (FF), which delivers the state transitions. This will be the most efficient way to output state machine output as combinational logics.

If describing combinational part in *always construct*, which generates register (FF), output signal is latched. The details are given in “2.11.3. Separate FF inference and case statements”.

State machine itself would not cause timing problem since the circuit size is not so large. State machine output however is normally used by data path, which often causes speed problems. In order to improve the speed, it would also be effective to think of the state allocation value as binary, gray code, one hot. Before that however, it is crucial that the state machine does not get too large.^[2] Also, readability will improve by defining state value by parameter.^[3]

Keep the number of states within 40 when using binary or gray code state numbers.^[4] 33 through 64 states are realizable with six registers. However, value allocation becomes difficult as the number of states approaches 60, so it is required to use more registers.

See “1.6.5. Separate the data path block from control block” and “2.11.2. Isolate state machine circuits” for more detail.

State machine description contradicts “the RTL description style that RTL description keeping circuit quality in mind” which has been explained in Chapter 2.

“2.6.1. Describe taking the circuit structure into account” recommends not describing too many output signals from *always construct*. However, in state machine description, signals output from this state machine are described in the same *always construct* so the number of output increases. Also, it becomes a large *case statement* to a certain extent, which conflicts with the rule that it is better to describe from small table to large table in “2.8.6. Beware of nesting that *if statements* and *case statements* coexist”.

Therefore, attention should be paid by limiting the size of the state machine so that circuit quality does not become so poor.

State machine is explained in RMM5.5.8.

2.11.2. Isolate state machine circuits

- [1] Isolate state machine circuits
- Different logic optimization methods can be tried
 - State allocation changes are possible

recommend 3

Explanation

State machine circuit should be an independent block whenever possible.^[1] The strategy for synthesizing state machine may differ from that for the data path or other combinational circuits.

State machine descriptions often have small *case statements* inside fairly large *case statements*. This description, in contrast to “2.8.2.Divide using *if statement*, etc. to avoid creating large tables” and “2.8.6.Beware of nesting that *if statements* and *case statements* coexist” is not suitable for logic synthesis.

In order to prevent this negative influence, either restrict the state machine size and reduce the negative influence as mentioned in “2.11.1.Use Mealy type and Moore type descriptions properly” or get redundancy using the method explained in “5.2.3.Flatten (set_flatten)” and “5.4.2.Speed optimization” to improve the speed and area.

If the flatten method is used, it may not be completed when the description gets too large. Also, if it is mixed with other descriptions, there may be cases in which executing flattening on other description portions may have the reverse effect. Therefore, we recommend isolating the state machine circuit from other descriptions.

See “1.6.5.Separate the data path block from control block” for information about the relationship between state machine and other blocks.

Example Code

```
module MULSTATE( CLK,RST_X,INST_IN,INST_OUT,DONE);
input          CLK,RST_X;
input[3:0] INST_IN;
output[3:0]INST_OUT;
output        DONE;
reg DONE;
reg[3:0] INST_OUT;

reg[2:0] CurrentState,NextState;
parameter P_INIT = 3'b000;
parameter P_ADC = 3'b001;
parameter P_DAD = 3'b010;
parameter P_SRF = 3'b011;
parameter P_FRF = 3'b100;
parameter P_CADD = 4'b1010;
parameter P_CADC = 4'b1011;
parameter P_CSRF = 4'b0100;
parameter P_CMUL = 4'b1100;
parameter P_CDAD = 4'b1101;
parameter STB = 1;
reg [2:0] Count;
```

constant declaration of state value

```

(continue from the previous page)
always @(posedge CLK or negedge RST_X)
  if(!RST_X)
    CurrentState <= #STB P_INIT;
  else
    CurrentState <= #STB NextState;

always @(CurrentState or INST_IN or Count)
  case (CurrentState)
    P_INIT : begin INST_OUT <= P_CADD; DONE <= 1'd0;
              if(INST_IN==P_CMUL)
                NextState <= P_ADC;
              else if(INST_IN==P_CDAD )
                NextState <= P_DAD;
              else
                NextState <= P_INIT;
            end
    P_DAD  : begin INST_OUT <= P_CADC; DONE <= 1'd1;
              NextState <= P_INIT;
            end
    P_ADC  : begin INST_OUT <= P_CSRF; DONE <= 1'd0;
              if(Count==3'h6)
                NextState <= P_FRF;
              else
                NextState <= P_SRF;
            end
    P_SRF  : begin INST_OUT <= P_CADC; DONE <= 1'd0;
              NextState <= P_ADC;
            end
    default : begin INST_OUT <= P_CADC; DONE <= 1'd1; // FRF
              NextState <= P_INIT;
            end
  endcase

always @(posedge CLK or negedge RST_X)
  if(!RST_X)
    Count <= #STB 3'h0;
  else if(CurrentState==P_INIT)
    Count <= #STB 3'h0;
  else if(CurrentState==P_ADC)
    Count <= #STB Count + 1;

endmodule

```

Simple description assigning state register NextState to CurrentState

Description of combinational circuit to create next state(NextState)

Example 2-55 State machine circuit description (binary)

State machine is explained in RMM: 5.5.8.

2.11.3. Separate FF inference and *case statements*

- | | |
|--|---|
| <p>[1] Separate the FF inferences and case statements defining state, in state machine circuits</p> <ul style="list-style-type: none"> - Describe FF inference using independent always constructs - Describe state definition with another always construct using if statements and case statements | <div style="border: 1px solid blue; border-radius: 50%; padding: 2px; display: inline-block;">recommend 3</div> |
| <p>[2] Unless <i>case statements</i> are separated, the output signal is latched and delayed by 1 clock cycle</p> <ul style="list-style-type: none"> - If it is allowed to be a 1 clock delay, there is no need to recognize Mealy or Moore type descriptions | <div style="border: 1px solid blue; border-radius: 50%; padding: 2px; display: inline-block;">reference</div> |

Explanation

It is a principle in state machine description that combinational circuit, which defines state, and sequential circuit, which infer FF, are separated. When describing combinational circuit, which defines state, readability will be better if the output signal from this state machine is also defined in the same if statement and case statement.

As explained in “2.11.1. Use Mealy type and Moore type descriptions properly”, the state machine output is advantageous from a circuit design standpoint if it is output as combinational logic.

In this case, in order to output the signal as combinational logic output, it must be described using either combinational logic *always constructs* or *assign statements*. If the state machine *case statements* are described in *always constructs* that infer FFs, the output signal must be described in another location. However, describing the state machine would become complex if it reaches a certain size.^[1]

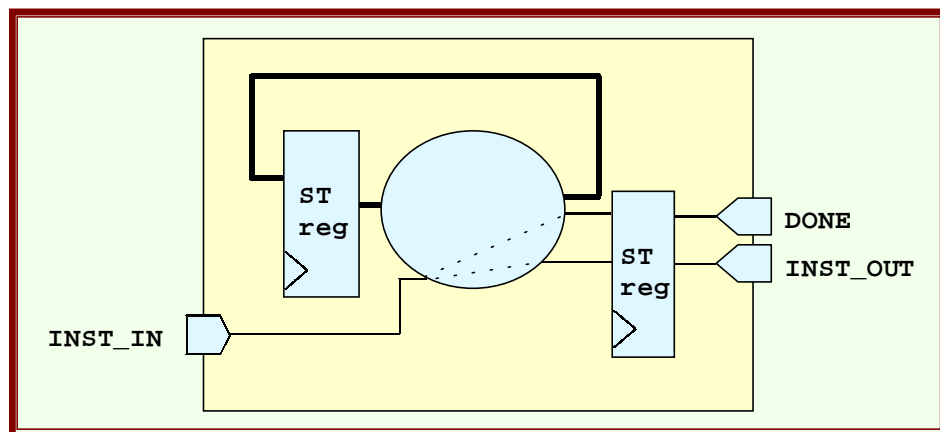


Figure 2-19 Output latch type state machine

In the case of the state machine that latches output signals using FFs, it is not necessary to separate combinational circuits and sequential circuits. If this method is used, all output from the state machine module becomes FF output, so the timing conditions become easier.^[2]

In the case of such descriptions, there is no particular problem with using a Mealy type state machine since paths that pass through this module as combinational logic no longer exist.

There is also no problem if there are up to about 100 states.

2.11.4. Consider the state allocation

[1] Allocate the states properly

- Basically gray code. Use one hot when speed is an issue.
- Allocate states so there are fewer bit changes
- One hot only increases the area 20-30% if there are up to about 15 states

recommend 3

[2] Use parameter or define to facilitate allocation changes

recommend 3

Explanation

There are three methods for allocating states for the state machine: binary, gray code, one hot. These state allocation values define constants using either parameter or define. It is possible to avoid directly specifying values in the state transition descriptions.

The binary method can minimize the number of registers required for state allocation, but timing becomes slower since a decoding circuit is required.

With the binary method, similar processes should be the same bits and the number of changes to each bit is reduced as much as possible.^[1]

Similar to the binary method, a decoding circuit is necessary with the gray code method, but since only one bit changes, it is possible to increase the speed and decrease power consumption. However, it is difficult to represent each state change by only changing one bit, so it would be impossible to create a complete gray code. Describe so that bit changes are kept to a minimum.

One-bit one hot is a description method that uses registers for the number of states. The number of registers increases, but since no decoding circuit is necessary, this method has the best timing performance. The binary, gray code, and one hot description methods make it possible to freely change values by defining constants.

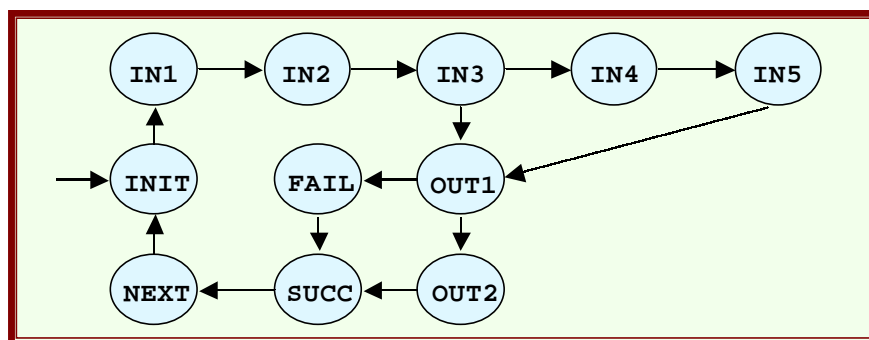


Figure 2-20 State shift

Example of value allocations and state transitions are illustrated in Figure 2-20 and Example 2-56. In the case of binary allocations, whenever possible, values are allocated to minimize bit changes by transitions between states. When there are many consecutive state transitions from IN1 to IN5 as in Figure 2-20, values are allocated to IN1 through IN5 as in the Example 2-56 by using gray code to define them so that the number of bit change can be minimized. Similarly, state values are allocated with the state transitions taken into account. Area and speed can be improved by decreasing the number of bit change since the state transits like gray code rather than binary type.

```

reg [3:0] CURRENT_STATE,NEXT_STATE;
parameter INIT    4'b1000
parameter IN1     4'b0000
parameter IN2     4'b0001
parameter IN3     4'b0011
parameter IN4     4'b0010
parameter IN5     4'b0110
parameter OUT1    4'b0111
parameter OUT2    4'b0101
parameter FAIL    4'b1111
parameter SUCC    4'b1101
parameter NEXT    4'b1001

```

Example 2-56 Binary allocations

It is important not only to keep the bit change small, but also to decide the value considering output signal of the state machine. For instance, when an output signal such as IN_MODE is necessary for IN1-IN5 in the state machine of the Example 2-56, if keeping a particular bit of the above state values to be 1, and always 0 for another operation, extra decoding is not necessary for output signal that the circuit becomes small and speed is improved.

In the state machine, the allocation of value is an important factor. Try to have values in one place in the *parameter* or *define* and consider state values.^[2]

```

reg [10:0] CURRENT_STATE,NEXT_STATE;
parameter INIT    11'b00000000001
parameter IN1     11'b00000000010
parameter IN2     11'b00000000100
parameter IN3     11'b00000001000
parameter IN4     11'b00000010000
parameter IN5     11'b00000100000
parameter OUT1    11'b00001000000
parameter OUT2    11'b00010000000
parameter FAIL    11'b00100000000
parameter SUCC    11'b01000000000
parameter NEXT    11'b10000000000

```

Example 2-57 One hot allocation

When the state machine value is parameterized, it is easy to change circuit structure to the one hot type state machine. In the case of one hot, bits for the state count are required. Only one bit is set to 1 in one hot descriptions. Moreover, one hot allocation is faster than binary allocation since no extra decoding circuit is required that and speed is improved. The change from binary to one hot in the Example 2-55 can be easily performed by changing this parameter value and bit widths of CURRENT_STATE,NEXT_STATE. In one hot type, speed is improved but area increases. One hot type can be employed when the number of state is up to about 18.

Chapter 3 RTL Design Methodology

This chapter introduces the methodology for creating function libraries, the parameterization of design resources, test facilitation design, low power consumption design, and methods for managing design data. This chapter also introduces the methodology for improving design quality and the reusability of design resources.

Contents

- 3.1 Create function libraries
- 3.2 Using function libraries
- 3.3 Design for Test (DFT)
- 3.4 Low power consumption design
- 3.5 Source codes and design data management

3.1. Create function libraries

3.1.1. Create and utilize function libraries

- | | |
|---|-------------|
| [1] Create sub-programs which can be used in common | recommend 3 |
| [2] Create reusable component libraries and streamline the design | recommend 3 |
| [3] Standardization of design data is necessary for design libraries to be reused | recommend 3 |
| [4] Intellectual Properties (IP) can be purchased from third-parties | reference |

Explanation

The term function library here refers to sub-programs such as common tasks or functions that can be used when designing circuitry and reusable design circuits (components).^[1] Libraries that are easy for designers to use are limited to only simple functions that do not include FFs. We recommend preparing special operation circuits such as selectors, encoders, and combined operations in function libraries.^[2] When creating design libraries for sequential circuits that include FFs, reuse becomes difficult unless the specifications appear to be simple on the outside. We recommend creating simple I/O pin specifications if the library reaches a certain size.

By using the function library as a common data source for the designers, not only will the number of individual development tasks be reduced, but the differences in design quality between individual members will be reduced as well.^[3]

Moreover, reusing circuits that have already been verified and have a proven track record will shorten the development period and increase circuit reliability.

Standardizing a design style that takes reuse into account is necessary for commonality of the data. Standardization unifies the function library descriptions and facilitates reuse.

Standardization of design properties is currently being carried out worldwide by an organization called VSIA (Virtual Socket Interface Alliance). When this standardization is realized, it will be possible to purchase design properties from third-party vendors and integrate them into your chips.^[4] Design properties at VSIA are classified into three categories. Below is an introduction to the design property categories and the advantages and disadvantages of each.

IP categories	Data expression	Merits	Demerits
Software IP	Design data described with synthesizable RTL	Much generality, easy spec changes	Performance varies greatly according to the synthesis methodology. Much time required until chips are realized.
Firmware IP	Technology specific, or individual netlist-level design data	Chips can be easily realized by converting technology	Description changes are limited. Specifications are difficult to understand.
Hardware IP	Polygon expression layout level design data	Can be realized in chips quickly. Guaranteed performance.	Changes are difficult. Dependent on layout tools.

Table 3-1 IP categories (reference)

3.1.2. Describe a style that takes reuse into account

[1] Follow the basic naming conventions	mandatory
[2] Do not make technology-dependent descriptions	recommend 1
[3] Use a simple clock system	recommend 3
[4] Insert appropriate comments to make it easier to understand	recommend 3
[5] Describe with the circuit structure in mind	recommend 3
[6] Parameterize whenever possible	recommend 3
[7] Do not use embedded logic synthesis scripts in the source code	mandatory

Explanation

Creating function libraries based on a description style that takes reuse into account makes it possible to streamline the design. If the description style is not standardized, understanding the function library specifications will take more time and integrating the function library into the circuit will become more labor-intensive.

Keep the following points in mind for description styles that take standardization into account.

* Follow basic naming conventions^[1]

Name circuits, signals, instances, etc. according to basic naming conventions. See “1.1.1. Following basic naming conventions” and “1.1.2. Circuit and *pin name* conventions should consider the hierarchy” for more information.

* Do not make the description dependent on technology^[2]

Do not define *cell names* that are technology library dependent directly in the description.

* Use simple clock system^[3]

In order to reuse designs, the clock system must be made as simple as possible.

If the clock system is complicated, it will become necessary to consider delay, synthesis, and layout problems.

Make the clock system as simple as possible and eliminate problems other than the logic.

* Insert appropriate comments to make it easier to understand^[4]

Adding more comments and clarifying the role and objective of each signal and description will make the descriptions easier to understand. See “3.5.6. Use comments often” for more information.

3.1. Create function libraries

- * Describe with the circuit structure in mind^[5]

Describe with the circuit structure in mind so the circuit structure can be easily understood.

See “2.6.1.Describe taking the circuit structure into account” for more information.

- * Parameterize whenever possible^[6]

Parameterize so changes to the I/O port bit width and the constants in the description can be made easily. See “3.1.5.Parameterize array range of module I/O” for more information regarding parameterization.

- * Do not use embedded dc_shell scripts in the source code^[7]

It is possible to describe Design Compiler dc_shell scripts in the source code. However, description will be tool dependent, so create dc_shell scripts as separate files without embedding them in the RTL description.

3.1.3. Unify description order of module I/O ports

- | | |
|--|-------------|
| [1] Unify the description order of the <i>port declaration</i> , <i>port list</i> and <i>module instantiation port lists</i> defined in the modules
- Fewer errors will be made when defining ports, so they will become easier to understand when viewed | recommend 3 |
| [2] The <i>port</i> description order should be clock, reset, input, output, I/O | recommend 2 |
| [3] Separate the <i>reg</i> declarations of sequential circuits and combinational circuits | recommend 3 |
| [4] Define one signal per line in I/O, <i>reg</i> declaration, <i>wire</i> declaration.
Always add comments | recommend 3 |

Example Code

<code>module CNT8(CLK, RST_X, CARRYIN, DIN, LD, CARRYOUT, DOUT);</code>	
<code>input CLK;</code>	Describe input signal first Describe clock and reset signal first
<code>input RST_X;</code>	
<code>input CARRYIN;</code>	
<code>input[7:0] DIN;</code>	
<code>input LD;</code>	
 <code>output CARRYOUT;</code>	Describe output signal
<code>output[7:0] DOUT;</code>	
 <code>reg[7:0] CNT;</code>	reg declaration to generate FF
<code>reg COUT;</code>	reg declaration which does not generate FF
 <code>wire LSB8, MSB8;</code>	wire declaration

Example 3-1 A good example of the I/O port sequence

Explanation

Defining the port description order according to the convention makes it possible to reduce errors when calling the function library from an upper level.^[1]

The port description should be in order of basic control signals such as clock, reset, and enable, and then input, output and I/O. There is no particular description order within input, output and I/O, but collecting signals according to application whenever possible will contribute to reducing errors.^[2]

It is recommended to describe I/O ports for the module instantiation in the same order as its module declaration.

Moreover, those signals for which *reg* is declared either become registers or are used as combinational circuits. Declare these signals separately without mixing them in the declarations.^[3] When describing simple combinational circuits, using *continuous assignment statements* makes it possible to clearly distinguish them.

3.1. Create function libraries

Define one signal per line for I/O, *reg* declarations (when generating registers) and *wire* declarations, and adding some comments is helpful. See “3.5.6. Use comments often” for more information.^[4]

RTqualify performs following checks.

- | | |
|-----------|---|
| 3131 (N) | Add a comment to input or output signal. |
| 3131a(W3) | Port list description order is other than clock, reset, input, output, input/output. |
| 3131b(W3) | Port list does not match order of <i>port declarations</i> . |
| 3131c(W3) | Lower layer <i>port list</i> and <i>instance statement</i> port connection description do not match. |
| 3131d(W3) | Declarations of <i>reg signals</i> for sequential circuits and combinational circuits are mixed. |
| 3132(N) | Declarations are not in recommended order. |
| 3134 (N) | <i>Wire declaration</i> with no comment. The comment of <i>reg declaration</i> is checked by 2114. (excluded from check as default setting) |

The comments description method is explained in RMM: 5.2.5, 5.2.6 and 5.2.7. The port description order is explained in RMM: 5.2.10. RMM prescribes to separate comments lines from RTL lines.

3.1.4. Consider RTL description readability

- | | |
|--|-------------|
| [1] Signal naming is essential for RTL description readability | reference |
| [2] Unify the number of indents used in <i>always constructs</i> , <i>if statements</i> and <i>case statements</i> (2 spaces are standard) | reference |
| [3] Replace tabs with spaces after editing | recommend 3 |
| [4] Do not describe multiple assignments in 1 line | recommend 3 |
| [5] The maximum number of characters in 1 line should be about 110 | recommend 3 |

Explanation

Naming that is easy to understand is most important for the readability of the RTL description.^[1] Keep naming conventions in “1.1.2.Naming conventions of circuit and pin names should be considered by the hierarchy” and “1.1.3.Give meaningful names for signals” in mind. For further improvement of readability, unify the indent positions and avoid describing multiple assignments in 1 line.

As seen in the description examples of the Style Guide, indents are used in *if statements* and *case statements* in *always constructs*, and in *always constructs* itself. Also use indents for every nesting. Unify the number of indents so that the description is easy to read. Given the fact that approximately 5 nests are generated in a general description, a smaller number of indents is better. The standard should be about 2 spaces.^[2]

Using tabs to insert indents is advisable for smooth editing. However, when using tabs, if editors with a different number of tab stops are used in a different environment, the indication position shifts. In this case, it is recommended that you convert the tabs into spaces before saving the design properties.^[3]

Example Code

Bad example:

```
assign A = B & C, D = K & M;

if(EN==1'b1) begin
  A = B & C; D = K & M;
end
```



A description that concatenates by comma (,) in *assign* statements like that shown above is a bad description style from the standpoint of readability. Instead, try to describe one expression per line.^[4]

Inserting many comments can also improve RTL description readability. The description can easily be checked when a comment description is added next to the RTL description rather than completely separating it from the RTL description. Refer to “3.5.6. Use comments often” for comment descriptions.

3.1. Create function libraries

The number of characters of the display terminal was fixed at 80 in former systems. Currently, there are some editors available that can handle more characters per line. However, if the number of lines is too great, folding occurs and readability decreases. Therefore, it is better to limit the number of characters in 1 line to about 110.

RTqualify performs following checks.

- 3142 (N) Indents not uniform.
- 3143 (N) Tab character is used.
- 3144 (N) Multiple assignment statements in a single line.
- 3145 (N) Do not exceed <N=110> characters per line.
- (All of the above are excluded from check as default setting)

Methods to improve readability such as the compliance with 1 line 1 command are explained in RMM: from 5.2.6 onward.

3.1.5. Parameterize the array range of module I/O

- | | | |
|-----|--|-----------|
| [1] | It is preferable to parameterize the array range of module I/O as much as possible
- Makes it possible to use common modules even in circuits with different bit widths | reference |
| [2] | Specify the bit width using parameters or text macros
- Overriding from an upper level is possible when using <i>parameter statements</i>
- Specifying entire design circuits is possible with text macros | reference |
| [3] | Set parameter default values | reference |

Example Code

```

module SEL4TO1(DIN0,DIN1,DIN2,DIN3,SEL,Y);
    parameter K = 16;
    input [K-1:0] DIN0,DIN1,DIN2,DIN3;
    input [3:0] SEL;
    output [K-1:0] Y;

    assign Y = ({K{SEL[0]}} & DIN0) | ({K{SEL[1]}} & DIN1)
              | ({K{SEL[2]}} & DIN2) | ({K{SEL[3]}} & DIN3);
endmodule

```

Example 3-2 Parameterization of the input bit width

Explanation

Using parameters to specify the array range makes it possible to use common descriptions even if the bit widths of the I/O signals differ.^[1] Parameters are specified when their modules are called from the top level as components.

Bit widths can be specified either with *parameter statements* or text macros to improve reusability.^[2] In the case of function libraries, *parameter statement* definition is recommended since multiple bit widths may be used in a single circuit. Moreover, it is necessary to assign constants as default values to *parameter statements* without exception.^[3]

An example description when specifying using text macros is shown below.

```

module SEL4TO1(DIN0,DIN1,DIN2,DIN3,SEL,Y);
    `define K 8
    input [`K-1:0] DIN0,DIN1,DIN2,DIN3;
    input [3:0] SEL;
    output [`K-1:0] Y;

```

Example 3-3 Specifying the input bit width using text macros

3.1. Create function libraries

Specifying the bit width with text macros is useful when changing the bit widths of all modules at once. However, as described in “Naming conventions for including file, parameter and define”, care should be taken when using *define statements* to avoid complication. When using parameters, you can read in the parameter file using *include*.

In order to use a function library with variable bit widths at logic synthesis, you need to load the function library as a template in advance. Once the data is loaded, it can be saved in a working directory with an object format appropriate for specific tools. See “3.2.4. Use # (value) when overriding parameters from an upper level” for a usage with synthesis tools.

RTqualify performs following checks.

3151 (N)	An input/output port bit width is not parameterized. (excluded from check as default setting)
----------	--

3.1.6. Parameter description using *'ifdef* (Verilog only)

- | | |
|--|-----------|
| [1] Parameter may be described by using <i>'ifdef</i> (Verilog only) | reference |
| [2] <i>define names</i> defined by <i>'ifdef</i> selects simulation description or circuit description within one description (Verilog only) | reference |

Example Code

```

`ifdef K24
    always @(posedge CLK)
        if(EN==P_POS24)
            D <= A;
`else
    always @(posedge CLK)
        if(EN==P_POS16)
            D <= B;
`endif
`ifdef K24
    always @(posedge CLK)
        if(EN==P_POS24 && CZERO)
            K <= GT_SKYLINE;
`endif
always @(posedge CLK or NRST)
    if(NRST)
        `ifdef K24
            A <= 24'hfffffff;
        `else
            A <= 16'hffff;
        `endif
    else if(

```

Example 3-4 Description using *'ifdef*

Explanation

ifdef is a compilation specifier for performing conditional compilation. It will compile a statement that follows *'ifdef* if the specified *define names* are defined using *'define* in the source files to be compiled together, and if not it enables statements that follow *'else*. In addition, you could use following options in the command line when using a compilation tool.

- “-d <define name>”
- “+define+<define name>”

(Options specification depends on the tools)

With Design Compiler *'ifdef* can be used in version 1998.02 or later.

When using *'ifdef*, set the environment variable of the Design Compiler to

```
hdlin_enable_vpp = true
```

then load it using the following command:

```
analyze -f verilog -d K24 <file_name>
elaborate <design> -lib WORK
```

The *'else item* is executed if the *define name* is not specified using -d. Use *'ifdef* if parameterize descriptions cannot be used.

3.1. Create function libraries

The latest standard IEEE-1364-2001 of Verilog-HDL supports *generate statements* similar to those in VHDL. It is better to use it after IEEE-1364-2001 is supported by each EDA tool. 'ifdef' is just for simple use, so it cannot be expected that it is supported by all logic synthesis tools in the future.

Verilog-2001 is the new standard authorized in 2001. Major enhancements are listed below.

1. Support for signed operation

```
reg signed [63:0] data;
wire signed [11:0] address;
```

2. Support for multi-dimensional array

```
reg [31:0] array_2 [0:255][0:15];
```

3. Additional syntax for sensitivity lists

```
always @* @(A,B,C)
```

4. ANSI type *port* declaration

```
module mux2 (output reg [7:0] y0,
             input wire [7:0] a1,
```

5. Support for attributes

```
(*parallel case*) case (1'b1) //one hot FSM
```

6. Support for random function that generates correct results irrespective of the OS or tool environment

7. Support for generates syntax

```
generate if ( ) ...;
           else
endgenerate
```

8. Support for library and configuration

```
config cfg4
  default liblist rtlLib gateLib;
  instance top.couif.u3 liblist gateLib;
endconfig
```

9. Support for constant functions

```
parameter RAM_SIZE = 1024;
input [address_cal(RAM_SIZE)-1: 0] address_bus;
```

Some simulation tools and synthesis tools are beginning to support Verilog-2001. However, it is still not fully supported across different EDA vendors, so it might be a good idea to wait a while before actually using Verilog-2001 in your designs.

3.2. Using function libraries

3.2.1. Manage libraries in a common directory (differ from VHDL)

- | | |
|--|-------------|
| [1] Create separate directories, then store a function library for each function | recommend 3 |
| [2] Only allow specific administrators to make modifications to function libraries | recommend 3 |
| [3] Place the common “.synopsys_dc.setup” file in a common directory also | recommend 3 |

Explanation

In the design of large-scale circuits, libraries of functions such as calculators, selectors, and so forth, written in RTL description, are used, along with other design resources that have already been designed. A directory that can be read by all of the designers should be set up, so that this type of function library is controlled therein. However, when a library such as this can be modified by everyone, there will be the potential for the library to be modified unintentionally. Because of this, modifications to the function library should be limited to specified administrators only.^[2]

Once function libraries are created, they should be managed as files created for each library, placed under a shared directory.^[1] Having the library as individual files rather than having the entire library in a single file makes it easier to manage the library on a file-by-file basis.

When logic synthesis is executed, they can be stored in the synthesis library. When using the Design Compiler, this can be done by creating a data storage directory under the synthesis execution directory;

```
unix> mkdir work
and
```

```
define_design_lib WORK -path ./work
```

be specified in the following file: .synopsys_dc.setup

If the function file is a parameter description (see “3.2.4. Use # (*value*) when overwriting parameters from an upper level”), this is the method that must be used. A method for storing objects in the library (in RTL code) is also provided for the simulation tools. The method is different for each individual simulation tool. Additionally, when they are managed as objects, there is a risk that those objects will not match the RTL code version.

It is recommended that the function library be controlled not as object but as RTL code. If it is necessary to store them as objects, it is probably best to generate objects from the RTL code each time a tool is launched.

3.2.2. Define global parameters in separate files (differ from VHDL)

- | | |
|---|-------------|
| [1] Describe constants by parameters as much as possible | recommend 3 |
| [2] Define <i>global parameters</i> relating to the overall design in a separate file | recommend 3 |
| [3] Read parameter files using <i>'include</i> (Verilog only) | recommend 3 |
| [4] Make the file names specified by <i>'include</i> into relative paths (../include/common.h) (Verilog only) | recommend 1 |
| [5] Do not nest text macros | recommend 2 |
| [6] Distinguish the <i>'include</i> files used in the overall design project from those for individual use (Verilog only) | reference |
| [7] Use only parameter in the overall design project (Do not use <i>'define</i>) | recommend 3 |

Example Code

```
//-----
// FILE NAME : localpara.h          --
// TYPE      : PACKAGE              --
// FUNCTION   : LOCAL package       --
// PRODUCER   : hasegawa            --
// DATE      : 97/07/01             --
//-----

//-----
// Declare constants --
//-----
parameter  SLTLEN      = 4'd10;      // Slot counter bit width
parameter  RTMLEN      = 'd3;        // TMGCR register bit width
parameter  ADRLLEN     = 'd8;        // Address bit width
parameter  STLEN       = 'd3;        // State machine bit width
parameter  P_STVOEN1   = 10'd0;      // First half audio data rise
parameter  P_ENDVOEN1  = 10'd111;    // First half audio data fall
parameter  P_STVOEN2   = 10'd168;    // Last half audio data rise
parameter  P_ENDVOEN2  = 10'd279;    // Last half audio data fall
parameter  P_STCDATEN  = 10'd112;    // Control data rise
parameter  P_ENDCDATEN = 10'd167;    // Control data fall
parameter  P_ENDSLTC   = 10'd836;    // Slot counter value

//-----
// Declare addresses --
//-----
parameter[7:0] P_ATMGCR = 8'b00000000; // TMG control register
parameter[7:0] P_AS LTCR0 = 8'b00000010; // SLTC register :
// Read slot counter
```

Example 3-5 Local parameter declaration file

Explanation

Constants should be described with parameters as much as possible to handle bit width change or to share with other designers.^[1]

It makes debugging easier when the values are put in a place as parameters. It is not necessary to parameterize numeric values of 0 to 7, and vector values of all 0 and all 1.

Create a separate file for common parameter management in the design project, specify its name using the *'include*, and then read it from the other files.^{[2] [3]} Reading by using *'include* makes it possible to shorten the time required to make a parameter description. Moreover, when changing the parameter values, there is no need to change all of the associated modules. All that is needed is to change the parameter files.

A sample *'include* description is shown below.

```
'include "../include/commpara.h" // Common constant
'include "../rtl/localpara.h"    // Local constant
```

For the file specified by *'include*, path should be described as relative path and that it returns to the directory which is 1 level higher. Please refer to "3.5.1. Create a directory for each objective" for details. Execution of simulation and logic synthesis are done in separate directories. It is hazardous if simply using "commpara.h" since another file may be called.

In a large scale design, RTL descriptions are stored in some places. With this type of design, there is a limit in specifying the file using relative path. In this case, options of each tool can be used to specify file name. File name can be specified by +incdir+<directory name> with most of Verilog simulators.^[4]

The situation becomes complex if the *'include* has to read many files. If *'includes* have only a *constant* declaration, we recommend reading two files or so as shown above. One file is the include file that is used in common in the design group, and the other is the include file used only by individual designers. It may be good idea to name the *'include* file used by an individual designer "<an individual name>para.h".^[6] In the case of Verilog-HDL, both define or parameter can be used for constants. However, mixing them can be complicated, so it would be advisable to concentrate and use just one.

Do not nest *'define* (text macro) definitions^[5]. *'define* definitions are subject to text replacement, and syntax checks are not performed.

In the following definitions, VLEN will be $1024 + 128 = 1152$

```
'define VDRS_SET 1024
'define DDRS_SPLIT 128
'define VLEN 'VDRS_SET+'DDRS_SPLIT
```

In the example above, if an error is made in the VLEN line that the '+' sign is omitted, then

```
'define VLEN 'VDRS_SETDDRS_SPLIT
```

will end up as "1024128" a completely different value will be used. If nested *'define* definitions were allowed, it would not be possible to check for these types of problems. It would be extremely risky to allow nested *'define* definitions without some sort of constraint.

3.2. Using function libraries

When describing the bit width, it may be necessary to pass the bit width information between modules explained in “3.1.5. Parameterize the array range of module I/O”. This is a particularly effective way to describe function libraries. Parameters must be used for this passing. It may be better to use parameters for the constant that are used by the entire design group.^[7]

```
//-----
// FILE NAME : commpara.h          --
// TYPE      : PACKAGE             --
// FUNCTION   : Common constant    --
// PRODUCER   : hdlab              --
// DATE      : 96/07/01            --
//-----

//-----
// Declare constants  --
//-----
parameter P_FALLI  = 1'b0; // module para falling edge
parameter P_LOWI   = 1'b0; // module para Low level
parameter P_HIGHI  = 1'b1; // module para High level
parameter P_LOWS   = 1'b0; // Low level
parameter P_HIGHS  = 1'b1; // High level

parameter P_ONSN   = 1'b0; // Meaning is ON, expresses Active, but actual signal is 0
parameter P_OFFSN  = 1'b1; // Meaning OFF, expresses Inactive, but actual signal is 1

parameter P_FIRST  = 3'b001; // State machine status expression
parameter P_SECOND = 3'b011; // State machine status expression
parameter P_THIRD  = 3'b010; // State machine status expression
parameter P_FOURTH = 3'b110; // State machine status expression
parameter P_FIFTH  = 3'b100; // State machine status expression

parameter[7:0] P_BIT0=8'b00000001;
parameter[7:0] P_BIT1=8'b00000010;
parameter[7:0] P_BIT2=8'b00000100;
parameter[7:0] P_BIT3=8'b00001000;
parameter[7:0] P_BIT4=8'b00010000;
parameter[7:0] P_BIT5=8'b00100000;
parameter[7:0] P_BIT6=8'b01000000;
parameter[7:0] P_BIT7=8'b10000000;
```

Example 3-6 Global *parameter* declaration

RTqualify performs following check.

- | | |
|----------|--|
| 3221(W3) | Whenever possible, describe constants using parameter.
(excluded from check as default setting) |
| 3222(W3) | <i>Parameters</i> and <i>defines</i> are not described in independent files.
(excluded from check as default setting) |
| 3224(W1) | An include file is not specified using a relative path from the first layer
in file structure. |
| 3225(W2) | There are nested <i>defines</i> . |
| 3227(W3) | A <i>define</i> is used. (excluded from check as default setting)
The check for <i>define</i> is also available in 1144a and 1144b. |

3.2.3. Connect ports by name for component instantiations

- | | | |
|-----|--|-----------|
| [1] | For component instantiations, <i>connect ports by name</i> connections, not <i>by ordered list</i> | mandatory |
| [2] | Match the bit width of the component port and the bit width of the net to be connected | mandatory |

Example Code

Port name connection

```
CNT8 CNT8(.CLK(CLK1), .RST(RESET3), .CARRYIN(CI1),
           .CARRYOUT(DATA1), .DOUT(CI2));
```

Port order connection

```
CNT8 CNT8(CLK1, RESET3, CI1, DATA1, CI2);
```

This connection is prohibited

Example 3-7 Lower level block connection

Explanation

There are two types of connections to the lower components. The first is the *connection of ports by name*, which clearly describes the correlation between the component port and the connected *net name*. The second is the *connection of ports by ordered list* that describes the net in the port description order of the lower components.

With the *connection of ports by name*, the description amount increases since the *port names* of the lower components are indicated, but the description will be easier to understand afterwards since it will be possible to match upper *net names* with the *port names*.^[1] With the *connection of ports by ordered list*, not as much effort is required to describe them since it is only necessary to describe the *net names*. However, incorrect connections may result. The net description order must match the lower component port order in the case of port order connection, but the net description order is irrelevant in the case of *connection of ports by name*.

The I/O ports of each level are frequently changed as designs progress. However, if ports are connected by ordered list first, port orders are often mistaken when making modifications. In order to prevent description errors, always connect ports by name.

When making connections, be sure that the bit width of the lower component port and the bit width of the net to be connected match.^[2]

RTqualify performs following check.

- | | |
|-----------|--|
| 3231 (W1) | Order-based connection is used in lower-level modules connections. |
| 3232 (E) | In lower-layer module connections, signals connected to ports have different bit widths. |

RMM regulates the above in 5.2.11.

3.2.4. Use # (value) when overwriting parameters from an upper level (differ from VHDL)

- | | |
|---|-------------|
| [1] Use #(value) to overwrite parameters from an upper level (Verilog only) | reference |
| [2] Specify all parameters of a lower level (Verilog only) | reference |
| [3] Do not use <i>defparam statements</i> (Verilog only) | recommend 1 |

Example Code

```
SEL4TO1 #(32) MSEL1(.DIN0(DIN), .DIN1(AIN), .DIN2(BIN),
                  .DIN3(CIN), .SEL(SEL1), .Y(Y1));
SEL4TO1 #(16,8) MSEL2(.DIN0(LIN), .DIN1(RIN), .DIN2(SIN),
                  .DIN3(TIN), .SEL(SEL2), .Y(Y2));
```

Example 3-8 Overwriting parameters from an upper level

Explanation

Parameters defined in lower components can be overwritten from upper levels.^[1] Parameter overwriting is executed by specifying '#' and then the value to be overwritten when calling the lower components.^[2]

'#' is also used for expressing the delay value, but it is used as parameter overwriting when calling components. When multiple parameters exist in a component, specify the multiple values marking off the parameters by ',' in the order in which the parameters were declared.

For Design Compiler, registration to the synthesis library is required in order to replace the lower-level components. (See "3.2.1. Manage libraries in a common directory"). Use the "analyze" command as shown below when storing a lower-level components:

```
analyze -f verilog SEL4TO1.v
```

If a registered design are being used in a higher-level, then the lower-level will be instantiated automatically when you "read" this level as shown below:

```
read -f verilog upperhier.v
```

Specify the following in order to change the bit length of a registered design:

```
elaborate SEL4TO1 -parameter 32,8
```

The specified parameter values will be added to the design. If, in a large-scale design, the synthesis of the lower-level and the upper level is performed separately, the connections will not be formed correctly because of the design names mismatch. While it is possible to change the design name using the logic synthesis tool, such an approach is not recommended because it makes the work more complicated.

The replacement of parameters from an upper-level should be limited to small-scale function libraries only; it should not be attempted for large-scale designs.

To synthesize the description of Example 3-8, the parameterized circuits must already exist as object files or they must be included in the design library. See “3.2.1. Manage libraries in a common directory” for more information on reading parameterized circuits. *defparam* can also be used for rewriting parameters. *defparam* specifies the *instance names* and the *parameter names*, then defines the override values.

```
defparam U1.K = 32;
```

While *defparam* has the benefit of making, it possible to replace the parameter values all at once, it should not be used because some logic synthesis tools do not support *defparam*.^[3]

RTqualify performs following check.

- 3242a(W3) There is a parameter not overwritten using '#'.
(excluded from check as default setting)
- 3242b(W2) There are too many parameter value assignments.

3.2.5. The function library of an arithmetic operation improves data path design performance

[1]	Using a library described by a logical operation for selector improves speed	reference
[2]	Use a function library described by a logical operation for a multiplier with 16 bits or more output	recommend 1
[3]	Using a function library for multiple arithmetic operations improves speed and decreases the area	recommend 3
[4]	Use an exclusive tool when including a multiplier and multiple arithmetic operations	reference

Explanation

When it comes to design of the data path portion, a style that has the parts used by the data path as a function library and that makes *module instantiations* of these parts is linked to improved speed and decreased area.^[1]

For instance, using a library described by a logical operation, as explained in “3.1.5. Parameterize array range of module I/O” for such parts as a selector improves the speed and decreases the area.

Logic synthesis tools produce high quality performance in the case of adders and subtractors, but do not generate high efficiency when it comes to multipliers. You should not generate a multiplier with an output of at least 16 or more bits by logic synthesis with ‘*’.^[2] Multiplier is explained also in “[5]Do not infer large multiplier by the RTL description but describe the contents of multiplier by logical operation” of “2.10.6. Notes on arithmetic operations”.

In those designs in which the performance of area and speed are critical, an operation library described by a logical operation should be used for all the multipliers and multiple arithmetic operations regardless of how large the scale is.^[3] The current logic synthesis tools process operators individually. In fact, 3 adders or MAC (Multiply and accumulate) can be improved from 1.3 times to 1.8 times in speed by optimization of the carry signal, which goes through 2 operations without processing operations respectively.

Logic synthesis tools generate high performance adders for both the ripple carry type and carry look ahead type. However, the performance of the function for selecting between the two of them is not so good. Taking advantage of the logic synthesis tool technique can be used when making this selection. It may be better though to select other tools or to use *module instantiations* frequently.

The design save by logical operation does not necessarily become gate circuit with the best structure at the time of logic synthesis. It is preferable that it is described with gates, but synthesizing the logical operator would not decrease performance greatly as long as the inside is hierarchized to a certain extent.

When using the netlist described in the gate circuit (or the hard macro of a particular vendor), it is better to register it in the logic synthesis tool side.

However, this method is very complicated and not applicable to other design tools. Even when using hard macro, instance described by logical expression instead of hard macro should be called in RTL and then replaced with hard macro at gate level.

In those designs including many arithmetical operations, exclusive tools such as Module Compiler are often used. Module Compiler generates multiple arithmetic operation circuits efficiently by taking a combination of operators into account. Moreover, as a single multiplier generates a high performance multiplier, it reinforces the improvement of speed and the decrease of area.^[4] Also, a generated operation circuit at a logical operation level can be output.

It is also a good method to use the circuit generated by Module Compiler as a function library.

RTqualify checks the following:

3252 (W2) There is a multiplier using a '*' operator where output exceeds <N=15> bits

Explanation

The LSI differs from the design of FPGA circuits in that it is necessary to perform shipping tests on the integrated circuits after manufacturing in order to insure quality. The shipping test is not required for the FPGA because it is performed as LSI at the time of n shipping from FPGA manufacturers.

The LSI shipping tests differ from the functional tests that check the functions of the circuits, testing instead whether or not there are physical defects within the integrated circuit.

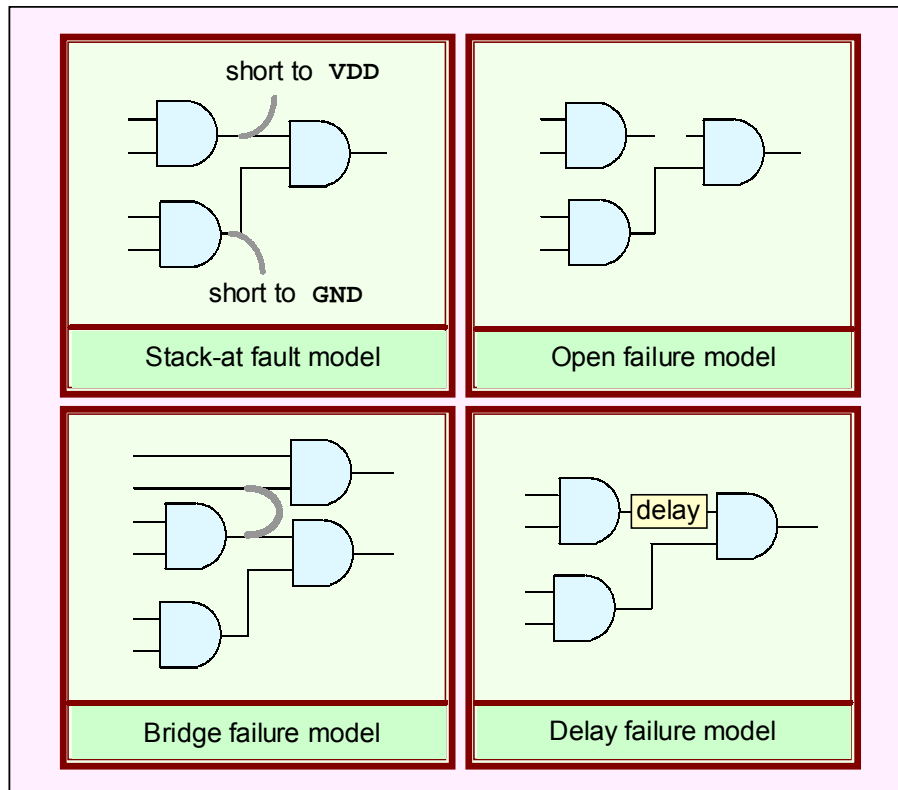


Figure 3-1 Models of four types of defects

A variety of different defects can occur in integrated circuits. These defects are simplified into “fault models”. The types of fault models include stack-at faults, open faults, bridge faults, and delay faults. (Figure 3-1) Note that these types of faults are not necessarily mutually independent.

At present, the probabilistic fault detection method uses the stack-at fault model, so the fault detection is done by assuming that of all the nodes there is a single point fixed on VDD or GND (a single stack-at fault). The state where a node is stack at VDD is described as “Stack-at-1”, and if the node is stack at GND it is termed a “Stack-at-0” fault. Open faults and bridge faults are investigated indirectly by looking at stack-at faults. Even if the detection is done using stack-at faults alone, if the ratio of nodes for which detection can be performed is near to 100%, then it is possible to discover most of the faults. However, there is no guarantee that all the faults will have been discovered, so this process is not perfect. Furthermore, the detection of delay faults requires the performance of tests on all active nodes at the actual operating speed.

As the scope of integrated circuits grows larger, the automatic generation of test patterns becomes more common. In this case, the testing is performed at the actual operating speed even on multi-cycle paths and false paths. Therefore, the actual shipping tests are often done at a speed that is less than the actual operating speed, or the testing is done at the actual operating speed on only a portion of the nodes. In such testing, it is not possible to detect all delay faults. The detection of these delay faults is currently a problem in fault detection.

Stack-at faults are detected using a test pattern. By inputting a test pattern provided from outside of the LSI, it is possible to change the values of specific nodes to 0 or 1. If the given node is stack-at 0 or 1, the output of the subsequent logical element will not change as expected. Ultimately, it is when a given node has stack-at 0 or 1 and this degradation ultimately propagates to outside of the LSI that a fault is discovered in the node. In the shipping test for LSI manufacturing, the approach taken to detecting the internal fault is tied to reducing the probability of shipping manufacturing defects.

At present, there is the need to have a fault coverage rate (i.e., the percentage of nodes for which detection is possible) of at least 97% (although opinions regarding this value vary from manufacturer to manufacturer).

Until several years ago, the designers themselves were required to generate test patterns for detecting faults. The test patterns for detecting faults either used test patterns for functional verification, or specialized test patterns were created to detect faults. When the circuits grew larger, however, it became difficult to create manually test patterns with high fault coverage rate. Additionally, the test pattern used in RTL simulations in million-gate-scale designs would require substantially more than 10 million steps. These types of test patterns are too long and would take too much time in the shipping tests, and thus are not practical in actual fault detection.

At present, automatic test pattern generator (ATPG) tools are used to generate automatically test patterns with high fault coverage rates. The ATPG tools are able to generate test patterns with high fault coverage rates for combination circuits, but not for sequential circuits. Given this, the “scan” technique is used in order to use ATPG tools efficiently. Typical scan methods include multiplexed scans and level-sensitive scan designs (LSSD). With multiplexed scans, the scan paths (scan chains) are formed by the insertion of a multiplexer before the inputs of the FFs, and switching the multiplexer with the test signals. (Figure 3.2) In practice, it is after an EDA tool is used to replace a FF cell with a single cell that has a FF combined with the multiplexer (i.e., a scan cell), that the scan path is linked up.

A FF is connected to the subsequent FF by a scan path. The signal that is input into SCAN_IN is set sequentially into the FFs when the SCAN_SEL signal is 1. After a value has been set for a FF, SCAN_SEL is set to 0 and the FF is caused to operate for one clock cycle in its normal operating mode. The result of the operation is stored in the FF. Afterwards, SCAN_SEL is again set to 1, and the value stored in the FF is then output to the SCAN-OUT port.

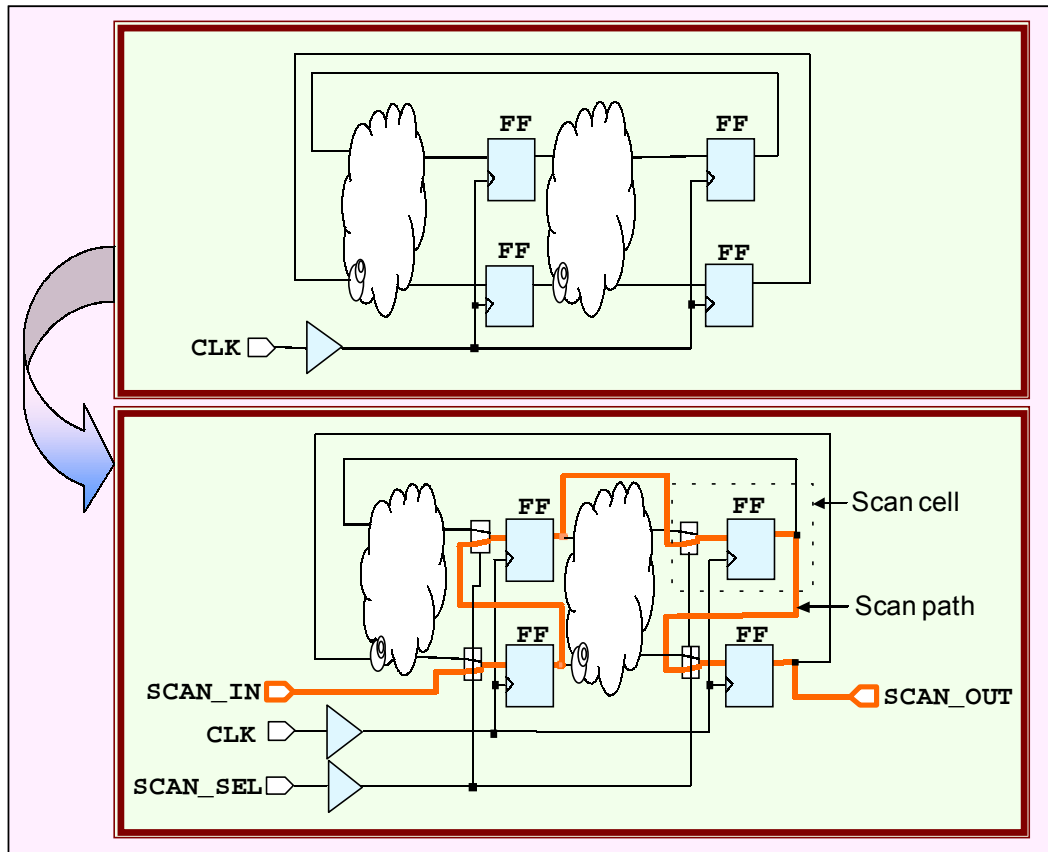


Figure 3-2 Conceptual diagram of the insertion of a scan circuit

Through the use of this method, all of the circuits in the LSI can be viewed as combinational and the ATPG tool will generate automatically a test pattern with high coverage. Consequently, in order to use this ATPG tool, those circuits that would present problems for this multiplexed scanning must be eliminated, and the number of nodes wherein stack-at faults cannot be detected must be reduced.

The level-sensitive scan design (LSSD) technique uses a different approach from the multiplexed-type scanning. While for convenience of this documentation will omit an explanation of the LSSD approach, the circuitry required for LSSD is the same as for the multiplexed type.

In this documentation, the insertion of circuitry for LSSD and for multiplexed scanning, for use by the ATPG tool, will be referred to as “scan insertion”.

In the design of large-scale logic circuits, it is important to be aware of the issues surrounding scan insertion and to be careful to insure that the design is compatible with ATPG tools; otherwise it will be difficult to increase the fault coverage rate. In large-scale LSIs, macro cells, such as RAM and ROM cells, are used in addition to gates. In this type of circuitry, the faults are detected either through the use of a technology known as “built-in self test”(BIST) or through the use of test patterns made by the designer (or provided by the ASIC vendor). The creation of a test pattern to discover faults from manufacturing is tremendously time-consuming. Manufacturing tests should be automated as much as possible. To simplify testing using techniques such as ATPG and BIST, the circuitry must be made so that the faults can be detected automatically. Creating this type of circuitry is known as “Design For Testability”(DFT).

3.3.1. Clocks and Resets for DFT

[1] The clocks must be directly controllable from external input ports	mandatory
[2] When there are two selectable clock systems, one clock system must be selected throughout testing	mandatory
[3] The output of random logic should not be used as a clock	recommend 1
[4] The reset for the FFs must be directly controllable from an external input port	mandatory

Explanation

When inserting scans, the most important consideration is to structure the circuitry so that the scan shift is performed safely during the scan test. A circuit structure wherein the scan shift is performed safely is a circuit structure wherein the clock pins of all FFs connected to the scan path at the time of the scan shift can be controlled directly from an external input port, and where none of the asynchronous set or reset signals for any of the FFs connected to the scan path will become active during the scan shift. If each circuit structure is one wherein an asynchronous set or reset signal becomes active during the scan shift, the data in the FF will be lost during the scan shift (i.e., the FF will be set or reset during the scan shift), meaning that the data will not be shifted as it should. Because of this, it is necessary to structure the circuit so that the asynchronous set and reset signals will be inactive during the scan shift.

* The clocks must be directly controllable from external input ports

In order to insert scans, it must be possible to control the clock pins of the FFs from an external input port during the scan test. If it is not possible to control directly the clock pins of the FFs from an external input port, then the scan insert tool will exclude the affected FFs from the scan. Note also that it will be difficult to detect faults using the ATPG tool for any parts for which scans have not been inserted.

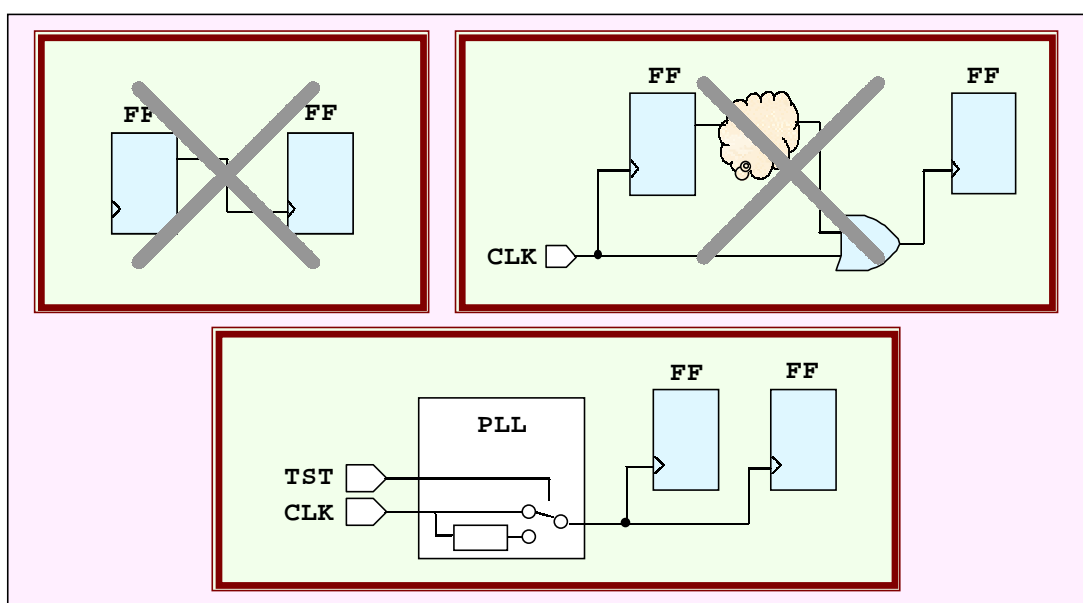


Figure 3-3 Examples of internally generated clocks, clocks that cannot be controlled directly, and PLLs

3.3. Design for Test(DFT)

FFs wherein the clock lines are gated, and FFs where, as is shown at the top of Figure 3-3, the output of a FF is connected to a FF clock pin, will be excluded from the scan. In such cases, it is necessary, for example, to switch the clock lines during the scan test so that they can be controlled directly from an external input port. Note that it is also not possible to insert scans when the output of a hard macro within the LSI acts as a clock. PLLs and DLLs are used as clock generator macros in LSIs. These macros usually have test ports such as shown by the bottom figure of Figure 3-3 so that the reference clock that is input into the PLL when in test mode is output without modification to the clock lines. If PLLs or DLLs not equipped with this type of switching circuitry are used, the designer will have to insert circuitry to switch to an external clock.

When it comes to “the clock must be directly controllable from an external input port”, see “3.3.5.DFT in Clock Lines” for specific measures to be taken.

- * When there are two selectable clock systems, one clock system must be selected throughout testing
Even if the clocks can be controlled from an external input port, if, as shown in Figure 3-4, it is possible to switch between two clocks, it will still not be possible to insert a scan. A test signal to control the select signal for the selector as shown in the figure must be used so that, during testing, the same clock will be selected throughout the entire process.

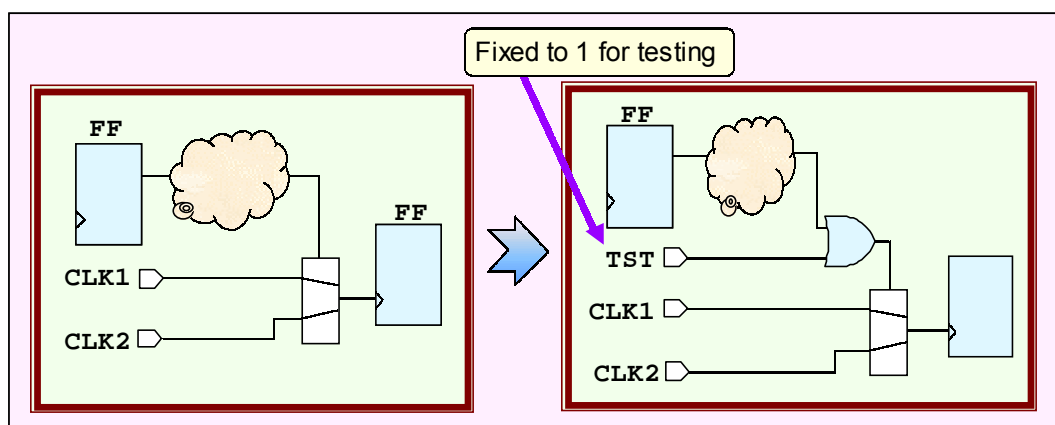


Figure 3-4 DFT for circuits wherein there are two selectable clock systems

- * The output of random logic should not be used as a clock

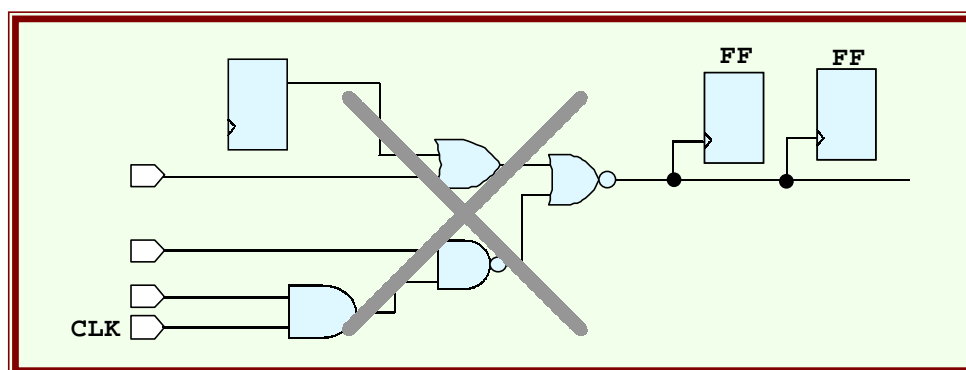


Figure 3-5 The output of random logic should not be used as a clock

As is shown in Figure 3-5, if the output of random logic is to be used as a clock, insert a selector at the final output of the random logic to make it possible to select an external clock.

The technique for using AND gates, OR gates, or selectors in clock lines is described in “3.3.5. DFT in Clock Lines”.

- * The reset for the FFs must be directly controllable from an external input port

In addition to paying attention to the clock systems when inserting scans, the designer must also pay attention to the reset lines. DFT for reset lines requires that they be structured so that no reset is applied to the FF during the scan shift, which would cause it to lose its data. While it is possible to insert scans even in cases where test signals, etc., are used to force the reset lines within the LSI to be inactive, doing so will make it impossible for the ATPG tool to detect faults in the reset lines. As a result, it is necessary for the reset lines to be directly controllable from external input ports.

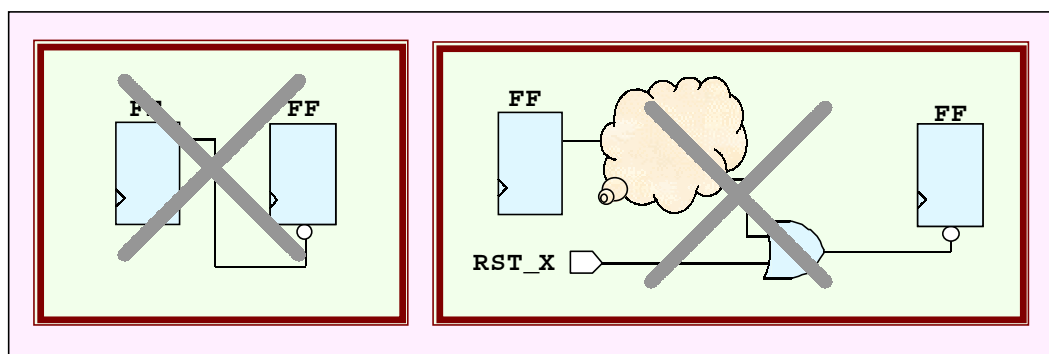


Figure 3-6 Internally generated resets and circuitry wherein there is no direct control from an external input port

As is shown on the left-hand side of Figure 3-6, when the output of a FF is connected to the reset pin of a FF, the FF will be excluded from scanning. In addition, in the right-hand figure in Figure 3-6, it is possible to disable the reset for the FF by tying RST_X to '1' and thus it is possible to insert a scan. However, in such cases, faults in the reset lines of the FF cannot be checked. As a result, it is necessary to make it possible to control these circuits from external input ports.

While generally resets are synchronized by inserting multiple stages of FFs as countermeasures for noise or timing problems, in such cases the resets cannot be controlled directly from an external port, and thus measures such as described above are required. The various problems pertaining to resets are described in “3.3.6.DFT in Reset Lines”.

3.3.2. Dealing With Hard Macros and Asynchronous Circuits

- | | |
|--|-------------|
| [1] Consider the use of original test circuits for asynchronous circuits and hard macros | recommend 1 |
| [2] Do not connect clock pins, reset pins, or tristate outputs to black boxes | recommend 3 |
| [3] Do not connect the outputs of a black box to clock pins, reset pins, or tristate-enable pins
- Prepare the hard macro library | mandatory |

Explanation

Scans cannot be inserted for asynchronous circuitry or for hard macros such as PLLs, DLLs, RAMs, or ROMs, and, for such hard macros and circuitry, it is also difficult to use the ATPG tool to generate test patterns able to detect faults. For these types of blocks, you must think about special fault detection tests. At present, in the design of large-scale LSI, a technology known as memory BIST is used to generate test patterns automatically within the LSIs for RAMs and ROMs. Generally, it is the ASIC vendors who think of the test methods for hard macros that include ROM and RAM. Consequently, when it comes to testing hard macros, all the designer must do is to design the interface for the test circuitry, following the directions from the ASIC vendor. However, when it comes to the interface with the hard macro during normal operations, the designer will still need to perform tests, and thus must pay attention to this point. More details are explained in “3.3.9. Handling Hard macros and Asynchronous Circuits, and Test Strategies for Large-scale ASICs”.

*** Consider the use of special test circuits for asynchronous circuits and hard macros**

When it comes to hard macros that are supplied by the ASIC vendor, normally it is the provider of the macros who must consider the testing within the hard macro. However, there are also those places where no automatic fault detection test environment is in place for the hard macros. Contact the ASIC vendor to investigate ways to handle the fault detection testing for hard macros.

*** Do not connect clock pins, reset pins, or tristate outputs to black boxes**

In some cases, a hard macro library may not exist when generating LSI design data. In such cases, define at least the inputs and outputs of the hard macro. When a clock is provided to a black box, it may be determined that there is a collision between the clock signal and the output of the black box. In some cases this may make it impossible to insert a scan. Even if it is possible to insert a scan, it may not be possible to generate a test pattern using the ATPG tool. Rather than using the black box the way it is in blocks such as these, either generate RTL code wherein only the inputs and outputs are defined, or obtain a library from the ASIC vendor.

*** Do not connect the outputs of a black box to clock pins, reset pins, or tristate-enable pins**

PLLs or modules that generate resets should not be left as black boxes. When it comes to macro cells such as PLLs, obtain the library for the macro cell from the ASIC vendor.

Even if there is a clock switching circuit for the test mode for the PLL, when the PLL is left as a black box, it may become impossible to determine whether or not the clock signal can be controlled directly from an external pin of the LSI, making it impossible to insert the scan. Additionally, even if it is possible to insert the scan, it still may not be possible to generate test patterns using the ATPG tool.

There are also cases wherein the initial reset uses a power-on reset module in the LSI. Even if the reset is activated on power-on, if there is no library, the DFT tool will not know. As is discussed in "3.3.6.DFT in Reset Lines", for this type of reset, switch to a reset signal that can be controlled directly from an external pin during testing.

In addition, when using a tristate circuit within the LSI, design the circuits so that the tristate enable signal is not output from a black box. In this type of circuit, it may not be possible to insert a scan, and even if a scan can be inserted, it may not be possible to generate test patterns using the ATPG tool.

3.3.3. Constraints on the Use of Flip-Flops

- [1] A clock must not be connected to the D input of a FF
- [2] Do not connect the input of a FF to VDD or GND

mandatory

recommend 2

Explanation

- * A clock must not be connected to the D input of a FF

Figure 3.7 is an example of dealing with the metastable problem as described in “1.5.1.[5] Latching the Clock Signal and Using It As an Enable Signal In Order to Prevent the Acceptance of Incorrect Data”, however, inputting a clock into the D input of a FF in this way may cause a problem for the ATPG tool.

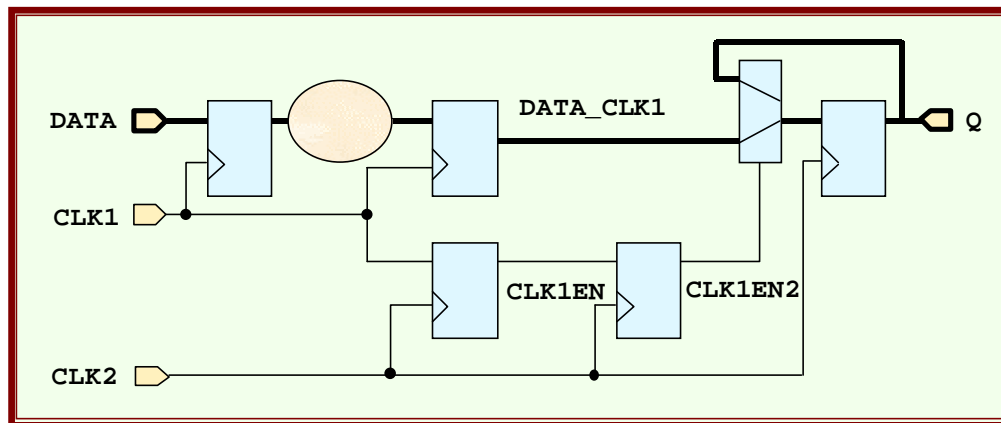


Figure 3-7 An example of a FF where a clock is input into the D input

Generally, the ATPG tool generates a test pattern while performing the simulation with zero delays. Because of this, inputting a clock into the D input of a FF runs a tremendous risk of generating an incorrect test pattern because of the racing problem. In “1.4.3.[4] Do Not Supply a Clock Signal to pins other than FF clock input pins (such as a D input)”, this was given a “recommend 1” because of the problem of timing analysis; however, this should be considered a critical requirement when it comes to the use of the ATPG tool. Note that when such a use is unavoidable, it is necessary to insert circuitry for DFT so that the clock will not be input into the D input of the FF when the ATPG tool is used.

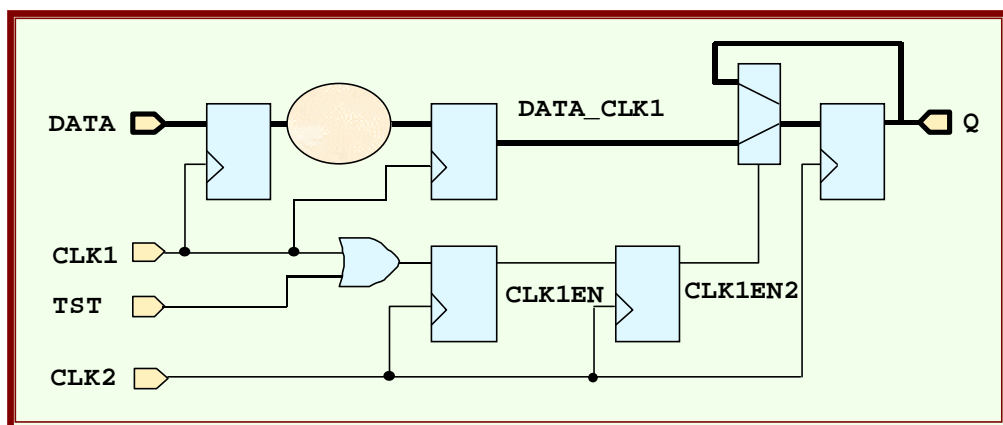


Figure 3-8 DFT for FFs where a clock is input into the D input

* Do not connect the input of a FF to VDD or GND

FFs where the input to the FF is fixed at a given voltage level, as shown in the left-hand side of Figure 3-9, should not be used. Modern scan insertion tools can insert scans even when there are FFs wherein the inputs are fixed to a specific value. If fault coverage rate is measured by ATPG and a list of undetected path is generated, the nodes, wherein fixed values are input, will be included in the list. Of course, because these nodes are fixed to either 1 or 0, it is not always the case. While this is only a dummy error in fault detection, it is better to not generate such errors. A FF wherein the input value is fixed to a specific value is, itself, redundant and thus not necessary, or it forms asynchronous circuit wherein the enable signal is connected to the CLK pin. This type of FF is undesirable in the design of synchronous circuits, and should not be used.

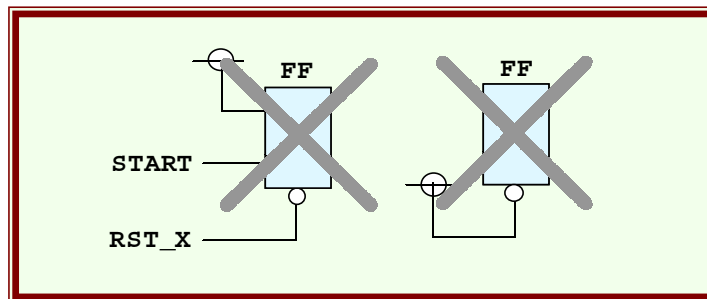


Figure 3-9 A FF wherein the input is fixed, and a FF where the reset pin is fixed

3.3.4. Cautions When Using Latches

- [1] Do not create feedback loop circuits including latches
- [2] Put the latches into a through state during testing
- [3] Use latches, which operate on inverted clocks, in 1 stage
- [4] When more than 2 stages of latches are used in a two-phase clock design, use test patterns created by the designer to detect faults

mandatory

recommend 1

recommend 2

recommend 3

Explanation

* Do not generate feedback loop circuits including latches

The latch must pass the input data throughout the time when the gate is active. When there is a feedback loop including a latch, such as in Figure 3-10, then it will be handled the same way as a combination loop during the time in which the gate is active. It may not be possible to insert a scan when there is this type of circuitry. Even if the scan can be inserted, it may not be possible to generate the test pattern using the ATPG tool. In practice, even if there is a circuit that operates as an inactive path or operates as a circuit that maintains an asynchronous state, such circuits must absolutely be avoided when using the ATPG tool.

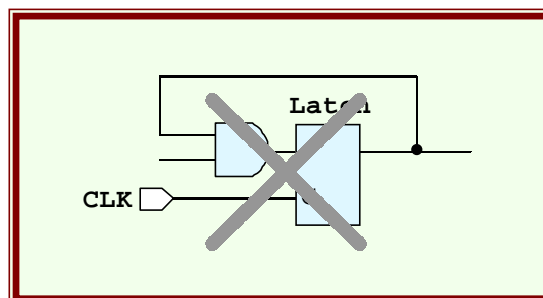


Figure 3-10 Latch feedback

* Put the latches into a through state during testing

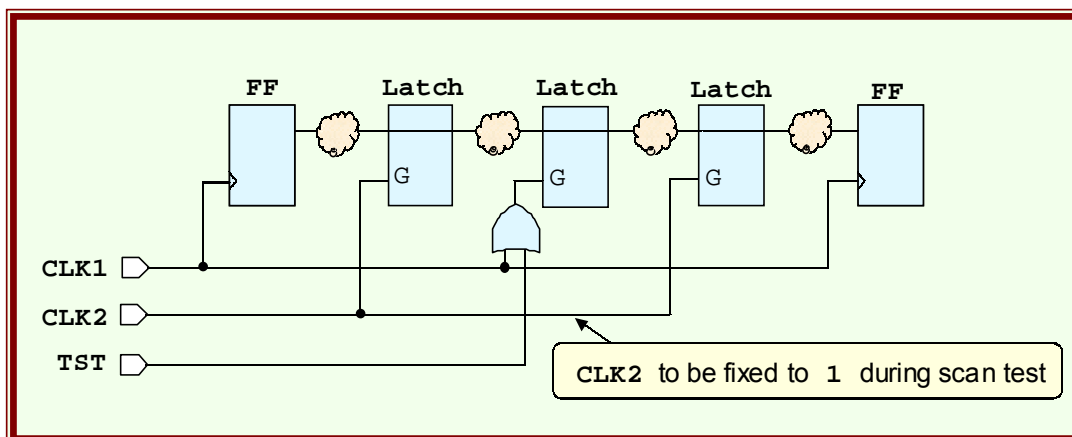


Figure 3-11 Detecting errors in the combinational circuit before and after the latch by causing the latch to be in a through state

Latch is normally not replaced by scan cell. Fault in logic part before and after latches can be detected by putting the latches into a through state during scan test. Latches can be used by the following methods; switching to through during test, fixing clock line with OR or AND and switching output by selector.

In a design with a small number of latch, fault detection rate can be improved by putting all the latches into through state. In a two-phase latch base design however, feed back loop circuit including latch may be generated if all the latches are put into through state. Long timing path may also be generated. Moreover, timing violation may occur during test.

When latches are used in a two-phase clock design, by tying the latch gate high during the scan testing, faults can be detected in the combination circuits before and after the latch. For a two-phase clock design such as shown in Figure 3-11, the latch can be placed in a through state by tying CLK2 to '1' and then having the gate for the latch which CLK1 is connected be tied to '1' by the test signal, making it possible to detect faults in the combination circuit before and after the latch.

- * Use latches, which operate on inverted clocks, in 1 stage

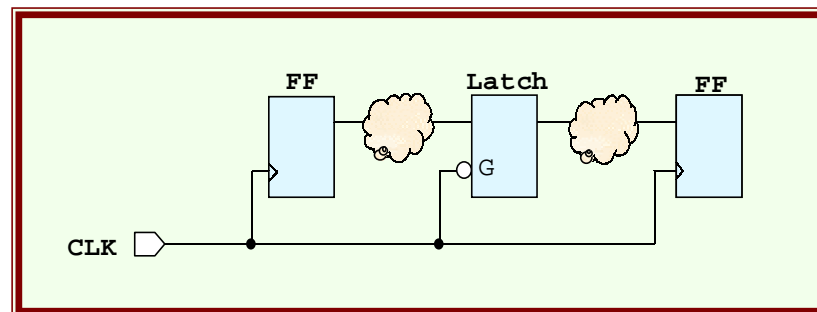


Figure 3-12 Only one latch for inversed clock is not a problem

As shown in Figure 3-12, when one latch is placed that operates with inverted clock, there is no need to perform any special measures for DFT. Latches that operate on inverted clock sample the data when the clock is low. By doing so, it is possible to detect faults in combination circuits between FFs even if no other special measures are taken.

When it is guaranteed that one latch is used that operates with inverted clock, the method described above, which is to put into through state, is not necessary.

- * When more than 2 stages of latches are used in a two-phase clock design, use test patterns created by the designer

When using latch in the design with two or three phases clock, there are often cases wherein the ATPG tools give up detecting faults in those particular parts. In such cases, the detection of faults in those parts must be done using test patterns generated by the designers.

In two-phase/three-phase design using latches, the designer must be aware that the designer will have to generate the test pattern or that there will be some degree of reduction in the test detection coverage.

3.3.5. DFT in Clock Lines

- | | |
|--|-------------|
| [1] During testing, switch to the same phase any clock lines having different phases | recommend 1 |
| [2] When the output of a FF is used as a clock, switch using a selector | recommend 1 |
| [3] Circuits that use OR gating of clocks and internally generated signals should be tied to a specific voltage level using an AND gate | recommend 1 |
| [4] Circuits that use AND gating of clocks and internally generated signals should be tied to a specific voltage level using an OR gate | recommend 1 |
| [5] Circuits wherein gating is performed on clocks and latch outputs should have an OR performed on the scan select and the stage prior to the latch | recommend 2 |
| [6] When gating a clock and the output of a FF, tie to a specific voltage level by performing AND gating with the stage after the FF | recommend 2 |
| [7] When the output of a latch is used as a clock, tie to a specific voltage level by performing OR gating with the latch clock input | recommend 2 |
| [8] In order to add logic to clock lines for safe DFT, generate the clock-generating modules in the top level | recommend 3 |

Explanation

In DFT, the requirement in “3.3.1.[1] The clocks must be directly controllable from external input ports”, is essential. Those clocks that cannot be controlled directly from the outside require switching to an external clock during testing. The simplest method by which to do this is by switching using a selector, such as shown below.

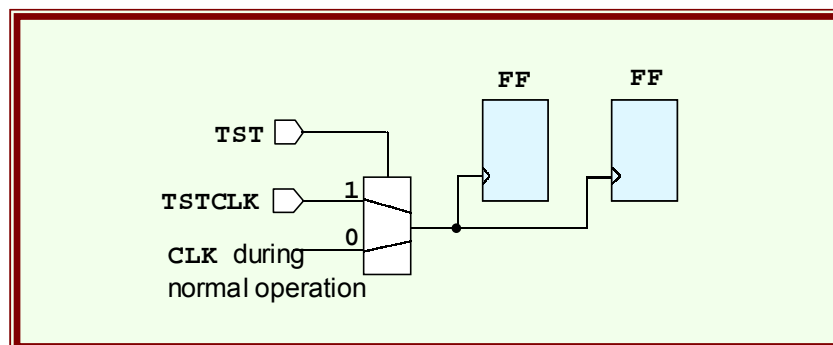


Figure 3-13 Switching to a test clock input from the outside during testing

However, when a selector is inserted into a clock line, gates are inserted into the clock line during normal operations as well. Because of this, there is also the approach taken in “3.3.1.[1] The clocks must be directly controllable from external input ports”, using a method that is somewhat simpler, depending on the circuit. In this section, specific measures by which to ensure that the “The clocks must be directly controllable from external input ports”, as required by 3.3.1.[1], for different types of circuits.

- * During testing, switch to the same phase any clock lines having different phases

When performing scan tests, all clocks must have the same phase. If an inverted clock is used in the circuitry, it is necessary to switch the clock as shown in Figure 3-14.

Some of the latest ATPG tools support inverted clock. However, still not many ATPG provide the support. Also, tools which rewire scan path during layout do not support inverted clock. Therefore, the scan path connected to scan FF with inverted clock can not be re-constructed and wire length may become long.

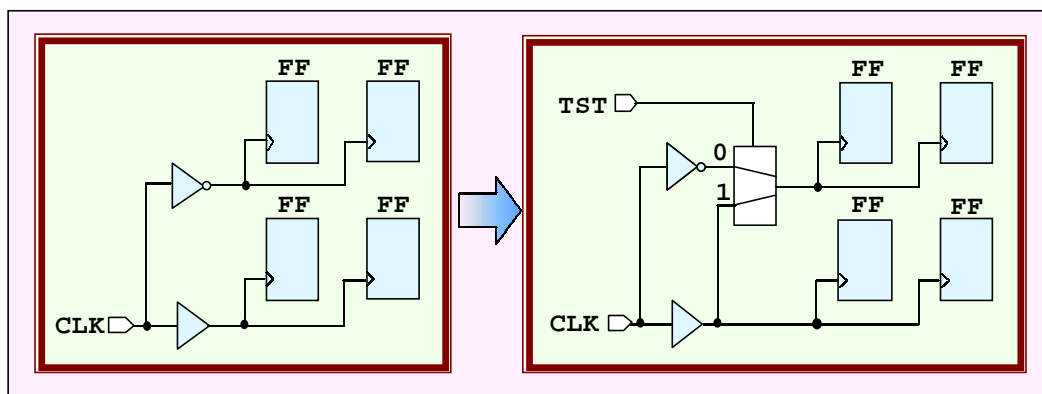


Figure 3-14 Switching an inverted clock to a non-inverted clock during testing

When it comes to latches (D latches, transparent latches), if the inverted clocks are used in only 1 stage, then there will be no problems in terms of DFT. When it comes to using latches. Please see “3.3.4.Cautions When Using Latches”.

- * When the output of a FF is used as a clock, switch using a selector

When the output of a FF is used as a clock, the clock pin of that FF cannot be controlled directly from the outside, and thus it is not compatible with scan insertion. Consequently, the ATPG tool cannot be used to detect faults in nodes pertaining to FFs connected to that clock line. Switch to a clock that is input from the outside during testing, as shown in Figure 3-15.

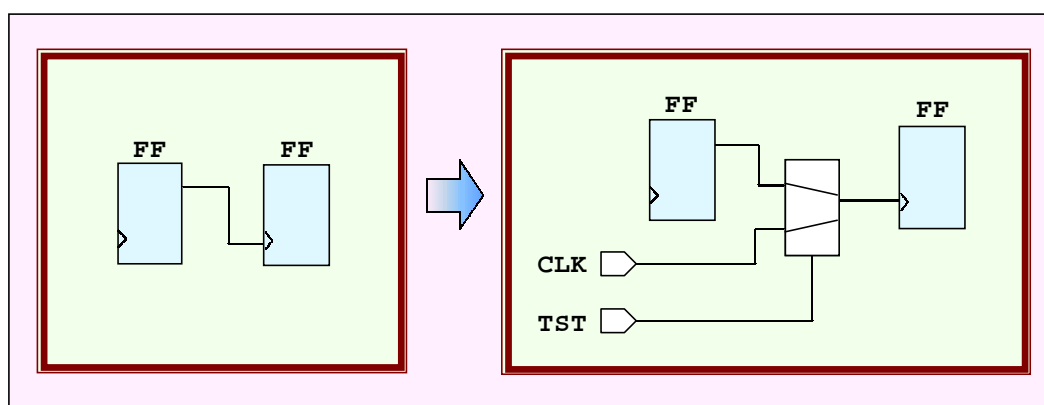


Figure 3-15 DFT for circuits wherein the output of the FF is used as a clock

3.3. Design for Test(DFT)

- * Circuits that use OR gating between clocks and internally generated signals should be fixed to a specific voltage level using an AND gate

In DFT, it is best if the number of clock lines is kept down, and the use of gated clocks could be avoided. However, use of gated clocks may be unavoidable in efforts to reduce power consumption.

There are two techniques for gated clocks. The first is the method of enabling the clock line through the use of an OR gate, and the second is that of enabling the clock line through the use of an AND gate.

As is shown in Figure 3-16, when the clock line is enabled using the OR gate, the enable signal EN must arrive within a half-cycle interval. If it does not arrive within a half-cycle interval, the clock pulse width will be different, and the circuit will not function safely.

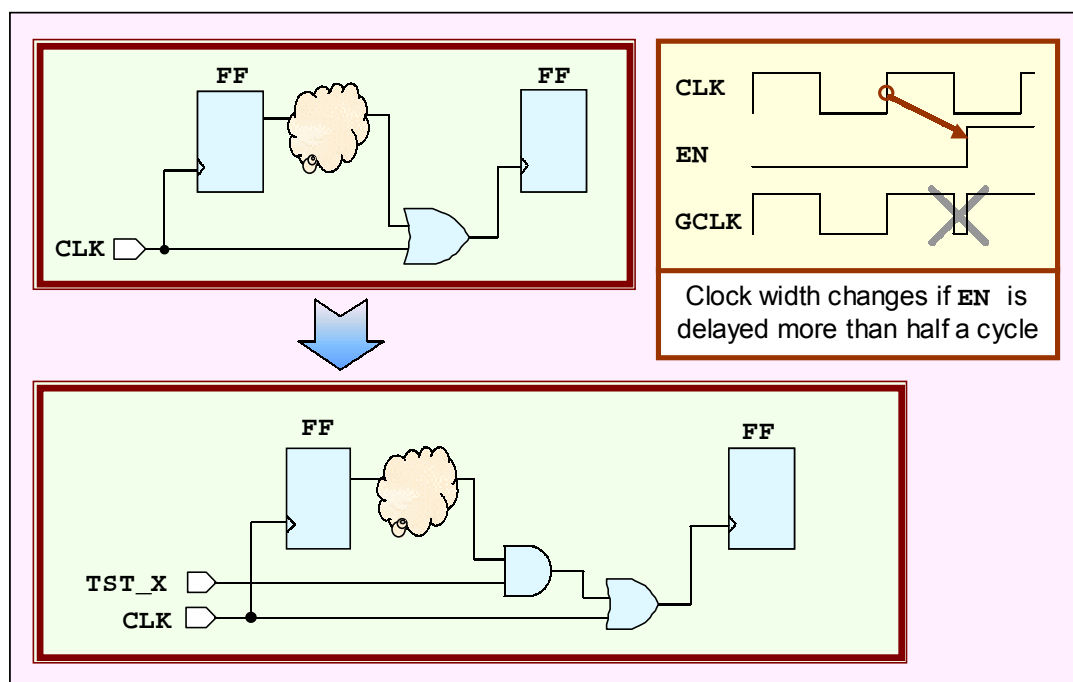


Figure 3-16 DFT for OR gated circuits

When OR gating is used, then when the EN signal goes to '1' at the time of the scan shift, the clock will no longer be input into the flip flop, so the scan shift will not be performed properly. At the time of scan shift, this enable signal must be tied to '0' by the test signal that is input from outside of the LSI. As shown in the top figure, the AND output can be fixed to a specific voltage by inserting an AND gate in the stage prior to the OR gate and then inputting '0' into one side.

- * Circuits that use AND gating between clocks and internally generated signals should be fixed to a specific voltage level using an OR gate

In AND gating an inverted clock serves as the base. When gating is performed on a positive clock signal, a hazard will be produced on the clock line, causing a malfunction. In gated clocks, either OR gating is performed with a positive clock, or AND gating is performed with an inverted clock. In order to prevent the occurrence of a

hazard on the clock line, one of these clock-gating circuits should be selected and the other should not be used at all.

As in OR gating, in AND gating the enable signal must arrive within a half-cycle interval. When in test mode, the test signal must cause the signal to be tied to a '1' before the AND gate, because this enable signal is fixed to a specific voltage value.

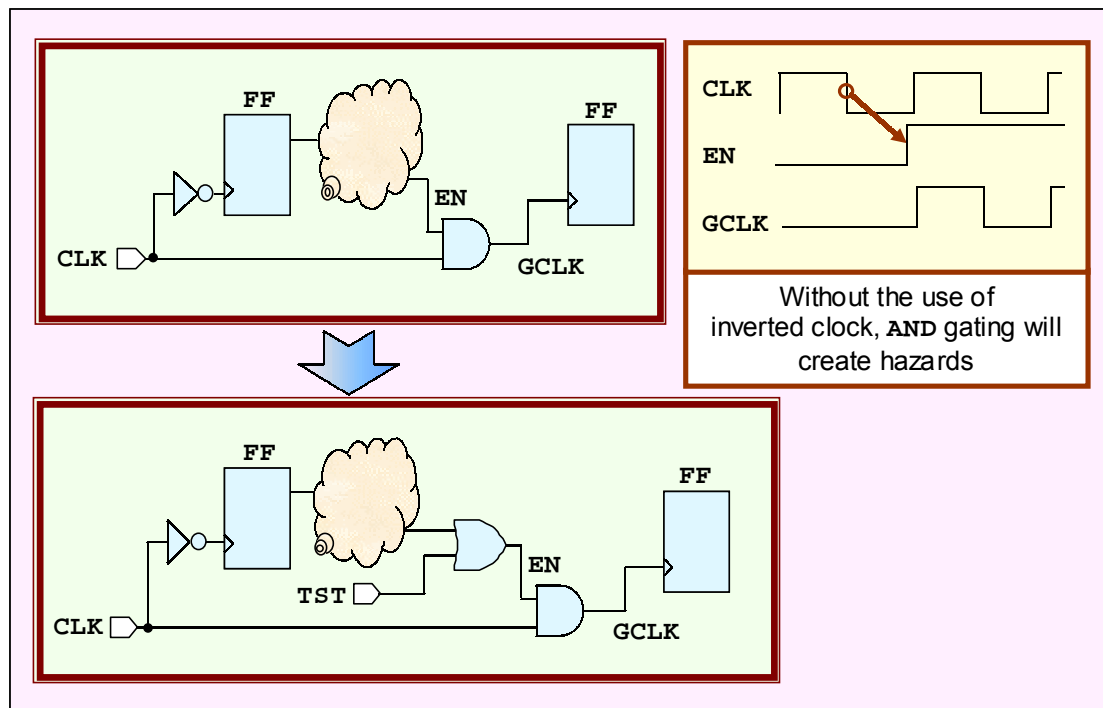


Figure 3-17 DFT for AND gated circuits

When AND gating is performed, the enable signal will be the output of a FF using an inverted clock. In the case shown in Figure 3-17, only the FFs for the enable signal will be different from the phase of the normal clock. Warnings may be output regarding these types of circuits in the scan insertion tool and/or the ATPG tool. If it becomes problematic with the tool that is used, it may be necessary to operate all clock lines with inverted clocks.

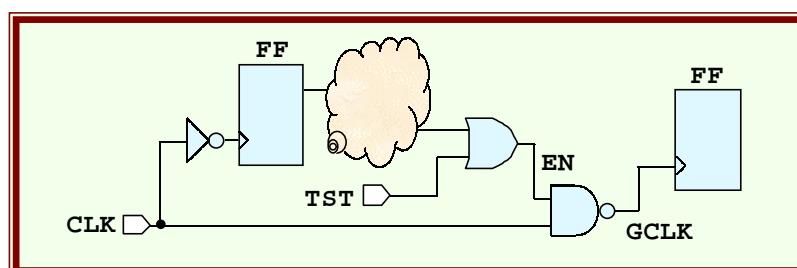


Figure 3-18 AND gating circuit for an inverted clock line

In Figure 3-18, both the FF that produces the enable signal and the FF that uses the GCLK as clock operate with inverted clocks. This is one possible way for a design wherein everything operates on a inverted clock. If the system is one wherein everything within the LSI operates on a positive clock, it would not be recommended that the circuits be operated by inverting specific clock lines for AND gating.

The DFT measures used for the OR gating and the AND gating described above have the benefit of making it easy to adjust the skew between different clocks. This is because there will not be too many gates inserted in the clock line. Of course, the DFT measures used could also be the switching technique mentioned earlier in this section, where the clock is switched for testing using a selector.

- * Circuits wherein gating is performed on clocks and latch outputs should be fixed by OR

As already discussed, the AND gating and OR gating techniques require the timing of the enable signal EN to be within the half-cycle interval, making the timing analysis more complicated. In addition to the above, there are also methods that use latches and FFs to produce gated clocks. When a latch or an inverted clock is placed prior to the AND gate as in Figure 3-19, the value of A1 does not change when the clock is a high pulse. The latch goes to a through state when the pulse goes low after the falling edge, at which time the value for EN propagates to A1. Consequently, if the AND with the clock signal is performed after the enable signal EN passes through the latch, then the enable signal will be accepted with the timing for the clock cycle being from the rising edge of the clock to the next rising edge of the clock.

Gated clocks using latches such as this normally fix latch output with test pin by OR. Besides this, some automatic test pattern generation tools support the method wherein an OR is performed on the SCAN_SEL signal prior to the latch. When SCAN_SEL is '1' (at the time of the scan shift), the output of the OR will always '1', so a '1' will always be input into the AND input. As a result, the CLK to the FF will not stop during the scan shift.

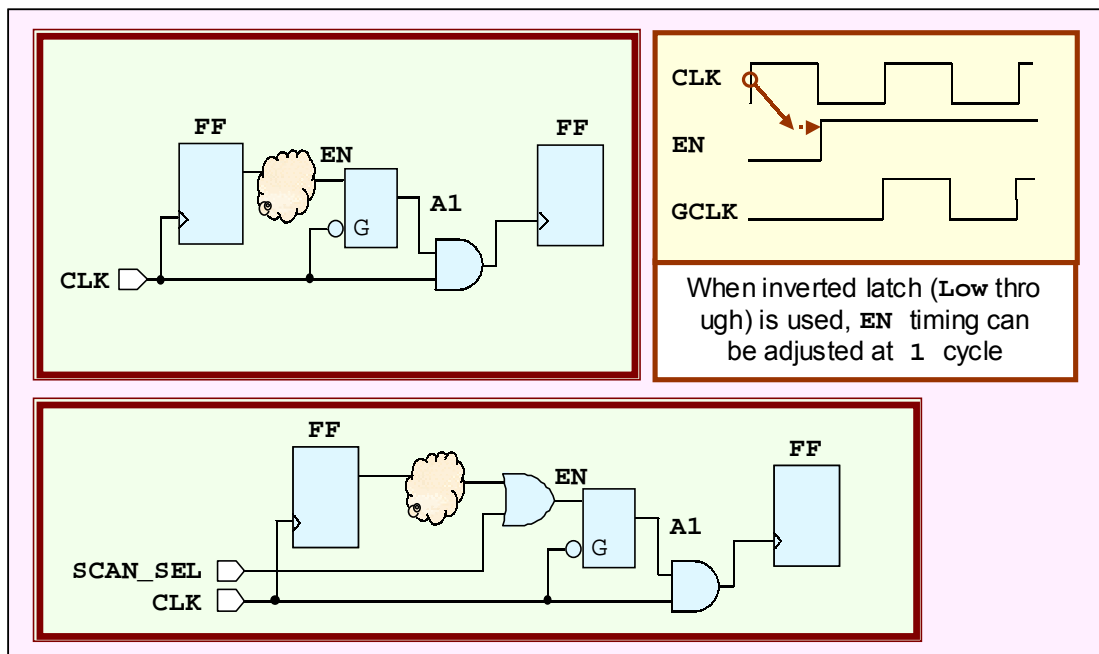


Figure 3-19 DFT for circuits wherein the clock and the latch output are AND gated

When SCAN_SEL is '0', then there will be a case where the OR output is '1' and a case where the OR output is '0'. If the OR output goes to '0' the next rising edge of the clock will be skipped.

The method using SCAN_SEL has higher fault coverage rate compared to the method fixing by test signal. However, attention should be paid since some ATPG tools may not recognize that SCAN_SEL is used.

- * When gating a clock and the output of a FF, tie to a specific voltage level by performing AND gating with the stage after the FF

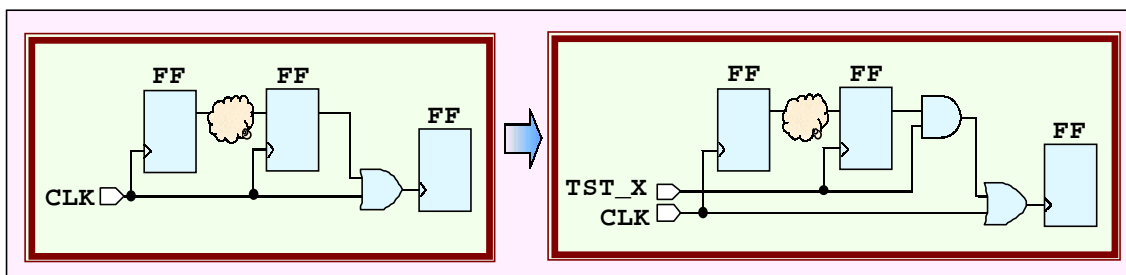


Figure 3-20 DFT for circuits where the clock and the FF output are OR gated

When OR gating is used, a FF is used as the previous stage, rather than a latch. There is a huge risk that there will be a change to the width of the rising edge of the clock when a hazard is produced in the AND gating. When OR gating is performed, an AND gate should be inserted in the previous stage. When, during the test, a '0' is input into one side of the AND gate, the output of the AND will go to 0, allowing a stable clock be supplied to the FF.

- * When the output of a latch is used as a clock, tie to a specific voltage level by performing OR gating with the latch clock input

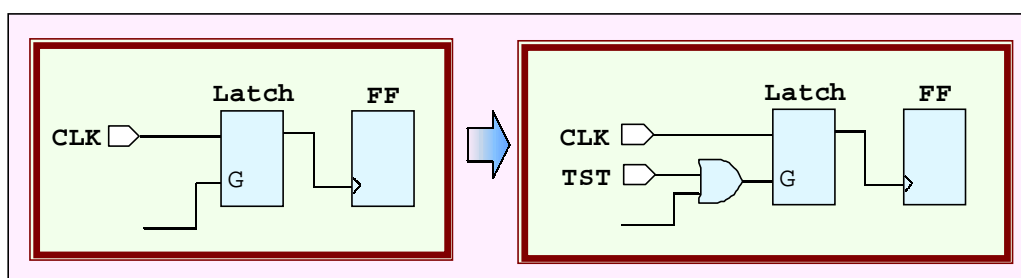


Figure 3-21 DFT for circuitry wherein the output of a latch is used as a clock

There are also methods to control the clock lines where a latch is used instead of a gated clock. Such methods are not recommended because they are not desirable in terms of timing analysis. If this method is being used to control the clock lines, be sure to set to '1' the signal that is input into the clock pin for the latch. In Figure 3-21 the test input and the latch clock input are be tied to a specific value through the use of an OR gate so that the signal always passes through the latch.

* In order to add logic to clock lines for safe DFT, generate the clock-generating modules in the top level

Be sure to have the clock-generating modules in the top level, and generate the gated clocks, the period-divided clocks, etc., therein. In the local levels, do not use gated clocks, do not use the outputs of FFs as clocks, and do not perform switching on clock lines. As was described in “3.3.1.Clocks and Resets for DFT”, all clocks must be able to be controlled from outside of the LSI during testing. These operations cannot be performed safely when there are many gated clocks in the local levels.

3.3.6. DFT in Reset Lines

- | | |
|--|-------------|
| [1] When inputting the output of random logic into an asynchronous set or reset pin, fix to a specific voltage using an AND gate | mandatory |
| [2] Do not mix clock lines and reset lines | mandatory |
| [3] Do not connect the output of a FF directly to the asynchronous set or reset pin of a FF | mandatory |
| [4] Do not connect the output of a latch directly to the asynchronous set or reset pin of a FF | mandatory |
| [5] In order to add DFT logic to a reset line safely, create the reset-generation module in the top level | recommend 3 |

Explanation

The requirements in “3.3.1. [4] The resets for the FFs must be directly controllable from an external input port”, are critical requirements for DFT. The ATPG tool does not understand power-on reset circuitry. Be sure to create circuits such that the resets of all FFs are enabled by inputting resets into ports that are external to the LSI. This section explains for various types circuits specific approaches that can be used to fulfill the requirements of “3.3.1.[4] The resets for the FFs must be directly controllable from an external input port”.

* When inputting the output of random logic into an asynchronous set or reset pin, fix to a specific voltage using an AND gate

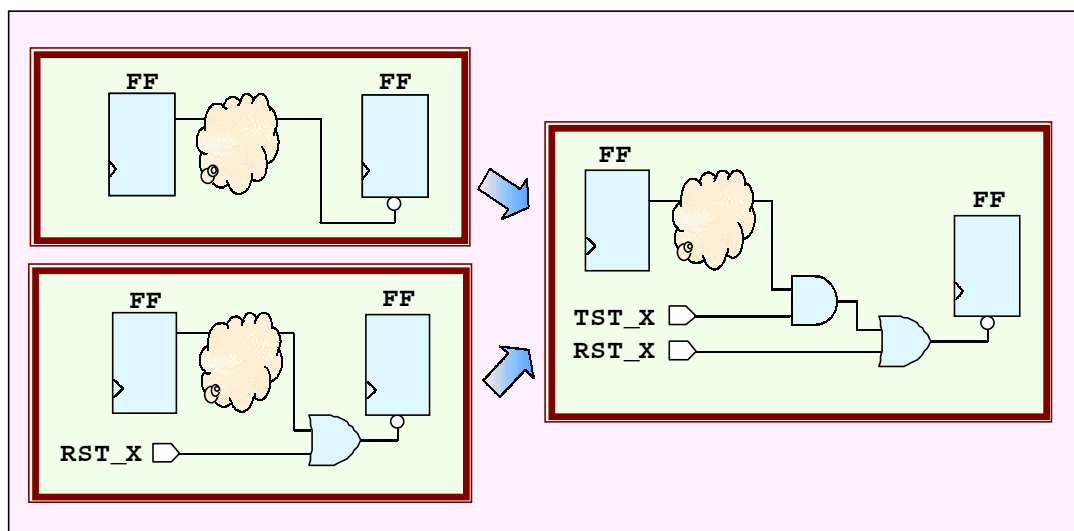


Figure 3-22 DFT for circuits wherein random logic provides the input into an asynchronous reset pin

Circuits wherein there is logic on an asynchronous reset line are not desirable. Specific DFT measures are required for circuitry wherein combination logic signals are input into asynchronous set or reset pins, and also required for circuitry wherein OR gating is performed on a reset input. By using an AND gate on the output of the random logic and the test signal (which is input from the external port), these inputs

3.3. Design for Test(DFT)

can be tied to '0' during testing. Next, the circuit can be controlled by an input from an external reset during testing through OR gating the reset signal that is input from outside of the LSI.

*** Do not mix clock lines and reset lines**

Do not mix clock lines and reset lines. In circuits where clock lines and reset lines are mixed, all FFs to which the CLK signal is connected will be excluded from scanning.

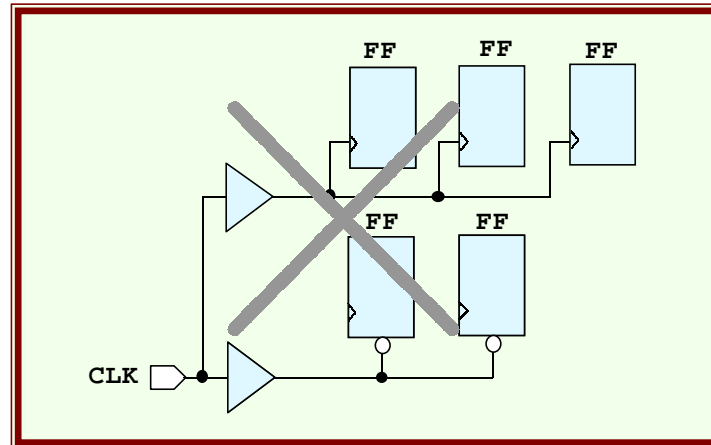


Figure 3-23 A circuit where clock lines and reset lines are mixed

Normally, clock lines and reset lines are not mixed in design. However, there are some rare cases that use a circuitry as shown in Figure 3-23, in an attempt to provide asynchronous control from outside the LSI. This type of CLK input is not a clock signal that will prove a clock constantly, but rather constitutes an asynchronous circuit wherein the status of the FF is changed by this signal, when the signal works as an enable signal that is used only very rarely. This type of high-risk design should not be used, even when no scan is to be inserted.

- * Do not connect the output of a FF directly to the asynchronous set or reset pin of a FF

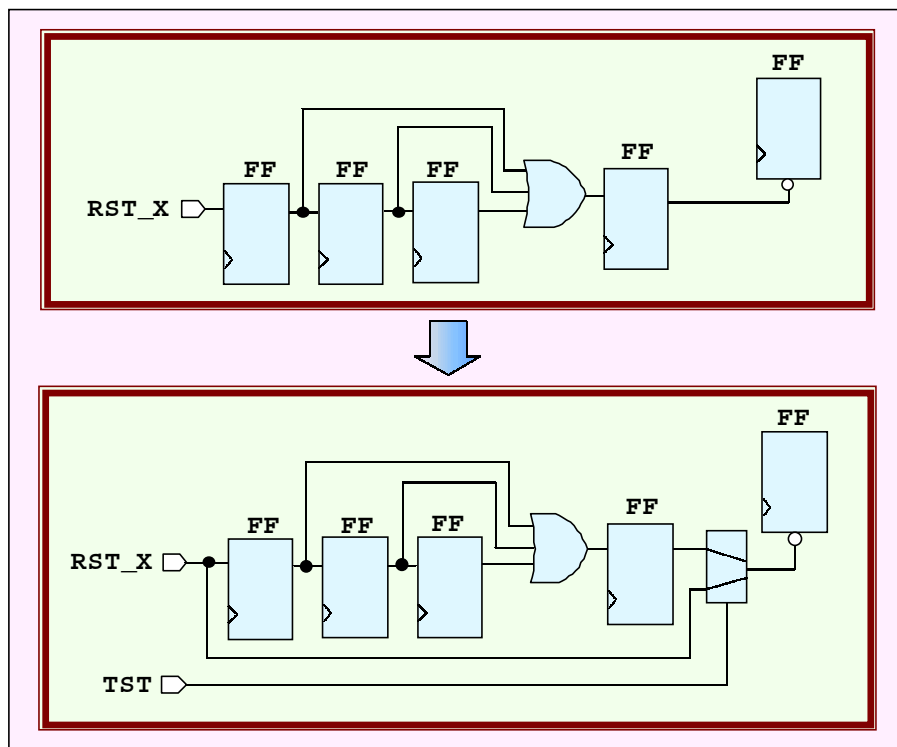


Figure 3-24 Noise countermeasure circuit for resets

Similar to the case discussed above where the output of random logic should not be input into an asynchronous set or reset input pin, in some cases the output of a FF is input directly into an asynchronous set or reset pin. As explained in “1.3.3.Cautions Pertaining to External Noise in the Initial Reset”, the reset signal is sometimes synchronized in order to control the reset timing, doing so as a noise prevention measure at the time of the layout.

For this type of circuitry, a selector should be inserted as shown in Figure 3-24 to switch to a reset signal that can be controlled directly from an external port, rather than using the synchronized reset signal.

- * Do not connect the output of a latch directly to the asynchronous set or reset pin of a FF
Having a latch on a reset line may cause an error in the check tools that pertain to DFT, or may cause the ATPG tools to malfunction. For this type of circuitry, tie the latch gate to '1' during testing, as shown in Figure 3-25, to place the latch in a through mode.

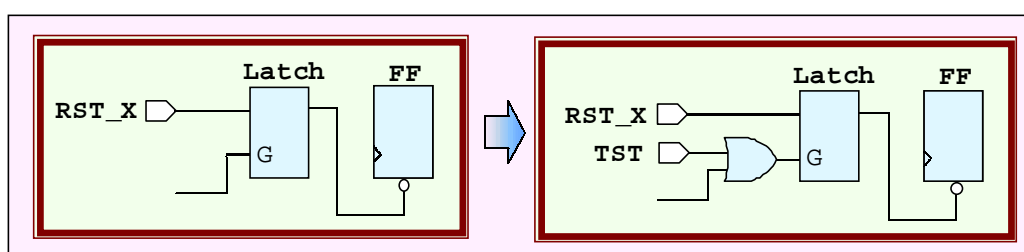


Figure 3-25 Countermeasure for the circuit where the output of a latch is input into a reset

3.3. Design for Test(DFT)

- * In order to add DFT logic to a reset line safely, create the reset-generation module in the top level

For resets, create the reset generator module in the top level and perform the measures for DFT and for synchronizing the resets therein. In local levels, do not use gated reset lines, do not use the output of FFs as resets, and do not perform switching on the reset lines. As described in “3.3.1.Clocks and Resets for DFT”, DFT requires that all resets be controllable from outside of the LSI during testing. If gated operations are performed in a local level, the operations will not be safe.

3.3.7. Handling of Different Clocks

[1] Measures to prevent clock skew between different clocks are required

recommend 1

[2] Insert a latch with inverted clock when transmitting between asynchronous clocks

recommend 3

Explanation

Generally LSIs have several different clocks. When there is a large number of clock lines, it becomes difficult to perform timing analysis and difficult to adjust the clock lines during layout. While the number of clock lines, of course, should be as small as possible, the use of many clock lines is often unavoidable in asynchronous interfaces to the outside.

* Measures to prevent clock skew between difference clocks are required

When performing tests on a circuit that uses different clocks, such as shown in Figure 3-26, using a scan chain, the same scan clock must be input into both CLK1 and CLK2. If this CLK1 and CLK2 are asynchronous in normal operation, there is no need to insure a hold time from CLK1 to CLK2. However, in order to detect stuck-at faults in nodes between CLK1 and CLK2, one must consider assuring the hold time between CLK1 and CLK2.

When there are many clock systems, it becomes difficult to adjust for skew during the layout. In such cases, it is necessary to align the clock arrival times between these asynchronous clocks. It is difficult to completely sequence the clock delay times between different clocks during the layout process. The delayed arrival times between the clocks should be made as near to each other as possible, and then, in the end, layout tools or logic synthesis tools should be used to insert buffers in order to insure the hold on the data lines or the scan path.

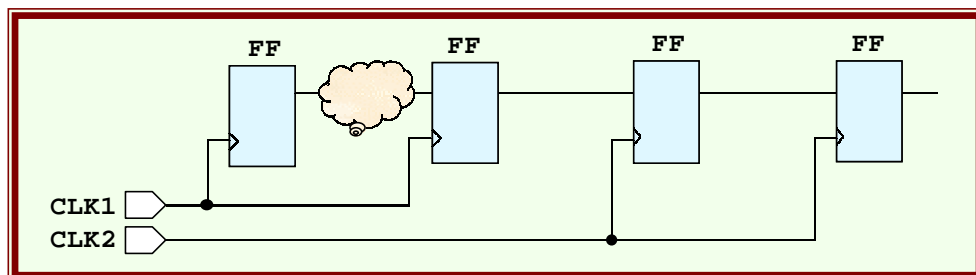


Figure 3-26 Handling the metastable problem between asynchronous clocks

* Insert a latch with inverted clock when transmitting between asynchronous clocks

One method for structuring a single scan chain between asynchronous clocks is that of using a latch with inverted clock in the data line, such as in Figure 3-27. If a latch with inverted clock is used, then the data will be held during the time in which the clock is high. If the clock is low, then the latch goes to a through state, causing the data to be transmitted. If the skew value in the clock line is no more than $\frac{1}{2}$ of a clock cycle, then even if the clock line arrival times are adjusted, there will be no need to insure the hold. However, even if these measures are used, it is still necessary to insure the hold time on the scan path.

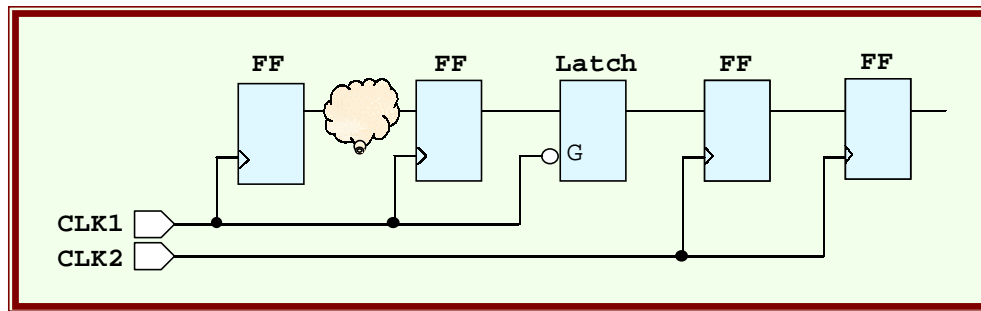


Figure 3-27 Inserting a latch with inverted clock between asynchronous clocks

In practice, it is difficult to sequence the clock line arrival times. Intentions to adjust the clock lines are often ultimately abandoned or forgotten. The insertion of these latch with inverted clocks between asynchronous clocks is recommended.

When scan are inserted in large-scale LSIs, inserting only a single scan line would mean that there would be many FFs chained together, making it difficult to keep the test vectors within the allowable range for the logic circuit tester using ATPG tools. Usually between 4 and 32 scan chains are created for LSIs in the range of about a million gates.

If there are only a few clock lines, then one method that might be used is to generate a scan chain for each of the clock lines; however, in large-scale LSIs with a large number of clock lines, this would be difficult. Additionally, when there is a correspondence between the clock lines and the scan chains, there are cases wherein the balance for each scan chain is lost.

When there are many clock lines, the scan chains must be combined to some degree. If there is no data transmitted between asynchronous clocks, then the measures in Figure 3-27 probably do not need to be taken. However, if the asynchronous clocks are on the same scan chain, then, as is shown in Figure 3-27, the scan data will be transmitted.

The issues described above must be considered when a scan is inserted. Recently some scan insertion tools available in the market can insert latches into the scan path as shown in Figure 3-27.

3.3.8. DFT for Tristate Circuits

- | | |
|--|-------------|
| [1] Tristate enable signals should be able to be fixed from an external input port | recommend 2 |
| [2] Tristate enable signals should be controllable directly from the outside or should be controlled by a decoder that is controlled directly from the outside | recommend 3 |
| [3] External bidirectional pins should be set during the scan shift | recommend 3 |

Explanation

When there is a tristate (unidirectional, or bidirectional) within the structure of the LSI or at an external port, this value must be set to a constant value during testing.

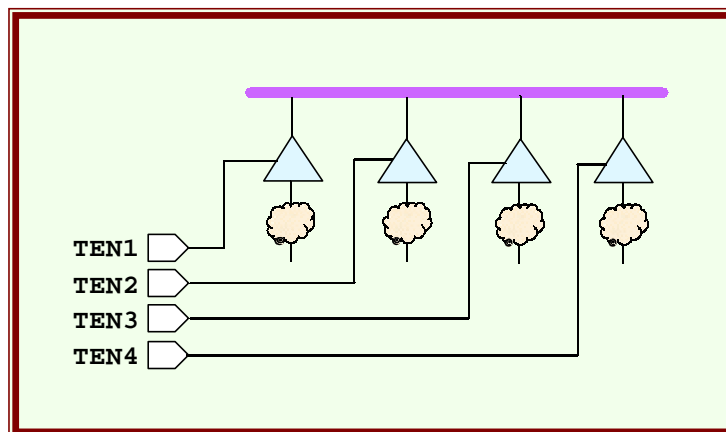


Figure 3-28 An ideal bidirectional circuit from the perspective of DFT

Tristate design that takes DFT into consideration is, ideally, a design wherein all of the enable signals can be controlled individually from outside of the LSI. With such circuitry, there will be no problems with ATPG tools.

- * Create circuit structure so that enable signal of tri-state can be fixed from an external input port

In actual bidirectional design, the tristate enable signal is usually generated within the LSI itself. In such cases, there is a method by which the tristate buffer enable pin can be tied to a specific value from outside of the LSI.

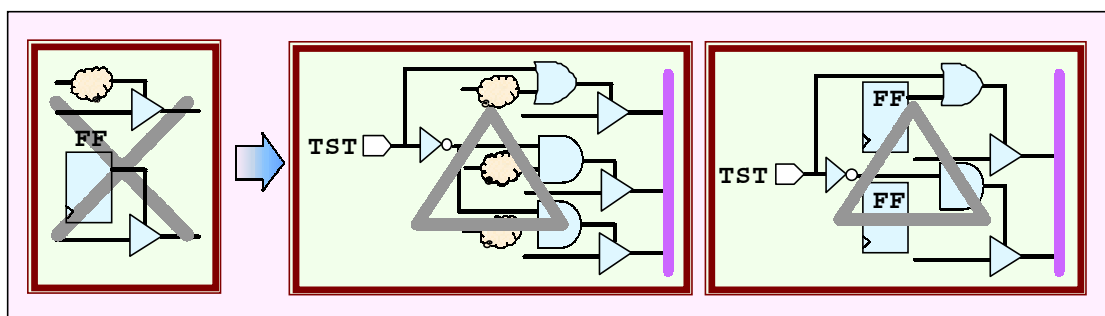


Figure 3-29 Circuit wherein the tristate output is tied to a specific value during testing

3.3. Design for Test(DFT)

In the circuit in Figure 3-29, when TST is '1', only one of the tristate buffers connected to the bi-directional bus goes into an ON state, and all of the other buffers go into an OFF state (i.e., high impedance output). Such circuitry eliminates collisions from the output of the tristate buffers during testing. However this method is, in the end, rather simplistic, and is not ideal for DFT. When only a specific tristate buffer is caused to produce an output during testing, it is not possible to detect faults in the other driver outputs.

- * Tristate enable signals should be controllable directly from the outside or should be controlled by a decoder that is controlled directly from the outside

The circuit shown in Figure 3-30 is ideal. When it comes to circuits wherein the tristate elements are controlled internally, it is ideal to create the circuitry shown in Figure 3-29. All of the tristate buffer enable signals can be controlled individually during testing from outside of the LSI.

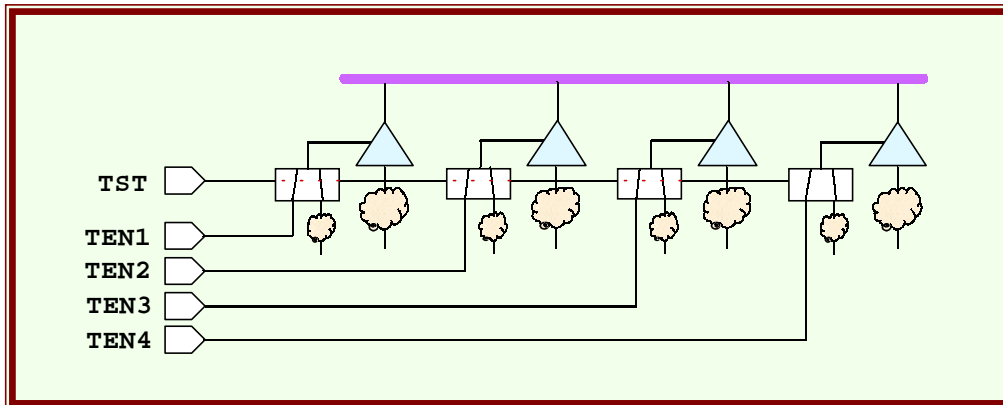


Figure 3-30 All of the enable signals can be controlled individually from the outside during testing

In LSI design, it is desirable to reduce the number of I/O pins as far as possible. In LSIs containing many tristate circuits, it is, as a practical matter, impossible to provide enable signals to all of the tristate elements from outside of the LSI. In such cases, one method that is used broadly is that of designing using decoders that control the tristate elements internally.

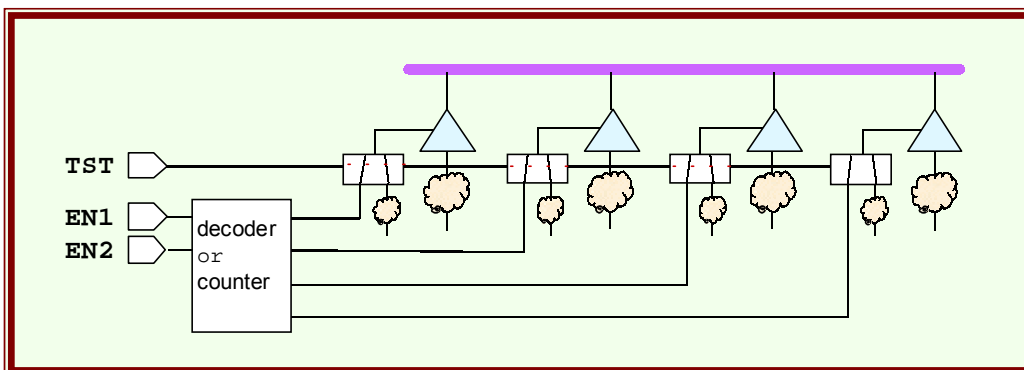


Figure 3-31 Control using decoders that are able to control the enable signals directly from the outside

Figure 3-31 shows the provision of decoder circuits instead of direct input in order to reduce the number of input pins. While decoders are generally made from combina-

tion circuits directly, there is also a method wherein a counter is included in the combination in order to turn the enable signals ON automatically. When decoder is created with combinational circuits, there is a risk of hazard being produced on the decoder output. When a hazard is produced, there is a risk for more than one tristate signal to be ON simultaneously. Additionally, when the delay time it takes for the signals to actually arrive at the tristate buffers from the decoder is not uniform, there may be two tristate signals ON at the same time because of the timing with which the switching occurs. When circuits such as these are created, the designer must be very careful during layout and during timing analysis. Actually, even if this type of decoder is used, problems in the algorithm for the ATPG tool may prevent improvement in the fault detection coverage. In the circuit shown in Figure 3-31, checkers related to DFT will be unable to make accurate determinations whether or not the circuit is correct. Ultimately, it will be necessary to confirm whether or not there are problems by performing a gate simulation using the test pattern produced by the ATPG tool.

The design of LSIs that use tristate elements (and in particular, bidirectional tristate elements) are those that are the least applicable when it comes to DFT. It is best to design so as not to use tristate elements whenever possible.

* External bidirectional ports should be fixed during the scan shift

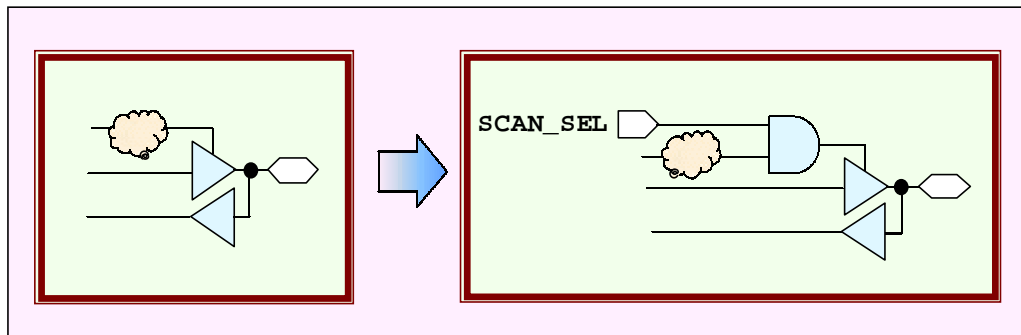


Figure 3-32 External bidirectional ports of the LSI are set as inputs during the scan shift

Be sure to set the external bidirectional ports of the LSI to be inputs during the scan shift. The direction to fix is better to be output as fault detection rate increases. However, if all the bidirectional ports are set to output, too much current flow that may generate problem. It is up to a designer's judgement which way to fix.

Some automatic fault test pattern generation tools support the control from outside during test. In this case, the rules of "3.3.3.[1] A clock must not be connected to the D input of a FF" should be applied as same as internal tri-state signals. IN reality, however, there is a limitation in the number of I/O pins of LSI that it is difficult to input all the enable signals from outside.

3.3.9. Handling Hard macros and Asynchronous Circuits, and Test Strategies for Large-scale ASICs

- | | |
|--|-------------|
| [1] Use boundary scanning on the I/O for hard macros and asynchronous circuits | recommend 1 |
| [2] Bypass the I/O instead of using boundary scanning on the I/O | reference |
| [3] Pay attention to fault detection in the RAM I/O when RAM is used | recommend 3 |
| [4] Consider creating test systems for large-scale ASICs | reference |
| [5] Boundary scanning is recommended for blocks larger than a specific size in large-scale ASICs | reference |

Explanation

* For hard macros and asynchronous circuits, use I/O boundary scanning

As described in “3.3.2.[1] Consider the use of independent test circuits for asynchronous circuits and hard macros”, independent test circuits are needed for hard macros and asynchronous circuits. Recently a unique test circuitry called “memory BIST” has been built into circuits such as RAM and ROM, and test patterns are generated automatically within the LSIs. Memory BIST does not only detect stuck-at faults, but also open faults and bridge faults as well. When broken down in more detail, bridge faults can be classified as inversion faults (wherein the data is inverted), state faults (where a constant value is output), and independent faults (where the value is the same as at another node). All of these faults can be detected using the BIST tools. Recently it has also become possible to perform tests in real time (at the RAM operating speed), and if the BIST tools are used correctly, it is possible to discover virtually all faults in RAMs and ROMs.

Through the use of BIST, more faults can be detected than designers inserting a selector circuit for RAM testing and writing data to, and then reading data from, all addresses in the RAM, from outside of the LSI.

This is not to say that fault detection using BIST is perfect, even if it can perform the detection process in RAMs, ROMs, and other hard macros. There still remains the issue of detection faults in the interface between the hard macros (wherein BIST was used to perform the detection process) and the other circuitry.

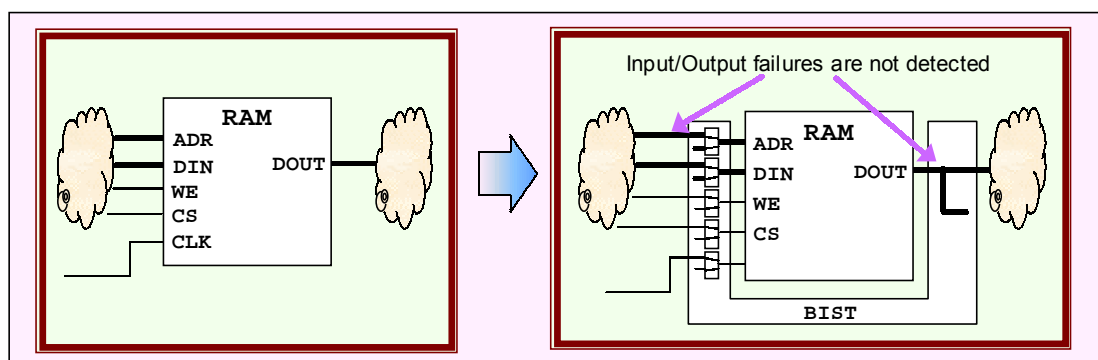


Figure 3-33 Insert a BIST circuit into the RAM test

Figure 3-33 is a circuit in which a BIST was inserted into the RAM. The BIST circuit makes a connection through a selector to the RAM input. When the BIST is operating, the selector switches, and the data is provided to RAM. The output signals are examined automatically by the BIST circuit, and a pass/fail decision is made. However, even if a BIST circuit is inserted, it cannot detect failures in the input and output nodes to which the RAM is connected, as shown in Figure 3-33.

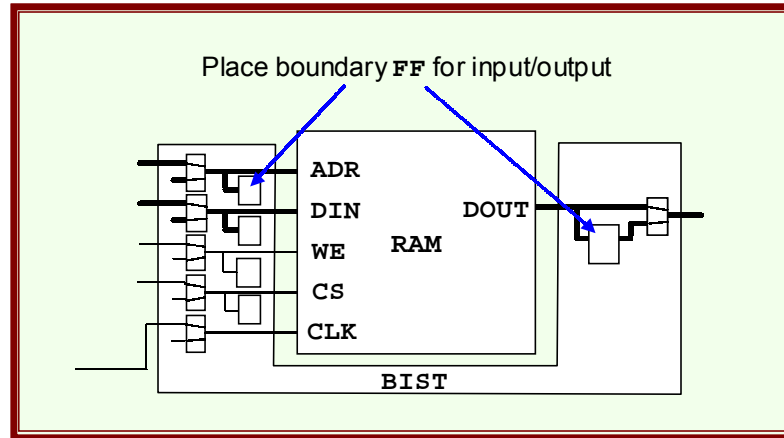


Figure 3-34 Insert FFs in a BIST circuit and scan

Figure 3-34 shows a case wherein FFs are placed in the BIST circuit in order to solve this problem with BIST. If the FFs that are placed at the inputs and outputs are chained by inserting scans, the ATPG tool will be able to perform fault detection on the RAM inputs/outputs automatically. This is an extremely good method for examining the RAM. Although it would be great if this can be done automatically by the BIST insertion tool, only a very few of the BIST insertion tools of today can insert the FFs or scans automatically. After BIST insertion, the designer must insert the FFs into the net list, and must go through the manual work of creating the scan chains. For large-scale LSIs, the work on the net list level can be time consuming.

* Bypass the I/O instead of using boundary scanning on the I/O

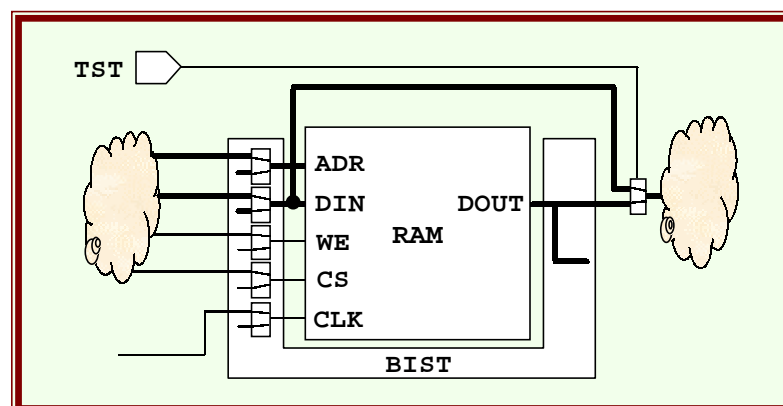


Figure 3-35 I/O bypass method

In addition to the method where FFs are inserted into circuits after the BIST insertion, there is a method where, in the RTL coding stage (or the gate circuit stage), the

3.3. Design for Test(DFT)

circuit is set up so that the RAM I/O is bypassed completely during testing. Because this process is performed in a stage prior to the BIST insertion (i.e, the RTL coding stage, where readability is particularly high), this method is easier to implement.

However, when this method is used, there will be paths that pass straight from the input to the output during testing, so there is the danger that these paths will show up as violations in timing analysis. In practice, this is not a problem because the operating frequency is low during testing; however, the fact that dummy errors will appear during timing analysis is problematic. One method for solving the timing problem is to insert a FF in the bypass line.

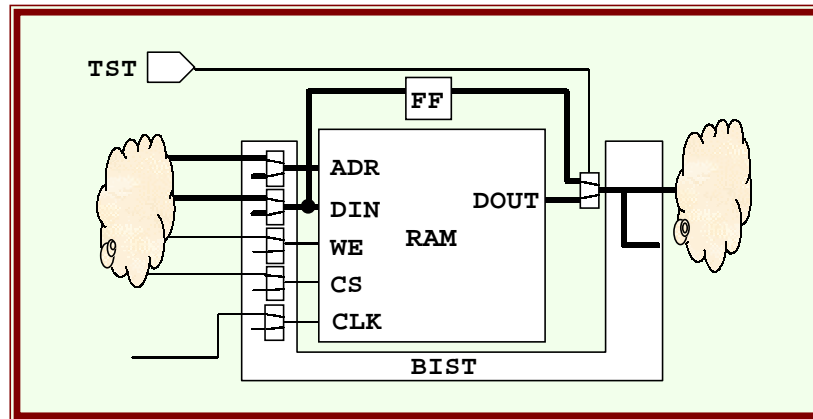


Figure 3-36 A FF inserted in the I/O bypass line

The use of these methods will eliminate the problem in timing analysis; however, while the DIN input can be examined using this method, RAM also has ADR input signals, along with WE and CS input signals. If the number of inputs does not match the number of outputs, there will still be nodes that cannot be examined using this method. As is shown in Figure 3-37, if there are more inputs than there are outputs, use EX-OR gates to combine multiple inputs into a single input. If one of the inputs of EX-OR gate is tied to either '1' or '0' then the signal input to the other side will be transmitted, which can improve efficiency when generating test patterns. Furthermore, when the number of outputs is more than the number of inputs, problems can be avoided by connecting selectors so that a single input can go to multiple outputs.

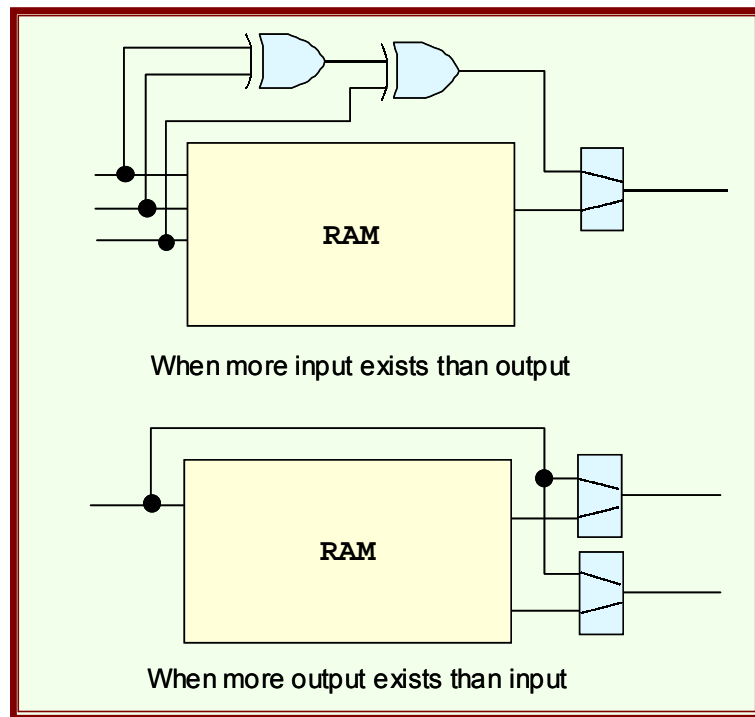


Figure 3-37 Bypassing when the number of inputs differs from the number of outputs in a hard macro

The approaches described above should not be limited to RAM, but should be considered for other hard macros as well. If the methods described above cannot be applied easily, then faults must be detected using test patterns generated by the designer. The designer should consider test patterns that cause all inputs and all outputs to transition between 0 and 1.

* Consider creating test systems for large-scale ASICs

For large-scale LSIs, one must take into account the problem that the scan insertion and the test pattern generation by the ATPG tool depend on the layout and the performance of the ATPG tool.

In circuits of more than a million gates, it is difficult to keep the test pattern within the allowable capacity for logic circuit testers when a single scan chain is used. Because the number and length of scan paths that can be used depend on the specifications of the logic circuit tester used, these parameters must be verified by the ASIC vendor. Additionally, when a scan is made to a place that is physically distant in the layout, the result will be an increase in the number of interconnects, which makes the layout more complicated.

With current methodologies, the scan chain lines are not determined automatically by the tools, but rather must be determined by a designer. In large-scale design, hierarchical levels are made in units between about 50,000 and 300,000 gates, and the scan chain at the top level is generally constructed manually. The designer should pay attention to the size of the blocks in the top level when it comes to creating scan chains.

3.3. Design for Test(DFT)

Conventionally, scan insertion has been performed prior to layout. In practice positioning FFs after layout causes the layout to become disorganized and confusing. When the positions of the FFs are disorganized and confusing, the scan path lengths get longer, putting pressure on the layout (the interconnects). The latest scan insertion tools now available on the market reroute after layout the scans that are created before layout. When it comes to the results of timing analysis after logic synthesis, the timing will change when scans are inserted because of the delays of the multiplexers and the increase in the fanout from the Q outputs. It is recommended that scan cells be used from the start when performing logic syntheses. (The “Design Compiler” synthesis tool uses a compile-scan process rather than just a compile process at the initial synthesis.)

- * Boundary scanning is recommended for blocks larger than a specific size in large-scale ASICs

In ultra large-scale LSIs (in excess of three million gates), the fault test strategy is of central importance. As described above, it is necessary not only to split the scan chains into several lines, but also to be innovative in partitioning the generation of test patterns using the ATPG tool. While the speed at which ATPG tools operate increases from year to year, it is still difficult to run these tools for circuits in excess of two million gates. This has given rise to the concept of partitioning not only the scan chains, but also partitioning all aspects from the generation of the test patterns by the ATPG tool.

As described above, if a hard macro is used, scan insertion is done in advance for the hard macro, and test patterns are produced in advance as well, separating the hard macro from the testing of the other circuitry. In ultra large-scale LSI design, it is said that this same approach of scan insertion and pattern generation by the ATPG tool that is performed on hard macros should be performed in units of about 500,000 gates. When the circuitry is partitioned into individual parts for examining the blocks using scans before making connections between the individual blocks, one method that can be used is to place FFs at the inputs and the outputs and then perform the scan (i.e., I/O boundary scanning can be used).

As a practical matter, this type of technique is not yet widely used. However, if the size of circuit designs continues to grow along present trajectories, I/O boundary scanning may become commonplace in fault detection tests.

In large-scale LSIs, it is currently difficult to reach a 100% fault detection coverage rate. Furthermore, as explained in the introduction, the ATPG and BIST tools for scan insertion examine only a single fault model, that of stack-at faults. Consequently, even if one could achieve a 100% fault detection coverage rate, the testing would still not be perfect. At present, other systems for detecting faults from other perspectives are under development, in addition to ATPG. The so-called Iddq method, where electric currents are examined in a static state, and the so-called Idds method, where electric currents are examined in a dynamic state, have also progressed to the point where they can be used.

3.4. Low Power-Consumption Design

3.4.1. Low Power-Consumption Design Using Gated Clocks

[1] In the design of standard ASICs, gated clocks can be used only in the top-level

mandatory

[2] When gated clocks are to be used in smaller units, use the EDA tools

reference

Explanation

One issue facing designs today is that of reducing power consumption. The most effective thing that designers can do to reduce power consumption using the circuit design itself is to use gated clocks or divided clocks. Through the use of gated clocks, it is possible to reduce power consumption by stopping clocks when they are not needed. The use of divided clocks is explained in “3.4.2.Low power design hints by the logic circuits”.

Using gated clocks or divided clocks, on the other hand, increases the likelihood of problems in DFT design, logic synthesis, timing analysis, and layout. Be sure to study the use of gated clocks carefully.

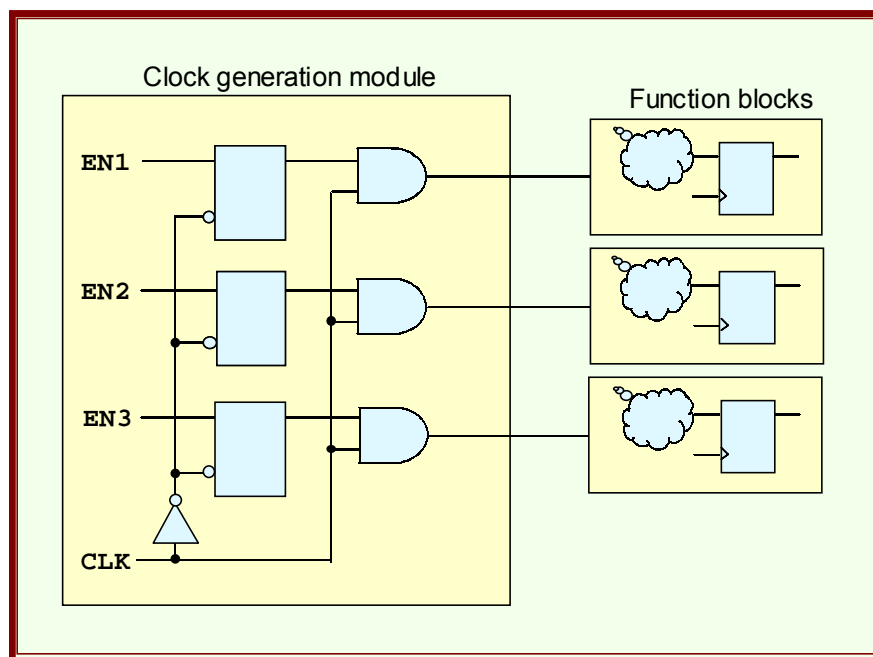


Figure 3-38 Gated clocks within a clock generator module

When gated clocks are used in the design of standard ASICs, the gated clocks should only be placed in the clock generator module at the top-level, as shown in the example in Figure 3-38, so as to gate the clocks to each functional block.^[1] When gated clocks exist in the local level, the result will be the presence of many incorrect cells in the clock tree when generating the clock tree using clock tree synthesis, and correcting clock skew is more difficult. Additionally, using gated clocks in only the clock generator module makes it relatively easy to insert circuits for DFT.

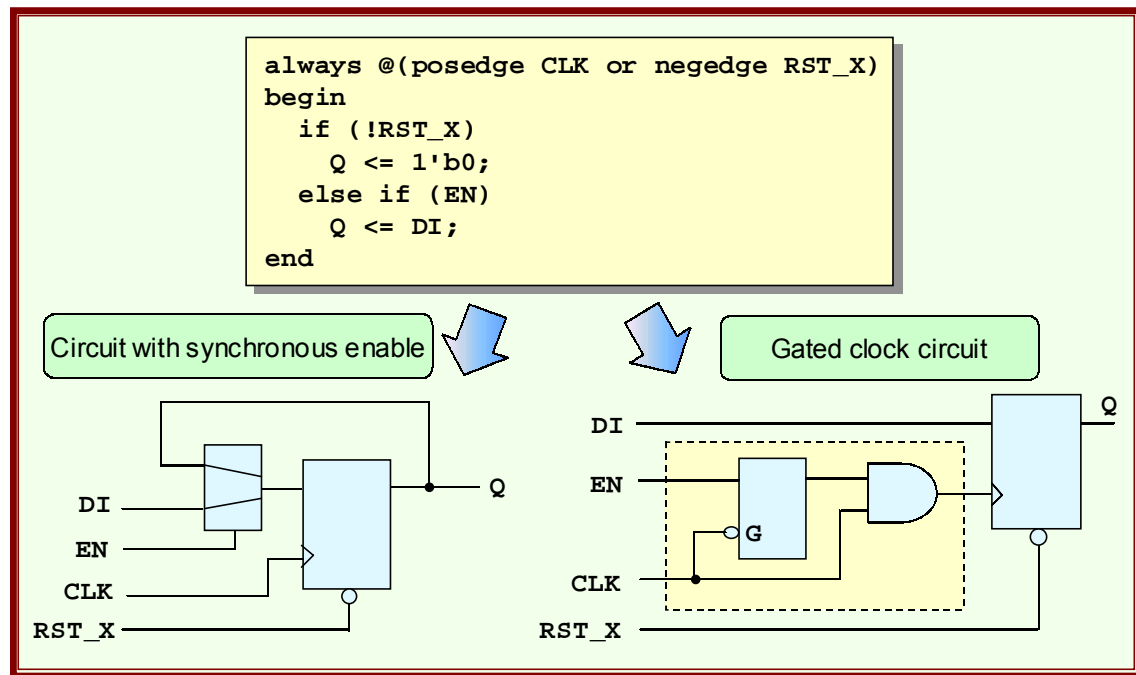


Figure 3-39 Creation of a gated clock circuit

When gated clocks are used in detailed units in the local level, the EDA tool can be used to generate them.^[2] The amount of power saved through the use of the EDA tool averages about 30%, but, depending on the circuit, it may be possible to cut power consumption by 80% or more.

In RTL description, when synthesizing the code for an FF with synchronized enable, as shown in Figure 3-39, the circuit shown in the left-hand side of Figure 3-39 is generated. A selector is connected in front of the FF data input, and when the enable signal is '1' the value of the input data is read, and when it is '0' the output of the FF is fed back so the output of the FF is read in. At this time, the clock is constantly input into the FF regardless of the value of the enable signal, and thus power is consumed in the circuitry within the FF depending on the clock frequency.

The EDA tools can be used to generate the gated clock circuits, such as shown on the right-hand side of Figure 3-39, from the code for FFs with synchronous enable. Were a designer to do this manually, the designer would have to place a description of the gated clock in the RTL description, requiring a tremendous amount of work and time.

In Figure 3-39, a gated clock was created using a latch with inverted clock and an AND gate for an FF with a synchronous enable. Because the clock is always supplied to the latch, there is no gated-clock effect. Additionally, because the selector is replaced with a latch with inverted clock and an AND gate, the area will be increased. Because of this, the gated clock will not be effective unless a gated clock is connected to a large number of FFs.

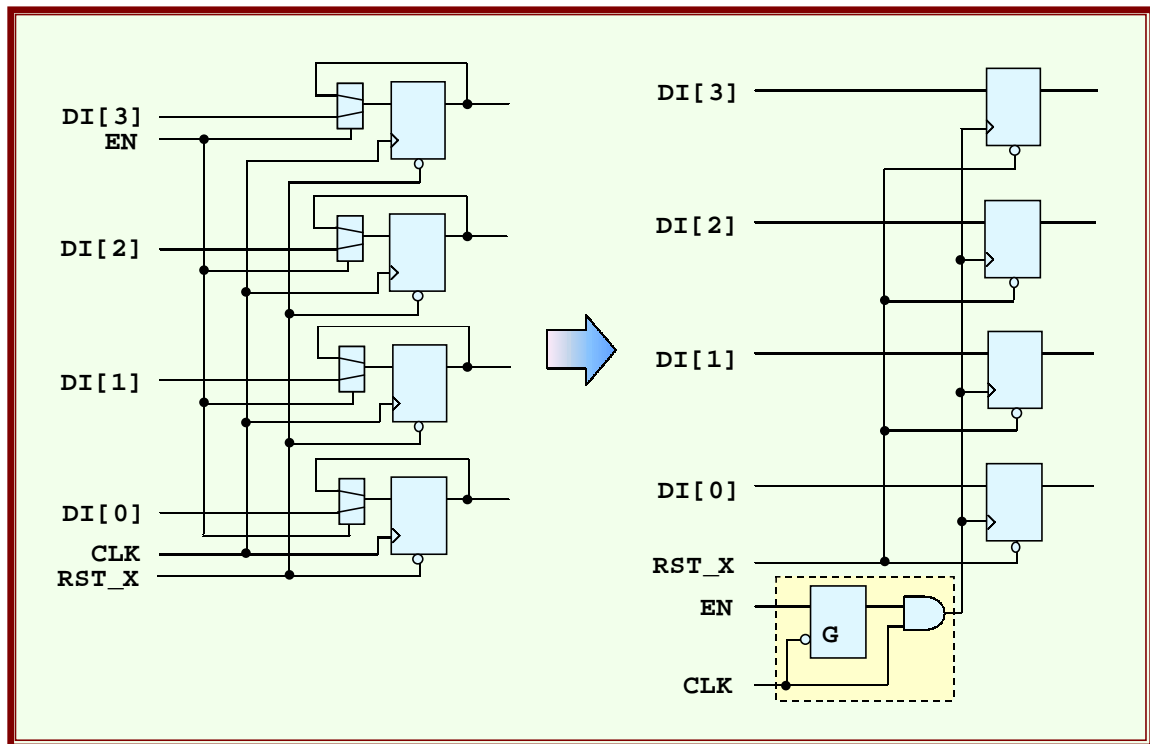


Figure 3-40 Implementation of the gated clock circuit

In Figure 3-40, four FFs are connected to a common enable signal. In a case such as this, the clocks that are provided to the clock pins of the four FFs can be supplied only to a single latch with inverted clock alone, thereby reducing power consumption. Furthermore, the four selectors are replaced by a latch with inverted clock and an AND gate, reducing the area used.

Aside from using the EDA tools, the use of gated clocks on detailed units at local levels should be avoided. Because it is very easy for the designer to make an error when inserting gated clocks on detailed units, such a practice is extremely dangerous. The EDA tool has a command for controlling the number of clocks connected to a single clock. Normally this value should be set to between 4 and 7.

In the end, the usage of EDA tools is just a simple method. It will be more effective from the standpoint of reducing power consumption when a designer does this on the top-level module with consideration of clock supply. When gated clocks are made on the detailed units in the local levels, the amount of energy consumed in the clock lines will not be reduced, and the effect may be less than expected. The problem of the clock tree synthesis will be explained in “3.4.3.Low power design hints by the clock tree”.

3.4.2. Low power-consumption design hints by the logic circuits

- [1] Power consumption can be reduced through dividing the clock to reduce the operating speed
- [2] Dividing clock and parallel execution should be applied for data path part
- [3] Use a small number of enable signals to frequently stop the operation of the circuits

reference

reference

reference

Explanation

After gated clocks, the next most effective method of low-power design is to divide the clock in half or in quarters, thereby dropping the operating frequency of unneeded parts. In MOS transistor logic circuits, cutting the clock frequency in half reduces power consumption by about one half as well.

Dividing the clock frequencies, however, at too detailed a unit will cause problems for clock tree generation and for DFT. To prevent there from being too many clock lines, it is recommended that there be only one system of divided clocks.

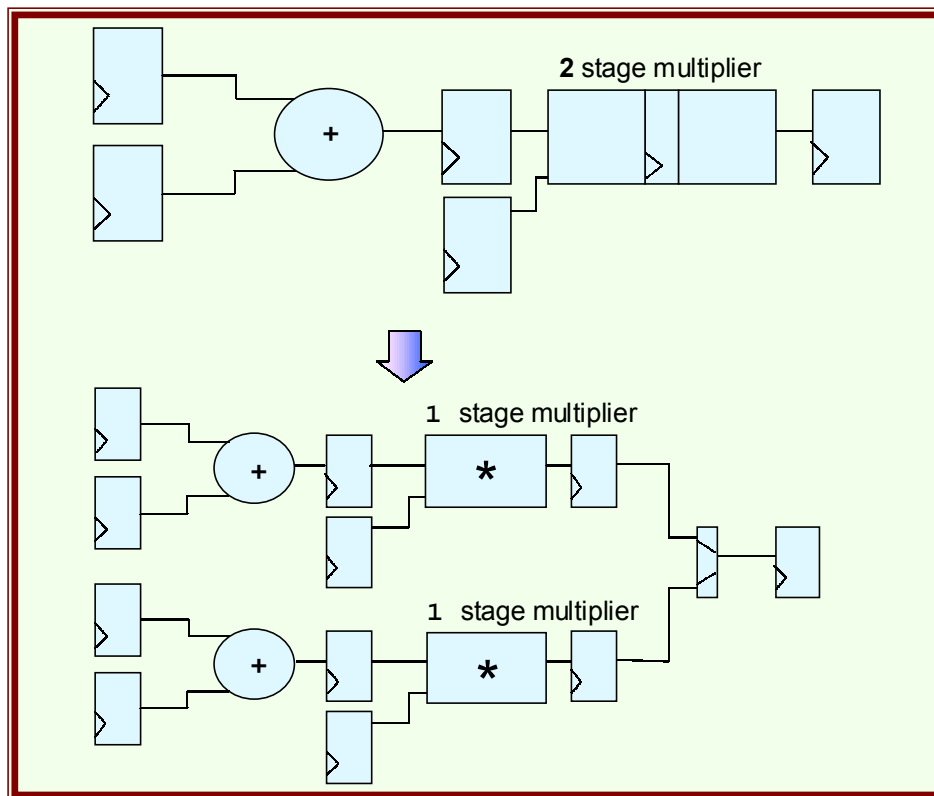


Figure 3-41 Frequency division used on the data path

Even in parts that are always in operation, such as the data path, it is still possible to reduce power consumption by clock division.^[2] As is shown in Figure 3-41, the number of data lines is doubled when the clock is divided. While of course this type of modification increases the size of the circuit, the increase in the amount of area consumed can be minimized by tuning the circuit design or by changing adders, etc., from a high-speed carry look-ahead type to a ripple carry type. However, dividing the clock of the data path

part, which operates at the normal operating frequency, would require a considerable amount of work from the circuit designers.

Other than using gated clocks or divided clocks, one way to save power is to use the enable signal to stop the operation of the circuits.^[3] For example, one could use an enable signal to control a circuit rather than dividing the clock for the circuit.

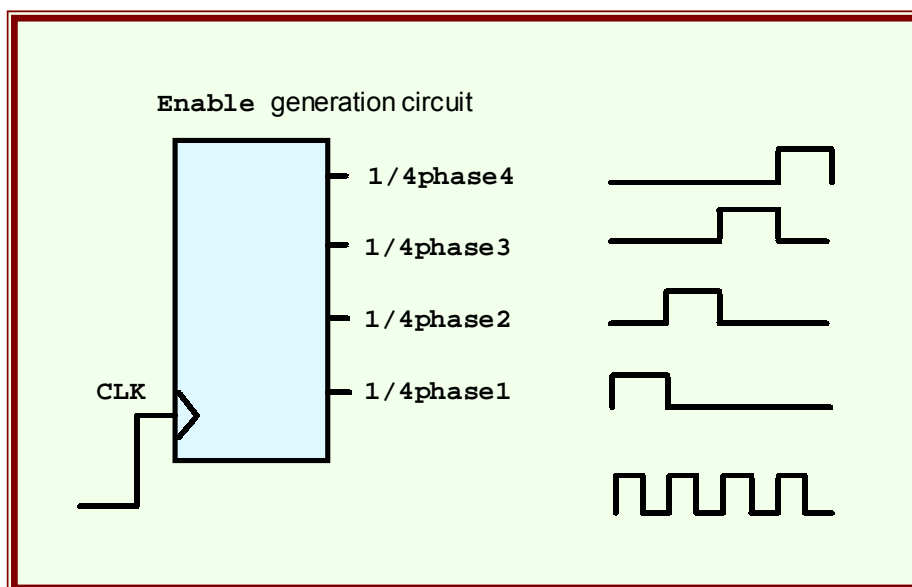


Figure 3-42 Control using an enable signal

The proportion of the circuit that is active can be reduced, reducing the power consumption, by having the circuit operate only one phase out of four. Placing a synchronous enable directly before the FF makes it possible to reduce the power consumption even further through the use of the EDA tools to produce gated clocks. However, this method will not produce the desired level of power savings unless the circuit is one wherein the activity of the enable signal is quite low.

As was explained in “3.4.1.Low Power-Consumption Design Using Gated Clocks” control using the enable signal in this way is more effective when performed on the combinational circuit at the previous stage than it is when input into an FF with a synchronous enable. However, doing so is a complicated, time-consuming process. When compared to the amount of work involved, the amount of power saved is not much.

In addition to attempting to save power in the design of the logic circuits themselves, it is wise also to consider the following approaches:

- (1) Reducing the voltage. (Moving to 1.8 volts instead of 2.5 volts will reduce the power consumption by about half.)
- (2) Using a technology with a smaller design geometry. (Moving from 0.25mm to 0.18mm will reduce the power consumption by approximately two thirds.)
- (3) Using specialty processes for low power consumption.

3.4.3. Low power-consumption design hints by the clock tree

[1] Power consumption can be reduced through partitioning the clock lines

reference

Explanation

The clock line structure requires special attention when gated clocks are used. Recent clock tree synthesis (CTS) tools are able to generate clock trees even when multiple gates are present in the clock lines. However, when it comes to adjusting clock skew between gated clocks, a large number of clock buffers may be inserted for the skew adjustment in the stage before the gated element. When clock buffers are inserted in the stage before the gated element, the clocks will always be supplied to the buffers that have been inserted, regardless of the enable signal for the gated clock, canceling out the power-saving effect of the gated clocks.

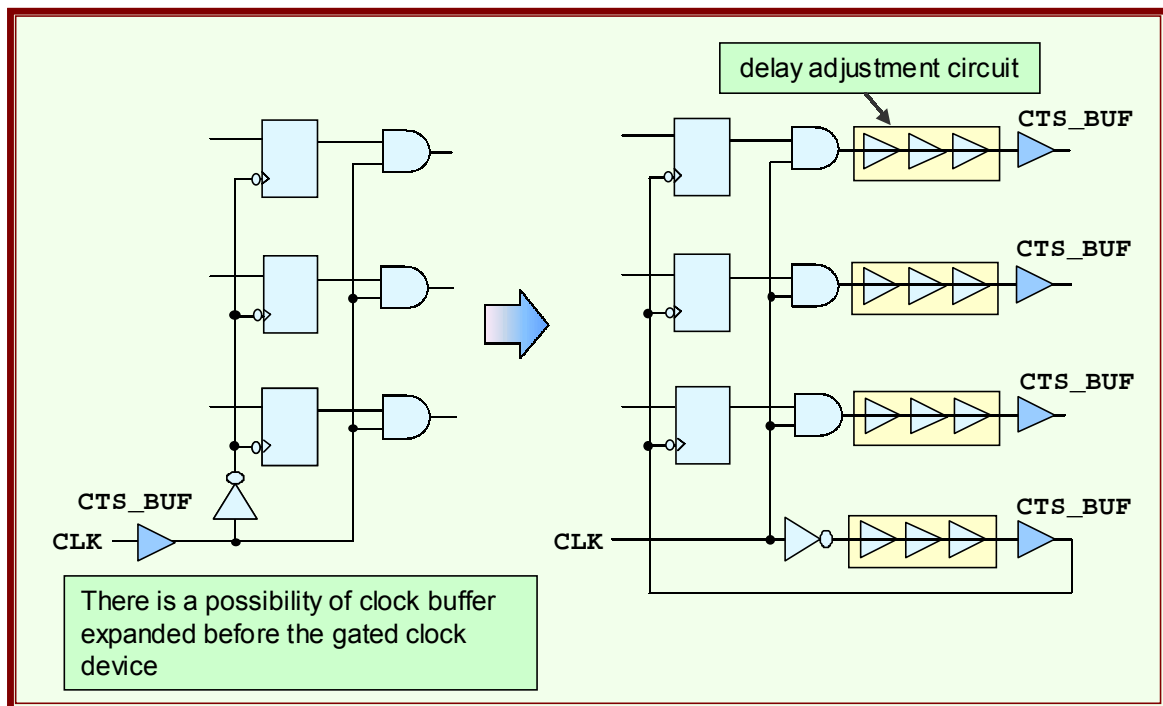


Figure 3-43 A clock tree structure when gated clocks are used

One way to avoid this problem is to partition the clocks into different gated clocks as shown in Figure 3-43 and then perform clock synthesis on the individual clock lines. If the clock lines are separated, they will not be affected by the other clock lines. Additionally, when one considers the floor plan in the layout, if the blocks that are supplied by the gated clock are combined in one place, then it will be possible to save power by generating compact clock lines for those blocks. Even if gated blocks are not implemented, it is possible to save power by merely being aware of the placement of the blocks in the layout and by partitioning the clock trees.^[1]

Note, however, that the clock skew between different gated clocks must be adjusted manually when this method is used, making it necessary to insert in advance delay adjustment circuits in the stages following the gated elements. The delay adjustment circuits should be implemented hierarchically, and buffers can be inserted manually based on the layout results.

3.5. Source codes and design data management

3.5.1. Create a directory for each objective

- | | | |
|-----|--|-------------|
| [1] | Save RTL descriptions, logic synthesis scripts, logic synthesis results, and simulation results in different directories | mandatory |
| [2] | Do not save test vectors in RTL description directories | recommend 1 |
| [3] | For data version management, copy the directories and delete unnecessary files to save in a compact size | recommend 2 |
| [4] | File names should be specified as relative paths | recommend 1 |

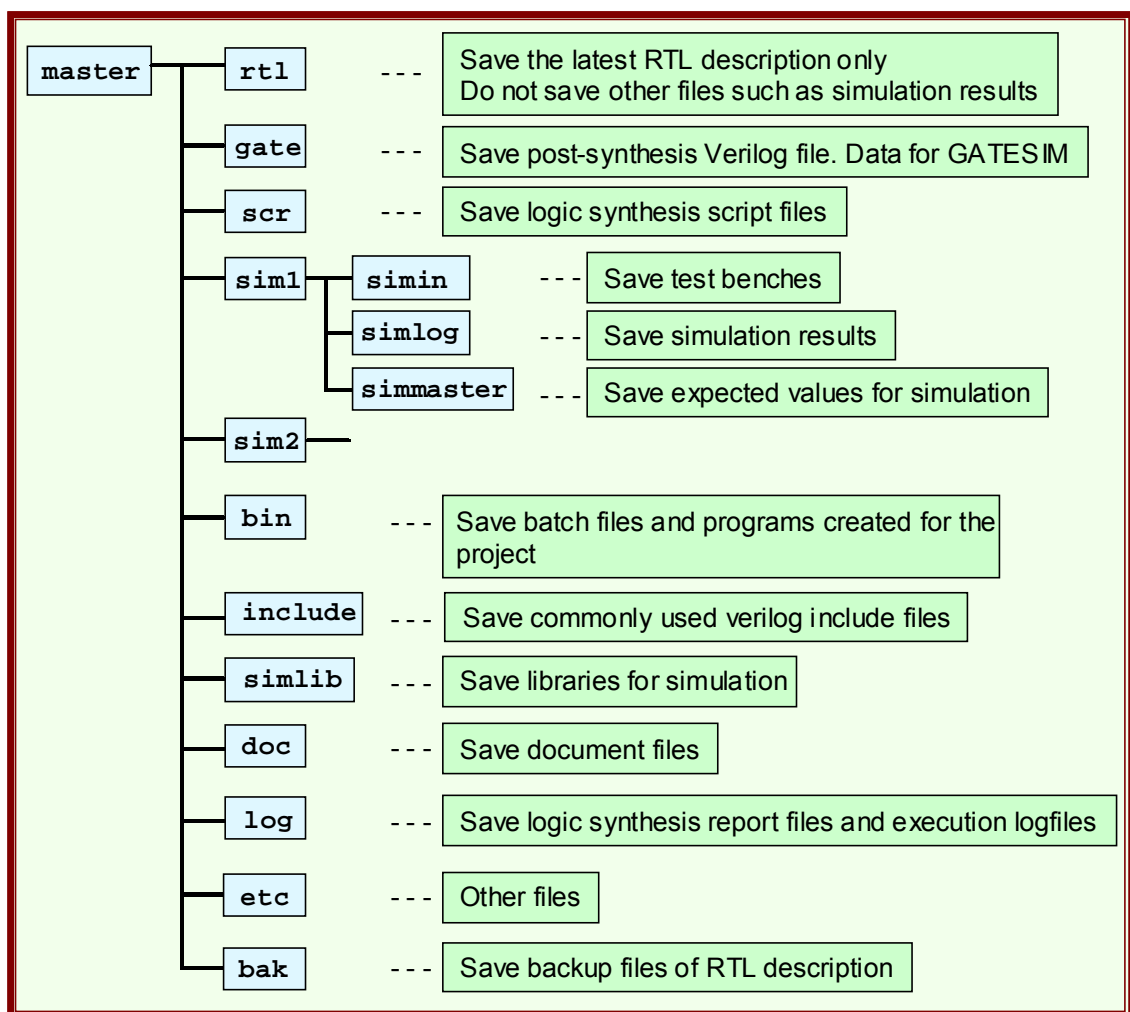


Figure 3-44 Example directory structure

Explanation

Design data require many files in addition to RTL and test bench files. If these files are all saved in the same directory, it will become impossible to distinguish between necessary and unnecessary files. Therefore, design data should be divided and maintained in a separate directory for each data type.^[1]

3.5. Source codes and design data management

Figure 3-44 provides an example of a *directory structure* for design data. This chapter will provide a description based on this example of a *directory name*. However, as long as each *directory name* and *structure* are easy to understand designers can decide any *name* and *structure*. There is no problem if all the file reference is described by relative path, and the top directory and directories under it can function no matter where they are copied.^[4]

First, save only *rtl* files in the *rtl* directory and be sure not to mix these with other data such as those for test patterns in the case of Figure 3-44. RTL descriptions are important data that directly lead to the final product. Always be sure to separate RTL descriptions from other files.^[2]

Save test patterns in the *sim* directory, the test input in *simin* and the test results or logs in *simlog*. Run simulations in the *sim* directory, and be sure that the waveform output and files that are automatically created by the simulator are saved in this directory.

Either directly output data such as waveform data to this directory or create a new directory named “wave” and output the data to it. Separate these data from other data so they can be easily deleted. In some cases, the amount of waveform data (such as VCD files, etc.) reaches several Giga Bytes. Make it possible to retrieve such data easily and to erase it; otherwise, the capacity of hard disk drive will be filled up soon.

Save all the simulation test benches in *simin*. The *include* files, which have the description of parameters and tasks used on the test bench, should be placed in the *include* directory if they are for common use in the development project. Have the simulation results output to *simlog*.

The simulation results will be compared with previous results, and the RTL simulation results will be compared with the gate level simulation results. For this reason, copy all verified simulation results to be saved to *simmaster*.

As shown in the Figure 3-44, the test directory is best named according to actual test specifications rather than as *sim1* and *sim2*. From the standpoint of test operation, creating many detailed test patterns to link test patterns with test specifications is preferable for reuse. If many verification items are tested with one test pattern, it will be difficult to analyze simulation results. Give due thought to an effective directory structure to manage several test benches.

The logic synthesis script creates *scr* and detailed sub-directory structures for each kind thereof. Large designs may have more than 1000 files. A directory structure must be in place to manage these data.

Save documents in the *doc* directory. It does not matter which tools create the documents, but if the documents are saved as *.pdf* or *.html* files, it will be easier for all designers to access the documents.

The directory structure in the Figure 3-44 is just an example. The actual directory structure should be decided for each design item. It may be easier for a designer to manage the directory structure that matches the design hierarchy structure than the directory struc-

ture, which is flat like the one shown in Figure 3-44.

Figure 3-45 is an example of the directory structure created from the top level. In this case, note that it becomes difficult to understand the data when the hierarchy structure is too deep. Moreover, give due consideration to an environment where simulation and logic synthesis for each level can be performed automatically from the upper level.

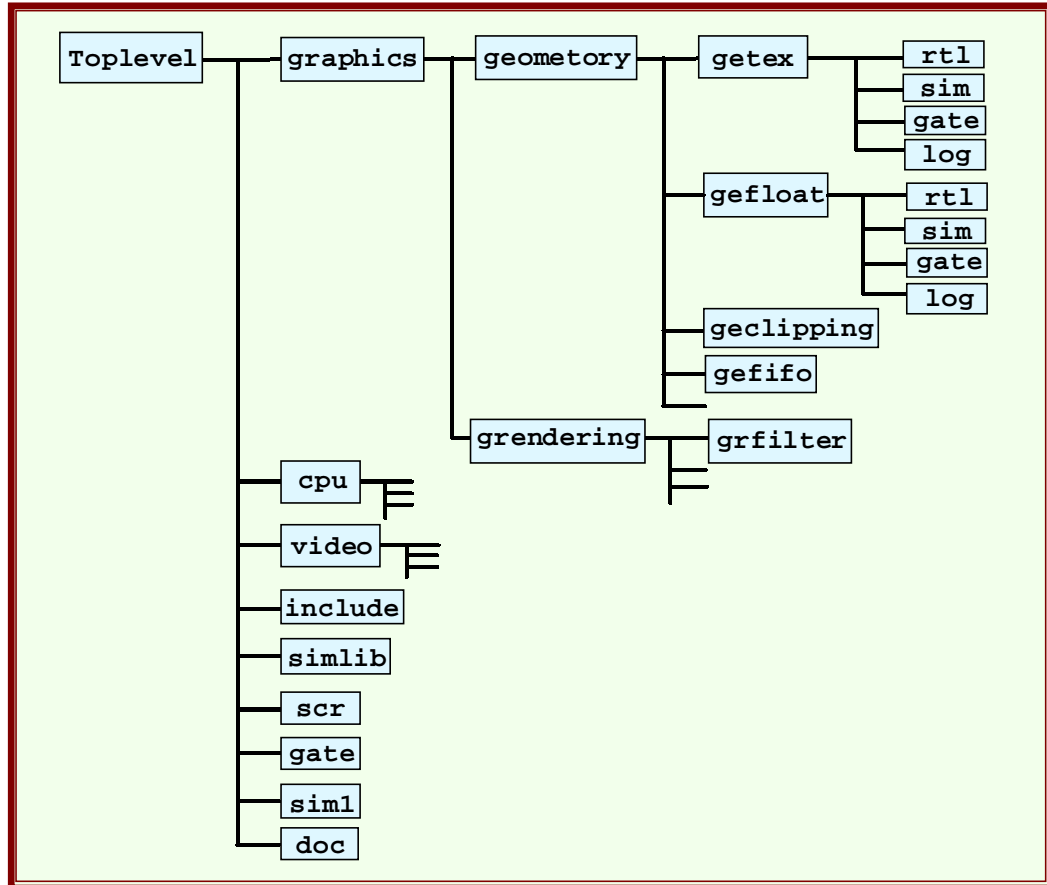


Figure 3-45 Directory example matching design hierarchy structure

Manage the data versions by copying to each design directory, changing them to reflect directory names that include the date or version name, and then saving them. In the case of design data, other data such as the logic synthesis scripts and the test patterns are involved in addition to the *rtl* source codes. Therefore, the optimal method is to store and save these data together. ^[3]

However, if the design data were stored as is, there exists no hard disk that could store them all. For designs on the order of 1 million gates, it is not uncommon for the design data to require more than 10 GB of disk space. Therefore, after copying the data, either delete unnecessary directories or establish links for data that are the same in other versions. It is also necessary to create directories properly in order to organize all of these data.

3.5.2. File suffix names

[1] Each file's reference should be specified by a relative path (return to the master directory once)	recommend 1
[2] The file name of RTL description should consist of <i>module name</i> + ".v"	recommend 1
[3] The file name of the test bench should be "_tb.v", "_test.v" or ".vt"	recommend 3
[4] The file name of the gate description should consist of <i>module name</i> + ".v" or ".vnet"	recommend 3
[5] The include file name should be ".h", ".vh" or ".inc" for RTL description, ".inc", ".ht" or ".tsk" for test bench (Verilog only)	recommend 2
[6] Execution file names on UNIX should end with ".run"	recommend 3
[7] Simulation (Verilog) script file names should end with ".v_scr"	recommend 3
[8] lint script file names should end with ".l_scr")	recommend 3
[9] The file name of logic synthesis scripts should end with ".scr"	recommend 3
[10] All logic file name for synthesis, simulation, and layout logs should end with ".log"	recommend 3
[11] The file name of lint logs should end with ".l_log"	recommend 3
[12] Logic synthesis report file names should end with ".rep", ".tim", or ".ara"	recommend 3
[13] SDF file names must end with ".sdf"	recommend 1
[14] EDIF file names must end with ".edif" or ".edf"	recommend 1
[15] The file name of logic synthesis databases must end with ".db"	recommend 1

Explanation

Circuit designs handle many files. Apply an extension to each file name so each name will be distinguishable. Have file name references return to the master directory in the relative path before specifying file names.^[1] If file names are specified by an absolute path, then it will have to be changed when the data are moved to another directory.

If, for example, the RTL description is placed in a directory called rtl, then the *include* file referred to in the RTL description must end with "../rtl/common.h". In addition, specify a file's reference in each script by a relative path. The simulation executes in directory sim1, and logic synthesis is executed starting from directories other than those under scr. It becomes necessary to specify a return to the master directory to avoid confusion.

3.5.3. Define necessary information for file header

- | | |
|---|-------------|
| [1] Indicate the circuit name, circuit function, author, and creation date in the file header | recommend 1 |
| [2] Indicate who made changes and which item was modified in the case of reuse | recommend 1 |
| [3] Make file headers common | recommend 1 |
| [4] Consider using CVS if necessary | recommend 3 |

Example Code

```

/*-----
-- FILE NAME : tmg.v                      --
-- TYPE : CIRCUIT                        --
-- FUNCTION : Timing generation block      --
-- edit : Bob                            --
-- Author : Robert                       --
-- Rev,Date : 1.0  97/08/01              --
--           : 1.1  97/08/01              --
-----*/

module TMG( CP_DATB, CP_ADRB, CP_RDX, CP_WRLX, RST, CP_MTMG,
            TM_VOEN, TM_CDATEN, SL_SYNCST, TM_DOUT, SCLK);

```

Example 3-9 File header definition

Explanation

Define the information necessary for the file header.^[1] Define the file, and make it possible to know when the file was modified by whom and for what purpose it was created. Doing this makes it possible to improve readability when the description is reused by designers other than the original designer.

In the Example 3-9 file header, file names, circuit type, function, modifier, originator, and revision are defined.^[2] The section, test bench name, comments, etc. should be added when necessary.

Readability decreases if the file headers' formats differ among designers, so make them common as much as possible.^[3] Adding parameters that can be handled by CVS to the file header makes it possible to fill in the change history automatically in the source code.^[4]

RT qualify performs following checks.

3531(N) File header missing, or data is incomplete.

3.5.4. Managing the version of a file

[1] In multiple designer development, separate the master data from the individual data

recommend 1

[2] CVS can be used to manage file versions

reference

Explanation

Before a design is completed, many files needed to be created in addition to the RTL and the test bench. In addition to RTL descriptions, multiple forms of data such as logic synthesis script files and logic synthesis result gate files must be managed. If these files are not managed properly, the newest files cannot be referenced and design errors will occur when executing simulations or logic synthesis.

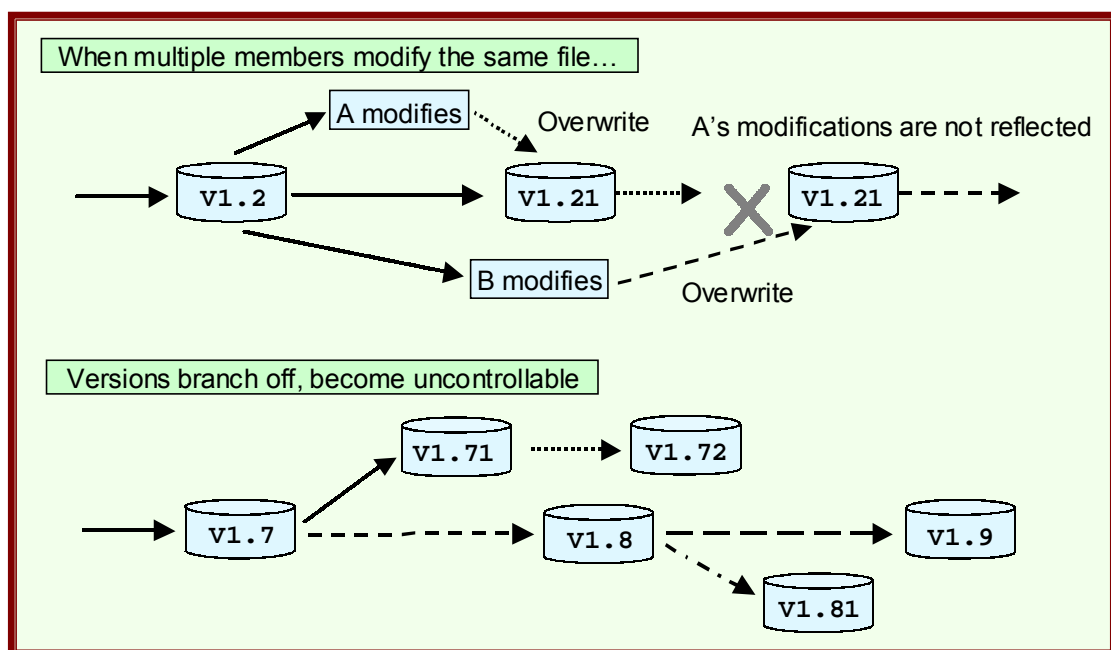


Figure 3-46 Necessity of version management

Multiple members may simultaneously modify files in the case of development. Even if the handling of same files by multiple designers is prohibited, there are cases in which such access is inadvertently provided. To prevent this problem, always separate the master file from the individual data when there are more than two designers.^[1]

An individual designer's data should as a rule have the same directory structure as the master. The designer (the individual) only copies from the master the necessary directories as necessary. Moreover, after the design work is concluded, the verified data are uploaded to the master.

When individuals upload data at this time, they should use some scripts such as `csh`, etc. so the other necessary data in the master are not inadvertently deleted. If you are uploading data unnecessarily, data with bugs may be uploaded, so be careful with the data you upload.

When a manager (design team leader) uploads data, he should be careful not to upload

data that have not been completely modified yet. When managing data for multiple designers, teamwork between the manager and the members is important. Take steps to ensure that all designers clearly communicate with one another and that there are no mistakes.

When data (RTL, *simlib*) are updated, there are cases in which simulations that had worked fine up to that point (they were executed by a different designer) cease to function. Therefore, when there are many team members, there are cases in which it is necessary to prepare *master_tmp* (as a temporary directory) in addition to the *master*, and to have a two-stage structure that uploads data to the *master* after all verification is completed in that directory.

Design data can be managed with file administration systems such as SCCS, RCS (Revision Control System), or CVS (Concurrent Version System).^[2]

CVS is the most suitable for managing design data. CVS is the management tool of choice for two reasons: first, it can manage data in each directory; second, it is possible to overcome a problem with the tool since it always stores the most recent version as the master. Even when one uses this tool, it is not possible to judge automatically the interval for copying data and for merging. Communication between the administrator (team leader) and each designer is still important.

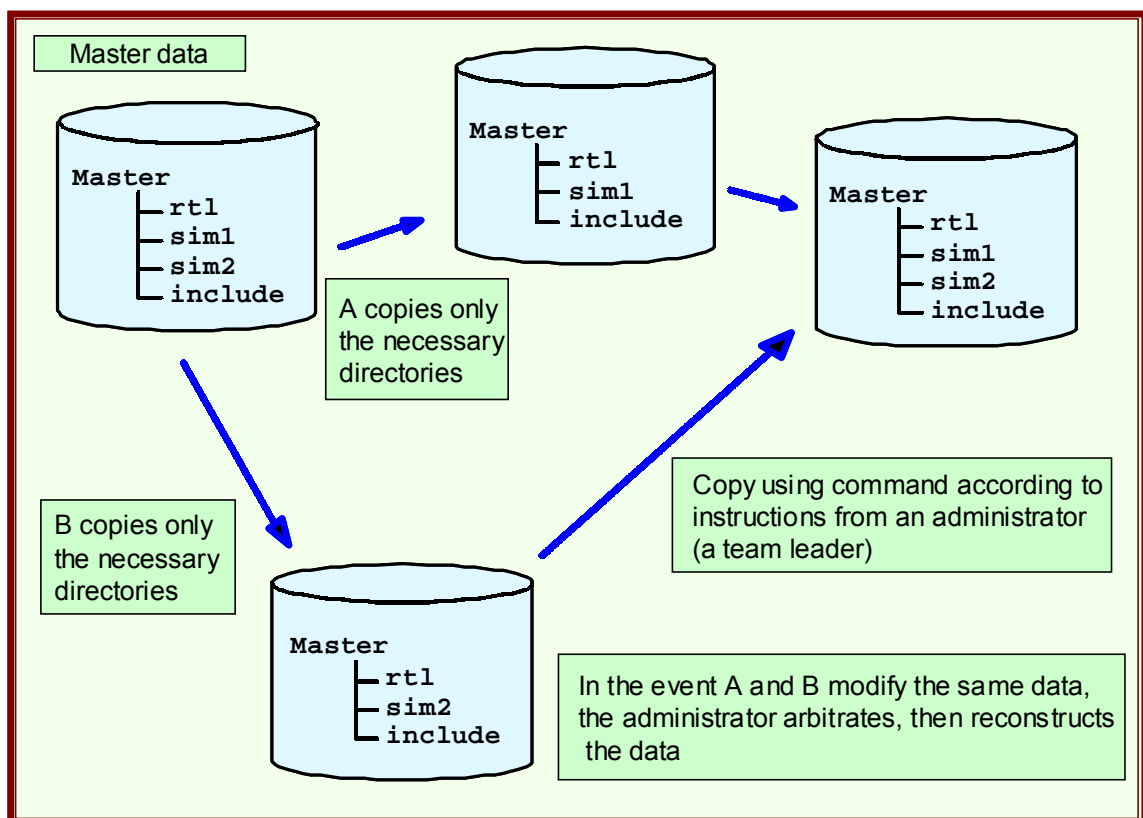


Figure 3-47 File management method

3.5.5. Periodically back up files

[1] Back up files

recommend 2

[2] Periodically back up all design directories

recommend 2

Example Code

```
#!/bin/csh -f
@ num= 1
set foo = "First"
set Bakfilename = ""
while ( "" != $foo )
    set Bakfilename = $foo
    set foo = `echo $argv[1] | cut -f $num -d/`
    @ num++
    if ( $foo == $Bakfilename ) then
        break
    endif
end
if( -f ~/vibak/$Bakfilename.bak2 ) then
    mv ~/vibak/$Bakfilename.bak2 ~/vibak/$Bakfilename.bak3
endif
if( -f ~/vibak/$Bakfilename.bak ) then
    mv ~/vibak/$Bakfilename.bak ~/vibak/$Bakfilename.bak2
endif
if( -f $argv[1] ) then
    cp -p $argv[1] ~/vibak/$Bakfilename.bak
endif
/usr/ucb/vi $argv[1]
exit 1
```

Example 3-10 Example script for file backup

Explanation

Periodically make file backups if necessary.^[1] The Example 3-10 is an example of a script that automatically backs up specified files and then makes it possible to edit the file.

If a backup file is created in the same directory, the number of files increases. Rather than showing design data management, this example instead shows a simple script for a case where a designer (individual) has made only slight changes without employing a data management method or during the period until the overall data management method starts at the initial design stage. This is a good method for retrieving previous data that were accidentally deleted.

Backup files can also be created manually, but it is possible to easily recover modifications that were inappropriate by automatically creating backups.

Moreover, many files are created during the design process, so we recommend that you regularly back up the entire design directory on a tape drive or hard disk on a daily basis.^[2] However, it has become more difficult to back up all of the data since the amount of design data has recently increased. In such a situation, try to ensure file system security by using RAID, for example. In addition, it is best to back up at least the RTL description, Sim description and logic synthesis script everyday.

3.5.6. Use comments often

[1] Use comments often to improve the readability of the source code	recommend 2
[2] Add comments that indicate the objective and content to operators in the description	recommend 2
[3] Describe the I/O ports and declarations in one line and always add comments	recommend 2
[4] Provide the comments in English as much as possible	recommend 2
[5] Some EDA tools may not read comments in languages other than English	reference
[6] EUC has fewer problems than any other Japanese codes	recommend 1
[7] Comments should start with "//"	recommend 3

Explanation

The frequent use of comments improves the readability of the source code.^[1] As a result, it becomes easier to maintain the source code and descriptions become easier for other designers to understand, and this in turn leads to improved reuse efficiency.

Comments should be added indicating the objective and contents to operators in the description, sub-programs, *always construct*, and *condition statements*.^[2] Moreover, the I/O port or internal register declarations should be described in one line for each signal or register, and comments should always be added.^[3]

It is better to provide comments in English, if possible.^[4] Design resources may be used internationally. Global expansion becomes easier if the comments are written in English. Also, there are many EDA tools that do not ensure proper operation by source codes including languages other than English.

However, if it is hard to provide comments in English, sometimes the comments become inadequate and sometimes the number of comments may diminish. It is acceptable to write the comments in a language other than English if writing the comments in English is too difficult. Always be sure that detailed comments are added to the greatest extent possible.

At the present time many EDA tools do not ensure proper operation if comments are provided in languages other than English.^[5] If possible, it is ideal to create tools that automatically delete comments in languages other than English.

Among the Japanese codes 7bit-JIS, shift-JIS and EUC, ECU has the fewest problems.^[6] If you use Japanese codes other than shift-JIS, problems will not occur so frequently. It is advisable for you to delete the comment lines when problems occur.

A description with added comments is shown on the following page. At a minimum, insert comments in all I/O signal and register *signal names*. It is best to insert comments next to the applicable lines in *always constructs* and *assign statements*.^[7] In this case, there is no point in describing something as a RTL description, so describe it by changing the

3.5. Source codes and design data management

expression as much as possible .

The volume of comments is generally said to be about 20 to 40% of the source code.

RTqualify performs following checks.

3564(N)	There is a comment in Japanese. (excluded from check as default setting)
3566(N)	Japanese EUC character code is not used in a comment. (excluded from check as default setting)

Comment description example

```
module TMG( CP_DATB, CP_ADRB, CP_RDX, CP_WRLX, RST, CP_MTMG, TM_VOEN, TM_CDATEN,
           SL_SYNCSTLT, TM_DOUT, SCLK);

`include "commpara.h"           //common parameter file
`include "localpara.h"         //local parameter define file

/*****input-output pin list*****/

/**bus signal, bus control signal
input[`RTMLEN-1:0] CP_DATB;           //internal databus           from cpu_if block
input[`ADRLN-1:0] CP_ADRB;           //internal addressbus       from cpu_if_block
input CP_RDX;                       //read flag.'0'active        from cpu_if_block
input CP_WRLX;                       //write flag.'0'active       from cpu_if_block
input CP_MTMG;                       //macroselect.'1'active      from cpu_if_block
//addressbus in all 16 bit. use lower 8 bit in this block
//higher 8 bit is make a decode by cpu_if_block
//higher 8 bit of CP_MTMG are address MG function
//enable signal in case of ~

/**signal from part of de-modulator
input SL_SYNCSTLT;                   //initial for slotcounter    from slot block
input RST;                           //global reset.'1'active     from outside of IC
input SCLK;                           //system clock               from PLL 1 block

output TM_VOEN;                       //'1'by sound data section of receive slot
output TM_CDATEN;                     //'1'by control data section of receive slot
output[`RTMLEN-1:0] TM_DOUT;          //output for databus

/*****internal register list*****/
reg[`RTMLEN-1:0] TM_DOUT;              //register of databus output
reg[`SLTLEN-1:0] SLTC;                 //slot counter
reg[`RTMLEN-1:0] RTMGCR;               //TMGCR register output signal. exclude SLTINI
reg[`STLEN-1:0] SINIFSM;               //variable state machine for SLTINI signal.the present
//state

/*****internal signal list*****/
reg[`STLEN-1:0] NSINIFSM;              //variable state machine for SLTINI.the next state
reg[`RTMLEN-1:0] ADOUT;                //transform value integer into vector of slot counter
reg RSTINIFF;                         //reset signal of SLTINI signal maintenance register
reg SELARRY;                          //clock latch signal for write register by negative edge
reg NSTFSM;                           //clock latch signal for write register by negative edge
reg SLTINI;                           //slot counter reset TMGCR(3)
reg C;                                 //
wire SYNCEN;                          //slot counter reset pulse enable TMGCR(2)
//pulse from negative edge to negative edge of SCLK from
//part of de-modulator

wire CDATENC;                          //CDATEN output enable, enable by '1' TMGCR(1)
wire VOENC;                            //VOEN output enable, enable by '1' TMGCR(0)

CLTM #(`RTMLEN) RTMGCR1 (CP_DATB,SCLK,RST,RTMGCR); //CP_DBUS signal write in RTMGCR reg
//utilization reg for correspondence change spec bus
CLTM #(`RTMLEN) TM_DO1 (ADOUT,SCLK,RST, TM_DOUT); //output of bus after latched ADOUT signal
//utilization reg for correspondence change spec bus
assign SYNCEN = RTMGCR(2);             //resolution of write in register into each bit signal

assign CDATENC = RTMGCR(1);
assign VOENC = RTMGCR(0);
```

Example 3-11 Added comments (1/3)

Comment description example(continued)

```

//-----
//  PROCESS NAME: ADRDEC      --
//  FUCTION      : DATA Select  --
//-----
always @( CP_RDx or CP_WRLX or CP_MTMG or RTMGCR or SLTC or CP_ADRB ) begin
    if( CP_MTMG == OFFS ) begin          //data select mode
        if( CP_RDx == OFFS ) begin        //read register
            SELARRY <= OFFS;              //not generation latch pulse for register
            case (CP_ADRB)                 //based on address lower 1 bite.
                ATMGCR : begin
                    ADOUT[RTMLEN-1:0] <= SLTINI & RTMGCR;
                    C <= ONSN;             //2000
                end
                ASLTCR0 : begin
                    ADOUT[7:0] <= SLTC[7:0];
                    C <= ONSN;             //2002
                end
                ASLTCR1 : begin
                    ADOUT[1:0] <= SLTC[SLTLEN-1:SLTLEN-2];
                    C <= ONSN;             //2003
                end
                default : begin
                    ADOUT <= 8'bXXXXXXXX;
                    C<=OFFSN; end
            endcase
        end
        else if( CP_WRLX ==OFFS ) begin    //write in register
            ADOUT <= 8'bXXXXXXXX; C<=OFFSN; //bus for Hi-Z
            case (CP_ADRB)
                ATMGCR : SELARRY <= ONS;    //latch pulse by TMGCR
                default: SELARRY <= OFFS;    //not generation latch pulse for register
            endcase
        end
        else begin                        //In case of no turn OFF CP_RDx and CP_WRLX
            ADOUT <= 8'bXXXXXXXX;           //ADOUT will do. Reduce the area to a minimum
            C <= OFFSN;                     //Hi-Z is 3 state buffer
            SELARRY <= OFFS;                //not generation latch pulse for register
        end
    end
end

//-----
//  PROCESS : SINIFSMC
//  FUNCTION : SLTINI latch by state machine
//-----
always @( posedge SCLK or posedge RST) begin //state machine clear SLTINI after reset
    if(RST == 1'b1)
        SINIFSM <= FIRST;                 //reset condition, global reset
    else
        SINIFSM <= NSINIFSM;              //initial of FSM variable
    end
    else
        SINIFSM <= NSINIFSM;              //FSM state moves at SCLK positive edge
    end
    else
        SINIFSM <= NSINIFSM;              //moves next state
    end

//-----
//  PROCESS : SINIFSM
//  FUNCTION : SLTINI decoder by state machine
//-----
always @(SLTINI or SINIFSM) begin
    case (SINIFSM)
        FIRST : begin                    //initial state
            RSTSINIFF <= OFFS;            //not creation pulse turns 0 into SLTINI reg
            if (SLTINI == ONS)            //slot counter indication of initial
                NSINIFSM <= SECOND;        //slot counter preparation of reset
            else
                NSINIFSM <= FIRST;          //state(FIRST)maintenance
            end
        SECOND : begin                    //creation state at slot counter reset pulse
            RSTSINIFF <= ONS;              //creation at pulse of clear SLTINI register
            NSTFSM <= FIRST;               //return to initial state
            end
        default : begin
            RSTSINIFF <= 1'bx;             //creation at pulse of clear SLTINI register
            NSTFSM <= 1'bx;               //return to initial state
            end
    endcase
end
end

```

Example 3-11 Added comments (2/3)

Comment description example(continued)

```

//-----
//  PROCESS  : SINIRFF
//  FUNCTION  : SLTINI(TIMGCR(3))FF
//-----
always @(negedge SELARRY or posedge RST) begin
    if( RST==1'b1)                //global reset?
        SLTINI <= OFFS;           //reset FF
    else if(RSTSINIFF==ONS)        //slot counter reset are complete?
        SLTINI <= OFFS;           //reset FF
    else                           //CP_WRLX positive edge=SELARRY negedge
        SLTINI <= DATR(3);        //latch at bit 3 of data bus
end

//-----
//  PROCESS  : SLTCT
//  FUNCTION  : slot counter(0-839)
//-----
always @( posedge SCLK or posedge RST) begin //SCLK is count up from sync of posedge
    if (RST==1'b1)                //make an integer at counter of 0~ENDSLTC.
        SLTC <= 0;                //global reset?
    else if( SL_SYNCSLT==ONS || SLTINI==ONS || SLTC == ENDSLTC ) //initialize SLTC (move 0)
        SLTC <= 0;                //from de-module?TMGCR.SLTINI?SLTC over flow
    else                           //0 turn SLTC
        SLTC <= SLTC + 1;         //slot counter increment
end

//-----
//  PROCESS  : VOENG
//  FUNCTION  : create sound data enable signal
//-----
always @(SLTC or VOENC) begin
    if( VOENC==OFFS )              //VOEN output not permission, 0 fix
        TM_VOEN <= OFFS;
    else if((SLTC>=STVOEN1 && SLTC<=ENDVOEN1) //the part of first half of VOEN
        || (SLTC>=STVOEN2 && SLTC<=ENDVOEN2)) //the part of latter half of VOEN
        TM_VOEN <= ONS;
    else                           //VOEN fault output section
        TM_VOEN <= OFFS;
end

//-----
//  PROCESS  : CDATENG
//  FUNCTION  : create control data section enable signal
//-----
always @(SLTC or CDATENC) begin
    if( CDATENC==OFFS )            //CDATEN output fault permission, 0 fix
        TM_CDATEN <= OFFS;
    else if(SLTC>=STCDATEN && SLTC<=ENDCDATEN) //CDATEN output section
        TM_CDATEN <= ONS;
    else                           //CDATEN fault output section
        TM_CDATEN <= OFFS;
end

endmodule

```

Example 3-11 Added comments (3/3)

3.5.7. Using CVS when managing the versions

[1] RTL codes may be managed by CVS

reference

[2] What is easily managed by CVS is the source code such as RTL, test bench and script file. CVS is not well-suited to manage comprehensive sets of all the data

reference

Explanation

We recommend using CVS (Concurrent Version System) for version management of the design data. It can be used mainly for source code management of circuit description.^[1] It is not suitable for management of log file, database file after logic synthesis and such.^[2] CVS was originally a software development tool, but is now a system for managing versions when developing software. CVS, one of several GNU tools that run on UNIX, is a front-end tool for RCS (Revision Control System), which is a UNIX version management tool. Therefore, RCS is required to run CVS.

Neither CVS nor RCS are bundled with Solaris, so they must be installed before use. If it is Version 1.10 or later, however, RCS and CVS are bundled in the same package. Since CVS is a GNU software application, it can easily be installed for free.

CVS source code or binary can be obtained from the following sites and such.

<http://sunsite.sut.ac.jp/sun/solbin/>

<http://www.cvshome.org/>

Execute the following at the unzipped directory for installation.

./configure

/usr/ccs/bin/make install

If C language can be used, set-up is done by the above 2 commands. cvs command is automatically created under /usr/local/bin.

One advantage of using CVS is its superior version management feature. Management of an entire directory is possible with CVS. Multiple versions are managed by differences (revisions), so little disk space is required. For this reason, it is possible to retrieve previous versions, recheck the items that have been changed, and return to an arbitrary version. In addition, version branching is supported and it is possible for multiple designers to edit the same file.

In CVS, all modifications are recorded in a log file, so it is possible to check the modified items, modification date, designer who made the modifications, and comments by referring to the log file.

3.5.8. Using CVS basic commands *checkout* and *commit*

[1] With CVS, sharing and management information is stored in repositories

reference

[2] The CVS has the basic commands; *checkout* and *commit*

reference

Explanation

With CVS, all information pertaining to shared items or management information is stored in the repository.^[1] The repository can be placed anywhere as long as it can read/write. It can be placed anywhere if shared disks and individual regions can be accessed, but the users are prevented from directly manipulating the information in the repository.

The repository specifies the directory to be used by UNIX environment variable \$CVSROOT. If \$CVSROOT is changed, then it will be possible to properly use multiple repositories. Source files are saved as RCS management files, and data such as the modification items, log, modification dates, and name of designers who made modifications are saved.

```
setenv CVSROOT /home/cvsdata    (specify the directory for storage)
cvs init                        (create initial data of repository)
```

If directory structure and file already exist, move to the directory and then execute

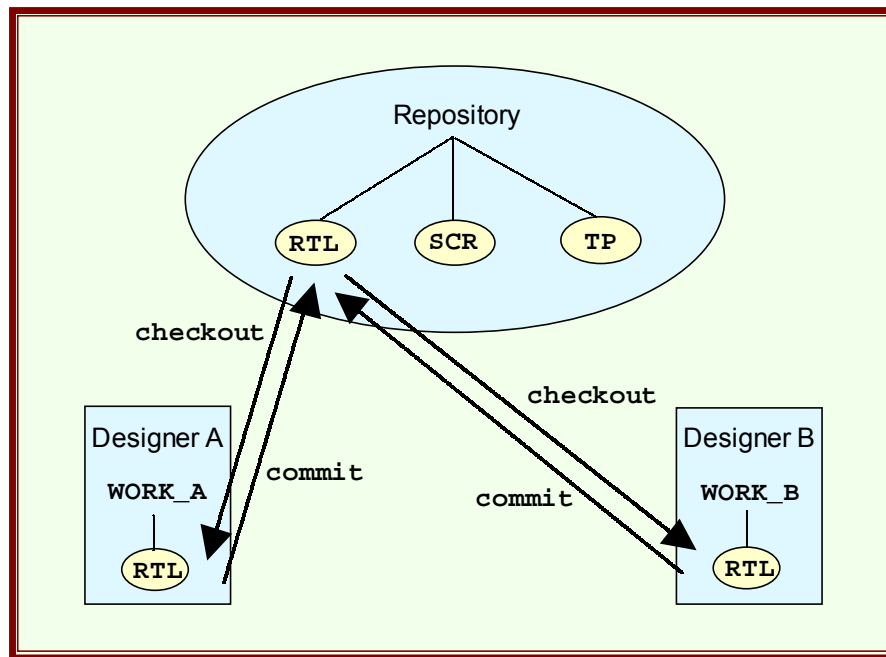
```
cd /home/hasegawa/ProjectX
cvs import -m "ProjectX" xxxxx hdLab start
```

xxxxx directory is created under by this command \$CVSROOT(/home/cvsdata) where data is saved under as directory structure.

Normally, the user name of source provider should be specified for the place, which is "hdLab" in the above description. The identifier of this release should be specified for "start". These names are used just as comment or such that it does not really matter as long as it is an appropriate name.

CVS should be used from the initial stage of design. If data is not available at that point, import empty directory. When file is created, register each one by cvs add command (3.5.9.).

With CVS, data are copied from the repositories to each region. Dedicated commands are used to copy data between the repositories and the work regions. This copying task is performed using “checkout” and “commit” which are the basic CVS commands.^[2] “checkout” copies data from a repository to a work region. “commit” copies data from a work region to a repository.

Figure 3-48 Running *checkout* and *commit*

3.5. Source codes and design data management

3.5.9. Managing versions using CVS (as example)

- [1] Managing file versions using CVS commands
 1) Copying from the repository 2) Debugging and simulation
 3) Storing in the repository 4) Releasing the work region
- [2] Beware of data contention messages

reference

reference

Explanation

Figure 3-49 shows the basic CVS commands. In the case of CVS, an execution command should be added after `cvs`. There are three basic commands: “checkout”, which retrieves files from repositories, “commit”, which returns files to the repositories, and “update”, which updates the files. There is also the “log” command, which displays the modification history, and “status”, which displays the version information.

```
cvs checkout module name
cvs commit [-m log message] [file name]
cvs update
cvs log [file name]
cvs status [file name]
cvs diff [option] [file name]
cvs add file name
cvs remove file name
```

- • fetch file
- • Write file
- • Update file
- • Modification history
- • Version information
- • Display difference
- • Add file
- • Delete file

[] indicates items that can be omitted, “*module name*” is a directory name

Figure 3-49 Basic CVS commands

The following figure illustrates a sequence of basic CVS editing work. First, a file is retrieved from a repository using the “checkout” command. Then, either debug or simulation is executed and the file is modified. Afterwards, the file is stored in a repository and the work region is released.^[1]

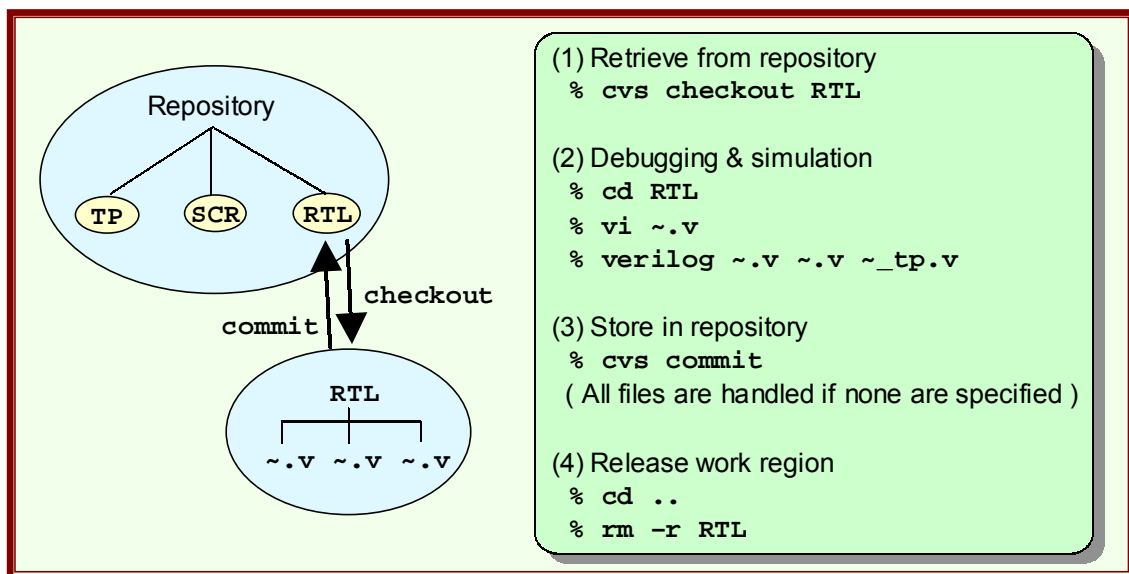


Figure 3-50 Standard editing task

Figure 3-51 illustrates a situation in which multiple users copy identical data to a work region, execute the “commit” command, and then data contention occurs when the data are returned to the repository.^[2] After Designer A executed “commit” to return data to the repository, Designer B executed “commit” for the same data, so a conflict occurred. CVS therefore displays a warning message, and Designer A’s corrections and Designer B’s corrections are merged when the update command is specified.

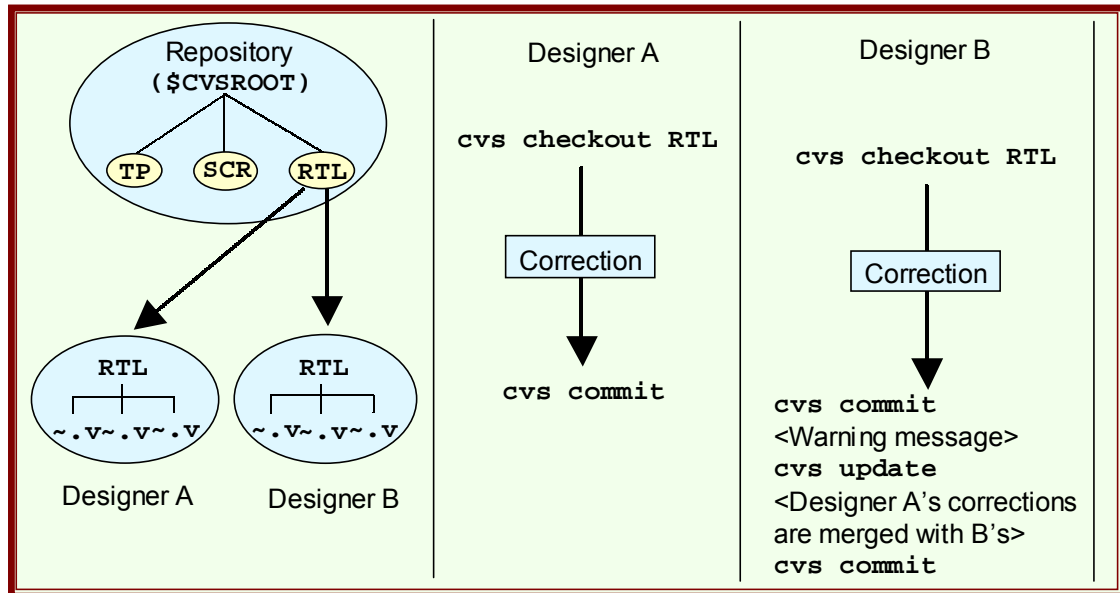


Figure 3-51 Contention when implementing CVS

The following figure illustrates an example of the implementation of CVS where version branches are created. Branches can be created for module (directory), but branches cannot be created to one single file.

1) Create branch ^[1]

cvsv rtag -b -r *version_number branch_name module_name*

Example) **cvsv rtag -b -r 1.2 patch1_2 RTL**

2) Call branch ^[2]

cvsv checkout -r *branch_name module_name*

Example) **cvsv checkout -r patch1_2 RTL**

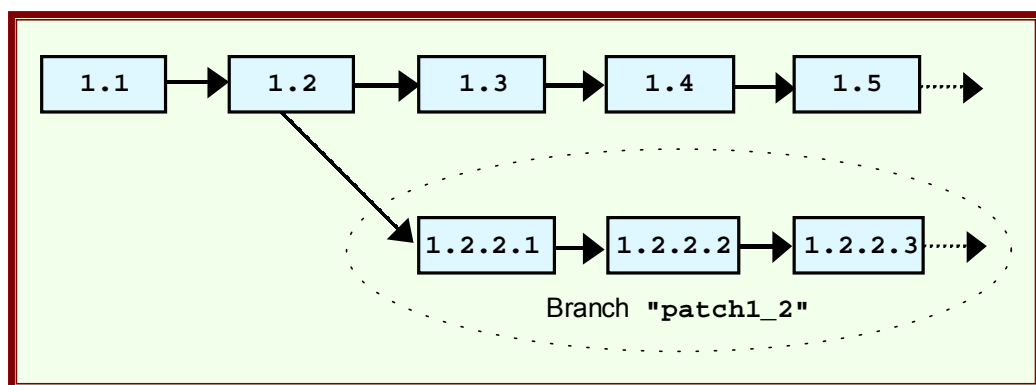


Figure 3-52 Branch creation

3.5.10. Check modified contents using the CVS history

- | | |
|--|-----------|
| [1] Display the file version information in the work region using <i>status</i> | reference |
| [2] Display such detailed information as each version, branch, modification date | reference |

Explanation

It is possible to display and check such information as the version information and the modification history by using the “status” or “log” commands.

* Status command example

cvs **status** [*file name*]

In the case of the “status” command, the version information for those files that have been checked out to the work area is displayed.

```

-----:
File: clock24.vhd           status: Up-to-date

Working revision:          1.7 Fri May 22 11:26:01 2001
Repository revision:      1.7 /koba/RTL/clock24.vhd,v
Sticky Tag:                (none)
Sticky Date:               (none)
Sticky Options:           (none)

```

Example 3-12 *status* command execution

* log command example

cvs **log** [*file name*]

Executing the “log” command displays the modification history of those files that have been checked out to a work region. The “log” command displays the details of each version and the branches such as the modification date, name of the designer who made the modifications, and comments made at the time the modifications were being made. Comments that can be added when running “cvs commit” are important in managing information so be sure that these are meaningful.

```

RCS file: /koba/RTL/bcd60count.vhd,v
Working file: RTL/bcd60count.vhd
...
-----
Revision 1.4
Date: 1998/05/22 11:51:14; author: koba; state: Exp; lines: +2 -0
Corrected so XPOSCNT counter value is 9 and conditions are branched (kobayashi)
-----

```

Example 3-13 *log* command execution

Chapter 4 Verification Techniques

This chapter explains techniques for test bench description using Verilog-HDL, methods for running simulations, guidelines for proceeding with verification work, mismatches in gate level simulation with RTL simulation, and simulation-related techniques.

Contents

- 4.1 Test bench description
- 4.2 Task description
- 4.3 Verification process
- 4.4 Gate level simulation
- 4.5 Static timing analysis



hd Lab, inc.

4.1. Test bench description

4.1.1. Hierarchizing the test bench

[1] Hierarchize the test bench using hierarchical structure or task/function

recommend 1

[2] Define common timing using parameter statements or text macros

recommend 2

Explanation

If the circuits to be verified are complex, the test bench description required for verification will also be complex and the amount to be described will tend to increase more than the circuit description. Therefore, hierarchization using the test bench division or using the task/function is required.^[1]

Verilog test benches range from simple descriptions that indicate only the input time and signal values to descriptions that read test vector files and structural descriptions that use functions or tasks. Select the appropriate test bench description taking into account the size and complexity of the circuit to be verified and the verification objectives.

Description style	Description method	Merits	Target circuits
Signal value	Description signal names and signal values that execute elapsed time and assignments like <code>#100 A=4'b1000;</code>	Easy description is possible	Cell libraries and small circuits
Vector file	Create test vectors using separate file, runs verification by reading them	Corresponds to tester file	Circuits subject to the tester
Hierarchization Description	Constant test bench in a program-like manner by calling tasks	Is possible to run complex tests	Large system circuits

Table 4-1 The respective test bench description styles and features

With cell library verification or small circuits, signal values can be described in the simplest manner.

When consideration is given to correspondence with the tester, the conversion of test vectors into testers is facilitated by making the test vectors into separate files and reading them. However, since designed circuits continue to get ever larger, debugging efficiency is improved by preparing general tasks that generate test vectors and then making functional descriptions that call them.^[2]

Verilog-HDL is a simple language from the standpoint of the programming language. Structuring complex data and formula manipulation operation are however not the strong points of Verilog-HDL. Recently, there have been many cases where the test bench is described by C language and the test results are also analyzed by C language. In such cases, the test vector file is read or output in Verilog-HDL. However, the throughput is too great for I/O of a simple test vector file. The data should be processed up to a point by using tasks, etc. and then file input/output should be performed.

4.1.2. Use basic test vector descriptions

[1] Use parameters - Do not use embedded values in the test descriptions	recommend 1
[2] Use blocking assignment "=" - When "<=" is used, the operation is postponed - Simulation speed is faster	reference
[3] The number of lines in <i>fork-join</i> should be a maximum of 5 (Verilog only)	recommend 2
[4] Specify time units using `timescale (Verilog only)	recommend 2

Example Code

```

`timescale 1 ps / 1 ps
module BCD_TEST;
parameter CYCLE = 1000 ;
parameter HALF_CYCLE = 500 ;
parameter STB = 100 ;
reg CLK,COUNTON,RST_X;
wire CO;
wire[3:0] LSB,MSB;

BCD100 U1(.COUNTON(COUNTON), .RST_X(RST_X),
          .CLK(CLK), .LSB(LSB), .MSB(MSB), .CO(CO));
always begin
    CLK = 1'b1;
    #(HALF_CYCLE) CLK = 1'b0;
    #(HALF_CYCLE);
end
initial begin
    RST_X = 1'b1; COUNTON = 1'b0;
    #(CYCLE) ;
    #(STB) RST_X = 1'b0;
    #(CYCLE) RST_X = 1'b1;
    #(CYCLE) COUNTON = 1'b1;
    #(300 * CYCLE) $finish;
end
endmodule

```

Example 4-1 Sample test bench described by Verilog

Example 4-1 shows a basic test bench described by Verilog. First, the simulation time unit is defined by `timescale.^[4] Then, the circuit name is defined after the module, but since this test bench description is the top layer, the module has no I/O ports. Next, the parameters necessary for running the simulation are defined by the parameter statement, and *reg* or *wire* declarations are made on the signals used by the test bench description. Define the values by parameters as much as possible to facilitate making changes to the values used by the test bench description.^[1] Simulations are often run with different cycles. In order to ensure the correct operation, all values relating to time should be defined with parameters.

4.1. Test bench description

Next, the circuit components subject to verification are instantiated. With component instantiations, use port name connections instead of port order connections and clearly specify the port names of the lower components. Please refer to "3.2.3. Use the port name connection for component instantiations". The description style up to the component instantiation is common to nearly all test benches.

Then, the descriptions that generate the test vectors and the descriptions to observe the output signals are made. Descriptions that are repeatedly executed like clocks are defined by an *always construct*. Descriptions that are sequentially executed are defined by *initial constructs*. Clock signals described by an *always construct* can be explained as the Example 4-2. However, we recommend describing clock generation by an *always construct* as the Example 4-1 so that the descriptions can be clearly distinguished from other descriptions.

```
initial begin
    CLK = 0;
    forever
        #HALF_CYCLE CLK = ~CLK;
end
```

Example 4-2 Clock generation description (non-recommended example)

fork-join is a syntax that supports simultaneous execution (see "4.2.2. Describe function operations using tasks"). When fork-join is used effectively, improved flexibility in description is possible and this is convenient. However, fork-join also slows down simulation speed. Moreover, when there are too many assign statements in fork-join the readability declines. Limit the fork-join contents to a maximum of 5 execution lines .

4.1.3. Note input signal timing

[1] Shift the timing of assignments to input signal from the clock's rising edge

mandatory

[2] When the clock's falling edge is used, it should also be shifted from the falling edges

recommend 1

Explanation

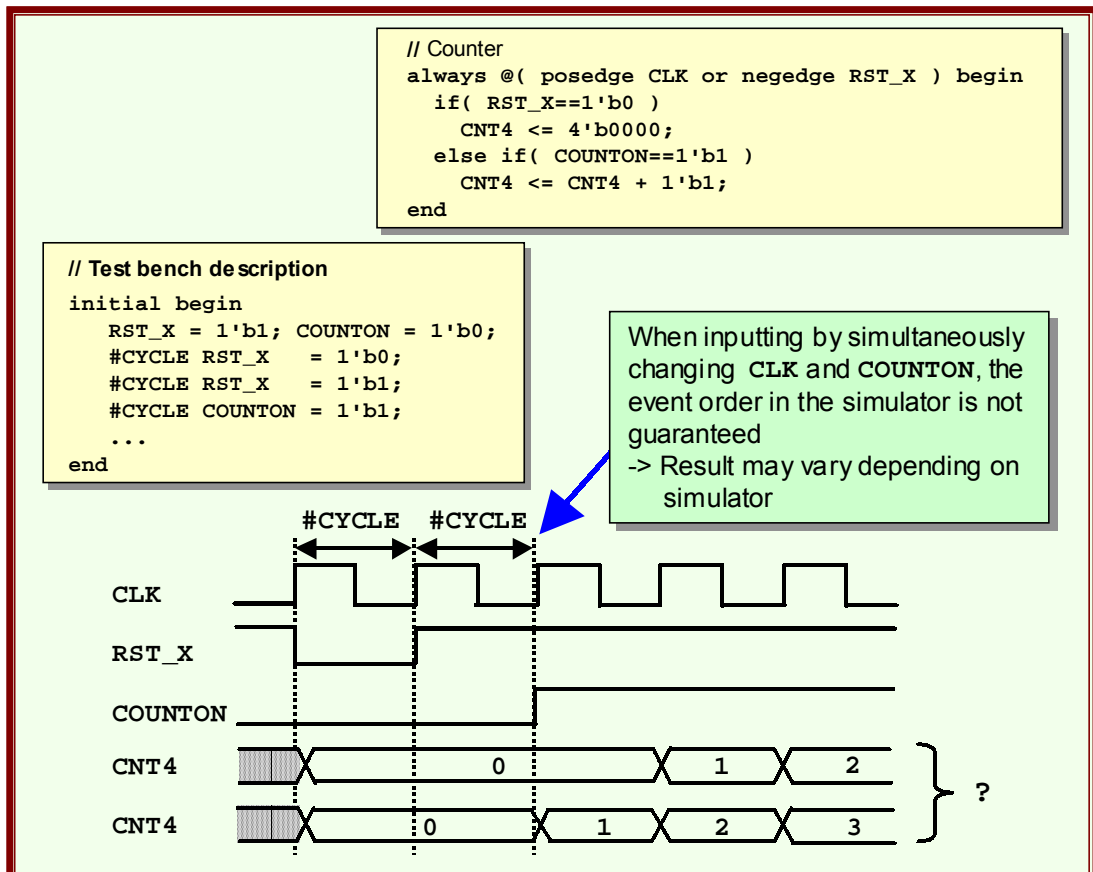


Figure 4-1 Problems with input timing

The Figure 4-1 illustrates a 4-bit counter circuit description and a part of the test bench description for the circuit. In the test bench description, the signals CLK, RST_X and COUNTON are generated based on the CYCLE time.

In this case, CLK and COUNTON change after delta delay from the CYCLE time so that CLK rise and COUNTON change simultaneously. When this CLK and COUNTON are input to the circuit, it is unknown whether the CLK signal rise fetches the value 0 before the COUNTON changed or fetches the value 1 after it changed. Therefore, due to the simultaneous changes of the COUNTON and the CLK signal, two types of CNT4 output are expected as illustrated in the Figure 4-1.

Input timing such as this may cause an error message to be output either by a setup check or by a hold check in the case of gate level circuit. No particular error message is output in the case of RTL descriptions, however. In such situations, it is important to note that making minor corrections to the test bench description or RTL description may render

4.1. Test bench description

them inoperable even if normal function was verified by RTL simulation.

It is ideal for all ASIC circuits to operate only at rising edges, but it is difficult to have the external I/O parts all operate only at the rising edges. Shifting of the input timing is also applied to reversed phase edges as well.^[2]

In order to avoid substitution occurring at the same timing, define the delay values for the input signals in the test bench description, and set the input timing so that the clock satisfies the setup time and hold time and a stable value is read.^[1] Doing so makes it easier to use the same test bench description in gate level circuits.

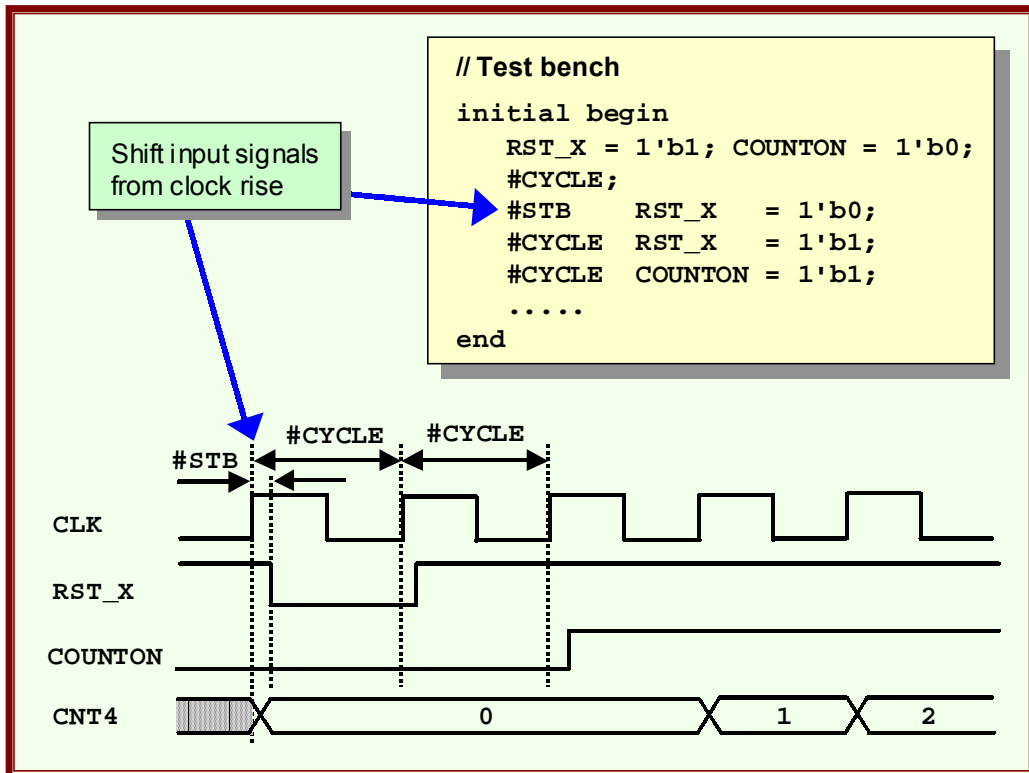


Figure 4-2 Shifting the input timing

With the input timing in the above figure, the timing is delayed only enough for the #STB portion after the initial clock cycle (#CYCLE) elapses. After delaying the timing enough only for the #STB portion, signals are generated synchronous with the clock cycle. Providing timing as explained here makes it possible to guarantee the timing between the clock and the data input.

As far as the time for delaying the timing is concerned, we recommend setting the necessary setup time and hold time carefully to delay timing out of consideration of cases where the same test bench description is used for the gate level simulations.

4.1.4. Avoid assigning from multiple *initial constructs* (differ from VHDL)

- [1] Avoid assigning from multiple *initial constructs* to the same signal (Verilog only)
 - Resulting value is unknown if racing occurs

recommend 1

- [2] Define one signal using one description block (Verilog only)

recommend 1

Example Code

```
initial begin
    #(CYCLE*8) ;
    #(STB) ;
    for(i=0;i<101;i=i+1) begin
        #(CYCLE2BAI) FOO = FOO + 1;
    end
    #(CYCLE*281) FOO = 197;
end;
.....
initial begin
    FOO = 0;
    #(CYCLE*152)
    FOO = 126; .....
```

Example 4-3 Assigning from multiple *initial constructs*

Explanation

If a value is assigned to the same signal in multiple *initial constructs* at the same timing, no guarantee can be made as to which value is assigned. If assignments to the signal FOO conflict with one another as in the Example 4-3, the results may vary depending on which simulator is used. Even when the test bench description is modified, the results may differ.

Avoid describing assignment expressions that assign in multiple *initial constructs* or *always constructs* to the same signal.^[1]

Defining one signal inside one description block makes the pattern definition easier to comprehend.^[2] Use task statements as described in “4.2. Task description” to perform assignments, instead of using multiple *initial constructs* and *always constructs*. In addition to using task statements, an alternative method is to use event statements. When using event statements, pay special attention to avoid the problem of racing in event expressions.

```
event END_PROC_A, END_PROC_B;

initial begin
    INA = 2'b00;
    @(posedge CLK) INA = #DLY 2'b01;
    ...
    @(posedge CLK) -> END_PROC_A;
    #DLY;
    @(END_PROC_B);
    @(posedge CLK) INA = #DLY 2'b11;
    ...
end

initial begin
    #DLY;
    @(END_PROC_A);
    if(FIS == 1'b1)
        INA = #DLY 2'b11;
    end
    #DLY -> END_PROC_B;
end
```

Example 4-4 Execution control using event statement

4.1.5. Describe on a clock edge basis

- [1] Input test vectors synchronizing with clock
 - Reflect the clock cycle changes to other signals
 - Making the timing relationships more comprehensive
- [2] Use *event statements* for clock synchronization

recommend 3

reference

Example Code

```

`timescale 1 ps / 1 ps
module BCD_TEST;
parameter CYCLE = 1000 ;
parameter CYCLE2 = 500 ;
parameter DLY = 10 ;
reg CLK,COUNTON,RST_X;
wire CO;
wire[3:0] LSB,MSB;

BCD100 U1(.COUNTON(COUNTON), .RST_X(RST_X),
          .CLK(CLK), .LSB(LSB), .MSB(MSB), .CO(CO));

always begin
    CLK = 1'b0;
    #(CYCLE2) CLK = 1'b1;
    #(CYCLE2);
end
initial begin
    RST_X = 1'b1; COUNTON = 1'b0;
    #DLY;
    @(posedge CLK) #DLY RST_X = 1'b0;
    @(posedge CLK) #DLY RST_X = 1'b1;
    @(posedge CLK) #DLY COUNTON = 1'b1;
    repeat(300) @(posedge CLK);
    #DLY $finish;
end
endmodule

```

Example 4-5 Input signals synchronized to the clock

Explanation

Clock signals can be used as sync signals of test bench description. If the other signals are synchronized to the clock, it becomes easier to describe timing relationships dependent on the clock.^[1] For test bench description in which delays are defined based on the cycles in blocks in the order indicated above in Example 4-1, the cycles might shift part way through due to description errors.

In the clock edge base description, the input timing of signals can be definite since they are synchronous with the clock. In Example 4-5, signals are generated by @(posedge CLK) synchronous with the clock rise. Moreover, in the description a *repeat* statement is used to count the clock cycles, and simulation continues only for the specified number of clock cycles, after which it is stopped.

In this example, the necessary number of @(posedge CLK) #DLY clock rising edge detection descriptions have been described. However, making descriptions like these for each

and every cycle is complicated. Depending on the situation, it may be better to create a clock for simulation, which is delayed a little from the main clock system. Using an *event* statement as in the following example makes it possible to reduce the size of the description used.^[2]

```
event TRIG;
always @(posedge CLK)
    #DLY -> TRIG;

initial begin
    RST_X = 1'b1; COUNTON = 1'b0;
    @TRIG RST_X = 1'b0;
    @TRIG RST_X = 1'b1;
    @TRIG COUNTON = 1'b1;
    repeat(300) @TRIG;
    #DLY $finish;
    .....
end
```

Example 4-6 Sample description using an *event* statement for clock synchronization

Unlike *reg* declarations or *wire* declarations, *event* declarations do not have Boolean values in the declared signals. *event* declarations merely convey the fact that there has been a change. *Event* statements are not supported by logic synthesis, so they cannot be used in RTL descriptions. However, they are convenient factors in test bench descriptions when synchronizing or controlling flow. When creating test vectors, the assignment position of each signal should be shifted from clock rise to avoid racing in RTL description.

The problem of racing may occur not only between a simulation description and a RTL description, but also between simulation descriptions. A simulation description simulates RTL as well as the gate level circuit. In this case, the simulation will differ unless due consideration is given to the problem of racing.

Taking this issue into consideration, it would be safer when the delay value is described in the assignment statement with non-blocking assignment (*<=*) as follows in Example 4-5:

```
@(posedge CLK ) RST_X      <= #DELAY 1'b0;
@(posedge CLK ) RST_X      <= #DELAY 1'b1;
@(posedge CLK ) COUNTON    <= #DELAY 1'b1;
```

In Example 4-5, the COUNTON signal is not problematic since a constant is assigned. However, if

```
COUNTON <= ANOTHER_SIM_SIGNAL & RTL_OUTPUT_SIGNAL;
```

and the event statement in Example 4-6 is used, the value of ANOTHER_SIM_SIGNAL will be evaluated after the #DLY delay from @(posedge CLK). This causes the evaluation of ANOTHER_SIM_SIGNAL value and its assignment to COUNTON to occur simultaneously, such that there is a risk of the problem of racing. This problem will be discussed in “4.1.8. Descriptions where the results do not differ due to the simulators”.

4.1.6. Set the cycle using parameters

- [1] Do not put cycle computation expressions inside an *always construct*
 - Simulation will be slow. Errors will occur easily

recommend 1

Example Code

```
parameter CYCLE = 1000;
parameter HALF_CYCLE = 500;
parameter STB = 100;

always begin
    CLK = 1'b1;
    #HALF_CYCLE CLK = 1'b0;
    #HALF_CYCLE;
end;
```

Time should all be parameterized.
Do not directly describe time using #.

Do not define this as $CYCLE/2$
Division will round off decimals and may lead to erroneous shifting of cycles

Example 4-7 Setting the cycle

Explanation

In the above example, clocks are generated by setting the clock cycle to 1000 time units and setting the duty cycle to 50%, and assigning 0 and 1 for each 500 time unit. In this case, the parameter statement defines the CYCLE cycle and $CYCLE/2$ as HALF_CYCLE, and is described such that 1 and 0 are repeated for each HALF_CYCLE.

$CYCLE/2$ can be calculated within the *always* statement, but if the CYCLE value is not even, for example, is an odd value such as 9899, the decimal fractions will be cut off in $CYCLE/2$. This problem can be easily overlooked when such time calculations are performed in the test bench description. Values concerning time should be defined in one place using parameter definitions. It is better to avoid using `add(+)`, `subtract(-)` or `divide(/)` for values related to time in test descriptions other than parameter statement.^[1] This will eliminate mistakes because it enables the designer to consider all values.

4.1.7. Define a signal for each clock in a multiple clocks design

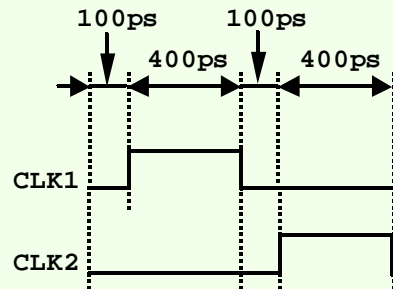
- [1] Separate the descriptions for each clock
- Facilitates clock cycle modifications
 - Facilitates understanding of waveforms

recommend 2

Example Code

```
parameter CYCLE = 1000;
parameter HALF_CYCLE = CYCLE/2;
parameter WIDTH = 400;
parameter DELAY = 100;

always begin
    CLK1 = 0;
    CLK2 = 0;
    #DELAY CLK1 = 1;
    #WIDTH CLK1 = 0;
    #DELAY CLK2 = 1;
    #WIDTH;
end
```



Example 4-8 Sample description of multiple clocks (bad example)

Explanation

Isolate clock signal generation for each clock. The above example involves a description that defined a two-phased clock, but the two clock signals CLK1 and CLK2 are defined in one *always construct*. Since in this description the CLK2 description coexists with the CLK1 description, it cannot be changed easily, even when you try to change only CLK2. Moreover, this description is difficult to understand.^[1] Therefore, be sure to describe each clock independently as shown below.

```
always begin
    CLK1 = 0;
    #DELAY CLK1 = 1;
    #WIDTH CLK1 = 0;
    #HALF_CYCLE;
end

always begin
    CLK2 = 0;
    #HALF_CYCLE;
    #DELAY CLK2 = 1;
    #WIDTH;
end
```

Example 4-9 Improvement of the description in Example 4-8

4.1. Test bench description

4.1.8. Description where the results do not differ due to the simulators (differ from VHDL)

[1]	Shift the observation point of a signal from an assignment point	recommend 1
[2]	Clarify the observation point and the assignment point in clock edge-based descriptions	reference
[3]	Pay due attention to time 0 events in clock edge-based descriptions	recommend 3
[4]	Avoid using edges other than clock signals to the greatest extent possible (posedge CLK only)	recommend 1
[5]	Pay due attention to the timing of asynchronous resets	recommend 2

Explanation

With Verilog-HDL, there are often cases where the output results differ depending on the simulator used. Particularly when moving from RTL development to Gate simulation, different simulators are often used, and designers will frequently be troubled by this problem.

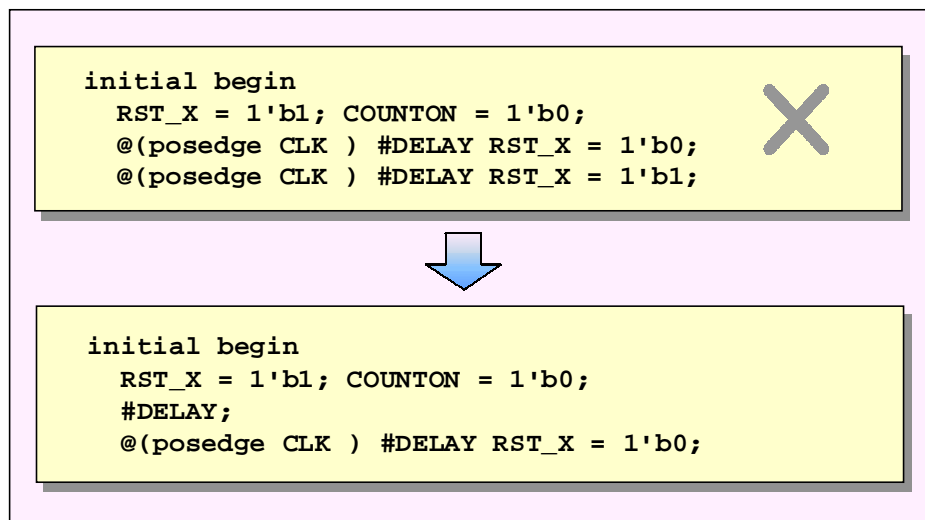
Pay due attention to the following items so that this problem does not arise.

Strictly adhere to items 2.2.2, 2.2.3, and 2.3.1 in the RTL description.

Strictly adhere to items 4.1.3 and 4.1.4 in the simulation description.

In addition to this, there is the problem of edge detection at time 0.^[3] To illustrate this by using the edge-based description introduced in Example 4-5, the first `@(posedge CLK) #DELAY RST_X = 1'b1;` is executed at time 0.

At this time, the clock rising edge may be accidentally detected at time 0 depending on the simulator. Although there should not originally be an edge at time 0, this phenomenon still occurs since at time 0 a value is thought to have changed from 'x' to 1. Insert a delay to avoid this problem.^[1]



Example 4-10 Beware of time 0 edge in clock-based description

The problem that is most likely to occur, when it comes to the results of the simulation differing depending on the simulator, is a problem termed “racing”. In order to avoid racing, pay attention to use a correct approach for RTL description and the simulation description such as described above.

As described in “4.1.3. Note input signal timing”, even though it is necessary to shift the input of the test signal away from the rising edge of the clock, what is needed is not merely to shift the signal, but, if the signal is within the same clock system, all of the signals should be changed with the same timing. Actually, in order to avoid the racing problem, not only must the transition points of the test signals input into the circuit be aligned, but also necessary to align the points, where the signal values are observed, to certain extent. If the positions of the signal observation points are the same as the transition points of the input test signals, the racing problem will result. Pay attention to this issue when coding simulations.

It is recommended that the signal observation points be placed just in front of the rising edge of the clock signal.

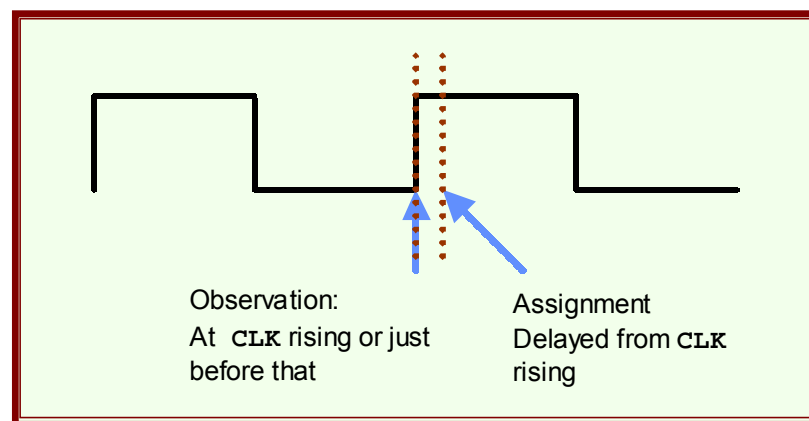
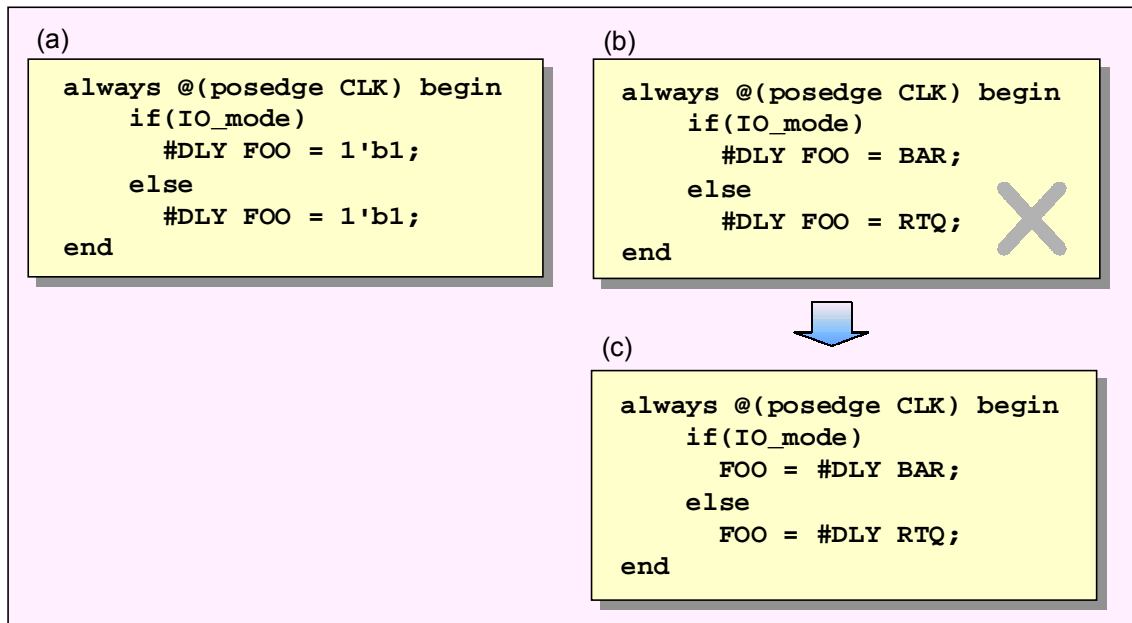


Figure 4-3 Signal observation point and assignment point

`@(posedge CLK)` can be used in order to observe the value just in front of the rising edge of the clock.

If, in the RTL description, non-blocking assignments are used for all FF substitutions, the assignment of the value will be delayed. Additionally, if, in the simulation code, the signal assignment is delayed beyond the rising edge of the clock, then the value of the signal at the point in time of the `@(posedge CLK)` can be confirmed to provide the same observation as would observing the value directly before the rising edge of the CLK.



Example 4-11 Signal observation point and assignment point

In Example 4-11, the code (a) is correct, but the code (b) will result in the racing problem. The code (b) is run by the rising edge of the CLK. The *if statement* condition and the IO_mode value are observed next. The observation of the IO_mode value is made at the point in time of the rising edge of the CLK.

Next, even though the

```
#DLY FOO = BAR;
```

statement is executed, if the delay equation is at the start of the executable statement, the statement will be executed after the DLY delay. As a result, the BAR value is observed after a delay of DLY after the rising edge of the CLK.

In the previous example, the code (b) must be described as (c).

In a statement such as

```
FOO = #DLY BAR;
```

if the delay equation is in the assigned statement (i.e., on the right-hand side), then the observation of the BAR value will be at the rising edge of the CLK, and the assignment of the value for FOO will then be done after a #DLY delay.

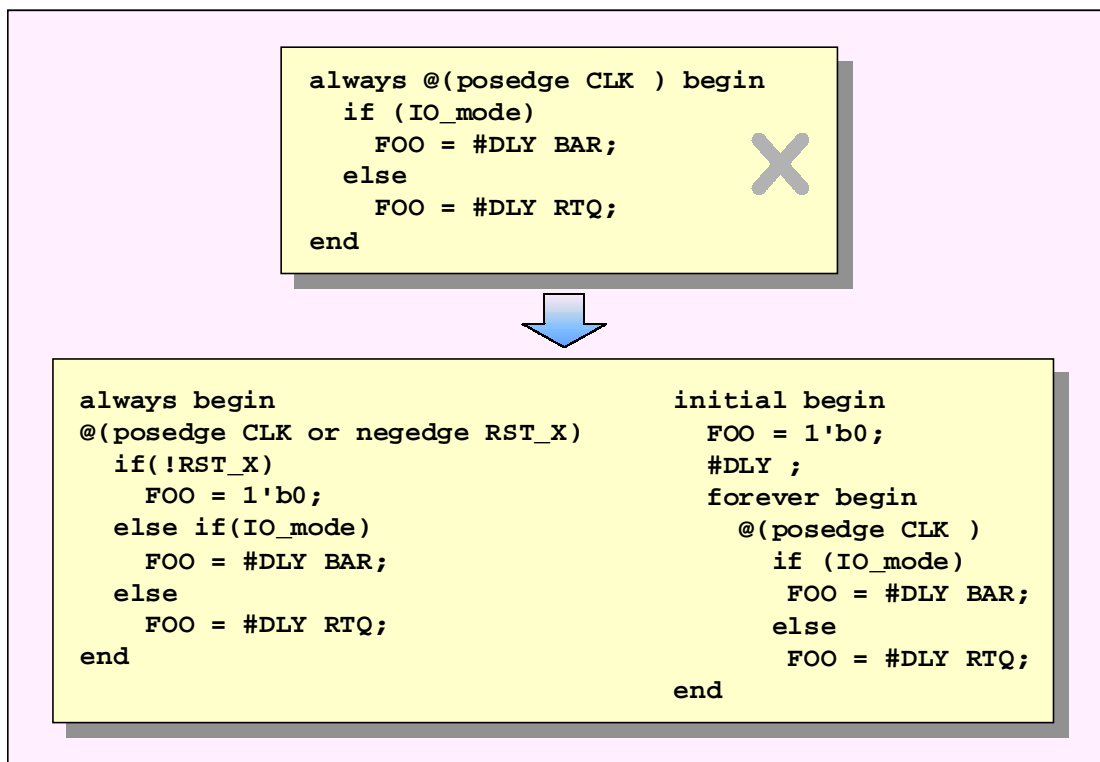
There is no problem with the code (a). In this code, a constant value is assigned for FOO. If the value is a constant value, there is no need to observe the value, so there will be no problems whatsoever with the position of the delay equation.

Actually, the test bench of Example 4-1, presented in “4.1.2. Use basic test vector descriptions” is correct for assigning a constant value for a signal; however, it is not correct for observing the value of a signal and then calculating and performing the assignment. Use the style described in “4.1.5. Describe on a clock edge basis” for the code in this type of

active test bench. When doing so, pay special attention to the position of the delay equation, such as shown in Example 4-11.

In Example 4-11, if CLK signal does not rise other than time 0, then the FOO value is unknown, and 'x' will be output until it does. When a 'x' is produced on an input signal, the RTL simulation results may not match the simulation results at the gate level. See the detailed **Explanation** pertaining to this in “4.4.2. Inconsistencies can occur in RTL and at gate level with the propagation of X”.

In Example 4-11, it is necessary to correct the code so that the value of the signal is determined at time 0. If the reset signal RST_X (negative logic) goes to '0' in the first operation of the simulation, then the value can be determined by the code on the left-hand side of Example 4-12. If the reset signal is not used, then the code on the right-hand side of Example 4-12 may be considered.



Example 4-12 Description that resets to prevent X at time 0

In the simulation code, it does not matter whether the assignment of the signal uses blocking assignment (=) or uses non-blocking assignment(<=). If a delay equation is used in the assignment, the point at which the value is observed and the point at which the assignment is made should be clearly defined. If values are assigned to two different signals at the same time, then either the non-blocking assignment such as shown in Example 4-13 should be used, or a fork-join should be used for a blocking assignment.

4.1. Test bench description

```

always begin
  @(posedge CLK or negedge RST_X)
    if(!RST_X) begin
      EST <= 1'b0;
      FOO <= 1'b0;
    end
    else if(IO_mode) begin
      EST <= #DLY ETH;
      FOO <= #DLY BAR;
    end
    else begin
      FOO <= #DLY RTQ;
    end
end

always begin
  @(posedge CLK or negedge RST_X)
    if(!RST_X) begin
      EST = 1'b0;
      FOO = 1'b0;
    end
    else if(IO_mode) fork
      EST = #DLY ETH;
      FOO = #DLY BAR;
    join
    else begin
      FOO = #DLY RTQ;
    end
end

```

Example 4-13 2 signals described in edge-based description

In Example 4-14, the code is such that the operation is performed with the signal edge. In this type of code the point at which the signal is assigned and the point at which the signal is observed are at the same point, and thus a racing problem may occur. As is shown in Example 4-15, changes should be made for all signals so that the values are observed on the rising clock edges.


```

wire FIN_signal;

TEST_CURCUIT U1(..., .FIN_signal(FIN_signal))

initial begin
  FIN_count = 0;
  @(posedge FIN_signal)
    if(FIN_Enable)
      FIN_count <= #DLY FIN_count + 1;
end

```



Example 4-14 Description that uses the signal edge

```

initial begin
  FIN_count = 0;
  @(posedge CLK)
    if(FIN_Enable && FIN_signal && !FIN_signal_mae)
      FIN_count <= #DLY FIN_count + 1;
end
always @(posedge CLK)
  FIN_signal_mae = #DLY FIN_signal;

```

Example 4-15 Modified example 4-14 in clock edge-based description

When the operations outside of the LSI or the FPGA are asynchronous, then it may be necessary to use the signal edge in order to express the operation correctly. This type of asynchronous operation is prone to have racing problems. Use caution to ensure safe code

so that there is no racing problem, doing so by either assuming a virtual synchronization or by using the correct delay equations.

As was described in “2.3.1. Unify the description style of FF inferences”, it is wise to insert delay values in assignments to FFs in order to prevent racing problems in the circuits that include, for example, gated clocks. Note, however, that racing problems may still occur even when delay values are inserted in the assignments to the FFs. While in the code in Figure 4-4 there is a delay of #DLY (10 units) from the CLK rise when making the assignment to the FF, the asynchronous reset signal is shifted by #STB (5 units) from the CLK rise. In such cases, different simulators may produce different results if the point at which the assignment is made to RST_X is shorter than the delay value DLY that is inserted when making the assignment to the FF. While one simulator may set the output signal Q to '0' because RST_X is '1' after five units have elapsed, after five units it will return to '1'. This is because even though Q = #DLY D; is executed at the rising edge of the CLK, the value of D is assigned to Q after 10 units delay, causing the Q output to go to '1'. However, depending on the simulator, the result at the bottom (simulator B) may be produced, the results may not match.

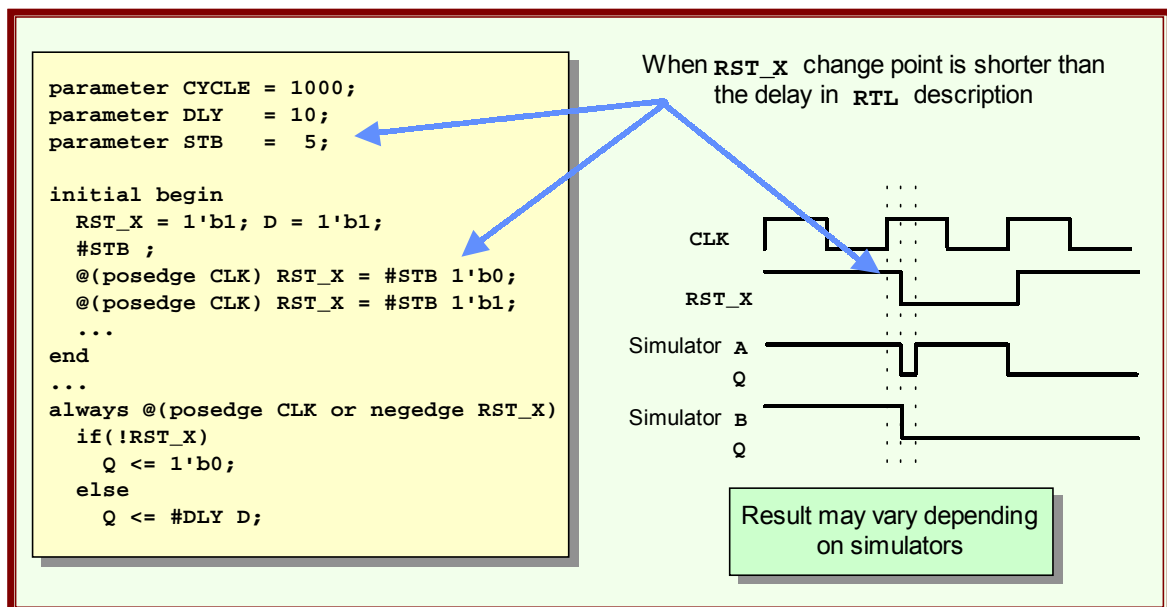


Figure 4-4 Simulation result will not match if RST_X delay value is smaller than the one used for FF assignment

In Figure 4-5, a delay of #DLY is added to the asynchronous reset input as well. When this is done, the results in the large majority of simulators will be the results at the bottom (simulator B); however, this does not fully solve the problem because there are still some simulators that will produce the results at the top (simulator A).

The value shorter than the delay value, which is inserted when assigning the value to the FF should not be used for the delay value of asynchronous reset input. There will be no problem if the change is either at the same time as the delay time that is inserted when assigning the value to the FF, or later.

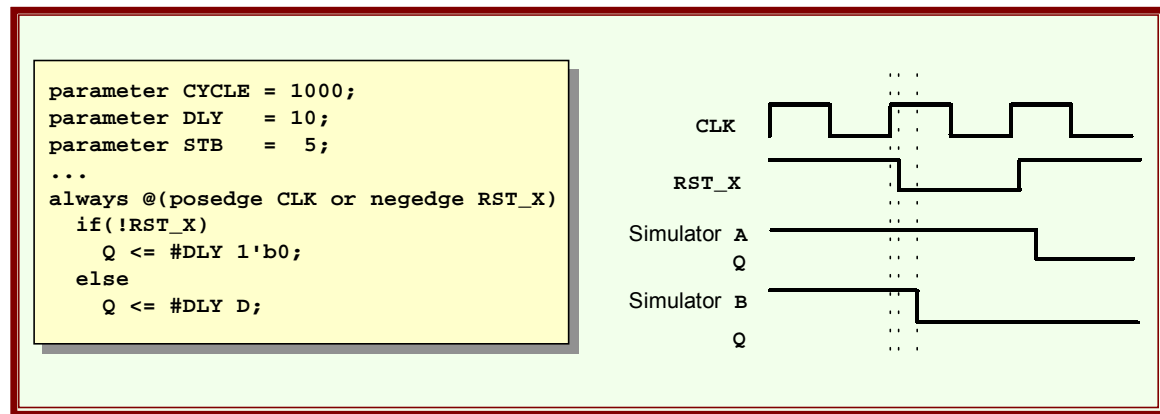


Figure 4-5 Even if adding delay value to RST_X assignment, there is no guarantee that the result will match

4.1.9. Simulation between asynchronous clocks

- | | |
|---|-------------|
| [1] Pay attention to racing in case of the simulation between asynchronous clocks | reference |
| [2] Use (<=) for the simulation description between asynchronous clocks | recommend 2 |
| [3] Change delay values to check racing problems | reference |

Example Code

```

parameter MAINCYCLE = 100;
parameter MAIN_HALF_CYCLE = 50;
parameter MAINDELAY = 1;
parameter CPUCYCLE = 270;
parameter CPU_HALF_CYCLE = 135;
parameter CPUDELAY = 1;

always begin
    CLK_M = 1'b1;
    #(MAIN_HALF_CYCLE) CLK_M = 1'b0;
    #(MAIN_HALF_CYCLE);
end
always begin
    CLK_CPU = 1'b1;
    #(CPUCYCLE) CLK_CPU = 1'b0;
    #(CPUCYCLE);
end
task write_reg;
input ADDRESS;
input DATA;
    @(posedge CLK_CPU) AB <= #MAINDELAY ADDRESS; DIN = DATA;
    @(posedge CLK_CPU) IOWB <= #MAINDELAY 1'b0;
    @(posedge CLK_CPU) IOWB <= #MAINDELAY 1'b1;
endtask

```

Example 4-16 Description using asynchronous clocks

Explanation

In simulations involving asynchronous clocks, attention should be paid to racing problems. In Example 4-16, the cycle 100 of the main clock system CLK_M and the cycle 270 of the CPU interface clock CLK_CPU operate asynchronously.

In descriptions involving asynchronous clocks, each cycle should be described by separate *always* constructs. The signals to be synchronized with each clock should be synchronized with the each clock using @(posedge) without exception. In simulations between asynchronous clocks, each cycle must be changed for operation so that the problems of the circuits of the asynchronous part are checked. Moreover, when creating patterns for a LSI test, this cycle should be the same (or a multiple of one clock).

Make descriptions while paying due attention so that operation is proper even if you change the cycles. Operations between asynchronous clocks will cause racing problems at places where you change at the same timing, as shown in Figure 4-6.^[1]

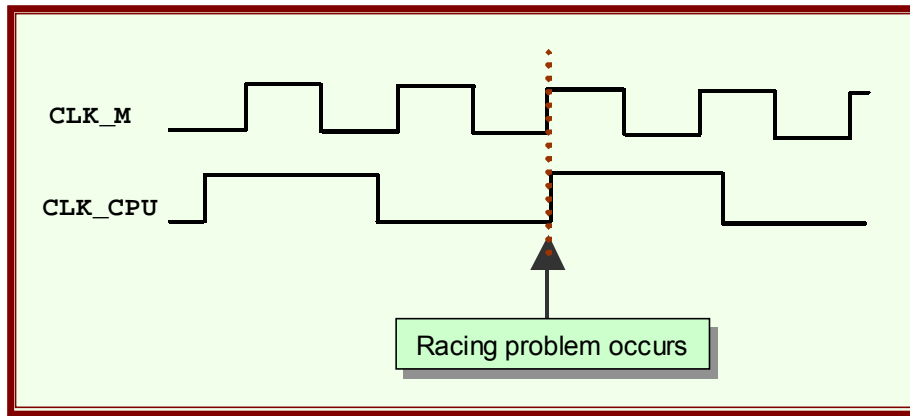


Figure 4-6 Racing of CLK_M and CLK_CPU

As explained in “4.1.3.Note input signal timing” and “4.1.5.Describe on a clock edge basis”, this description,

```
@(posedge CLK_CPU ) #DELAY AB <= ADDRESS;
```

makes the signal evaluation timing and the assignment timing simultaneous, so the risk that racing problems will occur increases. As Example 4-16, be sure to make descriptions with delay values using non-blocking assignment statements (`<=`). ^[2]

The DELAY value specified by the parameter statement is normally set at a very small value (1 in Example 4-16). When a large value is set for the cycle, the operation is delayed for 1 clock as the Figure 4-7, if the rising edge shift of CLK_CPU side is the same as the delay value or less at the clock crossing point.

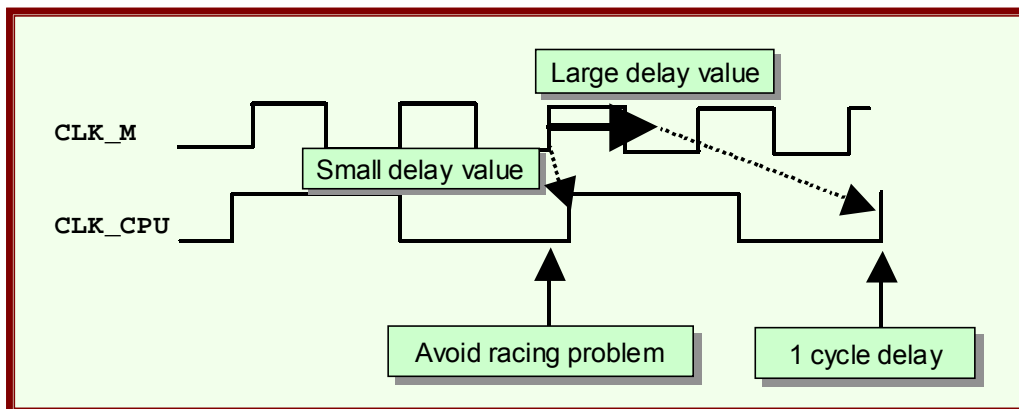


Figure 4-7 A large delay value may cause a 1 cycle delay

The DELAY value is normally set at a small value. There is a method to analyze the data transfer between asynchronous clocks by the check that the same data is correctly transferred after a value is changed to a large value (slightly shorter than the cycle depending on the situation).

4.1.10. Use handshakes when the process cycle count is unclear

- [1] Use handshakes when the process cycle count is unclear
 - You can change the test input dynamically by observing the circuit status

recommend 2

Explanation

In order to clearly evaluate the simulation results, it is necessary to compare them with expected values. Please refer to "4.3.5. Compare expected value files with the simulation results" and "4.3.6. Simulating during comparison of expected values" for details on comparison of expected values. Handshakes should be considered when comparing the simulation results with expected values.^[1] In circuit design, a circuit has been modified in the course of design, so at such times there are many instances where the number of cycles from input to output ends up changing from what it was at the outset of design. If you compare by fixing the position when you verify the expected values, correction also becomes necessary when you change the circuit. Owing to this, it is necessary to make it into a description that can be used generically even if you change the cycle count. If the RDY and DONE signals exist in the circuit in such situations, then a test bench should be described using these signals.

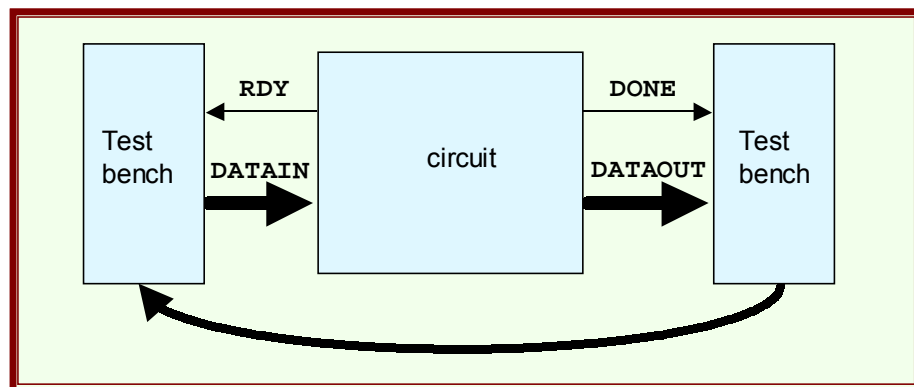


Figure 4-8 Handshake via signal observation

In Figure 4-8, data are input from the test bench when the RDY signal is output from a circuit, and completion of the operation process is acknowledged by the DONE signal. Furthermore, the next RDY signal approves inputs of the data required for the next operation.

4.1. Test bench description

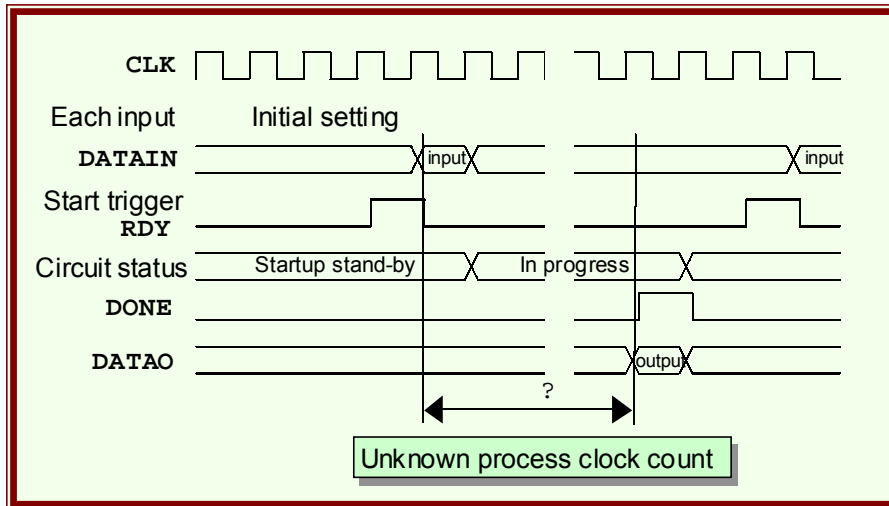


Figure 4-9 Necessity of handshakes

As illustrated in Figure 4-9, when you input the signal while waiting for the RDY signal, use for example “while” to describe it as follows.

```
initial begin
    DATAIN = 8'h00;
    #DLY;
    while (RDY==1'b0) begin
        @(posedge CLK) ;
    end
    DATAIN = #DLY A + B;
```

Similarly, for DONE signal, describe as follows:

```
always begin
    @(posedge CLK)
    if(DONE == 1'b1)
        Res = DATAO;
    ....
```

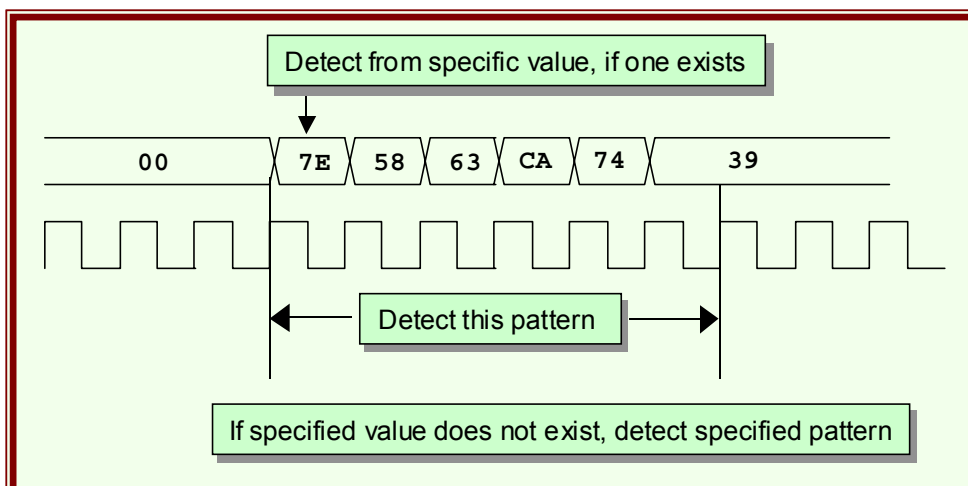


Figure 4-10 Sample implementation of handshakes

Some circuits may not have output equivalent of the RDY or DONE signal.

However, it should be possible to determine handshakes even when a specific output signal is not available. For example, when the value "7E" is observed for the output, it may not be adequate to determine that some output has begun by that value alone. But it may be possible to make determinations if we detect the values up to the next output "58". Some decision should be possible if you consider the matter in terms of sequential circuits rather than trying to rely on one signal or one cycle of a value. Handshakes will be useful since even suppositions are acceptable.

4.1.11. Output simulation results to a file

- | | |
|--|-----------|
| [1] Use "\$fstrobe" rather than "\$fdisplay" for outputting the results (Verilog only) | reference |
| [2] Use \$display, \$fdisplay to display error messages (Verilog only) | reference |

Example Code

```

parameter CYCLE = 20;
parameter PNUM = 131019;
integer I, J, mcd1;

initial begin
    mcd1 = $fopen("pattern");
    #(CYCLE - 2)
    for(J=; J<=PNUM; J=J+1) begin
        #(CYCLE)
        $fstrobe(mcd1, "%b%b%b%b%b%",
                SECLSB, SECMSB, MINLSB, MINMSB, HOURLSB, HOURMSB);
    end

    $finish;
end

```

Example 4-17 Sample description of file output

Explanation

Simulation results can be output to files. When writing to files, using \$fstrobe is safer than using \$fdisplay. \$fstrobe executes file output after all event changes at the specified time are completed.

If event changes occur simultaneously with file output, \$fdisplay outputs at the same time as the execution, so neither the value from before when an event changes nor the value from after when the event changes is guaranteed. This is why the output value may vary depending on the execution environment. Since in the case of \$fstrobe all event changes at that specified time are complete and are executed immediately before proceeding to the next specified time, the output of the value from when the event change ends is guaranteed. Of course, as explained in "4.1.8.Descriptions where the results do not differ due to the simulators", if the observation point of a signal is different from assignment point, this kind of problem will not occur. If it is a correct simulation description, either one can be used.

Use \$fdisplay when defining error messages in the description. The reason is that if an error occurs because of debugging, it is necessary to stop the simulation to check the signal condition at event change.

Only one bit is set to 1 in order from the second bit of the LSB of mcd (ex. 000010, 000100, 001000) each time \$fopen is executed. If you need output to multiple files, add the mcd value as mcd1 + mcd2 and then define it. Since the LSB is the bit that determines whether or not to output to the display, in the event that you would like it to be forcibly output to the display during debugging, execute it from a command as "force mcd1 = mcd1+1;"

4.1.12. Use PLI (Verilog only)

(reference: There may be cases where PLI use is not be permitted due to use flows)

- | | |
|--|-----------|
| [1] Useful for interfacing with other tools (Verilog only) | reference |
| [2] Useful for creating test benches that require complex operations (Verilog only)
- Verilog-HDL has no operation functions. Even if such operation functions were defined, they would have poor functionality and thus would be difficult to use | reference |
| [3] There is little speed advantage (Verilog only)
- For compile type or cycle based type simulators, it would be faster to describe bus models etc. using Verilog-HDL
- It is more advantageous to describe simulation models such as macro cells using Verilog-HDL when taking generality into account | reference |

Explanation

PLI (Programming Language Interface) is for interfacing C language programs and Verilog-HDL simulators. It can be used to build system tasks or functions. Using PLI makes it possible to not just create simulation models, but to add many functions, such as test bench generation, system task debugging, netlist analysis, and simulation result dump, to the simulator.

When PLI is used in the test bench as shown below, for example, it is possible to execute DCT on data from an image file using the C language, and then input the results to a circuit. There is no need to describe a large amount of data in HDL when using the C language. It is also possible to revert the results output from the circuit and use C language routines to display them.

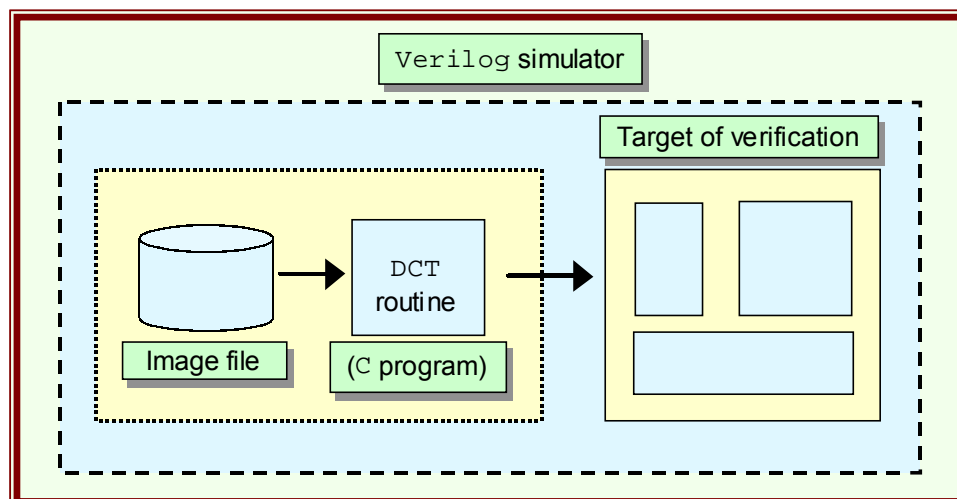


Figure 4-11 Creation of test benches using PLI

Many C routines that can be linked to simulators are provided with PLI. Describe programs using such C routines, compile and link them using a C compiler, and then execute user-defined tasks on the simulator.

```

#include "stdio.h"
#include "veriusert.h"
#include "acc_user.h"
pat_gen()
{
    int count;
    int wait, load, inc, d;
    int wait_count;
    acc_initialize();
    count=(int)agg_fetch_tflag(1);

    switch ( count ) {
        case 6: load = 1; d = 8; break;
        case 12: load = 1; d = 2; break;
        default: load = 0; d = 0; break;
    }
    switch( count ) {
        case 9: wait = 1; break;
        default: wait = 0; break;
    }
    switch( count ) {
        case 9: wait_count++;
                printf( "%d\\n", wait_count );
                if( wait_count==3 ) {
                    wait=0;
                    wait_count=0;
                }
                break;
        default: wait_count=0;
                break;
    }
    inc=1;
    tf_putp( 2, load );
    tf_putp( 3, wait );
    tf_putp( 4, inc);
    tf_putp( 5, d );
    acc_close();
}

char *veri_user_version_str = "";
int (*endofcompile_routines[])() = { 0 };
bool err_intercept( level, facility, code )
int level;
char *facility;
char *code
{ return(true);
}

s_tfcell veriusertfs[] = {
    { usertask, 0, 0, 0, pat_gen,
      0, "$pat_gen", 1 }, { 0 }
};

```

Example 4-18 Sample PLI description

```

reg LOAD, WAIT, INC, RST;
reg [3:0] D;
wire [3:0] count_out;

counter ul( RST, CLK, D, WAIT, LOAD, INC, count_out);

always @( negedge CLK )
    $pat_gen( count_out, LOAD, WAIT, INC, D );

```

Example 4-19 Sample PLI call

4.2. Task description

4.2.1. Describe using tasks to provide structure

[1] Make it possible for test bench description to be more efficient and structuralized

reference

[2] Define application tasks by combining basic tasks

recommend 3

Explanation

Using tasks makes it possible to describe structural test benches. The difference between tasks and functions is that functions call and execute at the same time but cannot internally define timing descriptions, while tasks can internally define timing descriptions such as event control or delay. In addition, functions have only one return value while tasks can return multiple values. Given this, functions are suitable for describing combinational circuits, and tasks are suitable for test bench descriptions.

Describing basic operational procedures using tasks and executing simulations while calling them decreases the amount to be described, reduces errors, and makes it possible to create highly readable test bench descriptions with high debugging efficiency.

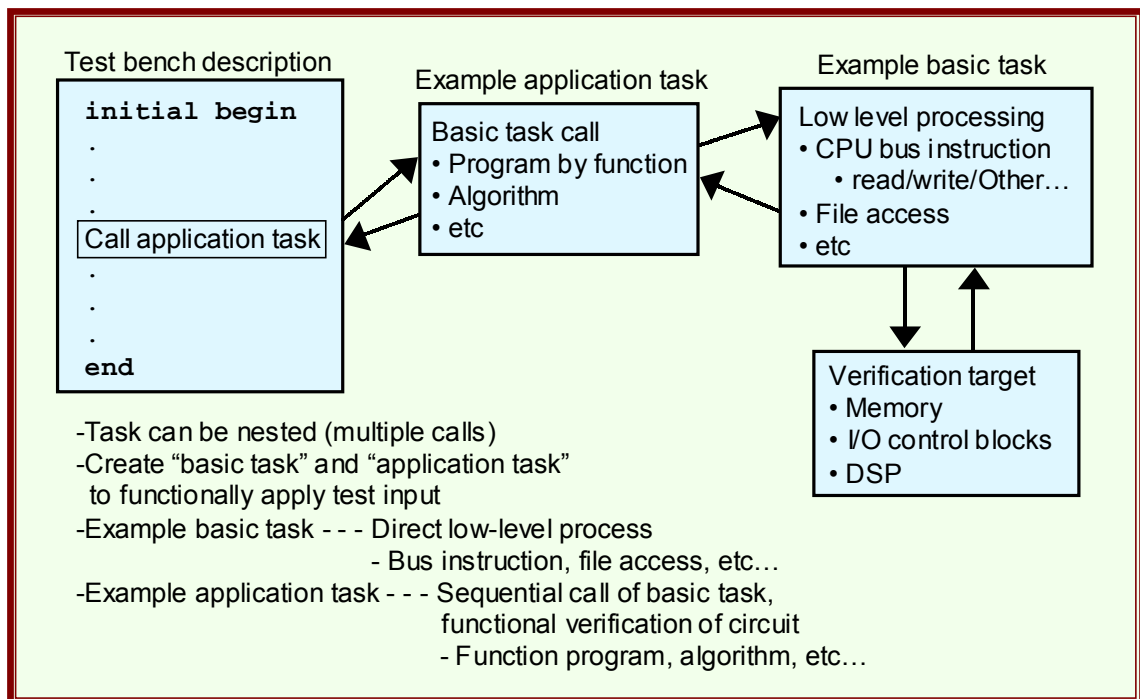


Figure 4-12 Structural test benches implemented by tasks

As illustrated in Figure 4-12, direct low-level processes such as bus instructions and file access are defined by basic tasks. In addition, combining these basic tasks in accordance with the procedure for application tasks, and defining a series of algorithms and functions makes it possible to define structural test bench descriptions.^[2]

4.2.2. Describe function operations using tasks

- | | |
|---|-----------|
| [1] Model functional operations like bus operations using tasks | reference |
| [2] It is possible to define functional operations by calling tasks
- Describing detailed waveforms is not necessary
- The descriptiveness and debugging efficiency improve | reference |
| [3] It is also possible to use commercially available bus model descriptions | reference |

Explanation

When the circuits to be verified are connected to standard busses, it is necessary to define the environment for reading data from the busses and the environment for writing data to the busses. It is possible to write each bus operation in a waveform description, but this involves a greater risk of mistakes since the amount of code increases accordingly. If there are errors in the timing description, it will be difficult to distinguish whether the problems are in the circuit specifications or in the bus description, and debugging will require a long time for completion.

Such basic bus operations do not necessitate describing each waveform by using tasks to define them, and it is possible to describe bus operations in the execution command format.^[1] As a result, it becomes possible to define efficiently the test bench descriptions and improve the debugging efficiency.

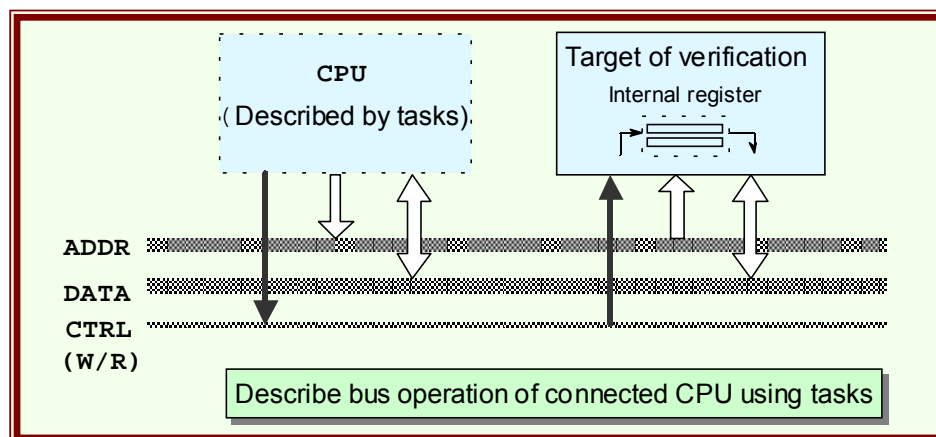


Figure 4-13 Verification using description of bus operation with tasks

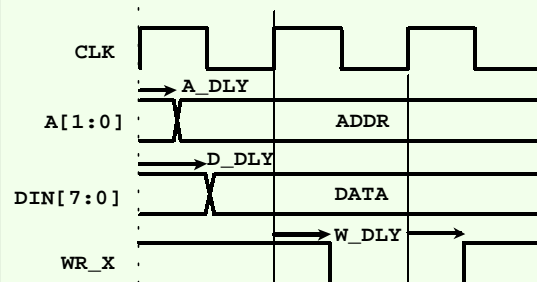
In Figure 4-13, the CPU and the target circuit are connected by a bus. Describing data reading and writing to the CPU data bus using tasks confirms the operation of the circuit that is the target of verification.^[2] The following is a sample task description for reading and writing data to an actual bus. Tasks, unlike functions, can describe the timing. Example 4-20 shows the operation for multiple cycles.

There are two arguments to the task: address (ADDR) and data (DATA). The other signals described in the task are the signals in the module (A, DIN and DOUT). That means that the tasks directly access these signals in the module. Therefore, be careful not to access the same signal from other descriptions.

The output arguments defined by “output” in a task are usually not defined. Tasks read in the value of the input argument at the time it is called, and it transfers the value to the output argument when the task processing is completed. Therefore, if you need to create an event on a signal in task processing, you will have to directly access signals within a module without the use of arguments.

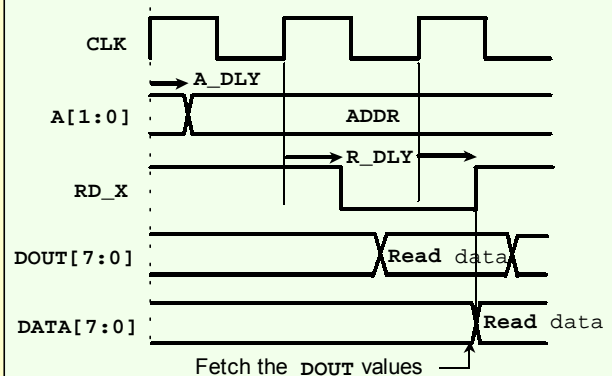
```
// Write to register
// (CPU bus write cycle)
// Method used: write_reg(ADDR, DATA);

task write_reg;
  input  [1:0] ADDR;
  input  [7:0] DATA;
begin
  @(posedge CLK)
    A   <= #A_DLY ADDR;
    DIN <= #D_DLY DATA;
  @(posedge CLK)
    WR_X <= #W_DLY 1'b0;
  @(posedge CLK)
    WR_X <= #W_DLY 1'b1;
end
endtask
```



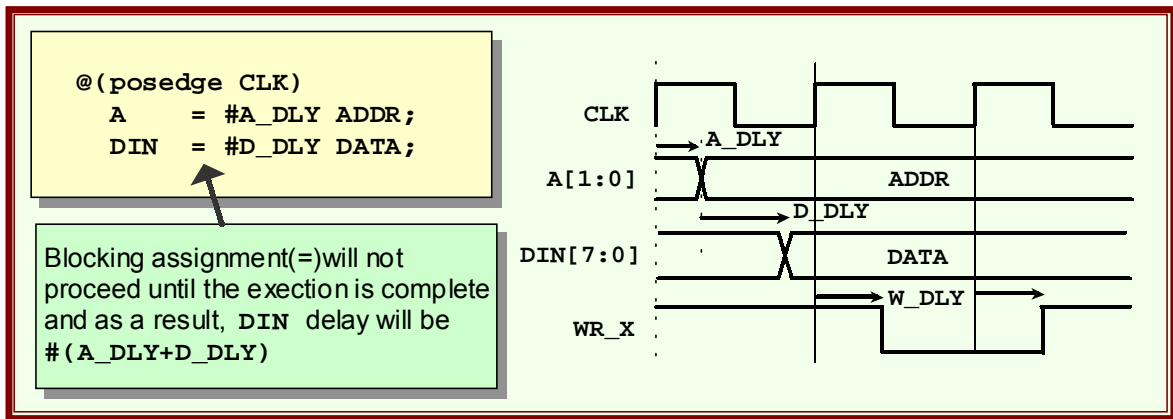
```
// Read from register
// (CPU bus read cycle)
// Method used: read_reg(ADDR);

task read_reg;
  input  [1:0] ADDR;
begin
  @(posedge CLK)
    A   <= #A_DLY ADDR;
  @(posedge CLK)
    RD_X <= #R_DLY 1'b0;
  @(posedge CLK)
    RD_X <= #R_DLY 1'b1;
    DATA <= DOUT;
end
endtask
```



Example 4-20 Sample task description of read/write operation to register

With task, you should be careful about blocking assignment and non-blocking assignment usage. As illustrated in Example 4-21, the delay values are handled differently.



Example 4-21 Different assignment causes different delays

The following is an example of tasks in Example 4-20 used in a test bench. In Example 4-22, the arguments for the task are not defined, but this is a viable description. In this task example, the arguments are not defined and instead a reg signal within a module is directly accessed. See “4.2.3. Take note of task I/O arguments” for details.

This example defines a structural test bench by calling a task (write_reg and read_reg) different from the task (IoRegVerificationTask) according to the procedure.

```

// Task execution order : 1) Reset, 2) Start process, 3) Wait for process finished

parameter COMMAND=2'b00, STATUS=2'b00, DATAREG=2'b01; //Internal register address
parameter SOFT_RESET=8'h03, BUSY_FLAG=8'h01, TRIG_CMD=8'h01; //Command

task IoRegVerificationTask; //Command I/O block function verification
task
  write_reg( COMMAND, SOFT_RESET ); //Soft reset
  write_reg( COMMAND, TRIG_CMD ); //Launch command
  while (DOUT == BUSY_FLAG) //BUSY check
    read_reg(STATUS); //Read status register
endtask

initial begin //Call application task
  IoRegVerificationTask;
end

```

Example 4-22 Process description example of I/O block functional verification

The test bench can be described in a simple and organized way by preparing several structured tasks and describing them in the order of verification, as illustrated in Example 4-23. In this example, concurrent execution of tasks is done using fork-join. In this example, a fork-join block is created in the main test bench, from which a several tasks are called. Execution statements in a fork-join are processed simultaneously, and all three tasks are executed at the same time. (See Figure 4-14).

```

initial begin
  fork // Simultaneously process inside fork-join, the following 3 tasks will operate simultaneously
    IoRegVerificationTask;
    TimerVerificationTask;
    IrqVerificationTask;
  join // Proceed after all tasks inside fork-join have completed

  fork
    ...
  join

  FuncAVerificationTask; // Serially process tasks inside begin-end
  FuncBVerificationTask; // After one task ends, the next will be executed
  ...

  $finish // End of simulation
end

```

Example 4-23 Description of tasks executed concurrently

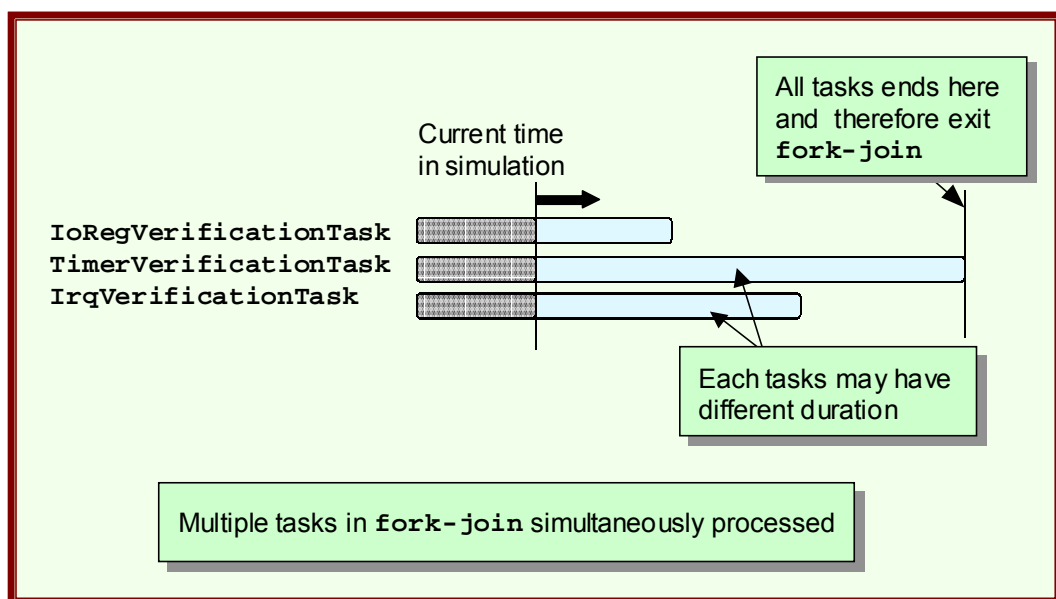


Figure 4-14 Simultaneous processing inside fork-join

4.2.3. Pay due attention to task I/O arguments (differ from VHDL)

- | | |
|--|-------------|
| [1] It is not always necessary to describe arguments to tasks (Verilog only) | reference |
| [2] Do not define output arguments when generating test vectors using tasks | recommend 1 |
| [3] Tasks can define multiple I/O arguments | reference |
| [4] Do not define as input arguments when regularly observing the signals inside a task (Verilog only) | recommend 1 |

Explanation

With tasks, it is possible either to define I/O or not to define I/O.^[1] When defining I/O, input values are transferred to the tasks when the tasks are executed, but output values are returned after tasks are complete. When not defining I/O to tasks, either the signals or variables in the module can be read in real time or directly assigned to the variables.

If you define an output to a task, a test vector cannot be described in accordance with the simulation time because a task will return the value to the output after the execution is completed.^[2] In such a case, you can directly assign values to the signals in a module (A, DIN).

```
//Write to register
// (CPU bus write cycle)
// Method used : write_reg(ADDR,DATA);

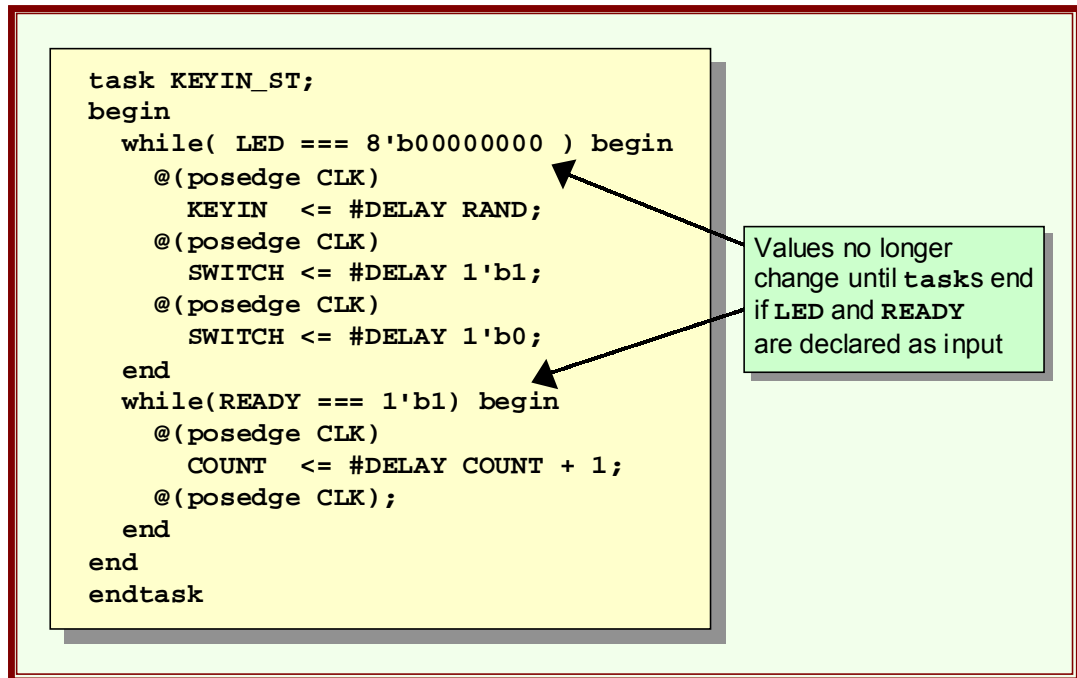
task write_reg;
    input  [1:0] ADDR;
    input  [7:0] DATA;
begin
    @(posedge CLK)
        A    <= #A_DLY ADDR;
        DIN  <= #D_DLY DATA;
    @(posedge CLK)
        WR_X <= #W_DLY 1'b0;
    @(posedge CLK)
        WR_X <= #W_DLY 1'b1;
end
endtask
```

I/O can be omitted.
To be described only for specific clarification.
Output is not defined in this case.

Example 4-24 Abbreviated task output list

With this task, the address and data values are passed to task, and the test vector that is written to the register is generated.

Moreover, if the LED or READY signals are declared as inputs to a task as shown below, these signals are passed to the task only when the task is called. It will not always be possible to observe the signal values.



Example 4-25 Caution about the task output list

Avoid declaring these signals as an input argument to the task when it is always necessary to observe signal changes inside the task.^[4]

4.3. Verification Process

4.3.1. Making verification flow

[1] Create test specifications, and use them as a checklist	recommend 1
[2] Initial debugging stage with RTL is done by checking the waveform output	reference
[3] To clarify verification, generate expected values by Verilog description and then compare them	recommend 2
[4] If generating expected values by Verilog-HDL description is difficult, compare with the previous results	recommend 1
[5] With gate level simulation, verify only the matching with RTL simulation	recommend 1

Explanation

Verification is an important work, so it must be performed efficiently. If verification is inadequate, malfunctions will occur after a device is made. Verification must therefore be performed with a thorough understanding of the operation specifications. An index for how to proceed with verification is introduced below.

* Create test specifications and a checklist^[1]

Create test specifications as design specifications for the test bench. Use the test specifications to organize the verification strategy and check for omissions in the functionality checks. Test specifications should not define test bench signals in detail, but should instead specify what kind of test is to be performed. Test benches are described based on item of the test specifications. Create checklists for the test and manage the overall verification process.

* Output results check and Debugging^[2]

RTL mainly checks the results by checking the output of signal values for each waveform output or strobe. The normal operation of the blocks that perform external control such as memory controllers and peripherals can be checked even with a partial understanding of the control specifications as long as the target operation model is connected and verification proceeds.

If the expected operation was not achieved as a result of verification, debugging continues by tracing the values of specific signals. Debugging uses breakpoint settings, step tracing, or internal node probing. There are also cases in which source line debugging is effective in resolving problems in the initial description, but it will be rendered ineffective if the description is too complex. It is difficult to check all specifications from just the waveforms, so output the necessary internal node values and check them.

* Compare with expected value^{[3] [4]}

It is difficult to check all results accurately by waveforms. Making comparisons with the expected values is effective in improving verification reliability. However, creating expected values using 0 or 1 is difficult in the case of large designs. The best

method is to generate the expected values in the test bench using Verilog-HDL descriptions, and then to compare these with the values derived when the simulation is executed. If describing by Verilog-HDL is difficult, you may instead compare the simulation results with the expected values generated by using another language such as C.

In large systems, a considerable amount of time is devoted to creating these expected values. If it is difficult to describe the expected values, use the method of comparison with the previous simulation results, as explained in “4.3.4. Use batch file for simulation”.

*** Gate level simulation^[5]**

In gate level simulation, check whether or not the results match those of the RTL simulation. This check is performed by output the simulation results as text (\$fstrobe), and using the *diff* command in Unix, etc., for comparison with the RTL results. Gate level simulation mainly uses maximum delays, but we recommended running a simulation with minimum delay as well.

If the gate level simulation and RTL simulation results differ, you should first suspect 'x' propagation. See “4.4. Gate level simulation” for more information regarding gate level simulation.

4.3.2. Create test specifications

- | | |
|--|-------------|
| [1] Use test specifications to organize the verification contents and to check for functional check omissions
- It is important to know what tests were performed | recommend 1 |
| [2] List the test items in simple terms | recommend 1 |
| [3] Create test specifications before starting verification | recommend 1 |

Explanation

In the design of large scale LSI circuits, it is essential to prepare the test specification as the test bench design specification. The test specification is a document that defines the tests to be performed, and thus it is created for the purpose of ensuring that there are no omissions in the functional checks.^[1] The test benches for the test of smaller LSI circuits on a more conventional scale were created by defining an in-house pattern creation tool, a program in C language, or by defining input vectors from an editor. At that time, the documentation pertaining to the design itself took priority over verification documentation. Although circuit diagrams and design specifications were carefully organized, clearly defined test specification documents were less common. However, as the scope of the circuits to be tested became larger, it became difficult to create test patterns and confirm the test results, leading to many omissions in the items tested. Therefore, the details of testing should carefully be examined based on LSI specification given by a project leader of a large scale LSI design.

If the test specification defines details of logic levels, then the test specification for a large-scale circuit design would become extremely complex. It is important that the test specification define what type of tests (verifications) to be performed. The test specification should be written in simple, easily understood language.^[2]

*** Specific Example of a Test Specification**

In order to explain test specifications, Figure 4-15 shows an example of a simple circuit. There is a DSP in this drawing, which should be considered to be an IP that contains internal RAM (a design that is sold by RTL, layout data, etc.). Let us assume that we have a perfect design, in which there is not a single bug. While of course there will never be an IP purchased from the outside that is completely bug-free, let us assume, for convenience in this discussion, that the IP is perfect.

For this DSP, let us assume that the circuits have been designed for the surrounding circuits, the ISA bus interface part, the part that connects the DSP with the program memory (PRAM), and the parts that connect the A/D and D/A connectors. Although nowadays the ISA bus has nearly vanished completely, in the past it was a famous interface specification for I/O in PCs. This system is a PC expansion board that downloads program data from the CPU and performs signal processing, such as sound generation, on that data.

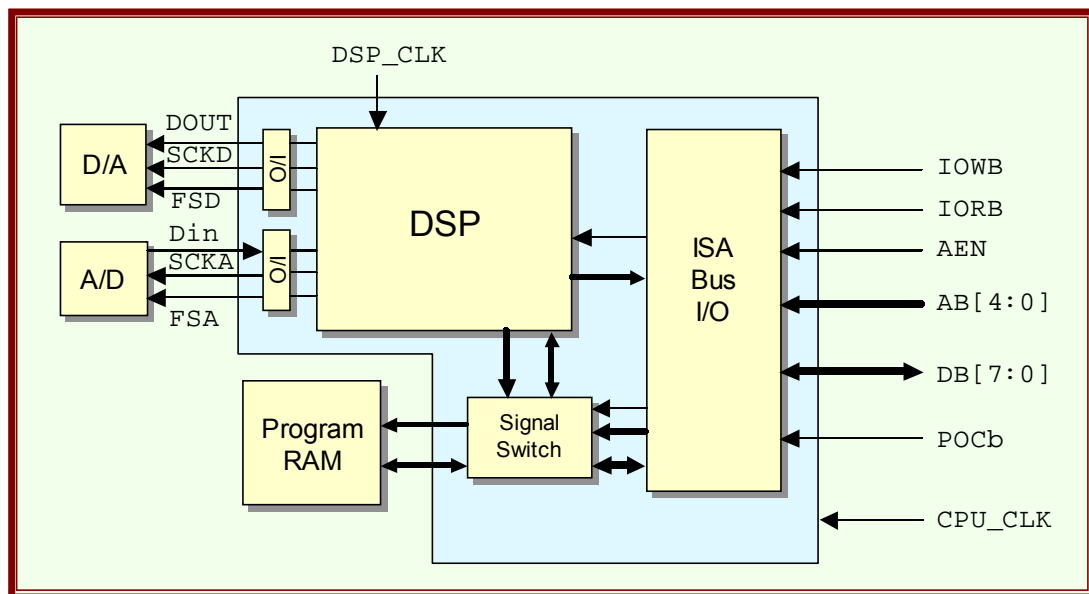


Figure 4-15 A sample circuit for the creation of a test specification.

Figure 4-16 is the test specification for testing the circuit that was created in this case. The first test item is described as “write program data from the ISA bus”. This can be interpreted as the desire to check the ISA bus interface by writing data from the CPU side, and as wanting to check the signal switchover in the lines for writing data to the PRAM.

No	Test items
1	Write program data from ISA bus to Pram (Normal behavior)
2	Write program data from ISA bus to Pram (Abnormal input from AEN,IOWB)
3	Write program data from ISA bus to Pram (Other than specified address)
4	Read serial data(12 data) from A/D (Normal behavior)
5	Read serial data(12 data) from A/D (Abnormal behavior)
	.
	.
	.

Figure 4-16 A List of test items

With the details of each of these items, it cannot be seen exactly what type of test the designer wishes to have performed. However, the details written in the test specification do not specify the entire test flow or all of the items to be checked. It is not uncommon to have more than 1000 test items in a large-scale design. If the test items are written in too much detail, then it will be impossible to organize all of the details of the test items.

* Test Detail Speciation

The detailed flow of testing is written in what is called the “Detailed Specifications”. Figure 4-17 shows detailed test specifications. In order to download data from the PC through the ISA bus, first the data “42H” is written to the control register address 4. Then the address value and data value are written. The lower byte is written to address 1 in the control register while the upper byte is written to address 2. Next, the address data is written to address 3 and address 4. Finally, the value in the control register address 4 is overwritten to write one byte worth of data to the PRAM. The data download is completed when this operation is executed for every line of the program data A.

The actual test flow is not as short as shown in Figure 4-17. When the test flow is described in detail it may be 100 or even 1000 lines long. When the design is large, writing out the test flow in this way is extremely time consuming. The test detail specifications are redundant work because they are the same test items rewritten in a different notation. Additionally, the larger the specification, the more difficult it is to prevent errors when rewriting the content of the test bench specifications into details.

No.1	Test item	Write SERI.DATA from the ISA bus to Pram (Normal mode)
		<ol style="list-style-type: none"> 1. Write 'h42 to ctr_reg4 from ISA bus (set to data load mode) 2. Write all of program data A to PRAM (repetitive behavior address using while statement will increment from FFFE one address at a time) - Use DataLoadFromISA task Write program data [7:0] from ISA bus to ctr_reg0 Write program data [15:8] from ISA bus to ctr_reg1 Write program address [7:0] from ISA bus to ctr_reg2 Write program address [15:8] from ISA bus to ctr_reg3 Write 'h53 from ISA bus to ctr_reg4 Write 'h52 from ISA bus to ctr_reg4 3. Write 'h04 from ISA bus to ctr_reg4 (change to DSP start mode)

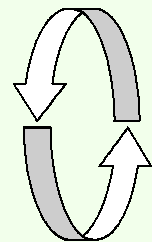


Figure 4-17 Example of the detailed test specification

* Systematic Test Bench Description

Even if no test detail specification has been written, as long as the test bench description is systematic, it is more effective to check the details of the tests using this description. Example 4-26 is the task description for Test Item 1. In this test description, the program data that is stored in the file is saved to an array using the `$readmemh` command. Next the data “42H” is written to the control register address 4, thereby setting up the data load mode. Then a **while** statement is used to increment the address, one address at a time from **FFFE**, during which the **DataLoadFromISA** task is called.

In the **DataLoadFromISA** task (Example 4-27), the task known as “writeISA” is called, and thus the same test procedure as in the test detail specification has been written. Finally, the **WriteISA** task is notated as shown in Example 4-28, causing an external input to the **ISA** bus.

```

task AllDataLoadFromISA;                                //Task to load all program data from ISA bus
input[15*8:1] file_name;                                //Enter program data file name
reg [15:0] prog_adr;                                    //Address to send to DataLoadFromISA(used only in this task)
reg [15:0] prog_array[0:65535];                         //Storage array for program data
begin $write("loading program %s",file_name);            //Read all of program data to the array
$readmemh(file_name, prog_array);
write_ISA(3'h4, 8'h42);                                //Data load mode to write 42 to the control register address 4
prog_adr = 16'hFFFE;                                    //
while(prog_array[prog_adr] != 16'hxxxx)                //Program data end with xxxx for identification
begin
  $display("p_adr=%d ,p_da=%h",prog_adr,prog_array[prog_adr]);
  DataLoadFromISA( prog_adr, prog_array[prog_adr]);
  prog_adr = prog_adr + 1;
end
write_ISA(3'h4, 8'h04);
end
endtask

```

Example 4-26 The task for the test item 1 in Figure 4-16

```

task DataLoadFromISA;                                // Task to load 1 byte data from ISA bus

input [15:0] write_prog_adr;
input [15:0] write_prog_data;

begin
  write_ISA(3'h0,write_prog_data[7:0]);                // Write Seq: IOWB-0
  write_ISA(3'h1,write_prog_data[15:8]);                // ADDRESS 0 : Putting Prog Data lower-bit
  write_ISA(3'h2,write_prog_adr[7:0]);                  // ADDRESS 1 : Putting Prog Data higher-bit
  write_ISA(3'h3,write_prog_adr[15:8]);                // ADDRESS 2 : Putting Prog Address lower-bit
  write_ISA(3'h4,write_prog_adr[15:8]);                // ADDRESS 3 : Putting Prog Address Higher-bit
  // ADDRESS 4 : control register
  // +----- DSP RESET 1-on 0-off
  // | +----- RDY 1-notrun 0-running
  // | | +----- CLOCK 1-ckin 0-1/8ckin
  // | | | +----- STATUS7 1-set 0-off
  // | | | | +--- PD 1-PRAM 0-Emu data
  // | | | | | +--- PA 1-Emudata 0-PRAM
  // | | | | | | +--- INT2B 1-on 0-off
  // 0101_0011 resetINT2B-on with data load mode
  // 0101_0010 reset with data load mode

  write_ISA(3'h4, 8'h53);
  write_ISA(3'h4, 8'h52);
end
endtask

```

Example 4-27 The tasks used in the test description of Example 4-26

```

task write_ISA;
input [3:0] write_ISA_adr;
input [7:0] write_ISA_data;
begin
  @(negedge CLK) AB = write_ISA_adr;
  DB = write_ISA_data;
  @(negedge CLK) IOWB = 1'b0;AEN=1'b0;
  @(negedge CLK) IOWB = 1'b1;
  @(negedge CLK) AEN = 1'b1;
  DB = 8'b11111111;
end
endtask

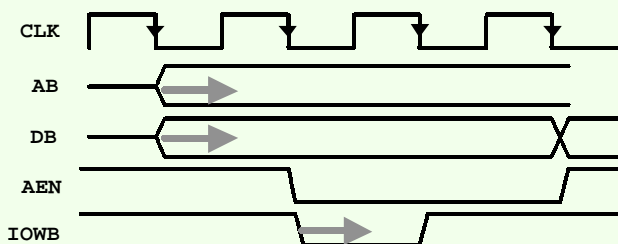
```

Example code with added delay:

```

task write_ISA;
input [3:0] write_ISA_adr;
input [7:0] write_ISA_data;
begin
  @(negedge CLK) fork
    #(Tab) AB = write_ISA_adr;
    #(Tdb) DB = write_ISA_data;
  join
  @(negedge CLK) AEN = 1'b0; #(Twb) IOWB = 1'b0;
  @(negedge CLK) #(Twb) IOWB = 1'b1;
  @(negedge CLK) AEN = 1'b1;
  DB = 8'b11111111;
end
endtask

```



Example 4-28 The lowest-level task used in Example 4-27

*** Establishing the Relationship Between the Test Specifications and the Test Bench Description**

The test specification introduced by Figure 4-16 is used for performing a design review on the items covered by the tests and on the test results. If the participants in the design review are familiar with the existence of the tasks such as `DataLoadFromISA` or `WriteISA`, which were writing for testing the circuits, then they will know exactly what type of tests were performed for the material covered by Test Item 1, which at first appeared to be rather vague.

The test specification cannot describe comprehensively all the material covered by the tests. Although there will inevitably be some remaining ambiguity, this ambiguity is eliminated by the use of tasks, etc., in the test description and understanding that the tests are performed in a hierarchical structure. Care should be taken that the test specification be more than a mere description, but rather that it clearly defines its relationship with the test bench description.

*** Write the Test Specification Before Starting Tests**

The test specification should be written prior to the commencement of testing (i.e., before starting logic circuit verification).^[3] The creation of the test bench description and the logic circuit verification both must be done according to the test specification. Of course the test specification can be modified or amended during the test process. Although it is not possible to create the perfect test specification before starting the logic circuit testing, care should be used to ensure that the test specification is quite complete before starting the logic circuit testing.

The test specification is used to clearly define the test operations for the logic circuits. The value of the test specification is cut in half if it is written after the test operations have been completed.

4.3.3. Create a simulation checklist

[1] Clarify how test items and the test bench descriptions are related	recommend 2
[2] Clarify what have to be confirmed in the test items	recommend 3
[3] Note the circuit revisions	recommend 2
[4] List all possible items even if it seems impossible to complete testing for all of them during design phase	reference
[5] Consider testing priorities	reference
[6] Consider bug curve graphs	reference

Explanation

* Use the Test Specification as a Checklist

The test specification is convenient when used as a checklist during the logic circuit test operations, and when used in checking the test results. When there is too many test items, it is easy to lose track of whether or not the testing has been completed. As is shown in Figure 4-18, the state of testing can be shown clearly by checking off the test items as they are completed.

Ideally, there should be a 1-to-1 correspondence between the items in the test specification and the test bench description.^[1] However, when there is a large number of a test items, it would take too much time to write a single set of test bench description for each individual test item and to perform the simulations separately. When this is the case, multiple test items should be tested using a single set of test bench description. When this is done, it is wise to clearly define the relationships between the test items and the test bench description.

The test specification items in Figure 4-16 and Figure 4-18 were written after envisioning the type of test vector that would be input into the circuit. In this case, the test items are written assuming the test bench description, and thus the test bench description will have a 1-to-1 correspondence with the test items, and will be easy to execute.

However, the test items that are written in this way do not clearly express the material that is to be checked, and thus there is the danger that the test items will be unclear. Additionally, when the test items are written after envisioning the type of test vector that will be input, it is difficult to envision the situation when the circuit does not operate correctly, increasing the likelihood that the test items to test for incorrect functioning may be missed.

Primary item	Secondary item	Sub item	Test description	Circuit Rev[3]	Test Priority	In charge	Status	Approved by.
1. ETHER test	1. Normal packet	1. Random To/From, data length 42-50	eth111	1.2	A	Tony	✓	Jack
		2. Random To/From, data length 1499-1501	eth112	1.2	A	Tony	✓	Jack
		3. Data length, 10 points in 51-1498	eth113	1.3	B	John	✓	Jack
		4. TYPE other than TCP/IP	ofdm84	1.3	D	John	✓	Jack
	2. Frame gap							

Figure 4-18 Using the test specification as a checklist

* Test Items Are to Clearly Define All of the Details to Be Checked in the Test

One way of thinking about test items is that it is best to list all of the items to be checked.^[2] Figure 4-19 is a test specification of the type that lists the items to be checked. Certainly, writing this type of specification will minimize the test items that would have been forgotten, doing so by showing all the items to be checked. However, when the test items are written using this method, there may be too many test items to handle, making it impossible to describe test bench for each item. Systems in excess of 1 million gates may have 10,000 test items or more. Such a number is too much to handle, and the task of even attempting to organize this many items is tremendously time consuming.

Primary item	Secondary item	Sub item	Test description
1. OSD	1. Single Functionality check	1. Display fringe ON characters (all characters)	sampledisp1
		2. Display fringe ON characters (all invisible characters)	sampledisp24
		3. Display inverted charecters (all characters)	sampledisp2
	.	.	.
	.	.	.
	.	.	.
	2. Complex Functionality check	1. Display fringe ON characters, change 7 colors	sampledisp1
		2. Display fringe ON characters, change 7 colors, at TimeFlag overflow	sampledisp28
		3. Display fringe ON characters, change 7 colors, inverted display	sampledisp2

Figure 4-19 Test specifications based on items to be checked

When it comes to writing test items, the designer must pay attention to two things: making sure that they are compatible with test bench description, and making sure that they clearly define the items to be checked. One method of doing this is to add “check items” to the test items. While test items have a 1-to-1 correspondence with test bench description, the check items can be written in parallel to indicate what it is that is checked for each of the individual test items. Figure 4-20 is an example of rewriting the test specification of Figure 4-16 into check items. As few as two or as

many as ten check items are written for a single test item. The use of check items in parallel with test items increases the level of confidence when reviewing the results of the tests.

No	Test items	Check items
1	Write program data from ISA bus to Pram (normal behavior)	Write to ctr_reg1-4 (check all bits)
		Behavior of all bits in PRAMAddress
		Behavior of all bits in PRAMData
		Behavior of DSP stop mode (ctr_reg[6]0->1)
2	Write program data from ISA bus to Pram (abnormal input from AEN , IOWB)	Check that it will not write (IOWB =0) when AEN =1
		Check that it will not write (AEN =0) when IOWB =1
		Behavior when AEN rises before IOWB
		Behavior when IOWB is shorter than CLK
	.	.
	.	.
	.	.

Figure 4-20 The addition of check items to the test specification of Figure 4-16

Another method is to write the list of test items and the list of check items separately. The test items are written first, while keeping the test bench description in mind. Next the list of the check items is made, and for each check item, the test item wherein it is checked is noted. If the check item is not checked in any of the test items, then either an existing test item must be modified or a new test item must be added. After the relationships between the list of test items and the list of check items have been defined, then the test bench is described following the test items, and the simulation is performed. When this is done, a careful evaluation is made whether or not the check items have been tested, and the check items on the list are checked off one at a time. The test is completed after all of the check items have been checked.

In practice, there are not that many examples wherein the list of test items and the list of check items have been written separately. It is also a fact that the more precisely the tests are performed, the longer the testing takes. However, it is not possible to eliminate the bugs unless the test items are quite complete. There is no guarantee that any specifics not tested will actually work correctly. Thus it should be regarded as imperative that the test items be reviewed carefully from a different perspective after they have been written.^[4]

When the test items are generated based on test bench description, you considers from another perspective what it is that must be checked. You must look for gaps in the test specification by considering the check items, going through the output signals and the circuit structures with a fine toothcomb. Conversely, when the test items are written by first listing the check items, you must consider the actual test bench description, and restructure the check items from the perspective of practical operations.

* Prioritizing Test Items

In large-scale logic circuits, the number of test items alone will be prodigious. There are always deadlines when it comes to circuit design, and often there is not enough time to test all of the test items. While, of course, it would be ideal if all test items could be checked, one can expect some degree of validation by simply listing all of the test items alone, even if they are not checked. To put it differently, it is important to consider and list all sorts of check items to such an extent that covering them before the circuit design deadline seems impossible.

When there is too many test items to test them all, the test items should be prioritized as shown in Figure 4-18.^[5] Priority ranking from A to E are assigned, and the test items are performed starting with A. It is not a particular problem if you are only able to make it about halfway through the Ds and Es. In actual large-scale designs, upon finding bugs after LSI production, we often realize that test items for the bug had been prepared but not tested in limited time scale. Nevertheless, the reality is that it is not possible to test all of the items. Thus prioritizing the test items is critical if the design is to be as correct as possible.

* The Bug Curve Graph

It has become difficult to test all imaginable sets of conditions in large-scale logic circuits. Furthermore, even if you were to test all of the sets of conditions that you might be able to think up, this does not mean that all possible conditions have been tested. There will always be bugs due to factors that are difficult to anticipate. Bug curve graphs are used in software development, and this approach has been used increasingly in hardware development as well.^[6] Figure 4-21 shows a bug curve graph. Many bugs are discovered in the initial testing stages, and then, little by little the frequency with which bugs are discovered diminishes. When you divides the number of test items into four parts, the frequency with which bugs are discovered should fall as the testing proceeds in a proper testing process. If the number of bugs discovered in the last quarter of the testing process is extremely low, then the circuit can be considered to be in a stable state, and you can project that there will be few bugs thereafter.

When you draws a graph of the bug curve, however, it rarely actually looks like what is shown in Figure 4-21. Instead, sometimes many bugs are discovered when the testing is nearly completed as shown in Figure 4-22. There are two possible causes for this. The first is that the number of items to be tested is inadequate. There is some degree of variability in the number of bugs discovered. The stable period (a) might be simply a valley and bugs are discovered near the end of the testing period. Therefore it is possible that bugs will continue to be found meaning that it will take considerably more time to find a truly stabilized state. The second possible cause is a problem with the sequencing of the tests. There are cases where those critical tests that are most likely to find bugs are mistaken to be lower priority, causing a surge in the number of bugs discovered near the end of the testing period. In either of these cases it is possible that there will still be many bugs that have not been discovered, and thus it is risky to complete testing.

At present, it is difficult to guarantee that large-scale designs will be absolutely bug-free. There is still the need for innovations in the use of statistical method to reduce the probability that bugs will occur.

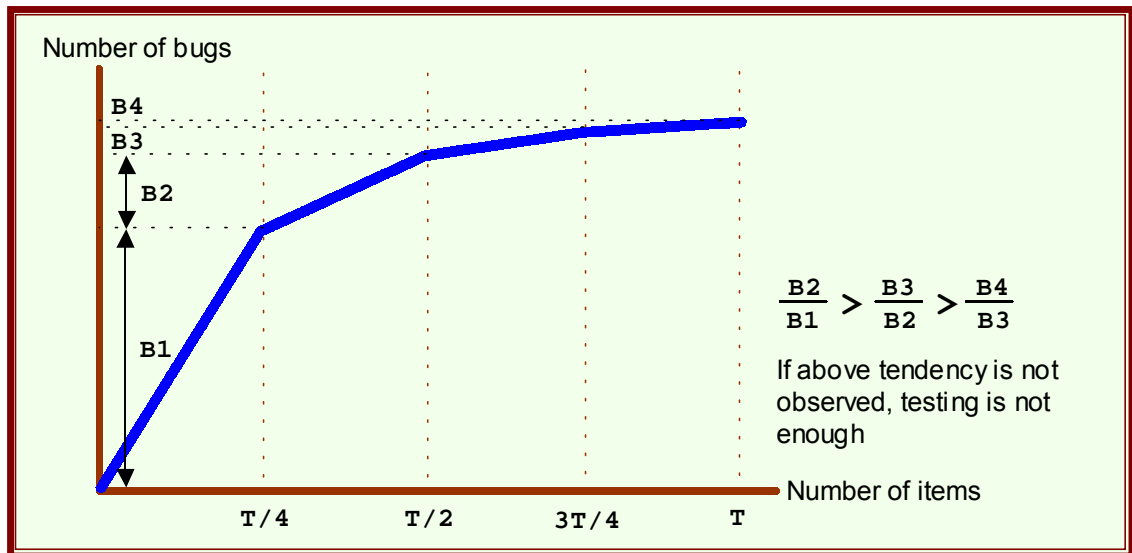


Figure 4-21 The bug curve graph

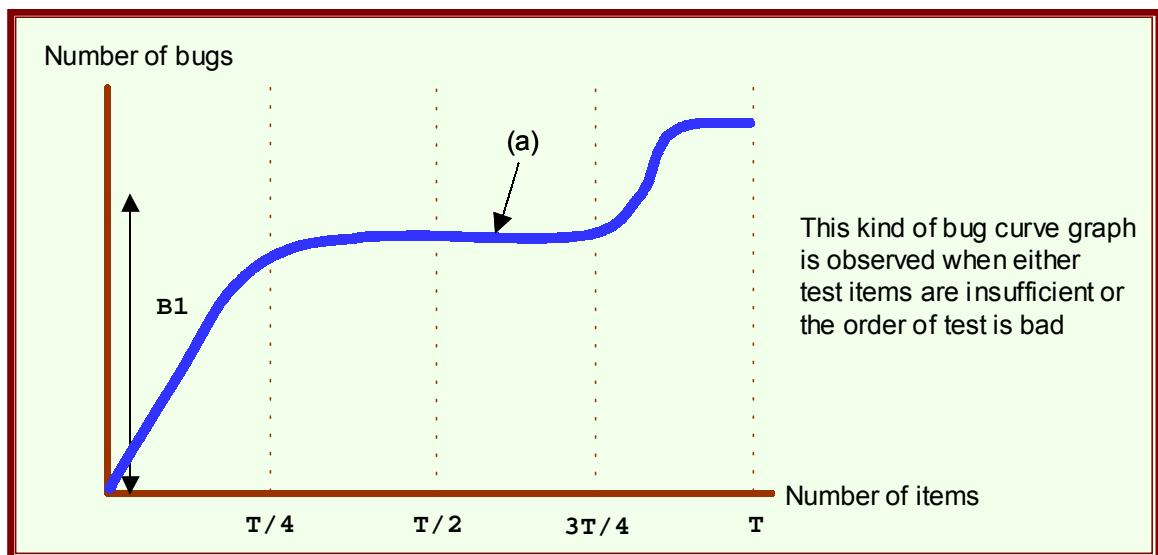


Figure 4-22 A non-ideal bug curve graph

4.3.4. Clarify simulation results

[1] Clarify simulation results not only by waveforms but also by comparing with the previous results

recommend 2

[2] Creating a batch file makes it possible to run multiple jobs

reference

Explanation

An effective method is to output internal nodes as multiple waveforms and to search for errant signals in the initial simulation debugging phase. However, continuing to check items using waveforms is not really an efficient method from the standpoint of clarifying the verification.

Let us assume, for example, that checking the results of one simulation by looking at all waveforms takes 3 hours. Let us then posit that an error was discovered in a function when checking the waveforms. In such a situation, designers have to modify the description and perform the simulation again.

In this case, however, most designers will only check the waveforms relating to the changes that were made. Even if codes that had been correct up to now became incorrect after the description was changed, these codes will be inadvertently overlooked. Conversely, if the waveforms are carefully checked for 3 hours for every simulation, unnecessary waveform checks will be repeated.

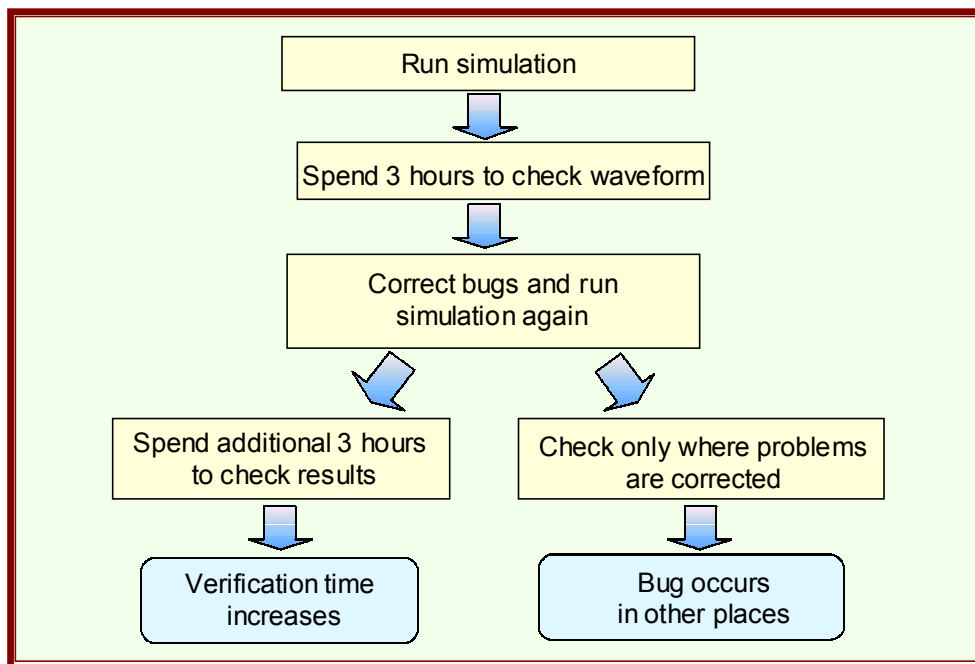


Figure 4-23 Checking waveform alone can not clarify simulation results

The best method for clarifying the simulation results is to generate the expected values using the Verilog-HDL description indicated in “4.3.6. Simulating while executing expected value verification” and then to automatically compare the results.^[1]

However, a considerable amount of time is required to make descriptions to generate expected values in this way. Accordingly, when comparison with expected results is difficult, you can compare it with previous results.

This is a method for saving the output results from the previous simulation as a file, running a simulation, and then comparing the previous output results with the current ones. The previous results may not be corrected when employing this method. However, the difference with the previous results should at least become clear. If there are differences with the previous results, either the current or the previous results are incorrect, so you should then check carefully only the waveforms for the areas where these differences exist.

If no differences appear when the simulation is executed again, the portions previously checked and judged to be correct can be judged to be correct without checking the results by looking at the waveforms. It is possible to say that this method of comparison with the previous results is very compatible with simulation process management that uses checklists. In addition, running simulations as a batch process makes it possible to execute multiple jobs.^[2]

Interactive environments are effective in the early stage of verifications, but it becomes necessary to provide many test vectors when verification has progressed. Batch processing is most suitable for such situations.

In the case of gate level simulations, the expected values have already been created, so making batch files is effective. If the parameters are changed and multiple simulations are executed for the same circuit, then control that uses parameter files as separate files is effective. Example 4-29 is a description that saves the output results to a file at each cycle. A good observation point would probably be immediately before the clock rise while taking gate level simulation into account.

Example Code

```
parameter CYCLE = 200;
parameter STROBE = 190;

integer I,J,mcd1;

initial begin
    mcd1 = $fopen("CD2450a.pat")
    forever begin
        @(posedge CLK) #Out_STB ;
        $fstrobe (mcd1, "%b%b%b%b%b%b",
                    SECLSB,SECMSB,MINLSB,MINMSB,HOURLSB,HOURMSB) ;
    end
end
```

Example 4-29 Output signals to a file at each cycle

4.3.5. Compare the expected value files with the simulation results

- | | |
|---|-------------|
| [1] Compare the previous results (expected values) with the simulation results
- Compare files using the UNIX <i>diff</i> command
- However, debugging to discover which output caused a mismatch is not easy | recommend 1 |
| [2] Compare the gate level results with the RTL simulation results | reference |
| [3] Previous results are not always correct, but the number of correct portions will increase | reference |

Explanation

This is a method where comparisons are made to the material described in “4.3.4. Use batch file for simulation”. Here the results output from \$strobe or \$fstrobe task can be compared using the UNIX *diff* command.^[1] If the simulation is run using this method, it will be possible to build upon the areas that are correct, even if not all of the previous results were right. This makes it possible to define the tests more clearly.

The results that are output by the \$strobe or \$fstrobe task can be used in comparisons with the results from the gate-level simulation. The gate-level simulation is explained in “4.4. Gate level simulation”.

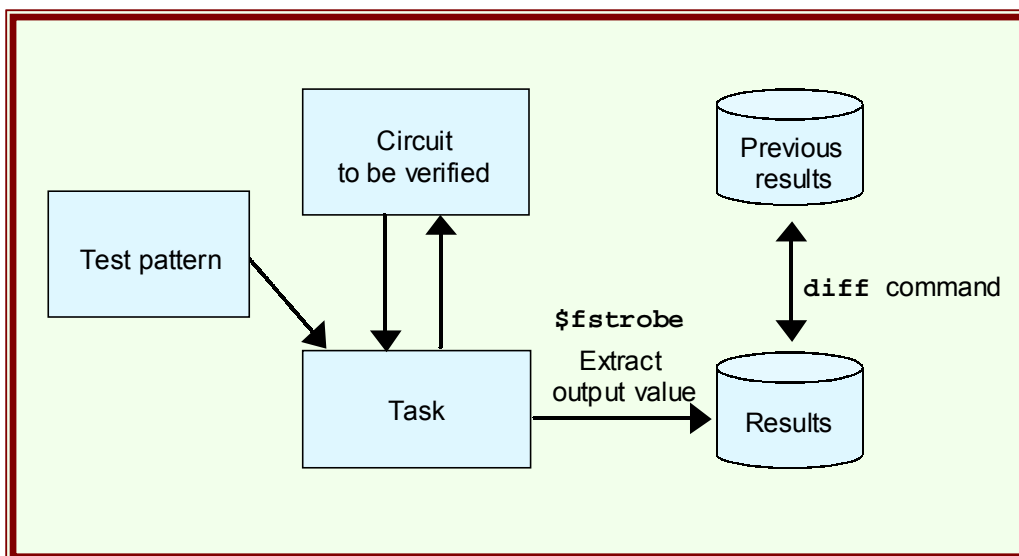


Figure 4-24 Comparison flow of simulation results using the previous results

One way to output all of the result with each simulation run is to compare them to the previous results, which is relatively easy to do. However, when the design is large and complex there may be a very large number of differences. In such a case, it often would take too much time to compare the differences on a line-by-line basis, and the task would be arduous. Under such circumstances, handshaking should be performed in Verilog (explained in “4.1.10. Use handshakes when the process cycle count is not clear”) in order to output the significant content. In some cases, the output results are compared using analysis tools written in C, Perl, or some other programming language.

4.3.6. Simulating while comparing with the expected values

- | | |
|--|-------------------------------------|
| <p>[1] Run the simulation while comparing with the expected values</p> <ul style="list-style-type: none"> - Compare with the expected values, and stop the simulation if mismatches occur <p>[2] Some techniques are required for comparing with don't-care</p> | <p>reference</p> <p>recommend 2</p> |
|--|-------------------------------------|

Explanation

As explained in “4.3.4. Clearly Defining Simulation Results”, the ideal is to compare the results of the simulation to the expected values in order to clarify the results of the simulation.^[1] One way to compare to the expected values is to directly compare the output results with the expected values. Another way is to generate the expected values in Verilog code and to compare with the results output by the simulation.

Example 4-30 is an example of expected results comparison description in Verilog. The circuit being analyzed in this example is the one shown in Figure 4-25. This circuit outputs by a RS-232 a signal that is input in parallel (8 bits). In the output signal TXD of the RS-232, the output value ‘0’ means the start signal. After that, 8 bits of data are output in series, starting with the least significant bit.

When a parity is selected, the parity bit is output next, and then, if the output is completed, TXD goes to ‘1’. This RS-232C output circuit has a “full” output. If the circuit is in a full state, the “full” bit outputs a ‘1’ and no more data is accepted.

(a) in Example 4-30 stores `send_din_bank` in a storage array in the sequence of transmission as the parallel data is input into `din`. At this time, the parity input value is also stored, in `send_parity_bank`. The task for writing data of (a) is called in the *initial construct* of (b) and the specified data is transmitted the specified number of times.

The value of the output signal TXD is processed by the *always construct*, the value output from the serial output TXD is converted to a parallel format, and is compared sequentially to the value that was stored in `send_parity_bank`, which we saw above (c). When this is done, the expected value comparison results are output in a message. If the parity input is ‘1’, the parity value that was output from the TXD will be calculated and compared with the value of the parity bit.

In the simulation, “(==,!=)” is used when comparing the signal values. In (e), when “!=” were used, a malfunction would occur if the value of TXD is ‘x’ because “false” will be returned.^[2]

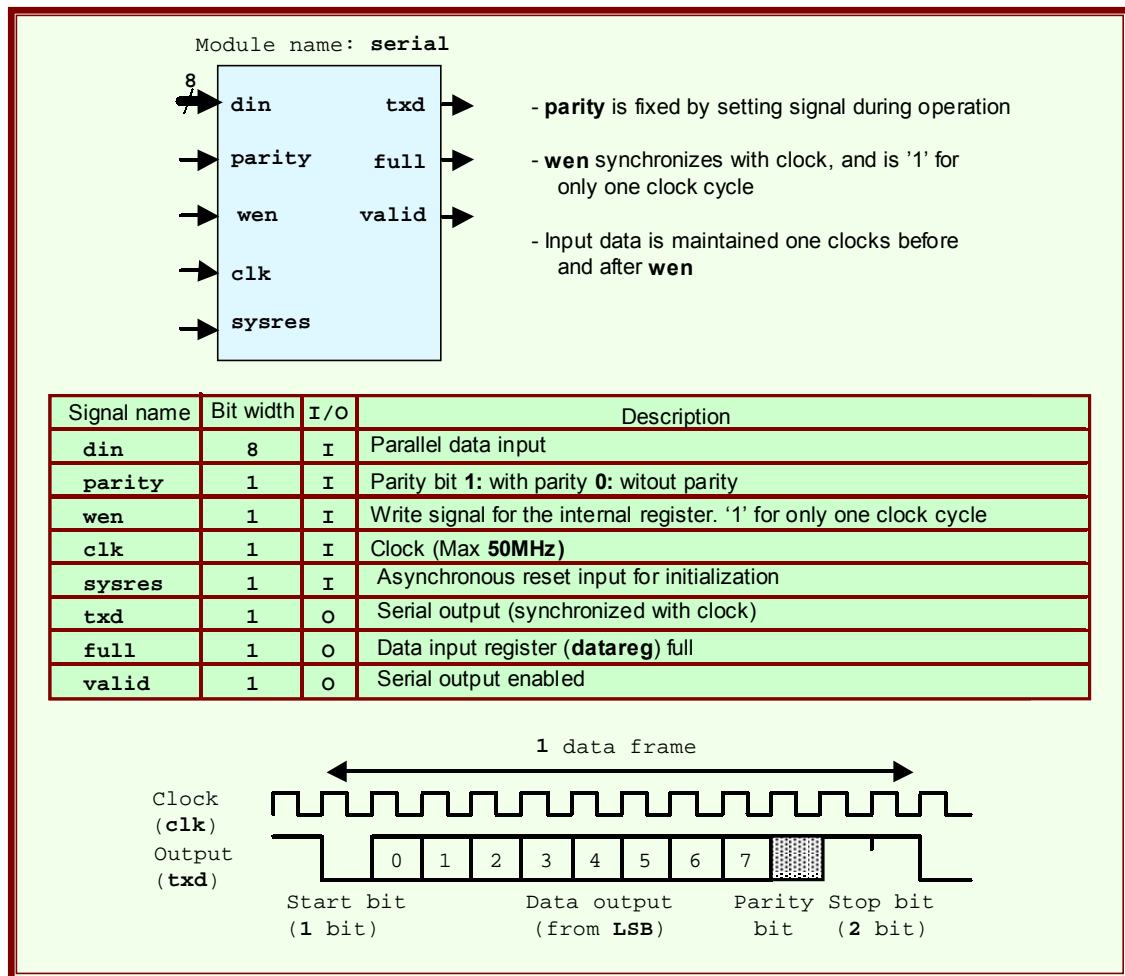


Figure 4-25 RS232C output circuit

```

module serial_tp;
reg      clk, sysres, wen, parity;
reg      [7:0] din;
reg      [7:0] send_din_bank [0:1023];           //Stock of data sent
reg      [0:1023] send_parity_bank;              //Parity value of data sent
integer  send_number;                            //number of data sent
integer  read_number;                            //number of data received
reg      [7:0] read_data;                        //Data received
reg      read_parity;                            //Parity received
wire     txd, full, valid;
integer  i,j;
parameter CYCLE      = 20000;                    //Cycle setting for one clock
parameter HALF_CYCLE = 10000;                    //Cycle setting for half a clock
parameter DELAY      = 2000;                     //Input delay for clock
serial serial( .clk(clk), .sysres(sysres), .din(din), .wen(wen),
               .parity(parity), .txd(txd), .full(full), .valid(valid) );

always begin
    clk = 1;                                     // Clock generation description
    #HALF_CYCLE clk = 0;
    #HALF_CYCLE ;
end

(to be continued)

```

```

task write_reg1;
begin
  @(posedge clk)
  if(full != 1'b1) begin
    #DELAY din = i; wen = 1;
    send_din_bank[send_number] = din;
    send_parity_bank[send_number] = parity;
    send_number = send_number + 1; end
  else
    #DELAY din = i; wen = 1;
  @(posedge clk) #DELAY wen = 0;
end
endtask

initial begin
  sysres = 0; wen = 0; parity = 1; send_number = 0; read_number = 0;
  #DELAY;
  @(posedge clk) #DELAY sysres = 1;
  @(posedge clk) #DELAY sysres = 0;

  $display( "all Data sending with Parity ON mode" ); // Check item 1
  for ( i=0; i<256; i=i+1 )begin
    while ( full ) @(posedge clk);
    write_reg1;
  end

  $display( $stime, "Checking Parity OFF Mode" ); // Check item 2
  parity = 0;
  for ( i=0; i<256; i=i+7 ) begin
    while ( full ) @(posedge clk);
    write_reg1;
  end

  $display( $stime, "Checking port full" ); // Check item
  for ( i=0; i<256; i=i+17 ) begin
    repeat(i/29) @(posedge clk);
    write_reg1;
  end
  end
  $finish;

end
always begin
  while (txd != 1'b0)
  @(posedge clk);
  for (j=0; j<8; j=j+1) begin
    @(posedge clk) read_data[j] = txd;
  end
  if(read_data == send_din_bank[read_number])
    $write("%d, Send=%h, Read=%h, Match!!", read_number, send_din_bank[read_number], read_data);
  else
    $write("%d, Send=%h, Read=%h, Error Unmatched!!", read_number, send_din_bank[read_number], read_data);
  if(send_parity_bank[read_number] == 1'b1) begin
    parity = txd;
    if(read_parity == ^read_data)
      $write("Parity OK\n");
    else
      $write("Parity NG!! read_parity=%d, Exp=%d\n", read_parity, ^read_data);
  end
  else
    $write("No Parity\n");
  @(posedge clk);
  read_number = read_number + 1;
  if(txd != 1'b1)
    $write("Stop bit Error!!\n");
end

```

(a) points to the `if(full != 1'b1)` condition in the `write_reg1` task.

(b) points to the `while (full)` loop in the `initial` block.

(c) points to the `if(read_data == send_din_bank[read_number])` condition in the `always` block.

(d) points to the `if(read_parity == ^read_data)` condition in the `always` block.

(e) points to the `while (txd != 1'b0)` condition in the `always` block.

Example 4-30 Comparison with expected values

4.3.7. Efficient verification using a behavior model

- | | |
|--|-----------|
| [1] Verify the functionality of the entire circuit and blocks using behavior simulations | reference |
| - Appropriate block partitioning is not possible unless the overall circuit is considered | |
| - It is not possible to simulate each block unless the I/O specifications for each block are completed | |
| - More than 2 million gates are required, but not less than 500,000 gates | |
| [2] Signal events only (untime) description | reference |
| [3] Cycle-accurate description | reference |
| [4] System verification using C language | reference |

Explanation

The testing process is extremely labor-intensive for large-scale LSI circuit designs. In the conventional design method, the detailed design has been described in RTL code, and time-consuming simulations have been run. When bugs are discovered at this stage, if those bugs trace back to problems in the specification, fixing the bugs will be time consuming. Recently, behavior models have been used incorporating an approach where problems are tested in the specification before starting the actual design in RTL.^[1]

The benefits of testing using behavior models include:

- the amount of coding is minimized and
- the simulations run faster.

Behavior code can be described in HDL language such as Verilog-HDL or VHDL, or in C language.

The benefits of using the HDL language include:

- Because the same environment can be used for both RTL code and the actual design, it is easy to check whether or not the simulation results for the behavior match the RTL results.
- The hardware model can be expressed easily.

Behavior coding is done in two steps. The first is termed “Untime” where the code has no concept of time.^[2] Programming languages such as C use sequential processes, so cannot express simultaneous execution. For example, if there are individual circuits, even if the contents of those circuits are expressed in sequential processes, the hardware will execute several of the circuits (systems) simultaneously. Because the C language does not support simultaneous (parallel) execution, it is difficult to represent the system as a whole. In behavior design using the C language, these problems are handled in one of two ways. One is to use tools that support simultaneous execution as a specialty language (an extended C language) and the design is done using those language extensions. Another is to use C++ language under certain restrictions to represent simultaneous execution. Using C++ language, it is possible to represent simultaneous processing even without using some specialty language and relying on certain kind of tools. In designs where a variety of parts are connected to the CPU, the individual parts are represented using sequential processing, and the behavior verification is performed using a method where simultaneous execution is supported for the bus only.

Because both sequential processing and simultaneous execution is supported in the HDL language, these processes can be represented freely in the Untime stage. HDL code in the Untime stage does not have a concept of a clock, instead assuming a style where behaviors are represented as signal events.

In the logic circuit systems of today, however, clocks do indeed exist, and design is done based on clock synchronization. Even if some degree of testing and validation is possible in Untime, ultimately testing using clock synchronization must be performed. There are many problems that are only found in clock synchronization, even after through verification in Untime. The clock-synchronous behavior code is called “cycle accurate” code.^[3] The difference between RTL code and cycle-accurate code is that RTL code shows the registers and you could perform logic synthesis, where cycle-accurate code does not need to show the registers. Cycle-accurate description is not as specific as RTL, and because of that, the simulation speed could be faster.

However, the cycle-accurate code is too specific when compared with Untime, and so does not produce the benefits of high-speed testing of behavior code.

Some behavior design are started and verified in Untime and then redefined in cycle-accurate code or directly to RTL, and some are started in cycle-accurate code. In either case, it is possible to do the coding faster than it is to do the coding in RTL, and the simulations run faster as well.

Behavior testing can be expected to improve the efficiency of design in complex, large systems. However, in order to perform testing processes in Untime, the existing design resources written in RTL have to be converted to Untime, an aspect of the move to Untime that is not so simple.

“SystemC”, the Standardization Committee for C++ class library, is working on the standardization of behavior code in the C language.^[4] However, at present the specific C design environment depends on specific tools. When it comes to whether or not a global standard will emerge for the language that is used for behavior coding (Verilog, VHDL, Extended Verilog, a C language with a specialty specification, C++, etc.), the jury is still out.

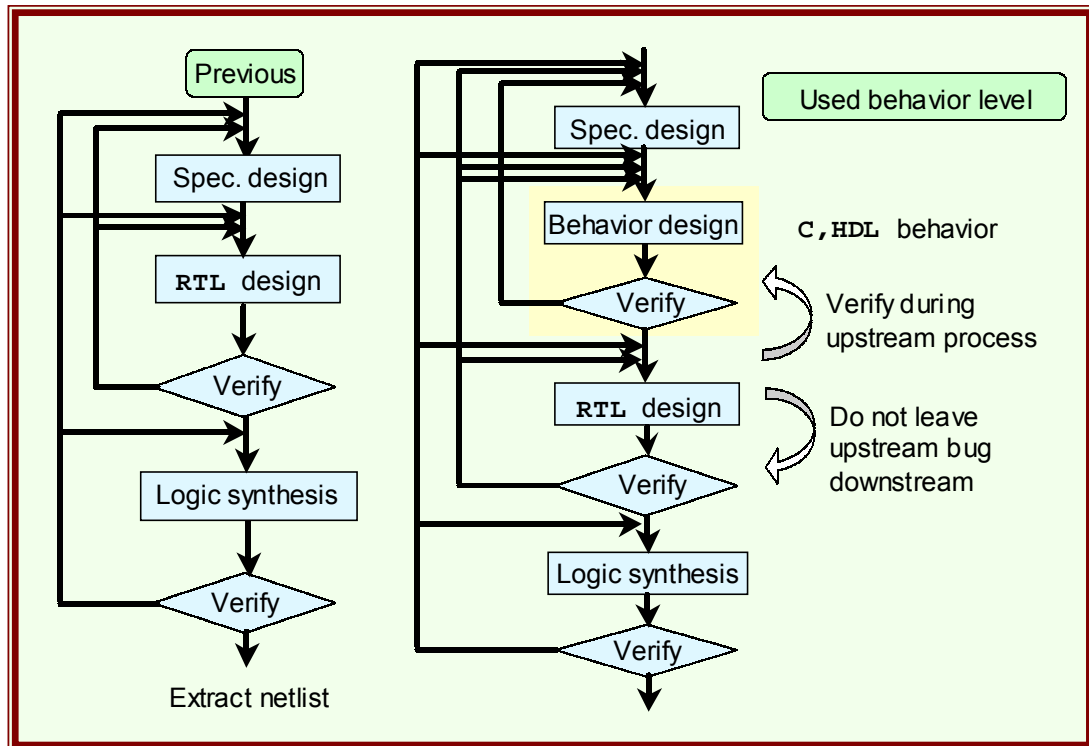


Figure 4-26 Verification flow using a behavior model

4.3.8. Verification methods for large-scale design

[1] Hardware debugging using FPGA	reference
[2] Use of hardware emulators	reference
[3] Use of Co-Sim	reference
[4] Use of automatic function verification tool, model-checking and property-checking	reference

Explanation

As was described in “4.3.2.Create test specifications” and “4.3.3.Create simulation checklist”, testing large-scale circuits is an extremely difficult work. While test specifications are used to define the tests, in practice the number of test items may get out of control, and the HDL simulations would require massive amounts of time.

The most effective testing for logic circuits would be to actually build hardware and to verify it at speeds near to the actual speed. However, if bugs were found after the actual LSI has been built, reworking the design at that point would entail vast expenditures of time and money.

Recently, the use of FPGAs has become common for prototype testing. When an FPGA is used, the tests can be performed in a fraction of the time required for software simulation using, for example, HDL language, and the test accuracy will be improved.

It has recently become possible to use a single FPGA for large circuits up to 200,000 gates. If three or four FPGAs are used, logic circuits in the range of up to about 50,000 gates can be prototype. In prototyping using FPGAs, 50,000 gates is thought to be about the limit. This is because a large number of FPGA has to be connected by the wire on board in order to realize a large scale circuit by FPGA, modification of circuit is unavoidable during logic design and therefore wires between FPGA will have to be changed. For example, if 30 FPGAs were used for prototyping a 3 million-gate logic circuit, it would be extremely difficult to manually correct the interconnections when there is a design change, and instead a new breadboard would have to be constructed. Having to make a whole series of breadboards would slow down the development process.

There are also cases where emulators are used for testing large-scale logic circuits. Emulators are systems containing between 10 and 200 FPGAs, where the interconnections between the FPGAs can be changed dynamically. The use of these emulators makes it possible to make changes using software, instead of requiring changes in hardware such as breadboards even when logic circuit is changed.

The use of emulators is most effective in the development of the large scale LSI circuits in excess of 1 million gates from the view point of verification speed. Rather than using software such as the HDL language to perform excessively time-consuming testing, testing can be performed rapidly, nearly as fast as in the actual circuit. However, emulators are extremely expensive, and require a huge development investment. Additionally, rather than performing all of the processes in software, hardware is actually constructed; thus it takes somewhat more time to actually start up verification environment.

4.3. Verification Process

Using FPGAs or emulators is the good way to improve the speed of testing, however, even when FPGAs or emulators are used, software simulation using HDL or the C language, etc., is still required. If a problem is found in the FPGAs or the emulator, the source of the problem is difficult to identify because it is difficult to monitor the behavior of the internal logic circuits. Detailed debugging requires the use of the software simulators. In addition, there are times where it is difficult to reproduce specific detailed conditions in actual operations on hardware; thus it is difficult to eliminate bugs using hardware testing alone. In large-scale LSI circuit development, it would be good to consider speeding up the software testing (such as when using HDL), when FPGAs or emulators cannot be used. The software testing can be speeded up through the use of the methods described in "4.3.7. Efficient verification using a behavior model" or using hardware/software collaborative simulator, called Co-Sim (Seamless, CVE, Eagle-i etc.). The Co-Sim tools have been shown to be effective in simulations of systems where a variety of circuits are connected to CPUs. The operating speed of the HDL simulator can be improved by a factor between 2 and 10 by simplifying the operations of bus or CPU. The use of the Co-Sim tools cannot produce the 1,000 to 10,000 x increase in operating speed that can be obtained by using hardware. However, the use of Co-Sim tools is the simplest, relatively inexpensive method, and is thus a useful way to improve simulation speed.

In the development of large-scale LSI circuits, there is another test method available to test hardware behavior specification, which is described by RTL, by using exclusive tools in addition to merely improving the efficiency of testing by increasing the speed of the simulations that is to verify by a method, which is not simple simulation like property check. For example, in systems where a variety of circuits are attached to the CPU, circuits that perform DMA through arbitration with several circuits are both prone to have bugs and are difficult to test. Because when all of the circuits are connected, it is difficult to reproduce all of the possible arbitration situations to perform simulations on all of them. In such a case, there is a method to statistically analyze the arbitration behavior by generating random patterns from output of each circuit and the arbitration behavior can be analyzed statistically; or conversely, the static analysis can be performed on a arbitration status.

As explained above, specific simulation methods or static analysis tools are also used in large-scale LSI development.

4.3.9. Separate the function verification patterns and the fault detection patterns

[1] Separate the function verification patterns and the fault detection patterns

recommend 1

Explanation

Since circuits have become larger, the number of cycles in the test vectors used in function checks will be over several million lines. Such test vectors are too large to be used as failure detection patterns. Even if such patterns were used, it would be difficult to expect high fault coverage since these patterns are intended for function verification.

In circuit designs that presuppose testability, no more test vectors are created than are needed to check operation for function checking. ATPG should be used for fault detection, and test patterns with high fault coverage are automatically created. Even if ATPG is not used for circuits on the order of 100,000 gates, patterns for function verification and patterns for fault detection should be viewed differently.

Even if you decide to use ATPG tools, you will need to create hundreds of thousands of test vector lines. For such test vectors, a part of those created for RTL functional verifications cannot be used. They should be created from scratch with due consideration given to the items required for failure detection.

The circuit structure required for ATPG was discussed in “3.3.Design For Testability (DFT)” but some structures such as the gated clock may not be detected for clock line behavior. Moreover, if you are using a bi-directional signal inside the circuit, ATPG might just fix the direction to the process. Some special patterns have to be created for these nets.

4.3.10. Verification method using random function

[1] Use a random function for the pattern to verify functions

reference

Explanation

In the verifications using specific pattern, such as "111111" where 1 continues, "000000" wherein 0 is reiterated, and "10101010", wherein 1 and 0 alternate, as input, function detection for the circuit is not always sufficient. As the circuit function becomes more complicated, creating a pattern for function detection becomes more difficult. In this case, a random function could be used for the input pattern.^[1]

A random pattern for ASIC detection is required to ensure that all the values appear and that they can be repeated. A random value that cannot be repeated is obstructed when the expected values are checked.

Random data is generated by a pseudorandom code generation circuit (M sequence), which uses the shift register (LFSR: Linear Feedback Shift Register). This circuit consists of a shift register and EX-OR gate, which feedbacks the value in the middle.

From this circuit, data that changes on a $2^{10}-1=1023$ bit cycle when the number of shift registers level is 10 can be acquired. This data includes all the patterns (1023 patterns) except for those cases when all the values of shift register become 0.

When using this random data as a circuit input pattern, input data of all the patterns can be generated. Therefore, a verification pattern is can be generated easily.

The structure of the pseudorandom code generation circuit changes the feedback structure by the order of the primitive polynomial (= the number of the shift register stage), as shown in the Figure 4-27.

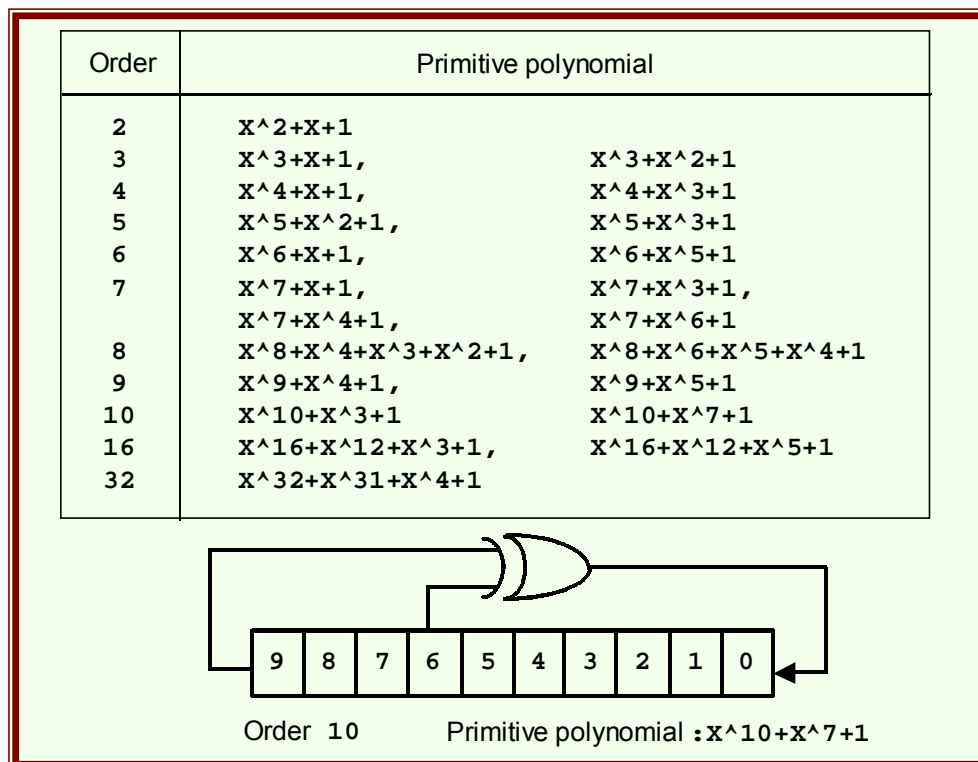
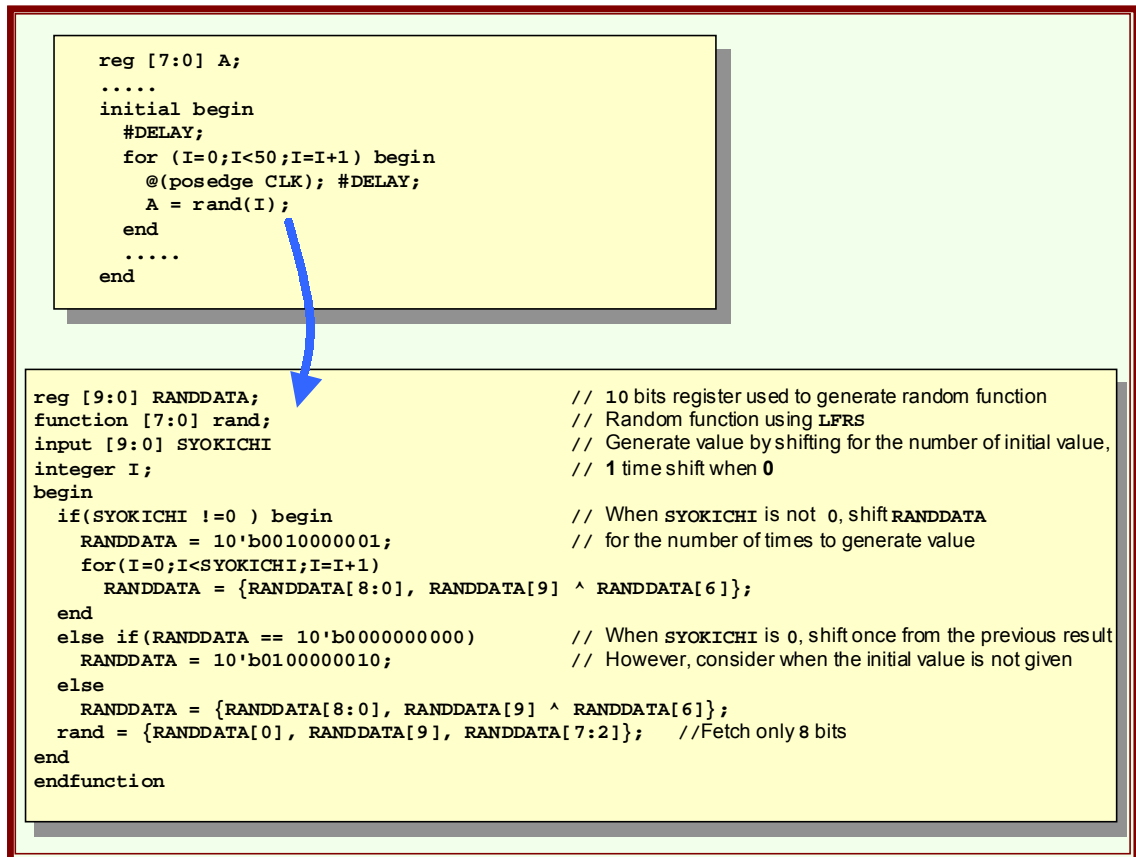


Figure 4-27 Pseudorandom code generation circuit

Example 4-31 is the description for the random function and the test bench using the pseudorandom code generation circuit. By using the random function, all the patterns of the input data are generated. To improve the randomness further, random data generated with the bits of which number is more than the necessary number of bits is used. For instance, if you are using 8 bits from the data operated by 10 or 11 bits shift register when 8 bits data is necessary as a pattern, the continued data pattern is also included such that the randomness of the data is improved.



Example 4-31 Generation of a pattern using a random function

Verilog-HDL has a random function task called \$random. With this function, the results vary depending on the simulator, and unlike LFSR not every value can be generated. As a result, it is better to use the description introduced here.

Random function is explained in RMM:7.1.2.

4.3.11. Efficient debugging by embedding descriptions using *'ifdef* (Verilog only)

- | | |
|--|-----------|
| [1] Switch the active location of the description using <i>'ifdef</i> (Verilog only) | reference |
| [2] Switch the description used by the test bench (Verilog only) | reference |
| [3] The debug descriptions in the RTL are not synthesizable, so use "translate_off" | reference |

Example Code

```
// Test bench description
`ifdef `FAST_CLK
always begin
    CLK=0;
    #50 CLK=1;
    #50;
end
`else
always begin
    CLK=0;
    #250 CLK=1;
    #250;
end
`elseif

// RTL description debug routine
// synopsys translate_off

`ifdef `DEBUG_A
    initial
        $monitor($stime,,, "ADDR= %h, DATA= %h", ADDR, DATA);
`endif

// synopsys translate_on
```

Example 4-32 Switching the description by using *'ifdef*

Explanation

If a text macro name is specified after *'ifdef* and the text macro is specified by a *'define* statement, the description under *'ifdef* becomes active. However, the description under *'else* becomes active if the text macro is not defined.

If *'ifdef* is used, it is possible to switch part of the test bench description depending on the purpose, to define a debugging routine in the RTL description, and then to use a text macro to control the routine.^{[1] [2]}

It is recommended not to put unsynthesizable descriptions such as debug routines in your RTL code. However, when you really need to use them, either use *'ifdef* or *//synopsys translate_off* to exclude them from the synthesis process.^[3] Note that this specification is valid in the Design Compiler only.

4.4. Gate level simulation

4.4.1. Gate level simulation problems

- | | |
|---|---|
| <p>[1] Both RTL verification and gate level verification are necessary
<Reason></p> <ul style="list-style-type: none"> - Timing verification is required (Static timing analysis is currently the standard method) - The mismatch is caused by an 'x' problem - Malfunction of an asynchronous block or the problem of racing - Mismatches may occur due to sensitivity list problems (the combinatorial circuit by an <i>always construct</i>) | <div style="border: 1px solid red; padding: 2px; text-align: center;">mandatory</div> |
| <p>[2] Gate level verification for a large design can only execute a part of pattern in the RTL verification</p> | <div style="border: 1px solid blue; border-radius: 15px; padding: 2px; text-align: center;">reference</div> |
| <p>[3] Use unit delay for gate level simulation of large-scale design</p> | <div style="border: 1px solid blue; border-radius: 15px; padding: 2px; text-align: center;">reference</div> |
| <p>[4] Use a formal verification (equivalence checking) tool</p> | <div style="border: 1px solid blue; border-radius: 15px; padding: 2px; text-align: center;">reference</div> |

Explanation

Timing analysis including critical path analysis and setup/hold timing check is possible for gate level circuits now that static timing analysis tools have come into wide use. See “4.5.Static timing analysis” for more details on static timing. Static timing analysis is more efficient for synchronous circuits than performing all the test benches with time-consuming gate-level simulation. See “4.5. Static timing analysis” for more information on static timing analysis.

However, in addition to analyzing timing problems, it is necessary to perform simulations on gate-level circuits after synthesis.^[1] The reason for this is that even if it is verified in RTL for the correct operation there is no guarantee that the gate-level circuit will work exactly the same way as in RTL. There are cases when the generated gate-level circuit will not match the expected value. Some of the causes for this kind of mismatch will be discussed below.

This problem can be solved by comparing the gate-level simulation results against the expected values obtained with RTL simulation. However, as chip sizes become larger, the time required for simulation has significantly increased, such that the gate-level simulation is becoming more difficult.^[2] In order to verify that the RTL and gate-level circuit are logically the same, you could now use formal verification (equivalence checking) tools.^[4] Moreover, static timing analysis tools can be used for the timing analysis of synchronous circuits. But it should be noted that the use of formal verification and static timing analysis does not give you precisely the same kind of verification that you could perform with gate-level simulation.

* 'x' propagation

RTL and gate-level simulation results may not match because the treatment of an unknown 'x' is different. In addition, an unknown 'x' value could remain when initialization of synchronous reset of FF is not correctly performed in gate-level simulation. This problem can be detected with gate-level simulation, but not with formal verification tools. Thus, it will be necessary to take some measures by using such things as a RTL syntax checking tool. (See “4.4.2.Mismatches in X propagation may occur between the RTL and gate levels” for details.)

4.4. Gate level simulation

* Timing problem

Timing failure may occur because of a clock timing problem with gated clocks, or when static timing analysis is insufficient. In addition, when the initial value assignment is done in a RTL description, timing problems could be overlooked in simulation. (See “4.4.2.Mismatches in X propagation may occur between the RTL and gate levels” for details).

* Sensitivity list problem

This is a problem explained in “2.2.2.Define all input signals in the *always* construct in the sensitivity list” that can lead to a mismatch between RTL and gate-level simulation results. Moreover, latches may be formed when the condition expression is not well defined in an if statement or a case statement. (See “4.4.2.Mismatches in X propagation may occur between the RTL and gate levels” for details)

* Library problem

There may be errors in the synthesis libraries or simulation libraries, causing the results to be different. Although this should not occur, it does happen surprisingly often. There are in particular many cases in which errors occur in the RAM model. There are some ways to solve this problem with formal verification tools.^[4]

* Directive problem peculiar to synthesis

There is a possibility of this problem occurring when you use directives like “*async_reset*”, “*parallel_case*”, “*full_case*”. Avoid using these directives wherever possible.

* Synthesis tool bug problem

A synthesis bug may cause different results. Almost all of these problems can be eliminated by using a reliable synthesis tool. Nonetheless, these problems can occur when using special commands, etc.

There are some ways to solve this problem with formal verification tools.^[4]

Many of these problems can be reduced by paying attention to the description style or reports when logic synthesis is executed

With large designs, not all of the test benches used during RTL verification can be used due to the limited simulation time. Adeptly uncovering potential problem spots and generating short patterns is the key. They cannot all be checked at the current time, so you should design so as to eliminate every mismatch between RTL and the gate-level.

To reduce problems of RTL and gate-level mismatch, adhere to the following.

“1.3.1. Use asynchronous reset for initial reset”

“1.4.3. Gated clocks should be used with special care”

“2.2.2. Define all input signals in the *always construct* in the sensitivity list”

“2.2.3. Default initial value description of the *always construct*”

“2.3.1. Unify the description style of the F/F inference”

- “2.8.5. Description relying on a `parallel_case` is prohibited”
- “4.1.3. Take note of input signal timing”
- “4.1.4. Avoid assigning from multiple *initial constructs*”
- “4.1.8. Description must not depend on each simulator”

Gate level simulation is performed usually by the input of delay information data called SDF by layout tool or timing analysis tool. However, it takes enormous amount of time if delay value is taken into consideration for gate level simulation.

What really has to be confirmed by gate level simulation is the conformity with RTL simulation. To verify the conformity, reflecting accurate delay value is not necessary. There is a method to simulate by making logic gate, such as AND and OR, and sequential cell delay value of FF and latch, fixed value (unit value) and by concerning only logic structure.^[3] The simulation by unit delay can operate 3 to 5 times faster than the delay simulation using SDF.

When simulation by unit delay, library for unit delay is sometimes provided. Safety can be improved by performing many high-speed gate level simulations using these libraries.

4.4.2. Inconsistencies can occur between RTL and at gate level with the propagation of X

[1]	Handling of unknown values 'x' differs between RTL and the gate level	reference
[2]	Be sure there is no 'x' status FF after releasing the initial reset	recommend 1
[3]	Circuits in which the values are not determined from the synchronous reset description may be synthesized	reference
[4]	'x' will not propagate when an if statement is used	reference
[5]	Propagation of 'x' and 'z' are different in case/casex/casex statements	reference

Explanation

Because the handling of unknown value 'x' is different between RTL and the gate level, it is possible that the two will not match. In the code shown in Example 4-33, the *if statement* on the left-hand side, might be evaluated as being false when A is 'x' causing the logical level 1'b1 to be output to B. Because in RTL simulation the value that was 'x' will be a fixed value in Gate level, the simulation results will not match. If, as shown in the code at the upper left, A is coded as the 1'b1 condition and B is coded as the 1'b0 condition, then the *else item* would be executed only when the A value is 'x' or 'z'. Although in this code the 'x' value will be propagated, the code at the upper left must not be used.

<pre> if (A==1'b1) B = 1'b0; else B = 1'b1; </pre>	<pre> if (A==1'b1) B = 1'b0; else if (A==1'b0) B = 1'b1; else B = 1'bx; </pre>
<pre> case (A) 1'b1: B = 1'b0; 1'b0: B = 1'b1; endcase </pre>	<pre> case (A) 1'b1: B = 1'b0; 1'b0: B = 1'b1; default: B = 1'bx; endcase </pre>

Example 4-33 X value propagation in a simulation

The *if statement* at the top is a simple code where the *if statement* condition is based on A alone. In actual RTL code, however, it is not as easy as this type of *if statement*. If, by mistake, you do not include all possible cases in the code, then the 'x' will be generated by the *else item*. While this 'x' indicates a don't-care 'x', when it is then used subsequently in an *if statement*, then the value will be gone.^[4] This assignment of 'x' should be considered "risky coding", which could cause the RTL and the results of gate-level simulation to contradict each other.

Note, however, that, when it comes to the *case statement*, the number of gates will increase if there is no assignment for 'x' in the *default clause*. Thus the don't-care 'x' is used carefully only in the *default clause* for the *case statement*.^[5]

See “2.5.Tri-state buffers”, “2.8.3.Use *default clauses*” and “2.10.1.Order of operators and assignment of 'x'” for further Explanation of this point.

The solution to this 'x' problem is to not produce an unknown or *don't-care* 'x' in the RTL simulation. Be sure that there are no FFs that are not initialized when a synchronous reset is used for the initial reset. 'x' will be output from any FFs that are not initialized. Starting an RTL simulation while the output of an FF is 'x' may result in a problem with discrepancies in the results.

When FFs with synchronous resets are used, the logic synthesis tool may produce circuits wherein the FFs are not initialized in the gate-level simulation.^[3] In the gate-level simulation, this type of circuit does not initialize the FF even if the reset is active, and thus 'x' will remain.

For more information about this problem, see “1.3.1.Use asynchronous reset for initial reset” and “2.2.3.Initial value description in *always constructs*”. Also, see “2.5.2.Consider high-impedance propagation in tri-state bus” for 'x' propagation in tristate buses.

don't-care values are handled differently in the *case*, *casex*, and *casez statements*.^[5] In a *case statement*, the line is executed when there is a perfect match between the input value and the value of the selection. However, in the *casex statement*, 'x' and 'z' are seen as don't-care, so the 'x' value cannot be sent to the next stage as one might expect, as shown in the example. *casez statement* recognize 'z' only as don't care. If the input is 'x', default clause will be executed, however, 'z' will be handled as don't-care.

When 4'bxxxx, 4'bzzzz is entered in DIN ...

```
case( DIN ) // Complete match comparison
  4'b0001: Y=2'b00;
  4'b0010: Y=2'b01;
  4'b0100: Y=2'b10;
  4'b1000: Y=2'b11;
  default: Y=2'bxx; // Branch off to this line
endcase

casex( DIN ) // Match comparison with x and z as don't care
  4'b0001: Y=2'b00; // Branch off to this line
  4'b001x: Y=2'b01;
  4'b01xx: Y=2'b10;
  4'b1xxx: Y=2'b11;
  default: Y=2'bxx;
endcase

casez( DIN ) // Match comparison with z only as don't care
  4'b0001: Y=2'b00; //Branch off to this line if 4'bzzzz
  4'b001?: Y=2'b01;
  4'b01??: Y=2'b10;
  4'b1????: Y=2'b11;
  default: Y=2'bxx; // Branch off to this line if 4'bxxxx
endcase
```

Example 4-34 *don't-care* handling in *case* and *casex*

The result of casex will be same whether 'x', '?' or 'z' is used for clause. With casez, 'z' and '?' are don't care but 'x' become to match if input is 'x'. 'x' should not be used for clause in casez. 'z', '?' and 'x' can be used also for case statement, however, should not be used. This problem is explained in "2.5.Tri-state buffers" and "2.8.3.Use default clauses". It is more important to prevent the generation of the 'x' value in RTL simulation then it is to completely perform the propagation of 'x' in RTL coding.

4.4.3. Beware of malfunctions caused by the timing

[1] Eliminate racing problems	recommend 1
[2] Verify the asynchronous part with a gate-level simulation	recommend 1
[3] Use of static timing analysis	recommend 1

Explanation

When racing occurs in simulation, the result may vary depending on the simulation tools. Racing can be caused by a 0 time delay in RTL, and by an actual time delay in simulation. Racing caused by a 0 time delay can occur when a circuit description includes gated clocks. See “2.3.1. Unify the description style of FF inferences” for details. Also, a description that causes an assignment to the same signal twice in a single *always* construct can also lead to racing with a 0 time delay. See “2.2.3. Initial value description in *always constructs*” for details.

Racing caused by an actual time delay in simulation can be avoided by shifting the input signal events from clock edges.^[1] There are various other problems. Please refer to;

“4.1.3. Note input signal timing”

“4.1.8. Description where the results do not differ due to the simulators”

“4.1.9. Simulation between asynchronous clocks”

for details.

Clock synchronization design is essential for designs that use HDL and logic synthesis tools. Use of static timing analysis tools is effective for verifying the timing for synchronous circuits.^[3]

It is not so easy to define synthesis constraints for asynchronous circuits that are prone to errors. As a result, an unexpected circuit may be generated. Therefore, it is very important to carefully examine whether the generated circuit satisfies the timing requirements. The asynchronous part may not be so large in your entire design, and it is the best to verify this part with gate-level simulation.^[2] See “1.2.1. Clock synchronous design” and “1.5. Handling of asynchronous circuits” for clock synchronization design.

4.4.4. Other causes of mismatches between RTL and gate level

- [1] Confirm that the all necessary items are on the sensitivity lists
- [2] Incomplete sensitivity lists cause mismatches between RTL and gate level simulation results
- [3] Bugs in libraries, and tools also cause mismatches

reference

reference

reference

Explanation

When describing a combinational circuit with an `always` construct, all input signals must be defined in a sensitivity list. (See “2.2.2. Define every input signal in an *always construct* in the sensitivity list “ for details). However, the logic synthesis tool ignores the sensitivity list described in RTL when generating combinational circuits. In other words, the functions in an `always` construct will be generated as a circuit even when some necessary items are omitted in the sensitivity list. This may result in a simulation mismatch.^[1] In order to avoid this problem, either a RTL checking tool or a formal verification tool should be used.

Note as well that in Verilog-2001 standard, `">*` can be used in sensitivity list for an `always` construct, as shown in Example 4-35. By specifying `*` all the input signals are included that there will be no omission problem. However, not all the tools support this syntax at this point, so it is better not to use this yet.

Conventional Verilog description

```
always @(A or B or SEL)
  if (SEL==1'b1)
    Y = A;
  else
    Y = B;
```

**Verilog-2001 description**

```
always @*    // "*" can now be set in sensitivity list
  if (SEL==1'b1)
    Y = A;
  else
    Y = B;
```

Example 4-35 Verilog-2001 Sensitivity list

Simulations may mismatch due to improper RTL description.^[2] There are many cases of this such as the difference in ideas of logic synthesis tool and the usage of directive, which depends on logic synthesis tool. Please refer to the following item for details.

“2.3.1.[7] Whether you use an asynchronous or a synchronous reset, be sure to use only one of them in a line”

“2.5.1.[6] inout should not be directly connected to input/output”

“2.8.1.[5] Do not force full_case for case statement directives that depend on a particular logic synthesis tool”

“2.8.5. Description relying on parallel_case is prohibited”

Mismatch can be caused by bug of libraries or tools.^[3] Libraries and tools are something you are given but it is risky to completely trust them. In reality, however, you can not design without them and there is no other thing you can rely on. Verification has to be done from not one point but multiple points of views. Therefore, it is better to use more than 2 tools for final check. To check the correctness of description, it is better to use tools in order of priority given below by allowing as much time as possible.

1. RTL check tool
2. Formal Verification tool
3. Gate level simulation

If a bug is found in logic synthesis library, fault can be found by comparing RTL and gate level by Formal Verification tool. Also, if there is a bug in Formal Verification tool, checking same function by gate level simulation can find fault.

4.5. Static timing analysis

4.5.1. Key points of static timing analysis

[1] Not only gate simulation but also static timing analysis is necessary	mandatory
[2] Check all timing violations	recommend 1
[3] When violations occur, other violation paths may not have been displayed	reference
[4] Do not specify "set_false_path" for signals inside of ASIC as much as possible	recommend 3
[5] Do not specify set_multicycle_path inside ASIC as much as possible	recommend 3
[6] Specify set_failelse_path, set_multicycle_path carefully by pin-to-pin	recommend 1
[7] In the case of false path, insert FFs and cut the path on the circuit	reference
[8] Perform timing analysis of the ASIC I/O pins one by one	reference
[9] If false_path is used between asynchronous clocks, then check each path	recommend 2

Explanation

In today's LSI design, checking timing by static timing analysis has become the standard method.^[1] A static timing analysis feature is included in synthesis tools such as Design Compiler or BuildGates, but stand-alone tools such as PrimeTime or tools provided by ASIC vendors are also used.

One problem of static timing analysis is when paths that do not need to be analyzed are also shown. For example, signals not required for actual operation such as test circuits, or paths that operate within 1 cycle in the circuit are displayed as violations.^[2] In addition, the real violation paths may be hidden in these unnecessarily displayed paths, so a careful examination must be done.^[3]

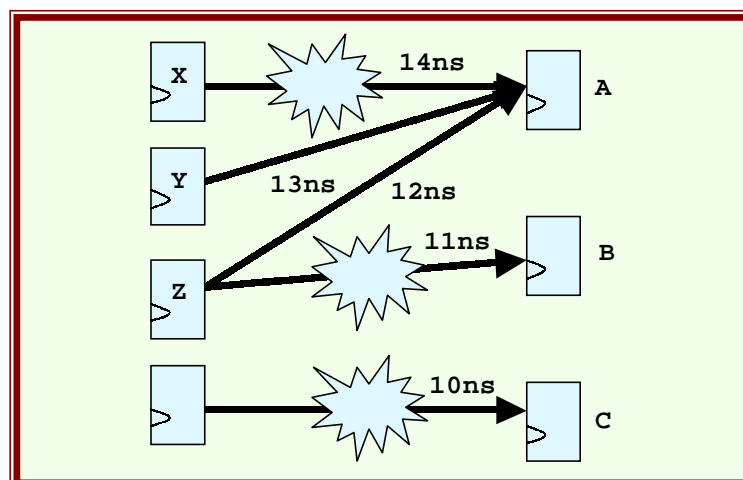


Figure 4-28 Sample timing analysis

If there is a circuit like that illustrated above in Figure 4-28, then

A : 14ns

B : 11ns

C : 10ns

is displayed in the static timing analysis tool. If the target operation cycle is 12 ns, then the only path violating is 14 ns path. There is no problem with B:11ns and C:10ns because they are within the target delay value. In addition, A:14ns path could be a false path from X which is not a violation. But there may be other paths that arrive at A, for example, a path that arrives from Y to A may be a violation target. But this path will not be shown without some special setting so this violation could be overlooked.

Timing analysis tools have a constraint called `set_false_path`. This command can remove paths that reach A from X without these being subject to analysis. If these paths are removed from the targets of timing analysis, then the paths that reach A from Y are displayed and analysis becomes possible. However, if this method is frequently specified in an ASIC, the necessary paths may then be removed as well. Too many specifications for this may cut out the necessary paths.

Cutting out the timing paths (`set_false_path`) presents such problems, so consider it a rule that this constraint is seldom used in an ASIC.^[4] Whenever you specify, specify safely as follows

```
set_false_path -from X/CK -to A/D
```

with pin-to-pin.

```
set_false_path -to A/D
```

As in the above specification, if you specify only the end or the beginning of the analysis, it becomes unclear whether it was safely done.

If `set_false_path` needs to be specified in the timing analysis, cut out the timing paths in advance by inserting FFs, then design so that the speed is kept within 1 cycle. Inserting FFs naturally increases the gate area that was not previously necessary.^[7] However, it is generally understood that even if there is some increase in the area due to the insertion of FFs, this is more desirable than setting a huge number of `set_false_paths`. If you set too many `set_false_paths`, it becomes more difficult to guarantee that there are no timing errors.

LSI fault analysis is usually performed at substantially lower speeds than the actual operation speed. The failure detection circuit does not have to run as fast as the actual LSI operation speed. If LSI fault analysis is to be performed when a specific test terminal is active, you can set a signal value for this test terminal so that it will not be included in the timing path.

```
set_case_analysis 0 TST_mode
```

By specifying this constraint, those sections where a path is separated by the fixed value will be excluded from analysis. Use this constraint to separately perform timing analysis

4.5. Static timing analysis

for the normal operation mode and test mode.

The timing analysis results are displayed in order starting from those with the most violations relative to the target cycles. This checking, which is done downwards from the top, may not cover paths that require careful checking since some of these may not have been analyzed properly due to insufficient specification. Be sure to separately display and check paths for at least all ASIC I/O pins and RAM.^[8]

```
report_timing -from all_inputs() -to all_registers(-data_pin) -max_path 300 -nworst 3
report_timing -from all_registers(-output_pin) -to all_outputs() -max_path 300 -nworst 3
report_timing -from all_inputs() -to all_outputs() -max_path 300 -nworst 3
```

As shown above, the signals that are output from the input port to FF data input, from the Q output of FF to the output port and from the input port directly to the output port should be displayed separately from other timing analysis, and delay values should be checked for every 1 pin.

Some circuits may operate on 2 cycles because the throughput is not sufficient on 1 cycle. For this type of path, the current timing analysis tool does not automatically recognize that the operation is on 2 cycles. In this case, the `set_multicycle_path` constraint is used to specify on how many cycles the circuit is operated.

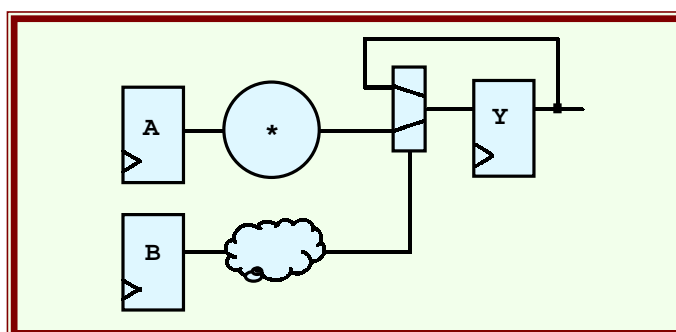


Figure 4-29 Multiplier operated on 2 cycles

In the above figure, if operation is on 2 cycles due to the slow speed of the multiplier(*), set

```
set_multicycle_path 2 -from A/CK -to Y/D
```

`set_multicycle_path`, like `set_false_path`, should be specified pin-to-pin.^[6] If specifying only the end in the above circuit,

```
set_multicycle_path 2 -to Y/D
```

the path that goes from B, which originally operates with 1 cycle, to Y should be specified also as 2 cycles. As for `set_multicycle_path`, there is a risk of setting one, to be operated as 1 cycle, to 2 cycle mistakenly like `set_false_path`. Therefore please try to create a circuit structure without setting `set_multicycle_path`.^[5]

A similar Explanation is given in RMM:5.6.7.

4.5.2. Circuit design according to static timing analysis

- | | |
|--|-------------|
| [1] Too many specifications of <code>set_false_path</code> , <code>set_multicycle_path</code> makes analysis difficult | reference |
| [2] Avoid letting the circuit operate on multicycle as much as possible | recommend 1 |
| [3] For false paths violating timing analysis, insert FF or cut the path | recommend 2 |

Explanation

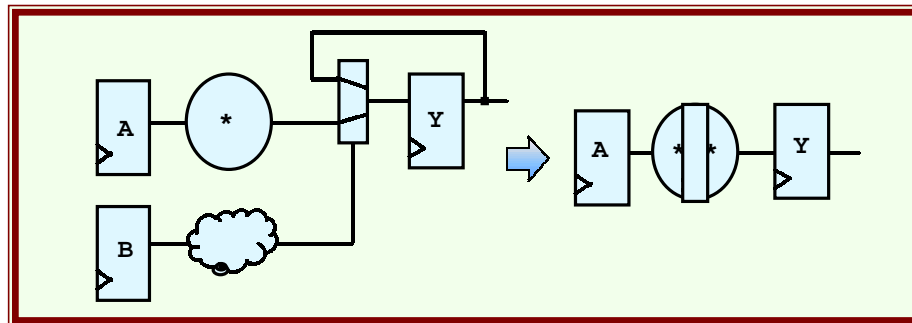


Figure 4-30 Changing multiplier to operate in 2 stages

As explained in “4.5.1.Key points of static timing analysis”, timing analysis is essential with current designs. With the current design style, the circuit structure has to be made so that simple timing analysis is possible. To accomplish this, pay attention to the following:

- Make the clock system simple. (Decrease the number of clock as much as possible.)
- Do not insert a logical circuit between asynchronous clocks.
- Do not use tri-state.
- Divide the circuits operating on multicycle into 2 stages or use divide clock.
- For false paths in timing analysis, insert FFs or cut paths.

When too many `set_false_path` and `set_multicycle_path` are specified, the timing path analysis cannot be performed safely.^[1]

When a multiplier that you want to use set a `multicycle_path` is slow, you should use instead a multiplier that operates in two stages as shown in Figure 4-30, or divide the clock and change it to a circuit that operates in 1 cycle.^[2]

When dividing the clock in 1/2, special attention must be paid, since when a clock system gets complicated, it also complicates the timing analysis.

Circuits should be designed to avoid the use of `set_false_path` as much as possible.^[3] When there are false paths, consider inserting FFs, or disconnect it if that path is not a part of the actual operation. Designs with bus structures create more false paths because busses are binded. Care should be taken to avoid occurrence of false paths to the greatest extent possible.

Chapter5 Logic Synthesis

by DesignCompiler

This chapter introduces examples of creating commands and scripts, and the methods for optimizing circuits using the Design Compiler made by Synopsys, Inc. Creating scripts is important in order to implement optimization that meets the required performance of the circuits optimized. The commands introduced here are compliant with the 2001.08 version. Part of the descriptions in this chapter is related to the versions before 2000.11-SP2.

Some commands have changed from previous versions. Version names were added to commands that were added or changed after 1997.01 so refer to them. The interfaces used in this chapter relate to `dc_shell`. The Tcl interface is not explained here.

Contents

- 5.1 `dc_shell` basic commands
- 5.2 `dc_shell` advanced commands
- 5.3. Basic principles of Synthesis
- 5.4 Script examples
- 5.5 *characterize* optimization
- 5.6 Circuit optimization including operators
- 5.7 2001.08 performance and comparison
with past versions



hd Lab, Inc.

5.1. *dc_shell* basic commands

5.1.1. Command script example

- [1] Create a batch file and execute batch processing using *dc_shell*
- [2] Consider that *design_analyzer* is only for checking circuit diagram or testing optimization
- [3] Prepare various standard scripts to execute optimization without errors

recommend 1

reference

reference

Standard Script Example

```
sh date
Design_name = mansell
file_name = "../rtl/" + Design_name + ".v"
gate_name = "../gate/" + Design_name + ".v"
rep_name = "../log/" + Design_name + ".rep"
read -f verilog file_name
current_design Design_name
set_operating_conditions -max worst -max_library asc018_MAX \
                        -min worst -min_library asc018_MIN

set_wire_load_model -name asc_5000area
set_wire_load_mode enclosed
create_clock -p 9 -w { 0 4.5 } -name CLK CLK
set_input_delay 1 -clock CLK all_inputs()
set_output_delay 4 -clock CLK all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))

set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_fanout 11 find(design,"*")
set_max_area 0
derive_clocks()

compile
ungroup -all -flatten
compile -incremental
check_design
rep2 = rep_name + "_1"
rep3 = rep_name + "_2"
sh cp rep2 rep3
sh cp rep_name rep2
report_timing -max_path 4 -net > rep_name
report_timing -path end -max_path 100 >> rep_name
report_constraint -verbose >> rep_name
report_area >> rep_name
write -f verilog -hier -o gate_name
quit
```

Explanation

For logic synthesis, stable operation should be given priority over improvement of the area or speed performance. We recommend preparing standard scripts in order to execute logic synthesis without any errors. The designers will modify these standard scripts, and then execute logic synthesis.^{[1] [3]}

Among the commands described in the above script examples, *set_wire_load_model* is

is not necessary when an `auto_wire_load_selection` (a function that automatically selects the `wire_load` model according to the design size) is specified in the technology library. Be sure to execute other commands from `set_operating_conditions` to `derive_clocks`.

The above script is for versions 1999.10 to 2001.08. If you are using versions from v3.4a to 1999.05, please replace

```
set_wire_load_model -name asc_5000area
set_wire_load_mode enclosed
```

with

```
set_wire_load asc_5000area -mode enclosed
```

The example described assumes that the library delay value is described in *ns* units. The above script has 1 clock, which can be used as it is provided the circuit has the structure explained in "1.6.2. Make basic block FF output & combinational circuit input".

The following may have to be specified for some libraries besides this command, depending on ASIC vendors.

```
set_max_transition 0.8 find(Design,Design_name) (0.8 in case of ns units)
```

Normally, libraries created by ASIC vendors have specifications of `max_capacitance` (maximum capacitance) on the output pin of each cell. When the vendor has this specification, the circuit should be generated so that the total amount of the capacity of the input pin of cell loaded from the driver and the capacity of wire does not exceed the maximum capacity, so the `set_max_transition` constraint is not necessary. However, if a designer determines that the `max_capacitance` value specified for output pins of each library cells is too lenient, this constraint can be used.

When your design is targeted for deep submicron technology from 0.18 to 0.13 μm , this should be carefully considered. This is because the appropriate value is technology dependent. Although it varies among different vendors, the appropriate value should be around 0.25-0.4 ns for 0.13 μm , 0.35 - 0.9ns for 0.18 μm and 0.5-1.7ns for 0.25 μm technologies.

The `set_max_fanout` constraint also requires some special consideration for creating deep submicron layouts. The number of fanouts will be relatively high for cells with higher drive capacity when only `max_capacitance` is set for each output. During layout, nets with many fanouts tends to require longer wire than expected, and this is not desirable even if it is driven by higher drive capacity cell.

```
set_driving_cell -cell NIIL -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
```

By specifying these constraints, a buffer tree will not be generated for the `RST_X` signal. This is on the assumption that a buffer tree is generated at layout like a clock tree synthesis (CTS). Please refer to "1.3.1. Use asynchronous reset for initial reset" and "1.4.2. Use clock tree synthesis for clock balancing" for details.

With versions from 1999.05-1999.10.5, you have to specify `set_ideal_net` or buffers will be inserted for asynchronous reset lines. `set_ideal_net` can only be specified for nets, and thus is very inconvenient. In the releases planned for 2002, a new command `set_ideal_network` is slated to be introduced.

If an asynchronous reset line is used, the following can be specified.

```
set_driving_cell -cell NI1L -pin X all_inputs() - CLK
set_max_fanout 1 RST_X
```

5.1.2. *read* command (reading design)

- `read -format <format file name> [-netlist]`
- `analyze -format [verilog|vhdl] <filename> [-work <library name>]`
- `elaborate <design name> [-parameters <parameter> | -file_parameters <parameter>] [-gate_clock]`

read Command

<code>-format</code>	<code>vhdl</code>	- VHDL file
	<code>verilog</code>	- Verilog-HDL file
	<code>db</code>	- Synopsys object data
	<code>edif</code>	- edif file
<code>-netlist</code>		- Accelerates lading of the gate-level netlist written in Verilog
<code>-define</code>	macro name	- Verilog only
		Defines macro name (2001.08 or later)

Usage notes

- *The VHDL syntax is not case sensitive, but the Design Compiler is. Pay careful attention to the fact that if names such as port names have the same name but different letter cases, these signals will be treated as different signals.
- *It is possible to set the compiler so that it is not case sensitive by using options, but pay careful attention to the fact that problems may occur in other operations (depends on flow tool used).
- *With Verilog, a new default for 2001.08 or later is "presto". `hdlin_enable_presto=true` can be specified for 2000.05 and 2000.11. However, there are many bugs in versions prior to 2001.11-SP2, so the use of this setting might be dangerous.
- *To read a Verilog description that includes `'ifdef`, `hdlin_enable_vpp=true` should be specified.

analyze Command

Use *analyze* and *elaborate* commands instead of *read* when using packages or *generic declarations* in VHDL, or parameter descriptions in Verilog.

The analyzed data will be stored as an intermediate file in the directory specified by `.synopsys_dc.setup`.

```
.synopsys_dc.setup
```

```
define_design_lib work -path work
define_design_lib USBM -path /home/library/syn/1999.10-4/
```

A directory called "work" is created under the current working directory with the *library name* "work".

elaborate Command

This command will create circuit objects from the *analyzed* and stored data. The *read* command will perform the same thing as executing *analyze* and *elaborate* commands. Specify the top level when using the *elaborate* command. Its sub-levels will be automatically linked if the analyzed data has been stored.

Usage Example

Verilog:

```
module LOWLEVEL(...);
parameter W=5;
parameter K=5
...
endmodule

module HIGHLEVEL(...);
...
LOWLEVEL #(13,5) LOWLEVEL
...
endmodule
```

VHDL :

```
entity LOWLEVEL is
generic( W : integer := 5;
          K : integer := 5;
);
...
end RTL;

entity HIGHLEVEL is
...
LOW1 : LOWLEVEL generic map(13,5)
...
end RTL;
```

```
analyze -f [verilog|vhdl] LOWLEVEL.v
analyze -f [verilog|vhdl] HIGHLEVEL.v
elaborate HIGHLEVEL          ---- -Lower levels will be automatically elaborated
```

```
analyze -f [verilog|vhdl] LOWLEVEL.v
elaborate LOWLEVEL -parameters "W=5,K=13"  -- -Elaborates lower levels only
```

Gated clock generation with Power Compiler

```
elaborate design_name -gate_clock
```

5.1.3. *current_design* command (design specification)

- **current_design** <design name>

Function

Specifies the *design name* (entity name, module name) to be compiled.

Explanation

The read design automatically specifies the current design after the design is read (after executing the read command), but this command should be used to specify the top level when there are multiple modules or entities in a single file. This command is also used to move between levels when synthesizing levels.

5.1.4. *set_operating_conditions* command (specify operating conditions)

- **set_operating_conditions** [-library <library name>] [<operating con name>]
- **set_operating_conditions** [-library <library name>]
[-max <max ope env name>] [-min <min ope env name>]
[-max_library <max lib name>] [-min_library <min lib name>]

- [1] It is easier to consider executing all the operating conditions by MAX for logic synthesis
- [2] MIN and MAX can be specified at the same time, but you must not first synthesize with the MAX operating environment and then change it to the MIN operating environment to run the hold time guarantee optimization

reference

mandatory

Function

- The operating conditions are equivalent to the K factor resulting from delay changes attributed to the operating voltage and operating temperature.
- For 0.18 and 0.13um technology, libraries are provided for each operating environment that depending on the vendor, this command may not be necessary.

Sample operating conditions model				Check the name and contents of the operating condition. report_lib <Library name>
NAME	Proc	Temp	Volt	
WCCOM	1.57	70.00	1.81	
WCIND	1.74	100.00	1.81	
WCMIL	2.25	125.00	1.72	

Example Specification

- With logic synthesis, it is necessary to set the maximum delay value to meet the constrained delay value even in the worst conditions. If you are using a library set for two or more of the worst conditions, choose one of these.

```
set_operating_conditions -library asic018 WCCOM
```

For 0.25 and 0.35 um technology libraries, the typical conditions and the best conditions are also included in the library. In this case, choose the worst conditions.

Sample operating conditions model			
NAME	Proc	Temp	Volt
-----	-----	-----	-----
WORST	1.74	100.00	2.37
TYP	1.00	100.00	2.50
BEST	0.61	-25.00	2.63

```
set_operating_conditions -library asic025 WORST
```

Usually, the worst conditions are used for both maximum delay optimization and the hold time guarantee. However, in order to more accurately perform the hold time guarantee, the best conditions (MIN) should be used. In this case, both best conditions and worst conditions have to be set. If separate libraries are provided for these conditions, you can usually set

```
target_library = { aisc018_MAX.db }
set_min_library aisc018_MAX.db -min_version aisc018_MIN.db
```

in `.synopsys_dc.setup`, and then set the

```
set_operating_conditions -max WCCOM -max_library aisc018_MAX ¥
                        -min BCCOM -min_library aisc018_MIN
```

command. This simultaneous usage of MAX and MIN has been supported from 1998.02. The names to be specified by `max_library` and `min_library` are not the *library file name*, but rather the *library name* included in the file.

If the MAX and MIN library names are the same, and if several operating conditions are included in the library, you can set it as follows.

```
set_operating_conditions -max WCCOM -max_library "aisc018_MAX.db:aisc018"
                        -min BCCOM -max_library "aisc018_MIN.db:aisc018"
```

If you are just running synthesis for which detailed timing analysis is not necessary, the synthesis run will be easier if you maintain the worst operating conditions when doing hold time guarantee.^[1]

Use the settings explained above when you want to run hold time analysis with best conditions.

You should never synthesize first with worst (MAX) condition, and then changing it to best (MIN) to run hold time guarantee.^[2] If you take this approach for logic synthesis, hold time guarantee may result in an increased delay value.

5.1.5. `set_wire_load_model` command (wire load model specification)

- `set_wire_load_model -name <wire_load_name> [-max] [-min] <design_name>` (1999.10 or later)
- `set_wire_load_mode [top|enclosed|segmented]` (1999.10 or later)
- `set_wire_load <wire_load_name> [-library <library_name>] [-mode top|enclosed|segmented]` (1999.05 or before)

[1] In principle, the `wire_load` model should be set for each basic level

recommend 3

[2] The `wire_load` model of min should be set uniformly for all the levels

recommend 1

Function & Points

- Specifies the wire load model appropriate for each circuit size.
- It is possible to automatically select the wire load according to the circuit area if you are using a library whose auto wire load attribute has been set. (Auto wire load selection)
It is not necessary to use the `set_wire_load_model` in this situation. However, note that there are many cases in which a particular wire load model is required for Top level synthesis or timing analysis.

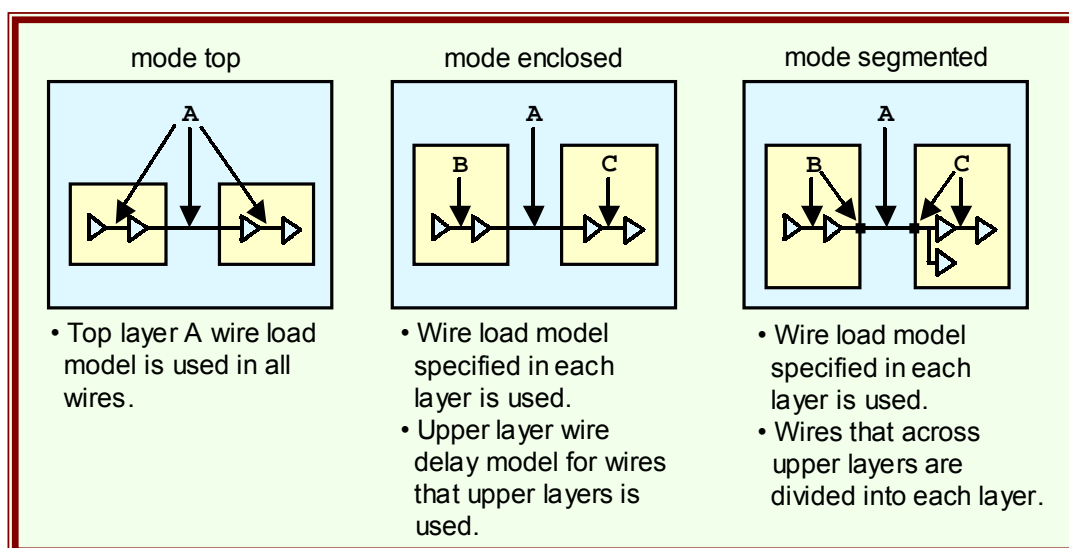
Setting the wire load for circuits with levels (example usage of `set_wire_load_mode`)

When using separate wire load models in lower levels, specify either

```
set_wire_load_mode enclosed
```

or

```
set_wire_load_mode segmented
```



Sample wire load model

asc_5000area	Block with < less than 5,000 gates>	} Specified at lower layer
asc_20000area	Block with 5,000-20,000 gates	
asc_45000area	Block with 20,000-45,000 gates	
asc_80000area	Block with 45,000-80,000 gates	
asc_125000area		
asc134_chip	Specify according to the die size	} Specified at the top layer
asc304_chip		
asc474_chip		
asc664_chip		
asc914_chip		
asc115_chip		

Usage Example

The constraint of the wire_load model for each level is as follows.

```
set_wire_load_model -name asc5000area(for each level) -library asc035
set_wire_load_mode enclosed
(1999.05 or before)
set_wire_load asc5000area -mode enclosed
```

The setting of wire_load model for the top level is as follows according to the die size.

```
set_wire_load_model -name asc134_chip(for chip) -library asc035
set_wire_load_mode enclosed
```

In lower levels, those libraries in which the auto wire load selection is set are automatically set. In this case, it is not necessary to specify the wire_load model for each lower level. However, wire_load should not be decided based only on the size of each level but, actual wire length changes depending on how the level is layout and whether hard macro exist. To improve performance during layout, it is recommended to specify appropriate wire_load for each level rather than to judge wire_load by the size of each level.

If wire_load is specified to synthesize lower level and saved in db format, the wire_load becomes valid by reading the db file of the lower level during upper level synthesis.

However this will be invalid if it is saved in Verilog format. Moreover, there are some cases where the specification for the lower levels is not adequate, so it should be separately set starting from the top level.

Add the following for each level.^[1]

```
set_wire_load_model -name asc5000_area -library asc035
find(design,"Design name")
```

If the wire_load model is changed for each level, the number of buffers inserted between FFs for hold time analysis (MIN path analysis) will not be uniform. To prevent this, it is better to use the same wire_load model for MIN path analysis.^[2]

```
set_wire_load_model -name asc5000_area -min find(design,"")
set_wire_load_model -name asc5000_area -max Design name(set for each level)
```

In the case of ASIC with technology below 0.25 um, the difference between a wire delay calculation with virtual placement/routing and a delay calculation after layout has become largely incompatible. This is particularly true for a large-module, which is problematic. It is not advisable to try to improve speed during logic synthesis by setting a wire_load model, which indicates an excessively large virtual wire length, because area will increase. It is recommended instead to execute logic synthesis by taking the area into account as explained in "5.4.1. Area optimization".

If you are applying a uniform wire load model for the entire design, set

```
set_wire_load_model -name asc115_chip find(design,"TOP level name")
set_wire_load_model top
```

for the top level only.

WireloadModel is explained in RMM:6.2.6.

5.1.6. The *create_clock* command (clock definition)

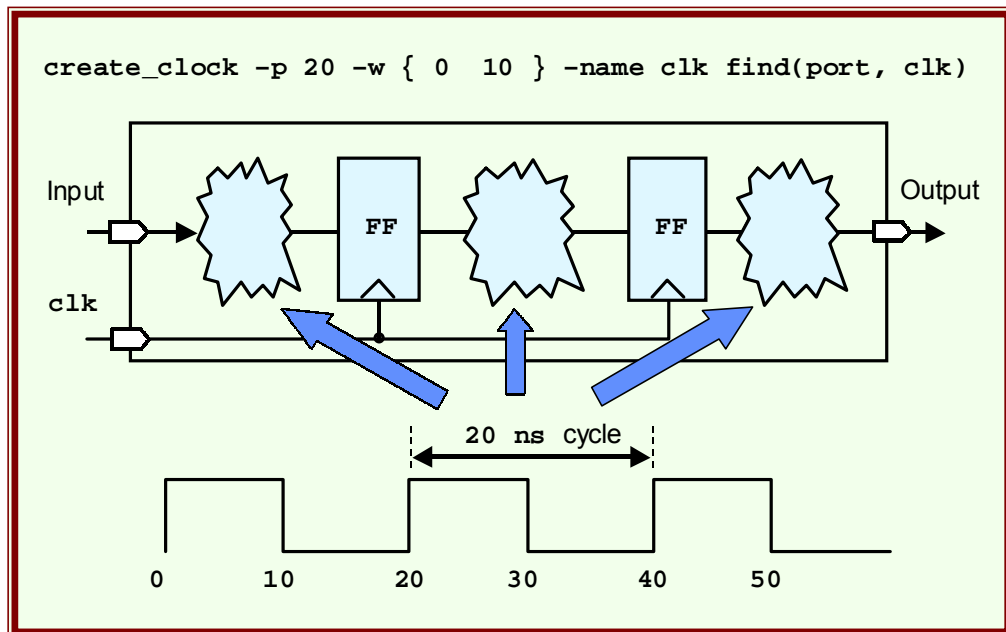
- **create_clock** -p <cycle> -w { <rise> <fall> } [-name <clock_name>]
- clock port name or cell output pin name>

- | | |
|---|-------------|
| [1] When there are no clocks whose phases are different, the rising edge time should be 0 | recommend 1 |
| [2] Do not use decimals for the cycle, rise and fall | recommend 3 |
| [3] Check any missing clock settings by derive_clocks() command | recommend 1 |
| [4] Add create_clock to the input port of the top level or the primitive cell pin | recommend 1 |

Function & Points

- *create_clock* settings are required (check using *derive_clocks*) for all clock pins.
- Optimization of the speed is not performed unless the clock cycle is set, and inoperative IC might be generated because the fact that there were speed violations is not noticed.
- The rise time should be set to 0 for a single-phase clock^[1] (in order to make the report easier to read and to avoid unnecessary problems).
- Sets the reference name of the clock using the "-name" option.
- Cycle will be applied as timing constraint

Constraint Example



- The *port name* becomes the *clock name* when *-name* is omitted
`create_clock -p 6 -w { 0 3 } find(port, clk)`

- Set a virtual clock only for combinatorial circuits
`create_clock -p 6 -w { 0 3 } -name clk`

- Set a two-phase clock
`create_clock -p 6 -w { 0 3 } -name clk1 find(port, clk1)`
`create_clock -p 6 -w { 3 6 } -name clk2 find(port, clk2)`

- Command checking for missing settings
`derive_clocks()`

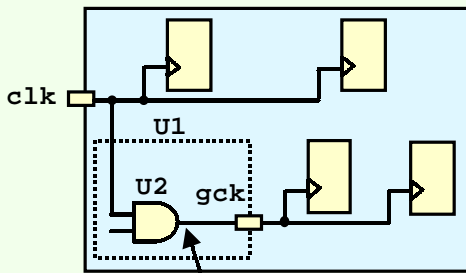
When clock settings are missing, `derive_clocks` command checks which timing is not analyzed.

For gated clocks, the cell's output terminal names should be specified instead of the port names.

Constraint Example

```
create_clock -p 6 -w { 0 3 } -name clk find(port, clk)
create_clock -p 6 -w { 0 3 } -name gateclk find(pin, "U1/U2/X")
```

Gated clocks slightly delay from clk, but rise specifies 0, same as clk



- Specify as pin
- Do not specify to the internal port (gck)

find (type, object name)

- Search for objects with the specified name by the specified type. Return the object name if it is found.
- When using multi phase clocks, "find (port, clk)" is mandatory since it is necessary to analyze timing with the port names and clocks lines separated.

5.1.7. set_input_delay, set_output_delay (input, output delay setting)

```
● set_input_delay <delay_value> -clock <clock_name> <port_name>
  set_output_delay <delay_value> -clock <clock_name> <port_name>
```

```
● create_clock -p 20 -w { 0 10 } -name clk find(port,"clk")
```

```
  set_input_delay 6 -clock clk all_inputs()
```

```
  set_output_delay 9 -clock clk all_outputs()
```

* all_inputs() : All input ports

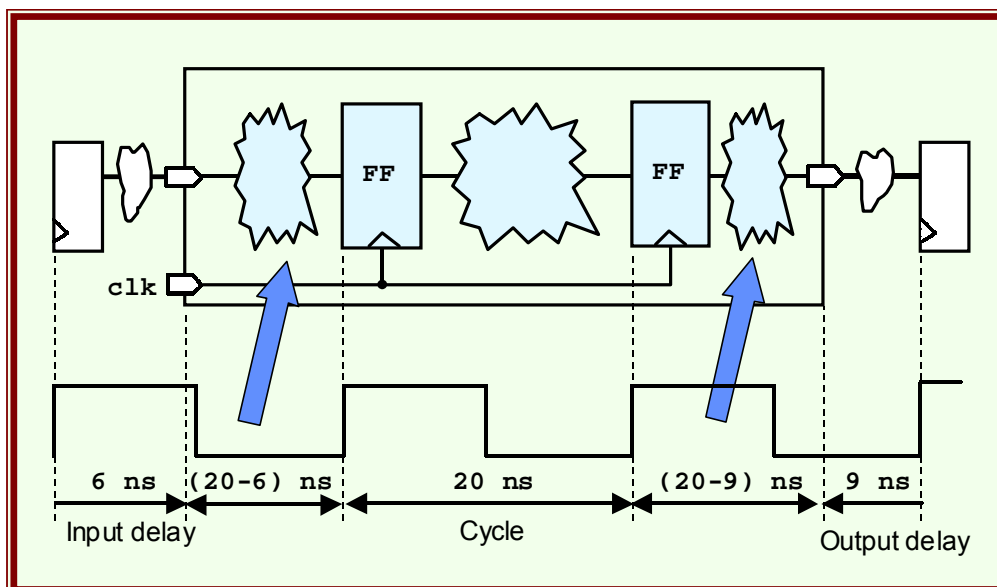
* all_outputs() : All output ports

[1] Always specify "-clock" for set_input_delay and set_output_delay

mandatory

Points

- Forgetting the -clock option is very hazardous since problems may occur during the timing analysis. Always be sure to specify the -clock option.^[1]
- There is also the method of specifying delay using the *max_delay* constraint. However, this method should not be used since the path groups are different and there is a danger of set_input_delay, and set_output_delay being lost when these path groups coexist.
- The (cycle - delay) value will be applied as a constraint.



Example Specification

If the structure of the basic block is close to the structures of combinational logic input and FF output, set the amount of delay of FF (about 1 ns at 0.35 μ m). Since combinational logic may be over 2 levels, it is standard to specify about half a cycle.

```
create_clock -p 10 -w { 0 5 } -name CLK find(port, CLK)
set_input_delay 1 -clock CLK all_inputs()
set_output_delay 5 -clock CLK all_outputs()
```

When the delay value is known (or when it does not have a basic block circuit structure), the delay value should be added for each port.

```
set_input_delay 8 -clock clk find(port, SECCO)
set_input_delay 1.4 -clock clk find(port, MINCO)
set_output_delay 7 -clock clk find(port, HOURLSB)
```

Specification of blocks containing only combinational circuits

```
create_clock -p 20 -w { 0 10 } -name clk
set_input_delay 0 -clock clk all_inputs()
set_output_delay 10 -clock clk all_outputs()
```

Port specification unnecessary

Specify virtual clocks, and then specify set_input_delay and set_output_delay based on that clock

5.1.8. The `set_driving_cell` command (drive strength setting of input port connection)

- `set_driving_cell -cell <cell_name> [-pin <pin_name>] <input_port_name> [-lib <name>]`

[1] Always specify `set_driving_cell` for basic block optimization except for the Chip level

mandatory

[2] Use `set_ideal_net` for asynchronous reset lines

mandatory

[3] Pay attention to specifications to the clock and reset

reference

Points

Unless this command is specified, the delay value of the net connected to the input ports as in the following example will become 0 even if 1,000 cells have been connected. Always remember to use this command.^[1]

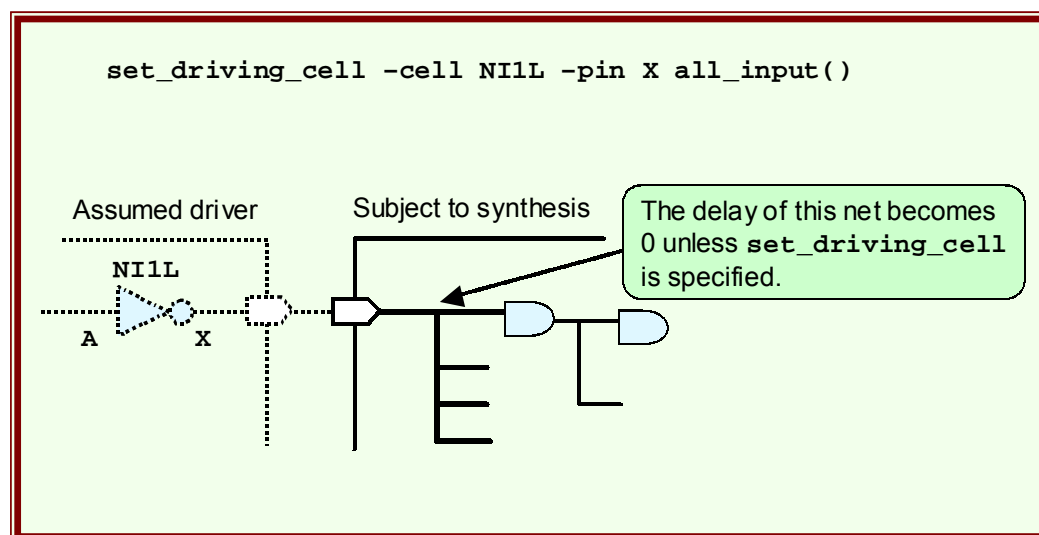
fanout (or *max_capacitance*) constraints can be added as of 1997.01 release.

(The following warning message will be output when the constraints are set.)

Warning:Design rule attributes from the driving cell will be set on the port.
(UID-401)

Ports other than the clock port and the reset port are used as is.

Constraint Example



Settings for the clock input port

- Optimizing clock lines using current logic synthesis tools is hazardous. In this case, insert clock buffers using a method other than a synthesis tool.
- Normal clock lines (and occasionally reset lines) automatically insert buffers on the layout side. In this case, specify the following constraints so the buffers are not inserted into the clock line in the logic optimization phase (reset line).

```
set_drive 0 find(port,clk)
```

OR

```
set_driving_cell -cell NIIL all_inputs() - CLK
```

The objective of this setting is to set the clock delay to 0. (Depending on the library, there are cases in which the clock line delay applies an unexpected delay to the connected FFs.) This prevents more buffer cells from being inserted by the *fanout* or *max_capacitance* constraints.

Among recent ASICs, it has become common for the reset tree synthesis to be performed to the reset line like the clock line. The same settings should be performed when executing reset tree synthesis.

```
set_driving_cell -cell NIIL all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
```

However, when `set_ideal_net` is specified, that net will be excluded from the design rule check and the buffers will not be inserted.^[2] Even if the buffers have been inserted, they will be removed. If you have placed buffers for CTS, specify `set_dont_touch` for that cell.

As for the clock lines, `set_ideal_net` will be automatically applied for nets connected to the clock pins of FFs at the time when `create_clock` is specified. Designers have to set this manually for asynchronous reset lines (1999.05 or later). `set_ideal_net` will not be added if the clock line has gated clocks.

`set_ideal_net` can only be applied to nets. Use the `all_connected` command to search for nets connected to reset ports.

For synchronous resets, or if you are not using reset tree synthesis for asynchronous resets, a buffer tree for the reset line needs to be generated with logic synthesis.

In this case, add a *fanout* constraint or a *max_capacitance* constraint for the reset ports instead of using `set_ideal_net`.

```
set_max_fanout 1 find(port,RST_X)
```

OR

```
set_max_capacitance load_of(iv/a) find(port,RST_X)
```

The cell drive capacity is specified to the clock line either after the I/O buffers are inserted or gated clocks are used. The following is specified in this case:

```
set_clock_transition 0 all_clocks()
```

Refer to "5.2.2.5.set_clock_transition" for more details.

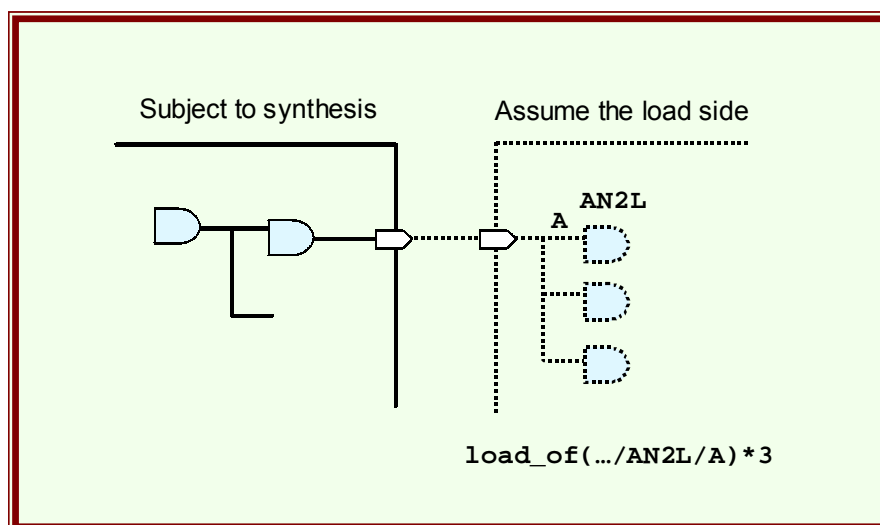
5.1.9. The `set_load` command (set output port load)

- `set_load <capacitance_value> <port_name>`

Points

Load capacitance is not as important as `set_driving_cell`, but adding it is a good idea. Specify the input pin name of standard low drive cell using `load_of` for the capacitance values added by `set_load`.

However, the wire capacitance is not reflected, so it would be better to specify more than the estimated amount. Specifying `x3` for estimated `x2`, and `x5` for `x3`, is preferable.



Constraint Example

```
set_load load_of(asic018/AN2L/A) * 3 all_outputs()
```

In the chip level specification, it is recommended that you do not specify `set_load`, or that you set the direct value with pf units such as 5pf and 10pf rather than internal cells.

```
set_load 10 all_outputs()
```

5.1.10. The `set_max_area` command (set area constraints)

- `set_max_area <value> [-ignore_tns]`

[1] Ordinarily, set 0 for the value

recommend 1

[2] In synthesis of a large scale design, run time may become long by constraining `set_max_area 0`

reference

Points

Optimizing to the area direction has a lower priority than in speed, so always specify 0 (minimum area).^[1] (It would be pointless to specify any other value than 0.)

Setting `set_max_area 0` should be considered carefully in a large scale design.^[2]

Adding `set_max_area 0` may make synthesis run time 10 times in some design of more than 100 thousands gates.

If possible, use `set_max_area 0` in small logic synthesis.

```
set_max_area 0
```

* Total negative slack:

From the 1998.02 version, all the paths that violate constraints are the objects of optimization of speed. Until that version, only 1 was the object of such optimization in 1 path group. However, unlike the specification of the `group_path -critical_range`, the aim is not to improve completely all the violating paths of speed. Therefore, the validity of the `group_path` command does not change.

Since this function is supported, it is necessary to pay due attention to the speed constraints (the cycle set by `create_clock`). If the final speed attained greatly exceeds the given constraint, the paths of the second or later are shortened and the area may increase.

Moreover, since the paths of the second or later are extra short, the most delayed delay value may become rather large. It is better for the final speed attained to be close to the given constraint.

This function can be eliminated by the `set_max_area` command.

```
set_max_area 0 -ignore_tns
```

Total negative slack itself has been supported since the 1998.02 version, but this command is available only from 1999.05.

To decrease the area, or depending on the situation, specify the above constraint.

In the benchmark result, about 50% of the designs are somewhat improved in speed when the total negative slack function is eliminated.

5.1.11. The *compile* command (logic synthesis and circuit optimization)

```
● compile [-map_effort low|medium|high] [-incremental_mapping]
          [-only_design_rule] [-no_design_rule] [-in_place]
```

[1] Hierarchical synthesis and optimization after `ungroup` are
`compile -incremental`

recommend 1

[2] Use `-map_effort high` with `-incremental`

recommend 1

Options

<code>-map_effort</code>	low	Optimization with weak optimization capability (used to obtain results quickly)
	medium	Normal optimization (default)
	high	Optimization that is strong in speed optimization (Normally used with <code>-incremental_mapping</code> in the second or subsequent optimization execution) ^[2]
<code>-incremental_mapping</code>		Optimization that improves the circuit quality based on the current gate structure. Used for hierarchical synthesis or the second or subsequent optimization ^[1]
<code>-only_design_rule</code>		Optimizes only by correcting design rule violations Design rules: - Max fanout count, max capacitance of each wire, skew time -> Predefined in libraries, automatically executed - Min delay value (fix hold time) -> Activated by <code>set_clock_skew</code> and <code>set_fix_hold</code> commands
<code>-no_design_rule</code>		Synthesizes without correcting design rule violations
<code>-boundary_optimization</code>		Changes the I/O port polarity. Apply when there are small levels under the basic block and they are to be retained.
<code>-top</code>		Added in 1999.05 release as a new option. Is used in optimization from the top level. In contrast to <code>-inc</code> , only the paths that pass through the top level ports are subject to optimization, and the lower level is not optimized.
<code>-area_effort</code>	low	Specify the intensity for attempting to decrease the area (1999.05 or later)
	medium	Decrease the area 1 or less % by specifying high
	high	
<code>-ungroup</code>		After synthesizing the Boolean equations, it ungroups the hierarchy and performs gate level synthesis. It is better to execute the <code>ungroup</code> command explicitly than to execute it when compiling.

5.1.12. The *report* command group (synthesis results report)

5.1.12.1. *report_timing*

```
● report_timing [-max_path <output_count>] [-path full|end] [-net] [-to <end_point>]
  [-from <start_point>] [-net] [-delay max|min] [-input_pins] [-nworst]
```

Points

- The distribution of delay values can be observed when using the "-path end" option (the delay values of each end point in each line) and outputting many delay results. This is used to consider the synthesis strategy and the layout.
- Use -from -to to observe certain paths.
- We recommend checking the *fanout* count of each cell on paths where the delay is large by using -net.

Options

-max_path <i>output count</i>	Outputs a timing report for the provided paths
-path full end	Displays all cells that pass through the timing path (default) Use when displaying only the end point in one line and outputting many points
-net	Displays the names of nets that pass through the timing path (Can check the fanout count of each cell)
-to <i>end point</i>	Specifies the end point of the timing path to be displayed. When specifying a point other than the analysis end point (FF D input or output port), it displays the timing path that passes through that point
-from <i>start point</i>	Specifies the start point of the timing path to be displayed. When specifying a point other than the analysis start point (FF CK input or input port), it displays the timing path that passes through that point
-delay min max	Specifies the maximum delay value or minimum delay value (for hold time guarantee)
-input_pins	Displays the input pins of each cell in the timing path report
-nworst <i>count</i>	Specifies the number of paths to display for each end point. Only 1 path is displayed in the case of the default setting

Execution Example

```
report_timing
report_timing -net -max_path 3
report_timing -path end -max_path 100
report_timing -to COF_reg/D
report_timing -delay min -path end -max_path 50
report_timing -from all_inputs() -to all_registers(-data_pin) -max_path 50
```

Output Result

```
dc_shell> report_timing -net
```

Startpoint: U47/U30/U166/Q_reg ← The starting point is FF in this case.
(rising edge-triggered flip-flop clocked by P_CKIN)

Endpoint: P_I Eb (output port clocked by P_CKIN)

Path Group: P_CKIN ← End point in this case is output port **P_I Eb**

Path Type: max ← Displays which clock the end point is dependent on

Point	Fanout	Incr	Path	

clock P_CKIN (rise edge)		0.00	0.00	Delay value of the cell
clock network delay (ideal)		0.00	0.00	in question
U47/U30/U166/Q_reg/CK (FD1AM)		0.00	0.00 r	
U47/U30/U166/Q_reg/Q (FD1AM)		556.08	556.08 f	
U47/U30/U166/Q_reg/Q (net)	4	0.00	556.08 f	
U47/U313/a1 (cla2_10)		0.00	556.08 f	
U47/U313/a1 (net)		0.00	556.08 f	f: fall delay, r: rise delay
U47/U313/U6/X (RT2L)		262.70	818.78 r	
U47/U313/n1862 (net)	1	0.00	818.78 r	
U47/U313/U5/X (AR224L)		286.09	1104.87 f	Cumulative delay value
U47/U313/n1860 (net)	2	0.00	1104.87 f	to this point
U47/U313/U7/X (RT2L)		253.42	1358.30 f	
.....				
U7/U9/X (NI1L)		127.83	15058.71 f	
U7/n555 (net)	6	0.00	15058.71	f fanout count
U7/U41/X (RN4L)		320.69	15379.40 r	
U7/PIN_I Eb (net)	1	0.00	15379.40 r	
U7/PIN_I Eb (ctl_reg)		0.00	15379.40 r	
P_I Eb (net)		0.00	15379.40 r	
P_I Eb (out)		0.00	15379.40 r	
data arrival time			15379.40	Total delay value of this path

clock P_CKIN (rise edge)		14000.00	14000.00	
clock network delay (ideal)		0.00	14000.00	
output external delay		-3000.00	11000.00	
data required time			11000.00	

data required time		11000.00		
data arrival time		-15379.40		

slack (VIOLATED)		-4379.40		Violates constraint by 4379.40ps

(continue from the previous page)

```
dc_shell> report_timing -path end -max_path 4
```

Endpoint	Path Delay	Path Required	Slack

P_IEb (out)	15379.40 r	11000.00	-4379.40
gb[17] (inout)	15319.11 f	11000.00	-4319.11
DTR0b (out)	15193.39 f	11000.00	-4193.39
DTR1b (out)	15067.07 f	11000.00	-4067.07

Displays four delay values for each end point in order from the slowest.

5.1.12.2. report_constraint

- `report_constraint [-all_violators] [-verbose]`

Function

Displays the provided constraints and the items that violated them.

Options

- | | |
|-----------------------------|--|
| <code>-all_violators</code> | Displays all violating items. |
| <code>-verbose</code> | Displays a detailed report for each violation (only one for the single worst violation). |

Execution Example

```
report_constraint
report_constraint -verbose
report_constraint -all_violators
```

Output Result

```
dc_shell> report_constraint
```

Group	(max_delay/setup)	Cost	Weight	Cost
BCLK		0.80	1.00	0.80
IO_CLK		0.00	1.00	0.00
default		0.00	1.00	0.00
max_delay/setup				0.80

Maximum delay:

Created path group (refer to 5.2.5) for each clock (by `create_clock`). Worst path is displayed. `cost` displays the cumulative violation value.

`weight` displays the compiled weight value.

Group	(min_delay/hold)	Cost	Weight	Cost
BCLK (no fix_hold)		0.00	1.00	0.00
IO_CLK		0.00	1.00	0.00
default		0.00	1.00	0.00
min_delay/hold				0.00

Minimum delay:

Constraint	Cost
max_delay/setup	0.80 (VIOLATED)
min_delay/hold	0.00 (MET)
max_fanout	0.00 (MET)
max_capacitance	0.00 (MET)
max_area	4071.60 (VIOLATED)

Displays whether or not the constraints are met. (MET) means that constraints have been met.

```
dc_shell> report_constraint -verbose (The speed violation report is omitted)
```

```
Net: RLPC[12]
  max_fanout      200.00
- Fanout          42.00
  Slack          158.00 (MET)
```

Name of net that violated **fanout** constraint

fanout count

```
Net: CPARREY/A10W
  max_capacitance 11.76
- Capacitance      9.08
  Slack           2.68 (MET)
```

Name of net that violated output specified capacitance with `max_capacitance` for each cell

```
Design: PCANEW
```

```
  max_area      0.00
- Current Area  405.54
  Slack         -405.54 (VIOLATED)
```

The constraints have been violated, but there is no problem since `set_max_area 0` is set to 0.

5.1.12.3. report_reference

- report_reference

Function

Displays the cell count and area used. When checking only the cell area, this command is more convenient than report_area.

Output Result

Reference	Library	Unit Area	Count	Total Area	Attributes

AH1L	asc035	6.600000	10	66.000000	r
AN4L	asc035	7.600000	4	30.400000	↑ Cell attributes
AN4M	asc035	8.000000	4	32.000000	

Total 58 references				5592.500488	

Meaning of attributes names

b - black box (unknown)	Blocks that do not contain logic (RAM, ROM, etc.)
d - dont_touch	Cells not modified on synthesis
r - removable	Cells not removed on synthesis
u - dont_use	Cells not used in synthesis (I/O cells, etc.)
s - statetable	Cells with state descriptions (cells such as FFs, latches)

5.1.12.4. report_area

- **report_area**

[1] Grasp not only the gate area but also the wire area value by the **report_area** command

mandatory

Function & Points

Displays the area used. If the wire area is added to the wire_load model, then this command also reports the wire area.^[1]

We recommend using the **report_reference** command if the wire area is not output.

With current ASIC, not only the gate area but also the wire area rate (wire/gate area) is very important depending on the layout. The wire area of each level should also be grasped. If the wire area is not specified to the library to be used, ask the ASIC vendor to define the area in the library.

```

Number of ports:      254 ← I/O port total
Number of nets:      1231
Number of cells:      994 ← Cell count total
Number of references:  58

Combinational area:   4198.100098 ← Gate count of combinatorial logic
Noncombinational area: 1394.399902 ← FF latch gate count
Net Interconnect area: undefined (Wire load has zero net area)
                        ↗ Becomes undefined if the wire area is not set

Total cell area:      5592.500000
Total area:           undefined ← Total gate count
  
```

5.1.13. The write command (saving a design, generating a netlist)

- **write [-format <format>] [-hier] [-output <file_name>] [<design_name>]**

Options

-format verilog|vhdl|db|edif|xnf

Specifies the output format

- The database data are read quickly and are small

The constraints provided for synthesis can also be retained using db format.

-hierarchy

Saves the lower level data too (should be specified in most cases)

-output file_name

Specifies the file_name

Always add this option since it becomes <design name>.db when omitted

5.1.14. The *check_design* command

- **check_design**

Main checked items and how to fix them

Warning : In design <module>, there are X multiple drivers nets with unknown wired logic type

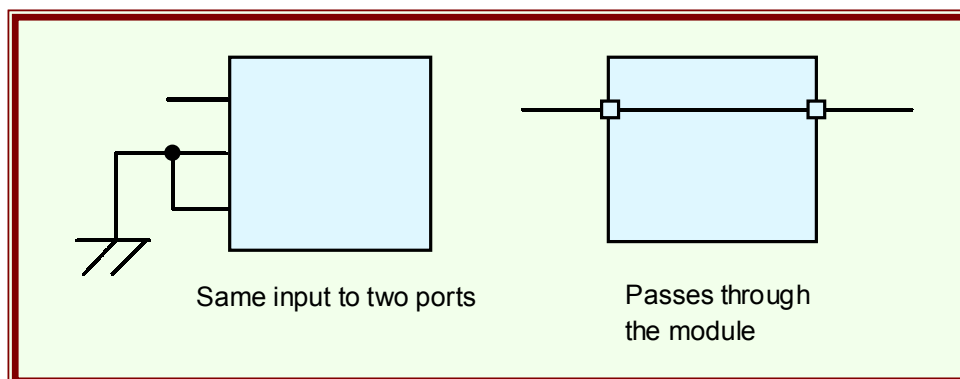
In situations such as that of driving one signal from two *always* constructs, the Design Compiler inadvertently generates Wired AND. This is a hazard warning, so correct the description immediately.

Warning : In design <module>, port <name> is not connected to any nets

There are unconnected ports. Either correct the description or use the commands `ungroup`, `remove_unconnected_ports`.

Warning : In design <module>, the same net is connected more than once
Warning : In design <module>, port <name> is connected directly to output

There is no problem in the circuit, but the ASIC vendor rules may have been violated. If violation occurs, correct the description or automatically insert buffer in the violated part by `set_fix_multiple_port_nets` command.



5.2. *dc_shell* advanced commands

5.2.1. Setting the hold time guarantee

- `set_clock_uncertainty <value> [-from <clock_name>] [-to <clock_name>] [-setup] [-hold] [-rise] [-fall] [<clock_name> | <cell_output_pin> | <port_name>]`
- `set_clock_latency value [-max] [-min] [-rise] [-fall] [-source] <clock_name> | <cell_output_pin> | <port_name>`
- `set_propagated_clock <clock_name> | <cell_output_pin> | <port_name>`
- `set_clock_skew [-uncertainty <value>] [-plus_uncertainty <value>] (until 1998.08) [-minus_uncertainty <value>] [-delay <value>] [-propagated]`

[1] Do not use `set_propagated_clock` for logic optimization unless it is analysis after layout

mandatory

[2] Specify `set_clock_uncertainty -from, -to` to change the skew value between different clocks

reference

[3] Do not use `-source` option of `set_clock_latency` for logic optimization

- `set_fix_hold <clock_name>`

[4] Log the timing report by **MIN** before **compile** when `set_fix_hold` is specified

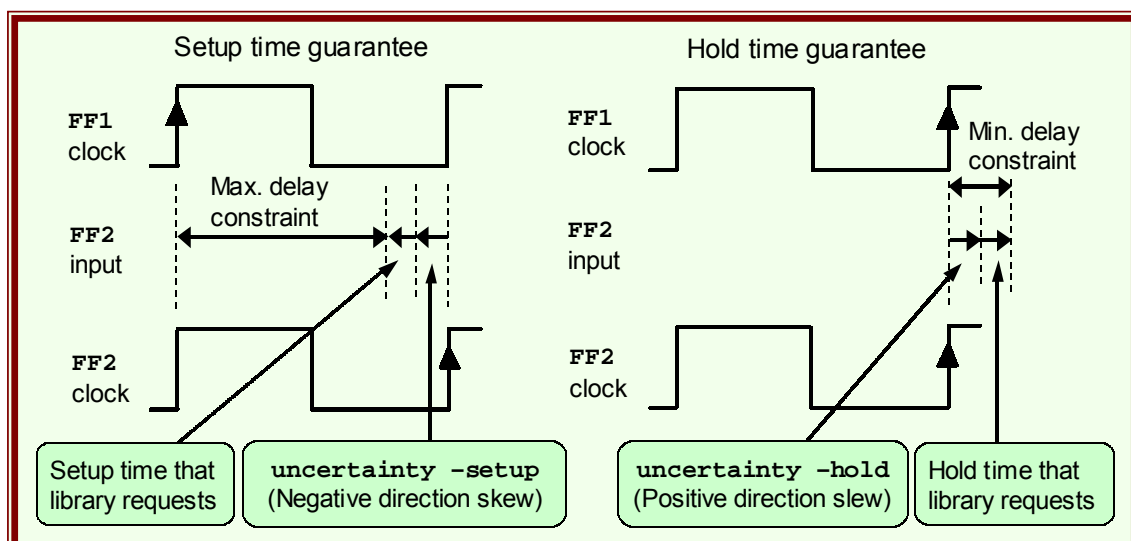
recommend 1

[5] Pay attention to the paths from the input port, to the output port and between different clocks, for the hold time guarantee

mandatory

Function

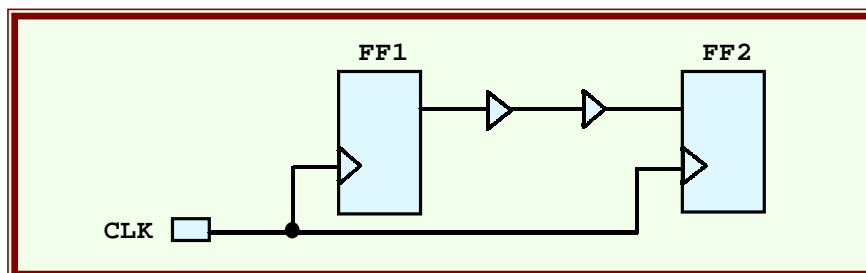
- Sets the clock delay and skew.
Unless it is set, the delay value, clock line delay and skew are assumed to be 0.
- Guarantee the hold time
- Buffers for guaranteeing hold time are not inserted unless this constraint is specified.



Constraint and Execution Example

```
(1999.05 or later)
set_clock_uncertainty 0.3 find(clock, CLK)
set_fix_hold find(clock, CLK)
report_timing -delay min -max_path 30 > Chip_min_mae.rpt
compile -inc
report_timing -delay min -max_path 30 >> Chip_min.rpt

(1998.08 or before)
set_clock_skew -uncertainty 0.3 find(clock, CLK)
set_fix_hold find(clock, CLK)
```



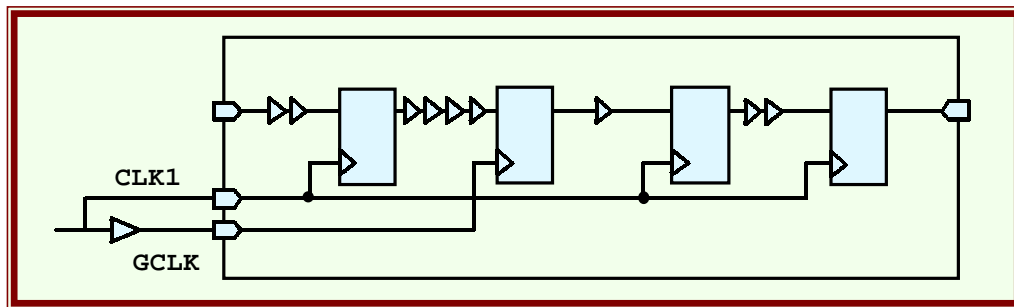
When executing compile, use `set_fix_hold`, output the MIN timing report before and after compile.^[4] It is difficult to fine after compile that a large number of buffers are inserted (failing in minimum timing analysis). If outputting the MIN timing report before Compile, and a delay value that causes a large violation is found, adjust by the constraints; `set_clock_uncertainty`, `set_input_delay` and `set_multicycle_path`. Attention should be paid particularly to the paths, from input port, to the output port and between the different clocks.^[5]

The hold time guarantee is re-buffered at the upper level synthesis even if buffers are inserted by specification at lower level synthesis. One execution of the hold time guarantee at the end of synthesis of upper level is sufficient.

* The hold time guarantee when there is a 0.7ns delay in the GCLK line

```
(1999.05 or later)
set_clock_uncertainty 0.3 find(clock,CLK1)
set_clock_uncertainty 0.3 find(clock,GCLK)
set_clock_latency 0.7 find(clock,GCLK)
set_fix_hold find(clock,CLK1)

(1998.08 or before)
set_clock_skew -uncertainty 0.3 find(clock,CLK1)
set_clock_skew -uncertainty 0.3 -delay 0.7 find(clock,GCLK)
set_fix_hold find(clock,CLK1)
```

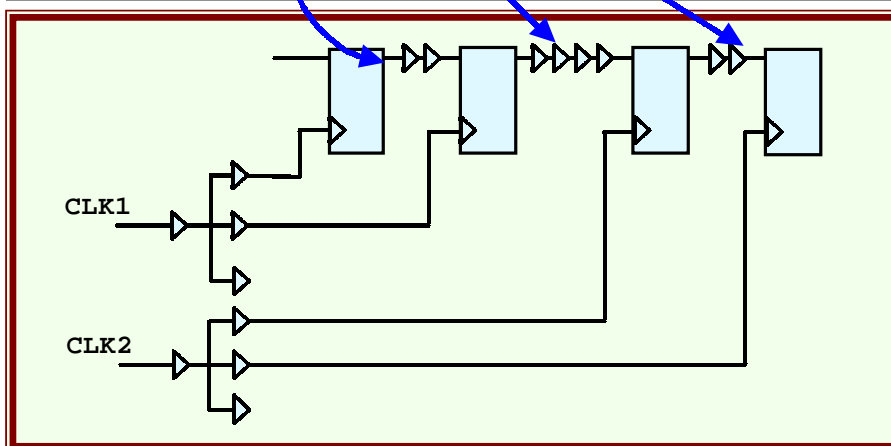


The clock line delay value cannot be clarified unless clock tree synthesis is performed. It is verbose to set `-rise`, `-fall`, `-min` and `-max` separately, so it is better to constrain clock line as a constant delay value by clock system constraints such as `set_clock_latency`. If the assumption is defeated, adjust the problem on the layout side. However, `set_propagated_clock` should not be set by using dummy buffers for clock tree synthesis. After thorough consideration, set the value by `set_clock_latency`. The value set by `set_clock_latency` can be reset by setting `set_propagated_clock` during analysis after layout.^[1]

The values set by `set_clock_latency -source` are not reset by the `set_propagated_clock` command. At the logic synthesis before layout, it is better not to use the `-source` option.^[3] Attention should be paid so that trouble does not occur due to the values remaining during analysis after layout.

* In the event that there are two clocks (CLK1, CLK2) and you are guaranteeing the hold time by changing the number of buffers to be inserted between CLK1 and CLK2^[2]

```
(1999.05 and after only)
create_clock -p 10 -w { 0 5 } CLK1
create_clock -p 10 -w { 0 5 } CLK2
set_clock_uncertainty 0.6 -from find(clock,CLK1) -to find(clock, CLK1)
set_clock_uncertainty 0.6 -from find(clock,CLK2) -to find(clock,CLK2)
set_clock_uncertainty 0.9 -from find(clock,CLK1) -to find(clock,CLK2)
set_clock_uncertainty 0.9 -from find(clock,CLK2) -to find(clock,CLK1)
set_fix_hold find clock,clk)
```



* Concept of `set_operating_condition` for the hold time guarantee

The hold time guarantee is normally performed by setting MAX (max. delay value) for the operating conditions. Do not set MIN for the operating conditions for guaranteeing hold time. In terms of guaranteeing the hold time, Design Compiler has a feature to ensure that the maximum delay value will not exceed the constraint value. When MIN is set for the operating conditions, the maximum delay value diminishes and is accidentally judged to be a path that has slack to meet the constraints. When this problem occurs, there is a danger of the maximum delay value not meeting the constraints after hold time is fixed.

(1999.05 and later)

```
set_operating_conditions -library asc018_MAX WCCOM
create_clock -p 10 -w { 0 5 } CLK2
set_clock_uncertainty 1 find(clock,CLK1)
set_fix_hold find(clock,clk)
```

(1998.08 and before)

```
set_operating_conditions -library asc035 MAX174
create_clock -p 10 -w { 0 5 } CLK2
set_clock_skew -uncertainty 1 find(clock,CLK1)
set_fix_hold find(clock,clk)
```

From the release of 1998.02, it became possible to simultaneously set MAX (maximum delay value) and MIN (minimum delay value). When both are specified, there is no problem since the hold time and the maximum delay values use the MIN and MAX operating conditions, respectively.

(1998.02 and later)

```
set_operating_conditions -library asc018 -max MAX174 -min MIN047
create_clock -p 10 -w { 0 5 } CLK2
set_clock_skew -uncertainty 0.3 find(clock,CLK1)
set_fix_hold find(clock,clk)
```

Recently, it has become common that the setting of operation conditions is included in the library. In such cases, set as follows.

```
set_operating_conditions -max MAX -max_library CMOS_MAX ¥
                        -min MIN -min_library CMOS_MIN
```

* `set_wire_load` for the hold time guarantee

Refer to "5.1.5. `set_wire_load` command (wire load model *specification*)".

* The hold time guarantee at layout

It has become possible to insert buffers for the hold time guarantee with layout tools. When using PBopt, adjustment is eventually made by layout and therefore it is not necessary to have completely fixed timing at logic optimization. However, the place to

insert buffers at layout should be saved before layout. This depends on how much domain is saved. However, it is not possible to insert too many buffers. Therefore, the buffers should be inserted temporarily at logic optimization and fine adjustment should be made during layout.

When adjusting during layout, use `script`, which is used at logic synthesis (output the given constraint by the `write_script` command and PBopt read it). Therefore, unless the appropriate setting is made at logic synthesis, the hold time guarantee will not be performed automatically.

The operation may become inappropriate, when using many tri-states and there are constraints of `set_input_delay` and `set_output_delay` the in primitive circuit cell pins, or when many `set_false_paths` and `set_multicycle_paths` are set. In such a case, the buffers must be inserted into the circuit manually for the hold time guarantee after layout.

When adjusting during layout, due attention should be paid if multiple clocks exist. In the analysis after layout, set

```
set_propagated_clock
```

to the root point of each clock and modify the script so that the delay value of clock line is correctly reflected.

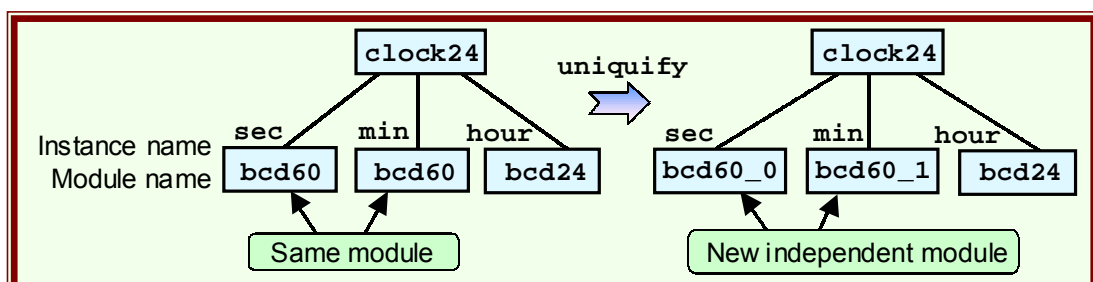
5.2.2. Hierarchical synthesis (`uniquify`, `set_dont_touch`, `ungroup`, `compile -inc`)

5.2.2.1. `uniquify`

- `uniquify`

Function

- Uniquify modules when there are two or more of the same modules in lower levels.
- Automatically changed module names are handled as different modules.
- The `compile` command does not operate unless it is executed using multiple modules.



5.2.2.2. `set_dont_touch`● `set_dont_touch <instance name>`

- [1] Specify **dont_touch** only for asynchronous circuits, tri-state buffers and buffers for which drive capacity has to be fixed
- [2] Avoid hierarchical synthesis that sets **dont_touch** for the lower levels

reference

reference

Function

- Specifies the instances to be excluded from being subject to optimization.
- Specifies for cases that you do not want to be subject optimization effects, e.g. in the case of asynchronous circuits.
- The *compile* command can be executed when specified for cases where there are two or more of the same modules. (*uniquify* becomes unnecessary.)

Execution Example

```
set_dont_touch find(cell, "A_block")
set_dont_touch find(cell, "B_block")
```

5.2.2.3. `ungroup`● `ungroup [-all | <instance_name>] -flatten`

- [1] `ungroup` sub levels of basic blocks and lower
- [2] Do not `ungroup` above basic blocks

recommend 1

mandatory

Function

Ungroup hierarchy. All the levels of blocks under basic blocks stipulated in "1.6. Hierarchical design" should be ungrouped. This is because the layout performance does not decrease much even without consideration being given to the small size blocks during layout. Moreover, ungrouping lower levels by logic synthesis tool removes ports of each block. When ports are removed, the combinational logic part, which passes ports, is further optimized that makes area increase.

If you do not want to ungroup lower blocks out of consideration of the grouping function by layout, process the open port of lower levels (especially the level of arithmetic operation generated by logic synthesis DW01??) by `remove_unconnected_ports` command.

The above basic blocks should be used during layout and therefore should not be ungrouped by logic synthesis tools.

Options

-all	Ungroup all instance levels in the design
<i>instance name</i>	Ungroup only the specified instance levels
-flatten	Ungroup all levels under the specified instance
	Normally only ungroup one level

Execution Example

```
ungroup -all -flatten
ungroup find(cell, "cell name")
```

5.2.2.4. Hierarchical synthesis (compile -incremental_mapping)

[1] Use incremental compile as the basis of hierarchical synthesis (lower level optimization has already completed)

recommend 2

[2] Always use incremental compile after ungrouping the lower levels

recommend 1

Explanation

If synthesis is performed without this option, 2 step optimization, Boolean equation optimization -> gate level optimization, is executed (see "5.2.3.Flatten (set_flatten)" for more information). In the case of hierarchical synthesis, results may become worse if the Boolean equations are optimized from an upper level when the lower levels have already been synthesized.

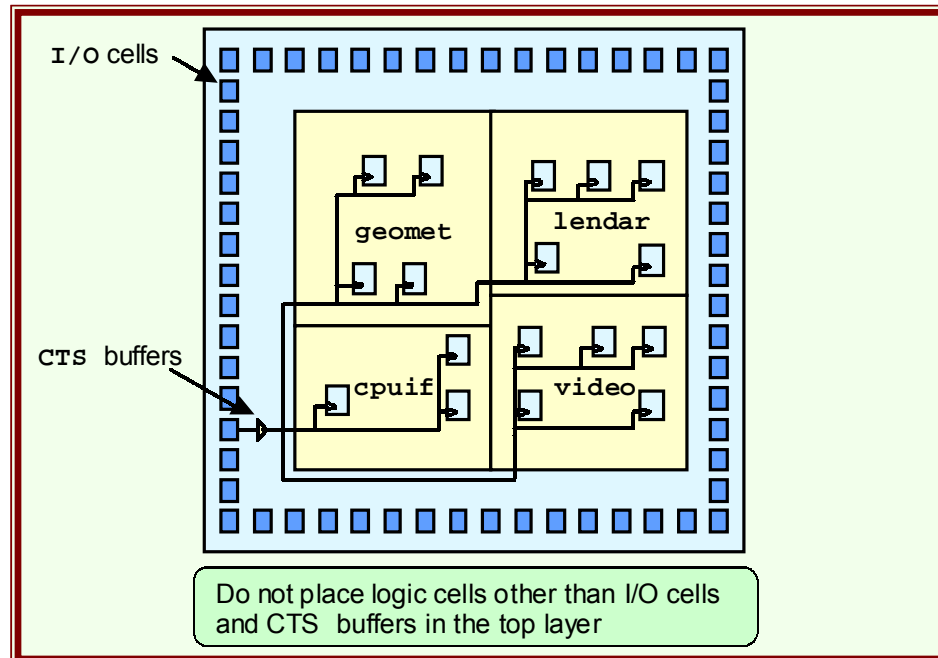
Therefore, for hierarchical synthesis it is better to execute not optimize Boolean equations, but only gate level optimization by adding the -incremental_mapping option.^[1] Optimization proceeds from circuits that have already become gates in the case of incremental optimization, so the results will never get worse than the original results. Especially when operators such as lower levels or DW01_add exist as levels, you should always use incremental compile after first collapsing the levels using the ungroup command.^[2] See "5.4.5.Hierarchical optimization and its concept" for more detail.

5.2.2.5. *set_clock_transition*

[1] Specify after inserting buffers for clock tree synthesis in the top level, etc.

mandatory

Explanation



Recently, it has been common to use clock tree synthesis(CTS) during layout to synthesize clock trees. When using clock tree synthesis, place only one clock buffer in netlists during logic synthesis (before layout). After that, clock tree synthesis inserts the buffers taking into consideration the balance based on the layout.

In the case of netlists before layout, only one buffer should be placed, and then it drives many FF clock pins. For circuits on the order of 100,000 gates, there would be from 2,000 to 3,000 FFs. If the drive capacity is specified for the clock buffer, the clock line will have a large delay value (2ns - 10ns). Clock line delay is not a problem with the Design Compiler since it is ignored. If it is actually due to a transition delay from the library, the clock line transition delay (propagation delay, delay due to wires being added) increases the delay value of the FF cells. When this situation occurs, the cell delay value from the FF clock pins to the Q output used to be about 0.8ns, but is now about 3ns to 6ns. In order to prevent this, specify

```
set_clock_transition 0 all_clocks()
```

after inserting clock tree synthesis buffers.^[1]

If the drive capacity of the clock tree synthesis buffer is set as infinite (resistance value is 0), there is no problem if the drive capacity is not specified. However, the drive capacity is set in some ASIC libraries.

5.2. *dc_shell* advanced commands

In addition, even if the buffer's drive capacity is infinite, when gated clocks exist, you cannot assume that a clock tree synthesis buffer exists for the output of every gated clock. In such cases, this command will be necessary, so it is a good practice always to use this command.

When returning post-layout data to the Design Compiler or Primetime, do not use this specification since it is necessary to have the drive capacity of the clock buffer reflected in the data. (It is also necessary to eliminate this command in the event that you have the logic synthesis constraints reflected to layout tool by the `write_script` command.)

5.2.2.6. `remove_unconnected_ports`

- `remove_unconnected_ports [-blast_buses] <instance name>`

[1] Use this command when not using the `ungroup` command for basic block optimization (not ungrouping the level of the operator generated by Design Compiler)

recommend 1

Explanation

This command is used to process unconnected ports. When the “`-blast_buses`” option is specified, this command eliminates the unused bits among bus signals.^[1]

Use this command when levels are not ungrouped using `ungroup` or other commands. Use

```
remove_unconnected_ports -blast filter(find(cell, "*"), "@is_hierarchical
== true" )
```

to handle all cells in the lower levels.

This command processes the open port. However, specification is incomplete so that not all the open ports (or ports fixed by VDD and GND) can be processed.

In order to completely process open ports, you can ungroup and then group again as follows.

```
ungroup Sub_module
group find(cell,"Sub_module") -design_name Sub_module -cell_name Sub_module
```

However, the *cell name* (Sub_module) will be added to the *instance name*, so it would repeat when displaying signal names with hierarchy information.

5.2.3. Flatten (*set_flatten*)

● `set_flatten true [-phase true] [-effort high]`

[1] Specify with `-effort high`

recommend 1

[2] Executable range is up to 2,000 gates taking run time into consideration

reference

Function & Points

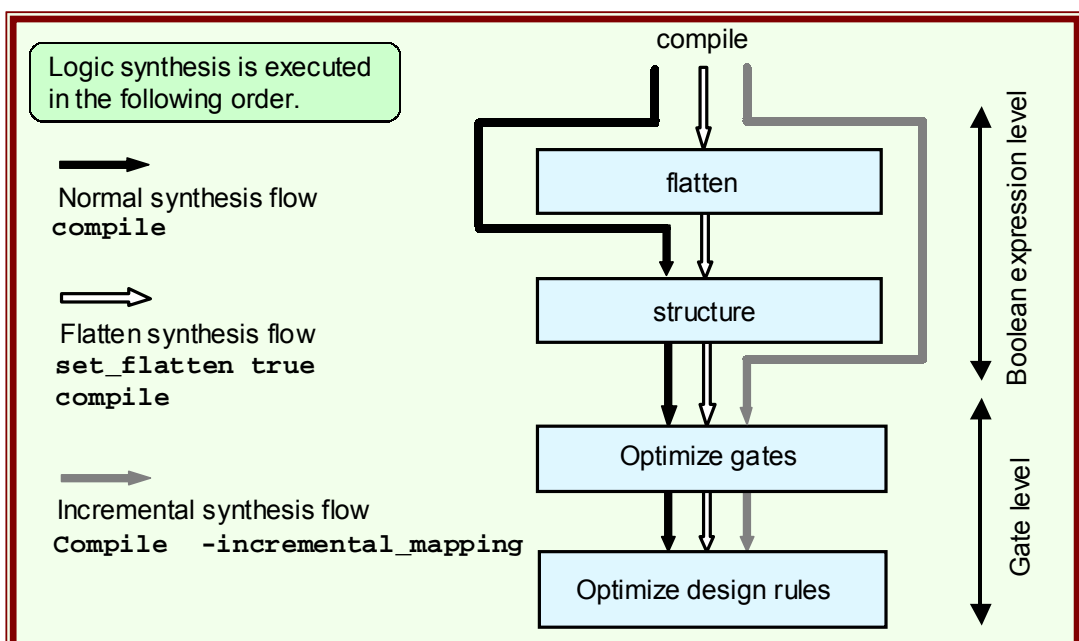
- Converts all logic to AND-OR logic.
- This constraint is often effective for the speed (is often efficient for both speed and area in the case of *case* statements). However, this constraint increases the amount of synthesis run time. Taking the run time into account, it will likely be difficult to execute for more than 2,000 gates.^[2]

Options

- | | |
|---------------------------|---|
| <code>true</code> | Executes flattening at compiling |
| <code>-effort high</code> | <ul style="list-style-type: none"> - Executes flattening regardless of the circuit size. - Use if necessary this option since <i>flatten</i> by default (low) is not executed when the circuit is large^[1] |
| <code>-phase true</code> | <ul style="list-style-type: none"> - Implements flattening by both positive logic and negative logic, then selects the better of the two. - Execution takes twice as long, but the execution results may improve. |

Execution Example

```
set_flatten true -phase true -effort high
```



5.2.4. Structuring (*set_structure*)

● `set_structure [true|false] [-boolean true|false]`

[1] `-boolean true` is effective to decrease area

reference

Function

- Factorize common items of the circuits at the Boolean equation level.
- It is necessary to pay attention since structuring reduces the circuit area, but the speed is prone to slowing down.
- Make it possible to reduce the circuit size by `set_structure -boolean true`.

Options

<code>true</code>	Structures Boolean equations (default value: normally True, is seldom set to False)
<code>false</code>	Does not structure Boolean equations (to improve speed) However, it may increase the area. Is seldom set to False unless a special optimization safety is designed.
<code>-boolean true false</code>	Executes Boolean equations optimization There is little trouble with speed, but when it is necessary to reduce the area, set <code>set_structure -boolean true</code> before <i>compile</i> . ^[1] The area can be reduced but it may lead to lower speed

Boolean equation optimization

Simplifies meaningless numerical expressions in the Boolean equations
-> Implements structuring that takes speed constraints into account.

```
A * !A  -> 0
A + A   -> A
A + !A  -> 1
```

The area reduction effect is great, but TDS may become ineffective and the speed may be decreased.

Example of Boolean equations optimization

```
set_structure -boolean true
```

5.2.5. The *group_path* command

- `group_path -name <clock_name> [-from <port_name or pin_name>] [-to <port_name or pin_name>] [-weight <value>] [-critical_range <value>]`

Function & Points

- The Design Compiler creates one new path group by the *create_clock* command.
- The only path that is subject to optimization in the speed direction is the single slowest path in this group. When the slowest path gets faster, the next slowest path then becomes the target of optimization. Therefore, if there is even one path that is difficult to optimize, another path in the group will not be optimized.
- The *group_path* command makes it possible to generate new path groups and to control path synthesis targets.

Example 1

Apply to circuits where the speed is not satisfactory

```
create_clock -p 12 -w { 0 6 } clk
set_input_delay 2 -clock clk all_inputs()
set_output_delay 2 -clock clk all_outputs()
set_driving_cell -cell FF1 -pin Q all_inputs()
set_driving_cell -cell none clk
set_max_area 0
compile
report_timing -path end -delay max -max_path 80 -nworst 1
```

For instance, after optimizing as above and creating a speed report, the following table results:

Endpoint	Path Delay	Path Required	Slack
-----	-----	-----	-----
s00_reg[6]/D (FF1)	8.36 f	7.17	-1.19
s00_reg[5]/D (FF1)	8.22 f	7.17	-1.05
gengoro/D (FF1)	7.65 f	7.17	-0.48
tanishi/D (FF1)	7.43 f	7.17	-0.26
s00_reg[2]/D (FF1)	7.20 f	7.17	-0.03
s00_reg[3]/D (FF1)	7.18 f	7.17	-0.01
medaka_reg[1]/D (FD1E)	7.16 f	7.17	0.01
mizusumasi/D (FF1)	7.16 f	7.17	0.01

Slow paths with a potential reduction effect are grouped as follows:

```
set_driving_cell -cell FF1 -pin Q all_inputs()
set_driving_cell -cell none clk
group_path -to find(cell,"s00_reg*") -name FOO -weight 5
set_max_area 0
compile
report_timing -path end -delay max -max_path 80 -nworst 1
```

The results are then

Endpoint	Path Delay	Path Required	Slack
-----	-----	-----	-----
s00_reg[6]/D (FF1)	7.17 f	7.17	0.00
s00_reg[2]/D (FF1)	7.17 f	7.17	0.00
gengoro/D (FF1)	7.16 f	7.17	0.01
tanishi/D (FF1)	7.16 f	7.17	0.01

This method is effective when there are few paths with speed violations.

The higher the weight value is, the easier the delay can be optimized compared with other groups. (See "5.1.12.2.report_constraint" for more information.) Note that results may actually worsen when there are too many paths with violated speed.

Example 2 Use the "-critical_range" option

Let us assume that as a result of optimization, the speed constraints were barely met. However, it is the estimated wire delay models that are optimized in the logic synthesis phase. Paths that barely meet the speed constraints would not always meet them after they are actually placed and routed in the layout.

Timing driven layouts have recently been developed, and it is becoming possible to consider the speed constraints during logic synthesis. In addition, if many paths that barely meet the speed constraints exist, it will become difficult to meet them at the layout phase. Normally, the slowest path is subject to being optimized in the speed direction when optimization is executed; the subsequent paths are not optimized. Consequently, even if the speed constraints are met, the number of paths that barely meet the speed constraints will increase.

Speed		
12 ns	-----	Synthesis constraints
11 ns	*****	
10 ns	*****	
9 ns	**	
8 ns	***	
	*	
	**	
	*	

In such situations, there is a higher possibility of the speed not being satisfied by the layout.

When

```
group_path -name CLK -critical_range
```

is specified, the second and subsequent paths also become the target of optimization, and it becomes possible to lower the overall number of paths that barely meet the timing constraints.

Speed		
12 ns	-----	Synthesis constraints
11 ns	****	
10 ns	*****	
9 ns	*****	
8 ns	***	
	*	
	**	
	*	

An actual script is shown below.

```
.....
create_clock -p 12000 -w { 0 6000 } clk
set_input_delay 2000 -clock clk all_inputs()
set_output_delay 2000 -clock clk all_outputs()
set_driving_cell -cell FF1 -pin Q all_inputs()
set_driving_cell -cell none clk
set_max_area 0
group_path -name clk -critical_range 1
compile
report_timing -path end -delay max -max_path 80 -nworst 1
```

Values specified by the `critical_range` option determine how wide a range from the worst value should be optimized for speed. The bigger the specified value is, the wider the range gets.

It would be a good idea not to let this range become too wide. If it is too wide, then there will be too many subject paths and the optimization run time will increase. The area will increase too much as well, actually making the layout results worse. Use roughly 10% or less of the clock cycle as a guideline.

This option actually ends up making it more difficult to speed up the slowest paths. In addition, if there are too many paths that barely meet the constraints, the area may be increased and this option may have a negative effect.

5.2.6. `set_false_path`

- `set_false_path [-hold] [-from <pin_name>|<port_name>|<clock_name>]`
`[-to <pin_name>|<port_name>|<clock_name>]`

[1] In principle, specify pin-to-pin

recommend 1

Function & Points

- Specifies locations such as reset input or input fixed in the power supply GND where timing analysis is not required (or not desired). However, when these locations are specified, the timing is no longer considered so that specifying too many locations is hazardous. Try to specify no more than about 5 locations.^[1]
- Specifying locations is recommended when disabling analysis between clocks that are asynchronous.
- The `-hold` option is used only to disable timing analysis in the case of minimum delay paths (guaranteeing the hold time). Refer to "4.5.Static timing analysis" regarding `set_false_path`.

Example Specification

```
set_false_path -from find(port,RST)
set_false_path -from find(port,PRM1) -to find(pin,U1/Q_reg/D)

set_false_path -from find(clock,CLK1) -to find(clock,CLK2)
set_false_path -from find(clock,CLK2) -to find(clock,CLK1)
```

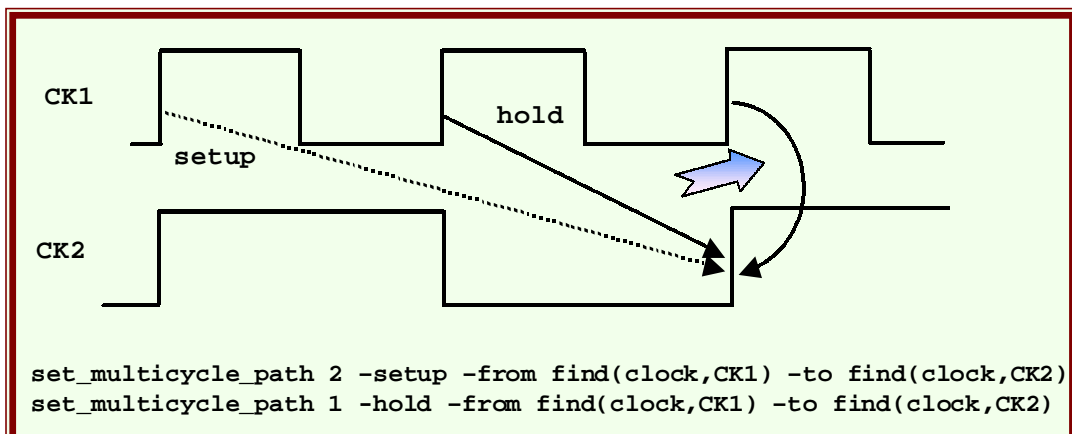
`set_false_path` is explained in RMM:5.6.7.

5.2.7. `set_multicycle_path`

- `set_multicycle_path` count [-from <pin_name> | <port_name> | <clock_name>]
[-to <pin_name> | <port_name> | <clock_name>]
[-hold]

Function & Points

- Should be specified in paths that are processed in two or more cycles. However, it is difficult for this command to specify particular locations. Moreover, if this command is used frequently, it will be difficult to understand the timing analysis results. Avoid using this command as much as possible.
- `set_multicycle_path` should be specified for pin-to-pin. Specifying `-from` or `-to` only runs the risk of applying it to paths that should not be multicycle.
- pin-to-pin for `set_multicycle_path` means from a clock input terminal of FF to a data output of FF.
- For logic synthesis with multiple clocks or when `set_multicycle_path` is set, the position of the hold time guarantee analysis may be insufficient. In this case, the position should be shifted only by the "hold time" length.



5.2.8. write_script, write_sdf

- `write_script <file name>`
- `write_sdf [-version v2.1/v2.0] <file name>`

Function & Points

`write_script` is used to pass the constraint of logic synthesis to layout, or to perform file output for the results of *characterize* execution.

Pay attention to the following when passing the constraints of logic synthesis to layout:

- * `-through` is not used to `set_false_path` and `set_multicycle_path`
- * The `set_clock_transition` command is not executed
- * With or without `set_propagated_clock` command specification
(it is better to calculate clock line analysis by this command at layout. However this command should not specified by logic synthesis. It should be executed just before `write_script`)
- * With or without `set_clock_latency -source` option usage
(or delay value setting with clock source)

5.2.9. fanout, capacitance, transition

- | | |
|---|-------------|
| [1] Specify <code>set_max_transition</code> to circuit on the whole if <code>max_capacitance</code> constraint is not in library cell | mandatory |
| [2] Set <code>set_max_fanout</code> or <code>set_max_capacitance</code> for the input port in the event of large level analysis | recommend 1 |

Points

Currently, most ASIC vendors in Japan have `max_capacitance` constraints for each library cell. Some libraries with technology (0.35um or older) do not have `set_max_transition` set. For such cases, use this command.^[1]

```
set_max_transition 2.8 find(design,"")
```

Here, the value (transition time) is usually about 2.2-3ns for 0.35um technology after layout.

5.2. *dc_shell* advanced commands

With 0.18 or 0.13 um technology, the post-layout delay tends to be worse for nets with a larger transition time. Moreover, there are cases when an unexpected delay will be added because of the cross talk effect. The transition time for each net is constrained by `max_capacitance` specified for the output of the library cells, but there are cases where you will want to add stricter constraints. For such cases as well, you can add `set_max_transition` for the entire circuit.

```
set_max_transition 0.45 find(design,"*")
```

Here, the values should be 0.35 - 0.9ns for 0.18 um technology and 0.20-0.38ns for 0.13 um technology. However it is hard to determine intuitively what should be the appropriate value.

```
report_timing -transition_time -capacitance -net
```

You can use this command to generate a report and search for nets with 4-8 fanouts, and estimate from that value.

`set_max_transition` is a command that very hard to set. In its place, `set_max_fanout` can be specified for the entire circuit. By reducing the number of fanouts, the transition time will also be reduced. Note in addition that layout results for nets with large number of fanouts tend to be worse, even if the transition time is shorter.

```
set_max_fanout 9 find(design,"*")
```

For 0.18 or 0.13 um technology, a value from 8 to 11 will be an appropriate value for fanout limits.

Of course, fanout constraints and transition constraints are not the same. Some designers specify both commands.

Regarding the input port, the limitation of fanout and capacitance is automatically added when the `set_driving_cell` command is executed. In a large-scale synthesis, it is recommended that the input port receive with 1 cell, whose capacity is small, from the standpoint of the upper level. In this case,

```
set_max_fanout 1 all_inputs()
```

or

```
set_max_capacitance load_of(.../iv/a) all_inputs()
```

should be specified.^[2]

5.2.10. Variable of Design Compiler better to be specified

```

[1] hlo_resource_allocation = area_only
[2] compile_implementation_selection = false
[3] edifout_netlist_only = true
[4] vhdlout_use_packages = { "IEEE.std_logic_1164" } (VHDL only)
[5] verilout_no_tri = true (Verilog only)
[6] verilout_single_bit = true (Verilog only)
[7] hdlin_check_no_latch = true
[8] hdlin_enable_vpp = true
[9] hdlin_use_cin = true
[10] hlo_transform_constant_multiplication = true
[11] hdl_keep_license = false
[12] suppress_errors = {message number , message number, ...}

```

Explanation

```
hlo_resource_allocation = area_only
```

Design Compiler has a function called resource sharing. As explained in "2.10.5. Do not share resources in critical speed circuits", it is better not to depend on this function from the stage of RTL description.

However, if you are using a part of it (or if the area decreases when resource sharing is executed, notwithstanding the description), change this variable to `area_only`.^[1] Default is constant. In this case, it will be optimized for speed. Resource sharing in the speed direction is very hazardous.

```
compile_implementation_selection = false
```

Since 1998.02, the selection of ripple carry and carry look ahead has been performed at incremental optimization. Due to this function, compile is performed again in a large-scale synthesis in the event of `compile -incremental` at the upper level without performing `ungroup` after compile at the lower level.

In the optimization in the lower level, there is no problem as long as `ungroup` is performed. By setting this variable to false, recompile at incremental optimization can be cancelled. In addition, it is possible to reduce synthesis time and prevent performance problems, but depending on the case the synthesis results may be affected.

```
edifout_netlist_only = true
```

Unless this variable is set to true, drawing circuit information is included in that edif

data, so this leads to an enormous output of edif data.^[3]

```
vhdlout_use_packages = { "IEEE.std_logic_1164" }
```

For this variable, describe `std_logic_1164` and the library name that has the component name offered by vendor.^[4]

```
verilogout_no_tri = true  
verilogout_single_bit = true
```

Unless this variable is set to true, gate level description of Verilog-HDL may cause errors at a subsequent stage in some ASIC vendors.^{[5] [6]}

`verilogout_single_bit=true` break vector signal into 1 bit. An error may occur with tools in subsequent stages by specifying this variable that ASIC vendor's instructions should be followed for variable specification.

```
hdlin_check_no_latch = true
```

This outputs a warning message when a latch is generated (the warning is not output with default) ^[7]

```
hdlin_enable_vpp = true
```

This supports `'ifdef` on Verilog.^[8]

```
hdlin_use_cin = true
```

This uses the operator carry-in pin actively^[9]

```
hlo_transform_constant_multiplication = true
```

This does not generate a multiplier but rather a combinational logic circuit when multiplying a constant with a variable. Consequently, decreased area and improved speed are achieved.^[10]

However, resource sharing for multiplication of `A * B` and multiplication of `A * 6` is no longer performed. Normally, this type of resource sharing is not recommended and therefore this variable should always be true.

```
hdl_keep_license = false
```

After reading the HDL description, separate the license of the VHDL compiler or the HDL compiler_for_verilog.^[11]

```
suppress_errors = {message number, message number, ...}
```

Eliminate unnecessary messages by this variable so as not to overlook necessary warning messages.^[12] Numbers are given to all the error messages and warnings. (EQN-10, UID-615 etc.) Specify these numbers.

5.3. Basic principles of Synthesis

[1]	Logic synthesis should be performed in the bottom-up manner from the basic block	recommend 2
[2]	Ungroup all the levels under the basic block	reference
[3]	Logic synthesis should be performed in logic area first	recommend 1
[4]	Expanding logic area using too many high drive cells is not good for layout	reference
[5]	The library needs wire area specification. Specify <code>dont_use</code> for cells with 6 inputs or more	reference
[6]	The drive strength should be adjusted on the layout side	reference
[7]	For up to 200 thousand gates, hierarchical synthesis should be performed by <code>compile -inc</code>	recommend 1
[8]	In case of chip level optimization, results should be checked for paths related to the input port or the output port	mandatory

Explanation

Basically logic synthesis should be performed in a bottom-up manner. However, it will become verbose if it is performed from enormously small blocks in a large-scale design. In addition, in the case of extremely small levels, the structure is not similar to the concept of the basic block (combinational logic input, FF output), so it is difficult to provide logic synthesis constraints. Therefore, in order to perform logic synthesis, first circuit design should be performed based on the concept of the basic block, and then synthesized from the basic block in a bottom-up manner.^[1]

Regarding only the results of logic synthesis, it is better to ungroup small levels.^[2] In addition, the principle is to ungroup sub-levels of the basic block and below and the levels of the arithmetical operator generated by Design Compiler (DW01_add...).

However, there is also the view that it is not advisable to ungroup small levels since the layout tools have the function to draw cells with same name close. (If compile is performed again after ungrouping, the name of the cell may be switched with the name of another cell or a cell may be inserted in the middle). In this case, the appropriate operation, such as execution of `remove_unconnected_ports` for the level or the setting of variable `compile_implementation_selection = false`, should be done. At the logic synthesis stage, it is recommended not to ungroup the basic block and above.

When optimization is either in its initial stages or at lower levels, we recommend giving

The appropriate view of optimization: Area aware optimization

priority to the area when executing optimization.^[3] This is because in the case of speed problems, the designer can quickly judge what the cause for the timing delay is just by looking at the timing analysis results. However, the designer cannot judge what caused the area to increase by just looking at a report.

For instance, let us assume that the results for the first optimization were 10,000 gates,

but that the speed at this time was only -0.5 ns away from meeting the value of 10 ns. Optimization that is robust in speed optimization is then performed so that the speed can meet 10 ns. Let us assume that the speed value was satisfied as a result, becoming 0.0 ns. However, the area increased, becoming 14,000 gates. In this case, which results will have superior speed after layout, these results or those after the previous optimization, is not known. However, if layout is done, it will be easier to reduce the speed of the one with the larger area. Design Compiler may increase the circuit area approximately 50-60 % with only a slight improvement in speed.

If attention is paid only to the speed from the beginning of optimization, then low quality ISIs with larger area may be produced.

Therefore, it is recommended to first

Synthesize in area

and then gradually improve the speed. Logic synthesis is not something that is optimized just once. Optimization is performed at least three or four times, so it is not necessary to meet completely the speed constraints from the first optimization.

To decrease the area, the method introduced in "5.4.1.Area optimization" whereby synthesis is performed without constraints being applied is effective, but this cannot improve speed so much. To synthesize in speed, a timing constraint is added but attention should be paid so that you do not specify the clock cycle in too much detail.

Area increase by logic optimization is caused by using many cells with a high drive capacity.^[4] Recently it has become possible to increase the drive capacity automatically on the layout side by tools such as Pbopt. As the design rule is 0.25um or less these days, the wire delay is unpredictable before layout. The drive capacity should be adjusted with the layout tools.^[6]

However, since the layouts tool replace them with buffers with high drive capacity after setting allocation, it cannot replace that many. We recommend performing this at logic optimization to a certain extent (stop logic optimization before the area dramatically increases) and leave the rest to the layout tools.

As explained above, the synthesis results should be evaluated in terms of area and not just speed. When you perform synthesis, prepare a table as shown below in order to understand the trends for area and speed before finalizing the entire design. By using a table like this, the designer can track how the area changes to check that it has not increased too much. In addition, checking the delay value enables modification of the RTL code to increase speed.

Cycle 10ns, Setupskew 0.5ns

No	Date	Block name	Sub block name	Gate	Wire	in-ff	ff-ff	ff-out	in-out	Notes
1	01.9.22	geomet	g_tex	8955	7384	4.45	8.96	FF output	None	
2	01.9.24		g_float	15654	10852	7.95	8.96	FF output	None	
3	01.9.27		g_clip	7298	11862	8.82	10.19	FF output	7.82	
4	01.9.25		g_mappi	12560	7103	6.54	8.13	3.52	None	
5	01.8.27	rendarin	r_fil	13810	6954	3.92	8.96	FF output	None	
6	01.9.27		r_map	4315	2140	5.01	7.52	FF output	None	

Logic synthesis result check list

Design Compiler became even stronger for optimizing speed in the 1998.02 version. In addition, not only was the maximum delay value is the target of optimization, but the concept of TNS (Total Negative Slack) was also introduced. Give due consideration so that the delay values of the subsequent slowest paths do not increase. As a result, the area is more susceptible to increasing.

When considering optimization in area, in addition to the gate area, the wire area is an important element. The wire area after actual layout differs from the value of the virtual wire model specified in the library. Some have raised questions about the effectiveness of specification of the wire area in the library.

However, by specifying the wire area, at least the wire area is recognized by a designer, and this is likely to be effective. When considering the wire area after layout, multiple input cells gather many cells to one place, which is not desirable.

By specifying the wire area, usage of this kind of cell can be decreased but this is far from perfect. If you are using a library with 5 or more inputs, it is recommended that you not use cells with many inputs by specifying dont_use.^[5]

Input and output timing

Attention should be paid especially to the delay value setting of I/O and delay value confirmation in chip level optimization and to timing analysis. When sending and receiving data synchronously with outside of chip, data may not be sent or received properly if correct assumption is not made for the delay value of outside, at IO buffer and between IO buffer and FF. Especially precise assumption of the delay value is preferred for the outside LSI and appropriate margin should be set.

In logic synthesis, the delay value of clock line is usually not taken into account for timing analysis. Actual clock line however, create some levels of buffer as tree state that delay occurs. How to deal with CTS delay value is also an issue.

The delay of clock line synchronize with clock input from outside by PLL or DLL. In this case, the timing during logic synthesis and the timing of actual behavior match. Even in this case however, during timing analysis after layout, the delay value of clock line is analyzed that it is necessary to perform addition or subtraction of those to input delay value, output delay value

When input and output timing constraint is strict, sometimes arrival time of clock is changed only for input and output FF. Please pay extra attention for this kind of timing analysis.

5.4. Script examples

5.4.1. Area optimization

[1] **set_structure -boolean true** and **-area-effort high** are effective in the area optimization

recommend 3

[2] In the case of area optimization, compile without timing constraints, then execute incremental optimization

recommend 2

[3] It is better for logic optimization to start from area optimization

recommend 2

Explanation

The Synopsys synthesis tool is very strong in speed optimization. However, it is not good at decreasing area. Certain scripts should be used in order to reduce the area.

The **-boolean** option is very effective at reducing area. This command makes it possible to eliminate circuit redundancy and decrease area.^[1] It is especially effective when the description style is not adequate. In order to optimize in the area, optimization should be executed once without adding any timing constraints at all.^[2] The area can be reduced when this restriction is completely eliminated since equation optimization is executed (structurize) while adding a restriction with the timing taken into account.^[3] With all versions from 1998.02 release or later, the area reduction effectiveness of **-boolean true** unfortunately seem to be reduced. With version 1999.05 or later, this command may increase area and therefore may not be effective for every case.

-area_effort high is a command for area reduction available since version 1999.05. This command is used with incremental synthesis. This command can reduce area for some 5% of the designs but is not effective for 80% of the designs.

From 1999.05, area optimization is not performed unless **set_max_area 0** is specified. Please specify **set_max_area 0**. From 1999.05 however, logic synthesis sometimes take long time when **set_max_area 0** is specified. This tendency becomes high when **-area_effort high** is also set. Please pay attention to run time.

Area Optimization Script Example

```
sh date
Design_name = mansell
file_name   = "../rtl/" + Design_name + ".v"
gate_name   = "../gate/" + Design_name + ".v"
tim_name     = "../log/" + Design_name + ".tim"
ara_name     = "../log/" + Design_name + ".ara"
read -f verilog file_name
current_design Design_name
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
set_wire_load_model -name asc_5000area find(design,Design_name)
set_structure -boolean true
compile
(continue)
```

```
(continuation)
create_clock -p 9 -w { 0 4.5 } -name CLK CLK
set_input_delay 1 -clock clk all_inputs()
set_output_delay 4 -clock clk all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_area 0
set_max_fanout 10 find(design,"*")
derive_clocks()
ungroup -all -flatten
compile -incremental -area_effort high
check_design
report_timing -max_path 2 > tim_name
report_timing -path end -max_path 100 >> tim_name
report_constraint -verbose > ara_name
report_reference >> ara_name
write -f verilog -hier -o gate_name
quit
```

5.4.2. Speed optimization

- | | |
|--|-------------|
| [1] set_flatten true -ef high is effective for state machine descriptions and large <i>case statement</i> description (however, only up to about 2,000 gates) (See 5.2.3) | recommend 2 |
| [2] The easiest method is compile -inc -map_ef high | recommend 2 |
| [3] Effective in speed optimization when cells other than the power cell are set to dont_use | reference |
| [4] If there is a problem in a particular path, group_path command is effective (See 5.2.5.) | reference |
| [5] Beware of the area increasing too much when performing speed optimization | mandatory |

Example Code

set_flatten usage example

```
sh date
Design_name = mansell
file_name = "../rtl/" + Design_name + ".v"
gate_name = "../gate/" + Design_name + ".v"
tim_name = "../log/" + Design_name + ".tim"
ara_name = "../log/" + Design_name + ".ara"
read -f verilog file_name
current_design Design_name
(continue)
```

```
(continuation)
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
set_wire_load_model -name asc_5000area find(design,Design_name)
set_max_area 0
create_clock -p 9 -w { 0 4.5 } -name CLK CLK
set_input_delay 1 -clock clk all_inputs()
set_output_delay 4 -clock clk all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_area 0
set_max_fanout 10 find(design,"*")
derive_clocks()

/* set_structure -boolean true */
set_flatten true -ph true -effort high

compile
ungroup -all -fla
compile -inc -map_effort high
check_design
report_timing -max_path 2 > tim_name
report_timing -path end -max_path 100 >> tim_name
report_constraint -verbose > ara_name
report_reference >> ara_name
write -f verilog -hier -o gate_name
quit
```

Note: `set_structure -boolean true` is not normally specified, but there are situations in which the results may be better if it is specified.

The `flatten` method is effective in the optimization of lower levels when the circuit is not too large. It is especially effective for state machine description and large *case statement* description. If the area becomes too large when this command is specified, even if the speed has improved, pay attention to the method for increasing area since the results occasionally become worse after layout.

Simple two optimizations

```
sh date
Design_name = mansell
file_name   = "../rtl/" + Design_name + ".v"
gate_name   = "../gate/" + Design_name + ".v"
tim_name    = "../log/" + Design_name + ".tim"
ara_name    = "../log/" + Design_name + ".ara"
read -f verilog file_name
current_design Design_name
(continue)
```

```
(continuation)
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
set_wire_load_model -name asc_5000area find(design,Design_name)
set_max_area 0
create_clock -p 9 -w { 0 4.5 } -name CLK CLK
set_input_delay 1 -clock clk all_inputs()
set_output_delay 4 -clock clk all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_area 0
set_max_fanout 10 find(design,"*")
derive_clocks()

compile
compile

ungroup -all -flatten
compile -inc
check_design
report_timing -max_path 2 > tim_name
report_timing -path end -max_path 100 >> tim_name
report_constraint -verbose > ara_name
report_reference >> ara_name
write -f verilog -hier -o gate_name
quit
```

Sometimes the results improve by just repeating the *compile* command twice. This is because TDS (the timing driven structure) operates when structuring is done during the initial optimization, but optimization is performed without proper timing constraints being provided since no actual circuit has been optimized yet. This is because the circuit will meet constraints to some extent after optimization is executed once, so structuring functions with more realistic timing after the second optimization.

This method is especially effective when the compilation subject has small sub-levels , or for levels with a large number of arithmetic operations or relational operations.

-incremental -map_ef high

```
sh date
Design_name = mansell
file_name = "../rtl/" + Design_name + ".v"
gate_name = "../gate/" + Design_name + ".v"
tim_name = "../log/" + Design_name + ".tim"
ara_name = "../log/" + Design_name + ".ara"
read -f verilog file_name
(continue)
```

```

(continuation)
current_design  Design_name
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
set_wire_load_model -name asc_5000area find(design,Design_name)
set_max_area 0
create_clock -p 9 -w { 0 4.5 } -name CLK CLK
set_input_delay 1 -clock clk all_inputs()
set_output_delay 4 -clock clk all_outputs()
set_driving_cell -cell NIIL -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
set_load load_of (asc018_MAX/NIIL/A) * 3 all_outputs()
set_max_area 0
set_max_fanout 10 find(design,"*")
derive_clocks()

compile
ungroup -all -flatten
compile -incremental -map_ef high

check_design
report_timing -max_path 2 > tim_name
report_timing -path end -max_path 100 >> tim_name
report_constraint -verbose > ara_name
report_reference >> ara_name
write -f verilog -hier -o gate_name
quit

```

`-map_ef high` automatically groups critical paths that violate timing constraints, then executes optimization that is strong in area. If precise timing information is not provided, `-map_ef high` will not function properly, so it is used in second order and subsequent optimization. In addition, this option may be effective during gate level optimization, so it is usually used together with `-incremental`.

To quickly optimize the speed, this method is the simplest and is the most effective method. We would recommend that this method be used first. (It is most effective when synthesizing levels.)

`ungroup -all -flatten` ungroups lower levels, but it is important to note that collapsing levels is not always necessarily desirable.

Floor planning will become more difficult when laying out the circuit if the basic blocks are ungrouped when optimizing at the level above the basic blocks. Do not ungroup levels in such situations. Levels are normally ungrouped for sub-blocks under the basic blocks or levels of operators that are generated by optimization. However, there are cases in which it is better to keep the levels for the convenience of the layout. The `ungroup` command should not be executed in such situations.

See "5.4.5. Hierarchical optimization and its concept" for more information.

Ungroup Optimization

```

sh date
Design_name = mansell
read -f verilog file_name
current_design Design_name
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
set_wire_load_model -name asc_5000area find(design,Design_name)
set_max_area 0
create_clock -p 9 -w { 0 4.5 } -name CLK CLK
set_input_delay 1 -clock clk all_inputs()
set_output_delay 4 -clock clk all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_area 0
set_max_fanout 10 find(design,"*")
derive_clocks()

ungroup -all -flatten ← Ungroup before compile

compile
compile -incremental -map_ef high
write -f verilog -hier -o gate_name
quit

```

Normally, the synthesis commands would be:

```

compile
ungroup -all -flatten
compile -incremental

```

However it sometimes improves the results if ungroup is done before initial compile. Synthesis at the logic equation level (structure, flatten) is executed for each hierarchical unit. When a circuit is not well structured, logic equation level synthesis may become inefficient. For such cases, logic equation level synthesis can be performed for the entire circuit by ungrouping before compile. In fact, almost 20% of designs could benefit both in area and speed from this command. However, please note that if you can improve the area or speed with this command, the design may not be adequately structured.

In logic synthesis at chip level or timing analysis, check the result of I/O pins. For the interface with the outside of chip, those with FF input and without can be found by logging timing report.

5.4.3. Circuit synthesis consisting of only combinatorial circuit

- | | |
|--|-------------|
| [1] Do not use <code>set_max_delay</code> | mandatory |
| [2] Use a virtual clock by specifying <code>create_clock -name</code> | mandatory |
| [3] If <code>flatten</code> is not used, then do not execute optimization of less than 1,000 gates | recommend 2 |

Example Code

Combinatorial circuit synthesis

```

sh date
Design_name = mansell
file_name   = "../rtl/" + Design_name + ".v"
gate_name   = "../gate/" + Design_name + ".v"
tim_name    = "../log/" + Design_name + ".tim"
ara_name    = "../log/" + Design_name + ".ara"
read -f verilog file_name
current_design Design_name
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
set_wire_load_model -name asc_5000area find(design,Design_name)

create_clock -p 9 -w { 0 4.5 } -name CLK
                        ↑
                Specify virtual clock without specifying clock port

set_input_delay 1 -clock clk all_inputs()
set_output_delay 4 -clock clk all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST*")))
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_area 0
derive_clocks()

compile

check_design
report_timing -max_path 2 > tim_name
report_timing -path end -max_path 100 >> tim_name
report_constraint -verbose > ara_name
report_reference >> ara_name
write -f verilog -hier -o gate_name
quit

```

Sub-levels that are only constructed from combinatorial logic are generated by virtual clocks. Do not use the `set_max_delay` command since one path group will be created for each command. (See 5.2.5. Group path command (`group_path`) for more information.) Satisfactory results cannot be obtained even if optimization is executed in levels that are too small irrespective of combinatorial logic. This is because it is difficult to provide appropriate delay constraints and drive capacity to the sub-levels. We recommend optimizing starting from about 5,000 gates. However, it is better to specify `set_flatten` for levels that have large *case statements*, so we recommend optimizing from small units.

5.4.4. Multiple clock optimization

- [1] Be careful when setting clocks with different phases
- [2] Be careful when setting clocks that are asynchronous

recommend 2

recommend 2

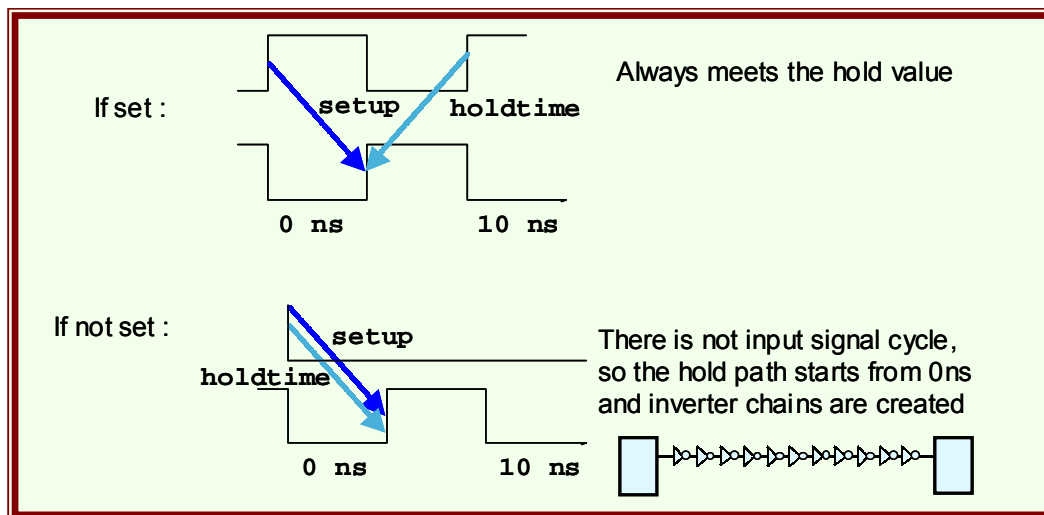
Explanation

When there is only one clock, set the clock rise time to 0 to make the timing report easier to examine (add this value and display when 1000 ps is specified) and to prevent unnecessary trouble. If there are two clocks with different phases, it is impossible to set 0 for both of them.

```
create_clock -p 10 -w {0 5} CLKM
create_clock -p 10 -w {5 10} CLKS
```

Be careful of the `set_input_delay` and `set_output_delay` settings in this situation. The first thing to remember is that you must not forget to specify `-clock`. If the `-clock` option is missing, problems will occur when trying to guarantee the hold time.

If these commands are set properly, then there will be no problem since the hold time will always be met as shown in the upper portion of the following figure due to 5 ns being analyzed from the following clock edge (at 10 ns) in the forward direction. If the `-clock` option is missing, however, the ports that set the `set_input_delay` command will start hold time guarantee analysis from 0 ns since the cycle has not been specified. When the hold time guarantee is executed in this situation, inverter chains will be generated.

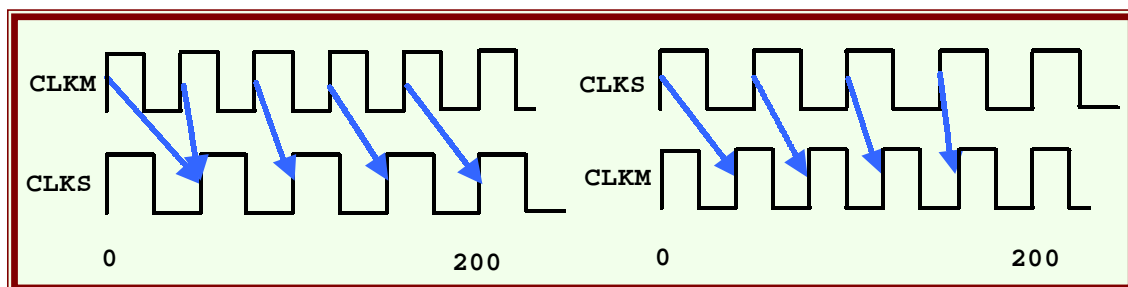


Moreover, if there are clocks with more than two phases, then a clock setup that is dependent on CLKM starts from 0 and clocks that start from CLKS start from 5 ns. If the clock names are set incorrectly, then the delay value may differ from the constraint value. Note that paths that originally should have been optimized may not be optimized or may be optimized too much.

It is necessary to pay careful attention when clocks with different cycles exist asynchronously with each other. For example, when two cycles are set to 4 ns and 5 ns respectively,

```
create_clock -p 40 -w {0 20} CLKM
create_clock -p 50 -w {0 25} CLKS
```

from a human viewpoint, it is easily understood that these two clocks are asynchronous. However, the Design Compiler judges these two clocks to be synchronous if nothing is specified.



When these two clocks are judged to be synchronous, the lowest common multiple of the two cycles (4 ns and 5 ns) is calculated, so the cycle is set at 20 ns. Timing analysis is then performed on the paths as illustrated above. Design Compiler then chooses the path that is constrained to the shortest length.

As shown in the above illustration, the shortest timing path increases the logic area between these two clocks since they are constrained by a very short time. Either that, or the number of timing analyses targeted will increase a factor of five. There is no problem with such a five-fold increase, but if the provided cycle is a more haphazard number (difficult to derive the lowest common multiple), then a large number of timing analyses will be required and there may even be cases in which the analysis is not completed.

Using `set_false_path` is one method to disable analysis of different paths and thereby avoid this problem.

Script example using asynchronous clock `false_path`

```
read -f verilog file_name
current_design Design_name
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
set_wire_load_model -name asc_5000area find(design,Design_name)

create_clock -p 40 -w { 0 20 } -name CLKM CLKM
create_clock -p 50 -w { 0 25 } -name CLKS CLKS

set_false_path -from find(clock,CLKM) -to find(clock,CLKS)
set_false_path -from find(clock,CLKS) -to find(clock,CLKM)

set_input_delay 1 -clock CLKM all_inputs() - CLKS - CLKM
set_output_delay 5 -clock CLKS all_outputs()

set_driving_cell -cell NI1L -pin X all_inputs() - CLKS - CLKM \
                - RSTS_X - RSTM_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_area 0
derive_clocks()

compile
```

Stop mutual analysis

With this script, `false_path` is specified twice since there are two clocks. However, if there were three clocks, it would be necessary to specify `false_path` 9 times, and it would be necessary to specify `false_path` 100 times if there were 10 clocks.

This script disables mutual analyses, but there are actually circuits in which logic exists between these paths. In this case, timing analysis is not performed on this portion and no report is issued either.

If there is logic between the asynchronous clocks, you can change the clock cycles in a pseudo manner, so it would probably be better to make it a multiple (2*, 3 *, 4*, 10*) of the master clock. If a multiple clock is specified, the timing analysis during this period is performed according to the master clock cycle.

Script Example of multiple asynchronous clock

When clock cycle is 4ns and 8ns.

```
read -f verilog file_name
current_design Design_name
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
set_wire_load_model -name asc_5000area find(design,Design_name)

create_clock -p 4 -w { 0 2 } -name CLKM CLKM
create_clock -p 8 -w { 0 4 } -name CLKS CLKS

set_input_delay 1 -clock CLKM all_inputs() - CLKM - CLKS
set_output_delay 2 -clock CLKS all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLKM - CLKS
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_area 0
derive_clocks()

compile
```

When there are either multiple clocks or clocks with different phases, hold time guarantee analysis may not be performed properly even if the maximum delay optimization is performed properly. Normal hold time guarantee analysis starts one clock after the timing point of maximum delay analysis. Use "5.2.7.Set_multicycle_path" to change this position.

5.4.5. Hierarchical optimization and its concept

[1] Start optimization from basic blocks (2,000 – 20,000)	recommend 1
[2] Perform sub-block optimization only when special optimization such as flatten is required	recommend 1
[3] Either ungroup or process unconnected ports in sub-blocks or DW01 levels using ungroup or remove_unconnected_ports	recommend 1
[4] In hierarchical optimization, use compile -incremental (up to 200,000 gates)	mandatory
[5] Perform partitioned optimization for 200,000 gates or more (do not use characterize)	recommend 1

Example Code

Hierarchical synthesis standard script

```

sh date
Design_name = mansell
file_name   = "../rtl/" + Design_name + ".v"
gate_name   = "../gate/" + Design_name + ".v"
rep_name    = "../log/" + Design_name + ".rep"

read basic block

read -f verilog file_name
current_design Design_name
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
set_wire_load_model -name asc_20000 -max find(design,Design_name)
set_wire_load_model -name asc_5000  -max find(design,"Sub_A")
set_wire_load_model -name asc_5000  -max find(design,"Sub_B")
set_wire_load_model -name asc_20000 -min find(design,"*")

create_clock -p 9 -w { 0 4.5 } -name CLK CLK
set_input_delay 1 -clock CLK all_inputs()
set_output_delay 4 -clock CLK all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_fanout 10 find(design,"*")
set_max_area 0
derive_clocks()

compile -incremental -map_effort high

(continue)

```

5.4. Script examples

```
(continuous)
check_design
rep2 = rep_name + "_1"
rep3 = rep_name + "_2"
sh cp rep2 rep3
sh cp rep_name rep2
report_timing -max_path 4 -net > rep_name
report_timing -path end -max_path 100 >> rep_name
report_constraint -verbose >> rep_name
report_reference >> rep_name
write -f verilog -hier -o gate_name
quit
```

Bottom-up compilation is the basic logic optimization method.^[1] However, performing optimization from excessively small units is verbose. Basic blocks normally start from units such as those in "1.6.1.Consider limitations based on hierarchical scale". See "5.1.1.Command script example" and "5.4.1.Area optimization" for more information on optimization in basic blocks.

Logic synthesis optimizes by reading the lower level data and then optimizing only the gate level using

```
compile -incremental
```

Design Compiler cannot produce good results when optimization is performed in excessively large units. Always synthesize the levels above a certain level after basic block optimization is complete. However, Design Compiler optimization efficiency worsens and the time required for optimization gets longer when there are more than 200,000 gates. Execute logic optimization on levels with less than 200,000 gates.^[4] Circuits any larger than that should be optimized using a partitioning method.^[5]

Load a 100,000-gate circuit into the Design Compiler, and then perform only timing analysis. Characterization should not be normally performed at this stage. If a problem occurs with timing, etc., then either the synthesis constraints of each block or the RTL description itself will have to be reviewed.

For optimization of less than 100,000 gates, try to obtain good results using this hierarchical synthesis. If good results cannot be obtained by hierarchical synthesis, then *characterize* is used on the basic blocks under that particular level. *characterize* provides the precise delay information, drive capacity, etc. relating to a certain block. Therefore, using *characterize* makes it possible to synthesize low basic blocks more precisely.

However, *characterize* is also a command that is difficult to use. It is best to try to manage without having to use this command, if possible.

Script example when basic block has sub-blocks

```

sh date
Design_name = mansell
file_name   = "../rtl/" + Design_name + ".v"
gate_name   = "../gate/" + Design_name + ".v"
rep_name    = "../log/" + Design_name + ".rep"

read call lower level

read -f verilog file_name
current_design Design_name
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
create_clock -p 9 -w { 0 4.5 } -name CLK CLK
set_input_delay 1 -clock CLK all_inputs()
set_output_delay 4 -clock CLK all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_fanout 10 find(design,Design_name)
set_max_area 0
derive_clocks()

set_dont_touch Instance name of lower level
compile

remove_attribute Instance name of lower level dont_use
compile -incremental -map_effort high

```

Compile from sub-blocks is not normally performed. This is because basic blocks use combinatorial circuit input and FF output and delay values can easily be provided to I/O ports using `all_inputs()`, `all_outputs()`. However, sub-levels are not structured to have combinatorial circuit input and FF output, so delay values cannot be easily provided. There is also the approach wherein delay values are carefully set for each port, but this is very complex. Carefully setting delay values would yield the best synthesis results, but synthesis would have to be executed in detailed units several times in order to calculate the optimum delay value (not the precise delay value, but the value that improves the synthesis results the most).

If compile is performed from sub-blocks without making such detailed settings, good results cannot be obtained even though someone has gone through the trouble of synthesizing from sub-blocks.

However, there is a case where it is possible to improve synthesis results without setting delay values to detailed I/O in synthesis from sub-blocks. This is when the sub-blocks are described by *case* statements and the descriptions are highly redundant. In such cases,

`set_flatten true` will be efficient, but this command is applied only to small blocks due to time limitations.

We would recommend synthesizing from sub-blocks when specifying `flatten` to improve the synthesis results.^[2]

When starting optimization from sub-blocks, as in the above script, optimization of the upper basic blocks first executes `dont_touch` on those blocks with gates that have already been optimized and then removes them from being subject to optimization. This is because the results may actually worsen if Boolean equation optimization is performed on circuits that already have logic gates.

If optimization is to be performed from basic blocks, then execute it using the scripts described in "5.4.1. Area optimization" and "5.4.2. Speed optimization". Either script will ungroup lower levels using

```
ungroup -all -flatten
```

The optimization results will improve since areas fixed by the unused I/O ports will disappear when the lower levels are ungrouped.

Moreover, note that ASIC design rules may be violated depending on the vendor used when there are unconnected ports in the lower levels. Adders, subtracters, multipliers and other devices automatically generated by Design Compiler also create levels, but there will also be unconnected ports in these levels. In this case, the problem is resolved by collapsing the levels using the `ungroup` command.

In most cases, it would be better to ungroup levels that are too small, but there are also situations in which it would be best occasionally to leave levels in the layout. In such situations, executing

```
remove_unconnected_ports -blast_buses instance name
```

can eliminate unnecessary ports.^[3]

Using this command with `compile -boundary_optimization` processes not just unconnected ports, but also processes ports that merely pass through a block or items in which the same input is input by two ports.

Constraint Example

```
compile -bo

remove_unconnected_ports -blast_buses
filter(find(cell,""),"@is_hierarchical==true")
```

5.5. *characterize* optimization

[1] Base on hierarchical synthesis, and apply if the constraints are not met	mandatory
[2] If the speed constraint is exceeded by more than 10%, do not perform <i>characterize</i> optimization	mandatory
[3] Eliminate unconnected ports when executing <i>characterize</i>	recommend 1
[4] Do not execute <i>characterize</i> on circuits that include latches	recommend 1

Explanation

When resynthesizing lower levels, it will be impossible to provide precise delay values, drive strength, and capacity to all ports if default scripts are used. *characterize* can be used to optimize more precisely.^[1] As shown in the following script, *characterize* is executed from the upper level after synthesizing the upper level. The line

```
filter(find(cell, "**"), "@is_hierarchical == true" )
```

in the script means that the command will be executed on all designs that are not primitive gates.

After executing *characterize* optimization, first save the constraints (the input delay values, output delay values, drive strength, capacity, clock cycle, etc.) provided to that design to a file. If execution is to continue without exiting logic synthesis, it is acceptable to move levels using the *current_design* command.

Let us assume that the saved script files are read when synthesizing the lower levels. After reading the script file, add some commands that appear to obtain good results and execute them.

Script Example

```
compile -incremental -map_effort high

check_design
characterize filter(find(cell, "**"), "@is_hierarchical == true")

foreach(Sub_design,filter(find(reference,"**"),"@is_hierarchical==true"))
{
    current_design find(design,Sub_design)
    write_script > "../scr/" + foo + ".wscr"
}

Create constraint file with the script of upper level and use it for the optimization of lower level.

sh date
Design_name = mansell
file_name    = "../rtl/" + Design_name + ".v"
gate_name    = "../gate/" + Design_name + ".v"
(continue)
```



```

(continuous)
rep_name      = "../log/" + Design_name + ".rep"
chara_name    = "../scr/" + Design_name + ".wscr"
read -f verilog file_name
current_design Design_name

include chara_name

/* group_path flatten etc. executed */
/* according to the circumstances, overwrite the delay value of arbitrary port */

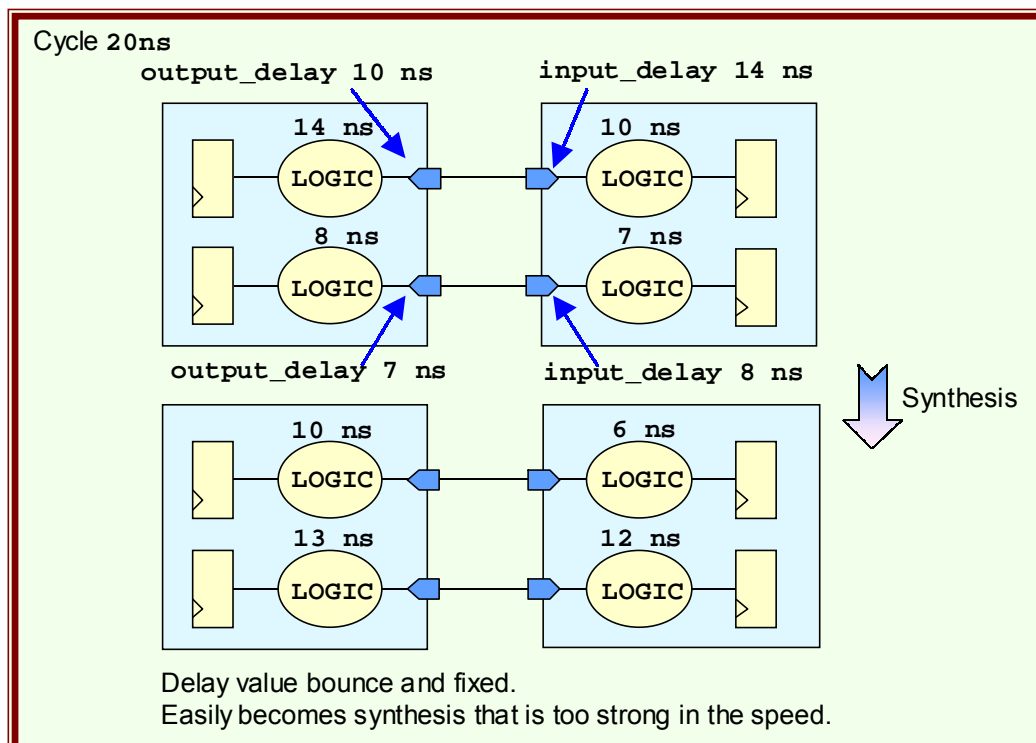
compile

compile -inc -map_effort high

check_design
rep2 = rep_name + "_1"
rep3 = rep_name + "_2"
sh cp rep2 rep3
sh cp rep_name rep2
report_timing -max_path 4 -net > rep_name
report_timing -path end -max_path 100 >> rep_name
report_constraint -verbose >> rep_name
report_reference >> rep_name
write -f verilog -hier -o gate_name
quit

```

characterize is a very good command for obtaining precise values, but be careful when using this command.



Let us assume that the cycle is 20 ns. The `set_output_delay` value obtained by *characterize* is the delay value until it reaches the circuit. The timing path in the top part of the above figure is 14 ns at the left side circuit, so the value specified by `set_output_delay` is 14 ns. By contrast, the `set_output_delay` value is the delay value that is applied to the logic passage of the right side (10 ns).

If optimization is performed in this situation, the delay for the right circuit becomes 6 ns, and the left circuit becomes 10 ns. The cycle is currently 20 ns, but the above path ends up being shortened as far down as to 16 ns. When *characterize* is used in this manner, further shortening the delay value may cause the area to increase. In addition, the lower paths, which are 8 ns and 7 ns (totaling 15 ns) have easily met 20 ns cycles, may be 13 ns + 12 ns = 25 ns, thereby exceeding the cycle.

Due to such effects, even if attempts are made to shorten the delay values by calculating them using *characterize*, new paths will violate the constraints one after another and the delay values will never be fixed.

Such area increase problems and delay bounding frequently occur when delay values are too large with respect to the provided constraints. Therefore, when using *characterize*, first execute optimization, shorten the delay values until they exceed the target cycle by no more than 7%, then use *characterize*.^[2]

The delay-bounding phenomenon occurs more often as the timing paths span more blocks. This phenomenon also occurs more frequently as the gaps between levels are separated by intermediate points such as 10 ns and 14 ns. The more the provisions in Chapter 1 regarding creating levels are followed, the less influence bounding will have. Executing *characterize* on levels that are solely combinatorial logic input or FF output will reduce the effect of the bounding phenomenon to negligible levels. In other words, if there are levels that are solely combinatorial logic inputs or FF outputs, there is no need to execute *characterize*.

When executing *characterize*, execute it in a state where there are no unconnected ports for each input and output.^[3] Moreover, problems are more likely to occur when there are signals fixed to LOW and HIGH. We recommend eliminating these signals before executing *characterize*.

`characterize` command is used for smaller synthesis jobs rather than large-ones. For example, when you want to run `set_flatten` but it is hard to add timing constraints to the target hierarchical level, combining it with *characterize* may sometimes generate a good result.

A time budget function was added in the 1998.08 release. (See "5.7.4.[2] Time budget". This new function executes *characterize* while taking the bounding phenomenon into account. This function is executed by a product other than the Design Compiler. There are fewer problems using this function than *characterize*, but it would be advisable to exercise caution when executing this new function as well.

Time budget is explained in RMM: 6.2.2.

5.6. Circuit synthesis including operators

- | | |
|--|-------------|
| [1] In order to select properly the <i>rpl</i> type adders and subtractors, execute compile without setting the clock cycle | recommend 1 |
| [2] Do not specify selection of implementation from within an RTL description | mandatory |
| [3] Before executing <code>compile -inc</code> for those circuits including operators, execute either <code>ungroup</code> or <code>remove_unconnected_port</code> | recommend 1 |

Explanation

Design Compiler is not very good at selecting *rpl* or *cla*. There may be cases where *cla* is selected even if the speed is sufficient or where *rpl* is selected although *cla* should be selected to satisfy speed. With Design Compiler, a method is available for specifying *rpl* or *cla* in the VERILOG description.

However, this method is complex and the description is only valid for the Design Compiler, so this method should not be used.^[2] Design Compiler is not good at selecting *rpl* and *cla*, but it is better to have them selected by the logic synthesis constraints whenever possible.

In order to set all adders and subtractors in the circuit to *rpl*, there is a method for optimizing without setting the cycle (without specifying the speed constraints).^[1] If the speed constraints are met by gate level optimization `compile -incremental`, this method will result in the smallest area.

Gate level optimization improved in versions issued on or after the 1998.02 release, so it became easier to obtain good results for the speed. Even if *cla* is not selected from the beginning or if *rpl* is selected and gate level optimization is performed, there is no appreciable difference if the bit width is not large. We recommend this method when using only adders or subtractors less than 11 bits.

However, this method may decrease the speed too much in parts other than operators. If the speed is problematic in parts other than operators, then use the following optimization script example in the speed aware.

Sample script of area optimization for circuit that includes operations

```
sh date
Design_name = mansell
file_name   = "../rtl/" + Design_name + ".v"
gate_name   = "../gate/" + Design_name + ".v"
tim_name    = "../log/" + Design_name + ".tim"
ara_name    = "../log/" + Design_name + ".ara"
read -f verilog file_name
current_design Design_name
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
(continue)
```

```

(continuous)
set_wire_load_model -name asc_5000area find(design,Design_name)

set_structure -boolean true
compile
create_clock -p 9 -w { 0 4.5 } -name CLK CLK
set_input_delay 1 -clock CLK all_inputs()
set_output_delay 4 -clock CLK all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_fanout 10 find(design,Design_name)
set_max_area 0
derive_clocks()
ungroup -all -flatten
compile -incremental

check_design
report_timing -max_path 2 > tim_name
report_timing -path end -max_path 100 >> tim_name
report_constraint -verbose > ara_name
report_reference >> ara_name
write -f verilog -hier -o gate_name
quit

```

In the 1998.02 release it became possible to select *rpl* and *cla* when optimizing gate levels as well. However, caution must be exercised with this function modification. If *rpl* or *cla* is selected again when `compile -inc` is executed, all optimizations made to *rpl* and *cla* up to that point will be lost and will have to be performed again from the beginning.

When synthesizing the top level, reselecting adders that have already been optimized may seriously worsen both the area and speed results. Therefore, when optimizing circuits that include adders, always execute either `ungroup` or `remove_unconnected_ports` when synthesizing that level.^[3] There is no hazard that areas where levels have been collapsed will be selected again when `compile -inc` is executed.

The following script is a sample script in which *rpl* and *cla* are selected and executed in a manner that gives priority to speed. Since *rpl* and *cla* are not selected well with just one optimization, they are compiled three times. Ultimately, `ungroup` is first executed to prevent adverse effects on optimization of the upper levels and then `compile -inc` is executed.

Sample script of circuit that includes operations to select cla at gate level optimization

```

sh date
Design_name = mansell
file_name   = "../rtl/" + Design_name + ".v"
gate_name   = "../gate/" + Design_name + ".v"
tim_name    = "../log/" + Design_name + ".tim"
ara_name    = "../log/" + Design_name + ".ara"
read -f verilog file_name
current_design Design_name
set_operating_conditions -max_library asc018_MAX \
                        -min_library asc018_MIN
set_wire_load_model -name asc_5000area find(design,Design_name)

compile

create_clock -p 9 -w { 0 4.5 } -name CLK CLK
set_input_delay 1 -clock CLK all_inputs()
set_output_delay 4 -clock CLK all_outputs()
set_driving_cell -cell NI1L -pin X all_inputs() - CLK - RST_X
set_ideal_net find(net,all_connected(find(port,"RST_X")))
set_load load_of (asc018_MAX/NI1L/A) * 3 all_outputs()
set_max_area 0
derive_clocks()

compile
compile -incremental
ungroup -all -flatten
compile -incremental

check_design
report_timing -max_path 2 > tim_name
report_timing -path end -max_path 100 >> tim_name
report_constraint -verbose > ara_name
report_reference >> ara_name
write -f verilog -hier -o gate_name
quit

```

5.7. 2001.08 performance and comparison with past versions

5.7.1. Improving performance in execution speed

[1] 2001.08: execution speed improved over 2000.11-SP2

[2] 1999.10-4: execution speed twice or more over 1998.08

[3] 1998.02, 1998.08: problem of increasing optimization time by `set_max_area 0`

Explanation

The performance exhibited by each version of Design Compiler is extremely different. Therefore, the selection of the version will have a significant impact on the results. Bugs are inevitable in software. It is hazardous to use Design Compiler without knowing about the various bugs that have been found for each function enhancement.

In recent tools, hazardous bugs have been found in the 1998.02 (02-2 and later is OK), 1999.05 and 1999.10 (10-5 and later is OK) versions, so attention should be paid before starting to use these versions. The 1998.02-2, 1998.08, 1999.10-4 and 1999.10-5 versions are relatively stable so it is possible to rely on these to a certain extent. Among the year 2000 versions, 2000.05-1 is considered to be relatively stable. However, it is better not to use the Verilog presto parser with this version. Two critical bugs in Verilog have been reported. It is better to upgrade 2000.11 to 2000.11-SP2. Version 2001.08, which was released last fall, has not been used much and therefore has not been thoroughly evaluated.

The run time by Design Compiler had become faster as every version-up until 1999.10-4. With 2000.05 and 2000.11 versions, the run time has improved 1.1 times faster on the average. 2001.08 version however, because specifications of `ungroup` and `ideal_net` have modified that the run time of these part became slower than before. The operating speed changes depending on the target circuit and constraint. Especially in design with many operators, the latest version shows the shortest run time of resource sharing that the overall run time becomes fast.

From 1999.10, run time sometimes become slow if area decrease was performed by specifying `set_max_area 0`. In case of 200 thousand gates optimization, if specifying `set_max_area 0`, it may become 4 times slower. In the area optimization, it is recommended to perform it at the optimization for the small size circuit.

Starting from the 1999.05 version, the Ultra version of Design Compiler appeared along with the Expert version. This tool displays its greatest strength in data path optimization, and is not suitable for a circuit with large random logic. Normal Design Compiler (Expert) is not suitable for data path optimization.

Ultra synthesizes the circuit, where one signal such as an enable signal is supplied to 200 to 300 points, whereas Expert is not effective at generating tree structures, which are apt for this type of net.

5.7.2. Improving speed performance

- [1] 2001.08: some area reduction compared to 2000.11-SP2
- [2] 2001.08: faster operating speed compared to 2000.11-SP2
- [3] 1999.10-4: fastest operating speed on the average
- [4] 2001.08: best operating speed for circuits with large number of operators
- [5] 1999.10-4 : circuit area tend to increase

Explanation

	area	speed
2001.08	41072	5.32ns
2000.11-SP2	37953	5.76ns
2000.05-1	38018	5.76ns
1999.10-4	44586	5.08ns

Comparison of synthesis results according to Design Compiler version of a circuit

	area	speed
1999.10-5	1175	18.66ns
1998.08	1193	18.61ns
1997.08	921	21.32ns
V3.5a	1062	20.18ns
V3.4a	1079	21.37ns
V3.3a	1042	21.42ns
V3.2b	1085	21.59ns
V3.1b	1028	21.42ns
V3.0c	905	21.67ns

Performance comparison between old versions

The synthesis result of Design Compiler differs considerably for each version. The result shown above is a result from one particular circuit. The logic synthesis result differs to a large degree according to circuit and constraint. Logic synthesis of 50 circuits were compared that showed similar result as above.

With version 2001.08, area increases compared to 2000.05 on the average. However, area is smaller than 1999.10-4. With version 2000.05 and 2000.11, speed optimization is weak that 2001.08 shows better result. 1999.10-4 is the version with which the best speed result can easily be obtained, however, its area tends to increase that attention should be paid.

The tendency of the difference in area and operating speed according to Design Compiler version likely to become remarkable when the nest of *if statement* and *for statement* is deep or the logical cone is large in circuit. Also, the circuit written in VHDL shows more difference in the result of logic synthesis due to Design Compiler's version compared to the circuit written in Verilog.

The description with the a large amount of operators have been improved version by version. Result regarding speed and area is better with 2000.05 and 2000.11 than 1999.10-4, and with 2001.08 than 2000.05 and 2000.11.

The difference in synthesis result according to Design Compiler past versions is shown for your reference. Design Compiler has shown improvement in operating design since 1998.02 version. However, area tends to increase compared to past versions. Attention should be paid not to increase area when using current Design Compiler.

5.7.3. Performance degradation by *if statements* and *case statements*

- [1] The area increases when *if statements* and *case statements* are used with versions from 1998.08 onwards.
- [2] Good results for both area and speed are obtained if only Boolean equations are used with versions from 1998.08 onwards.

Explanation

Synth. Method	Described by <i>case statements</i>				Described by Boolean expression	
	Normal area		Use <code>Flatten, boolean</code>		Normal synthesis	
	area	speed	area	speed	area	speed
2001.08	175	3.57ns	181	3.69ns	127	3.57ns
1999.10-5	175	3.57ns	181	3.69ns	127	3.57ns
1997.01	148	3.94ns	127	3.57ns	127	3.57ns

Comparison of synthesis results by description method

If the logic of circuits described by *if statements* and *case statements* is to be synthesized by versions 1999.10-4 or 1998.08, the area will increase in most cases. Especially when a description uses above a certain scale of *case statements*, the difference will be significant. The data in the above table is a comparison of the results when circuits are optimized using only a certain *case statement*. The results are about 15% less compared with version 1997.01 even when synthesized in a normal manner. This phenomenon has not changed since 1998.08.

With versions prior to 1997.01, it was possible to eliminate the redundancy included in the *case statements* by using the *flatten* ("5.2.3. Flatten (set_flatten)" or *boolean* ("5.2.4. Structuring(set_structure)") commands and thereby reducing the area, but using this command does not reduce the area much anymore. With respect to the speed, better results

than those with previous versions are often obtained, but there are also cases in which speed is affected by redundancy that is not eliminated and is thereby reduced. When using versions 1999.10-5 or 1998.08 therefore, observe the following criteria when making the RTL description so as not to increase the area to an extreme degree.

- 1) Create the size of the *case statements* as described in "2.8. case statements".
- 2) Try to have as little redundancy as possible when describing *if statements* and *case statements*.

The area of circuits described by Boolean equations will not increase.

Design only described by Boolean equations		
	Area	Speed
2001.08	55230	6.14ns
2000.11-1	53495	6.33ns
1999.10-5	56174	6.17ns

Synthesis result of a circuit(comparison among newer versions)

Design only described by Boolean equations		
	Area	Speed
1999.10-5	46853	14.32ns
1997.01	52708	15.91ns
V3.5a	53175	15.75ns

Synthesis result of a circuit(comparison among older versions)

The above table compares the results of logic synthesis that was performed on designs that were only described by Boolean equations. The 1999.10-5 results show about a 10% improvement in both area and speed. This is because the gate level optimization in versions 1999.10-5 was very strong and this indicates that this version was not particularly good at optimizing Boolean equations. As was explained in "5.2.3.Flatten (set_flatten)" logic optimization is started from Boolean equations.

Boolean equations change the RTL descriptions or logic gate circuits to expressions such as

```
A = B&C | A&E;
```

and then do things such as change the logical structure at the Boolean equation level or group common logic. After Boolean equation level optimization is complete, mapping to the actual logic cells is performed, resulting in a logic circuit. After this, logic optimization is performed on this logic circuit and individual cells are exchanged in an attempt to improve the speed and area.

Optimization at the Boolean equation stage was changed to a method that hardly functions for area (one that gives priority to speed) in an attempt to improve speed in Design Compiler versions 1999.10-5 and 1998.08. Therefore, the logic (especially *case statements*) described by *if statements* or *case statements* has a circuit area that is substantially worse compared with previous versions.

If circuits are described by Boolean equations, there is virtually no change in the circuits during optimization at the Boolean equation level if the expressions do not include too much redundancy. (This is because the logical structure is more appropriate than when already described by *case statements*.) As a result, there is little increase in area for circuits described using Boolean equations compared with versions prior to 1997.08.

Gate level optimization has improved considerably compared with the 1997.08 version, and the speed has improved at an average of 10%. There is an increase in area attributed to the increased speed, but when optimized to be the same speed, the area will not increase.

It can therefore be said that the latest versions (2001.08 etc.) are more suitable for Boolean equation description than *if statements* and *case statements*. However, it is important to keep in mind that a description not containing *if* or *case statements* will be more difficult to read. Logic optimization performance changes according to the version used. We feel that versions will be released in which the increases in area will be kept in check by *case statements*, etc., so it may be possible to consider the current phenomenon a temporary one.

The area tends to get smaller with the version 1999.10-5 compared to 1998.08. With the versions 2000.05 and 2000.11, area tends to get even smaller but operation speed tends to be slow. The operation speed has been improved for the version 2001.08.

5.7.4. New functions in 1998.03 through 2001.08

- [1] Operating conditions MAX, MIN support
- [2] Time budgeter (`characterize` that takes timing allocation into account)
- [3] Select implementation (cla, rpl) when `compile -inc` is specified
- [4] `TNS` (Total Negative Slack, subsequent delay paths taken into account)
- [5] The new `boolean` optimization function
- [6] Multi-bit cell support
- [7] `set_simple_compile_mode` (1999.05)
- [8] `compile -top` (1999.05)
- [9] `propagate_constraints` (1999.05)
- [10] ACS compile (2000.05)
- [11] `case_analysis` (2000.11)
- [12] clock gating (2000.11)
- [13] `set_isolate_ports` (2001.08)

5.7.4.1. Operating environment MAX, MIN support (useful)

MIN and MAX for the operating environment have been supported from 1998.02.

```
set_operating_conditions -library asc025 -max MAX174 -min MIN065
```

Specifying `-max`, `-min` by the operating environment set constraint makes it possible to analyze the maximum delay value under the worst conditions and the hold time guarantee under the best conditions. This function is very useful, so this is a command that should be used often. See "5.1.4.set_operating_condition constraint (specify operating conditions)" for details.

5.7.4.2. Time budgeter (useful but rarely used)

A time budgeting function was added from the 1998.08 release. This function, which performs `characterize` with the bounding phenomenon taken into account, is executed by a different product than the Design Compiler. The Design Compiler data are saved to a db file, and then `budget_shell` is started. Then inputting the following series of commands serves to obtain a synthesis constraints file.

There are fewer problems than using `characterize`, but this command should still be ex-

ecuted with care. With *budget_shell*, the bounding phenomenon of the delay that was a problem with *characterize* ("5.5.*characterize* optimization") is now quite limited in paths that span two blocks. However, this command is still not complete for paths that span three blocks or more.

In addition, when executing *characterize* or *budget_shell* and partition compiling, the drive capacity cannot easily be adjusted.

For example, before using *budget_shell*, when two circuits are connected to their respective low drive cell, if these are partitioned and synthesized, using the cell with high drive capacity for the one circuit will not affect the optimization of the other circuit. Problems are more likely to occur in the budget of the small circuits. In the case of small circuits, pay the same attention to this command as was paid for the *characterize* command. In addition, since *dc_shell* and the execution program differ, the usage method is very complex. Instead of considering this command as being executed on small levels, consider it as being executed on large designs of from 20,000 to 100,000 gates.

However, it would be ideal to make the circuit structure (see Chapter 1) such that *characterize* and *budget_shell* would not be needed on levels of this size. The significance of using this command in this situation would disappear.

budget_shell Script Example

```
after synthesis on dc_shell,
write -f db -o foo.db

on budget_shell
read_db foo.db
current_design Top_design
link_design
allocate_budgets instance name (design where budget is to be executed)
check_budget -verbose
create_timing_context instance name
write_context instance name -format dcsh -o foo.scr
quit

Then partitioned and synthesized again with dc_shell
read -f db <design name of circuit which is the target of budget_shell>
include foo.scr
.....
compile -inc
report_timing -max_path 20
report_area
```

5.7.4.3. Selecting implementation (*cla*, *rpl*) when *compile -inc* is specified (caution)

It has become possible to select *rpl*, or *cla* during gate level optimization from version 1998.02. However, special attention must be paid to this function modification. If *rpl*, *cla* is selected again when *compile -inc* has been specified, all *rpl* and *cla* selected up to that point will become invalid and will be selected again from the beginning.

When synthesizing the top level, reselecting adders that have been optimized may seriously worsen both the area and speed results. Therefore, when optimizing circuits that contain adders, always execute either the *ungroup* or the *remove_unconnected_ports* commands. There is no danger of reselecting levels that have been ungrouped when executing *compile -inc*.

5.7.4.4. Taking TNS (Total Negative Slug) and subsequent delay paths into account (caution)

A total negative slug function was added from 1998.02 release. Up to version 1997.01, only the delay time of the slowest path within a single timing group (one group specified once by *create_clock*) could be optimized.

From version 1998.02, optimization for attempting to shorten the paths for subsequent paths was also enabled. This is different from the *group path* command, however, so the path is not aggressively optimized. Use the *group path* command when you really wish to optimize the subsequent paths. The TNS function is applied to all optimization. When using *create_clock* to set the cycle, set the cycle while being cognizant of a realistic reduction range. If the cycle is set too short, then the influence of the TNS function may cause an unexpected increase in area.

5.7.4.5. The new *boolean* optimization function (useful but not effective)

```
compile_new_boolean_structure = true
set_structure -boolean true -boolean_effort high
```

From version 1997.01, it is possible to use *boolean* optimization of a new algorithm by the above settings. Using this variable may reduce the area further. However, there are also situations in which the area may increase slightly. According to the results of this command being executed on multiple designs, there is practically no overall change in the area, so this function is not effective enough to become a command that one would use regularly.

5.7.4.6. Multi-bit cell support (unnecessary)

From version 1998.02, it became possible to use multi-bit FF and selectors.
When specifying

```
hdlin_infer_multibit = default_all
```

as many cells that support multi-bits as possible are used in all Verilog-HDL descriptions. However, this option is a hazardous command, so it would be better not to use it unless absolutely necessary. When multi-bits are used, the pre-layout area will decrease, but large cells may be placed in the layout, wiring may be obstructed, and the post-layout area may increase. Excessive use of multi-bit cells will cause wires to be convergent to a single location within a cell, and can potentially increase the area and decrease the operating speed.

There is of course a way to effectively use multi-bits as long as placement that takes the data flow into account is performed in the data path blocks. With the current layout tools however, few placements are made that take the data flow into account, so this is not a preferable situation for the auto-layout tools either. The usage of multi-bit cells is effective for layout tools that can take the data flow into account. Therefore, it would probably be better to have the layout tool side support multi-bits (by merging single-bit cells) in these cases too.

If there is a particular reason for having logic optimization support multi-bits, then it is specified in the RTL description. In this case however, directives that are unique to Synopsys will be used, so this would not be preferable from a design reuse standpoint.

Sample multi-bit supporting RTL description

```
module FFBANK(D,CK,RSTB,Q);
input [7:0] D;
input CK,RSTB;
output [7:0] Q;
reg [7:0] Q;

//synopsys infer_multibit"Q"

always @(posedge CK or negedge RSTB) begin
    if(!RSTB)
        Q <= 0;
    else
        Q <= D;
end
endmodule
```

5.7.4.7. `set_simple_compile_mode` (from 1999.05, updated in 2000.11)

When this constraint is specified, the *compile* execution time will be shorter without much consideration given to speed constraints. The *uniquify* command is automatically executed if this mode is enabled.

```
set_simple_compile_mode true
compile
```

Executing this command will also automatically set resource sharing to *area_only* (area will be given priority, and speed will not be considered). While it will vary according to the design, setting this mode to true may also reduce the area. From 2001.11, a budgeting feature has been provided for synthesis of lower levels, such that it is possible to give some consideration to speed as well.

```
set_simple_compile_mode true -budget
compile
```

However, using *-budget* will increase the area, and the care accorded to speed is inadequate. Therefore, it maybe better not to use budget for the *simple_compile_mode*, and to forget the speed and focus entirely on simple compilation.

5.7.4.8. `compile -top` (from 1999.05)

This *compile* option became available in 1999.05. This command marks for synthesis only those paths that violate speed constraints in the top level. Execution time can be dramatically improved over the *compile -inc* option since the area subject to optimize is decreased. This command is useful for synthesizing levels that are 10,000 or 20,000 gates in size. When this command is executed, optimization of the design rule does not change from normal optimization, so it will be performed on the entire circuit.

Owing to the introduction of this command, it may be better to execute logic synthesis with the following strategy.

Basic block 200 gates to 20,000 gates	<code>compile</code> (initial optimization)
20,000 gates to 200,000 gates above basic blocks	<code>compile -inc</code>
200,000 gates to 1,000,000 gates of top level	<code>compile -top</code>

5.7.4.9. propagate_constraints (1999.05)

This is a command to bring the timing constraint specified at lower level up to top level.

```
current_design SUB
set_false_path -from find(port, "InputA") -to find(pin, "A_reg[0]/D")
```

, which specified to input port at lower level, is switched to -through by this command.

```
current_design TOP
propagate_constraints
-> set_false_path -through find(pin, "USUB/InputA") -to find(pin, "USUB/A_reg/D")
```

This command is very useful when the setting of `set_false_path`, `set_multicycle_path` and `create_clock` in the design of lower level is complicated. It is however risky to bring the constraint given to lower level unnecessarily up to upper level. This is not a command, which you should actively use.

When using this command, reset all the constraints at lower level and then specify only necessary constraints to lower level before executing `propagate_constraint` in upper level.

```
current_design SUB
reset_design

<necessary constraints only>
current_design TOP
propagate_constraints
```

5.7.4.10. ACS compile (2000.05 or later)

This automatically performs hierarchical synthesis using budget. This method relatively suits to synthesize a large scale circuit at high speed. As in the `simple_compile_mode`, it *uniquifies* the lower hierarchy and then executes budget, and subsequently, incremental synthesis is performed at the top level. Unlike the `simple_compile_mode`, it performs resource-sharing and hierarchical synthesis at upper levels, so it produces reasonably good speed results by a simple operation. However, it does not ungroup hierarchy, so the area will be larger compared to manually performed hierarchical synthesis. ACS compile is available only with the Tcl mode.

```
acs_read_hdl ~hasegawa/LMS/rtl      <--Simply specify RTL directory
source TOP.tcl
acs_compile_design TOP               <-- Specify top level
```


In addition, the following modes are available with ACS compile.

```
acs_refine_design      :budget->partition->incremental synthesis
                        for synthesized netlist
acs_recompile_design  :budget -> back to RTL of each level and synthesize again
                        for synthesized netlist
```

If you have multiple Design Compiler licenses and a multi-CPU server or LSF is in use, synthesis can be executed with multiple machines.

ACS compile is not recommended when using inverted clock, 2 phase or 3 phase clock or asynchronous reset.

Because it generate problem when 2 clocks exist as follows.

```
create_clock -p 10 -w {0 5 } CLK
create_clock -p 10 -w {5 10} CLK_X
```

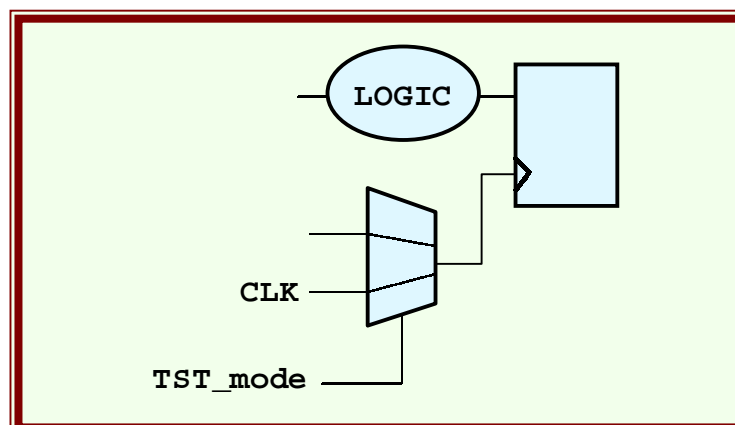
If a signal, which is connected to power or GND, is input at upper level or separate level. can not be judged which clock this input depends on. Therefore, `-clock` option is not added to `set_input_delay`. Consequently, timing of setup and hold may not be analyzed correctly, which is problematic.

5.7.4.11. case_analysis (2000.11 or later)

When a fixed value input such as the TST mode is used, paths that do not propagate can be cut off. This command was available with the PrimeTime timing analysis tool, and is now provided also with Design Compiler.

```
set_case_analysis 0 TST_mode
```

By setting the above, it will fix the selector shown in the following figure, and the inactive path will not be analyzed.

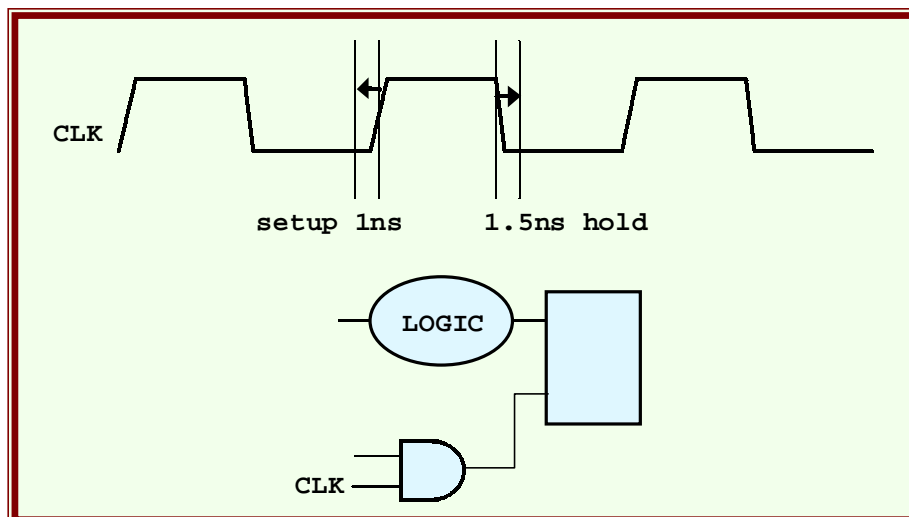


5.7.4.12. clock gating (2000.11)

Analyzes gated clocks

```
set_clock_gating_check -rise -setup 1 -fall -hold 1.5 -high CLK
```

Performs setup/hold analysis for the clocks as shown. With PrimeTime, clock gating is applied automatically for all gated clocks. Designers manually disable the unnecessary ones. With Design Compiler, the analysis will be performed only when above command is specified by the designer.



5.7.4.13. set_isolate_ports (2001.08)

In hard macro or a large size sub design (100 - 500 thousands gates), allocating buffer in output port to separate from inside path or using buffer with high drive capacity is sometimes necessary. In this case, I/O buffer levels can be used as explained in "1.6.6. Designate buffer outputs in upper levels with of 200,000 or more gates".

set_isolate_ports command is available for specification from 2001.08.

```
set_isolate_ports all_outputs()
```

Allocate buffer to output port where path, which makes internal regression, exist. The drive capacity of cell to be allocated changes depending on timing constraint.

```
set_isolate_ports -driver BUF16 -force all_outputs()
```

Allocate designated buffers to all the output ports.

Even with the version 2000.11-SP2 or earlier, setting similar to `set_isolate_ports` command is possible by using a command like `create_cell`.

```
foreach(OutputName, all_outputs()){  
    OutputNet = all_connected(find(port, OutputName))  
    disconnect_net OutputNet find(port, OutputName)  
    create_cell "U" + OutputName asic018/bufx16  
    connect_net find(net, OutputNet) "U" + OutputName + "/a"  
    NewNet = "Net" + OutputName  
    create_net find(net, NewNet)  
    connect_net find(net, NewNet) "U" + OutputName + "/y"  
    connect_net find(net, NewNet) find(port, OutputName)  
    set_dont_touch "U" + OutputName  
}
```

A-1 Index

Items, which appear on the Design Style Guide, are itemized in the alphabetical order to respond to questions and issues regarding design using Verilog-HDL or logic synthesis.

A

always construct

- 2.1.1. Use the *always construct* and *function statements* correctly
(avoiding Sim and synthesis malfunctions, description style)
- 2.2.1. Avoid the risk of generating latches (avoiding synthesis malfunction)
- 2.2.2. Define every input signal in an always construct in the sensitivity list
(avoiding Sim malfunctions)
- 2.3.1. Unify the description style of FF inferences
(avoiding Sim malfunctions, avoiding synthesis malfunctions)
- 2.3.3. Do not mix descriptions that have different edges (avoiding synthesis malfunction)
- 2.6.1. Describe taking the circuit structure into account (speed improvement, area decrease)
- 2.6.2. Avoid defining multiple output signals in a single always construct
(speed improvement, area decrease)
- 2.10.5. Do not share resource in speed critical circuits (area decrease)

area

- 2.6.1. Describe taking the circuit structure into account (area decrease, speed improvement)
- 2.6.2. Avoid defining multiple output signals in a single *always* construct
(area decrease, speed improvement)
- 2.7.1. *if* statements create prioritized circuits (area decrease)
- 2.7.2. Reduce *conditional expressions* of *if statement* with the same contents (area decrease)
- 2.8.2. Divide using if statement, etc. to avoid creating large tables
(area decrease, speed improvement)
- 2.8.6. Beware of nesting that *if* statements and *case* statements coexist
(area decrease, speed improvement)
- 2.10.7. Take share items out of conditional branches
- 5.1.10. The `set_max_area` command (set area constraints)

- 5.1.11. The compile command (logic synthesis and circuit optimization)
(area decrease, synthesis method)
- 5.3. Basic principles of Synthesis
- 5.4.1. Area optimization (area decrease, synthesis method)
- 5.6. Circuit synthesis including operators (area decrease, avoiding synthesis malfunction)
- 5.7.2. Improving speed performance (execution time increases due to decrease)
- 5.7.3. Performance degradation by if statements and case statements
(area increase reason, synthesis method)
- 5.7.4.3. Select implementation (cla, rpl) when "compile -inc" is specified (area increase reason)
- 5.7.4.7. set_simple_compile_mode (1999.05) (area decrease, synthesis method)

A

arithmetic operation circuit

- 2.1.6. Specifying the range of an array
- 2.6.1. Describe taking the circuit structure into account
- 2.8.5. Description relying on parallel_case is prohibited
- 2.9.2. Limiting loop-variable operation in *for statements*
- 2.10.1. Order of operators and assignment of 'x'
- 2.10.5. Do not share resource in speed critical circuits
- 2.10.6. Notes on arithmetic operations
- 3.2.5. The function library of an arithmetic operation improves data path design
performance
- 5.2.2.3. ungroup (synthesis method)
- 5.3. Basic principles of Synthesis

assignment statement

- 2.1.5. Use conditional operator ((A)?B:C) only once (description style)
- 2.3.2. Circuits will vary with non-blocking and blocking assignment statements
(avoiding Sim malfunctions, description style)
- 2.10.3. Match the bit width of the left side and the right side
(avoiding Sim malfunctions, description style)
- 2.10.4. Take note of the different data types between the left and right sides
(avoiding Sim malfunctions, description style)

asynchronous reset

- 1.3.1. Use asynchronous reset for initial reset (circuit structure, avoiding Sim malfunctions)
- 1.3.2. Reset line hazard (circuit structure)
- 1.3.3. Be careful about external noise on an initial reset signal (circuit structure)
- 2.3.1. Unify the description style of the FF inference
- 2.3.6. Do not mix FF inference with asynchronous resets and without (description style)
- 2.4.1. Clearly distinguish a latch inference from a combinational circuit
- 3.3.1. Clocks and Resets for DFT
- 3.3.6. DFT in reset lines
- 5.1.8. The set_driving_cell command (drive strength setting of input port connection)

bit width

- 2.1.3. In a *function statement*, be careful to check arguments and bit width (avoiding Sim malfunctions, description style)
- 2.1.6. Specifying the range of an array (description style)
- 2.10.3. Match the bit width of the left side and the right side (avoiding Sim malfunctions, description style)
- 4.2.3. Pay due attention to task I/O arguments (avoiding Sim malfunction, Sim method)
- 5.7.4.6. Multi-bit cell support (synthesis method)

bug

- 4.3.2. Create test specifications (Sim method)
- 4.3.3. Create a simulation checklist (Sim method)
- 4.3.4. Clarify simulation results (Sim method)
- 4.3.7. Efficient verification using a behavior model (Sim method)
- 4.4.1. Gate level simulation problems (Sim method)

case statement

- 2.8.1. *case* statements facilitate decoder/encoder description (description style)
- 2.8.2. Divide using *if* statement, etc. to avoid creating large tables (speed improvement, area decrease)
- 2.8.3. Use *default* clauses (speed improvement, area decrease)
- 2.8.4. Do not use complex *casex* statements (avoiding synthesis malfunction, speed improvement, area decrease)
- 2.8.5. Description relying on *parallel_case* is prohibited (avoiding synthesis malfunction, description style)
- 2.8.6. Beware of nesting that *if statements* and *case* statements coexist (speed improvement, area decrease)
- 2.11.3. Separate FF inference and case statements (description style)
- 5.2.3. Flatten (*set_flatten*) (speed improvement, area decrease)
- 5.7.3. Performance degradation by *if* statements and case statements (avoiding synthesis malfunction, area decrease)

circuit (-> circuit segmentation, -> high speed circuit, -> area, -> state machine circuit,

**-> function library, -> combinational circuit, -> arithmetic operation circuit,
-> flip flop(FF), -> latch, -> priority logic, -> encoder/decoder)**

circuit segmentation

- 1.6.1. Consider limitations based on hierarchical scale (speed improvement)
- 1.6.2. Make basic block FF output & combinational circuit input (speed improvement)

- 1.6.3. Follow sub-block (the levels below basic clocks) constraints (speed improvement)
- 1.6.5. Separate the data path section from the controller
(area decrease, speed improvement, synthesis method)
- 2.11.2. Isolate state machine circuits (description style, synthesis method)
- 3.1.3. Unify description order of module I/O ports (description style)
- 5.5. *characterize optimization* (speed improvement, synthesis method)

clock (-> multiple clocks, -> gated clock, -> edge, -> clock tree synthesis(synthesis))

-> Clock tree synthesis

- 1.2.1. Clock synchronous design (circuit structure)
- 1.4.1. Modularizing clock generation circuit (layout, synthesis method)
- 1.4.2. Use clock tree synthesis for clock balancing (layout)
- 1.4.3. Gated clocks should be used with special care (circuit structure, layout)
- 1.4.4. Multiple clock systems (Sim method)
- 1.5.1. Consider meta stable in signals between asynchronous clocks
- 1.5.2. Use memory in transfers between asynchronous same-period clocks
- 3.1.2. Describe a style that takes reuse into account (description style)
- 3.1.3. Unify description order of module I/O ports(DFT)
- 3.3.1. Clocks and Resets for DFT (DFT)
- 3.3.5. DFT in clock lines (DFT)
- 4.1.5. Describe on a clock edge basis (Sim method)
- 4.1.6. Set the cycle using parameters (Sim method)
- 4.1.7. Define a signal for each clock in a multiple clocks design (Sim method)
- 4.1.8. Description where the results do not differ due to the simulators
(avoiding Sim malfunctions)
- 5.1.6. The create_clock command (clock definition) (synthesis method)
- 5.2.1. Setting the hold time guarantee
- 5.4.4. Multiple clock optimization (avoiding synthesis malfunction)

clock tree synthesis(synthesis)

- 1.4.2. Use clock tree synthesis for clock balancing (layout)
- 1.4.3. Gated clocks should be used with special care (circuit structure, layout)
- 3.4.2. Low power-consumption design hints by the logic circuits
- 3.4.3. Low power-consumption design hints by the clock tree
- 5.2.1. Setting the hold time guarantee

combinational circuit

- 1.6.2. Make basic block FF output & combinational circuit input (speed improvement)
- 2.1.1. Use the always construct and function statements correctly
(avoiding Sim malfunctions, description style)
- 2.1.2. Define combinational circuits using the *function* statement (description style)
- 2.6.1. Describe taking the circuit structure into account (speed improvement, area decrease)

- 2.6.2. Avoid defining multiple output signals in a single *always* construct (speed improvement)
- 2.7.1. *if* statements create prioritized circuits (area decrease)
- 2.8.1. *case* statements facilitate decoder/encoder description (speed improvement)
- 2.10.1. Order of operators and assignment of 'x' (speed improvement)
- 2.10.2. Efficient description using logical operation expressions (description style)
- 5.4.3. Circuit synthesis consisting of only combinational circuit (speed improvement, synthesis method)

CPU interface

- 1.4.4. Multiple clock systems
- 4.1.9. Simulation between asynchronous clocks
- 4.3.1. Making verification flow

C

D

D

data path

- 1.6.5. Separate the data path section from the controller (avoiding synthesis malfunction, description style)
- 2.5.1. Make a block for a tri-state buffer (avoiding synthesis malfunction, circuit structure, description style)
- 2.10.1. Order of operators and assignment of 'x' (speed improvement, avoiding synthesis malfunctions, description style)
- 2.10.2. Efficient description using logical operation expressions (speed improvement, description style)
- 2.10.5. Do not share resource in speed critical circuits (speed improvement, area decrease)
- 3.1.1. Create and utilize function libraries (speed improvement, area decrease, design management method)
- 3.2.4. Use # (*value*) when overwriting parameters from an upper level (description style, design management method)
- 3.2.5. The function library of an arithmetic operation improves data path design performance (speed improvement, area decrease,)
- 5.6. Circuit synthesis including operators (avoiding synthesis malfunction, speed improvement, area decrease)
- 5.7.4.6. Multi-bit cell support (synthesis method)
- 5.7.4.7. `set_simple_compile_mode` (1999.05) (synthesis method)

decoded circuit (-> encoder/decoder)

decoder (-> encoder/decoder)

delay

- 1.3.3. Be careful about external noise on an initial reset signal (use delay element)
- 1.4.2. Use clock tree synthesis for clock balancing (wire delay)
- 1.6.6. Designate buffer outputs in upper levels with 200,000 or more gates (big delay)

- 2.3.1. Unify the description style of the FF inference (using delay value)
- 2.11.1. Use Mealy type and Moore type descriptions properly (propagation delay)
- 4.1.3. Note input signal timing (delay value of Sim input pattern)
- 4.1.5. Describe on a clock edge basis (edge relative delay value)
- 4.1.6. Set the cycle using parameters (specifying delay value)
- 4.1.8. Description where the results do not differ due to the simulators (specifying delay value)
- 4.1.9. Simulation between asynchronous clocks (specifying delay value)
- 4.3.1. Making verification flow (delay simulation)
- 4.5.1. Key points of static timing analysis (delay analysis)

design library

- 3.1.1. Create and utilize function libraries (speed improvement, area decrease, design management method)
- 3.2.1. Manage libraries in a common directory (design management method)
- 3.5.1. Create a directory for each objective (design management method)

Design Ware

- 3.2.1. Manage libraries in a common directory
- 5.2.2.3. ungroup
- 5.2.2.4. Hierarchical synthesis (compile -incremental_mapping)
- 5.2.2.6. remove_unconnected_ports
- 5.3. Basic principles of Synthesis
- 5.4.5. Hierarchical optimization and its concept

drive capacity

- 1.4.2. Use clock tree synthesis for clock balancing (circuit structure)
- 1.6.6. Designate buffer outputs in upper levels with 200,000 or more gates (avoiding synthesis malfunction)
- 5.1.8. The set_driving_cell command (drive strength setting of input port connection) (avoiding synthesis malfunction)
- 5.1.9. set_load command (set output port load) (avoiding synthesis malfunction)
- 5.5. *characterize optimization (synthesis method)*
- 5.7.4.2. Time budgeter (characterize that takes timing allocation into account) (synthesis method)

E

edge (Ref: -> event, -> sensitivity list)

- 1.2.1. Clock synchronous design
- 1.4.3. Gated clocks should be used with special care
- 1.5.3. Guaranteeing the setup/hold margin for synchronous RAM
- 2.3.1. Unify the description style of the FF inference
- 2.3.3. Do not mix descriptions that have different edges
- 4.1.3. Note input signal timing
- 4.1.5. Describe on a clock edge basis

- 4.1.8. Description where the results do not differ due to the simulators
- 4.1.9. Simulation between asynchronous clocks
- 5.1.6. The `create_clock` command (clock definition)
- 5.4.4. Multiple clock optimization

encoded circuit (-> encoder/decoder)

encoder/decoder

- 2.1.4. Instructions for equation level descriptions (speed improvement, description style)
- 2.8.1. *case* statements facilitate decoder/encoder description (description style)
- 2.8.3. Use *default* clauses (description style, area decrease)
- 2.8.4. Do not use complex *casex* statements (description style)

event (Ref: -> edge, -> sensitivity list)

- 2.2.2. Define every input signal in an *always* construct in the sensitivity list (description to avoid Sim malfunctions)
- 2.3.1. Unify the description style of FF inferences (avoiding Sim malfunctions)
- 2.6.2. Avoid defining multiple output signals in a single *always* construct (description style)
- 4.1.4. Avoid assigning from multiple initial constructs (avoiding Sim malfunctions)
- 4.1.5. Describe on a clock edge basis
- 4.1.10. Use handshakes when the process cycle count is unclear
- 4.1.11. Output simulation results to a file
- 4.2.1. Describe using tasks to provide structure
- 4.3.7. Efficient verification using a behavior model
- 4.3.8. Verification methods for large-scale design

execution speed (simulation)

- 4.3.1. Making verification flow
- 4.3.4. Clarify simulation results
- 4.4.1. Gate level simulation problems

execution speed (synthesis)

- 5.7.2. Improving speed performance (execution speed comparison, reason of synthesis speed increase)
- 5.7.4.7. `set_simple_compile_mode` (1999.05)(synthesis execution speed, area decrease)
- 5.7.4.8. `compile -top` (1999.05) (method of execution speed decrease at level synthesis)

expectation value verification

- 4.3.1. Making verification flow (avoiding Sim malfunctions, method)
- 4.3.6. Simulating while comparing with the expected values (Sim method)

F

failure (-> DFT)

file management

- 3.5.2. File suffix names (design management method)
- 3.5.5. Periodically back up files (design management method)

flip flop (FF)

- 1.3.1. Use asynchronous reset for initial reset (circuit structure, description style)
- 2.1.1. Use the *always* construct and function statements correctly (avoiding synthesis malfunction, description style)
- 2.1.2. Define combinational circuits using the *function* statement (avoiding synthesis malfunction)
- 2.2.1. Avoid the risk of generating latches (avoiding synthesis malfunction, description style)
- 2.2.2. Define every input signal in an *always* construct in the sensitivity list (avoiding Sim malfunctions)
- 2.3.1. Unify the description style of the FF inference (avoiding Sim malfunctions, description style)
- 2.3.2. Circuits will vary with non-blocking and blocking assignment statements (avoiding synthesis and Sim malfunctions)
- 2.3.3. Do not mix descriptions that have different edges (avoiding synthesis malfunction, description style)
- 2.3.4. Do not specify an initial FF value in a description (avoiding synthesis malfunction, description style)
- 2.3.5. Do not describe to generate FFs having fixed input values
- 2.4.1. Clearly distinguish a latch inference from a combinational circuit (avoiding synthesis malfunction, description style)
- 5.7.4.6. Multi-bit cell support (circuit structure, description style)

for statement

- 2.9.1. Do not use *for statement* for other than simple repeating statements (description style)
- 2.9.2. Limiting loop-variable operation in *for statements* (description style)

function library

- 2.1.4. Instructions for equation level descriptions
- 2.9.2. Limiting loop-variable operation in *for statements*
- 2.10.6. Notes on arithmetic operations
- 3.1. Create function libraries
- 3.2. Using function libraries

function statement

- 2.1.1. Use the *always construct* and *function statements* correctly (avoiding Sim malfunctions, description style)
- 2.1.2. Define combinational circuits using the *function statement* (avoiding Sim malfunctions, description style)
- 2.1.3. In a *function statement*, be careful to check arguments and bit width (avoiding Sim malfunctions, description style)

gate level simulation

- 1.4.3. Gated clocks should be used with special care (avoiding Sim malfunctions)
- 2.1.1. Use the always construct and function statements correctly (avoiding Sim malfunctions)
- 2.2.2. Define every input signal in an always construct in the sensitivity list (avoiding Sim malfunctions)
- 2.2.3. Initial value description in always constructs (description style)
- 2.3.1. Unify the description style of the FF inference (avoiding Sim malfunctions, description style)
- 4.1.8. Description where the results do not differ due to the simulators (avoiding Simmalfunctions)
- 4.4.1. Gate level simulation problems (design method)
- 4.4.2. Inconsistencies can occur between RTL and at gate level with the propagation of X (avoiding Sim malfunctions)
- 4.4.3. Beware of malfunctions caused by the timing (avoiding Sim malfunctions)

gated clock

- 1.4.1. Modularizing clock generation circuit (circuit structure)
- 1.4.3. Gated clocks should be used with special care (avoiding Sim malfunctions)
- 2.3.1. Unify the description style of the FF inference (avoiding Sim malfunctions)
- 3.3.5. DFT in clock lines (DFT)
- 3.4.1. Low power consumption design using gated clocks (circuit structure)
- 3.4.2. Low power-consumption design hints by the logic circuits
- 5.4.4. Multiple clock optimization (avoiding synthesis malfunction)

handshake

- 4.1.1. Hierarchizing the test bench (Sim method)
- 4.1.10. Use handshakes when the process cycle count is unclear (Sim method)

high speed circuit

- 2.1.4. Instructions for equation level descriptions (speed improvement)
- 2.6.1. Describe taking the circuit structure into account (speed improvement)
- 2.8.1. *case* statements facilitate decoder/encoder description (speed improvement)
- 2.8.6. Beware of nesting that *if* statements and *case* statements coexist (speed improvement)
- 2.10.5. Do not share resource in speed critical circuits (speed improvement, area decrease)
- 3.2.5. The function library of an arithmetic operation improves data path design performance
- 5.1.11. The compile command (logic synthesis and circuit optimization) (speed improvement)
- 5.2.3. Flatten (set_flatten) (speed improvement)
- 5.2.5. The group_path command (group_path) (speed improvement)

- 5.4.2. Speed optimization (synthesis method, speed improvement)
- 5.5. *characterize optimization* (synthesis method, speed improvement)
- 5.7.2. Improving speed performance
(synthesis performance comparison, speed improvement)

hold time guarantee

- 1.4.2. Use clock tree synthesis for clock balancing (circuit structure)
- 1.4.3. Gated clocks should be used with special care (circuit structure)
- 1.4.4. Multiple clock systems (circuit structure)
- 3.4.1. Low power consumption design using gated clocks
- 5.2.1. Setting the hold time guarantee
- 5.4.4. Multiple clock optimization
- 5.7.4.1. Operating conditions MAX, MIN support

I

if statement

- 2.1.4. Instructions for equation level descriptions
(reasons of speed decrease and area increase)
- 2.2.3. Initial value description in *always constructs*
(avoiding Sim malfunctions, description style)
- 2.3.6. Do not mix FF inference with asynchronous resets and without
(avoiding Sim malfunctions, description style)
- 2.4.1. Clearly distinguish a latch inference from a combinational circuit (description style)
- 2.7.1. *if statements* create prioritized circuits
(reasons of area decrease and speed decrease, description style)
- 2.7.2. Reduce conditional expressions of *if statement* with the same contents
(speed improvement, area decrease, description style)
- 2.7.3. Decrease the number of *if statement* nests (speed improvement, description style)
- 2.7.4. Always surround multiple statements using *block statements* (begin-end)
(avoiding Sim malfunctions, description style)
- 2.8.1. *case statements* facilitate decoder/encoder description
(speed improvement, area decrease, description style)
- 2.8.2. Divide using *if statement*, etc. to avoid creating large tables
(speed improvement, area decrease, description style)
- 2.8.6. Beware of nesting that if statements and case statements coexist
(avoiding Sim malfunctions, description style)
- 2.10.2. Efficient description using logical operation expressions
(Sim efficiency improvement, description style)
- 2.10.7. Take share items out of conditional branches
(avoiding Sim malfunctions, description style)
- 3.1.4. Consider RTL description readability (description style)
- 5.7.3. Performance degradation by if statements and case statements (area increase reason)

initial reset

- 1.3.1. Use asynchronous reset for initial reset

- 1.3.2. Reset line hazard
- 1.3.3. Be careful about external noise on an initial reset signal
- 1.5.2. Use memory in transfers between asynchronous same-period clocks
- 2.3.4. Do not specify an initial FF value in a description
- 3.3.1. Clocks and Resets for DFT (DFT)
- 3.3.6. DFT in reset lines
- 4.4.2. Inconsistencies can occur between RTL and at gate level with the propagation of X

interface (-> CPU interface)

L

latch

- 2.1.1. Use the always construct and function statements correctly
(avoiding Sim and synthesis malfunctions)
- 2.2.1. Avoid the risk of generating latches (avoiding synthesis malfunction)
- 2.4.1. Clearly distinguish a latch inference from a combinational circuit
(avoiding synthesis malfunction)
- 3.3.4. Cautions when using latches

layer design

- 1.6.2. Make basic block FF output & combinational circuit input (speed improvement)
- 1.6.4. Do not insert gates into upper levels of basic blocks
(speed improvement, avoiding synthesis malfunction)
- 1.6.6. Designate buffer outputs in upper levels with 200,000 or more gates
(speed improvement)
- 2.4.1. Clearly distinguish a latch inference from a combinational circuit
(avoiding synthesis malfunction, description style)
- 2.5.1. Make a block for a tri-state buffer (avoiding synthesis malfunction)
- 2.5.2. Consider high-impedance propagation in tri-state bus (avoiding Sim malfunctions)
- 2.10.3. Match the bit width of the left side and the right side
(operator is hierarchized by logical synthesis tool) (logical synthesis)
- 3.1.3. Unify description order of module I/O ports (description style)
- 3.1.5. Parameterize the array range of module I/O (description style)
- 3.2.1. Manage libraries in a common directory (design management method)
- 3.2.3. Connect ports by name for component instantiations (avoiding Sim malfunctions)
- 3.2.4. Use # (value) when overwriting parameters from an upper level (description style)
- 5.2.2. Hierarchical Synthesis (uniquify, set_dont_touch, ungroup, compile -inc)
(speed improvement, area decrease)
- 5.2.2.3. ungroup
- 5.2.2.4. Hierarchical synthesis (compile -incremental_mapping)
- 5.4.5. Hierarchical optimization and its concept (speed improvement, area decrease)
- 5.7.4.8. compile -top (1999.05) (execution time decrease)

library(-> function library, -> logic synthesis library, -> Design Ware,
-> simulation library)

logic synthesis (-> logic synthesis library, -> flip flop(FF), -> latch,
-> Synopsys directive, -> area , -> drive capacity)

logic synthesis library

- 1.6.6. Designate buffer outputs in upper levels with 200,000 or more gates
(circuit structure, layout)
- 3.2.1. Manage libraries in a common directory
- 3.2.5. The function library of an arithmetic operation improves data path design
performance
- 4.4.1. Gate level simulation problems
- 5.1.1. Command script example
- 5.1.4. set_operating_conditions command (specify operating conditions)
- 5.1.5. set_wire_load_model command (wire load model specification)
- 5.1.12.3. report_reference
- 5.2.1. Setting the hold time guarantee
- 5.2.9. fanout, capacitance, transition
- 5.3. Basic principles of Synthesis

Logical operation

- 2.1.4. Instructions for equation level descriptions (description style)
- 2.1.5. Use conditional operator ((A)?B:C) only once (description style)
- 2.10.3. Match the bit width of the left side and the right side (description style)
- 2.10.4. Take note of the different data types between the left and right sides
(description style)
- 3.1.5. Parameterize the array range of module I/O (speed improvement, area decrease)
- 3.2.5. The function library of an arithmetic operation improves data path design
performance (speed improvement)
- 5.7.3. Performance degradation by if statements and case statements
(speed improvement, area decrease, synthesis method)

M

management (-> parameterization, -> file management, -> revision management)

meta stable

- 1.4.4 Multiple clock systems
- 1.5.1. Consider meta stable in signals between asynchronous clocks (circuit structure)

multiple bits (-> bit width)

multiple clocks

- 1.2.1. Clock synchronous design (circuit structure)
- 1.4.1. Modularizing clock generation circuit (circuit structure, description style)
- 1.4.4. Multiple clock systems (circuit structure)
- 1.5.1. Consider meta stable in signals between asynchronous clocks (circuit structure)
- 1.5.2. Use memory in transfers between asynchronous same-period clocks (circuit structure)
- 4.1.7. Define a signal for each clock in a multiple clocks design
(avoiding Sim malfunctions, circuit structure, description style)
- 5.4.4. Multiple clock optimization (avoiding synthesis malfunction, synthesis method)

N

naming conventions

- 1.1.1. Basic naming conventions (description style)
- 1.1.2. Naming conventions of circuit and pin names should be considered by the hierarchy
- 1.1.3. Give meaningful names for signals (description style)
- 1.1.4. Naming conventions of *include* file, *parameter* and *define* (description style)
- 1.1.5. Give register output names that suggest clocks or registers (description style)
- 3.1.3. Unify description order of module I/O ports (avoiding Sim malfunctions, description style)

noise

- 1.3.2. Reset line hazard (circuit structure)
- 1.3.3. Be careful about external noise on an initial reset signal (circuit structure)
- 1.5.1. Consider meta stable in signals between asynchronous clocks (circuit structure)

O

operator

- 1.6.5. Separate the data path section from the controller
- 2.9.2. Limiting loop-variable operation in *for statements*
- 2.10.1. Order of operators and assignment of 'x' (speed improvement, description style)
- 2.10.2. Efficient description using logical operation expressions (description style)
- 2.10.3. Match the bit width of the left side and the right side
(avoiding Sim and synthesis malfunctions)
- 2.10.4. Take note of the different data types between the left and right sides
(avoiding Sim malfunctions)
- 2.10.5. Do not share resource in speed critical circuits (area decrease, speed improvement)
- 2.10.6. Notes on arithmetic operations (area decrease, speed improvement)
- 3.2.5. The function library of an arithmetic operation improves data path design
performance (area decrease, speed improvement,)
- 5.6. Circuit synthesis including operators (area decrease, speed improvement)
- 5.7.4.3. Select implementation (cla, rpl) when "compile -inc" is specified
(avoiding synthesis malfunction)

parameterization

- 1.1.4. Naming conventions of *include file*, *parameter* and *define*
- 3.1.1. Create and utilize function libraries (description style)
- 3.1.2. Describe a style that takes reuse into account (description style)
- 3.1.5. Parameterize the array range of module I/O
(avoiding Sim malfunctions, description style)
- 3.2.2. Define global parameters in separate files (Sim method, description style)
- 3.2.4. Use # (value) when overwriting parameters from an upper level
(Sim method, description style)

power consumption lowering

- 1.4.3. Gated clocks should be used with special care (circuit structure)
- 3.4.1. Low power consumption design using gated clocks (circuit structure)
- 3.4.2. Low power-consumption design hints by the logic circuits (circuit structure)
- 3.4.3. Low power-consumption design hints by the clock tree

priority logic

- 2.6.2. Avoid defining multiple output signals in a single *always* construct
- 2.7.1. *if* statements create prioritized circuits
- 2.7.2. Reduce conditional expressions of *if statement* with the same contents
- 2.7.3. Decrease the number of *if* statement nests
- 2.8.1. *case* statements facilitate decoder/encoder description
- 2.8.3. Use *default* clauses
- 2.8.4. Do not use complex *casex* statements
- 4.4.2. Inconsistencies can occur between RTL and at gate level with the propagation of X

racing issue

- 1.4.3. Gated clocks should be used with special care (avoiding Sim malfunctions)
- 2.2.3. Initial value description in *always* constructs
- 2.3.1. Unify the description style of the FF inference
- 4.1.3. Note input signal timing (avoiding Sim malfunctions)
- 4.1.4. Avoid assigning from multiple initial constructs (avoiding Sim malfunctions)
- 4.1.8. Description where the results do not differ due to the simulators (avoiding Sim malfunctions)
- 4.4.1. Gate level simulation problems (avoiding Sim malfunctions)
- 4.4.3. Beware of malfunctions caused by the timing

RAM

- 1.5.2. Use memory in transfers between asynchronous same-period clocks
- 1.5.3. Guaranteeing the setup/hold margin for synchronous RAM

readability

- 1.1.2. Naming conventions of circuit and pin names should be considered by the hierarchy

- 1.1.3. Give meaningful names for signals
- 2.1.4. Instructions for equation level descriptions
- 3.1.1. Create and utilize function libraries
(speed improvement, area decrease, design management method)
- 3.1.2. Describe a style that takes reuse into account (design management method)
- 3.1.3. Unify description order of module I/O ports
- 3.1.4. Consider RTL description readability (description style)
- 3.5.3. Define necessary information for file header (design management method)
- 3.5.6. Use comments often (design management method)
- 4.1.1. Hierarchizing the test bench (Sim method)

reset (-> asynchronous reset, -> synchronous reset, -> initial reset)

revision management

- 3.5.1. Create a directory for each objective (design management method)
- 3.5.3. Define necessary information for file header (design management method)
- 3.5.4. Managing the version of a file (design management method)
- 3.5.7. Using CVS when managing the versions (design management method)
- 3.5.8. Using CVS basic commands *checkout* and *commit*
- 3.5.9. Managing versions using CVS (as example)
- 3.5.10. Check modified contents using the CVS history

S

scan circuit

- 3.3. Design for Test (DFT)

sensitivity list (Ref: -> event , -> edge)

- 2.1.1. Use the always construct and function statements correctly
- 2.2.2. Define every input signal in an always construct in the sensitivity list
- 2.3.1. Unify the description style of the FF inference
- 2.6.1. Describe taking the circuit structure into account
- 2.6.2. Avoid defining multiple output signals in a single *always* construct
- 4.3.8. Verification methods for large-scale design
- 4.4.1. Gate level simulation problems
- 4.4.4. Other causes of mismatches between RTL and gate level

**simulation (-> expectation value verification, -> gate level simulation,
-> execution speed (simulation), -> simulation result mismatch
-> simulation library, -> test bench description,
-> handshake, -> unknown value(X), -> delay, -> task, -> verification method)**

simulation library

- 3.5.1. Create a directory for each objective

4.4.1. Gate level simulation problems

simulation result mismatch

- 1.3.1. Use asynchronous reset for initial reset (avoiding Sim malfunctions)
- 1.4.3. Gated clocks should be used with special care (avoiding Sim malfunctions)
- 2.1.1. Use the always construct and function statements correctly (avoiding Sim malfunctions)
- 2.2.2. Define every input signal in an always construct in the sensitivity list (avoiding Sim malfunctions)
- 2.2.3. Initial value description in always constructs (description style)
- 2.3.1. Unify the description style of the FF inference (avoiding Sim malfunctions, description style)
- 2.5.2. Consider high-impedance propagation in tri-state bus (avoiding Sim malfunctions)
- 4.1.8. Description where the results do not differ due to the simulators (avoiding Sim malfunctions)
- 4.4.1. Gate level simulation problems (design method)
- 4.4.2. Inconsistencies can occur between RTL and at gate level with the propagation of X (avoiding Sim malfunctions)
- 4.4.3. Beware of malfunctions caused by the timing (avoiding Sim malfunctions)
- 4.4.4. Other causes of mismatches between RTL and gate level

source code readability (-> readability)

speed (-> high speed circuit, -> execution speed(synthesis), -> execution speed(simulation))

state machine circuit

- 1.6.5. Separate the data path section from the controller (avoiding synthesis malfunction, description style)
- 2.11.1. Use Mealy type and Moore type descriptions properly (circuit structure, avoiding synthesis malfunction)
- 2.11.2. Isolate state machine circuits (speed improvement, avoiding synthesis malfunction, circuit structure)
- 2.11.3. Separate FF inference and case statements (description style)
- 2.11.4. Consider the state allocation (speed improvement, area decrease, description style)

synchronous reset

- 1.3.1. Use asynchronous reset for initial reset
- 2.2.3. Initial value description in always constructs (description style)
- 2.3.1. Unify the description style of the FF inference
- 4.4.2. Inconsistencies can occur between RTL and at gate level with the propagation of X
- 5.1.8. The set_driving_cell command (drive strength setting of input port connection)

Synopsys directive

- 1.3.1. Use asynchronous reset for initial reset
- 2.8.1. *case* statements facilitate decoder/encoder description

- 2.8.3. Use *default* clauses
- 2.8.5. Description relying on `parallel_case` is prohibited
- 4.3.11. Efficient debugging by embedding descriptions using ``ifdef`
- 5.7.4.6. Multi-bit cell support

synthesis (-> logic synthesis , -> clock tree synthesis(synthesis))

T

task

- 4.1.1. Hierarchizing the test bench (Sim method)
- 4.2.1. Describe using tasks to provide structure (Sim method)
- 4.2.2. Describe function operations using tasks (Sim method)
- 4.2.3. Pay due attention to task I/O arguments (Sim method)
- 4.3.2. Create test specifications (Sim method, design management method)

test bench description

- 4.1.1. Hierarchizing the test bench (Sim method)
- 4.1.2. Use basic test vector descriptions (Sim method)
- 4.2.1. Describe using tasks to provide structure (Sim method)
- 4.2.2. Describe function operations using tasks (Sim method)
- 4.3.1. Making verification flow (Sim method)
- 4.3.2. Create test specifications (Sim method)
- 4.3.3. Create a simulation checklist (Sim method)
- 4.3.5. Compare the expected value files with the simulation results (Sim method)
- 4.3.7. Efficient verification using a behavior model (Sim method)
- 4.3.11. Efficient debugging by embedding descriptions using ``ifdef` (Verilog only) (Sim method)

Test facility

- 1.4.1. Modularizing clock generation circuit (circuit structure, description style)
- 1.4.3. Gated clocks should be used with special care (avoiding Sim malfunctions, circuit structure)
- 1.4.4. Multiple clock systems (circuit structure)
- 2.3.5. Do not describe to generate FFs having fixed input values (description style)
- 3.3.1. Clocks and Resets for DFT
- 3.3.2. Dealing with hard macros and asynchronous circuits
- 3.3.3. Constraints on the use of Flip Flops
- 3.3.4. Cautions when using latches
- 3.3.5. DFT in clock lines
- 3.3.6. DFT in reset lines
- 3.3.7. Handling of different clocks
- 3.3.8. DFT for tri-state circuits
- 3.3.9. Handling hard macros and asynchronous circuits, and test strategies for large-scale ASICs

timing

- 1.5.1. Consider meta stable in signals between asynchronous clocks (circuit structure, avoiding synthesis malfunction, description style)
- 1.6.2. Make basic block FF output & combinational circuit input (circuit structure, avoiding synthesis malfunction)
- 4.4.3. Beware of malfunctions caused by the timing
- 4.5.1. Key points of static timing analysis
- 4.5.2. Circuit design according to static timing analysis
- 5.2.6. set_false_path (avoiding synthesis malfunction, synthesis method)
- 5.2.7. set_multicycle_path (avoiding synthesis malfunction, synthesis method)
- 5.4.2. Speed optimization (speed improvement, synthesis method)

tri-state buffer

- 2.5.1. Make a block for a tri-state buffer (avoiding synthesis malfunction)
- 2.5.2. Consider high-impedance propagation in tri-state bus (avoiding Sim malfunctions)

U**unknown value(X)**

- 1.3.1. Use asynchronous reset for initial reset (avoiding Sim malfunctions, circuit structure, description style)
- 4.4.2. Inconsistencies can occur between RTL and at gate level with the propagation of X (avoiding Sim malfunctions)

V**verification method**

- 4.1.1. Hierarchizing the test bench (Sim method)
- 4.1.3. Note input signal timing (Sim method)
- 4.1.4. Avoid assigning from multiple initial constructs (Sim method)
- 4.1.9. Simulation between asynchronous clocks (Sim method)
- 4.1.11. Output simulation results to a file (Sim method)
- 4.1.12. Use PLI (Sim method)
- 4.3.1. Making verification flow (Sim method)
- 4.3.2. Create test specifications (Sim method)
- 4.3.3. Create a simulation checklist (Sim method)
- 4.3.4. Clarify simulation results
- 4.3.8. Verification methods for large-scale design (Sim method)
- 4.3.9. Separate the function verification patterns and fault detection patterns (Sim method)
- 4.3.10. Verification method using random function (Sim method)
- 4.5.1. key points of static timing analysis (Sim method)
- 4.5.2. Circuit design according to static timing analysis (Sim method)

Verilog-2001

- 3.1.6. Parameter description using *`ifdef*
- 4.4.4 Other causes of mismatches between RTL and gate level
- 5.1.2 read command (reading design)

List of Items by Level

Level	Item No.	Contents
mandatory	1.1.1.2.	Only alphanumeric characters and the underscore '_' should be used, and the first character should be a letter of the alphabet
	1.1.1.3.	Key words in Verilog-HDL(IEEE1364), VHDL(IEEE1076.X) may not be used
	1.1.1.4.	Names containing "NC_0", "NC_1", "_TSB", "VDD ", "VSS", "VCC" or "GND" (uppercase or lowercase) must not be used
	1.1.1.5.	Do not distinguish names by using upper or lower case English letters (Abc, abc)
	1.1.1.10.	Do not use the same instance name or cell name as the ASIC library being used
	1.1.1.11.	All names should be within 40 characters in length
	1.1.2.1.	Module names and instance names should be between 2 and 32 characters in length - The length of 16 or fewer characters is recommended (recommend 2)
	1.1.3.3.	Signal names, port names, parameter names, define names and function names should be between 2 and 40 characters in length - The length of 24 or fewer characters is recommended
	1.1.4.9.	Specify the bit width if it is greater than 32 bits (Verilog only)
	1.2.1.2.	Do not create a RS latch or FF using primitive cells such as AND, OR
	1.2.1.3.	Do not use feedback in combinational circuits
	1.3.1.6.	Do not have both asynchronous reset and synchronous reset on the same reset line
	1.5.1.1.	To avoid meta stable conditions, do not locate logic between asynchronous clocks
	1.5.1.3.	Do not have a feedback loop at the first-stage FF after transfers between asynchronous clocks
	1.5.3.1.	Guarantee the setup/hold margin
	1.6.1.4.	Assign clock generation modules, reset generation modules, RAM, I/O cells, Setup/Hold guarantee buffers, and the top level of RTL description only to the top level
	1.6.4.1.	The upper levels of basic blocks should contain only the connections of each block
	2.1.1.2.	Describe every case of conditions in a <i>function statement</i>
	2.1.2.1.	The function statement should not be used for asynchronous reset line logic in an <i>always construct</i> for FF inference
	2.1.2.2.	A <i>non-blocking assignment</i> (<=) should not be used in <i>function statements</i> (Verilog only)
	2.1.2.3.	All arguments are defined as <i>function statement</i> inputs
	2.1.2.5.	Clock edge descriptions should not be used in task statements (Verilog only)
	2.1.3.1.	Match the argument bit width with the bit width of the function statement input declaration (Verilog only)
	2.1.3.2.	Match the return value bit width with the bit width of the <i>assignment</i> destination signal (Verilog only)

Level	Item No.	Contents
mandatory	2.1.3.5.	In a function statement, global signal assignment should not be performed (Verilog only)
	2.1.6.5.	For an array index, 'x' and 'z' should not be used
	2.2.1.1.	Latches are generated unless all conditions have been described -Care should be taken not to create latches
	2.2.2.1.	Define all the signals, which are in conditional expression and in the right-hand side of <i>assignment statements</i> in the <i>always constructs</i> , in the sensitivity list
	2.2.2.3.	Multiple event expressions should not be described with <i>always constructs</i> (an event expression is necessary for one)
	2.2.3.1.	Do not mix <i>blocking assignments</i> (=) and <i>non-blocking assignments</i> (<=) in combinational <i>always construct</i>
	2.2.3.2.	Do not assign over the same signal using a <i>non-blocking assignment</i> for combinational circuit
	2.2.3.3.	Do not assign over the same signal in an <i>always construct</i> for sequential circuit
	2.3.1.1.	Use <i>non-blocking assignment</i> in FF inferences
	2.3.1.2.	Do not use unsynthesizable FF inference styles
	2.3.1.5.	Specify delay values with integral numbers and do not use minus delay values
	2.3.1.6.	In FF inference with asynchronous reset, pay attention to the negedge or the posedge of the reset signal
	2.3.1.7.	Do not use both asynchronous set and reset
	2.3.2.2.	Do not mix <i>blocking</i> and <i>non-blocking assignments</i> in FF inference <i>always construct</i> (Verilog only)
	2.3.3.1.	Do not use two or more different clock edges within a single <i>always construct</i>
	2.3.3.2.	Do not use two or more identical clock edges within a single <i>always construct</i>
	2.3.4.1.	Do not specify FF initial values explicitly in initial constructs
	2.3.4.2.	Logic synthesis ignores initial constructs, so it should not be used
	2.4.1.4.	Avoid conditional feedback loops that contain latches
	2.5.1.5.	Do not connect 2 or more outputs other than tri-state buffer even under the same conditions
	2.5.1.6.	inout should not directly be connected to input/output
	2.6.2.1.	Do not describe more than one <i>if</i> or <i>case</i> in one <i>always construct</i>
	2.6.2.2.	Signals assigned in <i>always construct</i> should not be described on sensitivity list in the same <i>always construct</i>
	2.7.4.3.	Do not use fork-join in RTL descriptions (Verilog only)
	2.8.1.5.	Do not force <i>full_case</i> for <i>case statement</i> directives that depend on a particular logic synthesis tool (Verilog only)
	2.8.3.5.	Describe a default clause at the end of a <i>case statement</i> (Verilog only)
	2.8.5.3.	Do not describe variables (or the expression: a+b) in the clause of <i>case statement</i> (Verilog only)
	2.9.1.2.	Initial value and conditions of for statement should be constant, and do not change the values within loop variable
	2.9.2.1.	Do not describe any arithmetic operations other than with loop variable and constant

Level	Item No.	Contents
mandatory	2.9.2.4.	Use for loop separately from reset part and logic part
	2.10.1.5.	Do not assign 'x' except for the default clause of a <i>case statement</i>
	2.10.1.6.	Do not use values including 'x' or 'z'
	2.10.3.6.	Specify bit width for constants used in conditional expressions and such (Verilog only)
	2.10.4.1.	Do not use data types other than reg, wire and integer (Verilog only)
	2.10.4.5.	Do not assign negative value to integer
	2.10.6.6.	Do not use division
	3.1.2.1.	Follow the basic naming conventions
	3.1.2.7.	Do not use embedded logic synthesis scripts in the source code
	3.2.3.1.	For component instantiations, connect ports by name, not by ordered list
	3.2.3.2.	Match the bit width of the component port and the bit width of the net to be connected
	3.3.1.1.	The clocks must be directly controllable from external input ports
	3.3.1.2.	When there are two selectable clock systems, one clock system must be selected throughout testing
	3.3.1.4.	The reset for the FFs must be directly controllable from an external input port
	3.3.2.3.	Do not connect the outputs of a black box to clock pins, reset pins, or tristate-enable pins - Prepare the hard macro library
	3.3.3.1.	A clock must not be connected to the D input of a FF
	3.3.4.1.	Do not create feedback loop circuits including latches
	3.3.6.1.	When inputting the output of random logic into an asynchronous set or reset pin, fix to a specific voltage using an AND gate
	3.3.6.2.	Do not mix clock lines and reset lines
	3.3.6.3.	Do not connect the output of a FF directly to the asynchronous set or reset pin of a FF
	3.3.6.4.	Do not connect the output of a latch directly to the asynchronous set or reset pin of a FF
	3.4.1.1.	In the design of standard ASICs, gated clocks can be used only in the top-level
	3.5.1.1.	Save RTL descriptions, logic synthesis scripts, logic synthesis results, and simulation results in different directories
	4.1.3.1.	Shift the timing of <i>assignments</i> to input signal from the clock's rising edge
	4.4.1.1.	Both RTL verification and gate level verification are necessary <Reason> - Timing verification is required (Static timing analysis is currently the standard method) - The mismatch is caused by an `x` problem - Malfunction of an asynchronous block or the problem of racing - Mismatches may occur due to sensitivity list problems (the combinatorial circuit by an <i>always construct</i>)
	4.5.1.1.	Not only gate simulation but also static timing analysis is necessary

Level	Item No.	Contents
mandatory	5.1.4.2.	MIN and MAX can be specified at the same time, but you must not first synthesize with the MAX operating environment and then change it to the MIN operating environment to run the hold time guarantee optimization
	5.1.7.1.	Always specify -clock for set_input_delay and set_output_delay
	5.1.8.1.	Always specify set_driving_cell for basic block optimization except for the chip level
	5.1.8.2.	Use set_ideal_net for asynchronous reset lines
	5.1.12.4.1	Grasp not only the gate area but also the wire area value by the report_area command
	5.2.1.1.	Do not use set_propagated_clock for logic optimization unless it is analysis after layout
	5.2.1.4	Log the timing report by MIN before compile when set_fix_hold is specified
	5.2.2.3.2.	Do not ungroup above basic blocks
	5.2.2.5.1.	Specify after inserting buffers for clock tree synthesis in the top level, etc.
	5.2.9.1.	Specify set_max_transition to circuit on the whole if max_capacitance constraint is not in library cell
	5.3.8.	In case of chip level optimization, results should be checked for paths related to the input port or the output port
	5.4.2.5.	Beware of the area increasing too much when performing speed optimization
	5.4.3.1.	Do not use set_max_delay
	5.4.3.2.	Use a virtual clock by specifying create_clock -name
	5.4.5.4.	In hierarchical optimization, use compile -incremental (up to 200,000 gates)
	5.5.1.	Base on hierarchical synthesis, and apply if the constraints are not met
	5.5.2.	If the speed constraint is exceeded by more than 10%, do not perform characterize optimization
	5.6.2.	Do not specify selection of implementation from within an RTL description

Level	Item No.	Contents
recommend 1	1.1.1.6.	Do not use an '_' (underscore) at the end of the primary port name or module name, and do not use '_' consecutively
	1.1.1.9.	At the top level, module names and port names should consist of 16 or fewer characters and should not be distinguished by upper or lower case English letters
	1.1.4.4.	Use <i>define statements</i> , declared in the same module only (Verilog only)
	1.1.4.6.	Do not propagate parameters through ports
	1.1.4.8.	Clarify <value>'b, 'h, 'd, 'o specification for parameters
	1.2.1.1.	Designs should use a single clock/single edge as much as possible
	1.3.1.3.	Do not use asynchronous set/reset pins for anything other than initial reset

Level	Item No.	Contents
recommend 1	1.3.1.7.	Do not use a FF with both asynchronous set and asynchronous reset
	1.3.2.1.	Do not insert logical operands (AND,OR, XOR) in a reset line at the local level, and create circuits that supply resets as separate modules - Logic order may be replaced by synthesis - Hazards cannot be prevented in the RTL description
	1.3.2.2.	Do not insert signals other than initial reset to FF asynchronous reset pins
	1.4.1.1.	Modularize circuits that supply clocks separately - Put gated clocks and multiplication clocks together in a single level of hierarchy at the top level
	1.4.3.2.	Do not input a FF output pin to other FF clock pins
	1.4.3.4.	Do not supply clock signals to pins other than FF clock input pins (such as D input)
	1.4.3.6.	Do not use FFs with inverted edges
	1.5.1.2.	To avoid meta stable conditions, do not locate logic between FF for data retrieval and the next FF
	1.6.1.2.	Consider hierarchies on a scale of 2,000~10,000 gates to be the standard, and do not place logic gates in the upper levels (except for CTS buffers and I/O cells)
	1.6.2.2.	When the above is impossible, have the timing path cover no more than two blocks
	2.1.2.4.	task statements should not be used (Verilog only)
	2.1.3.4.	A function statement should end with a return value <i>assignment</i>
	2.1.4.5.	Logical operator should not be used for vector (Verilog only)
	2.3.1.4.	Do not use delay values other than in <i>always constructs</i> that infer FFs
	2.3.5.1.	Do not describe to generate FFs having fixed input values
	2.3.6.1.	Do not mix FF inference with asynchronous resets and without in the same <i>always construct</i>
	2.4.1.1.	Clearly distinguish a latch inference from the logic in other combinational circuits
	2.4.1.3.	Do not use latches with an asynchronous set/reset
	2.4.1.5.	Do not use two level latches in the same phase clock
	2.6.1.2.	Use <i>always constructs</i> as single execution units (The number of outputs should be limited to 15 at most)
	2.7.1.3.	<i>if statement</i> in combinational circuit ends with else (not with else if)
	2.7.2.1.	Reduce conditional expressions with the same contents
	2.7.3.3.	The number of nest for if-if and else if should be 10 at most
	2.8.1.3.	Avoid the overlapping of <i>case items</i>
	2.8.1.4.	Always add default clauses
	2.8.3.4.	Do not use the signal to which a don't-care condition is assigned for a conditional expression of <i>if statement</i>
	2.8.3.6.	Do not use the signal to which don't-care condition is assigned for selection expression of <i>case statement</i> that do not assign 'x' in default clause

Level	Item No.	Contents
recommend 1	2.8.3.7.	Do not use the signal, to which don't-care condition is assigned, for selection expression of casex or casez statement
	2.8.4.4.	Do not indiscriminately describe 'x' of casex statement for each bit (Verilog only)
	2.8.5.1.	Do not force parallel_case in a <i>case statement</i> directives that depends on a particular logic synthesis tool (Verilog only)
	2.8.5.2.	Do not describe fixed values in the selection expression of <i>case statement</i> (Verilog only)
	2.9.2.3.	The number of loops range is up to 10 if operating logically or relationally other than with loop variable and constant
	2.10.1.4.	Do not compare with 'x' and 'z'
	2.10.1.7.	Do not use RAM output signals for a conditional expression of <i>if statement</i>
	2.10.1.8.	Do not use RAM output signals for the selection expression of <i>case statement</i> that is not assigned 'x' in default clause
	2.10.2.3.	Do not perform logical negation to vector (Verilog only)
	2.10.3.2.	Match the bit width of <i>assignment</i> signal and operand of logical operator (Verilog only)
	2.10.3.3.	The bit width of the right-hand side <i>assignment</i> should not be wider than the left-hand side <i>assignment</i> (Verilog only)
	2.10.4.3.	Pay attention to bit widths when assigning integer to reg or wire
	2.10.5.3.	Do not describe 3 or more shared arithmetic operations
	2.10.6.2.	Beware of the sign extension and adjust bit widths in signed operations
	2.10.6.5.	Do not infer large multiplier by the RTL description but describe the contents of multiplier by logical operation
	2.10.7.1.	Do not use arithmetic operation in the conditional expression of <i>if statements</i>
	3.1.2.2.	Do not make technology-dependent descriptions
	3.2.2.4.	Make the file names specified by <code>`include</code> into relative paths (<code>../include/common.h</code>) (Verilog only)
	3.2.4.3.	Do not use <i>defparam statements</i> (Verilog only)
	3.3.1.3.	The output of random logic should not be used as a clock
	3.3.2.1.	Consider the use of original test circuits for asynchronous circuits and hard macros
	3.3.4.2.	Put the latches into a through state during testing
	3.3.5.1.	During testing, switch to the same phase any clock lines having different phases
	3.3.5.2.	When the output of a FF is used as a clock, switch using a selector
	3.3.5.3.	Circuits that use OR gating of clocks and internally generated signals should be tied to a specific voltage level using an AND gate
	3.3.5.4.	Circuits that use AND gating of clocks and internally generated signals should be tied to a specific voltage level using an OR gate
	3.3.7.1.	Measures to prevent clock skew between different clocks are required
	3.3.9.1.	Use boundary scanning on the I/O for hard macros and asynchronous circuits

Level	Item No.	Contents
recommend 1	3.5.1.2.	Do not save test vectors in RTL description directories
	3.5.1.4.	File names should be specified as relative paths
	3.5.2.1.	Each file's reference should be specified by a relative path (return to the master directory once)
	3.5.2.2.	The file name of the RTL description should consist of module name + ".v"
	3.5.2.13.	SDF file names must end with ".sdf"
	3.5.2.14.	EDIF file names must end with ".edif" or ".edf"
	3.5.2.15.	The file name of the logic synthesis databases must end with ".db"
	3.5.3.1.	Indicate the circuit name, circuit function, author, and creation date in the file header
	3.5.3.2.	Indicate who made changes and which item was modified in the case of reuse
	3.5.3.3.	Make file headers common
	3.5.4.1.	In multiple designer development, separate the master data from the individual data
	3.5.6.6.	EUC has fewer problems than any other Japanese codes
	4.1.1.1.	Hierarchize the test bench using hierarchical structure or task/function
	4.1.2.1.	Use parameters - Do not use embedded values in the test descriptions
	4.1.3.2.	When the clock's falling edge is used, it should also be shifted from the falling edges
	4.1.4.1.	Avoid assigning from multiple initial constructs to the same signal (Verilog only) - Resulting value is unknown if racing occurs
	4.1.4.2.	Define one signal using one description block (Verilog only)
	4.1.6.1.	Do not put cycle computation expressions inside an <i>always construct</i> - Simulation will be slow. Errors will occur easily.
	4.1.8.1.	Shift the observation point of a signal from an <i>assignment</i> point
	4.1.8.4.	Avoid using edges other than clock signals to the greatest extent possible (posedge CLK only)
	4.2.3.2.	Do not define output arguments when generating test vector using tasks
	4.2.3.4.	Do not define as input arguments when regularly observing the signals inside a task (Verilog only)
	4.3.1.1.	Create test specifications, and use them as a checklist
	4.3.1.4.	If generating expected values by Verilog-HDL description is difficult, compare with the previous results
	4.3.1.5.	With gate level simulation, verify only the matching with RTL simulation
	4.3.2.1.	Use test specifications to organize the verification contents and to check for functional check omissions - It is important to know what tests were performed
	4.3.2.2.	List the test items in simple terms
	4.3.2.3.	Create test specifications before starting verification

Level	Item No.	Contents
recommend 1	4.3.5.1.	Compare the previous results (expected values) with the simulation results - Compare files using the UNIX diff command - However, debugging to discover which output caused a mismatch is not easy
	4.3.9.1.	Separate the function verification patterns and the fault detection patterns
	4.4.2.2.	Be sure there is no 'x' status FF after releasing the initial reset
	4.4.3.1.	Eliminate racing problems
	4.4.3.2.	Verify the asynchronous part with a gate-level simulation
	4.4.3.3.	Use of static timing analysis
	4.5.1.2.	Check all timing violations
	4.5.1.6.	Specify set_failelse_path, set_multicycle_path carefully by pin-to-pin
	4.5.2.2.	Avoid letting the circuit operate on multicycle as much as possible
	5.1.1.1.	Create a batch file and execute batch processing using dc_shell
	5.1.5.2.	The wire_load_model of min should be set uniformly for all the levels
	5.1.6.1.	When there are no clocks whose phases are different, the rising edge time should be 0
	5.1.6.3.	Check any missing clock settings by derive_clocks() command
	5.1.6.4.	Add create_clock to the input port of the top level or the primitive cell pin
	5.1.10.1.	Ordinarily, set 0 for the value
	5.1.11.1.	Hierarchical synthesis and optimization after ungroup are compile -incremental
	5.1.11.2.	Use -map_effort high with -incremental
	5.2.1.3.	Do not use -source option of set_clock_latency for logic optimization
	5.2.2.3.1.	ungroup sub levels of basic blocks and lower
	5.2.2.4.2.	Always use incremental compile after ungrouping the lower levels
	5.2.2.6.1.	Use this command when not using the ungroup command for basic block optimization (not ungrouping the level of the operator generated by Design Compiler)
	5.2.3.1.	Specify with -effort high
	5.2.6.1.	In principle, specify pin-to-pin
	5.2.9.2.	Set set_max_fanout or set_max_capacitance for the input port in the event of large level analysis
	5.3.3.	Logic synthesis should be performed in logic area first
	5.3.7.	For up to 200 thousand gates, hierarchical synthesis should be performed by compile -inc
	5.4.5.1.	Start optimization from basic blocks (2,000 - 20,000 gates)
	5.4.5.2.	Perform sub-block optimization only when special optimization such as flatten is required
	5.4.5.3.	Either ungroup or process unconnected ports in sub-blocks or DW01 levels using ungroup or remove_unconnected_ports

Level	Item No.	Contents
recommend 1	5.4.5.5.	Perform partitioned optimization for 200,000 gates or more (do not use characterize)
	5.5.3.	Eliminate unconnected ports when executing characterize
	5.5.4.	Do not execute characterize on circuits that include latches
	5.6.1.	In order to select properly the rpl type adders and subtractors, execute compile without setting the clock cycle
	5.6.3.	Before executing compile -inc for those circuits including operators, execute either ungroup or remove_unconnected_port

Level	Item No.	Contents
recommend 2	1.1.1.1.	File names should be as follows: "<module name>.v".
	1.1.1.7.	Add an identifying symbol at the end of the name so the polarity of negative logic signals is clearly identified ("_X", "_N" for example)
	1.1.2.6.	Output pin names and the connected net names should be the same - Upper level net names and the input pin names should be the same (recommend 2)
	1.1.2.7.	Do not use signal names that have sub-block instance names at the beginning
	1.1.4.1.	Use either ".h", ".vh" or ".inc" for RTL description and ".inc", ".ht" or ".tsk" for test bench as the include file (Verilog only)
	1.4.3.1.	Avoid reversals on the same clock line, using gated clocks and using FFs with different edges
	1.5.1.4.	Meta stable measures should be taken even for input from outside LSIs that are susceptible to noise
	1.6.1.1.	Limit the gate size of a single level to 10,000 gates or fewer for safety's sake - Keep the size of a single level as small as possible as the operating frequency grows higher - Limit the gate size of a level to 20,000 gates or fewer even when the operating frequency is low (mandatory)
	1.6.4.3.	If the scale of a basic block is about 10,000 gates, the number of I/O ports should be specified to no more than 200
	2.1.5.3.	The result should not be a vector in the conditional expression of <i>if statement</i> and conditional expression(?)
	2.1.6.1.	Specification of an array should be [MSB:LSB], if it is one-dimensional
	2.1.6.4.	The range of an array index should be appropriately specified
	2.2.2.2.	Do not define constants and unnecessary signals in the sensitivity list
	2.5.1.2.	Do not describe logic in conditional expression to infer tri-state
	2.5.1.4.	Specify up to five tri-state buffer connectors at most
	2.5.1.7.	Tri-state output should not be used in a conditional expression of an <i>if statement</i>
	2.5.1.8.	Tri-state output should not be used in a selection expression of a <i>case statement</i> that is not assigned 'x' as default clause

Level	Item No.	Contents
recommend 2	2.5.1.9.	Tri-state output should not be entered in the selection expression of <i>casex</i> or <i>casez</i> statements
	2.5.2.1.	Not only a block for a tri-state buffer but also a block for input cell connected directly from bidirectional bus should be made
	2.6.1.1.	Describe with the structure of the circuit to be generated in mind
	2.6.1.5.	Use intermediate variables when a same logic is used in more than two places
	2.7.1.2.	Avoid unnecessary priorities
	2.8.1.6.	Pay attention to the selective range of <i>case statement</i> and bit width of each item (Verilog only)
	2.8.5.4.	Do not describe logical operations and arithmetic operations in the selection expression of <i>case statement</i> (Verilog only)
	2.9.1.1.	Do not use for statements for other than simple repeating statements (that do not generate priorities)
	2.10.1.2.	Use parentheses for logical operation to make the description more readable even if the precedences are clear
	2.10.1.3.	Use parentheses for two or more arithmetical operations
	2.10.3.1.	Clearly match the bit width between the right side and the left side of relational operator (Verilog only)
	2.10.3.4.	The bit width of the right-hand side <i>assignment</i> should not be narrower than the left-hand side <i>assignment</i> (Verilog only)
	2.10.3.5.	Specify base format('d','b','h','o') for constant and these base formats should be kept in mind (Verilog only)
	2.10.4.6.	Do not assign negative value to signals, which are declared with <i>reg</i> or <i>wire</i>
	2.10.5.5.	Do not describe arithmetic operations with conditional operators (?) in assign statement (resource sharing will not be performed)
	2.10.7.2.	Clearly describe common operational expressions
	2.11.1.4.	Keep the number of states to 40 or less
	3.1.3.2.	The port description order should be clock, reset, input, output and I/O
	3.2.2.5.	Do not nest text macros (Verilog only)
	3.3.3.2.	Do not connect the input of a FF to VDD or GND
	3.3.4.3.	Use latches, which operate on inverted clocks, in 1 stage
	3.3.5.5.	Circuits wherein gating is performed on clocks and latch outputs should have an OR performed on the scan select and the stage prior to the latch
	3.3.5.6.	When gating a clock and the output of a FF, tie to a specific voltage level by performing AND gating with the stage after the FF
	3.3.5.7.	When the output of a latch is used as a clock, tie to a specific voltage level by performing OR gating with the latch clock input
	3.3.8.1.	Tristate enable signals should be able to be fixed from an external input port
	3.5.1.3.	For data version management, copy the directories and delete unnecessary files to save in a compact size
	3.5.2.5.	The include file name should be ".h", ".vh" or ".inc" for RTL description, ".inc", ".ht" or ".tsk" for test bench (Verilog only)

Level	Item No.	Contents
recommend 2	3.5.5.1.	Back up files
	3.5.5.2.	Periodically back up all design directories
	3.5.6.1.	Use comments often to improve the readability of the source code
	3.5.6.2.	Add comments that indicate the objective and content to operators in the description
	3.5.6.3.	Describe the I/O ports and declarations in one line and always add comments
	3.5.6.4.	Provide the comments in English as much as possible
	4.1.1.2.	Define common timing using parameter statements or text macros
	4.1.2.3.	The number of lines in <i>fork-join</i> should be a maximum of 5 (Verilog only)
	4.1.2.4.	Specify time units using 'timescale (Verilog only)
	4.1.7.1.	Separate the descriptions for each clock - Facilitates clock cycle modifications - Facilitates understanding of waveforms
	4.1.8.5.	Pay due attention to the timing of asynchronous resets
	4.1.9.2.	Use (<=) for the simulation description between asynchronous clocks
	4.1.10.1.	Use handshakes when the process cycle count is unclear - You can change the test input dynamically by observing the circuit status
	4.3.1.3.	To clarify verification, generate expected values by Verilog description and then compare them
	4.3.3.1.	Clarify how test items and the test bench descriptions are related
	4.3.3.3.	Note the circuit revisions
	4.3.4.1.	Clarify simulation results not only by waveforms but also by comparing with the previous results
	4.3.6.2.	Some techniques are required for comparing with don't-care
	4.5.1.9.	If false_path is used between asynchronous clocks, then check each path
	4.5.2.3.	For false paths violating timing analysis, insert FF or cut the path
	5.2.2.4.1.	Use incremental compile as hierarchical synthesis (lower level optimization has already completed)
	5.3.1.	Logic synthesis should be performed in the bottom-up manner from the basic level block
	5.4.1.2.	In the case of area optimization, compile without timing constraints, then execute incremental optimization
	5.4.1.3.	It is better for logic optimization to start from area optimization
	5.4.2.1.	set_flatten true -ef high is effective for state machine descriptions and large <i>case statement</i> description (however, only up to about 2,000 gates) (See 5.2.3)
	5.4.2.2.	The easiest method is compile -inc -map_ef high
	5.4.3.3.	If flatten is not used, then do not execute optimization of less than 1,000 gates
	5.4.4.1.	Be careful when setting clocks with different phases
	5.4.4.2.	Be careful when setting clocks that are asynchronous

Level	Item No.	Contents
recommend 3	1.1.1.8.	Instance names should basically be the module names. Instance names that are used more than once should be "<module name>_<quantity>"
	1.1.2.2.	Add a hierarchy identification character to the beginning of the top level block name
	1.1.2.3.	Add the hierarchy identification character of the upper level to the hierarchy identification character of the sub-block
	1.1.2.4.	The first string of the output pin name for each block should be "<hierarchy identification character>" + "_"
	1.1.4.2.	<i>Parameter names</i> should have different naming convention
	1.1.4.3.	Do not use parameters with same name for different modules
	1.1.4.7.	Parameterize the bit width of ports required for circuits that will be reused
	1.1.5.1.	Give register output signal names that suggest the clock system or register
	1.1.5.2.	Basically, use "CLK" or "CK" for clock signal names, "RST_X" or "RESET_X" for reset signal names and "EN" for enable signal names. Add identifier to the end of these basic names.
	1.3.1.2.	It is safer to use asynchronous reset for initial reset to a register - Unlike ordinary paths, initial reset is not critical in timing - Reset tree synthesis at layout is easy - Values may not be fixed in a gate-level simulation with synchronous reset
	1.3.1.5.	Do not use synchronous reset directives for a particular logic synthesis tool
	1.3.3.2.	An initial reset may have to be synchronized or else a noise elimination circuit may be needed
	1.4.3.5.	Clock signals should not be connected to black boxes, bi-directional pins or reset lines
	1.5.3.2.	Synchronous RAM has a long hold time, so some measures are necessary
	1.6.2.1.	Make all basic blocks combinational circuit input and FF output
	1.6.3.2.	Stay within three sub-blocks whenever possible if there are any problems in terms of speed
	1.6.4.2.	Make paths with severe speed constraints similar, even in the sub-blocks
	1.6.6.1.	ASIC I/O cells should be inserted only in the top level or the I/O cell level
	2.1.4.6.	Be careful about bit width for reduction operators (Verilog only)
	2.1.5.1.	Use nesting of conditional operator (?) only once (Verilog only) - Use of (?) should be limited to 10 times at most (MANDATORY)
	2.1.6.2.	The LSB of an array should be 0
	2.1.6.3.	The index of an array should be simple signal names only
	2.3.1.3.	Set delay values for FF inference
	2.3.6.2.	Asynchronous reset is only one bit, which is low active with negedge
	2.4.1.2.	Create latch only blocks and infer latch in these blocks only

Level	Item No.	Contents
recommend 3	2.5.1.1.	Make a block for a tri-state buffer
	2.6.1.3.	The number of signal output from one <i>always construct</i> should be five or less, if possible
	2.6.1.4.	The number of lines in an <i>always construct</i> should be up to 200 - 2000 lines at most (mandatory)
	2.7.1.1.	Bring signals with critical timing or signals with many changes to the beginning of the conditional branch - Timing acceleration is possible - Advantageous in reducing power consumption
	2.7.3.1.	The number of nest for if-if and else if is best at 5 or less
	2.7.3.4.	Unify <i>if statements</i> that are mergeable
	2.8.2.1.	Division criterion is within 24 I/O (input: 16, output: 8)
	2.8.2.2.	Item count should be within 100
	2.8.3.1.	The don't-care condition is defined by using `x` as the default clauses (only for default clauses, the extensive use of don't care is recommended)
	2.8.4.3.	It is best to avoid using casex statements and casez statements (Verilog only)
	2.8.6.1.	In the case of nesting that <i>if statements</i> and <i>case statements</i> coexist, it is better to have a large table in a small condition than to have a small condition in a large table
	2.9.2.2.	Do not describe any logic or relation operations other than with loop variable and constant
	2.10.1.1.	Describe with the operation precedence in mind
	2.10.3.7.	Match the bit width of value with the base number part (2`b) (Verilog only)
	2.10.5.4.	It is hazardous to depend too much on resource sharing. Resource sharing is not perfect.
	2.10.5.6.	When performing resource sharing, always check generated circuits
	2.10.6.1.	Carry-out should be considered for bit width of signals to which operation result will be assigned
	2.10.6.3.	Signed operations and unsigned operations should not be mixed in one statement
	2.10.6.4.	Use continuous <i>assignments</i> to clearly define data path structures
	2.10.6.7.	Do not describe more than one arithmetic operation in one line (except for carry-in A+B+CIN(1bit))
	2.11.1.1.	Use the Moore type in principle - Either Mealy or Moore type descriptions are okay if all output signals are FF
	2.11.1.2.	Minimize the bit change in state transitions (fundamentally gray code)
	2.11.1.3.	Define state values by parameters
	2.11.1.5.	If there are more than 40 states, divide the states and describe using a separate module
	2.11.2.1.	Isolate state machine circuits - Different logic optimization methods can be tried - State allocation changes are possible

Level	Item No.	Contents
recommend 3	2.11.3.1.	Separate the FF inferences and case statements defining state, in state machine circuits - Describe FF inference using independent <i>always constructs</i> - Describe state machine with another <i>always construct</i> using <i>if statements</i> and <i>case statements</i>
	2.11.4.1.	Allocate the states properly - Basically gray code. Use one hot when speed is an issue. - Allocate states so there are fewer bit changes - One hot only increases the area 20-30% if there are up to about 15 states
	2.11.4.2.	Use parameter or define to facilitate allocation changes
	3.1.1.1.	Create sub-programs which can be used in common
	3.1.1.2.	Create reusable component libraries and streamline the design
	3.1.1.3.	Standardization of design data is necessary for design libraries to be reused
	3.1.2.3.	Use a simple clock system
	3.1.2.4.	Insert appropriate comments to make it easier to understand
	3.1.2.5.	Describe with the circuit structure in mind
	3.1.2.6.	Parameterize whenever possible
	3.1.3.1.	Unify the description order of the port declaration, port list and module instantiation port lists defined in the modules - Fewer errors will be made when defining ports, so they will become easier to understand when viewed
	3.1.3.3.	Separate the reg declarations of sequential circuits and combinational circuits
	3.1.3.4.	Define one signal per line in I/O, <i>reg declaration</i> , <i>wire declaration</i> . Always add comments
	3.1.4.3.	Replace tabs with spaces after editing
	3.1.4.4.	Do not describe multiple <i>assignments</i> in 1 line
	3.1.4.5.	The maximum number of characters in 1 line should be about 110
	3.2.1.1.	Create separate directories, then store a function library for each function
	3.2.1.2.	Only allow specific administrators to make modifications to function libraries
	3.2.1.3.	Place the common ".synopsys_dc.setup" file in a common directory also
	3.2.2.1.	Describe constants by parameters as much as possible
	3.2.2.2.	Define global parameters relating to the overall design in a separate file
	3.2.2.3.	Read parameter files using <code>`include</code> (Verilog only)
	3.2.2.7.	Use only <i>parameter</i> in the overall design project (Do not use <code>`define</code>)
	3.3.2.2.	Do not connect clock pins, reset pins, or tristate outputs to black boxes
	3.3.4.4.	When more than 2 stages of latches are used in a two-phase clock design, use test patterns created by the designer to detect faults

Level	Item No.	Contents
recommend 3	3.3.5.8.	In order to add logic to clock lines for safe DFT, generate the clock-generating modules in the top level
	3.3.6.5.	In order to add DFT logic to a reset line safely, create the reset-generation module in the top level
	3.3.7.2.	Insert a latch with inverted clock when transmitting between asynchronous clocks
	3.3.8.2.	Tristate enable signals should be controllable directly from the outside or should be controlled by a decoder that is controlled directly from the outside
	3.3.8.3.	External bidirectional pins should be set during the scan shift
	3.3.9.3.	Pay attention to fault detection in the RAM I/O when RAM is used
	3.5.2.3.	The file name of the test bench should be "_tb.v", "_test.v" or ".vt"
	3.5.2.4.	The file name of the gate description should consist of module name + ".v" or ".vnet"
	3.5.2.6.	Execution file names on UNIX should end with ".run"
	3.5.2.7.	Simulation (Verilog) script file names should end with ".v_scr"
	3.5.2.8.	lint script file names should end with ".l_scr"
	3.5.2.9.	The file name of the logic synthesis scripts should end with ".scr"
	3.5.2.10.	All logic file names for synthesis, simulation, and layout logs should end with ".log"
	3.5.2.11.	The file name of the lint logs should end with ".l_log"
	3.5.2.12.	Logic synthesis report file names should end with ".rep", ".tim" or ".ara"
	3.5.3.4.	Consider using CVS if necessary
	3.5.6.7.	Comments should start with "/"
	4.1.5.1.	Input test vectors synchronizing with clock - Reflect the clock cycle changes to other signals - Making the timing relationships more comprehensive
	4.1.8.3.	Pay due attention to time 0 events in clock edge-based descriptions
	4.2.1.2.	Define application tasks by combining basic tasks
	4.3.3.2.	Clarify what have to be confirmed in the test items
	4.5.1.4.	Do not specify set_false_path for signals inside of ASIC as much as possible
	4.5.1.5.	Do not specify set_multicycle_path inside ASIC as much as possible
	5.1.5.1.	In principle, the wire_load model should be set for each basic level
	5.1.6.2.	Do not use decimals for the cycle, rise and fall
	5.4.1.1.	set_structure -boolean true and -area-effort high are effective in the area optimization

Level	Item No.	Contents
reference	1.1.2.5.	Naming conventions for input signal names and output signal names for each block should be different from those for internal signal names

Level	Item No.	Contents
reference	1.1.3.1.	Naming conventions for internal signals of blocks should be different from those for input and output signals
	1.1.3.2.	Give meaningful and comprehensive names for internal signals of hierarchy
	1.1.4.5.	Add hierarchy ID characters to parameters, which are used only for within a level of hierarchy
	1.1.5.3.	Names that suggest the clock
	1.1.5.4.	Names that suggest the register
	1.3.1.1.	Circuits which can not be reset by synchronous reset may be optimized
	1.3.1.4.	When using synchronous reset circuits, establish new hierarchy for the register with synchronous reset
	1.3.2.3.	Optimize by keeping in mind the timing of the signal that outputs the reset line
	1.3.3.1.	There is danger of malfunction unless attention is paid to reset lines on the circuit board
	1.3.3.3.	In some systems, an initial reset signal is asserted before the clock - The asynchronous method is preferred in this case, but you should add a noise elimination circuit
	1.4.1.2.	Do not subject clocks to synthesis
	1.4.2.1.	For clock lines, do not use primitive cells other than dummy buffers on a clock tree
	1.4.2.2.	Clock should be balanced in accordance with the number of cells to be connected to each clock tree
	1.4.3.3.	Using gated clocks is an effective method for achieving low power consumption
	1.4.4.1.	Divide up hierarchical blocks in each clock whenever possible
	1.4.4.2.	When inputting multiple clocks in the same block, provide an integral multiple period as a clock constraint
	1.5.1.5.	To avoid erroneous input data, latch the clock signals and use them as enable signals
	1.5.2.1.	Use FIFO for transfers between asynchronous clocks with the same clock period - Clocks have the same clock period
	1.5.2.2.	Use frame memory for transfers in asynchronous with different periods
	1.5.3.3.	Margin can be guaranteed by using a latch or by using buffer, inverter or delay cells
	1.5.3.4.	Allocate RAM to the top level if possible
	1.5.3.5.	There is also a method for avoiding modification of I/O for each ASIC vendor by creating a general-purpose RAM module and fixing the I/O
	1.6.1.3.	Lower-levels of the basic blocks (on a 2,000~10,000 gate scale) are optional
	1.6.2.3.	The above restrictions do not apply to smaller levels below the basic blocks (2,000 – 10,000 gates in scale)
	1.6.3.1.	Limit paths that are critical in terms of speed to within two sub-blocks inside each basic block, if possible

Level	Item No.	Contents
reference	1.6.5.1.	Description styles are different for the data path section and controller
	1.6.5.2.	Different synthesis methods can be chosen for the data path section and the controller
	1.6.6.2.	In large-scale ASICs, it is not possible to synthesize everything from the top level
	1.6.6.3.	When a high drive capacity buffer is required for the output of a level with 200,000 to 800,000 gates, create a separate level containing only buffers
	2.1.1.1.	For the effective reuse of design resources, standardize the use of <i>always construct</i> or function statement+assign statement (Verilog only)
	2.1.1.3.	Use the syntax checker tool to avoid mistakes
	2.1.1.4.	For <i>always construct</i> , add a comment for each reg variable (Verilog only) - Make clear whether they are for combinational circuits or sequential circuits
	2.1.3.3.	When returning values to multiple signals, use concatenation (Verilog only)
	2.1.4.1.	Operators having the same precedence are evaluated from the left (Verilog only)
	2.1.4.2.	Use bit-wise operators instead of equation levels, even in single-bit operator expressions (Verilog only)
	2.1.4.3.	Descriptions of equation levels have advantages in terms of area and speed (especially in selectors, encoders, and descriptions with large bit width)
	2.1.4.4.	Readability deteriorates if complex logic is described with equation levels
	2.1.5.2.	Where a conditional expression is an unknown value `x`, the <i>assignment</i> values are compared for each bit of the branch statement - Where the conditional expression is an unknown value `x`, a value will be more precise using an <i>if statement</i> (Verilog only)
	2.2.1.2.	Pay attention to warning messages in RTL check tool and logic synthesis tool, and verify the presence or absence of latch generation
	2.2.2.4.	The sensitivity list should be verified with a RTL check tool rather than a logic synthesis tool
	2.3.2.1.	Circuits generated by <i>non-blocking assignments</i> (\leq) and <i>blocking assignments</i> ($=$) may be different in FF inferences
	2.5.1.3.	Infer a tri-state buffer by assigning high impedance `z`
	2.5.2.2.	z' becomes 'x' when passing through gates, so circuits that mask using gates may be added as a gate level verification means to prevent 'x' propagation
	2.5.2.3.	Mask circuit may be optimized after logic synthesis, so 'x' propagation may not be prevented
	2.5.2.4.	To do RTL simulation, remove bus holder temporarily

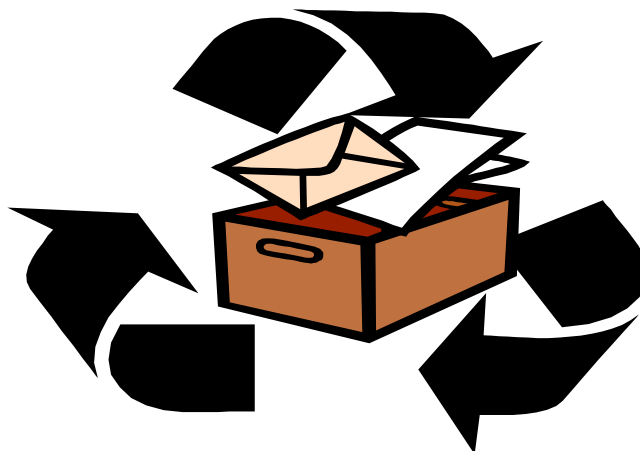
Level	Item No.	Contents
reference	2.7.3.2.	Associating <i>if statements</i> that have deep nesting with else items is difficult
	2.7.3.5.	Use of tabs (indenting) will reduce mistakes in <i>if statement</i> nesting
	2.7.4.1.	It is not necessary to surround a single statement by block statements (Verilog only)
	2.7.4.2.	Be sure to remember to attach begin-end (Verilog-only)
	2.8.1.1.	<i>case statements</i> are suitable for describing decoders/encoders with many branches
	2.8.1.2.	Described properly, <i>case statements</i> create faster circuits than <i>if statements</i> do
	2.8.2.3.	Do not put the truth table in <i>case statements</i> as is - Create truth table that takes the HDL description into account - Divide in advance <i>case statements</i> that are easily divisible
	2.8.3.2.	When 'x' is used as default clauses, 'x' is propagated by the simulation even if 'x' is not don't-care
	2.8.3.3.	Can suppress latch generation (Verilog only) - Use default clause, do not use //synopsys full_case
	2.8.4.1.	Complex casex statements decrease circuit quality (Verilog only)
	2.8.4.2.	Readability declines with complex casex statements (Verilog only)
	2.10.2.1.	Is possible to handle logical operation results (TRUE, FALSE) in the same manner as logic values, and use them to simplify the description (Verilog only)
	2.10.2.2.	Results of logical operation will be 1 bit (Verilog only)
	2.10.4.2.	Integers are defined as 32-bit values
	2.10.4.4.	Pay attention when comparing integer variables and reg/wire variables
	2.10.5.1.	Take notice that arithmetic operators (+, -, *) and relational operators (<, >, =) with large bit width generate large circuits - Take notice specially of relational operators, which are 8-bits or more
	2.10.5.2.	Logic synthesis tools can share resources which are arithmetic operators and relational operators
	2.11.3.2.	Unless <i>case statements</i> are separated, the output signal is latched and delayed by 1 clock cycle - If it is allowed to be a 1 clock delay, there is no need to recognize Mealy or Moore type descriptions
	3.1.1.4.	Intellectual Properties (IP) can be purchased from third-parties
	3.1.4.1.	Signal naming is essential for RTL description readability
	3.1.4.2.	Unify the number of indents used in <i>always constructs</i> , <i>if statements</i> and <i>case statements</i> (2 spaces are standard)
	3.1.5.1.	It is preferable to parameterize the array range of module I/O as much as possible - Makes it possible to use common modules even in circuits with different bit widths
	3.1.5.2.	Specify the bit width using parameters or text macros - Overriding from an upper level is possible when using parameter statements - Specifying entire design circuits is possible with text macros

Level	Item No.	Contents
reference	3.1.5.3.	Set parameter default values
	3.1.6.1.	Parameter may be described by using `ifdef (Verilog only)
	3.1.6.2.	define names defined by `ifdef selects simulation description or circuit description within one description (Verilog only)
	3.2.2.6.	Distinguish the `include files used in the overall design project from those for individual use (Verilog only)
	3.2.4.1.	Use #(value) to overwrite parameters from an upper level (Verilog only)
	3.2.4.2.	Specify all parameters of a lower level (Verilog only)
	3.2.5.1.	Using library described by a logical operation for selector improves speed
	3.2.5.2.	Use a function library described by a logical operation for a multiplier with 16 bits or more output
	3.2.5.3.	Using a function library for multiple arithmetic operations improves speed and decreases the area
	3.2.5.4.	Use an exclusive tool when including a multiplier and multiple arithmetic operations
	3.3.9.2.	Bypass the I/O instead of using boundary scanning on the I/O
	3.3.9.4.	Consider creating test systems for large-scale ASICs
	3.3.9.5.	Boundary scanning is recommended for blocks larger than a specific size in large-scale ASICs
	3.4.1.2.	When gated clocks are to be used in smaller units, use the EDA tools
	3.4.2.1.	Power consumption can be reduced through dividing the clock to reduce the operating speed
	3.4.2.2.	Dividing clock and parallel execution should be applied for data path part
	3.4.2.3.	Use a small number of enable signals to frequently stop the operation of the circuits
	3.4.3.1.	Power consumption can be reduced through partitioning the clock lines
	3.5.4.2.	CVS can be used to manage file versions
	3.5.6.5.	Some EDA tools may not read comments in languages other than English
	3.5.7.1.	RTL codes may be managed by CVS
	3.5.7.2.	What is easily managed by CVS is the source code such as RTL, test bench and script file. CVS is not well-suited to manage comprehensive sets of all the data
	3.5.8.1.	With CVS, sharing and management information is stored in repositories
	3.5.8.2.	The CVS has the basic commands; checkout and commit
	3.5.9.1.	Managing file versions using CVS commands 1) Copying from the repository 2) Debugging and simulation 3) Storing in the repository 4) Releasing the work region
	3.5.9.2.	Beware of data contention messages

Level	Item No.	Contents
reference	3.5.10.1	Display the file version information in the work region using <i>status</i>
	3.5.10.2.	Display such detailed information as each version, branch, modification date
	4.1.2.2.	Use <i>blocking assignment</i> '=' - When '<=' is used, the operation is postponed - Simulation speed is faster
	4.1.5.2.	Use <i>event statements</i> for clock synchronization
	4.1.8.2.	Clarify the observation point and the <i>assignment</i> point in clock edge-based descriptions
	4.1.9.1.	Pay attention to racing in case of the simulation between asynchronous clocks
	4.1.9.3.	Change delay values to check racing problems
	4.1.11.1.	Use "\$fstrobe" rather than "\$fdisplay" for outputting the results (Verilog only)
	4.1.11.2.	Use \$display, \$fdisplay to display error messages (Verilog only)
	4.1.12.1.	Useful for interfacing with other tools (Verilog only)
	4.1.12.2.	Useful for creating test benches that require complex operations (Verilog only) - Verilog-HDL has no operation functions. Even if such operation functions were defined, they would have poor functionality and thus would be difficult to use
	4.1.12.3.	There is little speed advantage (Verilog only) - For compile type or cycle based type simulators, it would be faster to describe bus models etc. using Verilog-HDL - It is more advantageous to describe simulation models such as macro cells using Verilog-HDL when taking generality into account
	4.2.1.1.	Make it possible for test bench description to be more efficient and structuralized
	4.2.2.1.	Model functional operations like bus operations using tasks
	4.2.2.2.	It is possible to define functional operations by calling tasks - Describing detailed waveforms is not necessary - The descriptiveness and debugging efficiency improve
	4.2.2.3.	It is also possible to use commercially available bus model descriptions
	4.2.3.1.	It is not always necessary to describe arguments to tasks (Verilog only)
	4.2.3.3.	Tasks can define multiple I/O arguments
	4.3.1.2.	Initial debugging stage with RTL is done by checking the waveform output
	4.3.3.4.	List all possible items even if it seems impossible to complete testing for all of them during design phase
	4.3.3.5.	Consider testing priorities
	4.3.3.6.	Consider bug curve graphs
	4.3.4.2.	Creating a batch file makes it possible to run multiple jobs
	4.3.5.2.	Compare the gate level results with the RTL simulation results
	4.3.5.3.	Previous results are not always correct, but the number of correct portions will increase

Level	Item No.	Contents
reference	4.3.6.1.	Run the simulation while comparing with the expected values - Compare with the expected values, and stop the simulation if mismatches occur
	4.3.7.1.	Verify the functionality of the entire circuit and blocks using behavior simulations - Appropriate block partitioning is not possible unless the overall circuit is considered - It is not possible to simulate each block unless the I/O specifications for each block are completed - More than 2 million or more gates are required, but not less than 500,000 gates
	4.3.7.2.	Signal events only (untime) description
	4.3.7.3.	Cycle-accurate description
	4.3.7.4.	System verification using C language
	4.3.8.1.	Hardware debugging using FPGA
	4.3.8.2.	Use of hardware emulators
	4.3.8.3.	Use of Co-Sim
	4.3.8.4.	Use of automatic function verification tool, model-checking and property-checking
	4.3.10.1.	Use a random function for the pattern to verify functions
	4.3.11.1.	Switch the active location of the description using `ifdef (Verilog only)
	4.3.11.2.	Switch the description used by the test bench (Verilog only)
	4.3.11.3.	The debug descriptions in the RTL are not synthesizable, so use "translate_off"
	4.4.1.2.	Gate level verification for a large design can only execute a part of pattern in the RTL verification
	4.4.1.3.	Use unit delay for gate level simulation of large-scale design
	4.4.1.4.	Use a formal verification (equivalence checking) tool
	4.4.2.1.	Handling of unknown values `x` differs between RTL and the gate level
	4.4.2.3.	Circuits in which the values are not determined from the synchronous reset description may be synthesized
	4.4.2.4.	`x` will not propagate when an <i>if statement</i> is used
	4.4.2.5.	Propagation of `x` and z` are different in <i>case/casex/casez statements</i>
	4.4.4.1.	Confirm that the all necessary items are on the sensitivity lists
	4.4.4.2.	Incomplete sensitivity lists cause mismatches between RTL and gate level simulation results
	4.4.4.3.	Bugs in libraries, and tools also cause mismatches
	4.5.1.3.	When violations occur, other violation paths may not have been displayed
	4.5.1.7.	In the case of false path, insert FFs and cut the path on the circuit
	4.5.1.8.	Perform timing analysis of the ASIC I/O pins one by one
	4.5.2.1.	Too many specifications of set_false_path, set_multicycle_path makes analysis difficult
	5.1.1.2.	Consider that design_analyzer is only for checking circuit diagram or testing optimization

Level	Item No.	Contents
reference	5.1.1.3.	Prepare various standard scripts to execute optimization without errors
	5.1.4.1.	It is easier to consider executing all the operating conditions by MAX for logic synthesis
	5.1.8.3.	Pay attention to specifications to the clock and reset
	5.2.1.2.	Specify set_clock_uncertainty -from-to to change the skew value between different clocks
	5.2.2.2.1.	Specify dont_touch only for asynchronous circuits, tri-state buffers and buffers for which drive capacity has to be fixed
	5.2.2.2.2.	Avoid hierarchical synthesis that sets dont_touch for the lower levels
	5.2.3.2.	Executable range is up to 2,000 gates taking run time into consideration
	5.2.4.1.	-boolean true is effective to decrease area
	5.3.2.	Ungroup all the levels under the basic block
	5.3.4.	Expanding logic area using too many high drive cells is not good for layout
	5.3.5.	The library needs wire area specification. Specify dont_use for cells with 6 inputs or more
	5.3.6.	The drive strength should be adjusted on the layout side
	5.4.2.3.	Effective in speed optimization when cells other than the power cell are set to dont_use
	5.4.2.4.	If there is a problem in a particular path, group_path command is effective (See 5.2.5.)
	5.1.10.2.	In synthesis of a large scale design, run time may become long by constraining set_max_area 0



Thank you for your cooperation in recycling design
for design environment

All rights reserved by hd Lab, inc. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photocopying, recording, or otherwise, without the prior written permission of hd Lab, inc.

We will not assume any responsibility for the results of the use of any design methods described or shown herein.

Design Style Guide fourth edition(Rev4.02)

first edition issued on February 1st, 1998

second edition issued on March 1st, 1999

third edition issued on April 20th, 2001

fourth edition issued on May 17th, 2002



Editor: Toshiba Corporation

hd Lab, inc.

Author & Publisher: hd Lab, inc.

Plustaria Bldg.6F, 3-1-4 Sin-yokohama, Kohoku-ku,
Yokohama, 222-0033 Japan.

TEL 045-477-4315

FAX 045-477-4316

MAIL: info@hdlab.co.jp

URL <http://www.hdlab.co.jp>

Design Style Guide Verilog-Version Rev4.02

Toshiba Corporation

