Verilog HDL Coding

Semiconductor Reuse Standard

IPMXDSRSHDL0001
SRS V3.2

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.
3.0	30 APR 2001	SoC-IP Design Systems	Change summary location: http://socdt.sps.mot.com/ddts/ddts_main
3.0.1	01 DEC 2001	SoC&IP	Edit
3.0.2	15 MAR 2002	SoC&IP	Changed from MCP to MIUO; Changed Motorola font batwing to batwing gif.
3.1	1 NOV 2002	SoC&IP	Changed to reflect changes to SRS V3.1.
3.1.1	1 APR 2003	SoC&IP	Changed to reflect changes to SRS V3.1.1; added eight new paragraph tags
3.2	01 FEB 2005	DEO	Added updates for SRS V3.2.

Table of Contents

Section 7 Verilog HDL Coding

List of Figures

List of Tables

Rule and Guideline Reference

Introduction

Reference Information

Naming Conventions

R 7.3.1 At most one module per file
R 7.3.2 File naming conventions
R 7.3.3 Separate analog, digital, and mixed-signal Verilog files
R 7.3.4 HDL Code items naming convention
R 7.3.5 Document abbreviations and additional naming conventions
R 7.3.6 Global text macros include module name
R 7.3.7 Instance naming conventions
R 7.3.8 Signal naming convention - suffixes
R 7.3.9 Signal naming convention - prefixes
G 7.3.10 Consistent signal names throughout hierarchy
R 7.3.11 Signal name length does not exceed 32 characters

Comments

R 7.4.1 Each file must contain a file header with required fields
R 7.4.2 Additional constructs in file use a header with required fields
R 7.4.3 Comment conventions

Code Style

R 7.5.1 Write code in a tabular format
G 7.5.2 Use consistent code indentation with spaces
R 7.5.3 One Verilog statement per line
R 7.5.4 One port declaration per line
G 7.5.5 Preserve port order
R 7.5.6 Declare internal nets
G 7.5.7 Line length not to exceed 80 characters

Module Partitioning and Reusability

R 7.6.1 No accesses to nets and variables outside the scope of a module
G 7.6.2 The use of `` `include `` compiler directives should be avoided
R 7.6.3 Mask plugs outside of top most module
R 7.6.4 Avoid potentially non-portable constructs
G 7.6.5 Partitioning conventions - general
G 7.6.6 Partitioning conventions - clocks

Modeling Practices

R 7.7.1 No hard-coded global distribution nets
R 7.7.2 Synchronize asynchronous interface signals
R 7.7.3 Use technology independent code for noninferred blocks

R 7.7.4 Glitch-free gated clock enables and direct action signals
R 7.7.5 Known state of powered down signals
R 7.7.6 Initialize control storage elements
G 7.7.7 Initialize datapath storage elements
G 7.7.8 Use synchronous design practices
R 7.7.9 No combinational feedback loops

General Coding Techniques

R 7.8.1 Expression in condition must be a 1-bit value
R 7.8.2 Use consistent ordering of bus bits
R 7.8.3 Do not assign *x* value to signals
R 7.8.4 No `reg` assign in two `always` constructs
G 7.8.5 Use parameters instead of text macros for symbolic constants
R 7.8.6 Text macros must not be redefined
G 7.8.7 Preserve relationships between constants
R 7.8.8 Use parameters for state encodings
G 7.8.9 `` `define `` usage includes `` `undef ``
R 7.8.10 Use programmable base addresses
R 7.8.11 Use text macros for base addresses
R 7.8.12 Use base + offset for address generation
G 7.8.13 Use symbolic constants for register field values
G 7.8.14 Limit ‘ifdef nesting to three levels
G 7.8.15 Use text macros for signal hierarchy paths
R 7.8.16 Macromodules are not allowed
R 7.8.17 Operand sizes must match
R 7.8.18 Connect ports by name in module instantiations
R 7.8.19 Ranges match for vector port and net/variable declarations
R 7.8.20 Port connection widths must match
G 7.8.21 Avoid ports of type inout
G 7.8.22 Use parentheses in complex equations
G 7.8.23 No disables on named blocks or tasks containing nonblocking assignments with delays
G 7.8.24 Use task guards
G 7.8.25 Next-state encoding of state machines should be made through the use of case statements
R 7.8.26 No internal three-state logic
G 7.8.27 Avoid three-state outputs
R 7.8.28 Replication multiplier must be greater than zero

Standards for Structured Test Techniques

R 7.9.1 Use additional logic for scanning high-impedance devices
R 7.9.2 Allow PLL bypass
G 7.9.3 Allow clock divider bypass
R 7.9.4 Scan support logic for gated clocks
R 7.9.5 Externally control asynchronous reset of storage elements
R 7.9.6 Latches transparent during scan
R 7.9.7 No simultaneous master/slave latch clocking
G 7.9.8 Segregate opposing phase clocks

General Standards for Synthesis

R 7.10.1	Complete `always` sensitivity list
R 7.10.2	One clock per `always` sensitivity list
R 7.10.3	Only use synthesizable constructs
R 7.10.4	Specify combinational logic completely
G 7.10.5	Assign default values to outputs before case statements
G 7.10.6	Avoid full_case synthesis directive
R 7.10.7	No `disable` in looping constructs
R 7.10.8	Avoid unbounded loops
R 7.10.9	Expressions are not allowed in port connections
G 7.10.10	Avoid top-level glue logic
R 7.10.11	Verilog primitives are prohibited
R 7.10.12	Use nonblocking assignments when inferring flip-flops and latches
R 7.10.13	Drive all unused module inputs
G 7.10.14	Connect unused module outputs
R 7.10.15	Do not infer latches in functions
R 7.10.16	Use of `casex` is not allowed
R 7.10.17	Embedded synthesis scripts are not allowed
G 7.10.18	Use a cycle-wide enable signal for signals with multicycle paths
G 7.10.19	Model high-impedance devices explicitly
G 7.10.20	Avoid direct instantiation of standard library cells

Section 7 Verilog HDL Coding

7.1 Introduction

The Verilog HDL coding standards pertain to virtual component (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 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 that a model can be adapted to an application. The modeling practices 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. Additionally, these standards apply to all other code written in Verilog, such as testbenches and monitors. Some of the standards explicitly state the type of code to which they apply, and exceptions to the standards are noted.
The rules were determined to be items that enable rapid SoC design, integration, and production, as well as enable maintainability by someone other than the original author. Note that in many cases, a guideline may fit this definition, however, at this point it may have a large number of exceptions, tool limitations, or a deeply entrenched opposing usage which prohibited the rule designation.
Note: rules and guidelines as described in V3.2 of the SRS are required for compliance only in new IP (i.e. IP coded after the release date of V3.2). But it is possible to certify older IP with V3.2 if there is no issue with the changes introduced by the new version of the standard.

7.1.1 Deliverables

The deliverables to the IP repository are defined in Section 2 VC Deliverables. These deliverables include:

- Synthesizable RTL Source Code (L1)
- Testbench (V1)
- Drivers (V2)
- Monitors (V3)
- Detailed Behavioral Model (V4)
- HDL Interface Model (V5)
- Stub Model (V6)
- Emulation (V13)

7.2 Reference Information

7.2.1 Referenced Documents

[1] IEEE Verilog Hardware Description Language, IEEE Standard 1364-1995.

[2] IEEE Verilog Hardware Description Language, IEEE Standard 1364-2001, Version C.

[3] Verilog-AMS Language Reference Manual, Version 2.2. November 2004, Accellera http://www.eda.org/verilog-ams/htmlpages/public-docs/lrm/2.2/AMS-LRM-2-2.pdf

[4] SystemVerilog 3.1a Language Reference Manual, Accellera's Extensions to Verilog, Accellera, May 2004. http://www.eda.org/sv/SystemVerilog_3.1a.pdf

7.2.2 Terminology

Base address - An address in the allocated address space of the SoC to which offsets are added to enable access registers.
Deliverables - VC deliverables are a set of files that make up a design. They are provided by the virtual component creator. Deliverables are assigned a unique identifier that consists of a letter followed by a number. A complete description of the SRS deliverables can be found in document IPMXDSRSDEL00001, Semiconductor Reuse Standard: VC Block Deliverables.
Guideline - A guideline is a "recommended" practice that enhances rapid SoC design, integration, and production, reduces the need to modify IP deliverables, and increases maintainability.
HDL - Hardware Description Language
Mask plug - Physically a mask plug is just a wire either connected to VDD or VSS, or a choice of two inputs (hardwired switch) used to configure a module without changing anything internal to that module. This avoids resynthesis when changing the configuration.
PLL - Phase-Locked Loop
Properties - Properties are variables that are assigned a value. Values are unique to each VC but the property names are common to all VC blocks. Properties are also referred to as "Metadata ." Properties are also used in equations to determine if a rule is applicable to a deliverable. If the equation holds true, the rule applies to the deliverables.
RTL - Register Transfer Level
Rule - A rule is a "required" practice that enables rapid SoC design, integration, and production, eliminates the need to modify IP deliverables, and supports maintainability.
Text macro - 'define
Top-level module - Module at the highest level of the VC design hierarchy.
UDP - User-Defined Primitive
VC - Virtual Component. A block in the virtual socket design environment. A pre-implemented, reusable module of intellectual property that can be quickly inserted and verified to create a single-chip system. The usage of the term VC is not an indication of compliance to the VSIA standards.

7.3 Naming Conventions

7.3.1 File Naming

R 7.3.1 At most one module per file

A file must contain at most one module.

Reason: Simplifies design modifications.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.3.2 File naming conventions

The file name must be composed in the following way:

<top level name>[_<sub level name>][_<file type>].<ext>

where:

<*top level name*> is the name of the top level module (e.g., duart.v)

<*sub level name*> is the module name extension for a module under the top level module (e.g., fifo for module duart_fifo.v)

<*file type*> indicates the file type:

`task`	file consists of tasks
`func`	file consists of functions
`defines`	file consists of text macros (see **G 7.8.9**)

For regular synthesizable RTL source code, _<file type> is omitted.

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

`.v`	Verilog file
`.va`	Verilog-A file
`.vams`	Verilog-AMS file

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

Example: `spooler.v`: File containing Verilog code for module spooler
`spooler_task.v`: File containing Verilog code for tasks used by module spooler

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.3.3 Separate analog, digital, and mixed-signal Verilog files

A file must contain either: (1) digital-only Verilog code (files with `.v` extension); (2) analog-only Verilog code (files with .va or `.vams` extension); or (3) mixed-signal Verilog code (files with `.vams` extension).

Reason: Digital compilers may not handle analog constructs or mixed-signal constructs; analog compilers may not handle digital or mixed-signal constructs.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

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). The following naming conventions do not apply to third-party PLI tasks.

R 7.3.4 HDL Code items naming convention

These items include: nets, variables, parameters, module/primitive instances, and constructs such as functions, modules, and tasks.

a. Names must describe the purpose of the item. Items must be named according to *what* they do rather than *how* they do it. Use meaningful names.
b. English must be used for all names.
c. Names must start with a letter, be composed of alphanumeric characters or underscores [A-Z, a-z, 0-9,_].
d. Consecutive underscores and escaped names are not allowed.
e. For names composed of several words, underscore separated words must be used.
f. Consistent usage in the spelling and naming style of nets and variables must be used throughout the design.
g. All signals and modules in the RTL that are referenced in the documentation must maintain the same name.
h. Names representing constants must be upper case (parameters and text macros), all other names NOT representing constants must be lower case. Case must not be used to differentiate construct, net, or variable names.
i. SystemVerilog, Verilog-AMS, VHDL and VHDL-AMS keywords must not be used for signals or any other user code item.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.3.5 Document abbreviations and additional naming conventions

Abbreviations used in a module must be documented and uncommon abbreviations should be avoided. Any naming conventions used in the module which are in addition to the conventions required or recommended in the SRS should be documented. The keyword section of the header should be used to document the abbreviations and additional naming conventions used. Alternately, the keyword section may contain the name of the file that contains these items. Document abbreviations and naming conventions in the Creation Guide as well.

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

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.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.3.6 Global text macros include module name

Global text macros specified by the ‘define directive must be preceded with the top-level module name, as in:

<top level module name>_<text macro name> (see **G 7.8.5**).

Reason: Avoids inadvertent redefinition of macros at the SoC level.

Example: ‘define SPOOLER_ADDR_BUS_WIDTH 32 //address bus width for module spooler

Exception: If the text macro is undefined within the same module (see **G 7.8.9**).

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.3.7 Instance naming conventions

Module instance name must be the same as the module name (with or without the top-level module name prefix, see **R 7.3.2**), optionally with an appropriate suffix.

For singly instantiated modules, it is recommended that the instance name be the same as the module name (with or without the top-level prefix). For example, module ‘duart_fifo’ may be instantiated as ‘duart_fifo’ or as ‘fifo’.

The top-level block name can also be shortened with an acronym. For example, if the block is a magenta-to-magenta gasket named mag2mag, m2m_ is a possible acronym to be used as prefix for a submodule’s instance name.

For multiply instantiated modules, instance names should have a numbered suffix (an underscore followed by an integer) or a functionally meaningful suffix.

Reason: Clear association between module name and instance name, improves readability, removes confusion.

Example:

```
mag2mag_fifo mag2mag_fifo (...);

mag2mag_fifo m2m_fifo (...);

mag2mag_fifo fifo (...);
```

```
mag2mag_fifo mag2mag_fifo_0 (...);
mag2mag_fifo mag2mag_fifo_1 (...);

mag2mag_fifo m2m_fifo_0 (...);
mag2mag_fifo m2m_fifo_1 (...);

mag2mag_fifo fifo_tx (...);
mag2mag_fifo fifo_rx (...);
```

Exception: Only applies at the Verilog source level, and does not include the wrappers.

Exception: Generic building blocks

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.3.8 Signal naming convention - suffixes

The signal name must be composed in the following way:

<signal_prefix>_<signal name>*[_p*n*][_async|_sync][_ns][_ff][_l][_clk][_z][_b][_nc][_test][_se]*

The prefix component must be used according to the specifications of rule **R 7.3.9**. The suffix component may only be used as described below and, in the case of multiple suffixes being used in the same signal name, must only be used in the order specified in the signal name descriptions above.

a. *_p*n - Pipeline stage where signal originates from. n indicates the level of the pipeline stage, where 0 indicates the first stage. Optional usage, but reserved suffix for denoting pipeline stage for pipelined designs.
b. *_async* - Asynchronous signal, or first stage of synchronizing latch when used as _async_ff.
c. *_sync* - Synchronous signal, which identifies a version of an asynchronous signal that has been synchronized with the destination clock domain.
d. *_ns* - State machine next state.
e. *_ff* - Register or flip-flop output. Optional.
f. *_l* - latch output. Optional for signals leaving top-level module or sub-module, required for signals internal to a module.
g. *_clk* - Clock signal. Exception: Signals whose names obviously indicate clocks (e.g. `system_clock` or `clk32m`).
h. *_z* - High impedance signal.
i. *_b* - Active low signal.
j. *_nc* - Not connected. This suffix can be used at the instantiation of a module, but may not be used in the output section of a module.
k. *_test* - Test signal.
l. *_se* - Scan enable. Exception: Signals whose names obviously indicate scan enable functionality (e.g. `scan_en`, `se`).

Reason: Consistent naming conventions aid in understanding the design.

Example: Use of _nc suffix

```
blockx    blockx (.result({result[15:3], result_nc[2:0]}), ...);
    or
blockx    blockx (.result(result[15:0]), ...);
wire      result_nc[2:0] = result[2:0];
```

Exception: Signals which are defined in a standard or specification (e.g., IPI or AMBA) do not have to follow the above requirements.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.3.9 Signal naming convention - prefixes

For signals internal to a module or in the port list of the module, the signal name must be composed in the following way, where:

<*top level name*> is the name of the top level module

<*sub level name*> is the name of a module under the top level module name

<*signal name*> is a meaningful signal name

a. <top level name>_<signal name>_<signal_suffix> for signals leaving the top level module
b. <sub level name>_<signal name>_<signal_suffix> for signals leaving a sub-module, but not leaving top level module. The <sub level name> may optionally be prefixed by <top level name> as well.
c. <signal name>_<signal_suffix> for internal signals. The <signal name> may optionally be prefixed by <top level name> and/or <sub level name>.

The suffix component must be composed according to rule **R 7.3.8**.

Whenever used in a signal name, the top-level module name can be shortened with an acronym, as indicated in **R 7.3.7**.

At the module instantiation level, the prefix for signals connected to module outputs should be the module instance name instead of the module name. (Note that for singly instantiated modules, the instance name should normally be the same as the module name.) However, the top-level block name prefix may be omitted or shortened to an acronym, as in **R 7.3.7**.

Reason: Consistent naming conventions aid in understanding the design.

Example: Signals leaving the top-level module:

```
gsm_crypto gsm_crypto (.gsm_cr_dout(gsm_cr_dout[31:0]), ...);
```

Exception: Signals which are defined in a standard or specification (e.g., IPI or AMBA) do not have to follow the above requirements. However, it is permitted for the outputs to be prefixed by the top level name.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.3.10 Consistent signal names throughout hierarchy

Signal names should remain the same throughout the design hierarchy.

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

Exception: Multiply-instantiated modules and generic building blocks.

Exception: Vector part selects. Selected vector bit(s) can have a name different from the vector name, as in the following case:

```
reg[7:0] status_reg_ff;
wire int_pend;
int_pend = status_reg_ff[1];
```

or

```
module modx (modx_qq);
output [1:0]        modx_qq;
wire [3:0]          qq1;
assign              modx_qq   = qq1[1:0];
```

Example: The exception above applies also to vector part selects connected to output ports, as in the following example:

```
modulex modulex (   .modulex_out2(modulex_internal_bus[2],
                    .modulex_out1(modulex_internal_bus[1],
                    .modulex_out0(modulex_internal_bus[0]),...
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.3.11 Signal name length does not exceed 32 characters

Signal name length should not exceed 32 characters. The 32 characters do not include the hierarchy.

Reason: Shorter names increase readability.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

7.4 Comments

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

7.4.1 File Headers

Every RTL and behavioral Verilog file will be documented with the header shown in **Figure 7-1**. The format of the header must match the figure to ensure the ability to parse the header with a software tool. The capitalized field identifiers in the header may be used as search points for types of information. This template format assures consistency. The header shown is the minimum required, and additions may be made after the REUSE ISSUES section. Additionally, a copyright and company confidential header should be included at the top.

```
// +FHDR-------------------------------------------------------------------------
// Copyright (c) 2004 Freescale Semiconductor, Inc. All rights reserved
// Freescale Confidential Proprietary
// ------------------------------------------------------------------------------
// FILE NAME      :
// DEPARTMENT     :
// AUTHOR         :
// AUTHOR'S EMAIL :
// ------------------------------------------------------------------------------
// RELEASE HISTORY
// VERSION DATE        AUTHOR  DESCRIPTION
// 1.0     YYYY-MM-DD  name
// ------------------------------------------------------------------------------
// KEYWORDS    : General file searching keywords, leave blank if none.
// ------------------------------------------------------------------------------
// PURPOSE     : Short description of functionality
// ------------------------------------------------------------------------------
// PARAMETERS
//     PARAM NAME      RANGE    : DESCRIPTION       : DEFAULT : UNITS
// e.g.DATA_WIDTH     [32,16]   : width of the data : 32      :
// ------------------------------------------------------------------------------
// REUSE ISSUES
//   Reset Strategy        :
//   Clock Domains         :
//   Critical Timing       :
//   Test Features         :
//   Asynchronous I/F      :
//   Scan Methodology      :
//   Instantiations        :
//   Synthesizable (y/n)   :
//   Other                 :
// -FHDR-------------------------------------------------------------------------
```

Figure 7-1 Verilog File Header

R 7.4.1 Each file must contain a file header with required fields

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

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

b. The header must include the name of the file.

c. The file header must include the originating department, including group, division, and physical location, author, and author's email address.

d. The header must include a release history only for the VC changes checked into the VC Repository, with the most recent release listed last. The date format YYYY-MM-DD must be used. This information is useful to the integrator. A local release history should not be included in the header.

e. The header must contain a section of the searching field identifiers. This string may contain a brief synopsis of the construct's functionality, or list systems and buses with which the construct was designed to work. Abbreviations and additional naming conventions may also be listed.

f. The header must contain a purpose section describing the construct's functionality. The purpose must describe *what* the construct provides and not *how*.

g. Headers must contain information describing the parameters being used in the construct. The default value must be listed. The valid parameter values must also be indicated in the range field.

h. 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.
i. All clock domains and clocking strategies must be documented.
j. Critical timing including external timing relationships must be documented. The header location may contain the name of the file that contains the critical timing information (e.g., creation guide).
k. Any specific test features that are added to the code to speed up testing must be documented.
l. The asynchronous interfaces must be described including the timing relationships and frequency.
m. Notation must be used to indicate what scan style is used, if any.
n. Headers must contain information detailing what cells, modules, function calls, and task enables are instantiated within.
o. The ability to synthesize the construct must be indicated by specifying *yes* or *no*.
p. The header should include additional pertinent information which is useful to the integrator or which makes code more understandable. This field is to be used at the designers discretion, and maintains consistency in the location of additional information

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

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

7.4.2 Additional Construct Headers

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

```
// +HDR -------------------------------------------------------------------------
// NAME     :
// TYPE     : TYPE can be func, task, primitive
// -------------------------------------------------------------------------------
// PURPOSE  : Short description of functionality
// -------------------------------------------------------------------------------
// PARAMETERS
//     PARAM NAME      RANGE   : DESCRIPTION       : DEFAULT : 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

R 7.4.2 Additional constructs in file use a header with required fields

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

a. The +HDR/-HDR tags must be used to define the boundary of the header information.
b. Construct headers must include the name of the additional construct.
c. Construct headers must include the construct type.

d. Construct headers must contain a purpose section describing the construct functionality. The purpose must describe *what* the unit provides and not *how*.
e. Construct headers must contain information describing the parameters being used in the construct. The default value must be listed.
f. The construct header should include additional pertinent information which is useful to the integrator or makes the code more understandable.

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

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

7.4.3 Other Comments

Comments are required to describe the functionality and flow of HDL code. They must be sufficient for another designer to understand and maintain the code. Comments should be used liberally throughout the code to describe the code intent, functionality, design process, and special handling. Avoid obvious comments (e.g., `a <= b; // save b into a`). Comments must be in English, and be up-to-date with the release of the code.

R 7.4.3 Comment conventions

Comments usage is as follows:

a. Each functional section of the code must be preceded by comments describing the code's intent and function.
b. Unusual or non obvious implementations must be explained and their limitations documented with a comment.
c. One line comments (//) must be used. Do not use multiline (/*...*/) comments.
d. Detailed documentation must be provided for cases in which a designer decides to group unrelated signals together.
e. Old code, or unused code must be deleted as opposed to commented out.
f. A comment must be used to explain the functionality of any instantiated cells and why the cell is instantiated and not inferred. These cells can be from a library, not modeled in an HDL language like Verilog, cells with hidden functionality, or custom implemented cells.
g. Each port declaration must have a descriptive comment, preferably on the same line. If the comment is not on the same line, it should be on the preceding line.
h. For other declarations, such as nets and variables, it is recommended to have a descriptive comment, preferably on the same line. If the comment is not on the same line, it should be on the preceding line. Optional for auto-generated code.
i. Document SR latch usage (Recommended).
j. Gated clock usage that is not inferred by tools must be documented in the code.
k. Multicycle paths must be documented in the code.
l. All synthesis-specific directives must be documented where used, identifying the reason they are used, the tool and the directive used.
m. Compiler directives such as `` `ifdef ``, `` `else `` and `` `endif `` must have a comment where used, describing the usage of the directive.
n. Comment end and endcase statements with an annotation of the construct ended. Recommended for sections of code with more than 10 lines of code, but not required.

Example:

```
always @(p or q)
begin
...
end                 // always @(p or q)
```

or

```
case (...)          // <case function>
```

```
...
endcase             // <case function>
```

Reason: Aids understanding of the code.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

7.5 Code Style

R 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 code block (begin, case, if statements, etc.), it is useful to complete the frame first, in particular to align the *end* of the code block with the *begin*.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.5.2 Use consistent code indentation with spaces

Consistent indentation should be used for code alignment. Spaces should be used instead of tab stops.

Reason: Improves readability. Tab stops should not be used because they may be interpreted differently in different systems.

Note: Excessive indentation may actually hinder readability (see **G 7.5.7**).

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.5.3 One Verilog statement per line

One line must not contain more than one statement. Do not concatenate multiple semicolon separated Verilog statements on the same line. Comments are allowed on the same line as a Verilog statement.

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]);
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.5.4 One port declaration per line

Port types must be indicated individually; that is, one port per line must be declared, using the direction indication with each net.

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

Example: Use:

```
input     a;        // port a description
input     b;        // port b description
```

Do not use:

```
input     a, b;
```

or:

```
input     a,
```

```
                b;
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.5.5 Preserve port order

It is recommended that the port declarations be listed in the same order as the port list of the module declaration.

Reason: Improves readability.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.5.6 Declare internal nets

a. Internal nets must be declared explicitly, not implicitly. Port nets need not be redeclared in wire declarations in addition to the input/output/inout declarations.

b. It is recommended that all wire declarations be grouped together in one section following the input/output/inout declarations at the top of the module.

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

Exception: Auto-generated code.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.5.7 Line length not to exceed 80 characters

It is recommended that line length not exceed 80 characters.

Reason: Improves readability, and avoids inadvertent line wraps.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

Figure 7-4 is an example of good Verilog code format.

```
// +FHDR-------------------------------------------------------------------------
// Copyright (c) 2004 Freescale Semiconductor, Inc. All rights reserved
// Freescale Confidential Proprietary
// ------------------------------------------------------------------------------
// FILE NAME      : prescaler.v
// DEPARTMENT     : SPS SoCDT, Austin TX
// AUTHOR         : Mike Kentley
// AUTHOR'S EMAIL : r6476c@freescale.com
// ------------------------------------------------------------------------------
// RELEASE HISTORY
// VERSION DATE        AUTHOR           DESCRIPTION
// 1.0     1998-09-12  tommyk           initial version
// 2.0     1998-11-11  mkentley         Updated for SRS compatibility
// 2.1     1999-10-25  mark lancaster   Cleaned up prescaler bypass.
// ------------------------------------------------------------------------------
// KEYWORDS    : clock divider, divide by 16
// ------------------------------------------------------------------------------
// PURPOSE     : divide input clock by 16.
// ------------------------------------------------------------------------------
// PARAMETERS
//    PARAM NAME   RANGE : DESCRIPTION : DEFAULT : UNITS
//    N/A
// ------------------------------------------------------------------------------
// REUSE ISSUES
//   Reset Strategy   : Asynchronous, active low system level reset
//   Clock Domains    : core_32m_clk, system_clk
//   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
//   Synthesizable    : Y
//   Other            : uses synthesis directive to infer a mux to
//                      avoid glitching clock_out and clock_out_b
// -FHDR-------------------------------------------------------------------------
module prescaler(
                  core_32m_clk,
                  system_clock,
                  scan_mode_test,
                  reset_b,
                  div16_clk,
                  div16_clk_b
                );
input     core_32m_clk;       // 32 MHz clock
input     system_clk;         // system clock
input     scan_mode_test;     // scan mode clock
input     reset_b;            // active low hard reset, synch w/ system_clock
output    div16_clk;          // input clock divided by 16
output    div16_clk_b;        // input clock divided by 16 and inverted
```

Figure 7-3 Verilog Coding Format (Page 1)

```
reg[3:0]  count_ff;           // counter to make clock divider
reg       div16_clk;          // input clock divided by 16
reg       div16_clk_b;        // input clock divided by 16 and inverted

wire[3:0] count_ns;           // clock divider next state input

// 4-bit counter; count_ff[3] is the divide by 16

assign count_ns = count_ff  + 4'b0001;  // increment counter

always @(posedge core_32m_clk or negedge reset_b)
          if (!reset_b)
                    count_ff <= 4'b0000;          // reset counter
          else
                    count_ff <= count_ns;         // update 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 on the rising edge of the system clock for test purposes.

// synopsys infer_mux "clk_mux"
// ensure that mux is not optimized away during synthesis

always @(scan_mode_test or system_clk or count_ff)
begin: clk_mux
          if (!scan_mode_test)          // normal operation clock assign
          begin
                    div16_clk  =  count_ff[3];
                    div16_clk_b = ~count_ff[3];
          end
          else
          begin                         // scan mode clock assign
                    div16_clk   = system_clk;
                    div16_clk_b = system_clk;
          end
end                                     // clk_mux

endmodule                               // prescaler
```

Figure 7-4 Verilog Coding Format (Page 2)

7.6 Module Partitioning and Reusability

R 7.6.1 No accesses to nets and variables outside the scope of a module

Modules, tasks, and functions must not modify nets or variables not passed as ports into the module. Verilog users must not use a hierarchical reference to read or modify a net or variable.

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

Exception: Non-synthesizable blocks, e.g. behavioral models and testbenches.

Deliverables: L1,V13

G 7.6.2 The use of `'include` compiler directives should be avoided

The use of `'include` compiler directives should be avoided. Where necessary, only the name of the file being referenced (no relative nor absolute pathnames) should be specified.

Reason: The `'include` compiler directive affects reusability by introducing an additional file dependency.

Exception: `'define` files.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.6.3 Mask plugs outside of top most module

Mask plugs must be placed outside of the top most module.

Exception: Internal mask plugs may be used when a generic building block is instantiated or a single sub-block has multiple instantiations within a block, and the mask plugs configure the instances for specific function that does not change with block reuse. The synthesis flow will remove the unused logic.

Reason: Resynthesis will not be required for base changes.

Deliverables: L1

R 7.6.4 Avoid potentially non-portable constructs

The constructs used in the code must be compliant to the latest version of the IEEE 1364 standard, and not dependent on simulator specific extensions or behavior.

Reason: May compromise reusability across vendors and tools.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.6.5 Partitioning conventions - general

Good partitioning improves adaptability to different applications and helps to emphasize special logic, which aids in testing, debug and integration. The following partitioning guidelines should be followed:

a. The design should be partitioned such that the module boundaries match the physical boundaries.
b. The application-specific (e.g, bus interface) parts of the code should be partitioned from the more general portion of the code.
c. Speed critical logic that may require specific timing constraints for synthesis should be partitioned into its own module.
d. Data-path logic should be partitioned from nondata path logic.
e. Any BIST logic should be coded into its own module or with the module that instantiates the array associated with the BIST logic. This makes it easier to port BIST to a different memory.
f. Memory control logic and the memory array functions should be partitioned into separate modules.

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: Non-synthesizable code.

Deliverables: L1,V13

G 7.6.6 Partitioning conventions - clocks

Proper partition of clock signals aids in the automatic generation of the clock distribution network, as well as in the overall synthesis of the module. The guidelines below should be followed when distributing the clock signal through the design:

a. If it is necessary to use a gated clock, internally generated clocks, or use both edges of a clock, the clock generation circuitry should be kept in a separate module at the top level of the VC module or at the same logical level in the hierarchy as the module to which the clocks apply.
b. Partition separate clock domains into separate modules. The synchronization logic should be part of the receiving clock domain.

c. Asynchronous logic should be partitioned from synchronous logic. Asynchronous logic is that which is not timed by a clock signal. Synchronous logic with an asynchronous set or reset pin is considered synchronous.

Reason: Eases clock network synthesis. 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: Non-synthesizable code.

Deliverables: L1,V13

7.7 Modeling Practices

R 7.7.1 No hard-coded 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 typically nonsynthesizable modules(e.g., Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation. Not applicable to logic that has timing and/or sizing specification constraints that cannot be met with synthesis.

Deliverables: L1,V13

R 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). The connection of register inputs asynchronous to the register clock, other than for synchronization purposes as previously mentioned, must be avoided. See **G 7.6.6**.

Reason: Limit asynchronous signals to a minimum (asynchronous design practice is not yet adequately supported by design tools). Register inputs asynchronous to the register clock are the source of metastability problems, which can show up both during static timing analysis and gate-level simulation.

Note: Pay particular attention to the interface in which there is a frequency difference. For example, a lower frequency clock domain cannot guarantee reception of 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 an 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 also be taken into consideration.

Exception: Not applicable to typically nonsynthesizable modules(e.g., Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation.

Exception: Bus signals are not guaranteed to change together on the same clock cycle at the receiving clock domain. For example, a multi-bit value may have intermediate values. In such a case, one must use other techniques such as gray-code value or strobing signal which ensures coherent usage of synchronized signal groups. Furthermore, in the case in which an asynchonous signal is synchronized in more than one receiver, each synchronized signal must be considered independent in the receiving clock domain, in order to avoid problems due to race conditions between the different synchronizers.

Exception: Registers specifically instantiated to synchronize an asynchronous source of data (also known as synchronizing registers).

Deliverables: L1,V13

Figure 7-5 shows the problem that this rule addresses. Register r1 is clocked by d1_clk, and it sources data to register r2, which is clocked by d2_clk. As an example, the combinational logic between r1 and r2 can exist in a module in which r2 is part of a state machine, such that the logic implements the 'next state' transition. In this case there can be no guarantee that the data at the input of r2 will meet setup and hold requirements with respect to d2_clk. Note also that, in another situation, the register clocked by d1_clk and the combinationanl logic can be located in a different module than the register clocked by d2_clk, in which case the input to the latter would be an asynchronous signal that would have to be synchronized close to the interface.

Figure 7-6 shows the proper implementation of the exception to the rule. In this case, register r1 is clocked by d1_clk, and registers r2s and r2 are clocked by d2_clk. Data at the output of r2 can be considered stable, and synchronous to clock d2_clk.

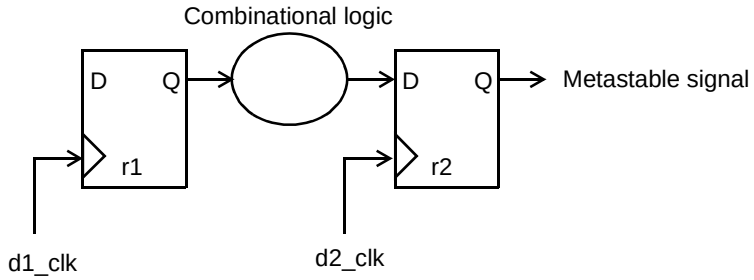

Figure 7-5 Metastability Hazard Due to a Violation of this rule

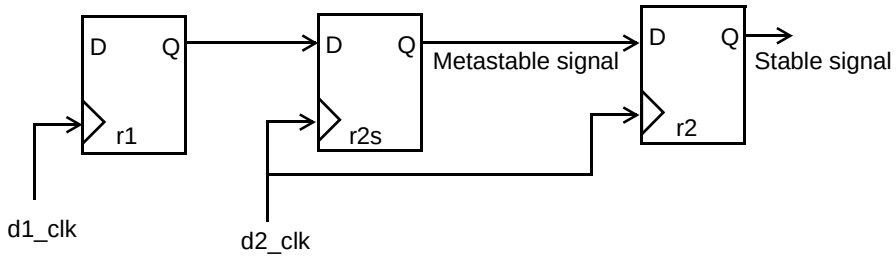

Figure 7-6 Proper Use of Synchronization Register According to this rule

R 7.7.3 Use technology independent code for noninferred blocks

A model of an instantiated noninferred block (i.e., custom, not intended to be synthesized) must be written in a technology-independent coding style (see **R 7.4.1**(n), **R 7.4.3**(f) and **G 7.10.20**).

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

Example: Instantiations in a separate function.

Deliverables: L1,V13

R 7.7.4 Glitch-free gated clock enables and direct action signals

a. If gated clocks are used, there must not be any glitches during the clocking time.
b. Internally generated direct action signals (e.g., reset, set, chip_select) should be glitch free.

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, which can cause faulty operation. Glitches on direct action signals may cause faulty operation.

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

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.7.5 Known state of 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 its 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 VC block when the signal source is powered down.

Deliverables: L1,V13

R 7.7.6 Initialize control storage elements

All latches and registers in a control path must be initialized to a predetermined value, 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"). Also, the propagation of X's makes verification more difficult.

Exception: An FSM that does something during reset (such as a reset time counter) or that must complete/continue an action during reset - such as SDRAM refresh cycle or complete a memory access without damage. Each illegal state in such FSM must transition to a well known legal state.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.7.7 Initialize datapath storage elements

All latches and registers in a data path should be set or reset, as appropriate. Initialization of storage elements can be achieved in a number of ways, such as the use of set/reset components or the flushing of scan chains.

Reason: Aids in performing latch divergence tests while performing speed tests and hunting for timing paths.

Exception: Where there will be a major cost in area and route, as long as the implementation and proper block activation takes care of blocking non-initialized values from propagating out.

Deliverables: L1,V13

G 7.7.8 Use synchronous design practices

Synchronous design practices should be followed whenever possible. Asynchronous design circuitry should be used only 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 typically nonsynthesizable modules(e.g., Bus Functional Models, Bus Monitors or Analog Behavioral Models) unless they are intended to be synthesized for emulation.

Deliverables: L1,V13

R 7.7.9 No combinational feedback loops

Combinational feedback loops must not be used.

Reason: Combinational feedback loops do not work with cycle-based simulators or formal verification tools.

Exception: Highly specialized VC blocks such as digital phase-locked loops.

Deliverables: L1,V13

7.8 General Coding Techniques

R 7.8.1 Expression in condition must be a 1-bit value

The conditions in the following constructs must be an expression that results in a 1-bit value:

- *if(condition)*
- *while(condition)*
- *for(initial_assignment; condition; step_assignment)*
- @(posedge condition or ...)
- @(negedge condition or ...)
- (condition)? exp1 : exp2

Reason: Avoid nonstandard simulator behavior.

Example:

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

if (bus) bus_is_active = 1;    // bus is a multi-bit value,
                              // which violates the rule

if (bus > 0) bus_is_active = 1;// the resulting control condition is a 1-bit
                               // expression, as suggested by the rule,
                               // which is intuitively easier to read
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.2 Use consistent ordering of bus bits

When describing multibit buses, a consistent ordering of the bits must be maintained.

Reason: Eases readability and reduces inadvertent order swapping between buses.

Exception: VC blocks which internally use one convention, but interface to the other convention at the VC block boundary.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.3 Do not assign `x` value to signals

The value of `x` must not be assigned to signals. Known legal signal values must be assigned to all signals.

Reason: Avoids `x` propagation through the circuitry.

Exception: Testbenches.

Deliverables: L1,V13

R 7.8.4 No `reg` assign in two `always` constructs

`regs` must not be assigned in two separate `always` constructs.

Reason: Avoid internal buses, and execution order simulator dependence.

Exception: Behavioral code.

Deliverables: L1,V13

G 7.8.5 Use parameters instead of text macros for symbolic constants

It is recommended that parameters be used instead of text macros (‘define) to specify symbolic constants.

(See **G 7.8.9**)

Reason: The scope of ‘define text macros is global unless they are explicitly undefined. This may cause compilation order dependencies and conflicts at the SoC level. Also, parameter values may be customized for each instance of a module.

Exception: Global constants.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.6 Text macros must not be redefined

Text macros must not be redefined to a different value. This applies to both locally and globally-defined macros.

Reason: Avoids inadvertent redefinition of macros.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.8.7 Preserve relationships between constants

If a constant is dependent on the value of another constant, the dependency should be shown in the definition. Where a text 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)
```

instead of:

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

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.8 Use parameters for state encodings

Enumerated parameters must be used to encode the different states of an explicitly coded state machine.

Reason: This eases retargeting to different state machine implementations, for example changing from an encoded 1-hot style to gray code. Using symbolic names enhances readability, and provides a single point for state changes.

Example:

```
parameter [1:0]     RESET_STATE;        // synthesis enum state_info
parameter [1:0]     TX_STATE;           // synthesis enum state_info
parameter [1:0]     RX_STATE;           // synthesis enum state_info
parameter [1:0]     ILLEGAL_STATE;      // synthesis enum state_info
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.8.9 `‘define` usage includes `‘undef`

If a ‘define statement is used within the design code, the macro name should be undefined using ‘undef in the same module (see **R 7.3.6**).

Reason: 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. Otherwise, compilation-order dependencies may unintentionally be introduced in the code.

Exception: Testbenches

Deliverables: L1,V13

R 7.8.10 Use programmable base addresses

The base address of a module must be programmable.

Reason: Eases changing the memory map of a module.

Exception: IP blocks whose registers are accessed through SkyBlue need not contain a base address; it is enough to define a region with a large enough offset to cover the registers defined for the module, including reserve space. For example, a module may allocate a 4KB address space for the registers defined in the IP (including reserve space for expansion). This permits the integrator of the IP to relocate the base address for a given IP to any desired location within the device's address space.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.11 Use text macros for base addresses

If the base address of a module appears in the Verilog code, it must be assigned with a text macro.

Reason: Eases changing the memory map of a module.

Exception: The base address is defined in software.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.12 Use base + offset for address generation

All accesses to a register/memory location within a module must be formed by using a base address and the offset from the base.

Reason: Eases retargeting to different system configurations.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.8.13 Use symbolic constants for register field values

Symbolic constants should be used for register field values rather than fixed numerical constants. The fields may be one or more bits or the entire register.

Reason: Improves readability and maintainability. Reduces the chance of using the wrong value.

```
Example:
 parameter  CNT_ADDR = 2'b00;        // count address matches

 ...
if (cnt_ctrl_reg[6:5] == CNT_ADDR)
   if (addr_in[31:0] == cnt_ref[31:0])
      counter <= counter + 1;
...
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.8.14 Limit `ifdef nesting to three levels

Nesting of 'ifdef directives should not exceed three levels.

Reason: Eases understanding of the code.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.8.15 Use text macros for signal hierarchy paths

Commonly used signal hierarchy paths should be specified using a text macro. Hierarchical signal paths are not allowed in code that is intended to be synthesized.

Reason: Simplifies remapping to a new environment

Example:

```
      `define XXX_TESTBENCH si
      `define XXX_TOP_SCOPE `TESTBENCH.xxx // xxx is the top-level module
      `define XXX_YYY_SCOPE `XXX_TOP_SCOPE.yyy // yyy is a submodule of xxx
```

Deliverables: V1, V2, V3, V4, V5, V6

R 7.8.16 Macromodules are not allowed

The macromodule construct must not be used.

Reason: Simulation results may be simulator dependent. Different simulators treat macromodules differently.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.17 Operand sizes must match

No expression may have its size implicitly extended or reduced. In a case statement, all the case item expressions and the case expression must have the same 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: The following should all be avoided:

```
wire [63:0] wire64bit;
wire [ 7:0] wire8bit;

assign wire8bit   = 1;           // assigns integer (32-bit) to 8-bit net
assign wire64bit  = 1;           // assigns integer (32-bit) to 64-bit net
assign wire64bit  = 'b1;         // assigns unsized number to 64-bit net
assign wire64bit  = wire8bit;    // assigns 8-bit expression to 64-bit net
assign wire64bit  = wire64bit & wire8bit;  // bit-wise AND of 8-bit and 64-bit nets
```

Exception: If the right-hand side of an assignment is of the form a+b, then the left-hand side may be wider than the right-hand side.

Exception: Does not apply to variables of type integer since they have no direct hardware intent.

Exception: Expressions of the type +1 or -1 do not have to follow this rule.

Exception: Assignments of either constant 0 or 1.

Example:

```
always @(posedge clk or negedge rst_b)
          if (rst_b == 1'b0) my_wide_reg <= 0;
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.18 Connect ports by name in module instantiations

In module instantiations, ports must be connected by name instead of by position. For multi-bit ports, it is recommended to include explicit widths in the connection.

Reason: Module instantiation explicitly shows port connections, which improves readability and adaptability. For multi-bit ports, including bit widths highlights the nature of the port.

Exception: Auto-generated code.

Example:

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

Example:

```
block block_1 (.vector(vector[7:0]));
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.19 Ranges match for vector port and net/variable declarations

The ranges in both the vector port declaration and the net/variable declaration must be equal.

Reason: Nonmatching ranges are forbidden by the IEEE Verilog standard. Simulators may treat the situation differently. Matching ranges ease understanding of the code, help avoid inadvertent unconnected nets, and ease portability of the code between tools.

Example: Improper usage:

```
input  [23:0] a;
output [23:0] b;
input  [23:0] c;
wire   [31:0] a;
```

```
reg    [15:0] b;
wire   [31:8] c;
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.20 Port connection widths must match

In module instantiations, nets connected to ports must have the same width as the respective port declaration.

Reason: In cases where the net and port widths differ in a module instantiation, different tools may treat the connection differently. Where source is shorter than sink, some tools may either zero- or sign-extend, in order to match the sink width, whereas other tools may leave the extra sink bits unconnected.

Example: A port declaration as in the following example:

```
module bus_bridge (input_bus,
                   ...
          input [31:0] input_bus;
                   ...
```

should have a corresponding net connection in the module instantiation as follows:

```
bus_bridge bus_bridge (.input_bus(internal_bus[31:0]),
                   ...
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.8.21 Avoid ports of type `inout`

Ports of type input or output should be used. Ports of type inout (bidirectional) should be avoided. If the block is connected to a bidirectional bus, the port should be split into separate input and output ports which will be connected to each other outside the block. (See **R 7.9.1** for treatment of three-state outputs; see also **G 7.8.27**.)

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

Deliverables: L1,V13

G 7.8.22 Use parentheses in complex equations

Parentheses should 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, and clearly convey the functionality.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.8.23 No disables on named blocks or tasks containing nonblocking assignments with delays

In order to ensure consistent simulation results, a named block or a task containing nonblocking assignments with delays should not be disabled.

Reason: Tools handle pending scheduled values differently.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

G 7.8.24 Use task guards

Error guards should be used in tasks containing delays.

Reason: Avoids re-entrant task data corruption.

Example:

```
task <name>;
reg in_use;
begin
    if (in_use == 1'b1) $stop;
    in_use = 1'b1;
```

```
    #10
    ...
    in_use = 1'b0;
end
endtask
```

Deliverables: V1,V2,V3,V4,V5,V6

G 7.8.25 Next-state encoding of state machines should be made through the use of case statements

The preferred construct to encode the next-state logic in state machines is the case statement.

Reason: Case statements provide a readable way to specify state transitions in a state machine. They are also easier to maintain, particularly in situations in which a state has to be either added or deleted.

Example: The following is a typical example of a state machine written with case statements:

```
always @(posedge clock)
begin        // sequential part of state machine
    if (!reset_b)
        state <= RESET_STATE;
    else
        state <= state_ns;
end

always @(state or done)
begin    // combinational part of state machine
    state_ns = INIT_STATE;
    case (state)
    RESET_STATE:
        state_ns = INIT_STATE;
    INIT_STATE:
        state_ns = DATA_WAIT_STATE;
    DATA_WAIT_STATE:
        state_ns = DATA_PROCESS_STATE;
    DATA_PROCESS_STATE:
        if (done)
            state_ns = INIT_STATE;
        else
            state_ns = DATA_PROCESS_STATE;
    endcase    // end case (state)
end            // end combinational part of state machine
```

Note: When using case statements to encode the next-state logic in a state machine, consideration should be given to **R 7.10.4**, **G 7.10.18** and **G 7.10.19**, as they address other aspects of the specification of combinational logic.

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

R 7.8.26 No internal three-state logic

Do not use three-state logic within the block. Use multiplexers instead.

Reason: Three-state logic complicates testability, equivalence checking and requires special verification.

Deliverables: L1,V13

G 7.8.27 Avoid three-state outputs

Avoid three-state outputs of the block. Use external multiplexers instead.

Reason: Three-state logic complicates testability, equivalence checking and requires special verification. In addition, they typically are slow and complicate chip-level synthesis methodologies.

Deliverables: L1,V13

R 7.8.28 Replication multiplier must be greater than zero

In a replication operation (e.g. `{4{1'b0}}`) the replication multiplier must be greater than zero.

Reason: The use of a replication multiplier that is either negative or zero is forbidden by the IEEE standard.

Example:

```
x = {P-1{1'b0},1'b1};          // P must be greater than 1
```

Deliverables: L1,V1,V2,V3,V4,V5,V6,V7,V13

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 document IPMXDSRSDFT00001, Semiconductor Reuse Standard: VC Design-for-Test .

R 7.9.1 Use additional logic for scanning high-impedance devices

Additional logic must be used to prevent signal contention. All multisourced signals and buses must be driven one-hot mutually-exclusive during the launch and capture test sequence (see **R 7.8.26**, **G 7.8.27**, **G 7.10.19** and **G 10.4.10**).

Reason: To prevent propagation of `x` and scan vector mismatches.

Example: 1-hot driver high-impedance device enables during scan shift.

Deliverables: L1

R 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 bypassing or disabling the PLL must be documented (see **G 10.10.11** and **R 10.4.25**).

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

Deliverables: L1

G 7.9.3 Allow clock divider bypass

There should be a method to bypass clock dividers (see **R 10.4.25**).

Reason: Decreases simulation and tester time.

Deliverables: L1

R 7.9.4 Scan support logic for gated clocks

Gated clocks usage must be accompanied by scan support logic (see **R 10.4.25**).

Reason: To avoid ATPG loss of fault coverage.

Deliverables: L1

R 7.9.5 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 (see **R 10.7.3;** see also document IPMXDSRSIPI00001, Semiconductor Reuse Standard: IP Interface).

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.

Deliverables: L1

R 7.9.6 Latches transparent during scan

If latches are connected to the clock, they must be transparent during scan or otherwise scannable (see **R 10.4.17**).

Refer to **Figure 7-7** for a two-phase implementation.

Reason: To avoid fault coverage loss due to the latches blocking upstream and downstream logic.

Deliverables: L1

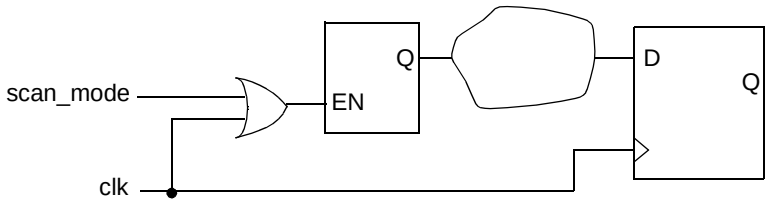

Figure 7-7 Scan Support for Mixed Latch/Flip-Flop Designs

R 7.9.7 No simultaneous 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.

Deliverables: L1

G 7.9.8 Segregate opposing phase clocks

Logic which uses both the positive and the negative edges of the clock should have segregated clocks (i.e., each clock is a separate input to the VC/block (see **R 10.7.2**).

Reason: Eases scan insertion.

Deliverables: L1

7.10 General Standards for Synthesis

This section describes the synthesis standards which are applicable to 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 modules that are not intended for synthesis (e.g., Bus Functional Models, Bus Monitors, data path modules, Analog Behavioral Models, testbenches, or behavioral modules) unless they are intended to be synthesized for emulation.

R 7.10.1 Complete `always` sensitivity list

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

Reason: Synthesis creates a structure that depends on all values read regardless of the sensitivity list, which can lead to mismatches between RTL simulation and gate-level simulation or real-world behavior.

Deliverables: L1,V13

R 7.10.2 One clock per `always` sensitivity list

Only one clock per Verilog `always` construct sensitivity list must be used in a synchronous process.

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

Deliverables: L1,V13

R 7.10.3 Only use synthesizable constructs

Only synthesizable Verilog constructs must be used. Verilog code that is not to be synthesized must be separated from the code to be synthesized. No waves, checkers, or other tasks and/or statements not supported by synthesis (e.g., $display, force, release) are allowed in the HDL. Examples of things that must not be found in synthesizable logic are:

- `# delay control`
- `initial` statement.

Reason: May compromise reusability across vendors and tools.

Exception: Print statements may be embedded in an `` `ifdef `` used for simulation purposes only.

Deliverables: L1,V13

R 7.10.4 Specify combinational logic completely

Combinational logic must be specified completely (i.e., a value must be assigned to the logic outputs for all input combinations). In a construct derived from either a `case` or an `if` statement, the outputs may be assigned default values before the `case` or `if` statement, and then the logic is completely specified (see **G 7.10.5**).

Reason: Synthesis tools infer memory elements if combinational logic is not completely specified.

Example: In the following section of code, memory elements will not be erroneously inferred, as the logic is completely specified:

```
always @(signal_name)
begin
    output = 4'b0000;
    case (signal_name)
        3'b001: output = 4'b0000;
        3'b010: output = 4'b1010;
        3'b101: output = 4'b0101;
        3'b111: output = 4'b0001;
    endcase
end    // always @(signal_name)
```

Example: In the following section of code, the `if` statement is inferred into a memory element, because there is no corresponding `else` clause:

```
always @(signal_name)
    if (signal_name) output = 4'b0;
```

Deliverables: L1,V13

G 7.10.5 Assign default values to outputs before case statements

It is recommended to precede combinational logic case statements with a default value assignment to the logic outputs. This is the safest way to ensure that the outputs are assigned the intended value under all conditions. This is the preferred practice over the use of a default clause or the `full_case` synthesis directive in the case statement.

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

Example:

```
always @(state_cur or frame or irdy or hit or
    terminate or stop or trdy)
begin
    state_ns = state_cur ;
    case (state_cur)
        IDLE : if (frame) state_ns = BUSY;
        BUSY : begin
            if (!frame & !irdy) state_ns = IDLE;
```

```
            if (frame & hit & !terminate ) state_ns = XDATA;
        end
        XDATA : if (frame & stop & !trdy ) state_ns = BACKOFF;
        BACKOFF : if ( !frame) state_ns = TURNA;
        TURNA : if (!frame) state_ns = IDLE;
    endcase               // state_cur
end                       // always @(state_ns or ...)
```

Deliverables: L1,V13

G 7.10.6 Avoid `full_case` synthesis directive

It is recommended that the `full_case` synthesis directive be avoided, unless it is essential in order to meet area or timing targets.

Reason: The `full_case` directive informs the synthesis tool that the assignments of the unused cases are don't cares. This may result in pre- and post-synthesis simulation mismatches. It may also cause incorrect logic to be formed. In addition, the use of the `full_case` directive can make it more difficult to run a formal verification tool.

Example: The following example illustrates the incorrect usage of the `full_case` directive. The enable input may be optimized away because the `full_case` directive overrides the initial default assignment:

```
always @(a or en)
begin
    y = 4'b0;
        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          // case {(en, a)}
end                      // always @(a or en)
```

Deliverables: L1,V13

R 7.10.7 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. Also, such a statement inside a loop is not supported by some tools.

Deliverables: L1,V13

R 7.10.8 Avoid unbounded loops

Looping constructs must have a static range.

Reason: Unbounded constructs may not be synthesizable or portable.

Deliverables: L1,V13

R 7.10.9 Expressions are not allowed in port connections

Expressions must not be used in port connections.

Reason: An expression inside a port connection may result in glue logic between modules and affect synthesis. It may also make the code more difficult to understand.

Exception: Bus concatenations and constants are allowed.

Deliverables: L1,V13

G 7.10.10 Avoid top-level glue logic

Gates should 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.

Exception: If the top level is a block flattened by synthesis.

Deliverables: L1,V13

R 7.10.11 Verilog primitives are prohibited

Verilog primitives such as `and`, `or`, or UDPs must not be used.

Reason: Eases understanding of the HDL.

Deliverables: L1,V13

R 7.10.12 Use nonblocking assignments when inferring flip-flops and latches

a. Nonblocking assignments (<=) must be used in edge-sensitive sequential code blocks. Blocking assignments (=) are not allowed.
b. Latches must be written with nonblocking assignments.
c. Use blocking assignments for combinational always blocks.

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

Exception: Do not generate a clock signal from another clock signal using a non-blocking assignment. For example:

```
always @(posedge clka)
          clkb <= ~clkb;
```

Example: Edge-sensitive code written with nonblocking assignments:

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

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

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

In pre-synthesis simulation, the second always block may execute before the first always block. In that case, `data` will run through to regb on the same posedge of the clock. However, this will synthesize to a shift register, where `data` will not run through to `regb` on the same posedge of the clock.

Deliverables: L1,V13

R 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 zero or one.

Reason: All ports appear in the module instantiation. None are hidden or forgotten. Avoids floating nodes.

Deliverables: L1,V13

G 7.10.14 Connect unused module outputs

All unused module instance outputs should be connected to a "dummy" wire declaration that indicates no connect.

Reason: Indicates that the unconnected ports are intentional.

Example: The following is an example of unused outputs:

```
wire bit_3;
wire bit_2_nc;
wire bit_1_nc;
wire bit_0_nc;

counter  counter (.bit_3(bit_3), .bit_2(bit_2_nc), .bit_1(bit_1_nc), .bit_0(bit_0_nc) );
```

Deliverables: L1,V13

R 7.10.15 Do not infer latches in functions

Latches must not be inferred in any function call.

Reason: Functions always synthesize to combinational logic.

Deliverables: L1,V13

R 7.10.16 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 whether the true don't care conditions will affect the operation of the design, rather than cover up a problem.

Deliverables: L1,V13

R 7.10.17 Embedded synthesis scripts are not allowed

Embedded synthesis scripts must not be used. If embedded scripts are used, they must be documented as to purpose and functionality.

Reason: Later reuse of the VC 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., async_set_reset). These must be documented in the reuse issues section of the header.

Deliverables: L1,V13

G 7.10.18 Use a cycle-wide enable signal for signals with multicycle paths

A signal propagated through a multicycle path should 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 testing and their usage must be carefully evaluated and completely documented.

Deliverables: L1,V13

G 7.10.19 Model high-impedance devices explicitly

High-impedance devices should be modeled explicitly with *z* assignments and multiple concurrent assignments. All select combinations for high-impedance devices should be defined with mutually exclusive logic.

Reason: Multiple concurrent assignments and assignment of value *z* results in the inference of three-state buffers. Lack of bus contention must be ensured. Three-state devices may cause test problems (see **R 7.9.1**).

Deliverables: L1,V13

G 7.10.20 Avoid direct instantiation of standard library cells

Avoid the direct instantiation of standard library cells in HDL code (see **R 7.4.3**(f) and **R 7.7.3**).

Reason: Eases reuse and portability across libraries.

Deliverables: L1,V13

Standard End Sheet

FINAL PAGE OF
44
PAGES