Verilog HDL Coding

Semiconductor Reuse Standard

SRS07HDL
V2.0

MOTOROLA

Revision History

Version Number	Date	Author	Summary of Changes
1.0	29 JAN 1999	SoCDT	Original
1.1	08 MAR 1999	SoCDT	Revision based on SRS development process. Detailed history contained in DWG records.
2.0	06 DEC 1999	SoCDT	Revision based on SRS development process. Detailed history contained in DWG records.

Table of Contents

Section 7 Verilog HDL Coding

7.1 Introduction 7-9
7.2 Reference Information 7-9
7.2.1 Documented References 7-9
7.2.2 Terminology 7-9
7.3 Naming Conventions 7-9
7.3.1 File Naming 7-9
7.3.2 Naming of HDL Code Items 7-10
7.4 Comments 7-13
7.4.1 File Headers 7-13
7.4.2 Additional Construct Headers 7-16
7.4.3 Other Comments 7-17
7.5 Code Style 7-19
7.6 Module Partitioning & Reusability 7-21
7.7 Modeling Complex Behavior 7-23
7.8 General Coding Techniques 7-25
7.9 Standards for Structured Test Techniques 7-26
7.10 General Standards for Synthesis 7-27

Appendix A

A.1 Example Header Files 7-33

List of Figures

Figure 7-1 Verilog File Header 7-14
Figure 7-2 Verilog Functions, User-defined Primitives and Tasks Header 7-16
Figure 7-3 Verilog Coding Format. 7-20
Figure 7-4 Clock Domain Partitioning 7-22
Figure 7-5 Partition Asynchronous Logic 7-23
Figure 7-6 Scan Support for Mixed Latch/Flip-Flop Designs 7-27

Section 7 Verilog HDL Coding

7.1 Introduction

The general coding standards pertain to IP/VC generation and deal with naming conventions, documentation of the code and the format, or style, of the code. Conformity to these standards simplifies reuse by describing insight that is absent from the code, making the code more readable and assuring compatibility with most tools. Any exceptions to the rules specified in this standard, except as noted, must be justified and documented.

The methodology standards promote reuse by ensuring a high adaptability among applications. The intent of this document is to ensure that the gate level implementation is identical to the HDL code as it is understood by a standard Verilog simulator. Partitioning can affect the ease with which a model can be adapted to an application. The modeling complex behavior section deals with structures that are typically difficult to address well in a synthesis environment and are needed to ensure pre- and post-synthesis consistency. These standards apply to behavioral as well as synthesizable code.

These Verilog-centric standards were developed after an analysis of the Motorola design community, and are heavily based on the existing Module Board coding guidelines. However, several other sources were also considered, including VSIA, M-CORE, and Star12.

7.2 Reference Information

7.2.1 Documented References

[1] Reuse Methodology Manual for System-on-a-Chip Designs, M. Keating, P. Bricaud, Kluwer Academic Publishers, 2nd Edition, 1999.

7.2.2 Terminology

HDL: Hardware Description Language

MTBF: Mean Time Between Failures

PLL: Phase Locked Loop

RTL: Register Transfer Level

7.3 Naming Conventions

7.3.1 File Naming

RULE 7.3.1 One design unit per file

A file must not contain more than one design unit. Everything contained in a design unit must be completely contained in a single module/endmodule construct.

reason: Simplifies design modifications.

RULE 7.3.2 File naming conventions

The file name must be composed in the following way:

<design unit name>.<ext>

where:

<*design unit name*> is the name of the design unit (*i.e.*, module name).

<*ext*> signifies that it is a Verilog file:

`.v` Verilog file

`.va` Verilog-AMS file

`.v.mpp` Motorola Pre-processor file

reason: Simplifies understanding the design structure, and file contents.

example: `spooler.v` Synthesizable Verilog code for module spooler

note: Refer to the IP Block Deliverables section for model naming conventions.

RULE 7.3.3 Analog and digital Verilog files

An individual file must contain: (1) only analog Verilog (with the `.va` file extension); or (2) only digital Verilog (with the `.v` file extension; or (3) only explicit mixed analog/digital Verilog (with the `.va` file extension).

reason: Analog compilers may not handle digital constructs, and vice versa.

7.3.2 Naming of HDL Code Items

A meaningful name very often helps more than several lines of comment. Therefore, names should be meaningful (*i.e.*, the nature and purpose of the object it refers to should be obvious and unambiguous).

RULE 7.3.4 Allowable character set

Names must be composed of alphanumeric characters or underscores [A-Z, a-z, 0-9,_] (see **RULE 7.3.5**).

exception: Consecutive underscores are not allowed.

reason: Double underscores will not work with hardware emulation (Quickturn)

RULE 7.3.5 First character of a name

Names must start with a letter, *not* a number or underscore (see **RULE 7.3.4**).

reason: Names starting with a number or underscore may cause conflicts with tools.

RULE 7.3.6 All names must be unique irrespective of case

Case must not be used to differentiate cell names or signals.

reason: Use of some tools may not be able to differentiate on the basis of case. Moving designs between Verilog (case sensitive) to VHDL (case insensitive) is facilitated by a case insensitive design style.

RULE 7.3.7 Consistent use of signal names

Consistent usage in the spelling and naming style of signals and variables must be used. This includes case and naming conventions.

reason: In contrast to VHDL, Verilog as well as some tools supporting VHDL design are case sensitive. Immediate identification of signal types (*e.g.*, active low signals or clocks) eases debug.

RULE 7.3.8 Constant, parameter and block label are upper case

Consistent upper case spelling of parameter and constant names must be used. Therefore, all letters must be upper case for:

- constants

- parameters
- block labels

reason: Provides a mechanism to identify objects that do not go through data changes during simulation.

RULE 7.3.9 Signal, constructs and instance labels are lower case

Consistent lower case spelling of signal and construct names, and instance labels must be used. Therefore, all letters must be lower case for:

- signals
- constructs
- and instance labels

reason: Differentiates signals and constructs from objects that do not change data during simulation, and maintains a consistent look and feel between designs.

RULE 7.3.10 Meaningful signal and variable names

The lower-case name must contain the purpose of the variable/signal.

reason: Description of *what* not *how* aids in understanding the design.

example: `data_bus, set_priority`

RULE 7.3.11 Meaningful constant names

A constant name must describe the purpose of the constant. The type of the constant must be obvious by the purpose and is not part of the name. Constants are upper-case.

reason: Meaningful names are particularly important for constants.

example: Good naming: `SBUS_DATA_BITS, MEMORY_WIDTH, CLK_PERIOD`

example: Bad naming: `ADDRESS_SIZE` is not clear if it refers to the number of bits or the size of the address space.

RULE 7.3.12 Meaningful construct names

Construct names such as functions, modules, tasks, etc. must be named according to what they do rather than how they do it. Construct names are lower case.

reason: Description of what, not how, aids in understanding the design. *How* can be seen from the code, *what* may not be immediately obvious.)

RULE 7.3.13 Meaningful instance labels

Instance labels must be named according to what task is carried out by the construct, not according to how it is done. Instance labels are lower case.

reason: Description of what, not how, aids in understanding the design.

example: `addr_decode, bit_stuff, sbus_if`

RULE 7.3.14 Underscore separate names composed of several words

For names composed of several words, underscore separated lower case letters must be used. Also see **RULE 7.3.4**, **RULE 7.3.5**, **RULE 7.3.6**, and **RULE 7.3.7**.

reason: Improves readability.

example: `ram_addr`

RULE 7.3.15 Active low signal names use *_b*

Where a signal uses active low polarity, it must use the suffix *_b*.

reason: Meaningful, consistent names aid in understanding the design.

example: `enable_data_b, reset_b`

RULE 7.3.16 Clock signal names use *_clk*

Signals that are used for clocking that do not have the word *clock* or *clk* already in their names must use the suffix *_clk*.

reason: Meaningful, consistent names aid in understanding the design.

example: `fifo_transmit_clk`

exception: Signals whose names obviously indicate clocks (*i.e.*, system_clock or clk32m).

RULE 7.3.17 Unconnected output signals use *_nc*

Unused module outputs must use a signal name with the suffix *_nc* (nc: not connected).

reason: When warnings about unconnected signals appear, if the name ends in *_nc*, it is obvous that the signal is known to be unconnected and not an error.

RULE 7.3.18 Signal bundling

Unrelated signals must not be bundled into buses.

reason: Eases understanding of the module.

GUIDELINE 7.3.19 Three-state signal names use *_z*

It is recommended that Tri-state signals use the suffix *_z*.

reason: Meaningful, consistent names aid in understanding the design.

example: `ram_data1_z`

GUIDELINE 7.3.20 State machine signal names use *_next*

It is recommended that state machine next state signals use the suffix *_next*.

reason: Meaningful, consistent names aid in understanding the design.

example: `transmit_next`

GUIDELINE 7.3.21 Test mode signal names use *_test*

It is recommended that test mode signals use the suffix *_test*.

reason: Meaningful, consistent names aid in understanding the design.

example: `parallel_clk_test`

GUIDELINE 7.3.22 Scan enable signal names use *_se*

It is recommended that scan enable signals use the suffix *_se*.

reason: Meaningful, consistent names aid in understanding the design.

GUIDELINE 7.3.23 Analog signal names use *_ana*

It is recommended that analog signals use the suffix *_ana*.

reason: Aids in understanding the design when analog signals are differentiated. Especially useful when using a graphical viewer.

GUIDELINE 7.3.24 Multiple suffix signal name priority

For signals that may contain multiple suffixes, the following priority, from highest to lowest, is recommended:

1. *_b*
2. *_z*
3. *_clk*
4. *_next*

The highest priority suffix is recommended to be the last suffix to the signal.

example: `ram_data1_z_b, receive_clk_b`

GUIDELINE 7.3.25 Parameterized variable names use *_PP*

It is recommended that parameterized variables use the suffix *_PP*. This is applicable to parameters that may change from one design to the next, as opposed to enumerated parameters. This enables the user to easily determine which signals are parameterized.

reason: Meaningful, consistent names aid in understanding the design.

example: `NUM_COLUMNS_PP.`

GUIDELINE 7.3.26 Signal name lengths less than 28 characters

It is recommended that signal name length not exceed 28 characters. Shorter names increase readability. The 28 characters does not include the hierarchy.

reason: Longer name lengths may cause conflicts with tools.

GUIDELINE 7.3.27 Avoid abbreviations except for commonly known acronyms

It is recommended that abbreviations, especially abbreviations with only one letter, be avoided unless it is a commonly known acronym.

reason: Use of meaningful names.

exception: Generally known abbreviations or acronyms, like RAM, and loop counters. Loop counters may be named with a single letter like `I` or `N`, because they represent an index. Some back end tools concatenate all hierarchy names and put a limit on the total name length. In that case abbreviations might be required for hierarchy names. The abbreviations must be explained in a comment.

GUIDELINE 7.3.28 Document abbreviations and additional naming conventions

It is recommended that any abbreviations used in the module be documented. Any naming conventions used in the module that are in addition to the conventions required or recommended in the SRS should be documented.

reason: What may be an obvious abbreviation to the original designer could be obscure when the module is reused.

GUIDELINE 7.3.29 Consistent names throughout the hierarchy

It is recommended that signal or design unit names remain the same throughout the hierarchy of the entire IP. Names associated with multiple instances are recommended to have the name indexed by an integer.

reason: Improves readability, removes confusion, avoids buffer insertion during synthesis.

GUIDELINE 7.3.30 Verilog names are equivalent to documentation names

All signals and blocks in the Verilog RTL that are referenced in the documentation are recommended to maintain the same name as in the documentation.

reason: Simplifies understanding of an HDL model.

7.4 Comments

Comments are required to describe the functionality of a design unit. In particular, comments must supply context information that is not seen locally.

7.4.1 File Headers

Figure 7-1 shows the standard file headers to be used for Verilog files. This template format assures consistency.

RULE 7.4.1 Each file must contain a file header

Every file must contain a header as shown in **Figure 7-1**. All fields must be included, even if the data is N/A.

reason: Provides a standard means of supplying pertinent design information.

RULE 7.4.2 Include file name

The header must include the name of the file.

reason: Provides an easy way to determine what a file contains.

RULE 7.4.3 Include file construct type

The header must include the highest level construct contained in the file.

reason: Provides an easy way to determine what a file contains.

example: `module, macromodule`

RULE 7.4.4 Include point of contact information

Every file header must include the originating department, author, and author's email address.

reason: Required for inquiries beyond the scope of the documentation for the design.

RULE 7.4.5 Include a release history

Header must include release history only for the IP changes checked into the Repository. This information is useful to the integrator. Local release history should not be included in the header.

reason: Required to track the revision history of the design.

```
// +FHDR-------------------------------------------------------------------------
// Copyright (c) 1999, Motorola.
// Motorola Confidential Proprietary
// ------------------------------------------------------------------------------
// FILE NAME :
// TYPE           : TYPE can be module, macromodule
// DEPARTMENT :
// AUTHOR         :
// AUTHOR'S EMAIL :
// ------------------------------------------------------------------------------
// Release history
// VERSION Date AUTHOR DESCRIPTION
// 1.0 6 Oct 98  name
// ------------------------------------------------------------------------------
// KEYWORDS    : General file searching keywords, leave blank if none.
// ------------------------------------------------------------------------------
// PURPOSE     : Short description of functionality
// ------------------------------------------------------------------------------
// PARAMETERS
//     PARAM NAME      RANGE    : DESCRIPTION       : DEFAULT : VA UNITS
// e.g.DATA_WIDTH_PP  [32,16]   : width of the data : 32      :
// ------------------------------------------------------------------------------
// REUSE ISSUES
//   Reset Strategy        :
//   Clock Domains         :
//   Critical Timing       :
//   Test Features         :
//   Asynchronous I/F      :
//   Scan Methodology      :
//   Instantiations        :
//   Other                 :
// -FHDR-------------------------------------------------------------------------
```

Figure 7-1 Verilog File Header

RULE 7.4.6 Include a keyword section

The header must contain a section of the searching keywords. This string may contain a few word synopsis of the module's functionality, or list systems and buses that the module was designed to work with.

reason: Keywords provide a quick searching mechanism and aid the IP integrator in the appropriate selection of blocks. If there are no keywords, the entry should be left blank.

example: `address decoder, coldfire, sbus`

RULE 7.4.7 Include a purpose section

The header must contain a purpose section describing the modules functionality. The purpose must describe *what* the unit provides and not *how*.

reason: Aids understanding of module functionality.

RULE 7.4.8 Include a parameter description

Headers must contain information describing the parameters being used in the construct. The default value must be listed. For Verilog-AMS files, the units of the parameter must also be listed.

reason: Aids understanding of HDL code.

RULE 7.4.9 Document the reset strategy

The reset strategy must be documented, including whether the reset is synchronous or asynchronous, internal or external power-on reset, hard versus soft reset, and whether the module is individually resettable for debug purposes.

reason: Improves readability of the code, and highlights test and synthesis steps that must be taken.

RULE 7.4.10 Document the clock domains

The number of clock domains and clocking strategies must be documented.

reason: Documenting internally generated clocks, or divided down clocks allows better understanding of the code and clocking schemes.

RULE 7.4.11 Document the critical timing

Critical timing including external timing relationships must be documented.

reason: The setup-hold and output timing relationships highlight the test and synthesis steps that must be taken.

RULE 7.4.12 Test features

Any specific test features that are added to the code to speed up testing must be documented.

reason: Improves the ability to understand and test the code once the block is integrated.

example: `parallel clocking`

RULE 7.4.13 Detail asynchronous interfaces

The asynchronous interfaces must be described including the timing relationships and frequency.

reason: Aids understanding of the design, and helps determine if additional synchronization stages are needed for a retargeted application.

RULE 7.4.14 Scan methodology

Notation must be used to indicate what scan style is used.

reason: Aids integration of the design.

example: Mux-D or LSSD

RULE 7.4.15 Document instantiations

Cell and construct instantiations must be listed (see **RULE 7.4.27**).

reason: Indicates areas that must be addressed for technology retarget, and aids understanding of the design hierarchy.

RULE 7.4.16 Use file header boundary tags (+FHDR & -FHDR)

The +FHDR/-FHDR tags must be used to define the boundary of the header information.

reason: Easy way to identify the header. Indicates that the header is a file header.

GUIDELINE 7.4.17 Other header documentation

It is recommended that header include additional pertinent information that is useful to the integrator or that makes code more understandable.

reason: Aids in understanding the design

7.4.2 Additional Construct Headers

Each additional construct within files will also be documented with the following header. The format of the header must match the figure to ensure the ability to parse the header with a software tool. The capitalized keywords in the headers may be used as search points for types of information. This template format assures consistency. **Figure 7-2** contains the header for Verilog functions, user-defined primitives, and tasks.

```
// +HDR -------------------------------------------------------------------------
// NAME  :
// TYPE  :TYPE can be func, task, primitive
// -------------------------------------------------------------------------------
// PURPOSE    : Short description of functionality
// -------------------------------------------------------------------------------
// INSTANTIATES    : Leave blank if none.
// -------------------------------------------------------------------------------
// PARAMETERS
//     PARAM NAME      RANGE   : DESCRIPTION       : DEFAULT : VA UNITS
// e.g. DATA_WIDTH_PP  [32,16] : width of the data : 32      :
// -------------------------------------------------------------------------------
// Other        : Leave blank if none.
// -HDR -------------------------------------------------------------------------
```

Figure 7-2 Verilog Functions, User-defined Primitives and Tasks Header

RULE 7.4.18 Additional constructs in file use a header

All of the additional constructs used in a file must be documented with a header as illustrated (see **Figure 7-2**). All fields must be included, even if the data is N/A.

reason: Provides a standard means of supplying pertinent design information.

RULE 7.4.19 Include construct name

The header must include the name of the additional construct.

reason: Provides an easy way to determine what a file contains.

RULE 7.4.20 Include construct type

Header must include the construct type.

reason: Provides an easy way to determine what a file contains.

RULE 7.4.21 Include a purpose section

Header must contain a purpose section describing the construct functionality. The purpose must describe *what* the unit provides and not *how*.

reason: Aids understanding of construct functionality.

RULE 7.4.22 Include an instantiation list

Headers must contain information detailing what constructs are instantiated within it. This includes modules, tasks, primitives, and functions.

reason: Aids understanding of HDL code dependencies, and achieves completeness.

RULE 7.4.23 Include a parameter description

Headers must contain information describing the parameters being used in the construct. The default value must be listed. For Verilog-AMS files, the units of the parameter must also be listed.

reason: Aids understanding of HDL code.

RULE 7.4.24 Construct header boundary tags (+HDR & -HDR)

The +HDR/-HDR tags must be used to define the boundary of the header information.

reason: Easy way to identify the header. Indicates that the header is a construct header.

GUIDELINE 7.4.25 Other header documentation

It is recommended that header include additional pertinent information that is useful to the integrator or that makes code more understandable.

reason: Aids in understanding the design

7.4.3 Other Comments

RULE 7.4.26 Section comments

Each functional section of the code must be preceded by comments describing the code's intent and function.

reason: Aids understanding of the code.

RULE 7.4.27 Cell instantiation

A comment must be used to explain the functionality of any instantiated cells and why this cell is instantiated and not synthesized. These cells can be from a library, not modeled in an HDL language like Verilog, cells with hidden functionality, or custom implemented cells. For custom instantiation, indicate if the custom code is synthesizable. See **RULE 7.4.15** and **GUIDELINE 7.10.29**.

reason: Instantiated cells limit the technological portability of a design. These cells can also have their functionality hidden. Improves the ability to understand the code.

exception: Not applicable to non-synthesizable blocks (*e.g.*, Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation.

RULE 7.4.28 Document unusual or non-obvious implementations

Unusual or non-obvious implementations must be explained and their limitations documented with a comment.

reason: Improves readability of the code. Purpose and implications of unusual or non-obvious implementations will, in general require an explanation.

RULE 7.4.29 List compiler directives

Compiler directives such as 'ifdef must be listed and their usage described.

reason: Improves understanding and portability of the code.

GUIDELINE 7.4.30 Comment end statements

It is recommended that every end have a comment telling what construct it ends.

reason: Improves readability. Easier to identify the bounds of a construct.

GUIDELINE 7.4.31 Use comments liberally

It is recommended that comments be used liberally throughout the code to describe functionality, design process, and special handling.

reason: Improves understanding of the code.

GUIDELINE 7.4.32 'else, 'ifdef should be commented

The opening 'ifdef declaration should have a comment including the name of the defined item.

7.5 Code Style

Figure 7-3 is an example of good Verilog code format.

```
// +FHDR-------------------------------------------------------------------------
// Copyright (c) 1999, Motorola.
// Motorola Confidential Proprietary
// ------------------------------------------------------------------------------
// FILE NAME : prescaler.v
// TYPE : module
// DEPARTMENT : SoCDT foundry, Austin TX
// AUTHOR : Mike Kentley
// AUTHOR'S EMAIL : r6476c@email.sps.mot.com
// ------------------------------------------------------------------------------
// Release history
// VERSION Date AUTHOR DESCRIPTION
// 1.0 12 Sep tommyk initial version
// 2.0 Nov 98 mkentley Updated for SRS compatibility
// 2.1 Oct 99 mark lancaster Cleaned up prescaler bypass.
//                           Added comments in header.
// ------------------------------------------------------------------------------
// KEYWORDS    : clock divider, divide by 16
// ------------------------------------------------------------------------------
// PURPOSE     : divide input clock by 16.
// ------------------------------------------------------------------------------
// REUSE ISSUES
//   Reset Strategy : Asynchronous, active low system level reset
//   Clock Domains : system_clock, clock_in
//   Critical Timing : N/A
//   Test Features : Prescaler is bypassed when scan_mode is asserted
//   Asynchronous I/F : reset_b
//   Scan Methodology : Mux-D
//   Instantiations : N/A
//   Other : uses synthesis directive to infer a mux to
//   avoid glitching clock_out and clock_out_b
// -FHDR-------------------------------------------------------------------------

module prescaler(
                 clock_out, clock_out_b,
                 clock_in, system_clock,
                 scan_mode, reset_b
                 );

output clock_out; // input clock divided by 16
output clock_out_b; // input clock divided by 16 and inverted

input clock_in; // 32 MHz clock
input system_clock; // system clock
input scan_mode; // scan mode clock
input reset_b; // active low hard reset, synch w/ system_clock

reg [3:0] count; // counter to make clock divider
reg clock_out; // input clock divided by 16
reg clock_out_b; // input clock divided by 16 and inverted
```

```
// 4-bit counter; count[3] is the divide by 16
always @(posedge clock_in or negedge reset_b)
  if (!reset_b)
    count <= 4'b0000; // reset counter
  else if (count == 4'b1111)
    count <= 4'b0000; // roll the counter over
  else
    count <= count + 4'b0001; // increment counter

// Bypass the prescaler during scan testing.  It guarantees
// that the mux will not be optimized away which could
// result in a glitchy test clock.
// Also make sure that the clock_out and clock_out_b are active
// high clocks during scan testing.  This ensures that flops
// connected to clock_out and clock_out_b are all on the rising
// edge of the system clock for test purposes.

always @(scan_mode or system_clock or count)
  case (scan_mode) //insert appropriate synthesis mux inference
                   //directive here
    1'b0 : begin
             clock_out    = count[3];
             clock_out_b  = !count[3];
           end // normal operation clock assign
    1'b1 : begin
             clock_out   = system_clock;
             clock_out_b = system_clock;
           end // scan mode clock assign
  endcase // scan_mode_clock
endmodule // prescaler
```

Figure 7-3 Verilog Coding Format

RULE 7.5.1 Write code in a tabular format

Code must be written in a tabular manner (*i.e.*, code items of the same kind are aligned).

reason: Improves readability. When writing a block (begin, case, if statements etc.), it is useful to complete the frame first, in particular to put the *end* of the block down with the correct indentation.

RULE 7.5.2 Use two to four space code indentation

A constant indentation of two to four spaces must be used for code alignment. Do not use tab stops. Use spaces and empty lines to increase the readability of the code. The tab key of the text editor may be mapped to insert spaces.

reason: Improves readability. Tab stops must not be used because they are represented differently from one system to another.

RULE 7.5.3 One Verilog statement per line

One line must not contain more than one statement. Do not concatenate multiple semicolon separated Verilog statement on the same line.

reason: Improves readability. Easier to parse code with a design tool.

example: use:

```
upper_en = (p5type && xadr1[0]);
lower_en = (p5type && !xadr1[0]);
```

do not use:

```
upper_en = (p5type && xadr1[0]); lower_en = (p5type && !xadr1[0]);
```

exception: Comments are allowed on the same line as a Verilog statement.

RULE 7.5.4 Use one line comments

One line comments (//) must be used. Do not use multi-line (/*...*/) comments.

reason: Improves readability and improves code parsing.

RULE 7.5.5 Explicitly indicate port type

Explicit port typing indication must be used. One port per line must be declared.

reason: Improves readability and understanding of the code, as well as parsing the code with scripts.

example: use:

```
input a;
input b;
```

do not use:

```
input a, b;
input a,
      b;
```

GUIDELINE 7.5.6 Keep line length less than 80

It is recommended that line length does not exceed 80 characters.

reason: Improves readability, and avoids inadvertent line wraps.

GUIDELINE 7.5.7 Port declaration order

It is recommended that ports be declared in the same order as in the port list.

reason: Improves readability

GUIDELINE 7.5.8 Comment port listings

It is recommended that a descriptive comment follow each port listing, preferably on the same line.

reason: Improves readability.

7.6 Module Partitioning & Reusability

RULE 7.6.1 Powered down signals

An input pin that is driven from a source whose power supply can be powered down must either be logically gated to it's inactive level, using a nand gate or a nor gate or handled by the library. The input must be controlled so that when the source signal's power supply is powered down the input is in a known state.

reason: Avoid propagating unknowns into the block when the signal source is powered down.

RULE 7.6.2 No accesses to variable/signals outside the scope of a module

Procedures, tasks, and functions must not modify signals or variables not passed as parameters into the module. Verilog users must not use a hierarchical signal reference to read or modify a signal.

reason: Increases readability, and eases debugging. Improves adaptability and reuse of sub-blocks of the design.

exception: Non-circuit purposes, for example behavioral modeling and testbenches.

RULE 7.6.3 'include statements are not allowed

'include statements must not be used.

reason: Ease portability by avoiding explicit path names.

exception: Model wrappers around memory structures. Not applicable to non-synthesizable blocks (*e.g.*, Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation. Not applicable to the Verilog-AMS disciplines.h and constants.h files. See **RULE 7.6.4**.

RULE 7.6.4 Standard discipline and constants files are to be used in Verilog-AMS files

Standard Verilog-AMS header files must be used. These files are defined in **A.1 Example Header Files**.

reason: Consistent definitions of natures and disciplines eases portability.

RULE 7.6.5 Simulation tasks are not allowed

No waves, checkers, force statements or other simulation related tasks are allowed in the HDL.

reason: Eases understanding and portability.

RULE 7.6.6 Mask plug usage

Mask plugs must be placed outside of the topmost module.

reason: Re-synthesis will not be required for base changes.

GUIDELINE 7.6.7 Partition clocks into separate block

If you must use a gated clock, internally generated clocks, or use both edges of a clock, the clock generation circuitry must be kept in a separate module at the top level of the IP module or at the same logical level in the hierarchy as the block to which the clocks apply.

reason: Eases test strategy generation, and limits exceptions to the coding standards to a small module. It also improves the portability of the code to a different end use clocking scheme.

exception: Not applicable to non-synthesizable blocks (*e.g.*, Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation.

GUIDELINE 7.6.8 Partition clock domains

It is recommended that separate clock domains be partitioned into separate blocks, as shown in **Figure 7-4**. The synchronization logic between the two domains is recommended to also be partitioned into a separate block.

reason: Eases synthesis. Avoids over-constraining one clock domain.

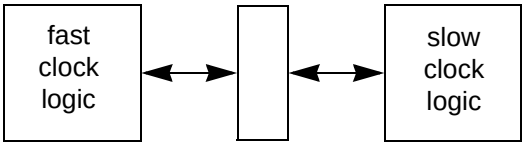

Figure 7-4 Clock Domain Partitioning

GUIDELINE 7.6.9 Minimize interface signals

It is recommended that the design be partitioned in a way to minimize the interface signals.

reason: Reduces the size of an interface boundary.

GUIDELINE 7.6.10 Match physical and logical boundaries

It is recommended that the design be partitioned such that the module boundaries match the physical boundaries. Refer to Physical Standards section.

reason: Improves adaptability to different applications. Simplifies understanding and debugging.

GUIDELINE 7.6.11 Register all module outputs

It is recommended that all module outputs be registered.

reason: Simplifies the timing interface to other modules and the synthesis process.

exception: Not applicable to non-synthesizable modules (*e.g.*, Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation.

GUIDELINE 7.6.12 Partition application specific code from general

It is recommended that the application specific (e.g, bus interface) parts of the code be partitioned from the more general portion of the code.

reason: Improves Adaptability to different applications.

GUIDELINE 7.6.13 Partition speed critical logic

It is recommended that speed critical logic be partitioned into its own block.

reason: Avoids synthesis problems and eases constraints and scripting.

exception: Not applicable to non-synthesizable blocks (*e.g.*, Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation.

GUIDELINE 7.6.14 Partition random logic

It is recommended that random logic be partitioned from data path logic.

reason: Avoids synthesis problems and eases constraints and scripting.

exception: Not applicable to non-synthesizable blocks (*e.g.*, Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation.

GUIDELINE 7.6.15 Partition asynchronous logic

It is recommended that asynchronous logic be partitioned from synchronous logic.

reason: Avoids synthesis problems and eases constraints and scripting.

exception: Not applicable to non-synthesizable blocks (*e.g.*, Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation.

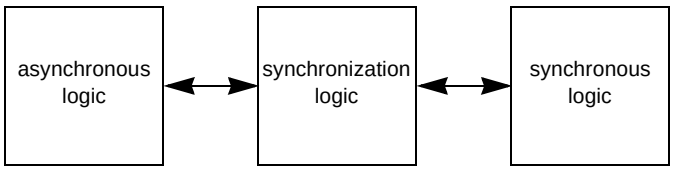

Figure 7-5 Partition Asynchronous Logic

GUIDELINE 7.6.16 Partition state machines

It is recommended that the FSMs be coded into their own block. Separate the state machine code into two processes, one for the combinational logic and one for the sequential logic.

reason: Avoids synthesis problems and eases constraints and scripting.

GUIDELINE 7.6.17 Do not mix structural and behavioral RTL code within a construct

It is recommended that the model partitioning isolate the structural code from the behavioral RTL code. Each construct must be either purely structural or purely behavioral RTL. Structural code is a line of code containing only connectivity information.

reason: This can result in synthesis problems and limitations.

GUIDELINE 7.6.18 Partition BIST

It is recommended that any BIST logic be coded into its own block.

reason: Eases understanding, portability, and constraints.

7.7 Modeling Complex Behavior

RULE 7.7.1 Global distribution nets

Buffering all global distribution nets on the chip (*e.g.*, clock or scan_enable) must be based on net load and placement. Do not hard code buffer trees in the RTL.

reason: Buffering global nets without considering net load and placement (*i.e.*, based solely on connectivity information) may result in serious timing problems.

exception: Not applicable to non-synthesizable blocks (*e.g.*, Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation. Not applicable to logic whose timing and/or sizing specification constraints cannot be met with synthesis.

RULE 7.7.2 Synchronize asynchronous interface signals

If clocks are available, asynchronous interface signals must be synchronized as near to the interface boundary as possible. Synchronization must be performed to avoid metastability (*e.g.*, use double registering).

reason: To limit asynchronous signals to a minimum (Asynchronous design practice is not yet adequately supported by design tools.). Double registering avoids metastability hazards.

note: Pay particular attention to the interface where there is a frequency difference. For example, a lower frequency clock domain cannot guarantee to receive a signal of single period width from a higher frequency clock domain. The higher frequency domain must supply signals with the following minimum active period:
$t_{active} = t_{slow} + t_{setup} + t_{hold}$, where:
t_{active} is the minimum active period of a interfacing signal
t_{slow} is the clock period of the receiving clock domain that has the low frequency
t_{setup} is the setup time of the receiving D-type flip-flop
t_{hold} is the hold time of the receiving D-type flip-flop
t_{active} must take into account the different propagation delay paths for the interfacing signal. In addition skew must be also taken into consideration.
From probability theory, the mean time between the failure (MTBF) of the output to be resolved within some time is given by
MTBF = $(1/ f_{c*}f_d)$ e**-(t_f / τ_r)
where:
τ_r = the time constant of resolution of the latch
f_c = the frequency of the clock
f_d = the frequency of the data
t_f = the time after the change in the clock by which the latch output must be resolved

exception: Not applicable to non-synthesizable blocks (*e.g.*, Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation.

RULE 7.7.3 Use technology independent code for non-inferred blocks

A model of an instantiated non-inferred block (*i.e.*, custom, not intended to be synthesized) must be written in a technology independent coding style.

reason: Allows easy retargetability to a new process technology or hardware emulation box.

RULE 7.7.4 Gated clock enables

If gated clocks are used there cannot be any glitches during the clocking time.

reason: Although gated clocks reduce power consumption, glitching can occur on the clocks due to the clock enable signal setup/hold to the active edge of the clock, causing faulty operation. Refer to **GUIDELINE 7.6.7** and **RULE 10.7.29**.

exception: Asynchronous interfaces, but the timing must be strictly defined.

GUIDELINE 7.7.5 Document gated clock usage

It is recommended that gated clock usage be documented in the code.

reason: Eases understanding of the HDL.

GUIDELINE 7.7.6 Asynchronous inputs to a register clock

Connecting asynchronous inputs to a register clock should be avoided.

reason: To avoid metastability problems on the data being registered.

exception: An asynchronous clock may be used to latch the asynchronous data into a holding register while the clock is being synchronized.

GUIDELINE 7.7.7 Reset all storage elements

It is recommended that all storage elements in a control path be initialized, as appropriate.

reason: For storage elements that are not reset, simulation results may be simulator dependent (Different simulators assume different initial values, (*i.e.*, "0" or "X").

GUIDELINE 7.7.8 Use synchronous design practices

It is recommended to follow synchronous design practice whenever possible. Asynchronous design circuitry should only be used when unavoidable.

reason: Design tools do not adequately support the development of asynchronous designs. Reliable timing verification, including the detection of glitches and hazards, will in general require extensive simulation with SPICE, which is expensive and time consuming.

exception: Not applicable to non-synthesizable blocks (*e.g.*, Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation.

GUIDELINE 7.7.9 Document SR Latch Usage

It is recommended that SR latch usage be documented in the code.

reason: Ease understanding of the HDL.

7.8 General Coding Techniques

RULE 7.8.1 Expression in condition must be a 1-bit value.

The condition in an `if, or` or a `while` statement must be an expression that results in a 1-bit value.

reason: Makes the code easier to read.

example: A signal called bus_is_active is set to 1 whenever bus, a multi-bit value, has a value other than 0.

a) `if (bus) bus_is_active = 1;`
In this example `bus` is a multi-bit value.
b) `if (bus > 0) bus_is_active = 1;`
Here the condition results in a 1-bit expression, which is intuitively easier to read.

RULE 7.8.2 Do not assign signals to *x*

Signals must not be assigned to *x*. Known legal signal values must be assigned to all signals.

reason: Avoids *x* propagation through the circuitry.

RULE 7.8.3 Do not infer latches in functions

Latches must not be inferred in any function call

reason: Functions always synthesize to combinational logic.

GUIDELINE 7.8.4 Operand sizes should match

The operands of an operation are not recommended to differ in size.

reason: With different operand sizes the operand is not explicitly defined, but depends on how Verilog resolves the size differences. Verilog allows this since it is not a highly typed language.

example: `wire [63:0] some_signal;`

```
some_signal <= 1;
```

Tools interpret the 1 as a 32 bit integer value. Due to this fact the bits 63 to 32 are not affected by this assignment.

GUIDELINE 7.8.5 Use parentheses in complex equations

It is recommended that parentheses be used to force the order of operations.

reason: Large equations without parentheses depend on the order preference of the language to determine the functionality of the equations. Parentheses explicitly order the operations of the equations, clearly conveys the functionality, and implies structure for synthesis.

7.9 Standards for Structured Test Techniques

These coding standards are intended to obtain the maximum amount of test coverage. More complete guidelines on implementing scan and “scan friendly” circuitry are detailed in the SCAN Design Standards.

RULE 7.9.1 Use additional logic for scanning three-state devices

Additional logic must be used to prevent signal contention. All multi-sourced signals/buses must be driven one-hot mutually-exclusive during the launch and capture test sequence.

reason: To prevent propagation of x’s and scan vector mismatches.

example: 1-hot driver three-state device enables during scan shift

RULE 7.9.2 Allow PLL bypass

ATPG tools must have clock control from an input pin. If a PLL is used for on-chip clock generation, then the means of by-passing or disabling the PLL must be documented.

reason: The PLL bypass makes testing and debug easier, and facilitates the use of hardware modelers.

RULE 7.9.3 Scan support logic for gated clocks

Gated clocks usage must be accompanied by scan support logic.

reason: To avoid ATPG loss of fault control.

RULE 7.9.4 Externally control asynchronous reset of storage elements

Asynchronous resets for initialization or power-up may only be used when they are controlled via a primary input in test mode. The asynchronous resets must be synchronously released after an internal clock edge or the reset must be reclocked in the module to avoid reset recovery problems. Refer to **Section 10 Design-for-Test**.

reason: Scan chains can be generated for storage elements only if the reset is synchronous or if an asynchronous reset is controlled via a primary input.

GUIDELINE 7.9.5 Segregate opposing phase clocks

Logic which uses both the positive and the negative edges of the clock is recommended to have segregated clocks (*i.e.*, each clock is a separate input to the block).

reason: Eases scan insertion.

RULE 7.9.6 Scan Support Logic for Latches

If latches are connected to the clock, they must be transparent during scan.

reason: To avoid fault coverage loss due to the latches blocking up-stream and down-stream logic.

example: Refer to **Figure 7-6** for a two-phase implementation

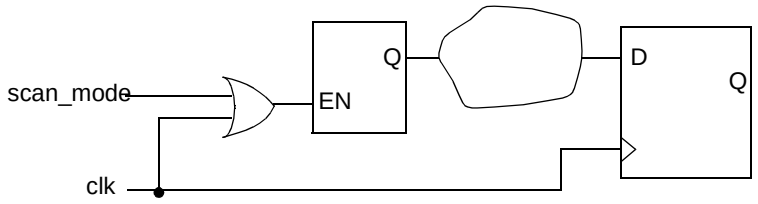

Figure 7-6 Scan Support for Mixed Latch/Flip-Flop Designs

RULE 7.9.7 Master/slave latch clocking

If master/slave latches are inferred in the code, clocks of a given phase must drive the corresponding stage of the latch. No phase of the clock can update both master and slave latches during test mode.

reason: Test tools do not understand timing and loss of coverage will occur.

GUIDELINE 7.9.8 Synchronously reset storage elements

It is recommended that synchronous resets be used wherever possible.

reason: Eases scan chain insertion and test.

7.10 General Standards for Synthesis

This section describes the language independent standards which are applicable to the Verilog. There has been a special effort to make as many of the standards as language and tool independent as possible. The synthesis tool specific coding guidelines should also be referenced in addition to the standards specified in this and other sections. It should be noted that the following standards are not applicable to non-synthesizable blocks (*e.g.*, Bus Functional Models, Bus Monitors, data path modules, Analog Behavioral Models, test benches, or behavioral modules) unless they are intended to be synthesized for emulation.

RULE 7.10.1 Complete always sensitivity list

All always blocks inferring combinational logic or a latch must have a sensitivity list. The sensitivity list must contain all input signals.

reason: Synthesis will create a structure that depends on all values read regardless of the sensitivity list, which can lead to potential mismatches between behavioral and gate-level simulation.

RULE 7.10.2 One clock per always block

Only one clock per Verilog always block must be used in a synchronous process.

reason: This is required to restrict each process to a single type of memory-element inference.

RULE 7.10.3 Wait statements and # delay statements are not allowed.

Wait statements, both explicit and implicit, must not be used in the design.

reason: The wait statement and the # delay are generally not supported by the RTL to gate level synthesis process. These constructs should be avoided to enable effective synthesis.

RULE 7.10.4 Specify combinational expressions completely

Conditional expressions must be specified completely (*i.e.*, assign a value to a variable or signal under all conditions).

reason: Synthesis tools infer memory elements if a combinational expression is not completely specified.

example: Memory elements are not inferred due to a completely specified case statement.

```
always @(signal_names)
  case (signal_names)
    3'b000,
    3'b001 : output = 4'b0000;
    3'b010 : output = 4'b1010;
    3'b011,
    3'b100,
    3'b101 : output = 4'b0101;
    3'b110,
    3'b111 : output = 4'b0001;
  endcase
```

example: The following `if` statement infers a memory element because there is no `else` clause.

```
always @(signal)
  begin
    if  (signal  = 1'h1)
  output = 4'b0;
   end
```

RULE 7.10.5 No `disable` in looping constructs

The use of the `disable` command in looping constructs is prohibited.

reason: It is good programming style to have defined ends for loops and to prohibit jumps to the next loop iteration.

exception: Allowed in testbench constructs.

RULE 7.10.6 Do not use the initial statement

Reset functionality along with signal and variable initialization must be modeled explicitly. Do not use the initial construct.

reason: The synthesized gate-level netlist will not use the initialization construct which will result in simulation mismatches between the behavioral and gate-level models.

exception: Allowed in testbench constructs.

RULE 7.10.7 Expressions are not allowed in port connections

Expressions must not be used in port connections.

reason: May result in glue logic between blocks. Eases understanding of the HDL.

exception: Bus concatenations are allowed.

RULE 7.10.8 Verilog primitives are prohibited

Verilog primitive must not be used.

exception: Eases understanding of the HDL.

RULE 7.10.9 Use non-blocking assignments (<=) in edge-sensitive constructs

Non-blocking assignments (<=) must be used in edge-sensitive sequential blocks. Blocking assignments (=) are not allowed.

reason: Use of blocking assignments in edge-sensitive sequential code can result in mismatches between pre and post-synthesis simulations.

example: Edge-sensitive code written with non-blocking assignments:

```
always @(posedge clk) begin
regb <= rega;
rega <= data;
end
```

example: Code that may result in pre and post-synthesis simulation mismatches:

```
always @(posedge clk) begin
rega = data;
regb = rega;
```

```
end
```

In pre-synthesis simulation data runs through to regb at each posedge clock. This will synthesize to a shift register, where data will not run through to regb on the same posedge of the clock.

RULE 7.10.10 Declare all internal wires in one section

The internal wire declaration must follow the port I/O declarations at the top of the module.

reason: Eases understanding of code.

RULE 7.10.11 Internal wires must be declared

All internal wires must be declared instead of implied.

reason: Although Verilog can handle implied wires, all internal wires must be declared to avoid confusion.

RULE 7.10.12 Use explicit port references in module instantiation

Modules must be instantiated with explicit port references instead of by port position. Expressions in port connections are not allowed.

reason: Module instantiation will explicitly show port connections, and will not result in extra logic at the instantiating level which improves readability and adaptability.

example:
```
block block_1 ( .signal_a(signal_a),
                .signal_b(signal_b) );
```

RULE 7.10.13 Drive all unused module inputs

All unused module instance inputs must be actively driven by some other signal or by a fixed logic 0 or 1.

reason: All ports appear in the module instantiation. None are hidden or forgotten.

RULE 7.10.14 Connect unused module outputs

All unused module instance outputs must be connected (see **RULE 7.3.17**).

reason: Warnings about unconnected ports or missing ports in the instance statement are eliminated.

RULE 7.10.15 Use of `casex` is not allowed

`case` or `casez` must be used for all case statements.

reason: Casex treats the *X* and *Z* states as don't cares in synthesis, which can result in different simulation behavior pre- and post-synthesis. The *X* state must be generated and handled in such a way to determine if the true don't care conditions will affect the operation of the design, rather than cover up a problem.

RULE 7.10.16 'define usage

If a 'define statement is used within a module, the macro name must be undefined using 'undef in the same module.

exception: Since the 'defines have no scope, they must remain associated with the intended code. Maintaining the name association and defining the macro in the source code eases reuse.

RULE 7.10.17 Do not use nested 'ifdefs

exception: Nested 'ifdef declarations must not be used.

GUIDELINE 7.10.18 Use parameters for state encodings

It is recommended that enumerated parameters be used to encode the different states of a state machine.

reason: This eases retargeting to different state machine implementations, for example changing from an encoded 1-hot style to gray code.

example:

```
parameter [1:0] // synthesis enum state_info
RESET_STATE = 2'b00,
TX_STATE = 2'b01,
RX_STATE = 2'b10,
ILLEGAL_STATE = 2'b11;
```

GUIDELINE 7.10.19 Use a cycle wide enable signal for signals with multicycle paths

It is recommended that a signal propagated through a multicycle path be qualified by a one clock cycle wide enable signal at the receiving register.

reason: Failure to do so will result in potential metastability problems at the output of the receiving register.

note: Using multicycle paths has severe implications on test and their usage must be carefully evaluated and completely documented.

GUIDELINE 7.10.20 Model three-state devices explicitly

It is recommended that three-state devices be modeled explicitly with *z* assignments and multiple concurrent assignments. All select combinations for three-state devices are recommended to be defined with mutually exclusive logic.

reason: Multiple concurrent assignments and assignment of value *z* results in the inference of three-state buffers. No bus contention must be ensured. Three-state devices may cause test problems (see **Section 10 Design-for-Test**).

GUIDELINE 7.10.21 Top level glue logic is not allowed

It is recommended that gates not be instantiated or inferred at the top level of the design hierarchy.

reason: Synthesis results are limited because the top level logic cannot be combined for optimization.

GUIDELINE 7.10.22 Avoid ports of type inout

It is recommended to use ports of type input or output. Ports of type inout (bidirectional) should be avoided.

reason: Bidirectional implementations may cause contention problems. Avoiding bidirectionals also eases synthesis and test insertion.

GUIDELINE 7.10.23 Preserve relationships between constants

If a constant is dependent on the value of another constant, it is recommended the dependency be shown in the definition. Where a ‘define macro defines an arithmetic or logical expression, it should be enclosed in parentheses.

reason: Increased adaptability, as code changes required for adaptation are reduced.

example: preferred:

```
‘define DATA_WORD 8
‘define DATA_LONG (4 * ‘DATA_WORD)
```

versus:

```
‘define DATA_WORD 8
‘define DATA_LONG 32
```

GUIDELINE 7.10.24 Wires must have descriptive contents

It is recommended that a descriptive comment follow each wire declaration (preferably on the same line).

reason: Improves readability.

GUIDELINE 7.10.25 Blocking assignment usage

In pure latch designs, blocking assignments may be used. It is recommended that combinational logic use blocking assignments.

reason: Blocking assignments typically simulate faster than non-blocking. However, to avoid dependencies on execution order, care must be taken when inferring latches.

exception: Latches in mixed latch/FF designs should be written with non-blocking assignments.

GUIDELINE 7.10.26 Default case assignments in case statements

It is recommended that default case assignments be used for all combinational logic case statement descriptions.

reason: It may be possible to decode unexpected combinations of the case selects, resulting in pre- and post-synthesis simulation mismatches.

note: Additional logic may be inferred using the default case.

example: The bus is assigned to an illegal value if a case select combination which is not explicitly decoded is selected, e.g. 3'b000.

```
casez (sbus_sel[2:0])
  3'b100: sbus_data[31:0] = in_bus[31:0];
  3'b?1?: sbus_data[31:0] = 32'b1;
  3'b?01: sbus_data[31:0] = data_rd[31:0];
  default: sbus_data[31:0] = 32'hdead;
endcase
```

GUIDELINE 7.10.27 Use of the *full_case* synthesis directive

It is recommended that the *full_case* synthesis directive only be used in case statements describing a state machine where the case selects consist only of the state vector.

reason: The full_case directive informs the synthesis tool that the assignments to the unused cases are don't cares. This may result in pre- and post-synthesis simulation mismatches.

example: The enable input may be optimized away:

```
case {(en, a)} // synthesis full_case directive
  3'b1_00: y[a] = 1'b1;
  3'b1_01: y[a] = 1'b1;
  3'b1_10: y[a] = 1'b1;
  3'b1_11: y[a] = 1'b1;
endcase
```

example: State machine:

```
case (cur_state) // synthesis full_case directive
  idle : if (frame) nxt_state = busy;
  busy : begin
    if (!frame && !irdy) nxt_state = idle;
    if (( frame && hit && !terminate )
       || ( frame && hit && terminate && ready )
       || ( irdy && hit && !terminate )
       || ( irdy && hit && terminate && ready )) nxt_state <= xdata;
    if (( frame && hit && terminate && !ready )
       || ( irdy && hit && terminate && !ready )) nxt_state <= backoff;
    end
  xdata :  begin
    if (frame && stop && !trdy ) nxt_state <= backoff;
    if (( !frame && trdy ) || ( !frame && stop )) nxt_state <= turna;
    end
  backoff :  if ( !frame) nxt_state <= turna;
  turna :   begin
    if ( !frame) nxt_state <= idle;
    if (frame && !hit) nxt_state <= busy;
    if ((frame && hit && !terminate)
       || (frame && hit && terminate && ready)) nxt_state <= xdata;
    if (frame && hit && terminate && !ready) nxt_state <= backoff;
    end
//synthesis translate off directive
  default: $display ($time,,"WARNING: unknown state, \
     cur_state=%h",cur_state);
//synthesis translate on directive
endcase
```

GUIDELINE 7.10.28 Embedded synthesis scripts are not allowed

It is recommended that embedded synthesis scripts not be used. If embedded scripts are used, they must be documented as to purpose and functionality.

reason: Later reuse of the blocks may have different synthesis goals and embedded scripts may cause future synthesis runs to return poor results. In addition, further releases of the synthesis tool may obsolete the embedded commands.

exception: Judicious use of synthesis directives (*e.g.*, translate_off and translate_on). These must be documented in the reuse issues section of the header.

GUIDELINE 7.10.29 Instantiate Gates Indirectly

If a gate must be instantiated, a technology independent gate, or an indirect instantiation in a function is recommended to be used (see **RULE 7.4.27**).

reason: Eases reuse and portability across libraries.

Appendix A

A.1 Example Header Files

```
// +FHDR-------------------------------------------------------------------------
// Copyright (c) 1999, Motorola.
// Motorola Confidential Proprietary
// ------------------------------------------------------------------------------
// FILE NAME : disciplines.h
// TYPE           : verilog-ams discipline include file
// DEPARTMENT :
// AUTHOR         :
// AUTHOR'S EMAIL :
// ------------------------------------------------------------------------------
// Release history
// VERSION Date AUTHOR DESCRIPTION
// 1.1 01/08/98 rice initial version
// 1.2 01/20/98 aisola pdated abstol in natures for
//                    changes & flux & correct typos
// ------------------------------------------------------------------------------
// KEYWORDS    : verilog-ams, discipline
// ------------------------------------------------------------------------------
// PURPOSE     : define verilog-ams disciplines
// ------------------------------------------------------------------------------
// REUSE ISSUES
//   Reset Strategy        :
//   Clock Domains         :
//   Critical Timing       :
//   Test Features         :
//   Asynchronous I/F      :
//   Scan Methodology      :
//   Instantiations        :
//   Other                 :
// -FHDR-------------------------------------------------------------------------

`ifdef DISCIPLINES_H
`else
`define DISCIPLINES_H 1
//
// Natures and Disciplines
//
//
// Default absolute tolerances may be overriden by setting the
// appropriate _ABSTOL prior to including this file
//
// Electrical
// Current in amperes
nature Current
          units      = "A";
          access     = I;
          idt_nature = Charge;
`ifdef CURRENT_ABSTOL
          abstol     = `CURRENT_ABSTOL;
`else
          abstol     = 1e-12;
`endif
```

```
endnature
// Charge in coulombs
nature Charge
          units      = "coul";
          access     = Q;
          ddt_nature = Current;
`ifdef CHARGE_ABSTOL
          abstol     = `CHARGE_ABSTOL;
`else
          abstol     = 1e-16;
`endif
endnature
// Potential in volts
nature Voltage
          units      = "V";
          access     = V;
          idt_nature = Flux;
`ifdef VOLTAGE_ABSTOL
          abstol     = `VOLTAGE_ABSTOL;
`else
          abstol     = 1e-6;
`endif
endnature
// Flux in Webers
nature Flux
          units      = "Wb";
          access     = Phi;
          ddt_nature = Voltage;
`ifdef FLUX_ABSTOL
          abstol     = `FLUX_ABSTOL;
`else
          abstol     = 1e-14;
`endif
endnature
// Conservative discipline
discipline electrical
          potential    Voltage;
          flow         Current;
enddiscipline
// Signal flow disciplines
discipline voltage
          potential    Voltage;
enddiscipline
discipline current
          potential    Current;
enddiscipline

// Magnetic
// Magnetomotive force in Ampere-Turns.
nature Magneto_Motive_Force
          units      = "A*turn";
          access     = MMF;
`ifdef MAGNETO_MOTIVE_FORCE_ABSTOL
          abstol     = `MAGNETO_MOTIVE_FORCE_ABSTOL;
`else
          abstol     = 1e-12;
`endif
endnature
// Conservative discipline
```

```
discipline magnetic
          potential    Magneto_Motive_Force;
          flow         Flux;
enddiscipline

// Thermal
// Temperature in Celsius
nature Temperature
          units      = "C";
          access     = Temp;
`ifdef TEMPERATURE_ABSTOL
          abstol     = `TEMPERATURE_ABSTOL;
`else
          abstol     = 1e-4;
`endif
endnature
// Power in Watts
nature Power
          units      = "W";
          access     = Pwr;
`ifdef POWER_ABSTOL
          abstol     = `POWER_ABSTOL;
`else
          abstol     = 1e-9;
`endif
endnature
// Conservative discipline
discipline thermal
          potential    Temperature;
          flow         Power;
enddiscipline

// Kinematic
// Position in meters
nature Position
          units      = "m";
          access     = Pos;
          ddt_nature = Velocity;
`ifdef POSITION_ABSTOL
          abstol     = `POSITION_ABSTOL;
`else
          abstol     = 1e-6;
`endif
endnature
// Velocity in meters per second
nature Velocity
          units      = "m/s";
          access     = Vel;
          ddt_nature = Acceleration;
          idt_nature = Position;
`ifdef VELOCITY_ABSTOL
          abstol     = `VELOCITY_ABSTOL;
`else
          abstol     = 1e-6;
`endif
endnature
// Acceleration in meters per second squared
nature Acceleration
          units      = "m/s^2";
```

```
          access     = Acc;
          ddt_nature = Impulse;
          idt_nature = Velocity;
`ifdef ACCELERATION_ABSTOL
          abstol     = `ACCELERATION_ABSTOL;
`else
          abstol     = 1e-6;
`endif
endnature
// Impulse in meters per second cubed
nature Impulse
          units      = "m/s^3";
          access     = Imp;
          idt_nature = Acceleration;
`ifdef IMPULSE_ABSTOL
          abstol     = `IMPULSE_ABSTOL;
`else
          abstol     = 1e-6;
`endif
endnature
// Force in newtons
nature Force
          units      = "n";
          access     = F;
`ifdef FORCE_ABSTOL
          abstol     = `FORCE_ABSTOL;
`else
          abstol     = 1e-6;
`endif
endnature
// Conservative disciplines
discipline kinematic
          potential    Position;
          flow         Force;
enddiscipline
discipline kinematic_v
          potential    Velocity;
          flow         Force;
enddiscipline
// Rotational
// Angle in radians
nature Angle
          units      = "rads";
          access     = Theta;
          ddt_nature = Angular_Velocity;
`ifdef ANGLE_ABSTOL
          abstol     = `ANGLE_ABSTOL;
`else
          abstol     = 1e-6;
`endif
endnature
// Angular Velocity in radians per second
nature Angular_Velocity
          units      = "rads/s";
          access     = Omega;
          ddt_nature = Angular_Acceleration;
          idt_nature = Angle;
`ifdef ANGULAR_VELOCITY_ABSTOL
          abstol     = `ANGULAR_VELOCITY_ABSTOL;
```

```
`else
          abstol     = 1e-6;
`endif
endnature
// Angular acceleration in radians per second squared
nature Angular_Acceleration
          units      = "rads/s^2";
          access     = Alpha;
          idt_nature = Angular_Velocity;
`ifdef ANGULAR_ACCELERATION_ABSTOL
          abstol     = `ANGULAR_ACCELERATION_ABSTOL;
`else
          abstol     = 1e-6;
`endif
endnature
// Force in newtons
nature Angular_Force
          units      = "n/m";
          access     = Tau;
`ifdef ANGULAR_FORCE_ABSTOL
          abstol     = `ANGULAR_FORCE_ABSTOL;
`else
          abstol     = 1e-6;
`endif
endnature
// Conservative disciplines
discipline rotational
          potential    Angle;
          flow         Angular_Force;
enddiscipline
discipline rotational_omega
          potential    Angular_Velocity;B
          flow         Angular_Force;
enddiscipline

`endif
```

```
// +FHDR-------------------------------------------------------------------------
// Copyright (c) 1999, Motorola.
// Motorola Confidential Proprietary
// ------------------------------------------------------------------------------
// FILE NAME :constants.h
// TYPE           : verilog-ams constant include file
// DEPARTMENT :
// AUTHOR         :
// AUTHOR'S EMAIL :
// ------------------------------------------------------------------------------
// Release history
// VERSION Date AUTHOR DESCRIPTION
// 1.0 01/08/98 rice initial version
// ------------------------------------------------------------------------------
// KEYWORDS    : verilog-ams, constants
// ------------------------------------------------------------------------------
// PURPOSE     : define verilog-ams constants
// ------------------------------------------------------------------------------
// REUSE ISSUES
//   Reset Strategy        :
//   Clock Domains         :
//   Critical Timing       :
//   Test Features         :
//   Asynchronous I/F      :
//   Scan Methodology      :
//   Instantiations        :
//   Other                 :
// -FHDR-------------------------------------------------------------------------

// Mathematical and physical constants

`ifdef CONSTANTS_H
`else
`define CONSTANTS_H 1

// M_ is a mathmatical constant

`define   M_E          2.7182818284590452354
`define   M_LOG2E      1.4426950408889634074
`define   M_LOG10E     0.43429448190325182765
`define   M_LN2        0.69314718055994530942
`define   M_LN10       2.30258509299404568402
`define   M_PI         3.14159265358979323846
`define   M_TWO_PI     6.28318530717958647652
`define   M_PI_2       1.57079632679489661923
`define   M_PI_4       0.78539816339744830962
`define   M_1_PI       0.31830988618379067154
`define   M_2_PI       0.63661977236758134308
`define   M_2_SQRTPI   1.12837916709551257390
`define   M_SQRT2      1.41421356237309504880
`define   M_SQRT1_2    0.70710678118654752440

// P_ is a physical constant

// charge of electron in coulombs
`define   P_Q          1.6021918e-19

// speed of light in vacuum in meters/sec
`define   P_C          2.997924562e8
```

```
// Boltzmann's constant in joules/kelvin
`define   P_K          1.3806226e-23

// Planck's constant in joules*sec
`define   P_H          6.6260755e-34

// permitivity of vacuum in farads/meter
`define   P_EPS0       8.854187923944200139 68e-12

// permeability of vacuum in henrys/meter
`define   P_U0         (4.0e-7 * `M_PI)

// zero celsius in kelvin
`define   P_CELSIUS0   273.15

`endif
```

Standard End Sheet

FINAL PAGE OF
42
PAGES