# IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language

IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Sponsored by the
Design Automation Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

**IEEE Std 1800™-2017**
(Revision of
IEEE Std 1800-2012)

**IEEE Std 1800™-2017**
(Revision of
IEEE Std 1800-2012)

# IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language

Sponsor

**Design Automation Standards Committee**
of the
**IEEE Computer Society**
and the
**IEEE Standards Association Corporate Advisory Group**

Approved 6 December 2017

**IEEE-SA Standards Board**

**Abstract:** The definition of the language syntax and semantics for SystemVerilog, which is a unified hardware design, specification, and verification language, is provided. This standard includes support for modeling hardware at the behavioral, register transfer level (RTL), and gate-level abstraction levels, and for writing testbenches using coverage, assertions, object-oriented programming, and constrained random verification. The standard also provides application programming interfaces (APIs) to foreign programming languages.

**Keywords:** assertions, design automation, design verification, hardware description language, HDL, HDVL, IEEE 1800™, PLI, programming language interface, SystemVerilog, Verilog®, VPI

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

| | | |
|---|---|---|
| Print: | ISBN 978-1-5044-4510-8 | STDPD22888 |
| PDF: | ISBN 978-1-5044-4509-2 | STDGT22888 |

## Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

## Official statements

A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

## Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854 USA

## Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

## Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

## Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

## Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE-SA Website at http://ieeexplore.ieee.org or contact IEEE at the address listed previously. For more information about the IEEE SA or IEEE's standards development process, visit the IEEE-SA Website at http://standards.ieee.org.

## Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE-SA Website at the following URL: http://standards.ieee.org/findstds/errata/index.html. Users are encouraged to check this URL for errata periodically.

## Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at http://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

## Participants

The **SystemVerilog Language Working Group** is entity based. At the time this standard was completed, the SystemVerilog Working Group had the following membership:

**Karen Pieper**, Accellera Systems Initiative, *Chair*
**Neil Korpusik**, Oracle Corporation, *Vice Chair*, *Technical Chair*
**Dennis Brophy**, Mentor, a Siemens Business, *Secretary*
**Shalom Bresticker,** Accellera Systems Initiative, *Editor*

Dave Rich, Mentor, a Siemens Business
Dmitry Korchemny, Synopsys, Inc.
Michiel Ligthart, Design Automation, Inc.
Matt Maidment, Intel Corporation
Scott Little, Mentor, a Siemens Business
Charles Dawson, Cadence Design Systems, Inc.

Work on this standard was divided among primary committees.

The **Design and Testbench Committee (SV-BC)** was responsible for the specification of the design and testbench features of SystemVerilog.

**Matt Maidment**, Intel Corporation, *Chair*
**Brad Pierce**, Synopsys, Inc., *Co-Chair*

Shalom Bresticker, Accellera Systems Initiative
Jonathan Bromley, Oracle Corporation
Eric Coffin, Mentor, a Siemens Business
Mark Hartoog, Synopsys, Inc.
Neil Korpusik, Oracle Corporation
Francoise Martinolle, Cadence Design Systems, Inc.
C. Venkat Ramana Rao, Mentor, a Siemens Business
Justin Refice, NVIDIA Corporation
Dave Rich, Mentor, a Siemens Business
Ray Ryan, Mentor, a Siemens Business
Arturo Salz, Synopsys, Inc.
Steven Sharp, Cadence Design Systems, Inc.
Mark Strickland, Cisco Systems, Inc.
Brandon Tipp, Intel Corporation

The **Assertions Committee (SV-AC)** was responsible for the specification of the assertion features of SystemVerilog.

**Dmitry Korchemny**, Synopsys, Inc., *Chair*
**Erik Seligman**, Intel Corporation, *Co-Chair*

Mehbub Ali, Intel Corporation
Shalom Bresticker, Accellera Systems Initiative
Eduard Cerny, Synopsys, Inc.
Ang Boon Chong, Intel Corporation
Ben Cohen, Accellera Systems Initiative
Manisha Kulshrestha, Mentor Graphics Corporation
Anupam Prabhakar, Mentor Graphics Corporation
Samik Sengupta, Synopsys, Inc.

The **Discrete Committee (SV-DC)** was responsible for defining features to support modeling of analog/mixed-signal circuit components in the discrete domain.

**Scott Little,** Mentor, a Siemens Business, *Chair*
**Scott Cranston**, Cadence Design Systems, Inc., *Co-Chair*

Kevin Cameron, Samsung
Shekar Chetput, Cadence Design Systems, Inc.
Dave Cronauer, Synopsys, Inc.
Mark Hartoog, Synopsys, Inc.
Arturo Salz, Synopsys, Inc.
Aaron Spratt, Cadence Design Systems, Inc.
Martin Vlach, Mentor, a Siemens Business
Gordon Vreugdenhil, Mentor, a Siemens Business

# Introduction

This introduction is not part of IEEE Std 1800-2017, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.

The purpose of this standard is to provide the electronic design automation (EDA), semiconductor, and system design communities with a well-defined and official IEEE unified hardware design, specification, and verification standard language. The language is designed to coexist and enhance the hardware description and verification languages (HDVLs) presently used by designers while providing the capabilities lacking in those languages.

SystemVerilog is a unified hardware design, specification, and verification language based on the Accellera SystemVerilog 3.1a extensions to the Verilog hardware description language (HDL) [B4], published in 2004. Accellera is a consortium of EDA, semiconductor, and system companies. IEEE Std 1800 enables a productivity boost in design and validation and covers design, simulation, validation, and formal assertion-based verification flows.

SystemVerilog enables the use of a unified language for abstract and detailed specification of the design, specification of assertions, coverage, and testbench verification based on manual or automatic methodologies. SystemVerilog offers application programming interfaces (APIs) for coverage and assertions, and a direct programming interface (DPI) to access proprietary functionality. SystemVerilog offers methods that allow designers to continue to use present design languages when necessary to leverage existing designs and intellectual property (IP). This standardization project will provide the VLSI design engineers with a well-defined IEEE standard, which meets their requirements in design and validation, and which enables a step function increase in their productivity. This standardization project will also provide the EDA industry with a standard to which they can adhere and that they can support in order to deliver their solutions in this area.

# Contents

## List of figures

## List of tables

# List of syntax excerpts

# Part One: Design and Verification Constructs

# IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language

## 1. Overview

### 1.1 Scope

This standard provides the definition of the language syntax and semantics for the IEEE 1800™ SystemVerilog language, which is a unified hardware design, specification, and verification language. The standard includes support for behavioral, register transfer level (RTL), and gate-level hardware descriptions; testbench, coverage, assertion, object-oriented, and constrained random constructs; and also provides application programming interfaces (APIs) to foreign programming languages.

### 1.2 Purpose

This standard develops the IEEE 1800 SystemVerilog language in order to meet the increasing usage of the language in specification, design, and verification of hardware. This revision corrects errors and clarifies aspects of the language definition in IEEE Std 1800-2012.[1] This revision also provides enhanced features that ease design, improve verification, and enhance cross-language interactions.

### 1.3 Content summary

This standard serves as a complete specification of the SystemVerilog language. This standard contains the following:

- The formal syntax and semantics of all SystemVerilog constructs
- Simulation system tasks and system functions, such as text output display commands
- Compiler directives, such as text substitution macros and simulation time scaling
- The Programming Language Interface (PLI) mechanism
- The formal syntax and semantics of the SystemVerilog Verification Procedural Interface (VPI)
- An Application Programming Interface (API) for coverage access not included in VPI

[1]Information on references can be found in Clause 2.

- Direct programming interface (DPI) for interoperation with the C programming language
- VPI, API, and DPI header files
- Concurrent assertion formal semantics
- The formal syntax and semantics of standard delay format (SDF) constructs
- Informative usage examples

NOTE—An earlier standard, IEEE Std 1800-2009, represented a merger of two previous standards: IEEE Std 1364™-2005 and IEEE Std 1800-2005. In these previous standards, Verilog® was the base language and defined a completely self-contained standard. SystemVerilog defined a number of significant extensions to Verilog, but IEEE Std 1800-2005 was not a self-contained standard; IEEE Std 1800-2005 referred to, and relied on, IEEE Std 1364-2005. These two standards were designed to be used as one language. Merging the base Verilog language into the SystemVerilog standard enabled users to have all information regarding syntax and semantics in a single document.[2, 3]

## 1.4 Special terms

Throughout this standard, the following terms apply:

- *SystemVerilog 3.1a* refers to the Accellera *SystemVerilog 3.1a Language Reference Manual* [B4], a precursor to IEEE Std 1800-2005.[4]
- *Verilog* refers to IEEE Std 1364-2005 for the Verilog hardware description language (HDL).
- *Language Reference Manual* (*LRM*) refers to the document describing a Verilog or SystemVerilog standard.
- *Tool* refers to a software implementation that reads SystemVerilog source code, such as a logic simulator.

NOTE—In IEEE Std 1800-2005, *SystemVerilog* referred to just the extensions to the IEEE 1364-2005 Verilog language and did not include the Verilog base language.

## 1.5 Conventions used in this standard

This standard is organized into clauses, each of which focuses on a specific area of the language. There are subclauses within each clause to discuss individual constructs and concepts. The discussion begins with an introduction and an optional rationale for the construct or the concept, followed by syntax and semantic descriptions, followed by examples and notes.

The terminology conventions used throughout this standard are as follows:

- The word *shall* is used to indicate mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*).
- The word *should* is used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*).
- The word *may* is used to indicate a course of action permissible within the limits of the standard (*may* equals *is permitted to*).
- The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals *is able to*).

[2]Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.
[3]Verilog is a registered trademark of Cadence Design Systems, Inc.
[4]The numbers in brackets correspond to those of the bibliography in Annex Q.

## 1.6 Syntactic description

The main text uses the following conventions:

— *Italicized* font when a term is being defined

— `Constant-width` font for examples, file names, and references to constants, especially `0`, `1`, `x`, and `z` values

— **`Boldface constant-width`** font for SystemVerilog keywords, when referring to the actual keyword

The formal syntax of SystemVerilog is described using Backus-Naur Form (BNF). The following conventions are used:

— Lowercase words, some containing embedded underscores, denote syntactic categories. For example:

module_declaration

— **Boldface-red** characters denote reserved keywords, operators, and punctuation marks as a required part of the syntax. For example:

**`module   =>   ;`**

— A vertical bar ( | ) that is not in boldface-red separates alternative items. For example:

unary_operator ::=
  **+** | **-** | **!** | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**

— Square brackets ( [ ] ) that are not in boldface-red enclose optional items. For example:

function_declaration ::= **`function`** [ lifetime ] function_body_declaration

— Braces ( { } ) that are not in boldface-red enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:

list_of_param_assignments ::= param_assignment { **,** param_assignment }

list_of_param_assignments ::=
  param_assignment
  | list_of_param_assignments **,** param_assignment

A *qualified term* in the syntax is a term such as *array_identifier* for which the "array" portion represents some semantic intent and the "identifier" term indicates that the qualified term reduces to the "identifier" term in the syntax. The syntax does not completely define the semantics of such qualified terms; for example while an identifier that would qualify semantically as an array_identifier is created by a declaration, such declaration forms are not explicitly described using *array_identifier* in the syntax.

## 1.7 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. The places where color is used are the following:

— Cross references that are hyperlinked to other portions of this standard are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).

— Syntactic keywords and tokens in the formal language definitions are shown in **`boldface-red text`**.

— Some figures use a minimal amount of color to enhance readability.

## 1.8 Contents of this standard

A synopsis of the clauses and annexes is presented as a quick reference. All clauses and several of the annexes are normative parts of this standard. Some annexes are included for informative purposes only.

### Part One: Design and Verification Constructs

Clause 1 describes the contents of this standard and the conventions used in this standard.

Clause 2 lists references to other standards that are required in order to implement this standard.

Clause 3 introduces the major building blocks that make up a SystemVerilog design and verification environment: modules, programs, interfaces, checkers, packages, and configurations. This clause also discusses primitives, name spaces, the `$unit` compilation space, and the concept of simulation time.

Clause 4 describes the SystemVerilog simulation scheduling semantics.

Clause 5 describes the lexical tokens used in SystemVerilog source text and their conventions.

Clause 6 describes SystemVerilog data objects and types, including nets and variables, their declaration syntax and usage rules, and charge strength of the values on nets. This clause also discusses strings and string methods, enumerated types, user-defined types, constants, data scope and lifetime, and type compatibility.

Clause 7 describes SystemVerilog compound data types: structures, unions, arrays, including packed and unpacked arrays, dynamic arrays, associative arrays, and queues. This clause also describes various array methods.

Clause 8 describes the object-oriented programming capabilities in SystemVerilog. Topics include defining classes, interface classes, dynamically constructing objects, inheritance and subclasses, data hiding and encapsulation, polymorphism, and parameterized classes.

Clause 9 describes the SystemVerilog procedural blocks: **`initial`**, **`always`**, **`always_comb`**, **`always_ff`**, **`always_latch`**, and **`final`**. Sequential and parallel statement grouping, block names, statement labels, and process control are also described.

Clause 10 describes continuous assignments, blocking and nonblocking procedural assignments, and procedural continuous assignments.

Clause 11 describes the operators and operands that can be used in expressions.

Clause 12 describes SystemVerilog procedural programming statements, such as decision statements and looping constructs.

Clause 13 describes tasks and functions, which are subroutines that can be called from more than one place in a behavioral model.

Clause 14 defines clocking blocks, input and output skews, cycle delays, and default clocking.

Clause 15 describes interprocess communications using event types and event controls, and built-in semaphore and mailbox classes.

Clause 16 describes immediate and concurrent assertions, properties, sequences, sequence operations, multiclock sequences, and clock resolution.

Clause 17 describes checkers. Checkers allow the encapsulation of assertions and modeling code to create a single verification entity.

Clause 18 describes generating random numbers, constraining random number generation, dynamically changing constraints, seeding random number generators (RNGs), and randomized **`case`** statement execution.

Clause 19 describes coverage groups, coverage points, cross coverage, coverage options, and coverage methods.

Clause 20 describes most of the built-in system tasks and system functions.

Clause 21 describes additional system tasks and system functions that are specific to input/output (I/O) operations.

Clause 22 describes various compiler directives, including a directive for controlling reserved keyword compatibility between versions of previous Verilog and SystemVerilog standards.

## Part Two: Hierarchy Constructs

Clause 23 describes how hierarchies are created in SystemVerilog using module instances and interface instances, and port connection rules. This clause also discusses the `$root` top-level instances, nested modules, extern modules, identifier search rules, how parameter values can be overridden, and binding auxiliary code to scopes or instances.

Clause 24 describes the testbench program construct, the elimination of testbench race conditions, and program control tasks.

Clause 25 describes interface syntax, interface ports, modports, interface subroutines, parameterized interfaces, virtual interfaces, and accessing objects within interfaces.

Clause 26 describes user-defined packages and the std built-in package.

Clause 27 describes the generate construct and how generated constructs can be used to do conditional or multiple instantiations of procedural code or hierarchy.

Clause 28 describes the gate- and switch-level primitives and logic strength modeling.

Clause 29 describes how a user-defined primitive (UDP) can be defined and how these primitives are included in SystemVerilog models.

Clause 30 describes how to specify timing relationships between input and output ports of a module.

Clause 31 describes how timing checks are used in specify blocks to determine whether signals obey the timing constraints.

Clause 32 describes the syntax and semantics of SDF constructs.

Clause 33 describes how to configure the contents of a design.

Clause 34 describes encryption and decryption of source text regions.

## Part Three: Application Programming Interfaces

Clause 35 describes SystemVerilog's direct programming interface (DPI), a direct interface to foreign languages and the syntax for importing functions from a foreign language and exporting subroutines to a foreign language.

Clause 36 provides an overview of the programming language interface (PLI and VPI).

Clause 37 presents the VPI data model diagrams, which document the VPI object relationships and access methods.

Clause 38 describes the VPI routines.

Clause 39 describes the assertion API in SystemVerilog.

Clause 40 describes the coverage API in SystemVerilog.

## Part Four: Annexes

Annex A (normative) defines the formal syntax of SystemVerilog, using BNF.

Annex B (normative) lists the SystemVerilog keywords.

Annex C (informative) lists constructs that have been deprecated from SystemVerilog. The annex also discusses the possible deprecation of the **`defparam`** statement and the procedural **`assign`**/**`deassign`** statements.

Annex D (informative) describes system tasks and system functions that are frequently used, but that are not required in this standard.

Annex E (informative) describes compiler directives that are frequently used, but that are not required in this standard.

Annex F (normative) describes a formal semantics for SystemVerilog concurrent assertions.

Annex G (normative) describes the SystemVerilog standard package, containing type definitions for mailbox, semaphore, randomize, and process.

Annex H (normative) defines the C language layer for the SystemVerilog DPI.

Annex I (normative) defines the standard `svdpi.h` include file for use with SystemVerilog DPI applications.

Annex J (normative) describes common guidelines for the inclusion of foreign language code into a SystemVerilog application.

Annex K (normative) provides a listing of the contents of the `vpi_user.h` file.

Annex L (normative) provides a listing of the contents of the `vpi_compatibility.h` file, which extends the `vpi_user.h` include file.

Annex M (normative) provides a listing of the contents of the `sv_vpi_user.h` file, which extends the `vpi_user.h` include file.

Annex N (normative) provides the C source code for the SystemVerilog random distribution system functions.

Annex O (informative) describes various scenarios that can be used for intellectual property (IP) protection, and it also shows how the relevant pragmas can be used to achieve the desired effect of securely protecting, distributing, and decrypting the model.

Annex P (informative) defines terms that are used in this standard.

Annex Q (informative) lists reference documents that are related to this standard.

## 1.9 Deprecated clauses

Annex C lists constructs that appeared in previous versions of either IEEE Std 1364 or IEEE Std 1800, but that have been deprecated and do not appear in this standard. This annex also lists constructs that appear in this standard, but that are under consideration for deprecation in a future version of this standard.

## 1.10 Examples

Small SystemVerilog code examples are shown throughout this standard. These examples are informative. They are intended to illustrate the usage of SystemVerilog constructs in a simple context and do not define the full syntax.

## 1.11 Prerequisites

Some clauses of this standard presuppose a working knowledge of the C programming language.

# 2. Normative references

The following referenced documents are indispensable for the application of this standard (i.e., they must be understood and used, so each referenced document is cited in the text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

Anderson, R., Biham, E., and Knudsen, L. “Serpent: A Proposal for the Advanced Encryption Standard,” NIST AES Proposal, 1998.[5]

ANSI X9.52-1998, American National Standard for Financial Services—Triple Data Encryption Algorithm Modes of Operation.[6]

ElGamal, T., “A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms,” *IEEE Transactions on Information Theory*, vol. IT-31, no. 4, pp. 469–472, July 1985.

FIPS 46-3 (October 1999), Data Encryption Standard (DES).[7]

FIPS 180-2 (August 2002), Secure Hash Standard (SHS).

FIPS 197 (November 2001), Advanced Encryption Standard (AES).

IEEE Std 754™, IEEE Standard for Floating-Point Arithmetic.[8, 9]

IEEE Std 1003.1™, IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®).

IEEE Std 1364™-1995, IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language.

IEEE Std 1364™-2001, IEEE Standard Verilog Hardware Description Language.

IEEE Std 1364™-2005, IEEE Standard for Verilog Hardware Description Language.

IEEE Std 1800™-2005, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.

IEEE Std 1800™-2009, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.

IEEE Std 1800™-2012, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.

IETF RFC 1319 (April 1992), The MD2 Message-Digest Algorithm.[10]

IETF RFC 1321 (April 1992), The MD5 Message-Digest Algorithm.

IETF RFC 2045 (November 1996), Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies.

---

[5]This document is available at http://www.cl.cam.ac.uk/~rja14/Papers/serpent.tar.gz.

[6]ANSI publications are available from the American National Standards Institute (http://www.ansi.org/).

[7]FIPS publications are available from the National Technical Information Service (http://www.ntis.gov/).

[8]IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).

[9]The IEEE standards or products referred to in this clause are trademarks of The Institute of Electrical and Electronics Engineers, Inc.

[10]IETF documents (i.e., RFCs) are available for download at http://www.rfc-archive.org/.

IETF RFC 2144 (May 1997), The CAST-128 Encryption Algorithm.

IETF RFC 2437 (October 1998), PKCS #1: RSA Cryptography Specifications, Version 2.0.

IETF RFC 2440 (November 1998), OpenPGP Message Format.

ISO/IEC 10118-3:2004, Information technology—Security techniques—Hash-functions—Part 3: Dedicated hash-functions.[11]

Schneier, B., "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993)*, Springer-Verlag, 1994, pp. 191–204.

Schneier, B., et al., *The Twofish Encryption Algorithm: A 128-Bit Block Cipher*, 1st ed., Wiley, 1999.

[11]ISO publications are available from the International Organization for Standardization (http://www.iso.org/). IEC publications are available from the International Electrotechnical Commission (http://www.iec.ch). ISO/IEC publications are available from the American National Standards Institute (http://www.ansi.org/)..

# 3. Design and verification building blocks

## 3.1 General

This clause describes the following:

- The purpose of modules, programs, interfaces, checkers, and primitives
- An overview of subroutines
- An overview of packages
- An overview of configurations
- An overview of design hierarchy
- Definition of compilation and elaboration
- Declaration name spaces
- Simulation time, time units, and time precision

This clause defines several important SystemVerilog terms and concepts that are used throughout this document. The clause also provides an overview of the purpose and usage of the modeling blocks used to represent a hardware design and its verification environment.

## 3.2 Design elements

A *design element* is a SystemVerilog module (see Clause 23), program (see Clause 24), interface (see Clause 25), checker (see Clause 17), package (see Clause 26), primitive (see Clause 28) or configuration (see Clause 33). These constructs are introduced by the keywords **`module`**, **`program`**, **`interface`**, **`checker`**, **`package`**, **`primitive`**, and **`config`**, respectively.

Design elements are the primary building blocks used to model and build up a design and verification environment. These building blocks are the containers for the declarations and procedural code that are discussed in subsequent clauses of this document.

This clause describes the purpose of these building blocks. Full details on the syntax and semantics of these blocks are defined in later clauses of this standard.

## 3.3 Modules

The basic building block in SystemVerilog is the module, enclosed between the keywords **`module`** and **`endmodule`**. Modules are primarily used to represent design blocks, but can also serve as containers for verification code and interconnections between verification blocks and design blocks. Some of the constructs that modules can contain include the following:

- Ports, with port declarations
- Data declarations, such as nets, variables, structures, and unions
- Constant declarations
- User-defined type definitions
- Class definitions
- Imports of declarations from packages
- Subroutine definitions
- Instantiations of other modules, programs, interfaces, checkers, and primitives
- Instantiations of class objects
- Continuous assignments

— Procedural blocks
— Generate blocks
— Specify blocks

Each of the constructs in the preceding list is discussed in detail in subsequent clauses of this standard.

NOTE—The preceding list is not all inclusive. Modules can contain additional constructs, which are also discussed in subsequent clauses of this standard.

Following is a simple example of a module that represents a 2-to-1 multiplexer:

```
module mux2to1 (input  wire  a, b, sel, // combined port and type declaration
                output logic y);

  always_comb begin    // procedural block
    if (sel) y = a;    // procedural statement
    else     y = b;
  end
endmodule: mux2to1
```

Modules are presented in more detail in Clause 23. See also 3.11 on creating design hierarchy with modules.

## 3.4 Programs

The program building block is enclosed between the keywords **program**...**endprogram**. This construct is provided for modeling the testbench environment. The module construct works well for the description of hardware. However, for the testbench, the emphasis is not on the hardware-level details such as wires, structural hierarchy, and interconnects, but in modeling the complete environment in which a design is verified.

The program block serves the following three basic purposes:

— It provides an entry point to the execution of testbenches.
— It creates a scope that encapsulates program-wide data, tasks, and functions.
— It provides a syntactic context that specifies scheduling in the Reactive region.

The program construct serves as a clear separator between design and testbench, and, more importantly, it specifies specialized simulation execution semantics. Together with **clocking** blocks (see Clause 14), the program construct provides for race-free interaction between the design and the testbench and enables cycle- and transaction-level abstractions.

A program block can contain data declarations, class definitions, subroutine definitions, object instances, and one or more initial or final procedures. It cannot contain always procedures, primitive instances, module instances, interface instances, or other program instances.

The abstraction and modeling constructs of SystemVerilog simplify the creation and maintenance of testbenches. The ability to instantiate and individually connect each program instance enables their use as generalized models.

A sample program declaration is as follows:

```
program test (input clk, input [16:1] addr, inout [7:0] data);
   initial begin
     ...
endprogram
```

The program construct is discussed more fully in Clause 24.

## 3.5 Interfaces

The interface construct, enclosed between the keywords **interface**...**endinterface**, encapsulates the communication between design blocks, and between design and verification blocks, allowing a smooth migration from abstract system-level design through successive refinement down to lower level register-transfer and structural views of the design. By encapsulating the communication between blocks, the interface construct also facilitates design reuse.

At its lowest level, an interface is a named bundle of nets or variables. The interface is instantiated in a design and can be connected to interface ports of other instantiated modules, interfaces and programs. An interface can be accessed through a port as a single item, and the component nets or variables referenced where needed. A significant proportion of a design often consists of port lists and port connection lists, which are just repetitions of names. The ability to replace a group of names by a single name can significantly reduce the size of a description and improve its maintainability.

Additional power of the interface comes from its ability to encapsulate functionality as well as connectivity, making an interface, at its highest level, more like a class template. An interface can have parameters, constants, variables, functions, and tasks. The types of elements in an interface can be declared, or the types can be passed in as parameters. The member variables and functions are referenced relative to the instance name of the interface as instance members. Thus, modules that are connected via an interface can simply call the subroutine members of that interface to drive the communication. With the functionality thus encapsulated in the interface and isolated from the module, the abstraction level and/or granularity of the communication protocol can be easily changed by replacing the interface with a different interface containing the same members, but implemented at a different level of abstraction. The modules connected via the interface do not need to change at all.

To provide direction information for module ports and to control the use of subroutines within particular modules, the **modport** construct is provided. As the name indicates, the directions are those seen from the module.

In addition to subroutine methods, an interface can also contain processes (i.e., initial or always procedures) and continuous assignments, which are useful for system-level modeling and testbench applications. This allows the interface to include, for example, its own protocol checker, which automatically verifies that all modules connected via the interface conform to the specified protocol. Other applications, such as functional coverage recording and reporting, protocol checking, and assertions can also be built into the interface.

A simple example of an interface definition and usage is as follows:

```
interface simple_bus(input logic clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;
endinterface: simple_bus

module memMod(simple_bus a); // simple_bus interface port
   logic avail;
   // When memMod is instantiated in module top, a.req is the req
   // signal in the sb_intf instance of the 'simple_bus' interface
   always @(posedge a.clk) a.gnt <= a.req & avail;
endmodule
```

```
module cpuMod(simple_bus b); // simple_bus interface port
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(.clk(clk)); // Instantiate the interface

   memMod mem(.a(sb_intf)); // Connect interface to module instance
   cpuMod cpu(.b(sb_intf)); // Connect interface to module instance

endmodule
```

See Clause 25 for a full description of interfaces.

## 3.6 Checkers

The checker construct, enclosed by the keywords **checker**...**endchecker**, represents a verification block encapsulating assertions along with the modeling code. The intended use of checkers is to serve as verification library units or as building blocks for creating abstract auxiliary models used in formal verification. The checker construct is discussed in detail in Clause 17.

## 3.7 Primitives

The *primitive* building block is used to represent low-level logic gates and switches. SystemVerilog includes a number of built-in primitive types. Designers can supplement the built-in primitives with *user-defined primitives* (UDPs). A UDP is enclosed between the keywords **primitive**...**endprimitive**. The built-in and UDP constructs allow modeling timing-accurate digital circuits, commonly referred to as *gate-level models*. Gate-level modeling is discussed more fully in Clause 28 through Clause 31.

## 3.8 Subroutines

*Subroutines* provide a mechanism to encapsulate executable code that can be invoked from one or more places. There are two forms of subroutines, tasks (13.3) and functions (13.4).

A task is called as a statement. A task can have any number of input, output, inout, and ref arguments, but does not return a value. Tasks can block simulation time during execution. That is, the task exit can occur at a later simulation time than when the task was called.

A function can return a value or can be defined as a void function, which does not return a value. A nonvoid function call is used as an operand within an expression. A void function is called as a statement. A function can have input, output, inout, and ref arguments. Functions must execute without blocking simulation time, but can fork off processes that do block time.

## 3.9 Packages

Modules, interfaces, programs, and checkers provide a local name space for declarations. Identifiers declared within a module, interface, program, or checker are local to that scope, and do not affect or conflict with declarations in other building blocks.

Packages provide a declaration space, which can be shared by other building blocks. Package declarations can be imported into other building blocks, including other packages.

A package is defined between the keywords **package**...**endpackage**. For example:

```
package ComplexPkg;
   typedef struct {
      shortreal i, r;
   } Complex;

   function Complex add(Complex a, b);
      add.r = a.r + b.r;
      add.i = a.i + b.i;
   endfunction

   function Complex mul(Complex a, b);
      mul.r = (a.r * b.r) - (a.i * b.i);
      mul.i = (a.r * b.i) + (a.i * b.r);
   endfunction
endpackage : ComplexPkg
```

The full syntax and semantics of packages are described in Clause 26.

## 3.10 Configurations

SystemVerilog provides the ability to specify design configurations, which specify the binding information of module instances to specific SystemVerilog source code. Configurations utilize libraries. A library is a collection of modules, interfaces, programs, checkers, primitives, packages, and other configurations. Separate library map files specify the source code location for the cells contained within the libraries. The names of the library map files are typically specified as invocation options to simulators or other software tools reading in SystemVerilog source code.

See Clause 33 for details of configurations.

## 3.11 Overview of hierarchy

The basic building blocks of modules, programs, interfaces, checkers, and primitives are used to build up a *design hierarchy*. Hierarchy is created by one building block *instantiating* another building block. When a module contains an *instance* of another module, interface, program, or checker, a new level of hierarchy is created. Communication through levels of hierarchy is primarily through connections to the ports of the instantiated module, interface, program, or checker.

Following is a simple example of two module declarations that utilize some simple declarations. Module `top` contains an instance of module `mux2to1`, creating a design with two levels of hierarchy.

```
module top;    // module with no ports
   logic in1, in2, select;     // variable declarations
   wire  out1;                 // net declaration

   mux2to1 m1 (.a(in1), .b(in2), .sel(select), .y(out1));  // module instance

endmodule: top

module mux2to1 (input  wire  a, b, sel, // combined port and type declaration
                output logic y);
```

```
      // netlist using built-in primitive instances
      not g1 (sel_n, sel);
      and g2 (a_s, a, sel_n);
      and g3 (b_s, b, sel);
      or  g4 (y, a_s, b_s);
   endmodule: mux2to1
```

Modules can instantiate other modules, programs, interfaces, checkers, and primitives, creating a hierarchy tree. Interfaces can also instantiate other building blocks and create a hierarchy tree. Programs and checkers can instantiate other checkers. Primitives cannot instantiate other building blocks; they are leaves in a hierarchy tree.

Normally, a module or program that is elaborated but not explicitly instantiated is implicitly instantiated once at the top of the hierarchy tree and becomes a *top-level hierarchy block* (see 23.3 and 24.3). SystemVerilog permits multiple top-level blocks.

Identifiers within any level of hierarchy can be referenced from any other level of hierarchy using *hierarchical path names* (see 23.6).

Instantiation syntax and design hierarchy are presented in more detail in Clause 23.

## 3.12 Compilation and elaboration

*Compilation* is the process of reading in SystemVerilog source code, decrypting encrypted code, and analyzing the source code for syntax and semantic errors. Implementations may execute compilation in one or more passes. Implementations may save compiled results in a proprietary intermediate format, or may pass the compiled results directly to an elaboration phase. Not all syntax and semantics can be checked during the compilation process. Some checking can only be done during or at the completion of elaboration.

SystemVerilog supports both single file and multiple file compilation through the use of compilation units (see 3.12.1).

*Elaboration* is the process of binding together the components that make up a design. These components can include module instances, program instances, interface instances, checker instances, primitive instances, and the top level of the design hierarchy. Elaboration occurs after parsing the source code and before simulation; and it involves expanding instantiations, computing parameter values, resolving hierarchical names, establishing net connectivity and in general preparing the design for simulation. See 23.10.4 for additional details on the elaboration process.

Although this standard defines the results of compilation and elaboration, the compilation and elaboration steps are not required to be distinct phases in an implementation. Throughout this standard the terms *compilation, compile,* and *compiler* normally refer to the combined compilation and elaboration process. So for example, when the standard refers to a “compile time error,” an implementation is permitted to report the error at any time prior to the start of simulation.

This standard does not normally specify requirements regarding the order of compilation for design elements. The two exceptions are the rules regarding “compilation units” (see 3.12.1) where actual file boundaries during compilation are significant, and the rules regarding references to package items (see 26.3) where the compilation of a package is required to precede references to it.

### 3.12.1 Compilation units

SystemVerilog supports separate compilation using compiled units. The following terms and definitions are provided:

— **compilation unit:** A collection of one or more SystemVerilog source files compiled together.
— **compilation-unit scope:** A scope that is local to the compilation unit. It contains all declarations that lie outside any other scope.
— **$unit:** The name used to explicitly access the identifiers in the compilation-unit scope.

The exact mechanism for defining which files constitute a compilation unit is tool-specific. However, compliant tools shall provide use models that allow both of the following cases:

a) All files on a given compilation command line make a single compilation unit (in which case the declarations within those files are accessible following normal visibility rules throughout the entire set of files).
b) Each file is a separate compilation unit (in which case the declarations in each compilation-unit scope are accessible only within its corresponding file).

The contents of files included using one or more `` `include `` directives become part of the compilation unit of the file within which they are included.

If there is a declaration that is incomplete at the end of a file, then the compilation unit including that file will extend through each successive file until there are no incomplete declarations at the end of the group of files.

There are other possible mappings of files to compilation units, and the mechanisms for defining them are tool-specific and may not be portable.

Although the compilation-unit scope is not a package, it can contain any item that can be defined within a package (see 26.2) and bind constructs as well (see 23.11). These items are in the compilation-unit scope name space (see 3.13).

The following items are visible in all compilation units: modules, primitives, programs, interfaces, and packages. Items defined in the compilation-unit scope cannot be accessed by name from outside the compilation unit. The items in a compilation-unit scope can be accessed using the PLI, which must provide an iterator to traverse all the compilation units.

Items in a compilation-unit scope can have hierarchical references to identifiers. For upwards name referencing (see 23.8), the compilation-unit scope is treated like a top-level design unit. This means that if these are not references to identifiers created within the compilation-unit scope or made visible by import of a package into the compilation unit scope, they are treated as full path names starting at the top of the design (`$root`, described in 23.3.1).

Within a separately compiled unit, compiler directives once seen by a tool apply to all subsequent source text. However, compiler directives from one separately compiled unit shall not affect other compilation units. This may result in a difference of behavior between compiling the units separately or as a single compilation unit containing the entire source.

When an identifier is referenced within a scope

— First, the nested scope is searched (see 23.9) (including nested module declarations), including any identifiers made available through package import declarations.
— Next, the portion of the compilation-unit scope defined prior to the reference is searched (including any identifiers made available through package import declarations).
— Finally, if the identifier follows hierarchical name resolution rules, the instance hierarchy is searched (see 23.8 and 23.9).

`$unit` is the name of the scope that encompasses a compilation unit. Its purpose is to allow the unambiguous reference to declarations at the outermost level of a compilation unit (i.e., those in the compilation-unit scope). This is done via the same scope resolution operator used to access package items (see 26.3).

For example:

```
bit b;
task t;
   int b;
   b = 5 + $unit::b;    // $unit::b is the one outside
endtask
```

Other than for task and function names (see 23.8.1), references shall only be made to names already defined in the compilation unit. The use of an explicit `$unit::` prefix only provides for name disambiguation and does not add the ability to refer to later compilation unit items.

For example:

```
task t;
   int x;
   x = 5 + b;         // illegal - "b" is defined later
   x = 5 + $unit::b; // illegal - $unit adds no special forward referencing
endtask
bit b;
```

The compilation-unit scope allows users to easily share declarations (e.g., types) across the unit of compilation, but without having to declare a package from which the declarations are subsequently imported. Because it has no name, the compilation-unit scope cannot be used with an import declaration, and the identifiers declared within the scope are not accessible via hierarchical references. Within a particular compilation unit, however, the special name `$unit` can be used to explicitly access the declarations of its compilation-unit scope.

## 3.13 Name spaces

SystemVerilog has eight name spaces for identifiers: two are global (definitions name space and package name space), two are global to the compilation unit (compilation unit name space and text macro name space), and four are local. The eight name spaces are described as follows:

a) The *definitions name space* unifies all the non-nested **`module`**, **`primitive`**, **`program`**, and **`interface`** identifiers defined outside all other declarations. Once a name is used to define a module, primitive, program, or interface within one compilation unit, the name shall not be used again (in any compilation unit) to declare another non-nested module, primitive, program, or interface outside all other declarations.

b) The *package name space* unifies all the **`package`** identifiers defined among all compilation units. Once a name is used to define a package within one compilation unit, the name shall not be used again to declare another package within any compilation unit.

c) The *compilation-unit scope name space* exists outside the **`module`**, **`interface`**, **`package`**, **`checker`**, **`program`**, and **`primitive`** constructs. It unifies the definitions of the functions, tasks, checkers, parameters, named events, net declarations, variable declarations, and user-defined types within the compilation-unit scope.

d) The *text macro name space* is global within the compilation unit. Because text macro names are introduced and used with a leading **`** character, they remain unambiguous with any other name space. The text macro names are defined in the linear order of appearance in the set of input files that make up the compilation unit. Subsequent definitions of the same name override the previous definitions for the balance of the input files.

e) The *module name space* is introduced by the **module**, **interface**, **package**, **program**, **checker**, and **primitive** constructs. It unifies the definition of modules, interfaces, programs, checkers, functions, tasks, named blocks, instance names, parameters, named events, net declarations, variable declarations, and user-defined types within the enclosing construct.

f) The *block name space* is introduced by named or unnamed blocks, the **specify**, **function**, and **task** constructs. It unifies the definitions of the named blocks, functions, tasks, parameters, named events, variable type of declaration, and user-defined types within the enclosing construct.

g) The *port name space* is introduced by the **module**, **interface**, **primitive**, and **program** constructs. It provides a means of structurally defining connections between two objects that are in two different name spaces. The connection can be unidirectional (either **input** or **output**) or bidirectional (**inout** or **ref**). The port name space overlaps the module and the block name spaces. Essentially, the port name space specifies the type of connection between names in different name spaces. The port type of declarations includes **input**, **output**, **inout**, and **ref**. A port name introduced in the port name space can be reintroduced in the module name space by declaring a variable or a net with the same name as the port name.

h) The *attribute name space* is enclosed by the `(*` and `*)` constructs attached to a language element (see 5.12). An attribute name can be defined and used only in the attribute name space. Any other type of name cannot be defined in this name space.

Within a name space, it shall be illegal to redeclare a name already declared by a prior declaration.

## 3.14 Simulation time units and precision

An important aspect of simulation is time. The term *simulation time* is used to refer to the time value maintained by the simulator to model the actual time it would take for the system description being simulated. The term *time* is used interchangeably with simulation time.

Time values are used in design elements to represent propagation delays and the amount of simulation time between when procedural statements execute. Time values have two components, a *time unit* and a *time precision*.

— The *time unit* represents the unit of measurement for times and delays, and can be specified in units ranging from 100 second units down to 1 femtosecond units.

— The *time precision* specifies the degree of accuracy for delays.

Both the time units and time precision are represented using one of the character strings: `s`, `ms`, `us`, `ns`, `ps`, and `fs` with an order of magnitude of 1, 10, or 100. The definition of these character strings is given in Table 3-1.

**Table 3-1—Time unit strings**

| Character string | Unit of measurement |
|---|---|
| `s` | seconds |
| `ms` | milliseconds |
| `us` | microseconds |
| `ns` | nanoseconds |
| `ps` | picoseconds |
| `fs` | femtoseconds |

NOTE—While s, ms, ns, ps, and fs are the usual SI unit symbols for second, millisecond, nanosecond, picosecond, and femtosecond, due to lack of the Greek letter µ (mu) in coding character sets, "us" represents the SI unit symbol for microsecond, properly µs.

The time precision of a design element shall be at least as precise as the time unit; it cannot be a longer unit of time than the time unit.

### 3.14.1 Time value rounding

Within a design element, such as a module, program or interface, the time precision specifies how delay values are rounded before being used in simulation.

The time precision is relative to the time units. If the precision is the same as the time units, then delay values are rounded off to whole numbers (integers). If the precision is one order of magnitude smaller than the time units, then delay values are rounded off to one decimal place. For example, if the time unit specified is `1ns` and the precision is `100ps`, then delay values are rounded off to one decimal place (`100ps` is equivalent to `0.1ns`). Thus, a delay of `2.75ns` would be rounded off to `2.8ns`.

The time values in a design element are accurate to within the unit of time precision specified for that design element, even if there is a smaller time precision specified elsewhere in the design.

### 3.14.2 Specifying time units and precision

The time unit and time precision can be specified in the following two ways:

— Using the compiler directive `` `timescale ``

— Using the keywords **`timeunit`** and **`timeprecision`**

#### 3.14.2.1 The `timescale compiler directive

The `` `timescale `` compiler directive specifies the default time unit and precision for all design elements that follow this directive and that do not have **`timeunit`** and **`timeprecision`** constructs specified within the design element. The `` `timescale `` directive remains in effect from when it is encountered in the source code until another `` `timescale `` compiler directive is read. The `` `timescale `` directive only affects the current compilation unit; it does not span multiple compilation units (see 3.12.1).

The general syntax for the `` `timescale `` directive is (see 22.7 for more details):

```
`timescale time_unit / time_precision
```

The following example specifies a time unit of 1 ns with a precision of 10 ps (2 decimal places of accuracy). The compiler directive affects both module `A` and `B`. A second `` `timescale `` directive replaces the first directive, specifying a time unit of 1 ps and precision of 1 ps (zero decimal places of accuracy) for module C.

```
`timescale 1ns / 10ps
module A (...);
   ...
endmodule

module B (...);
   ...
endmodule

`timescale 1ps/1ps
module C (...);
```

```
      ...
   endmodule
```

The `` `timescale `` directive can result in file order dependency problems. If the previous three modules were compiled in the order of `A`, `B`, `C` (as shown) then module `B` would simulate with time units in nanoseconds. If the same three files were compiled in the order of `C`, `B`, `A` then module would simulate with time units in picoseconds. This could cause very different simulation results, depending on the time values specified in module `B`.

#### 3.14.2.2 The timeunit and timeprecision keywords

The time unit and precision can be declared by the **`timeunit`** and **`timeprecision`** keywords, respectively, and set to a time literal (see 5.8). The time precision may also be declared using an optional second argument to the **`timeunit`** keyword using the slash separator. For example:

```
   module D (...);
      timeunit 100ps;
      timeprecision 10fs;
      ...
   endmodule

   module E (...);
      timeunit 100ps / 10fs; // timeunit with optional second argument
      ...
   endmodule
```

Defining the **`timeunit`** and **`timeprecision`** constructs within the design element removes the file order dependency problems with compiler directives.

There shall be at most one time unit and one time precision for any module, program, package, or interface definition or in any compilation-unit scope. This shall define a time scope. If specified, the **`timeunit`** and **`timeprecision`** declarations shall precede any other items in the current time scope. The **`timeunit`** and **`timeprecision`** declarations can be repeated as later items, but must match the previous declaration within the current time scope.

#### 3.14.2.3 Precedence of timeunit, timeprecision, and `timescale

If a **`timeunit`** is not specified within a module, program, package, or interface definition, then the time unit shall be determined using the following rules of precedence:

a) If the module or interface definition is nested, then the time unit shall be inherited from the enclosing module or interface (programs and packages cannot be nested).

b) Else, if a `` `timescale `` directive has been previously specified (within the compilation unit), then the time unit shall be set to the units of the last `` `timescale `` directive.

c) Else, if the compilation-unit scope specifies a time unit (outside all other declarations), then the time unit shall be set to the time units of the compilation unit.

d) Else, the default time unit shall be used.

The time unit of the compilation-unit scope can only be set by a **`timeunit`** declaration, not a `` `timescale `` directive. If it is not specified, then the default time unit shall be used.

If a **`timeprecision`** is not specified in the current time scope, then the time precision shall be determined following the same precedence as with time units.

The default time unit and precision are implementation-specific.

It shall be an error if some design elements have a time unit and precision specified and others do not.

### 3.14.3 Simulation time unit

The *global time precision*, also called the *simulation time unit*, is the minimum of all the **`timeprecision`** statements, all the time precision arguments to **`timeunit`** declarations, and the smallest time precision argument of all the `` `timescale `` compiler directives in the design.

The **`step`** time unit is equal to the global time precision. Unlike other time units, which represent physical units, a **`step`** cannot be used to set or modify either the precision or the time unit.

# 4. Scheduling semantics

## 4.1 General

This clause describes the following:

- Event-based simulation scheduling semantics
- SystemVerilog's stratified event scheduling algorithm
- Determinism and nondeterminism of event ordering
- Possible sources of race conditions
- PLI callback control points

## 4.2 Execution of a hardware model and its verification environment

The balance of the clauses of this standard describe the behavior of each of the elements of the language. This clause gives an overview of the interactions between these elements, especially with respect to the scheduling and execution of events.

The elements that make up the SystemVerilog language can be used to describe the behavior, at varying levels of abstraction, of electronic hardware. SystemVerilog is a parallel programming language. The execution of certain language constructs is defined by parallel execution of blocks or processes. It is important to understand what execution order is guaranteed to the user and what execution order is indeterminate.

Although SystemVerilog is used for more than simulation, the semantics of the language is defined for simulation, and everything else is abstracted from this base definition.

## 4.3 Event simulation

The SystemVerilog language is defined in terms of a discrete event execution model. The discrete event simulation is described in more detail in this clause to provide a context to describe the meaning and valid interpretation of SystemVerilog constructs. These resulting definitions provide the standard SystemVerilog reference algorithm for simulation, which all compliant simulators shall implement. Within the following event execution model definitions, there is a great deal of choice, and differences in some details of execution are to be expected between different simulators. In addition, SystemVerilog simulators are free to use different algorithms from those described in this clause, provided the user-visible effect is consistent with the reference algorithm.

A SystemVerilog description consists of connected threads of execution or processes. Processes are objects that can be evaluated, that can have state, and that can respond to changes on their inputs to produce outputs. Processes are concurrently scheduled elements, such as **`initial`** procedures. Examples of processes include, but are not limited to, primitives; **`initial`**, **`always`**, **`always_comb`**, **`always_latch`**, and **`always_ff`** procedures; continuous assignments; asynchronous tasks; and procedural assignment statements.

Every change in state of a net or variable in the system description being simulated is considered an *update event*.

Processes are sensitive to update events. When an update event is executed, all the processes that are sensitive to that event are considered for evaluation in an arbitrary order. The evaluation of a process is also an event, known as an *evaluation event*.

Evaluation events also include PLI callbacks, which are points in the execution model where PLI application routines can be called from the simulation kernel.

In addition to events, another key aspect of a simulator is time. The term *simulation time* is used to refer to the time value maintained by the simulator to model the actual time it would take for the system description being simulated. The term *time* is used interchangeably with *simulation time* in this clause.

To fully support clear and predictable interactions, a single time slot is divided into multiple regions where events can be scheduled that provide for an ordering of particular types of execution. This allows properties and checkers to sample data when the design under test is in a stable state. Property expressions can be safely evaluated, and testbenches can react to both properties and checkers with zero delay, all in a predictable manner. This same mechanism also allows for nonzero delays in the design, clock propagation, and/or stimulus and response code to be mixed freely and consistently with cycle-accurate descriptions.

## 4.4 Stratified event scheduler

A compliant SystemVerilog simulator shall maintain some form of data structure that allows events to be dynamically scheduled, executed, and removed as the simulator advances through time. The data structure is normally implemented as a time-ordered set of linked lists, which are divided and subdivided in a well-defined manner.

The first division is by time. Every event has one and only one simulation execution time, which at any given point during simulation can be the current time or some future time. All scheduled events at a specific time define a time slot. Simulation proceeds by executing and removing all events in the current simulation time slot before moving on to the next nonempty time slot, in time order. This procedure guarantees that the simulator never goes backwards in time.

A time slot is divided into a set of ordered regions, as follows:

a) Preponed
b) Pre-Active
c) Active
d) Inactive
e) Pre-NBA
f) NBA
g) Post-NBA
h) Pre-Observed
i) Observed
j) Post-Observed
k) Reactive
l) Re-Inactive
m) Pre-Re-NBA
n) Re-NBA
o) Post-Re-NBA
p) Pre-Postponed
q) Postponed

The purpose of dividing a time slot into these ordered regions is to provide predictable interactions between the design and testbench code.

NOTE—These regions essentially encompass the IEEE 1364-2005 reference model for simulation, with exactly the same level of determinism. In other words, legacy Verilog code should continue to run correctly without modification within the SystemVerilog mechanism.

### 3.4.1 Active region sets and reactive region sets

There are two important groupings of event regions that are used to help define the scheduling of SystemVerilog activity, the active region set and the reactive region set. Events scheduled in the Active, Inactive, Pre-NBA, NBA, and Post-NBA regions are *active region set* events. Events scheduled in the Reactive, Re-Inactive, Pre- Re-NBA, Re-NBA, and Post-Re-NBA regions are *reactive region set* events.

The Active, Inactive, Pre-NBA, NBA, Post-NBA, Pre-Observed, Observed, Post-Observed, Reactive, Re-Inactive, Pre-Re-NBA, Re-NBA, Post-Re-NBA, and Pre-Postponed regions are known as the *iterative regions*.

In addition to the active region set and reactive region set, all of the event regions of each time slot can be categorized as *simulation regions* (see 3.4.2) or *PLI regions* (see 3.4.3).

### 3.4.2 Simulation regions

The simulation regions of a time slot are the Preponed, Active, Inactive, NBA, Observed, Reactive, Re-Inactive, Re-NBA and Postponed regions. The flow of execution of the event regions is specified in Figure 4-1.

#### 4.4.2.1 Preponed events region

The `#1step` sampling delay provides the ability to sample data immediately before entering the current time slot. `#1step` sampling is identical to taking the data samples in the Preponed region of the current time slot. Sampling in the Preponed region is equivalent to sampling in the previous Postponed region.

Preponed region PLI events are also scheduled in this region (see 4.4.3.1).

#### 4.4.2.2 Active events region

The Active region holds the current active region set events being evaluated and can be processed in any order.

#### 4.4.2.3 Inactive events region

The Inactive region holds the events to be evaluated after all the Active events are processed.

If events are being executed in the active region set, an explicit `#0` delay control requires the process to be suspended and an event to be scheduled into the Inactive region of the current time slot so that the process can be resumed in the next Inactive to Active iteration.

#### 4.4.2.4 NBA events region

The NBA (nonblocking assignment update) region holds the events to be evaluated after all the Inactive events are processed.

If events are being executed in the active region set, a nonblocking assignment creates an event in the NBA region scheduled for the current or a later simulation time.

#### 4.4.2.5 Observed events region

The Observed region is for evaluation of property expressions when they are triggered. During property evaluation, pass/fail code shall be scheduled in the Reactive region of the current time slot. PLI callbacks are not allowed in the Observed region.

#### 4.4.2.6 Reactive events region

The Reactive region holds the current reactive region set events being evaluated and can be processed in any order.

The code specified by blocking assignments in checkers, program blocks and the code in action blocks of concurrent assertions are scheduled in the Reactive region. The Reactive region is the reactive region set dual of the Active region (see 4.4.2.2).

#### 4.4.2.7 Re-Inactive events region

The Re-Inactive region holds the events to be evaluated after all the Reactive events are processed.

If events are being executed in the reactive region set, an explicit #0 delay control requires the process to be suspended and an event to be scheduled into the Re-Inactive region of the current time slot so that the process can be resumed in the next Re-Inactive to Reactive iteration. The Re-Inactive region is the reactive region set dual of the Inactive region (see 4.4.2.3).

#### 4.4.2.8 Re-NBA events region

The Re-NBA region holds the events to be evaluated after all the Re-Inactive events are processed.

If events are being executed in the reactive region set, a nonblocking assignment creates an event in the Re-NBA update region scheduled for the current or a later simulation time. The Re-NBA region is the reactive region set dual of the NBA region (see 4.4.2.4).

#### 4.4.2.9 Postponed events region

`$monitor`, `$strobe`, and other similar events are scheduled in the Postponed region.

No new value changes are allowed to happen in the current time slot once the Postponed region is reached. Within this region, it is illegal to write values to any net or variable or to schedule an event in any previous region within the current time slot.

Postponed region PLI events are also scheduled in this region (see 4.4.3.10).

### 3.4.3 PLI regions

In addition to the simulation regions, where PLI callbacks can be scheduled, there are additional PLI-specific regions. The PLI regions of a time slot are the Preponed, Pre-Active, Pre-NBA, Post-NBA, Pre-Observed, Post-Observed, Pre-Re-NBA, Post-Re-NBA and Pre-Postponed regions. The flow of execution of the PLI regions is specified in Figure 4-1.

#### 4.4.3.1 Preponed PLI region

The Preponed region provides for a PLI callback control point that allows PLI application routines to access data at the current time slot before any net or variable has changed state. Within this region, it is illegal to write values to any net or variable or to schedule an event in any other region within the current time slot.

NOTE—The PLI currently does not schedule callbacks in the Preponed region.

#### 4.4.3.2 Pre-Active PLI region

The Pre-Active region provides for a PLI callback control point that allows PLI application routines to read and write values and create events before events in the Active region are evaluated (see 4.5).

#### 4.4.3.3 Pre-NBA PLI region

The Pre-NBA region provides for a PLI callback control point that allows PLI application routines to read and write values and create events before the events in the NBA region are evaluated (see 4.5).

#### 4.4.3.4 Post-NBA PLI region

The Post-NBA region provides for a PLI callback control point that allows PLI application routines to read and write values and create events after the events in the NBA region are evaluated (see 4.5).

#### 4.4.3.5 Pre-Observed PLI region

The Pre-Observed region provides for a PLI callback control point that allows PLI application routines to read values after the active region set has stabilized. Within this region, it is illegal to write values to any net or variable or to schedule an event within the current time slot.

#### 4.4.3.6 Post-Observed PLI region

The Post-Observed region provides for a PLI callback control point that allows PLI application routines to read values after properties are evaluated (in the Observed or an earlier region).

NOTE—The PLI currently does not schedule callbacks in the Post-Observed region.

#### 4.4.3.7 Pre-Re-NBA PLI region

The Pre-Re-NBA region provides for a PLI callback control point that allows PLI application routines to read and write values and create events before the events in the Re-NBA region are evaluated (see 4.5).

#### 4.4.3.8 Post-Re-NBA PLI region

The Post-Re-NBA region provides for a PLI callback control point that allows PLI application routines to read and write values and create events after the events in the Re- NBA region are evaluated (see 4.5).

#### 4.4.3.9 Pre-Postponed PLI region

The Pre-Postponed region provides a PLI callback control point that allows PLI application routines to read and write values and create events after processing all other regions except the Postponed region.

#### 4.4.3.10 Postponed PLI region

The Postponed region provides a PLI callback control point that allows PLI application routines to create read-only events after processing all other regions. PLI `cbReadOnlySynch` and other similar events are scheduled in the Postponed region.

The SystemVerilog flow of time slots and event regions is shown in Figure 4-1.

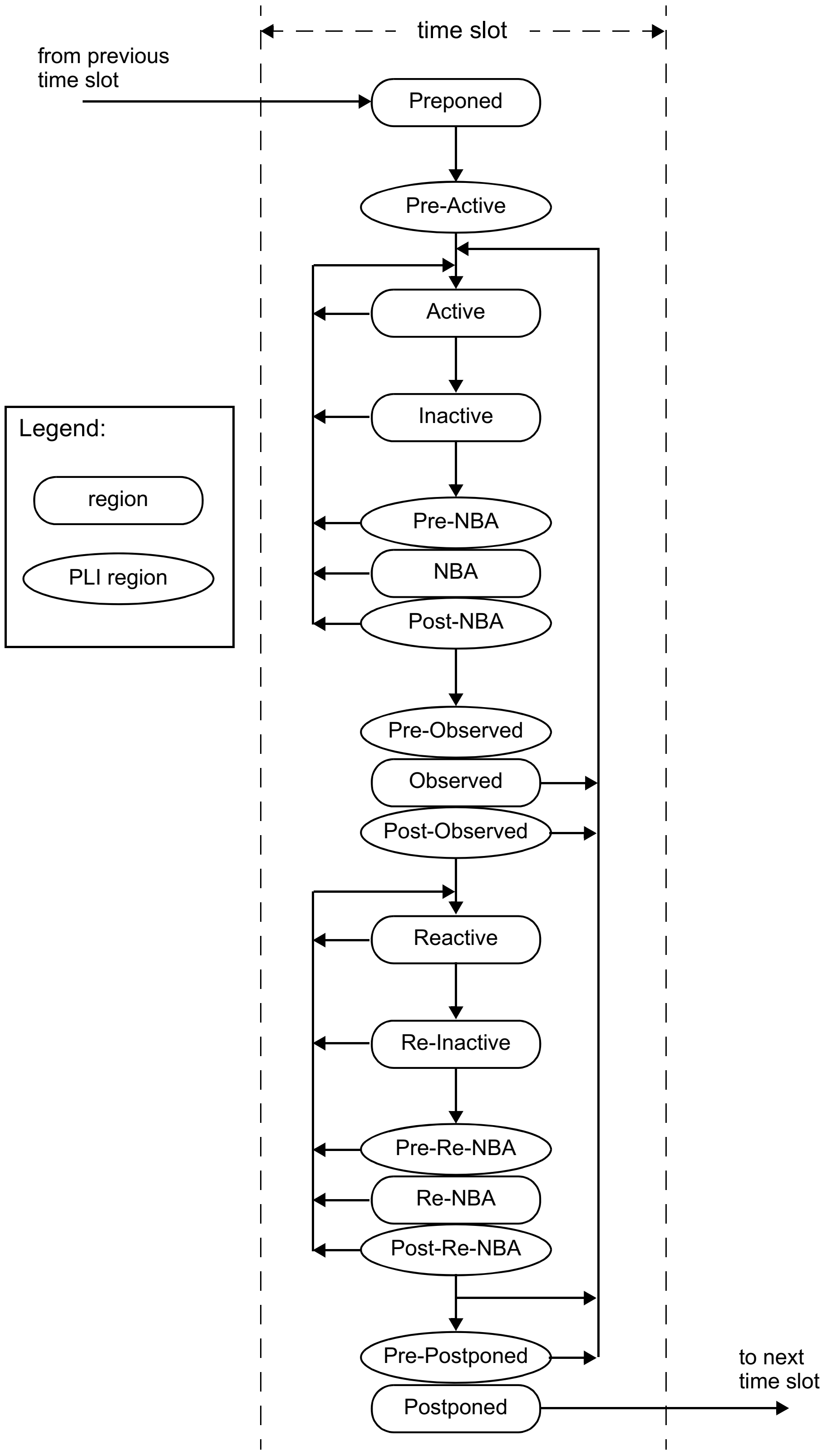


**Figure 4-1—Event scheduling regions**

## 4.5 SystemVerilog simulation reference algorithm

```
execute_simulation {
   T = 0;
   initialize the values of all nets and variables;
   schedule all initialization events into time zero slot;
   while (some time slot is nonempty) {
      move to the first nonempty time slot and set T;
      execute_time_slot (T);
   }
}

execute_time_slot {
   execute_region (Preponed);
   execute_region (Pre-Active);
   while (any region in [Active ... Pre-Postponed] is nonempty) {
      while (any region in [Active ... Post-Observed] is nonempty) {
         execute_region (Active);
         R = first nonempty region in [Active ... Post-Observed];
         if (R is nonempty)
            move events in R to the Active region;
      }
      while (any region in [Reactive ... Post-Re-NBA] is nonempty) {
         execute_region (Reactive);
         R = first nonempty region in [Reactive ... Post-Re-NBA];
         if (R is nonempty)
            move events in R to the Reactive region;
      }
      if (all regions in [Active ... Post-Re-NBA] are empty)
         execute_region (Pre-Postponed);
   }
   execute_region (Postponed);
}

execute_region {
   while (region is nonempty) {
      E = any event from region;
      remove E from the region;
      if (E is an update event) {
         update the modified object;
         schedule evaluation event for any process sensitive to the object;
      } else { /* E is an evaluation event */
         evaluate the process associated with the event and possibly
            schedule further events for execution;
      }
   }
}
```

The Iterative regions and their order are Active, Inactive, Pre-NBA, NBA, Post-NBA, Pre-Observed, Observed, Post-Observed, Reactive, Re-Inactive, Pre-Re-NBA, Re-NBA, Post-Re-NBA, and Pre-Postponed. As shown in the algorithm, once the Reactive, Re-Inactive, Pre-Re-NBA, Re-NBA, or Post-Re-NBA regions are processed, iteration over the other regions does not resume until these five regions are empty.

## 4.6 Determinism

This standard guarantees a certain scheduling order:

a) Statements within a begin-end block shall be executed in the order in which they appear in that begin-end block. Execution of statements in a particular begin-end block can be suspended in favor of other processes in the model; however, in no case shall the statements in a begin-end block be executed in any order other than that in which they appear in the source.
b) NBAs shall be performed in the order the statements were executed (see 10.4.2).

Consider the following example:

```
module test;
   logic a;
   initial begin
      a <= 0;
      a <= 1;
   end
endmodule
```

When this block is executed, there will be two events added to the NBA region. The previous rule requires that they be entered in the event region in execution order, which, in a sequential begin-end block, is source order. This rule requires that they be taken from the NBA region and performed in execution order as well. Hence, at the end of simulation time `0`, the variable `a` will be assigned 0 and then 1.

## 4.7 Nondeterminism

One source of nondeterminism is the fact that active events can be taken off the Active or Reactive event region and processed in any order. Another source of nondeterminism is that statements without time-control constructs in procedural blocks do not have to be executed as one event. Time control statements are the # expression and @ expression constructs (see 9.4). At any time while evaluating a procedural statement, the simulator may suspend execution and place the partially completed event as a pending event in the event region. The effect of this is to allow the interleaving of process execution, although the order of interleaved execution is nondeterministic and not under control of the user.

## 4.8 Race conditions

Because the execution of expression evaluation and net update events may be intermingled, race conditions are possible: For example:

```
assign p = q;
initial begin
   q = 1;
   #1 q = 0;
   $display(p);
end
```

The simulator is correct in displaying either a 1 or a 0. The assignment of 0 to `q` enables an update event for `p`. The simulator may either continue and execute the `$display` task or execute the update for `p`, followed by the `$display` task.

## 4.9 Scheduling implication of assignments

Assignments are translated into processes and events as detailed in 4.9.1 through 4.9.7.

### 4.9.1 Continuous assignment

A continuous assignment statement (see 10.3) corresponds to a process, sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event region, using current values to determine the target. A continuous assignment process is also evaluated at time zero in order to propagate constant values. This includes implicit continuous assignments inferred from port connections (see 3.9.6).

### 4.9.2 Procedural continuous assignment

A procedural continuous assignment (which is the **`assign`** or **`force`** statement; see 10.6) corresponds to a process that is sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event region, using current values to determine the target.

A **`deassign`** or a **`release`** statement deactivates any corresponding **`assign`** or **`force`** statement(s).

### 4.9.3 Blocking assignment

A blocking assignment statement (see 10.4.1) with an intra-assignment delay computes the right-hand side value using the current values, then causes the executing process to be suspended and scheduled as a future event. If the delay is 0, the process is scheduled as an Inactive event for the current time. If a blocking assignment with zero delay is executed from a Reactive region, the process is scheduled as a Re-Inactive event.

When the process is returned (or if it returns immediately if no delay is specified), the process performs the assignment to the left-hand side and enables any events based upon the update of the left-hand side. The values at the time the process resumes are used to determine the target(s). Execution may then continue with the next sequential statement or with other Active or Reactive events.

### 4.9.4 Nonblocking assignment

A nonblocking assignment statement (see 10.4.2) always computes the updated value and schedules the update as an NBA update event, either in the current time step if the delay is zero or as a future event if the delay is nonzero. The values in effect when the update is placed in the event region are used to compute both the right-hand value and the left-hand target.

### 4.9.5 Switch (transistor) processing

The event-driven simulation algorithm described in 4.5 depends on unidirectional signal flow and can process each event independently. The inputs are read, the result is computed, and the update is scheduled.

SystemVerilog provides switch-level modeling in addition to behavioral and gate-level modeling. Switches provide bidirectional signal flow and require coordinated processing of nodes connected by switches.

The source elements that model switches are various forms of transistors, called **`tran`**, **`tranif0`**, **`tranif1`**, **`rtran`**, **`rtranif0`**, and **`rtranif1`**.

Switch processing shall consider all the devices in a bidirectional switch-connected net before it can determine the appropriate value for any node on the net because the inputs and outputs interact. A simulator can do this using a relaxation technique. The simulator can process tran at any time. It can process a subset of tran-connected events at a particular time, intermingled with the execution of other active events.

Further refinement is required when some transistors have gate value `x`. A conceptually simple technique is to solve the network repeatedly with these transistors set to all possible combinations of fully conducting and nonconducting transistors. Any node that has a unique logic level in all cases has steady-state response equal to this level. All other nodes have steady-state response `x`.

### 3.9.6 Port connections

Ports connect processes through implicit continuous assignment statements or implicit bidirectional connections. Bidirectional connections are analogous to an always-enabled **`tran`** connection between the two nets, but without any strength reduction.

Ports can always be represented as declared objects connected, as follows:

- If an input port, then a continuous assignment from an outside expression to a local (input) net or variable
- If an output port, then a continuous assignment from a local output expression to an outside net or variable
- If an inout port, then a non-strength-reducing transistor connecting the local net to an outside net

Primitive terminals, including UDP terminals, are different from module ports. Primitive output and inout terminals shall be connected directly to 1-bit nets or 1-bit structural net expressions (see 23.3.3), with no intervening process that could alter the strength. Changes from primitive evaluations are scheduled as active update events in the connected nets. Input terminals connected to 1-bit nets or 1-bit structural net expressions are also connected directly, with no intervening process that could affect the strength. Input terminals connected to other kinds of expressions are represented as implicit continuous assignments from the expression to an implicit net that is connected to the input terminal.

### 4.9.7 Subroutines

Subroutine argument passing is by value, and it copies in on invocation and copies out on return. The copy-out-on-the-return function behaves in the same manner as does any blocking assignment.

## 4.10 PLI callback control points

There are two kinds of PLI callbacks: those that are executed immediately when some specific activity occurs, and those that are explicitly registered as a one-shot evaluation event.

Table 4-1 provides the mapping from the various PLI callbacks.

**Table 4-1—PLI callbacks**

| Callback | Event region |
|---|---|
| cbAfterDelay | Pre-Active |
| cbNextSimTime | Pre-Active |
| cbReadWriteSynch | Pre-NBA or Post-NBA |
| cbAtStartOfSimTime | Pre-Active |
| cbNBASynch | Pre-NBA |
| cbAtEndOfSimTime | Pre-Postponed |
| cbReadOnlySynch | Postponed |

# 5. Lexical conventions

## 5.1 General

This clause describes the following:

— Lexical tokens (white space, comments, operators)
— Integer, real, string, array, structure, and time literals
— Built-in method calls
— Attributes

## 5.2 Lexical tokens

SystemVerilog source text files shall be a stream of lexical tokens. A *lexical token* shall consist of one or more characters. The layout of tokens in a source file shall be free format; that is, spaces and newline characters shall not be syntactically significant other than being token separators, except for escaped identifiers (see 5.6.1).

The types of lexical tokens in the language are as follows:

— White space
— Comment
— Operator
— Number
— String literal
— Identifier
— Keyword

## 5.3 White space

White space shall contain the characters for spaces, tabs, newlines, and formfeeds. These characters shall be ignored except when they serve to separate other lexical tokens. However, blanks and tabs shall be considered significant characters in string literals (see 5.9).

## 5.4 Comments

SystemVerilog has two forms to introduce comments. A *one-line comment* shall start with the two characters `//` and end with a newline character. A *block comment* shall start with `/*` and end with `*/`. Block comments shall not be nested. The one-line comment token `//` shall not have any special meaning in a block comment.

## 5.5 Operators

Operators are single-, double-, or triple-character sequences and are used in expressions. Clause 11 discusses the use of operators in expressions.

*Unary operators* shall appear to the left of their operand. *Binary operators* shall appear between their operands. A *conditional operator* shall have two operator characters that separate three operands.

## 5.6 Identifiers, keywords, and system names

An *identifier* is used to give an object a unique name so it can be referenced. An identifier is either a *simple identifier* or an *escaped identifier* (see 5.6.1). A *simple identifier* shall be any sequence of letters, digits, dollar signs (`$`), and underscore characters (_).

The first character of a simple identifier shall not be a digit or `$`; it can be a letter or an underscore. Identifiers shall be case sensitive.

For example:

```
shiftreg_a
busa_index
error_condition
merge_ab
_bus3
n$657
```

Implementations may set a limit on the maximum length of identifiers, but the limit shall be at least 1024 characters. If an identifier exceeds the implementation-specific length limit, an error shall be reported.

### 5.6.1 Escaped identifiers

*Escaped identifiers* shall start with the backslash character (\) and end with white space (space, tab, newline). They provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

Neither the leading backslash character nor the terminating white space is considered to be part of the identifier. Therefore, an escaped identifier `\cpu3` is treated the same as a nonescaped identifier `cpu3`.

For example:

```
\busa+index
\-clock
\***error-condition***
\net1/\net2
\{a,b}
\a*(b+c)
```

### 5.6.2 Keywords

*Keywords* are predefined nonescaped identifiers that are used to define the language constructs. A SystemVerilog keyword preceded by an escape character is not interpreted as a keyword.

All keywords are defined in lowercase only. Annex B gives a list of all defined keywords. Subclause 22.14 discusses compatibility of reserved keywords with previous versions of IEEE Std 1364 and IEEE Std 1800.

### 5.6.3 System tasks and system functions

The dollar sign (**`$`**) introduces a language construct that enables development of user-defined system tasks and system functions. System constructs are not design semantics, but refer to simulator functionality. A name following the **`$`** is interpreted as a *system task* or a *system function*.

The syntax for system tasks and system functions is given in Syntax 5-1.

```
system_tf_call ::=                                                                // from A.8.2
      system_tf_identifier [ ( list_of_arguments ) ]
    | system_tf_identifier ( data_type [ , expression ] )
    | system_tf_identifier ( expression { , [ expression ] } [ , [ clocking_event ] ] )
system_tf_identifier[50] ::= $[ a-zA-Z0-9_$ ]{ [ a-zA-Z0-9_$ ] }                  // from A.9.3
```

50) The $ character in a system_tf_identifier shall not be followed by white_space. A system_tf_identifier shall not be escaped.

*Syntax 5-1—Syntax for system tasks and system functions (excerpt from Annex A)*

SystemVerilog defines a standard set of system tasks and system functions in this document (see Clause 20 and Clause 21). Additional *user-defined system tasks* and *system functions* can be defined using the PLI, as described in Clause 36. Software implementations can also specify additional system tasks and system functions, which may be tool-specific (see Annex D for some common additional system tasks and system functions). Additional system tasks and system functions are not part of this standard.

For example:

```
$display ("display a message");

$finish;
```

### 5.6.4 Compiler directives

The ` character (the ASCII value 0x60, called *grave accent*) introduces a language construct used to implement compiler directives. The compiler behavior dictated by a compiler directive shall take effect as soon as the compiler reads the directive. The directive shall remain in effect for the rest of the compilation unless a different compiler directive specifies otherwise. A compiler directive in one description file can, therefore, control compilation behavior in multiple description files. The effects of a compiler directive are limited to a compilation unit (see 3.12.1) and shall not affect other compilation units.

For example:

```
`define wordsize
```

SystemVerilog defines a standard set of compiler directives in this document (see Clause 22). Software implementations can also specify additional compiler directives, which may be tool-specific (see Annex E for some common additional compiler directives). Additional compiler directives are not part of this standard.

## 5.7 Numbers

*Constant numbers* can be specified as integer constants (see 5.7.1) or real constants (see 5.7.2). The formal syntax for numbers is listed in Syntax 5-2.

```
primary_literal ::= number | time_literal | unbased_unsized_literal | string_literal     // from A.8.4
time_literal[44] ::=
      unsigned_number time_unit
    | fixed_point_number time_unit
```

```
time_unit ::= s | ms | us | ns | ps | fs
number ::=                                                              // from A.8.7
      integral_number
    | real_number
integral_number ::=
      decimal_number
    | octal_number
    | binary_number
    | hex_number
decimal_number ::=
      unsigned_number
    | [ size ] decimal_base  unsigned_number
    | [ size ] decimal_base  x_digit { _ }
    | [ size ] decimal_base  z_digit { _ }
binary_number ::= [ size ] binary_base  binary_value
octal_number ::= [ size ] octal_base  octal_value
hex_number ::= [ size ] hex_base  hex_value
sign ::= + | -
size ::= non_zero_unsigned_number
non_zero_unsigned_number[33] ::= non_zero_decimal_digit { _ | decimal_digit}
real_number[33] ::=
      fixed_point_number
    | unsigned_number [ . unsigned_number ] exp [ sign ] unsigned_number
fixed_point_number[33] ::= unsigned_number . unsigned_number
exp ::= e | E
unsigned_number[33] ::= decimal_digit { _ | decimal_digit }
binary_value[33] ::= binary_digit { _ | binary_digit }
octal_value[33] ::= octal_digit { _ | octal_digit }
hex_value[33] ::= hex_digit { _ | hex_digit }
decimal_base[33] ::= '[s|S]d | '[s|S]D
binary_base[33] ::= '[s|S]b | '[s|S]B
octal_base[33] ::= '[s|S]o | '[s|S]O
hex_base[33] ::= '[s|S]h | '[s|S]H
non_zero_decimal_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
decimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
binary_digit ::= x_digit | z_digit | 0 | 1
octal_digit ::= x_digit | z_digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex_digit ::= x_digit | z_digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F
x_digit ::= x | X
z_digit ::= z | Z | ?
unbased_unsized_literal ::= '0 | '1 | 'z_or_x [48]
string_literal ::= " { Any_ASCII_Characters } "                      // from A.8.8
```

33) Embedded spaces are illegal.

44) The unsigned number or fixed-point number in time_literal shall not be followed by a white_space.

48) The apostrophe ( ' ) in unbased_unsized_literal shall not be followed by white_space.

*Syntax 5-2—Syntax for integer and real numbers (excerpt from Annex A)*

### 5.7.1 Integer literal constants

*Integer literal constants* can be specified in decimal, hexadecimal, octal, or binary format.

There are two forms to express integer literal constants. The first form is a simple decimal number, which shall be specified as a sequence of digits `0` through `9`, optionally starting with a plus or minus unary operator. The second form specifies a *based literal constant*, which shall be composed of up to three tokens—an optional size constant, an apostrophe character (`'`, ASCII 0x27) followed by a base format character, and the digits representing the value of the number. It shall be legal to macro-substitute these three tokens.

The first token, a size constant, shall specify the size of the integer literal constant in terms of its exact number of bits. It shall be specified as a nonzero unsigned decimal number. For example, the size specification for two hexadecimal digits is eight because one hexadecimal digit requires 4 bits.

The second token, a `base_format`, shall consist of a case insensitive letter specifying the base for the number, optionally preceded by the single character `s` (or `S`) to indicate a signed quantity, preceded by the apostrophe character. Legal base specifications are `d`, `D`, `h`, `H`, `o`, `O`, `b`, or `B` for the bases decimal, hexadecimal, octal, and binary, respectively.

The apostrophe character and the base format character shall not be separated by any white space.

The third token, an unsigned number, shall consist of digits that are legal for the specified base format. The unsigned number token shall immediately follow the base format, optionally preceded by white space. The hexadecimal digits `a` to `f` shall be case insensitive.

Simple decimal numbers without the size and the base format shall be treated as *signed integers*, whereas the numbers specified with the base format shall be treated as signed integers if the `s` designator is included or as *unsigned integers* if the base format only is used. The `s` designator does not affect the bit pattern specified, only its interpretation.

A plus or minus operator preceding the size constant is a unary plus or minus operator. A plus or minus operator between the base format and the number is an illegal syntax.

*Negative numbers* shall be represented in two's-complement form.

An `x` represents the *unknown value* in hexadecimal, octal, and binary literal constants. A `z` represents the *high-impedance value*. See 6.3 for a discussion of the SystemVerilog value set. An `x` shall set 4 bits to unknown in the hexadecimal base, 3 bits in the octal base, and 1 bit in the binary base. Similarly, a `z` shall set 4 bits, 3 bits, and 1 bit, respectively, to the high-impedance value.

If the size of the unsigned number is smaller than the size specified for the literal constant, the unsigned number shall be padded to the left with zeros. If the leftmost bit in the unsigned number is an `x` or a `z`, then an `x` or a `z` shall be used to pad to the left, respectively. If the size of the unsigned number is larger than the size specified for the literal constant, the unsigned number shall be truncated from the left.

The number of bits that make up an unsized number (which is a simple decimal number or a number with a base specifier but no size specification) shall be at least 32. Unsized unsigned literal constants where the high-order bit is unknown (`X` or `x`) or three-state (`Z` or `z`) shall be extended to the size of the expression containing the literal constant.

NOTE—In IEEE Std 1364-1995, in unsized literal constants where the high-order bit is unknown or three-state, the `x` or `z` was only extended to 32 bits.

An unsized single-bit value can be specified by preceding the single-bit value with an apostrophe ( ' ), but without the base specifier. All bits of the unsized value shall be set to the value of the specified bit. In a self-determined context, an unsized single-bit value shall have a width of 1 bit, and the value shall be treated as unsigned.

```
'0, '1, 'X, 'x, 'Z, 'z     // sets all bits to specified value
```

The use of `x` and `z` in defining the value of a number is case insensitive.

When used in a number, the question mark (`?`) character is a SystemVerilog alternative for the `z` character. It sets 4 bits to the high-impedance value in hexadecimal numbers, 3 bits in octal, and 1 bit in binary. The question mark can be used to enhance readability in cases where the high-impedance value is a do-not-care condition. See the discussion of **`casez`** and **`casex`** in 12.5.1. The question mark character is also used in UDP state tables. See Table 29-1 in 29.3.6.

In a decimal literal constant, the unsigned number token shall not contain any `x`, `z`, or `?` digits, unless there is exactly one digit in the token, indicating that every bit in the decimal literal constant is `x` or `z`.

The underscore character (_) shall be legal anywhere in a number except as the first character. The underscore character is ignored. This feature can be used to break up long numbers for readability purposes.

Several examples of specifying literal integer numbers are as follows:

*Example 1:* Unsized literal constant numbers

```
659         // is a decimal number
'h 837FF    // is a hexadecimal number
'o7460      // is an octal number
4af         // is illegal (hexadecimal format requires 'h)
```

*Example 2:* Sized literal constant numbers

```
4'b1001     // is a 4-bit binary number
5 'D 3      // is a 5-bit decimal number
3'b01x      // is a 3-bit number with the least
            // significant bit unknown
12'hx       // is a 12-bit unknown number
16'hz       // is a 16-bit high-impedance number
```

*Example 3:* Using sign with literal constant numbers

```
8 'd -6     // this is illegal syntax
-8 'd 6     // this defines the two's-complement of 6,
            // held in 8 bits—equivalent to -(8'd 6)
4 'shf      // this denotes the 4-bit number '1111', to
            // be interpreted as a two's-complement number,
            // or '-1'. This is equivalent to -4'h 1
-4 'sd15    // this is equivalent to -(-4'd 1), or '0001'
16'sd?      // the same as 16'sbz
```

*Example 4:* Automatic left padding of literal constant numbers

```
logic [11:0] a, b, c, d;
logic [84:0] e, f, g;
initial begin
        a = 'h x;      // yields xxx
```

```
         b = 'h 3x;      // yields 03x
         c = 'h z3;      // yields zz3
         d = 'h 0z3;     // yields 0z3

         e = 'h5;        // yields {82{1'b0},3'b101}
         f = 'hx;        // yields {85{1'hx}}
         g = 'hz;        // yields {85{1'hz}}
end
```

*Example 5:* Automatic left padding of constant literal numbers using a single-bit value

```
logic [15:0] a, b, c, d;
a = '0;     // sets all 16 bits to 0
b = '1;     // sets all 16 bits to 1
c = 'x;     // sets all 16 bits to x
d = 'z;     // sets all 16 bits to z
```

*Example 6:* Underscores in literal constant numbers

```
27_195_000                 // unsized decimal 27195000
16'b0011_0101_0001_1111    // 16-bit binary number
32 'h 12ab_f001            // 32-bit hexadecimal number
```

Sized negative literal constant numbers and sized signed literal constant numbers are sign-extended when assigned to a data object of type **`logic`**, regardless of whether the type itself is signed.

The default length of `x` and `z` is the same as the default length of an integer.

## 5.7.2 Real literal constants

The *real literal constant numbers* shall be represented as described by IEEE Std 754, an IEEE standard for double-precision floating-point numbers.

Real numbers can be specified in either decimal notation (for example, 14.72) or in scientific notation (for example, 39e8, which indicates 39 multiplied by 10 to the eighth power). Real numbers expressed with a decimal point shall have at least one digit on each side of the decimal point.

For example:

```
1.2
0.1
2394.26331
1.2E12  (the exponent symbol can be e or E)
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12  (underscores are ignored)
```

The following are invalid forms of real numbers because they do not have at least one digit on each side of the decimal point:

```
.12
9.
4.E3
.2e-7
```

The default type for fixed-point format (e.g., 1.2), and exponent format (e.g., 2.0e10) shall be **`real`**.

A cast can be used to convert literal real values to the **`shortreal`** type (e.g., **`shortreal`**`'(1.2)`). Casting is described in 6.24.

Real numbers shall be converted to integers by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion shall take place when a real number is assigned to an integer. The ties shall be rounded away from zero. For example:

— The real numbers 35.7 and 35.5 both become 36 when converted to an integer and 35.2 becomes 35.
— Converting –1.5 to integer yields –2, converting 1.5 to integer yields 2.

## 5.8 Time literals

Time is written in integer or fixed-point format, followed without a space by a time unit (**`fs ps ns us ms s`**). For example:

```
2.1ns
40ps
```

The time literal is interpreted as a **`realtime`** value scaled to the current time unit and rounded to the current time precision.

## 5.9 String literals

A *string literal* is a sequence of characters enclosed by double quotes (`""`).

Nonprinting and other special characters are preceded with a backslash.

A string literal shall be contained in a single line unless the newline character is immediately preceded by a **\** (backslash). In this case, the backslash and the newline character are ignored. There is no predefined limit to the length of a string literal.

*Example 1:*

```
$display("Humpty Dumpty sat on a wall. \
Humpty Dumpty had a great fall.");
```

prints

```
Humpty Dumpty sat on a wall. Humpty Dumpty had a great fall.
```

*Example 2:*

```
$display("Humpty Dumpty sat on a wall.\n\
Humpty Dumpty had a great fall.");
```

prints

```
Humpty Dumpty sat on a wall.
Humpty Dumpty had a great fall.
```

String literals used as operands in expressions and assignments shall be treated as unsigned integer constants represented by a sequence of 8-bit ASCII values, with one 8-bit ASCII value representing one character.

A string literal can be assigned to an integral type, such as a packed array. If the size differs, it is right justified. To fully store a string literal, the integral type should be declared with a width equal to the number of characters in the string multiplied by 8. For example:

```
byte c1 = "A" ;
bit [7:0] d = "\n" ;
```

The rules of SystemVerilog assignments shall be followed if the packed array width does not match the number of characters multiplied by 8. When an integral type is larger than required to hold the string literal value being assigned, the value is right-justified, and the leftmost bits are padded with zeros, as is done with nonstring values. If a string literal is larger than the destination integral type, the string is right-justified, and the leftmost characters are truncated.

For example, to store the 12-character string `"Hello world\n"` requires a variable 8 × 12, or 96 bits wide.

```
bit [8*12:1] stringvar = "Hello world\n";
```

Alternatively, a multidimensional packed array can be used, with 8-bit subfields, as in:

```
bit [0:11] [7:0] stringvar = "Hello world\n" ;
```

A string literal can be assigned to an unpacked array of bytes. If the size differs, it is left justified.

```
byte c3 [0:12] = "hello world\n" ;
```

Packed and unpacked arrays are discussed in 7.4.

String literals can also be cast to a packed or unpacked array type, which shall follow the same rules as assigning a string literal to a packed or unpacked array. Casting is discussed in 6.24.

SystemVerilog also includes a **`string`** data type to which a string literal can be assigned. Variables of type **`string`** have arbitrary length; they are dynamically resized to hold any string. String literals are packed arrays (of a width that is a multiple of 8 bits), and they are implicitly converted to the **`string`** type when assigned to a **`string`** type or used in an expression involving **`string`** type operands (see 6.16).

String literals stored in vectors can be manipulated using the SystemVerilog operators. The value being manipulated by the operator is the sequence of 8-bit ASCII values. See 11.10 for operations on string literals.

### 5.9.1 Special characters in strings

Certain characters can only be used in string literals when preceded by an introductory character called an *escape character*. Table 5-1 lists these characters in the right-hand column, with the escape sequence that represents the character in the left-hand column.

**Table 5-1—Specifying special characters in string literals**

| Escape string | Character produced by escape string |
|---|---|
| \n | Newline character |
| \t | Tab character |
| \\ | \ character |
| \" | " character |

**Table 5-1—Specifying special characters in string literals *(continued)***

| Escape string | Character produced by escape string |
|---|---|
| \v | vertical tab |
| \f | form feed |
| \a | bell |
| \ddd | A character specified in 1 to 3 octal_digits (see Syntax 5-2). If fewer than three characters are used, the following character shall not be an octal_digit. Implementations may issue an error if the character represented is greater than \377. It shall be illegal for an octal_digit in an escape sequence to be an x_digit or a z_digit (see Syntax 5-2). |
| \xdd | A character specified in 1 to 2 hex_digits (see Syntax 5-2). If only one digit is used, the following character shall not be a hex_digit. It shall be illegal for a hex_digit in an escape sequence to be an x_digit or a z_digit (see Syntax 5-2). |

## 5.10 Structure literals

Structure literals are structure assignment patterns or pattern expressions with constant member expressions (see 10.9.2). A structure literal must have a type, which may be either explicitly indicated with a prefix or implicitly indicated by an assignment-like context (see 10.8).

```
typedef struct {int a; shortreal b;} ab;
ab c;
c = '{0, 0.0}; // structure literal type determined from
               // the left-hand context (c)
```

Nested braces shall reflect the structure. For example:

```
ab abarr[1:0] = '{'{1, 1.0}, '{2, 2.0}};
```

The C-like alternative `'{1, 1.0, 2, 2.0}` for the preceding example is not allowed.

Structure literals can also use member name and value or use data type and default value (see 10.9.2):

```
c = '{a:0, b:0.0};               // member name and value for that member
c = '{default:0};                // all elements of structure c are set to 0
d = ab'{int:1, shortreal:1.0};   // data type and default value for all
                                 // members of that type
```

When an array of structures is initialized, the nested braces shall reflect the array and the structure. For example:

```
ab abarr[1:0] = '{'{1, 1.0}, '{2, 2.0}};
```

Replication operators can be used to set the values for the exact number of members. The inner pair of braces in a replication is removed.

```
struct {int X,Y,Z;} XYZ = '{3{1}};
typedef struct {int a,b[4];} ab_t;
int a,b,c;
ab_t v1[1:0] [2:0];
v1 = '{2{'{3{'{a,'{2{b,c}}}}}}};

/* expands to '{ '{3{ '{ a, '{2{ b, c }} } }},
                  '{3{ '{ a, '{2{ b, c }} } }}
                } */

/* expands to '{ '{ '{ a, '{2{ b, c }} },
                    '{ a, '{2{ b, c }} },
                    '{ a, '{2{ b, c }} }
                  },
                 '{ '{ a, '{2{ b, c }} },
                    '{ a, '{2{ b, c }} },
                    '{ a, '{2{ b, c }} }
                  }
                } */

/* expands to '{ '{ '{ a, '{ b, c, b, c } },
                    '{ a, '{ b, c, b, c } },
                    '{ a, '{ b, c, b, c } }
                  },
                 '{ '{ a, '{ b, c, b, c } },
                    '{ a, '{ b, c, b, c } },
                    '{ a, '{ b, c, b, c } }
                  }
                } */
```

## 5.11 Array literals

Array literals are syntactically similar to C initializers, but with the replication operator ( `{{}}` ) allowed.

```
int n[1:2][1:3] = '{'{0,1,2},'{3{4}}};
```

The nesting of braces shall follow the number of dimensions, unlike in C. However, replication operators can be nested. The inner pair of braces in a replication is removed. A replication expression only operates within one dimension.

```
int n[1:2][1:6] = '{2{'{3{4, 5}}}}; // same as
'{'{4,5,4,5,4,5},'{4,5,4,5,4,5}}
```

Array literals are array assignment patterns or pattern expressions with constant member expressions (see 10.9.1). An array literal must have a type, which may be either explicitly indicated with a prefix or implicitly indicated by an assignment-like context (see 10.8).

```
typedef int triple [1:3];
$mydisplay(triple'{0,1,2});
```

Array literals can also use their index or type as a key and use a default key value (see 10.9.1).

```
triple b = '{1:1, default:0};  // indices 2 and 3 assigned 0
```

## 5.12 Attributes

A mechanism is included for specifying properties about objects, statements, and groups of statements in the SystemVerilog source that can be used by various tools, including simulators, to control the operation or behavior of the tool. These properties are referred to as *attributes*. This subclause specifies the syntactic mechanism used for specifying attributes, without standardizing on any particular attributes.

The syntax for specifying an attribute is shown in Syntax 5-3.

```
attribute_instance ::= (* attr_spec { , attr_spec } *)                    // from A.9.1
attr_spec ::= attr_name [ = constant_expression ]
attr_name ::= identifier
```

*Syntax 5-3—Syntax for attributes (excerpt from Annex A)*

An attribute_instance can appear in the SystemVerilog description as a prefix attached to a declaration, a module item, a statement, or a port connection. It can appear as a suffix to an operator or a function name in an expression.

The default type of an attribute with no value is **bit**, with a value of 1. Otherwise, the attribute takes the type of the expression.

If the same attribute name is defined more than once for the same language element, the last attribute value shall be used, and a tool can issue a warning that a duplicate attribute specification has occurred.

Nesting of attribute instances is disallowed. It shall be illegal to specify the value of an attribute with a constant expression that contains an attribute instance.

Refer to Annex A for the syntax of specifying an attribute instance on specific language elements. Several examples are illustrated below.

*Example 1:* The following example shows how to attach attributes to a case statement:

```
(* full_case, parallel_case *)
case (a)
<rest_of_case_statement>
```

or

```
(* full_case=1 *)
(* parallel_case=1 *) // Multiple attribute instances also OK
case (a)
<rest_of_case_statement>
```

or

```
(* full_case, // no value assigned
   parallel_case=1 *)
case (a)
<rest_of_case_statement>
```

*Example 2:* To attach the `full_case` attribute, but not the `parallel_case` attribute:

```
(* full_case *) // parallel_case not specified
case (a)
<rest_of_case_statement>
```

or

```
(* full_case=1, parallel_case = 0 *)
case (a)
<rest_of_case_statement>
```

*Example 3:* To attach an attribute to a module definition:

```
(* optimize_power *)
module mod1 (<port_list>);
```

or

```
(* optimize_power=1 *)
module mod1 (<port_list>);
```

*Example 4:* To attach an attribute to a module instantiation:

```
(* optimize_power=0 *)
mod1 synth1 (<port_list>);
```

*Example 5:* To attach an attribute to a variable declaration:

```
(* fsm_state *) logic [7:0] state1;
(* fsm_state=1 *) logic [3:0] state2, state3;
logic [3:0] reg1;                  // reg1 does NOT have fsm_state set
(* fsm_state=0 *) logic [3:0] reg2; // nor does reg2
```

*Example 6:* To attach an attribute to an operator:

```
a = b + (* mode = "cla" *) c;    // sets the value for the attribute mode
                                 // to be the string cla.
```

*Example 7:* To attach an attribute to a function call:

```
a = add (* mode = "cla" *) (b, c);
```

*Example 8:* To attach an attribute to a conditional operator:

```
a = b ? (* no_glitch *) c : d;
```

## 5.13 Built-in methods

SystemVerilog uses a C++ -like class method calling syntax, in which a subroutine is called using the dot notation (`.`):

```
object.task_or_function()
```

The object uniquely identifies the data on which the subroutine operates. Hence, the method concept is naturally extended to built-in types in order to add functionality, which traditionally was done via system tasks or system functions. Unlike system tasks, built-in methods are not prefixed with a `$` because they require no special prefix to avoid collisions with user-defined identifiers. Thus, the method syntax allows extending the language without the addition of new keywords or the cluttering of the global name space with system tasks.

Built-in methods, unlike system tasks, cannot be redefined by users via PLI tasks. Thus, only functions that users should not be allowed to redefine are good candidates for built-in method calls.

In general, a built-in method is preferred over a system task when a particular functionality applies to all data types or when it applies to a specific data type. For example:

```
    dynamic_array.size, associative_array.num, and string.len
```

These are all similar concepts, but they represent different things. A dynamic array has a size, an associative array contains a given number of items, and a string has a given length. Using the same system task, such as `$size`, for all of them would be less clear and intuitive.

A built-in method can only be associated with a particular data type. Therefore, if some functionality is a simple side effect (i.e., `$stop` or `$reset`) or it operates on no specific data (i.e., `$random`), then a system task must be used.

When a subroutine built-in method call specifies no arguments, the empty parenthesis, `()`, following the subroutine name is optional. This is also true for subroutines that require arguments, when all arguments have defaults specified. For a method, this rule allows simple calls to appear as properties of the object or built-in type. Similar rules are defined for subroutines in 13.5.5.

# 6. Data types

## 6.1 General

This clause describes the following:

- SystemVerilog logic value and strength set
- Net declarations
- Singular variable declarations
- Constants
- Scope and lifetime of data
- Type compatibility
- Type operator and type casting

## 6.2 Data types and data objects

SystemVerilog makes a distinction between an object and its data type. A data type is a set of values and a set of operations that can be performed on those values. Data types can be used to declare data objects or to define user-defined data types that are constructed from other data types. A data object is a named entity that has a data value and a data type associated with it, such as a parameter, a variable, or a net.

## 6.3 Value set

### 6.3.1 Logic values

The SystemVerilog value set consists of the following four basic values:

**0**—represents a logic zero or a false condition
1—represents a logic one or a true condition
**x**—represents an unknown logic value
**z**—represents a high-impedance state

The values 0 and 1 are logical complements of one another.

When the z value is present at the input of a gate or when it is encountered in an expression, the effect is usually the same as an x value. Notable exceptions are the metal-oxide semiconductor (MOS) primitives, which can pass the z value.

The name of this primitive data type is **logic**. This name can be used to declare objects and to construct other data types from the 4-state data type.

Several SystemVerilog data types are 4-state types, which can store all four logic values. All bits of 4-state vectors can be independently set to one of the four basic values. Some SystemVerilog data types are 2-state, and only store 0 or 1 values in each bit of a vector. Other exceptions are the event type (see 6.17), which has no storage, and the real types (see 6.12).

### 6.3.2 Strengths

The language includes *strength* information in addition to the basic value information for nets. This is described in detail in Clause 28. The additional strength information associated with bits of a net is not considered part of the data type.

Two types of *strengths* can be specified in a net declaration, as follows:

— *Charge strength* shall only be used when declaring a net of type **trireg**.

— *Drive strength* shall only be used when placing a continuous assignment on a net in the same statement that declares the net.

Gate declarations can also specify a drive strength. See Clause 28 for more information on gates and for information on strengths.

#### 6.3.2.1 Charge strength

The charge strength specification shall be used only with trireg nets. A trireg net shall be used to model charge storage; charge strength shall specify the relative size of the capacitance indicated by one of the following keywords:

— **small**

— **medium**

— **large**

The default charge strength of a trireg net shall be **medium**.

A trireg net can model a charge storage node whose charge decays over time. The simulation time of a charge decay shall be specified in the delay specification for the trireg net (see 28.16.2).

For example:

```
trireg a;                         // trireg net of charge strength medium
trireg (large) #(0,0,50) cap1;    // trireg net of charge strength large
                                  // with charge decay time 50 time units
trireg (small) signed [3:0] cap2; // signed 4-bit trireg vector of
                                  // charge strength small
```

#### 6.3.2.2 Drive strength

The drive strength specification allows a continuous assignment to be placed on a net in the same statement that declares that net. See Clause 10 for more details. Net drive strength properties are described in detail in Clause 28.

## 6.4 Singular and aggregate types

Data types are categorized as either *singular* or *aggregate*. A singular type shall be any data type except an unpacked structure, unpacked union, or unpacked array (see 7.4 on arrays). An aggregate type shall be any unpacked structure, unpacked union, or unpacked array data type. A singular variable or expression represents a single value, symbol, or handle. Aggregate expressions and variables represent a set or collection of singular values. Integral types (see 6.11.1) are always singular even though they can be sliced into multiple singular values. The **string** data type is singular even though a string can be indexed in a similar way to an unpacked array of bytes.

These categories are defined so that operators and functions can simply refer to these data types as a collective group. For example, some functions recursively descend into an aggregate variable until reaching a singular value and then perform an operation on each singular value.

Although a class is a type, there are no variables or expressions of class type directly, only class object handles that are singular. Therefore, classes need not be categorized in this manner (see Clause 8 on classes).

## 6.5 Nets and variables

There are two main groups of data objects: variables and nets. These two groups differ in the way in which they are assigned and hold values.

A net can be written by one or more continuous assignments, by primitive outputs, or through module ports. The resultant value of multiple drivers is determined by the resolution function of the net type. A net cannot be procedurally assigned. If a net on one side of a port is driven by a variable on the other side, a continuous assignment is implied. A **force** statement can override the value of a net. When released, the net returns to the resolved value.

Variables can be written by one or more procedural statements, including procedural continuous assignments. The last write determines the value. Alternatively, variables can be written by one continuous assignment or one port.

Variables can be packed or unpacked aggregates of other types (see 7.4 for packed and unpacked types). Multiple assignments made to independent elements of a variable are examined individually. Independent elements include different members of a structure or different elements of an array. Each bit in a packed type is also an independent element. Thus, in an aggregate of packed types, each bit in the aggregate is an independent element.

An assignment where the left-hand side contains a slice is treated as a single assignment to the entire slice. Thus, a structure or array can have one element assigned procedurally and another element assigned continuously. And elements of a structure or array can be assigned with multiple continuous assignments, provided that each element is covered by no more than a single continuous assignment.

The precise rule is that it shall be an error to have multiple continuous assignments or a mixture of procedural and continuous assignments writing to any term in the expansion of a written longest static prefix of a variable (see 11.5.3 for the definition of a longest static prefix).

For example, assume the following structure declaration:

```
struct {
   bit [7:0] A;
   bit [7:0] B;
   byte C;
} abc;
```

The following statements are legal assignments to **struct** `abc`:

```
assign abc.C = sel ? 8'hBE : 8'hEF;

not   (abc.A[0],abc.B[0]),
      (abc.A[1],abc.B[1]),
      (abc.A[2],abc.B[2]),
      (abc.A[3],abc.B[3]);

always @(posedge clk) abc.B <= abc.B + 1;
```

The following additional statements are illegal assignments to **struct** `abc`:

```
// Multiple continuous assignments to abc.C
assign abc.C = sel ? 8'hDE : 8'hED;

// Mixing continuous and procedural assignments to abc.A[3]
always @(posedge clk) abc.A[7:3] <= !abc.B[7:3];
```

For the purposes of the preceding rule, a declared variable initialization or a procedural continuous assignment is considered a procedural assignment. The **force** statement overrides the procedural assign statement, which in turn overrides the normal assignments. A **force** statement is neither a continuous nor a procedural assignment.

A continuous assignment shall be implied when a variable is connected to an input port declaration. This makes assignments to a variable declared as an input port illegal. A continuous assignment shall be implied when a variable is connected to the output port of an instance. This makes additional procedural or continuous assignments to a variable connected to the output port of an instance illegal.

Variables cannot be connected to either side of an **inout** port. Variables can be shared across ports with the **ref** port type. See 23.3.3 for more information about ports and port connection rules.

The compiler can issue a warning if a continuous assignment could drive strengths other than `St0`, `St1`, `StX`, or `HiZ` to a variable. In any case, automatic type conversion shall be applied to the assignment, and the strength is lost.

Unlike nets, a variable cannot have an implicit continuous assignment as part of its declaration. An assignment as part of the declaration of a variable is a variable initialization, not a continuous assignment. For example:

```
wire w = vara & varb;              // net with a continuous assignment

logic v = consta & constb;         // variable with initialization

logic vw;                          // no initial assignment
assign vw = vara & varb;           // continuous assignment to a variable

real circ;
assign circ = 2.0 * PI * R;        // continuous assignment to a variable
```

Data shall be declared before they are used, apart from implicit nets (see 6.10).

Within a name space (see 3.13), it shall be illegal to redeclare a name already declared by a net, variable, or other declaration.

## 6.6 Net types

There are two different kinds of net types: built-in and user-defined. The *net* types can represent physical connections between structural entities, such as gates. A net shall not store a value (except for the **trireg** net). Instead, its value shall be determined by the values of its drivers, such as a continuous assignment or a gate. See Clause 10 and Clause 28 for definitions of these constructs. If no driver is connected to a net, its value shall be high-impedance (`z`) unless the net is a **trireg**, in which case it shall hold the previously driven value.

There are several distinct types of built-in net types, as shown in Table 6-1.

**Table 6-1—Built-in net types**

| | | | |
|---|---|---|---|
| **`wire`** | **`tri`** | **`tri0`** | **`supply0`** |
| **`wand`** | **`triand`** | **`tri1`** | **`supply1`** |
| **`wor`** | **`trior`** | **`trireg`** | **`uwire`** |

### 6.6.1 Wire and tri nets

The *wire* and *tri* nets connect elements. The net types **`wire`** and **`tri`** shall be identical in their syntax and functions; two names are provided so that the name of a net can indicate the purpose of the net in that model. A **`wire`** net can be used for nets that are driven by a single gate or continuous assignment. The **`tri`** net type can be used where multiple drivers drive a net.

Logical conflicts from multiple sources of the same strength on a **`wire`** or a **`tri`** net result in `x` (unknown) values.

Table 6-2 is a truth table for resolving multiple drivers on **`wire`** and **`tri`** nets. It assumes equal strengths for both drivers. See 28.11 for a discussion of logic strength modeling.

**Table 6-2—Truth table for wire and tri nets**

| **`wire/tri`** | **0** | **1** | **x** | **z** |
|---|---|---|---|---|
| **0** | 0 | x | x | 0 |
| **1** | x | 1 | x | 1 |
| **x** | x | x | x | x |
| **z** | 0 | 1 | x | z |

### 6.6.2 Unresolved nets

The **`uwire`** net is an unresolved or unidriver wire and is used to model nets that allow only a single driver. The **`uwire`** type can be used to enforce this restriction. It shall be an error to connect any bit of a **`uwire`** net to more than one driver. It shall be an error to connect a **`uwire`** net to a bidirectional terminal of a bidirectional pass switch.

The port connection rule in 23.3.3.6 enforces this restriction across the net hierarchy or shall issue a warning if not.

### 6.6.3 Wired nets

Wired nets are of type **`wor`**, **`wand`**, **`trior`**, and **`triand`** and are used to model wired logic configurations. Wired nets use different truth tables to resolve the conflicts that result when multiple drivers drive the same net. The **`wor`** and **`trior`** nets shall create *wired or* configurations so that when any of the drivers is 1, the resulting value of the net is 1. The **`wand`** and **`triand`** nets shall create *wired and* configurations so that if any driver is 0, the value of the net is 0.

The net types **`wor`** and **`trior`** shall be identical in their syntax and functionality. The net types **`wand`** and **`triand`** shall be identical in their syntax and functionality. Table 6-3 and Table 6-4 give the truth tables for wired nets, assuming equal strengths for both drivers. See 28.11 for a discussion of logic strength modeling.

**Table 6-3—Truth table for wand and triand nets**

| **`wand/triand`** | **0** | **1** | **x** | **z** |
|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 |
| **1** | 0 | 1 | x | 1 |
| **x** | 0 | x | x | x |
| **z** | 0 | 1 | x | z |

**Table 6-4—Truth table for wor and trior nets**

| **`wor/trior`** | **0** | **1** | **x** | **z** |
|---|---|---|---|---|
| **0** | 0 | 1 | x | 0 |
| **1** | 1 | 1 | 1 | 1 |
| **x** | x | 1 | x | x |
| **z** | 0 | 1 | x | z |

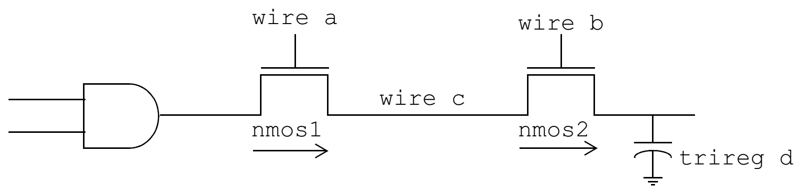
### 6.6.4 Trireg net

The *trireg* net stores a value and is used to model charge storage nodes. A trireg net can be in one of two states, as follows:

| | |
|---|---|
| *Driven state* | When at least one driver of a **`trireg`** net has a value of `1`, `0`, or `x`, the resolved value propagates into the **`trireg`** net and is the driven value of the **`trireg`** net. |
| *Capacitive state* | When all the drivers of a **`trireg`** net are at the high-impedance value (`z`), the **`trireg`** net retains its last driven value; the high-impedance value does not propagate from the driver to the **`trireg`**. |

The strength of the value on the **`trireg`** net in the capacitive state can be **`small`**, **`medium`**, or **`large`**, depending on the size specified in the declaration of the **`trireg`** net. The strength of a **`trireg`** net in the driven state can be **`supply`**, **`strong`**, **`pull`**, or **`weak`**, depending on the strength of the driver.

For example, Figure 6-1 shows a schematic that includes a **`trireg`** net whose size is **`medium`**, its driver, and the simulation results.



| simulation time | wire a | wire b | wire c | trireg d |
|---|---|---|---|---|
| 0 | 1 | 1 | strong 1 | strong 1 |
| 10 | 0 | 1 | HiZ | medium 1 |

**Figure 6-1—Simulation values of a trireg and its driver**

a) At simulation time 0, wire `a` and wire `b` have a value of `1`. A value of `1` with a **`strong`** strength propagates from the **`and`** gate through the **`nmos`** switches connected to each other by wire `c` into trireg net `d`.

b) At simulation time 10, wire `a` changes value to `0`, disconnecting wire `c` from the **`and`** gate. When wire `c` is no longer connected to the **`and`** gate, the value of wire `c` changes to `HiZ`. The value of wire `b` remains 1 so wire `c` remains connected to trireg net `d` through the `nmos2` switch. The `HiZ` value does not propagate from wire `c` into trireg net `d`. Instead, **`trireg`** net `d` enters the capacitive state, storing its last driven value of `1`. It stores the `1` with a **`medium`** strength.

### 6.6.4.1 Capacitive networks

A capacitive network is a connection between two or more trireg nets. In a capacitive network whose trireg nets are in the capacitive state, logic and strength values can propagate between trireg nets.

For example, Figure 6-2 shows a capacitive network in which the logic value of some trireg nets change the logic value of other trireg nets of equal or smaller size.

In Figure 6-2, the capacitive strength of `trireg_la` net is **`large`**, `trireg_me1` and `trireg_me2` are **`medium`**, and `trireg_sm` is **`small`**. Simulation reports the following sequence of events:

a) At simulation time 0, wire `a` and wire `b` have a value of `1`. The wire `c` drives a value of `1` into `trireg_la` and `trireg_sm`; wire `d` drives a value of `1` into `trireg_me1` and `trireg_me2`.

b) At simulation time 10, the value of wire `b` changes to `0`, disconnecting `trireg_sm` and `trireg_me2` from their drivers. These trireg nets enter the capacitive state and store the value `1`, their last driven value.

c) At simulation time 20, wire `c` drives a value of `0` into `trireg_la`.

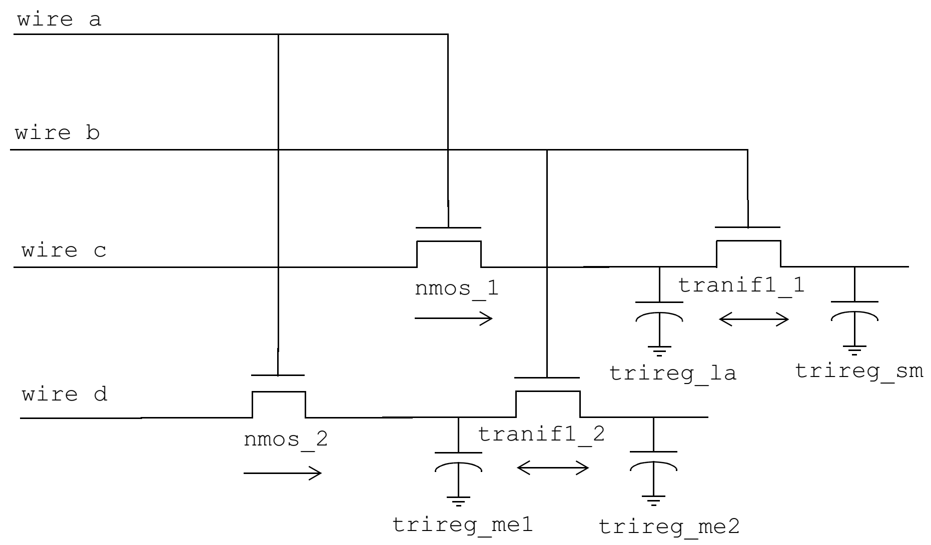
d) At simulation time 30, wire `d` drives a value of `0` into `trireg_me1`.

e) At simulation time 40, the value of wire `a` changes to `0`, disconnecting `trireg_la` and `trireg_me1` from their drivers. These trireg nets enter the capacitive state and store the value `0`.

f) At simulation time 50, the value of wire `b` changes to `1`.

g) This change of value in wire `b` connects `trireg_sm` to `trireg_la`; these trireg nets have different sizes and stored different values. This connection causes the smaller trireg net to store the value of the larger trireg net, and `trireg_sm` now stores a value of `0`.

   This change of value in wire `b` also connects `trireg_me1` to `trireg_me2`; these trireg nets have the same size and stored different values. The connection causes both `trireg_me1` and `trireg_me2` to change value to `x`.



| simulation time | wire a | wire b | wire c | wire d | trireg_la | trireg_sm | trireg_me1 | trireg_me2 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 1 | $\boxed{0}$ | 1 | 1 | 1 | 1 | 1 | 1 |
| 20 | 1 | 0 | $\boxed{0}$ | 1 | $\boxed{0}$ | 1 | 1 | 1 |
| 30 | 1 | 0 | 0 | $\boxed{0}$ | 0 | 1 | $\boxed{0}$ | 1 |
| 40 | $\boxed{0}$ | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 50 | 0 | $\boxed{1}$ | 0 | 0 | 0 | $\boxed{0}$ | $\boxed{x}$ | $\boxed{x}$ |

**Figure 6-2—Simulation results of a capacitive network**

In a capacitive network, charge strengths propagate from a larger trireg net to a smaller trireg net. Figure 6-3 shows a capacitive network and its simulation results.

In Figure 6-3, the capacitive strength of `trireg_la` is **`large`**, and the capacitive strength of `trireg_sm` is **`small`**. Simulation reports the following results:

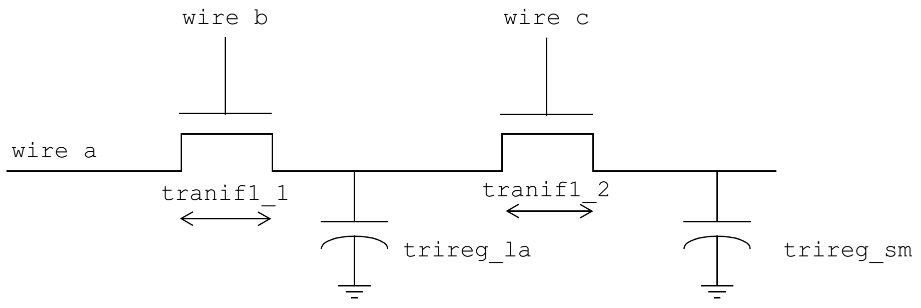
a) At simulation time 0, the values of wire `a`, wire `b`, and wire `c` are `1`, and wire `a` drives a **`strong`** `1` into `trireg_la` and `trireg_sm`.

b) At simulation time 10, the value of wire `b` changes to `0`, disconnecting `trireg_la` and `trireg_sm` from wire `a`. The `trireg_la` and `trireg_sm` nets enter the capacitive state. Both trireg nets share the **`large`** charge of `trireg_la` because they remain connected through `tranif1_2`.

c) At simulation time 20, the value of wire `c` changes to `0`, disconnecting `trireg_sm` from `trireg_la`. The `trireg_sm` no longer shares **`large`** charge of `trireg_la` and now stores a **`small`** charge.

d) At simulation time 30, the value of wire `c` changes to `1`, connecting the two trireg nets. These trireg nets now share the same charge.

e) At simulation time 40, the value of wire `c` changes again to `0`, disconnecting `trireg_sm` from `trireg_la`. Once again, `trireg_sm` no longer shares the **`large`** charge of `trireg_la` and now stores a **`small`** charge.



| simulation time | wire a | wire b | wire c | trireg_la | trireg_sm |
|---|---|---|---|---|---|
| 0 | strong 1 | 1 | 1 | strong 1 | strong 1 |
| 10 | strong 1 | 0 | 1 | large 1 | large 1 |
| 20 | strong 1 | 0 | 0 | large 1 | small 1 |
| 30 | strong 1 | 0 | 1 | large 1 | large 1 |
| 40 | strong 1 | 0 | 0 | large 1 | small 1 |

**Figure 6-3—Simulation results of charge sharing**

#### 6.6.4.2 Ideal capacitive state and charge decay

A **trireg** net can retain its value indefinitely, or its charge can decay over time. The simulation time of charge decay is specified in the delay specification of the **trireg** net. See 28.16.2 for charge decay explanation.

### 6.6.5 Tri0 and tri1 nets

The **tri0** and **tri1** nets model nets with resistive *pulldown* and resistive *pullup* devices on them. A **tri0** net is equivalent to a wire net with a continuous 0 value of **pull** strength driving it. A **tri1** net is equivalent to a wire net with a continuous 1 value of **pull** strength driving it.

When no driver drives a **tri0** net, its value is 0 with strength **pull**. When no driver drives a **tri1** net, its value is 1 with strength **pull**. When there are drivers on a **tri0** or **tri1** net, the drivers combine with the strength **pull** value implicitly driven on the net to determine the net's value. See 28.11 for a discussion of logic strength modeling.

Table 6-5 and Table 6-6 are truth tables for modeling multiple drivers of strength **strong** on **tri0** and **tri1** nets. The resulting value on the net has strength **strong**, unless both drivers are z, in which case the net has strength **pull**.

**Table 6-5—Truth table for tri0 net**

| **tri0** | **0** | **1** | **x** | **z** |
|---|---|---|---|---|
| **0** | 0 | x | x | 0 |
| **1** | x | 1 | x | 1 |
| **x** | x | x | x | x |
| **z** | 0 | 1 | x | 0 |

**Table 6-6—Truth table for tri1 net**

| **tri1** | **0** | **1** | **x** | **z** |
|---|---|---|---|---|
| **0** | 0 | x | x | 0 |
| **1** | x | 1 | x | 1 |
| **x** | x | x | x | x |
| **z** | 0 | 1 | x | 1 |

### 6.6.6 Supply nets

The **supply0** and **supply1** nets can be used to model the power supplies in a circuit. These nets shall have **supply** strengths.

### 6.6.7 User-defined nettypes

A user-defined **nettype** allows users to describe more general abstract values for a wire, including its resolution function. This **nettype** is similar to a **typedef** in some ways, but shall only be used in declaring a net. It provides a name for a particular data type and optionally an associated resolution function.

The syntax for net type declarations is given in Syntax 6-1.

```
net_type_declaration ::=                                                    // from A.2.1.3
    nettype data_type net_type_identifier
       [ with [ package_scope | class_scope ] tf_identifier ] ;
  | nettype [ package_scope | class_scope ] net_type_identifier net_type_identifier ;
```

*Syntax 6-1—Syntax for net type declarations (excerpt from Annex A)*

A net declared with a **nettype** therefore uses that data type and, if specified, the associated resolution function. An explicit data type is required for a user-defined **nettype**.

Certain restrictions apply to the data type of a net with a user-defined **nettype**. A valid data type shall be one of the following:

a) A 4-state integral type, including a packed array, packed structure or union.
b) A 2-state integral type, including a packed array, packed structure or union with 2-state data type members.
c) A **real** or **shortreal** type.
d) A fixed-size unpacked array, unpacked structure or union, where each element has a valid data type for a net of a user-defined **nettype**.

A second form of a **nettype** declaration is to create another name for an existing **nettype**.

An *atomic net* is a net whose value is updated and resolved as a whole. A net declared with a user-defined **nettype** is an atomic net. Similarly, a **logic** net is an atomic net, but a **logic** vector net is not an atomic net as each **logic** element is resolved and updated independently. While an atomic net may have a singular or aggregate value, each atomic net is intended to describe a single connection point in the design.

The resolution for a user-defined **nettype** is specified using a SystemVerilog function declaration. If a resolution function is specified, then when a driver of the net changes value, an update event is scheduled on the net in the Active (or Reactive) region. When the update event matures, the simulator calls the resolution function to compute the value of the net from the values of the drivers. The return type of the function shall match the data type of the **nettype**. The function shall accept an arbitrary number of drivers, since different instances of the net could be connected to different numbers of drivers. Any change in the value of one or more of the drivers shall trigger the evaluation of the resolution function associated with that **nettype**.

A user-defined *resolution function* for a net of a user-defined **nettype** with a data type `T` shall be a function with a return type of `T` and a single input argument whose type is a dynamic array of elements of type `T`. A resolution function shall be automatic (or preserve no state information) and have no side effects. A resolution function shall not resize the dynamic array input argument nor shall it write to any part of the dynamic array input argument. While a class function method may be used for a resolution function, such functions shall be class static methods as the method call occurs in a context where no class object is involved in the call. Parameterized variants of such methods can be created through the use of parameterized class methods as described in 13.8.

Two different nettypes can use the same data type, but have different resolution functions A **nettype** may be declared without a resolution function, in which case it shall be an error for a net of that **nettype** to have multiple drivers.

Due to nondeterminism within scheduling regions, if there are multiple driver updates within a scheduling region, there may be multiple evaluations of the resolution function.

A **force** statement can override the value of a net of a user-defined **nettype**. When released, the net returns to the resolved value.

```
// user-defined data type T
typedef struct {
   real field1;
   bit field2;
} T;

// user-defined resolution function Tsum
function automatic T Tsum (input T driver[]);
   Tsum.field1 = 0.0;
   foreach (driver[i])
      Tsum.field1 += driver[i].field1;
```

```
endfunction

nettype T wT;   // an unresolved nettype wT whose data type is T

// a nettype wTsum whose data type is T and
// resolution function is Tsum
nettype T wTsum with Tsum;

// user-defined data type TR
typedef real TR[5];

// an unresolved nettype wTR whose data type
// is an array of real
nettype TR wTR;

// declare another name nettypeid2 for nettype wTsum
nettype wTsum nettypeid2;
```

The following example shows how to use a combination of a parameterized class definition with class static methods to parameterize the data type of a user-defined **nettype**.

```
class Base #(parameter p = 1);
   typedef struct {
      real r;
      bit[p-1:0] data;
   } T;

   static function T Tsum (input T driver[]);
      Tsum.r = 0.0;
      Tsum.data = 0;
      foreach (driver[i])
         Tsum.data += driver[i].data;
      Tsum.r = $itor(Tsum.data);
   endfunction
endclass

typedef Base#(32) MyBaseT;
nettype MyBaseT::T narrowTsum with MyBaseT::Tsum;

typedef Base#(64) MyBaseType;
nettype MyBaseType::T wideTsum with MyBaseType::Tsum;

narrowTsum net1; // data is 32 bits wide
wideTsum net2; // data is 64 bits wide
```

### 6.6.8 Generic interconnect

In SystemVerilog it is possible to use net types and configurations to create design models with varying levels of abstraction. In order to support netlist designs, which primarily specify design element instances and the net connections between the design elements, SystemVerilog defines a generic form of nets. Such generic nets allow the separation of the specification of the net connections from the types of the connections.

A net or port declared as **interconnect** (an **interconnect** net or port) indicates a typeless or generic net. Such nets or ports are only able to express net port and terminal connections and shall not be used in any procedural context nor in any continuous or procedural continuous assignments. An **interconnect** net or port shall not be used in any expression other than a net_lvalue expression in which all nets or ports in the expression are also **interconnect** nets. An **interconnect** array shall be considered valid even if different bits in the array are resolved to different net types as demonstrated in the following example. It shall be legal to specify a net_alias statement with an **interconnect** net_lvalue. See 23.3.3.7.1 and 23.3.3.7.2 for port and terminal connection rules for **interconnect** nets.

```
package NetsPkg;
   nettype real realNet;
endpackage : NetsPkg

module top();
   interconnect [0:1] iBus;
   lDriver l1(iBus[0]);
   rDriver r1(iBus[1]);
   rlMod m1(iBus);
endmodule : top

module lDriver(output wire logic out);
endmodule : lDriver

module rDriver
   import NetsPkg::*;
   (output realNet out);
endmodule : rDriver

module rlMod(input interconnect [0:1] iBus);
   lMod l1(iBus[0]);
   rMod r1(iBus[1]);
endmodule : rlMod
```

The following simple example serves to illustrate the usefulness of an **interconnect** net. The example contains a top level module (`top`) that instantiates a stimulus module (`driver`) and a comparator module (`cmp`). This configuration is intended to compare two elements and determine if they are equal. There are two different versions of the configuration, as described by the two different **config** blocks: one that works on **real** values and one that works on **logic** values. By using the typeless **interconnect** net, we can use the same testbench with both configurations, without having to change anything in the testbench itself. The **interconnect** net `aBus` takes its data type from the type of its connections.

```
<file lib.map>
library realLib *.svr;
library logicLib *.sv;

config cfgReal;
   design logicLib.top;
   default liblist realLib logicLib;
endconfig

config cfgLogic;
   design logicLib.top;
   default liblist logicLib realLib;
endconfig

<file top.sv>
module top();
   interconnect [0:3] [0:1] aBus;
   logic [0:3] dBus;
   driver driverArray[0:3](aBus);
   cmp cmpArray[0:3](aBus,rst,dBus);
endmodule : top

<file nets.pkg>
package NetsPkg;
```

```
   nettype real realNet;
endpackage : NetsPkg

<file driver.svr>
module driver
   import NetsPkg::*;
   #(parameter int delay = 30,
               int iterations = 256)
   (output realNet [0:1] out);
   timeunit 1ns / 1ps;
   real outR[1:0];
   assign out = outR;
   initial begin
      outR[0] = 0.0;
      outR[1] = 3.3;
      for (int i = 0; i < iterations; i++) begin
         #delay outR[0] += 0.2;
         outR[1] -= 0.2;
      end
   end
endmodule : driver

<file driver.sv>
module driver #(parameter int delay = 30,
                          int iterations = 256)
               (output wire logic [0:1] out);
   timeunit 1ns / 1ps;
   logic [0:1] outvar;

   assign out = outvar;

   initial begin
      outvar = '0;
      for (int i = 0; i < iterations; i++)
         #delay outvar++;
   end
endmodule : driver

<file cmp.svr>
module cmp
   import NetsPkg::*;
   #(parameter real hyst = 0.65)
   (input realNet [0:1] inA,
    input  logic rst,
    output logic out);
   timeunit 1ns / 1ps;
   real updatePeriod = 100.0;

   initial out = 1'b0;

   always #updatePeriod begin
      if (rst) out <= 1'b0;
      else if (inA[0] > inA[1]) out <= 1'b1;
      else if (inA[0] < inA[1] - hyst) out <= 1'b0;
   end
endmodule : cmp

<file cmp.sv>
module cmp #(parameter real hyst = 0.65)
```

```
            (input wire logic [0:1] inA,
      input  logic rst,
      output logic out);

      initial out = 1'b0;

      always @(inA, rst) begin
         if (rst) out <= 1'b0;
         else if (inA[0] & ~inA[1]) out <= 1'b1;
         else out <= 1'b0;
      end
   endmodule : cmp
```

## 6.7 Net declarations

The syntax for net declarations is given in Syntax 6-2.

```
net_declaration12 ::=                                                         // from A.2.1.3
     net_type [ drive_strength | charge_strength ] [ vectored | scalared ]
        data_type_or_implicit [ delay3 ] list_of_net_decl_assignments ;
   | net_type_identifier [ delay_control ]
        list_of_net_decl_assignments ;
   | interconnect implicit_data_type [ # delay_value ]
        net_identifier { unpacked_dimension }
        [ , net_identifier { unpacked_dimension }] ;
net_type ::=                                                                  // from A.2.2.1
     supply0 | supply1 | tri | triand | trior | trireg | tri0 | tri1 | uwire | wire | wand | wor
drive_strength ::=                                                            // from A.2.2.2
     ( strength0 , strength1 )
   | ( strength1 , strength0 )
   | ( strength0 , highz1 )
   | ( strength1 , highz0 )
   | ( highz0 , strength1 )
   | ( highz1 , strength0 )
strength0 ::= supply0 | strong0 | pull0 | weak0
strength1 ::= supply1 | strong1 | pull1 | weak1
charge_strength ::= ( small ) | ( medium ) | ( large )
delay3 ::=                                                                    // from A.2.2.3
     # delay_value | # ( mintypmax_expression [ , mintypmax_expression [ , mintypmax_expression ] ] )
delay2 ::= # delay_value | # ( mintypmax_expression [ , mintypmax_expression ] )
delay_value ::=
     unsigned_number
   | real_number
   | ps_identifier
   | time_literal
   | 1step
list_of_net_decl_assignments ::= net_decl_assignment { , net_decl_assignment }    // from A.2.3
net_decl_assignment ::= net_identifier { unpacked_dimension } [ = expression ]     // from A.2.4
```

12) A charge strength shall only be used with the **trireg** keyword. When the **vectored** or **scalared** keyword is used, there shall be at least one packed dimension.

*Syntax 6-2—Syntax for net declarations (excerpt from Annex A)*

### 6.7.1 Net declarations with built-in net types

Net declarations without assignments and whose **nettype** is not a user-defined **nettype** are described in this subclause. Net declarations with assignments are described in Clause 10.

A net declaration begins with a net type that determines how the values of the nets in the declaration are resolved. The declaration can include optional information such as delay values, drive or charge strength, and a data type.

If a set of nets share the same characteristics, they can be declared in the same declaration statement.

Any 4-state data type can be used to declare a net. For example:

```
trireg (large) logic #(0,0,0) cap1;
typedef logic [31:0] addressT;
wire addressT w1;
wire struct packed { logic ecc; logic [7:0] data; } memsig;
```

If a data type is not specified in the net declaration or if only a range and/or signing is specified, then the data type of the net is implicitly declared as **logic**. For example:

```
wire w;           // equivalent to "wire logic w;"
wire [15:0] ww;   // equivalent to "wire logic [15:0] ww;"
```

A net declared as an **interconnect** net shall:

— have no data type but may have optional packed or unpacked dimensions;
— not specify drive_strength or charge_strength;
— not have assignment expressions;
— specify at most one delay value.

Certain restrictions apply to the data type of a net. A valid data type for a net shall be one of the following:

a) A 4-state integral type, including a packed array or packed structure.
b) A fixed-size unpacked array or unpacked structure, where each element has a valid data type for a net.

The effect of this recursive definition is that a net is composed entirely of 4-state bits and is treated accordingly. In addition to a signal value, each bit of a net shall have additional strength information. When bits of signals combine, the strength and value of the resulting signal shall be determined as described in 28.12.

A lexical restriction applies to the use of the **reg** keyword in a net or port declaration. A net type keyword shall not be followed directly by the **reg** keyword. Thus, the following declarations are in error:

```
tri reg r;
inout wire reg p;
```

The **reg** keyword can be used in a net or port declaration if there are lexical elements between the net type keyword and the **reg** keyword.

The default initialization value for a net shall be the value z. Nets with drivers shall assume the output value of their drivers. The **trireg** net is an exception. The **trireg** net shall default to the value x, with the strength specified in the net declaration (**small**, **medium**, or **large**).

As described in 6.6.8, an **interconnect** net is restricted in terms of its declaration and use. The following are some examples of legal and illegal **interconnect** net declarations:

```
interconnect w1;                // legal
interconnect [3:0] w2;          // legal
interconnect [3:0] w3 [1:0];    // legal
interconnect logic [3:0] w4;    // illegal – data type specified
interconnect #(1,2,3) w5;       // illegal – only one delay permitted
assign w1 = 1;                  // illegal – not allowed in a
                                // continuous assign
initial $display(w1);           // illegal – not allowed in a
                                // procedural context
```

### 6.7.2 Net declarations with user-defined nettypes

A net with a user-defined **nettype** allows users to describe more general abstract values for a wire. A net declared with a **nettype** uses the data type and any associated resolution function for that **nettype**.

```
// an unresolved nettype wT whose data type is T
// Refer to example in 6.6.7 for declaration of the data type T
nettype T wT;

// a nettype wTsum whose data type is T and
// resolution function is Tsum
// Refer to example in 6.6.7 for the declaration of Tsum
nettype T wTsum with Tsum;

// a net of unresolved nettype wT
wT w1;

// an array of nets, each net element is of unresolved nettype wT
wT w2[8];

// a net of resolved nettype wTsum and resolution function Tsum
wTsum w3;

// an array of nets, each net is of resolved nettype wTsum
wTsum w4[8];

// user-defined data type TR which is an array of reals
typedef real TR[5];

// an unresolved nettype wTR with data type TR
nettype TR wTR;

// a net with unresolved nettype wTR and data type TR
wTR w5;

// an array of nets, each net has an unresolved nettype wTR
// and data type TR
wTR w6[8];
```

### 6.7.3 Initialization of nets with user-defined nettypes

The resolution function for any net of a user-defined **nettype** shall be activated at time zero at least once. This activation occurs even for such nets with no drivers or no value changes on drivers at time zero. Since the actual evaluation of the resolution function is subject to scheduling nondeterminism, no assumptions can be made regarding the state of driven values during the guaranteed call, which may precede or follow any driver changes at time zero.

The initial value of a net with a user-defined **nettype** shall be set before any initial or always procedures are started and before the activation of the guaranteed time zero resolution call. The default initialization value for a net with a user-defined **nettype** shall be the default value defined by the data type. Table 6-7 defines the default value for data types of variables if no initializer is provided; those default values shall also apply to nets of user-defined nettypes for valid data types of a net. For a net with a user-defined **nettype** whose data type is a **struct** type, any initialization expressions for the members within the **struct** shall be applied.

NOTE— The default value for a **logic** net of a user-defined **nettype** is X. This default means that a bit of a **logic** data type in an unresolved user-defined **nettype** will be X if it has no drivers, not Z. For a net with a resolved **nettype**, the value would be determined by the resolution function executed with an empty array of driver values.

## 6.8 Variable declarations

A *variable* is an abstraction of a data storage element. A variable shall store a value from one assignment to the next. An assignment statement in a procedure acts as a trigger that changes the value in the data storage element.

The syntax for variable declarations is given in Syntax 6-3.

```
data_declaration ::=                                                        // from A.2.1.3
    [ const ] [ var ] [ lifetime ] data_type_or_implicit list_of_variable_decl_assignments ;[10]
  | type_declaration
  ...
data_type ::=                                                               // from A.2.2.1
    integer_vector_type [ signing ] { packed_dimension }
  | integer_atom_type [ signing ]
  | non_integer_type
  | struct_union [ packed [ signing ] ] { struct_union_member { struct_union_member } }
       { packed_dimension }[13]
  | enum [ enum_base_type ] { enum_name_declaration { , enum_name_declaration } }
       { packed_dimension }
  | string
  | chandle
  | virtual [ interface ] interface_identifier [ parameter_value_assignment ] [ . modport_identifier ]
  | [ class_scope | package_scope ] type_identifier { packed_dimension }
  | class_type
  | event
  | ps_covergroup_identifier
  | type_reference[14]
integer_type ::= integer_vector_type | integer_atom_type
integer_atom_type ::= byte | shortint | int | longint | integer | time
integer_vector_type ::= bit | logic | reg
non_integer_type ::= shortreal | real | realtime
signing ::= signed | unsigned
simple_type ::= integer_type | non_integer_type | ps_type_identifier | ps_parameter_identifier
data_type_or_void ::= data_type | void
```

```
variable_decl_assignment ::=                                                    // from A.2.4
      variable_identifier { variable_dimension } [ = expression ]
    | dynamic_array_variable_identifier unsized_dimension { variable_dimension }
          [ = dynamic_array_new ]
    | class_variable_identifier [ = class_new ]
```

10) In a data_declaration that is not within a procedural context, it shall be illegal to use the **automatic** keyword. In a data_declaration, it shall be illegal to omit the explicit data_type before a list_of_variable_decl_assignments unless the **var** keyword is used.

13) When a packed dimension is used with the **struct** or **union** keyword, the **packed** keyword shall also be used.

14) When a type_reference is used in a net declaration, it shall be preceded by a net type keyword; and when it is used in a variable declaration, it shall be preceded by the **var** keyword.

*Syntax 6-3—Syntax for variable declarations (excerpt from Annex A)*

One form of variable declaration consists of a data type followed by one or more instances.

```
shortint s1, s2[0:9];
```

Another form of variable declaration begins with the keyword **var**. The data type is optional in this case. If a data type is not specified or if only a range and/or signing is specified, then the data type is implicitly declared as **logic**.

```
var byte my_byte;    // equivalent to "byte my_byte;"
var v;               // equivalent to "var logic v;"
var [15:0] vw;       // equivalent to "var logic [15:0] vw;"
var enum bit { clear, error } status;
input var logic data_in;
var reg r;
```

If a set of variables share the same characteristics, they can be declared in the same declaration statement.

A variable can be declared with an initializer, for example:

```
int i = 0;
```

Setting the initial value of a static variable as part of the variable declaration (including static class members) shall occur before any initial or always procedures are started (also see 6.21 and 10.5 on variable initialization with static and automatic lifetimes).

NOTE—In IEEE Std 1364-2005, an initialization value specified as part of the declaration was executed as if the assignment were made from an initial procedure, after simulation has started.

Initial values are not constrained to simple constants; they can include run-time expressions, including dynamic memory allocation. For example, a static class handle or a mailbox can be created and initialized by calling its **new** method (see 15.4.1), or static variables can be initialized to random values by calling the `$urandom` system task. This may require a special pre-initial pass at run time.

Table 6-7 contains the default values for variables if no initializer is specified.

**Table 6-7—Default variable initial values**

| Type | Default initial value |
|---|---|
| 4-state integral | `'X` |
| 2-state integral | `'0` |
| **`real`**, **`shortreal`** | `0.0` |
| Enumeration | Base type default initial value |
| **`string`** | `""` (empty string) |
| **`event`** | New event |
| **`class`** | **`null`** |
| **`interface class`** | **`null`** |
| **`chandle`** (Opaque handle) | **`null`** |
| **`virtual interface`** | **`null`** |

Nets and variables can be assigned negative values, but only signed types shall retain the significance of the sign. The **`byte`**, **`shortint`**, **`int`**, **`integer`**, and **`longint`** types are signed types by default. Other net and variable types can be explicitly declared as signed. See 11.4.3.1 for a description of how signed and unsigned nets and variables are treated by certain operators.

## 6.9 Vector declarations

A data object declared as **`reg`**, **`logic`**, or **`bit`** (or as a matching user-defined type or implicitly as **`logic`**) without a range specification shall be considered 1-bit wide and is known as a *scalar*. A multibit data object of one of these types shall be declared by specifying a range and is known as a *vector*. Vectors are packed arrays of scalars (see 7.4).

### 6.9.1 Specifying vectors

The range specification ([msb_constant_expression : lsb_constant_expression]) gives addresses to the individual bits in a multibit **`reg`**, **`logic`,** or **`bit`** vector. The most significant bit, specified by the *msb* constant expression, is the left-hand value in the range, and the least significant bit, specified by the *lsb* constant expression, is the right-hand value in the range.

Both the msb constant expression and the lsb constant expression shall be constant integer expressions. The msb and lsb constant expressions (see 11.2.1) may be any integer value—positive, negative, or zero. It shall be illegal for them to contain any unknown (x) or high-impedance bits. The lsb value may be greater than, equal to, or less than the msb value.

Vectors shall obey laws of arithmetic modulo-2 to the power $n$ ($2^n$), where $n$ is the number of bits in the vector. Vectors of **`reg`**, **`logic`,** and **`bit`** types shall be treated as unsigned quantities, unless declared to be signed or connected to a port that is declared to be signed (see 23.2.2.1 and 23.3.3.8).

*Examples:*

```
wand w;                           // a scalar "wand" net
```

```
tri [15:0] busa;            // a 16-bit bus
trireg (small) storeit;     // a charge storage node of strength small
logic a;                    // a scalar variable
logic[3:0] v;               // a 4-bit vector made up of (from most to
                            // least significant)v[3], v[2], v[1], and v[0]
logic signed [3:0] signed_reg; // a 4-bit vector in range -8 to 7
logic [-1:4] b;             // a 6-bit vector
wire w1, w2;                // declares two nets
logic [4:0] x, y, z;        // declares three 5-bit variables
```

Implementations may set a limit on the maximum length of a vector, but the limit shall be at least 65536 ($2^{16}$) bits.

Implementations are not required to detect overflow of integer operations.

### 6.9.2 Vector net accessibility

*Vectored* and *scalared* shall be optional advisory keywords to be used in vector net declarations. If these keywords are implemented, certain operations on vector nets may be restricted. If the keyword **vectored** is used, bit-selects and part-selects and strength specifications may not be permitted, and the PLI may consider the net *unexpanded*. If the keyword **scalared** is used, bit-selects and part-selects of the net shall be permitted, and the PLI shall consider the net *expanded*.

For example:

```
tri1 scalared [63:0] bus64;   //a bus that will be expanded
tri vectored [31:0] data;     //a bus that may or may not be expanded
```

## 6.10 Implicit declarations

The syntax shown in 6.7 and 6.8 shall be used to declare nets and variables explicitly. In the absence of an explicit declaration, an implicit net of default net type shall be assumed in the following circumstances:

— If an identifier is used in a port expression declaration, then an implicit net of default net type shall be assumed, with the vector width of the port expression declaration. See 23.2.2.1 for a discussion of port expression declarations.

— If an identifier is used in the terminal list of a primitive instance or in the port connection list of a module, interface, program, or static checker instance (but not a procedural checker instance, see 17.3), and that identifier has not been declared previously in the scope where the instantiation appears or in any scope whose declarations can be directly referenced from the scope where the instantiation appears (see 23.9), then an implicit scalar net of default net type shall be assumed.

— If an identifier appears on the left-hand side of a continuous assignment statement, and that identifier has not been declared previously in the scope where the continuous assignment statement appears or in any scope whose declarations can be directly referenced from the scope where the continuous assignment statement appears (see 23.9), then an implicit scalar net of default net type shall be assumed. See 10.3 for a discussion of continuous assignment statements.

The implicit net declaration shall belong to the scope in which the net reference appears. For example, if the implicit net is declared by a reference in a generate block, then the net is implicitly declared only in that generate block. Subsequent references to the net from outside the generate block or in another generate block within the same module either would be illegal or would create another implicit declaration of a different net (depending on whether the reference meets the preceding criteria). See Clause 27 for information about generate blocks.

See 22.8 for a discussion of control of the type for implicitly declared nets with the `` `default_nettype `` compiler directive.

## 6.11 Integer data types

SystemVerilog provides several integer data types, as shown in Table 6-8.

**Table 6-8—Integer data types**

| | |
|---|---|
| **shortint** | 2-state data type, 16-bit signed integer |
| **int** | 2-state data type, 32-bit signed integer |
| **longint** | 2-state data type, 64-bit signed integer |
| **byte** | 2-state data type, 8-bit signed integer or ASCII character |
| **bit** | 2-state data type, user-defined vector size, unsigned |
| **logic** | 4-state data type, user-defined vector size, unsigned |
| **reg** | 4-state data type, user-defined vector size, unsigned |
| **integer** | 4-state data type, 32-bit signed integer |
| **time** | 4-state data type, 64-bit unsigned integer |

### 6.11.1 Integral types

The term *integral* is used throughout this standard to refer to the data types that can represent a single basic integer data type, packed array, packed structure, packed union, enum variable, or time variable.

The term *simple bit vector type* is used throughout this standard to refer to the data types that can directly represent a one-dimensional packed array of bits. The integer types listed in Table 6-8 are simple bit vector types with predefined widths. The packed structure types (see 7.2) and multidimensional packed array types (see 7.4) are not simple bit vector types, but each is equivalent (see 6.22.2) to some simple bit vector type, to and from which it can be easily converted.

### 6.11.2 2-state (two-value) and 4-state (four-value) data types

Types that can have unknown and high-impedance values are called *4-state types*. These are **logic**, **reg**, **integer**, and **time**. The other types do not have unknown values and are called *2-state types*, for example, **bit** and **int**.

The difference between **int** and **integer** is that **int** is a 2-state type and **integer** is a 4-state type. The 4-state values have additional bits, which encode the X and Z states. The 2-state data types can simulate faster, take less memory, and are preferred in some design styles.

The keyword **reg** does not always accurately describe user intent, as it could be perceived to imply a hardware register. The keyword **logic** is a more descriptive term. **logic** and **reg** denote the same type.

Automatic type conversions from a smaller number of bits to a larger number of bits involve zero extensions if unsigned or sign extensions if signed. Automatic type conversions from a larger number of bits to a smaller number of bits involve truncations of the most significant bits (MSBs). When a 4-state value is automatically converted to a 2-state value, any unknown or high-impedance bits shall be converted to zeros.

### 6.11.3 Signed and unsigned integer types

Integer types use integer arithmetic and can be signed or unsigned. This affects the meaning of certain operators (see Clause 11 on operators and expressions).

The data types **byte**, **shortint**, **int**, **integer**, and **longint** default to signed. The data types **time**, **bit**, **reg**, and **logic** default to unsigned, as do arrays of these types. The signedness can be explicitly defined using the keywords **signed** and **unsigned**.

```
int unsigned ui;
int signed si;
```

## 6.12 Real, shortreal, and realtime data types

The **real**[12] data type is the same as a C double. The **shortreal** data type is the same as a C float. The **realtime** declarations shall be treated synonymously with **real** declarations and can be used interchangeably. Variables of these three types are collectively referred to as *real variables*.

### 6.12.1 Operators and real numbers

The result of using logical or relational operators on real numbers and real variables is a single-bit scalar value. Not all operators can be used with expressions involving real numbers and real variables (see 11.3.1). Real number constants and real variables are also prohibited in the following cases:

- Edge event controls (**posedge**, **negedge**, **edge**) applied to real variables
- Bit-select or part-select references of variables declared as **real**
- Real number index expressions of bit-select or part-select references of vectors

### 6.12.2 Conversion

Real numbers shall be converted to integers by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion shall take place when a real number is assigned to an integer. If the fractional part of the real number is exactly 0.5, it shall be rounded away from zero.

Implicit conversion shall also take place when an expression is assigned to a real. Individual bits that are x or z in the net or the variable shall be treated as zero upon conversion.

Explicit conversion can be specified using casting (see 6.24) or using system tasks (see 20.5).

## 6.13 Void data type

The **void** data type represents nonexistent data. This type can be specified as the return type of functions to indicate no return value. This type can also be used for members of tagged unions (see 7.3.2).

## 6.14 Chandle data type

The **chandle** data type represents storage for pointers passed using the DPI (see Clause 35). The size of a value of this data type is platform dependent, but shall be at least large enough to hold a pointer on the machine on which the tool is running.

The syntax to declare a handle is as follows:

[12]The **real** and **shortreal** types are represented as described by IEEE Std 754.

```
    chandle variable_name ;
```

where `variable_name` is a valid identifier. Chandles shall always be initialized to the value **`null`**, which has a value of 0 on the C side. Chandles are restricted in their usage, with the only legal uses being as follows:

— Only the following operators are valid on **`chandle`** variables:
  - Equality (**==**), inequality (**!=**) with another **`chandle`** or with **`null`**
  - Case equality (**===**), case inequality (**!==**) with another **`chandle`** or with **`null`** (same semantics as **==** and **!=**)

— Chandles can be tested for a Boolean value, which shall be 0 if the **`chandle`** is **`null`** and 1 otherwise.

— Only the following assignments can be made to a **`chandle`**:
  - Assignment from another **`chandle`**
  - Assignment to **`null`**

— Chandles can be inserted into associative arrays (refer to 7.8), but the relative ordering of any two entries in such an associative array can vary, even between successive runs of the same tool.

— Chandles can be used within a class.

— Chandles can be passed as arguments to subroutines.

— Chandles can be returned from functions.

The use of chandles is restricted as follows:

— Ports shall not have the **`chandle`** data type.

— Chandles shall not be assigned to variables of any other type.

— Chandles shall not be used as follows:
  - In any expression other than as permitted in this subclause
  - As ports
  - In sensitivity lists or event expressions
  - In continuous assignments
  - In untagged unions
  - In packed types

## 6.15 Class

A class variable can hold a handle to a class object. Defining classes and creating objects is discussed in Clause 8.

## 6.16 String data type

The **`string`** data type is an ordered collection of characters. The length of a **`string`** variable is the number of characters in the collection. Variables of type **`string`** are dynamic as their length may vary during simulation. A single character of a **`string`** variable may be selected for reading or writing by indexing the variable. A single character of a **`string`** variable is of type **`byte`**.

SystemVerilog also includes a number of special methods to work with strings, which are defined in this subclause.

A string variable does not represent a string in the same way as a string literal (see 5.9). String literals behave like packed arrays of a width that is a multiple of 8 bits. A string literal assigned to a packed array of an integral variable of a different size is either truncated to the size of the variable or padded with zeros to the left as necessary. When using the **string** data type instead of an integral variable, strings can be of arbitrary length and no truncation occurs. String literals are implicitly converted to the **string** type when assigned to a **string** type or used in an expression involving **string** type operands.

The indices of string variables shall be numbered from 0 to *N*–1 (where *N* is the length of the string) so that index 0 corresponds to the first (leftmost) character of the string and index *N*–1 corresponds to the last (rightmost) character of the string. The string variables can take on the special value `""`, which is the empty string. Indexing an empty string variable shall be an out-of-bounds access.

A string variable shall not contain the special character `"\0"`. Assigning the value 0 to a string character shall be ignored.

The syntax to declare a string variable is as follows:

```
string variable_name [= initial_value];
```

where `variable_name` is a valid identifier and the optional `initial_value` can be a string literal, the value `""` for an empty string, or a string data type expression. For example:

```
parameter string default_name = "John Smith";
string myName = default_name;
```

If an initial value is not specified in the declaration, the variable is initialized to `""`, the empty string. An empty string has zero length.

SystemVerilog provides a set of operators that can be used to manipulate combinations of string variables and string literals. The basic operators defined on the **string** data type are listed in Table 6-9.

A string literal can be assigned to a variable of a **string** or an integral data type. When assigning to a variable of integral data type, if the number of bits of the data object is not equal to the number of characters in the string literal multiplied by 8, the literal is right justified and either truncated on the left or zero-filled on the left, as necessary. For example:

```
byte c = "A";                  // assigns to c "A"
bit [10:0] b = "\x41";         // assigns to b 'b000_0100_0001
bit [1:4][7:0] h = "hello" ;   // assigns to h "ello"
```

A string literal or an expression of **string** type can be assigned directly to a variable of **string** type (a *string variable*). Values of integral type can be assigned to a string variable, but require a cast. When casting an integral value to a string variable, that variable shall grow or shrink to accommodate the integral value. If the size of the integral value is not a multiple of 8 bits, then the value shall be zero-filled on the left so that its size is a multiple of 8 bits.

A string literal assigned to a string variable is converted according to the following steps:

— All `"\0"` characters in the string literal are ignored (i.e., removed from the string).
— If the result of the first step is an empty string literal, the string is assigned the empty string.
— Otherwise, the string is assigned the remaining characters in the string literal.

Casting an integral value to a string variable proceeds in the following steps:

— If the size (in bits) of the integral value is not a multiple of 8, the integral value is left extended and filled with zeros until its bit size is a multiple of 8. The extended value is then treated the same as a string literal, where each successive 8 bits represent a character.
— The steps described previously for string literal conversion are then applied to the extended value.

For example:

```
string s0 = "String literal assign";// sets s0 to "String literal assign"
string s1 = "hello\0world";         // sets s1 to "helloworld"
bit [11:0] b = 12'ha41;
string s2 = string'(b);             // sets s2 to 16'h0a41
```

As a second example:

```
typedef logic [15:0] r_t;
r_t r;
integer i = 1;
string b = "";
string a = {"Hi", b};

r = r_t'(a);          // OK
b = string'(r);       // OK
b = "Hi";             // OK
b = {5{"Hi"}};        // OK
a = {i{"Hi"}};        // OK (non-constant replication)
r = {i{"Hi"}};        // invalid (non-constant replication)
a = {i{b}};           // OK
a = {a,b};            // OK
a = {"Hi",b};         // OK
r = {"H",""};         // yields "H\0". "" is converted to 8'b0
b = {"H",""};         // yields "H". "" is the empty string
a[0] = "h";           // OK, same as a[0] = "cough"
a[0] = b;             // invalid, requires a cast
a[1] = "\0";          // ignored, a is unchanged
```

**Table 6-9—String operators**

| Operator | Semantics |
|---|---|
| `Str1 == Str2` | *Equality*. Checks whether the two string operands are equal. Result is 1 if they are equal and 0 if they are not. Both operands can be expressions of **`string`** type, or one can be an expression of **`string`** type and the other can be a string literal, which shall be implicitly converted to **`string`** type for the comparison. If both operands are string literals, the operator is the same equality operator as for integral types. |
| `Str1 != Str2` | *Inequality*. Logical negation of == |
| `Str1 < Str2`<br>`Str1 <= Str2`<br>`Str1 > Str2`<br>`Str1 >= Str2` | *Comparison*: Relational operators return 1 if the corresponding condition is true using the lexicographic ordering of the two strings `Str1` and `Str2`. The comparison uses the compare string method. Both operands can be expressions of **`string`** type, or one can be an expression of **`string`** type and the other can be a string literal, which shall be implicitly converted to **`string`** type for the comparison. If both operands are string literals, the operator is the same comparison operator as for integral types. |

**Table 6-9—String operators *(continued)***

| Operator | Semantics |
|---|---|
| `{Str1,Str2,...,Strn}` | *Concatenation*: Each operand can be a string literal or an expression of **`string`** type. If all the operands are string literals the expression shall behave as a concatenation of integral values; if the result of such a concatenation is used in an expression involving **`string`** types then it shall be implicitly converted to **`string`** type. If at least one operand is an expression of **`string`** type, then any operands that are string literals shall be converted to **`string`** type before the concatenation is performed, and the result of the concatenation shall be of **`string`** type. |
| `{multiplier{Str}}` | *Replication*: `Str` can be a string literal or an expression of **`string`** type. `multiplier` shall be an expression of integral type and is not required to be a constant expression. If `multiplier` is non-constant or `Str` is an expression of **`string`** type, the result is a string containing *N* concatenated copies of `Str`, where *N* is specified by the `multiplier`. If `Str` is a literal and the `multiplier` is constant, the expression behaves like numeric replication (if the result is used in another expression involving **`string`** types, it is implicitly converted to the **`string`** type). |
| `Str[index]` | *Indexing*. Returns a byte, the ASCII code at the given index. Indices range from 0 to *N*–1, where *N* is the number of characters in the string. If given an index out of range, returns 0. Semantically equivalent to `Str.getc(index)` in 6.16.3. |
| `Str.method(...)` | The dot (`.`) operator is used to invoke a specified method on strings. |

SystemVerilog also includes a number of special methods to work with strings, which use the built-in method notation. These methods are described in 6.16.1 through 6.16.15.

### 6.16.1 Len()

```
function int len();
```

— `str.len()` returns the length of the string, i.e., the number of characters in the string.

— If `str` is " ", then `str.len()` returns 0.

### 6.16.2 Putc()

```
function void putc(int i, byte c);
```

— `str.putc(i, c)` replaces the *i*th character in *str* with the given integral value.

— `putc` does not change the size of `str`: If `i` < 0 or `i` >= `str.len()`, then *str* is unchanged.

— If the second argument to `putc` is zero, the string is unaffected.

The putc method assignment `str.putc(j, x)` is semantically equivalent to `str[j] = x`.

### 6.16.3 Getc()

```
function byte getc(int i);
```

— `str.getc(i)` returns the ASCII code of the `i`th character in `str`.

— If `i` < 0 or `i` >= `str.len()`, then `str.getc(i)` returns 0.

The getc method assignment `x = str.getc(j)` is semantically equivalent to `x = str[j]`.

### 6.16.4 Toupper()

```
function string toupper();
```

- `str.toupper()` returns a string with characters in `str` converted to uppercase.
- `str` is unchanged.

### 6.16.5 Tolower()

```
function string tolower();
```

- `str.tolower()` returns a string with characters in `str` converted to lowercase.
- `str` is unchanged.

### 6.16.6 Compare()

```
function int compare(string s);
```

- `str.compare(s)` compares `str` and `s`, as in the ANSI C `strcmp` function with regard to lexical ordering and return value.

See the relational string operators in Table 6-9.

### 6.16.7 Icompare()

```
function int icompare(string s);
```

- `str.icompare(s)` compares `str` and `s`, like the ANSI C `strcmp` function with regard to lexical ordering and return value, but the comparison is case insensitive.

### 6.16.8 Substr()

```
function string substr(int i, int j);
```

- `str.substr(i, j)` returns a new string that is a substring formed by characters in position `i` through `j` of `str`.
- If `i` < 0, `j` < `i`, or `j` >= `str.len()`, `substr()` returns `""` (the empty string).

### 6.16.9 Atoi(), atohex(), atooct(), atobin()

```
function integer atoi();
function integer atohex();
function integer atooct();
function integer atobin();
```

- `str.atoi()` returns the integer corresponding to the ASCII decimal representation in `str`. For example:

```
str = "123";
int i = str.atoi();  // assigns 123 to i.
```

The conversion scans all leading digits and underscore characters ( _ ) and stops as soon as it encounters any other character or the end of the string. It returns zero if no digits were encountered. It does not parse the full syntax for integer literals (sign, size, apostrophe, base).

- `atohex` interprets the string as hexadecimal.
- `atooct` interprets the string as octal.
- `atobin` interprets the string as binary.

NOTE—These ASCII conversion functions return a 32-bit integer value. Truncation is possible without warning. For converting integer values greater than 32 bits, see `$sscanf` in 21.3.4.

### 6.16.10 Atoreal()

```
function real atoreal();
```

- `str.atoreal()` returns the real number corresponding to the ASCII decimal representation in `str`.

The conversion parses for real constants. The scan stops as soon as it encounters any character that does not conform to this syntax or the end of the string. It returns zero if no digits were encountered.

### 6.16.11 Itoa()

```
function void itoa(integer i);
```

- `str.itoa(i)` stores the ASCII decimal representation of `i` into `str` (inverse of `atoi`).

### 6.16.12 Hextoa()

```
function void hextoa(integer i);
```

- `str.hextoa(i)` stores the ASCII hexadecimal representation of `i` into `str` (inverse of `atohex`).

### 6.16.13 Octtoa()

```
function void octtoa(integer i);
```

- `str.octtoa(i)` stores the ASCII octal representation of `i` into `str` (inverse of `atooct`).

### 6.16.14 Bintoa()

```
function void bintoa(integer i);
```

- `str.bintoa(i)` stores the ASCII binary representation of `i` into `str` (inverse of `atobin`).

### 6.16.15 Realtoa()

```
function void realtoa(real r);
```

- `str.realtoa(r)` stores the ASCII real representation of `r` into `str` (inverse of `atoreal`).

## 6.17 Event data type

An event object gives a powerful and efficient means of describing the communication between, and synchronization of, two or more concurrently active processes. A basic example of this is a small waveform clock generator that synchronizes control of a synchronous circuit by signaling the occurrence of an explicit event periodically while the circuit waits for the event to occur.

The **event** data type provides a handle to a synchronization object. The object referenced by an event variable can be explicitly triggered and waited for. Furthermore, event variables have a persistent triggered state that lasts for the duration of the entire time step. Its occurrence can be recognized by using the event control syntax described in 9.4.2.

An event variable can be assigned or compared to another event variable or assigned the special value **null**. When assigned another event variable, both event variables refer to the same synchronization object. When assigned **null**, the association between the synchronization object and the event variable is broken.

If an initial value is not specified in the declaration of an event variable, then the variable is initialized to a new synchronization object.

*Examples:*

```
event done;                // declare a new event called done
event done_too = done;     // declare done_too as alias to done
event empty = null;        // event variable with no synchronization object
```

Event operations and semantics are discussed in detail in 15.5.

## 6.18 User-defined types

SystemVerilog's data types can be extended with user-defined types using **typedef**. The syntax for declaring user-defined types is shown in Syntax 6-4.

---

type_declaration ::= *// from A.2.1.3*
  **typedef** data_type type_identifier { variable_dimension } **;**
| **typedef** interface_instance_identifier constant_bit_select **.** type_identifier type_identifier **;**
| **typedef** [ **enum** | **struct** | **union** | **class** | **interface class** ] type_identifier **;**

---

*Syntax 6-4—User-defined types (excerpt from Annex A)*

A **typedef** may be used to give a user-defined name to an existing data type. For example:

```
typedef int intP;
```

The named data type can then be used as follows:

```
intP a, b;
```

User-defined data type names must be used for complex data types in casting (see 6.24), which only allows simple data type names, and as type parameter values (see 6.20.3) when unpacked array types are used.

A type parameter may also be used to declare a *type_identifier*. The declaration of a user-defined data type shall precede any reference to its *type_identifier*. User-defined data type identifiers have the same scoping rules as data identifiers, except that hierarchical references to *type_identifier* shall not be allowed. References to type identifiers defined within an interface through ports are not considered hierarchical references and are allowed provided they are locally redefined before being used. Such a **typedef** is called an *interface based typedef*.

```
interface intf_i;
   typedef int data_t;
endinterface

module sub(intf_i p);
   typedef p.data_t my_data_t;
   my_data_t data;
      // type of 'data' will be int when connected to interface above
endmodule
```

Sometimes a user-defined type needs to be declared before the contents of the type have been defined. This is of use with user-defined types derived from the basic data types: **enum**, **struct**, **union**, **interface class**, and **class**. Support for this is provided by the following forms for a *forward typedef*:

```
typedef enum type_identifier;
typedef struct type_identifier;
typedef union type_identifier;
typedef class type_identifier;
typedef interface class type_identifier;
typedef type_identifier;
```

NOTE—While an empty user-defined type declaration is useful for coupled definitions of classes as shown in 8.27, it cannot be used for coupled definitions of structures because structures are statically declared and there is no support for handles to structures.

The last form shows that the basic data type of the user-defined type does not have to be defined in the forward declaration.

The actual data type definition of a forward **typedef** declaration shall be resolved within the same local scope or **generate** block. It shall be an error if the *type_identifier* does not resolve to a data type. It shall be an error if a basic data type was specified by the forward type declaration and the actual type definition does not conform to the specified basic data type. It shall be legal to have a forward type declaration in the same scope, either before or after the final type definition. It shall be legal to have multiple forward type declarations for the same type identifier in the same scope. The use of the term *forward* type declaration does not require the forward type declaration to precede the final type definition.

A forward typedef shall be considered incomplete prior to the final type definition. While incomplete forward types, type parameters, and types defined by an interface based typedef may resolve to class types, use of the class scope resolution operator (see 8.23) to select a type with such a prefix shall be restricted to a typedef declaration. It shall be an error if the prefix does not resolve to a class.

*Example:*

```
typedef C;
C::T x;              // illegal; C is an incomplete forward type
typedef C::T c_t;    // legal; reference to C::T is made by a typedef
c_t y;
class C;
   typedef int T;
endclass
```

## 6.19 Enumerations

Enumerated types shall be defined using the syntax shown in Syntax 6-5.

```
data_type ::=                                                              // from A.2.2.1
    ...
  | enum [ enum_base_type ] { enum_name_declaration { , enum_name_declaration } }
        { packed_dimension }
    ...
enum_base_type ::=
    integer_atom_type [ signing ]
  | integer_vector_type [ signing ] [ packed_dimension ]
  | type_identifier [ packed_dimension ] 15
enum_name_declaration ::=
    enum_identifier [ [ integral_number [ : integral_number ] ] ] [ = constant_expression ]
```

15) A type_identifier shall be legal as an enum_base_type if it denotes an integer_atom_type, with which an additional packed dimension is not permitted, or an integer_vector_type.

*Syntax 6-5—Enumerated types (excerpt from Annex A)*

An enumerated type declares a set of integral named constants. Enumerated data types provide the capability to abstractly declare strongly typed variables without either a data type or data value(s) and later add the required data type and value(s) for designs that require more definition. Enumerated data types also can be easily referenced or displayed using the enumerated names as opposed to the enumerated values.

In the absence of a data type declaration, the default data type shall be **int**. Any other data type used with enumerated types shall require an explicit data type declaration.

An enumerated type defines a set of named values. In the following example, `light1` and `light2` are defined to be variables of the anonymous (unnamed) enumerated **int** type that includes the three members: `red`, `yellow`, and `green`.

```
enum {red, yellow, green} light1, light2; // anonymous int type
```

An enumerated name with `x` or `z` assignments assigned to an **enum** with no explicit data type or an explicit 2-state declaration shall be a syntax error.

```
// Syntax error: IDLE=2'b00, XX=2'bx <ERROR>, S1=2'b01, S2=2'b10
enum bit [1:0] {IDLE, XX='x, S1=2'b01, S2=2'b10} state, next;
```

An **enum** declaration of a 4-state type, such as **integer**, that includes one or more names with `x` or `z` assignments shall be permitted.

```
// Correct: IDLE=0, XX='x, S1=1, S2=2
enum integer {IDLE, XX='x, S1='b01, S2='b10} state, next;
```

An unassigned enumerated name that follows an **enum** name with `x` or `z` assignments shall be a syntax error.

```
// Syntax error: IDLE=0, XX='x, S1=??, S2=??
enum integer {IDLE, XX='x, S1, S2} state, next;
```

The values can be cast to integer types and increment from an initial value of 0. This can be overridden.

```
enum {bronze=3, silver, gold} medal; // silver=4, gold=5
```

The values can be set for some of the names and not set for other names. The optional value of an **enum** named constant is an elaboration time constant expression (see 6.20) and can include references to parameters, local parameters, genvars, other **enum** named constants, and constant functions of these. Hierarchical names and **const** variables are not allowed. A name without a value is automatically assigned an increment of the value of the previous name. It shall be an error to automatically increment the maximum representable value of the enum.

```
// c is automatically assigned the increment-value of 8
enum {a=3, b=7, c} alphabet;
```

Both the enumeration names and their integer values shall be unique. It shall be an error to set two values to the same name or to set the same value to two names, regardless of whether the values are set explicitly or by automatic incrementing.

```
// Error: c and d are both assigned 8
enum {a=0, b=7, c, d=8} alphabet;
```

If the first name is not assigned a value, it is given the initial value of 0.

```
// a=0, b=7, c=8
enum {a, b=7, c} alphabet;
```

The integer value expressions are evaluated in the context of a cast to the **enum** base type. Any enumeration encoding value that is outside the representable range of the **enum** base type shall be an error. For an unsigned base type, this occurs if the cast truncates the value and any of the discarded bits are nonzero. For a signed base type, this occurs if the cast truncates the value and any of the discarded bits are not equal to the sign bit of the result. If the integer value expression is a sized literal constant, it shall be an error if the size is different from the **enum** base type, even if the value is within the representable range. The value after the cast is the value used for the name, including in the uniqueness check and automatic incrementing to get a value for the next name.

```
// Correct declaration - bronze and gold are unsized
enum bit [3:0] {bronze='h3, silver, gold='h5} medal2;

// Correct declaration - bronze and gold sizes are redundant
enum bit [3:0] {bronze=4'h3, silver, gold=4'h5} medal3;

// Error in the bronze and gold member declarations
enum bit [3:0] {bronze=5'h13, silver, gold=3'h5} medal4;

// Error in c declaration, requires at least 2 bits
enum bit [0:0] {a,b,c} alphabet;
```

Type checking of enumerated types used in assignments, as arguments, and with operators is covered in 6.19.3. As in C, there is no overloading of literals; therefore, `medal2` and `medal3` cannot be defined in the same scope because they contain the same names.

### 6.19.1 Defining new data types as enumerated types

A type name can be given so that the same type can be used in many places.

```
typedef enum {NO, YES} boolean;
boolean myvar; // named type
```

### 6.19.2 Enumerated type ranges

A range of enumeration elements can be specified automatically, via the syntax shown in Table 6-10.

**Table 6-10—Enumeration element ranges**

| | |
|---|---|
| `name` | Associates the next consecutive number with name. |
| `name = C` | Associates the constant `C` to `name`. |
| `name[N]` | Generates *N* named constants in the sequence: name0, name1,..., nameN–1. *N* shall be a positive integral number. |
| `name[N] = C` | Optionally, a constant can be assigned to the generated named constants to associate that constant to the first generated named constant; subsequent generated named constants are associated consecutive values.<br>*N* shall be a positive integral number. |
| `name[N:M]` | Creates a sequence of named constants starting with nameN and incrementing or decrementing until reaching named constant nameM.<br>*N* and *M* shall be nonnegative integral numbers. |
| `name[N:M] = C` | Optionally, a constant can be assigned to the generated named constants to associate that constant to the first generated named constants; subsequent generated named constants are associated consecutive values.<br>*N* and *M* shall be nonnegative integral numbers. |

For example:

```
typedef enum { add=10, sub[5], jmp[6:8] } E1;
```

This example defines the enumerated type `E1`, which assigns the number 10 to the enumerated named constant `add`. It also creates the enumerated named constants `sub0`, `sub1`, `sub2`, `sub3`, and `sub4` and assigns them the values 11...15, respectively. Finally, the example creates the enumerated named constants `jmp6`, `jmp7`, and `jmp8` and assigns them the values 16 through 18, respectively.

```
enum { register[2] = 1, register[2:4] = 10 } vr;
```

The preceding example declares enumerated variable `vr`, which creates the enumerated named constants `register0` and `register1`, which are assigned the values 1 and 2, respectively. Next, it creates the enumerated named constants `register2`, `register3`, and `register4` and assigns them the values 10, 11, and 12.

### 6.19.3 Type checking

Enumerated types are strongly typed; thus, a variable of type **`enum`** cannot be directly assigned a value that lies outside the enumeration set unless an explicit cast is used or unless the **`enum`** variable is a member of a union. This is a powerful type-checking aid, which prevents users from accidentally assigning nonexistent values to variables of an enumerated type. The enumeration values can still be used as constants in expressions, and the results can be assigned to any variable of a compatible integral type.

Enumerated variables are type-checked in assignments, arguments, and relational operators. Enumerated variables are auto-cast into integral values, but assignment of arbitrary expressions to an enumerated variable requires an explicit cast.

For example:

```
typedef enum { red, green, blue, yellow, white, black } Colors;
```

This operation assigns a unique number to each of the color identifiers and creates the new data type `Colors`. This type can then be used to create variables of that type.

```
Colors c;
c = green;
c = 1;              // Invalid assignment
if ( 1 == c )       // OK. c is auto-cast to integer
```

In the preceding example, the value `green` is assigned to the variable `c` of type `Colors`. The second assignment is invalid because of the strict typing rules enforced by enumerated types.

Casting can be used to perform an assignment of a different data type, or an out-of-range value, to an enumerated type. Casting is discussed in 6.19.4, 6.24.1, and 6.24.2.

### 6.19.4 Enumerated types in numerical expressions

Elements of enumerated type variables can be used in numerical expressions. The value used in the expression is the numerical value associated with the enumerated value. For example:

```
typedef enum { red, green, blue, yellow, white, black } Colors;

Colors col;
integer a, b;

a = blue * 3;
col = yellow;
b = col + green;
```

From the previous declaration, `blue` has the numerical value 2. This example assigns `a` the value of 6 (2*3), and it assigns `b` a value of 4 (3+1).

An **enum** variable or identifier used as part of an expression is automatically cast to the base type of the **enum** declaration (either explicitly or using **int** as the default). A cast shall be required for an expression that is assigned to an **enum** variable where the type of the expression is not equivalent to the enumeration type of the variable.

Casting to an **enum** type shall cause a conversion of the expression to its base type without checking the validity of the value (unless a dynamic cast is used as described in 6.24.2).

```
typedef enum {Red, Green, Blue} Colors;
typedef enum {Mo,Tu,We,Th,Fr,Sa,Su} Week;
Colors C;
Week W;
int I;

C = Colors'(C+1);             // C is converted to an integer, then added to
                              // one, then converted back to a Colors type

C = C + 1; C++; C+=2; C = I;  // Illegal because they would all be
                              // assignments of expressions without a cast

C = Colors'(Su);              // Legal; puts an out of range value into C

I = C + W;                    // Legal; C and W are automatically cast to int
```

### 6.19.5 Enumerated type methods

SystemVerilog includes a set of specialized methods to enable iterating over the values of enumerated types, which are defined in 6.19.5.1 through 6.19.5.6.

#### 6.19.5.1 First()

The prototype for the `first()` method is as follows:

```
function enum first();
```

The `first()` method returns the value of the first member of the enumeration.

#### 6.19.5.2 Last()

The prototype for the `last()` method is as follows:

```
function enum last();
```

The `last()` method returns the value of the last member of the enumeration.

#### 6.19.5.3 Next()

The prototype for the `next()` method is as follows:

```
function enum next( int unsigned N = 1 );
```

The `next()` method returns the Nth next enumeration value (default is the next one) starting from the current value of the given variable. A wrap to the start of the enumeration occurs when the end of the enumeration is reached. If the given value is not a member of the enumeration, the `next()` method returns the default initial value for the enumeration (see Table 6-7).

#### 6.19.5.4 Prev()

The prototype for the `prev()` method is as follows:

```
function enum prev( int unsigned N = 1 );
```

The `prev()` method returns the Nth previous enumeration value (default is the previous one) starting from the current value of the given variable. A wrap to the end of the enumeration occurs when the start of the enumeration is reached. If the given value is not a member of the enumeration, the `prev()` method returns the default initial value for the enumeration (see Table 6-7).

#### 6.19.5.5 Num()

The prototype for the `num()` method is as follows:

```
function int num();
```

The `num()` method returns the number of elements in the given enumeration.

#### 6.19.5.6 Name()

The prototype for the name() method is as follows:

```
function string name();
```

The `name()` method returns the string representation of the given enumeration value. If the given value is not a member of the enumeration, the `name()` method returns the empty string.

### 6.19.5.7 Using enumerated type methods

The following code fragment shows how to display the name and value of all the members of an enumeration:

```
typedef enum { red, green, blue, yellow } Colors;
Colors c = c.first;
forever begin
   $display( "%s : %d\n", c.name, c );
   if( c == c.last ) break;
   c = c.next;
end
```

## 6.20 Constants

Constants are named data objects that never change. SystemVerilog provides three elaboration-time constants: **`parameter`**, **`localparam`**, and **`specparam`**. SystemVerilog also provides a run-time constant, **`const`** (see 6.20.6).

The **`parameter`**, **`localparam`**, and **`specparam`** constants are collectively referred to as *parameter* constants.

Parameter constants can be initialized with a literal.

```
localparam byte colon1 = ":" ;
specparam delay = 10 ; // specparams are used for specify blocks
parameter logic flag = 1 ;
```

SystemVerilog provides several methods for setting the value of parameter constants. Each parameter may be assigned a default value when declared. The value of a parameter of an instantiated module, interface, or program can be overridden in each instance using one of the following:

- Assignment by ordered list (e.g., `m #(value, value) u1 (...);` ) (see 23.10.2.1)
- Assignment by name
  (e.g., `m #(.param1(value), .param2(value)) u1 (...);` ) (see 23.10.2.2)
- **`defparam`** statements, using hierarchical path names to redefine each parameter (see 23.10.1)

NOTE—The **`defparam`** statement might be removed from future versions of the language. See C.4.1.

### 6.20.1 Parameter declaration syntax

local_parameter_declaration ::= *// from A.2.1.1*
    **`localparam`** data_type_or_implicit list_of_param_assignments
    | **`localparam type`** list_of_type_assignments

parameter_declaration ::=
    **`parameter`** data_type_or_implicit list_of_param_assignments
    | **`parameter type`** list_of_type_assignments

specparam_declaration ::=
    **`specparam`** [ packed_dimension ] list_of_specparam_assignments **`;`**

data_type_or_implicit ::= *// from A.2.2.1*

```
      data_type
    | implicit_data_type
implicit_data_type ::= [ signing ] { packed_dimension }
list_of_param_assignments ::= param_assignment { , param_assignment }          // from A.2.3
list_of_specparam_assignments ::= specparam_assignment { , specparam_assignment }
list_of_type_assignments ::= type_assignment { , type_assignment }
param_assignment ::=                                                            // from A.2.4
      parameter_identifier { unpacked_dimension } [ = constant_param_expression ][18]
specparam_assignment ::=
      specparam_identifier = constant_mintypmax_expression
    | pulse_control_specparam
type_assignment ::=
      type_identifier [ = data_type ][18]
parameter_port_list ::=                                                         // from A.1.3
      # ( list_of_param_assignments { , parameter_port_declaration } )
    | # ( parameter_port_declaration { , parameter_port_declaration } )
    | #( )
parameter_port_declaration ::=
      parameter_declaration
    | local_parameter_declaration
    | data_type list_of_param_assignments
    | type list_of_type_assignments
```

18) It shall be legal to omit the constant_param_expression from a param_assignment or the data_type from a type_assignment only within a parameter_port_list. However, it shall not be legal to omit them from localparam declarations in a parameter_port_list.

*Syntax 6-6—Parameter declaration syntax (excerpt from Annex A)*

The *list_of_param_assignments* can appear in a module, interface, program, class, or package or in the *parameter_port_list* of a module (see 23.2), interface, program, or class. If the declaration of a design element uses a *parameter_port_list* (even an empty one), then in any *parameter_declaration* directly contained within the declaration, the **parameter** keyword shall be a synonym for the **localparam** keyword (see 6.20.4). All *param_assignments* appearing within a class body shall become **localparam** declarations regardless of the presence or absence of a *parameter_port_list*. All *param_assignments* appearing within a **generate** block, package, or compilation-unit scope shall become **localparam** declarations.

The **parameter** keyword can be omitted in a parameter port list. For example:

```
class vector #(size = 1);  // size is a parameter in a parameter port list
   logic [size-1:0] v;
endclass

interface simple_bus #(AWIDTH = 64, type T = word)  // parameter port list
                      (input logic clk) ;             // port list
   ...
endinterface
```

In a list of parameter constants, a parameter can depend on earlier parameters. In the following declaration, the default value of the second parameter depends on the value of the first parameter. The third parameter is a type, and the fourth parameter is a value of that type.

```
module mc #(int N = 5, M = N*16, type T = int, T x = 0)
 ( ... );
...
endmodule
```

In the declaration of a parameter in a parameter port list, the specification for its default value may be omitted, in which case the parameter shall have no default value. If no default value is specified for a parameter of a design element, then an overriding parameter value shall be specified in every instantiation of that design element (see 23.10). Also, if no default value is specified for a parameter of a design element, then a tool shall not implicitly instantiate that design element (see 23.3, 23.4, and 24.3). If no default value is specified for a parameter of a class, then an overriding parameter value shall be specified in every specialization of that class (see 8.25).

```
class Mem #(type T, int size);
   T words[size];
   ...
endclass

typedef Mem#(byte, 1024) Kbyte;
```

### 6.20.2 Value parameters

A parameter constant can have a *type* specification and a *range* specification. The type and range of parameters shall be in accordance with the following rules:

— A parameter declaration with no type or range specification shall default to the type and range of the final value assigned to the parameter, after any value overrides have been applied. If the expression is real, the parameter is real. If the expression is integral, the parameter is a **logic** vector of the same size with range `[size-1:0]`.

— A parameter with a range specification, but with no type specification, shall have the range of the parameter declaration and shall be unsigned. The sign and range shall not be affected by value overrides.

— A parameter with a type specification, but with no range specification, shall be of the type specified. A signed parameter shall default to the range of the final value assigned to the parameter, after any value overrides have been applied.

— A parameter with a signed type specification and with a range specification shall be signed and shall have the range of its declaration. The sign and range shall not be affected by value overrides.

— A parameter with no range specification and with either a signed type specification or no type specification shall have an implied range with an *lsb* equal to `0` and an *msb* equal to one less than the size of the final value assigned to the parameter.

— A parameter with no range specification, with either a signed type specification or no type specification, and for which the final value assigned to it is unsized shall have an implied range with an *lsb* equal to `0` and an *msb* equal to an implementation-dependent value of at least 31.

In an assignment to, or override of, a parameter with an explicit type declaration, the type of the right-hand expression shall be assignment compatible with the declared type (see 6.22.3).

The conversion rules between real and integer values described in 6.12.2 apply to parameters as well.

Bit-selects and part-selects of parameters that are of integral types shall be allowed (see 6.11.1).

A value parameter (**parameter**, **localparam**, or **specparam**) can only be set to an expression of literals, value parameters or local parameters, genvars, enumerated names, or a constant function of these. Package references are allowed. Hierarchical names are not allowed. A **specparam** can also be set to an expression containing one or more specparams.

*Examples:*

```
parameter   msb = 7;                // defines msb as a constant value 7
parameter   e = 25, f = 9;          // defines two constant numbers
parameter   r = 5.7;                // declares r as a real parameter
parameter   byte_size = 8,
            byte_mask = byte_size - 1;
parameter   average_delay = (r + f) / 2;

parameter signed [3:0] mux_selector = 0;
parameter real r1 = 3.5e17;
parameter   p1 = 13'h7e;
parameter   [31:0] dec_const = 1'b1;   // value converted to 32 bits
parameter   newconst = 3'h4;           // implied range of [2:0]
parameter   newconst = 4;              // implied range of at least [31:0]
```

A parameter can also be declared as an aggregate type, such as an unpacked array or an unpacked structure. An aggregate parameter must be assigned to or overridden as a whole; individual members of an aggregate parameter may not be assigned or overridden separately. However, an individual member of an aggregate parameter may be used in an expression. For example:

```
parameter logic [31:0] P1 [3:0] = '{ 1, 2, 3, 4 } ; // unpacked array
                                                     // parameter declaration
initial begin
  if ( P1[2][7:0] ) ... // use part-select of individual element of the array
```

### 6.20.2.1 $ as a parameter value

The value **$** can be assigned to parameters of integer types. A parameter to which **$** is assigned shall only be used wherever **$** can be specified as a literal constant.

For example, $ represents unbounded range specification, where the upper index can be any integer.

```
parameter r2 = $;
property inq1(r1,r2);
   @(posedge clk) a ##[r1:r2] b ##1 c |=> d;
endproperty
assert property (inq1(3, r2));
```

A system function is provided to test whether a constant is a **$**. The syntax of the system function is

```
$isunbounded(constant_expression)
```

`$isunbounded` returns true (`1'b1`) if *constant_expression* is unbounded. Typically, `$isunbounded` would be used as a condition in the generate statement.

The following example illustrates the benefit of using $ in writing properties concisely where the range is parameterized. The checker in the example verifies that a bus driven by signal `en` remains 0, i.e, quiet for the specified minimum (`min_quiet`) and maximum (`max_quiet`) quiet time.

NOTE—The function `$isunbounded` is used for checking the validity of the actual arguments.

```
interface quiet_time_checker #(parameter min_quiet = 0,
                               parameter max_quiet = 0)
                              (input logic clk, reset_n, logic [1:0]en);

   generate
      if ( max_quiet == 0) begin
         property quiet_time;
            @(posedge clk) reset_n |-> ($countones(en) == 1);
         endproperty
         a1: assert property (quiet_time);
      end
      else begin
         property quiet_time;
            @(posedge clk)
               (reset_n && ($past(en) != 0) && en == 0)
               |->(en == 0)[*min_quiet:max_quiet]
            ##1 ($countones(en) == 1);
         endproperty
         a1: assert property (quiet_time);
      end
      if ((min_quiet == 0) && ($isunbounded(max_quiet))
         $warning(warning_msg);
   endgenerate
endinterface


quiet_time_checker #(0, 0) quiet_never (clk,1,enables);
quiet_time_checker #(2, 4) quiet_in_window (clk,1,enables);
quiet_time_checker #(0, $) quiet_any (clk,1,enables);
```

Another example below illustrates that by testing for `$`, a property can be configured according to the requirements. When parameter `max_cks` is unbounded, it is not required to test for `expr` to become false.

```
interface width_checker #(parameter min_cks = 1, parameter max_cks = 1)
                         (input logic clk, reset_n, expr);

   generate
      if ($isunbounded(max_cks)) begin
         property width;
            @(posedge clk)
               (reset_n && $rose(expr)) |-> (expr [* min_cks]);
         endproperty
         a2: assert property (width);
      end
      else begin
         property assert_width_p;
            @(posedge clk)
               (reset_n && $rose(expr)) |-> (expr[* min_cks:max_cks])
                  ##1 (!expr);
         endproperty
         a2: assert property (width);
      end
   endgenerate
endinterface


width_checker #(3, $) max_width_unspecified (clk,1,enables);
width_checker #(2, 4) width_specified (clk,1,enables);
```

### 6.20.3 Type parameters

A parameter constant can also specify a data type, allowing modules, interfaces, or programs to have ports and data objects whose type is set for each instance.

```
module ma   #( parameter p1 = 1, parameter type p2 = shortint )
            (input logic [p1:0] i, output logic [p1:0] o);
   p2 j = 0; // type of j is set by a parameter, (shortint unless redefined)
   always @(i) begin
      o = i;
      j++;
   end
endmodule

module mb;
   logic [3:0] i,o;
   ma #(.p1(3), .p2(int)) u1(i,o); //redefines p2 to a type of int
endmodule
```

In an assignment to, or override of, a type parameter, the right-hand expression shall represent a data type.

A data-type parameter (**parameter type**) can only be set to a data type. Package references are allowed. Hierarchical names are not allowed.

It shall be illegal to override a type parameter with a **defparam** statement.

### 6.20.4 Local parameters (localparam)

Local parameters are identical to parameters except that they cannot directly be modified by **defparam** statements (see 23.10.1) or instance parameter value assignments (see 23.10.2). Local parameters can be assigned constant expressions (see 11.2.1) containing parameters, which in turn *can* be modified with **defparam** statements or instance parameter value assignments.

Unlike nonlocal parameters, local parameters can be declared in a **generate** block, package, class body, or compilation-unit scope. In these contexts, the **parameter** keyword shall be a synonym for the **localparam** keyword.

Local parameters may be declared in a module's *parameter_port_list*. Any parameter declaration appearing in such a list between a **localparam** keyword and the next **parameter** keyword (or the end of the list, if there is no next **parameter** keyword) shall be a local parameter. Any other parameter declaration in such a list shall be a nonlocal parameter that may be overridden as described in 23.10.

### 6.20.5 Specify parameters

The keyword **specparam** declares a special type of parameter that is intended only for providing timing and delay values, but can appear in any expression that is not assigned to a parameter and is not part of the range specification of a declaration. Specify parameters (also called *specparams*) are permitted both within the specify block (see Clause 30) and in the main module body.

A specify parameter declared outside a specify block shall be declared before it is referenced. The value assigned to a specify parameter can be any constant expression. A specify parameter can be used as part of a constant expression for a subsequent specify parameter declaration. Unlike the **parameter** constant, a specify parameter cannot be modified from within the language, but it can be modified through SDF annotation (see Clause 32).

Specify parameters and **parameter** constants are not interchangeable. In addition, **parameter** and **localparam** shall not be assigned a constant expression that includes any specify parameters. Table 6-11 summarizes the differences between the two types of parameter declarations.

**Table 6-11—Differences between specparams and parameters**

| Specparams (specify parameter) | Parameters |
|---|---|
| Use keyword **specparam** | Use keyword **parameter** |
| Shall be declared *inside* a module or specify block | Shall be declared *outside* specify blocks |
| May only be used inside a module or specify block | May not be used inside specify blocks |
| May be assigned specparams and parameters | May not be assigned specparams |
| Use SDF annotation to override values | Use **defparam** or instance declaration parameter value passing to override values |

A specify parameter can have a range specification. The range of specify parameters shall be in accordance with the following rules:

— A **specparam** declaration with no range specification shall default to the range of the final value assigned to the parameter, after any value overrides have been applied.

— A **specparam** with a range specification shall have the range of the parameter declaration. The range shall not be affected by value overrides.

*Examples:*

```
specify
   specparam tRise_clk_q = 150, tFall_clk_q = 200;
   specparam tRise_control = 40, tFall_control = 50;
endspecify
```

The lines between the keywords **specify** and **endspecify** declare four specify parameters. The first line declares specify parameters called `tRise_clk_q` and `tFall_clk_q` with values 150 and 200, respectively; the second line declares `tRise_control` and `tFall_control` specify parameters with values 40 and 50, respectively.

```
module RAM16GEN ( output [7:0] DOUT,
                  input [7:0] DIN,
                  input [5:0] ADR,
                  input WE, CE);
   specparam dhold = 1.0;
   specparam ddly = 1.0;
   parameter width = 1;
   parameter regsize = dhold + 1.0;    // Illegal - cannot assign
                                       // specparams to parameters
endmodule
```

### 6.20.6 Const constants

A **const** form of constant differs from a **localparam** constant in that the **localparam** shall be set during elaboration, whereas a **const** can be set during simulation, such as in an automatic task.

A static constant declared with the **const** keyword can be set to an expression of literals, parameters, local parameters, genvars, enumerated names, a constant function of these, or other constants. Hierarchical names are allowed because constants declared with the **const** keyword are calculated after elaboration.

```
const logic option = a.b.c ;
```

An automatic constant declared with the **const** keyword can be set to any expression that would be legal without the **const** keyword.

An instance of a class (an object handle) can also be declared with the **const** keyword.

```
const class_name object = new(5,3);
```

In other words, the object acts like a variable that cannot be written. The arguments to the **new** method shall be constant expressions (see 11.2.1). The members of the object can be written (except for those members that are declared **const**).

## 6.21 Scope and lifetime

Variables declared outside a module, program, interface, checker, task, or function are local to the compilation unit and have a static lifetime (exist for the whole simulation). This is roughly equivalent to C static variables declared outside a function, which are local to a file. Variables declared inside a module, interface, program, or checker, but outside a task, process, or function, are local in scope and have a static lifetime.

Variables declared inside a static task, function, or block are local in scope and default to a static lifetime. Specific variables within a static task, function, or block can be explicitly declared as automatic. Such variables have the lifetime of the call or block and are initialized on each entry to the call or block (also see 6.8 on variable initialization). This is roughly equivalent to a C automatic variable.

Tasks and functions may be declared as **automatic**. Variables declared in an automatic task, function, or block are local in scope, default to the lifetime of the call or block, and are initialized on each entry to the call or block (also see 6.8 on variable initialization). An automatic block is one in which declarations are automatic by default. Specific variables within an automatic task, function, or block can be explicitly declared as static. Such variables have a static lifetime. This is roughly equivalent to C static variables declared within a function.

The lifetime of a fork-join block (see 9.3.2) shall encompass the execution of all processes spawned by the block. The lifetime of a scope enclosing any fork-join block includes the lifetime of the fork-join block.

A variable declaration shall precede any simple reference (non-hierarchical) to that variable. Variable declarations shall precede any statements within a procedural block. Variables may also be declared in unnamed blocks. These variables are visible to the unnamed block and any nested blocks below it. Hierarchical references shall not be used to access these variables by name.

```
module msl;
   int st0;                  // static
   initial begin
      int st1;               // static
      static int st2;        // static
      automatic int auto1;   // automatic
   end
   task automatic t1();
      int auto2;             // automatic
      static int st3;         // static
      automatic int auto3;    // automatic
   endtask
endmodule
```

Variables declared in a static task, function, or procedural block default to a static lifetime and a local scope. However, an explicit **static** keyword shall be required when an initialization value is specified as part of a static variable's declaration to indicate the user's intent of executing that initialization only once at the beginning of simulation. The **static** keyword shall be optional where it would not be legal to declare the variables as automatic. For example:

```
module top_legal;
   int svar1 = 1;                      // static keyword optional
   initial begin
      for (int i=0; i<3; i++) begin
         automatic int loop3 = 0;      // executes every loop
         for (int j=0; j<3; j++) begin
            loop3++;
            $display(loop3);
         end
      end // prints 1 2 3 1 2 3 1 2 3
      for (int i=0; i<3; i++) begin
         static int loop2 = 0;         // executes once at time zero
         for (int j=0; j<3; j++) begin
            loop2++;
            $display(loop2);
         end
      end // prints 1 2 3 4 5 6 7 8 9
   end
endmodule : top_legal

module top_illegal;                    // should not compile
   initial begin
      int svar2 = 2;                // static/automatic needed to show intent
      for (int i=0; i<3; i++) begin
         int loop3 = 0;                // illegal statement
         for (int i=0; i<3; i++) begin
            loop3++;
            $display(loop3);
         end
      end
   end
endmodule : top_illegal
```

An optional qualifier can be used to specify the default lifetime of all variables declared in a task, function, or block defined within a module, interface, package, or program. The lifetime qualifier is **automatic** or **static**. The default lifetime is **static**.

```
program automatic test ;
   int i;               // not within a procedural block - static
   task t ( int a );    // arguments and variables in t are automatic
      ...               //   unless explicitly declared static
   endtask
endprogram
```

It is permissible to hierarchically reference any static variable unless the variable is declared inside an unnamed block. This includes static variables declared inside automatic tasks and functions.

Class methods (see Clause 8) and declared **for** loop variables (see 12.7.1) are by default automatic, regardless of the lifetime attribute of the scope in which they are declared.

Automatic variables and elements of dynamically sized array variables shall not be written with nonblocking, continuous, or procedural continuous assignments. Non-static class properties shall not be written with continuous or procedural continuous assignments. References to automatic variables and elements or members of dynamic variables shall be limited to procedural blocks.

See also Clause 13 on tasks and functions.

## 6.22 Type compatibility

Some constructs and operations require a certain level of type compatibility for their operands to be legal. There are five levels of type compatibility, formally defined here: matching, equivalent, assignment compatible, cast compatible, and nonequivalent.

SystemVerilog does not require a category for identical types to be defined here because there is no construct in the SystemVerilog language that requires it. For example, as defined below, **int** can be interchanged with **bit signed** [31:0] wherever it is syntactically legal to do so. Users can define their own level of type identity by using the $typename system function (see 20.6.1) or through use of the PLI.

The scope of a data type identifier shall include the hierarchical instance scope. In other words, each instance with a user-defined type declared inside the instance creates a unique type. To have type matching or equivalence among multiple instances of the same module, interface, program, or checker, a class, **enum**, unpacked structure, or unpacked union type must be declared at a higher level in the compilation-unit scope than the declaration of the module, interface, program, or checker, or imported from a package. For type matching, this is true even for packed structure and packed union types.

### 6.22.1 Matching types

Two data types shall be defined as *matching data types* using the following inductive definition. If two data types do not match using the following definition, then they shall be defined to be nonmatching.

a) Any built-in type matches every other occurrence of itself, in every scope.

b) A simple **typedef** or type parameter override that renames a built-in or user-defined type matches that built-in or user-defined type within the scope of the type identifier.

```
typedef bit node;       // 'bit' and 'node' are matching types
typedef type1 type2;    // 'type1' and 'type2' are matching types
```

c) An anonymous **enum**, **struct**, or **union** type matches itself among data objects declared within the same declaration statement and no other data types.

```
struct packed {int A; int B;} AB1, AB2; // AB1, AB2 have matching types
struct packed {int A; int B;} AB3;  // the type of AB3 does not match
                                    // the type of AB1
```

d) A **typedef** for an **enum**, **struct**, **union**, or **class** matches itself and the type of data objects declared using that data type within the scope of the data type identifier.

```
typedef struct packed {int A; int B;} AB_t;
AB_t AB1; AB_t AB2;  // AB1 and AB2 have matching types

typedef struct packed {int A; int B;} otherAB_t;
```

```
otherAB_t AB3;    // the type of AB3 does not match the type of AB1 or AB2
```

e) A simple bit vector type that does not have a predefined width and one that does have a predefined width match if both are 2-state or both are 4-state, both are signed or both are unsigned, both have the same width, and the range of the simple bit vector type without a predefined width is [width–1:0].

```
typedef bit signed [7:0] BYTE;   // matches the byte type
typedef bit signed [0:7] ETYB;   // does not match the byte type
```

f) Two array types match if they are both packed or both unpacked, are the same kind of array (fixed-size, dynamic, associative, or queue), have matching index types (for associative arrays), and have matching element types. Fixed-size arrays shall also have the same left and right range bounds. Note that the element type of a multidimensional array is itself an array type.

```
typedef byte MEM_BYTES [256];
typedef bit signed [7:0] MY_MEM_BYTES [256];    // MY_MEM_BYTES matches
                                                // MEM_BYTES

typedef logic [1:0] [3:0] NIBBLES;
typedef logic [7:0] MY_BYTE; // MY_BYTE and NIBBLES are not matching types

typedef logic MD_ARY [][2:0];
typedef logic MD_ARY_TOO [][0:2];  // Does not match MD_ARY
```

g) Explicitly adding **signed** or **unsigned** modifiers to a type that does not change its default signing creates a type that matches the type without the explicit signing specification.

```
typedef byte signed MY_CHAR;  // MY_CHAR matches the byte type
```

h) A **typedef** for an **enum**, **struct**, **union**, or **class** type declared in a package always matches itself, regardless of the scope into which the type is imported.

### 6.22.2 Equivalent types

Two data types shall be defined as *equivalent data types* using the following inductive definition. If the two data types are not defined as equivalent using the following definition, then they shall be defined to be nonequivalent.

a) If two types match, they are equivalent.

b) An anonymous **enum**, unpacked **struct**, or unpacked **union** type is equivalent to itself among data objects declared within the same declaration statement and no other data types.

```
struct {int A; int B;} AB1, AB2;    // AB1, AB2 have equivalent types
struct {int A; int B;} AB3;         // AB3 is not type equivalent to AB1
```

c) Packed arrays, packed structures, packed unions, and built-in integral types are equivalent if they contain the same number of total bits, are either all 2-state or all 4-state, and are either all signed or all unsigned.

NOTE—If any bit of a packed structure or union is 4-state, the entire structure or union is considered 4-state.

```
typedef bit signed [7:0] BYTE;      // equivalent to the byte type
typedef struct packed signed {bit[3:0] a, b;} uint8;
                                    // equivalent to the byte type
```

d) Unpacked fixed-size array types are equivalent if they have equivalent element types and equal size; the actual range bounds may differ. Note that the element type of a multidimensional array is itself an array type.

```
bit [9:0]  A [0:5];
bit [1:10] B [6];
typedef bit [10:1] uint10;
uint10 C [6:1];               // A, B and C have equivalent types
typedef int anint [0:0];      // anint is not type equivalent to int
```

e) Dynamic array, associative array, and queue types are equivalent if they are the same kind of array (dynamic, associative, or queue), have equivalent index types (for associative arrays), and have equivalent element types.

The following example is assumed to be within one compilation unit, although the package declaration need not be in the same unit:

```
package p1;
   typedef struct {int A;} t_1;
endpackage

typedef struct {int A;} t_2;

module sub();
   import p1::t_1;
   parameter type t_3 = int;
   parameter type t_4 = int;
   typedef struct {int A;} t_5;
   t_1 v1; t_2 v2; t_3 v3; t_4 v4; t_5 v5;
endmodule

module top();
   typedef struct {int A;} t_6;
   sub #(.t_3(t_6)) s1 ();
   sub #(.t_3(t_6)) s2 ();

   initial begin
      s1.v1 = s2.v1; // legal - both types from package p1 (rule 8)
      s1.v2 = s2.v2; // legal - both types from $unit (rule 4)
      s1.v3 = s2.v3; // legal - both types from top (rule 2)
      s1.v4 = s2.v4; // legal - both types are int (rule 1)
      s1.v5 = s2.v5; // illegal - types from s1 and s2 (rule 4)
   end
endmodule
```

### 6.22.3 Assignment compatible

All equivalent types, and all nonequivalent types that have implicit casting rules defined between them, are *assignment-compatible types*. For example, all integral types are assignment compatible. Conversion between assignment-compatible types can involve loss of data by truncation or rounding.

Unpacked arrays are assignment compatible with certain other arrays that are not of equivalent type. Assignment compatibility of unpacked arrays is discussed in detail in 7.6.

Compatibility can be in one direction only. For example, an **enum** can be converted to an integral type without a cast, but not the other way around. Implicit casting rules are defined in 6.24.

### 6.22.4 Cast compatible

All assignment-compatible types, plus all nonequivalent types that have defined explicit casting rules, are *cast-compatible types*. For example, an integral type requires a cast to be assigned to an **enum**.

Explicit casting rules are defined in 6.24.

### 6.22.5 Type incompatible

*Type incompatible* includes all the remaining nonequivalent types that have no defined implicit or explicit casting rules. Class handles, interface class handles, and chandles are type incompatible with all other types.

### 6.22.6 Matching nettypes

a) A **nettype** matches itself and the **nettype** of nets declared using that **nettype** within the scope of the **nettype** type identifier.

b) A simple **nettype** that renames a user-defined **nettype** matches that user-defined **nettype** within the scope of the **nettype** identifier.

```
// declare another name nettypeid2 for nettype nettypeid1
nettype nettypeid1 nettypeid2;
```

## 6.23 Type operator

The **type** operator provides a way to refer to the data type of an expression. A type reference can be used like a type name or local type parameter, for example, in casts, data object declarations, and type parameter assignments and overrides. It can also be used in equality/inequality and case equality/inequality comparisons with other type references, and such comparisons are considered to be constant expressions (see 11.2.1). When a type reference is used in a net declaration, it shall be preceded by a net type keyword; and when it is used in a variable declaration, it shall be preceded by the **var** keyword.

```
var type(a+b) c, d;

c = type(i+3)'(v[15:0]);
```

The **type** operator applied to an expression shall represent the self-determined result type of that expression. The expression shall not be evaluated and shall not contain any hierarchical references or references to elements of dynamic objects.

The type operator can also be applied to a data type.

```
localparam type T = type(bit[12:0]);
```

When a type reference is used in an equality/inequality or case equality/inequality comparison, it shall only be compared with another type reference. Two type references shall be considered equal in such comparisons if, and only if, the types to which they refer match (see 6.22.1).

```
bit [12:0] A_bus, B_bus;
parameter type bus_t = type(A_bus);
generate
   case (type(bus_t))
      type(bit[12:0]): addfixed_int #(bus_t) (A_bus,B_bus);
      type(real): add_float #(type(A_bus)) (A_bus,B_bus);
   endcase
endgenerate
```

## 6.24 Casting

### 6.24.1 Cast operator

A data type can be changed by using a cast ( `'` ) operation. The syntax for cast operations is shown in Syntax 6-7.

---

constant_cast ::= *// from A.8.4*
  casting_type **'** **(** constant_expression **)**

cast ::=
  casting_type **'** **(** expression **)**

casting_type ::= simple_type | constant_primary | signing | **string** | **const** *// from A.2.2.1*

simple_type ::= integer_type | non_integer_type | ps_type_identifier | ps_parameter_identifier

---

*Syntax 6-7—Casting (excerpt from Annex A)*

In a static cast, the expression to be cast shall be enclosed in parentheses that are prefixed with the casting type and an apostrophe. If the expression is assignment compatible with the casting type, then the cast shall return the value that a variable of the casting type would hold after being assigned the expression. If the expression is not assignment compatible with the casting type, then if the casting type is an enumerated type, the behavior shall be as described as in 6.19.4, and if the casting type is a bit-stream type, the behavior shall be as described in 6.24.3.

```
int'(2.0 * 3.0)
shortint'({8'hFA,8'hCE})
```

Thus, in the following example, if expressions `expr_1` and `expr_2` are assignment compatible with data types `cast_t1` and `cast_t2`, respectively, then

```
A = cast_t1'(expr_1) + cast_t2'(expr_2);
```

is the same as

```
cast_t1 temp1;
cast_t2 temp2;

temp1 = expr_1;
temp2 = expr_2;
A = temp1 + temp2;
```

Thus, an implicit cast (e.g., `temp1 = expr1`), if defined, gives the same results as the corresponding explicit cast (`cast_t1'(expr1)`).

If the casting type is a constant expression with a positive integral value, the expression in parentheses shall be padded or truncated to the size specified. It shall be an error if the size specified is zero or negative.

*Examples:*

```
17'(x - 2)

parameter P = 16;
(P+1)'(x - 2)
```

The signedness can also be changed.

```
signed'(x)
```

The expression inside the cast shall be an integral value when changing the size or signing.

When changing the size, the cast shall return the value that a packed array type with a single `[n-1:0]` dimension would hold after being assigned the expression, where `n` is the cast size. The signedness shall pass through unchanged, i.e., the signedness of the result shall be the self-determined signedness of the expression inside the cast. The array elements shall be of type **`bit`** if the expression inside the cast is 2-state, otherwise they shall be of type **`logic`**.

When changing the signing, the cast shall return the value that a packed array type with a single `[n-1:0]` dimension would hold after being assigned the expression, where `n` is the number of bits in the expression to be cast (`$bits(expression)`). The signedness of the result shall be the signedness specified by the cast type. The array elements shall be of type **`bit`** if the expression inside the cast is 2-state; otherwise, they shall be of type **`logic`**.

NOTE—The `$signed()` and `$unsigned()` system functions (see 11.7) return the same results as **`signed'`**`()` and **`unsigned'`**`()`, respectively.

*Examples:*

```
logic [7:0] regA;
logic signed [7:0] regS;

regA = unsigned'(-4);      // regA = 8'b11111100
regS = signed'(4'b1100);   // regS = -4
```

An expression may be changed to a constant with a **`const`** cast.

```
const'(x)
```

When casting an expression as a constant, the type of the expression to be cast shall pass through unchanged. The only effect is to treat the value as though it had been used to define a **`const`** variable of the type of the expression.

When casting to a predefined type, the prefix of the cast shall be the predefined type keyword. When casting to a user-defined type, the prefix of the cast shall be the user-defined type identifier.

When a **`shortreal`** is converted to an **`int`** or to 32 bits using either casting or assignment, its value is rounded (see 6.12). Therefore, the conversion can lose information. To convert a **`shortreal`** to its underlying bit representation without a loss of information, use `$shortrealtobits` as defined in 20.5. To convert from the bit representation of a shortreal value into a **`shortreal`**, use `$bitstoshortreal` as defined in 20.5.

Structures can be converted to bits preserving the bit pattern. In other words, they can be converted back to the same value without any loss of information. When unpacked data are converted to the packed representation, the order of the data in the packed representation is such that the first field in the structure occupies the MSBs. The effect is the same as a concatenation of the data items (**`struct`** fields or array elements) in order. The type of the elements in an unpacked structure or array shall be valid for a packed representation in order to be cast to any other type, whether packed or unpacked.

An explicit cast between packed types is not required because they are implicitly cast as integral values, but a cast can be used by tools to perform stronger type checking.

The following example demonstrates how `$bits` can be used to obtain the size of a structure in bits (the `$bits` system function is discussed in 20.6.2), which facilitates conversion of the structure into a packed array:

```
typedef struct {
            bit isfloat;
            union { int i; shortreal f; } n; // anonymous type
} tagged_st;                        // named structure

typedef bit [$bits(tagged_st) - 1 : 0] tagbits;  // tagged_st defined above

tagged_st a [7:0];                  // unpacked array of structures

tagbits t = tagbits'(a[3]);         // convert structure to array of bits
a[4] = tagged_st'(t);               // convert array of bits back to structure
```

Note that the **`bit`** data type loses `X` values. If these are to be preserved, the **`logic`** type should be used instead.

The size of a union in bits is the size of its largest member. The size of a **`logic`** in bits is 1.

The functions `$itor`, `$rtoi`, `$bitstoreal`, `$realtobits`, `$signed`, and `$unsigned` can also be used to perform type conversions (see Clause 20).

### 6.24.2 $cast dynamic casting

The `$cast` system task can be used to assign values to variables that might not ordinarily be valid because of differing data type. `$cast` can be called as either a task or a function.

The syntax for `$cast` is as follows:

```
function int $cast( singular dest_var, singular source_exp );
```

or

```
task $cast( singular dest_var, singular source_exp );
```

The *dest_var* is the variable to which the assignment is made.

The *source_exp* is the expression that is to be assigned to the destination variable.

Use of `$cast` as either a task or a function determines how invalid assignments are handled.

When called as a task, `$cast` attempts to assign the source expression to the destination variable. If the assignment is invalid, a run-time error occurs, and the destination variable is left unchanged.

When called as a function, `$cast` attempts to assign the source expression to the destination variable and returns 1 if the cast is legal. If the cast fails, the function does not make the assignment and returns 0. When called as a function, no run-time error occurs, and the destination variable is left unchanged.

It is important to note that `$cast` performs a run-time check. No type checking is done by the compiler, except to check that the destination variable and source expression are singulars. The `$cast` behavior when applied to class handles is described in 8.16.

For example:

```
typedef enum { red, green, blue, yellow, white, black } Colors;
```

```
Colors col;
$cast( col, 2 + 3 );
```

This example assigns the expression `(5 => black)` to the enumerated type. Without `$cast` or a static compile-time cast operation, this type of assignment is illegal.

The following example shows how to use the `$cast` to check whether an assignment will succeed:

```
if ( ! $cast( col, 2 + 8 ) )     // 10: invalid cast
   $display( "Error in cast" );
```

Alternatively, the preceding examples can be cast using a static cast operation. For example:

```
col = Colors'(2 + 3);
```

However, this is a compile-time cast, i.e, a coercion that always succeeds at run time and does not provide for error checking or warn if the expression lies outside the enumeration values.

Allowing both types of casts gives full control to the user. If users know that certain expressions assigned to an enumerated variable lie within the enumeration values, the faster static compile-time cast can be used. If users need to check if an expression lies within the enumeration values, it is not necessary to write a lengthy case statement manually. The compiler automatically provides that functionality via the `$cast` function. By providing both types of casts, SystemVerilog enables users to balance the trade-offs of performance and checking associated with each cast type.

NOTE—`$cast` is similar to the dynamic_cast function available in C++. However, `$cast` allows users to check whether the operation will succeed, whereas dynamic_cast always raises a C++ exception.

### 6.24.3 Bit-stream casting

Type casting can also be applied to unpacked arrays and structs. It is thus possible to convert freely between bit-stream types using explicit casts. Types that can be packed into a stream of bits are called *bit-stream types*. A bit-stream type is a type consisting of the following:

— Any integral, packed, or string type
— Unpacked arrays, structures, or classes of the preceding types
— Dynamically sized arrays (dynamic, associative, or queues) of any of the preceding types

This definition is recursive so that, for example, a structure containing a queue of **`int`** is a bit-stream type.

Assuming `A` is of bit-stream type `source_t` and `B` is of bit-stream type `dest_t`, it is legal to convert `A` into `B` by an explicit cast:

```
B = dest_t'(A);
```

The conversion from `A` of type `source_t` to `B` of type `dest_t` proceeds in two steps:

a) Conversion from `source_t` to a generic packed value containing the same number of bits as `source_t`. If `source_t` contains any 4-state data, the entire packed value is 4-state; otherwise, it is 2-state.
b) Conversion from the generic packed value to `dest_t`. If the generic packed value is a 4-state type and parts of `dest_t` designate 2-state types, then those parts in `dest_t` are assigned as if cast to a 2-state.

When a dynamic array, queue, or string type is converted to the packed representation, the item at index 0 occupies the MSBs. When an associative array is converted to the packed representation, items are packed in index-sorted order with the first indexed element occupying the MSBs. An associative array type or class shall be illegal as a destination type. A class handle with local or protected members shall be illegal as a source type except when the handle is the current instance **this** (see 8.11 and 8.18).

Both `source_t` and `dest_t` can include one or more dynamically sized data in any position (for example, a structure containing a dynamic array followed by a queue of bytes). If the source type, `source_t`, includes dynamically sized variables, they are all included in the bit stream. If the destination type, `dest_t`, includes unbounded dynamically sized types, the conversion process is greedy: compute the size of the `source_t`, subtract the size of the fixed-size data items in the destination, and then adjust the size of the first dynamically sized item in the destination to the remaining size; any remaining dynamically sized items are left empty.

For the purposes of a bit-stream cast, a string type is considered a dynamic array of bytes.

Regardless of whether the destination type contains only fixed-size items or dynamically sized items, data are extracted into the destination in left-to-right order. It is thus legal to fill a dynamically sized item with data extracted from the middle of the packed representation.

If both `source_t` and `dest_t` are fixed-size types of different sizes and either type is unpacked, then a cast generates a compile-time error. If `source_t` or `dest_t` contain dynamically sized types, then a difference in their sizes will issue an error either at compile time or at run time, as soon as it is possible to determine the size mismatch. For example:

```
// Illegal conversion from 24-bit struct to 32 bit int - compile time error
struct {bit[7:0] a; shortint b;} a;
int b = int'(a);

// Illegal conversion from 20-bit struct to int (32 bits) - run time error
struct {bit a[$]; shortint b;} a = {{1,2,3,4}, 67};
int b = int'(a);

// Illegal conversion from int (32 bits) to struct dest_t (25 or 33 bits),
// compile time error
typedef struct {byte a[$]; bit b;} dest_t;
int a;
dest_t b = dest_t'(a);
```

Bit-stream casting can be used to convert between different aggregate types, such as two structure types, or a structure and an array or queue type. This conversion can be useful to model packet data transmission over serial communication streams. For example, the following code uses bit-stream casting to model a control packet transfer over a data stream:

```
typedef struct {
   shortint address;
   logic [3:0] code;
   byte command [2];
} Control;

typedef bit Bits [36:1];

Control p;
Bits stream[$];

p = ...                       // initialize control packet
stream.push_back(Bits'(p));   // append packet to unpacked queue of Bits
```

```
Bits b;
Control q;
b = stream.pop_front();       // get packet (as Bits) from stream
q = Control'(b);              // convert packet bits back to a Control packet
```

The following example uses bit-stream casting to model a data packet transfer over a byte stream:

```
typedef struct {
   byte length;
   shortint address;
   byte payload[];
   byte chksum;
} Packet;
```

The preceding type defines a generic data packet in which the size of the payload field is stored in the length field. Following is a function that randomly initializes the packet and computes the checksum.

```
function Packet genPkt();
   Packet p;

   void'( randomize( p.address, p.length, p.payload )
      with { p.length > 1 && p.payload.size == p.length; } );
   p.chksum = p.payload.xor();
   return p;
endfunction
```

The byte stream is modeled using a queue, and a bit-stream cast is used to send the packet over the stream.

```
typedef byte channel_type[$];
channel_type channel;
channel = {channel, channel_type'(genPkt())};
```

And the code to receive the packet:

```
Packet p;
int size;

size = channel[0] + 4;
p = Packet'( channel[0 : size - 1] );     // convert stream to Packet
channel = channel[ size : $ ];            // update the stream so it now
                                          // lacks that packet
```

## 6.25 Parameterized data types

SystemVerilog provides a way to create parameterized data types. A parameterized data type allows the user to generically define a data type and then conveniently create many varieties of that data type. When using such a data type one may provide the parameters that fully define its sets of values and operations. This allows for only one definition to be written and maintained instead of multiple definitions.

Parameterized data types are implemented through the use of type definitions in parameterized classes (see 8.25). The following example shows how to use type definitions and class parameterization to implement parameterized data types. The example has one class with three data types. The class may be declared **virtual** in order to prevent object construction and enforce its usage only for data type definition.

```
virtual class C#(parameter type T = logic, parameter SIZE = 1);
   typedef logic [SIZE-1:0] t_vector;
   typedef T t_array [SIZE-1:0];
   typedef struct {
      t_vector m0 [2*SIZE-1:0];
      t_array m1;
   } t_struct;
endclass
```

Class `C` contains three data types: `t_vector`, `t_array`, and `t_struct`. Each data type is parameterized by reusing the class parameters `T` and `SIZE`.

```
module top ();
   typedef logic [7:0] t_t0;
   C#(t_t0,3)::t_vector v0;
   C#(t_t0,3)::t_array a0;
   C#(bit,4)::t_struct s0;
endmodule
```

The top level module first defines a data type `t_t0`. Data type `t_t0` and the constant 3 are then used to declare variable `v0`. The number of bits in variable `t_vector` is determined by the specialized class parameter value of 3. Data type `t_vector` is referenced inside class `C` using the static class scope resolution operator `::` (see 8.23). Similarly for variable `a0`, the specialized class parameter values of `t_t0` and 3, declare `a0` as an unpacked array of 3 elements of type `t_t0`. Finally, variable `s0` is declared as an unpacked struct whose member data types are defined through the values of specialized class parameter values **`bit`** and 4.

# 7. Aggregate data types

## 7.1 General

This clause describes the following:

— Structure definitions and usage
— Union definitions and usage
— Packed arrays, unpacked arrays, dynamic arrays, associative arrays, and queues
— Array query and manipulation methods

## 7.2 Structures

A *structure* represents a collection of data types that can be referenced as a whole, or the individual data types that make up the structure can be referenced by name. By default, structures are unpacked, meaning that there is an implementation-dependent packing of the data types. Unpacked structures can contain any data type.

Structure declarations follow the C syntax, but without the optional structure tags before the "`{`". The syntax for structure declarations is shown in Syntax 7-1.

```
data_type ::=                                                          // from A.2.2.1
    ...
  | struct_union [ packed [ signing ] ] { struct_union_member { struct_union_member } }
        { packed_dimension }13
struct_union_member16 ::=
     { attribute_instance } [random_qualifier] data_type_or_void  list_of_variable_decl_assignments ;
data_type_or_void ::= data_type | void
struct_union ::= struct | union [ tagged ]
```

13) When a packed dimension is used with the **struct** or **union** keyword, the **packed** keyword shall also be used.

16) It shall be legal to declare a **void** struct_union_member only within tagged unions. It shall be legal to declare a random_qualifier only within unpacked structures.

*Syntax 7-1—Structure declaration syntax (excerpt from Annex A)*

Examples of declaring structures are as follows:

```
struct { bit [7:0] opcode; bit [23:0] addr; }IR;   // anonymous structure
                                                   // defines variable IR
IR.opcode = 1;   // set field in IR.

typedef struct {
            bit [7:0] opcode;
            bit [23:0] addr;
} instruction; // named structure type
instruction IR; // define variable
```

### 7.2.1 Packed structures

A packed structure is a mechanism for subdividing a vector into subfields, which can be conveniently accessed as members. Consequently, a packed structure consists of bit fields, which are packed together in memory without gaps. An unpacked structure has an implementation-dependent packing, normally matching the C compiler. A packed structure differs from an unpacked structure in that, when a packed structure appears as a primary, it shall be treated as a single vector.

A packed structure can also be used as a whole with arithmetic and logical operators, and its behavior is determined by its signedness, with unsigned being the default. The first member specified is the most significant and subsequent members follow in decreasing significance.

```
struct packed signed {
   int a;
   shortint b;
   byte c;
   bit [7:0] d;
} pack1; // signed, 2-state

struct packed unsigned {
   time a;
   integer b;
   logic [31:0] c;
} pack2; // unsigned, 4-state
```

The signing of unpacked structures is not allowed. The following declaration would be considered illegal:

```
typedef struct signed {
   int f1 ;
   logic f2 ;
} sIllegalSignedUnpackedStructType;    // illegal declaration
```

If all data types within a packed structure are 2-state, the structure as a whole is treated as a 2-state vector.

If any data type within a packed structure is 4-state, the structure as a whole is treated as a 4-state vector. If there are also 2-state members in the structure, there is an implicit conversion from 4-state to 2-state when reading those members and from 2-state to 4-state when writing them.

One or more bits of a packed structure can be selected as if it were a packed array with the range `[n-1:0]`:

```
pack1 [15:8] // c
```

Only packed data types and the integer data types summarized in Table 6-8 (see 6.11) shall be legal in packed structures.

A packed structure can be used with a **`typedef`**.

```
typedef struct packed { // default unsigned
   bit [3:0] GFC;
   bit [7:0] VPI;
   bit [11:0] VCI;
   bit CLP;
   bit [3:0] PT ;
   bit [7:0] HEC;
   bit [47:0] [7:0] Payload;
   bit [2:0] filler;
} s_atmcell;
```

### 7.2.2 Assigning to structures

A structure can be assigned as a whole and passed to or from a subroutine as a whole.

Members of a structure data type can be assigned individual default member values by using an initial assignment with the declaration of each member. The assigned expression shall be a constant expression.

An example of initializing members of a structure type is as follows:

```
typedef struct {
   int addr = 1 + constant;
   int crc;
   byte data [4] = '{4{1}};
} packet1;
```

The structure can then be instantiated.

```
packet1 p1;     // initialization defined by the typedef.
                // p1.crc will use the default value for an int
```

If an explicit initial value expression is used with the declaration of a variable, the initial assignment expression within the structure data type shall be ignored. Subclause 5.10 discusses assigning initial values to a structure. For example:

```
packet1 pi = '{1,2,'{2,3,4,5}}; //suppresses the typedef initialization
```

Members of unpacked structures containing a union as well as members of packed structures shall not be assigned individual default member values.

The initial assignment expression within a data type shall be ignored when using a data type to declare a net that does not have a user-defined **nettype** (see 6.7.1).

## 7.3 Unions

A *union* is a data type that represents a single piece of storage that can be accessed using one of the named member data types. Only one of the data types in the union can be used at a time. By default, a union is unpacked, meaning there is no required representation for how members of the union are stored. Dynamic types and chandle types can only be used in tagged unions.

The syntax for union declarations is shown in Syntax 7-2.

```
data_type ::=                                                          // from A.2.2.1
   ...
  | struct_union [ packed [ signing ] ] { struct_union_member { struct_union_member } }
       { packed_dimension }[13]
struct_union_member[16] ::=
   { attribute_instance } [random_qualifier] data_type_or_void list_of_variable_decl_assignments ;
  data_type_or_void ::= data_type | void
```

struct_union ::= **struct** | **union** [ **tagged** ]

13) When a packed dimension is used with the **struct** or **union** keyword, the **packed** keyword shall also be used.

16) It shall be legal to declare a **void** struct_union_member only within tagged unions. It shall be legal to declare a random_qualifier only within unpacked structures.

*Syntax 7-2—Union declaration syntax (excerpt from Annex A)*

*Examples:*

```
typedef union { int i; shortreal f; } num;         // named union type
num n;
n.f = 0.0; // set n in floating point format

typedef struct {
            bit isfloat;
            union { int i; shortreal f; } n;       // anonymous union type
} tagged_st;                                       // named structure
```

If no initial value is specified in the declaration of a variable of an unpacked union type, then the variable shall be initialized to the default initial value for variables of the type of the first member in declaration order of the union type.

One special provision exists in order to simplify the use of unpacked unions: if an unpacked union contains several unpacked structures that share a common initial sequence and if the unpacked union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the complete type of the union is visible. Two structures share a common initial sequence if corresponding members have equivalent types for a sequence of one or more initial members.

### 7.3.1 Packed unions

Packed unions shall only contain members that are of integral data types. The members of a packed, untagged union shall all be the same size (in contrast to an unpacked union or a packed, tagged union, where the members can be different sizes). Thus, a union member that was written as another member can be read back. A packed union differs from an unpacked union in that when a packed union appears as a *primary*, it shall be treated as a single vector.

A packed union can also be used as a whole with arithmetic and logical operators, and its behavior is determined by its signedness, with unsigned being the default. One or more bits of a packed union can be selected as if it were a packed array with the range `[n-1:0]`.

Only packed data types and the integer data types summarized in Table 6-8 (see 6.11) shall be legal in packed unions.

If a packed union contains a 2-state member and a 4-state member, the entire union is 4-state. There is an implicit conversion from 4-state to 2-state when reading and from 2-state to 4-state when writing the 2-state member.

For example, a union can be accessible with different access widths:

```
typedef union packed { // default unsigned
   s_atmcell acell;
   bit [423:0] bit_slice;
   bit [52:0][7:0] byte_slice;
} u_atmcell;

u_atmcell u1;
byte b; bit [3:0] nib;
b = u1.bit_slice[415:408]; // same as b = u1.byte_slice[51];
nib = u1.bit_slice [423:420]; // same as nib = u1.acell.GFC;
```

With packed unions, writing one member and reading another is independent of the byte ordering of the machine, unlike an unpacked union of unpacked structures, which are C-compatible and have members in ascending address order.

### 7.3.2 Tagged unions

The qualifier **tagged** in a union declares it as a *tagged union*, which is a type-checked union. An ordinary (untagged) union can be updated using a value of one member type and read as a value of another member type, which is a potential type loophole. A tagged union stores both the member value and a *tag*, i.e., additional bits representing the current member name. The tag and value can only be updated together consistently using a statically type-checked tagged union expression (see 11.9). The member value can only be read with a type that is consistent with the current tag value (i.e., member name). Thus, it is impossible to store a value of one type and (mis)interpret the bits as another type.

Dynamic types and chandle types shall not be used in untagged unions, but may be used in tagged unions.

Members of tagged unions can be referenced as tagged expressions. See 11.9.

In addition to type safety, the use of member names as tags also makes code simpler and smaller than code that has to track unions with explicit tags. Tagged unions can also be used with pattern matching (see 12.6), which improves readability even further.

In tagged unions, members can be declared with type **void**, when all the information is in the tag itself, as in the following example of an integer together with a valid bit:

```
typedef union tagged {
   void Invalid;
   int Valid;
} VInt;
```

A value of `VInt` type is either `Invalid` (and contains nothing) or `Valid` (and contains an **int**). Subclause 11.9 describes how to construct values of this type and also describes how it is impossible to read an integer out of a `VInt` value that currently has the `Invalid` tag.

For example:

```
typedef union tagged {
   struct {
      bit [4:0] reg1, reg2, regd;
   } Add;
   union tagged {
      bit [9:0] JmpU;
      struct {
         bit [1:0] cc;
         bit [9:0] addr;
      } JmpC;
   } Jmp;
} Instr;
```

A value of `Instr` type is either an `Add` instruction, in which case it contains three 5-bit register fields, or it is a `Jmp` instruction. In the latter case, it is either an unconditional jump, in which case it contains a 10-bit destination address, or it is a conditional jump, in which case it contains a 2-bit condition-code register field and a 10-bit destination address. Subclause 11.9 describes how to construct values of `Instr` type and describes how, in order to read the `cc` field, for example, the instruction must have opcode `Jmp` and sub-opcode `JmpC`.

When the **`packed`** qualifier is used on a tagged union, all the members shall have packed types, but they do not have to be of the same size. The (standard) representation for a packed tagged union is the following:

- The size is always equal to the number of bits needed to represent the tag plus the maximum of the sizes of the members.
- The size of the tag is the minimum number of bits needed to code for all the member names (e.g., five to eight members would need 3 tag bits).
- The tag bits are always left-justified (i.e., towards the MSBs).
- For each member, the member bits are always right-justified [i.e., towards the least significant bits (LSBs)].
- The bits between the tag bits and the member bits are undefined. In the extreme case of a void member, only the tag is significant and all the remaining bits are undefined.

The representation scheme is applied recursively to any nested tagged unions.

For example, if the `VInt` type definition had the **`packed`** qualifier, `Invalid` and `Valid` values will have the layouts shown in Figure 7-1, respectively.

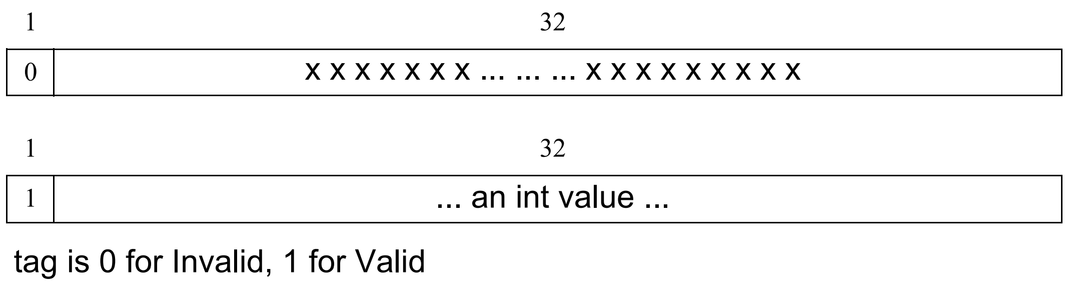

1
32
0
X X X X X X X ... ... ... X X X X X X X X X
1
32
1
... an int value ...
tag is 0 for Invalid, 1 for Valid

**Figure 7-1—VInt type with packed qualifier**

For example, if the `Instr` type had the **`packed`** qualifier, its values will have the layouts shown in Figure 7-2

.

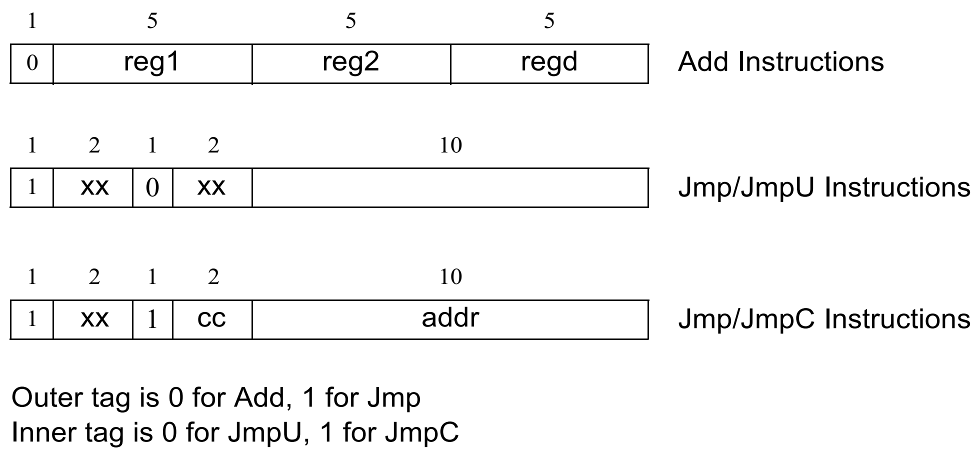


**Figure 7-2—Instr type with packed qualifier**

## 7.4 Packed and unpacked arrays

SystemVerilog supports both packed arrays and unpacked arrays of data. The term *packed array* is used to refer to the dimensions declared before the data identifier name. The term *unpacked array* is used to refer to the dimensions declared after the data identifier name.

```
bit [7:0] c1;         // packed array of scalar bit types
real u [7:0];         // unpacked array of real types
```

A one-dimensional packed array is often referred to as a *vector* (see 6.9). Multidimensional packed arrays are also allowed.

Unpacked arrays may be fixed-size arrays (see 7.4.2), dynamic arrays (see 7.5), associative arrays (see 7.8), or queues (see 7.10). Unpacked arrays are formed from any data type, including other packed or unpacked arrays (see 7.4.5).

### 7.4.1 Packed arrays

A packed array is a mechanism for subdividing a vector into subfields, which can be conveniently accessed as array elements. Consequently, a packed array is guaranteed to be represented as a contiguous set of bits. An unpacked array may or may not be so represented. A packed array differs from an unpacked array in that, when a packed array appears as a primary, it is treated as a single vector.

If a packed array is declared as signed, then the array viewed as a single vector shall be signed. The individual elements of the array are unsigned unless they are of a named type declared as signed. A part-select of a packed array shall be unsigned.

Packed arrays allow arbitrary length integer types; therefore, a 48-bit integer can be made up of 48 bits. These integers can then be used for 48-bit arithmetic. The maximum size of a packed array can be limited, but shall be at least 65 536 ($2^{16}$) bits.

Packed arrays can be made of only the single bit data types (**bit**, **logic**, **reg**), enumerated types, and recursively other packed arrays and packed structures.

Integer types with predefined widths shall not have packed array dimensions declared. These types are **byte**, **shortint**, **int**, **longint**, **integer**, and **time**. Although an integer type with a predefined width `n` is not a packed array, it matches (see 6.22), and can be selected from as if it were, a packed array type with a single `[n-1:0]` dimension.

```
byte c2;    // same as bit signed [7:0] c2;
integer i1; // same as logic signed [31:0] i1;
```

### 7.4.2 Unpacked arrays

Unpacked arrays can be made of any data type. Arrays whose elements are themselves arrays are declared as multidimensional arrays (see 7.4.5). Unpacked arrays shall be declared by specifying the element address range(s) after the declared identifier.

Elements of net arrays can be used in the same fashion as a scalar or vector net. Net arrays are useful for connecting to ports of module instances inside loop generate constructs (see 27.4).

Each fixed-size dimension shall be represented by an address range, such as `[1:1024]`, or a single positive number to specify the size of a fixed-size unpacked array, as in C. In other words, `[size]` becomes the same as `[0:size-1]`.

The following examples declare equivalent size two-dimensional fixed-size arrays of **int** variables:

```
int Array[0:7][0:31];   // array declaration using ranges

int Array[8][32];       // array declaration using sizes
```

The expressions that specify an address range shall be constant integer expressions. The value of the constant expression can be a positive integer, a negative integer, or zero. It shall be illegal for them to contain any unknown (x) or high-impedance bits.

Implementations may limit the maximum size of an array, but they shall allow at least 16 777 216 ($2^{24}$) elements.

### 7.4.3 Operations on arrays

The following operations can be performed on all arrays, packed or unpacked. The examples provided with these rules assume that `A` and `B` are arrays of the same shape and type.

— Reading and writing the array, e.g., `A = B`
— Reading and writing a slice of the array, e.g., `A[i:j] = B[i:j]`
— Reading and writing a variable slice of the array, e.g., `A[x+:c] = B[y+:c]`
— Reading and writing an element of the array, e.g., `A[i] = B[i]`
— Equality operations on the array or slice of the array, e.g., `A==B, A[i:j] != B[i:j]`

The following operations can be performed on packed arrays, but not on unpacked arrays. The examples provided with these rules assume that `A` is an array.

— Assignment from an integer, e.g., `A = 8'b11111111;`
— Treatment as an integer in an expression, e.g., `(A + 3)`

If an unpacked array is declared as signed, then this applies to the individual elements of the array because the whole array cannot be viewed as a single vector.

See 7.6 for rules for assigning to packed and unpacked arrays.

### 7.4.4 Memories

A one-dimensional array with elements of types **reg**, **logic**, or **bit** is also called a *memory*. Memory arrays can be used to model read-only memories (ROMs), random access memories (RAMs), and register files. An element of the packed dimension in the array is known as a memory *element* or *word*.

```
logic [7:0] mema [0:255];  // declares a memory array of 256 8-bit
                           // elements. The array indices are 0 to 255

mema[5] = 0;               // Write to word at address 5

data = mema[addr];         // Read word at address indexed by addr
```

### 7.4.5 Multidimensional arrays

A *multidimensional array* is an array of arrays. Multidimensional arrays can be declared by including multiple dimensions in a single declaration. The dimensions preceding the identifier set the packed dimensions. The dimensions following the identifier set the unpacked dimensions.

```
bit [3:0] [7:0] joe [1:10];   // 10 elements of 4 8-bit bytes
                              // (each element packed into 32 bits)
```

can be used as follows:

```
joe[9] = joe[8] + 1; // 4 byte add
joe[7][3:2] = joe[6][1:0]; // 2 byte copy
```

In a multidimensional declaration, the dimensions declared following the type and before the name (`[3:0][7:0]` in the preceding declaration) vary more rapidly than the dimensions following the name (`[1:10]` in the preceding declaration). When referenced, the packed dimensions (`[3:0], [7:0]`) follow the unpacked dimensions (`[1:10]`).

In a list of dimensions, the rightmost one varies most rapidly, as in C. However, a packed dimension varies more rapidly than an unpacked one.

```
bit [1:10] v1 [1:5];    // 1 to 10 varies most rapidly; compatible with
                           memory arrays

bit v2 [1:5] [1:10];    // 1 to 10 varies most rapidly, compatible with C

bit [1:5] [1:10] v3 ;   // 1 to 10 varies most rapidly

bit [1:5] [1:6] v4 [1:7] [1:8];  // 1 to 6 varies most rapidly, followed by
                                   // 1 to 5, then 1 to 8 and then 1 to 7
```

Multiple packed dimensions can also be defined in stages with **typedef**.

```
typedef bit [1:5] bsix;
bsix [1:10] v5; // 1 to 5 varies most rapidly
```

Multiple unpacked dimensions can also be defined in stages with **typedef**.

```
typedef bsix mem_type [0:3];  // array of four 'bsix' elements
mem_type ba [0:7];            // array of eight 'mem_type' elements
```

A *subarray* is an array that is an element of another array. As in the C language, subarrays are referenced by omitting indices for one or more array dimensions, always omitting the ones that vary most rapidly. Omitting indices for all the dimensions references the entire array.

```
int A[2][3][4], B[2][3][4], C[5][4];
...
A[0][2] = B[1][1];   // assign a subarray composed of four ints
A[1] = B[0];         // assign a subarray composed of three arrays of
                     // four ints each
A = B;               // assign an entire array
A[0][1] = C[4];      // assign compatible subarray of four ints
```

A comma-separated list of array declarations can be specified. All arrays in the list shall have the same data type and the same packed array dimensions.

```
bit [7:0] [31:0] v7 [1:5] [1:10], v8 [0:255];      // two arrays declared
```

### 7.4.6 Indexing and slicing of arrays

An expression can select part of a packed array, or any integer type, which is assumed to be numbered down to 0.

The term *part-select* refers to a selection of one or more contiguous bits of a single-dimension packed array.

```
logic [63:0] data;
logic [7:0] byte2;
byte2 = data[23:16]; // an 8-bit part-select from data
```

The term *slice* refers to a selection of one or more contiguous elements of an array.

NOTE—IEEE Std 1364-2005 only permitted a single element of an array to be selected.

A single element of a packed or unpacked array can be selected using an indexed name.

```
bit [3:0] [7:0] j;   // j is a packed array
byte k;
k = j[2]; // select a single 8-bit element from j
```

One or more contiguous elements can be selected using a slice name. A slice name of a packed array is a packed array. A slice name of an unpacked array is an unpacked array.

```
bit signed [31:0] busA [7:0] ;   // unpacked array of 8 32-bit vectors
int busB [1:0];                  // unpacked array of 2 integers
busB = busA[7:6];                // select a 2-vector slice from busA
```

The size of the part-select or slice shall be constant, but the position can be variable.

```
int i = bitvec[j +: k];    // k must be constant.
int a[x:y], b[y:z], e;
a = {b[c -: d], e};        // d must be constant
```

Slices of an array can only apply to one dimension, but other dimensions can have single index values in an expression.

If an index expression is out of bounds or if any bit in the index expression is `x` or `z`, then the index shall be invalid. Reading from an unpacked array of any kind with an invalid index shall return the value specified in Table 7-1. Writing to an array with an invalid index shall perform no operation, with the exceptions of writing to element `[$+1]` of a queue (described in 7.10.1) and creating a new element of an associative array (described in 7.8.6). Implementations may issue a warning if an invalid index occurs for a read or write operation on an array.

Access to a packed array with an invalid index is described in 11.5.1.

See 11.5.1 and 11.5.2 for more information on vector and array element selecting and slicing.

**Table 7-1—Value read from a nonexistent array entry**

| Type of array | Value read |
|---|---|
| 4-state integral type | `'X` |
| 2-state integral type | `'0` |
| Enumeration | Value specified in this table for the enumeration's base type |
| **real**, **shortreal** | `0.0` |
| **string** | `""` |
| **class** | **null** |
| **interface class** | **null** |
| **event** | **null** |
| **chandle** | **null** |
| **virtual interface** | **null** |
| Variable-size unpacked array (dynamic, queue or associative) | Array of size zero (no elements) |
| Fixed-size unpacked array | Array, all of whose elements have the value specified in this table for that array's element type |
| Unpacked **struct** | **struct**, each of whose members has the value specified in this table for that member's type, unless the member has an initial assignment as part of its declaration (see 7.2.2), in which case the member's value shall be as given by its initial assignment |
| Unpacked **union** | Value specified in this table for the type of the first member of the **union** |

## 7.5 Dynamic arrays

A dynamic array is an unpacked array whose size can be set or changed at run time. The default size of an uninitialized dynamic array is zero. The size of a dynamic array is set by the **new** constructor or array assignment, described in 7.5.1 and 7.6, respectively. Dynamic arrays support all variable data types as element types, including arrays.

Dynamic array dimensions are denoted in the array declaration by `[ ]`. Any unpacked dimension in an array declaration may be a dynamic array dimension.

For example:

```
bit [3:0] nibble[];     // Dynamic array of 4-bit vectors
```

```
integer mem[2][];       // Fixed-size unpacked array composed
                        // of 2 dynamic subarrays of integers
```

Note that in order for an identifier to represent a dynamic array, it must be declared with a dynamic array dimension as the leftmost unpacked dimension.

The **new**`[]` constructor is used to set or change the size of the array and initialize its elements (see 7.5.1).

The `size()` built-in method returns the current size of the array (see 7.5.2).

The `delete()` built-in method clears all the elements yielding an empty array (zero size) (see 7.5.3).

### 7.5.1 New[ ]

The **new** constructor sets the size of a dynamic array and initializes its elements. It may appear in place of the right-hand side expression of variable declaration assignments and blocking procedural assignments when the left-hand side indicates a dynamic array.

```
blocking_assignment ::=                                              // from A.6.2
    ...
  | nonrange_variable_lvalue = dynamic_array_new
    ...
dynamic_array_new ::= new [ expression ] [ ( expression ) ]          // from A.2.4
```

*Syntax 7-3—Dynamic array new constructor syntax (excerpt from Annex A)*

**[** expression **]**:

The desired size of the dynamic array. The type of this operand is **longint**. It shall be an error if the value of this operand is negative. If this operand is zero, the array shall become empty.

**(** expression **)**:

Optional. An array with which to initialize the dynamic array.

The **new** constructor follows the SystemVerilog precedence rules. Because both the square brackets `[]` and the parenthesis `()` have the same precedence, the arguments to the **new** constructor are evaluated left to right: **[** expression **]** first, and **(** expression **)** second.

Dynamic array declarations may include a declaration assignment with the **new** constructor as the right-hand side:

```
int arr1 [][2][3] = new [4]; // arr1 sized to length 4; elements are
                             // fixed-size arrays and so do not require
                             // initializing

int arr2 [][] = new [4];     // arr2 sized to length 4; dynamic subarrays
                             // remain unsized and uninitialized

int arr3 [1][2][] = new [4]; // Error – arr3 is not a dynamic array, though
                             // it contains dynamic subarrays
```

Dynamic arrays may be initialized in procedural contexts using the **new** constructor in blocking assignments:

```
int arr[2][][];
arr[0] = new [4];       // dynamic subarray arr[0] sized to length 4

arr[0][0] = new [2];    // legal, arr[0][n] created above for n = 0..3

arr[1][0] = new [2];    // illegal, arr[1] not initialized so arr[1][0] does
                        // not exist

arr[0][] = new [2];     // illegal, syntax error - dimension without
                        // subscript on left hand side

arr[0][1][1] = new[2];  // illegal, arr[0][1][1] is an int, not a dynamic
                        // array
```

In either case, if the **new** constructor call does not specify an initialization expression, the elements are initialized to the default value for their type.

The optional initialization expression is used to initialize the dynamic array. When present, it shall be an array that is assignment compatible with the left-hand-side dynamic array.

```
int idest[], isrc[3] = '{5, 6, 7};
idest = new [3] (isrc); // set size and array element data values (5, 6, 7)
```

The size argument need not match the size of the initialization array. When the initialization array's size is greater, it is truncated to match the size argument; when it is smaller, the initialized array is padded with default values to attain the specified size.

```
int src[3], dest1[], dest2[];
src = '{2, 3, 4};
dest1 = new[2] (src);   // dest1's elements are {2, 3}.
dest2 = new[4] (src);   // dest2's elements are {2, 3, 4, 0}.
```

This behavior provides a mechanism for resizing a dynamic array while preserving its contents. An existing dynamic array can be resized by using it both as the left-hand side term and the initialization expression.

```
integer addr[];   // Declare the dynamic array.
addr = new[100];  // Create a 100-element array.
...
// Double the array size, preserving previous values.
// Preexisting references to elements of addr are outdated.
addr = new[200](addr);
```

Resizing or reinitializing a previously initialized dynamic array using **new** is destructive; no preexisting array data is preserved (unless reinitialized with its old contents—see preceding), and all preexisting references to array elements become outdated.

### 7.5.2 Size()

The prototype for the `size()` method is as follows:

```
function int size();
```

The `size()` method returns the current size of a dynamic array or returns zero if the array has not been created.

```
int j = addr.size;
addr = new[ addr.size() * 4 ] (addr);  // quadruple addr array
```

The `size` dynamic array method is equivalent to `$size( addr, 1 )` array query system function (see 20.7).

### 7.5.3 Delete()

The prototype for the `delete()` method is as follows:

```
function void delete();
```

The `delete()` method empties the array, resulting in a zero-sized array.

```
int ab [] = new[ N ];          // create a temporary array of size N
// use ab
ab.delete;                     // delete the array contents
$display( "%d", ab.size );     // prints 0
```

## 7.6 Array assignments

For the purposes of assignment, a packed array is treated as a vector. Any vector expression can be assigned to any packed array. The packed array bounds of the target packed array do not affect the assignment. A packed array cannot be directly assigned to an unpacked array without an explicit cast.

Associative arrays are assignment compatible only with associative arrays, as described in 7.9.9. A fixed-size unpacked array, dynamic array, or queue, or a slice of such an array, shall be assignment compatible with any other such array or slice if all the following conditions are satisfied:

- The element types of source and target shall be equivalent.
- If the target is a fixed-size array or a slice, the source array shall have the same number of elements as the target.

Here *element* refers to elements of the slowest-varying array dimension. These elements may themselves be of some unpacked array type. Consequently, for two arrays to be assignment compatible it is necessary (but not sufficient) that they have the same number of unpacked dimensions. Assignment compatibility of unpacked arrays is a weaker condition than type equivalence because it does not require their slowest-varying dimensions to be of the same unpacked array kind (queue, dynamic, or fixed-size). This weaker condition applies only to the slowest-varying dimension. Any faster-varying dimensions must meet the requirements for equivalence (see 6.22.2) for the entire arrays to be assignment compatible.

Assignment shall be done by assigning each element of the source array to the corresponding element of the target array. Correspondence between elements is determined by the left-to-right order of elements in each array. For example, if array `A` is declared as `int A[7:0]` and array `B` is declared as `int B[1:8]`, the assignment `A = B;` will assign element `B[1]` to element `A[7]`, and so on. If the target of the assignment is a queue or dynamic array, it shall be resized to have the same number of elements as the source expression and assignment shall then follow the same left-to-right element correspondence as previously described.

```
int A[10:1];      // fixed-size array of 10 elements
int B[0:9];       // fixed-size array of 10 elements
int C[24:1];      // fixed-size array of 24 elements

A = B;            // ok. Compatible type and same size
A = C;            // type check error: different sizes
```

An array of wires can be assigned to an array of variables, and vice versa, if the source and target arrays' data types are assignment compatible.

```
logic [7:0] V1[10:1];
logic [7:0] V2[10];
wire [7:0] W[9:0];      // data type is logic [7:0] W[9:0]
assign W = V1;
initial #10 V2 = W;
```

When a dynamic array or queue is assigned to a fixed-size array, the size of the source array cannot be determined until run time. An attempt to copy a dynamic array or queue into a fixed-size array target having a different number of elements shall result in a run-time error and no operation shall be performed. Example code showing assignment of a dynamic array to a fixed-size array follows.

```
int A[2][100:1];
int B[] = new[100];     // dynamic array of 100 elements
int C[] = new[8];       // dynamic array of 8 elements
int D [3][][];          // multidimensional array with dynamic subarrays
D[2] = new [2];         // initialize one of D's dynamic subarrays
D[2][0] = new [100];

A[1] = B;               // OK. Both are arrays of 100 ints
A[1] = C;               // type check error: different sizes (100 vs. 8 ints)
A = D[2];               // A[0:1][100:1] and subarray D[2][0:1][0:99] both
                        // comprise 2 subarrays of 100 ints
```

Examples showing assignment to a dynamic array follow. (See 7.5.1 for additional assignment examples involving the dynamic array **new** constructor).

```
int A[100:1];           // fixed-size array of 100 elements
int B[];                // empty dynamic array
int C[] = new[8];       // dynamic array of size 8

B = A;                  // ok. B has 100 elements
B = C;                  // ok. B has 8 elements
```

The previous last statement is equivalent to:

```
B = new[ C.size ] (C);
```

Similarly, the source of an assignment can be a complex expression involving array slices or concatenations. For example:

```
string d[1:5] = '{ "a", "b", "c", "d", "e" };
string p[];
p = { d[1:3], "hello", d[4:5] };
```

The preceding example creates the dynamic array `p` with contents `"a"`, `"b"`, `"c"`, `"hello"`, `"d"`, `"e"`.

## 7.7 Arrays as arguments to subroutines

Arrays can be passed as arguments to subroutines. The rules that govern array argument passing by value are the same as for array assignment (see 7.6). When an array argument is passed by value, a copy of the array is passed to the called subroutine. This is true for all array types: fixed-size, dynamic, queue, or associative.

The rules that govern whether an array actual argument can be associated with a given formal argument are the same as the rules for whether a source array's values can be assigned to a destination array (see 7.6). If a dimension of a formal is unsized (unsized dimensions can occur in dynamic arrays, queues, and formal arguments of import DPI functions), then it matches any size of the actual argument's corresponding dimension.

For example, the declaration

```
task fun(int a[3:1][3:1]);
```

declares task `fun` that takes one argument, a two-dimensional array with each dimension of size 3. A call to `fun` must pass a two-dimensional array and with the same dimension size 3 for all the dimensions. For example, given the preceding description for `fun`, consider the following actuals:

```
int b[3:1][3:1];     // OK: same type, dimension, and size

int b[1:3][0:2];     // OK: same type, dimension, & size (different ranges)

logic b[3:1][3:1];   // error: incompatible element type

event b[3:1][3:1];   // error: incompatible type

int b[3:1];          // error: incompatible number of dimensions

int b[3:1][4:1];     // error: incompatible size (3 vs. 4)
```

A subroutine that accepts a fixed-size array can also be passed a dynamic array or queue with compatible type and equal size.

For example, the declaration

```
task t( string arr[4:1] );
```

declares a task that accepts one argument, an array of 4 strings. This task can accept the following actual arguments:

```
string b[4:1];       // OK: same type and size
string b[5:2];       // OK: same type and size (different range)
string b[] = new[4]; // OK: same type, number of dimensions, and
                     // dimension size; requires run-time check
```

A subroutine that accepts a dynamic array or queue can be passed a dynamic array, queue, or fixed-size array of a compatible type.

For example, the declaration

```
task t ( string arr[] );
```

declares a task that accepts one argument, a dynamic array of strings. This task can accept any one-dimensional unpacked array of strings or any one-dimensional dynamic array or queue of strings.

The rules that govern dynamic array and queue formal arguments also govern the behavior of unpacked dimensions of DPI open array formal arguments (see 7.6). DPI open arrays can also have a solitary unsized, packed dimension (see 34.5.6.1). A dynamic array or queue shall not be passed as an actual argument if the DPI formal argument has unsized dimensions and an output direction mode.

## 7.8 Associative arrays

Dynamic arrays are useful for dealing with contiguous collections of variables whose number changes dynamically. When the size of the collection is unknown or the data space is sparse, an associative array is a better option. Associative arrays do not have any storage allocated until it is used, and the index expression is not restricted to integral expressions, but can be of any type.

An associative array implements a lookup table of the elements of its declared type. The data type to be used as an index serves as the lookup key and imposes an ordering.

The syntax to declare an associative array is as follows:

```
data_type array_id [ index_type ];
```

where

`data_type` is the data type of the array elements. Can be any type allowed for fixed-size arrays.
`array_id` is the name of the array being declared.
`index_type` is the data-type to be used as an index or is **`*`**. If **`*`** is specified, then the array is indexed by any integral expression of arbitrary size. An index type restricts the indexing expressions to a particular type. It shall be illegal for `index_type` to declare a type.

Examples of associative array declarations are as follows:

```
integer i_array[*];           // associative array of integer (unspecified
                              // index)

bit [20:0] array_b[string];   // associative array of 21-bit vector,
                              // indexed by string

event ev_array[myClass];      // associative array of event indexed by class
                              // myClass
```

Array elements in associative arrays are allocated dynamically. An entry for a nonexistent associative array element shall be allocated when it is used as the target of an assignment or actual to an argument passed by reference. The associative array maintains the entries that have been assigned values and their relative order according to the index data type. Associative array elements are unpacked. In other words, other than for copying or comparing arrays, an individual element must be selected out of the array before it can be used in most expressions.

### 7.8.1 Wildcard index type

For example:

```
int array_name [*];
```

Associative arrays that specify a wildcard index type have the following properties:

— The array may be indexed by any integral expression. Because the index expressions may be of different sizes, the same numerical value can have multiple representations, each of a different size. SystemVerilog resolves this ambiguity by removing the leading zeros and computing the minimal length and using that representation for the value.
— Nonintegral index values are illegal and result in an error.
— A 4-state index value containing `X` or `Z` is invalid.
— Indexing expressions are self-determined and treated as unsigned.

— A string literal index is automatically cast to a bit vector of equivalent size.
— The ordering is numerical (smallest to largest).
— Associative arrays that specify a wildcard index type shall not be used in a **foreach** loop (see 12.7.3) or with an array manipulation method (see 7.12) that returns an index value or array of values.

### 7.8.2 String index

For example:

```
int array_name [ string ];
```

Associative arrays that specify a string index have the following properties:

— Indices can be strings or string literals of any length. Other types are illegal and shall result in a type check error.
— An empty string `""` index is valid.
— The ordering is lexicographical (lesser to greater).

### 7.8.3 Class index

For example:

```
int array_name [ some_Class ];
```

Associative arrays that specify a class index have the following properties:

— Indices can be objects of that particular type or derived from that type. Any other type is illegal and shall result in a type check error.
— A null index is valid.
— The ordering is deterministic but arbitrary.

### 7.8.4 Integral index

For example:

```
int array_name1 [ integer ];
typedef bit signed [4:1] SNibble;
int array_name2 [ SNibble ];
typedef bit [4:1] UNibble;
int array_name3 [ UNibble ];
```

Associative arrays that specify an index of integral data type shall have the following properties:

— The index expression shall be evaluated in terms of a cast to the index type, except that an implicit cast from a real or shortreal data type shall be illegal.
— A 4-state index expression containing `X` or `Z` is invalid.
— The ordering is signed or unsigned numerical, depending on the signedness of the index type.

### 7.8.5 Other user-defined types

For example:

```
typedef struct {byte B; int I[*];} Unpkt;
int array_name [ Unpkt ];
```

In general, associative arrays that specify an index of any type have the following properties:

— Declared indices shall have the equality operator defined for its type to be legal. This includes all of the dynamically sized types as legal index types. However, **real** or **shortreal** data types, or a type containing a **real** or **shortreal**, shall be an illegal index type.
— An index expression that is or contains X or Z in any of its elements is invalid.
— An index expression that is or contains an empty value or **null** for any of its elements does not make the index invalid.
— If the relational operator is defined for the index type, the ordering is as defined in the preceding clauses. If not, the relative ordering of any two entries in such an associative array can vary, even between successive runs of the same tool. However, the relative ordering shall remain the same within the same simulation run while no indices have been added or deleted.

### 7.8.6 Accessing invalid indices

If a read operation uses an index that is a 4-state expression with one or more x or z bits, or an attempt is made to read a nonexistent entry, then a warning shall be issued and the nonexistent entry value for the array type shall be returned, as shown in Table 7-1 (see 7.4.6). A user-specified default shall not issue a warning and returns the value specified in 7.9.11.

If an invalid index is used during a write operation, the write shall be ignored, and a warning shall be issued.

### 7.8.7 Allocating associative array elements

An entry for a nonexistent associative array element shall be allocated when it is used as the target of an assignment or actual to an argument passed by reference. Some constructs perform both a read and write operation as part of a single statement, such as with an increment operation. In those cases, the nonexistent element shall be allocated with its default or user-specified initial value before any reference to that element. For example:

```
int a[int] = '{default:1};
typedef struct { int x=1,y=2; } xy_t;
xy_t b[int];

begin
   a[1]++;
   b[2].x = 5;
end
```

Assume the references to a[1] and b[2] are nonexistent elements before these statements execute. Upon executing a[1]++, a[1] will be allocated and initialized to 1. After the increment, a[1] will be 2. Upon executing b[2].x = 5, b[2] will be allocated and b[2].x will be 1 and b[2].y will be 2. After executing the assignment, b[2].x will be updated to 5.

## 7.9 Associative array methods

In addition to the indexing operators, several built-in methods are provided, which allow users to analyze and manipulate associative arrays, as well as iterate over its indices or keys.

### 7.9.1 Num() and size()

The syntax for the num() and size() methods is as follows:

```
function int num();
function int size();
```

The `num()` and `size()` methods return the number of entries in the associative array. If the array is empty, they return 0.

```
int imem[int];
imem[ 3 ] = 1;
imem[ 16'hffff ] = 2;
imem[ 4'b1000 ] = 3;
$display( "%0d entries\n", imem.num );    // prints "3 entries"
```

### 7.9.2 Delete()

The syntax for the `delete()` method is as follows:

```
function void delete( [input index] );
```

where `index` is an optional index of the appropriate type for the array in question.

If the `index` is specified, then the `delete()` method removes the entry at the specified index. If the entry to be deleted does not exist, the method issues no warning.

If the `index` is not specified, then the `delete()` method removes all the elements in the array.

```
int map[ string ];
map[ "hello" ] = 1;
map[ "sad" ] = 2;
map[ "world" ] = 3;
map.delete( "sad" );  // remove entry whose index is "sad" from "map"
map.delete;           // remove all entries from the associative array "map"
```

### 7.9.3 Exists()

The syntax for the `exists()` method is as follows:

```
function int exists( input index );
```

where `index` is an index of the appropriate type for the array in question.

The `exists()` function checks whether an element exists at the specified index within the given array. It returns 1 if the element exists; otherwise, it returns 0.

```
if ( map.exists( "hello" ))
   map[ "hello" ] += 1;
else
   map[ "hello" ] = 0;
```

### 7.9.4 First()

The syntax for the `first()` method is as follows:

```
function int first( ref index );
```

where `index` is an index of the appropriate type for the array in question. Associative arrays that specify a wildcard index type shall not be allowed.

The `first()` method assigns to the given index variable the value of the first (smallest) index in the associative array. It returns 0 if the array is empty; otherwise, it returns 1.

```
string s;
if ( map.first( s ) )
   $display( "First entry is : map[ %s ] = %0d\n", s, map[s] );
```

### 7.9.5 Last()

The syntax for the `last()` method is as follows:

```
function int last( ref index );
```

where `index` is an index of the appropriate type for the array in question. Associative arrays that specify a wildcard index type shall not be allowed.

The `last()` method assigns to the given index variable the value of the last (largest) index in the associative array. It returns 0 if the array is empty; otherwise, it returns 1.

```
string s;
if ( map.last( s ) )
   $display( "Last entry is : map[ %s ] = %0d\n", s, map[s] );
```

### 7.9.6 Next()

The syntax for the `next()` method is as follows:

```
function int next( ref index );
```

where `index` is an index of the appropriate type for the array in question. Associative arrays that specify a wildcard index type shall not be allowed.

The `next()` method finds the smallest index whose value is greater than the given index argument.

If there is a next entry, the index variable is assigned the index of the next entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0.

```
string s;
if ( map.first( s ) )
   do
      $display( "%s : %d\n", s, map[ s ] );
   while ( map.next( s ) );
```

### 7.9.7 Prev()

The syntax for the `prev()` method is as follows:

```
function int prev( ref index );
```

where `index` is an index of the appropriate type for the array in question. Associative arrays that specify a wildcard index type shall not be allowed.

The `prev()` function finds the largest index whose value is smaller than the given index argument. If there is a previous entry, the index variable is assigned the index of the previous entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0.

```
string s;
if ( map.last( s ) )
   do
      $display( "%s : %d\n", s, map[ s ] );
   while ( map.prev( s ) );
```

### 7.9.8 Arguments to traversal methods

The argument that is passed to any of the four associative array traversal methods `first()`, `last()`, `next()`, and `prev()` shall be assignment compatible with the index type of the array. If the argument has an integral type that is smaller than the size of the corresponding array index type, then the function returns –1 and shall truncate in order to fit into the argument. For example:

```
string   aa[int];
byte     ix;
int      status;
aa[ 1000 ] = "a";
status = aa.first( ix );
   // status is –1
   // ix is 232 (least significant 8 bits of 1000)
```

### 7.9.9 Associative array assignment

Associative arrays can be assigned only to another associative array of a compatible type and with the same index type. Other types of arrays cannot be assigned to an associative array, nor can associative arrays be assigned to other types of arrays, whether fixed-size or dynamic.

Assigning an associative array to another associative array causes the target array to be cleared of any existing entries, and then each entry in the source array is copied into the target array.

### 7.9.10 Associative array arguments

Associative arrays can be passed as arguments only to associative arrays of a compatible type and with the same index type. Other types of arrays, whether fixed-size or dynamic, cannot be passed to subroutines that accept an associative array as an argument. Likewise, associative arrays cannot be passed to subroutines that accept other types of arrays.

Passing an associative array by value causes a local copy of the associative array to be created.

### 7.9.11 Associative array literals

Associative array literals use the `'{index:value}` syntax with an optional default index. Like all other arrays, an associative array can be written one entry at a time, or the whole array contents can be replaced using an array literal.

For example:

```
// an associative array of strings indexed by 2-state integers,
// default is "hello".
string words [int] = '{default: "hello"};

// an associative array of 4-state integers indexed by strings, default is –1
integer tab [string] = '{"Peter":20, "Paul":22, "Mary":23, default:-1 };
```

If a default value is specified, then reading a nonexistent element shall yield the specified default value, and no warning shall be issued. Otherwise, the value specified by Table 7-1 (see 7.4.6) shall be returned.

Defining a default value shall not affect the operation of the associative array methods (see 7.9).

## 7.10 Queues

A queue is a variable-size, ordered collection of homogeneous elements. A queue supports constant-time access to all its elements as well as constant-time insertion and removal at the beginning or the end of the queue. Each element in a queue is identified by an ordinal number that represents its position within the queue, with 0 representing the first, and **$** representing the last. A queue is analogous to a one-dimensional unpacked array that grows and shrinks automatically. Thus, like arrays, queues can be manipulated using the indexing, concatenation, slicing operator syntax, and equality operators.

Queues are declared using the same syntax as unpacked arrays, but specifying **$** as the array size. The maximum size of a queue can be limited by specifying its optional right bound (last index).

Queue values may be written using assignment patterns or unpacked array concatenations (see 10.9, 10.10).

The syntax for declaring queues is shown in Syntax 7-4.

```
variable_dimension ::=                                              // from A.2.5
      unsized_dimension
    | unpacked_dimension
    | associative_dimension
    | queue_dimension
queue_dimension ::= [ $ [ : constant_expression ] ]
```

*Syntax 7-4—Declaration of queue dimension (excerpt from Annex A)*

*constant_expression* shall evaluate to a positive integer value.

For example:

```
byte q1[$];                    // A queue of bytes
string names[$] = { "Bob" };   // A queue of strings with one element
integer Q[$] = { 3, 2, 7 };    // An initialized queue of integers
bit q2[$:255];                 // A queue whose maximum size is 256 bits
```

If an initial value is not provided in the declaration, the queue variable is initialized to the empty queue. The empty queue can be denoted by an empty unpacked array concatenation `{}`, as described in 10.10.

### 7.10.1 Queue operators

Queues shall support the same operations that can be performed on unpacked arrays. In addition, queues shall support the following operations:

— A queue shall resize itself to accommodate any queue value that is written to it, except that its maximum size may be bounded as described in 7.10.

— In a queue slice expression such as `Q[a:b]`, the slice bounds may be arbitrary integral expressions and, in particular, are not required to be constant expressions.

— Queues shall support methods as described in 7.10.2.

Unlike arrays, the empty queue, `{}`, is a valid queue and the result of some queue operations. The following rules govern queue operators:

— `Q[ a : b ]` yields a queue with `b - a + 1` elements.
  - If `a > b`, then `Q[a:b]` yields the empty queue `{}`.
  - `Q[ n : n ]` yields a queue with one item, the one at position `n`. Thus, `Q[ n : n ] === { Q[n] }`.
  - If `n` lies outside `Q`'s range (`n < 0 or n > $`), then `Q[n:n]` yields the empty queue `{}`.
  - If either `a` or `b` are 4-state expressions containing `X` or `Z` values, it yields the empty queue `{}`.

— `Q[ a : b ]` where `a < 0` is the same as `Q[ 0 : b ]`.

— `Q[ a : b ]` where `b > $` is the same as `Q[ a : $ ]`.

— An invalid index value (i.e., a 4-state expression whose value has one or more `x` or `z` bits, or a value that lies outside 0...$) shall cause a read operation to return the value appropriate for a nonexistent array entry of the queue's element type (as described in Table 7-1 in 7.4.6).

— An invalid index (i.e., a 4-state expression with `X`'s or `Z`'s, or a value that lies outside 0...$+1) shall cause a write operation to be ignored and a run-time warning to be issued; however, writing to `Q[$+1]` is legal.

— A queue declared with a right bound using the syntax `[$:N]` is known as a *bounded queue* and shall be limited to have indices not greater than N (its size shall not exceed N+1). The additional rules governing bounded queues are described in 7.10.5.

### 7.10.2 Queue methods

In addition to the array operators, queues provide several built-in methods. Assume these declarations for the examples that follow:

```
typedef mytype element_t;  // mytype is any legal type for a queue
typedef element_t queue_t[$];
element_t e;
queue_t Q;
int i;
```

#### 7.10.2.1 Size()

The prototype for the `size()` method is as follows:

```
function int size();
```

The `size()` method returns the number of items in the queue. If the queue is empty, it returns 0.

```
for ( int j = 0; j < Q.size; j++ ) $display( Q[j] );
```

#### 7.10.2.2 Insert()

The prototype of the `insert()` method is as follows:

```
function void insert(input integer index, input element_t item);
```

The `insert()` method inserts the given item at the specified index position.

If the index argument has any bits with unknown (`x`/`z`) value, or is negative, or is greater than the current size of the queue, then the method call shall have no effect on the queue and may cause a warning to be issued.

NOTE—The index argument is of type **integer** rather than **int** so that `x/z` values in the caller's actual argument value can be detected.

### 7.10.2.3 Delete()

The prototype for the `delete()` method is as follows:

```
function void delete( [input integer index] );
```

where `index` is an optional index.

If the index is not specified, then the `delete()` method deletes all the elements in the queue, leaving the queue empty.

If the index is specified, then the `delete()` method deletes the item at the specified index position. If the index argument has any bits with unknown (`x/z`) value, or is negative, or is greater than or equal to the current size of the queue, then the method call shall have no effect on the queue and may cause a warning to be issued.

### 7.10.2.4 Pop_front()

The prototype of the `pop_front()` method is as follows:

```
function element_t pop_front();
```

The `pop_front()` method removes and returns the first element of the queue.

If this method is called on an empty queue:

— Its return value shall be the same as that obtained by attempting to read a nonexistent array element of the same type as the queue's elements (as described in Table 7-1, in 7.4.6);

— It shall have no effect on the queue and may cause a warning to be issued.

### 7.10.2.5 Pop_back()

The prototype of the `pop_back()` method is as follows:

```
function element_t pop_back();
```

The `pop_back()` method removes and returns the last element of the queue.

If this method is called on an empty queue:

— Its return value shall be the same as that obtained by attempting to read a nonexistent array element of the same type as the queue's elements (as described in Table 7-1 in 7.4.6);

— It shall have no effect on the queue and may cause a warning to be issued.

### 7.10.2.6 Push_front()

The prototype of the `push_front()` method is as follows:

```
function void push_front(input element_t item);
```

The `push_front()` method inserts the given element at the front of the queue.

### 7.10.2.7 Push_back()

The prototype of the `push_back()` method is as follows:

```
function void push_back(input element_t item);
```

The `push_back()` method inserts the given element at the end of the queue.

### 7.10.3 Persistence of references to elements of a queue

As described in 13.5.2, it is possible for an element of a queue to be passed by reference to a task that continues to hold the reference while other operations are performed on the queue. Some operations on the queue shall cause any such reference to become outdated (as defined in 13.5.2). This subclause defines the situations in which a reference to a queue element shall become outdated.

When any of the queue methods described in 7.10.2 updates a queue, a reference to any existing element that is not deleted by the method shall not become outdated. All elements that are removed from the queue by the method shall become outdated references.

When the target of an assignment is an entire queue, references to any element of the original queue shall become outdated.

As a consequence of this clause, inserting elements in a queue using unpacked array concatenation syntax, as illustrated in the examples in 7.10.4, will cause all references to any element of the existing queue to become outdated. Use of the `delete`, `pop_front`, and `pop_back` methods will outdate any reference to the popped or deleted element, but will leave references to all other elements of the queue unaffected. By contrast, use of the `insert`, `push_back`, and `push_front` methods on a queue can never give rise to outdated references (except that `insert` or `push_front` on a bounded queue would cause the highest-numbered element of the queue to be deleted if the new size of the queue were to exceed the queue's bound).

### 7.10.4 Updating a queue using assignment and unpacked array concatenation

As described in 7.10, a queue variable may be updated by assignment. Together with unpacked array concatenation, this offers a flexible alternative to the queue methods described in 7.10.2 when performing operations on a queue variable.

The following examples show queue assignment operations that exhibit behaviors similar to those of queue methods. In each case the resulting value of the queue variable shall be the same as if the queue method had been applied, but any reference to elements of the queue will become outdated by the assignment operation (see 7.10.3):

```
int q[$] = { 2, 4, 8 };
int e, pos;

// assignment                       // method call yielding the
//                                  // same value in variable q
// -------------------------------- // -------------------------
q = { q, 6 };                       // q.push_back(6)
q = { e, q };                       // q.push_front(e)
q = q[1:$];                         // void'(q.pop_front()) or q.delete(0)
q = q[0:$-1];                       // void'(q.pop_back()) or
                                    // q.delete(q.size-1)
q = { q[0:pos-1], e, q[pos:$] };    // q.insert(pos, e)
q = { q[0:pos], e, q[pos+1:$] };    // q.insert(pos+1, e)
q = {};                             // q.delete()
```

Some useful operations that cannot be implemented as a single queue method call are illustrated in the following examples. As in the preceding examples, assignment to the queue variable outdates any reference to its elements.

```
q = q[2:$];        // a new queue lacking the first two items
q = q[1:$-1];      // a new queue lacking the first and last items
```

### 7.10.5 Bounded queues

A bounded queue shall not have an element whose index is higher than the queue's declared upper bound. Operations on bounded queues shall behave exactly as if the queue were unbounded except that if, after any operation that writes to a bounded queue variable, that variable has any elements beyond its bound, then all such out-of-bounds elements shall be discarded and a warning shall be issued.

NOTE—Implementations may meet this requirement in any way that achieves the same result. In particular, they are not required to write the out-of-bounds elements before discarding them.

## 7.11 Array querying functions

SystemVerilog provides system functions to return information about an array. These are **$left**, **$right**, **$low**, **$high**, **$increment**, **$size**, **$dimensions**, and **$unpacked_dimensions**. These functions are described in 20.7.

## 7.12 Array manipulation methods

SystemVerilog provides several built-in methods to facilitate array searching, ordering, and reduction.

The general syntax to call these array methods is as follows:

```
array_method_call ::=
    expression . array_method_name { attribute_instance } [ ( iterator_argument ) ]
       [ with ( expression ) ]
```

*Syntax 7-5—Array method call syntax (not in Annex A)*

The optional **with** clause accepts an expression enclosed in parentheses. In contrast, the **with** clause used by the randomize method (see 18.7) accepts a set of constraints enclosed in braces.

If the expression contained in the **with** clause includes any side effects, the results may be unpredictable.

Array manipulation methods iterate over the array elements, which are then used to evaluate the expression specified by the **with** clause. The *iterator_argument* optionally specifies the name of the variable used by the **with** expression to designate the element of the array at each iteration. If it is not specified, the name `item` is used by default. The scope for the iterator_argument is the **with** expression. Specifying an iterator_argument without also specifying a **with** clause shall be illegal.

### 7.12.1 Array locator methods

Array locator methods operate on any unpacked array, including queues, but their return type is a queue. These locator methods allow searching an array for elements (or their indices) that satisfy a given expression. Array locator methods traverse the array in an unspecified order.

Index locator methods return a queue of **int** for all arrays except associative arrays, which return a queue of the same type as the associative index type. Associative arrays that specify a wildcard index type shall not be allowed.

If no elements satisfy the given expression or the array is empty (in the case of a queue or dynamic array), then an empty queue is returned. Otherwise, these methods return a queue containing all items that satisfy the expression. Index locator methods return a queue with the indices of all items that satisfy the expression. The optional expression specified by the **with** clause shall evaluate to a Boolean value.

The following locator methods are supported (the **with** clause is mandatory):

— `find()` returns all the elements satisfying the given expression.

— `find_index()` returns the indices of all the elements satisfying the given expression.

— `find_first()` returns the first element satisfying the given expression.

— `find_first_index()` returns the index of the first element satisfying the given expression.

— `find_last()` returns the last element satisfying the given expression.

— `find_last_index()` returns the index of the last element satisfying the given expression.

The first or last element is defined as being closest to the leftmost or rightmost indexed element, respectively, except for an associative array, which shall use the element closest to the index returned by the first or last method for the associative array index type.

For the following locator methods, the **with** clause (and its expression) may be omitted if the relational operators (<, >, ==) are defined for the element type of the given array. If a **with** clause is specified, the relational operators (<, >, ==) shall be defined for the type of the expression.

— `min()` returns the element with the minimum value or whose expression evaluates to a minimum.

— `max()` returns the element with the maximum value or whose expression evaluates to a maximum.

— **`unique`**`()` returns all elements with unique values or whose expression evaluates to a unique value. The queue returned contains one and only one entry for each of the values found in the array. The ordering of the returned elements is unrelated to the ordering of the original array.

— `unique_index()` returns the indices of all elements with unique values or whose expression evaluates to a unique value. The queue returned contains one and only one entry for each of the values found in the array. The ordering of the returned elements is unrelated to the ordering of the original array. The index returned for duplicate valued entries may be the index for one of the duplicates.

*Examples:*

```
string SA[10], qs[$];
int IA[int], qi[$];

// Find all items greater than 5
qi = IA.find( x ) with ( x > 5 );
qi = IA.find( x );                          // shall be an error

// Find indices of all items equal to 3
qi = IA.find_index with ( item == 3 );

// Find first item equal to Bob
qs = SA.find_first with ( item == "Bob" );

// Find last item equal to Henry
qs = SA.find_last( y ) with ( y == "Henry" );
```

```
// Find index of last item greater than Z
qi = SA.find_last_index( s ) with ( s > "Z" );

// Find smallest item
qi = IA.min;

// Find string with largest numerical value
qs = SA.max with ( item.atoi );

// Find all unique string elements
qs = SA.unique;

// Find all unique strings in lowercase
qs = SA.unique( s ) with ( s.tolower );
```

### 7.12.2 Array ordering methods

Array ordering methods reorder the elements of any unpacked array (fixed or dynamically sized) except for associative arrays.

The prototype for the ordering methods is as follows:

```
function void ordering_method ( array_type iterator = item );
```

The following ordering methods are supported:

- `reverse()` reverses the order of the elements in the array. Specifying a **with** clause shall be a compiler error.
- `sort()` sorts the array in ascending order, optionally using the expression in the **with** clause. The **with** clause (and its expression) is optional when the relational operators (<, >, ==) are defined for the array element type. If a **with** clause is specified, the relational operators (<, >, ==) shall be defined for the type of the expression.
- `rsort()` sorts the array in descending order, optionally using the expression in the **with** clause. The **with** clause (and its expression) is optional when the relational operators (<, >, ==) are defined for the array element type. If a **with** clause is specified, the relational operators (<, >, ==) shall be defined for the type of the expression.
- `shuffle()` randomizes the order of the elements in the array. Specifying a **with** clause shall be a compiler error.

*Examples:*

```
string s[] = { "hello", "sad", "world" };
s.reverse;        // s becomes { "world", "sad", "hello" };

int q[$] = { 4, 5, 3, 1 };
q.sort;                                 // q becomes { 1, 3, 4, 5 }

struct { byte red, green, blue; } c [512];
c.sort with ( item.red );               // sort c using the red field only
c.sort( x ) with ( {x.blue, x.green} );   // sort by blue then green
```

### 7.12.3 Array reduction methods

Array reduction methods may be applied to any unpacked array of integral values to reduce the array to a single value. The expression within the optional **with** clause is used to specify the values to use in the reduction. The values produced by evaluating this expression for each array element are used by the reduction method. This is in contrast to the array locator methods (see 7.12.1) where the **with** clause is used as a selection criteria.

The prototype for these methods is as follows:

```
function expression_or_array_type reduction_method (array_type iterator = item);
```

The method returns a single value of the same type as the array element type or, if specified, the type of the expression in the **with** clause. The **with** clause may be omitted if the corresponding arithmetic or Boolean reduction operation is defined for the array element type. If a **with** clause is specified, the corresponding arithmetic or Boolean reduction operation shall be defined for the type of the expression.

The following reduction methods are supported:

— `sum()` returns the sum of all the array elements or, if a **with** clause is specified, returns the sum of the values yielded by evaluating the expression for each array element.
— `product()` returns the product of all the array elements or, if a **with** clause is specified, returns the product of the values yielded by evaluating the expression for each array element.
— **and**`()` returns the bitwise AND ( & ) of all the array elements or, if a **with** clause is specified, returns the bitwise AND of the values yielded by evaluating the expression for each array element.
— **or**`()` returns the bitwise OR ( | ) of all the array elements or, if a **with** clause is specified, returns the bitwise OR of the values yielded by evaluating the expression for each array element.
— **xor**`()` returns the bitwise XOR ( ^ ) of all the array elements or, if a **with** clause is specified, returns the bitwise XOR of the values yielded by evaluating the expression for each array element.

*Examples:*

```
byte b[] = { 1, 2, 3, 4 };
int y;
y = b.sum ;                   // y becomes 10 => 1 + 2 + 3 + 4
y = b.product ;               // y becomes 24 => 1 * 2 * 3 * 4
y = b.xor with ( item + 4 );  // y becomes 12 => 5 ^ 6 ^ 7 ^ 8

logic [7:0] m [2][2] = '{ '{5, 10}, '{15, 20} };
int y;
y = m.sum with (item.sum with (item)); // y becomes 50 => 5+10+15+20

logic bit_arr [1024];
int y;
y = bit_arr.sum with ( int'(item) );   // forces result to be 32-bit
```

The last example shows how the result of calling `sum` on a bit array can be forced to be a 32-bit quantity. By default, the result of calling `sum` would be of type **logic** in this example. Summing the values of 1024 bits could overflow the result. This overflow can be avoided by using a **with** clause. When specified, the **with** clause is used to determine the type of the result. Casting `item` to an **int** in the **with** clause causes the array elements to be extended to 32 bits before being summed. The result of calling `sum` in this example is 32 bits since the width of the reduction method result shall be the same as the width of the expression in the **with** clause.

### 7.12.4 Iterator index querying

The expressions used by array manipulation methods sometimes need the actual array indices at each iteration, not just the array element. The index method of an iterator returns the index value of the specified dimension. The prototype of the index method is as follows:

```
    function int_or_index_type index ( int dimension = 1 );
```

The array dimensions are numbered as defined in 20.7. The slowest varying is dimension 1. Successively faster varying dimensions have sequentially higher dimension numbers. If the dimension is not specified, the first dimension is used by default.

The return type of the index method is an **int** for all array iterator items except associative arrays, which return an index of the same type as the associative index type. Associative arrays that specify a wildcard index type shall not be allowed.

For example:

```
    int arr[];
    int q[$];
    ...

    // find all items equal to their position (index)

q = arr.find with ( item == item.index );
```

# 8. Classes

## 8.1 General

This clause describes the following:

- — Class definitions
- — Virtual classes and methods
- — Polymorphism
- — Parameterized classes
- — Interface classes
- — Memory management

## 8.2 Overview

A class is a type that includes data and subroutines (functions and tasks) that operate on those data. A class's data are referred to as *class properties*, and its subroutines are called *methods*; both are members of the class. The class properties and methods, taken together, define the contents and capabilities of some kind of object.

For example, a packet might be an object. It might have a command field, an address, a sequence number, a time stamp, and a packet payload. In addition, there are various things that can be done with a packet: initialize the packet, set the command, read the packet's status, or check the sequence number. Each packet is different, but as a class, packets have certain intrinsic properties that can be captured in a definition.

```
class Packet ;
   //data or class properties
   bit [3:0] command;
   bit [40:0] address;
   bit [4:0] master_id;
   integer time_requested;
   integer time_issued;
   integer status;
   typedef enum { ERR_OVERFLOW = 10, ERR_UNDERFLOW = 1123} PCKT_TYPE;
   const integer buffer_size = 100;
   const integer header_size;

   // initialization
   function new();
      command = 4'd0;
      address = 41'b0;
      master_id = 5'bx;
      header_size = 10;
   endfunction

   // methods
   // public access entry points
   task clean();
      command = 0; address = 0; master_id = 5'bx;
   endtask

   task issue_request( int delay );
      // send request to bus
   endtask

   function integer current_status();
```

```
        current_status = status;
    endfunction
endclass
```

The object-oriented class extension allows objects to be created and destroyed dynamically. Class instances, or objects, can be passed around via object handles, which provides a safe-pointer capability. An object can be declared as an argument with direction **input**, **output**, **inout**, or **ref**. In each case, the argument copied is the object handle, not the contents of the object.

## 8.3 Syntax

```
class_declaration ::=                                                          // from A.1.2
    [virtual ] class [ lifetime ] class_identifier [ parameter_port_list ]
        [ extends class_type [ ( list_of_arguments ) ] ]
        [ implements interface_class_type { , interface_class_type } ] ;
        { class_item }
    endclass [ : class_identifier]
interface_class_type ::= ps_class_identifier [ parameter_value_assignment ]
class_item ::=                                                                 // from A.1.9
      { attribute_instance } class_property
    | { attribute_instance } class_method
    | { attribute_instance } class_constraint
    | { attribute_instance } class_declaration
    | { attribute_instance } covergroup_declaration
    | local_parameter_declaration ;
    | parameter_declaration[7] ;
    | ;
class_property ::=
      { property_qualifier } data_declaration
    | const { class_item_qualifier } data_type const_identifier [ = constant_expression ] ;
class_method ::=
      { method_qualifier } task_declaration
    | { method_qualifier } function_declaration
    | pure virtual { class_item_qualifier } method_prototype ;
    | extern { method_qualifier } method_prototype ;
    | { method_qualifier } class_constructor_declaration
    | extern { method_qualifier } class_constructor_prototype
class_constructor_prototype ::=
    function new [ ( [ tf_port_list ] ) ] ;
class_constraint ::=
      constraint_prototype
    | constraint_declaration
class_item_qualifier[8] ::=
      static
    | protected
    | local
property_qualifier[8] ::=
      random_qualifier
    | class_item_qualifier
```

```
random_qualifier8 ::=
    rand
  | randc
method_qualifier8 ::=
    [ pure ] virtual
  | class_item_qualifier
method_prototype ::=
    task_prototype
  | function_prototype
```

7) In a parameter_declaration that is a class_item, the **parameter** keyword shall be a synonym for the **localparam** keyword.

8) In any one declaration, only one of **protected** or **local** is allowed, only one of **rand** or **randc** is allowed, and **static** and/or **virtual** can appear only once.

*Syntax 8-1—Class syntax (excerpt from Annex A)*

## 8.4 Objects (class instance)

A class defines a data type. An object is an instance of that class. An object is used by first declaring a variable of that class type (that holds an object handle) and then creating an object of that class (using the **new** function) and assigning it to the variable.

```
Packet p;   // declare a variable of class Packet
p = new;    // initialize variable to a new allocated object
            // of the class Packet
```

The variable `p` is said to hold an object handle to an object of class `Packet`.

Uninitialized object handles are set by default to the special value **null.** An uninitialized object can be detected by comparing its handle with **null**.

For example: The following task `task1` checks whether the object is initialized. If it is not, it creates a new object via the **new** function.

```
class obj_example;
      ...
endclass

task task1(integer a, obj_example myexample);
   if (myexample == null) myexample = new;
endtask
```

Accessing non-static members (see 8.9) or virtual methods (see 8.20) via a **null** object handle is illegal. The result of an illegal access via a null object is indeterminate, and implementations may issue an error.

SystemVerilog objects are referenced using an object handle. There are some differences between a C pointer and a SystemVerilog object handle (see Table 8-1). C pointers give programmers a lot of latitude in how a pointer can be used. The rules governing the usage of SystemVerilog object handles are much more restrictive. A C pointer can be incremented, for example, but a SystemVerilog object handle cannot. In addition to object handles, 6.14 introduces the **chandle** data type for use with the DPI (see Clause 35).

**Table 8-1—Comparison of pointer and handle types**

| Operation | C pointer | SV object handle | SV chandle |
|---|---|---|---|
| Arithmetic operations (such as incrementing) | Allowed | Not allowed | Not allowed |
| For arbitrary data types | Allowed | Not allowed | Not allowed |
| Dereference when **null** | Error | Error, see text above | Not allowed |
| Casting | Allowed | Limited | Not allowed |
| Assignment to an address of a data type | Allowed | Not allowed | Not allowed |
| Unreferenced objects are garbage collected | No | Yes | No |
| Default value | Undefined | **null** | **null** |
| For classes | (C++) | Allowed | Not allowed |

Only the following operators are valid on object handles:

— Equality (==), inequality (!=) with another class object or with **null**. One of the objects being compared must be assignment compatible with the other.

— Case equality (===), case inequality (!==) with another class object or with **null** (same semantics as == and !=).

— Conditional operator (see 11.4.11).

— Assignment of a class object whose class data type is assignment compatible with the target class object.

— Assignment of **null**.

## 8.5 Object properties and object parameter data

There are no restrictions on the data type of a class property. The class properties of an object can be used by qualifying class property names with an instance name. Using the earlier example (see 8.2), the properties for the `Packet` object `p` can be used as follows:

```
Packet p = new;
int var1;
p.command = INIT;
p.address = $random;
packet_time = p.time_requested;
var1 = p.buffer_size;
```

Class enum names, in addition to being accessed using a class scope resolution operator, can also be accessed by qualifying the class enum name with an instance name.

```
initial $display (p.ERR_OVERFLOW);
```

The parameter data values of an object can also be accessed by qualifying the class value parameter or local value parameter name with an instance name. Such an expression is not a constant expression. Accessing data types using a class handle is not allowed. For example:

```
class vector #(parameter width = 7, type T = int);
endclass

vector #(3) v = new;
initial $display (vector #(3)::T'(3.45));    // Typecasting
initial $display ((v.T)'(3.45));             //ILLEGAL
initial $display (v.width);
```

## 8.6 Object methods

An object's methods can be accessed using the same syntax used to access class properties:

```
Packet p = new;
status = p.current_status();
```

The preceding assignment to `status` cannot be written as follows:

```
status = current_status(p);
```

The focus in object-oriented programming is the object, in this case the packet, not the function call. Also, objects are self-contained, with their own methods for manipulating their own properties. Therefore, the object does not have to be passed as an argument to `current_status()`. A class's properties are freely and broadly available to the methods of the class, but each method only accesses the properties associated with its object, i.e., its instance.

The lifetime of methods declared as part of a class type shall be automatic. It shall be illegal to declare a class method with a static lifetime.

## 8.7 Constructors

SystemVerilog does not require the complex memory allocation and deallocation of C++. Construction of an object is straightforward; and garbage collection, as in Java, is implicit and automatic. There can be no memory leaks or other subtle behaviors, which are so often the bane of C++ programmers.

SystemVerilog provides a mechanism for initializing an instance at the time the object is created. When an object is created, for example,

```
Packet p = new;
```

the system executes the **`new`** function associated with the class:

```
class Packet;
   integer command;

   function new();
      command = IDLE;
   endfunction
endclass
```

As shown previously, **`new`** is now being used in two very different contexts with very different semantics. The variable declaration creates an object of class `Packet`. In the course of creating this instance, the **`new`** function is invoked, in which any specialized initialization required can be done. The **`new`** function is also called the *class constructor*.

The **new** operation is defined as a function with no return type, and like any other function, it shall be nonblocking. Even though **new** does not specify a return type, the left-hand side of the assignment determines the return type.

If a class does not provide an explicit user-defined **new** method, an implicit **new** method shall be provided automatically. The **new** method of a derived class shall first call its base class constructor [**super.new()** as described in 8.15]. After the base class constructor call (if any) has completed, each property defined in the class shall be initialized to its explicit default value or its uninitialized value if no default is provided. After the properties are initialized, the remaining code in a user-defined constructor shall be evaluated. The default constructor has no additional effect after the property initialization. The value of a property prior to its initialization shall be undefined.

*Example:*

```
class C;
   int c1 = 1;
   int c2 = 1;
   int c3 = 1;
   function new(int a);
      c2 = 2;
      c3 = a;
   endfunction
endclass

class D extends C;
   int d1 = 4;
   int d2 = c2;
   int d3 = 6;
   function new;
      super.new(d3);
   endfunction
endclass
```

After the construction of an object of type `D` is complete, the properties are as follows:

— `c1` has the value 1
— `c2` has the value 2 since the constructor assignment happens after the property initialization
— `c3` has an undefined value since the constructor call from `D` passes in the value of `d3`, which is undefined when the **super.new**`(d3)` call is made
— `d1` has the value 4
— `d2` has the value 2 since the **super.new** call is complete when `d2` is initialized
— `d3` has the value 6

It is also possible to pass arguments to the constructor, which allows run-time customization of an object:

```
Packet p = new(STARTUP, $random, $time);
```

where the **new** initialization task in `Packet` might now look like the following:

```
function new(int cmd = IDLE, bit[12:0] adrs = 0, int cmd_time );
   command = cmd;
   address = adrs;
   time_requested = cmd_time;
endfunction
```

The conventions for arguments are the same as for any other procedural subroutine calls, such as the use of default arguments.

A constructor may be declared as a **local** or **protected** method (see 8.18). A constructor shall not be declared as a **static** (see 8.10) or **virtual** method (see 8.20).

## 8.8 Typed constructor calls

class_new[19] ::= *// from A.2.4*
  [ class_scope ] **new** [ **(** list_of_arguments **)** ]
  | **new** expression

19) In a shallow copy, the expression shall evaluate to an object handle.

*Syntax 8-2—Calling a constructor (excerpt from Annex A)*

Uses of **new** described in earlier parts of this clause require that the type of the object to be constructed matches the assignment target's type. An alternative form of constructor invocation, the *typed constructor call*, adds *class_scope* immediately before the **new** keyword, specifying the constructed object's type independently of the assignment target. The specified type shall be assignment compatible with the target.

The following example illustrates a typed constructor call. The **extends** keyword is described in 8.13. The concept of a superclass type is described in 8.15.

```
class C; . . . endclass
class D extends C; . . . endclass
C c = D::new;     // variable c of superclass type C now references
                  // a newly constructed object of type D
```

NOTE—The effect of this typed constructor call is as if a temporary variable of type D had been declared, constructed, and then copied to variable c, as in this example fragment:

```
D d = new;
C c = d;
```

A typed constructor call shall create and initialize a new object of the specified type. Creation and initialization of the new object shall proceed exactly as it would for an ordinary constructor as described in 8.7. Arguments may be passed to a typed constructor call if appropriate, just as for an ordinary constructor.

If the type of object to be constructed is a parameterized class, as described in 8.25, the specified type may have parameter specializations. The following example, continuing the previous example, illustrates a typed constructor call for a parameterized class and also illustrates how arguments may be passed to the constructor as described in 8.7.

```
class E #(type T = int) extends C;
   T x;
   function new(T x_init);
      super.new();
      x = x_init;
   endfunction
endclass

initial begin
```

```
      c = E #(.T(byte))::new(.x_init(5));
   end
```

## 8.9 Static class properties

The previous examples have only declared instance class properties. Each instance of the class (i.e., each object of type `Packet`) has its own copy of each of its eight variables. Sometimes only one version of a variable is required to be shared by all instances. These class properties are created using the keyword **`static`**. Thus, for example, in the following case, all instances of a class need access to a common file descriptor:

```
class Packet ;
   static integer fileID = $fopen( "data", "r" );
```

Now, `fileID` shall be created and initialized once. Thereafter, every `Packet` object can access the file descriptor in the usual way:

```
Packet p;
c = $fgetc( p.fileID );
```

The static class properties can be used without creating an object of that type.

## 8.10 Static methods

Methods can be declared as **`static`**. A static method is subject to all the class scoping and access rules, but behaves like a regular subroutine that can be called outside the class, even with no class instantiation. A static method has no access to non-static members (class properties or methods), but it can directly access static class properties or call static methods of the same class. Access to non-static members or to the special **`this`** handle within the body of a static method is illegal and results in a compiler error. Static methods cannot be virtual.

```
class id;
   static int current = 0;
   static function int next_id();
      next_id = ++current; // OK to access static class property
   endfunction
endclass
```

A static method is different from a task with static lifetime. The former refers to the lifetime of the method within the class, while the latter refers to the lifetime of the arguments and variables within the task.

```
class TwoTasks;
   static task t1(); ... endtask    // static class method with
                                    // automatic variable lifetime

   task static t2(); ... endtask    // ILLEGAL: non-static class method with
                                    // static variable lifetime
endclass
```

## 8.11 This

The **`this`** keyword is used to unambiguously refer to class properties, value parameters, local value parameters, or methods of the current instance. The **`this`** keyword denotes a predefined object handle that refers to the object that was used to invoke the subroutine that **`this`** is used within. The **`this`** keyword shall only be used within non-static class methods, constraints, inlined constraint methods, or covergroups embedded within classes (see 19.4); otherwise, an error shall be issued. For example, the following declaration is a common way to write an initialization task:

```
class Demo ;
   integer x;

   function new (integer x);
      this.x = x;
   endfunction
endclass
```

The `x` is now both a property of the class and an argument to the function **new**. In the function **new**, an unqualified reference to `x` shall be resolved by looking at the innermost scope, in this case, the subroutine argument declaration. To access the instance class property, it is qualified with the **this** keyword, to refer to the current instance.

NOTE—In writing methods, members can be qualified with **this** to refer to the current instance, but it is usually unnecessary.

## 8.12 Assignment, renaming, and copying

Declaring a class variable only creates the name by which the object is known. Thus,

```
Packet p1;
```

creates a variable, `p1`, that can hold the handle of an object of class `Packet`, but the initial value of `p1` is **null**. The object does not exist, and `p1` does not contain an actual handle, until an instance of type `Packet` is created:

```
p1 = new;
```

Thus, if another variable is declared and assigned the old handle, `p1`, to the new one, as in

```
Packet p2;
p2 = p1;
```

then there is still only one object, which can be referred to with either the name `p1` or `p2`. In this example, **new** was executed only once; therefore, only one object has been created.

If, however, the preceding example is rewritten as follows, a copy of `p1` shall be made:

```
Packet p1;
Packet p2;
p1 = new;
p2 = new p1;
```

The last statement has **new** executing a second time, thus creating a new object `p2`, whose class properties are copied from `p1`. This is known as a *shallow copy*. All of the variables are copied: integers, strings, instance handles, etc. Objects, however, are not copied, only their handles; as before, two names for the same object have been created. This is true even if the class declaration includes the instantiation operator **new**.

It shall be illegal to use a typed constructor call for a shallow copy (see 8.8).

A *shallow copy* executes in the following manner:

1) An object of the class type being copied is allocated. This allocation shall not call the object's constructor or execute any variable declaration initialization assignments.
2) All class properties, including the internal states used for randomization and coverage, are copied to the new object. Object handles are copied; this includes the object handles for covergroup objects (see Clause 19). An exception is made for embedded covergroups (see 19.4). The object handle of an embedded covergroup shall be set to **null** in the new object. The internal states for randomization include the random number generator (RNG) state, the `constraint_mode` status of constraints, the `rand_mode` status of random variables, and the cyclic state of **randc** variables (see Clause 18).
3) A handle to the newly created object is assigned to the variable on the left-hand side.

NOTE—A shallow copy does not create new coverage objects (covergroup instances). As a result, the properties of the new object are not covered.

```
class baseA ;
   integer j = 5;
endclass

class B ;
   integer i = 1;
   baseA a = new;
endclass
class xtndA extends baseA;
   rand int x;
   constraint cst1 { x < 10; }
endclass

function integer test;
   xtndA xtnd1;
   baseA base2, base3;
   B b1 = new;          // Create an object of class B
   B b2 = new b1;       // Create an object that is a copy of b1
   b2.i = 10;           // i is changed in b2, but not in b1
   b2.a.j = 50;         // change a.j, shared by both b1 and b2
   test = b1.i;         // test is set to 1 (b1.i has not changed)
   test = b1.a.j;       // test is set to 50 (a.j has changed)
   xtnd1 = new;         // create a new instance of class xtndA
   xtnd1.x = 3;
   base2 = xtnd1;       // base2 refers to the same object as xtnd1
   base3 = new base2;   // Creates a shallow copy of xtnd1
endfunction
```

In the last statement `base3` is assigned a shallow copy of `base2`. The type of the variable `base3` is a handle to the base class `baseA`. When the shallow copy is invoked, this variable contains a handle to an instance of the extended class `xtndA`. The shallow copy creates a duplicate of the referenced object, resulting in a duplicate instance of the extended class `xntdA`. The handle to this instance is then assigned to the variable `base3`.

Several things are noteworthy. First, class properties and instantiated objects can be initialized directly in a class declaration. Second, the shallow copy does not copy objects. Third, instance qualifications can be chained as needed to reach into objects or to reach through objects:

```
b1.a.j                  // reaches into a, which is a property of b1
p.next.next.next.val    // chain through a sequence of handles to get to val
```

To do a full (deep) copy, where everything (including nested objects) is copied, custom code is typically needed. For example:

```
Packet p1 = new;
Packet p2 = new;
p2.copy(p1);
```

where `copy(Packet p)` is a custom method written to copy the object specified as its argument into its instance.

## 8.13 Inheritance and subclasses

The previous subclauses defined a class called `Packet`. This class can be extended so that the packets can be chained together into a list. One solution would be to create a new class called `LinkedPacket` that contains a variable of type `Packet` called `packet_c`.

To refer to a class property of `Packet`, the variable `packet_c` needs to be referenced.

```
class LinkedPacket;
   Packet packet_c;
   LinkedPacket next;

   function LinkedPacket get_next();
      get_next = next;
   endfunction
endclass
```

Because `LinkedPacket` is a special form of `Packet`, a more elegant solution is to extend the class creating a new subclass that *inherits* the members of the base class. Thus, for example:

```
class LinkedPacket extends Packet;
   LinkedPacket next;

   function LinkedPacket get_next();
      get_next = next;
   endfunction
endclass
```

Now, all of the methods and class properties of `Packet` are part of `LinkedPacket` (as if they were defined in `LinkedPacket`), and `LinkedPacket` has additional class properties and methods.

The methods of the base class can also be overridden to change their definitions.

The mechanism provided by SystemVerilog is called *single inheritance,* that is, each class is derived from a single base class.

## 8.14 Overridden members

Subclass objects are also legal representative objects of their base classes. For example, every `LinkedPacket` object is a perfectly legal `Packet` object.

The handle of a `LinkedPacket` object can be assigned to a `Packet` variable:

```
LinkedPacket lp = new;
Packet p = lp;
```

In this case, references to `p` access the methods and class properties of the `Packet` class. So, for example, if class properties and methods in `LinkedPacket` are overridden, these overridden members referred to through `p` get the original members in the `Packet` class. From `p`, **`new`** and all overridden members in `LinkedPacket` are now hidden.

```
class Packet;
   integer i = 1;
   function integer get();
      get = i;
   endfunction
endclass

class LinkedPacket extends Packet;
   integer i = 2;
   function integer get();
      get = -i;
   endfunction
endclass

LinkedPacket lp = new;
Packet p = lp;
j = p.i;              // j = 1, not 2
j = p.get();          // j = 1, not -1 or -2
```

To call the overridden method via a base class object (`p` in the example), the method needs to be declared **`virtual`** (see 8.20).

## 8.15 Super

The **`super`** keyword is used from within a derived class to refer to members, class value parameters, or local value parameters of the base class. It is necessary to use **`super`** to access members, value parameters, or local value parameters of a base class when those are overridden by the derived class. An expression using **`super`** to access the value parameter or local value parameter is not a constant expression.

```
class Packet;                         // base class
   integer value;
   function integer delay();
      delay = value * value;
   endfunction
endclass

class LinkedPacket extends Packet;    // derived class
   integer value;
   function integer delay();
      delay = super.delay()+ value * super.value;
   endfunction
endclass
```

The member, value parameter, or local value parameter can be declared a level up or be inherited by the class one level up. There is no way to reach higher (for example, `super.super.count` is not allowed).

Subclasses (or derived classes) are classes that are extensions of the current class whereas superclasses (base classes) are classes from which the current class is extended, beginning with the original base class.

A **`super.new`** call shall be the first statement executed in the constructor. This is because the superclass shall be initialized before the current class and, if the user code does not provide an initialization, the compiler shall insert a call to **`super.new`** automatically.

## 8.16 Casting

It is always legal to assign an expression of subclass type to a variable of a class type higher in the inheritance tree (a superclass or ancestor of the expression type). It shall be illegal to directly assign a variable of a superclass type to a variable of one of its subclass types. However, `$cast` may be used to assign a superclass handle to a variable of a subclass type provided the superclass handle refers to an object that is assignment compatible with the subclass variable.

To check whether the assignment is legal, the dynamic cast function `$cast` is used (see 6.24.2).

The prototype for `$cast` is as follows:

```
    function int $cast( singular dest_var, singular source_exp );
```
or
```
    task $cast( singular dest_var, singular source_exp );
```

When `$cast` is applied to class handles, it succeeds in only three cases:

1) The source expression and the destination type are assignment compatible, that is, the destination is the same type or a superclass of the source expression.
2) The type of the source expression is cast compatible with the destination type, that is, either:
   — the type of the source expression is a superclass of the destination type, or
   — the type of the source expression is an interface class (see 8.26)

   and the source is an object that is assignment compatible with the destination type. This type of assignment requires a run-time check as provided by `$cast`.
3) The source expression is the literal constant **`null`**.

In all other situations `$cast` shall fail, particularly when the source and destination types are not cast compatible, even if the source expression evaluates to **`null`**.

When `$cast` succeeds, it performs the assignment. Otherwise, the error handling is as described in 6.24.2.

## 8.17 Chaining constructors

When a subclass is instantiated, the class method **`new`**`()` is invoked. The first action that **`new`**`()` takes, before any code defined in the function is evaluated, is to invoke the **`new`**`()` method of its superclass and so on up the inheritance hierarchy. Thus, all the constructors are called, in the proper order, beginning with the root base class and ending with the current class. Class property initialization occurs during this sequence as described in 8.7.

If the initialization method of the superclass requires arguments, there are two choices: to always supply the same arguments or to use the **`super`** keyword. If the arguments are always the same, then they can be specified at the time the class is extended:

```
    class EtherPacket extends Packet(5);
```

This passes 5 to the **`new`** routine associated with `Packet`.

A more general approach is to use the **`super`** keyword, to call the superclass constructor:

```
function new();
   super.new(5);
endfunction
```

To use this approach, **`super.new`**`(...)` shall be the first executable statement in the function **`new`**.

If the arguments are specified at the time the class is extended, the subclass constructor shall not contain a **`super.new`**`()` call. The compiler shall insert a call to **`super.new`**`()` automatically, as whenever the subclass constructor does not contain a **`super.new`**`()` call (see 8.15).

NOTE 1—Declaring a class constructor as a **`local`** method makes that class inextensible since the reference to **`super.new`**`()` in a subclass would be illegal.

NOTE 2—When calling a virtual method from a constructor **`new()`**, the constructor calls the method as described in 8.20. However, users must be aware of the class property initialization sequence as described in 8.7, as properties the method refers to may not have been initialized, depending on where in the chain of constructors the method was called from.

## 8.18 Data hiding and encapsulation

In SystemVerilog, unqualified class properties and methods are public, available to anyone who has access to the object's name. Often, it is desirable to restrict access to class properties and methods from outside the class by hiding their names. This keeps other programmers from relying on a specific implementation, and it also protects against accidental modifications to class properties that are internal to the class. When all data become hidden (i.e., being accessed only by public methods), testing and maintenance of the code become much easier.

Class parameters and class local parameters are also public.

A member identified as **`local`** is available only to methods inside the class. Further, these local members are not visible within subclasses. Of course, nonlocal methods that access local class properties or methods can be inherited and work properly as methods of the subclass.

A **`protected`** class property or method has all of the characteristics of a **`local`** member, except that it can be inherited; it is visible to subclasses.

Within a class, a local method or class property of the same class can be referenced, even if it is in a different instance of the same class. For example:

```
class Packet;
   local integer i;
   function integer compare (Packet other);
      compare = (this.i == other.i);
   endfunction
endclass
```

A strict interpretation of encapsulation might say that `other.i` should not be visible inside this packet because it is a local class property being referenced from outside its instance. Within the same class, however, these references are allowed. In this case, `this.i` shall be compared to `other.i`, and the result of the logical comparison returned.

Class members can be identified as either **`local`** or **`protected`**; class properties can be further defined as **`const`**, and methods can be defined as **`virtual`**. There is no predefined ordering for specifying these modifiers; however, they can only appear once per member. It shall be an error to define members to be both **`local`** and **`protected`** or to duplicate any of the other modifiers.

## 8.19 Constant class properties

Class properties can be made read-only by a **const** declaration like any other SystemVerilog variable. However, because class objects are dynamic objects, class properties allow two forms of read-only variables: global constants and instance constants.

Global constant class properties include an initial value as part of their declaration. They are similar to other **const** variables in that they cannot be assigned a value anywhere other than in the declaration.

```
class Jumbo_Packet;
   const int max_size = 9 * 1024; // global constant
   byte payload [];
   function new( int size );
      payload = new[ size > max_size ? max_size : size ];
   endfunction
endclass
```

Instance constants do not include an initial value in their declaration, only the **const** qualifier. This type of constant can be assigned a value at run time, but the assignment can only be done once in the corresponding class constructor.

```
class Big_Packet;
   const int size; // instance constant
   byte payload [];
   function new();
      size = $urandom % 4096; //one assignment in new -> ok
      payload = new[ size ];
   endfunction
endclass
```

Typically, global constants are also declared **static** because they are the same for all instances of the class. However, an instance constant cannot be declared **static** because doing so would disallow all assignments in the constructor.

## 8.20 Virtual methods

A method of a class may be identified with the keyword **virtual**. Virtual methods are a basic polymorphic construct. A virtual method shall override a method in all of its base classes, whereas a non-virtual method shall only override a method in that class and its descendants. One way to view this is that there is only one implementation of a virtual method per class hierarchy, and it is always the one in the latest derived class.

Virtual methods provide prototypes for the methods that later override them, i.e., all of the information generally found on the first line of a method declaration: the encapsulation criteria, the type and number of arguments, and the return type if it is needed.

Virtual method overrides in subclasses shall have matching argument types, identical argument names, identical qualifiers, and identical directions to the prototype. The **virtual** qualifier is optional in the derived class method declarations. The return type of a virtual function shall be either:

— a matching type (see 6.22.1)

— or a derived class type

of the return type of the virtual function in the superclass. It is not necessary to have matching default expressions, but the presence of a default shall match.

Example 1 illustrates virtual method override.

*Example 1:*

```
class BasePacket;
   int A = 1;
   int B = 2;
   function void printA;
      $display("BasePacket::A is %d", A);
   endfunction : printA
   virtual function void printB;
      $display("BasePacket::B is %d", B);
   endfunction : printB
endclass : BasePacket

class My_Packet extends BasePacket;
   int A = 3;
   int B = 4;
   function void printA;
      $display("My_Packet::A is %d", A);
   endfunction: printA
   virtual function void printB;
      $display("My_Packet::B is %d", B);
   endfunction : printB
endclass : My_Packet

BasePacket P1 = new;
My_Packet P2 = new;

initial begin
   P1.printA;  // displays 'BasePacket::A is 1'
   P1.printB;  // displays 'BasePacket::B is 2'
   P1 = P2;    // P1 has a handle to a My_packet object
   P1.printA;  // displays 'BasePacket::A is 1'
   P1.printB;  // displays 'My_Packet::B is 4' - latest derived method
   P2.printA;  // displays 'My_Packet::A is 3'
   P2.printB;  // displays 'My_Packet::B is 4'
end
```

Example 2 illustrates the use of a derived class type for a virtual function return type and of matching formal argument types. In the derived class `D`, the virtual function return type is `D`, a derived class type of `C`. The formal argument data type is `T`, which is a matching data type of the predefined type **`int`**.

*Example 2:*

```
typedef int T;    // T and int are matching data types.

class C;
   virtual function C some_method(int a); endfunction
endclass

class D extends C;
   virtual function D some_method(T a); endfunction
endclass

class E #(type Y = logic) extends C;
   virtual function D some_method(Y a); endfunction
endclass
```

```
E #() v1;       // Illegal: type parameter Y resolves to logic, which is not
                // a matching type for argument a
E #(int) v2;    // Legal: type parameter Y resolves to int
```

A virtual method may override a non-virtual method, but once a method has been identified as virtual, it shall remain virtual in any subclass that overrides it. In that case, the **virtual** keyword may be used in later declarations, but is not required.

## 8.21 Abstract classes and pure virtual methods

A set of classes may be created that can be viewed as all being derived from a common base class. For example, a common base class of type `BasePacket` that sets out the structure of packets, but is incomplete, would never be constructed. This is characterized as an *abstract class*. From this abstract base class, however, a number of useful subclasses may be derived, such as Ethernet packets, token ring packets, GPS packets, and satellite packets. Each of these packets might look very similar, all needing the same set of methods, but they could vary significantly in terms of their internal details.

A base class may be characterized as being abstract by identifying it with the keyword **virtual**:

```
virtual class BasePacket;
   ...
endclass
```

An object of an abstract class shall not be constructed directly. Its constructor may only be called indirectly through the chaining of constructor calls originating in an extended non-abstract object.

A virtual method in an abstract class may be declared as a prototype without providing an implementation. This is called a *pure virtual method* and shall be indicated with the keyword **pure** together with not providing a method body. An extended subclass may provide an implementation by overriding the pure virtual method with a virtual method having a method body.

Abstract classes may be extended to further abstract classes, but all pure virtual methods shall have overridden implementations in order to be extended by a non-abstract class. By having implementations for all its methods, the class is complete and may now be constructed. Any class may be extended into an abstract class, and may provide additional or overridden pure virtual methods.

```
virtual class BasePacket;
   pure virtual function integer send(bit[31:0] data); // No implementation
endclass

class EtherPacket extends BasePacket;
   virtual function integer send(bit[31:0] data);
      // body of the function
      ...
   endfunction
endclass
```

`EtherPacket` is now a class that can have an object of its type constructed.

NOTE—A method without a statement body is still a legal, callable method. For example, if the function `send` was declared as follows, it would have an implementation:

```
virtual function integer send(bit[31:0] data); // Will return 'x
endfunction
```

## 8.22 Polymorphism: dynamic method lookup

Polymorphism allows the use of a variable of the superclass type to hold subclass objects and to reference the methods of those subclasses directly from the superclass variable. As an example, assume the base class for the `Packet` objects, `BasePacket`, defines, as virtual functions, all of the public methods that are to be generally used by its subclasses. Such methods include `send`, `receive`, and `print`. Even though `BasePacket` is abstract, it can still be used to declare a variable:

```
BasePacket packets[100];
```

Now, instances of various packet objects can be created and put into the array:

```
EtherPacket ep = new;   // extends BasePacket
TokenPacket tp = new;   // extends BasePacket
GPSPacket gp = new;     // extends EtherPacket
packets[0] = ep;
packets[1] = tp;
packets[2] = gp;
```

If the data types were, for example, integers, bits, and strings, all of these types could not be stored into a single array, but with polymorphism, it can be done. In this example, because the methods were declared as **`virtual`**, the appropriate subclass methods can be accessed from the superclass variable, even though the compiler did not know—at compile time—what was going to be loaded into it.

For example, `packets[1]`

```
packets[1].send();
```

shall invoke the `send` method associated with the `TokenPacket` class. At run time, the system correctly binds the method from the appropriate class.

This is a typical example of polymorphism at work, providing capabilities that are far more powerful than what is found in a nonobject-oriented framework.

## 8.23 Class scope resolution operator ::

The class scope resolution operator `::` is used to specify an identifier defined within the scope of a class. It has the following form:

```
class_type :: { class_type :: } identifier
```

The left operand of the scope resolution operator `::` shall be a class type name, package name (see 26.2), **`covergroup`** type name, **`coverpoint`** name, **`cross`** name (see 19.5, 19.6), **`typedef`** name, or type parameter name. When a type name is used, the name shall resolve to a class or covergroup type after elaboration.

Because classes and other scopes can have the same identifiers, the class scope resolution operator uniquely identifies a member, a parameter or local parameter of a particular class. In addition to disambiguating class scope identifiers, the `::` operator also allows access to static members (class properties and methods), class parameters, and class local parameters from outside the class, as well as access to public or protected elements of a superclass from within the derived classes. A class parameter or local parameter is a public element of a class. A class scoped parameter or local parameter is a constant expression.

```
class Base;
   typedef enum {bin,oct,dec,hex} radix;
   static task print( radix r, integer n ); ... endtask
endclass
...
Base b = new;
int bin = 123;
b.print( Base::bin, bin );    // Base::bin and bin are different
Base::print( Base::hex, 66 );
```

In SystemVerilog, the class scope resolution operator applies to all static elements of a class: static class properties, static methods, typedefs, enumerations, parameters, local parameters, constraints, structures, unions, and nested class declarations. Class scope resolved expressions can be read (in expressions), written (in assignments or subroutines calls), or triggered off (in event expressions). A class scope can also be used as the prefix of a type or a method call.

Like modules, classes are scopes and can nest. Nesting allows hiding of local names and local allocation of resources. This is often desirable when a new type is needed as part of the implementation of a class. Declaring types within a class helps prevent name collisions and the cluttering of the outer scope with symbols that are used only by that class. Type declarations nested inside a class scope are public and can be accessed outside the class.

```
class StringList;
   class Node; // Nested class for a node in a linked list.
      string name;
      Node link;
   endclass
endclass

class StringTree;
   class Node; // Nested class for a node in a binary tree.
      string name;
      Node left, right;
   endclass
endclass
// StringList::Node is different from StringTree::Node
```

The class scope resolution operator enables the following:

— Access to static public members (methods and class properties) from outside the class hierarchy.
— Access to public or protected class members of a superclass from within the derived classes.
— Access to constraints, type declarations, and enumeration named constants declared inside the class from outside the class hierarchy or from within derived classes.
— Access to parameters and local parameters declared inside the class from outside the class hierarchy or from within derived classes.

Nested classes shall have the same access rights as methods do in the containing class. They have full access rights to **local** and **protected** methods and properties of the containing class. Nested classes have lexically scoped, unqualified access to the **static** properties and methods, parameters, and local parameters of the containing class. They shall not have implicit access to non-static properties and methods except through a handle either passed to it or otherwise accessible by it. There is no implicit **this** handle to the outer class. For example:

```
class Outer;
   int             outerProp;
   local int       outerLocalProp;
   static int        outerStaticProp;
   static local int  outerLocalStaticProp;
   class Inner;
      function void innerMethod(Outer h);
         outerStaticProp = 0;
            // Legal, same as Outer::outerStaticProp
         outerLocalStaticProp = 0;
            // Legal, nested classes may access local's in outer class
         outerProp = 0;
            // Illegal, unqualified access to non-static outer
         h.outerProp = 0;
            // Legal, qualified access.
         h.outerLocalProp = 0;
            // Legal, qualified access and locals to outer class allowed.
      endfunction
   endclass
endclass
```

The class scope resolution operator has special rules when used with a prefix that is the name of a parameterized class; see 8.25.1 for details.

## 8.24 Out-of-block declarations

It is convenient to be able to move method definitions out of the body of the class declaration. This is done in two steps. First, within the class body, declare the method prototypes, i.e., whether it is a function or task, any qualifiers (**local**, **protected**, or **virtual**), and the full argument specification plus the **extern** qualifier. The **extern** qualifier indicates that the body of the method (its implementation) is to be found outside the declaration. Second, outside the class declaration, declare the full method (e.g., the prototype but without the qualifiers), and, to tie the method back to its class, qualify the method name with the class name and a pair of colons, as follows:

```
class Packet;
   Packet next;
   function Packet get_next();// single line
      get_next = next;
   endfunction

   // out-of-body (extern) declaration
   extern protected virtual function int send(int value);
endclass

function int Packet::send(int value);
   // dropped protected virtual, added Packet::
   // body of method
      ...
endfunction
```

The out-of-block method declaration shall match the prototype declaration exactly, with the following exceptions:

- The method name is preceded by the class name and the class scope resolution operator.
- A function return type may also require the addition of a class scope in the out-of-block declaration, as described below.
- A default argument value specified in the prototype may be omitted in the out-of-block declaration. If a default argument value is specified in the out-of-block declaration, then there shall be a syntactically identical default argument value specified in the prototype.

An out-of-block declaration shall be declared in the same scope as the class declaration and shall follow the class declaration. It shall be an error if more than one out-of-block declaration is provided for a particular **extern** method.

The class scope resolution operator is required in some situations in order to name the return type of a method with an out-of-block declaration. When the return type of the out-of-block declaration is defined within the class, the class scope resolution operator shall be used to indicate the internal return type.

*Example:*

```
typedef real T;

class C;
   typedef int T;
   extern function T f();
   extern function real f2();
endclass

function C::T C::f();   // the return must use the class scope resolution
                        // operator, since the type is defined within the
                        // class
   return 1;
endfunction

function real C::f2();
   return 1.0;
endfunction
```

An out-of-block method declaration shall be able to access all declarations of the class in which the corresponding prototype is declared. Following normal resolution rules, the prototype has access to class types only if they are declared prior to the prototype. It shall be an error if an identifier referenced in the prototype does not resolve to the same declaration as the declaration resolved for the corresponding identifier in the out-of-block method declaration's header.

*Example:*

```
typedef int T;
class C;
   extern function void f(T x);
   typedef real T;
endclass

function void C::f(T x);
endfunction
```

In this example, identifier `T` in the prototype for method `f` resolves to the declaration of `T` in the outer scope. In the out-of-block declaration for method `f` the identifier `T` resolves to `C::T` since the out-of-block declaration has visibility to all types in class `C`. Since the resolution of `T` in the out-of-block declaration does not match the resolution in the prototype, an error shall be reported.

## 8.25 Parameterized classes

It is often useful to define a generic class whose objects can be instantiated to have different array sizes or data types. This avoids writing similar code for each size or type and allows a single specification to be used for objects that are fundamentally different and (like a templated class in C++) not interchangeable.

The SystemVerilog parameter mechanism is used to parameterize a class:

```
class vector #(int size = 1);
   bit [size-1:0] a;
endclass
```

Instances of this class can then be instantiated like modules or interfaces:

```
   vector #(10) vten;         // object with vector of size 10
   vector #(.size(2)) vtwo;   // object with vector of size 2
   typedef vector#(4) Vfour;  // Class with vector of size 4
```

This feature is particularly useful when using types as parameters:

```
class stack #(type T = int);
   local T items[];
   task push( T a ); ... endtask
   task pop( ref T a ); ... endtask
endclass
```

The preceding class defines a generic *stack* class, which can be instantiated with any arbitrary type:

```
stack is;                  // default: a stack of ints
stack#(bit[1:10]) bs;      // a stack of 10-bit vectors
stack#(real) rs;           // a stack of real numbers
```

Any type can be supplied as a parameter, including a user-defined type such as a **class** or **struct**.

The combination of a generic class and the actual parameter values is called a *specialization*. Each specialization of a class has a separate set of **static** member variables (this is consistent with C++ templated classes). To share static member variables among several class specializations, they shall be placed in a nonparameterized base class.

```
class vector #(int size = 1);
   bit [size-1:0] a;
   static int count = 0;
   function void disp_count();
      $display( "count: %d of size %d", count, size );
   endfunction
endclass
```

The variable `count` in the preceding example can only be accessed by the corresponding `disp_count` method. Each specialization of the class `vector` has its own unique copy of `count`.

A specialization is the combination of a specific generic class with a unique set of parameters. Two sets of parameters shall be unique unless all parameters are the same, as defined by the following rules:

a) A parameter is a type parameter and the two types are matching types.

b) A parameter is a value parameter and both their type and their value are the same.

All matching specializations of a particular generic class shall represent the same type. The set of matching specializations of a generic class is defined by the context of the class declaration. Because generic classes in a package are visible throughout the system, all matching specializations of a package generic class are the same type. In other contexts, such as modules or programs, each instance of the scope containing the generic class declaration creates a unique generic class, thus defining a new set of matching specializations.

A generic class is not a type; only a concrete specialization represents a type. In the preceding example, the class vector becomes a concrete type only when it has had parameters applied to it, for example:

```
typedef vector my_vector;  // use default size of 1
vector#(6) vx;             // use size 6
```

To avoid having to repeat the specialization either in the declaration or to create parameters of that type, a **`typedef`** should be used:

```
typedef vector#(4) Vfour;
typedef stack#(Vfour) Stack4;
Stack4 s1, s2;                // declare objects of type Stack4
```

A parameterized class can extend another parameterized class. For example:

```
class C #(type T = bit); ... endclass              // base class
class D1 #(type P = real) extends C;               // T is bit (the default)
class D2 #(type P = real) extends C #(integer);    // T is integer
class D3 #(type P = real) extends C #(P);          // T is P
class D4 #(type P = C#(real)) extends P;           // for default, T is real
```

Class `D1` extends the base class `C` using the base class's default type (**`bit`**) parameter. Class `D2` extends the base class `C` using an **`integer`** parameter. Class `D3` extends the base class `C` using the parameterized type (`P`) with which the extended class is parameterized. Class `D4` extends the base class specified by the type parameter `P`.

When a type parameter or typedef name is used as a base class, as in class `D4` above, the name shall resolve to a class type after elaboration.

The default specialization of a parameterized class is the specialization of the parameterized class with an empty parameter override list. For a parameterized class `C`, the default specialization is `C#()`. Other than as the prefix of the scope resolution operator, use of the unadorned name of a parameterized class shall denote the default specialization of the class. Not all parameterized classes have a default specialization since it is legal for a class to not provide parameter defaults. In that case all specializations shall override at least those parameters with no defaults.

*Example:*

```
class C #(int p = 1);
   ...
endclass
class D #(int p);
   ...
endclass

C obj;   // legal; equivalent to "C#() obj";
D obj;   // illegal; D has no default specialization
```

### 8.25.1 Class scope resolution operator for parameterized classes

Use of the class scope resolution operator with a prefix that is the unadorned name of a parameterized class (see 8.25) shall be restricted to use within the scope of the named parameterized class and within its out-of-block declarations (see 8.24). In such cases, the unadorned name of the parameterized class does not denote the default specialization but is used to unambiguously refer to members of the parameterized class. When referring to the default specialization as the prefix to the class scope resolution operator, the explicit default specialization form of `#()` shall be used.

Outside the context of a parameterized class or its out-of-block declarations, the class scope resolution operator may be used to access any of the class parameters. In such a context, the explicit specialization form shall be used; the unadorned name of the parameterized class shall be illegal. The explicit specialization form may denote a specific parameter or the default specialization form. The class scope resolution operator may access value as well as type parameters that are either local or parameters to the class.

*Example:*

```
class C #(int p = 1);
   parameter int q = 5;  // local parameter
   static task t;
      int p;
      int x = C::p;  // C::p disambiguates p
                     // C::p is not p in the default specialization
   endtask
endclass

int x = C::p;        // illegal; C:: is not permitted in this context
int y = C#()::p;     // legal; refers to parameter p in the default
                     // specialization of C
typedef C T;         // T is a default specialization, not an alias to
                     // the name "C"
int z = T::p;        // legal; T::p refers to p in the default specialization
int v = C#(3)::p;    // legal; parameter p in the specialization of C#(3)
int w = C#()::q;     // legal; refers to the local parameter
T obj = new();
int u = obj.q;       // legal; refers to the local parameter
bit arr[obj.q];      // illegal: local parameter is not a constant expression
```

In the context of a parameterized class method out-of-block declaration, use of the class scope resolution operator shall be a reference to the name as though it was made inside the parameterized class; no specialization is implied.

*Example:*

```
class C #(int p = 1, type T = int);
   extern static function T f();
endclass

function C::T C::f();
   return p + C::p;
endfunction

initial $display("%0d %0d", C#()::f(),C#(5)::f()); // output is "2 10"
```

## 8.26 Interface classes

A set of classes may be created that can be viewed as all having a common set of behaviors. Such a common set of behaviors may be created using *interface classes*. An interface class makes it unnecessary for related classes to share a common abstract superclass or for that superclass to contain all method definitions needed by all subclasses. A non-interface class can be declared as implementing one or more interface classes. This creates a requirement for the non-interface class to provide implementations for a set of methods that shall satisfy the requirements of a virtual method override (see 8.20).

An **interface class** shall only contain pure virtual methods (see 8.21), type declarations (see 6.18), and parameter declarations (see 6.20, 8.25). Constraint blocks, covergroups, and nested classes (see 8.23) shall not be allowed in an interface class. An interface class shall not be nested within another class. An interface class can inherit from one or more interface classes through the **extends** keyword, meaning that it inherits all the member types, pure virtual methods and parameters of the interface classes it extends, except for any member types and parameters that it may hide. In the case of multiple inheritance, name conflicts may occur that must be resolved (see 8.26.6).

Classes can implement one or more interface classes through the **implements** keyword. No member types or parameters are inherited through the **implements** keyword. A subclass implicitly implements all of the interface classes implemented by its superclass. In the following example, class C implicitly implements interface class A and has all of the requirements and capabilities as if it explicitly implemented interface class A:

```
interface class A;
endclass

class B implements A;
endclass

class C extends B;
endclass
```

Each pure virtual method from an interface class shall have a virtual method implementation in order to be implemented by a non-abstract class. When an interface class is implemented by a class, the required implementations of interface class methods may be provided by inherited virtual method implementations. A **virtual class** shall define or inherit a **pure virtual** method prototype or **virtual** method implementation for each **pure virtual** method prototype in each implemented **interface class**. The keyword **virtual** shall be used unless the virtual method is inherited.

A variable whose declared type is an interface class type may have as its value a reference to any instance of a class that implements the specified interface class (see 8.22). It is not sufficient that a class provides implementations for all the pure virtual methods of an interface class; the class or one of its superclasses shall be declared to implement the interface class through the **implements** keyword, or else the class does not implement the interface class.

The following is a simple example of interface classes.

```
interface class PutImp#(type PUT_T = logic);
   pure virtual function void put(PUT_T a);
endclass

interface class GetImp#(type GET_T = logic);
   pure virtual function GET_T get();
endclass
```

```
class Fifo#(type T = logic, int DEPTH=1) implements PutImp#(T), GetImp#(T);
   T myFifo [$:DEPTH-1];
   virtual function void put(T a);
      myFifo.push_back(a);
   endfunction
   virtual function T get();
      get = myFifo.pop_front();
   endfunction
endclass

class Stack#(type T = logic, int DEPTH=1) implements PutImp#(T), GetImp#(T);
   T myFifo [$:DEPTH-1];
   virtual function void put(T a);
      myFifo.push_front(a);
   endfunction
   virtual function T get();
      get = myFifo.pop_front();
   endfunction
endclass
```

The example has two interface classes, `PutImp` and `GetImp`, which contain prototype pure virtual methods `put` and `get`. The `Fifo` and `Stack` classes use the keyword **`implements`** to implement the `PutImp` and `GetImp` interface classes and they provide implementations for `put` and `get`. These classes therefore share common behaviors without sharing a common implementation.

### 8.26.1 Interface class syntax

```
interface_class_declaration ::=                                          // from A.1.2
      interface class class_identifier [ parameter_port_list ]
         [ extends interface_class_type { , interface_class_type } ] ;
         { interface_class_item }
      endclass [ : class_identifier]
interface_class_item ::=
      type_declaration
    | { attribute_instance } interface_class_method
    | local_parameter_declaration ;
    | parameter_declaration[7] ;
    | ;
interface_class_method ::=
      pure virtual method_prototype ;
```

[7]) In a parameter_declaration that is a class_item, the **`parameter`** keyword shall be a synonym for the **`localparam`** keyword.

*Syntax 8-3—Class syntax (excerpt from Annex A)*

### 8.26.2 Extends versus implements

Conceptually **`extends`** is a mechanism to add to or modify the behavior of a superclass while **`implements`** is a requirement to provide implementations for the pure virtual methods in an interface class. When a class is extended, all members of the class are inherited into the subclass. When an interface class is implemented, nothing is inherited.

An interface class may extend, but not implement, one or more interface classes, meaning that the interface subclass inherits members from multiple interface classes and may add additional member types, pure virtual method prototypes, and parameters. A class or virtual class may implement, but not extend, one or more interface classes. Because virtual classes are abstract, they are not required to fully define the methods from their implemented classes (see 8.26.7). The following highlights these differences:

— An interface class
  - may extend zero or more interface classes
  - may not implement an interface class
  - may not extend a class or virtual class
  - may not implement a class or virtual class

— A class or virtual class
  - may not extend an interface class
  - may implement zero or more interface classes
  - may extend at most one other class or virtual class
  - may not implement a class or virtual class
  - may simultaneously extend a class and implement interface classes

In the following example, a class is both extending a base class and implementing two interface classes:

```
interface class PutImp#(type PUT_T = logic);
   pure virtual function void put(PUT_T a);
endclass

interface class GetImp#(type GET_T = logic);
   pure virtual function GET_T get();
endclass

class MyQueue #(type T = logic, int DEPTH = 1);
   T PipeQueue[$:DEPTH-1];
   virtual function void deleteQ();
      PipeQueue.delete();
   endfunction
endclass

class Fifo #(type T = logic, int DEPTH = 1)
      extends MyQueue#(T, DEPTH)
      implements PutImp#(T), GetImp#(T);
   virtual function void put(T a);
      PipeQueue.push_back(a);
   endfunction
   virtual function T get();
      get = PipeQueue.pop_front();
   endfunction
endclass
```

In this example, the `PipeQueue` property and `deleteQ` method are inherited in the `Fifo` class. In addition the `Fifo` class is also implementing the `PutImp` and `GetImp` interface classes so it shall provide implementations for the `put` and `get` methods, respectively.

The following example demonstrates that multiple types can be parameterized in the class definition and the resolved types used in the implemented classes `PutImp` and `GetImp`.

```
virtual class XFifo#(type T_in = logic, type T_out = logic, int DEPTH = 1)
                     extends MyQueue#(T_out)
                     implements PutImp#(T_in), GetImp#(T_out);
```

```
      pure virtual function T_out translate(T_in a);
      virtual function void put(T_in a);
         PipeQueue.push_back(translate(a));
      endfunction
      virtual function T_out get();
         get = PipeQueue.pop_front();
      endfunction
   endclass
```

An inherited virtual method can provide the implementation for a method of an implemented interface class. Here is an example:

```
   interface class IntfClass;
      pure virtual function bit funcBase();
      pure virtual function bit funcExt();
   endclass

   class BaseClass;
      virtual function bit funcBase();
         return (1);
      endfunction
   endclass

   class ExtClass extends BaseClass implements IntfClass;
      virtual function bit funcExt();
         return (0);
      endfunction
   endclass
```

`ExtClass` fulfills its requirement to implement `IntfClass` by providing an implementation of `funcExt` and by inheriting an implementation of `funcBase` from `BaseClass`.

An inherited non-virtual method does not provide an implementation for a method of an implemented interface class.

```
   interface class IntfClass;
      pure virtual function void f();
   endclass

   class BaseClass;
      function void f();
         $display("Called BaseClass::f()");
      endfunction
   endclass

   class ExtClass extends BaseClass implements IntfClass;
      virtual function void f();
         $display("Called ExtClass::f()");
      endfunction
   endclass
```

The non-virtual function `f()` in `BaseClass` does not fulfill the requirement to implement `IntfClass`. The implementation of `f()` in `ExtClass` simultaneously hides the `f()` of `BaseClass` and fulfills the requirement to implement `IntfClass`.

### 8.26.3 Type access

Parameters and typedefs within an interface class are inherited by extending interface classes, but are not inherited by implementing interface classes. All parameters and typedefs within an interface class are static and can be accessed through the class scope resolution operator `::` (see 8.23). Accessing parameters through an interface class handle has the same restrictions as accessing parameters through a class handle (see 8.5).

*Example 1*: Types and parameter declarations are inherited by **`extends`**.

```
interface class IntfA #(type T1 = logic);
   typedef T1[1:0] T2;
   pure virtual function T2 funcA();
endclass : IntfA

interface class IntfB #(type T = bit) extends IntfA #(T);
   pure virtual function T2 funcB(); // legal, type T2 is inherited
endclass : IntfB
```

*Example 2:* Type and parameter declarations are not inherited by **`implements`** and must be specified with the class scope resolution operator.

```
interface class IntfC;
   typedef enum {ONE, TWO, THREE} t1_t;
   pure virtual function t1_t funcC();
endclass : IntfC

class ClassA implements IntfC;
   t1_t t1_i;     // error, t1_t is not inherited from IntfC
   virtual function IntfC::t1_t funcC();     // correct
      return (IntfC::ONE);                    // correct
   endfunction : funcC
endclass : ClassA
```

### 8.26.4 Type usage restrictions

A class shall not implement a type parameter, nor shall an interface class extend a type parameter, even if the type parameter resolves to an interface class. The following examples illustrate this restriction and are illegal:

```
class Fifo #(type T = PutImp) implements T;
virtual class Fifo #(type T = PutImp) implements T;
interface class Fifo #(type T = PutImp) extends T;
```

A class shall not implement a forward typedef for an interface class. An interface class shall not extend from a forward typedef of an interface class. An interface class shall be declared before it is implemented or extended.

```
typedef interface class IntfD;

class ClassB implements IntfD #(bit);     // illegal
   virtual function bit[1:0] funcD();
endclass : ClassB

// This interface class declaration must be declared before ClassB
interface class IntfD #(type T1 = logic);
   typedef T1[1:0] T2;
   pure virtual function T2 funcD();
```

```
endclass : IntfD
```

### 8.26.5 Casting and object reference assignment

It shall be legal to assign an object handle to a variable of an interface class type that the object implements.

```
class Fifo #(type T = int) implements PutImp#(T), GetImp#(T);
endclass
Fifo#(int) fifo_obj = new;
PutImp#(int) put_ref = fifo_obj;
```

It shall be legal to dynamically cast between interface class variables if the actual class handle is valid to assign to the destination.

```
GetImp#(int) get_ref;
Fifo#(int) fifo_obj = new;
PutImp#(int) put_ref = fifo_obj;
$cast(get_ref, put_ref);
```

In the preceding, `put_ref` is an instance of `Fifo#(int)` that implements `GetImp#(int)`. It shall also be legal to cast from an object handle to an interface class type handle if the actual object implements the interface class type.

```
$cast(fifo_obj, put_ref); // legal
$cast(put_ref, fifo_obj); // legal, but casting is not required
```

Like abstract classes, an object of an interface class type shall not be constructed.

```
put_ref = new(); // illegal
```

Casting from a source interface class handle that is **null** is handled in the same manner as casting from a source class handle that is **null** (see 8.16).

### 8.26.6 Name conflicts and resolution

When a class implements multiple interface classes, or when an **interface class** extends multiple interface classes, identifiers are merged from different name spaces into a single name space. When this occurs, it is possible that the same identifier name from multiple name spaces may be simultaneously visible in a single name space creating a name conflict that must be resolved.

#### 8.26.6.1 Method name conflict resolution

It is possible that an interface class may inherit multiple methods, or a class may be required through **implements** to provide an implementation of multiple methods, where these methods have the same name. This is a method name conflict. A method name conflict shall be resolved with a single method prototype or implementation that simultaneously provides an implementation for all pure virtual methods of the same name of any implemented interface class. That method prototype or implementation must also be a valid virtual method override (see 8.20) for any inherited method of the same name.

*Example:*

```
interface class IntfBase1;
   pure virtual function bit funcBase();
endclass

interface class IntfBase2;
```

```
   pure virtual function bit funcBase();
endclass

virtual class ClassBase;
   pure virtual function bit funcBase();
endclass

class ClassExt extends ClassBase implements IntfBase1, IntfBase2;
   virtual function bit funcBase();
      return (0);
   endfunction
endclass
```

Class `ClassExt` provides an implementation of `funcBase` that overrides the pure virtual method prototype from `ClassBase` and simultaneously provides an implementation for `funcBase` from both `IntfBase1` and `IntfBase2`.

There are cases in which a method name conflict cannot be resolved.

*Example:*

```
interface class IntfBaseA;
   pure virtual function bit funcBase();
endclass

interface class IntfBaseB;
   pure virtual function string funcBase();
endclass

class ClassA implements IntfBaseA, IntfBaseB;
   virtual function bit funcBase();
      return (0);
   endfunction
endclass
```

In this case, `funcBase` is prototyped in both `IntfBaseA` and `IntfBaseB` but with different return types, **`bit`** and **`string`** respectively. Although the implementation of `funcBase` is a valid override of `IntfBaseA::funcBase`, it is not simultaneously a valid override of the prototype of `IntfBaseB::funcBase`, so an error shall occur.

### 8.26.6.2 Parameter and type declaration inheritance conflicts and resolution

Interface classes may inherit parameters and type declarations from multiple interface classes. A name collision will occur if the same name is inherited from different interface classes. The subclass shall provide parameter and/or type declarations that override all such name collisions.

*Example:*

```
interface class PutImp#(type T = logic);
   pure virtual function void put(T a);
endclass

interface class GetImp#(type T = logic);
   pure virtual function T get();
endclass

interface class PutGetIntf#(type TYPE = logic)
```

```
                 extends PutImp#(TYPE), GetImp#(TYPE);
   typedef TYPE T;
endclass
```

In the preceding example, the parameter `T` is inherited from both `PutImp` and `GetImp`. A conflict occurs despite the fact that `PutImp::T` matches `GetImp::T` and is never used by `PutGetIntf`. `PutGetIntf` overrides `T` with a type definition to resolve the conflict.

#### 8.26.6.3 Diamond relationship

A *diamond relationship* occurs if an interface class is implemented by the same class or inherited by the same interface class in multiple ways. In the case of a diamond relationship, only one copy of the symbols from any single interface class will be merged so as to avoid a name conflict. For example:

```
interface class IntfBase;
   parameter SIZE = 64;
endclass

interface class IntfExt1 extends IntfBase;
   pure virtual function bit funcExt1();
endclass

interface class IntfExt2 extends IntfBase;
   pure virtual function bit funcExt2();
endclass

interface class IntfExt3 extends IntfExt1, IntfExt2;
endclass
```

In the preceding example, the class `IntfExt3` inherits the parameter `SIZE` from `IntfExt1` and `IntfExt2`. Since these parameters originate from the same interface class, `IntfBase`, only one copy of `SIZE` shall be inherited into `IntfExt3` so it shall not be considered a conflict.

Each unique parameterization of a parameterized interface class is an interface class specialization. Each interface class specialization is considered as though it is a unique interface class type. Therefore, there is no diamond relationship if different specializations of the same parameterized interface class are inherited by the same interface class or implemented by the same class. As a result, method name conflicts as described in 8.26.6.1 and parameter and type declaration name conflicts as described in 8.26.6.2 may occur. For example:

```
interface class IntfBase #(type T = int);
   pure virtual function bit funcBase();
endclass

interface class IntfExt1 extends IntfBase#(bit);
   pure virtual function bit funcExt1();
endclass

interface class IntfExt2 extends IntfBase#(logic);
   pure virtual function bit funcExt2();
endclass

interface class IntfFinal extends IntfExt1, IntfExt2;
   typedef bit T; // Override the conflicting identifier name
   pure virtual function bit funcBase();
endclass
```

In the preceding example, there are two different parameterizations of the interface class `IntfBase`. Each of these parameterizations of `IntfBase` is a specialization; therefore there is no diamond relationship and there are conflicts of the parameter `T` and method `funcBase` that must be resolved.

### 8.26.7 Partial implementation

It is possible to create classes that are not fully defined and that take advantage of interface classes through the use of virtual classes (see 8.21). Because virtual classes do not have to fully define their implementation, they are free to partially define their methods. The following is an example of a partially implemented virtual class.

```
interface class IntfClass;
   pure virtual function bit funcA();
   pure virtual function bit funcB();
endclass

// Partial implementation of IntfClass
virtual class ClassA implements IntfClass;
   virtual function bit funcA();
      return (1);
   endfunction
   pure virtual function bit funcB();
endclass

// Complete implementation of IntfClass
class ClassB extends ClassA;
   virtual function bit funcB();
      return (1);
   endfunction
endclass
```

It shall be illegal to use an interface class to partially define a virtual class without fulfilling the interface class prototype requirements. In other words, when an interface class is implemented by a virtual class, the virtual class must do one of the following for each interface class method prototype:

— Provide a method implementation
— Re-declare the method prototype with the **pure** qualifier

In the preceding example `ClassA` fully defines `funcA`, but re-declares the prototype `funcB`.

### 8.26.8 Method default argument values

Method declarations within interface classes may have default argument values. The default expression shall be a constant expression and is evaluated in the scope containing the subroutine declaration. The value of the constant expression shall be the same for all the classes that implement the method. See 13.5.3 for more information.

### 8.26.9 Constraint blocks, covergroups, and randomization

Constraint blocks and covergroups shall not be declared in interface classes.

A `randomize` method call shall be legal with interface class handles. While in-line constraints shall also be legal, interface classes cannot contain any data meaning that in-line constraints will only be able to express conditions related to state variables and are therefore of very limited utility. Use of `rand_mode` and `constraint_mode` shall not be legal as a consequence of the name resolution rules and the fact that interface classes are not permitted to contain data members.

Interface classes contain two built-in empty virtual methods `pre_randomize()` and `post_randomize()` that are automatically called before and after randomization. These methods can be overridden. As a special case, `pre_randomize()` and `post_randomize()` shall not cause method name conflicts.

## 8.27 Typedef class

Sometimes a class variable needs to be declared before the class itself has been declared; for example, if two classes each need a handle to the other. When, in the course of processing the declaration for the first class, the compiler encounters the reference to the second class, that reference is undefined and the compiler flags it as an error.

This is resolved using **typedef** to provide a forward declaration for the second class:

```
typedef class C2;       // C2 is declared to be of type class
class C1;
      C2 c;
endclass
class C2;
   C1 c;
endclass
```

In this example, `C2` is declared to be of type **class**, a fact that is reinforced later in the source code. The **class** construct always creates a type and does not require a **typedef** declaration for that purpose (as in **typedef class** …).

In the preceding example, the **class** keyword in the statement **typedef class** `C2;` is not necessary and is used only for documentation purposes. The statement **typedef** `C2;` is equivalent and shall work the same way.

As with other forward typedefs as described in 6.18, the actual class definition of a forward class declaration shall be resolved within the same local scope or generate block.

A forward **typedef** to a class may refer to a class with a parameter port list.

*Example:*

```
typedef class C ;
module top ;
   C#(1, real) v2 ;           // positional parameter override
   C#(.p(2), .T(real)) v3 ;   // named parameter override
endmodule

class C #(parameter p = 2, type T = int);
endclass
```

## 8.28 Classes and structures

On the surface, it might appear that **class** and **struct** provide equivalent functionality, and only one of them is needed. However, that is not true; **class** differs from **struct** in the following three fundamental ways:

a) SystemVerilog structs are strictly static objects; they are created either in a static memory location (global or module scope) or on the stack of an automatic task. Conversely, SystemVerilog objects (i.e., class instances) are exclusively dynamic; their declaration does not create the object. Creating an object is done by calling **new**.

b) SystemVerilog objects are implemented using handles, thereby providing C-like pointer functionality. But SystemVerilog disallows casting handles onto other data types; thus, SystemVerilog handles do not have the risks associated with C pointers.

c) SystemVerilog objects form the basis of an Object-Oriented data abstraction that provides true polymorphism. Class inheritance, abstract classes, and dynamic casting are powerful mechanisms, which go way beyond the mere encapsulation mechanism provided by structs.

## 8.29 Memory management

Memory for objects, strings, and dynamic and associative arrays is allocated dynamically. When objects are created, SystemVerilog allocates more memory. When an object is no longer needed, SystemVerilog automatically reclaims the memory, making it available for reuse. The automatic memory management system is an integral part of SystemVerilog. Without automatic memory management, SystemVerilog's multithreaded, reentrant environment creates many opportunities for users to run into problems. A manual memory management system, such as the one provided by C's `malloc` and `free`, would not be sufficient.

Consider the following example:

```
myClass obj = new;
fork
   task1( obj );
   task2( obj );
join_none
```

In this example, the main process (the one that forks off the two tasks) does not know when the two processes might be done using the object `obj`. Similarly, neither `task1` nor `task2` knows when any of the other two processes will no longer be using the object `obj`. It is evident from this simple example that no single process has enough information to determine when it is safe to free the object. The only two options available to the user are as follows:

— Play it safe and never reclaim the object, or
— Add some form of reference count that can be used to determine when it might be safe to reclaim the object.

Adopting the first option can cause the system to quickly run out of memory. The second option places a large burden on users, who, in addition to managing their testbench, must also manage the memory using less than ideal schemes. To avoid these shortcomings, SystemVerilog manages all dynamic memory automatically.

Users do not need to worry about dangling references, premature deallocation, or memory leaks. The system shall automatically reclaim any object that is no longer being used. In the preceding example, all that users do is assign **`null`** to all the variables referencing handle `obj` when they no longer need it. An object shall not be reclaimed while there are outstanding references to that object in any active scope, or pending nonblocking assignments to non-static members of that object.

# 9. Processes

## 9.1 General

This clause describes the following:

— Structured procedures (initial procedures, always procedures, final procedures)
— Block statements (begin-end sequential blocks, fork-join parallel blocks)
— Timing control (delays, events, waits, intra-assignment)
— Process threads and process control

## 9.2 Structured procedures

All structured procedures in SystemVerilog are specified within one of the following constructs:

— *initial* procedure, denoted with the keyword **`initial`** (see 9.2.1)
— *always* procedure, denoted with the keywords:
  - **`always`** (see 9.2.2.1)
  - **`always_comb`** (see 9.2.2.2)
  - **`always_latch`** (see 9.2.2.3)
  - **`always_ff`** (see 9.2.2.4)

— *final* procedure, denoted with the keyword **`final`** (see 9.2.3)
— Task
— Function

The syntax for these structured procedures is shown in Syntax 9-1.

```
initial_construct ::= initial statement_or_null                                // from A.6.2
always_construct ::= always_keyword statement
always_keyword ::= always | always_comb | always_latch | always_ff
final_construct ::= final function_statement
function_declaration ::= function [ lifetime ] function_body_declaration       // from A.2.6
task_declaration ::= task [ lifetime ] task_body_declaration                   // from A.2.7
```

*Syntax 9-1—Syntax for structured procedures (excerpt from Annex A)*

The initial and always procedures are enabled at the beginning of a simulation. The initial procedure shall execute only once, and its activity shall cease when the statement has finished. In contrast, an always procedure shall execute repeatedly, and its activity shall cease only when the simulation is terminated.

There shall be no implied order of execution between initial and always procedures. The initial procedures need not be scheduled and executed before the always procedures. There shall be no limit to the number of initial and always procedures that can be defined in a module. See 6.8 for the order of variable initialization relative to the execution of procedures.

The final procedures are enabled at the end of simulation time and execute only once.

Tasks and functions are procedures that are enabled from one or more places in other procedures. Tasks and functions are described in Clause 13.

In addition to these structured procedures, SystemVerilog contains other procedural contexts, such as coverage point expressions (19.5), assertion sequence match items (16.10, 16.11), and action blocks (16.14).

SystemVerilog has the following types of control flow within a procedure:

— Selection, loops, and jumps (see Clause 12)
— Subroutine calls (see Clause 13)
— Sequential and parallel blocks (see 9.3)
— Timing control (see 9.4)
— Process control (see 9.5 through 9.7)

### 9.2.1 Initial procedures

An **initial** procedure shall execute only once, and its activity shall cease when the statement has finished.

The following example illustrates use of an initial procedure for initialization of variables at the start of simulation.

```
initial begin
   a = 0;      // initialize a
   for (int index = 0; index < size; index++)
          memory[index] = 0; // initialize memory word
end
```

Another typical usage of the initial procedure is specification of waveform descriptions that execute once to provide stimulus to the main part of the circuit being simulated.

```
initial begin
   inputs = 'b000000;     // initialize at time zero
   #10 inputs = 'b011001; // first pattern
   #10 inputs = 'b011011; // second pattern
   #10 inputs = 'b011000; // third pattern
   #10 inputs = 'b001000; // last pattern
end
```

### 9.2.2 Always procedures

There are four forms of always procedures: **always**, **always_comb**, **always_latch**, and **always_ff**. All forms of always procedures repeat continuously throughout the duration of the simulation.

#### 9.2.2.1 General purpose always procedure

The **always** keyword represents a general purpose always procedure, which can be used to represent repetitive behavior such as clock oscillators. The construct can also be used with proper timing controls to represent combinational, latched, and sequential hardware behavior.

The general purpose **always** procedure, because of its looping nature, is only useful when used in conjunction with some form of timing control. If an **always** procedure has no control for simulation time to advance, it will create a simulation deadlock condition.

The following code, for example, creates a zero-delay infinite loop:

```
always areg = ~areg;
```

Providing a timing control to the preceding code creates a potentially useful description as shown in the following:

```
always #half_period areg = ~areg;
```

#### 9.2.2.2 Combinational logic always_comb procedure

SystemVerilog provides a special **always_comb** procedure for modeling combinational logic behavior. For example:

```
always_comb
   a = b & c;

always_comb
   d <= #1ns b & c;
```

The **always_comb** procedure provides functionality that is different from the general purpose **always** procedure, as follows:

— There is an inferred sensitivity list that includes the expressions defined in 9.2.2.2.1.

— The variables written on the left-hand side of assignments shall not be written to by any other process. However, multiple assignments made to independent elements of a variable are allowed as long as their longest static prefixes do not overlap (see 11.5.3). For example, an unpacked structure or array can have one bit assigned by an **always_comb** procedure and another bit assigned continuously or by another **always_comb** procedure, etc. See 6.5 for more details.

— The procedure is automatically triggered once at time zero, after all **initial** and **always** procedures have been started so that the outputs of the procedure are consistent with the inputs.

Software tools should perform additional checks to warn if the behavior within an **always_comb** procedure does not represent combinational logic, such as if latched behavior can be inferred.

##### 9.2.2.2.1 Implicit always_comb sensitivities

The implicit sensitivity list of an **always_comb** includes the expansions of the longest static prefix of each variable or select expression that is read within the block or within any function called within the block with the following exceptions:

a) Any expansion of a variable declared within the block or within any function called within the block

b) Any expression that is also written within the block or within any function called within the block

For the definition of the longest static prefix, see 11.5.3.

Hierarchical function calls and function calls from packages are analyzed as normal functions, as are calls to static method functions referenced with the class scope resolution operator (see 8.23). References to class objects and method calls of class objects do not add anything to the sensitivity list of an **always_comb**, except for any contributions from argument expressions passed to these method calls.

Task calls are allowed in an **always_comb**, but the contents of the tasks do not add anything to the sensitivity list.

NOTE—A task that does not consume time may be replaced by a void function so that the contents will be analyzed for sensitivity.

An expression used in an immediate assertion (see 16.3) within the procedure, or in any function called within the procedure, contributes to the implicit sensitivity list of an **always_comb** as if that expression were used as a condition of an **if** statement. Expressions used in assertion action blocks do not contribute to the implicit sensitivity list of an **always_comb**. In the following example, the **always_comb** shall trigger whenever b, c or e change.

```
always_comb
begin
   a = b & c;
   A1:assert (a != e) else if (!disable_error) $error("failed");
end
```

#### 9.2.2.2.2 always_comb compared to always @*

The SystemVerilog **always_comb** procedure differs from **always @*** (see 9.4.2.2) in the following ways:

— **always_comb** automatically executes once at time zero, whereas **always @*** waits until a change occurs on a signal in the inferred sensitivity list.

— **always_comb** is sensitive to changes within the contents of a function, whereas **always @*** is only sensitive to changes to the arguments of a function.

— Variables on the left-hand side of assignments within an **always_comb** procedure, including variables from the contents of a called function, shall not be written to by any other processes, whereas **always @*** permits multiple processes to write to the same variable.

— Statements in an **always_comb** shall not include those that block, have blocking timing or event controls, or fork-join statements.

— **always_comb** is sensitive to expressions in immediate assertions within the procedure and within the contents of a function called in the procedure, whereas **always @*** is sensitive to expressions in immediate assertions within the procedure only.

### 9.2.2.3 Latched logic always_latch procedure

SystemVerilog also provides a special **always_latch** procedure for modeling latched logic behavior. For example:

```
always_latch
   if(ck) q <= d;
```

The **always_latch** construct is identical to the **always_comb** construct except that software tools should perform additional checks and warn if the behavior in an **always_latch** construct does not represent latched logic, whereas in an **always_comb** construct, tools should check and warn if the behavior does not represent combinational logic. All statements in 9.2.2.2 shall apply to **always_latch**.

### 9.2.2.4 Sequential logic always_ff procedure

The **always_ff** procedure can be used to model synthesizable sequential logic behavior. For example:

```
always_ff @(posedge clock iff reset == 0 or posedge reset) begin
   r1 <= reset ? 0 : r2 + 1;
   ...
end
```

The **always_ff** procedure imposes the restriction that it contains one and only one event control and no blocking timing controls. Variables on the left-hand side of assignments within an **always_ff** procedure, including variables from the contents of a called function, shall not be written to by any other process.

Software tools should perform additional checks to warn if the behavior within an **always_ff** procedure does not represent sequential logic.

### 9.2.3 Final procedures

The **`final`** procedure is like an **`initial`** procedure, defining a procedural block of statements, except that it occurs at the end of simulation time and executes without delays. A **`final`** procedure is typically used to display statistical information about the simulation.

The only statements allowed inside a **`final`** procedure are those permitted inside a function declaration, so that they execute within a single simulation cycle. Unlike an **`initial`** procedure, the **`final`** procedure does not execute as a separate process; instead, it executes in zero time, as a series of function calls from a single process. All **`final`** procedures shall execute in an arbitrary order. No remaining scheduled events shall execute after all final procedures have executed.

A **`final`** procedure executes when simulation ends due to an explicit or implicit call to `$finish`.

```
final
   begin
      $display("Number of cycles executed %d",$time/period);
      $display("Final PC = %h",PC);
   end
```

Execution of `$finish`, `tf_dofinish()`, or `vpi_control(vpiFinish,...)` from within a **`final`** procedure shall cause the simulation to end immediately. A **`final`** procedure can only trigger once in a simulation.

A **`final`** procedure shall execute before any PLI callbacks that indicate the end of simulation.

SystemVerilog **`final`** procedures execute in an arbitrary but deterministic sequential order. This is possible because **`final`** procedures are limited to the legal set of statements allowed for functions.

NOTE—SystemVerilog does not specify the ordering in which final procedures are executed, but implementations should define rules that preserve the ordering between runs. This helps keep the output log file stable because **`final`** procedures are mainly used for displaying statistics.

## 9.3 Block statements

*Block statements* are a means of grouping statements together so that they act syntactically like a single statement. There are two types of blocks, as follows:

— *Sequential block*, also called *begin-end block*
— *Parallel block*, also called *fork-join block*

The sequential block shall be delimited by the keywords **`begin`** and **`end`**. The procedural statements in a sequential block shall be executed sequentially in the given order.

The parallel block shall be delimited by the keywords **`fork`** and **`join`**, **`join_any`**, or **`join_none`**. The procedural statements in a parallel block shall be executed concurrently.

### 9.3.1 Sequential blocks

A *sequential block* shall have the following characteristics:

— Statements shall be executed in sequence, one after another.
— Delay values for each statement shall be treated relative to the simulation time of the execution of the previous statement.
— Control shall pass out of the block after the last statement executes.

Syntax 9-2 gives the formal syntax for a sequential block.

```
seq_block ::=                                                          // from A.6.3
      begin [ : block_identifier ] { block_item_declaration } { statement_or_null }
      end [ : block_identifier ]
block_item_declaration ::=                                             // from A.2.8
      { attribute_instance } data_declaration
    | { attribute_instance } local_parameter_declaration ;
    | { attribute_instance } parameter_declaration ;
    | { attribute_instance } let_declaration
```

*Syntax 9-2—Syntax for sequential block (excerpt from Annex A)*

*Example 1:* A sequential block enables the following two assignments to have a deterministic result:

```
begin
   areg = breg;
   creg = areg;      // creg stores the value of breg
end
```

The first assignment is performed, and `areg` is updated before control passes to the second assignment.

*Example 2:* An event control (see 9.4.2) can be used in a sequential block to separate the two assignments in time:

```
begin
   areg = breg;
   @(posedge clock) creg = areg; // assignment delayed until
end                              // posedge on clock
```

*Example 3:* The following example shows how the combination of the sequential block and delay control can be used to specify a time-sequenced waveform:

```
parameter d = 50;    // d declared as a parameter and
logic [7:0] r;       // r declared as an 8-bit variable

begin     // a waveform controlled by sequential delays
   #d r = 'h35;
   #d r = 'hE2;
   #d r = 'h00;
   #d r = 'hF7;
end
```

### 9.3.2 Parallel blocks

The fork-join *parallel block* construct enables the creation of concurrent processes from each of its parallel statements. A parallel block shall have the following characteristics:

- Statements shall execute concurrently.
- Delay values for each statement shall be considered relative to the simulation time of entering the block.
- Delay control can be used to provide time-ordering for assignments.
- Control shall pass out of the block when the last time-ordered statement executes based on the type of join keyword.

— Has restricted usage inside function calls (see 13.4).

Syntax 9-3 gives the formal syntax for a parallel block.

```
par_block ::=                                                              // from A.6.3
      fork [ : block_identifier ] { block_item_declaration } { statement_or_null }
      join_keyword [ : block_identifier ]
join_keyword ::= join | join_any | join_none
block_item_declaration ::=                                                 // from A.2.8
      { attribute_instance } data_declaration
    | { attribute_instance } local_parameter_declaration ;
    | { attribute_instance } parameter_declaration ;
    | { attribute_instance } let_declaration
```

*Syntax 9-3—Syntax for parallel block (excerpt from Annex A)*

One or more statements can be specified; each statement shall execute as a concurrent process. The timing controls in a fork-join block do not have to be ordered sequentially in time.

The following example codes the waveform description shown in Example 3 of 9.3.1 by using a parallel block instead of a sequential block. The waveform produced on the variable is exactly the same for both implementations.

```
fork
   #50 r = 'h35;
   #100 r = 'hE2;
   #150 r = 'h00;
   #200 r = 'hF7;
join
```

SystemVerilog provides three choices for specifying when the parent (forking) process resumes execution, which are summarized in Table 9-1.

**Table 9-1—fork-join control options**

| Option | Description |
|---|---|
| join | The parent process blocks until all the processes spawned by this fork complete. |
| join_any | The parent process blocks until any one of the processes spawned by this fork completes. |
| join_none | The parent process continues to execute concurrently with all the processes spawned by the fork. The spawned processes do not start executing until the parent thread executes a blocking statement or terminates. |

When defining a fork-join block, encapsulating the entire fork within a begin-end block causes the entire block to execute as a single process, with each statement executing sequentially.

```
fork
   begin
      statement1;    // one process with 2 statements
      statement2;
   end
join
```

In the following example, two processes are forked. The first one waits for 20 ns and the second one waits for the named event `eventA` to be triggered. Because the **`join`** keyword is specified, the parent process shall block until the two processes complete, i.e., until 20 ns have elapsed and `eventA` has been triggered.

```
fork
   begin
      $display( "First Block\n" );
      # 20ns;
   end
   begin
      $display( "Second Block\n" );
      @eventA;
   end
join
```

A **`return`** statement within the context of a fork-join block is illegal and shall result in a compilation error. For example:

```
task wait_20;
   fork
      # 20;
      return ;    // Illegal: cannot return; task lives in another process
   join_none
endtask
```

Variables declared in the *block_item_declaration* of a fork-join block shall be initialized to their initialization value expression whenever execution enters their scope and before any processes are spawned. Within a **`fork-join_any`** or **`fork-join_none`** block, it shall be illegal to refer to formal arguments passed by reference other than in the initialization value expressions of variables declared in a *block_item_declaration* of the fork. These variables are useful in processes spawned by looping constructs to store unique, per-iteration data. For example:

```
initial
   for( int j = 1; j <= 3; ++j )
      fork
         automatic int k = j; // local copy, k, for each value of j
         #k $write( "%0d", k );
         begin
            automatic int m = j; // the value of m is undetermined
            ...
         end
      join_none
```

The preceding example generates the output 123.

### 9.3.3 Statement block start and finish times

Both sequential and parallel blocks have the notion of a start and finish time. For sequential blocks, the start time is when the first statement is executed, and the finish time is when the last statement has been executed. For parallel blocks, the start time is the same for all the statements, and the finish time is controlled by the type of join construct used (see 9.3.2, Table 9-1).

Sequential and parallel blocks can be embedded within each other, allowing complex control structures to be expressed easily and with a high degree of structure. When blocks are embedded within each other, the timing of when a block starts and finishes is important. Execution shall not continue to the statement following a block until the finish time for the block has been reached, that is, until the block has completely finished executing.

*Example 1:* The following example shows the statements from the example in 9.3.2 written in the reverse order and still producing the same waveform.

```
fork
   #200 r = 'hF7;
   #150 r = 'h00;
   #100 r = 'hE2;
   #50 r = 'h35;
join
```

*Example 2:* When an assignment is to be made after two separate events have occurred, known as the *joining of events*, a **fork-join** block can be useful.

```
begin
   fork
         @Aevent;
         @Bevent;
   join
   areg = breg;
end
```

The two events can occur in any order (or even at the same simulation time), the **fork-join** block will complete once both events have occurred, and the assignment will be made. In contrast, if the **fork-join** block was a **begin-end** block and the `Bevent` occurred before the `Aevent`, then the block would be waiting for the next `Bevent`.

*Example 3:* This example shows two sequential blocks, each of which will execute when its controlling event occurs. Because the event controls are within a **fork-join** block, they execute in parallel, and the sequential blocks can, therefore, also execute in parallel.

```
fork
   @enable_a
         begin
            #ta wa = 0;
            #ta wa = 1;
            #ta wa = 0;
         end
   @enable_b
         begin
            #tb wb = 1;
            #tb wb = 0;
            #tb wb = 1;
         end
join
```

### 9.3.4 Block names

Both sequential and parallel blocks can be named by adding **:** `name_of_block` after the keywords **begin** or **fork**. A named block creates a new hierarchy scope. The naming of blocks serves the following purposes:

- — It allows local variables, parameters, and named events to be referenced hierarchically, using the block name.
- — It allows the block to be referenced in statements such as the **disable** statement (see 9.6.2).

An unnamed block creates a new hierarchy scope only if it directly contains a block item declaration, such as a variable declaration or a type declaration. This hierarchy scope is unnamed and the items declared in it cannot be hierarchically referenced (see 6.21).

All variables shall be static; that is, a unique location exists for all variables, and leaving or entering blocks shall not affect the values stored in them.

The block names give a means of uniquely identifying all variables at any simulation time.

A matching block name may be specified after the block **end**, **join**, **join_any**, or **join_none** keyword, preceded by a colon. This can help document which **end** or **join**, **join_any**, or **join_none** is associated with which **begin** or **fork** when there are nested blocks. A name at the end of the block is not required. It shall be an error if the name at the end is different from the block name at the beginning.

```
begin: blockB     // block name after the begin or fork
   ...
end: blockB
```

Similarly, a matching block name may be specified after the following block end keywords, preceded by a colon:

— **endchecker** (see 17.2)
— **endclass** (see 8.3)
— **endclocking** (see 14.3)
— **endconfig** (see 33.4)
— **endfunction** (see 13.4)
— **endgroup** (see 19.2)
— **endinterface** (see 25.3)
— **endmodule** (see 23.2.1)
— **endpackage** (see 26.2)
— **endprimitive** (see 29.3)
— **endprogram** (see 24.3)
— **endproperty** (see 16.2)
— **endsequence** (see 16.8)
— **endtask** (see 13.3)

A matching block name may also follow the keyword **end** at the end of a generate block (see 27.3). A name at the end of the block is not required. It shall be an error if the name at the end is different from the block name at the beginning.

### 9.3.5 Statement labels

A label can be specified before any procedural statement (any non-declaration statement that can appear inside a begin-end block), as in C. A statement label is used to identify a single statement. The label name is specified before the statement, followed by a colon.

```
labelA: statement
```

A begin-end or fork-join block is considered a statement and can have a statement label before the block. Specifying a statement label before a **begin** or **fork** keyword is equivalent to specifying a block name after the keyword, and a matching block name may be specified after the block **end**, **join**, **join_any**, or **join_none** keyword. For example:

```
labelB: fork   // label before the begin or fork
   ...
join_none : labelB
```

It shall be illegal to have both a label before a **begin** or **fork** and a block name after the **begin** or **fork**. A label cannot appear before the **end**, **join**, **join_any**, or **join_none**, as these keywords do not form a statement.

A statement label on a **foreach** loop, or on a **for** loop with variables declared as part of the for_initialization, names the implicit block created by the loop. For other types of statements, a statement label creates a named begin-end block around the statement and creates a new hierarchy scope.

A label may also be specified before a generate begin-end block (see 27.3).

A label may also be specified before a concurrent assertion (see 16.5).

A statement with a label can be disabled using a **disable** statement. Disabling a statement shall have the same behavior as disabling a named block. See 9.6.2 on **disable** statements and process control.

## 9.4 Procedural timing controls

SystemVerilog has two types of explicit timing control over when procedural statements can occur. The first type is a *delay control*, in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The delay expression can be a dynamic function of the state of the circuit, or it can be a simple number that separates statement executions in time. The delay control is an important feature when specifying stimulus waveform descriptions. It is described in 9.4.1 and 9.4.5.

The second type of timing control is the *event expression*, which allows statement execution to be delayed until the occurrence of some simulation event occurring in a procedure executing concurrently with this procedure. A simulation event can be a change of value on a net or variable (an *implicit event*) or the occurrence of an explicitly named event that is triggered from other procedures (an *explicit event*). Most often, an event control is a positive or negative edge on a clock signal. Event control is discussed in 9.4.2 through 9.4.5.

The procedural statements encountered so far all execute without advancing simulation time. Simulation time can advance by one of the following three methods:

— A *delay* control, which is introduced by the symbol #
— An *event* control, which is introduced by the symbol @
— The *wait* statement, which operates like a combination of the event control and the while loop

The three procedural timing control methods are discussed in 9.4.1 through 9.4.5. Syntax 9-4 shows the syntax of timing control in procedural statements.

```
procedural_timing_control_statement ::=                          // from A.6.5
    procedural_timing_control  statement_or_null
delay_or_event_control ::=
    delay_control
  | event_control
  | repeat ( expression ) event_control
```

```
delay_control ::=
      # delay_value
    | # ( mintypmax_expression )
event_control ::=
      @ hierarchical_event_identifier
    | @ ( event_expression )
    | @*
    | @ (*)
    | @ ps_or_hierarchical_sequence_identifier
event_expression31 ::=
      [ edge_identifier ] expression [ iff expression ]
    | sequence_instance [ iff expression ]
    | event_expression or event_expression
    | event_expression , event_expression
    | ( event_expression )
procedural_timing_control ::=
      delay_control
    | event_control
    | cycle_delay
...
wait_statement ::=
      wait ( expression ) statement_or_null
    | wait fork ;
    | wait_order ( hierarchical_identifier { , hierarchical_identifier } ) action_block
edge_identifier ::= posedge | negedge | edge                          // from A.7.4
```

31) Parentheses are required when an event expression that contains comma-separated event expressions is passed as an actual argument using positional binding.

*Syntax 9-4—Delay and event control syntax (excerpt from Annex A)*

The gate and net delays also advance simulation time, as discussed in Clause 28.

### 9.4.1 Delay control

A procedural statement following the delay control shall be delayed in its execution with respect to the procedural statement preceding the delay control by the specified delay. If the delay expression evaluates to an unknown or high-impedance value, it shall be interpreted as zero delay. If the delay expression evaluates to a negative value, it shall be interpreted as a two's-complement unsigned integer of the same size as a time variable. Specify parameters are permitted in the delay expression. They can be overridden by SDF annotation, in which case the expression is reevaluated.

*Example 1:* The following example delays the execution of the assignment by 10 time units:

```
#10 rega = regb;
```

*Example 2:* The next three examples provide an expression following the number sign (#). Execution of the assignment is delayed by the amount of simulation time specified by the value of the expression.

```
#d rega = regb;            // d is defined as a parameter
#((d+e)/2) rega = regb;    // delay is average of d and e
#regr regr = regr + 1;     // delay is the value in regr
```

### 9.4.2 Event control

The execution of a procedural statement can be synchronized with a value change on a net or variable or the occurrence of a declared event. The value changes on nets and variables can be used as events to trigger the execution of a statement. This is known as *detecting an implicit event*. The event can also be based on the direction of the change, that is, toward the value 1 (**posedge**) or toward the value 0 (**negedge**). The behavior of posedge and negedge events is shown in Table 9-2 and can be described as follows:

— A *negedge* shall be detected on the transition from 1 to `x`, `z`, or 0, and from `x` or `z` to 0

— A *posedge* shall be detected on the transition from 0 to `x`, `z`, or 1, and from `x` or `z` to 1

**Table 9-2—Detecting posedge and negedge**

| From | To | | | |
|---|---|---|---|---|
| | **0** | **1** | **x** | **z** |
| **0** | No edge | posedge | posedge | posedge |
| **1** | negedge | No edge | negedge | negedge |
| **x** | negedge | posedge | No edge | No edge |
| **z** | negedge | posedge | No edge | No edge |

In addition to **posedge** and **negedge**, a third edge event, **edge**, indicates a change towards either 1 or 0. More precisely, the behavior of an edge event can be described as:

— An *edge* shall be detected whenever *negedge* or *posedge* is detected.

An implicit event shall be detected on any change in the value of the expression. An edge event shall be detected only on the LSB of the expression. A change of value in any operand of the expression without a change in the result of the expression shall not be detected as an event.

The following example shows illustrations of edge-controlled statements:

```
@r rega = regb;   // controlled by any value change in the reg r
@(posedge clock) rega = regb;          // controlled by posedge on clock
forever @(negedge clock) rega = regb;  // controlled by negedge on clock
forever @(edge clock) rega = regb;     // controlled by edge on clock
```

If the expression denotes a **clocking** block **input** or **inout** (see Clause 14), the event control operator uses the synchronous values, that is, the values sampled by the clocking event. The expression can also denote a **clocking** block name (with no edge qualifier) to be triggered by the clocking event.

A variable used with the event control can be any one of the integral data types (see 6.11.1) or string. The variable can be either a simple variable or a **ref** argument (variable passed by reference); it can be a member of an array, associative array, or object (class instance) of the aforementioned types.

Event expressions shall return singular values. Aggregate types can be used in an expression provided the expression reduces to a singular value. The object members or aggregate elements can be any type as long as the result of the expression is a singular value.

If the event expression is a reference to a simple object handle or chandle variable, an event is created when a write to that variable is not equal to its previous value.

Nonvirtual methods of an object and built-in methods or system functions for an aggregate type are allowed in event control expressions as long as the type of the return value is singular and the method is defined as a function, not a task.

Changing the value of object data members, aggregate elements, or the size of a dynamically sized array referenced by a method or function shall cause the event expression to be reevaluated. An implementation can cause the event expression to be reevaluated when changing the value or size even if the members are not referenced by the method or function.

```
real AOR[];                               // dynamic array of reals
byte stream[$];                           // queue of bytes
initial wait(AOR.size() > 0) ....;        // waits for array to be allocated
initial wait($bits(stream) > 60)...;      // waits for total number of bits
                                          // in stream greater than 60

Packet p = new; // Packet 1 -- Packet is defined in 8.2
Packet q = new; // Packet 2
initial fork
   @(p.status);   // Wait for status in Packet 1 to change
   @p;            // Wait for a change to handle p
   # 10 p = q;    // triggers @p.
   // @(p.status) now waits for status in Packet 2 to change,
   // if not already different from Packet 1
join
```

### 9.4.2.1 Event OR operator

The logical OR of any number of events can be expressed so that the occurrence of any one of the events triggers the execution of the procedural statement that follows it. The keyword **or** or a comma character (**,**) is used as an event logical OR operator. A combination of these can be used in the same event expression. Comma-separated sensitivity lists shall be synonymous to **or**-separated sensitivity lists.

The next two examples show the logical or of two and three events, respectively:

```
@(trig or enable) rega = regb; // controlled by trig or enable

@(posedge clk_a or posedge clk_b or trig) rega = regb;
```

The following examples show the use of the comma (**,**) as an event logical **or** operator:

```
always @(a, b, c, d, e)

always @(posedge clk, negedge rstn)

always @(a or b, c, d or e)
```

### 9.4.2.2 Implicit event_expression list

An incomplete event_expression list of an event control is a common source of bugs in register transfer level (RTL) simulations. The implicit event_expression, **@***, is a convenient shorthand that eliminates these problems by adding all nets and variables that are read by the statement (which can be a statement group) of a *procedural_timing_ control_statement* to the *event_expression*.

NOTE—The **always_comb** procedure (see 9.2.2.2) is preferred over using the **@*** implicit event_expression list when used at the beginning of an always procedure as a sensitivity list. See 9.2.2.2.2 for a comparison of **always_comb** and **@***.

All net and variable identifiers that appear in the statement will be automatically added to the event expression with the following exceptions:

— Identifiers that only appear in wait or event expressions.
— Identifiers that only appear as a *hierarchical_variable_identifier* in the *variable_lvalue* of the left-hand side of assignments.

Nets and variables that appear on the right-hand side of assignments, in subroutine calls, in case and conditional expressions, as an index variable on the left-hand side of assignments, or as variables in case item expressions shall all be included by these rules.

*Example 1:*

```
always @(*)  // equivalent to @(a or b or c or d or f)
   y = (a & b) | (c & d) | myfunction(f);
```

*Example 2:*

```
always @* begin // equivalent to @(a or b or c or d or tmp1 or tmp2)
   tmp1 = a & b;
   tmp2 = c & d;
   y = tmp1 | tmp2;
end
```

*Example 3:*

```
always @* begin  // equivalent to @(b)
   @(i) kid = b;  // i is not added to @*
end
```

*Example 4:*

```
always @* begin  // equivalent to @(a or b or c or d)
   x = a ^ b;
   @*              // equivalent to @(c or d)
         x = c ^ d;
end
```

*Example 5:*

```
always @* begin  // same as @(a or en)
   y = 8'hff;
   y[a] = !en;
end
```

*Example 6:*

```
always @* begin  // same as @(state or go or ws)
   next = 4'b0;
   case (1'b1)
         state[IDLE]:   if (go)  next[READ] = 1'b1;
                        else     next[IDLE] = 1'b1;
         state[READ]:            next[DLY ] = 1'b1;
         state[DLY ]:   if (!ws) next[DONE] = 1'b1;
                        else     next[READ] = 1'b1;
         state[DONE]:            next[IDLE] = 1'b1;
   endcase
end
```

### 9.4.2.3 Conditional event controls

The `@` event control can have an **`iff`** qualifier.

```
module latch (output logic [31:0] y, input [31:0] a, input enable);
   always @(a iff enable == 1)
      y <= a; //latch is in transparent mode
endmodule
```

The event expression only triggers if the expression after the **`iff`** is true (as defined in 12.4), in this case when `enable` is equal to 1. This type of expression is evaluated when `a` changes and not when `enable` changes. Also, in similar event expressions of this type, **`iff`** has precedence over **`or`**. This can be made clearer by the use of parentheses.

### 9.4.2.4 Sequence events

A sequence instance can be used in event expressions to control the execution of procedural statements based on the successful match of the sequence. This allows the end point of a named sequence (see 16.7) to trigger multiple actions in other processes. Syntax 16-3 and Syntax 16-5 describe the syntax for declaring named sequences and sequence instances. A sequence instance can be used directly in an event expression, as shown in Syntax 9-4.

When a sequence instance is specified in an event expression, the process executing the event control shall block until the specified sequence reaches its end point. A process resumes execution following the Observed region in which the end point is detected.

An example of using a sequence as an event control follows:

```
sequence abc;
   @(posedge clk) a ##1 b ##1 c;
endsequence

program test;
   initial begin
      @ abc $display( "Saw a-b-c" );
      L1 : ...
   end
endprogram
```

In the preceding example, when the named sequence `abc` reaches its end point, the **`initial`** procedure in the program block `test` is unblocked, then displays the string `"Saw a-b-c"`, and continues execution with the statement labeled `L1`. In this case, the end of the sequence acts as the trigger to unblock the event.

A sequence used in an event control is instantiated (as if by an assert property statement); the event control is used to synchronize to the end of the sequence, regardless of its start time. Arguments to these sequences shall be static; automatic variables used as sequence arguments shall result in an error.

### 9.4.3 Level-sensitive event control

The execution of a procedural statement can also be delayed until a condition becomes true. This is accomplished using the *wait* statement, which is a special form of event control. The nature of the wait statement is level-sensitive, as opposed to basic event control (specified by the `@` character), which is edge-sensitive.

The wait statement shall evaluate a condition; and, if it is not true (as defined in 12.4), the procedural statements following the wait statement shall remain blocked until that condition becomes true before continuing. The wait statement has the form given in Syntax 9-5.

```
wait_statement ::=                                                        // from A.6.5
     wait ( expression ) statement_or_null
   | wait fork ;
   | wait_order ( hierarchical_identifier { , hierarchical_identifier } ) action_block
```

*Syntax 9-5—Syntax for wait statement (excerpt from Annex A)*

The following example shows the use of the wait statement to accomplish level-sensitive event control:

```
begin
   wait (!enable) #10 a = b;
   #10 c = d;
end
```

If the value of `enable` is 1 when the block is entered, the wait statement will delay the evaluation of the next statement (`#10 a = b;`) until the value of `enable` changes to 0. If `enable` is already 0 when the begin-end block is entered, then the assignment "`a = b;`" is evaluated after a delay of 10 and no additional delay occurs.

See also 9.6 on process control.

### 9.4.4 Level-sensitive sequence controls

The execution of procedural code can be delayed until a sequence termination status is true. This is accomplished using the level-sensitive **wait** statement in conjunction with the built-in method that returns the current end status of a named sequence: `triggered`.

The `triggered` sequence method evaluates to true (`1'b1`) if the given sequence has reached its end point (see 16.7) at that particular point in time (in the current time step) and false (`1'b0`) otherwise. The triggered status of a sequence is set during the Observed region and persists through the remainder of the time step (i.e., until simulation time advances).

For example:

```
sequence abc;
   @(posedge clk) a ##1 b ##1 c;
endsequence

sequence de;
   @(negedge clk) d ##[2:5] e;
endsequence

program check;
   initial begin
      wait( abc.triggered || de.triggered );
      if( abc.triggered )
         $display( "abc succeeded" );
      if( de.triggered )
         $display( "de succeeded" );
      L2 : ...
   end
endprogram
```

In the preceding example, the **initial** procedure in program `check` waits for the end point of either sequence `abc` or sequence `de`. When either condition evaluates to true, the **wait** statement unblocks the process, displays the sequences that caused the process to unblock, and then continues to execute the statement labeled `L2`.

See 16.9.11and 16.13.6 for a definition of sequence methods.

### 9.4.5 Intra-assignment timing controls

The delay and event control constructs previously described precede a statement and delay its execution. In contrast, *intra-assignment delay and event controls* are contained within an assignment statement and modify the flow of activity in a different way. This subclause describes the purpose of intra-assignment timing controls and the repeat timing control that can be used in intra-assignment delays.

An intra-assignment delay or event control shall delay the assignment of the new value to the left-hand side, but the right-hand expression shall be evaluated before the delay, instead of after the delay. The syntax for intra-assignment delay and event control is given in Syntax 9-6.

```
blocking_assignment ::=                                          // from A.6.2
    variable_lvalue = delay_or_event_control  expression
  | ...
nonblocking_assignment ::=
    variable_lvalue <= [ delay_or_event_control ] expression
```

*Syntax 9-6—Syntax for intra-assignment delay and event control (excerpt from Annex A)*

The delay_or_event_control syntax is shown in Syntax 9-4 in 9.4.

The intra-assignment delay and event control can be applied to both blocking assignments and nonblocking assignments. The *repeat* event control shall specify an intra-assignment delay of a specified number of occurrences of an event. If the *repeat* count literal, or signed variable holding the *repeat* count, is less than or equal to 0 at the time of evaluation, the assignment occurs as if there is no *repeat* construct.

For example:

```
repeat (3) @ (event_expression)
   // will execute event_expression three times

repeat (-3) @ (event_expression)
   // will not execute event_expression.

repeat (a) @ (event_expression)
   // if a is assigned -3, it will execute the event_expression if a is
   // declared as an unsigned variable, but not if a is signed
```

This construct is convenient when events have to be synchronized with counts of clock signals.

Table 9-3 illustrates the philosophy of intra-assignment timing controls by showing the code that could accomplish the same timing effect without using intra-assignment timing.

**Table 9-3—Intra-assignment timing control equivalence**

| *Intra-assignment timing control* | |
|---|---|
| **With intra-assignment construct** | **Without intra-assignment construct** |
| `a = #5 b;` | `begin` `temp = b;` `#5 a = temp;` `end` |
| `a = @(posedge clk) b;` | `begin` `temp = b;` `@(posedge clk) a = temp;` `end` |
| `a = repeat(3) @(posedge clk) b;` | `begin` `temp = b;` `@(posedge clk);` `@(posedge clk);` `@(posedge clk) a = temp;` `end` |

The next three examples use the fork-join behavioral construct. All statements between the keywords **`fork`** and **`join`** execute concurrently. This construct is described in more detail in 9.3.2.

The following example shows a race condition that could be prevented by using intra-assignment timing control:

```
fork
   #5 a = b;
   #5 b = a;
join
```

The code in this example samples and sets the values of both `a` and `b` at the same simulation time, thereby creating a race condition. The intra-assignment form of timing control used in the next example prevents this race condition.

```
fork                    // data swap
   a = #5 b;
   b = #5 a;
join
```

Intra-assignment timing control works because the intra-assignment delay causes the values of `a` and `b` to be evaluated before the delay and causes the assignments to be made after the delay.

Intra-assignment waiting for events is also effective. In the following example, the right-hand expressions are evaluated when the assignment statements are encountered, but the assignments are delayed until the rising edge of the clock signal:

```
fork                             // data shift
   a = @(posedge clk) b;
   b = @(posedge clk) c;
join
```

The following is an example of a repeat event control as the intra-assignment delay of a nonblocking assignment:

```
a <= repeat(5) @(posedge clk) data;
```

Figure 9-1 illustrates the activities that result from this **repeat** event control.

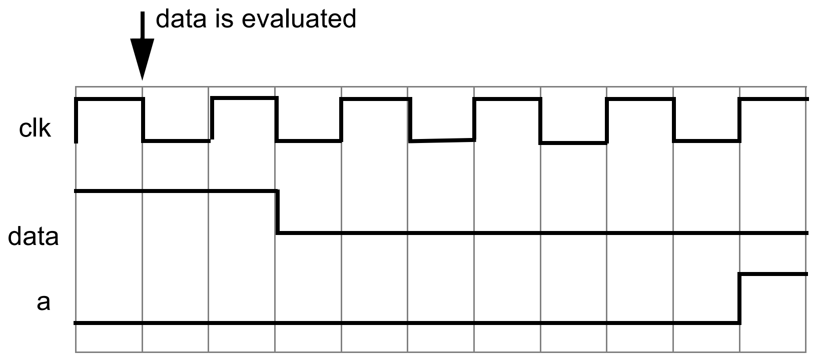


**Figure 9-1—Intra-assignment repeat event control utilizing a clock edge**

In this example, the value of `data` is evaluated when the assignment is encountered. After five occurrences of `posedge clk`, `a` is assigned the value of `data`.

The following is an example of a repeat event control as the intra-assignment delay of a procedural assignment:

```
a = repeat(num) @(clk) data;
```

In this example, the value of `data` is evaluated when the assignment is encountered. After the number of transitions of `clk` equals the value of `num`, `a` is assigned the value of `data`.

The following is an example of a repeat event control with expressions containing operations to specify both the number of event occurrences and the event that is counted:

```
a <= repeat(a+b) @(posedge phi1 or negedge phi2) data;
```

In this example, the value of `data` is evaluated when the assignment is encountered. After the sum of the positive edges of `phi1` and the negative edges of `phi2` equals the sum of `a` and `b`, `a` is assigned the value of `data`. Even if `posedge phi1` and `negedge phi2` occurred at the same simulation time, each will be detected and counted separately.

If `phi1` and `phi2` refer to the same signal, then the preceding assignment can be simplified as:

```
a <= repeat(a+b) @(edge phi1) data;
```

## 9.5 Process execution threads

SystemVerilog creates a thread of execution for the following:

— Each **initial** procedure
— Each **final** procedure
— Each **always**, **always_comb**, **always_latch**, and **always_ff** procedure

— Each parallel statement in a **fork-join** (or **join_any** or **join_none**) statement group
— Each dynamic process

Each continuous assignment can also be considered its own thread (see 10.3).

## 9.6 Process control

SystemVerilog provides constructs that allow one process to terminate or wait for the completion of other processes. The **wait fork** construct waits for the completion of processes. The **disable** construct stops the execution of all activity within a named block or task, without regard to parent-child relationship (a child process can terminate execution of a parent or one process can terminate execution of an unrelated process). The **disable fork** construct stops the execution of processes, but with consideration of parent-child relationships.

The process control statements have the syntax form shown in Syntax 9-7.

```
wait_statement ::=                                                    // from A.6.5
    wait ( expression ) statement_or_null
  | wait fork ;
  | wait_order ( hierarchical_identifier { , hierarchical_identifier } ) action_block
disable_statement ::=
    disable hierarchical_task_identifier ;
  | disable hierarchical_block_identifier ;
  | disable fork ;
```

*Syntax 9-7—Syntax for process control statements (excerpt from Annex A)*

### 9.6.1 Wait fork statement

The **wait fork** statement blocks process execution flow until all immediate child subprocesses (processes created by the current process, excluding their descendants) have completed their execution.

The syntax for **wait fork** is as follows:

```
    wait fork ;    // from A.6.5
```

Specifying **wait fork** causes the calling process to block until all its immediate child subprocesses have completed.

Simulation automatically terminates when there is no further activity of any kind. Simulation also automatically terminates when all its program blocks finish executing (i.e, they reach the end of their execute block), regardless of the status of any child processes (see 24.7). The **wait fork** statement allows a program block to wait for the completion of all its concurrent threads before exiting.

In the following example, two immediate child processes (`child1` and `child2`) are spawned before calling the task `do_test`. In the task `do_test`, three more immediate child processes (`child3`, `child4`, and `child5`) and two descendant processes (`descendant1` and `descendant2`) are spawned. Next, two more immediate child processes (`child6` and `child7`) are spawned by the function `do_sequence`. The **wait fork** statement blocks the execution flow of the task `do_test` until all seven immediate child processes complete before returning to its caller. The **wait fork** statement does not directly depend on the descendant processes spawned by `child5`.

```
initial begin : test
   fork
      child1();
      child2();
   join_none
   do_test();
end : test

task do_test();
   fork
      child3();
      child4();
      fork : child5  // nested fork-join_none is a child process
         descendant1();
         descendant2();
      join_none
   join_none
   do_sequence();
   wait fork;        // block until child1 ... child7 complete
endtask

function void do_sequence();
   fork
      child6();
      child7();
   join_none
endfunction
```

### 9.6.2 Disable statement

The *disable* statement provides the ability to terminate the activity associated with concurrently active processes, while maintaining the structured nature of procedural descriptions. The disable statement gives a mechanism for terminating a task before it executes all its statements, breaking from a looping statement, or skipping statements in order to continue with another iteration of a looping statement. It is useful for handling exception conditions such as hardware interrupts and global resets. The disable statement can also be used to terminate execution of a labeled statement, including a deferred assertion (see 16.4) or a procedural concurrent assertion (see 16.14.6).

The disable statement shall terminate the activity of a task or a named block. Execution shall resume at the statement following the block or following the task-enabling statement. All activities enabled within the named block or task shall be terminated as well. If task enable statements are nested (that is, one task enables another, and that one enables yet another), then disabling a task within the chain shall disable all tasks downward on the chain. If a task is enabled more than once, then disabling such a task shall disable all activations of the task.

The results of the following activities that can be initiated by a task are not specified if the task is disabled:

— Results of output and inout arguments
— Scheduled, but not executed, nonblocking assignments
— Procedural continuous assignments (assign and force statements)

The disable statement can be used within blocks and tasks to disable the particular block or task containing the disable statement. The disable statement can be used to disable named blocks within a function, but cannot be used to disable functions. In cases where a disable statement within a function disables a block or a task that called the function, the behavior is undefined. Disabling an automatic task or a block inside an automatic task proceeds as for regular tasks for all concurrent executions of the task.

*Example 1:* This example illustrates how a block disables itself.

```
begin : block_name
   rega = regb;
   disable block_name;
   regc = rega; // this assignment will never execute
end
```

*Example 2:* This example shows the disable statement being used within a named block in a manner similar to a forward *goto*. The next statement executed after the disable statement is the one following the named block.

```
begin : block_name
   ...
   ...
   if (a == 0)
      disable block_name;
   ...
end      // end of named block
// continue with code following named block
   ...
```

*Example 3*: This example illustrates using the disable construct to terminate execution of a named block that does not contain the disable statement. If the block is currently executing, this causes control to jump to the statement immediately after the block. If the block is a loop body, it acts like a **continue** (see 12.8). If the block is not currently executing, the disable has no effect.

```
module m (...);
   always
      begin : always1
         ...
         t1: task1( ); // task call
         ...
      end
   ...

   always
      begin
         ...
         disable m.always1;      // exit always1, which will exit task1,
                                 // if it was currently executing
      end
endmodule
```

*Example 4:* This example shows the disable statement being used as an early return from a task. SystemVerilog also has **return** from a task, which shall terminate execution of the process in which the return is executed (see 12.8). However, a task disabling itself using a disable statement is not a shorthand for the *return* statement. If **disable** is applied to a task, all currently active executions of the task are disabled.

```
task proc_a;
   begin
      ...
      ...
      if (a == 0)
         disable proc_a; // return if true
      ...
      ...
   end
endtask
```

*Example 5*: This example shows the disable statement being used in an equivalent way to the two statements **continue** and **break** (see 12.8). The example illustrates control code that would allow a named block to execute until a loop counter reaches `n` iterations or until the variable `a` is set to the value of `b`. The named block `outer_block` contains the code that executes until `a == b`, at which point the **disable** `outer_block;` statement terminates execution of that block. The named block `inner_block` contains the code that executes for each iteration of the **for** loop. Each time this code executes the **disable** `inner_block;` statement, the `inner_block` block terminates, and execution passes to the next iteration of the **for** loop. For each iteration of the `inner_block` block, a set of statements executes if `(a != 0)`. Another set of statements executes if `(a! = b)`.

```
   begin : outer_block
      for (i = 0; i < n; i = i+1) begin : inner_block
            @clk
               if (a == 0) // "continue" loop
                  disable inner_block ;
               ... // statements
               ... // statements
            @clk
               if (a == b) // "break" from loop
                  disable outer_block;
               ... // statements
               ... // statements
      end
   end
```

NOTE—The C-like **break** and **continue** statements (see 12.8) may be a more intuitive way to code the preceding example.

*Example 6*: This example shows the disable statement being used to disable concurrently a sequence of timing controls and the task named `action` when the `reset` event occurs. The example shows a fork-join block within which are a named sequential block (`event_expr`) and a disable statement that waits for occurrence of the event `reset`. The sequential block and the wait for `reset` execute in parallel. The `event_expr` block waits for one occurrence of event `ev1` and three occurrences of event `trig`. When these four events have happened, plus a delay of `d` time units, the task `action` executes. When the event `reset` occurs, regardless of events within the sequential block, the fork-join block terminates—including the task `action`.

```
   fork
      begin : event_expr
            @ev1;
            repeat (3) @trig;
            #d action (areg, breg);
      end
      @reset disable event_expr;
   join
```

*Example 7*: The next example is a behavioral description of a retriggerable monostable. The named event `retrig` restarts the monostable time period. If `retrig` continues to occur within 250 time units, then `q` will remain at 1.

```
   always begin : monostable
      #250 q = 0;
   end

   always @retrig begin
```

```
      disable monostable;
      q = 1;
   end
```

### 9.6.3 Disable fork statement

The **disable fork** statement terminates all active descendants (subprocesses) of the calling process.

The syntax for **disable fork** is as follows:

```
   disable fork ;     // from A.6.5
```

The **disable fork** statement terminates all descendants of the calling process as well as the descendants of the process's descendants. In other words, if any of the child processes have descendants of their own, the **disable fork** statement shall terminate them as well.

In the following example, the task `get_first` spawns three versions of a task that wait for a particular device (1, 7, or 13). The task `wait_device` waits for a particular device to become ready and then returns the device's address. When the first device becomes available, the `get_first` task shall resume execution and proceed to kill the outstanding `wait_device` processes.

```
   task get_first( output int adr );
      fork
         wait_device( 1, adr );
         wait_device( 7, adr );
         wait_device( 13, adr );
      join_any
      disable fork;
   endtask
```

The **disable** construct terminates a process when applied to the named block or statement being executed by the process. The **disable fork** statement differs from **disable** in that **disable fork** considers the dynamic parent-child relationship of the processes, whereas **disable** uses the static, syntactical information of the disabled block. Thus, **disable** shall end all processes executing a particular block, whether the processes were forked by the calling thread or not, while **disable fork** shall end only the processes that were spawned by the calling thread.

## 9.7 Fine-grain process control

A process is a built-in class that allows one process to access and control another process once it has started. Users can declare variables of type process and safely pass them through tasks or incorporate them into other objects. The prototype for the process class is as follows:

```
   class process;
      typedef enum { FINISHED, RUNNING, WAITING, SUSPENDED, KILLED } state;

      static function process self();
      function state status();
      function void kill();
      task await();
      function void suspend();
      function void resume();
      function void srandom( int seed );
      function string get_randstate();
      function void set_randstate( string state );
   endclass
```

Objects of type process are created internally when processes are spawned. Users cannot create objects of type process; attempts to call **`new`** shall not create a new process and shall instead result in an error. The process class cannot be extended. Attempts to extend it shall result in a compilation error. Objects of type process are unique; they become available for reuse once the underlying process terminates and all references to the object are discarded.

The **`self()`** function returns a handle to the current process, that is, a handle to the process making the call.

The **`status()`** function returns the process status, as defined by the state enumeration:

- **`FINISHED`** means the process terminated normally.
- **`RUNNING`** means the process is currently running (not in a blocking statement).
- **`WAITING`** means the process is waiting in a blocking statement.
- **`SUSPENDED`** means the process is stopped awaiting a resume.
- **`KILLED`** means the process was forcibly killed (via kill or disable).

The **`kill()`** function terminates the given process and all its subprocesses, that is, processes spawned using **`fork`** statements by the process being killed. If the process to be terminated is not blocked waiting on some other condition, such as an event, **`wait`** expression, or a delay, then the process shall be terminated at some unspecified time in the current time step.

The **`await()`** task allows one process to wait for the completion of another process. It shall be an error to call this task on the current process, i.e., a process cannot wait for its own completion.

The **`suspend()`** function allows a process to suspend either its own execution or that of another process. If the process to be suspended is not blocked waiting on some other condition, such as an event, **`wait`** expression, or a delay, then the process shall be suspended at some unspecified time in the current time step. Calling this method more than once, on the same (suspended) process, has no effect.

The **`resume()`** function restarts a previously suspended process. Calling `resume` on a process that was suspended while blocked on another condition shall resensitize the process to the event expression or to wait for the wait condition to become true or for the delay to expire. If the wait condition is now true or the original delay has transpired, the process is scheduled onto the Active or Reactive region to continue its execution in the current time step. Calling `resume` on a process that suspends itself causes the process to continue to execute at the statement following the call to suspend.

The methods `kill()`, `await()`, `suspend()`, and `resume()` shall be restricted to a process created by an initial procedure, always procedure, or fork block from one of those procedures.

The following example starts an arbitrary number of processes, as specified by the task argument *N*. Next, the task waits for the last process to start executing and then waits for the first process to terminate. At that point, the parent process forcibly terminates all forked processes that have not yet completed.

```
task automatic do_n_way( int N );
   process job[] = new [N];

   foreach (job[j])
      fork
         automatic int k = j;
         begin job[k] = process::self(); ... ; end
      join_none

   foreach (job[j])                  // wait for all processes to start
      wait( job[j] != null );
```

```
        job[1].await();                   // wait for first process to finish

        foreach (job[j]) begin
           if ( job[j].status != process::FINISHED )
              job[j].kill();
        end

endtask
```

# 10. Assignment statements

## 10.1 General

This clause describes the following:

- Continuous assignments
- Procedural blocking and nonblocking assignments
- Procedural continuous assignments (assign, deassign, force, release)
- Net aliasing

## 10.2 Overview

The assignment is the basic mechanism for placing values into nets and variables. There are two basic forms of assignments, as follows:

- The *continuous assignment*, which assigns values to nets or variables
- The *procedural assignment*, which assigns values to variables

Continuous assignments drive nets or variables in a manner similar to the way gates drive nets or variables. The expression on the right-hand side can be thought of as a combinational circuit that drives the net or variable continuously. In contrast, procedural assignments put values in variables. The assignment does not have duration; instead, the variable holds the value of the assignment until the next procedural assignment to that variable.

There are two additional forms of assignments, **`assign`**/**`deassign`** and **`force`**/**`release`**, which are called *procedural continuous assignments*, described in 10.6.

An assignment consists of two parts, a left-hand side and a right-hand side, separated by the equals ( **=** ) character; or, in the case of nonblocking procedural assignment, the less-than-equals ( **<=** ) character pair. The right-hand side can be any expression that evaluates to a value. The left-hand side indicates the net or variable to which the right-hand side value is to be assigned. The left-hand side can take one of the forms given in Table 10-1, depending on whether the assignment is a continuous assignment or a procedural assignment.

**Table 10-1—Legal left-hand forms in assignment statements**

| Statement type | Left-hand side |
|---|---|
| Continuous assignment | Net or variable (vector or scalar)<br>Constant bit-select of a vector net or packed variable<br>Constant part-select of a vector net or packed variable<br>Concatenation or nested concatenation of any of the above left-hand sides |
| Procedural assignment | Variable (vector or scalar)<br>Bit-select of a packed variable<br>Part-select of a packed variable<br>Memory word<br>Array<br>Array element select<br>Array slice<br>Concatenation or nested concatenation of any of the above left-hand sides |

SystemVerilog also allows a time unit to be specified in the assignment statement, as follows:

```
#1ns r = a;
r = #1ns a;
r <= #1ns a;
assign #2.5ns sum = a + b;
```

## 10.3 Continuous assignments

Continuous assignments shall drive values onto nets or variables, both vector (packed) and scalar. This assignment shall occur whenever the value of the right-hand side changes. Continuous assignments provide a way to model combinational logic without specifying an interconnection of gates. Instead, the model specifies the logical expression that drives the net or variable.

There are two forms of continuous assignments: *net declaration assignments* (see 10.3.1) and *continuous assign statements* (see 10.3.2).

The syntax for continuous assignments is given in Syntax 10-1.

```
net_declaration12 ::=                                                         // from A.2.1.3
    net_type [ drive_strength | charge_strength ] [ vectored | scalared ]
       data_type_or_implicit [ delay3 ] list_of_net_decl_assignments ;
  | net_type_identifier [ delay_control ]
       list_of_net_decl_assignments ;
  | interconnect implicit_data_type [ # delay_value ]
       net_identifier { unpacked_dimension }
       [ , net_identifier { unpacked_dimension }] ;
list_of_net_decl_assignments ::= net_decl_assignment { , net_decl_assignment }   // from A.2.3
net_decl_assignment ::= net_identifier { unpacked_dimension } [ = expression ]   // from A.2.4
continuous_assign ::=                                                         // from A.6.1
    assign [ drive_strength ] [ delay3 ] list_of_net_assignments ;
  | assign [ delay_control ] list_of_variable_assignments ;
list_of_net_assignments ::= net_assignment { , net_assignment }
list_of_variable_assignments ::= variable_assignment { , variable_assignment }
net_assignment ::= net_lvalue = expression
```

12) A charge strength shall only be used with the **trireg** keyword. When the **vectored** or **scalared** keyword is used, there shall be at least one packed dimension.

*Syntax 10-1—Syntax for continuous assignment (excerpt from Annex A)*

### 10.3.1 The net declaration assignment

The *net declaration assignment* allows a continuous assignment to be placed on a net in the same statement that declares the net.

The following is an example of the net declaration form of a continuous assignment:

```
wire (strong1, pull0) mynet = enable;
```

Because a net can be declared only once, only one net declaration assignment can be made for a particular net. This contrasts with the continuous assignment statement; one net can receive multiple assignments of the continuous assignment form.

An **interconnect** net (see 6.6.8) shall not have a net declaration assignment.

### 10.3.2 The continuous assignment statement

The continuous assignment statement shall place a continuous assignment on a net or variable data type. The net may be explicitly declared or may inherit an implicit declaration in accordance with the implicit declaration rules defined in 6.10. Variables shall be explicitly declared prior to the continuous assignment statement.

Assignments on nets or variables shall be continuous and automatic. In other words, whenever an operand in the right-hand expression changes value, the whole right-hand side shall be evaluated. If the new value is different from the previous value, then the new value shall be assigned to the left-hand side.

Nets can be driven by multiple continuous assignments or by a mixture of primitive outputs, module outputs, and continuous assignments. Variables can only be driven by one continuous assignment or by one primitive output or module output. It shall be an error for a variable driven by a continuous assignment or output to have an initializer in the declaration or any procedural assignment. See also 6.5.

A continuous assignment to an atomic net shall not drive part of the net; the entire **nettype** value shall be driven. Thus the left-hand side of a continuous assignment to a net of a user-defined **nettype** shall not contain any indexing or select operations into the data type of the **nettype**.

*Example 1:* The following is an example of a continuous assignment to a net that has been previously declared:

```
wire mynet ;
assign (strong1, pull0) mynet = enable;
```

*Example 2:* The following is an example of the use of a continuous assignment to model a 4-bit adder with carry. The assignment could not be specified directly in the declaration of the nets because it requires a concatenation on the left-hand side.

```
module adder (sum_out, carry_out, carry_in, ina, inb);
   output [3:0] sum_out;
   output carry_out;
   input [3:0] ina, inb;
   input carry_in;

   wire carry_out, carry_in;
   wire [3:0] sum_out, ina, inb;

   assign {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

*Example 3:* The following example describes a module with one 16-bit output bus. It selects between one of four input busses and connects the selected bus to the output bus.

```
module select_bus(busout, bus0, bus1, bus2, bus3, enable, s);
   parameter n = 16;
   parameter Zee = 16'bz;
   output [1:n] busout;
```

```
      input [1:n] bus0, bus1, bus2, bus3;
      input enable;
      input [1:2] s;

      tri [1:n] data;         // net declaration

      // net declaration with continuous assignment
      tri [1:n] busout = enable ? data : Zee;

      // assignment statement with four continuous assignments
      assign
         data = (s == 0) ? bus0 : Zee,
         data = (s == 1) ? bus1 : Zee,
         data = (s == 2) ? bus2 : Zee,
         data = (s == 3) ? bus3 : Zee;
   endmodule
```

The following sequence of events is experienced during simulation of this example:

a) The value of `s`, a bus selector input variable, is checked in the **`assign`** statement. Based on the value of `s`, the net `data` receives the data from one of the four input buses.
b) The setting of net `data` triggers the continuous assignment in the net declaration for `busout`. If `enable` is set, the contents of `data` are assigned to `busout`; if enable is 0, the contents of `Zee` are assigned to busout.

### 10.3.3 Continuous assignment delays

A delay given to a continuous assignment shall specify the time duration between a right-hand operand value change and the assignment made to the left-hand side. If the left-hand references a scalar net, then the delay shall be treated in the same way as for gate delays; that is, different delays can be given for the output rising, falling, and changing to high impedance (see 28.16).

If the left-hand references a vector net, then up to three delays can be applied. The following rules determine which delay controls the assignment:

— If the right-hand side makes a transition from nonzero to zero, then the falling delay shall be used.
— If the right-hand side makes a transition to `z`, then the turn-off delay shall be used.
— For all other cases, the rising delay shall be used.

If the left-hand side references a net of a user-defined **`nettype`** or an array of such nets, then only a single delay may be applied. The specific delay is used when any change occurs to the value of the net.

Specifying the delay in a continuous assignment that is part of the net declaration shall be treated differently from specifying a net delay and then making a continuous assignment to the net. A delay value can be applied to a net in a net declaration, as in the following example:

```
   wire #10 wireA;
```

This syntax, called a *net delay*, means that any value change that is to be applied to `wireA` by some other statement shall be delayed for ten time units before it takes effect. When there is a continuous assignment in a declaration, the delay is part of the continuous assignment and is not a net delay. Thus, it shall not be added to the delay of other drivers on the net. Furthermore, if the assignment is to a vector net, then the rising and falling delays shall not be applied to the individual bits if the assignment is included in the declaration.

In situations where a right-hand operand changes before a previous change has had time to propagate to the left-hand side, then the following steps are taken:

a) The value of the right-hand expression is evaluated.
b) If this right-hand side value differs from the value currently scheduled to propagate to the left-hand side, then the currently scheduled propagation event is descheduled.
c) If the new right-hand side value equals the current left-hand side value, no event is scheduled.
d) If the new right-hand side value differs from the current left-hand side value, a delay is calculated in the standard way using the current value of the left-hand side, the newly calculated value of the right-hand side, and the delays indicated on the statement; a new propagation event is then scheduled to occur delay time units in the future.

### 10.3.4 Continuous assignment strengths

The driving strength of a continuous assignment can be specified by the user. This applies only to assignments to scalar nets, except for nets of types **supply0** and **supply1**.

Continuous assignments driving strengths can be specified either in a net declaration or in a stand-alone assignment, using the **assign** keyword. The strength specification, if provided, shall immediately follow the keyword (either the keyword for the net type or **assign**) and precede any delay specified. Whenever the continuous assignment drives the net, the strength of the value shall be simulated as specified.

A drive strength specification shall contain one strength value that applies when the value being assigned to the net is 1 and a second strength value that applies when the assigned value is 0. The following keywords shall specify the strength value for an assignment of 1:

```
supply1      strong1      pull1      weak1      highz1
```

The following keywords shall specify the strength value for an assignment of 0:

```
supply0      strong0      pull0      weak0      highz0
```

The order of the two strength specifications shall be arbitrary. The following two rules shall constrain the use of drive strength specifications:

— The strength specifications (**highz1**, **highz0**) and (**highz0**, **highz1**) shall be treated as illegal constructs.
— If drive strength is not specified, it shall default to (**strong1**, **strong0**).

## 10.4 Procedural assignments

Procedural assignments occur within procedures such as **always**, **initial** (see 9.2), **task**, and **function** (see Clause 13) and can be thought of as "triggered" assignments. The trigger occurs when the flow of execution in the simulation reaches an assignment within a procedure. Reaching the assignment can be controlled by conditional statements. Event controls, delay controls, **if** statements, **case** statements, and looping statements can all be used to control whether assignments are evaluated. Clause 12 gives details and examples.

The right-hand side of a procedural assignment can be any expression that evaluates to a value, however the variable type on the left-hand side may restrict what is a legal expression on the right-hand side. The left-hand side shall be a variable that receives the assignment from the right-hand side. The left-hand side of a procedural assignment can take one of the following forms:

— Singular variables, as described in 6.4
— Aggregate variables, as described in Clause 7
— Bit-selects, part-selects, and slices of packed arrays
— Slices of unpacked arrays

SystemVerilog contains the following three types of procedural assignment statements:

— Blocking procedural assignment statements (see 10.4.1)
— Nonblocking procedural assignment statements (see 10.4.2)
— Assignment operators (see 11.4.1)

Blocking and nonblocking procedural assignment statements specify different procedural flows in sequential blocks.

### 10.4.1 Blocking procedural assignments

A *blocking procedural assignment* statement shall be executed before the execution of the statements that follow it in a sequential block (see 9.3.1). A blocking procedural assignment statement shall not prevent the execution of statements that follow it in a parallel block (see 9.3.2).

The syntax for a blocking procedural assignment is given in Syntax 10-2.

```
blocking_assignment ::=                                                          // from A.6.3
      variable_lvalue = delay_or_event_control  expression
    | nonrange_variable_lvalue = dynamic_array_new
    | [ implicit_class_handle . | class_scope | package_scope ] hierarchical_variable_identifier
         select = class_new
    | operator_assignment
operator_assignment ::= variable_lvalue  assignment_operator  expression
assignment_operator ::=
      = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>= | <<<= | >>>=
```

*Syntax 10-2—Blocking assignment syntax (excerpt from Annex A)*

In this syntax, *variable_lvalue* is a data type that is valid for a procedural assignment statement, **=** is the assignment operator, and *delay_or_event_control* is the optional intra-assignment timing control (see 9.4.5). The expression is the right-hand side value that shall be assigned to the left-hand side. If *variable_lvalue* requires an evaluation, it shall be evaluated at the time specified by the intra-assignment timing control. The order of evaluation of the *variable_lvalue* and the expression on the right-hand side is undefined if a timing control is not specified. See 4.9.3.

The = assignment operator used by blocking procedural assignments is also used by procedural continuous assignments and continuous assignments.

The following examples show blocking procedural assignments:

```
rega = 0;
rega[3] = 1;                 // a bit-select
rega[3:5] = 7;               // a part-select
mema[address] = 8'hff;       // assignment to a mem element
{carry, acc} = rega + regb;  // a concatenation
```

Additional assignment operators, such as +=, are described in 11.4.1.

### 10.4.2 Nonblocking procedural assignments

The *nonblocking procedural assignment* allows assignment scheduling without blocking the procedural flow. The nonblocking procedural assignment statement can be used whenever several variable assignments within the same time step can be made without regard to order or dependence upon each other.

It shall be illegal to make nonblocking assignments to automatic variables.

The syntax for a nonblocking procedural assignment is given in Syntax 10-3.

---

nonblocking_assignment ::= variable_lvalue **<=** [ delay_or_event_control ] expression *// from A.6.3*

---

*Syntax 10-3—Nonblocking assignment syntax (excerpt from Annex A)*

In this syntax, *variable_lvalue* is a data type that is valid for a procedural assignment statement, **<=** is the nonblocking assignment operator, and *delay_or_event_control* is the optional intra-assignment timing control (see 9.4.5). If *variable_lvalue* requires an evaluation, such as an index expression, class handle, or virtual interface reference, it shall be evaluated at the same time as the expression on the right-hand side. The order of evaluation of the *variable_lvalue* and the expression on the right-hand side is undefined (see 4.9.4).

The nonblocking assignment operator is the same operator as the less-than-or-equal-to relational operator. The interpretation shall be decided from the context in which **<=** appears. When **<=** is used in an expression, it shall be interpreted as a relational operator; and when it is used in a nonblocking procedural assignment, it shall be interpreted as an assignment operator.

The nonblocking procedural assignments shall be evaluated in two steps as discussed in Clause 4. These two steps are shown in the following example:

*Example 1:*

```
module evaluates (out);
   output out;
   logic a, b, c;

   initial begin
      a = 0;
      b = 1;
      c = 0;
   end

   always c = #5 ~c;

   always @(posedge c) begin
      a <= b; // evaluates, schedules,
      b <= a; // and executes in two steps
   end
endmodule
```

*Step 1:* At `posedge c`, the simulator evaluates the right-hand sides of the nonblocking assignments and schedules the assignments of the new values at the end of the *nonblocking assign update* events NBA region (see 4.5).

*Step 2:* When the simulator activates the *nonblocking assign update* events, the simulator updates the left-hand side of each nonblocking assignment statement.

*Nonblocking assignment schedules change at time 5*

| |
|---|
| *a = 0* |
| *b = 1* |

*assignment values*

| |
|---|
| *a = 1* |
| *b = 0* |

*At the end of the time step* means that the nonblocking assignments are the last assignments executed in a time step—with one exception. Nonblocking assignment events can create blocking assignment events. These blocking assignment events shall be processed after the scheduled nonblocking events.

Unlike an event or delay control for blocking assignments, the nonblocking assignment does not block the procedural flow. The nonblocking assignment evaluates and schedules the assignment, but it does not block the execution of subsequent statements in a begin-end block.

*Example 2:*

```
module nonblock1;
   logic a, b, c, d, e, f;

   // blocking assignments
   initial begin
      a = #10 1; // a will be assigned 1 at time 10
      b = #2 0;  // b will be assigned 0 at time 12
      c = #4 1;  // c will be assigned 1 at time 16
   end

   // nonblocking assignments
   initial begin
      d <= #10 1; // d will be assigned 1 at time 10
      e <= #2 0;  // e will be assigned 0 at time 2
      f <= #4 1;  // f will be assigned 1 at time 4
   end
endmodule
```

*scheduled changes at time 2*

| *e = 0* |
|---|

*scheduled changes at time 4*

| *f = 1* |
|---|

*scheduled changes at time 10*

| *d = 1* |
|---|

As shown in the previous example, the simulator evaluates and schedules assignments for the end of the current time step and can perform swapping operations with the nonblocking procedural assignments.

*Example 3:*

```
module nonblock2;
   logic a, b;
   initial begin
      a  = 0;
      b  = 1;
      a <= b; // evaluates, schedules,
      b <= a; // and executes in two steps
   end

   initial begin
      $monitor ($time, ,"a = %b b = %b", a, b);
      #100 $finish;
   end
endmodule
```

*Step 1:* The simulator evaluates the right-hand side of the nonblocking assignments and schedules the assignments for the end of the current time step.

*Step 2:* At the end of the current time step, the simulator updates the left-hand side of each nonblocking assignment statement.

*assignment values*

| *a = 1* |
|---|
| *b = 0* |

The order of the execution of distinct nonblocking assignments to a given variable shall be preserved. In other words, if there is clear ordering of the execution of a set of nonblocking assignments, then the order of the resulting updates of the destination of the nonblocking assignments shall be the same as the ordering of the execution (see 4.6).

*Example 4:*

```
module multiple;
   logic a;
   initial a = 1;
      // The assigned value of the variable is determinate

   initial begin
      a <= #4 0;     // schedules a = 0 at time 4
      a <= #4 1;     // schedules a = 1 at time 4
   end               // At time 4, a = 1
endmodule
```

If the simulator executes two procedural blocks concurrently and if these procedural blocks contain nonblocking assignment operators to the same variable, the final value of that variable is indeterminate. For example, the value of variable a is indeterminate in the following example:

*Example 5:*

```
module multiple2;
   logic a;

   initial a  = 1;
   initial a <= #4 0;   // schedules 0 at time 4
   initial a <= #4 1;   // schedules 1 at time 4

   // At time 4, a = ??
   // The assigned value of the variable is indeterminate
endmodule
```

The fact that two nonblocking assignments targeting the same variable are in different blocks is not by itself sufficient to make the order of assignments to a variable indeterminate. For example, the value of variable a at the end of time cycle 16 is determinate in the following example:

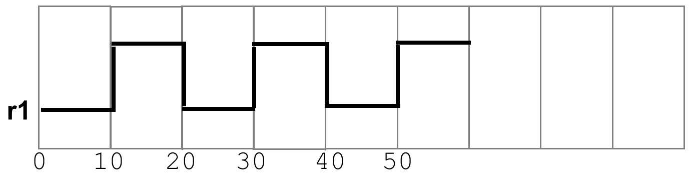
*Example 6:*

```
module multiple3;
   logic a;

   initial #8 a  <= #8 1;  // executed at time 8;
                           // schedules an update of 1 at time 16
   initial #12 a <= #4 0;  // executed at time 12;
                           // schedules an update of 0 at time 16

   // Because it is determinate that the update of a to the value 1
   // is scheduled before the update of a to the value 0,
   // then it is determinate that a will have the value 0
   // at the end of time slot 16.
endmodule
```

The following example shows how the value of i[0] is assigned to r1 and how the assignments are scheduled to occur after each time delay:

*Example 7:*

```
module multiple4;
   logic r1;
   logic [2:0] i;

   initial begin
      // makes assignments to r1 without cancelling previous assignments
      for (i = 0; i <= 5; i++)
         r1 <= # (i*10) i[0];
   end
endmodule
```



## 10.5 Variable declaration assignment (variable initialization)

Unlike nets, a variable cannot have an implicit continuous assignment as part of its declaration. An assignment as part of the declaration of a variable is a variable initialization, not a continuous assignment.

The variable declaration assignment is a special case of procedural assignment as it assigns a value to a variable. It allows an initial value to be placed in a variable in the same statement that declares the variable (see 6.8). The assignment does not have duration; instead, the variable holds the value until the next assignment to that variable.

For example:

```
wire w = vara & varb;              // net with a continuous assignment

logic v = consta & constb;         // variable with initialization
```

Setting the initial value of a static variable as part of the variable declaration (including static class members) shall occur before any initial or always procedures are started. See also 6.21.

## 10.6 Procedural continuous assignments

The *procedural continuous assignments* (using keywords **assign** and **force**) are procedural statements that allow expressions to be driven continuously onto variables or nets. The syntax for these statements is given in Syntax 10-4.

---

procedural_continuous_assignment ::= *// from A.6.2*
    **assign** variable_assignment
  | **deassign** variable_lvalue
  | **force** variable_assignment
  | **force** net_assignment
  | **release** variable_lvalue
  | **release** net_lvalue

variable_assignment ::= variable_lvalue **=** expression

net_assignment ::= net_lvalue **=** expression *// from A.6.1*

---

*Syntax 10-4—Syntax for procedural continuous assignments (excerpt from Annex A)*

The right-hand side of an **assign** procedural continuous assignment or a **force** statement can be an expression. This shall be treated just as a continuous assignment; that is, if any variable on the right-hand side of the assignment changes, the assignment shall be reevaluated while the assign or force is in effect. For example:

```
force a = b + f(c) ;
```

Here, if `b` changes or `c` changes, `a` will be forced to the new value of the expression `b+f(c)`.

### 10.6.1 The assign and deassign procedural statements

The **assign** procedural continuous assignment statement shall override all procedural assignments to a variable. The **deassign** procedural statement shall end a procedural continuous assignment to a variable. The value of the variable shall remain the same until the variable is assigned a new value through a procedural assignment or a procedural continuous assignment. The assign and deassign procedural statements allow, for example, modeling of asynchronous clear/preset on a D-type edge-triggered flip-flop, where the clock is inhibited when the clear or preset is active.

The left-hand side of the assignment in the *assign statement* shall be a singular variable reference or a concatenation of variables. It shall not be a bit-select or a part-select of a variable.

If the keyword **assign** is applied to a variable for which there is already a procedural continuous assignment, then this new procedural continuous assignment shall deassign the variable before making the new procedural continuous assignment.

The following example shows a use of the assign and deassign procedural statements in a behavioral description of a D-type flip-flop with preset and clear inputs:

```
module dff (q, d, clear, preset, clock);
   output q;
   input d, clear, preset, clock;
   logic q;

   always @(clear or preset)
      if (!clear)
            assign q = 0;
      else if (!preset)
            assign q = 1;
      else
            deassign q;

   always @(posedge clock)
      q = d;
endmodule
```

If either `clear` or `preset` is low, then the output `q` will be held continuously to the appropriate constant value, and a positive edge on the `clock` will not affect `q`. When both the `clear` and `preset` are high, then `q` is deassigned.

NOTE—The procedural assign and deassign constructs are under consideration for deprecation. See Annex C.

### 10.6.2 The force and release procedural statements

Another form of procedural continuous assignment is provided by the **force** and **release** procedural statements. These statements have a similar effect to the **assign-deassign** pair, but a force can be applied to nets as well as to variables. The left-hand side of the assignment can be a reference to a singular variable, a net, a constant bit-select of a vector net, a constant part-select of a vector net, or a concatenation of these. It shall not be a bit-select or a part-select of a variable or of a net with a user-defined **nettype**. A **force** or **release** statement shall not be applied to a variable that is being assigned by a mixture of continuous and procedural assignments.

A **force** statement to a variable shall override a procedural assignment, continuous assignment or an **assign** procedural continuous assignment to the variable until a **release** procedural statement is executed on the variable. When released, then if the variable is not driven by a continuous assignment and does not currently have an active **assign** procedural continuous assignment, the variable shall not immediately change value and shall maintain its current value until the next procedural assignment to the variable is executed. Releasing a variable that is driven by a continuous assignment or currently has an active **assign** procedural continuous assignment shall reestablish that assignment and schedule a reevaluation in the continuous assignment's scheduling region.

A **force** procedural statement on a net shall override all drivers of the net—gate outputs, module outputs, and continuous assignments—until a **release** procedural statement is executed on the net. When released, the net shall immediately be assigned the value determined by the drivers of the net.

For example:

```
module test;
   logic a, b, c, d;
   wire e;

   and and1 (e, a, b, c);

   initial begin
      $monitor("%d d=%b,e=%b", $stime, d, e);
      assign d = a & b & c;
      a = 1;
      b = 0;
      c = 1;
      #10;
      force d = (a | b | c);
      force e = (a | b | c);
      #10;
      release d;
      release e;
      #10 $finish;
   end
endmodule

Results:
    0 d=0,e=0
   10 d=1,e=1
   20 d=0,e=0
```

In this example, an **and** gate instance, `and1`, is "patched" to act like an **or** gate by a **force** procedural statement that forces its output to the value of its ORed inputs, and an **assign** procedural statement of ANDed values is "patched" to act like an assign statement of ORed values.

## 10.7 Assignment extension and truncation

The size of the left-hand side of an assignment forms the context for the right-hand expression.

The following are the steps for evaluating an assignment:

— Determine the size of the left-hand side and right-hand side by the standard expression size determination rules (see 11.8.1).
— When the right-hand side evaluates to fewer bits than the left-hand side, the right-hand side value is padded to the size of the left-hand side. If the right-hand side is unsigned, it is padded according to the rules specified in 11.6.1. If the right-hand side is signed, it is sign-extended.
— If the left-hand side is smaller than the right-hand side, truncation shall occur, as described in the following paragraphs.

If the width of the right-hand expression is larger than the width of the left-hand side in an assignment, the MSBs of the right-hand expression shall be discarded to match the size of the left-hand side. Implementations can, but are not required to, warn or report any errors related to assignment size mismatch or truncation. Size casting can be used to indicate explicit intent to change the size (see 6.24.1). Truncating the sign bit of a signed expression may change the sign of the result.

Some examples of assignment truncation follow.

*Example 1:*

```
logic [5:0] a;
logic signed [4:0] b;

initial begin
   a = 8'hff;     // After the assignment, a = 6'h3f
   b = 8'hff;     // After the assignment, b = 5'h1f
end
```

*Example 2:*

```
logic [0:5] a;
logic signed [0:4] b, c;

initial begin
   a = 8'sh8f;    // After the assignment, a = 6'h0f
   b = 8'sh8f;    // After the assignment, b = 5'h0f
   c = -113;      // After the assignment, c = 15
                  // 1000_1111 = (-'h71 = -113) truncates to ('h0F = 15)
end
```

*Example 3:*

```
logic [7:0] a;
logic signed [7:0] b;
logic signed [5:0] c, d;

initial begin
   a = 8'hff;
   c = a;      // After the assignment, c = 6'h3f
   b = -113;
   d = b;      // After the assignment, d = 6'h0f
end
```

## 10.8 Assignment-like contexts

An assignment-like context is as follows:

— A continuous or procedural assignment
— For a parameter with an explicit type declaration:
  - A parameter value assignment in a module, interface, program, or class
  - A parameter value override in the instantiation of a module, interface, or program
  - A parameter value override in the instantiation of a class or in the left-hand side of a class scope resolution operator
— A port connection to an input or output port of a module, interface, or program
— The passing of a value to a subroutine input, output, or inout port
— A return statement in a function
— A tagged union expression
— For an expression that is used as the right-hand value in an assignment-like context:
  - If a parenthesized expression, then the expression within the parentheses
  - If a mintypmax expression, then the colon-separated expressions
  - If a conditional operator expression, then the second and third operand
— A nondefault correspondence between an expression in an assignment pattern and a field or element in a data object or data value

No other contexts shall be considered assignment-like contexts. In particular, none of the following shall be considered assignment-like contexts:

— A static cast
— A default correspondence between an expression in an assignment pattern and a field or element in a data object or data value
— A port expression in a module, interface, or program declaration
— The passing of a value to a subroutine **ref** port
— A port connection to an **inout** or **ref** port of a module, interface, or program

## 10.9 Assignment patterns

Assignment patterns are used for assignments to describe patterns of assignments to structure fields and array elements.

An assignment pattern specifies a correspondence between a collection of expressions and the fields and elements in a data object or data value. An assignment pattern has no self-determined data type, but can be used as one of the sides in an assignment-like context (see 10.8) when the other side has a self-determined data type. An assignment pattern is built from braces, keys, and expressions and is prefixed with an apostrophe. For example:

```
var int A[N] = '{default:1};
var integer i = '{31:1, 23:1, 15:1, 8:1, default:0};

typedef struct {real r, th;} C;
var C x = '{th:PI/2.0, r:1.0};
var real y [0:1] = '{0.0, 1.1}, z [0:9] = '{default: 3.1416};
```

A positional notation without keys can also be used. For example:

```
var int B[4] = '{a, b, c, d};
var C y = '{1.0, PI/2.0};
'{a, b, c, d} = B;
```

When an assignment pattern is used as the left-hand side of an assignment-like context, the positional notation shall be required; and each member expression shall have a bit-stream data type that is assignment compatible with and has the same number of bits as the data type of the corresponding element on the right-hand side.

The assignment pattern syntax is listed in Syntax 10-5.

assignment_pattern ::= *// from A.6.7.1*
    **'{** expression { **,** expression } **}**
  | **'{** structure_pattern_key **:** expression { **,** structure_pattern_key **:** expression } **}**
  | **'{** array_pattern_key **:** expression { **,** array_pattern_key **:** expression } **}**
  | **'{** constant_expression **{** expression { **,** expression } **} }**

structure_pattern_key ::= member_identifier | assignment_pattern_key

array_pattern_key ::= constant_expression | assignment_pattern_key

assignment_pattern_key ::= simple_type | **default**

assignment_pattern_expression ::=
    [ assignment_pattern_expression_type ] assignment_pattern

assignment_pattern_expression_type ::=
    ps_type_identifier
  | ps_parameter_identifier
  | integer_atom_type
  | type_reference

constant_assignment_pattern_expression[32] ::= assignment_pattern_expression

32) In a constant_assignment_pattern_expression, all member expressions shall be constant expressions.

*Syntax 10-5—Assignment patterns syntax (excerpt from Annex A)*

An assignment pattern can be used to construct or deconstruct a structure or array by prefixing the pattern with the name of a data type to form an assignment pattern expression. Unlike an assignment pattern, an assignment pattern expression has a self-determined data type and is not restricted to being one of the sides in an assignment-like context. When an assignment pattern expression is used in a right-hand expression, it shall yield the value that a variable of the data type would hold if it were initialized using the assignment pattern.

```
typedef logic [1:0] [3:0] T;
shortint'({T'{1,2}, T'{3,4}})     // yields 16'sh1234
```

When an assignment pattern expression is used in a left-hand expression, the positional notation shall be required; and each member expression shall have a bit-stream data type that is assignment compatible with and has the same number of bits as the corresponding element in the data type of the assignment pattern expression. If the right-hand expression has a self-determined data type, then it shall be assignment compatible with and have the same number of bits as the data type of the assignment pattern expression.

```
typedef byte U[3];
var U A = '{1, 2, 3};
var byte a, b, c;
U'{a, b, c} = A;
U'{c, a, b} = '{a+1, b+1, c+1};
```

An assignment pattern expression shall not be used in a port expression in a module, interface, or program declaration.

### 10.9.1 Array assignment patterns

Concatenation braces are used to construct and deconstruct simple bit vectors. A similar syntax is used to support the construction and deconstruction of arrays. The expressions shall match element for element, and the braces shall match the array dimensions. Each expression item shall be evaluated in the context of an assignment to the type of the corresponding element in the array. In other words, the following examples are not required to cause size warnings:

```
bit unpackedbits [1:0] = '{1,1};          // no size warning required as
                                          // bit can be set to 1
int unpackedints [1:0] = '{1'b1, 1'b1};   // no size warning required as
                                          // int can be set to 1'b1
```

A syntax resembling replications (see 11.4.12.1) can be used in array assignment patterns as well. Each replication shall represent an entire single dimension.

```
unpackedbits = '{2 {y}} ;          // same as '{y, y}
int n[1:2][1:3] = '{2{'{3{y}}}};   // same as '{'{y,y,y},'{y,y,y}}
```

SystemVerilog determines the context of the braces when used in the context of an assignment.

It can sometimes be useful to set array elements to a value without having to keep track of how many members there are. This can be done with the **`default`** keyword:

```
initial unpackedints = '{default:2};   // sets elements to 2
```

For arrays of structures, it is useful to specify one or more matching type keys, as described under structure assignment patterns following in 10.9.2.

```
struct {int a; time b;} abkey[1:0];
abkey = '{'{a:1, b:2ns}, '{int:5, time:$time}};
```

The matching rules are as follows:

— An `index:value` specifies an explicit value for a keyed element index. The value is evaluated in the context of an assignment to the indexed element and shall be castable to its type. It shall be an error to specify the same index more than once in a single array pattern expression.

— For `type:value`, if the element or subarray type of the array matches this type, then each element or subarray that has not already been set by an index key above shall be set to the value. The value shall be castable to the array element or subarray type. Otherwise, if the array is multidimensional, then there is a recursive descent into each subarray of the array using the rules in this subclause and the type and default keys. Otherwise, if the array is an array of structures, there is a recursive descent into each element of the array using the rules for structure assignment patterns and the type and default keys. If more than one type matches the same element, the last value shall be used.

— The **`default`**`:value` applies to elements or subarrays that are not matched by either index or type key. If the type of the element or subarray is a simple bit vector type, matches the self-determined type of the value, or is not an array or structure type, then the value is evaluated in the context of each assignment to an element or subarray by the default and shall be castable to the type of the element or subarray; otherwise, an error is generated. For unmatched subarrays, the type and default specifiers are applied recursively according to the rules in this subclause to each of its elements or subarrays. For unmatched structure elements, the type and default keys are applied to the element according to the rules for structures.

Every element shall be covered by one of these rules.

If the type key, default key, or replication operator is used on an expression with side effects, the number of times that expression evaluates is undefined.

### 10.9.2 Structure assignment patterns

A structure can be constructed and deconstructed with a structure assignment pattern built from member expressions using braces and commas, with the members in declaration order. Replication operators can be used to set the values for the exact number of members. Each member expression shall be evaluated in the context of an assignment to the type of the corresponding member in the structure. It can also be built with the names of the members.

```
module mod1;

   typedef struct {
      int x;
      int y;
   } st;

   st s1;
   int k = 1;

   initial begin
      #1 s1 = '{1, 2+k};          // by position
      #1 $display( s1.x, s1.y);
      #1 s1 = '{x:2, y:3+k};      // by name
      #1 $display( s1.x, s1.y);
      #1 $finish;
   end
endmodule
```

It can sometimes be useful to set structure members to a value without having to keep track of how many members there are or what the names are. This can be done with the `default` keyword:

```
initial s1 = '{default:2}; // sets x and y to 2
```

The `'{member:value}` or `'{data_type: default_value}` syntax can also be used:

```
ab abkey[1:0] = '{'{a:1, b:1.0}, '{int:2, shortreal:2.0}};
```

Use of the `default` keyword applies to members in nested structures or elements in unpacked arrays in structures.

```
struct {
   int A;
   struct {
      int B, C;
   } BC1, BC2;
} ABC, DEF;

ABC = '{A:1, BC1:'{B:2, C:3}, BC2:'{B:4,C:5}};
DEF = '{default:10};
```

To deal with the problem of members of different types, a type can be used as the key. This overrides the default for members of that type:

```
typedef struct {
   logic [7:0] a;
   bit b;
   bit signed [31:0] c;
   string s;
} sa;

sa s2;
initial s2 = '{int:1, default:0, string:""};       // set all to 0 except the
                                                   // array of bits to 1 and
                                                   // string to ""
```

Similarly, an individual member can be set to override the general default and the type default:

```
initial #10 s2 = '{default:'1, s : ""}; // set all to 1 except s to ""
```

SystemVerilog determines the context of the braces when used in the context of an assignment.

The matching rules are as follows:

— A `member:value` specifies an explicit value for a named member of the structure. The named member shall be at the top level of the structure; a member with the same name in some level of substructure shall not be set. The value shall be castable to the member type and is evaluated in the context of an assignment to the named member; otherwise, an error is generated.

— The `type:value` specifies an explicit value for each field in the structure whose type matches the type (see 6.22.1) and has not been set by a field name key above. If the same type key is mentioned more than once, the last value is used. The value is evaluated in the context of an assignment to the matching type.

— The **`default`**`:value` applies to members that are not matched by either member name or type key. If the member type is a simple bit vector type, matches the self-determined type of the value, or is not an array or structure type, then the value is evaluated in the context of each assignment to a member by the default and shall be castable to the member type; otherwise, an error is generated. For unmatched structure members, the type and default specifiers are applied recursively according to the rules in this subclause to each member of the substructure. For unmatched array members, the type and default keys are applied to the array according to the rules for arrays.

Every member shall be covered by one of these rules.

If the type key, default key, or replication operator is used on an expression with side effects, the number of times that expression evaluates is undefined.

## 10.10 Unpacked array concatenation

Unpacked array concatenation provides a flexible way to compose an unpacked array value from a collection of elements and arrays. An unpacked array concatenation may appear as the source expression in an assignment-like context and shall not appear in any other context. The target of such assignment-like context shall be an array whose slowest-varying dimension is an unpacked fixed-size, queue, or dynamic dimension. A target of any other type (including associative array) shall be illegal.

An unpacked array concatenation shall be written as a comma-separated list, enclosed in braces, of zero or more items. If the list has zero items, then the concatenation shall denote an array value with no elements. Otherwise, each item shall represent one or more elements of the resulting array value, interpreted as follows:

- — An item whose self-determined type is assignment compatible with the element type of the target array shall represent a single element
- — An item whose self-determined type is an unpacked array whose slowest-varying dimension's element type is assignment compatible with the element type of the target array shall represent as many elements as exist in that item, arranged in the same left-to-right order as they would appear in the array item itself
- — An item of any other type, or an item that has no self-determined type, shall be illegal except that the literal value **`null`** shall be legal if the target array's elements are of event, class, interface class, chandle or virtual interface type

The elements thus represented shall be arranged in left-to-right order to form the resulting array. It shall be an error if the size of the resulting array differs from the number of elements in a fixed-size target. If the size exceeds the maximum number of elements of a bounded queue, then elements beyond the upper bound of the target shall be ignored and a warning shall be issued.

### 10.10.1 Unpacked array concatenations compared with array assignment patterns

Array assignment patterns have the advantage that they can be used to create assignment pattern expressions of self-determined type by prefixing the pattern with a type name. Furthermore, items in an assignment pattern can be replicated using syntax, such as `'{ n{element} }`, and can be defaulted using the **`default:`** syntax. However, every element item in an array assignment pattern must be of the same type as the element type of the target array. By contrast, unpacked array concatenations forbid replication, defaulting, and explicit typing, but they offer the additional flexibility of composing an array value from an arbitrary mix of elements and arrays. In some simple cases both forms can have the same effect, as in the following example:

```
int A3[1:3];
A3 = {1, 2, 3};   // unpacked array concatenation: A3[1]=1, A3[2]=2, A3[3]=3
A3 = '{1, 2, 3};  // array assignment pattern: A3[1]=1, A3[2]=2, A3[3]=3
```

The next examples illustrate some differences between the two forms:

```
typedef int AI3[1:3];
AI3 A3;
int A9[1:9];

A3 = '{1, 2, 3};
A9 = '{3{A3}};                // illegal, A3 is wrong element type
A9 = '{A3, 4, 5, 6, 7, 8, 9}; // illegal, A3 is wrong element type
A9 = {A3, 4, 5, A3, 6};       // legal, gives A9='{1,2,3,4,5,1,2,3,6}
A9 = '{9{1}};                 // legal, gives A9='{1,1,1,1,1,1,1,1,1}
A9 = {9{1}};                  // illegal, no replication in unpacked
                              // array concatenation
A9 = {A3, {4,5,6,7,8,9} };    // illegal, {...} is not self-determined here
A9 = {A3, '{4,5,6,7,8,9} };   // illegal, '{...} is not self-determined
A9 = {A3, 4, AI3'{5, 6, 7}, 8, 9};  // legal, A9='{1,2,3,4,5,6,7,8,9}
```

Unpacked array concatenation is especially useful for writing values of queue type, as shown in the examples in 7.10.4.

### 10.10.2 Relationship with other constructs that use concatenation syntax

Concatenation syntax with braces can be used in other SystemVerilog constructs, including vector concatenation and string concatenation. These forms of concatenation are expressions of self-determined type, unlike unpacked array concatenation that does not have a self-determined type and that must appear as the source expression in an assignment-like context. If concatenation braces appear in an assignment-like context with an unpacked array target, they unambiguously act as unpacked array concatenation and must conform to the rules given in 10.10. Otherwise, they form a vector or string concatenation according to the rules given in 11.4.12. The following examples illustrate how the same expression can have different meanings in different contexts without ambiguity.

```
string S, hello;
string SA[2];
byte B;
byte BA[2];

hello = "hello";

S = {hello, " world"};  // string concatenation: "hello world"
SA = {hello, " world"}; // array concatenation:
                        // SA[0]="hello", SA[1]=" world"

B = {4'h6, 4'hf};       // vector concatenation: B=8'h6f
BA = {4'h6, 4'hf};      // array concatenation: BA[0]=8'h06, BA[1]=8'h0f
```

### 10.10.3 Nesting of unpacked array concatenations

Each item of an unpacked array concatenation shall have a self-determined type (see 10.10), but a complete unpacked array concatenation has no self-determined type. Consequently it shall be illegal for an unpacked array concatenation to appear as an item in another unpacked array concatenation. This rule makes it possible for a vector or string concatenation to appear as an item in an unpacked array concatenation without ambiguity, as illustrated in the following example.

```
string S1, S2;
typedef string T_SQ[$];
T_SQ SQ;

S1 = "S1";
S2 = "S2";
SQ = '{"element 0", "element 1"};      // assignment pattern, two strings
SQ = {S1, SQ, {"element 3 is ", S2} };
```

In the last line of the preceding example, the outer pair of braces encloses an unpacked array concatenation whereas the inner pair of braces encloses a string concatenation, so that the resulting queue of strings is

```
'{"S1", "element 0", "element 1", "element 3 is S2"}
```

Alternatively the third item in the unpacked array concatenation could instead represent an array of strings, if it were written as an assignment pattern expression. The unpacked array concatenation would still be valid in this case, but now it would treat its third item as an array of two strings, each forming one element of the resulting array:

```
SQ = {S1, SQ, T_SQ'{"element 3 is ", S2} };
   // result: '{"S1", "element 0", "element 1", "element 3 is ", "S2"}
```

With the exception of **`default:`** items, each item of an assignment pattern or an assignment pattern expression is in an assignment-like context (see 10.9). Consequently an unpacked array concatenation may appear as a non-default item in an assignment pattern. The following example uses a two-dimensional queue to build a jagged array of arrays of int, using both an assignment pattern expression and unpacked array concatenations to represent the subarrays:

```
typedef int T_QI[$];
T_QI jagged_array[$] = '{ {1}, T_QI'{2,3,4}, {5,6} };

   // jagged_array[0][0] = 1 -- jagged_array[0] is a queue of 1 int

   // jagged_array[1][0] = 2 -- jagged_array[1] is a queue of 3 ints
   // jagged_array[1][1] = 3
   // jagged_array[1][2] = 4

   // jagged_array[2][0] = 5 -- jagged_array[2] is a queue of 2 ints
   // jagged_array[2][1] = 6
```

## 10.11 Net aliasing

An alias statement declares multiple names for the same physical net, or bits within a net. The syntax for an alias statement is as follows:

```
net_alias ::= alias net_lvalue = net_lvalue { = net_lvalue } ;            // from A.6.1
net_lvalue ::=                                                            // from A.8.5
    ps_or_hierarchical_net_identifier constant_select
  | { net_lvalue { , net_lvalue } }
  | [ assignment_pattern_expression_type ] assignment_pattern_net_lvalue
```

*Syntax 10-6—Syntax for net aliasing (excerpt from Annex A)*

The continuous **assign** statement is a unidirectional assignment and can incorporate a delay and strength change. To model a bidirectional short-circuit connection, it is necessary to use the **alias** statement. The members of an alias list are signals whose bits share the same physical nets. The following example implements a byte order swapping between bus A and bus B:

```
module byte_swap (inout wire [31:0] A, inout wire [31:0] B);
   alias {A[7:0],A[15:8],A[23:16],A[31:24]} = B;
endmodule
```

This example strips out the LSB and MSB from a 4-byte bus:

```
module byte_rip (inout wire [31:0] W, inout wire [7:0] LSB, MSB);
   alias W[7:0] = LSB;
   alias W[31:24] = MSB;
endmodule
```

The bit overlay rules are the same as for a packed union with the same member types: each member shall be the same size, and connectivity is independent of the simulation host. The nets connected with an **alias** statement shall be type compatible, that is, they have to be of the same net type. For example, it is illegal to connect a **wand** net to a **wor** net with an **alias** statement. This rule is stricter than the rule applied to nets joining at ports because the scope of an alias is limited and such connections are more likely to be a design error. Variables and hierarchical references cannot be used in **alias** statements. Any violation of these rules shall be considered a fatal error.

The same nets can appear in multiple **alias** statements. The effects are cumulative. The following two examples are equivalent. In either case, `low12[11:4]` and `high12[7:0]` share the same wires.

```
module overlap(inout wire [15:0] bus16, inout wire [11:0] low12, high12);
   alias bus16[11:0] = low12;
   alias bus16[15:4] = high12;
endmodule

module overlap(inout wire [15:0] bus16, inout wire [11:0] low12, high12);
   alias bus16 = {high12, low12[3:0]};
   alias high12[7:0] = low12[11:4];
endmodule
```

To avoid errors in specification, it is not allowed to specify an alias from an individual signal to itself or to specify a given alias more than once. The following version of the preceding code would be illegal because the top 4 bits and bottom 4 bits are the same in both statements:

```
alias bus16 = {high12[11:8], low12};
alias bus16 = {high12, low12[3:0]};
```

This alternative is also illegal because the bits of `bus16` are being aliased to itself:

```
alias bus16 = {high12, bus16[3:0]} = {bus16[15:12], low12};
```

**alias** statements can appear anywhere module instance statements can appear. If an identifier that has not been declared as a data type appears in an **alias** statement, then an implicit net is assumed, following the same rules as implicit nets for a module instance. The following example uses **alias** along with the automatic name binding to connect pins on cells from different libraries to create a standard macro:

```
module lib1_dff(Reset, Clk, Data, Q, Q_Bar);
   ...
endmodule

module lib2_dff(reset, clock, data, q, qbar);
   ...
endmodule

module lib3_dff(RST, CLK, D, Q, Q_);
   ...
endmodule

module my_dff(rst, clk, d, q, q_bar); // wrapper cell
   input rst, clk, d;
   output q, q_bar;
   alias rst = Reset = reset = RST;
   alias clk = Clk = clock = CLK;
   alias d = Data = data = D;
   alias q = Q;
   alias Q_ = q_bar = Q_Bar = qbar;
   `LIB_DFF my_dff (.*); // LIB_DFF is any of lib1_dff, lib2_dff or lib3_dff
endmodule
```

Using a net in an **alias** statement does not modify its syntactic behavior in other statements. Aliasing is performed at elaboration time and cannot be undone.

# 11. Operators and expressions

## 11.1 General

This clause describes the following:

- Expression semantics
- Operations on expressions
- Operator precedence
- Operand size extension rules
- Signed and unsigned operation rules
- Bit and part-select operations and longest static prefix
- Bit-stream operations

## 11.2 Overview

This clause describes the operators and operands available in SystemVerilog and how to use them to form expressions.

An *expression* is a construct that combines *operands* with *operators* to produce a result that is a function of the values of the operands and the semantic meaning of the operator. Any legal operand, such as a net bit-select, without any operator is considered an expression. Wherever a value is needed in a SystemVerilog statement, an expression can be used.

An *operand* can be one of the following:

- Constant literal number, including real literals
- String literal
- Parameter, including local and specify parameters
- Parameter bit-select or part-select, including local and specify parameters
- Net (see 6.7)
- Net bit-select or part-select
- Variable (see 6.8)
- Variable bit-select or part-select
- Structure, either packed or unpacked
- Structure member
- Packed structure bit-select or part-select
- Union, packed, unpacked, or tagged
- Union member
- Packed union bit-select or part-select
- Array, either packed or unpacked
- Packed array bit-select, part-select, element, or slice
- Unpacked array element bit-select or part-select, element, or slice
- A call to a user-defined function, system-defined function, or method that returns any of the above

### 11.2.1 Constant expressions

Some statement constructs require an expression to be a *constant expression*. The operands of a constant expression consist of constant numbers, strings, parameters, constant bit-selects and part-selects of parameters, *constant function calls* (see 13.4.3), and *constant system function calls* only. Constant expressions can use any of the operators defined in Table 11-1.

*Constant system function calls* are calls to certain built-in system functions where the arguments meet conditions outlined in this subclause. When used in constant expressions, these function calls shall be evaluated at elaboration time. The system functions that may be used in constant system function calls are *pure functions*, i.e., those whose value depends only on their input arguments and that have no side effects.

Certain built-in system functions where the arguments are constant expressions are constant system function calls. Specifically, these are the conversion system functions listed in 20.5, the mathematical system functions listed in 20.8, and the bit vector system functions listed in 20.9.

The data query system functions listed in 20.6 and the array query system functions listed in 20.7 are normally also constant system function calls even when their arguments are not constant. See those subclauses for the conditions under which these query system function calls are considered to be constant expressions.

### 11.2.2 Aggregate expressions

Unpacked structure and array data objects, as well as unpacked structure and array constructors, can all be used as aggregate expressions. A multi-element slice of an unpacked array can also be used as an aggregate expression.

Aggregate expressions can be copied in an assignment, through a port, or as an argument to a subroutine. Aggregate expressions can also be compared with equality or inequality operators.

If the two operands of a comparison operator are aggregate expressions, they shall be of equivalent type as defined in 6.22.2. Assignment compatibility of aggregate expressions is defined in 6.22.3 and, for arrays, in 7.6.

## 11.3 Operators

The symbols for the SystemVerilog operators are similar to those in the C programming language. Syntax 11-1 and Table 11-1 list these operators.

```
assignment_operator ::=                                                // from A.6.2
    = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>= | <<<= | >>>=
conditional_expression ::=                                             // from A.8.3
    cond_predicate ? { attribute_instance } expression : expression
unary_operator ::=                                                     // from A.8.6
    + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
binary_operator ::=
    + | - | * | / | % | == | != | === | !== | ==? | !=? | && | || | **
  | < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | >>> | <<<
  | -> | <->
inc_or_dec_operator ::= ++ | --
stream_operator ::= >> | <<                                            // from A.8.1
```

*Syntax 11-1—Operator syntax (excerpt from Annex A)*

**Table 11-1—Operators and data types**

| Operator token | Name | Operand data types |
|---|---|---|
| `=` | Binary assignment operator | Any |
| `+=  -=  /=  *=` | Binary arithmetic assignment operators | Integral, **`real`**, **`shortreal`** |
| `%=` | Binary arithmetic modulus assignment operator | Integral |
| `&=  \|=  ^=` | Binary bitwise assignment operators | Integral |
| `>>= <<=` | Binary logical shift assignment operators | Integral |
| `>>>= <<<=` | Binary arithmetic shift assignment operators | Integral |
| `?:` | Conditional operator | Any |
| `+  -` | Unary arithmetic operators | Integral, **`real`**, **`shortreal`** |
| `!` | Unary logical negation operator | Integral, **`real`**, **`shortreal`** |
| `~  &  ~&  \|  ~\|  ^`<br>`~^  ^~` | Unary logical reduction operators | Integral |
| `+  -  *  /  **` | Binary arithmetic operators | Integral, **`real`**, **`shortreal`** |
| `%` | Binary arithmetic modulus operator | Integral |
| `&  \|  ^  ^~  ~^` | Binary bitwise operators | Integral |
| `>> <<` | Binary logical shift operators | Integral |
| `>>> <<<` | Binary arithmetic shift operators | Integral |
| `&&  \|\|`<br>`-> <->` | Binary logical operators | Integral, **`real`**, **`shortreal`** |
| `<  <=  >  >=` | Binary relational operators | Integral, **`real`**, **`shortreal`** |
| `===  !==` | Binary case equality operators | Any except **`real`** and **`shortreal`** |
| `==  !=` | Binary logical equality operators | Any |
| `==?  !=?` | Binary wildcard equality operators | Integral |
| `++  --` | Unary increment, decrement operators | Integral, **`real`**, **`shortreal`** |
| **`inside`** | Binary set membership operator | Singular for the left operand |
| **`dist`**[a] | Binary distribution operator | Integral |
| `{}  {{}}` | Concatenation, replication operators | Integral |
| `{<<{}} {>>{}}` | Stream operators | Integral |

[a]The **`dist`** operator is described in 16.14.2 and 18.5.4.

### 11.3.1 Operators with real operands

Table 11-1 shows what operators may be applied to real operands.

The result of using logical or relational operators or the **`inside`** operator on real operands shall be a single-bit scalar value.

For other operators, if any operand, except before the ? in the conditional operator, is **real**, the result is **real**. Otherwise, if any operand, except before the ? in the conditional operator, is **shortreal**, the result is **shortreal**.

Real operands can also be used in the following expressions:

```
    str.realval // structure or union member
    realarray[intval] // array element
```

See 6.12.1 for more information on use of real numbers.

### 11.3.2 Operator precedence

Operator precedence and associativity are listed in Table 11-2. The highest precedence is listed first.

**Table 11-2—Operator precedence and associativity**

| Operator | Associativity | Precedence |
|---|---|---|
| `()` `[]` `::` `.` | Left | Highest |
| `+` `-` `!` `~` `&` `~&` `\|` `~\|` `^` `~^` `^~` `++` `--` (unary) | | |
| `**` | Left | |
| `*` `/` `%` | Left | |
| `+` `-` (binary) | Left | |
| `<<` `>>` `<<<` `>>>` | Left | |
| `<` `<=` `>` `>=` **`inside`** **`dist`** | Left | |
| `==` `!=` `===` `!==` `==?` `!=?` | Left | |
| `&` (binary) | Left | |
| `^` `~^` `^~` (binary) | Left | |
| `\|` (binary) | Left | |
| `&&` | Left | |
| `\|\|` | Left | |
| `?:` (conditional operator) | Right | |
| `->` `<->` | Right | |
| `=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `\|=` `<<=` `>>=` `<<<=` `>>>=` `:=` `:/` `<=` | None | |
| `{}` `{{}}` | Concatenation | Lowest |

Operators shown on the same row in Table 11-2 shall have the same precedence. Rows are arranged in order of decreasing precedence for the operators. For example, *****, /, and **%** all have the same precedence, which is higher than that of the binary + and – operators.

All operators shall associate left to right with the exception of the conditional (`?:`), implication (`->`), and equivalence (`<->`) operators, which shall associate right to left. Associativity refers to the order in which the operators having the same precedence are evaluated. Thus, in the following example, `B` is added to `A`, and then `C` is subtracted from the result of `A+B`.

```
A + B - C
```

When operators differ in precedence, the operators with higher precedence shall associate first. In the following example, `B` is divided by `C` (division has higher precedence than addition), and then the result is added to `A`.

```
A + B / C
```

Parentheses can be used to change the operator precedence.

```
(A + B) / C        // not the same as A + B / C
```

### 11.3.3 Using integer literals in expressions

Integer literals can be used as operands in expressions. An integer literal can be expressed as the following:

- An unsized, unbased integer (e.g., `12`)
- An unsized, based integer (e.g., `'d12`, `'sd12`)
- A sized, based integer (e.g., `16'd12`, `16'sd12`)

See 5.7.1 for integer literal syntax.

A negative value for an integer with no base specifier shall be interpreted differently from an integer with a base specifier. An integer with no base specifier shall be interpreted as a signed value in two's-complement form. An integer with an unsigned base specifier shall be interpreted as an unsigned value.

The following example shows four ways to write the expression "minus 12 divided by 3." Note that `-12` and `-'d12` both evaluate to the same two's-complement bit pattern, but, in an expression, the `-'d12` loses its identity as a signed negative number.

```
int IntA;
IntA = -12 / 3;         // The result is -4

IntA = -'d 12 / 3;      // The result is 1431655761

IntA = -'sd 12 / 3;     // The result is -4

IntA = -4'sd 12 / 3;    // -4'sd12 is the negative of the 4-bit
                        // quantity 1100, which is -4. -(-4) = 4
                        // The result is 1
```

### 11.3.4 Operations on logic (4-state) and bit (2-state) types

Operators may be applied to 2-state values or to a mixture of 2-state and 4-state values. The result is the same as if all values were treated as 4-state values. In most cases, if all operands are 2-state, the result is in the 2-state value set. The only exceptions involve operators that produce an `x` result for operands in the 2-state value set (e.g., division by zero).

```
int n = 8, zero = 0;
int res = 'b01xz | n;         // res gets 'b11xz coerced to int, or 'b1100
int sum = n + n;              // sum gets 16 coerced to int, or 16
int sumx = 'x + n;            // sumx gets 'x coerced to int, or 0
int div2 = n/zero + n;        // div2 gets 'x coerced to int, or 0
integer div4 = n/zero + n;    // div4 gets 'x
```

### 11.3.5 Operator expression short circuiting

The operators shall follow the associativity rules while evaluating an expression as described in 11.3.2. Some operators (`&&`, `||`, `->`, and `?:`) shall use *short-circuit evaluation*; in other words, some of their operand expressions shall not be evaluated if their value is not required to determine the final value of the operation. The detailed short-circuiting behavior of each of these operators is described in its corresponding subclause (11.4.7 and 11.4.11). All other operators shall not use short-circuit evaluation—all of their operand expressions are always evaluated. When short circuiting occurs, any side effects or runtime errors that would have occurred due to evaluation of the short-circuited operand expression shall not occur.

For example:

```
logic regA, regB, regC, result ;

function logic myFunc(logic x);
   ...
endfunction

result = regA & (regB | myFunc(regC)) ;
```

Even if `regA` is known to be zero, the subexpression (`regB | myFunc(regC)`) will be evaluated and any side effects caused by calling `myFunc(regC)` will occur.

Note that implementations are free to optimize by omitting evaluation of subexpressions as long as the simulation behavior (including side effects) is as if the standard rules were followed.

### 11.3.6 Assignment within an expression

An expression can include a blocking assignment, provided it does not have a timing control. These blocking assignments shall be enclosed in parentheses to avoid common mistakes such as using `a=b` for `a==b` or using `a|=b` for `a!=b`.

```
if ((a=b)) b = (a+=1);

a = (b = (c = 5));
```

The semantics of such an assignment expression is that of a function that evaluates the right-hand side, casts the right-hand side to the left-hand data type, stacks it, updates the left-hand side, and returns the stacked value. The data type of the value that is returned is the data type of the left-hand side. If the left-hand side is a concatenation, then the data type of the value that is returned shall be an unsigned integral data type whose bit length is the sum of the length of its operands.

It shall be illegal to include an assignment operator in an event expression, in an expression within a procedural continuous assignment, or in an expression that is not within a procedural statement.

## 11.4 Operator descriptions

### 11.4.1 Assignment operators

In addition to the simple assignment operator, =, SystemVerilog includes the C assignment operators and special bitwise assignment operators: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `<<<=`, and `>>>=`. An assignment operator is semantically equivalent to a blocking assignment, with the exception that any left-hand index expression is only evaluated once. For example:

```
a[i]+=2; // same as a[i] = a[i] +2;
```

### 11.4.2 Increment and decrement operators

SystemVerilog includes the C increment and decrement assignment operators `++i`, `--i`, `i++`, and `i--`. These do not need parentheses when used in expressions. These increment and decrement assignment operators behave as blocking assignments.

The ordering of assignment operations relative to any other operation within an expression is undefined. An implementation can warn whenever a variable is both written and read-or-written within an integral expression or in other contexts where an implementation cannot guarantee order of evaluation. For example:

```
i = 10;
j = i++ + (i = i - 1);
```

After execution, the value of `j` can be 18, 19, or 20 depending upon the relative ordering of the increment and the assignment statements.

The increment and decrement operators, when applied to real operands, increment or decrement the operand by 1.0.

### 11.4.3 Arithmetic operators

The binary arithmetic operators are given in Table 11-3.

**Table 11-3—Arithmetic operators defined**

| | |
|---|---|
| a + b | a plus b |
| a − b | a minus b |
| a * b | a multiplied by b (or a times b) |
| a / b | a divided by b |
| a % b | a modulo b |
| a ** b | a to the power of b |

The integer division shall truncate any fractional part toward zero. For the division or modulus operators, if the second operand is a zero, then the entire result value shall be `x`. The modulus operator (for example, `a % b`) gives the remainder when the first operand is divided by the second and thus is zero when `b` divides `a` exactly. The result of a modulus operation shall take the sign of the first operand.

If either operand of the power operator is real, then the result type shall be real (see 11.3.1). The result of the power operator is unspecified if the first operand is zero and the second operand is nonpositive or if the first operand is negative and the second operand is not an integral value.

If neither operand of the power operator is real, then the result type shall be determined as outlined in 11.6.1 and 11.8.1. The result value is `'x` if the first operand is zero and the second operand is negative. The result value is 1 if the second operand is zero.

In all cases, the second operand of the power operator shall be treated as self-determined.

These statements are illustrated in Table 11-4.

**Table 11-4—Power operator rules**

| | **op1 is negative < −1** | **op1 is −1** | **op1 is zero** | **op1 is 1** | **op1 is positive > 1** |
|---|---|---|---|---|---|
| **op2 is positive** | op1 ** op2 | op2 is odd -> −1<br>op2 is even -> 1 | 0 | 1 | op1 ** op2 |
| **op2 is zero** | 1 | 1 | 1 | 1 | 1 |
| **op2 is negative** | 0 | op2 is odd -> −1<br>op2 is even -> 1 | 'x | 1 | 0 |

The unary arithmetic operators shall take precedence over the binary operators. The unary operators are given in Table 11-5.

**Table 11-5—Unary operators defined**

| | |
|---|---|
| +m | Unary plus m (same as m) |
| −m | Unary minus m |

For the arithmetic operators, if any operand bit value is the unknown value x or the high-impedance value `z`, then the entire result value shall be `x`.

Table 11-6 gives examples of some modulus and power operations.

**Table 11-6—Examples of modulus and power operators**

| **Expression** | **Result** | **Comments** |
|---|---|---|
| 10 % 3 | 1 | 10/3 yields a remainder of 1. |
| 11 % 3 | 2 | 11/3 yields a remainder of 2. |
| 12 % 3 | 0 | 12/3 yields no remainder. |
| –10 % 3 | –1 | The result takes the sign of the first operand. |
| 11 % –3 | 2 | The result takes the sign of the first operand. |
| –4'd12 % 3 | 1 | –4'd12 is seen as a large positive number that leaves a remainder of 1 when divided by 3. |
| 3 ** 2 | 9 | 3 × 3 |
| 2 ** 3 | 8 | 2 × 2 × 2 |
| 2 ** 0 | 1 | Anything to the zero exponent is 1. |
| 0 ** 0 | 1 | Zero to the zero exponent is also 1. |
| 2.0 ** –3'sb1 | 0.5 | 2.0 is real, giving real reciprocal. |
| 2 ** –3 'sb1 | 0 | 2 ** –1 = 1/2. Integer division truncates to zero. |
| 0 ** –1 | 'x | 0 ** –1 = 1/0. Integer division by zero is 'x. |
| 9 ** 0.5 | 3.0 | Real square root. |

**Table 11-6—Examples of modulus and power operators *(continued)***

| Expression | Result | Comments |
|---|---|---|
| 9.0 ** (1/2) | 1.0 | Integer division truncates exponent to zero. |
| –3.0 ** 2.0 | 9.0 | Defined because real 2.0 is still integral value. |

### 11.4.3.1 Arithmetic expressions with unsigned and signed types

Nets and variables can be explicitly declared as unsigned or signed. The **`byte`**, **`shortint`**, **`int`**, **`integer`**, and **`longint`** data types are signed by default. Other data types are unsigned by default.

A value assigned to an unsigned variable or net shall be treated as an *unsigned* value. A value assigned to a signed variable or net shall be treated as *signed*. Signed values, except for those assigned to real variables, shall use a two's-complement representation. Values assigned to real variables shall use a floating-point representation. Conversions between signed and unsigned values shall keep the same bit representation; only the interpretation changes.

Table 11-7 lists how arithmetic operators interpret each data type.

**Table 11-7—Data type interpretation by arithmetic operators**

| Data type | Interpretation |
|---|---|
| Unsigned net | Unsigned |
| Signed net | Signed, two's-complement |
| Unsigned variable | Unsigned |
| Signed variable | Signed, two's-complement |
| Real variable | Signed, floating point |

The following example shows various ways to divide "minus twelve by three"—using **`integer`** and **`logic`** variables in expressions.

```
integer intS;
var logic [15:0] U;
var logic signed [15:0] S;

intS = -4'd12;
U = intS / 3;         // expression result is -4,
                      // intS is an integer data type, U is 65532

U = -4'd12;           // U is 65524
intS = U / 3;         // expression result is 21841,
                      // U is a logic data type

intS = -4'd12 / 3;    // expression result is 1431655761.
                      // -4'd12 is effectively a 32-bit logic data type

U = -12 / 3;          // expression result is -4, -12 is effectively
                      // an integer data type. U is 65532

S = -12 / 3;          // expression result is -4. S is a signed logic
```

```
S = -4'sd12 / 3;      // expression result is 1. -4'sd12 is actually 4.
                      // The rules for integer division yield 4/3==1
```

### 11.4.4 Relational operators

Table 11-8 lists and defines the relational operators.

**Table 11-8—Definitions of relational operators**

| | |
|---|---|
| a < b | a less than b |
| a > b | a greater than b |
| a <= b | a less than or equal to b |
| a >= b | a greater than or equal to b |

An expression using these *relational operators* shall yield the scalar value 0 if the specified relation is false or the value 1 if it is true. If either operand of a relational operator contains an unknown (x) or high-impedance (z) value, then the result shall be a 1-bit unknown value (x).

When one or both operands of a relational expression are unsigned, the expression shall be interpreted as a comparison between unsigned values. If the operands are of unequal bit lengths, the smaller operand shall be zero-extended to the size of the larger operand.

When both operands are signed, the expression shall be interpreted as a comparison between signed values. If the operands are of unequal bit lengths, the smaller operand shall be sign-extended to the size of the larger operand. See 11.8.2 for more information.

If either operand is a real operand, then the other operand shall be converted to an equivalent real value and the expression shall be interpreted as a comparison between real values.

All the relational operators shall have the same precedence. Relational operators shall have lower precedence than arithmetic operators.

The following examples illustrate the implications of this precedence rule:

```
a < b - 1             // this expression is the same as
a < (b - 1)           // this expression, but . . .
b - (1 < a)           // this one is not the same as
b - 1 < a             // this expression
```

When `b - (1 < a)` evaluates, the relational expression evaluates first, and then either zero or one is subtracted from `b`. When `b - 1 < a` evaluates, the value of `b` operand is reduced by one and then compared with `a`.

### 11.4.5 Equality operators

The *equality operators* shall rank lower in precedence than the relational operators. Table 11-9 lists and defines the equality operators.

**Table 11-9—Definitions of equality operators**

| | |
|---|---|
| a === b | a equal to b, including x and z |
| a !== b | a not equal to b, including x and z |
| a == b | a equal to b, result can be unknown |
| a != b | a not equal to b, result can be unknown |

All four equality operators shall have the same precedence. These four operators compare operands bit for bit. As with the relational operators, the result shall be 0 if comparison fails and 1 if it succeeds.

When one or both operands are unsigned, the expression shall be interpreted as a comparison between unsigned values. If the operands are of unequal bit lengths, the smaller operand shall be zero-extended to the size of the larger operand.

When both operands are signed, the expression shall be interpreted as a comparison between signed values. If the operands are of unequal bit lengths, the smaller operand shall be sign-extended to the size of the larger operand. See 11.8.2 for more information.

If either operand is a real operand, then the other operand shall be converted to an equivalent real value, and the expression shall be interpreted as a comparison between real values.

The logical equality (or case equality) operator is a legal operation if either operand is a class handle or the literal **null**, and one of the operands is assignment compatible with the other. The logical equality (or case equality) operator is a legal operation if either operand is a **chandle** or the literal **null**. In both cases, the operator compares the values of the class handles, interface class handles, or chandles.

For the *logical equality* and *logical inequality* operators (== and !=), if, due to unknown or high-impedance bits in the operands, the relation is ambiguous, then the result shall be a 1-bit unknown value (x).

For the *case equality* and *case inequality* operators (=== and !==), the comparison shall be done just as it is in the procedural case statement (see 12.5). Bits that are x or z shall be included in the comparison and shall match for the result to be considered equal. The result of these operators shall always be a known value, either 1 or 0.

### 11.4.6 Wildcard equality operators

The *wildcard equality operators* shall have the same precedence as the equality operators. Table 11-10 lists and defines the wildcard equality operators.

**Table 11-10—Wildcard equality and wildcard inequality operators**

| Operator | Usage | Description |
|---|---|---|
| ==? | a ==? b | a equals b, X and Z values in b act as wildcards |
| !=? | a !=? b | a does not equal b, X and Z values in b act as wildcards |

The wildcard equality operator (==?) and inequality operator (!=?) treat X and Z values in a given bit position of their right operand as a wildcard. X and Z values in the left operand are not treated as wildcards.

A wildcard bit matches any bit value (0, 1, `Z`, or `X`) in the corresponding bit of the left operand being compared against it. Any other bits are compared as for the logical equality and logical inequality operators.

These operators compare operands bit for bit and return a 1-bit self-determined result. If the operands to the wildcard equality/inequality are of unequal bit length, the operands are extended in the same manner as for the logical equality/inequality operators. If the relation is true, the operator yields a 1. If the relation is false, it yields a 0. If the relation is unknown, it yields `X`.

The different types of equality (and inequality) operators in SystemVerilog behave differently when their operands contain unknown values (`X` or `Z`). The `==` and `!=` operators may result in `x` if any of their operands contains an `X` or `Z`. The `===` and `!==` operators explicitly check for 4-state values; therefore, `X` and `Z` values shall either match or mismatch, never resulting in `X`. The `==?` and `!=?` operators may result in `X` if the left operand contains an `x` or `Z` that is not being compared with a wildcard in the right operand.

The wildcard equality operator is equivalent to the logical equality operator if its operands are class handles, interface class handles, chandles or the literal **`null`**.

### 11.4.7 Logical operators

The operators *logical AND* (`&&`), *logical OR* (`||`), *logical implication* (`->`), and *logical equivalence* (`<->`) are logical connectives. The result of the evaluation of a logical operation shall be 1 (defined as true), 0 (defined as false), or, if the result is ambiguous, the unknown value (`x`). The precedence of `&&` is greater than that of `||`, and both are lower than relational and equality operators. The precedence of `->` and `<->` is at the same level, the binding of operands between the two operations is governed by associativity (right), both are lower than other logical operators and the conditional operator.

The logical implication `expression1 -> expression2` is logically equivalent to `(!expression1 || expression2)`, and the logical equivalence `expression1 <-> expression2` is logically equivalent to `((expression1 -> expression2) && (expression2 -> expression1))`. Each of the two operands of the logical equivalence operator shall be evaluated exactly once.

The unary *logical negation* operator `(!)` converts a nonzero or true operand into 0 and a zero or false operand into 1. An ambiguous truth value remains as `x`.

*Example 1*: If variable `alpha` holds the integer value 237 and `beta` holds the value zero, then the following examples perform as described:

```
regA = alpha && beta;      // regA is set to 0
regB = alpha || beta;      // regB is set to 1
```

*Example 2:* The following expression performs a logical and of three subexpressions without needing any parentheses:

```
a < size-1 && b != c && index != lastone
```

However, it is recommended for readability purposes that parentheses be used to show very clearly the precedence intended, as in the following rewrite of this example:

```
(a < size-1) && (b != c) && (index != lastone)
```

*Example 3:* A common use of `!` is in constructions like the following:

```
if (!inword)
```

In some cases, the preceding construct makes more sense to someone reading the code than this equivalent construct:

```
if (inword == 0)
```

The `&&` and `||` operators shall use short circuit evaluation as follows:

— The first operand expression shall always be evaluated.

— For `&&`, if the first operand value is logically false then the second operand shall not be evaluated.

— For `||`, if the first operand value is logically true then the second operand shall not be evaluated.

### 11.4.8 Bitwise operators

The *bitwise operators* shall perform bitwise manipulations on the operands; that is, the operator shall combine a bit in one operand with its corresponding bit in the other operand to calculate 1 bit for the result. Table 11-11 through Table 11-15 show the results for each possible calculation.

**Table 11-11—Bitwise binary AND operator**

| & | 0 | 1 | x | z |
|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 |
| **1** | 0 | 1 | x | x |
| **x** | 0 | x | x | x |
| **z** | 0 | x | x | x |

**Table 11-12—Bitwise binary OR operator**

| \| | 0 | 1 | x | z |
|---|---|---|---|---|
| **0** | 0 | 1 | x | x |
| **1** | 1 | 1 | 1 | 1 |
| **x** | x | 1 | x | x |
| **z** | x | 1 | x | x |

**Table 11-13—Bitwise binary exclusive OR operator**

| **^** | **0** | **1** | **x** | **z** |
|---|---|---|---|---|
| **0** | 0 | 1 | x | x |
| **1** | 1 | 0 | x | x |
| **x** | x | x | x | x |
| **z** | x | x | x | x |

**Table 11-14—Bitwise binary exclusive NOR operator**

| **^~** <br> **~^** | **0** | **1** | **x** | **z** |
|---|---|---|---|---|
| **0** | 1 | 0 | x | x |
| **1** | 0 | 1 | x | x |
| **x** | x | x | x | x |
| **z** | x | x | x | x |

**Table 11-15—Bitwise unary negation operator**

| ~ | |
|---|---|
| 0 | 1 |
| 1 | 0 |
| x | x |
| z | x |

### 11.4.9 Reduction operators

The *unary reduction operators* shall perform a bitwise operation on a single operand to produce a single-bit result. For *reduction AND*, *reduction OR*, and *reduction XOR* operators, the first step of the operation shall apply the operator between the first bit of the operand and the second using Table 11-16 through Table 11-18. The second and subsequent steps shall apply the operator between the 1-bit result of the prior step and the next bit of the operand using the same logic table. For *reduction NAND*, *reduction NOR*, and *reduction XNOR* operators, the result shall be computed by inverting the result of the reduction AND, reduction OR, and reduction XOR operation, respectively.

**Table 11-16—Reduction unary AND operator**

| & | 0 | 1 | x | z |
|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 |
| **1** | 0 | 1 | x | x |
| **x** | 0 | x | x | x |
| **z** | 0 | x | x | x |

**Table 11-17—Reduction unary OR operator**

| \| | 0 | 1 | x | z |
|---|---|---|---|---|
| **0** | 0 | 1 | x | x |
| **1** | 1 | 1 | 1 | 1 |
| **x** | x | 1 | x | x |
| **z** | x | 1 | x | x |

**Table 11-18—Reduction unary exclusive OR operator**

| ^ | 0 | 1 | x | z |
|---|---|---|---|---|
| **0** | 0 | 1 | x | x |
| **1** | 1 | 0 | x | x |
| **x** | x | x | x | x |
| **z** | x | x | x | x |

For example, Table 11-19 shows the results of applying reduction operators on different operands.

**Table 11-19—Results of unary reduction operations**

| Operand | & | ~& | \| | ~\| | ^ | ~^ | Comments |
|---|---|---|---|---|---|---|---|
| 4'b0000 | 0 | 1 | 0 | 1 | 0 | 1 | No bits set |
| 4'b1111 | 1 | 0 | 1 | 0 | 0 | 1 | All bits set |
| 4'b0110 | 0 | 1 | 1 | 0 | 0 | 1 | Even number of bits set |
| 4'b1000 | 0 | 1 | 1 | 0 | 1 | 0 | Odd number of bits set |

### 11.4.10 Shift operators

There are two types of *shift operators*: the logical shift operators, << and >>, and the arithmetic shift operators, <<< and >>>. The left shift operators, << and <<<, shall shift their left operand to the left by the number of bit positions given by the right operand. In both cases, the vacated bit positions shall be filled with zeros. The right shift operators, >> and >>>, shall shift their left operand to the right by the number of bit positions given by the right operand. The logical right shift shall fill the vacated bit positions with zeros. The arithmetic right shift shall fill the vacated bit positions with zeros if the result type is unsigned. It shall fill the vacated bit positions with the value of the most significant (i.e., *sign*) bit of the left operand if the result type is signed. If the right operand has an `x` or `z` value, then the result shall be unknown. The right operand is always treated as an unsigned number and has no effect on the signedness of the result. The result signedness is determined by the left-hand operand and the remainder of the expression, as outlined in 11.8.1.

*Example 1:* In this example, the variable `result` is assigned the binary value `0100`, which is `0001` shifted to the left two positions and zero-filled.

```
module shift;
   logic [3:0] start, result;
   initial begin
      start = 1;
      result = (start << 2);
   end
endmodule
```

*Example 2*: In this example, the variable `result` is assigned the binary value `1110`, which is `1000` shifted to the right two positions and sign-filled.

```
module ashift;
   logic signed [3:0] start, result;
   initial begin
      start = 4'b1000;
      result = (start >>> 2);
   end
endmodule
```

### 11.4.11 Conditional operator

The *conditional operator* shall be right associative and shall be constructed using three operands separated by two operators in the format given in Syntax 11-2.

```
conditional_expression ::=                                                  // from A.8.3
    cond_predicate ? { attribute_instance } expression : expression
cond_predicate ::=                                                          // from A.6.6
    expression_or_cond_pattern { &&& expression_or_cond_pattern }
expression_or_cond_pattern ::=
    expression | cond_pattern
cond_pattern ::= expression matches pattern
```

*Syntax 11-2—Conditional operator syntax (excerpt from Annex A)*

This subclause describes the traditional notation where *cond_predicate* is just a single expression. SystemVerilog also allows *cond_predicate* to perform pattern matching, which is described in 12.6.

If *cond_predicate* is true, the operator returns the value of the first *expression* without evaluating the second expression; if false, it returns the value of the second *expression* without evaluating the first expression. If *cond_predicate* evaluates to an ambiguous value (`x` or `z`), then both the first *expression* and the second *expression* shall be evaluated, and compared for logical equivalence as described in 11.4.5. If that comparison is true (1), the operator shall return either the first or second *expression*. Otherwise the operator returns a result based on the data types of the expressions.

When both the first and second expressions are of integral types, if the *cond_predicate* evaluates to an ambiguous value and the *expressions* are not logically equivalent, their results shall be combined bit by bit using Table 11-20 to calculate the final result. The first and second expressions are extended to the same width, as described in 11.6.1 and 11.8.2.

**Table 11-20—Ambiguous condition results for conditional operator**

| **?:** | **0** | **1** | **x** | **z** |
|---|---|---|---|---|
| **0** | 0 | x | x | x |
| **1** | x | 1 | x | x |
| **x** | x | x | x | x |
| **z** | x | x | x | x |

The following example of a three-state output bus illustrates a common use of the conditional operator:

```
wire [15:0] busa = drive_busa ? data : 16'bz;
```

The bus called `data` is driven onto `busa` when `drive_busa` is `1`. If `drive_busa` is unknown, then an unknown value is driven onto `busa`. Otherwise, `busa` is not driven.

The conditional operator can be used with nonintegral types (see 6.11.1) and aggregate expressions (see 11.2.2) using the following rules:

— If both the first *expression* and second *expression* are of integral types, the operation proceeds as defined.

— If both *expressions* are **`real`**, then the resulting type is **`real`**. If one *expression* is **`real`** and the other *expression* is **`shortreal`** or integral, the other *expression* is cast to **`real`**, and the resulting type is **`real`**. If one *expression* is **`shortreal`** and the other *expression* is integral, the integral *expression* is cast to **`shortreal`**, and the resulting type is **`shortreal`**.

— Otherwise, if the first *expression* or second *expression* is an integral type and the opposing expression can be implicitly cast to an integral type, the cast is made and proceeds as defined.

— If the first *expression* or second *expression* is a class or interface class data type, the condition expression is legal in the following cases:

a) If both first *expression* and second *expression* are the literal value **`null`**, the result of the entire conditional expression is as if the expression were the literal **`null`**.

b) Else, if either first *expression* or second *expression* is the literal **`null`**, the resulting type is the type of the non-null expression.

c) Else, if the first *expression* is assignment compatible with the second *expression*, the resulting type is the type of the second *expression*,

d) Else, if the second *expression* is assignment compatible with the first *expression*, the resulting type is the type of the first *expression*,

e) Else, if the first *expression* and second *expression* are of a class type deriving from a common base class type, the resulting type is the closest common inherited class type.

In the preceding cases, the resulting value is the value of the first *expression* if the condition evaluates to true or the value of the second *expression* if the condition evaluates to false.

— For all other cases, the type of the first *expression* and second *expression* shall be equivalent (see 6.22.2).

For nonintegral and aggregate expressions, if *cond_predicate* evaluates to an ambiguous value and the expressions are not logically equivalent, then:

— For aggregate array data types, except associative arrays, where both expressions contain the same number of elements their results shall be combined element by element. If the elements match, the element shall be returned. If they do not match, then the value specified in Table 7-1 for that element's type shall be returned.

— For all other data types, the value specified in Table 7-1 for the resulting type as defined previously shall be returned.

#### 11.4.12 Concatenation operators

A concatenation is the result of the joining together of bits resulting from one or more expressions. The concatenation shall be expressed using the brace characters `{` and `}`, with commas separating the expressions within.

Unsized constant numbers shall not be allowed in concatenations. This is because the size of each operand in the concatenation is needed to calculate the complete size of the concatenation.

The following example concatenates four expressions:

```
{a, b[3:0], w, 3'b101}
```

The preceding example is equivalent to the following example:

```
{a, b[3], b[2], b[1], b[0], w, 1'b1, 1'b0, 1'b1}
```

The concatenation is treated as a packed vector of bits. It can be used on the left-hand side of an assignment or in an expression.

```
logic log1, log2, log3;
{log1, log2, log3} = 3'b111;
{log1, log2, log3} = {1'b1, 1'b1, 1'b1}; // same effect as 3'b111
```

One or more bits of a concatenation can be selected as if the concatenation were a packed array with the range `[n-1:0]`. Such a select shall not be legal as a *net_lvalue*, *variable_lvalue*, or in any equivalent use, such as on the left-hand side of an assignment. For example:

```
byte a, b ;
bit [1:0] c ;
c = {a + b}[1:0]; // 2 lsb's of sum of a and b
```

A concatenation is not the same as a structure literal (see 5.10) or an array literal (see 5.11). Concatenations are enclosed in just braces ( `{ }` ), whereas structure and array literals are enclosed in braces that begin with an apostrophe ( `'{ }` ).

### 11.4.12.1 Replication operator

A *replication* operator (also called a *multiple concatenation*) is expressed by a concatenation preceded by a non-negative, non-x, and non-z constant expression, called a *replication constant*, enclosed together within brace characters. A replication indicates a joining together of that many copies of the concatenation. Unlike regular concatenations, expressions containing replications shall not appear on the left-hand side of an assignment and shall not be connected to **output** or **inout** ports.

This example replicates w four times.

```
{4{w}}   // yields the same value as {w, w, w, w}
```

The following examples show illegal replications:

```
{1'bz{1'b0}}   // illegal
{1'bx{1'b0}}   // illegal
```

The next example illustrates a replication nested within a concatenation:

```
{b, {3{a, b}}}    // yields the same value as {b, a, b, a, b, a, b}
```

A replication operation may have a replication constant with a value of zero. This is useful in parameterized code. A replication with a zero replication constant is considered to have a size of zero and is ignored. Such a replication shall appear only within a concatenation in which at least one of the operands of the concatenation has a positive size.

For example:

```
parameter P = 32;

// The following is legal for all P from 1 to 32

assign b[31:0] = { {32-P{1'b1}}, a[P-1:0] } ;

// The following is illegal for P=32 because the zero
// replication appears alone within a concatenation

assign c[31:0] = { {{32-P{1'b1}}}, a[P-1:0] }

// The following is illegal for P=32

initial
  $displayb({32-P{1'b1}}, a[P-1:0]);
```

When a replication expression is evaluated, the operands shall be evaluated exactly once, even if the replication constant is zero. For example:

```
result = {4{func(w)}} ;
```

would be computed as

```
y = func(w) ;
result = {y, y, y, y} ;
```

#### 11.4.12.2 String concatenation

A concatenation of data objects of type **string** is allowed. In general, if any of the operands is of the data type **string**, the concatenation is treated as a string, and all other arguments are implicitly converted to the **string** data type (as described in 6.16). String concatenation is not allowed on the left-hand side of an assignment, only as an expression.

```
string hello = "hello";
string s;
s = { hello, " ", "world" };
$display( "%s\n", s );        // displays 'hello world'
s = { s, " and goodbye" };
$display( "%s\n", s );        // displays 'hello world and goodbye'
```

The replication operator form of braces can also be used with data objects of type **string**. In the case of string replication, a non-constant multiplier is allowed.

```
int n = 3;
string s = {n { "boo " }};
$display( "%s\n", s );  // displays 'boo boo boo '
```

Unlike bit concatenation, the result of a string concatenation or replication is not truncated. Instead, the destination variable (of type **string**) is resized to accommodate the resulting string.

### 11.4.13 Set membership operator

SystemVerilog supports singular value sets and set membership operators.

The syntax for the set membership operator is as follows:

inside_expression ::= expression **inside {** open_range_list **}** *// from A.8.3*

*Syntax 11-3—Inside expression syntax (excerpt from Annex A)*

The *expression* on the left-hand side of the **inside** operator is any singular expression.

The set-membership open_range_list on the right-hand side of the inside operator is a comma-separated list of expressions or ranges. If an expression in the list is an unpacked array, its elements are traversed by descending into the array until reaching a singular value. The members of the set are scanned until a match is found and the operation returns `1'b1`. Values can be repeated; therefore, values and value ranges can overlap. The order of evaluation of the expressions and ranges is nondeterministic.

```
int a, b, c;
if ( a inside {b, c} ) ...
int array [$] = '{3,4,5};
if ( ex inside {1, 2, array} ) ... // same as { 1, 2, 3, 4, 5}
```

The **inside** operator uses the equality ( == ) operator on nonintegral expressions to perform the comparison. If no match is found, the **inside** operator returns `1'b0`. Integral expressions use the wildcard equality (==?) operator so that an `x` or `z` bit in a value in the set is treated as a do-not-care in that bit position (see 11.4.6). As with wildcard equality, an `x` or `z` in the expression on the left-hand side of the inside operator is not treated as a do-not-care.

```
logic [2:0] val;
```

```
    while ( val inside {3'b1?1} ) ... // matches 3'b101, 3'b111, 3'b1x1, 3'b1z1
```

If no match is found, but some of the comparisons result in x, the inside operator shall return `1'bx`. The return value is effectively the or reduction of all the comparisons in the set with the expression on the left-hand side.

```
    wire r;
    assign r=3'bz11 inside {3'b1?1, 3'b011};  // r = 1'bx
```

A range can be specified with a low and high bound enclosed by square braces `[ ]` and separated by a colon ( **:** ), as in `[low_bound:high_bound]`. A bound specified by `$` shall represent the lowest or highest value for the type of the expression on the left-hand side. A match is found if the expression on the left-hand side is inclusively within the range. When specifying a range, the expressions shall be of a singular type for which the relation operators ( <=, >= ) are defined. If the bound to the left of the colon is greater than the bound to the right, the range is empty and contains no values.

For example:

```
    bit ba = a inside { [16:23], [32:47] };
    string I;
    if (I inside {["a rock":"hard place"]}) ...
       // I between "a rock" and a "hard place"
```

### 11.4.14 Streaming operators (pack/unpack)

The bit-stream casting described in 6.24.3 is most useful when the conversion operation can be easily expressed using only a type cast and the specific ordering of the bit stream is not important. Sometimes, however, a stream that matches a particular machine organization is required. The streaming operators perform packing of bit-stream types (see 6.24.3) into a sequence of bits in a user-specified order. When used in the left-hand side, the streaming operators perform the reverse operation, i.e., unpack a stream of bits into one or more variables.

If the data being packed contains any 4-state types, the result of a pack operation is a 4-state stream; otherwise, the result of a pack is a 2-state stream. In the remainder of this subclause, the word *bit*, without other qualification, denotes either a 2-state or a 4-state bit as required by this paragraph.

The syntax of the bit-stream concatenation is as follows:

---

streaming_concatenation ::= **{** stream_operator [ slice_size ] stream_concatenation **}** *// from A.8.1*

stream_operator ::= **>>** | **<<**

slice_size ::= simple_type | constant_expression

stream_concatenation ::= **{** stream_expression { **,** stream_expression } **}**

stream_expression ::= expression [ **with** **[** array_range_expression **]** ]

array_range_expression ::=
  expression
 | expression **:** expression
 | expression **+:** expression
 | expression **-:** expression

primary ::= *// from A.8.4*
  ...
 | streaming_concatenation

---

*Syntax 11-4—Streaming concatenation syntax (excerpt from Annex A)*

A *streaming_concatenation* (as specified in Syntax 11-4) shall be used either as the target of an assignment, or as the source of an assignment, or as the operand of a bit-stream cast, or as a *stream_expression* in another *streaming_concatenation*. Use of *streaming_concatenation* as the target of an assignment, and the associated unpack operation, is described in 11.4.14.3.

It shall be an error to use a *streaming_concatenation* as an operand in an expression without first casting it to a bit-stream type. When a *streaming_concatenation* is used as the source of an assignment, the target of that assignment shall be either a data object of bit-stream type or a *streaming_concatenation*.

If the target is a data object of bit-stream type, the stream created by the source *streaming_concatenation* shall first be widened if necessary to left-align it in the target, as follows:

— If the target represents a fixed-size variable that is narrower (has fewer bits) than the stream, an error shall be generated.

— If the target represents a fixed-size variable that is wider than the stream, the stream shall be widened to match it by filling with zero bits on the right.

— If the target represents a dynamically sized variable, such as a queue or dynamic array, the variable shall first be resized so that it has the smallest number of elements that make it as wide as or wider than the stream. If the resized variable is wider than the stream, the stream shall then be widened to match it by filling with zero bits on the right.

The stream, widened if necessary as described previously, shall then be implicitly cast to the type of the target.

The pack operation performed by a *streaming_concatenation* is described in two steps for convenience, but the intermediate result between the two steps is never visible and therefore tools are free to implement it in any way that yields the same overall result. First, all integral data in the *stream_expressions* are concatenated into a single stream of bits, similarly to bit-stream casting (as described in 6.24.3) but with fewer restrictions. Second, the resulting stream may be re-ordered in a manner specified by the *stream_operator* and *slice_size*. These two steps are described in more detail in 11.4.14.1 and 11.4.14.2.

#### 11.4.14.1 Concatenation of stream_expressions

Each *stream_expression* within the *stream_concatenation*, starting with the leftmost and proceeding from left to right through the comma-separated list of *stream_expressions*, is converted to a bit-stream and appended to a packed array (*stream*) of bits, the *generic stream*, by recursively applying the following procedure:

if the expression is a *streaming_concatenation* or it is of any bit-stream type,

  it shall be cast to a packed array of bit using a bit-stream cast, including casting 2-state to 4-state if necessary, and that packed array shall then be appended to the right-hand end of the generic stream;

else if the expression is an unpacked array (i.e., a queue, dynamic array, associative array, or fixed-size unpacked array)

  this procedure shall be applied in turn to each element of the array. An associative array is processed in index-sorted order. Other unpacked arrays are processed in the order in which they would be traversed by a **`foreach`** loop (see 12.7.3) having exactly one index variable;

else if the expression is of a struct type

  this procedure shall be applied in turn to each element of the struct, in declaration order;

else if the expression is of an untagged union type

  this procedure shall be applied to the first-declared member of the union;

else if the expression is a null class handle

> the expression shall be skipped (not streamed), and a warning may be issued;

else if the expression is a non-null class handle

> this procedure shall be applied in turn to each data member of the referenced object, and not the handle itself. Class members shall be streamed in declaration order. Extended class members shall be streamed after members of their base class. The result of streaming an object hierarchy that contains cycles shall be undefined, and an error may be issued. It shall be illegal to stream a class handle with local or protected members if those members would not be accessible at the point of the streaming operator;

else

> the expression shall be skipped (not streamed), and an error shall be issued.

In the preceding description, the phrase *skipped* (not *streamed*) means that the expression in question is not appended to the stream, and operation of the procedure then proceeds with the next item in turn. Implementations are not required to continue the procedure after issuing an error.

### 11.4.14.2 Re-ordering of the generic stream

The stream resulting from the operation described in 11.4.14.1 is then re-ordered by slicing it into blocks and then re-ordering those blocks.

The *slice_size* determines the size of each block, measured in bits. If a *slice_size* is not specified, the default is 1. If specified, it may be a constant integral expression or a simple type. If a type is used, the block size shall be the number of bits in that type. If a constant integral expression is used, it shall be an error for the value of the expression to be zero or negative.

The *stream_operator* << or >> determines the order in which blocks of data are streamed: >> causes blocks of data to be streamed in left-to-right order, while << causes blocks of data to be streamed in right-to-left order. Left-to-right streaming using >> shall cause the *slice_size* to be ignored and no re-ordering performed. Right-to-left streaming using << shall reverse the order of blocks in the stream, preserving the order of bits within each block. For right-to-left streaming using <<, the stream is sliced into blocks with the specified number of bits, starting with the right-most bit. If as a result of slicing the last (left-most) block has fewer bits than the block size, the last block has the size of the remaining bits; there is no padding or truncation.

For example:

```
int j = { "A", "B", "C", "D" };
{ >> {j}}              // generates stream "A" "B" "C" "D"
{ << byte {j}}         // generates stream "D" "C" "B" "A" (little endian)
{ << 16 {j}}           // generates stream "C" "D" "A" "B"
{ << { 8'b0011_0101 }}  // generates stream 'b1010_1100 (bit reverse)
{ << 4 { 6'b11_0101 }}  // generates stream 'b0101_11
{ >> 4 { 6'b11_0101 }}  // generates stream 'b1101_01 (same)
{ << 2 { { << { 4'b1101 }} }} // generates stream 'b1110
```

### 11.4.14.3 Streaming concatenation as an assignment target (unpack)

When a *streaming_concatenation* appears as the target of an assignment, the streaming operators perform the reverse operation; i.e., to unpack a stream of bits into one or more variables. The source expression shall be of bit-stream type, or the result of another *streaming_concatenation*. If the source expression contains more bits than are needed, the appropriate number of bits shall be consumed from its left (most significant) end. However, if more bits are needed than are provided by the source expression, an error shall be generated.

Unpacking a 4-state stream into a 2-state target is done by casting to a 2-state type, and vice versa. Null handles are skipped by both the pack and unpack operations; therefore, the unpack operation shall not create class objects. If a particular object hierarchy is to be reconstructed from a stream, the object hierarchy into which the stream is to be unpacked must be created before the streaming operator is applied. The unpack operation shall only modify explicitly declared properties; it will not modify implicitly declared properties such as random modes (see Clause 18).

For example:

```
int a, b, c;
logic [10:0] up [3:0];
logic [11:1] p1, p2, p3, p4;
bit [96:1] y = {>>{ a, b, c }};       // OK: pack a, b, c
int j = {>>{ a, b, c }};              // error: j is 32 bits < 96 bits
bit [99:0] d = {>>{ a, b, c }};       // OK: d is padded with 4 bits
{>>{ a, b, c }} = 23'b1;              // error: too few bits in stream
{>>{ a, b, c }} = 96'b1;              // OK: unpack a = 0, b = 0, c = 1
{>>{ a, b, c }} = 100'b11111;         // OK: unpack a = 0, b = 0, c = 1
                                      // 96 MSBs unpacked, 4 LSBs truncated
{ >> {p1, p2, p3, p4}} = up;          // OK: unpack p1 = up[3], p2 = up[2],
                                      // p3 = up[1], p4 = up[0]
```

#### 11.4.14.4 Streaming dynamically sized data

If the unpack operation includes unbounded dynamically sized types, the process is greedy (as in a cast): the first dynamically sized item is resized to accept all the available data (excluding subsequent fixed-size items) in the stream; any remaining dynamically sized items are left empty. This mechanism is sufficient to unpack a packet-sized stream that contains only one dynamically sized data item. However, when the stream contains multiple variable-size data packets, or each data packet contains more than one variable-size data item, or the size of the data to be unpacked is stored in the middle of the stream, this mechanism can become cumbersome and error-prone. To overcome these problems, the unpack operation allows a **with** expression to explicitly specify the extent of a variable-size field within the unpack operation.

The syntax of the **with** expression is as follows:

stream_expression ::= expression [ **with** **[** array_range_expression **]** ] *// from A.8.1*

array_range_expression ::=
  expression
| expression **:** expression
| expression **+:** expression
| expression **-:** expression

*Syntax 11-5—With expression syntax (excerpt from Annex A)*

The array range expression within the **with** construct shall be of integral type and evaluate to values that lie within the bounds of a fixed-size array or to positive values for dynamic arrays or queues. The expression before the **with** can be any one-dimensional unpacked array (including a queue). The expression within the **with** is evaluated immediately before its corresponding array is streamed (i.e., packed or unpacked). Thus, the expression can refer to data that are unpacked by the same operator but before the array. If the expression refers to variables that are unpacked after the corresponding array (to the right of the array), then the expression is evaluated using the previous values of the variables.

When used within the context of an unpack operation and the array is a variable-size array, it shall be resized to accommodate the range expression. If the array is a fixed-size array and the range expression evaluates to a range outside the extent of the array, only the range that lies within the array is unpacked and an error is generated. If the range expression evaluates to a range smaller than the extent of the array (fixed or variable size), only the specified items are unpacked into the designated array locations; the remainder of the array is unmodified.

When used within the context of a pack (on the right-hand side), it behaves the same as an array slice. The specified number of array items are packed into the stream. If the range expression evaluates to a range smaller than the extent of the array, only the specified array items are streamed. If the range expression evaluates to a range greater than the extent of the array size, the entire array is streamed, and the remaining items are generated using the nonexistent array entry value (as described in Table 7-1 in 7.4.6) for the given array.

For example, the following code uses streaming operators to model a packet transfer over a byte stream that uses little-endian encoding:

```
byte stream[$];   // byte stream

class Packet;
   rand int header;
   rand int len;
   rand byte payload[];
   int crc;

   constraint G { len > 1; payload.size == len ; }

   function void post_randomize; crc = payload.sum; endfunction
endclass

...
send: begin                    // Create random packet and transmit
   byte q[$];
   Packet p = new;
   void'(p.randomize());
   q = {<< byte{p.header, p.len, p.payload, p.crc}};  // pack
   stream = {stream, q};                              // append to stream
end

...
receive: begin       // Receive packet, unpack, and remove
   byte q[$];
   Packet p = new;
   {<< byte{ p.header, p.len, p.payload with [0 +: p.len], p.crc }} = stream;
   stream = stream[ $bits(p) / 8 : $ ];   // remove packet
end
```

In the preceding example, the pack operation could have been written as either:

```
q = {<<byte{p.header, p.len, p.payload with [0 +: p.len], p.crc}};
```

or

```
q = {<<byte{p.header, p.len, p.payload with [0 : p.len-1], p.crc}};
```

or

```
q = {<<byte{p}};
```

The result in this case would be the same because `p.len` is the size of `p.payload` as specified by the constraint.

## 11.5 Operands

There are several types of operands that can be specified in expressions. The simplest type is a reference to a net, variable, or parameter in its complete form; that is, just the name of the net, variable, or parameter is given. In this case, all of the bits making up the net, variable, or parameter value shall be used as the operand.

If a single bit of a vector net, vector variable, packed array, packed structure or parameter is required, then a bit-select operand shall be used. A part-select operand shall be used to reference a group of adjacent bits in a vector net, vector variable, packed array, packed structure, or parameter.

An unpacked array element can be referenced as an operand.

A concatenation of other operands (including nested concatenations) can be specified as an operand.

A function call is an operand.

Each of the types of operands mentioned previously is an example of a *simple operand*. An operand is simple if it is not parenthesized and is a *primary* as defined in A.8.4. In the following example, the expressions `1'b1 - 2'b00` and `(1'b1 + 1'b1)` are operands, but are not simple operands.

```
1'b1 - 2'b00 + (1'b1 + 1'b1)
```

### 11.5.1 Vector bit-select and part-select addressing

*Bit-selects* extract a particular bit from a vector, packed array, packed structure, parameter, or concatenation. The bit can be addressed using an expression that shall be evaluated in a self-determined context. If the bit-select address is invalid (it is out of bounds or has one or more `x` or `z` bits), then the value returned by the reference shall be `x` for 4-state and 0 for 2-state values. A bit-select or part-select of a scalar, or of a real variable or real parameter, shall be illegal.

Several contiguous bits can be addressed and are known as *part-selects*. There are two types of part-selects: a *non-indexed part-select* and an *indexed part-select*. A *non-indexed part-select* is given with the following syntax:

```
vect[msb_expr:lsb_expr]
```

Both *msb_expr* and *lsb_expr* shall be constant integer expressions. Each of these expressions shall be evaluated in a self-determined context. The first expression shall address a more significant bit than the second expression.

An *indexed part-select* is given with the following syntax:

```
logic [15:0] down_vect;
logic [0:15] up_vect;

down_vect[lsb_base_expr +: width_expr]
up_vect[msb_base_expr +: width_expr]

down_vect[msb_base_expr -: width_expr]
up_vect[lsb_base_expr -: width_expr]
```

The *msb_base_expr* and *lsb_base_expr* shall be integer expressions, and the *width_expr* shall be a positive constant integer expression. Each of these expressions shall be evaluated in a self-determined context. The *lsb_base_expr* and *msb_base_expr* can vary at run time. The first two examples select bits starting at the base and ascending the bit range. The number of bits selected is equal to the width expression. The second two examples select bits starting at the base and descending the bit range.

A *constant bit-select* is a bit-select whose position is constant. A *constant part-select* is a part-select whose position and width are both constant. The width of a part-select is always constant. Thus, a non-indexed part-select is always a constant part-select, and an indexed part-select is a constant part-select if its base is a constant value as well as its width.

A part-select that addresses a range of bits that are completely out of the address bounds of the vector, packed array, packed structure, parameter or concatenation, or a part-select that is `x` or `z` shall yield the value `x` when read and shall have no effect on the data stored when written. Part-selects that are partially out of range shall, when read, return `x` for the bits that are out of range and shall, when written, only affect the bits that are in range.

For example:

```
logic [31: 0] a_vect;
logic [0 :31] b_vect;
logic [63: 0] dword;
integer sel;

a_vect[ 0 +: 8]         // == a_vect[ 7 : 0]
a_vect[15 -: 8]         // == a_vect[15 : 8]

b_vect[ 0 +: 8]         // == b_vect[0 : 7]
b_vect[15 -: 8]         // == b_vect[8 :15]

dword[8*sel +: 8]       // variable part-select with fixed width
```

The following example specifies the single bit of vector `acc` that is addressed by the operand `index`:

```
acc[index]
```

The actual bit that is accessed by an address is, in part, determined by the declaration of `acc`. For instance, each of the declarations of `acc` shown in the next example causes a particular value of `index` to access a different bit:

```
logic [15:0] acc;
logic [2:17] acc;
```

The next example and the bullet items that follow it illustrate the principles of bit addressing. The code declares an 8-bit variable called `vect` and initializes it to a value of 4. The list describes how the separate bits of that vector can be addressed.

```
logic [7:0] vect;
vect = 4;   // fills vect with the pattern 00000100
            // msb is bit 7, lsb is bit 0
```

— If the value of `addr` is 2, then `vect[addr]` returns `1`.
— If the value of `addr` is out of bounds, then `vect[addr]` returns `x`.
— If `addr` is 0, 1, or 3 through 7, `vect[addr]` returns `0`.
— `vect[3:0]` returns the bits `0100`.
— `vect[5:1]` returns the bits `00010`.
— `vect[`*expression that returns x*`]` returns `x`.

— `vect[`*expression that returns* `z]` returns `x`.

— If any bit of `addr` is `x` or `z`, then the value of `addr` is `x`.

NOTE 1—Part-select indices that evaluate to `x` or `z` may be flagged as a compile time error.

NOTE 2—Bit-select or part-select indices that are outside the declared range may be flagged as a compile time error.

### 11.5.2 Array and memory addressing

Declaration of arrays and memories (one-dimensional arrays of **`reg`**, **`logic`**, or **`bit`**) are discussed in 7.4. This subclause discusses array addressing.

The following example declares a memory of 1024 8-bit words:

```
logic [7:0] mem_name[0:1023];
```

The syntax for a memory address shall consist of the name of the memory and an expression for the address, specified with the following format:

```
mem_name[addr_expr]
```

The *addr_expr* can be any integer expression; therefore, memory indirections can be specified in a single expression. The next example illustrates memory indirection:

```
mem_name[mem_name[3]]
```

In this example, `mem_name[3]` addresses word three of the memory called `mem_name`. The value at word three is the index into `mem_name` that is used by the memory address `mem_name[mem_name[3]]`. As with bit-selects, the address bounds given in the declaration of the memory determine the effect of the address expression. If the address is invalid (it is out of bounds or has one or more `x` or `z` bits), then the value of the reference shall be as described in 7.4.6.

The next example declares an array of 256-by-256 8-bit elements and an array 256-by-256-by-8 1-bit elements:

```
logic [7:0] twod_array[0:255][0:255];
wire threed_array[0:255][0:255][0:7];
```

The syntax for access to the array shall consist of the name of the memory or array and an integer expression for each addressed dimension:

```
twod_array[addr_expr][addr_expr]
threed_array[addr_expr][addr_expr][addr_expr]
```

As before, the `addr_expr` can be any integer expression. The array `twod_array` accesses a whole 8-bit vector, while the array `threed_array` accesses a single bit of the three-dimensional array.

To express bit-selects or part-selects of array elements, the desired word shall first be selected by supplying an address for each dimension. Once selected, bit-selects and part-selects shall be addressed in the same manner as net and variable bit-selects and part-selects (see 11.5.1).

For example:

```
twod_array[14][1][3:0]      // access lower 4 bits of word
twod_array[1][3][6]         // access bit 6 of word
twod_array[1][3][sel]       // use variable bit-select
threed_array[14][1][3:0]    // Illegal
```

### 11.5.3 Longest static prefix

Informally, the *longest static prefix* of a select is the longest part of the select for which an analysis tool has known values following elaboration. This concept is used when describing implicit sensitivity lists (see 9.2.2.2) and when describing error conditions for drivers of logic ports (see 6.5). The remainder of this subclause defines what constitutes the "longest static prefix" of a select.

A field select is defined as a hierarchical name where the right-hand side of the last "." hierarchy separator identifies a field of a variable whose type is a **struct** or **union** declaration. The field select prefix is defined to be the left-hand side of the final "." hierarchy separator in a field select.

An indexing select is a single indexing operation. The indexing select prefix is either an identifier or, in the case of a multidimensional select, another indexing select. Array selects, bit-selects, part-selects, and indexed part-selects are examples of indexing selects.

The definition of a static prefix is recursive and is defined as follows:

— An identifier is a static prefix.
— A hierarchical reference to an object is a static prefix.
— A package reference to net or variable is a static prefix.
— A field select is a static prefix if the field select prefix is a static prefix.
— An indexing select is a static prefix if the indexing select prefix is a static prefix and the select expression is a constant expression.

The definition of the longest static prefix is defined as follows:

— An identifier that is not the field select prefix or indexing select prefix of an expression that is a static prefix.
— A field select that is not the field select prefix or indexing select prefix of an expression that is a static prefix.
— An indexing select that is not the field select prefix or indexing select prefix of an expression that is a static prefix.

*Examples:*

```
localparam p = 7;
reg [7:0] m [5:1][5:1];
integer i;

m[1][i]     // longest static prefix is m[1]

m[p][1]     // longest static prefix is m[p][1]

m[i][1]     // longest static prefix is m
```

## 11.6 Expression bit lengths

The number of bits of an expression is determined by the operands and the context. Casting can be used to set the size context of an intermediate value (see 6.24).

Controlling the number of bits that are used in expression evaluations is important if consistent results are to be achieved. Some situations have a simple solution; for example, if a bitwise AND operation is specified on two 16-bit variables, then the result is a 16-bit value. However, in some situations, it is not obvious how many bits are used to evaluate an expression or what size the result should be.

For example, should an arithmetic add of two 16-bit values perform the evaluation using 16 bits, or should the evaluation use 17 bits in order to allow for a possible carry overflow? The answer depends on the type of device being modeled and whether that device handles carry overflow.

SystemVerilog uses the bit length of the operands to determine how many bits to use while evaluating an expression. The bit length rules are given in 11.6.1. In the case of the addition operator, the bit length of the largest operand, including the left-hand side of an assignment, shall be used.

For example:

```
logic [15:0] a, b;    // 16-bit variables
logic [15:0] sumA;    // 16-bit variable
logic [16:0] sumB;    // 17-bit variable

sumA = a + b;         // expression evaluates using 16 bits
sumB = a + b;         // expression evaluates using 17 bits
```

### 11.6.1 Rules for expression bit lengths

The rules governing the expression bit lengths have been formulated so that most practical situations have a natural solution.

The number of bits of an expression (known as the *size* of the expression) shall be determined by the operands involved in the expression and the context in which the expression is given.

A *self-determined expression* is one where the bit length of the expression is solely determined by the expression itself—for example, an expression representing a delay value.

A *context-determined expression* is one where the bit length of the expression is determined by the bit length of the expression and by the fact that it is part of another expression. For example, the bit size of the right-hand expression of an assignment depends on itself and the size of the left-hand side.

Table 11-21 shows how the form of an expression shall determine the bit lengths of the results of the expression. In Table 11-21, `i`, `j`, and `k` represent expressions of an operand, and `L(i)` represents the bit length of the operand represented by `i`.

**Table 11-21—Bit lengths resulting from self-determined expressions**

| Expression | Bit length | Comments |
|---|---|---|
| Unsized constant number | Same as integer | |
| Sized constant number | As given | |
| i op j, where op is:<br>`+ - * / % & \| ^ ^~ ~^` | max(L(i),L(j)) | |
| op i, where op is:<br>`+ - ~` | L(i) | |
| i op j, where op is:<br>`=== !== == != > >= < <=` | 1 bit | Operands are sized to max(L(i),L(j)) |
| i op j, where op is:<br>`&& \|\| -> <->` | 1 bit | All operands are self-determined |
| op i, where op is:<br>`& ~& \| ~\| ^ ~^ ^~ !` | 1 bit | All operands are self-determined |

**Table 11-21—Bit lengths resulting from self-determined expressions *(continued)***

| Expression | Bit length | Comments |
|---|---|---|
| i op j, where op is:<br>`>>    <<    **    >>>    <<<` | L(i) | j is self-determined |
| i ? j : k | max(L(j),L(k)) | i is self-determined |
| `{i,...,j}` | L(i)+..+L(j) | All operands are self-determined |
| `{i{j,..,k}}` | i × (L(j)+..+L(k)) | All operands are self-determined |

Multiplication may be performed without losing any overflow bits by assigning the result to something wide enough to hold it.

### 11.6.2 Example of expression bit-length problem

During the evaluation of an expression, interim results shall take the size of the largest operand (in case of an assignment, this also includes the left-hand side). Care has to be taken to prevent loss of a significant bit during expression evaluation. The following example describes how the bit lengths of the operands could result in the loss of a significant bit.

Given the following declarations:

```
logic [15:0] a, b, answer; // 16-bit variables
```

the intent is to evaluate the expression

```
answer = (a + b) >> 1; // will not work properly
```

where `a` and `b` are to be added, which can result in an overflow, and then shifted right by 1 bit to preserve the carry bit in the 16-bit answer.

A problem arises, however, because all operands in the expression are of a 16-bit width. Therefore, the expression `(a + b)` produces an interim result that is only 16 bits wide, thus losing the carry bit before the evaluation performs the 1-bit right shift operation.

The solution is to force the expression `(a + b)` to evaluate using at least 17 bits. For example, adding an integer value of `0` to the expression will cause the evaluation to be performed using the bit size of integers. The following example will produce the intended result:

```
answer = (a + b + 0) >> 1; // will work correctly
```

In the following example:

```
module bitlength();
   logic [3:0] a, b, c;
   logic [4:0] d;

   initial begin
      a = 9;
      b = 8;
      c = 1;
      $display("answer = %b", c ? (a&b) : d);
   end
endmodule
```

the `$display` statement will display

```
answer = 01000
```

By itself, the expression `a&b` would have the bit length 4, but because it is in the context of the conditional expression, which uses the maximum bit length, the expression `a&b` actually has length 5, the length of `d`.

### 11.6.3 Example of self-determined expressions

```
logic [3:0] a;
logic [5:0] b;
logic [15:0] c;

initial begin
   a = 4'hF;
   b = 6'hA;
   $display("a*b=%h", a*b);   // expression size is self-determined
   c = {a**b};                // expression a**b is self-determined
                              // due to concatenation operator {}
   $display("a**b=%h", c);
   c = a**b;                  // expression size is determined by c
   $display("c=%h", c);
end
```

Simulator output for this example:

```
a*b=16   // 'h96 was truncated to 'h16 since expression size is 6
a**b=1   // expression size is  4 bits (size of a)
c=ac61   // expression size is 16 bits (size of c)
```

## 11.7 Signed expressions

Controlling the sign of an expression is important if consistent results are to be achieved. 11.8.1 outlines the rules that determine if an expression is signed or unsigned.

The cast operator can be used to change either the signedness or type of an expression (see 6.24.1). In addition to the cast operator, the **`$signed`** and **`$unsigned`** system functions are available for casting the signedness of expressions. These functions shall evaluate the input expression and return a one-dimensional packed array with the same number of bits and value of the input expression and the signedness defined by the function.

**`$signed`**—returned value is signed

**`$unsigned`**—returned value is unsigned

For example:

```
logic [7:0] regA, regB;
logic signed [7:0] regS;

regA = $unsigned(-4);         // regA = 8'b11111100
regB = $unsigned(-4'sd4);     // regB = 8'b00001100
regS = $signed  (4'b1100);    // regS = -4

regA = unsigned'(-4);         // regA = 8'b11111100
regS = signed'(4'b1100);      // regS = -4
regS = regA + regB;                    // will do unsigned addition
regS = byte'(regA) + byte'(regB);      // will do signed addition
regS = signed'(regA) + signed'(regB);  // will do signed addition
regS = $signed(regA) + $signed(regB);  // will do signed addition
```

## 11.8 Expression evaluation rules

### 11.8.1 Rules for expression types

The following are the rules for determining the resulting type of an expression:

— Expression type depends only on the operands. It does not depend on the left-hand side (if any).

— Decimal numbers are signed.

— Based numbers are unsigned, except where the `s` notation is used in the base specifier (as in `4'sd12`).

— Bit-select results are unsigned, regardless of the operands.

— Part-select results are unsigned, regardless of the operands even if the part-select specifies the entire vector.

```
logic [15:0] a;
logic signed [7:0] b;

initial
   a = b[7:0];    // b[7:0] is unsigned and therefore zero-extended
```

— Concatenate results are unsigned, regardless of the operands.

— Comparison and reduction operator results are unsigned, regardless of the operands.

— Reals converted to integers by type coercion are signed

— The sign and size of any self-determined operand are determined by the operand itself and independent of the remainder of the expression.

— For non-self-determined operands, the following rules apply:

- If any operand is real, the result is real.
- If any operand is unsigned, the result is unsigned, regardless of the operator.
- If all operands are signed, the result will be signed, regardless of operator, except when specified otherwise.

### 11.8.2 Steps for evaluating an expression

The following are the steps for evaluating an expression:

— Determine the expression size based upon the standard rules of expression size determination.

— Determine the sign of the expression using the rules outlined in 11.8.1.

— Propagate the type and size of the expression (or self-determined subexpression) back down to the context-determined operands of the expression. In general, any context-determined operand of an operator shall be the same type and size as the result of the operator. However, there are two exceptions:

- If the result type of the operator is real and if it has a context-determined operand that is not real, that operand shall be treated as if it were self-determined and then converted to real just before the operator is applied.
- The relational and equality operators have operands that are neither fully self-determined nor fully context-determined. The operands shall affect each other as if they were context-determined

operands with a result type and size (maximum of the two operand sizes) determined from them. However, the actual result type shall always be 1 bit unsigned. The type and size of the operand shall be independent of the rest of the expression and vice versa.

— When propagation reaches a simple operand as defined in 11.5, then that operand shall be converted to the propagated type and size. If the operand shall be extended, then it shall be sign-extended only if the propagated type is signed.

### 11.8.3 Steps for evaluating an assignment

The following are the steps for evaluating an assignment:

— Determine the size of the right-hand side by the standard assignment size determination rules (see 11.6).

— If needed, extend the size of the right-hand side, performing sign extension if, and only if, the type of the right-hand side is signed.

### 11.8.4 Handling X and Z in signed expressions

If a signed operand is to be resized to a larger signed width and the value of the sign bit is `x`, the resulting value shall be bit-filled with `x`. If the sign bit of the value is `z`, then the resulting value shall be bit-filled with `z`. If any bit of a signed value is `x` or `z`, then any nonlogical operation involving the value shall result in the entire resultant value being an `x` and the type consistent with the expression's type.

## 11.9 Tagged union expressions and member access

```
expression ::=                                                   // from A.8.3
    ...
  | tagged_union_expression
tagged_union_expression ::=
    tagged member_identifier [ expression ]
```

*Syntax 11-6—Tagged union syntax (excerpt from Annex A)*

A tagged union expression (packed or unpacked) is expressed using the keyword **tagged** followed by a tagged union member identifier, followed by an expression representing the corresponding member value. For **void** members the member value expression is omitted.

*Example:*

```
typedef union tagged {
   void Invalid;
   int Valid;
} VInt;

VInt vi1, vi2;

vi1 = tagged Valid (23+34);   // Create Valid int
vi2 = tagged Invalid;         // Create an Invalid value
```

In the following tagged union expressions, the expressions in braces are structure assignment patterns (see 10.9.2).

```
typedef union tagged {
   struct {
      bit [4:0] reg1, reg2, regd;
   } Add;
   union tagged {
      bit [9:0] JmpU;
      struct {
         bit [1:0] cc;
         bit [9:0] addr;
      } JmpC;
   } Jmp;
} Instr;

Instr i1, i2;

// Create an Add instruction with its 3 register fields
i1 = ( e
   ? tagged Add '{ e1, 4, ed };                 // struct members by position
   : tagged Add '{ reg2:e2, regd:3, reg1:19 }); // by name (order irrelevant)

// Create a Jump instruction, with "unconditional" sub-opcode
i1 = tagged Jmp (tagged JmpU 239);

// Create a Jump instruction, with "conditional" sub-opcode
i2 = tagged Jmp (tagged JmpC '{ 2, 83 });  // inner struct by position
i2 = tagged Jmp (tagged JmpC '{ cc:2, addr:83 });  // by name
```

The type of a tagged union expression shall be known from its context (e.g., it is used in the right-hand side of an assignment to a variable whose type is known, or it has a cast, or it is used inside another expression from which its type is known). The expression evaluates to a tagged union value of that type. The tagged union expression can be completely type-checked statically; the only member names allowed after the **tagged** keyword are the member names for the expression type, and the member expression shall have the corresponding member type.

An uninitialized variable of tagged union type shall be undefined. This includes the tag bits. A variable of tagged union type can be initialized with a tagged union expression provided the member value expression is a legal initializer for the member type.

Members of tagged unions can be read or assigned using the usual dot notation. Such accesses are completely type-checked, i.e., the value read or assigned shall be consistent with the current tag. In general, this can require a run-time check. An attempt to read or assign a value whose type is inconsistent with the tag results in a run-time error.

All of the following examples are legal only if the instruction variable `i1` currently has tag `Add`:

```
x = i1.Add.reg1;
i1.Add = '{19, 4, 3};
i1.Add.reg2 = 4;
```

## 11.10 String literal expressions

This subclause discusses operations on string literals (see 5.9) and string literals stored in bit vectors and other packed types. SystemVerilog also has string variables, which store strings differently than vectors. The **string** data type has several special built-in methods for manipulating strings. See 6.16 for a discussion of the **string** data type and associated methods.

String literal operands shall be treated as constant numbers consisting of a sequence of 8-bit ASCII codes, one per character. Any SystemVerilog operator can manipulate string literal operands. The operator shall behave as though the entire string were a single numeric value.

When a vector is larger than required to hold the string literal value being assigned, the contents after the assignment shall be padded on the left with zeros. This is consistent with the padding that occurs during assignment of nonstring unsigned values.

The following example declares a vector variable large enough to hold 14 characters and assigns a value to it. The example then manipulates the stored value using the concatenation operator.

```
module string_test;
   bit [8*14:1] stringvar;

   initial begin
      stringvar = "Hello world";
      $display("%s is stored as %h", stringvar, stringvar);
      stringvar = {stringvar,"!!!"};
      $display("%s is stored as %h", stringvar, stringvar);
   end
endmodule
```

The result of simulating the preceding description is as follows:

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

### 11.10.1 String literal operations

SystemVerilog operators support the common string operations *copy*, *concatenate*, and *compare* for string literals and string literals stored in vectors. Copy is provided by simple assignment. Concatenation is provided by the concatenation operator. Comparison is provided by the equality operators.

When manipulating string literal values in vectors, the vectors should be at least `8*n` bits (where `n` is the number of ASCII characters) in order to preserve the 8-bit ASCII code.

### 11.10.2 String literal value padding and potential problems

When string literals are assigned to vectors, the values stored shall be padded on the left with zeros. Padding can affect the results of comparison and concatenation operations. The comparison and concatenation operators shall not distinguish between zeros resulting from padding and the original string characters (`\0`, ASCII `NUL`).

The following example illustrates the potential problem:

```
bit [8*10:1] s1, s2;
initial begin
   s1 = "Hello";
   s2 = " world!";
   if ({s1,s2} == "Hello world!")
      $display("strings are equal");
end
```

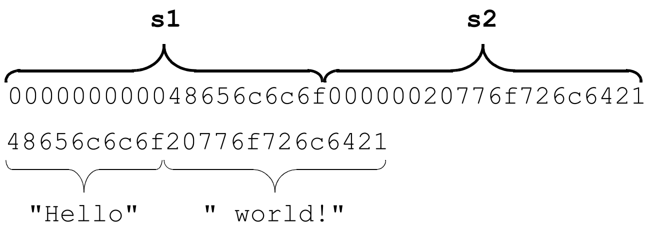
The comparison in this example fails because during the assignment the variables `s1` and `s2` are padded as illustrated in the next example:

```
s1 = 000000000048656c6c6f
s2 = 00000020776f726c6421
```

The concatenation of `s1` and `s2` includes the zero padding, resulting in the following value:

```
000000000048656c6c6f00000020776f726c6421
```

Because the string literal `"Hello world!"` contains no zero padding, the comparison fails, as shown in the following example:



This comparison yields a result of zero, which represents false.

### 11.10.3 Empty string literal handling

The empty string literal (`""`) shall be considered equivalent to the ASCII NUL (`"\0"`), which has a value zero (`0`), which is different from a string `"0"`.

## 11.11 Minimum, typical, and maximum delay expressions

SystemVerilog delay expressions can be specified as three expressions separated by colons and enclosed by parentheses. This is intended to represent minimum, typical, and maximum values—in that order. The syntax is given in Syntax 11-7.

```
mintypmax_expression ::=                                                    // from A.8.3
      expression
    | expression : expression : expression
constant_mintypmax_expression ::=
      constant_expression
    | constant_expression : constant_expression : constant_expression
expression ::=
      primary
    | unary_operator { attribute_instance } primary
    | inc_or_dec_expression
    | ( operator_assignment )
    | expression binary_operator { attribute_instance } expression
    | conditional_expression
    | inside_expression
    | tagged_union_expression
constant_expression ::=
      constant_primary
    | unary_operator { attribute_instance } constant_primary
```

| constant_expression binary_operator { attribute_instance } constant_expression
| constant_expression **?** { attribute_instance } constant_expression **:** constant_expression

constant_primary ::= *// from A.8.4*
primary_literal
| ps_parameter_identifier constant_select
| specparam_identifier [ **[** constant_range_expression **]** ]
| genvar_identifier[39]
| formal_port_identifier constant_select
| [ package_scope | class_scope ] enum_identifier
| constant_concatenation [ **[** constant_range_expression **]** ]
| constant_multiple_concatenation [ **[** constant_range_expression **]** ]
| constant_function_call
| constant_let_expression
| **(** constant_mintypmax_expression **)**
| constant_cast
| constant_assignment_pattern_expression
| type_reference[40]
| **null**

primary_literal ::= number | time_literal | unbased_unsized_literal | string_literal

39) A genvar_identifier shall be legal in a constant_primary only within a genvar_expression.

40) It shall be legal to use a type_reference constant_primary as the casting_type in a static cast. It shall be illegal for a type_reference constant_primary to be used with any operators except the equality/inequality and case equality/inequality operators.

*Syntax 11-7—Syntax for min:typ:max expression (excerpt from Annex A)*

SystemVerilog models typically specify three values for delay expressions. The three values allow a design to be tested with minimum, typical, or maximum delay values, known as a *min:typ:max expression*.

Values expressed in min:typ:max format can be used in expressions. The min:typ:max format can be used wherever expressions can appear.

*Example 1*: This example shows an expression that defines a single triplet of delay values. The minimum value is the sum of `a+d`; the typical value is `b+e`; the maximum value is `c+f`, as follows:

```
(a:b:c) + (d:e:f)
```

*Example 2*: The next example shows a typical expression that is used to specify min:typ:max format values:

```
val - (32'd 50: 32'd 75: 32'd 100)
```

## 11.12 Let construct

let_declaration ::= *// from A.2.12*
**let** let_identifier [ **(** [ let_port_list ] **)** ] **=** expression **;**

let_identifier ::=
identifier

let_port_list ::=
let_port_item { **,** let_port_item}

let_port_item ::=
  { attribute_instance } let_formal_type formal_port_identifier { variable_dimension } [ **=** expression ]

let_formal_type ::=
  data_type_or_implicit
  | **untyped**

let_expression ::=
  [ package_scope ] let_identifier [ **(** [ let_list_of_arguments ] **)** ]

let_list_of_arguments ::=
  [ let_actual_arg ] { **,** [ let_actual_arg ] } { **,** **.** identifier **(** [ let_actual_arg ] **)** }
  | **.** identifier **(** [ let_actual_arg ] **)** { **,** **.** identifier **(** [ let_actual_arg ] **)** }

let_actual_arg ::=
  expression

*Syntax 11-8—Let syntax (excerpt from Annex A)*

A **let** declaration defines a template expression (a **let** body), customized by its ports. A **let** construct may be instantiated in other expressions.

**let** declarations can be used for customization and can replace the text macros in many cases. The **let** construct is safer because it has a local scope, while the scope of compiler directives is global within the compilation unit. Including **let** declarations in packages (see Clause 26) is a natural way to implement a well-structured customization for the design code.

*Example 1:*

```
package pex_gen9_common_expressions;
   let valid_arb(request, valid, override) = |(request & valid) || override;
   ...
endpackage

module my_checker;
   import pex_gen9_common_expressions::*;
   logic a, b;
   wire [1:0] req;
   wire [1:0] vld;
   logic ovr;
   ...
      if (valid_arb(.request(req), .valid(vld), .override(ovr))) begin
         ...
      end
   ...
endmodule
```

*Example 2:*

```
let mult(x, y) = ($bits(x) + $bits(y))'(x * y);
```

Just as properties and sequences serve as templates for concurrent assertions (see 16.5), the **let** construct can serve this purpose for immediate assertions. For example:

```
let at_least_two(sig, rst = 1'b0) = rst || ($countones(sig) >= 2);
logic [15:0] sig1;
logic [3:0] sig2;
```

```
always_comb begin
   q1: assert (at_least_two(sig1));
   q2: assert (at_least_two(~sig2));
end
```

Another intended use of **`let`** is to provide shortcuts for identifiers or subexpressions. For example:

```
task write_value;
   input logic [31:0] addr;
   input logic [31:0] value;
   ...
endtask
...
let addr = top.block1.unit1.base + top.block1.unit2.displ;
...
write_value(addr, 0);
```

The formal arguments may optionally be typed and also may have optional default values. If a formal argument of a **`let`** is typed, then the type shall be **`event`** or one of the types allowed in 16.6. The following rules apply to typed formal arguments and their corresponding actual arguments, including default actual arguments declared in a **`let`**:

1) If the formal argument is of type **`event`**, then the actual argument shall be an *event_expression* and each reference to the formal argument shall be in a place where an *event_expression* may be written.
2) Otherwise, the self-determined result type of the actual argument shall be cast compatible (see 6.22.4) with the type of the formal argument. The actual argument shall be cast to the type of the formal argument before being substituted for a reference to the formal argument in the rewriting algorithm (see F.4).

Variables used in a **`let`** that are not formal arguments to the **`let`** are resolved according to the scoping rules from the scope in which the **`let`** is declared. In the scope of declaration, a **`let`** body shall be defined before it is used. No hierarchical references to **`let`** declarations are allowed.

The **`let`** body gets expanded with the actual arguments by replacing the formal arguments with the actual arguments. Semantic checks are performed to verify that the expanded **`let`** body with the actual arguments is legal. The result of the substitution is enclosed in parentheses (...) so as to preserve the priority of evaluation of the **`let`** body. Recursive **`let`** instantiations are not permitted.

A **`let`** body may contain sampled value function calls (see 16.9.3 and 16.9.4). Their clock, if not explicitly specified, is inferred in the instantiation context in the same way as if the functions were used directly in the instantiation context. It shall be an error if the clock is required, but cannot be inferred in the instantiation context.

A **`let`** may be declared in any of the following:

— A module
— An interface
— A program
— A checker
— A clocking block
— A package
— A compilation-unit scope
— A generate block

— A sequential or parallel block
— A subroutine

*Examples:*

a) **let** with arguments and without arguments

```
module m;
   logic clk, a, b;
   logic p, q, r;

   // let with formal arguments and default value on y
   let eq(x, y = b) = x == y;

   // without parameters, binds to a, b above
   let tmp = a && b;
   ...
   a1: assert property (@(posedge clk) eq(p,q));
   always_comb begin
      a2: assert (eq(r)); // use default for y
      a3: assert (tmp);
   end
endmodule : m
```

The effective code after expanding **let** expressions:

```
module m;
   bit clk, a, b;
   logic p, q, r;
   // let eq(x, y = b) = x == y;
   // let tmp = a && b;
   ...
   a1: assert property (@(posedge clk) (m.p == m.q));
   always_comb begin
      a2: assert ((m.r == m.b)); // use default for y
      a3: assert ((m.a && m.b));
   end
endmodule : m
```

b) Declarative context binding of **let** arguments

```
module top;
   logic x = 1'b1;
   logic a, b;
   let y = x;
   ...
   always_comb begin
      // y binds to preceding definition of x
      // in the declarative context of let
      bit x = 1'b0;
      b = a | y;
   end
endmodule : top
```

The effective code after expanding **let** expressions:

```
module top;
   bit x = 1'b1;
   bit a;
   // let y = x;
```

```
      ...
      always_comb begin
         // y binds to preceding definition of x
         // in the declarative context of let
         bit x = 1'b0;
         b = a | (top.x);
      end
   endmodule : top
```

c) Sequences (and properties) with **let** in structural context (see 16.8)

```
   module top;
      logic a, b;
      let x = a || b;
      sequence s;
         x ##1 b;
      endsequence : s
      ...
   endmodule : top
```

The effective code after expanding **let** expressions:

```
   module top;
      logic a, b;
      // let x = a || b;
      sequence s;
         (top.a || top.b) ##1 b;
      endsequence : s
      ...
   endmodule : top
```

d) **let** declared in a **generate** block

```
   module m(...);
      wire a, b;
      wire [2:0] c;
      wire [2:0] d;
      wire e;
      ...
      for (genvar i = 0; i < 3; i++) begin : L0
         if (i !=1) begin : L1
            let my_let(x) = !x || b && c[i];
            s1: assign d[2 - i] = my_let(a)); // OK
         end : L1
      end : L0
      s2: assign e = L0[0].L1.my_let(a)); // Illegal
   endmodule : m
```

Statement `s1` becomes two statements `L0[0].L1.s1` and `L0[2].L1.s1`, the first of them being

```
assign d[2] = (!m.a || m.b && m.c[0]);
```

and the second one being

```
assign d[0] = (!m.a || m.b && m.c[2]);
```

Statement `s2` is illegal since it references the **let** expression hierarchically, while hierarchical references to **let** expressions are not allowed.

e) **let** with typed arguments

```
   module m(input clock);
   logic [15:0] a, b;
   logic c, d;
   typedef bit [15:0] bits;
   ...
   let ones_match(bits x, y) = x == y;
   let same(logic x, y) = x === y;

   always_comb
      a1: assert(ones_match(a, b));

   property toggles(bit x, y);
      same(x, y) |=> !same(x, y);
   endproperty

   a2: assert property (@(posedge clock) toggles(c, d));
endmodule : m
```

In this example the **let** expression `ones_match` checks that both arguments have bits set to 1 at the same position. Because of the explicit specification of the formal arguments to be of the 2-state type **bit** in the **let** declaration, all argument bits having unknown logic value or a high-impedance value become 0, and therefore the comparison captures the match of the bits set to 1. The **let** expression same tests for the case equality (see 11.4.6) of its operands. When instantiated in the property `toggles` its actual arguments will be of type **bit**. The effective code after expanding **let** expressions:

```
module m(input clock);
   logic [15:0] a, b;
   logic c, d;
   typedef bit [15:0] bits;
   ...
   // let ones_match(bits x, y) = x == y;
   // let same(logic x, y) = x === y;

   always_comb
      a1:assert((bits'(a) == bits'(b)));

   property toggles(bit x, y);
      (logic'(x) === logic'(y)) |=> ! (logic'(x) === logic'(y));
   endproperty

   a2: assert property (@(posedge clock) toggles(c, d));
endmodule : m
```

f) Sampled value functions in **let**

```
module m(input clock);
   logic a;
   let p1(x) = $past(x);
   let p2(x) = $past(x,,,@(posedge clock));
   let s(x) = $sampled(x);
   always_comb begin
      a1: assert(p1(a));
      a2: assert(p2(a));
      a3: assert(s(a));
   end
   a4: assert property(@(posedge clock) p1(a));
   ...
endmodule : m
```

The effective code after expanding **let** expressions:

```
 module m(input clock);
    logic a;
    // let p1(x) = $past(x);
    // let p2(x) = $past(x,,,@(posedge clock));
    // let s(x) = $sampled(x);
    always_comb begin
       a1: assert(($past(a))); // Illegal: no clock can be inferred
       a2: assert(($past(a,,,@(posedge clock))));
       a3: assert(($sampled (a)));
    end
    a4: assert property(@(posedge clock)($past(a)));    // @(posedge clock)
                                                         // is inferred
    ...

endmodule : m
```

# 12. Procedural programming statements

## 12.1 General

This clause describes the following:

- Selection statements (if–else, case, casez, casex, unique, unique0, priority)
- Loop statements (for, repeat, foreach, while, do...while, forever)
- Jump statements (break, continue, return)

## 12.2 Overview

Procedural programming statements shall be contained within any of the following constructs:

- Procedural blocks that automatically activate, introduced with one of the keywords:
  - **`initial`**
  - **`always`**
  - **`always_comb`**
  - **`always_latch`**
  - **`always_ff`**
  - **`final`**

  See Clause 9 for a description of each type of procedural block.
- Procedural blocks that activate when called, introduced with one of the keywords:
  - **`task`**
  - **`function`**

  See Clause 13 for a description of tasks and functions.

Procedural programming statements include the following:

- Selection statements (see 12.4 and 12.5)
- Loop statements (see 12.7)
- Jump statements (see 12.8)
- Sequential and parallel blocks (see 9.3)
- Timing controls (see 9.4)
- Process control (see 9.5 through 9.7)
- Procedural assignments (see 10.4 through 10.9)
- Subroutine calls (see Clause 13)

## 12.3 Syntax

The syntax for procedural statements is as follows in Syntax 12-1:

```
statement_or_null ::=                                                    // from A.6.4
      statement
    | { attribute_instance } ;
statement ::= [ block_identifier : ] { attribute_instance } statement_item
statement_item ::=
      blocking_assignment ;
```

| nonblocking_assignment **;**
| procedural_continuous_assignment **;**
| case_statement
| conditional_statement
| inc_or_dec_expression **;**
| subroutine_call_statement
| disable_statement
| event_trigger
| loop_statement
| jump_statement
| par_block
| procedural_timing_control_statement
| seq_block
| wait_statement
| procedural_assertion_statement
| clocking_drive **;**
| randsequence_statement
| randcase_statement
| expect_property_statement

*Syntax 12-1—Procedural statement syntax (excerpt from Annex A)*

## 12.4 Conditional if–else statement

The *conditional statement* (or *if–else* statement) is used to make a decision about whether a statement is executed. Formally, the syntax is given in Syntax 12-2.

conditional_statement ::= *// from A.6.6*
[ unique_priority ] **if** **(** cond_predicate **)** statement_or_null
{ **else** **if** **(** cond_predicate **)** statement_or_null }
[ **else** statement_or_null ]

unique_priority ::= **unique** | **unique0** | **priority**

cond_predicate ::=
expression_or_cond_pattern { **&&&** expression_or_cond_pattern }

expression_or_cond_pattern ::=
expression | cond_pattern

cond_pattern ::= expression **matches** pattern

*Syntax 12-2—Syntax for if–else statement (excerpt from Annex A)*

If the *cond_predicate* expression evaluates to true (that is, has a nonzero known value), the first statement shall be executed. If it evaluates to false (that is, has a zero value or the value is `x` or `z`), the first statement shall not execute. If there is an else statement and the *cond_predicate* expression is false, the else statement shall be executed.

Because the numeric value of the `if` expression is tested for being zero, certain shortcuts are possible. For example, the following two statements express the same logic:

```
if (expression)
if (expression != 0)
```

Because the **else** part of an if–else is optional, there can be confusion when an **else** is omitted from a nested **if** sequence. This is resolved by always associating the else with the closest previous if that lacks an else. In the following example, the else goes with the inner **if**, as shown by indentation.

```
if (index > 0)
   if (rega > regb)
         result = rega;
   else        // else applies to preceding if
         result = regb;
```

If that association is not desired, a begin-end block statement shall be used to force the proper association, as in the following example:

```
if (index > 0)
   begin
      if (rega > regb)
         result = rega;
   end
else result = regb;
```

### 12.4.1 if–else–if construct

The if–else construct can be chained.

```
if (expression)         statement;
else if (expression)    statement;
else if (expression)    statement;
else                    statement;
```

This sequence of if–else statements (known as an *if–else–if* construct) is the most general way of writing a multiway decision. The expressions shall be evaluated in order. If any expression is true, the statement associated with it shall be executed, and this shall terminate the whole chain. Each statement is either a single statement or a block of statements.

The last else of the if–else–if construct handles the none-of-the-above or default case where none of the other conditions were satisfied. Sometimes there is no explicit action for the default. In that case, the trailing else statement can be omitted, or it can be used for error checking to catch an unexpected condition.

The following module fragment uses the if–else statement to test the variable `index` to decide whether one of three `modify_segn` variables has to be added to the memory address and which increment is to be added to the `index` variable.

```
// declare variables and parameters
logic [31:0]   instruction,
               segment_area[255:0];
logic [7:0]    index;
logic [5:0]    modify_seg1,
               modify_seg2,
               modify_seg3;
parameter
      segment1 = 0,  inc_seg1 = 1,
      segment2 = 20, inc_seg2 = 2,
      segment3 = 64, inc_seg3 = 4,
      data = 128;

// test the index variable
if (index < segment2) begin
```

```
      instruction = segment_area [index + modify_seg1];
      index = index + inc_seg1;
   end
   else if (index < segment3) begin
      instruction = segment_area [index + modify_seg2];
      index = index + inc_seg2;
   end
   else if (index < data) begin
      instruction = segment_area [index + modify_seg3];
      index = index + inc_seg3;
   end
   else
      instruction = segment_area [index];
```

### 12.4.2 unique-if, unique0-if, and priority-if

The keywords **unique**, **unique0**, and **priority** can be used before an **if** to perform certain *violation checks*.

If the keywords **unique** or **priority** are used, a *violation report* shall be issued if no condition matches unless there is an explicit **else**. For example:

```
   unique if ((a==0) || (a==1)) $display("0 or 1");
   else if (a == 2) $display("2");
   else if (a == 4) $display("4"); // values 3,5,6,7 cause a violation report

   priority if (a[2:1]==0) $display("0 or 1");
   else if (a[2] == 0) $display("2 or 3");
   else $display("4 to 7");      // covers all other possible values,
                                 // so no violation report
```

If the keyword **unique0** is used, there shall be no violation if no condition is matched. For example:

```
   unique0 if ((a==0) || (a==1)) $display("0 or 1");
   else    if (a == 2) $display("2");
   else    if (a == 4) $display("4");  // values 3,5,6,7
                                       // cause no violation report
```

*Unique-if* and *unique0-if* assert that there is no overlap in a series of if–else–if conditions, i.e., they are mutually exclusive and hence it is safe for the conditions to be evaluated in parallel.

In unique-if and unique0-if, the conditions may be evaluated and compared in any order. The implementation shall continue the evaluations and comparisons after finding a true condition. A unique-if or unique0-if is *violated* if more than one condition is found true. The implementation shall issue a violation report and execute the statement associated with the true condition that appears first in the **if** statement, but not the statements associated with other true conditions.

After finding a uniqueness violation, the implementation is not required to continue evaluating and comparing additional conditions. The implementation is not required to try more than one order of evaluations and comparisons of conditions. The presence of side effects in conditions may cause nondeterministic results.

A *priority-if* indicates that a series of if–else–if conditions shall be evaluated in the order listed. In the preceding example, if the variable a had a value of 0, it would satisfy both the first and second conditions, requiring priority logic.

The **unique**, **unique0**, and **priority** keywords apply to the entire series of if–else–if conditions. In the preceding examples, it would have been illegal to insert any of these keywords after any of the occurrences of **else**. To nest another **if** statement within such a series of conditions, a begin-end block should be used.

#### 12.4.2.1 Violation reports generated by unique-if, unique0-if, and priority-if constructs

The descriptions in 12.4.2 mention several cases in which a violation report shall be generated by unique-if, unique0-if, or priority-if statements. These violation checks shall be immune to false violation reports due to zero-delay glitches in the active region set (see 3.4.1).

A unique, unique0, or priority violation check is evaluated at the time the statement is executed, but violation reporting is deferred until the Observed region of the current time step (see 4.4). The violation reporting can be controlled by using assertion control system tasks (see 20.12).

Once a violation is detected, a *pending violation report* is scheduled in the Observed region of the current time step. It is scheduled on a *violation report queue* associated with the currently executing process. A *violation report flush point* is said to be reached if any of the following conditions are met:

— The procedure, having been suspended earlier due to reaching an event control or wait statement, resumes execution.

— The procedure was declared by an **always_comb** or **always_latch** statement, and its execution is resumed due to a transition on one of its dependent signals.

If a violation report flush point is reached in a process, its violation report queue is cleared. Any pending violation reports are discarded.

In the Observed region of each simulation time step, each pending violation report shall mature or be confirmed for reporting. Once a report matures, it shall no longer be flushed. A tool-specific violation report mechanism is then used to report each violation, and the pending violation report is cleared from the appropriate process violation report queue.

The following is an example of a unique-if that is immune to zero-delay glitches in the active region set:

```
always_comb begin
   not_a = !a;
end

always_comb begin : a1
   u1: unique if (a)
      z = a | b;
   else if (not_a)
      z = a | c;
end
```

In this example, **unique if** `u1` is checking for overlap in the two conditional expressions. When `a` and `not_a` are in a state of 0 and 1, respectively, and `a` transitions to 1, this **unique if** could be executed while `a` and `not_a` are both true, so the violation check for uniqueness will fail. Since this check is in the active region set, the failure is not immediately reported. After the update to `not_a`, process `a1` will be rescheduled, which results in a flush of the original violation report. The violation check will now pass, and no violation will be reported.

Another example shows how looping constructs are likewise immune to zero-delay glitches in the active region set:

```
always_comb begin
   for (int j = 0; j < 3; j++)
```

```
      not_a[j] = !a[j];
end

always_comb begin : a1
   for (int j = 0; j < 3; j++)
      unique if (a[j])
         z[j] = a[j] | b[j];
      else if (not_a[j])
         z[j] = a[j] | c[j];
end
```

This example is identical to the previous example but adds loop statements. Each loop iteration independently checks for a uniqueness violation in the exact same manner as the previous example. Any iteration in the loop can report a uniqueness violation. If the process a1 is rescheduled, all violations in the loop are flushed and the entire loop is reevaluated.

### 12.4.2.2 If statement violation reports and multiple processes

As described in the previous subclauses (see 12.4.2 and 12.4.2.1), violation reports are inherently associated with the process in which they are executed. This means that a violation check within a task or function may be executed several times due to the task or function being called by several different processes, and each of these different process executions is independent. The following example illustrates this situation:

```
module fsm(...);
   function bit f1(bit a, bit not_a, ...)
      ...
      a1: unique if (a)
         ...
      else if (not_a)
         ...
   endfunction
   ...
   always_comb begin : b1
      some_stuff = f1(c, d, ...);
      ...
   end

   always_comb begin : b2
      other_stuff = f1(e, f, ...);
      ...
   end
endmodule
```

In this case, there are two different processes that may call process a1: b1 and b2. Suppose simulation executes the following scenario in the first passage through the Active region of each time step. Note that this example refers to three distinct points in simulation time and how glitch resolution is handled for each specific time step:

a) In time step 1, b1 executes with c=1 and d=1, and b2 executes with e=1 and f=1.

   In this first time step, since a1 fails independently for processes b1 and b2, its failure is reported twice.

b) In time step 2, b1 executes with c=1 and d=1, then again with c=1 and d=0.

   In this second time step, the failure of a1 in process b1 is flushed when the process is re-triggered, and since the final execution passes, no failure is reported.

c) In time step 3, `b1` executes with `c=1` and `d=1`, then `b2` executes with `e=1` and `f=0`.

   In this third time step, the failure in process `b1` does not see a flush point, so that failure is reported. In process `b2`, the violation check passes, so no failure is reported from that process.

## 12.5 Case statement

The *case* statement is a multiway decision statement that tests whether an expression matches one of a number of other expressions and branches accordingly. The case statement has the syntax shown in Syntax 12-3.

```
case_statement ::=                                                    // from A.6.7
    [ unique_priority ] case_keyword ( case_expression )
        case_item { case_item } endcase
  | [ unique_priority ] case_keyword (case_expression )matches
        case_pattern_item { case_pattern_item } endcase
  | [ unique_priority ] case ( case_expression ) inside
        case_inside_item { case_inside_item } endcase
case_keyword ::= case | casez | casex
case_expression ::= expression
case_item ::=
    case_item_expression { , case_item_expression } : statement_or_null
  | default [ : ] statement_or_null
case_pattern_item ::=
    pattern [ &&& expression ] : statement_or_null
  | default [ : ] statement_or_null
case_inside_item ::=
    open_range_list : statement_or_null
  | default [ : ] statement_or_null
case_item_expression ::= expression
```

*Syntax 12-3—Syntax for case statements (excerpt from Annex A)*

The *default* statement shall be optional. Use of multiple default statements in one case statement shall be illegal.

The *case_expression* and *case_item_expressions* are not required to be constant expressions.

A simple example of the use of the case statement is the decoding of variable `data` to produce a value for `result` as follows:

```
logic [15:0] data;
logic [9:0] result;

case (data)
   16'd0:  result = 10'b0111111111;
   16'd1:  result = 10'b1011111111;
   16'd2:  result = 10'b1101111111;
   16'd3:  result = 10'b1110111111;
   16'd4:  result = 10'b1111011111;
   16'd5:  result = 10'b1111101111;
```

```
      16'd6:  result = 10'b1111110111;
      16'd7:  result = 10'b1111111011;
      16'd8:  result = 10'b1111111101;
      16'd9:  result = 10'b1111111110;
      default result = 'x;
   endcase
```

The *case_expression* shall be evaluated exactly once and before any of the *case_item_expressions*. The *case_item_expressions* shall be evaluated and then compared in the exact order in which they appear. If there is a default *case_item*, it is ignored during this linear search. During the linear search, if one of the *case_item_expressions* matches the *case_expression*, then the statement associated with that *case_item* shall be executed, and the linear search shall terminate. If all comparisons fail and the default item is given, then the default item statement shall be executed. If the default statement is not given and all of the comparisons fail, then none of the *case_item* statements shall be executed.

Apart from syntax, the case statement differs from the multiway if–else–if construct in two important ways:

a) The conditional expressions in the if–else–if construct are more general than comparing one expression with several others, as in the case statement.
b) The case statement provides a definitive result when there are `x` and `z` values in an expression.

In a *case_expression* comparison, the comparison only succeeds when each bit matches exactly with respect to the values `0`, `1`, `x`, and `z`. As a consequence, care is needed in specifying the expressions in the case statement. The bit length of all the expressions needs to be equal, so that exact bitwise matching can be performed. Therefore, the length of all the *case_item_expressions*, as well as the *case_expression*, shall be made equal to the length of the longest *case_expression* and *case_item_expressions*. If any of these expressions is unsigned, then all of them shall be treated as unsigned. If all of these expressions are signed, then they shall be treated as signed.

The reason for providing a *case_expression* comparison that handles the `x` and `z` values is that it provides a mechanism for detecting such values and reducing the pessimism that can be generated by their presence.

*Example 1:* The following example illustrates the use of a case statement to handle `x` and `z` values properly:

```
   case (select[1:2])
      2'b00:  result = 0;
      2'b01:  result = flaga;
      2'b0x,
      2'b0z:  result = flaga ? 'x : 0;
      2'b10:  result = flagb;
      2'bx0,
      2'bz0:  result = flagb ? 'x : 0;
      default result = 'x;
   endcase
```

In this example, if `select[1]` is 0 and `flaga` is 0, then even if the value of `select[2]` is `x` or `z`, `result` should be 0—which is resolved by the third *case_item*.

*Example 2:* The following example shows another way to use a case statement to detect `x` and `z` values:

```
   case (sig)
      1'bz:    $display("signal is floating");
      1'bx:    $display("signal is unknown");
      default: $display("signal is %b", sig);
   endcase
```

### 12.5.1 Case statement with do-not-cares

Two other types of case statements are provided to allow handling of do-not-care conditions in the case comparisons. One of these treats high-impedance values (`z`) as do-not-cares, and the other treats both high-impedance and unknown (`x`) values as do-not-cares. These case statements can be used in the same way as the traditional case statement, but they begin with keywords **`casez`** and **`casex`**, respectively.

Do-not-care values (`z` values for **`casez`**, `z` and `x` values for **`casex`**) in any bit of either the *case_expression* or the *case_items* shall be treated as do-not-care conditions during the comparison, and that bit position shall not be considered.

The syntax of literal numbers allows the use of the question mark `(?)` in place of `z` in these case statements. This provides a convenient format for specification of do-not-care bits in case statements.

*Example 1*: The following is an example of the casez statement. It demonstrates an instruction decode, where values of the MSBs select which task should be called. If the MSB of `ir` is a 1, then the task instruction1 is called, regardless of the values of the other bits of `ir`.

```
logic [7:0] ir;

casez (ir)
   8'b1???????: instruction1(ir);
   8'b01??????: instruction2(ir);
   8'b00010???: instruction3(ir);
   8'b000001??: instruction4(ir);
endcase
```

*Example 2*: The following is an example of the casex statement. It demonstrates an extreme case of how do-not-care conditions can be dynamically controlled during simulation. In this example, if `r = 8'b01100110`, then the task `stat2` is called.

```
logic [7:0] r, mask;

mask = 8'bx0x0x0x0;
casex (r ^ mask)
   8'b001100xx: stat1;
   8'b1100xx00: stat2;
   8'b00xx0011: stat3;
   8'bxx010100: stat4;
endcase
```

### 12.5.2 Constant expression in case statement

A constant expression can be used for the *case_expression*. The value of the constant expression shall be compared against the *case_item_expressions*.

The following example demonstrates the usage by modeling a 3-bit priority encoder:

```
logic [2:0] encode ;

case (1)
   encode[2] : $display("Select Line 2") ;
   encode[1] : $display("Select Line 1") ;
   encode[0] : $display("Select Line 0") ;
   default     $display("Error: One of the bits expected ON");
endcase
```

In this example, the *case_expression* is a constant expression (`1`). The *case_items* are expressions (bit-selects) and are compared against the constant expression for a match.

### 12.5.3 unique-case, unique0-case, and priority-case

The **`case`**, **`casez`**, and **`casex`** keywords can be qualified by **`priority`**, **`unique`**, or **`unique0`** keywords to perform certain *violation checks*. These are collectively referred to as a *priority-case*, *unique-case*, or *unique0-case*. A priority-case shall act on the first match only. Unique-case and unique0-case assert that there are no overlapping *case_items* and hence that it is safe for the *case_items* to be evaluated in parallel.

In unique-case and unique0-case, the *case_expression* shall be evaluated exactly once and before any of the *case_item_expressions*. The *case_item_expressions* may be evaluated in any order and compared in any order. The implementation shall continue the evaluations and comparisons after finding a matching *case_item*. Unique-case and unique0-case are *violated* if more than one *case_item* is found to match the *case_expression*. The implementation shall issue a violation report and execute the statement associated with the matching *case_item* that appears first in the case statement, but not the statements associated with other matching *case_items*.

After finding a uniqueness violation, the implementation is not required to continue evaluating and comparing additional *case_items*. It is not a violation of uniqueness for a single *case_item* to contain more than one *case_item_expression* that matches the *case_expression*. If a *case_item_expression* matches the *case_expression*, the implementation is not required to evaluate additional *case_item_expressions* in the same *case_item*. The implementation is not required to try more than one order of evaluations and comparisons of *case_item_expressions*. The presence of side-effects in *case_item_expressions* may cause nondeterministic results.

If the case is qualified as priority or unique, the simulator shall issue a violation report if no *case_item* matches. A violation report may be issued at compile time if it is possible then to determine the violation. If it is not possible to determine the violation at compile time, a violation report shall be issued during run time. If the case is qualified as **`unique0`**, the implementation shall not issue a violation report if no *case_item* matches.

NOTE—By specifying **`unique`** or **`priority`**, it is not necessary to code a **`default`** case to trap unexpected case values.

Consider the following example:

```
bit [2:0] a;
unique case(a) // values 3,5,6,7 cause a violation report
   0,1: $display("0 or 1");
   2: $display("2");
   4: $display("4");
endcase

priority casez(a) // values 4,5,6,7 cause a violation report
   3'b00?: $display("0 or 1");
   3'b0??: $display("2 or 3");
endcase

unique0 case(a) // values 3,5,6,7 do not cause a violation report
   0,1: $display("0 or 1");
   2: $display("2");
   4: $display("4");
endcase
```

#### 12.5.3.1 Violation reports generated by unique-case, unique0-case, and priority-case constructs

The descriptions in 12.5.3 mention several cases in which a violation report shall be generated by unique-case, unique0-case, or priority-case statements. These violation checks shall be immune to false violation reports due to zero-delay glitches in the active region set (see 3.4.1). The violation reporting can be controlled by using assertion control system tasks (see 20.12).

The mechanics of handling zero-delay glitches shall be identical to those used when processing zero-delay glitches for unique-if, unique0-if, and priority-if constructs (see 12.4.2.1).

The following is an example of a unique-case that is immune to zero-delay glitches in the active region set:

```
always_comb begin
   not_a = !a;
end

always_comb begin : a1
   unique case (1'b1)
      a     : z = b;
      not_a : z = c;
   endcase
end
```

In this example the **unique case** is checking for overlap in the two *case_item* selects. When a and not_a are in state 0 and 1, respectively, and a transitions to 1, this **unique case** could be executed while a and not_a are both true, so the violation check for uniqueness will fail. But since this violation check is in the active region set, the failure is not reported immediately. After the update to not_a, process a1 will be rescheduled, which results in a flush of the original violation report. The violation check will now pass, and no violation will be reported.

#### 12.5.3.2 Case statement violation reports and multiple processes

*Case* violation reports shall behave in the same manner as *if* violation reports when dealing with multiple processes (see 12.4.2.2).

### 12.5.4 Set membership case statement

The keyword **inside** can be used after the parenthesized expression to indicate a set membership (see 11.4.13). In a *case-inside* statement, the *case_expression* shall be compared with each *case_item_expression* (*open_range_list*) using the set membership inside operator. The **inside** operator uses asymmetric wildcard matching (see 11.4.6). Accordingly, the *case_expression* shall be the left operand, and each *case_item_expression* shall be the right operand. The *case_expression* and each *case_item_expression* in braces shall be evaluated in the order specified by a normal case, unique-case, or priority-case statement. A *case_item* shall be matched when the inside operation compares the *case_expression* to the *case_item_expressions* and returns 1'b1 and no match when the operation returns 1'b0 or 1'bx. If all comparisons do not match and the default item is given, the default item statement shall be executed.

For example:

```
logic [2:0] status;
always @(posedge clock)
   priority case (status) inside
   1, 3 : task1; // matches 'b001 and 'b011
   3'b0?0, [4:7]: task2;   // matches 'b000 'b010 'b0x0 'b0z0
```

```
                           // 'b100 'b101 'b110 'b111
endcase  // priority case fails all other values including
         // 'b00x 'b01x 'bxxx
```

## 12.6 Pattern matching conditional statements

Pattern matching provides a visual and succinct notation to compare a value against structures, tagged unions, and constants and to access their members. Pattern matching can be used with case and if–else statements and with conditional expressions. Before describing pattern matching in those contexts, the general concepts are described first.

A pattern is a nesting of tagged union and structure expressions with identifiers, constant expressions (see 11.2.1), and the wildcard pattern ".*" at the leaves. For tagged union patterns, the identifier following the **tagged** keyword is a union member name. For **void** members, the nested member pattern is omitted.

```
pattern ::=                                                          // from A.6.7.1
    . variable_identifier
  | .*
  | constant_expression
  | tagged member_identifier [ pattern ]
  | '{ pattern { , pattern } }
  | '{ member_identifier : pattern { , member_identifier : pattern } }
```

*Syntax 12-4—Pattern syntax (excerpt from Annex A)*

A pattern always occurs in a context of known type because it is matched against an expression of known type. Recursively, its nested patterns also have known type. A constant expression pattern shall be of integral type. Thus a pattern can always be statically type-checked.

Each pattern introduces a new scope; the extent of this scope is described separately below for case statements, if–else statements, and conditional expressions. Each pattern identifier is implicitly declared as a new variable within the pattern's scope, with the unique type that it shall have based on its position in the pattern. Pattern identifiers shall be unique in the pattern, i.e., the same identifier cannot be used in more than one position in a single pattern.

In pattern-matching, the value *V* of an expression is always matched against a pattern, and static type checking verifies that *V* and the pattern have the same type. The result of a pattern match is as follows:

— A 1-bit determined value: 0 (false, or fail) or 1 (true, or succeed). The result cannot be x or z even if the value and pattern contain such bits.

— If the match succeeds, the pattern identifiers are bound to the corresponding members from *V*, using ordinary procedural assignment.

— Each pattern is matched using the following simple recursive rule:
  - An identifier pattern always succeeds (matches any value), and the identifier is bound to that value (using ordinary procedural assignment).
  - The wildcard pattern ".*" always succeeds.
  - A constant expression pattern succeeds if *V* is equal to its value.
  - A tagged union pattern succeeds if the value has the same tag and, recursively, if the nested pattern matches the member value of the tagged union.
  - A structure pattern succeeds if, recursively, each of the nested member patterns matches the corresponding member values in *V*. In structure patterns with named members, the textual order of members does not matter, and members can be omitted. Omitted members are ignored.

Conceptually, if the value *V* is seen as a flattened vector of bits, the pattern specifies which bits to match, with what values they should be matched, and, if the match is successful, which bits to extract and bind to the pattern identifiers. Matching can be insensitive to `x` and `z` values, as described in the following individual constructs.

### 12.6.1 Pattern matching in case statements

In a pattern-matching case statement, the expression in parentheses is followed by the keyword **`matches`**, and the statement contains a series of *case_pattern_items*. The left-hand side of a case item, before the colon ( : ), consists of a *pattern* and, optionally, the operator **`&&&`** followed by a Boolean filter *expression*. The right-hand side of a case item is a statement. Each pattern introduces a new scope, in which its pattern identifiers are implicitly declared; this scope extends to the optional filter expression and the statement in the right-hand side of the same case item. Thus different case items can reuse pattern identifiers.

All the patterns are completely statically type-checked. The expression being tested in the pattern-matching case statement shall have a known type, which is the same as the type of the pattern in each case item.

The expression in parentheses in a pattern-matching case statement shall be evaluated exactly once. Its value *V* shall be matched against the left-hand sides of the case items, one at a time, in the exact order given, ignoring the default case item if any. During this linear search, if a case item is selected, its statement is executed and the linear search is terminated. If no case item is selected and a default case item is given, then its statement is executed. If no case item is selected and no default case item is given, no statement is executed.

For a case item to be selected, the value *V* shall match the pattern (and the pattern identifiers are assigned the corresponding member values in *V*), and then the Boolean filter expression shall evaluate to true (a determined value other than 0).

*Example 1:*

```
typedef union tagged {
   void Invalid;
   int Valid;
} VInt;
...
VInt v;
...
case (v) matches
   tagged Invalid  : $display ("v is Invalid");
   tagged Valid .n : $display ("v is Valid with value %d", n);
endcase
```

In the case statement, if `v` currently has the `Invalid` tag, the first pattern is matched. Otherwise, it must have the `Valid` tag, and the second pattern is matched. The identifier `n` is bound to the value of the `Valid` member, and this value is displayed. It is impossible to access the integer member value (`n`) when the tag is `Invalid`.

*Example 2:*

```
typedef union tagged {
   struct {
      bit [4:0] reg1, reg2, regd;
   } Add;
   union tagged {
      bit [9:0] JmpU;
```

```
        struct {
           bit [1:0] cc;
           bit [9:0] addr;
        } JmpC;
     } Jmp;
  } Instr;
  ...
  Instr instr;
  ...
  case (instr) matches
     tagged Add '{.r1, .r2, .rd} &&& (rd != 0) : rf[rd] = rf[r1] + rf[r2];
     tagged Jmp .j : case (j) matches
                        tagged JmpU .a       : pc = pc + a;
                        tagged JmpC '{.c, .a}: if (rf[c]) pc = a;
                     endcase
  endcase
```

If `instr` holds an `Add` instruction, the first pattern is matched, and the identifiers `r1`, `r2`, and `rd` are bound to the three register fields in the nested structure value. The right-hand statement executes the instruction on the register file `rf`. It is impossible to access these register fields if the tag is `Jmp`. If `instr` holds a `Jmp` instruction, the second pattern is matched, and the identifier `j` is bound to the nested tagged union value. The inner case statement, in turn, matches this value against `JmpU` and `JmpC` patterns and so on.

*Example 3* (same as previous example, but using wildcard and constant patterns to eliminate the `rd = 0` case; in some processors, register 0 is a special "discard" register):

```
  case (instr) matches
     tagged Add '{.*, .*, 0}     : ; // no op
     tagged Add '{.r1, .r2, .rd}  : rf[rd] = rf[r1] + rf[r2];
     tagged Jmp .j : case (j) matches
                        tagged JmpU .a        : pc = pc + a;
                        tagged JmpC '{.c, .a} : if (rf[c]) pc = a;
                     endcase
  endcase
```

*Example 4* (same as previous example except that the first inner case statement involves only structures and constants but no tagged unions):

```
  case (instr) matches
     tagged Add s:  case (s) matches
                       '{.*, .*, 0}     : ; // no op
                       '{.r1, .r2, .rd}  : rf[rd] = rf[r1] + rf[r2];
                    endcase
     tagged Jmp .j: case (j) matches
                      tagged JmpU .a        : pc = pc + a;
                      tagged JmpC '{.c, .a} : if (rf[c]) pc = a;
                    endcase
  endcase
```

*Example 5* (same as previous example, but using nested tagged union patterns):

```
  case (instr) matches
     tagged Add '{.r1, .r2, .rd} &&& (rd != 0) : rf[rd] = rf[r1] + rf[r2];
     tagged Jmp (tagged JmpU .a)               : pc = pc + a;
     tagged Jmp (tagged JmpC '{.c, .a})        : if (rf[c]) pc = a;
  endcase
```

*Example 6* (same as previous example, with named structure components):

```
case (instr) matches
   tagged Add '{reg2:.r2,regd:.rd,reg1:.r1} &&& (rd != 0):
                                            rf[rd] = rf[r1] + rf[r2];
   tagged Jmp (tagged JmpU .a)               : pc = pc + a;
   tagged Jmp (tagged JmpC '{addr:.a,cc:.c}) : if (rf[c]) pc = a;
endcase
```

The **casez** and **casex** keywords can be used instead of **case**, with the same semantics. In other words, during pattern matching, wherever 2 bits are compared (whether they are tag bits or members), the **casez** form ignores `z` bits, and the **casex** form ignores both `z` and `x` bits.

The **priority** and **unique** qualifiers can also be used. **priority** implies that some case item must be selected. **unique** implies that some case item must be selected and also implies that exactly one case item will be selected so that they can be evaluated in parallel.

### 12.6.2 Pattern matching in if statements

The predicate in an if–else statement can be a series of clauses separated with the **&&&** operator. Each clause is either an expression (used as a Boolean filter) or has the form: *expression* **matches** *pattern*. The clauses represent a sequential conjunction from left to right (i.e., if any clause fails, the remaining clauses are not evaluated) and all of them shall succeed for the predicate to be true. Boolean expression clauses are evaluated as usual. Each pattern introduces a new scope, in which its pattern identifiers are implicitly declared; this scope extends to the remaining clauses in the predicate and to the corresponding "true" arm of the if–else statement.

In each *e* **matches** *p* clause, *e* and *p* shall have the same known statically known type. The value of *e* is matched against the pattern *p* as previously described.

Even though the pattern matching clauses always return a defined 1-bit result, the overall result can be ambiguous because of the Boolean filter expressions in the predicate. The standard semantics of if–else statements holds, i.e., the first statement is executed if, and only if, the result is a determined value other than 0.

*Example 1:*

```
if (e matches (tagged Jmp (tagged JmpC '{cc:.c,addr:.a})))
   ... // c and a can be used here
else
...
```

*Example 2* (same as previous example, illustrating a sequence of two pattern matches with identifiers bound in the first pattern used in the second pattern):

```
if (e matches (tagged Jmp .j) &&&
    j matches (tagged JmpC '{cc:.c,addr:.a}))
      ... // c and a can be used here
else
...
```

*Example 3* (same as first example, but adding a Boolean expression to the sequence of clauses). The idea expressed is "if `e` is a conditional jump instruction and the condition register is not equal to zero ...".

```
    if (e matches (tagged Jmp (tagged JmpC '{cc:.c,addr:.a}))
       &&& (rf[c] != 0))
       ... // c and a can be used here
    else
    ...
```

The **priority** and **unique** qualifiers can be used even if they use pattern matching.

### 12.6.3 Pattern matching in conditional expressions

A conditional expression (`e1 ? e2 : e3`) can also use pattern matching, i.e., the predicate `e1` can be a sequence of expressions and "*expression* **matches** *pattern*" clauses separated with the **&&&** operator, just like the predicate of an if–else statement. The clauses represent a sequential conjunction from left to right, (i.e., if any clause fails, the remaining clauses are not evaluated) and all of them shall succeed for the predicate to be true. Boolean expression clauses are evaluated as usual. Each pattern introduces a new scope, in which its pattern identifiers are implicitly declared; this scope extends to the remaining clauses in the predicate and to the consequent expression `e2`.

As described in the previous subclause, `e1` can evaluate to true, false, or an ambiguous value. The semantics of the overall conditional expression is described in 11.4.11, based on these three possible outcomes for `e1`.

## 12.7 Loop statements

SystemVerilog provides six types of looping constructs, as shown in Syntax 12-5.

```
loop_statement ::=                                                      // from A.6.8
      forever statement_or_null
    | repeat ( expression ) statement_or_null
    | while ( expression ) statement_or_null
    | for ( [ for_initialization ] ; [ expression ] ; [ for_step ] )
         statement_or_null
    | do statement_or_null while ( expression ) ;
    | foreach ( ps_or_hierarchical_array_identifier [ loop_variables ] ) statement
for_initialization ::=
      list_of_variable_assignments
    | for_variable_declaration { , for_variable_declaration }
for_variable_declaration ::=
    [ var ] data_type variable_identifier = expression { , variable_identifier = expression }14
for_step ::= for_step_assignment { , for_step_assignment }
for_step_assignment ::=
      operator_assignment
    | inc_or_dec_expression
    | function_subroutine_call
loop_variables ::= [ index_variable_identifier ] { , [ index_variable_identifier ] }
```

14) When a type_reference is used in a net declaration, it shall be preceded by a net type keyword; and when it is used in a variable declaration, it shall be preceded by the **var** keyword.

*Syntax 12-5—Loop statement syntax (excerpt from Annex A)*

### 12.7.1 The for-loop

The *for-loop* controls execution of its associated statement(s) by a three-step process, as follows:

a) Unless the optional *for_initialization* is omitted, executes one or more *for_initialization* assignments, which are normally used to initialize a variable that controls the number of times the loop is executed.

b) Unless the optional *expression* is omitted, evaluates the *expression*. If the result is false (as defined in 12.4), the for-loop shall exit. Otherwise, or if the *expression* is omitted, the for-loop shall execute its associated statement(s) and then perform step c). If the expression evaluates to an unknown or high-impedance value, it shall be treated as zero.

c) Unless the optional *for_step* is omitted, executes one or more *for_step* assignments, normally used to modify the value of the loop-control variable, then repeats step b).

The variables used to control a for-loop can be declared prior to the loop. If loops in two or more parallel processes use the same loop control variable, there is a potential of one loop modifying the variable while other loops are still using it.

The variables used to control a for-loop can also be declared within the loop, as part of the *for_initialization* assignments. This creates an implicit begin-end block around the loop, containing declarations of the loop variables with automatic lifetime. This block creates a new hierarchical scope, making the variables local to the loop scope. The block is unnamed by default, but can be named by adding a statement label (9.3.5) to the for-loop statement. Thus, other parallel loops cannot inadvertently affect the loop control variable. For example:

```
module m;

   initial begin
      for (int i = 0; i <= 255; i++)
         ...
   end

   initial begin
      loop2: for (int i = 15; i >= 0; i--)
         ...
   end
endmodule
```

This is equivalent to the following:

```
module m;
   initial begin
      begin
         automatic int i;
         for (i = 0; i <= 255; i++)
            ...
      end
   end

   initial begin
      begin : loop2
         automatic int i;
         for (i = 15; i >= 0; i--)
            ...
      end
   end
endmodule
```

Only for-loop statements containing variable declarations as part of the for-initialization assignments create implicit begin-end blocks around them.

The initial declaration or assignment statement can be one or more comma-separated statements. The step assignment can also be one or more comma-separated assignment statements, increment or decrement expressions, or function calls.

```
for ( int count = 0; count < 3; count++ )
   value = value +((a[count]) * (count+1));

for ( int count = 0, done = 0, j = 0; j * count < 125; j++, count++)
   $display("Value j = %d\n", j );
```

In a for-loop initialization, either all or none of the control variables shall be locally declared. In the second loop of the example above, `count`, `done`, and `j` are all locally declared. The following would be illegal because it attempts to locally declare `y` whereas `x` was not locally declared:

```
for (x = 0, int y = 0; ...)
...
```

In a for-loop initialization that declares multiple local variables, the initialization expression of a local variable can use earlier local variables.

```
for (int i = 0, j = i+offset; i < N; i++,j++)
...
```

### 12.7.2 The repeat loop

The *repeat-loop* executes a statement a fixed number of times. If the expression evaluates to unknown or high impedance, it shall be treated as zero, and no statement shall be executed.

In the following example of a repeat-loop, add and shift operators implement a multiplier:

```
parameter size = 8, longsize = 16;
logic [size:1] opa, opb;
logic [longsize:1] result;

begin : mult
   logic [longsize:1] shift_opa, shift_opb;
   shift_opa = opa;
   shift_opb = opb;
   result = 0;
   repeat (size) begin
         if (shift_opb[1])
            result = result + shift_opa;
         shift_opa = shift_opa << 1;
         shift_opb = shift_opb >> 1;
   end
end
```

### 12.7.3 The foreach-loop

The *foreach-loop* construct specifies iteration over the elements of an array. Its argument is an identifier that designates any type of array followed by a comma-separated list of loop variables enclosed in square brackets. Each loop variable corresponds to one of the dimensions of the array. The foreach-loop is similar to a repeat-loop that uses the array bounds to specify the repeat count instead of an expression.

*Examples:*

```
string words [2] = '{ "hello", "world" };
int prod [1:8] [1:3];

foreach( words [ j ] )
   $display( j , words[j] );     // print each index and value

foreach( prod[ k, m ] )
   prod[k][m] = k * m;           // initialize
```

The number of loop variables shall not be greater than the number of dimensions of the array variable. Loop variables may be omitted to indicate no iteration over that dimension of the array, and trailing commas in the list may also be omitted. As in a for-loop (12.7.1), a foreach-loop creates an implicit begin-end block around the loop statement, containing declarations of the loop variables with automatic lifetime. This block creates a new hierarchical scope, making the variables local to the loop scope. The block is unnamed by default, but can be named by adding a statement label (9.3.5) to the foreach statement. foreach-loop variables are read-only. The type of each loop variable is implicitly declared to be consistent with the type of array index. It shall be an error for any loop variable to have the same identifier as the array.

The mapping of loop variables to array indices is determined by the dimension cardinality, as described in 20.7. The foreach-loop arranges for higher cardinality indices to change more rapidly.

```
//      1  2  3             3    4       1   2   -> Dimension numbers
int A [2][3][4];     bit [3:0][2:1] B [5:1][4];

foreach( A [ i, j, k ] ) ...
foreach( B [ q, r, , s ] ) ...
```

The first foreach-loop causes `i` to iterate from 0 to 1, `j` from 0 to 2, and `k` from 0 to 3. The second foreach-loop causes `q` to iterate from 5 to 1, `r` from 0 to 3, and `s` from 2 to 1 (iteration over the third index is skipped).

If the dimensions of a dynamically sized array are changed while iterating over a foreach-loop construct, the results are undefined and may cause invalid index values to be generated.

Multiple loop variables correspond to nested loops that iterate over the given indices. The nesting of the loops is determined by the dimension cardinality; outer loops correspond to lower cardinality indices. In the preceding first example, the outermost loop iterates over `i`, and the innermost loop iterates over `k`.

When loop variables are used in expressions other than as indices to the designated array, they are auto-cast into a type consistent with the type of index. For fixed-size and dynamic arrays, the auto-cast type is **`int`**. For associative arrays indexed by a specific index type, the auto-cast type is the same as the index type. To use different types, an explicit cast can be used.

### 12.7.4 The while-loop

The *while-loop* repeatedly executes a statement as long as a control expression is true (as defined in 12.4). If the expression is not true at the beginning of the execution of the while-loop, the statement shall not be executed at all.

The following example counts the number of logic 1 values in `data`:

```
begin : count1s
   logic [7:0] tempreg;
```

```
      count = 0;
      tempreg = data;
      while (tempreg) begin
            if (tempreg[0])
               count++;
            tempreg >>= 1;
      end
   end
```

### 12.7.5 The do...while-loop

The *do...while-loop* differs from the while-loop in that a do...while-loop tests its control expression at the end of the loop. Loops with a test at the end are sometimes useful to save duplication of the loop body.

```
   string s;
   if ( map.first( s ) )
      do
         $display( "%s : %d\n", s, map[ s ] );
      while ( map.next( s ) );
```

The condition can be any expression that can be treated as a Boolean. It is evaluated after the statement.

### 12.7.6 The forever-loop

The *forever-loop* repeatedly executes a statement. To avoid a zero-delay infinite loop, which could hang the simulation event scheduler, the forever loop should only be used in conjunction with the timing controls or the disable statement. For example:

```
   initial begin
      clock1 <= 0;
      clock2 <= 0;
      fork
         forever #10 clock1 = ~clock1;
         #5 forever #10 clock2 = ~clock2;
      join
   end
```

## 12.8 Jump statements

```
jump_statement ::=                                  // from A.6.5
     return [ expression ] ;
   | break ;
   | continue ;
```

*Syntax 12-6—Jump statement syntax (excerpt from Annex A)*

SystemVerilog provides the C-like jump statements **break**, **continue**, and **return**.

```
   break                 // break out of loop, as in C
   continue              // skip to end of loop, as in C
   return expression     // exit from a function
   return                // exit from a task or void function
```

The **continue** and **break** statements can only be used in a loop. The **continue** statement jumps to the end of the loop and executes the loop control if present. The **break** statement jumps out of the loop.

The **continue** and **break** statements cannot be used inside a fork-join block to control a loop outside the fork-join block.

The **return** statement can only be used in a subroutine. In a function returning a value, the return statement shall have an expression of the correct type.

NOTE—SystemVerilog does not include the C **goto** statement.

# 13. Tasks and functions (subroutines)

## 13.1 General

This clause describes the following:

— Task declarations
— Function declarations
— Calling tasks and functions

## 13.2 Overview

Tasks and functions provide the ability to execute common procedures from several different places in a description. They also provide a means of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions. This clause discusses the differences between tasks and functions, describes how to define and invoke tasks and functions, and presents examples of each.

Tasks and functions are collectively referred to as *subroutines*.

The following rules distinguish tasks from functions, with exceptions noted in 13.4.4:

— The statements in the body of a function shall execute in one simulation time unit; a task may contain time-controlling statements.
— A function cannot enable a task; a task can enable other tasks and functions.
— A nonvoid function shall return a single value; a task or void function shall not return a value.
— A nonvoid function can be used as an operand in an expression; the value of that operand is the value returned by the function.

For example:

Either a task or a function can be defined to switch bytes in a 16-bit word. The task would return the switched word in an output argument; therefore, the source code to enable a task called `switch_bytes` could look like the following example:

```
switch_bytes (old_word, new_word);
```

The task `switch_bytes` would take the bytes in `old_word`, reverse their order, and place the reversed bytes in `new_word`.

A word-switching function would return the switched word as the return value of the function. Thus, the function call for the function `switch_bytes` could look like the following example:

```
new_word = switch_bytes (old_word);
```

## 13.3 Tasks

A task shall be enabled from a statement that defines the argument values to be passed to the task and the variables that receive the results. Control shall be passed back to the enabling process after the task has completed. Thus, if a task has timing controls inside it, then the time of enabling a task can be different from the time at which the control is returned. A task can enable other tasks, which in turn can enable still other tasks—with no limit on the number of tasks enabled. Regardless of how many tasks have been enabled, control shall not return until all enabled tasks have completed.

The syntax for task declarations is as follows in Syntax 13-1.

```
task_declaration ::= task [ lifetime ] task_body_declaration                         // from A.2.7
task_body_declaration ::=
      [ interface_identifier . | class_scope ] task_identifier ;
      { tf_item_declaration }
      { statement_or_null }
      endtask [ : task_identifier ]
    | [ interface_identifier . | class_scope ] task_identifier ( [ tf_port_list ] ) ;
      { block_item_declaration }
      { statement_or_null }
      endtask [ : task_identifier ]
tf_item_declaration ::=
      block_item_declaration
    | tf_port_declaration
tf_port_list ::=
      tf_port_item { , tf_port_item }
tf_port_item[23] ::=
      { attribute_instance }
         [ tf_port_direction ] [ var ] data_type_or_implicit
         [ port_identifier { variable_dimension } [ = expression ] ]
tf_port_direction ::= port_direction | const ref
tf_port_declaration ::=
      { attribute_instance } tf_port_direction [ var ] data_type_or_implicit list_of_tf_variable_identifiers ;
lifetime ::= static | automatic                                                       // from A.2.1
signing ::= signed | unsigned                                                         // from A.2.2.1
data_type_or_implicit ::=
      data_type
    | implicit_data_type
implicit_data_type ::= [ signing ] { packed_dimension }
```

23) In a tf_port_item, it shall be illegal to omit the explicit port_identifier except within a function_prototype or task_prototype.

*Syntax 13-1—Task syntax (excerpt from Annex A)*

A task declaration has the formal arguments either in parentheses (like ANSI C) or in declarations and directions.

```
task mytask1 (output int x, input logic y);
   ...
endtask

task mytask2;
   output x;
   input y;
   int x;
   logic y;
   ...
endtask
```

Each formal argument has one of the following directions:

| | |
|---|---|
| **input** | // copy value in at beginning |
| **output** | // copy value out at end |
| **inout** | // copy in at beginning and out at end |
| **ref** | // pass reference (see 13.5.2) |

There is a default direction of **input** if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments `a` and `b` default to inputs, and `u` and `v` are both outputs:

```
task mytask3(a, b, output logic [15:0] u, v);
   ...
endtask
```

Each formal argument has a data type that can be explicitly declared or inherited from the previous argument. If the data type is not explicitly declared, then the default data type is **logic** if it is the first argument or if the argument direction is explicitly specified. Otherwise, the data type is inherited from the previous argument.

An array can be specified as a formal argument to a task. For example:

```
// the resultant declaration of b is input [3:0][7:0] b[3:0]
task mytask4(input [3:0][7:0] a, b[3:0], output [3:0][7:0] y[1:0]);
   ...
endtask
```

Multiple statements can be written between the task declaration and **endtask**. Statements are executed sequentially, the same as if they were enclosed in a **begin** .... **end** group. It shall also be legal to have no statements at all.

A task exits when the **endtask** is reached. The **return** statement can be used to exit the task before the **endtask** keyword.

A call to a task is also referred to as a *task enable* (see 13.5 for more details on calling tasks).

*Example 1:* The following example illustrates the basic structure of a task definition with five arguments:

```
task my_task;
   input a, b;
   inout c;
   output d, e;
   . . .        // statements that perform the work of the task
   . . .
   c = a;       // the assignments that initialize result outputs
   d = b;
   e = c;
endtask
```

Or using the second form of a task declaration, the task could be defined as follows:

```
task my_task (input a, b, inout c, output d, e);
   . . .        // statements that perform the work of the task
   . . .
   c = a;       // the assignments that initialize result variables
   d = b;
```

```
      e = c;
   endtask
```

The following statement calls the task:

```
   initial
      my_task (v, w, x, y, z);
```

The task call arguments (v, w, x, y, and z) correspond to the arguments (a, b, c, d, and e) defined by the task. At the time of the call, the input and inout type arguments (a, b, and c) receive the values passed in v, w, and x. Thus, execution of the call effectively causes the following assignments:

```
   a = v;
   b = w;
   c = x;
```

As part of the processing of the task, the task definition for my_task places the computed result values into c, d, and e. When the task completes, the following assignments to return the computed values to the calling process are performed:

```
   x = c;
   y = d;
   z = e;
```

*Example 2:* The following example illustrates the use of tasks by describing a traffic light sequencer:

```
   module traffic_lights;
      logic clock, red, amber, green;
      parameter   on = 1, off = 0, red_tics = 350,
                  amber_tics = 30, green_tics = 200;

      // initialize colors
      initial red = off;
      initial amber = off;
      initial green = off;

      always begin                     // sequence to control the lights
         red = on;                     // turn red light on
         light(red, red_tics);         // and wait.
         green = on;                   // turn green light on
         light(green, green_tics);     // and wait.
         amber = on;                   // turn amber light on
         light(amber, amber_tics);     // and wait.
      end

      // task to wait for 'tics' positive edge clocks
      // before turning 'color' light off
      task light (output color, input [31:0] tics);
         repeat (tics) @ (posedge clock);
         color = off;                  // turn light off.
      endtask: light

      always begin                     // waveform for the clock
         #100 clock = 0;
         #100 clock = 1;
      end
   endmodule: traffic_lights
```

### 13.3.1 Static and automatic tasks

Tasks defined within a module, interface, program, or package default to being static, with all declared items being statically allocated. These items shall be shared across all uses of the task executing concurrently.

Tasks can be defined to use automatic storage in the following two ways:

— Explicitly declared using the optional automatic keyword as part of the task declaration.
— Implicitly declared by defining the task within a module, interface, program, or package that is defined as automatic.

Tasks defined within a class are always automatic (see 8.6).

All items declared inside automatic tasks are allocated dynamically for each invocation. All formal arguments and local variables are stored on the stack.

Automatic task items cannot be accessed by hierarchical references. Automatic tasks can be invoked through use of their hierarchical name.

Specific local variables can be declared as **`automatic`** within a static task or as **`static`** within an automatic task.

### 13.3.2 Task memory usage and concurrent activation

A task may be enabled more than once concurrently. All variables of an automatic task shall be replicated on each concurrent task invocation to store state specific to that invocation. All variables of a static task shall be static in that there shall be a single variable corresponding to each declared local variable in a module instance, regardless of the number of concurrent activations of the task. However, static tasks in different instances of a module shall have separate storage from each other.

Variables declared in static tasks, including **`input`**, **`output`**, and **`inout`** type arguments, shall retain their values between invocations. They shall be initialized to the default initialization value as described in 6.8.

Variables declared in automatic tasks, including **`output`** type arguments, shall be initialized to the default initialization value whenever execution enters their scope. **`input`** and **`inout`** type arguments shall be initialized to the values passed from the expressions corresponding to these arguments listed in the task-enabling statements.

Because variables declared in automatic tasks are deallocated at the end of the task invocation, they shall not be used in certain constructs that might refer to them after that point:

— They shall not be assigned values using nonblocking assignments or procedural continuous assignments.
— They shall not be referenced by procedural continuous assignments or procedural force statements.
— They shall not be referenced in intra-assignment event controls of nonblocking assignments.
— They shall not be traced with system tasks such as **`$monitor`** and **`$dumpvars`**.

## 13.4 Functions

The primary purpose of a function is to return a value that is to be used in an expression. A void function can also be used instead of a task to define a subroutine that executes and returns within a single time step. The rest of this clause explains how to define and use functions.

Functions have restrictions that make certain they return without suspending the process that enables them. The following rules shall govern their usage, with exceptions noted in 13.4.4:

a) A function shall not contain any time-controlled statements. That is, any statements containing `#`, `##`, `@`, **`fork-join`**, **`fork-join_any`**, **`wait`**, **`wait_order`**, or **`expect`**.

b) A function shall not enable tasks regardless of whether those tasks contain time-controlling statements.

c) Functions may enable fine-grain process control methods to suspend its own or another process (see 9.7).

The syntax for defining a function is given in Syntax 13-2.

---

function_declaration ::= **function** [ lifetime ] function_body_declaration *// from A.2.6*

function_body_declaration ::=
    function_data_type_or_implicit
      [ interface_identifier **.** | class_scope ] function_identifier **;**
    { tf_item_declaration }
    { function_statement_or_null }
    **endfunction** [ **:** function_identifier ]
  | function_data_type_or_implicit
      [ interface_identifier **.** | class_scope ] function_identifier **(** [ tf_port_list ] **)** **;**
    { block_item_declaration }
    { function_statement_or_null }
    **endfunction** [ **:** function_identifier ]

function_data_type_or_implicit ::=
    data_type_or_void
  | implicit_data_type

data_type ::= *// from A.2.2.1*
    integer_vector_type [ signing ] { packed_dimension }
  | integer_atom_type [ signing ]
  | non_integer_type
  | struct_union [ **packed** [ signing ] ] **{** struct_union_member { struct_union_member } **}**
      { packed_dimension }[13]
  | **enum** [ enum_base_type ] **{** enum_name_declaration { **,** enum_name_declaration } **}**
      { packed_dimension }
  | **string**
  | **chandle**
  | **virtual** [ **interface** ] interface_identifier [ parameter_value_assignment ] [ **.** modport_identifier ]
  | [ class_scope | package_scope ] type_identifier { packed_dimension }
  | class_type
  | **event**
  | ps_covergroup_identifier
  | type_reference[14]

signing ::= **signed** | **unsigned**

lifetime ::= **static** | **automatic** *// from A.2.1.3*

13) When a packed dimension is used with the **struct** or **union** keyword, the **packed** keyword shall also be used.

14) When a type_reference is used in a net declaration, it shall be preceded by a net type keyword; and when it is used in a variable declaration, it shall be preceded by the **var** keyword.

---

*Syntax 13-2—Function syntax (excerpt from Annex A)*

To indicate the return type of a function, its declaration can either include an explicit *data_type_or_void* or use an implicit syntax that indicates only the ranges of the packed dimensions and, optionally, the signedness. When the implicit syntax is used, the return type is the same as if the implicit syntax had been immediately preceded by the **logic** keyword. In particular, the implicit syntax can be empty, in which case the return type is a **logic** scalar. A function can also be **void**, without a return value (see 13.4.1).

A function declaration has the formal arguments either in parentheses (like ANSI C) or in declarations and directions, as follows:

```
function logic [15:0] myfunc1(int x, int y);
   ...
endfunction

function logic [15:0] myfunc2;
   input int x;
   input int y;
   ...
endfunction
```

Functions can have the same formal arguments as tasks. Function argument directions are as follows:

```
   input        // copy value in at beginning
   output       // copy value out at end
   inout        // copy in at beginning and out at end
   ref          // pass reference (see 13.5.2)
```

Function declarations default to the formal direction **input** if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments `a` and `b` default to inputs, and `u` and `v` are both outputs:

```
function logic [15:0] myfunc3(int a, int b, output logic [15:0] u, v);
   ...
endfunction
```

Each formal argument has a data type that can be explicitly declared or inherited from the previous argument. If the data type is not explicitly declared, then the default data type is **logic** if it is the first argument or if the argument direction is explicitly specified. Otherwise the data type is inherited from the previous argument. An array can be specified as a formal argument to a function, for example:

```
function [3:0][7:0] myfunc4(input [3:0][7:0] a, b[3:0]);
   ...
endfunction
```

It shall be illegal to call a function with **output**, **inout**, or **ref** arguments in an event expression, in an expression within a procedural continuous assignment, or in an expression that is not within a procedural statement. However, a **const ref** function argument shall be legal in this context (see 13.5.2).

Multiple statements can be written between the function header and **endfunction**. Statements are executed sequentially, as if they were enclosed in a **begin-end** group. It is also legal to have no statements at all, in which case the function returns the current value of the implicit variable that has the same name as the function.

### 13.4.1 Return values and void functions

The function definition shall implicitly declare a variable, internal to the function, with the same name as the function. This variable has the same type as the function return value. Function return values can be specified in two ways, either by using a **return** statement or by assigning a value to the internal variable with the same name as the function. For example:

```
function [15:0] myfunc1 (input [7:0] x,y);
   myfunc1 = x * y - 1;    // return value assigned to function name
endfunction

function [15:0] myfunc2 (input [7:0] x,y);
   return x * y - 1;    //return value is specified using return statement
endfunction
```

The **return** statement shall override any value assigned to the function name. When the return statement is used, nonvoid functions shall specify an expression with the return.

A function return can be a structure or union. In this case, a hierarchical name used inside the function and beginning with the function name is interpreted as a member of the return value. If the function name is used outside the function, the name indicates the scope of the whole function. If the function name is used within a hierarchical name, it also indicates the scope of the whole function.

Functions can be declared as type **void**, which do not have a return value. Function calls may be used as expressions unless of type **void**, which are statements:

```
a = b + myfunc1(c, d);  // call myfunc1 (defined above) as an expression

myprint(a);             // call myprint (defined below) as a statement

function void myprint (int a);
   ...
endfunction
```

Functions that return a value may be used in an assignment or an expression. Calling a nonvoid function as if it has no return value shall be legal, but shall issue a warning. The function can be used as a statement and the return value discarded without a warning by casting the function call to the **void** type.

```
void'(some_function());
```

It shall be illegal to declare another object with the same name as the function in the scope where the function is declared or explicitly imported. It shall also be illegal to declare another object with the same name as the function inside the function scope.

### 13.4.2 Static and automatic functions

Functions defined within a module, interface, program, or package default to being static, with all declared items being statically allocated. These items shall be shared across all uses of the function executing concurrently.

Functions can be defined to use automatic storage in the following two ways:

— Explicitly declared using the optional automatic keyword as part of the function declaration.
— Implicitly declared by defining the function within a module, interface, program, or package that is defined as automatic.

Functions defined within a class are always automatic (see 8.6).

An automatic function is reentrant, with all the function declarations allocated dynamically for each concurrent function call. Automatic function items cannot be accessed by hierarchical references. Automatic functions can be invoked through the use of their hierarchical name.

Specific local variables can be declared as **automatic** within a static function or as **static** within an automatic function.

The following example defines a function called `factorial` that returns an integer value. The `factorial` function is called iteratively and the results are printed.

```
module tryfact;

   // define the function
   function automatic integer factorial (input [31:0] operand);
      if (operand >= 2)
         factorial = factorial (operand - 1) * operand;
      else
         factorial = 1;
   endfunction: factorial

   // test the function
   integer result;
   initial begin
      for (int n = 0; n <= 7; n++) begin
         result = factorial(n);
         $display("%0d factorial=%0d", n, result);
      end
   end
endmodule: tryfact
```

The simulation results are as follows:

```
0 factorial=1
1 factorial=1
2 factorial=2
3 factorial=6
4 factorial=24
5 factorial=120
6 factorial=720
7 factorial=5040
```

### 13.4.3 Constant functions

Constant functions are a subset of normal functions that shall meet the following constraints:

— A constant function shall not have output, inout, or ref arguments.
— A void function shall not be a constant function.
— A DPI import function (see 35.2.1) shall not be a constant function.
— A constant function shall not contain a statement that directly schedules an event to execute after the function has returned.
— A constant function shall not contain any fork constructs.
— Constant functions shall contain no hierarchical references.

— Any function invoked within a constant function shall be a constant function local to the current module.
— It shall be legal to call any system function that is allowed in a constant_expression (see 11.2.1). This includes `$bits` and the array query functions. Calls to other system functions shall be illegal.
— All system task calls within a constant function shall be ignored.
— All parameter values used within the function shall be defined before the use of the invoking constant function call (i.e., any parameter use in the evaluation of a constant function call constitutes a use of that parameter at the site of the original constant function call). A constant function may reference parameters defined in packages or `$unit`.
— All identifiers that are not parameters or functions shall be declared locally to the current function.
— If constant functions use any parameter value that is affected directly or indirectly by a **defparam** statement (see 23.10.1), the result shall be undefined. This can produce an error or the constant function can return an indeterminate value.
— Constant functions shall not be declared inside a generate block (see Clause 27).
— Constant functions shall not themselves use constant functions in any context requiring a constant expression.
— A constant function may have default argument values (see 13.5.3), but any such default argument value shall be a constant expression.

Constant function calls are used to support the building of complex calculations of values at elaboration time (see 3.12). A constant function call is a function call of a constant function local to the calling module or from a package or `$unit` where the arguments to the function are all constant expressions (see 11.2.1).

Constant function calls are evaluated at elaboration time. Their execution has no effect on the initial values of the variables used either at simulation time or among multiple invocations of a function at elaboration time. In each of these cases, the variables are initialized as they would be for normal simulation.

The following example defines a function called `clogb2` that returns an integer with the value of the ceiling of the log base 2.

```
module ram_model (address, write, chip_select, data);
   parameter data_width = 8;
   parameter ram_depth = 256;
   localparam addr_width = clogb2(ram_depth);
   input [addr_width - 1:0] address;
   input write, chip_select;
   inout [data_width - 1:0] data;

   //define the clogb2 function
   function integer clogb2 (input [31:0] value);
      value = value - 1;
      for (clogb2 = 0; value > 0; clogb2 = clogb2 + 1)
         value = value >> 1;
   endfunction

   logic [data_width - 1:0] data_store[0:ram_depth - 1];
      //the rest of the ram model
endmodule: ram_model
```

An instance of this `ram_model` with parameters assigned is as follows:

```
ram_model #(32,421) ram_a0(a_addr,a_wr,a_cs,a_data);
```

### 13.4.4 Background processes spawned by function calls

Functions shall execute with no delay. Thus, a process calling a function shall return immediately. Statements that do not block shall be allowed inside a function; specifically, nonblocking assignments, event triggers, clocking drives, and **fork-join_none** constructs shall be allowed inside a function.

Calling a function that tries to schedule an event that cannot become active until after that function returns shall be allowed provided that the thread calling the function is created by an initial procedure, always procedure, or fork block from one of those procedures and in a context in which a side effect is allowed. Implementations shall issue an error either at compile time or run time when these provisions have not been met.

Within a function, a **fork-join_none** construct may contain any statements that are legal within a task. Examples of a legal and illegal usage of **fork-join_none** in a function are shown as follows:

```
class IntClass;
   int a;
endclass

IntClass address=new(), stack=new();

function automatic bit watch_for_zero( IntClass p );
   fork
      forever @p.a begin
         if ( p.a == 0 ) $display (“Unexpected zero”);
      end
   join_none
   return ( p.a == 0 );
endfunction

function bit start_check();
   return ( watch_for_zero( address ) | watch_for_zero( stack ) );
endfunction

bit y = watch_for_zero( stack );                // illegal

initial if ( start_check() ) $display ( “OK”); // legal

initial fork
   if (start_check() ) $display( “OK too”);     // legal
join_none
```

## 13.5 Subroutine calls and argument passing

Tasks and void functions are called as statements within procedural blocks (see 9.2). A nonvoid *function call* may be an operand within an expression.

The syntax for calling a subroutine as a statement is shown in Syntax 13-3:

```
subroutine_call_statement ::=                                  // from A.6.9
    subroutine_call ;
  | void ' ( function_subroutine_call ) ;
subroutine_call ::=                                            // from A.8.2
    tf_call
```

| system_tf_call
| method_call
| [ **std ::** ] randomize_call

tf_call[37] ::= ps_or_hierarchical_tf_identifier { attribute_instance } [ **(** list_of_arguments **)** ]

list_of_arguments ::=
[ expression ] { **,** [ expression ] } { **, .** identifier **(** [ expression ] **)** }
| **.** identifier **(** [ expression ] **)** { **, .** identifier **(** [ expression ] **)** }

ps_or_hierarchical_tf_identifier ::= *// from A.9.3*
[ package_scope ] tf_identifier
| hierarchical_tf_identifier

37) It shall be illegal to omit the parentheses in a tf_call unless the subroutine is a task, void function, or class method. If the subroutine is a nonvoid class function method, it shall be illegal to omit the parentheses if the call is directly recursive.

*Syntax 13-3—Task or function call syntax (excerpt from Annex A)*

If an argument in the subroutine is declared as an **input**, then the corresponding expression in the subroutine call can be any expression. The order of evaluation of the expressions in the argument list is undefined.

If the argument in the subroutine is declared as an **output** or an **inout**, then the corresponding expression in the subroutine call shall be restricted to an expression that is valid on the left-hand side of a procedural assignment (see 10.4).

The execution of the subroutine call shall pass input values from the expressions listed in the arguments of the call. Execution of the return from the subroutine shall pass values from the **output** and **inout** type arguments to the corresponding variables in the subroutine call.

SystemVerilog provides two means for passing arguments to tasks and functions: by value and by reference. Arguments can also be bound by name as well as by position. Subroutine arguments can also be given default values, allowing the call to the subroutine to not pass arguments.

### 13.5.1 Pass by value

Pass by value is the default mechanism for passing arguments to subroutines. This argument passing mechanism works by copying each argument into the subroutine area. If the subroutine is automatic, then the subroutine retains a local copy of the arguments in its stack. If the arguments are changed within the subroutine, the changes are not visible outside the subroutine. When the arguments are large, it can be undesirable to copy the arguments. Also, programs sometimes need to share a common piece of data that is not declared global.

For example, calling the following function copies 1000 bytes each time the call is made.

```
function automatic int crc( byte packet [1000:1] );
   for( int j= 1; j <= 1000; j++ ) begin
      crc ^= packet[j];
   end
endfunction
```

### 13.5.2 Pass by reference

Arguments passed by reference are not copied into the subroutine area, rather, a reference to the original argument is passed to the subroutine. The subroutine can then access the argument data via the reference. Arguments passed by reference shall be matched with equivalent data types (see 6.22.2). No casting shall be permitted. To indicate argument passing by reference, the argument declaration is preceded by the **ref** keyword. It shall be illegal to use argument passing by reference for subroutines with a lifetime of **static**. The general syntax is as follows:

```
subroutine( ref type argument );
```

For example, the preceding example can be written as follows:

```
function automatic int crc( ref byte packet [1000:1] );
   for( int j= 1; j <= 1000; j++ ) begin
      crc ^= packet[j];
   end
endfunction
```

As shown in the preceding example, no change other than addition of the **ref** keyword is needed. The compiler knows that `packet` is now addressed via a reference, but users do not need to make these references explicit either in the callee or at the point of the call. In other words, the call to either version of the `crc` function remains the same:

```
byte packet1[1000:1];
int k = crc( packet1 ); // pass by value or by reference: call is the same
```

When the argument is passed by reference, both the caller and the subroutine share the same representation of the argument; therefore, any changes made to the argument, within either the caller or the subroutine, shall be visible to each other. The semantics of assignments to variables passed by reference is that changes are seen outside the subroutine immediately (before the subroutine returns).

Only the following shall be legal to pass by reference:

- A variable,
- A class property,
- A member of an unpacked structure, or
- An element of an unpacked array.

Nets and selects into nets shall not be passed by reference.

Because a variable passed by reference may be an automatic variable, a **ref** argument shall not be used in any context forbidden for automatic variables.

Elements of dynamic arrays, queues, and associative arrays that are passed by reference may get removed from the array or the array may get resized before the called subroutine completes. The specific array element passed by reference shall continue to exist within the scope of the called subroutines until they complete. Changes made to the values of array elements by the called subroutine shall not be visible outside the scope of those subroutines if those array elements were removed from the array before the changes were made. These references shall be called *outdated references*.

The following operations on a variable-size array shall cause existing references to elements of that array to become outdated references:

- A dynamic array is resized with an implicit or explicit **new**`[]`.
- A dynamic array is deleted with the `delete()` method.

- The element of an associative array being referenced is deleted with the `delete()` method.
- The queue or dynamic array containing the referenced element is updated by assignment.
- The element of a queue being referenced is deleted by a queue method.

Passing an argument by reference is a unique argument-passing qualifier, different from **`input`**, **`output`**, or **`inout`**. Combining **`ref`** with any other directional qualifier shall be illegal. For example, the following declaration results in a compiler error:

```
task automatic incr( ref input int a );// incorrect: ref cannot be qualified
```

A **`ref`** argument is similar to an **`inout`** argument except that an **`inout`** argument is copied twice: once from the actual into the argument when the subroutine is called and once from the argument into the actual when the subroutine returns. Passing object handles is no exception and has similar semantics when passed as **`ref`** or **`inout`** arguments. Thus, a **`ref`** of an object handle allows changes to the object handle (for example, assigning a new object) in addition to modification of the contents of the object.

To protect arguments passed by reference from being modified by a subroutine, the **`const`** qualifier can be used together with **`ref`** to indicate that the argument, although passed by reference, is a read-only variable.

```
task automatic show ( const ref byte data [] );
   for ( int j = 0; j < data.size ; j++ )
      $display( data[j] ); // data can be read but not written
endtask
```

When the formal argument is declared as a **`const ref`**, the subroutine cannot alter the variable, and an attempt to do so shall generate a compiler error.

### 13.5.3 Default argument values

To handle common cases or allow for unused arguments, SystemVerilog allows a subroutine declaration to specify a default value for each singular argument.

The syntax to declare a default argument in a subroutine is as follows:

```
subroutine( [ direction ] [ type ] argument = default_expression);
```

The optional direction can be **`input`**, **`inout`**, **`output`**, or **`ref`**.

The *default_expression* is evaluated in the scope containing the subroutine declaration each time a call using the default is made. If the default is not used, the default expression is not evaluated. The use of defaults shall only be allowed with the ANSI style declarations.

When the subroutine is called, arguments with defaults can be omitted from the call, and the compiler shall insert their corresponding values. Unspecified (or empty) arguments can be used as placeholders for default arguments. If an unspecified argument is used for an argument that does not have a default, a compiler error shall be issued.

```
task read(int j = 0, int k, int data = 1 );
...
endtask
```

This example declares a task `read()` with two default arguments, `j` and `data`. The task can then be called using various default arguments:

```
read( , 5 );         // is equivalent to  read( 0, 5, 1 );
read( 2, 5 );        // is equivalent to  read( 2, 5, 1 );
read( , 5, );        // is equivalent to  read( 0, 5, 1 );
read( , 5, 7 );      // is equivalent to  read( 0, 5, 7 );
read( 1, 5, 2 );     // is equivalent to  read( 1, 5, 2 );
read( );             // error; k has no default value
read( 1, , 7 );      // error; k has no default value
```

The following example shows an output argument with a default expression:

```
module m;
   logic a, w;

   task t1 (output o = a) ; // default binds to m.a
      ...
   endtask :t1

   task t2 (output o = b) ; // illegal, b cannot be resolved
      ...
   endtask :t2

   task t3 (inout io = w) ; // default binds to m.w
      ...
   endtask :t1
endmodule :m

module n;
   logic a;

   initial begin
      m.t1();  // same as m.t1(m.a), not m.t1(n.a);
               // at end of task, value of t1.o is copied to m.a
      m.t3();  // same as m.t3(m.w)
               // value of m.w is copied to t3.io at start of task and
               // value of t3.io is copied to m.w at end of task
   end
endmodule :n
```

### 13.5.4 Argument binding by name

SystemVerilog allows arguments to tasks and functions to be bound by name as well as by position. This allows specifying nonconsecutive default arguments and easily specifying the argument to be passed at the call. For example:

```
   function int fun( int j = 1, string s = "no" );
      ...
   endfunction
```

The `fun` function can be called as follows:

```
fun( .j(2), .s("yes") );      // fun( 2, "yes" );
fun( .s("yes") );             // fun( 1, "yes" );
fun( , "yes" );               // fun( 1, "yes" );
fun( .j(2) );                 // fun( 2, "no" );
fun( .s("yes"), .j(2) );      // fun( 2, "yes" );
fun( .s(), .j() );            // fun( 1, "no" );
fun( 2 );                     // fun( 2, "no" );
fun( );                       // fun( 1, "no" );
```

If the arguments have default values, they are treated like parameters to module instances. If the arguments do not have a default, then they shall be given, or the compiler shall issue an error.

If both positional and named arguments are specified in a single subroutine call, then all the positional arguments shall come before the named arguments. Then, using the same example as above:

```
fun( .s("yes"), 2 );        // illegal
fun( 2, .s("yes") );        // OK
```

### 13.5.5 Optional argument list

When a void function or class function method specifies no arguments, the empty parenthesis, `()`, following the subroutine name shall be optional. This is also true for tasks, void functions, and class methods that require arguments, when all arguments have defaults specified. It shall be illegal to omit the parenthesis in a directly recursive nonvoid class function method call that is not hierarchically qualified.

## 13.6 Import and export functions

SystemVerilog provides a direct programming interface (DPI) that allows importing foreign language subroutines, such as C functions, into SystemVerilog. An imported subroutine is called in the same way as a SystemVerilog subroutine. SystemVerilog tasks and functions can also be exported to a foreign language. See Clause 35 for details on the DPI.

## 13.7 Task and function names

Task and function names are resolved following slightly different rules than other references. Even when used as a simple name, a task or function name follows a modified form of the upwards hierarchical name resolution rules. This means that “forward” references to a task or function defined later in the same or an enclosing scope can be resolved. See 23.8.1 for the rules that govern task and function name resolution.

## 13.8 Parameterized tasks and functions

SystemVerilog provides a way to create parameterized tasks and functions, also known as *parameterized subroutines.* A parameterized subroutine allows the user to generically specify or define an implementation. When using that subroutine one may provide the parameters that fully define its behavior. This allows for only one definition to be written and maintained instead of multiple subroutines with different array sizes, data types, and variable widths.

The way to implement parameterized subroutines is through the use of static methods in parameterized classes (see 8.10 and 8.25). The following generic encoder and decoder example shows how to use static class methods along with class parameterization to implement parameterized subroutines. The example has one class with two subroutines that, in this case, share parameterization. The class may be declared virtual in order to prevent object construction and enforce the strict static usage of the method.

```
virtual class C#(parameter DECODE_W, parameter ENCODE_W = $clog2(DECODE_W));
   static function logic [ENCODE_W-1:0] ENCODER_f
         (input logic [DECODE_W-1:0] DecodeIn);
      ENCODER_f = '0;
      for (int i=0; i<DECODE_W; i++) begin
         if(DecodeIn[i]) begin
            ENCODER_f = i[ENCODE_W-1:0];
            break;
         end
      end
      endfunction

   static function logic [DECODE_W-1:0] DECODER_f
         (input logic [ENCODE_W-1:0] EncodeIn);
      DECODER_f = '0;
      DECODER_f[EncodeIn] = 1'b1;
   endfunction
endclass
```

Class `C` contains two static subroutines, `ENCODER_f` and `DECODER_f`. Each subroutine is parameterized by reusing the class parameters `DECODE_W` and `ENCODE_W`. The default value of parameter `ENCODE_W` is determined by using the system function `$clog2` (see 20.8.1). These parameters are used within the subroutines to specify the size of the encoder and the size of the decoder.

```
module top ();
   logic [7:0] encoder_in;
   logic [2:0] encoder_out;
   logic [1:0] decoder_in;
   logic [3:0] decoder_out;

   // Encoder and Decoder Input Assignments
   assign encoder_in = 8'b0100_0000;
   assign decoder_in = 2'b11;

   // Encoder and Decoder Function calls
   assign encoder_out = C#(8)::ENCODER_f(encoder_in);
   assign decoder_out = C#(4)::DECODER_f(decoder_in);

   initial begin
      #50;
      $display("Encoder input = %b Encoder output = %b\n",
         encoder_in, encoder_out );
      $display("Decoder input = %b Decoder output = %b\n",
         decoder_in, decoder_out );
   end
endmodule
```

The top level module first defines some intermediate variables used in this example, and then assigns constant values to the encoder and decoder inputs. The subroutine call of the generic encoder, `ENCODER_f`, uses the specialized class parameter value of 8 that represents the decoder width value for that specific instance of the encoder while at the same time passing the input encoded value, `encoder_in`. This expression uses the static class scope resolution operator `::` (see 8.23) to access the encoder subroutine. The expression is assigned to an output variable to hold the result of the operation. The subroutine call for the generic decoder, `DECODER_f`, is similar, using the parameter value of 4.

# 14. Clocking blocks

## 14.1 General

This clause describes the following:

- Clocking block declarations
- Input and output skews
- Clocking block signal events
- Cycle delays
- Synchronous events
- Synchronous drives

## 14.2 Overview

Module port connections and interfaces can specify the signals or nets through which a testbench communicates with a device under test (DUT). However, such specification does not explicitly denote any timing disciplines, synchronization requirements, or clocking paradigms.

The *clocking block* construct identifies clock signals and captures the timing and synchronization requirements of the blocks being modeled. A clocking block is defined between the keywords **`clocking`** and **`endclocking`**.

A clocking block assembles signals that are synchronous to a particular clock and makes their timing explicit. The clocking block is a key element in a cycle-based methodology, which enables users to write testbenches at a higher level of abstraction. Rather than focusing on signals and transitions in time, the test can be defined in terms of cycles and transactions. Depending on the environment, a testbench can contain one or more clocking blocks, each containing its own clock plus an arbitrary number of signals.

The clocking block separates the timing and synchronization details from the structural, functional, and procedural elements of a testbench. Thus, the timing for sampling and driving clocking block signals is implicit and relative to the clocking block’s clock. This enables a set of key operations to be written very succinctly, without explicitly using clocks or specifying timing. These operations are as follows:

- Synchronous events
- Input sampling
- Synchronous drives

## 14.3 Clocking block declaration

The syntax for the clocking block is as follows in Syntax 14-1.

```
clocking_declaration ::=                                                        // from A.6.11
    [ default ] clocking [ clocking_identifier ] clocking_event ;
        { clocking_item }
    endclocking [ : clocking_identifier ]
  | global clocking [ clocking_identifier ] clocking_event ; endclocking [ : clocking_identifier ]
clocking_event ::=
    @ identifier
  | @ ( event_expression )
```

```
clocking_item ::=
      default default_skew ;
    | clocking_direction list_of_clocking_decl_assign ;
    | { attribute_instance } assertion_item_declaration
default_skew ::=
      input clocking_skew
    | output clocking_skew
    | input clocking_skew output clocking_skew
clocking_direction ::=
      input [ clocking_skew ]
    | output [ clocking_skew ]
    | input [ clocking_skew ] output [ clocking_skew ]
    | inout
list_of_clocking_decl_assign ::= clocking_decl_assign { , clocking_decl_assign }
clocking_decl_assign ::= signal_identifier [ = expression ]
clocking_skew ::=
      edge_identifier [ delay_control ]
    | delay_control
edge_identifier ::= posedge | negedge | edge                          // from A.7.4
delay_control ::=                                                     // from A.6.5
      # delay_value
    | # ( mintypmax_expression )
```

Syntax 14-1—Clocking block syntax (excerpt from Annex A)

The *delay_control* shall be either a time literal or a constant expression that evaluates to a positive integer value.

The *clocking_identifier* specifies the name of the clocking block being declared. Only default clocking blocks may be unnamed. Declarations in unnamed clocking blocks may not be referenced.

The *signal_identifier* specifies a signal (a net or variable) in the scope enclosing the clocking block declaration, and defines a *clockvar* in the clocking block. The specified signal is called a *clocking signal*. Unless a hierarchical expression is used (see 14.5), both the clocking signal and the clockvar names shall be the same. It shall be illegal for a clocking signal to designate a variable restricted to a procedural block (see 6.21).

The *clocking_event* designates a particular event to act as the clock for the clocking block. The timing used to drive and sample all other signals specified in a given clocking block is governed by its clocking event. See 14.13 and 14.16 for details on the precise timing semantics of sampling and driving clocking signals.

It shall be illegal to write to any *clockvar* whose *clocking_direction* is **input**.

It shall be illegal to read the value of any *clockvar* whose *clocking_direction* is **output**.

A *clockvar* whose *clocking_direction* is **inout** shall behave as if it were two *clockvar*s, one **input** and one **output**, having the same name and the same *clocking_signal*. Reading the value of such an **inout** *clockvar* shall be equivalent to reading the corresponding **input** *clockvar*. Writing to such an **inout** *clockvar* shall be equivalent to writing to the corresponding **output** *clockvar*.

The *clocking_skew* determines how many time units away from the clock event a signal is to be sampled or driven. Input skews are implicitly negative, that is, they always refer to a time before the clock, whereas output skews always refer to a time after the clock (see 14.4). When the clocking event specifies a simple edge, instead of a number, the skew can be specified as the specific edge of the signal. A single skew can be specified for the entire block by using a **default** clocking item.

```
clocking ck1 @(posedge clk);
   default input #1step output negedge; // legal
   // outputs driven on the negedge clk
   input ... ;
   output ... ;
endclocking

clocking ck2 @(clk); // no edge specified!
   default input #1step output negedge; // legal
   input ... ;
   output ... ;
endclocking
```

The *expression* assigned to the *clockvar* specifies that the signal to be associated with the clocking block is associated with the specified hierarchical expression. For example, a cross-module reference can be used instead of a local port. See 14.5 for more information.

*Example:*

```
clocking bus @(posedge clock1);
   default input #10ns output #2ns;
   input data, ready, enable = top.mem1.enable;
   output negedge ack;
   input #1step addr;
endclocking
```

In the preceding example, the first line declares a clocking block called `bus` that is to be clocked on the positive edge of the signal `clock1`. The second line specifies that by default all signals in the clocking block shall use a `10ns` input skew and a `2ns` output skew. The next line adds three input signals to the clocking block: `data`, `ready`, and `enable`; the last signal refers to the hierarchical signal `top.mem1.enable`. The fourth line adds the signal `ack` to the clocking block and overrides the default output skew so that `ack` is driven on the negative edge of the clock. The last line adds the signal `addr` and overrides the default input skew so that `addr` is sampled one step before the positive edge of the clock.

Unless otherwise specified, the default **`input`** skew is `1step` and the default **`output`** skew is **`0`**. A step is a special time unit whose value is defined in 3.14.3. A `1step` input skew allows input signals to sample their steady-state values in the time step immediately before the clock event (i.e., in the preceding Postponed region).

## 14.4 Input and output skews

Input (or inout) signals are sampled at the designated clock event. If an input skew is specified, then the signal is sampled at *skew* time units *before* the clock event. Similarly, output (or inout) signals are driven *skew* simulation time units *after* the corresponding clock event. Figure 14-1 shows the basic sample and drive timing for a positive edge clock.

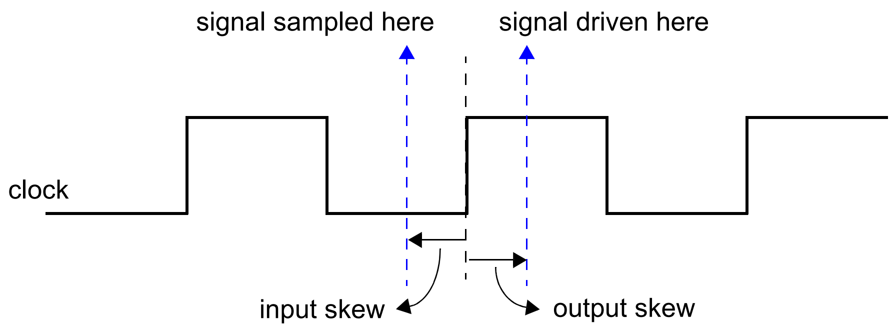


**Figure 14-1—Sample and drive times including skew with respect to the positive edge of the clock**

A skew shall be a constant expression and can be specified as a parameter. If the skew does not specify a time unit, the current time unit is used. If a number is used, the skew is interpreted using the timescale of the current scope.

```
clocking dram @(clk);
   input #1ps address;
   input #5 output #6 data;
endclocking
```

An input skew of `1step` indicates that the signal is to be sampled at the end of the previous time step. In other words, the value sampled is always the signal's last value immediately before the corresponding clock edge.

NOTE—A clocking block does not eliminate potential races when an event control outside a program block is sensitive to the same clock as the clocking block and a statement after the event control attempts to read a member of the clocking block. The race is between reading the old sampled value and the new sampled value.

Inputs with explicit #0 skew shall be sampled at the same time as their corresponding clocking event, but to avoid races, they are sampled in the Observed region. Likewise, clocking block outputs with no skew (or explicit #0 skew) shall be driven at the same time as their specified clocking event, in the Re-NBA region.

Skews are declarative constructs; thus, they are semantically very different from the syntactically similar procedural delay statement. In particular, an explicit #0 skew does not suspend any process, nor does it execute or sample values in the Inactive region.

## 14.5 Hierarchical expressions

Any signal in a clocking block can be associated with an arbitrary hierarchical expression. As described in 14.3, a hierarchical expression is introduced by appending an equal sign (=) followed by the hierarchical expression:

```
clocking cd1 @(posedge phi1);
      input #1step state = top.cpu1.state;
endclocking
```

However, hierarchical expressions are not limited to simple names or signals in other scopes. They can be used to declare slices and concatenations (or combinations thereof) of signals in other scopes or in the current scope.

```
clocking mem @(clock);
   input instruction = { opcode, regA, regB[3:1] };
endclocking
```

In a clocking block, any expression assigned to a signal in its declaration shall be an expression that would be legal in a port connection to a port of appropriate direction. Any expression assigned to a signal in a clocking **input** or **inout** declaration shall be an expression that would be legal for connection to a module's input port. Any expression assigned to a signal in a clocking **output** or **inout** declaration shall be an expression that would be legal for connection to a module's output port.

A clocking **inout** declaration is not an inout port; it is shorthand for two clocking declarations, one input and one output, with the same signal. Consequently, such a signal must meet the requirements for both a clocking input and a clocking output, but it is not required to meet the stricter requirements for connection to a module's inout port. In particular, it is acceptable to specify a variable as a clocking inout signal.

## 14.6 Signals in multiple clocking blocks

The same signals—clock, inputs, inouts, or outputs—can appear in more than one clocking block. When clocking blocks use the same clock (or clocking expression), they shall share the same synchronization event, in the same manner as several latches can be controlled by the same clock. Input semantics is described in 14.13, and output semantics is described in 14.16.

## 14.7 Clocking block scope and lifetime

A clocking block is both a declaration and an instance of that declaration. A separate instantiation step is not necessary. Instead, one copy is created for each instance of the block containing the declaration (like an always procedure). Once declared, the clocking signals are available via the clocking block name and the dot (`.`) operator:

```
dom.sig  // signal sig in clocking dom
```

Multiple clocking blocks cannot be nested. They cannot be declared inside functions, tasks, or packages or outside all declarations in a compilation unit. A clocking block can only be declared inside a module, interface, checker, or program (see Clause 24).

A clocking block has static lifetime and scope local to its enclosing module, interface, or program.

## 14.8 Multiple clocking blocks example

In this example, a simple test program includes two clocking blocks. The program construct used in this example is discussed in Clause 24.

```
program test(  input phi1, input [15:0] data, output logic write,
               input phi2, inout [8:1] cmd, input enable
            );
   reg [8:1] cmd_reg;

   clocking cd1 @(posedge phi1);
      input data;
      output write;
      input state = top.cpu1.state;
   endclocking
```

```
      clocking cd2 @(posedge phi2);
         input #2 output #4ps cmd;
         input enable;
      endclocking

      initial begin
         // program begins here
         ...
         // user can access cd1.data , cd2.cmd , etc…
      end
      assign cmd = enable ? cmd_reg: 'x;
   endprogram
```

The test program can be instantiated and connected to a DUT (`cpu` and `mem`).

```
   module top;
      logic phi1, phi2;
      wire [8:1] cmd; // cannot be logic (two bidirectional drivers)
      logic [15:0] data;

      test main (phi1, data, write, phi2, cmd, enable);
      cpu cpu1 (phi1, data, write);
      mem mem1 (phi2, cmd, enable);
   endmodule
```

## 14.9 Interfaces and clocking blocks

A clocking block encapsulates a set of signals that share a common clock; therefore, specifying a clocking block using a SystemVerilog **`interface`** (see Clause 25) can significantly reduce the amount of code needed to connect the testbench. Furthermore, because the signal directions in the clocking block within the testbench are with respect to the testbench and not the design under test, a **`modport`** declaration (see 25.5) can appropriately describe either direction. A testbench program can be contained within a program, and its ports can be interfaces that correspond to the signals declared in each clocking block. The interface's wires would have the same direction as specified in the clocking block when viewed from the testbench side (i.e., **`modport`** `test`) and reversed when viewed from the DUT (i.e., **`modport`** `dut`).

For example, the previous example could be rewritten using interfaces as follows:

```
   interface bus_A (input clk);
      logic [15:0] data;
      logic write;
      modport test (input data, output write);
      modport dut (output data, input write);
   endinterface

   interface bus_B (input clk);
      logic [8:1] cmd;
      logic enable;
      modport test (input enable);
      modport dut (output enable);
   endinterface

   program test( bus_A.test a, bus_B.test b );

      clocking cd1 @(posedge a.clk);
         input data = a.data;
         output write = a.write;
         inout state = top.cpu1.state;
      endclocking

      clocking cd2 @(posedge b.clk);
         input #2 output #4ps cmd = b.cmd;
         input en = b.enable;
      endclocking

      initial begin
         // program begins here
         ...
         // user can access cd1.data, cd1.write, cd1.state,
         // cd2.cmd, and cd2.en
      end
   endprogram
```

The test module can be instantiated and connected as before:

```
module top;
   logic phi1, phi2;

   bus_A a (phi1);
   bus_B b (phi2);

   test main (a, b);
   cpu cpu1 (a);
   mem mem1 (b);
endmodule
```

## 14.10 Clocking block events

The clocking event of a clocking block is available directly by using the clocking block name, regardless of the actual clocking event used to declare the clocking block.

For example:

```
clocking dram @(posedge phi1);
   inout data;
   output negedge #1 address;
endclocking
```

The clocking event of the `dram` clocking block can be used to wait for that particular event:

```
@(dram);
```

The preceding statement is equivalent to `@(posedge phi1)`.

## 14.11 Cycle delay: ##

The `##` operator can be used to delay execution by a specified number of clocking events or clock cycles.

The syntax for the cycle delay statement is as follows in Syntax 14-2.

---

procedural_timing_control_statement ::= *// from A.6.5*
    procedural_timing_control statement_or_null

procedural_timing_control ::=
    ...
  | cycle_delay

cycle_delay ::= *// from A.6.11*
    **##** integral_number
  | **##** identifier
  | **##** **(** expression **)**

---

*Syntax 14-2—Cycle delay syntax (excerpt from Annex A)*

The *expression* can be any SystemVerilog expression that evaluates to a positive integer value.

What constitutes a cycle is determined by the default clocking in effect (see 14.12). If no default clocking has been specified for the current module, interface, checker, or program, then the compiler shall issue an error.

*Example:*

```
    ##5;        // wait 5 cycles (clocking events) using the default clocking

    ##(j + 1); // wait j+1 cycles (clocking events) using the default clocking
```

The cycle delay timing control shall wait for the specified number of clocking events. This implies that for a `##1` statement that is executed at a simulation time that is not coincident with the associated clocking event, the calling process shall be delayed a fraction of the associated clock cycle.

Cycle delays of `##0` are treated specially. If a clocking event has not yet occurred in the current time step, a `##0` cycle delay shall suspend the calling process until the clocking event occurs. When a process executes a `##0` cycle delay and the associated clocking event has already occurred in the current time step, the process shall continue execution without suspension. When used on the right-hand side of a synchronous drive, a `##0` cycle delay shall have no effect, as if it were not present.

Cycle delay timing controls shall not be legal for use in intra-assignment delays in either blocking or nonblocking assignment statements.

## 14.12 Default clocking

One clocking block can be specified as the default for all cycle delay operations within a given module, interface, program, or checker.

The syntax for default clocking specification statement is as follows in Syntax 14-3.

---

module_or_generate_item_declaration ::= *// from A.1.4*
    ...
  | **default clocking** clocking_identifier **;**
  ...

checker_or_generate_item_declaration ::= *// from A.1.8*
    ...

```
  | default clocking clocking_identifier ;
   ...
clocking_declaration ::=                                          // from A.6.11
     [ default ] clocking [ clocking_identifier ] clocking_event ;
         { clocking_item }
     endclocking [ : clocking_identifier ]
  ...
```

*Syntax 14-3—Default clocking syntax (excerpt from Annex A)*

The *clocking_identifier* shall be the name of a clocking block.

Only one default clocking can be specified in a module, interface, program, or checker. Specifying a default clocking more than once in the same module, interface, program, or checker shall result in a compiler error.

A default clocking is valid only within the scope containing the default clocking specification statement. This scope includes the module, interface, program, or checker that contains the declaration as well as any nested modules, interfaces, or checkers. It does not include instantiated modules, interfaces, or checkers.

*Example 1*: Declaring a clocking as the default:

```
program test(input logic clk, input logic [15:0] data);
   default clocking bus @(posedge clk);
      inout data;
   endclocking

   initial begin
      ## 5;
      if (bus.data == 10)
         ## 1;
      else
         ...
   end
endprogram
```

*Example 2*: Assigning an existing clocking to be the default:

```
module processor ...
   clocking busA @(posedge clk1); ... endclocking
   clocking busB @(negedge clk2); ... endclocking
   module cpu( interface y );
      default clocking busA ;
      initial begin
         ## 5; // use busA => (posedge clk1)
         ...
      end
   endmodule
endmodule
```

## 14.13 Input sampling

All clocking block inputs (input or inout) are sampled at the corresponding clocking event. If the input skew is not an explicit `#0`, then the value sampled corresponds to the signal value at the Postponed region of the time step skew time units prior to the clocking event (see Figure 14-1 in 14.4). If the input skew is an explicit `#0`, then the value sampled corresponds to the signal value in the Observed region. In this case, the newly sampled values shall be available for reading at the end of the Observed region processing. If upon processing the Reactive region, the simulation must process Active events without advancing time (thereby executing the Observed region more than once), clocking inputs sampled with `#0` skew shall not be resampled unless a new clocking event occurs in the active region set.

NOTE—When the clocking event is triggered by the execution of a program, there is a potential race between the update of a clocking block input value and programs that read that value without synchronizing with the corresponding clocking event. This race does not exist when the clocking block event is triggered from within a module.

Upon processing its specified clocking event, a clocking block shall update its sampled values before triggering the event associated with the clocking block name. This event shall be triggered in the Observed region. Thus, a process that waits for the clocking block itself is guaranteed to read the updated sampled values, regardless of the scheduling region in which either the waiting or the triggering processes execute. For example:

```
clocking cb @(negedge clk);
   input v;
endclocking

always @(cb) $display(cb.v);

always @(negedge clk) $display(cb.v);
```

The preceding first **`always`** procedure is guaranteed to display the updated sampled value of signal `v`. In contrast, the second always exhibits a potential race and may display the old or the newly updated sampled value.

When an **`input`** or **`inout`** clockvar appears in any expression its value is the signal's sampled value; that is, the value that the clocking block sampled at the most recent clocking event.

When the same signal is an input to multiple clocking blocks, the semantics is straightforward; each clocking block samples the corresponding signal with its own clocking event.

## 14.14 Global clocking

A clocking block may be declared as the global clocking for all or part of the design hierarchy. The main purpose of global clocking is to specify which clocking event in simulation corresponds to the primary system clock used in formal verification (see F.3.1). Such a specification may be done for a whole design, or separately for different subsystems in a design, when there are multiple clocks and building a single global clocking event is challenging.

The syntax for the global clocking declaration is as follows in Syntax 14-4.

```
clocking_declaration ::=                                                    // from A.6.11
   ...
   | global clocking [ clocking_identifier ] clocking_event ; endclocking [ : clocking_identifier ]
```

*Syntax 14-4—Global clocking syntax (excerpt from Annex A)*

Global clocking may be declared in a module, an interface, a checker, or a program. A given module, interface, checker, or program shall contain at most one global clocking declaration. Global clocking shall not be declared in a generate block.

Although more than one global clocking declaration may appear in different parts of the design hierarchy, at most one global clocking declaration is effective at each point in the elaborated design hierarchy. The `$global_clock` system function shall be used to explicitly refer to the event expression in the effective global clocking declaration. The effective global clocking declaration for a specific reference is determined using the following hierarchical lookup rules, which iteratively check the design hierarchy to find the global clocking declaration closest to the point of reference.

a) Look for a global clocking declaration in the enclosing module, interface, checker, or program instance scope. If found, the lookup terminates with the result being the event expression of that global clocking declaration. If not found and the current scope is a top-level hierarchy block (see 3.11), the lookup terminates and shall result in an error. Otherwise, proceed to step b).

b) Look for a global clocking declaration in the parent module, interface, or checker instance scope of the enclosing instantiation. If found, the lookup terminates with the result being the event expression of that global clocking declaration. Otherwise, continue up the hierarchy until a global clocking declaration is found or a top-level hierarchy block is reached. If no global clocking declaration is found and a top-level hierarchy block is reached, the lookup terminates and shall result in an error.

When global clocking is referenced in a sequence declaration, a property declaration, or as an actual argument to a named sequence instance, a named property instance, or a checker instance, the point of reference shall be considered after the application of the rewriting algorithms defined in F.4. As a result, when a property or a sequence declaration is instantiated in an assertion statement, the hierarchical lookup rules described previously shall be applied from the point of the assertion statement appearance in the source description, not from the point of the sequence or the property declaration. Similarly, the lookup rules shall be applied after the substitution of the actual argument in place of the corresponding formal argument inside the checker body.

The following is an example of a **`global clocking`** declaration:

```
module top;
   logic clk1, clk2;
   global clocking sys @(clk1 or clk2); endclocking
   // ...
endmodule
```

In this example, `sys` is declared as the global clocking event and is defined to occur if, and only if, there is a change of either of two signals, `clk1` and `clk2`. Specification of the name `sys` in the global clocking declaration is optional since the global clocking event may be referenced by `$global_clock` .

In the following example the design hierarchy contains two global clocking declarations. The call to `$global_clock` in `top.sub1.common` resolves to the clocking event `top.sub1.sub_sys1`. The call to `$global_clock` in `top.sub2.common` resolves to the clocking event `top.sub2.sub_sys2`.

```
module top;
   subsystem1 sub1();
   subsystem2 sub2();
endmodule

module subsystem1;
   logic subclk1;
   global clocking sub_sys1 @(subclk1); endclocking
   // ...
   common_sub common();
endmodule

module subsystem2;
   logic subclk2;
   global clocking sub_sys2 @(subclk2); endclocking
   // ...
   common_sub common();
endmodule

module common_sub;
   always @($global_clock) begin
      // ...
   end
endmodule
```

In the following example the property `p` is declared in a module containing a global clocking declaration. However, that global clocking declaration is not effective where the property `p` is instantiated. Similar to the previous example, the call to `$global_clock` in `top.sub1.checks` resolves to the clocking event `top.sub1.sub_sys1`, and the call to `$global_clock` in `top.sub2.checks` resolves to the clocking event `top.sub2.sub_sys2`.

```
module top;
   subsystem1 sub1();
   subsystem2 sub2();
endmodule

module subsystem1;
   logic subclk1, req, ack;
   global clocking sub_sys1 @(subclk1); endclocking
   // ...
   common_checks checks(req, ack);
endmodule

module subsystem2;
   logic subclk2, req, ack;
   global clocking sub_sys2 @(subclk2); endclocking
   // ...
   common_checks checks(req, ack);
endmodule

module another_module;
   logic another_clk;
   global clocking another_clocking @(another_clk); endclocking
   // ...
   property p(req, ack);
      @($global_clock) req |=> ack;
   endproperty
endmodule

checker common_checks(logic req, logic ack);
   assert property (another_module.p(req, ack));
endchecker
```

In the following example, `$global_clock` is used in a task. The call to `$global_clock` in the task `another_module.t` resolves to the clocking event `another_module.another_clocking`.

```
module top;
   subsystem1 sub1();
   subsystem2 sub2();
endmodule

module subsystem1;
```

```
   logic subclk1, req, ack;
   global clocking sub_sys1 @(subclk1); endclocking
   // ...
   always another_module.t(req, ack);
endmodule

module subsystem2;
   logic subclk2, req, ack;
   global clocking sub_sys2 @(subclk2); endclocking
   // ...
   always another_module.t(req, ack);
endmodule

module another_module;
   logic another_clk;
   global clocking another_clocking @(another_clk); endclocking

   task t(input req, input ack);
      @($global_clock);
      // ...
   endtask
endmodule
```

The following example demonstrates the usage of `$global_clock` in checker arguments. The resolution of the calls to `$global_clock` is performed after the substitution of the actual checker arguments in place of the corresponding formal arguments, and flattening of the properties in the assertion statements. All calls to `$global_clock` in this example refer to the clocking event `top.check.checker_clocking`.

```
module top;
   logic a, b, c, clk;
   global clocking top_clocking @(clk); endclocking
   // ...

   property p1(req, ack);
      @($global_clock) req |=> ack;
   endproperty

   property p2(req, ack, interrupt);
      @($global_clock) accept_on(interrupt) p1(req, ack);
   endproperty

   my_checker check(
      p2(a, b, c),
      @$global_clock a[*1:$] ##1 b);
endmodule

checker my_checker(property p, sequence s);
   logic checker_clk;
   global clocking checker_clocking @(checker_clk); endclocking
   // ...
   assert property (p);
   cover property (s);
endchecker
```

NOTE—This is an area of backward incompatibility between this standard and 14.14 of IEEE Std 1800-2009. In the 2009 definition, only one global clocking declaration was allowed in the elaborated design description; it could be specified in any module, interface, or checker and referenced everywhere in the description. A design conforming to IEEE Std 1800-2009 could have a global clocking declaration defined in a non-top-level module and use `$global_clock` outside the subhierarchy of that module. Such a design shall not conform to this standard.

## 14.15 Synchronous events

Explicit synchronization is done via the event control operator, `@`, which allows a process to wait for a particular signal value change or a clocking event (see 14.10).

The syntax for the synchronization operator is given in 9.4.2.

The expression used with the event control can denote clocking block input (**`input`** or **`inout`**) or a slice thereof. Slices can include dynamic indices, which are evaluated once when the `@` expression executes.

Following are some examples of synchronization statements:

— Wait for the next change of signal `ack_1` of clocking block `ram_bus`

```
@(ram_bus.ack_l);
```

— Wait for the next clocking event in clocking block `ram_bus`

```
@(ram_bus);
```

— Wait for the positive edge of the signal `ram_bus.enable`

```
@(posedge ram_bus.enable);
```

— Wait for the falling edge of the specified 1-bit slice `dom.sign[a]`

```
@(negedge dom.sign[a]);
```

NOTE—The index `a` is evaluated at run time.

— Wait for either the next positive edge of `dom.sig1` or the next change of `dom.sig2`, whichever happens first

```
@(posedge dom.sig1 or dom.sig2);
```

— Wait for either the negative edge of `dom.sig1` or the positive edge of `dom.sig2`, whichever happens first

```
@(negedge dom.sig1 or posedge dom.sig2);
```

— Wait for the edge (either the negative edge or the positive edge, whichever happens first) of `dom.sig1`.

```
@(edge dom.sig1);
```

Or equivalently

```
@(negedge dom.sig1 or posedge dom.sig1);
```

The values used by the synchronization event control are the synchronous values, that is, the values sampled at the corresponding clocking event.

## 14.16 Synchronous drives

Clocking block outputs (**`output`** or **`inout`**) are used to drive values onto their corresponding signals, but at a specified time. In other words, the corresponding signal changes value at the indicated clocking event as modified by the output skew.

For zero skew clocking block outputs with no cycle delay, synchronous drives shall schedule new values in the Re-NBA region of the time step corresponding to the clocking event. For clocking block outputs with nonzero skew, or drives with nonzero cycle delay, the corresponding signal shall be scheduled to change value in the Re-NBA region of a future time step.

For each clocking block output whose target is a net, a driver on that net shall be created. The driver so created shall have (**strong1**, **strong0**) drive strength and shall be updated as if by a continuous assignment from a variable inside the clocking block. This implicit variable, which is invisible to user code, shall be updated in the Re-NBA region by the execution of a synchronous drive to the corresponding clockvar. The created driver shall be initialized to `'z`; hence, the driver has no influence on its target net until a synchronous drive is performed to the corresponding clockvar.

The syntax to specify a synchronous drive is similar to an assignment and is shown in Syntax 14-5.

---

statement ::= [ block_identifier **:** ] { attribute_instance } statement_item *// from A.6.4*

statement_item ::=
 ...
 | clocking_drive **;**

clocking_drive ::= *// from A.6.11*
 clockvar_expression **<=** [ cycle_delay ] expression

cycle_delay ::= **##** expression

clockvar ::= hierarchical_identifier

clockvar_expression ::= clockvar select

---

*Syntax 14-5—Synchronous drive syntax (excerpt from Annex A)*

The *clockvar_expression* is a bit-select, slice, or the entire clocking block output whose corresponding signal is to be driven (concatenation is not allowed):

```
dom.sig          // entire clockvar

dom.sig[2]       // bit-select

dom.sig[8:2]     // slice
```

The *expression* (in the *clocking_drive* production) can be any valid expression that is assignment compatible with the type of the corresponding signal.

The optional *cycle_delay* construct, appearing on the right-hand side of a *clocking_drive* statement, is syntactically similar to an intra-assignment delay in a nonblocking assignment. Like a nonblocking intra-assignment delay, it shall not cause execution of the statement to block. The right-hand side expression shall be evaluated immediately even when a *cycle_delay* is present. However, updating of the target signal shall be postponed for the specified number of cycles of the target clockvar's clocking block, plus any clocking output skew specified for that clockvar.

No other form of intra-assignment delay syntax shall be legal in a synchronous drive to a clockvar.

A procedural cycle delay, as described in 14.11, can be used as a prefix to any procedural statement. If a procedural cycle delay is used as a prefix to a synchronous drive, it shall block for its specified number of cycles of the default clocking exactly as it would if used as a prefix to any other procedural statement.

*Examples:*

```
bus.data[3:0] <= 4'h5;  // drive data in Re-NBA region of the current cycle

##1 bus.data <= 8'hz;   // wait 1 default clocking cycle, then drive data

##2; bus.data <= 2;     // wait 2 default clocking cycles, then drive data

bus.data <= ##2 r;      // remember the value of r and then drive
                        // data 2 (bus) cycles later

bus.data <= #4 r;       // error: regular intra-assignment delay not allowed
                        // in synchronous drives
```

Regardless of whether the synchronous drive takes effect on the current clocking event or at some future clocking event as a result of a *cycle_delay*, the corresponding signal shall be updated at a time after that clocking event as specified by the output skew.

It is possible for a drive statement to execute at a time that is not coincident with its clocking event. Such drive statements shall execute without blocking, but shall perform their drive action as if they had executed at the time of the next clocking event. The expression on the right-hand side of the drive statement shall be evaluated immediately, but the processing of the drive is delayed until the time of the next clocking event.

For example:

```
default clocking cb @(posedge clk);  // Assume clk has a period of #10 units
   output v;
endclocking

initial begin
   #3 cb.v <= expr1;  // Matures in cycle 1; equivalent to ##1 cb.v <= expr1
end
```

It shall be an error to write to a clockvar except by using the synchronous drive syntax described in this subclause. Thus, it is illegal to use any continuous assignment, force statement, or procedural continuous assignment to write to a clockvar.

### 14.16.1 Drives and nonblocking assignments

Although synchronous drives use the same operator syntax as nonblocking variable assignments, they are not the same. One difference is that synchronous drives do not support intra-assignment delay syntax. A key feature of synchronous drives to inout clockvars is that a drive does not change the clocking block input. This is because reading the input always yields the last sampled value, and not the driven value.

For example, consider the following code:

```
clocking cb @(posedge clk);
   inout a;
   output b;
endclocking

initial begin
   cb.a <= c;     // The value of a will change in the Re-NBA region
   cb.b <= cb.a;  // b is assigned the value of a before the change
end
```

### 14.16.2 Driving clocking output signals

When more than one synchronous drive on the same clocking block output (or inout) is scheduled to mature in the same Re-NBA region of the same time step, the last value is the only value driven onto the output signal. This is true whether the synchronous drives execute at times coincident with clocking events, or at times in between clocking events (within the same clock cycle).

For example:

```
default clocking pe @(posedge clk);
   output nibble;    // four bit output
endclocking

initial begin
  ##2;
  pe.nibble <= 4'b0101;
  pe.nibble <= 4'b0011;
end
```

The driven value of `nibble` is `4'b0011`, regardless of whether `nibble` is a variable or a net.

It is possible for the scheduling loop described in 4.4 to iterate through the Re-NBA region more than once in a given time step. If this happens, synchronous drives will cause their associated clocking signal to glitch (i.e., change value more than once in a time step) if they assign different values to their associated clockvar in different iterations of the scheduling loop.

In the following example, variable `a` will glitch 1 –> 0 –> 1 at the first posedge of `clk`.

```
module m;
   bit a = 1'b1;
   default clocking cb @(posedge clk);
      output a;
   endclocking

   initial begin
      ## 1;
      cb.a <= 1'b0;
      @(x); // x is triggered by reactive stimulus running in same time step
      cb.a <= 1'b1;
   end
endmodule
```

If a given clocking output is driven by multiple synchronous drives that are scheduled to mature at different future times due to the use of cycle delay, the drives shall each mature in their corresponding future cycles.

For example:

```
bit v;
default clocking cb @(posedge clk);
  output v;
endclocking

initial begin
   ##1;                     // Wait until cycle 1
   cb.v <= expr1;           // Matures in cycle 1, v is assigned expr1
   cb.v <= ##2 expr2;       // Matures in cycle 3
   #1 cb.v <= ##2 expr3;    // Matures in cycle 3
```

```
      ##1 cb.v <= ##1 expr4;  // Matures in cycle 3, v is assigned expr4
   end
```

When the same variable is an output from multiple clocking blocks, the last drive determines the value of the variable. This allows a single module to model multirate devices, such as a DDR memory, using a different **clocking** block to model each active edge. For example:

```
   reg j;

   clocking pe @(posedge clk);
      output j;
   endclocking

   clocking ne @(negedge clk);
      output j;
   endclocking
```

The variable `j` is an output from two clocking blocks using different clocking events, `@(`**`posedge`**` clk)` versus `@(`**`negedge`**` clk)`. When driven, the variable `j` shall take on the value most recently assigned by either **clocking** block. A clocking block output only assigns a value to its associated signal in clock cycles where a synchronous drive occurs.

With the **edge** event, this is equivalent to the following simplified declaration:

```
   reg j;
   clocking e @(edge clk);
      output j;
   endclocking
```

Multiple clocking block outputs driving a net cause the net to be driven to its resolved signal value. As described in 14.16, when a clocking block output corresponds to a net, a driver on that net is created. This semantic model applies to each clocking block output that drives the net. The driving values of all these driver(s), together with any other drivers on the net, shall be resolved as determined by the net's type.

It is possible to use a procedural assignment to assign to a signal associated with an output clockvar. When the associated signal is a variable, the procedural assignment assigns a new value to the variable, and the variable shall hold that value until another assignment occurs (either from a drive to a clocking block output or another procedural assignment).

If a synchronous drive and a procedural nonblocking assignment write to the same variable in the same time step, the writes shall take place in an arbitrary order.

It shall be illegal to write to a variable with a continuous assignment, a procedural continuous assignment, or a primitive when that variable is associated with an output clockvar.

# 15. Interprocess synchronization and communication

## 15.1 General

This clause describes the following:

— Semaphores
— Mailboxes
— Named events

## 15.2 Overview

High-level and easy-to-use synchronization and communication mechanisms are essential to control the kinds of interactions that occur between dynamic processes used to model a complex system or a highly reactive testbench.

The basic synchronization mechanism is the named event type, along with the event trigger and event control constructs (i.e., `->` and `@`). This type of control is limited to static objects. It is adequate for synchronization at the hardware level and simple system level, but falls short of the needs of a highly dynamic, reactive testbench.

SystemVerilog also provides a powerful and easy-to-use set of synchronization and communication mechanisms that can be created and reclaimed dynamically. This set comprises of a semaphore built-in class, which can be used for synchronization and mutual exclusion to shared resources, and a mailbox built-in class, which can be used as a communication channel between processes.

Semaphores and mailboxes are built-in types; nonetheless, they are classes and can be used as base classes for deriving additional higher level classes. These built-in classes reside in the built-in `std` package (see 26.7); thus, they can be redefined by user code in any other scope.

## 15.3 Semaphores

Conceptually, a semaphore is a bucket. When a semaphore is allocated, a bucket that contains a fixed number of keys is created. Processes using semaphores must first procure a key from the bucket before they can continue to execute. If a specific process requires a key, only a fixed number of occurrences of that process can be in progress simultaneously. All others must wait until a sufficient number of keys is returned to the bucket. Semaphores are typically used for mutual exclusion, access control to shared resources, and basic synchronization.

An example of creating a semaphore is as follows:

```
    semaphore smTx;
```

Semaphore is a built-in class that provides the following methods:

— Create a semaphore with a specified number of keys: **`new`**`()`
— Obtain one or more keys from the bucket: `get()`
— Return one or more keys into the bucket: `put()`
— Try to obtain one or more keys without blocking: `try_get()`

### 15.3.1 New()

Semaphores are created with the **new**() method.

The prototype for **new**() is as follows:

```
function new(int keyCount = 0 );
```

The keyCount specifies the number of keys initially allocated to the semaphore bucket. The number of keys in the bucket can increase beyond keyCount when more keys are put into the semaphore than are removed. The default value for keyCount is 0.

The **new**() function returns the semaphore handle.

### 15.3.2 Put()

The semaphore put() method is used to return keys to a semaphore.

The prototype for put() is as follows:

```
function void put(int keyCount = 1);
```

The keyCount specifies the number of keys being returned to the semaphore. The default is 1.

When the semaphore.put() function is called, the specified number of keys is returned to the semaphore. If a process has been suspended waiting for a key, that process shall execute if enough keys have been returned.

### 15.3.3 Get()

The semaphore get() method is used to procure a specified number of keys from a semaphore.

The prototype for get() is as follows:

```
task get(int keyCount = 1);
```

The keyCount specifies the required number of keys to obtain from the semaphore. The default is 1.

If the specified number of keys is available, the method returns and execution continues. If the specified number of keys is not available, the process blocks until the keys become available.

The semaphore waiting queue is first-in first-out (FIFO). This does not guarantee the order in which processes arrive at the queue, only that their arrival order shall be preserved by the semaphore.

### 15.3.4 Try_get()

The semaphore try_get() method is used to procure a specified number of keys from a semaphore, but without blocking.

The prototype for try_get() is as follows:

```
function int try_get(int keyCount = 1);
```

The keyCount specifies the required number of keys to obtain from the semaphore. The default is 1.

If the specified number of keys is available, the method returns a positive integer and execution continues. If the specified number of keys is not available, the method returns 0.

## 15.4 Mailboxes

A mailbox is a communication mechanism that allows messages to be exchanged between processes. Data can be sent to a mailbox by one process and retrieved by another.

Conceptually, mailboxes behave like real mailboxes. When a letter is delivered and put into the mailbox, a person can retrieve the letter (and any data stored within). However, if the letter has not been delivered when the mailbox is checked, the person must choose whether to wait for the letter or to retrieve the letter on a subsequent trip to the mailbox. Similarly, SystemVerilog's mailboxes provide processes to transfer and retrieve data in a controlled manner. Mailboxes are created as having either a bounded or unbounded queue size. A bounded mailbox becomes full when it contains the bounded number of messages. A process that attempts to place a message into a full mailbox shall be suspended until enough room becomes available in the mailbox queue. Unbounded mailboxes never suspend a thread in a send operation.

An example of creating a mailbox is as follows:

```
mailbox mbxRcv;
```

Mailbox is a built-in class that provides the following methods:

- Create a mailbox: **new**()
- Place a message in a mailbox: `put()`
- Try to place a message in a mailbox without blocking: `try_put()`
- Retrieve a message from a mailbox: `get()` or `peek()`
- Try to retrieve a message from a mailbox without blocking: `try_get()` or `try_peek()`
- Retrieve the number of messages in the mailbox: `num()`

### 15.4.1 New()

Mailboxes are created with the **new**() method.

The prototype for mailbox **new**() is as follows:

```
function new(int bound = 0);
```

The **new**() function returns the mailbox handle. If the `bound` argument is 0, then the mailbox is unbounded (the default) and a `put()` operation shall never block. If `bound` is nonzero, it represents the size of the mailbox queue.

The bound shall be positive. Negative bounds are illegal and can result in indeterminate behavior, but implementations can issue a warning.

### 15.4.2 Num()

The number of messages in a mailbox can be obtained via the `num()` method.

The prototype for `num()` is as follows:

```
function int num();
```

The `num()` method returns the number of messages currently in the mailbox.

The returned value should be used with care because it is valid only until the next `get()` or `put()` is executed on the mailbox. These mailbox operations can be from different processes from the one executing the `num()` method. Therefore, the validity of the returned value depends on the time that the other methods start and finish.

### 15.4.3 Put()

The `put()` method places a message in a mailbox.

The prototype for `put()` is as follows:

```
task put( singular message);
```

The `message` is any singular expression, including object handles.

The `put()` method stores a message in the mailbox in strict FIFO order. If the mailbox was created with a bounded queue, the process shall be suspended until there is enough room in the queue.

### 15.4.4 Try_put()

The `try_put()` method attempts to place a message in a mailbox.

The prototype for `try_put()` is as follows:

```
function int try_put( singular message);
```

The `message` is any singular expression, including object handles.

The `try_put()` method stores a message in the mailbox in strict FIFO order. This method is meaningful only for bounded mailboxes. If the mailbox is not full, then the specified message is placed in the mailbox, and the function returns a positive integer. If the mailbox is full, the method returns 0.

### 15.4.5 Get()

The `get()` method retrieves a message from a mailbox.

The prototype for `get()` is as follows:

```
task get( ref singular message );
```

The `message` can be any singular expression, and it shall be a valid left-hand expression.

The `get()` method retrieves one message from the mailbox, that is, removes one message from the mailbox queue. If the mailbox is empty, then the current process blocks until a message is placed in the mailbox. If the type of the `message` variable is not equivalent to the type of the message in the mailbox, a run-time error is generated.

Nonparameterized mailboxes are typeless (see 15.4.9), that is, a single mailbox can send and receive different types of data. Thus, in addition to the data being sent (i.e., the message queue), a mailbox implementation shall maintain the message data type placed by `put()`. This is required in order to enable the run-time type checking.

The mailbox waiting queue is FIFO. This does not guarantee the order in which processes arrive at the queue, only that their arrival order shall be preserved by the mailbox.

### 15.4.6 Try_get()

The `try_get()` method attempts to retrieves a message from a mailbox without blocking.

The prototype for `try_get()` is as follows:

```
function int try_get( ref singular message );
```

The `message` can be any singular expression, and it shall be a valid left-hand expression.

The `try_get()` method tries to retrieve one message from the mailbox. If the mailbox is empty, then the method returns 0. If the type of the `message` variable is not equivalent to the type of the message in the mailbox, the method returns a negative integer. If a message is available and the message type is equivalent to the type of the `message` variable, the message is retrieved, and the method returns a positive integer.

### 15.4.7 Peek()

The `peek()` method copies a message from a mailbox without removing the message from the queue.

The prototype for `peek()` is as follows:

```
task peek( ref singular message );
```

The `message` can be any singular expression, and it shall be a valid left-hand expression.

The `peek()` method copies one message from the mailbox without removing the message from the mailbox queue. If the mailbox is empty, then the current process blocks until a message is placed in the mailbox. If the type of the `message` variable is not equivalent to the type of the message in the mailbox, a run-time error is generated.

Calling the `peek()` method can also cause one message to unblock more than one process. As long as a message remains in the mailbox queue, any process blocked in either a `peek()` or `get()` operation shall become unblocked.

### 15.4.8 Try_peek()

The `try_peek()` method attempts to copy a message from a mailbox without blocking.

The prototype for `try_peek()` is as follows:

```
function int try_peek( ref singular message );
```

The `message` can be any singular expression, and it shall be a valid left-hand expression.

The `try_peek()` method tries to copy one message from the mailbox without removing the message from the mailbox queue. If the mailbox is empty, then the method returns 0. If the type of the `message` variable is not equivalent to the type of the message in the mailbox, the method returns a negative integer. If a message is available and its type is equivalent to the type of the message variable, the message is copied, and the method returns a positive integer.

### 15.4.9 Parameterized mailboxes

The default mailbox is typeless, that is, a single mailbox can send and receive any type of data. This is a very powerful mechanism, which, unfortunately, can also result in run-time errors due to type mismatches (types not equivalent) between a message and the type of the variable used to retrieve the message. Frequently, a mailbox is used to transfer a particular message type, and, in that case, it is useful to detect type mismatches at compile time.

Parameterized mailboxes use the same parameter mechanism as parameterized classes (see 8.25), modules, and interfaces:

```
mailbox #(type = dynamic_type)
```

where `dynamic_type` represents a special type that enables run-time type checking (the default).

A parameterized mailbox of a specific type is declared by specifying the type:

```
typedef mailbox #(string) s_mbox;

s_mbox sm = new;
string s;

sm.put( "hello" );
...
sm.get( s );    // s <- "hello"
```

Parameterized mailboxes provide all the same standard methods as dynamic mailboxes: `num()`, **`new`**`()`, `get()`, `peek()`, `put()`, `try_get()`, `try_peek()`, `try_put()`.

The only difference between a generic (dynamic) mailbox and a parameterized mailbox is that for a parameterized mailbox, the compiler verifies that the calls to `put`, `try_put`, `peek`, `try_peek`, `get`, and `try_get` methods use argument types equivalent to the mailbox type so that all type mismatches are caught by the compiler and not at run time.

## 15.5 Named events

An identifier declared as an event data type is called a *named event*. A named event can be triggered explicitly. It can be used in an event expression to control the execution of procedural statements in the same manner as event controls described in 9.4.2. A named event may also be used as a handle assigned from another named event.

A named event provides a handle to an underlying synchronization object. When a process waits for an event to be triggered, the process is put on a queue maintained within the synchronization object. Processes can wait for a named event to be triggered either via the `@` operator or by the use of the `wait()` construct to examine their triggered state.

### 15.5.1 Triggering an event

An event is made to occur by the activation of an event triggering statement with the syntax given in Syntax 15-1.

```
event_trigger ::=                                                        // from A.6.5
      -> hierarchical_event_identifier ;
    | ->> [ delay_or_event_control ] hierarchical_event_identifier ;
```

*Syntax 15-1—Event trigger syntax (excerpt from Annex A)*

An event is not made to occur by changing the index of a named event array in an event control expression.

Named events triggered via the `->` operator unblock all processes currently waiting on that event. When triggered, named events behave like a one shot, i.e., the trigger state itself is not observable, only its effect. This is similar to the way in which an edge can trigger a flip-flop, but the state of the edge cannot be ascertained, i.e., `if (posedge clock)` is illegal.

Nonblocking events are triggered using the `->>` operator. The effect of the `->>` operator is that the statement executes without blocking, and it creates a nonblocking assign update event in the time in which the delay control expires or the event control occurs. The effect of this update event shall be to trigger the referenced event in the nonblocking assignment region of the simulation cycle.

### 15.5.2 Waiting for an event

The basic mechanism to wait for an event to be triggered is via the event control operator, `@`.

```
@ hierarchical_event_identifier;
```

The `@` operator blocks the calling process until the given event is triggered.

For a trigger to unblock a process waiting on an event, the waiting process shall execute the `@` statement before the triggering process executes the trigger operator, `->`. If the trigger executes first, then the waiting process remains blocked.

### 15.5.3 Persistent trigger: triggered built-in method

SystemVerilog can distinguish the event trigger itself, which is instantaneous, from the named event's triggered state, which persists throughout the time step (i.e., until simulation time advances). The `triggered` built-in method of a named event allows users to examine this state.

The prototype for the `triggered()` method is as follows:

```
function bit triggered();
```

The `triggered` method evaluates to true (`1'b1`) if the given event has been triggered in the current time step and false (`1'b0`) otherwise. If the named event is **`null`**, then the `triggered` method returns false.

The `triggered` method is most useful when used in the context of a **`wait`** construct:

```
wait ( hierarchical_event_identifier.triggered )
```

Using this mechanism, an event trigger shall unblock the waiting process whether the **`wait`** executes before or at the same simulation time as the trigger operation. The `triggered` method, thus, helps eliminate a common race condition that occurs when both the trigger and the **`wait`** happen at the same time. A process that blocks waiting for an event might or might not unblock, depending on the execution order of the waiting and triggering processes. However, a process that waits on the triggered state always unblocks, regardless of the order of execution of the wait and trigger operations.

*Example:*

```
event done, blast;          // declare two new events
event done_too = done;      // declare done_too as alias to done

task trigger( event ev );
   -> ev;
endtask

...

fork
   @ done_too;              // wait for done through done_too
   #1 trigger( done );      // trigger done through task trigger
join

fork
   -> blast;
   wait ( blast.triggered );
join
```

The first fork in the example shows how two event identifiers, `done` and `done_too`, refer to the same synchronization object and also how an event can be passed to a generic task that triggers the event. In the example, one process waits for the event via `done_too`, while the actual triggering is done via the `trigger` task that is passed `done` as an argument.

In the second fork, one process can trigger the event `blast` before the other process (if the processes in the **`fork-join`** execute in source order) has a chance to execute, and wait for the event. Nonetheless, the second process unblocks and the fork terminates. This is because the process waits for the event's triggered state, which remains in its triggered state for the duration of the time step.

An event expression or wait condition is only reevaluated on a change to an operand in the expression, such as the event prefix of the `triggered` method. This means that the change of the return value of the `triggered` method from `1'b1` to `1'b0` at the end of the current time step will not affect an event control or wait statement waiting on the `triggered` method.

### 15.5.4 Event sequencing: wait_order()

The **`wait_order`** construct suspends the calling process until all of the specified events are triggered in the given order (left to right) or any of the untriggered events are triggered out of order and thus causes the operation to fail.

The syntax for the **`wait_order`** construct is as follows in Syntax 15-2.

wait_statement ::= *// from A.6.5*
  ...
  | **wait_order (** hierarchical_identifier { **,** hierarchical_identifier } **)** action_block
action_block ::=
  statement _or_null
  | [ statement ] **else** statement

*Syntax 15-2—Wait_order event sequencing syntax (excerpt from Annex A)*

For **wait_order** to succeed, at any point in the sequence, the subsequent events, which shall all be untriggered at this point or the sequence would have already failed, shall be triggered in the prescribed order. Preceding events are not limited to occur only once. In other words, once an event occurs in the prescribed order, it can be triggered again without causing the construct to fail.

Only the first event in the list can wait for the persistent `triggered` event.

The action taken when the construct fails depends on whether the optional action_block **else** statement (the fail statement) is specified. If it is specified, then the given statement is executed upon failure of the construct. If the fail statement is not specified, a failure generates a run-time error.

For example:

```
wait_order( a, b, c);
```

suspends the current process until events `a`, `b`, and `c` trigger in the order `a -> b -> c`. If the events trigger out of order, a run-time error is generated.

For example:

```
wait_order( a, b, c ) else $display( "Error: events out of order" );
```

In this example, the fail statement specifies that, upon failure of the construct, a user message be displayed, but without an error being generated.

For example:

```
bit success;
wait_order( a, b, c ) success = 1; else success = 0;
```

In this example, the completion status is stored in the variable `success`, without an error being generated.

### 15.5.5 Operations on named event variables

An event is a unique data type with several important properties. Named events can be assigned to one another. When one event is assigned to another, the synchronization queue of the source event is shared by both the source and the destination event. In this sense, events act as full-fledged variables and not merely as labels.

#### 15.5.5.1 Merging events

When one event variable is assigned to another, the two become merged. Thus, executing `->` on either event variable affects processes waiting on either event variable.

For example:

```
event a, b, c;
a = b;
-> c;
-> a;    // also triggers b
-> b;    // also triggers a
a = c;
b = a;
-> a;    // also triggers b and c
-> b;    // also triggers a and c
-> c;    // also triggers a and b
```

When events are merged, the assignment only affects the execution of subsequent event control or wait operations. If a process is blocked waiting for `event1` when another event is assigned to `event1`, the currently waiting process shall never unblock. For example:

```
fork
   T1: forever @ E2;
   T2: forever @ E1;
   T3: begin
         E2 = E1;
         forever -> E2;
   end
join
```

This example forks off three concurrent processes. Each process starts at the same time. Thus, at the same time that processes `T1` and `T2` are blocked, process `T3` assigns event `E1` to `E2`. As a result, process `T1` shall never unblock because the event `E2` is now `E1`. To unblock both threads `T1` and `T2`, the merger of `E2` and `E1` must take place before the fork.

### 15.5.5.2 Reclaiming events

When an event variable is assigned the special **`null`** value, the association between the event variable and the underlying synchronization queue is broken. When no event variable is associated with an underlying synchronization queue, the resources of the queue itself become available for reuse.

Triggering a **`null`** event shall have no effect. The outcome of waiting on a **`null`** event is undefined, and implementations can issue a run-time warning.

For example:

```
event E1 = null;
@ E1;                  // undefined: might block forever or not at all
wait( E1.triggered );  // undefined
-> E1;                 // no effect
```

### 15.5.5.3 Events comparison

Event variables can be compared against other event variables or the special value **`null`**. Only the following operators are allowed for comparing event variables:

— Equality (==) with another event or with **`null`**
— Inequality (`!=`) with another event or with **`null`**
— Case equality (===) with another event or with **`null`** (same semantics as ==)
— Case inequality (`!==`) with another event or with **`null`** (same semantics as `!=`)
— Test for a Boolean value that shall be 0 if the event is **`null`** and 1 otherwise

*Example:*

```
event E1, E2;
if ( E1 )   // same as if ( E1 != null )
   E1 = E2;
if ( E1 == E2 )
   $display( "E1 and E2 are the same event" );
```

# 16. Assertions

## 16.1 General

This clause describes the following:

— Immediate assertions
— Concurrent assertions
— Sequence specifications
— Property specifications

## 16.2 Overview

An assertion specifies a behavior of the system. Assertions are primarily used to validate the behavior of a design. In addition, assertions can be used to provide functional coverage and to flag that input stimulus, which is used for validation, does not conform to assumed requirements.

An assertion appears as an assertion statement that states the verification function to be performed. The statement shall be of one of the following kinds:

— **assert**, to specify the property as an obligation for the design that is to be checked to verify that the property holds.
— **assume**, to specify the property as an assumption for the environment. Simulators check that the property holds, while formal tools use the information to generate input stimulus.
— **cover**, to monitor the property evaluation for coverage.
— **restrict**, to specify the property as a constraint on formal verification computations. Simulators do not check the property.

There are two kinds of assertions: *concurrent* and *immediate*.

— Immediate assertions follow simulation event semantics for their execution and are executed like a statement in a procedural block. Immediate assertions are primarily intended to be used with simulation. There is no immediate **restrict** assertion statement.
— Concurrent assertions are based on clock semantics and use sampled values of their expressions (see 16.5.1). One of the goals of SystemVerilog assertions is to provide a common semantic meaning for assertions so that they can be used to drive various design and verification tools. Many tools, such as formal verification tools, evaluate circuit descriptions using cycle-based semantics, which typically rely on a clock signal or signals to drive the evaluation of the circuit. Any timing or event behavior between clock edges is abstracted away. Concurrent assertions incorporate this clock semantics. While this approach generally simplifies the evaluation of a circuit description, there are a number of scenarios under which this cycle-based evaluation provides different behavior from the standard event-based evaluation of SystemVerilog.

This clause describes both types of assertions.

## 16.3 Immediate assertions

The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code. The expression is nontemporal and is interpreted the same way as an expression in the condition of a procedural **if** statement. In other words, if the expression evaluates to X, Z, or 0, then it is interpreted as being false, and the assertion statement is said to fail. Otherwise, the expression is interpreted as being true, and the assertion statement is said to pass or, equivalently, to succeed.

There are two modes of immediate assertions, *simple immediate assertions* and *deferred immediate assertions*. In a simple immediate assertion, pass and fail actions take place immediately upon assertion evaluation. In a deferred immediate assertion, the actions are delayed until later in the time step, providing some level of protection against unintended multiple executions on transient or "glitch" values. Deferred immediate assertions are described in detail in 16.4.

The *immediate_assertion_statement* is a *statement_item* and can be specified anywhere a procedural statement is specified. The execution of immediate assertions can be controlled by using assertion control system tasks (see 20.12).

```
procedural_assertion_statement ::=                                    // from A.6.10
      ...
    | immediate_assertion_statement
      ...
immediate_assertion_statement ::=
      simple_immediate_assertion_statement
    | deferred_immediate_assertion_statement
simple_immediate_assertion_statement ::=
      simple_immediate_assert_statement
    | simple_immediate_assume_statement
    | simple_immediate_cover_statement
simple_immediate_assert_statement ::=
    assert ( expression ) action_block
simple_immediate_assume_statement ::=
    assume ( expression ) action_block
simple_immediate_cover_statement ::=
    cover ( expression ) statement_or_null
deferred_immediate_assertion_item ::= [ block_identifier : ] deferred_immediate_assertion_statement
deferred_immediate_assertion_statement ::=
      deferred_immediate_assert_statement
    | deferred_immediate_assume_statement
    | deferred_immediate_cover_statement
deferred_immediate_assert_statement ::=
      assert #0 ( expression ) action_block
    | assert final ( expression ) action_block
deferred_immediate_assume_statement ::=
      assume #0 ( expression ) action_block
    | assume final ( expression ) action_block
deferred_immediate_cover_statement ::=
      cover #0 ( expression ) statement_or_null
    | cover final ( expression ) statement_or_null
action_block ::=                                                       // from A.6.3
      statement _or_null
    | [ statement ] else statement_or_null
```

*Syntax 16-1—Immediate assertion syntax (excerpt from Annex A)*

An immediate assertion statement may be an immediate **assert**, an immediate **assume**, or an immediate **cover**.

The immediate **assert** statement specifies that its expression is required to hold. Failure of an immediate **assert** statement indicates a violation of the requirement and thus a potential error in the design. If an **assert** statement fails and no **else** clause is specified, the tool shall, by default, call `$error`, unless `$assertcontrol` with `control_type` 9 (FailOff) is used to suppress the failure.

The immediate **assume** statement specifies that its expression is assumed to hold. For example, immediate **assume** statements can be used with formal verification tools to specify assumptions on design inputs that constrain the verification computation. When used in this way, they specify the expected behavior of the environment of the design as opposed to that of the design itself. In simulation, an immediate assume may behave as an immediate **assert** to verify that the environment behaves as assumed. A simulation tool shall provide the capability to check the immediate **assume** statement in this way.

The *action_block* of an immediate **assert** or **assume** statement specifies what actions are taken upon success or failure of the assertion. The statement associated with success is the first statement. It is called the *pass statement* and shall be executed if the *expression* evaluates to true. The pass statement can, for example, record the number of successes for a coverage log, but can be omitted altogether. If the pass statement is omitted, then no user-specified action is taken when the assert *expression* of the immediate **assert** or **assume** statement is true. The statement associated with **else** is called the *fail statement* and shall be executed if the *expression* evaluates to false. The **else** statement can also be omitted. The action block shall be enabled to execute immediately after the evaluation of the assert *expression* of the immediate **assert** or **assume** statement. The execution of pass and fail statements can be controlled by using assertion action control tasks. The assertion action control tasks are described in 20.12.

The immediate **cover** statement specifies that successful evaluation of its *expression* is a coverage goal. Tools shall collect coverage information and report the results at the end of simulation or on demand via an assertion API (see Clause 39). The results of coverage for an immediate **cover** statement shall contain the following:

— Number of times evaluated
— Number of times succeeded

A pass statement for an immediate **cover** may be specified in *statement_or_null*. The pass statement shall be executed if the expression evaluates to true. The pass statement shall be enabled to execute immediately after the evaluation of the expression of the immediate **cover**.

The optional statement label (identifier and colon) creates a named block around the assertion statement (or any other statement), and the hierarchical name of the scope can be displayed using the `%m` format specification.

The information about assertion failure can be printed using one of the following severity system tasks in the action block:

— `$fatal` is a run-time fatal.
— `$error` is a run-time error.
— `$warning` is a run-time warning.
— `$info` indicates that the assertion failure carries no specific severity.

The syntax for these severity system tasks is shown in 20.10.

The severity system tasks can be used in assertion pass or fail statements. These tasks shall print the same tool-specific message when used either in a pass or a fail statement. For example:

```
assert_f: assert(f) $info("passed"); else $error("failed");

assume_inputs: assume (in_a || in_b) $info("assumption holds");
```

```
    else $error("assumption does not hold");

    cover_a_and_b: cover (in_a && in_b) $info("in_a && in_b == 1 covered");
```

For example, a formal verification tool might prove `assert_f` under the assumption `assume_inputs` expressing the condition that `in_a` and `in_b` are not both 0 at the same time. The **`cover`** statement detects whether `in_a` and `in_b` are both simultaneously 1.

If more than one of these system tasks is included in the action block, then each shall be executed as specified.

If the severity system task is executed at a time other than when the immediate **`assert`** or **`assume`** fails, the actual failure time of the immediate **`assert`** or **`assume`** can be recorded and displayed programmatically. For example:

```
time t;

always @(posedge clk)
   if (state == REQ)
      assert (req1 || req2)
      else begin
         t = $time;
         #5 $error("assert failed at time %0t",t);
      end
```

If the immediate **`assert`** fails at time 10, the error message shall be printed at time 15, but the user-defined string printed will be "assert failed at time 10."

Because the fail statement, like the pass statement, is any legal SystemVerilog procedural statement, it can also be used to signal a failure to another part of the testbench.

```
assert (myfunc(a,b)) count1 = count + 1; else ->event1;
assert (y == 0) else flag = 1;
```

## 16.4 Deferred assertions

---

immediate_assertion_statement ::= *// from A.6.10*
  ...
  | deferred_immediate_assertion_statement

deferred_immediate_assertion_item ::= [ block_identifier **:** ] deferred_immediate_assertion_statement

deferred_immediate_assertion_statement ::=
  deferred_immediate_assert_statement
  | deferred_immediate_assume_statement
  | deferred_immediate_cover_statement

deferred_immediate_assert_statement ::=
  **assert #0 (** expression **)** action_block
  | **assert final (** expression **)** action_block

deferred_immediate_assume_statement ::=
  **assume #0 (** expression **)** action_block
  | **assume final (** expression **)** action_block

deferred_immediate_cover_statement ::=
    **cover #0 (** expression **)** statement_or_null
  | **cover final (** expression **)** statement_or_null

*Syntax 16-2—Deferred immediate assertion syntax (excerpt from Annex A)*

Deferred assertions are a kind of immediate assertion. They can be used to suppress false reports that occur due to glitching activity on combinational inputs to immediate assertions. Since deferred assertions are a subset of immediate assertions, the term *deferred assertion* (often used for brevity) is equivalent to the term *deferred immediate assertion*. The term *simple immediate assertion* refers to an immediate assertion that is not deferred. In addition, there are two different kinds of deferred assertions: *observed deferred immediate assertions* and *final deferred immediate assertions*.

A deferred assertion is similar to a simple immediate assertion, but with the following key differences:

— Syntax: Deferred assertions use `#0` (for an observed deferred assertion) or **final** (for a final deferred assertion) after the verification directive.
— Deferral: Reporting is delayed rather than being reported immediately.
— Action block limitations: Action blocks may only contain a single subroutine call.
— Use outside procedures: A deferred assertion may be used as a *module_common_item*.

Deferred assertion syntax is similar to simple immediate assertion syntax, with the difference being the specification of a `#0` or **final** after the **assert**, **assume**, or **cover**:

```
assert #0 (expression) action_block
assert final (expression) action_block
```

As with all immediate assertions, a deferred assertion's *expression* is evaluated at the time the deferred assertion statement is processed. However, in order to facilitate glitch avoidance, the reporting or action blocks are scheduled at a later point in the current time step.

The pass and fail statements in a deferred assertion's *action_block*, if present, shall each consist of a single subroutine call. The subroutine can be a task, task method, void function, void function method, or system task. The requirement of a single subroutine call implies that no begin-end block shall surround the pass or fail statements, as **begin** is itself a statement that is not a subroutine call. In the case of a final deferred assertion, the subroutine shall be one that may be legally called in the Postponed region (see 4.4.2.9). A subroutine argument may be passed by value as an input or passed by reference as a **ref** or **const ref**. Actual argument expressions that are passed by value, including function calls, shall be fully evaluated at the instant the deferred assertion expression is evaluated. It shall be an error to pass automatic or dynamic variables as actuals to a **ref** or **const ref** formal. The processing of the action block differs between observed and final deferred assertions as follows:

— For an observed deferred assertion, the subroutine shall be scheduled in the Reactive region. Actual argument expressions that are passed by reference use or assign the current values of the underlying variables in the Reactive region.
— For a final deferred assertion, the subroutine shall be scheduled in the Postponed region. Actual argument expressions that are passed by reference use the current values of the underlying variables in the Postponed region.

Deferred assertions may also be used outside procedural code, as a *module_common_item*. This is explained in more detail in 16.4.3.

In addition to deferred **assert** statements, deferred **assume** and **cover** statements are also defined. Other than the deferred evaluation as described in this subclause, these **assume** and **cover** statements behave the same way as the simple immediate **assume** and **cover** statements described in 16.3. A deferred **assume** will often be useful in cases where a combinational condition is checked in a function, but needs to be used as an assumption rather than a proof target by formal tools. A deferred cover is useful to avoid crediting tests for covering a condition that is only met in passing by glitched values.

### 16.4.1 Deferred assertion reporting

When a deferred assertion passes or fails, the action block is not executed immediately. Instead, the action block subroutine call (or `$error`, if an **assert** or **assume** fails and no *action_block* is present) and the current values of its input arguments are placed in a *deferred assertion report queue* associated with the currently executing process. Such a call is said to be a *pending assertion report*.

If a *deferred assertion flush point* (see 16.4.2) is reached in a process, its deferred assertion report queue is cleared. Any pending assertion reports will not be executed.

In the Observed region of each simulation time step, each pending observed deferred assertion report that has not been flushed from its queue shall *mature*, or be confirmed for reporting. Once a report matures, it may no longer be flushed. Then the associated subroutine call (or `$error`, if the assertion fails and no action block is present) is executed in the Reactive region, and the pending assertion report is cleared from the appropriate process's deferred assertion report queue.

Note that if code in the Reactive region modifies signals and causes another pass to the Active region to occur, this still may create glitching behavior in observed deferred assertions, as the new passage in the Active region may re-execute some of the deferred assertions with different reported results. In general, observed deferred assertions prevent glitches due to order of procedural execution, but do not prevent glitches caused by execution loops between regions that the assignments from the Reactive region may cause.

In the Postponed region of each simulation time step, each pending final deferred assertion report that has not been flushed from its queue shall mature. Then the associated subroutine call (or `$error`, if the assertion fails and no action block is present) is scheduled in the same Postponed region, and the pending assertion report is cleared from the appropriate process's deferred assertion report queue. Due to their execution in the non-iterative Postponed region, final deferred assertions are not vulnerable to the potential glitch behavior previously described for observed deferred assertions.

### 16.4.2 Deferred assertion flush points

A process is defined to have reached a deferred assertion flush point if any of the following occur:

— The process, having been suspended earlier due to reaching an event control or wait statement, resumes execution.

— The process was declared by an **always_comb** or **always_latch**, and its execution is resumed due to a transition on one of its dependent signals.

— The outermost scope of the process is disabled by a **disable** statement (see 16.4.4)

The following example shows how deferred assertions might be used to avoid undesired reports of a failure due to transitional combinational values in a single simulation time step:

```
assign not_a = !a;
always_comb begin : b1
   a1: assert (not_a != a);
   a2: assert #0 (not_a != a); // Should pass once values have settled
end
```

When a changes, a simulator could evaluate assertions a1 and a2 twice—once for the change in a and once for the change in not_a after the evaluation of the continuous assignment. A failure could thus be reported during the first execution of a1. The failure during the first execution of a2 will be scheduled on the process's deferred assertion report queue. When not_a changes, the deferred assertion queue is flushed due to the activation of b1, so no failure of a2 will be reported.

This example illustrates the behavior of deferred assertions in the presence of time delays:

```
always @(a or b) begin : b1
   a3: assert #0 (a == b) rptobj.success(0); else rptobj.error(0, a, b);
   #1;
   a4: assert #0 (a == b) rptobj.success(1); else rptobj.error(1, a, b);
end
```

In this case, due to the time delay in the middle of the procedure, an Observed region will always be reached after the execution of a3 and before a flush point. Thus any passes or failures of a3 will always be reported. For a4, during cycles where either a or b changes after it has been executed, failures will be flushed and never reported. In general, deferred assertions must be used carefully when mixed with time delays.

The following example illustrates a typical use of a deferred **cover** statement:

```
assign a = ...;
assign b = ...;
always_comb begin : b1
   c1: cover (b != a);
   c2: cover #0 (b != a);
end
```

In this example, it is important to make sure some test is covering the case where a and b have different values. Due to the arbitrary order of the assignments in the simulator, it might be the case that in a cycle where both variables are being assigned the same value, b1 executes while a has been assigned but b still holds its previous value. Thus c1 will be triggered, but this is actually a glitch, and probably not a useful piece of coverage information. In the case of c2, this coverage will get added to the deferred report queue, but when b1 is executed the next time (after b has also been assigned its new value), that coverage point will be flushed, and c2 will correctly not get reported as having been covered during that time step.

The next example illustrates a case where, due to short-circuiting (see 11.3.5), the result of a deferred assertion may not appear at first glance to be consistent with the signal values at the end of a time step.

```
function f(bit v);
   p: assert #0 (v);
   ...
endfunction
always_comb begin: myblk
   a = b || f(c);
end
```

Suppose, during some time step, the following sequence of events occurs:

— b is set to 0 while c==1, and myblk is entered. When f is called, assertion p has a passing value.
— Later in the time step b settles at a value of 1, while c becomes 0. When the procedure resumes, the previous execution is flushed. This time, due to short-circuiting, f is never evaluated—so the new failing value of assertion p is never seen.
— In the Reactive region, no passing or failing execution is reported by the simulator on p.

NOTE—If the bitwise | operator, which does not allow short-circuiting, were used instead of || in the assignment to a, then f would be evaluated each time the assignment was reached.

The following example illustrates the evaluation of subroutine arguments to deferred assertion action blocks.

```
function int error_type (int opcode);
   func_assert: assert (opcode < 64) else $display("Opcode error.");
   if (opcode < 32)
      return (0);
   else
      return (1);
   endfunction

   always_comb begin : b1
      a1: assert #0 (my_cond) else
      $error("Error on operation of type %d\n", error_type(opcode));
      a2: assert #0 (my_cond) else
      error_type(opcode);
   ...
   end
```

Suppose block `b1` is executed twice in the Active region of a single time step, with `my_cond == 0`, so it fails assertions `a1` and `a2` both times. Also suppose `opcode` is 64 the first time it is executed, and 0 the second time. The following will occur during simulation:

- Upon each deferred assertion failure, the subroutine arguments of the action block are evaluated, even though the action block itself is not executed.
  - Upon the first failure of `a1`, the arguments of `$error` are examined. Since the second argument contains a function call, that function (`error_type(opcode)`, with `opcode=64`) is evaluated. During this function call, `func_assert` fails and displays the message "Opcode error."
  - Upon the first failure of `a2`, the arguments of `error_type` are examined. Since its only argument is the expression `opcode`, its value 64 is used and no further evaluation is needed at this time.
  - The pending reports with `opcode=64` are placed on the deferred assertion report queue.
- When block `b1` is executed again, the pending reports are flushed from the deferred assertion report queue.
  - Upon the second failure of `a1`, function `error_type` is called with `opcode==0`, so assertion `func_assert` passes.
  - Upon the second failure of `a2`, the value of 0 is used for the expression `opcode`, and no further evaluation is needed at this time.
- When the assertions later mature, the `$error` task will be called for `a1`, and the function `error_type` will be called for `a2`.

The deferral and flushing prevented a report from the first failure of `a1` as expected. But the evaluation of action block subroutine arguments, which happens every time a pending assertion report is queued, caused a function to be called upon each failure. In general, users must be cautious about the contents of action blocks for deferred assertions, since the evaluation of their subroutine arguments on every failure may seem inconsistent with the deferral in some usages.

The following example illustrates the differences between observed deferred assertions and final deferred assertions.

```
module dut(input logic clk, input logic a, input logic b);
   logic c;
   always_ff @(posedge clk)
      c <= b;
   a1: assert #0 (!(a & c)) $display("Pass"); else $display("Fail");
   a2: assert final (!(a & c)) $display("Pass"); else $display("Fail");
endmodule

program tb(input logic clk, output logic a, output logic b);
   default clocking m @(posedge clk);
      default input #0;
      default output #0;
      output a;
      output b;
   endclocking

   initial begin
      a = 1;
      b = 0;
      ##10;
      b = 1;
      ##1;
      a = 0;
   end
endprogram

module sva_svtb;
   bit clk;
   logic a, b;
   ...
   dut dut (.*);
   tb tb (.*);
endmodule
```

In the 11th clock cycle, observed deferred assertion `a1` will first execute in the Active region, and it will fail since at this point `a` and `c` are both 1. This pending assertion report will mature in the Observed region, and the failure report will be scheduled in the Reactive region. However, in the Reactive region of the same time step, the testbench will set `a` to 0, triggering another execution of the implied **`always_comb`** block containing assertion `a1`. This time `a1` will pass. So both a pass and a fail message will be displayed for `a1` during this time step.

For final deferred assertion `a2`, the behavior will be different. As with `a1`, a pending assertion report will be generated when the assertion fails in the Active region. However, when the value of `a` changes in the Reactive region and the assertion's implicit **`always_comb`** is resumed, this creates a flush point, so this pending report will be flushed. `a2` will be executed again with the new value, and the new result will be put on the deferred assertion report queue. In the Postponed region, this will mature, and the final passing result of this assertion will be the only one reported.

### 16.4.3 Deferred assertions outside procedural code

A deferred assertion statement may also appear outside procedural code, in which case it is referred to as a *static deferred assertion*. In such cases, it is treated as if it were contained in an **`always_comb`** procedure. For example:

```
module m (input a, b);
   a1: assert #0 (a == b);
endmodule
```

This is equivalent to the following:

```
module m (input a, b);
   always_comb begin
```

```
      a1: assert #0 (a == b);
   end
endmodule
```

Static deferred assertions in checkers are described in 17.3.

### 16.4.4 Disabling deferred assertions

The **disable** statement shall interact with deferred assertions as follows:

- A specific deferred assertion may be disabled. Any pending assertion reports for that assertion are cancelled.
- When a **disable** is applied to the outermost scope of a procedure that has an active deferred assertion queue, in addition to normal disable activities (see 9.6.2), the deferred assertion report queue is flushed and all pending assertion reports on the queue are cleared.

Disabling a task or a non-outermost scope of a procedure does not cause flushing of any pending reports.

The following example illustrates how user code can explicitly flush a pending assertion report. In this case, failures of `a1` are only reported in time steps where `bad_val_ok` does not settle at a value of 1.

```
always @(bad_val or bad_val_ok) begin : b1
   a1: assert #0 (bad_val) else $fatal(1, "Sorry");
   if (bad_val_ok) begin
      disable a1;
   end
end
```

The following example illustrates how user code can explicitly flush all pending assertion reports on the deferred assertion queue of process `b2`:

```
always @(a or b or c) begin : b2
   if (c == 8'hff) begin
      a2: assert #0 (a && b);
   end else begin
      a3: assert #0 (a || b);
   end
end

always @(clear_b2) begin : b3
   disable b2;
end
```

### 16.4.5 Deferred assertions and multiple processes

As described in the previous subclauses, deferred assertions are inherently associated with the process in which they are executed. This means that a deferred assertion within a function may be executed several times due to the function being called by several different processes, and each of these different process executions is independent. The following example illustrates this situation:

```
module fsm(...);
   function bit f (int a, int b)
      ...
      a1: assert #0 (a == b);
      ...
   endfunction
   ...
```

```
      always_comb begin : b1
         some_stuff = f(x,y) ? ...
         ...
      end
      always_comb begin : b2
         other_stuff = f(z,w) ? ...
         ...
      end
   endmodule
```

In this case, there are two different processes that may call assertion `a1`: `b1` and `b2`. Suppose simulation executes the following scenario in the first passage through the Active region of each time step:

— In time step 1, `b1` executes with `x!=y`, and `b2` executes with `z!=w`.

— In time step 2, `b1` executes with `x!=y`, then again with `x==y`.

— In time step 3, `b1` executes with `x!=y`, then `b2` executes with `z==w`.

In the first time step, since `a1` fails independently for processes `b1` and `b2`, its failure is reported twice.

In the second time step, the failure of `a1` in process `b1` is flushed when the process is re-triggered, and since the final execution passes, no failure is reported.

In the third time step, the failure in process `b1` does not see a flush point, so that failure is reported. In process `b2`, the assertion passes, so no failure is reported from that process.

## 16.5 Concurrent assertions overview

Concurrent assertions describe behavior that spans over time. Unlike immediate assertions, the evaluation model is based on a clock so that a concurrent assertion is evaluated only at the occurrence of a clock tick. The term *clock tick* refers to a time step when a clocking event of a sequence, property, sampled value function, or assertion statement occurs. Due to the need to verify proper behavior of the system and conform as closely as possible to cycle-based semantics, the clocking event should be glitch-free and only transition once during any time step. If the clocking event transitions more than once during a time step, the resulting behavior is undefined.

Concurrent assertions use the sampled values of their expressions except for disable conditions (see 16.15) and clocking events. Expression sampling is explained in 16.5.1. Concurrent assertions are evaluated in the Observed region.

### 16.5.1 Sampling

Concurrent assertions and several other constructs (such as variables referenced in an **`always_ff`** procedure in a checker, see 17.5) have special rules for sampling values of their expressions. The value of an expression sampled in one of these constructs is called a *sampled value*. In most cases the sampled value of an expression is its value in the Preponed region. This rule has, however, several important exceptions. The rest of this subclause provides the formal definition of sampling.

The default sampled value of an expression is defined as follows:

— The default sampled value of a static variable is the value assigned in its declaration, or, in the absence of such an assignment, it is the default (or uninitialized) value of the corresponding type (see 6.8, Table 6-7).

— The default sampled value of any other variable or net is the default value of the corresponding type (see 6.8, Table 6-7). For example, the default sampled value of variable `y` of type **`logic`** is `1'bx`.

— The default sampled value of the `triggered` event method (see 15.5.3) and the sequence methods `triggered` and `matched` is false (`1'b0`).

— The default sampled value of an expression is defined recursively by evaluating the expression using the default sampled values of its component subexpressions and variables.

A default sampled value is used in the definition of a sampled value of an expression as explained below, and in the definition of sampled value functions when there is a need to reference a sampled value of an expression before time zero (see 16.9.3).

The definition of a sampled value of an expression is based on the definition of a sampled value of a variable. The general rule for variable sampling is as follows:

— The sampled value of a variable in a time slot corresponding to time greater than 0 is the value of this variable in the Preponed region of this time slot.

— The sampled value of a variable in a time slot corresponding to time 0 is its default sampled value.

This rule has the following exceptions:

— Sampled values of automatic variables (see 16.14.6), local variables (see 16.10), and active free checker variables (see 17.7.2) are their current values. However,

- When a past or a future value of an active free checker variable is referenced by a sampled value function (see 16.9.3 and 16.9.4), this value is sampled in the Postponed region of the corresponding past or future clock tick;
- When a past or a future value of an automatic variable is referenced by a sampled value function, the current value of the automatic variable is taken instead.

— If a variable is an input variable of a clocking block, the variable shall be sampled by the clocking block with `#1step` sampling. Any other type of sampling for the clocking block variable shall result in an error. The sampled value of a such variable is the sampled value produced by the clocking block. This is explained in Clause 14.

The sampled value of an expression is defined as follows:

— The sampled value of an expression consisting of a single variable is the sampled value of this variable.

— The sampled value of a **`const`** cast expression (see 6.24.1 and 16.14.6) is defined as the current value of its argument. For example, if `a` is a variable, then the sampled value of **`const`**`'(a)` is the current value of `a`. When a past or a future value of a **`const`** cast expression is referenced by a sampled value function, the current value of this expression is taken instead.

— The sampled value of the `triggered` event method and the sequence methods `triggered` and `matched` (see 16.13.6) is defined as the current value returned by the event property or sequence method. When a past or a future value of an event property or sequence method is referenced by a sampled value function (see 16.9.3 and 16.9.4), this value is sampled in the Postponed region of the corresponding past or future clock tick.

— The sampled value of any other expression is defined recursively using the values of its arguments. For example, the sampled value of an expression `e1 & e2`, where `e1` and `e2` are expressions, is the bitwise AND of the sampled values of `e1` and `e2`. In particular, if an expression contains a function call, to evaluate the sampled value of this expression, the function is called on the sampled values of its arguments at the time of the expression evaluation. For example, if `a` is a static module variable, `s` is a sequence, and `f` is a function, the sampled value of `f(a, s.triggered)` is the result of the application of `f` to the sampled values of `a` and `s.triggered`, i.e., to the value of `a` taken from the Preponed region and to the current value of `s.triggered`.

### 16.5.2 Assertion clock

The timing model employed in a concurrent assertion specification is based on clock ticks and uses a generalized notion of clock cycles. The definition of a clock is explicitly specified by the user and can vary from one expression to another.

In an assertion, the sampled value is the only valid value of a variable during a clock tick. Figure 16-1 shows the values of a variable as the clock progresses. The value of signal `req` is low at clock ticks 1 and 2. At clock tick 3, the value is sampled as high and remains high until clock tick 6. The sampled value of variable `req` at clock tick 6 is low and remains low up to and including clock tick 9. Notice that the simulation value transitions to high at clock tick 9. However, the sampled value at clock tick 9 is low.

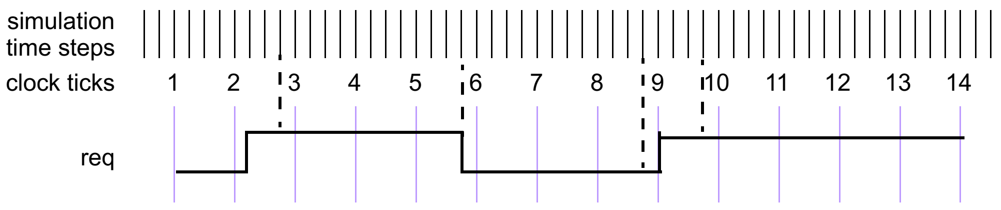


**Figure 16-1—Sampling a variable in a simulation time step**

An expression used in an assertion is always tied to a clock definition, except for the use of constant or automatic values from procedural code (see 16.14.6). The sampled values are used to evaluate value change expressions or Boolean subexpressions that are required to determine a match of a sequence.

For concurrent assertions, the following statements apply:

— It is important that the defined clock behavior be glitch free. Otherwise, wrong values can be sampled.

— If a variable that appears in the expression for clock also appears in an expression with an assertion, the values of the two usages of the variable can be different. The current value of the variable is used in the clock expression, while the sampled value of the variable is used within the assertion.

The clock expression that controls evaluation of a sequence can be more complex than just a single signal name. Expressions such as `clk` **`iff`** `gating_signal` can be used to represent a gated clock. Other more complex expressions are possible. However, in order to verify proper behavior of the system and conform as closely as possible to truly cycle-based semantics, the user should ensure that the clock expression is glitch-free and only transitions once at any simulation time. (See 9.4.2.3 for the reasons that the expression `clk` **`iff`** `gating_signal` avoids the potential glitch in the expression `clk && gating_signal`.)

A reference to `$global_clock` (see 14.14) is understood to be a reference to a *clocking_event* defined in a **`global clocking`** declaration. A global clock behaves just as any other clocking event. In formal verification, however, `$global_clock` has additional significance, as it is considered to be the primary system clock (see F.3.1 ). Thus, in the following example:

```
global clocking @clk; endclocking
   ...
assert property(@$global_clock a);
```

the assertion states that `a` is true at each tick of the global clock. This assertion is logically equivalent to:

```
assert property(@clk a);
```

An example of a concurrent assertion is as follows:

```
base_rule1: assert property (cont_prop(rst,in1,in2)) $display("%m, passing");
            else $display("%m, failed");
```

The keyword **property** distinguishes a concurrent assertion from an immediate assertion. The syntax of concurrent assertions is discussed in 16.14.

## 16.6 Boolean expressions

The outcome of the evaluation of an expression is Boolean and is interpreted the same way as an expression is interpreted in the condition of a procedural **if** statement. In other words, if the expression evaluates to `X`, `Z`, or `0`, then it is interpreted as being false. Otherwise, it is true.

Expressions that appear in concurrent assertions shall satisfy the following requirements:

— An expression shall result in a type that is cast compatible with an integral type. Subexpressions need not meet this requirement as long as the overall expression is cast compatible with an integral type.

— Elements of dynamic arrays, queues, and associative arrays that are sampled for assertion expression evaluation may get removed from the array or the array may get resized before the assertion expression is evaluated. These specific array elements sampled for assertion expression evaluation shall continue to exist within the scope of the assertion until the assertion expression evaluation completes.

— Expressions that appear in procedural concurrent assertions may reference automatic variables as described in 16.14.6.1. Otherwise, expressions in concurrent assertions shall not reference automatic variables.

— Expressions shall not reference non-static class properties or methods.

— Expressions shall not reference variables of the **chandle** data type.

— Sequence match items with a local variable as the *variable_lvalue* may use the C assignment, increment, and decrement operators. Otherwise, evaluation of an expression shall not have any side effects (e.g., the increment and decrement operators are not allowed).

— Functions that appear in expressions shall not contain output or **ref** arguments (**const ref** is allowed).

— Functions shall be automatic (or preserve no state information) and have no side effects.

Care should be taken when accessing large data structures, especially large dynamic data structures, in concurrent assertions. Some types of access may require creating a copy of the entire data structure, which could incur a significant performance penalty. The following example illustrates how the need to copy an entire data structure may arise. In `p1` only a single byte of `q` must be sampled by the assertion, and the location of that byte is constant. However, in `p2` there will be multiple active threads with potentially different values of `l_b`. This increases the difficulty of determining which bytes of `q` to sample and likely results in sampling all of `q`.

```
bit a;
integer b;
byte q[$];

property p1;
   $rose(a) |-> q[0];
endproperty

property p2;
   integer l_b;
   ($rose(a), l_b = b) |-> ##[3:10] q[l_b];
endproperty
```

There are two places where Boolean expressions occur in concurrent assertions. They are as follows:

— In a sequence or property expression
— In the disable condition inferred for an assertion, specified either in a top-level **disable iff** clause (see 16.12) or in a **default disable iff** declaration (see 16.15)

The Boolean expressions used in defining a sequence or property expression shall be evaluated over the sampled values of all variables. The preceding rule shall not, however, apply to expressions in a clocking event (see 16.5).

The expressions in a disable condition are evaluated using the current values of variables (not sampled) and may contain the sequence Boolean method `triggered`. They shall not contain any reference to local variables or to the sequence method `matched`.

Assertions that perform checks based on time values should capture these values in the same context. It is not recommended to capture time outside of the assertion. Time should be captured within the assertion using local variables. The following example illustrates how a problem may arise when capturing time in different contexts. In property `p1`, a time value, `t`, is captured in a procedural context based on the current value of `count`. Within the assertion, a comparison is made between the time value `t` and the time value returned by `$realtime` in the assertion context based on the sampled value of `count`. In both contexts, `$realtime` returns the current time value. As a result, the comparison between values of time captured in the different contexts yields an inconsistent result. The inconsistency results in the computation for `p1` checking the amount of time that elapses between 8 periods of `clk` instead of the intended 7. In property `p2`, both time values are captured within the assertion context. This strategy yields a consistent result.

```
bit [2:0] count;
realtime t;

initial count = 0;
always @(posedge clk) begin
   if (count == 0) t = $realtime; //capture t in a procedural context
   count++;
end

property p1;
   @(posedge clk)
   count == 7 |-> $realtime – t < 50.5;
endproperty

property p2;
   realtime l_t;
   @(posedge clk)
   (count == 0, l_t = $realtime) ##1 (count == 7)[->1] |->
      $realtime – l_t < 50.5;
endproperty
```

## 16.7 Sequences

sequence_expr ::= *// from A.2.10*
  cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
 | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
 | expression_or_dist [ boolean_abbrev ]
 | sequence_instance [ sequence_abbrev ]
 | **(** sequence_expr {**,** sequence_match_item } **)** [ sequence_abbrev ]

```
    | sequence_expr and sequence_expr
    | sequence_expr intersect sequence_expr
    | sequence_expr or sequence_expr
    | first_match ( sequence_expr {, sequence_match_item} )
    | expression_or_dist throughout sequence_expr
    | sequence_expr within sequence_expr
    | clocking_event sequence_expr
cycle_delay_range ::=
      ## constant_primary
    | ## [ cycle_delay_const_range_expression ]
    | ##[*]
    | ##[+]
sequence_match_item ::=
      operator_assignment
    | inc_or_dec_expression
    | subroutine_call
sequence_instance ::=
      ps_or_hierarchical_sequence_identifier [ ( [ sequence_list_of_arguments ] ) ]
sequence_list_of_arguments ::=
      [sequence_actual_arg] { , [sequence_actual_arg] } { , . identifier ( [sequence_actual_arg] ) }
    | . identifier ( [sequence_actual_arg] ) { , . identifier ( [sequence_actual_arg] ) }
sequence_actual_arg ::=
      event_expression
    | sequence_expr
boolean_abbrev ::=
      consecutive_repetition
    | non_consecutive_repetition
    | goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::=
      [* const_or_range_expression ]
    | [*]
    | [+]
non_consecutive_repetition ::= [= const_or_range_expression ]
goto_repetition ::= [-> const_or_range_expression ]
const_or_range_expression ::=
      constant_expression
    | cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
      constant_expression : constant_expression
    | constant_expression : $
expression_or_dist ::= expression [ dist { dist_list } ]
```

*Syntax 16-3—Sequence syntax (excerpt from Annex A)*

Properties are often constructed out of sequential behaviors. The **sequence** feature provides the capability to build and manipulate sequential behaviors. The simplest sequential behaviors are linear. A linear sequence is a finite list of SystemVerilog Boolean expressions in a linear order of increasing time. The linear sequence is said to match along a finite interval of consecutive clock ticks provided the first Boolean expression evaluates to true at the first clock tick, the second Boolean expression evaluates to true at the second clock tick, and so forth, up to and including the last Boolean expression evaluating to true at the last clock tick. A single Boolean expression is an example of a simple linear sequence, and it matches at a single clock tick provided the Boolean expression evaluates to true at that clock tick.

More complex sequential behaviors are described by SystemVerilog sequences. A sequence is a regular expression over the SystemVerilog Boolean expressions that concisely specifies a set of zero, finitely many, or infinitely many linear sequences. If at least one of the linear sequences from this set matches along a finite interval of consecutive clock ticks, then the sequence is said to match along that interval.

A property may involve checking of one or more sequential behaviors beginning at various times. An attempted evaluation of a sequence is a search for a match of the sequence beginning at a particular clock tick. To determine whether such a match exists, appropriate Boolean expressions are evaluated beginning at the particular clock tick and continuing at each successive clock tick until either a match is found or it is deduced that no match can exist.

A sequence may admit an *empty match*, a match that occurs over an interval of length 0. (See a formal definition at 16.12.22, and see 16.9.2.1 for more discussion of empty matches). An *end point* of a sequence is the time step of any nonempty match of the sequence. An end point is reached whenever the ending clock tick of a match of the sequence is reached, regardless of the starting clock tick of the match. A *match point* includes both empty and nonempty matches, and is reached either at an end point or, in the case of an empty match, at the length-0 time interval at the beginning of the time step when sequence evaluation begins. A sequence that admits only empty matches is referred to as an *empty sequence*.

Sequences can be composed by concatenation, analogous to a concatenation of lists. The concatenation specifies a delay, using `##`, from the end of the first sequence until the beginning of the second sequence.

The syntax for sequence concatenation is shown in Syntax 16-4.

```
sequence_expr ::=                                                         // from A.2.10
     cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
   | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
   ...
cycle_delay_range ::=
     ## constant_primary
   | ## [ cycle_delay_const_range_expression ]
   | ##[*]
   | ##[+]
cycle_delay_const_range_expression ::=
     constant_expression : constant_expression
   | constant_expression : $
```

*Syntax 16-4—Sequence concatenation syntax (excerpt from Annex A)*

In this syntax, the following statements apply:

— *constant_primary* is a *constant_expression*, which is computed at compile time and shall result in an integer value. Furthermore, *constant_expression* and the bounds in *cycle_delay_const_range_expression* can only be 0 or greater.

— The **$** token is used to indicate a finite, but unbounded, maximum.

— `##[*]` is used as an equivalent representation of `##[0:$]`.

— `##[+]` is used as an equivalent representation of `##[1:$]`.

— When a range is specified with two expressions, the second expression shall be greater than or equal to the first expression.

— In a *cycle_delay_range*, it shall be illegal for a *constant_primary* to contain a *constant_mintypmax_expression* that is not also a *constant_expression*.

The context in which a sequence occurs determines when the sequence is evaluated. The first expression in a sequence is checked at the first occurrence of the clock tick at or after the expression that triggered evaluation of the sequence. Each successive element (if any) in the sequence is checked at the next subsequent occurrence of the clock.

A ## followed by a number or range specifies the delay from the current clock tick to the beginning of the sequence that follows. The delay ##1 indicates that the beginning of the sequence that follows is one clock tick later than the current clock tick. The delay ##0 indicates that the beginning of the sequence that follows is at the same clock tick as the current clock tick.

When used as a concatenation between two sequences, the delay is from the end of the first sequence to the beginning of the second sequence. The delay ##1 indicates that the beginning of the second sequence is one clock tick later than the end of the first sequence. The delay ##0 indicates that the beginning of the second sequence is at the same clock tick as the end of the first sequence.

In the examples in this clause, `` `true `` is a Boolean expression that always evaluates to `1'b1` and is used for visual clarity. It is defined as follows:

```
`define true 1'b1

##0 a       // means a
##1 a       // means `true ##1 a
##2 a       // means `true ##1 `true ##1 a
##[0:3]a    // means (a) or (`true ##1 a) or (`true ##1 `true ##1 a) or
            // (`true ##1 `true ##1 `true ##1 a)
a ##2 b     // means a ##1 `true ##1 b
```

The sequence

```
req ##1 gnt ##1 !req
```

specifies that `req` be true on the current clock tick, `gnt` shall be true on the first subsequent tick, and `req` shall be false on the next clock tick after that. The `##1` operator specifies one clock tick separation. A delay of more than one clock tick can be specified, as in the following:

```
req ##2 gnt
```

This specifies that `req` shall be true on the current clock tick, and `gnt` shall be true on the second subsequent clock tick, as shown in Figure 16-2.

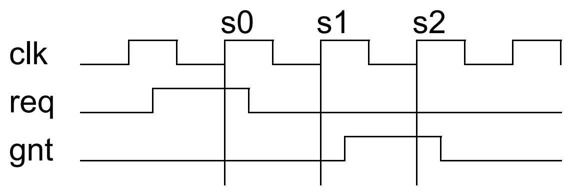


**Figure 16-2—Concatenation of sequences**

The following specifies that signal `b` shall be true on the Nth clock tick after signal `a`:

```
a ##N b     // check b on the Nth sample
```

To specify a concatenation of overlapped sequences, where the end point of one sequence coincides with the start of the next sequence, a value of `0` is used, as follows:

```
a ##1 b ##1 c // first sequence seq1
d ##1 e ##1 f // second sequence seq2
(a ##1 b ##1 c) ##0 (d ##1 e ##1 f) // overlapped concatenation
```

In the preceding example, `c` must be true at the end point of sequence `seq1`, and `d` must be true at the start of sequence `seq2`. When concatenated with `0` clock tick delay, `c` and `d` must be true at the same time, resulting in a concatenated sequence equivalent to the following:

```
a ##1 b ##1 c&&d ##1 e ##1 f
```

It should be noted that no other form of overlapping between the sequences can be expressed using the concatenation operation.

In cases where the delay can be any value in a range, a time window can be specified as follows:

```
req ##[4:32] gnt
```

In the preceding case, signal `req` must be true at the current clock tick, and signal `gnt` must be true at some clock tick between the 4th and the 32nd clock tick after the current clock tick.

The time window can extend to a finite, but unbounded, range by using `$` as in the following example:

```
req ##[4:$] gnt
```

A sequence can be unconditionally extended by concatenation with `` `true``.

```
a ##1 b ##1 c ##3 `true
```

After satisfying signal `c`, the sequence length is extended by three clock ticks. Such adjustments in the length of sequences can be required when complex sequences are constructed by combining simpler sequences.

## 16.8 Declaring sequences

A named **`sequence`** may be declared in the following:

— A module
— An interface
— A program
— A **`clocking`** block
— A package
— A compilation-unit scope
— A checker
— A generate block

Named sequences are declared using Syntax 16-5.

```
assertion_item_declaration ::=                                              // from A.2.10
    ...
  | sequence_declaration
sequence_declaration ::=
    sequence sequence_identifier [ ( [ sequence_port_list ] ) ] ;
        { assertion_variable_declaration }
        sequence_expr [ ; ]
    endsequence [ : sequence_identifier ]
sequence_port_list ::=
    sequence_port_item {, sequence_port_item}
sequence_port_item ::=
    { attribute_instance } [ local [ sequence_lvar_port_direction ] ] sequence_formal_type
        formal_port_identifier {variable_dimension} [ = sequence_actual_arg ]
sequence_lvar_port_direction ::= input | inout | output
sequence_formal_type ::=
    data_type_or_implicit
  | sequence
  | untyped
formal_port_identifier ::= identifier                                        // from A.9.3
sequence_instance ::=                                                        // from A.2.10
    ps_or_hierarchical_sequence_identifier [ ( [ sequence_list_of_arguments ] ) ]
sequence_list_of_arguments ::=
    [sequence_actual_arg] { , [sequence_actual_arg] } { , . identifier ( [sequence_actual_arg] ) }
  | . identifier ( [sequence_actual_arg] ) { , . identifier ( [sequence_actual_arg] ) }
sequence_actual_arg ::=
    event_expression
  | sequence_expr
assertion_variable_declaration ::=
    var_data_type list_of_variable_decl_assignments ;
```

*Syntax 16-5—Sequence declaration syntax (excerpt from Annex A)*

A named sequence may be declared with formal arguments in the optional *sequence_port_list*.

A formal argument may be typed by specifying the type prior to the *formal_port_identifier* of the formal argument. A type shall apply to all formal arguments whose identifiers both follow the type and precede the next type, if any, specified in the port list. Rules particular to the specification and use of typed formal arguments are discussed in 16.8.1.

Rules particular to the specification and use of local variable formal arguments are discussed in 16.8.2.

A formal argument is said to be *untyped* if there is no type specified prior to its *formal_port_identifier* in the port list. There is no default type for a formal argument.

The supported data types for sequence formal arguments are the types that are allowed for operands in assertion expressions (see 16.6) and the keyword **untyped**.

A default actual argument may be specified for a formal argument in the optional associated declaration assignment. The *default_expression* is resolved in the scope containing the sequence declaration. Requirements for the type of the default actual argument of a typed formal argument are described in 16.8.1.

The default actual argument of an untyped formal argument may be of any type provided its substitution results in a valid sequence as described in the rewriting algorithm (see F.4.1).

A formal argument may be referenced in the body of the declaration of the named sequence. A reference to a formal argument may be written in place of various syntactic entities, such as the following:

- identifier
- expression
- sequence_expr
- event_expression
- terminal $ in a *cycle_delay_const_range_expression*

A named sequence may be instantiated by referencing its name. The reference may be a hierarchical name (see 23.6). A named sequence may be instantiated anywhere that a *sequence_expr* may be written, including prior to its declaration. A named sequence may also be instantiated as part of a *sequence_method_call* (see 16.9.11, 16.13.5) or as an event_expression (see 9.4.2.4). It shall be an error if a cyclic dependency among named sequences results from their instantiations. A cyclic dependency among named sequences results if, and only if, there is a cycle in the directed graph whose nodes are the named sequences and whose edges are defined by the following rule: there is a directed edge from one named sequence to a second named sequence if, and only if, either the first named sequence instantiates the second named sequence within its declaration, including an instance within the declaration of a default actual argument, or there is an instance of the first named sequence that instantiates the second named sequence within an actual argument.

In an instance of a named sequence, actual arguments may be passed to formal arguments. The instance shall provide an actual argument in the list of arguments for each formal argument that does not have a default actual argument declared. The instance may provide an actual argument for a formal argument that has a default actual argument, thereby overriding the default. Actual arguments in the list of arguments may be bound to formal arguments by name or by position.

The terminal $ may be an actual argument in an instance of a named sequence, either declared as a default actual argument or passed in the list of arguments of the instance. If $ is an actual argument, then the corresponding formal argument shall be untyped and each of its references either shall be an upper bound in a *cycle_delay_const_range_expression* or shall itself be an actual argument in an instance of a named sequence.

If an instance of a named sequence is within the scope of a local variable (see 16.10), then an actual argument in the list of arguments of the instance may reference the local variable.

Names other than formal arguments that appear in the declaration of a named sequence, including those that appear in default actual arguments, shall be resolved according to the scoping rules from the scope of the declaration of the named sequence. Names appearing in actual arguments in the list of arguments of the instance shall be resolved according to the scoping rules from the scope of the instance of the named sequence.

The sequential behavior and matching semantics of an instance of a named sequence are the same as those of the flattened sequence that is obtained from the body of the declaration of the named sequence by the rewriting algorithm defined in F.4.1. The rewriting algorithm substitutes actual arguments for references to the corresponding formal arguments in the body of the declaration of the named sequence. The rewriting algorithm does not itself account for name resolution and assumes that names have been resolved prior to the substitution of actual arguments. If the flattened sequence is not legal, then the instance is not legal and there shall be an error.

The substitution of an actual argument for a reference to the corresponding untyped formal argument in the rewriting algorithm retains the actual as an expression term. An actual argument shall be enclosed in parentheses and shall be cast to its self-determined type before being substituted for a reference to the corresponding formal argument unless one of the following conditions holds:

— The actual argument is $.

— The actual argument is a *variable_lvalue*.

If the result of the rewriting algorithm is an invalid sequence, an error shall occur.

For example, a reference to an untyped formal argument may appear in the specification of a *cycle_delay_range*, a *boolean_abbrev*, or a *sequence_abbrev* (see 16.9.2) only if the actual argument is an elaboration-time constant. The following example illustrates such usage of formal arguments:

```
sequence delay_example(x, y, min, max, delay1);
   x ##delay1 y[*min:max];
endsequence

// Legal
a1: assert property (@(posedge clk) delay_example(x, y, 3, $, 2));

int z, d;

// Illegal: z and d are not elaboration-time constants
a2_illegal: assert property (@(posedge clk) delay_example(x, y, z, $, d));
```

In the following example, named sequences `s1` and `s2` are evaluated on successive **posedge** events of `clk`. The named sequence `s3` is evaluated on successive **negedge** events of `clk`. The named sequence `s4` is evaluated on successive alternating **posedge** and **negedge** events of `clk`.

```
sequence s1;
   @(posedge clk) a ##1 b ##1 c;
endsequence
sequence s2;
   @(posedge clk) d ##1 e ##1 f;
endsequence
sequence s3;
   @(negedge clk) g ##1 h ##1 i;
endsequence
sequence s4;
   @(edge clk) j ##1 k ##1 l;
endsequence
```

Another example of named sequence declaration, which includes arguments, follows:

```
sequence s20_1(data,en);
   (!frame && (data==data_bus)) ##1 (c_be[0:3] == en);
endsequence
```

Named sequence `s20_1` does not specify a clock. In this case, a clock would be inherited from some external source, such as a **property** or an **assert** statement. An example of instantiating a named sequence is shown as follows:

```
sequence s;
   a ##1 b ##1 c;
endsequence
sequence rule;
   @(posedge sysclk)
   trans ##1 start_trans ##1 s ##1 end_trans;
endsequence
```

Named sequence `rule` in the preceding example is equivalent to the following:

```
sequence rule;
   @(posedge sysclk)
   trans ##1 start_trans ##1 (a ##1 b ##1 c) ##1 end_trans ;
endsequence
```

The following example illustrates an illegal cyclic dependency among the named sequences `s1` and `s2`:

```
sequence s1;
   @(posedge sysclk) (x ##1 s2);
endsequence
sequence s2;
   @(posedge sysclk) (y ##1 s1);
endsequence
```

### 16.8.1 Typed formal arguments in sequence declarations

The data type specified for a formal argument of a sequence may be the keyword **`untyped`**. A formal argument shall be untyped (see 16.8) if its data type is **`untyped`**. The semantics of binding an actual argument expression to a formal with a data type of **`untyped`** shall be the same as the semantics for an untyped formal. The keyword **`untyped`** shall be used if an untyped formal argument follows a data type in the formal argument list.

If a formal argument of a named sequence is typed, then the type shall be **`sequence`** or one of the types allowed in 16.6. The following rules apply to typed formal arguments and their corresponding actual arguments, including default actual arguments declared in a named sequence:

a) If the formal argument is of type **`sequence`**, then the actual argument shall be a *sequence_expr*. A reference to the formal argument of type **`sequence`** shall either be in a place where a *sequence_expr* is legal, or as an operand of sequence methods `triggered` and `matched`.

b) If the formal argument is of type **`event`**, then the actual argument shall be an *event_expression* and each reference to the formal argument shall be in a place where an *event_expression* may be written.

c) Otherwise, the self-determined result type of the actual argument shall be cast compatible (see 6.22.4) with the type of the formal argument. If the actual argument is a *variable_lvalue*, references to the formal shall be considered as having the formal's type with any assignment to the formal being treated as though there was a subsequent assignment from the formal to the actual argument. If the actual argument is not a *variable_lvalue*, the actual argument shall be cast to the type of the formal argument before being substituted for a reference to the formal argument in the rewriting algorithm (see F.4.1).

For example, a Boolean expression may be passed as an actual argument to a formal argument of type **`sequence`** because a Boolean expression is a *sequence_expr*. A formal argument of type **`sequence`** may not be referenced as the *expression_or_dist* operand of a *goto_repetition* (see 16.9.2), regardless of the corresponding actual argument, because a *sequence_expr* may not be written in that position.

A reference to a typed formal argument within a *sequence_match_item* (see 16.10) shall not stand as the *variable_lvalue* in either an *operator_assignment* or an *inc_or_dec_expression* unless the formal argument is a local variable argument (see 16.8.2, 16.12.19).

Two examples of declaring formal arguments follow. All of the formal arguments of `s1` are untyped. The formal arguments `w` and `y` of `s2` are untyped, while the formal argument `x` has type **`bit`**.

```
sequence s1(w, x, y);
   w ##1 x ##[2:10] y;
endsequence
```

```
sequence s2(w, y, bit x);
   w ##1 x ##[2:10] y;
endsequence
```

The following instances of s1 and s2 are equivalent:

```
s1(.w(a), .x(bit'(b)), .y(c))
s2(.w(a), .x(b), .y(c))
```

In the instance of s2 above, if b happens to be 8 bits wide then it will be cast to **bit** by truncation since it is being passed to a formal argument of type **bit**. Similarly, if an expression of type **bit** is passed as actual argument to a formal argument of type **byte**, then the expression is extended to a **byte**.

If a reference to a typed formal argument appears in the specification of a *cycle_delay_range*, a *boolean_abbrev*, or a *sequence_abbrev* (see 16.9.2), then the type of the formal argument shall be **shortint**, **int**, or **longint**. The following example illustrates such usage of formal arguments:

```
sequence delay_arg_example (max, shortint delay1, delay2, min);
   x ##delay1 y[*min:max] ##delay2 z;
endsequence

parameter my_delay=2;
cover property (delay_arg_example($, my_delay, my_delay-1, 3));
```

The cover property in the preceding example is equivalent to the following:

```
cover property (x ##2 y[*3:$] ##1 z);
```

The following shows an example of a formal argument with **event** type:

```
sequence event_arg_example (event ev);
   @(ev) x ##1 y;
endsequence

cover property (event_arg_example(posedge clk));
```

The cover property in the preceding example is equivalent to the following:

```
cover property (@(posedge clk) x ##1 y));
```

If the intent is to pass as actual argument an expression that will be combined with an *edge_identifier* to create an *event_expression*, then the formal argument shall not be typed with type **event**. The following example illustrates such usage:

```
sequence event_arg_example2 (reg sig);
   @(posedge sig) x ##1 y;
endsequence

cover property (event_arg_example2(clk));
```

The cover property in the preceding example is equivalent to the following:

```
cover property (@(posedge clk) x ##1 y));
```

Another example, in which a local variable is used to sample a formal argument, shows how to get the effect of “pass by value.” Pass by value is not currently supported as a mode of argument passing.

```
sequence s(bit a, bit b);
   bit loc_a;
   (1'b1, loc_a = a) ##0
   (t == loc_a) [*0:$] ##1 b;
endsequence
```

### 16.8.2 Local variable formal arguments in sequence declarations

This subclause describes mechanisms for declaring local variable formal arguments and rules specific to their use. Local variable formal arguments are special cases of local variables (see 16.10).

A formal argument of a named sequence may be designated as a local variable argument by specifying the keyword **local** in the port item, followed optionally by one of the directions **input**, **inout**, or **output**. If no direction is specified explicitly, then the direction **input** shall be inferred. If the keyword **local** is specified in a port item, then the type of that argument shall be specified explicitly in that port item and shall not be inferred from a previous argument. The type of a local variable argument shall be one of the types allowed in 16.6. If one of the directions **input**, **inout**, or **output** is specified in a port item, then the keyword **local** shall be specified in that port item.

The designation of a formal argument as a local variable argument of a given direction and type shall apply to subsequent identifiers in the port list as long as none of the subsequent port items specifies the keyword **local** or an explicit type. In other words, if a port item consists only of an identifier and if the nearest preceding argument with an explicitly specified type also specifies the keyword **local**, then the port item is a local variable argument with the same direction and type as that preceding argument.

If a local variable formal argument has direction **input**, then a default actual argument may be specified for that argument in the optional declaration assignment in the port item, subject to the rules for default actual arguments described in 16.8. It shall be illegal to specify a default actual argument for a local variable argument of direction **inout** or **output**.

An example showing legal declaration of a named sequence using local variable formal arguments is as follows:

```
logic b_d, d_d;
sequence legal_loc_var_formal (
   local inout logic a,
   local logic b = b_d, // input inferred, default actual argument b_d
               c,       // local input logic inferred, no default
                        // actual argument
               d = d_d, // local input logic inferred, default actual
                        // argument d_d
   logic e, f           // e and f are not local variable formal arguments
);
   logic g = c, h = g || d;
   ...
endsequence
```

An example showing illegal declaration of a named sequence using local variable formal arguments is shown as follows:

```
sequence illegal_loc_var_formal (
   output logic     a,       // illegal: local must be specified with
                             // direction
```

```
    local inout logic b,
                      c = 1'b0,// default actual argument illegal for inout
    local             d = expr,// illegal: type must be specified explicitly
    local event       e,       // illegal: event is a type disallowed in
                               // 16.6
    local logic       f = g    // g shall not refer to the local variable
                               // below and must be resolved upward from
                               // this declaration
 );
    logic g = b;
    ...
 endsequence
```

In general, a local variable formal argument behaves in the same way as a local variable declared in an *assertion_variable_declaration*. The rules in 16.10 for assigning to and referencing local variables, including the rules of local variable flow, apply to local variable formal arguments with the following provisions:

— Without further specification, the term *local variable* shall mean either a local variable formal argument or a local variable declared in an *assertion_variable_declaration*.

— At the beginning of each evaluation attempt of an instance of a named sequence, a new copy of each of its local variable formal arguments shall be created.

— A local variable formal argument with direction **input** or **inout** shall be treated like a local variable declared in an *assertion_variable_declaration* with a declaration assignment. The initial value for the local variable formal argument is provided by the associated actual argument for the instance. The self-determined result type of the actual argument shall be cast compatible (see 6.22.4) with the type of the local variable formal argument. The value of the actual argument shall be cast to the type of the local variable formal argument before being assigned as initial value to the local variable formal argument. This assignment is referred to as the *initialization assignment* of the local variable formal argument. Initialization of all input and inout local variable formal arguments shall be performed before initialization of any local variable declared in an *assertion_variable_declaration*. The expression of a declaration assignment to a local variable declared in an *assertion_variable_declaration* may refer to a local variable formal argument of direction **input** or **inout**.

— If a local variable formal argument of direction **input** or **inout** is bound to an actual argument in the argument list of an instance and if the actual argument references a local variable, then it shall be an error if that local variable is unassigned at the point of the reference in the context of the instance.

— A local variable formal argument of direction **output** shall be unassigned at the beginning of the evaluation attempt of the instance.

— The entire actual argument expression bound to an **inout** or **output** local variable formal argument shall itself be a reference to a local variable whose scope includes the instance and with whose type the type of the local variable formal argument is cast compatible. It shall be an error if references to the same local variable are bound as actual arguments to two or more local variable formal arguments of direction **inout** or **output**. It shall be an error if there exists a match of the named sequence for which an inout or output local variable formal argument is unassigned at the completion of the match. At the completion of a match of the instance of the named sequence, the value of the inout or output local variable formal argument shall be cast to the type of and assigned to the local variable whose reference is the associated actual argument. If multiple threads of evaluation of the instance of the named sequence match, then multiple threads of evaluation shall continue in the instantiation context, each with its own copy of the actual argument local variable. For each matching thread of the instance of the named sequence, at the completion of the match of that thread the value of the local variable formal argument in that thread shall be cast to the type of and assigned to the associated copy of the actual argument local variable.

— It shall be an error for an instance of a named sequence with an **inout** or **output** local variable formal argument to admit an empty match (see 16.12.22).

— It shall be an error to apply any of the sequence methods `triggered` (see 16.9.11, 16.13.6) or `matched` (see 16.13.5) to an instance of a named sequence with an input or inout local variable formal argument.

The following example illustrates legal usage of a local variable formal argument:

```
sequence sub_seq2(local inout int lv);
   (a ##1 !a, lv += data_in)
   ##1 !b[*0:$] ##1 b && (data_out == lv);
endsequence
sequence seq2;
   int v1;
   (c, v1 = data)
   ##1 sub_seq2(v1)  // lv is initialized by assigning it the value of v1;
                     // when the instance sub_seq2(v1) matches, v1 is
                     // assigned the value of lv
   ##1 (do1 == v1);
endsequence
```

The matching behavior of `seq2` is equivalent to that of `seq2_inlined` as follows:

```
sequence seq2_inlined;
   int v1, lv;
   (c, v1 = data) ##1
   (
      (1, lv = v1) ##0
      (a ##1 !a, lv += data_in)
      ##1 (!b[*0:$] ##1 b && (data_out == lv), v1 = lv)
   )
   ##1 (do1 == v1);
endsequence
```

Untyped arguments provide an alternative mechanism for passing local variables to an instance of a subsequence, including the capability to assign to the local variable in the subsequence and later reference the value assigned in the instantiation context (see 16.10).

## 16.9 Sequence operations

### 16.9.1 Operator precedence

Operator precedence and associativity are listed in Table 16-1. The highest precedence is listed first.

**Table 16-1—Operator precedence and associativity**

| SystemVerilog expression operators | Associativity |
|---|---|
| `[* ]  [= ]  [-> ]` | — |
| `##` | Left |
| **throughout** | Right |
| **within** | Left |
| **intersect** | Left |

**Table 16-1—Operator precedence and associativity *(continued)***

| SystemVerilog expression operators | Associativity |
|---|---|
| **and** | Left |
| **or** | Left |

### 16.9.2 Repetition in sequences

The syntax for sequence repetition is shown in Syntax 16-6.

```
sequence_expr ::=                                                        // from A.2.10
   ...
   | expression_or_dist [ boolean_abbrev ]
   | sequence_instance [ sequence_abbrev ]
   | ( sequence_expr {, sequence_match_item} ) [ sequence_abbrev ]
   ...
boolean_abbrev ::=
      consecutive_repetition
   | non_consecutive_repetition
   | goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::=
      [* const_or_range_expression ]
   | [*]
   | [+]
non_consecutive_repetition ::= [= const_or_range_expression ]
goto_repetition ::= [-> const_or_range_expression ]
const_or_range_expression ::=
      constant_expression
   | cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
      constant_expression : constant_expression
   | constant_expression : $
```

*Syntax 16-6—Sequence repetition syntax (excerpt from Annex A)*

The number of iterations of a repetition can either be specified by exact count or be required to fall within a finite range. If specified by exact count, then the number of iterations is defined by a non-negative integer constant expression. If required to fall within a finite range, then the minimum number of iterations is defined by a non-negative integer constant expression; and the maximum number of iterations either is defined by a non-negative integer constant expression or is $, indicating a finite, but unbounded, maximum.

If both the minimum and maximum numbers of iterations are defined by non-negative integer constant expressions (see 11.2.1), then the minimum number shall be less than or equal to the maximum number.

See 16.9.2.1 for discussion of the special case where the number of iterations is 0.

The following three kinds of repetition are provided:

— *Consecutive repetition* ( [**const_or_range_expression*] ): Consecutive repetition specifies finitely many iterative matches of the operand sequence, with a delay of one clock tick from the end of one match to the beginning of the next. The overall repetition sequence matches at the end of the last iterative match of the operand. `[*]` is an equivalent representation of `[*0:$]` and `[+]` is an equivalent representation of `[*1:$]`.

— *Goto repetition* ( `[->`*const_or_range_expression*`]` ): Goto repetition specifies finitely many iterative matches of the operand Boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at the last iterative match of the operand.

— *Nonconsecutive repetition* ( `[=`*const_or_range_expression*`]` ): Nonconsecutive repetition specifies finitely many iterative matches of the operand Boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at or after the last iterative match of the operand, but before any later match of the operand.

The effect of consecutive repetition of a subsequence within a sequence can be achieved by explicitly iterating the subsequence, as follows:

```
a ##1 b ##1 b ##1 b ##1 c
```

Using the consecutive repetition operator `[*3]`, which indicates three iterations, this sequential behavior is specified more succinctly:

```
a ##1 b [*3] ##1 c
```

A consecutive repetition specifies that the operand sequence shall match a specified number of times. The consecutive repetition operator `[*N]` specifies that the operand sequence must match *N* times in succession. For example:

```
a [*3] means a ##1 a ##1 a
```

The syntax allows the combination of a delay and repetition in the same sequence. The following are both allowed:

```
a ##3 (b[*3])  // means a ##1 `true ##1 `true ##1 (b ##1 b ##1 b)
(a ##2 b)[*3]  // means (a ##2 b) ##1 (a ##2 b) ##1 (a ##2 b),
               // which in turn means
      // (a ##1 `true ##1 b) ##1 (a ##1 `true ##1 b) ##1 (a ##1 `true ##1 b)
```

A repetition with a range of minimum `min` and maximum `max` number of iterations can be expressed with the consecutive repetition operator `[* min:max]`.

For example:

```
(a ##2 b)[*1:5]
```

is equivalent to

```
(a ##2 b)
or (a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
```

Similarly,

```
(a[*0:3] ##1 b ##1 c)
```

is equivalent to

```
(b ##1 c)
or (a ##1 b ##1 c)
or (a ##1 a ##1 b ##1 c)
or (a ##1 a ##1 a ##1 b ##1 c)
```

To specify a finite, but unbounded, number of iterations, the dollar sign ( `$` ) is used. For example, the repetition

```
a ##1 b [*1:$] ##1 c
```

matches over an interval of three or more consecutive clock ticks if `a` is true on the first clock tick, `c` is true on the last clock tick, and `b` is true at every clock tick strictly in between the first and the last.

Specifying the number of iterations of a repetition by exact count is equivalent to specifying a range in which the minimum number of repetitions is equal to the maximum number of repetitions. In other words, `seq[*n]` is equivalent to `seq[*n:n]`.

The goto repetition (nonconsecutive exact repetition) takes a Boolean expression rather than a sequence as operand. It specifies the iterative matching of the Boolean expression at clock ticks that are not necessarily consecutive and ends at the last iterative match. For example:

```
a ##1 b [->2:10] ##1 c
```

matches over an interval of consecutive clock ticks provided `a` is true on the first clock tick, `c` is true on the last clock tick, `b` is true on the penultimate clock tick, and, including the penultimate, there are at least 2 and at most 10 not necessarily consecutive clock ticks strictly in between the first and last on which `b` is true. This sequence is equivalent to the following:

```
a ##1 ((!b[*0:$] ##1 b) [*2:10]) ##1 c
```

The nonconsecutive repetition is like the goto repetition except that a match does not have to end at the last iterative match of the operand Boolean expression. The use of nonconsecutive repetition instead of goto repetition allows the match to be extended by arbitrarily many clock ticks provided the Boolean expression is false on all of the extra clock ticks. For example:

```
a ##1 b [=2:10] ##1 c
```

matches over an interval of consecutive clock ticks provided `a` is true on the first clock tick, `c` is true on the last clock tick, and there are at least 2 and at most 10 not necessarily consecutive clock ticks strictly in between the first and last on which `b` is true. This sequence is equivalent to the following:

```
a ##1 ((!b [*0:$] ##1 b) [*2:10]) ##1 !b[*0:$] ##1 c
```

The consecutive repetition operator can be applied to general sequence expressions, but the goto repetition and nonconsecutive repetition operators can be applied only to Boolean expressions. In particular, goto repetition and nonconsecutive repetition cannot be applied to a Boolean expression to which a sequence match item (see 16.10, 16.11) has been attached. For example, the following is a legal sequence expression:

```
(b[->1], v = e)[*2]
```

but the following is illegal:

```
(b, v = e)[->2]
```

#### 16.9.2.1 Repetition, concatenation, and empty matches

Using `0` as a sequence repetition number, an empty sequence (see 16.7) results, as in this example:

```
a [*0]
```

Because empty matches occur over an interval of zero clock ticks and are thus of length 0, they follow the set of concatenation rules specified below. In the following rules, an empty sequence is denoted as *empty*, and another sequence (which may be empty or nonempty) is denoted as *seq*.

— `(empty ##0 seq)` does not result in a match.
— `(seq ##0 empty)` does not result in a match.
— `(empty ##n seq)`, where `n` is greater than 0, is equivalent to `(##(n-1) seq)`.
— `(seq ##n empty)`, where `n` is greater than 0, is equivalent to `(seq ##(n-1) `true)`.

For example, compare the following two sequences:

```
a[*0] ##0 b
`true ##0 b
```

As defined by the preceding rules, the first sequence can never be matched: there is no point in time when the end point of the length-0 sequence `a[*0]` and the length-1 sequence `##0 b` are aligned. In contrast, the second is a well-defined sequence representing the fusion of two sequences of length 1. It will match during any time step when the sampled value of `b` is true.

To apply these rules to a sequence admitting both empty and nonempty matches, rewrite the sequence as the OR of its empty and nonempty cases. Consider the multiple concatenation example:

```
b ##1 a[*0:1] ##2 c
```

This is equivalent to:

```
(b ##1 a[*0] ##2 c) or (b ##1 a[*1] ##2 c)
```

which can be rewritten as:

```
(b ##1 ##1 c) or (b ##1 a ##2 c)
```

or, more concisely:

```
(b ##2 c) or (b ##1 a ##2 c)
```

From this example, we can see that when matching the 0-tick interval specified by the empty case `a[*0]`, the total execution time of the sequence is one less than when using the 1-tick interval specified by `a[*1]`.

### 16.9.3 Sampled value functions

This subclause describes the system functions available for accessing sampled values of an expression. These functions include the capability to access current sampled value, access sampled value in the past, or detect changes in sampled value of an expression. Sampling of an expression is explained in 16.5.1. Local variables (see 16.10) and the sequence method `matched` are not allowed in the argument expressions passed to these functions. The following functions are provided:

```
$sampled ( expression )
$rose ( expression [ , [clocking_event] ] )
$fell ( expression [ , [clocking_event] ] )
$stable ( expression [ , [clocking_event] ] )
$changed ( expression [ , [ clocking_event ] ] )
$past ( expression1 [ , [number_of_ticks ] [ , [expression2 ] [ , [clocking_event]]] ] )
```

The use of these functions is not limited to assertion features; they may be used as expressions in procedural code as well. The clocking event, although optional as an explicit argument to the functions `$past`, `$rose`, `$stable`, `$changed`, and `$fell`, is required for their semantics. The clocking event is used to sample the value of the argument expression.

The function `$sampled` does not use a clocking event.

For a sampled value function other than `$sampled`, the clocking event shall be explicitly specified as an argument or inferred from the code where the function is called. The following rules are used to infer the clocking event:

— If called in an assertion, sequence, or property, the appropriate clocking event as determined by clock flow rules (see 16.13.3) is used.
— Otherwise, if called in a disable condition or a clock expression in an assertion, sequence, or property, it shall be explicitly clocked.
— Otherwise, if called in an action block of an assertion, the leading clock of the assertion is used.
— Otherwise, if called in a procedure, the inferred clock, if any, from the procedural context (see 16.14.6) is used.
— Otherwise, if called outside an assertion, default clocking (see 14.12) is used.

The function `$sampled` returns the sampled value of its argument (see 16.5.1). The use of `$sampled` in concurrent assertions, although allowed, is redundant, as the result of the function is identical to the sampled value of the expression itself used in the assertion. The use of `$sampled` in a **disable iff** clause is meaningful since the disable condition by default is not sampled (see 16.12).

The function `$sampled` is useful to access the value of expressions used in concurrent assertions in their action blocks. Consider the following example:

```
logic a, b, clk;
// ...
a1_bad: assert property (@clk a == b)
   else $error("Different values: a = %b, b = %b", a, b);
a2_ok: assert property (@clk a == b)
   else $error("Different values: a = %b, b = %b",
      $sampled(a), $sampled(b));
```

If in some clock tick the sampled value of `a` is 0 and of `b` is 1, but their current values in the Reactive region of this tick are 0, then assertion `a1_bad` will report `Different values: a = 0, b = 0`. This is because action blocks are evaluated in the Reactive region (see 16.14.1). Assertion `a2_ok` reports the intended message `Different values: a = 0, b = 1` because the values of `a` and `b` in its action block are evaluated in the same context as in the assertion.

The following functions are called value change functions and are provided to detect changes in sampled values: `$rose`, `$fell`, `$stable`, and `$changed`.

A value change function detects a change (or, in the case of `$stable`, lack of change) in the sampled value of an expression. The change (or lack of change) is determined by comparing the sampled value of the expression with the sampled value of the expression from the most recent strictly prior time step in which the clocking event occurred (see 16.5.1 for the definition of sampling in past clock ticks and the following description of `$past` for how past values are evaluated). The result of a value change function is true or false and a call to a value change function may be used as a Boolean expression. The results of value change functions shall be determined as follows:

— `$rose` returns true (`1'b1`) if the LSB of the expression changed to `1`. Otherwise, it returns false (`1'b0`).

— `$fell` returns true (`1'b1`) if the LSB of the expression changed to `0`. Otherwise, it returns false (`1'b0`).

— `$stable` returns true (`1'b1`) if the value of the expression did not change. Otherwise, it returns false (`1'b0`).

— `$changed` returns true (`1'b1`) if the value of the expression changed. Otherwise, it returns false (`1'b0`).

When these functions are called at or before the simulation time step in which the first clocking event occurs, the results are computed by comparing the sampled value of the expression with its default sampled value (see 16.5.1).

Figure 16-3 illustrates two examples of value changes:

— Value change expression `e1` is defined as `$rose(req)`.

— Value change expression `e2` is defined as `$fell(ack)`.

The clock ticks used for sampling the variables are derived from the clock for the property, which is different from the simulation time steps. Assume, for now, that this clock is defined elsewhere. At clock tick 3, `e1` occurs because the value of `req` at clock tick 2 was low and the value at clock tick 3 is high. Similarly, `e2` occurs at clock tick 6 because the value of `ack` was sampled as high at clock tick 5 and sampled as low at clock tick 6.

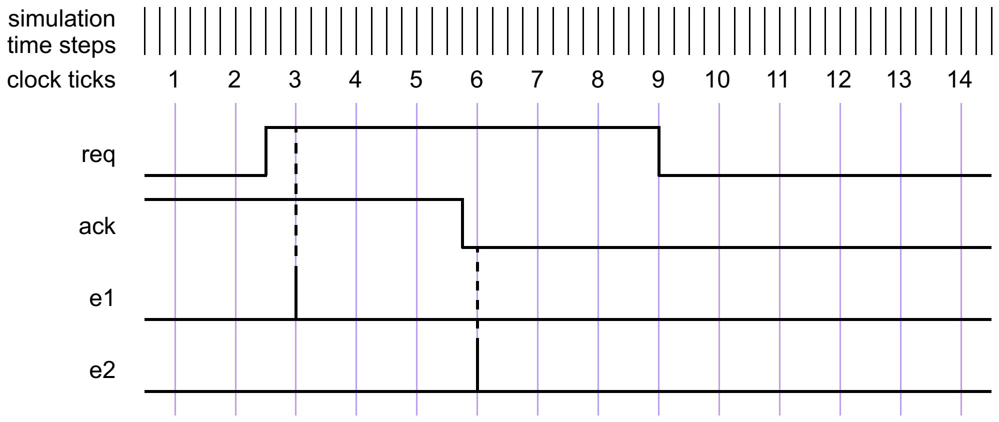


**Figure 16-3—Value change expressions**

The following example illustrates the use of `$rose` in SystemVerilog code outside assertions:

```
always @(posedge clk)
  reg1 <= a & $rose(b);
```

In this example, the clocking event `@(posedge clk)` is applied to `$rose`. `$rose` is true whenever the sampled value of `b` changed to `1` from its sampled value at the previous tick of the clocking event.

Past sampled values can be accessed with the `$past` function. The following three optional arguments are provided:

— *expression2* is used as a gating expression for the clocking event.
— *number_of_ticks* specifies the number of clock ticks in the past.
— *clocking_event* specifies the clocking event for sampling *expression1*.

*expression1* and *expression2* may be any expression allowed in assertions. If *expression2* is not specified, then it defaults to `1'b1`.

*number_of_ticks* shall be 1 or greater and shall be an elaboration-time constant expression. If *number_of_ticks* is not specified, then it defaults to 1.

`$past` returns the sampled value of *expression1* in a particular time step strictly prior to the one in which `$past` is evaluated (see 16.5.1 for the definition of sampling in past clock ticks). If *number_of_ticks* equals *k* and if *ev* is the event expression underlying *clocking_event*, then the particular time step is the *k*th strictly prior time step in which the event *ev* **iff** *expression2* occurred. If there do not exist *k* strictly prior time steps in which the event *ev* **iff** *expression2* occurred, then the value returned from the `$past` function is the default sampled value of *expression1* (see 16.5.1).

The clocking event for `$past` shall be explicitly specified through the *clocking_event* argument or inferred from the code where `$past` is called. The rules for inferring the clocking event are described previously.

When intermediate optional arguments between two arguments are not needed, a comma shall be placed for each omitted argument. For example:

```
$past(in1, , enable);
```

Here, a comma is specified to omit *number_of_ticks*. The default of `1` is used for the empty *number_of_ticks* argument. There is no need to include a comma for the omitted *clocking_event* argument, as it does not fall within the specified arguments.

`$past` can be used in any SystemVerilog expression. An example follows.

```
always @(posedge clk)
  reg1 <= a & $past(b);
```

In this example, the inferred clocking event `@(`**`posedge`**` clk)` is applied to `$past`. `$past` is evaluated in the current occurrence of (**`posedge`** `clk`) and returns the value of `b` sampled at the previous occurrence of (**`posedge`** `clk`).

The function `$past` may refer to automatic variables, e.g., to procedural loop variables, as follows:

```
always @(posedge clk)
   for (int i = 0; i < 4; i ++)
      if (cond[i])
         reg1[i] <= $past(b[i]);
```

According to the definition of the past sampled value (see 16.5.1), `$past` returns at each loop iteration the past value of the `i`-th bit of `b`.

When *expression2* is specified, the sampling of *expression1* is performed based on its clock gated with *expression2*. For example:

```
always @(posedge clk)
  if (enable) q <= d;

always @(posedge clk)
assert property (done |=> (out == $past(q, 2,enable)) );
```

In this example, the sampling of `q` for evaluating `$past` is based on the following clocking expression:

```
posedge clk iff enable
```

The clocking event argument of a sampled value function may be different from the clocking event of the context in which it is called, as determined by the clock resolution (see 16.16).

Consider the following assertions:

```
bit clk, fclk, req, gnt, en;
...
a1: assert property
      (@(posedge clk) en && $rose(req) |=> gnt);

a2: assert property
      (@(posedge clk) en && $rose(req, @(posedge fclk)) |=> gnt);
```

Both assertions `a1` and `a2` read: “whenever `en` is high and `req` rises, at the next cycle `gnt` must be asserted.” In both assertions, the rise of `req` occurs if and only if the sampled value of `req` at the current posedge of `clk` is `1'b1` and the sampled value of `req` at a particular prior point is distinct from `1'b1`. The assertions differ in the specification of the prior point. In `a1` the prior point is the preceding posedge of `clk`, while in `a2` the prior point is the most recent prior posedge of `fclk`.

As another example,

```
always_ff @(posedge clk1)
   reg1 <= $rose(b, @(posedge clk2));
```

Here, `reg1` is updated in each time step in which **posedge** `clk1` occurs, using the value returned from the `$rose` sampled value function in that time step. `$rose` compares the sampled value of the LSB of `b` from the current time step (one in which **posedge** `clk1` occurs) with the sampled value of the LSB of `b` in the strictly prior time step in which **posedge** `clk2` occurs.

The following example is illegal if it is not within the scope of a default clocking because no clock can be inferred:

```
always @(posedge clk) begin
   ...
   @(negedge clk2);
   x = $past(y, 5); // illegal if not within default clocking
end
```

This example is legal if it is within the scope of a default clocking.

### 16.9.4 Global clocking past and future sampled value functions

This subclause describes the system functions available for accessing the nearest past and future values of an expression as sampled by the global clock. They may be used only if global clocking is defined (see 14.14). These functions include the capability to access the sampled value at the global clock tick that immediately precedes or follows the time step at which the function is called. Sampled value is explained in 16.5.1. The following functions are provided.

Global clocking past sampled value functions are as follows:

**$past_gclk (** expression **)**

**$rose_gclk (** expression **)**

**$fell_gclk (** expression **)**

**$stable_gclk (** expression **)**

**$changed_gclk (** expression **)**

Global clocking future sampled value functions are as follows:

**$future_gclk (** expression **)**

**$rising_gclk (** expression **)**

**$falling_gclk (** expression **)**

**$steady_gclk (** expression **)**

**$changing_gclk (** expression **)**

The behavior of the global clocking past sampled value functions can be defined using the sampled value functions as follows (the symbol ≡ means here "is equivalent by definition"):

```
    $past_gclk(v)      ≡ $past(v,,,  @$global_clock)
    $rose_gclk(v)      ≡ $rose(v,    @$global_clock)
    $fell_gclk(v)      ≡ $fell(v,    @$global_clock)
    $stable_gclk(v)    ≡ $stable(v,  @$global_clock)
    $changed_gclk(v)   ≡ $changed(v, @$global_clock)
```

The global clocking future sampled value functions are similar except that they use the subsequent value of the expression.

`$future_gclk(v)` is the sampled value of `v` at the next global clock tick (see 16.5.1 for the definition of sampling in future clock ticks).

The other functions are defined as follows:

— `$rising_gclk(expression)` returns true (`1'b1`) if the sampled value of the LSB of the expression is changing to 1 at the next global clocking tick. Otherwise, it returns false (`1'b0`).

— `$falling_gclk(expression)` returns true (`1'b1`) if the sampled value of the LSB of the expression is changing to 0 at the next global clocking tick. Otherwise, it returns false (`1'b0`).

— `$steady_gclk(expression)` returns true (`1'b1`) if the sampled value of the expression does not change at the next global clock tick. Otherwise, it returns false (`1'b0`).

— `$changing_gclk(expression)` is the complement of `$steady_gclk`, i.e., `!$steady_gclk(expression)`.

The global clocking future sampled value functions may be invoked only in *property_expr* or in *sequence_expr*; this implies that they shall not be used in assertion action blocks. The global clocking past sampled value functions are a special case of the sampled value functions, and therefore the regular restrictions imposed on the sampled value functions and their arguments apply (see 16.9.3). In particular, the global clocking past sampled value functions are usable in general procedural code and action blocks. Additional restrictions are imposed on the usage of the global clocking future sampled value functions: they shall not be nested and they shall not be used in assertions containing sequence match items (see 16.10, 16.11).

The following example illustrates the illegal usage of the global clocking future sampled value functions:

```
// Illegal: global clocking future sampled value functions
// shall not be nested
a1: assert property (@clk $future_gclk(a || $rising_gclk(b));
sequence s;
   bit v;
   (a, v = a) ##1 (b == v)[->1];
endsequence : s

// Illegal: a global clocking future sampled value function shall not
// be used in an assertion containing sequence match items
a2: assert property (@clk s |=> $future_gclk(c));
```

Even though global clocking future sampled value functions depend on future values of their arguments, the interval of simulation time steps for an evaluation attempt of an assertion containing global clocking future sampled value functions is defined as though the future sampled values were known in advance. The end of the evaluation attempt is defined to be the last tick of the assertion clock and is not delayed any additional time steps up to the next global clocking tick.

The behavior of **disable iff** and other asynchronous assertion related controls such as `$assertcontrol` (see 20.12) is with respect to the interval of the evaluation attempt previously defined. If, for example, `$assertcontrol` with `control_type` 5 (Kill) is executed in a time step strictly after the last tick of the assertion clock for the evaluation attempt, then it shall not affect that attempt, even if `$assertcontrol` is executed no later than the next global clocking tick.

Execution of the action block of an assertion containing global clocking future sampled value functions shall be delayed until the global clocking tick that follows the last tick of the assertion clock for the attempt. If the evaluation attempt fails and `$error` is called by default (see 16.14.1), then `$error` shall be called at the global clocking tick that follows the last tick of the assertion clock.

A tool specific message that reports the starting or ending time step of an evaluation attempt of an assertion containing global clocking future sampled functions shall be consistent with the preceding definition of the interval of simulation time steps for the evaluation attempt. The message may also report the time step in which it is written, which may be that of the global clocking tick that follows the last tick of the assertion clock.

*Example 1:*

Table 16-2 shows the values returned by the global clocking future sampled value functions for `sig` at different time moments.

The following assertion states that the signal may change only on falling clock:

```
a1: assert property (@$global_clock $changing_gclk(sig)
                                |-> $falling_gclk(clk))
    else $error(”sig is not stable”);
```

In Figure 16-4, the vertical arrows indicate the ticks of the global clock. The assertion `a1` is violated at time 80 because `$changing_gclk(sig)` is true and `$falling_gclk(clk)` is false. Because the assertion contains global clocking future sampled value functions, the error task `$error("sig is not stable")` in the action block is executed at time 90. If, as part of the tool-specific message printed by `$error`, a tool reports the ending or failing time of this evaluation attempt, the time reported is 80.

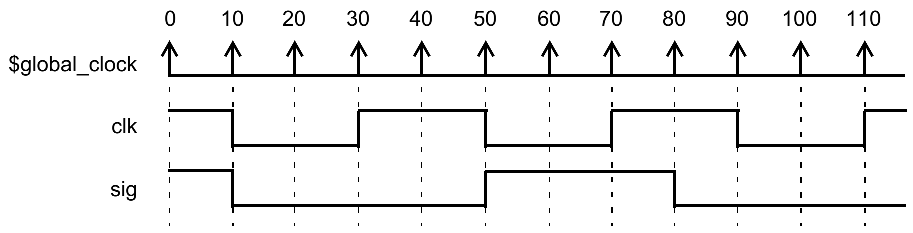


**Figure 16-4—Future value change**

**Table 16-2—Global clocking future sampled value functions**

| Time | $sampled(sig) | $future_gclk(sig) | $rising_gclk(sig) | $falling_gclk(sig) | $changing_gclk(sig) | $steady_gclk(sig) |
|---|---|---|---|---|---|---|
| 10 | 1'b1 | 1'b0 | 1'b0 | 1'b1 | 1'b1 | 1'b0 |
| 30 | 1'b0 | 1'b0 | 1'b0 | 1'b0 | 1'b0 | 1'b1 |
| 40 | 1'b0 | 1'b0 | 1'b0 | 1'b0 | 1'b0 | 1'b1 |
| 50 | 1'b0 | 1'b1 | 1'b1 | 1'b0 | 1'b1 | 1'b0 |
| 80 | 1'b1 | 1'b0 | 1'b0 | 1'b1 | 1'b1 | 1'b0 |

*Example 2:*

The following assumption states that a signal `sig` must remain stable between two falling edges of a clock `clk` as sampled by global clocking. This differs from the property in Example 1 in the case where the first falling edge of `clk` has not yet occurred. In Example 1, `sig` is not allowed to change in that case, but in this example `sig` can toggle freely while waiting for `clk` to begin.

```
a2: assume property(@$global_clock
          $falling_gclk(clk) ##1 (!$falling_gclk(clk)[*1:$]) |->
                                                  $steady_gclk(sig));
```

*Example 3:*

Assume that the signal `rst` is high between times 82 and 84, and is low at all other time moments. Then the following assertion:

```
a3: assert property (@$global_clock disable iff (rst) $changing_gclk(sig)
                                     |-> $falling_gclk(clk))
    else $error(”sig is not stable”);
```

fails at time 80 (see Figure 16-4) since `rst` is inactive at time 80. The interval of the failing evaluation attempt starts and ends at time 80. Although `rst` is active prior to the execution of the action block at time 90, the attempt is not disabled.

*Example 4:*

In some cases, the global clocking future value functions provide a more natural expression of a property than the past value functions. For example, the following two assertions are equivalent:

```
// A ##1 is needed in a4 due to the corner case at cycle 0
a4: assert property (##1 $stable_gclk(sig));
```

```
    // In a5, there is no issue at cycle 0
    a5: assert property ($steady_gclk(sig));
```

### 16.9.5 AND operation

The binary operator **and** is used when both operands are expected to match, but the end times of the operand sequences can be different (see Syntax 16-7).

```
sequence_expr ::=                                          // from A.2.10
   ...
   | sequence_expr and sequence_expr
```

*Syntax 16-7—And operator syntax (excerpt from Annex A)*

The two operands of **and** are sequences. The requirement for the match of the **and** operation is that both the operands shall match. The operand sequences start at the same time. When one of the operand sequences matches, it waits for the other to match. The end time of the composite sequence is the end time of the operand sequence that completes last.

When `te1` and `te2` are sequences, then the composite sequence

```
    te1 and te2
```

matches if `te1` and `te2` match. The end time is the end time of either `te1` or `te2`, whichever matches last.

The following example is a sequence with operator **and**, where the two operands are sequences:

```
    (te1 ##2 te2) and (te3 ##2 te4 ##2 te5)
```

The operation as illustrated in Figure 16-5 shows the evaluation attempt at clock tick 8. Here, the two operand sequences are `(te1 ##2 te2)` and `(te3 ##2 te4 ##2 te5)`. The first operand sequence requires that first `te1` evaluates to true followed by `te2` two clock ticks later. The second sequence requires that first `te3` evaluates to true followed by `te4` two clock ticks later, followed by `te5` two clock ticks later.

This attempt results in a match because both operand sequences match. The end times of matches for the individual sequences are clock ticks 10 and 12. The end time for the composite sequence is the later of the two end times; therefore, a match is recognized for the composite sequence at clock tick 12.

In the following example, the first operand sequence has a concatenation operator with range from 1 to 5:

```
    (te1 ##[1:5] te2) and (te3 ##2 te4 ##2 te5)
```

The first operand sequence requires that `te1` evaluate to true and that `te2` evaluate to true 1, 2, 3, 4, or 5 clock ticks later. The second operand sequence is the same as in the previous example. To consider all possibilities of a match of the composite sequence, the following steps can be taken:

a) Five threads of evaluation are started for the five possible linear sequences associated with the first sequence operand.

b) The second operand sequence has only one associated linear sequence; therefore, only one thread of evaluation is started for it.

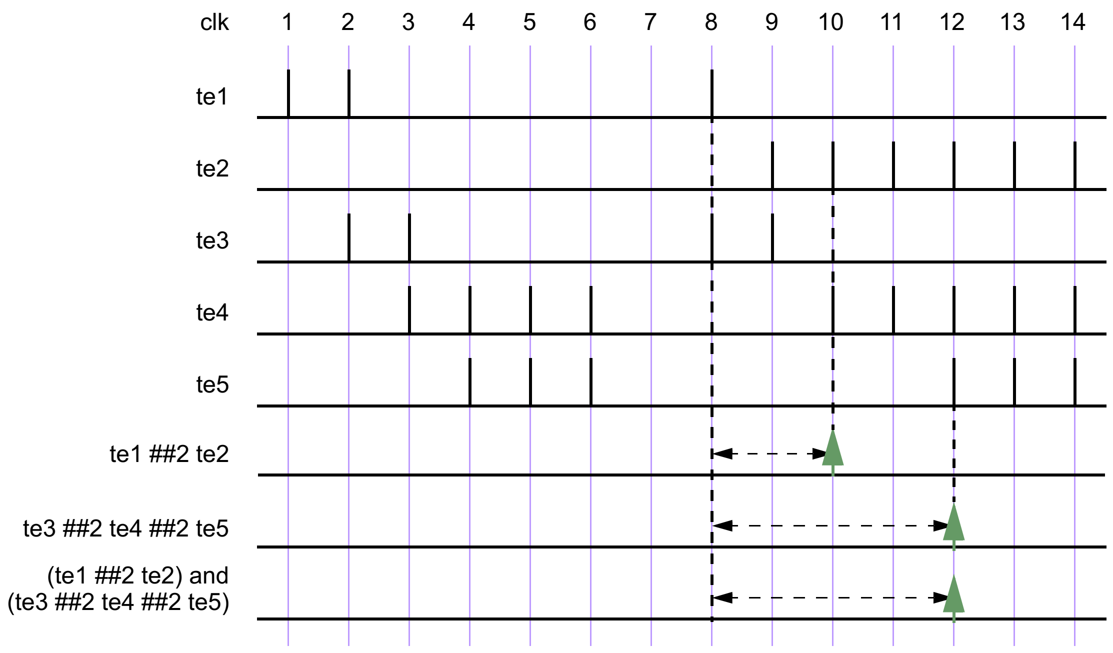


**Figure 16-5—ANDing (and) two sequences**

c) Figure 16-6 shows the evaluation attempt beginning at clock tick 8. All five linear sequences for the first operand sequence match, as shown in a time window; therefore, there are five matches of the first operand sequence, ending at clock ticks 9, 10, 11, 12, and 13, respectively. The second operand sequence matches at clock tick 12.

d) Each match of the first operand sequence is combined with the single match of the second operand sequence, and the rules of the AND operation determine the end time of the resulting match of the composite sequence.

The result of this computation is five matches of the composite sequence, four of them ending at clock tick 12, and the fifth ending at clock tick 13. Figure 16-6 shows the matches of the composite sequence ending at clock ticks 12 and 13.

If `te1` and `te2` are sampled expressions (not sequences), the sequence (`te1` **`and`** `te2`) matches if `te1` and `te2` both evaluate to true.

An example is illustrated in Figure 16-7, which shows the results for attempts at every clock tick. The sequence matches at clock tick 1, 3, 8, and 14 because both `te1` and `te2` are simultaneously true. At all other clock ticks, match of the AND operation fails because either `te1` or `te2` is false.

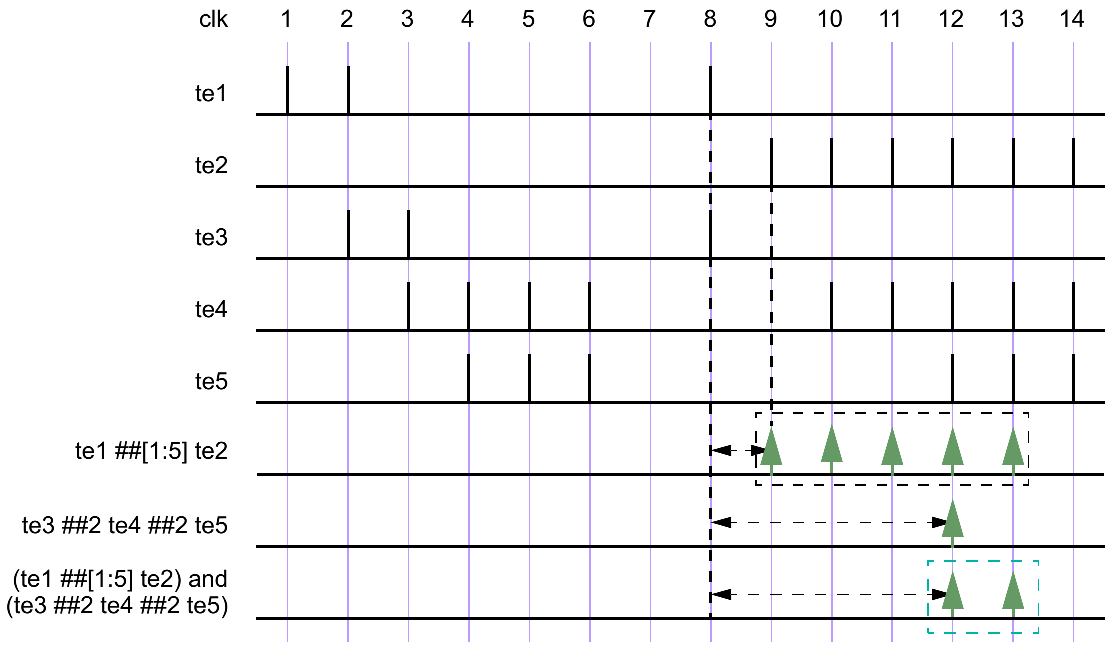


**Figure 16-6—ANDing (and) two sequences, including a time range**

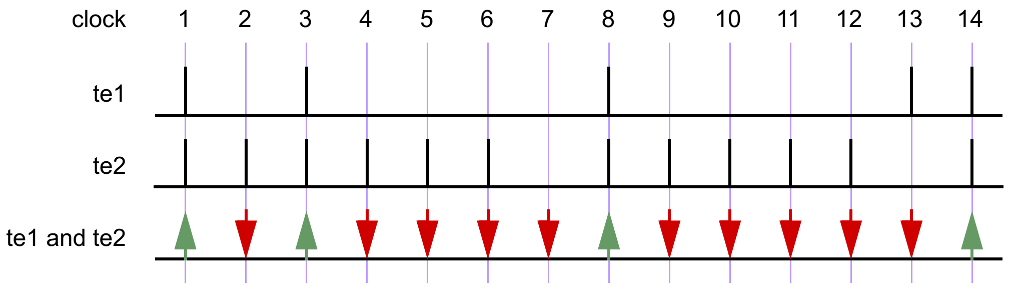


**Figure 16-7—ANDing (and) two Boolean expressions**

### 16.9.6 Intersection (AND with length restriction)

The binary operator **`intersect`** is used when both operand sequences are expected to match, and the end times of the operand sequences must be the same (see Syntax 16-8).

```
sequence_expr ::=                                                    // from A.2.10
    ...
  | sequence_expr intersect  sequence_expr
```

*Syntax 16-8—Intersect operator syntax (excerpt from Annex A)*

The two operands of **`intersect`** are sequences. The requirements for match of the **`intersect`** operation are as follows:

— Both the operands shall match.

— The lengths of the two matches of the operand sequences shall be the same.

The additional requirement on the length of the sequences is the basic difference between **`and`** and **`intersect`**.

An attempted evaluation of an **`intersect`** sequence can result in multiple matches. The results of such an attempt can be computed as follows:

- — Matches of the first and second operands that are of the same length are paired. Each such pair results in a match of the composite sequence, with length and match point equal to the shared length and match point of the paired matches of the operand sequences.
- — If no such pair is found, then there is no match of the composite sequence.

Figure 16-8 is similar to Figure 16-6, except that **`and`** is replaced by **`intersect`**. In this case, unlike in Figure 16-6, there is only a single match at clock tick 12.

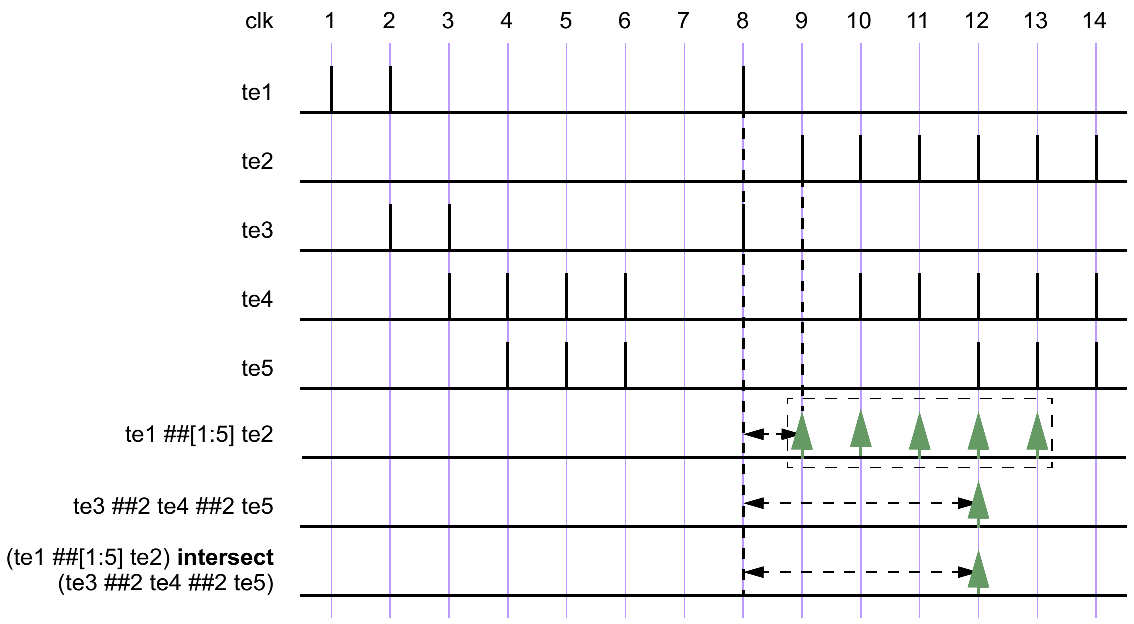


**Figure 16-8—Intersecting two sequences**

### 16.9.7 OR operation

The operator **`or`** is used when at least one of the two operand sequences is expected to match (Syntax 16-9).

---

sequence_expr ::= *// from A.2.10*
  ...
  | sequence_expr **or** sequence_expr

---

*Syntax 16-9—Or operator syntax (excerpt from Annex A)*

The two operands of **`or`** are sequences.

If the operands `te1` and `te2` are expressions, then

```
te1 or te2
```

matches at any clock tick on which at least one of `te1` and `te2` evaluates to true.

Figure 16-9 illustrates an OR operation for which the operands `te1` and `te2` are expressions. The composite sequence does not match at clock ticks 7 and 13 because `te1` and `te2` are both false at those times. At all other clock ticks, the composite sequence matches, as at least one of the two operands evaluates to true.

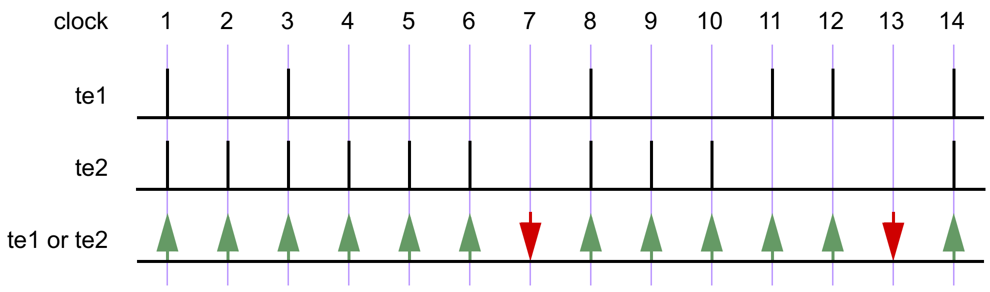


**Figure 16-9—ORing (or) two Boolean expressions**

When `te1` and `te2` are sequences, then the sequence

```
te1 or te2
```

matches if at least one of the two operand sequences `te1` and `te2` matches. Each match of either `te1` or `te2` constitutes a match of the composite sequence, and its end time as a match of the composite sequence is the same as its end time as a match of `te1` or of `te2`. In other words, the set of matches of `te1 or te2` is the union of the set of matches of `te1` with the set of matches of `te2`.

The following example shows a sequence with operator **`or`** where the two operands are sequences. Figure 16-10 illustrates this example.

```
(te1 ##2 te2) or (te3 ##2 te4 ##2 te5)
```

The two operand sequences in the preceding example are `(te1 ##2 te2)` and `(te3 ##2 te4 ##2 te5)`. The first sequence requires that `te1` first evaluates to true, followed by `te2` two clock ticks later. The second sequence requires that `te3` evaluates to true, followed by `te4` two clock ticks later, followed by `te5` two clock ticks later. In Figure 16-10, the evaluation attempt for clock tick 8 is shown. The first sequence matches at clock tick 10, and the second sequence matches at clock tick 12. Therefore, two matches for the composite sequence are recognized.

In the following example, the first operand sequence has a concatenation operator with range from 1 to 5:

```
(te1 ##[1:5] te2) or (te3 ##2 te4 ##2 te5)
```

The first operand sequence requires that `te1` evaluate to true and that `te2` evaluate to true 1, 2, 3, 4, or 5 clock ticks later. The second operand sequence requires that `te3` evaluate to true, that `te4` evaluate to true two clock ticks later, and that `te5` evaluate to true another two clock ticks later. The composite sequence matches at any clock tick on which at least one of the operand sequences matches. As shown in Figure 16-11, for the attempt at clock tick 8, the first operand sequence matches at clock ticks 9, 10, 11, 12, and 13, while the second operand matches at clock tick 12. The composite sequence, therefore, has one match at each of clock ticks 9, 10, 11, and 13 and has two matches at clock tick 12.

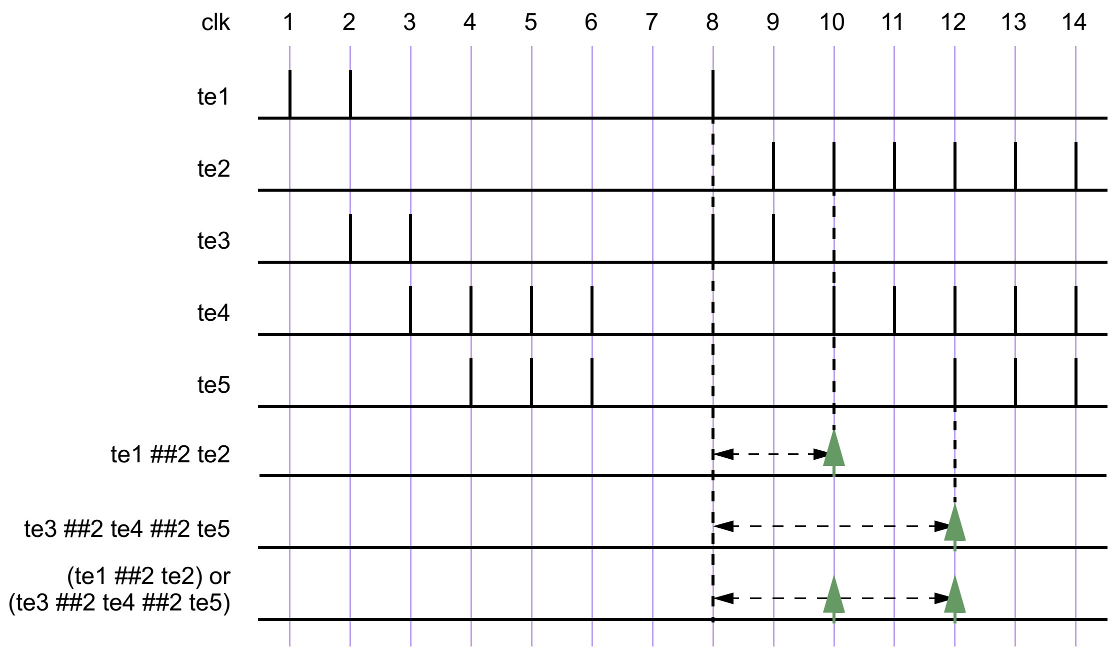


**Figure 16-10—ORing (or) two sequences**

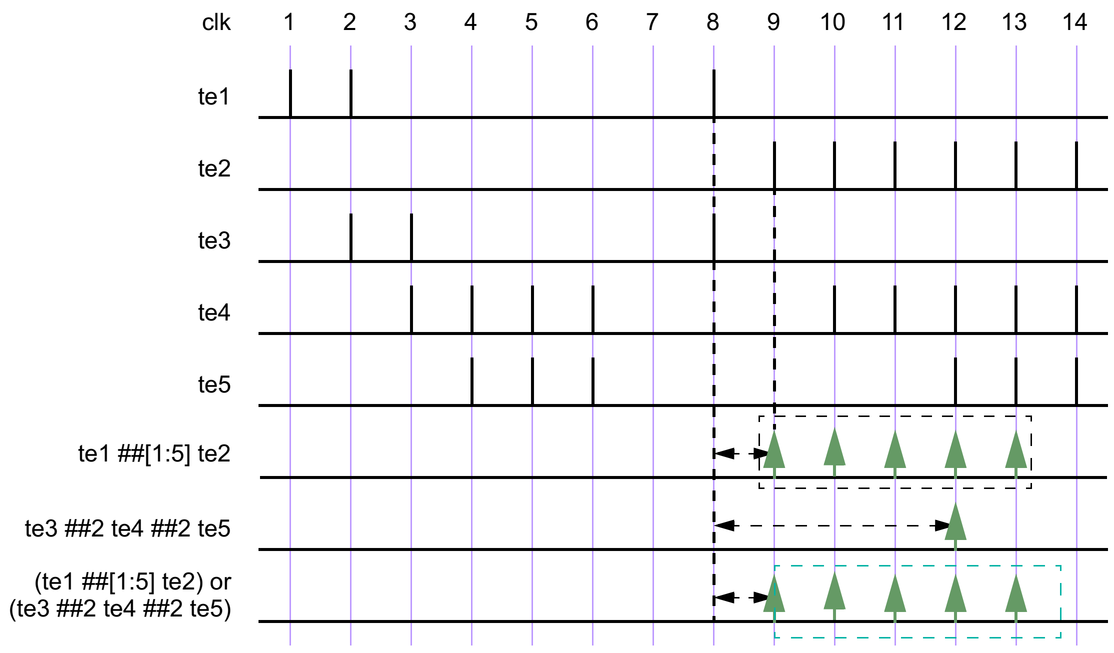


**Figure 16-11—ORing (or) two sequences, including a time range**

### 16.9.8 First_match operation

The **first_match** operator matches only the first of possibly multiple matches for an evaluation attempt of its operand sequence. This allows all subsequent matches to be discarded from consideration. In particular, when a sequence is a subsequence of a larger sequence, then applying the **first_match** operator has significant effect on the evaluation of the enclosing sequence (see Syntax 16-10).

```
sequence_expr ::=                                                    // from A.2.10
    ...
  | first_match ( sequence_expr {, sequence_match_item} )
```

*Syntax 16-10—First_match operator syntax (excerpt from Annex A)*

An evaluation attempt of **first_match** (*seq*) results in an evaluation attempt for the operand *seq* beginning at the same clock tick. If the evaluation attempt for *seq* produces no match, then the evaluation attempt for **first_match** (*seq*) produces no match. Otherwise, the match of *seq* with the earliest ending clock tick is a match of **first_match** (*seq*). If there are multiple matches of *seq* with the same ending clock tick as the earliest one, then all those matches are matches of **first_match** (*seq*).

The following example shows a variable delay specification:

```
sequence t1;
   te1 ## [2:5] te2;
endsequence
sequence ts1;
   first_match(te1 ## [2:5] te2);
endsequence
```

Here, te1 and te2 are expressions. Each attempt of sequence t1 can result in matches for up to four of the following sequences:

```
te1 ##2 te2
te1 ##3 te2
te1 ##4 te2
te1 ##5 te2
```

However, sequence ts1 can result in a match for only one of the preceding four sequences. Whichever match of the preceding four sequences ends first is a match of sequence ts1.

For example:

```
sequence t2;
   (a ##[2:3] b) or (c ##[1:2] d);
endsequence
sequence ts2;
   first_match(t2);
endsequence
```

Each attempt of sequence t2 can result in matches for up to four of the following sequences:

```
a ##2 b
a ##3 b
c ##1 d
c ##2 d
```

Sequence `ts2` matches only the earliest ending match of these sequences. If `a`, `b`, `c`, and `d` are expressions, then it is possible to have matches ending at the same time for both.

```
a ##2 b
c ##2 d
```

If both of these sequences match and (`c ##1 d`) does not match, then evaluation of `ts2` results in these two matches.

Sequence match items can be attached to the operand sequence of the **`first_match`** operator. The sequence match items are placed within the same set of parentheses that enclose the operand. Thus, for example, the local variable assignment `x = e` can be attached to the first match of seq via

```
first_match(seq, x = e)
```

which is equivalent to the following:

```
first_match((seq, x = e))
```

See 16.10 and 16.11 for discussion of sequence match items.

### 16.9.9 Conditions over sequences

Sequences often occur under the assumptions of some conditions for correct behavior. A logical condition must hold true, for instance, while processing a transaction. Also, occurrence of certain values is prohibited while processing a transaction. Such situations can be expressed directly using the construct shown in Syntax 16-11.

```
sequence_expr ::=                                        // from A.2.10
   ...
   | expression_or_dist throughout sequence_expr
```

*Syntax 16-11—Throughout construct syntax (excerpt from Annex A)*

The construct `exp` **`throughout`** `seq` is an abbreviation for the following:

```
(exp) [*0:$] intersect seq
```

The composite sequence, `exp` **`throughout`** `seq`, matches along a finite interval of consecutive clock ticks provided `seq` matches along the interval and `exp` evaluates to true at each clock tick of the interval.

The following example is illustrated in Figure 16-12.

```
sequence burst_rule1;
   @(posedge mclk)
      $fell(burst_mode) ##0
      ((!burst_mode) throughout (##2 ((trdy==0)&&(irdy==0)) [*7]));
endsequence
```

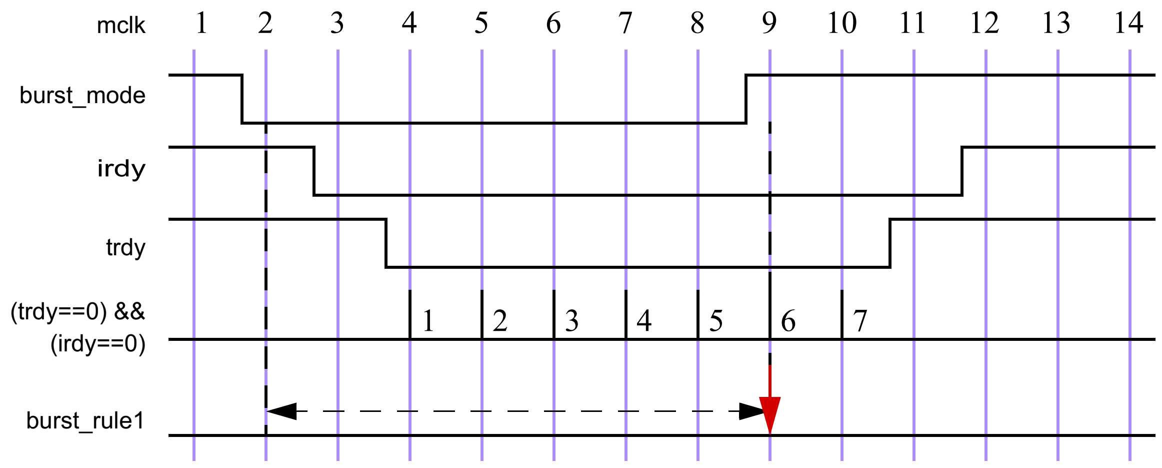


**Figure 16-12—Match with throughout restriction fails**

Figure 16-13 illustrates the evaluation attempt for sequence `burst_rule1` beginning at clock tick 2. Because signal `burst_mode` is high at clock tick 1 and low at clock tick 2, `$fell(burst_mode)` is true at clock tick 2. To complete the match of `burst_rule1`, the value of `burst_mode` is required to be low throughout a match of the subsequence (`##2 ((trdy==0)&&(irdy==0)) [*7]`) beginning at clock tick 2. This subsequence matches from clock tick 2 to clock tick 10. However, at clock tick 9 `burst_mode` becomes high, thereby failing to match according to the rules for **`throughout`**.

If signal `burst_mode` were instead to remain low through at least clock tick 10, then there would be a match of `burst_rule1` from clock tick 2 to clock tick 10, as shown in Figure 16-13.

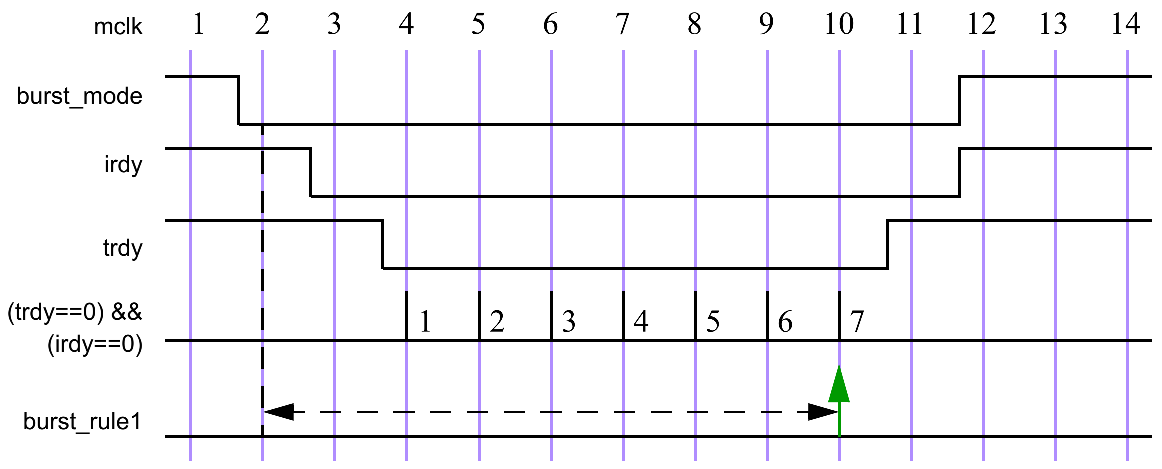


**Figure 16-13—Match with throughout restriction succeeds**

### 16.9.10 Sequence contained within another sequence

The containment of a sequence within another sequence is expressed as follows in Syntax 16-12.

```
sequence_expr ::=                                                    // from A.2.10
   ...
   | sequence_expr within sequence_expr
```

*Syntax 16-12—Within construct syntax (excerpt from Annex A)*

The construct `seq1` **`within`** `seq2` is an abbreviation for the following:

```
(1[*0:$] ##1 seq1 ##1 1[*0:$]) intersect seq2
```

The composite sequence `seq1` **`within`** `seq2` matches along a finite interval of consecutive clock ticks provided `seq2` matches along the interval and `seq1` matches along some subinterval of consecutive clock ticks. In other words, the matches of `seq1` and `seq2` must satisfy the following:

— The start point of the match of `seq1` shall be no earlier than the start point of the match of `seq2`.

— The match point of `seq1` shall be no later than the match point of `seq2`.

For example, the sequence

```
!trdy[*7] within ($fell(irdy) ##1 !irdy[*8])
```

matches from clock tick 3 to clock tick 11 on the trace shown in Figure 16-13.

### 16.9.11 Composing sequences from simpler subsequences

There are two ways in which a complex sequence can be composed using simpler subsequences.

One is to instantiate a named sequence by referencing its name. Evaluation of such a reference requires the named sequence to match starting from the clock tick at which the reference is reached during the evaluation of the enclosing sequence. For example:

```
sequence s;
   a ##1 b ##1 c;
endsequence
sequence rule;
   @(posedge sysclk)
      trans ##1 start_trans ##1 s ##1 end_trans;
endsequence
```

Sequence `s` is evaluated beginning one tick after the evaluation of `start_trans` in the sequence `rule`.

Another way to use a sequence is to detect its end point in another sequence. The reaching of the end point (see 16.7) can be tested by using the method `triggered`.

To detect the end point, the `triggered` method may be applied to a named sequence instance, with or without arguments, an untyped formal argument, or a formal argument of type **`sequence`**, where such is allowed, as follows:

```
sequence_instance.triggered
```

or

```
formal_argument_sequence.triggered
```

`triggered` is a method on a sequence. The result of its operation is true (`1'b1`) or false (`1'b0`) . When method `triggered` is evaluated in an expression, it tests whether its operand sequence has reached its end point at that particular point in time. The result of `triggered` does not depend upon the starting point of the match of its operand sequence. An example is shown as follows:

```
sequence e1;
   @(posedge sysclk) $rose(ready) ##1 proc1 ##1 proc2 ;
endsequence
sequence rule;
   @(posedge sysclk) reset ##1 inst ##1 e1.triggered ##1 branch_back;
endsequence
```

In this example, sequence `e1` must match one clock tick after `inst`. If the method `triggered` is replaced with an instance of sequence `e1`, a match of `e1` must start one clock tick after `inst`. Notice that method `triggered` only tests for the end point of `e1` and has no bearing on the starting point of `e1`.

The following example demonstrates an application of the method `triggered` on a named sequence instance with arguments:

```
sequence e2(a,b,c);
   @(posedge sysclk) $rose(a) ##1 b ##1 c;
endsequence
sequence rule2;
   @(posedge sysclk) reset ##1 inst ##1 e2(ready,proc1,proc2).triggered
      ##1 branch_back;
endsequence
```

`rule2` is equivalent to `rule2a` as follows:

```
sequence e2_instantiated;
  e2(ready,proc1,proc2);
endsequence
sequence rule2a;
  @(posedge sysclk) reset ##1 inst ##1 e2_instantiated.triggered ##1
branch_back;
endsequence
```

The following example demonstrates an application of method `triggered` on a formal argument of type **`sequence`**:

```
sequence e3(sequence a, untyped b);
   @(posedge sysclk) a.triggered ##1 b;
endsequence

sequence rule3;
   @(posedge sysclk) reset ##1 e3(ready ##1 proc1, proc2) ##1 branch_back;
endsequence
```

There are additional restrictions on passing local variables into an instance of a sequence to which `triggered` is applied (see 16.10).

The method `triggered` can be used in the presence of multiple clocks. However, the ending clock of the sequence instance to which `triggered` is applied shall always be the same as the clock in the context where the application of method `triggered` appears (see 16.13.5).

If a sequence admits an empty match, such empty matches shall not activate the `.triggered` method. For example, consider the following sequence, which admits both empty and nonempty matches:

```
sequence zero_or_one_req;
   (req==1'b1)[*0:1];
endsequence
```

The method `zero_or_one_req.triggered()` will only return true (`1'b1`) when the sampled value of `req` is `1'b1`, resulting in a nonempty match.

## 16.10 Local variables

Data can be manipulated within named sequences (see 16.8) and properties (see 16.12) using dynamically created local variables. The use of a static SystemVerilog variable implies that only one copy exists. If data values need to be checked in pipelined designs, then for each quantum of data entering the pipeline, a separate variable can be used to store the predicted output of the pipeline for later comparison when the result actually exits the pipe. This storage can be built by using an array of variables arranged in a shift register to mimic the data propagating through the pipeline. However, in more complex situations where the latency of the pipe is variable and out of order, this construction could become very complex and error prone. Therefore, variables are needed that are local to and are used within a particular transaction check that can span an arbitrary interval of time and can overlap with other transaction checks. Such a variable will thus be dynamically created when needed within an instance of a sequence and removed when the end of the sequence is reached.

The dynamic creation of a local variable and its assignment is achieved by either using a local variable formal argument declaration (see 16.8.2, 16.12.18) or using an assertion variable declaration within the declaration of a named sequence or property (see 16.12). Without further specification, the term *local variable* shall mean either a local variable formal argument or a local variable declared in an *assertion_variable_declaration*. Without further specification, the term *local variable initialization assignment* shall mean either an initialization assignment to a local variable formal argument of direction **`input`** or **`inout`** of the value of the corresponding actual argument or a declaration assignment to a local variable declared in an *assertion_variable_declaration* (see Syntax 16-13).

```
assertion_variable_declaration ::=                                   // from A.2.10
    var_data_type list_of_variable_decl_assignments ;
```

*Syntax 16-13—Assertion variable declaration syntax (excerpt from Annex A)*

The data type of an assertion variable declaration shall be specified explicitly. The data type shall be one of the types allowed within assertions as defined in 16.6. The data type shall be followed by a comma-separated list of one or more identifiers with optional declaration assignments. A declaration assignment, if present, defines the initial value to be placed in the corresponding local variable. The initial value is defined by an expression, which need not be constant.

The sampled value of a local variable is defined as the current value (see 16.5.1).

At the beginning of each evaluation attempt of an instance of a named sequence or property, a new copy of each of its local variables shall be created and, if present, the corresponding initialization assignment shall be performed. Initialization assignments shall be performed in the Observed region in the order that they appear in the sequence or property declaration. For the purposes of this rule, all initialization assignments to local variable formal arguments shall be performed before any initialization assignment to a local variable declared in an *assertion_variable_declaration*. An initialization assignment to a local variable uses the sampled value of its expression in the time slot in which the evaluation attempt begins. The expression of an initialization assignment to a given local variable may refer to a previously declared local variable. In this case the previously declared local variable shall itself have an initialization assignment, and the initial value assigned to the previously declared local variable shall be used in the evaluation of the expression assigned to the given local variable. Local variables do not have default initial values. A local variable without an initialization assignment shall be unassigned at the beginning of the evaluation attempt.

For example, at the beginning of an evaluation attempt of an instance of

```
sequence s;
   logic u, v = a, w = v || b;
   ...
endsequence
```

the assignment of `a` to `v` is performed first, and the assignment of `v || b` to `w` is performed second. The value assigned to `w` is the same as would result from the declaration assignment `w = a || b`. The local variable `u` is unassigned at the beginning of the evaluation attempt.

Local variables may be assigned and reassigned within the body of the sequence or property in which they are declared.

```
sequence_expr ::=                                                  // from A.2.10
    ...
  | ( sequence_expr {, sequence_match_item} ) [ sequence_abbrev ]
    ...
sequence_match_item ::=
    operator_assignment
  | inc_or_dec_expression
    ...
```

*Syntax 16-14—Variable assignment syntax (excerpt from Annex A)*

One or more local variables may be assigned at the end point of a syntactic subsequence by placing the subsequence, comma-separated from the list of local variable assignments, in parentheses. At the end of any nonempty match of the subsequence, the local variable assignments are performed in the order that they appear in the list. For example, if in

```
a ##1 b[->1] ##1 c[*2]
```

it is desired to assign `x = e` and then `y = x && f` at the match of `b[->1]`, the sequence can be rewritten as

```
a ##1 (b[->1], x = e, y = x && f) ##1 c[*2]
```

A local variable may be reassigned later in the sequence or property, as in

```
a ##1 (b[->1], x = e, y = x && f) ##1 (c[*2], x &= g)
```

The subsequence to which a local variable assignment is attached shall not admit an empty match (see 16.12.22). For example, the sequence

```
a ##1 (b[*0:1], x = e) ##1 c[*2] // illegal
```

is illegal because the subsequence `b[*0:1]` can match the empty word. The sequence

```
(a ##1 b[*0:1], x = e) ##1 c[*2] // legal
```

is legal because the concatenated subsequence `a ##1 b[*0:1]` cannot match the empty word.

A local variable may be referenced within the sequence or property in which it is declared. The sequence or property shall assign a value to the local variable prior to the point at which the reference is made. The prior assignment may be an initialization assignment or an assignment attached to a subsequence. There is an implicit reference associated with the use of an *inc_or_dec_operator* or an assignment operator other than "=". Therefore, a local variable shall be assigned a value prior to being updated with an *inc_or_dec_operator* or with an assignment operator other than "=".

Under certain circumstances, a local variable that is assigned later becomes unassigned. If a local variable does not flow out of a subsequence (see the following), then the local variable shall become unassigned at the end of that subsequence, regardless of whether it was assigned a value prior to that point. The local variable shall not be referenced after the point from which it does not flow until after it has again been assigned a value. See F.5.4 for precise conditions defining local variable flow.

Hierarchical references to a local variable are not allowed.

As an example of local variable usage, assume a pipeline that has a fixed latency of five clock cycles. The data enter the pipe on `pipe_in` when `valid_in` is true, and the value computed by the pipeline appears five clock cycles later on the signal `pipe_out1`. The data as transformed by the pipe are predicted by a function that increments the data. The following property verifies this behavior:

```
property e;
   int x;
   (valid_in, x = pipe_in) |-> ##5 (pipe_out1 == (x+1));
endproperty
```

Property `e` is evaluated as follows:

— When `valid_in` is true, `x` is assigned the value of `pipe_in`. If five cycles later, `pipe_out1` is equal to `x+1`, then property `e` is true. Otherwise, property `e` is false.
— When `valid_in` is false, property `e` evaluates to true.

A local variable can be used to form expressions in the same way that a static variable of the same type can be used. This includes the use of local variables in expressions for bit-selects and part-selects of vectors or for indices of arrays. A local variable shall not be used in a clocking event expression.

Local variables may be used in sequences or properties.

```
sequence data_check;
   int x;
   a ##1 (!a, x = data_in) ##1 !b[*0:$] ##1 b && (data_out == x);
endsequence
property data_check_p
   int x;
   a ##1 (!a, x = data_in) |=> !b[*0:$] ##1 b && (data_out == x);
endproperty
```

Local variable assignments may be attached to the operand sequence of a repetition and accomplish accumulation of values.

```
sequence rep_v;
   int x = 0;
   (a[->1], x += data)[*4] ##1 b ##1 c && (data_out == x);
endsequence
```

An accumulating local variable may be used to count the number of times a condition is repeated, as in the following example:

```
sequence count_a_cycles;
   int x;
   ($rose(a), x = 1)
   ##1 (a, x++)[*0:$]
   ##1 !a && (x <= MAX);
endsequence
```

The local variables declared within a sequence or property are not visible in the context where the sequence or property is instantiated. The following example illustrates an illegal access to local variable `v1` of sequence `sub_seq1` in sequence `seq1`.

```
sequence sub_seq1;
   int v1;
   (a ##1 !a, v1 = data_in) ##1 !b[*0:$] ##1 b && (data_out == v1);
endsequence
sequence seq1;
   c ##1 sub_seq1 ##1 (do1 == v1); // error because v1 is not visible
endsequence
```

It can be useful to assign a value to a local variable within an instance of a named sequence and reference the local variable in the instantiating context at or after the completion of a match of the instance. The rules for assigning values to a local variable within an instance of a named sequence are described in 16.8.2. This capability is also supported under the following conditions:

— The local variable shall be declared outside the named sequence, and its scope shall include both the instance of the named sequence and the desired reference in the instantiating context.
— The local variable shall be passed as an entire actual argument in the list of arguments of the instance of the named sequence.
— The corresponding formal argument shall be untyped.

The named sequence may specify assignments to the formal argument in one or more *sequence_match_items*.

The following example illustrates this usage:

```
sequence sub_seq2(lv);
   (a ##1 !a, lv = data_in) ##1 !b[*0:$] ##1 b && (data_out == lv);
endsequence
sequence seq2;
   int v1;
   c ##1 sub_seq2(v1)   // v1 is bound to lv
   ##1 (do1 == v1);     // v1 holds the value that was assigned to lv
endsequence
```

An alternative way to achieve a similar capability is by using local variable formal arguments (see 16.8.2).

Local variables can be passed into an instance of a named sequence to which `triggered` is applied and accessed in a similar manner. For example:

```
sequence seq2a;
   int v1; c ##1 sub_seq2(v1).triggered ##1 (do1 == v1);
   // v1 is now bound to lv
endsequence
```

There are additional restrictions when passing local variables into an instance of a named sequence to which `triggered` is applied:

- Local variables can be passed in only as entire actual arguments, not as proper subexpressions of actual arguments.
- In the declaration of the named sequence, the formal argument to which the local variable is bound shall not be referenced before it is assigned.

The second restriction is met by `sub_seq2` because the assignment `lv = data_in` occurs before the reference to `lv` in `data_out == lv`.

If a local variable is assigned before being passed into an instance of a named sequence to which `triggered` is applied, then the restrictions prevent this assigned value from being visible within the named sequence. The restrictions are important because the use of `triggered` means that there is no guaranteed relationship between the point in time at which the local variable is assigned outside the named sequence and the beginning of the match of the instance.

A local variable that is passed in as actual argument to an instance of a named sequence to which `triggered` is applied will flow out of the application of `triggered` to that instance provided both of the following conditions are met:

- The local variable flows out of the end of the named sequence instance, as defined by the local variable flow rules for sequences. (See the following and F.5.4.)
- The application of `triggered` to this instance is a maximal Boolean expression. In other words, the application of `triggered` cannot have negation or any other expression operator applied to it.

Both conditions are satisfied by `sub_seq2` and `seq2a`. Thus, in `seq2a`, the value in `v1` in the comparison `do1 == v1` is the value assigned to `lv` in `sub_seq2` by the assignment `lv = data_in`. However, in

```
sequence seq2b;
   int v1; c ##1 !sub_seq2(v1).triggered ##1 (do1 == v1); // v1 unassigned
endsequence
```

the second condition is violated because of the negation applied to `sub_seq2(v1).triggered`. Therefore, `v1` does not flow out of the application of `triggered` to this instance, and the reference to `v1` in `do1 == v1` is to an unassigned variable.

In a single cycle, there can be multiple matches of a sequence instance to which `triggered` is applied, and these matches can have different valuations of the local variables. The multiple matches are treated semantically the same way as matching both disjuncts of an **or** (see the following). In other words, the thread evaluating the instance to which `triggered` is applied will fork to account for such distinct local variable valuations.

When a local variable is a formal argument of a sequence declaration, it is illegal to declare the variable, as shown in the following example:

```
sequence sub_seq3(lv);
   int lv; // illegal because lv is a formal argument
   (a ##1 !a, lv = data_in) ##1 !b[*0:$] ##1 b && (data_out == lv);
endsequence
```

There are special considerations when using local variables in sequences involving the branching operators **`or`**, **`and`**, and **`intersect`**. The evaluation of a composite sequence constructed from one of these operators can be thought of as forking two threads to evaluate the operand sequences in parallel. A local variable may have been assigned a value before the start of the evaluation of the composite sequence, either from an initialization assignment or from an assignment attached to a preceding subsequence. Such a local variable is said to *flow in* to each of the operand sequences. The local variable may be assigned or reassigned in one or both of the operand sequences. In general, there is no guarantee that evaluation of the two threads results in consistent values for the local variable, or even that there is a consistent view of whether the local variable has been assigned a value. Therefore, the values assigned to the local variable before and during the evaluation of the composite sequence are not always allowed to be visible after the evaluation of the composite sequence.

In some cases, inconsistency in the view of the local variable's value does not matter, while in others it does. Precise conditions are given in F.5.4 to define static (i.e., compile-time computable) conditions under which a sufficiently consistent view of the local variable's value after the evaluation of the composite sequence is provided. If these conditions are satisfied, then the local variable is said to *flow out* of the composite sequence. Otherwise, the local variable shall become unassigned at the end of the composite sequence. An intuitive description of the conditions for local variable flow follows:

a) Variables assigned on parallel threads cannot be accessed in sibling threads. For example:

```
sequence s4;
    int x;
    (a ##1 (b, x = data) ##1 c) or (d ##1 (e==x)); // illegal
endsequence
```

b) In the case of **`or`**, a local variable flows out of the composite sequence if, and only if, it flows out of each of the operand sequences. If the local variable is not assigned before the start of the composite sequence and it is assigned in only one of the operand sequences, then it does not flow out of the composite sequence.

c) Each thread for an operand of an **`or`** that matches its operand sequence continues as a separate thread, carrying with it its own latest assignments to the local variables that flow out of the composite sequence. These threads do not have to have consistent valuations for the local variables. For example:

```
sequence s5;
   int x,y;
   ((a ##1 (b, x = data, y = data1) ##1 c)
      or (d ##1 (`true, x = data) ##0 (e==x))) ##1 (y==data2);
   // illegal because y is not in the intersection
endsequence
sequence s6;
   int x,y;
   ((a ##1 (b, x = data, y = data1) ##1 c)
      or (d ##1 (`true, x = data) ##0 (e==x))) ##1 (x==data2);
   // legal because x is in the intersection
endsequence
```

d) In the case of **`and`** and **`intersect`**, a local variable that flows out of at least one operand shall flow out of the composite sequence unless it is blocked. A local variable is blocked from flowing out of the composite sequence if either of the following statements applies:

   1) The local variable is assigned in and flows out of each operand of the composite sequence, or
   2) The local variable is blocked from flowing out of at least one of the operand sequences.

   The value of a local variable that flows out of the composite sequence is the latest assigned value. The threads for the two operands are merged into one at completion of evaluation of the composite sequence.

```
sequence s7;
   int x,y;
   ((a ##1 (b, x = data, y = data1) ##1 c)
      and (d ##1 (`true, x = data) ##0 (e==x))) ##1 (x==data2);
   // illegal because x is common to both threads
endsequence
sequence s8;
   int x,y;
   ((a ##1 (b, x = data, y = data1) ##1 c)
      and (d ##1 (`true, x = data) ##0 (e==x))) ##1 (y==data2);
   // legal because y is in the difference
endsequence
```

## 16.11 Calling subroutines on match of a sequence

Tasks, task methods, void functions, void function methods, and system tasks can be called at the end of a successful nonempty match of a sequence. The subroutine calls, like local variable assignments, appear in the comma-separated list that follows the sequence. The subroutine calls are said to be attached to the sequence. It shall be an error to attach a subroutine call or any *sequence_match_item* to a sequence that admits an empty match (see 16.12.22). The sequence and the list that follows are enclosed in parentheses (see Syntax 16-15).

```
sequence_expr ::=                                                        // from A.2.10
    ...
  | ( sequence_expr {, sequence_match_item} ) [ sequence_abbrev ]
    ...
sequence_match_item ::=
    operator_assignment
  | inc_or_dec_expression
  | subroutine_call
```

*Syntax 16-15—Subroutine call in sequence syntax (excerpt from Annex A)*

For example:

```
sequence s1;
   logic v, w;
   (a, v = e) ##1
   (b[->1], w = f, $display("b after a with v = %h, w = %h\n", v, w));
endsequence
```

defines a sequence `s1` that matches at the first occurrence of `b` strictly after an occurrence of `a`. At the match, the system task `$display` is executed to write a message that announces the match and shows the values assigned to the local variables `v` and `w`.

All subroutine calls attached to a sequence are executed at every end point of the sequence. For each end point, the attached calls are executed in the order they appear in the list. Assertion evaluation does not wait on or receive data back from any attached subroutine. The subroutines are scheduled in the Reactive region, like an action block.

Each argument of a subroutine call attached to a sequence shall either be passed by value as an input or be passed by reference (either **ref** or **const ref**; see 13.5.2). Actual argument expressions that are passed by value use sampled values of the underlying variables and are consistent with the variable values used to evaluate the sequence match. The variable passed by value as an input shall be of a type allowed in 16.6. An automatic variable may be passed as a constant input for a subroutine call from an assertion statement in procedural code (see 16.14.6.1). An automatic variable shall not be passed by reference nor passed as a non-constant input to a subroutine call from an assertion statement in procedural code. The rules for passing elements of dynamic arrays, queues, and associative arrays as ref arguments are described in 13.5.2.

Local variables can be passed into subroutine calls attached to a sequence. Any local variable that flows out of the sequence or that is assigned in the list following the sequence, but before the subroutine call, can be used in an actual argument expression for the call. If a local variable appears in an actual argument expression, then that argument shall be passed by value.

## 16.12 Declaring properties

A property defines a behavior of the design. A named property may be used for verification as an assumption, an obligation, or a coverage specification. In order to use the behavior for verification, an **assert**, **assume**, or **cover** statement must be used. A property declaration by itself does not produce any result.

A named **property** may be declared in any of the following:

- A module
- An interface
- A program
- A **clocking** block
- A package
- A compilation-unit scope
- A generate block
- A checker

To declare a named property, the **property** construct is used as shown in Syntax 16-16.

```
assertion_item_declaration ::=                                          // from A.2.10
    property_declaration
    ...
property_declaration ::=
    property property_identifier [ ( [ property_port_list ] ) ] ;
        { assertion_variable_declaration }
        property_spec [ ; ]
    endproperty [ : property_identifier ]
property_port_list ::=
    property_port_item {, property_port_item}
property_port_item ::=
    { attribute_instance } [ local [ property_lvar_port_direction ] ] property_formal_type
        formal_port_identifier {variable_dimension} [ = property_actual_arg ]
property_lvar_port_direction ::= input
property_formal_type ::=
    sequence_formal_type
  | property
property_spec ::=
    [clocking_event ] [ disable iff ( expression_or_dist ) ] property_expr
```

```
property_expr ::=
      sequence_expr
    | strong ( sequence_expr )
    | weak ( sequence_expr )
    | ( property_expr )
    | not property_expr
    | property_expr or property_expr
    | property_expr and property_expr
    | sequence_expr |-> property_expr
    | sequence_expr |=> property_expr
    | if ( expression_or_dist ) property_expr [ else property_expr ]
    | case ( expression_or_dist ) property_case_item { property_case_item } endcase
    | sequence_expr #-# property_expr
    | sequence_expr #=# property_expr
    | nexttime property_expr
    | nexttime [ constant _expression ] property_expr
    | s_nexttime property_expr
    | s_nexttime [ constant_expression ] property_expr
    | always property_expr
    | always [ cycle_delay_const_range_expression ] property_expr
    | s_always [ constant_range ] property_expr
    | s_eventually property_expr
    | eventually [ constant_range ] property_expr
    | s_eventually [ cycle_delay_const_range_expression ] property_expr
    | property_expr until property_expr
    | property_expr s_until property_expr
    | property_expr until_with property_expr
    | property_expr s_until_with property_expr
    | property_expr implies property_expr
    | property_expr iff property_expr
    | accept_on ( expression_or_dist ) property_expr
    | reject_on ( expression_or_dist ) property_expr
    | sync_accept_on ( expression_or_dist ) property_expr
    | sync_reject_on ( expression_or_dist ) property_expr
    | property_instance
    | clocking_event property_expr
property_case_item ::=
      expression_or_dist { , expression_or_dist } : property_expr ;
    | default [ : ] property_expr ;
assertion_variable_declaration ::=
      var_data_type list_of_variable_decl_assignments ;
property_instance ::=
      ps_or_hierarchical_property_identifier [ ( [ property_list_of_arguments ] ) ]
property_list_of_arguments ::=
      [property_actual_arg] { , [property_actual_arg] } { , . identifier ( [property_actual_arg] ) }
    | . identifier ( [property_actual_arg] ) { , . identifier ( [property_actual_arg] ) }
property_actual_arg ::=
      property_expr
    | sequence_actual_arg
```

*Syntax 16-16—Property construct syntax (excerpt from Annex A)*

A named property may be declared with formal arguments in the optional *property_port_list*.

Except as described in 16.12.18, 16.12.19, and 16.12.17, the rules for declaring formal arguments and default actual arguments in named properties and for instantiating named properties with actual arguments are the same as those for named sequences as described in 16.8, 16.8.1, and 16.8.2.

Rules particular to the specification and use of typed formal arguments in named properties are discussed in 16.12.18.

Rules particular to the specification and use of local variable formal arguments in named properties are discussed in 16.12.19.

A formal argument may be referenced in the body *property_spec* of the declaration of the named property. A reference to a formal argument may be written in place of various syntactic entities, including, in addition to those listed in 16.8, the following:

— *property_expr*

— *property_spec*

A named property may be instantiated prior to its declaration. A named property may be instantiated anywhere a *property_spec* may be written. A named property may be instantiated in a place where a *property_expr* may be written provided the instance does not produce an illegal **`disable iff`** clause (see the following). There may be cyclic dependencies among named properties resulting from their instantiations. A cyclic dependency among named properties results if, and only if, there is a cycle in the directed graph whose nodes are the named properties and whose edges are defined by the following rule: there is a directed edge from one named property to a second named property if, and only if, either the first named property instantiates the second named property within its declaration, including an instance within the declaration of a default actual argument, or there is an instance of the first named property that instantiates the second named property within an actual argument. Named properties with such cyclic dependencies are called *recursive* and are discussed in 16.12.17.

If $ is an actual argument to an instance of a named property, then the corresponding formal argument shall be untyped and each of its references either shall be an upper bound in a *cycle_delay_const_range_expression* or shall itself be an actual argument in an instance of a named sequence or property.

The behavior and semantics of an instance of a nonrecursive named property are the same as those of the flattened property that is obtained from the body of the declaration of the named property by the rewriting algorithm defined in F.4.1. The rewriting algorithm substitutes actual arguments for references to the corresponding formal arguments in the body of the declaration of the named property. The rewriting algorithm does not itself account for name resolution and assumes that names have been resolved prior to the substitution of actual arguments. If the flattened property is not legal, then the instance is not legal and there shall be an error.

The result of property evaluation is either true or false. Properties may be built from other properties or sequences using instantiation and the operators described in the following subclauses.

Table 16-3 lists the sequence and property operators from highest to lowest precedence and shows the associativity of the non-unary operators. The precedence for the strong and weak sequence operators is not defined because these operators require parentheses. The operators described in Table 11-2 have higher precedence than the sequence and property operators.

**Table 16-3—Sequence and property operator precedence and associativity**

| Sequence operators | Property operators | Associativity |
|---|---|---|
| `[*]`, `[=]`, `[->]` | | — |
| `##` | | Left |
| **`throughout`** | | Right |
| **`within`** | | Left |
| **`intersect`** | | Left |
| | **`not`**, **`nexttime`**, **`s_nexttime`** | — |
| **`and`** | **`and`** | Left |
| **`or`** | **`or`** | Left |
| | **`iff`** | Right |
| | **`until`**, **`s_until`**, **`until_with`**, **`s_until_with`**, **`implies`** | Right |
| | `\|->`, `\|=>`, `#-#`, `#=#` | Right |
| | **`always`**, **`s_always`**, **`eventually`**, **`s_eventually`**, **`if-else`**, **`case`** , **`accept_on`**, **`reject_on`**, **`sync_accept_on`**, **`sync_reject_on`** | — |

A **`disable iff`** clause can be attached to a *property_expr* to yield a *property_spec*.

```
disable iff (expression_or_dist) property_expr
```

The expression of the **`disable iff`** is called the *disable condition*. The **`disable iff`** clause allows preemptive resets to be specified. For an evaluation of the property_spec, there is an evaluation of the underlying *property_expr*. If the disable condition is true at anytime between the start of the attempt in the Observed region, inclusive, and the end of the evaluation attempt, inclusive, then the overall evaluation of the property results in disabled. A property has disabled evaluation if it was preempted due to a **`disable iff`** condition. A disabled evaluation of a property does not result in success or failure. Otherwise, the evaluation of the *property_spec* is the same as that of the *property_expr*. The disable condition is tested independently for different evaluation attempts of the *property_spec*. The values of variables used in the disable condition are those in the current simulation cycle, i.e., not sampled. The expression may contain a reference to an end point of a sequence by using the method `triggered` of that sequence. The disable conditions shall not contain any reference to local variables or the sequence method `matched`. If a sampled value function other than `$sampled` is used in the disable condition, the sampling clock shall be explicitly specified in its actual argument list as described in 16.9.3. Nesting of **`disable iff`** clauses, explicitly or through property instantiations, is not allowed.

## 16.12.1 Property instantiation

An instance of a named property can be used as a *property_expr* or *property_spec*. In general, the instance is legal provided the body *property_spec* of the named property can be substituted in place of the instance, with actual arguments substituted for formal arguments, and result in a legal *property_expr* or *property_spec*. For example, if an instance of a named property is used as a *property_expr* operand for any property-building operator, then the named property must not have a **`disable iff`** clause.

### 16.12.2 Sequence property

Sequence properties have three forms: *sequence_expr*, **weak**(*sequence_expr*), and **strong**(*sequence_expr*). The **strong** and **weak** operators are called *sequence operators*. **strong**(*sequence_expr*) evaluates to true if, and only if, there is a nonempty match of the *sequence_expr*. **weak**(*sequence_expr*) evaluates to true if, and only if, there is no finite prefix that witnesses inability to match the *sequence_expr*. The *sequence_expr* of a sequential property shall not admit an empty match (see 16.12.22).

If the **strong** or **weak** operator is omitted, then the evaluation of the *sequence_expr* depends on the assertion statement in which it is used. If the assertion statement is **assert property** or **assume property**, then the *sequence_expr* is evaluated as **weak**(*sequence_expr*). Otherwise, the *sequence_expr* is evaluated as **strong**(*sequence_expr*).

NOTE—The semantics for a *sequence_expr* definition in IEEE Std 1800-2009 and on is not backward compatible with IEEE Std 1800-2005. The current equivalent to a *sequence_expr* as defined in IEEE Std 1800-2005 is **strong**(*sequence_expr*).

Since only one match of a *sequence_expr* is needed for **strong**(*sequence_expr*) to hold, a property of the form **strong**(*sequence_expr*) evaluates to true if, and only if, the property **strong**(**first_match**(*sequence_expr*)) evaluates to true.

Similarly, a property of the form **weak**(*sequence_expr*) evaluates to true if, and only if, the property **weak**(**first_match**(*sequence_expr*)) evaluates to true. This is because a prefix witnesses inability to match *sequence_expr* if, and only if, it witnesses inability to match **first_match**(*sequence_expr*).

The following examples illustrate the sequential property forms:

```
property p3;
   b ##1 c;
endproperty

c1: cover property (@(posedge clk) a #-# p3);
a1: assert property (@(posedge clk) a |-> p3);
```

The sequential property `p3` is interpreted as strong in the cover property `c1`. An evaluation attempt of `c1` returns true if, and only if, `a` is true at the tick of **posedge** `clk` at which the attempt begins and both of the following conditions are satisfied:

— `b` is true at the tick of **posedge** `clk` at which the attempt begins.
— There exists a subsequent tick of **posedge** `clk` and `c` is true at the first such tick.

The sequential property `p3` is interpreted as weak in the **assert property** `a1`. An evaluation attempt of `a1` returns true if, and only if, either `a` is false at the tick of **posedge** `clk` at which the attempt begins or both of the following conditions are satisfied:

— `b` is true at the tick of **posedge** `clk` at which the attempt begins.
— If there exists a subsequent tick of **posedge** `clk`, then `c` is true at the first such tick.

### 16.12.3 Negation property

A property is a *negation* if it has the form **not** *property_expr*. For each evaluation attempt of the property, there is an evaluation attempt of *property_expr*. The keyword **not** states that the evaluation of the property returns the opposite of the evaluation of the underlying *property_expr*. Thus, if *property_expr* evaluates to true, then **not** *property_expr* evaluates to false; and if *property_expr* evaluates to false, then **not** *property_expr* evaluates to true.

The **`not`** operator switches the strength of a property. In particular, one should be careful when negating a sequence. For example, consider the following assertion:

```
a1: assert property (@clk not a ##1 b);
```

Since the sequential property `a ##1 b` is used in an assertion, it is weak. This means that if `clk` stops ticking and a holds at the last tick of `clk`, the weak sequential property `a ##1 b` will also hold beginning at that tick, and so the assertion `a1` will fail. In this case it is more reasonable to use:

```
a2: assert property (@clk not strong(a ##1 b));
```

### 16.12.4 Disjunction property

A property is a *disjunction* if it has the following form:

property_expr **`or`** property_expr

The property evaluates to true if, and only if, at least one of *property_expr1* and *property_expr2* evaluates to true.

### 16.12.5 Conjunction property

A property is a *conjunction* if it has the following form:

property_expr **`and`** property_expr

The property evaluates to true if, and only if, both *property_expr1* and *property_expr2* evaluate to true.

### 16.12.6 If-else property

A property is an *if–else* if it has either the following form:

**`if (`** expression_or_dist **`)`** property_expr

or the following form:

**`if (`** expression_or_dist **`)`** property_expr **`else`** property_expr

A property of the first form evaluates to true if, and only if, either *expression_or_dist* evaluates to false or *property_expr* evaluates to true. A property of the second form evaluates to true if, and only if, either *expression_or_dist* evaluates to true and *property_expr1* evaluates to true or *expression_or_dist* evaluates to false and *property_expr2* evaluates to true.

### 16.12.7 Implication

The implication construct specifies that the checking of a property is performed conditionally on the match of a sequential antecedent (see Syntax 16-17).

```
property_expr ::=                                          // from A.2.10
   ...
   | sequence_expr |-> property_expr
   | sequence_expr |=> property_expr
```

*Syntax 16-17—Implication syntax (excerpt from Annex A)*

This construct is used to precondition monitoring of a property expression and is allowed at the property level. The result of the implication is either true or false. The left-hand operand *sequence_expr* is called the *antecedent*, while the right-hand operand *property_expr* is called the *consequent*.

The following points should be noted for `|->` implication:

— From a given start point, the antecedent *sequence_expr* can have zero, one, or more than one successful match.

— If there is no match of the antecedent *sequence_expr* from a given start point, then evaluation of the implication from that start point succeeds and returns true.

— For each successful match of the antecedent *sequence_expr*, the consequent *property_expr* is separately evaluated. The end point of the match of the antecedent *sequence_expr* is the start point of the evaluation of the consequent *property_expr*.

— From a given start point, evaluation of the implication succeeds and returns true if, and only if, for every match of the antecedent *sequence_expr* beginning at the start point, the evaluation of the consequent *property_expr* beginning at the end point of the match succeeds and returns true.

Two forms of implication are provided: overlapped using operator `|->` and nonoverlapped using operator `|=>`. For overlapped implication, if there is a match for the antecedent *sequence_expr*, then the end point of the match is the start point of the evaluation of the consequent *property_expr*. For nonoverlapped implication, the evaluation of the consequent is described by two cases, depending on whether the implication is triggered by a nonempty match or by an empty match:

— If triggered by a nonempty match, the start point of the evaluation of the consequent *property_expr* is the clock tick after the end point of the match.

— If triggered by an empty match, the start point of the evaluation of the consequent *property_expr* is its nearest clock tick, starting from the tick when evaluation of the *sequence_expr* begins. For a singly clocked property, this coincides with the current clock tick.

Therefore,

```
sequence_expr |=> property_expr
```

is equivalent to the following:

```
sequence_expr ##1 `true |-> property_expr
```

The use of implication when multiclock sequences and properties are involved is explained in 16.13.

The following example illustrates a bus operation for data transfer from a master to a target device. When the bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, a data phase completes on any rising clock edge on which `irdy` is asserted and either `trdy` or `stop` is asserted. In this example, an asserted signal implies a value of low. The requirement for the end of a data phase can be expressed as follows:

```
let ready_exp = (irdy == 0) && ($fell(trdy) || $fell(stop));
property data_end;
   @(posedge mclk)
   $rose(data_phase) |-> ##[1:5] ready_exp;
endproperty
a1: assert property(data_end);
```

Each time the sequence `$rose(data_phase)` matches, an evaluation of the consequent property begins. In Figure 16-14, a match for `$rose(data_phase)` occurs at clock tick 2. This begins the evaluation of the consequent property. Then, at clock tick 6, the assertion attempt evaluates to true because `$fell(stop)` and `irdy==0` both evaluate to true.

In another example, `data_end_exp` is used to verify that `frame` is deasserted (value high) within two clock ticks after `data_end_exp` occurs. Further, it is also required that `irdy` is deasserted (value high) one clock tick after `frame` is deasserted.

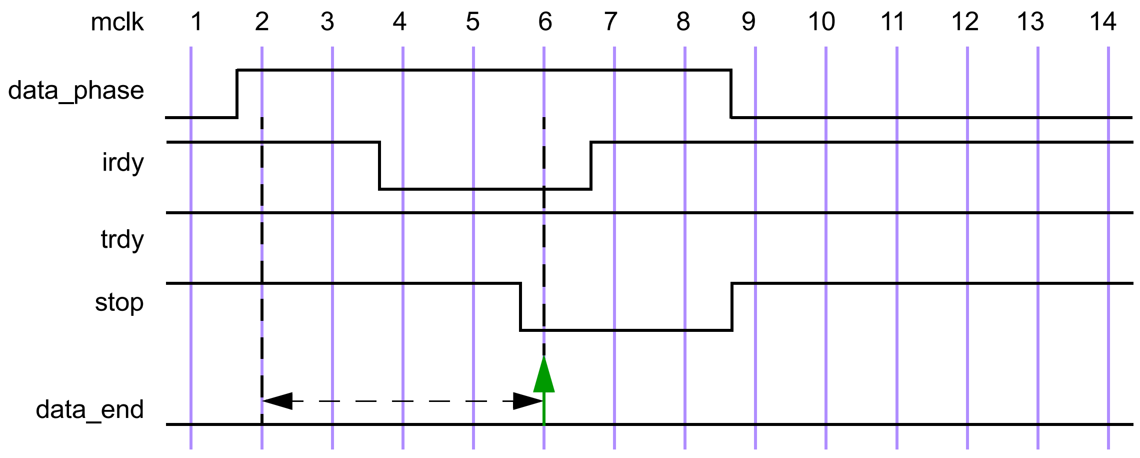


**Figure 16-14—Conditional sequence matching**

A property written to express this condition is as follows:

```
let data_end_exp = data_phase && ready_exp;
property data_end_rule1;
   @(posedge mclk)
   data_end_exp |-> ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
a2: assert property(data_end_rule1);
```

Property `data_end_rule1` first evaluates `data_end_exp` at every clock tick to test if its value is true. If the value is false, then that particular attempt to evaluate `data_end_rule1` is considered true. Otherwise, the following sequence is evaluated:

```
##[1:2] $rose(frame) ##1 $rose(irdy)
```

that specifies looking for the rising edge of `frame` within two clock ticks in the future. After `frame` toggles high, `irdy` must also toggle high after one clock tick. This is illustrated in Figure 16-15 for the evaluation attempt at clock tick 6. `data_end_exp` is acknowledged at clock tick 6. Next, `frame` toggles high at clock tick 7. Because this falls within the timing constraint imposed by `[1:2]`, it satisfies the sequence and continues to evaluate further. At clock tick 8, `irdy` is evaluated. Signal `irdy` transitions to high at clock tick 8, matching the sequence specification completely for the attempt that began at clock tick 6.

Generally, assertions are associated with preconditions so that the checking is performed only under certain specified conditions. As seen from the previous example, the `|->` operator provides this capability to specify preconditions with sequences that must be satisfied before evaluating their consequent properties. The next example modifies the preceding example to see the effect on the results of the assertion by removing the precondition for the consequent. This is shown below and illustrated in Figure 16-16.

```
property data_end_rule2;
   @(posedge mclk) ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
a3: assert property(data_end_rule2);
```

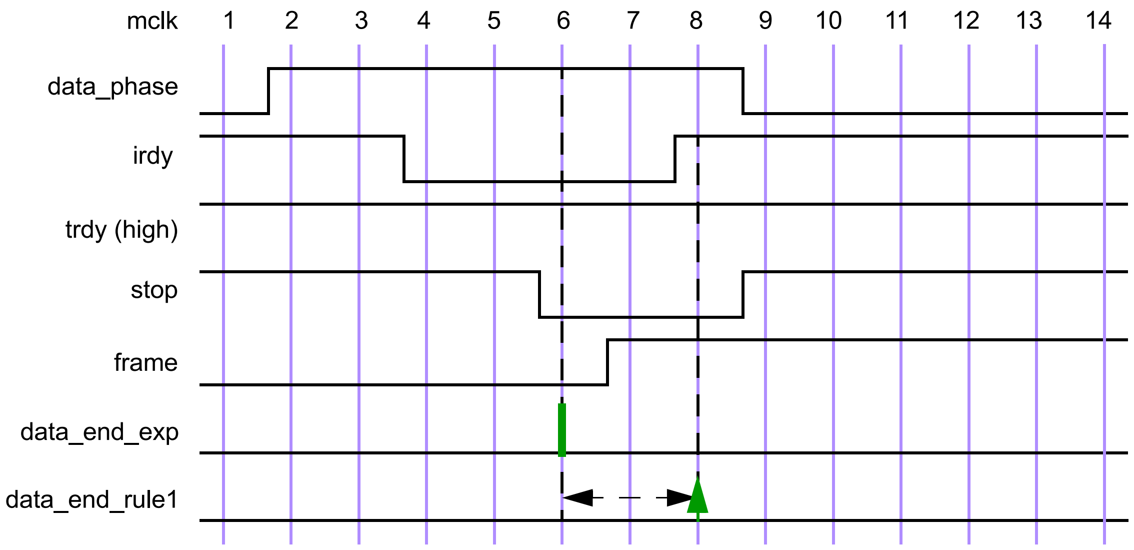


**Figure 16-15—Conditional sequences**

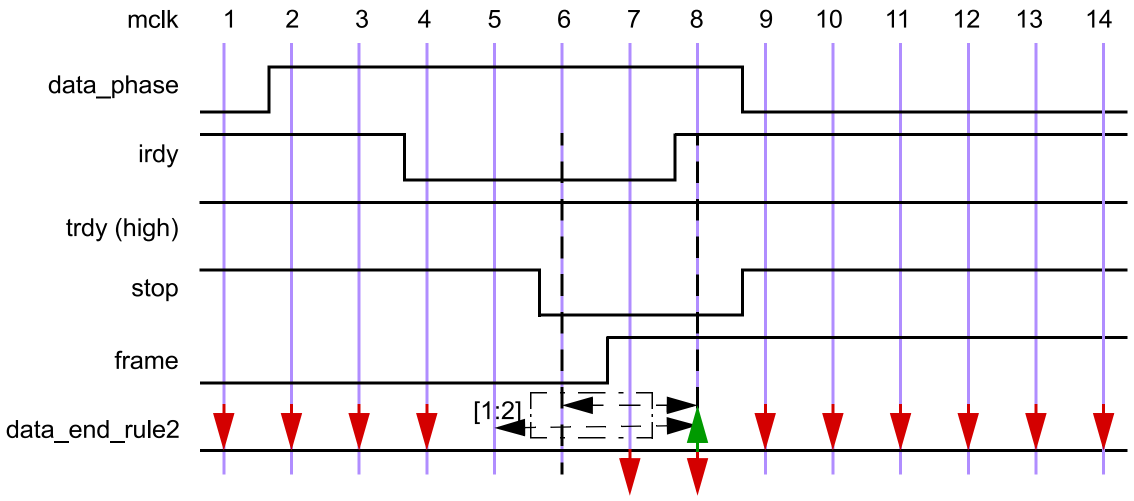


**Figure 16-16—Results without the condition**

The property is evaluated at every clock tick. For the evaluation at clock tick 1, the rising edge of signal `frame` does not occur at clock tick 2 or 3; therefore, the property fails at clock tick 1. Similarly, there is a failure at clock ticks 2, 3, and 4. For attempts starting at clock ticks 5 and 6, the rising edge of signal `frame` at clock tick 7 allows checking further. At clock tick 8, the sequences complete according to the specification, resulting in a match for attempts starting at clock ticks 5 and 6. All later attempts to match the sequence fail because `$rose(frame)` does not occur again.

Figure 16-16 shows that removing the precondition of checking `data_end_exp` from the assertion causes failures that are not relevant to the verification objective. It is important from the validation standpoint to determine these preconditions and use them to filter out inappropriate or extraneous situations.

An example of implication where the antecedent is a sequence follows:

```
(a ##1 b ##1 c) |-> (d ##1 e)
```

If the sequence `(a ##1 b ##1 c)` matches, then the sequence `(d ##1 e)` must also match. On the other hand, if the sequence `(a ##1 b ##1 c)` does not match, then the result is true.

Another example of implication is as follows:

```
property write_to_addr;
   (write_en & data_valid) ##0
   (write_en && (retire_address[0:4]==addr)) [*2] |->
   ##[3:8] write_en && !data_valid &&(write_address[0:4]==addr);
endproperty
```

This property can be coded alternatively as a nested implication:

```
property write_to_addr_nested;
   (write_en & data_valid) |->
      (write_en && (retire_address[0:4]==addr)) [*2] |->
         ##[3:8] write_en && !data_valid && (write_address[0:4]==addr);
endproperty
```

### 16.12.8 Implies and iff properties

A property is an *implies* if it has the following form:

property_expr1 **implies** property_expr2

A property of this form evaluates to true if, and only if, either *property_expr1* evaluates to false or *property_expr2* evaluates to true.

A property is an *iff* if it has the following form:

property_expr1 **iff** property_expr2

A property of this form evaluates to true if, and only if, either both *property_expr1* evaluates to false and *property_expr2* evaluates to false or both *property_expr1* evaluates to true and *property_expr2* evaluates to true.

### 16.12.9 Followed-by property

A property is a *followed-by* if it has one of the following forms that use the followed-by operators shown in Syntax 16-18.

```
property_expr ::=                                           // from A.2.10
   ...
   | sequence_expr #-# property_expr
   | sequence_expr #=# property_expr
```

*Syntax 16-18—Followed-by syntax (excerpt from Annex A)*

This clause is used to trigger monitoring of a property expression and is allowed at the property level.

The result of the followed-by is either true or false. The left-hand operand *sequence_expr* is called the *antecedent*, while the right-hand operand *property_expr* is called the *consequent*. For the followed-by property to succeed, the following must hold:

— From a given start point *sequence_expr* shall have at least one successful match.

— *property_expr* shall be successfully evaluated starting from one of the match points of the *sequence_expr*.

From a given start point, evaluation of the followed-by succeeds and returns true if, and only if, there exists a match of the antecedent *sequence_expr* beginning at the start point, and the evaluation of the consequent *property_expr* beginning at the tick specified by the rules in the next paragraph succeeds and returns true.

Two forms of followed-by are provided: overlapped using operator `#-#` and nonoverlapped using operator `#=#`. For overlapped followed-by, there shall be a match for the antecedent *sequence_expr*, where the end point of this match is the start point of the evaluation of the consequent *property_expr*. For nonoverlapped followed-by, the evaluation of the consequent is described by two cases, depending on whether the antecedent *sequence_expr* attains a nonempty match or an empty match:

— If a nonempty match, the start point of the evaluation of the consequent *property_expr* is the clock tick after the end point of the match.

— If an empty match, the start point of the evaluation of the consequent *property_expr* is its nearest clock tick, starting from the tick when evaluation of the *sequence_expr* begins. For a singly clocked property, this coincides with the current clock tick.

The followed-by operators are the duals of the implication operators. Therefore, *sequence_expr* `#-#` *property_expr* is equivalent to the following:

```
not (sequence_expr |-> not property_expr)
```

and *sequence_expr* `#=#` *property_expr* is equivalent to the following:

```
not (sequence_expr |=> not property_expr)
```

*Examples:*

```
property p1;
   ##[0:5] done #-# always !rst;
endproperty

property p2;
   ##[0:5] done #=# always !rst;
endproperty
```

Property `p1` says that `done` shall be asserted at some clock tick during the first 6 clock ticks, and starting from one of the clock ticks when `done` is asserted, `rst` shall always be low. Property `p2` says that `done` shall be asserted at some clock tick during the first 6 clock ticks, and starting the clock tick after one of the clock ticks when `done` is asserted, `rst` shall always be low.

*sequence_expr* `#-#` **`strong`**`(`*sequence_expr1*`)` is semantically equivalent to **`strong`**`(`*sequence_expr* `##0` *sequence_expr1*`)`, and *sequence_expr* `#=#` **`strong`**`(`*sequence_expr1*`)` is semantically equivalent to **`strong`**`(`*sequence_expr* `##1` *sequence_expr1*`)`.

A followed-by operator is especially convenient for specifying a **`cover property`** directive over a sequence followed by a property.

### 16.12.10 Nexttime property

A property is a *nexttime* if it has one of the following forms that use the nexttime operators:

— Weak nexttime

**nexttime** property_expr

The weak nexttime property **nexttime** *property_expr* evaluates to true if, and only if, either the *property_expr* evaluates to true beginning at the next clock tick or there is no further clock tick.

— Indexed form of weak nexttime

**nexttime [** constant_expression **]** property_expr

The indexed weak nexttime property **nexttime** [*constant_expression*] *property_expr* evaluates to true if, and only if, either there are not *constant_expression* clock ticks or *property_expr* evaluates to true beginning at the last of the next *constant_expression* clock ticks.

— Strong nexttime

**s_nexttime** property_expr

The strong nexttime property **s_nexttime** *property_expr* evaluates to true if, and only if, there exists a next clock tick and *property_expr* evaluates to true beginning at that clock tick.

— Indexed form of strong nexttime

**s_nexttime [** constant_expression **]** property_expr

The indexed strong nexttime property **s_nexttime** [*constant_expression*] *property_expr* evaluates to true if, and only if, there exist *constant_expression* clock ticks and *property_expr* evaluates to true beginning at the last of the next *constant_expression* clock ticks.

The number of clock ticks given by *constant_expression* shall be a non-negative integer constant expression.

The preceding explanations refer to the case where the nexttime property is evaluated in a time step that is a tick of the clock of the nexttime property. When the nexttime property is evaluated in a time step that is not a tick of the clock of the nexttime property, an alignment to the tick of the clock of the nexttime property should be applied before the preceding description. Thus, it is more precise to say that **s_nexttime**[n] *property_expr* evaluates to true if, and only if, there exist n+1 ticks of the clock of the nexttime property, including the current time step, and *property_expr* evaluates to true on the n+1 clock tick, where counting starts at the current time step. In particular **nexttime**[0] and **s_nexttime**[0] act as alignment operators.

The comments in the following examples describe the conditions for the properties to be evaluated to true:

```
// if the clock ticks once more, then a shall be true at the next clock tick
property p1;
   nexttime a;
endproperty

// the clock shall tick once more and a shall be true at the next clock tick.
property p2;
   s_nexttime a;
endproperty

// as long as the clock ticks, a shall be true at each future clock tick
// starting from the next clock tick
property p3;
   nexttime always a;
```

```
endproperty

// the clock shall tick at least once more and as long as it ticks, a shall
// be true at every clock tick starting from the next one
property p4;
   s_nexttime always a;
endproperty

// if the clock ticks at least once more, it shall tick enough times for a to
// be true at some point in the future starting from the next clock tick
property p5;
   nexttime s_eventually a;
endproperty

// a shall be true sometime in the strict future
property p6;
   s_nexttime s_eventually a;
endproperty

// if there are at least two more clock ticks, a shall be true at the second
// future clock tick
property p7;
   nexttime[2] a;
endproperty

// there shall be at least two more clock ticks, and a shall be true at the
// second future clock tick
property p8;
   s_nexttime[2] a;
endproperty
```

### 16.12.11 Always property

A property is an *always* if it has one of the following forms that use the always operators:

— Weak always

**always** property_expr

A property **always** *property_expr* evaluates to true if, and only if, *property_expr* holds at every current or future clock tick.

— Ranged form of weak always

**always [** cycle_delay_const_range_expression **]** property_expr

A property **always** [*cycle_delay_const_range_expression*] *property_expr* evaluates to true if, and only if, *property_expr* holds at every current or future clock tick that is within the range of clock ticks specified by *cycle_delay_const_range_expression*. It is not required that all clock ticks within this range exist. The range for a weak always may be unbounded.

— Ranged form of strong always

**s_always [** constant_range **]** property_expr

A property **s_always** [*constant_range*] *property_expr* evaluates to true if, and only if, all current or future clock ticks specified by *constant_range* exist and *property_expr* holds at each of these clock ticks. The range for a strong always shall be bounded.

The range of clock ticks given by *constant_range* shall adhere to the following restrictions. The minimum number of clock ticks is defined by a non-negative integer constant expression; and the maximum number of clock ticks either is defined by a non-negative integer constant expression or is $, indicating a finite, but unbounded, maximum. If both the minimum and maximum numbers of clock ticks are defined by non-negative integer constant expressions (see 11.2.1), then the minimum number shall be less than or equal to the maximum number.

The preceding explanations refer to the case where the always property is evaluated in a time step that is a tick of the clock of the always property. When the always property is evaluated in a time step that is not a tick of the clock of the always property, an alignment to the tick of the clock of the always property should be applied before the preceding description. Thus, it is more precise to say that **s_always**`[n:m]` *property_expr* evaluates to true if, and only if, there exist `m+1` ticks of the clock of the always property, including the current time step, and *property_expr* evaluates to true beginning in all of the `n+1` to `m+1` clock ticks, where counting starts at the current time step.

There is also the implicit always that is associated with concurrent assertions (see 16.5). A verification statement that is not placed inside an initial block specifies that an evaluation attempt of its top-level property shall begin at each occurrence of its leading clocking event. In the following two examples, there is a one-to-one correspondence between the evaluation attempts of `p` specified by the implicit always from the verification statement `implicit_always` and the evaluation attempts of `p` specified by the explicit **always** operator in `explicit_always`:

Implicit form:

```
implicit_always: assert property(p);
```

Explicit form:

```
initial explicit_always: assert property(always p);
```

This is not shown as a practical example, but only for illustration of the meaning of **always**.

*Examples:*

```
initial a1: assume property( @(posedge clk) reset[*5] #=# always !reset);

property p1;
   a ##1 b |=> always c;
endproperty

property p2;
   always [2:5] a;
endproperty

property p3;
   s_always [2:5] a;
endproperty

property p4;
   always [2:$] a;
endproperty

property p5;
   s_always [2:$] a; // Illegal
endproperty
```

The assertion `a1` says that `reset` shall be true for the first 5 clock ticks and then remain 0 for the rest of the computation. The assumption is being evaluated once starting at the first clock tick. The property `p1` evaluates to true provided that if `a` is true at the first clock tick and `b` is true at the second clock tick, then `c` shall be true at every clock tick that follows the second. The properties `p2` and `p3` evaluate to true provided that `a` is true at each of the second through fifth clock ticks after the starting clock tick of the evaluation attempt. Property `p3` evaluates to true provided that these clock ticks exist, while property `p2` does not require that. The property `p4` evaluates to true if, and only if, `a` is true at every clock tick that is at least two clock ticks after the starting clock tick of the evaluation attempt. These clock ticks are not required to exist. The property `p5` is illegal since specifying an unbounded range is not permitted with the strong form of an always property.

### 16.12.12 Until property

A property is an *until* if it has one of the following forms that use the until operators:

— Weak non-overlapping form

property_expr1 **`until`** property_expr2

— Strong non-overlapping form

property_expr1 **`s_until`** property_expr2

— Weak overlapping form

property_expr1 **`until_with`** property_expr2

— Strong overlapping form

property_expr1 **`s_until_with`** property_expr2

An until property of the non-overlapping form evaluates to true if *property_expr1* evaluates to true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until at least one tick before a clock tick where *property_expr2* evaluates to true. An until property of one of the overlapping forms evaluates to true if *property_expr1* evaluates to true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until and including a clock tick at which *property_expr2* evaluates to true. An until property of one of the strong forms requires a current or future clock tick exist at which *property_expr2* evaluates to true, while an until property of one of the weak forms does not make this requirement. An until property of one of the weak forms evaluates to true if *property_expr1* evaluates to true at each clock tick, even if *property_expr2* never holds.

*Examples:*

```
property p1;
   a until b;
endproperty

property p2;
   a s_until b;
endproperty

property p3;
   a until_with b;
endproperty

property p4;
   a s_until_with b;
endproperty
```

Property `p1` evaluates to true if, and only if, `a` is true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until, but not necessarily including, a clock tick at which `b` is true. If there is no current or future clock tick at which b is true, than a shall be true at every current or future clock tick. If b is true at the starting clock tick of the evaluation attempt, then a need not be true at that clock tick. The property p2 evaluates to true provided that there exists a current or future clock tick at which b is true and that a is true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until, but not necessarily including, the clock tick at which b is true. If b is true at the starting clock tick of the evaluation attempt, then a need not be true at that clock tick. The property p3 evaluates to true provided that a is true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until and including a clock tick at which b is true. If there is no current or future clock tick at which b is true, than a shall be true at every current or future clock tick. The property p4 evaluates to true provided there exists a current or future clock tick at which b is true and that a is true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until and including the clock tick at which b is true. The property p4 is equivalent to **strong**(a[*1:$] ##0 b) (here a and b are Boolean expressions).

### 16.12.13 Eventually property

A property is an *eventually* if it has one of the following forms that use the eventually operators:

— Strong eventually

**s_eventually** property_expr

A property s_eventually property_expr evaluates to true if, and only if, there exists a current or future clock tick at which property_expr evaluates to true.

— Ranged form of weak eventually

**eventually [** constant_range **]** property_expr

A property eventually [constant_range] property_expr evaluates to true if, and only if, either there exists a current or future clock tick within the range specified by constant_range at which property_expr evaluates to true or not all the current or future clock ticks within the range specified by constant_range exist. The range for a weak eventually shall be bounded.

— Ranged form of strong eventually

**s_eventually [** cycle_delay_const_range_expression **]** property_expr

A property s_eventually [cycle_delay_const_range_expression] property_expr evaluates to true if, and only if, there exists a current or future clock tick within the range specified by cycle_delay_const_range_expression at which property_expr evaluates to true. The range for a strong eventually may be unbounded.

In the following examples, a and b are Boolean expressions:

```
property p1;
   s_eventually a;
endproperty

property p2;
   s_eventually always a;
endproperty

property p3;
   always s_eventually a;
endproperty

property p4;
   eventually [2:5] a;
endproperty
```

```
property p5;
   s_eventually [2:5] a;
endproperty

property p6;
   eventually [2:$] a; // Illegal
endproperty

property p7;
   s_eventually [2:$] a;
endproperty
```

The property `p1` evaluates to true if, and only if, there exists a current or future clock tick at which `a` is true. It is equivalent to **`strong`**`(##[*0:$] a)`. The property `p2` evaluates to true if, and only if, there exists a current or future clock tick such that `a` is true both at that clock tick and also at every subsequent clock tick. On a computation with infinitely many clock ticks, the property `p3` evaluates to true if, and only if, `a` is true at infinitely many of those clock ticks. On a computation with finitely many clock ticks, the property `p3` evaluates to true provided that if there is at least one clock tick, then `a` holds at the last clock tick. The property `p4` evaluates to true provided that if the second through fifth clock ticks from the starting clock tick of the evaluation attempt all exist, then `a` is true at one of these clock ticks. `p4` is equivalent to **`weak`**`(##[2:5] a)`. The property `p5` evaluates to true if, and only if, there exists a clock tick at which `a` is true and that it is between the second and fifth clock ticks, inclusive, from the starting clock tick of the evaluation attempt. `p5` is equivalent to **`strong`**`(##[2:5] a)`. The property `p7` evaluates to true if, and only if, there exists a clock tick at which `a` is true and that it is no earlier than the second clock tick after the starting clock tick of the evaluation attempt.

The preceding explanations refer to the case where the eventually property is evaluated in a time step that is a tick of the clock of the eventually property. When the eventually property is evaluated in a time step that is not a tick of the clock of the eventually property, an alignment to the tick of the clock of the eventually property should be applied before the preceding description. Thus, it is more precise to say that **`s_eventually`**`[n:m]` *property_expr* evaluates to true if, and only if, there exist at least `n+1` ticks of the clock of the eventually property, including the current time step, and *property_expr* evaluates to true beginning in one of the `n+1` to `m+1` clock ticks, where counting starts at the current time step.

### 16.12.14 Abort properties

A property is an *abort* if it has one of the following forms:

**`accept_on (`** expression_or_dist **`)`** property_expr

**`reject_on (`** expression_or_dist **`)`** property_expr

**`sync_accept_on (`** expression_or_dist **`)`** property_expr

**`sync_reject_on (`** expression_or_dist **`)`** property_expr

where the *expression_or_dist* is called the *abort condition*. The properties **`accept_on`** and **`reject_on`** are called *asynchronous abort properties*, and the properties **`sync_accept_on`** and **`sync_reject_on`** are called *synchronous abort properties*.

For an evaluation of **`accept_on`** `(`*expression_or_dist*`)` *property_expr* and of **`sync_accept_on`** `(`*expression_or_dist*`)` *property_expr*, there is an evaluation of the underlying *property_expr*. If during the evaluation, the abort condition becomes true, then the overall evaluation of the property results in true. Otherwise, the overall evaluation of the property is equal to the evaluation of the *property_expr*.

For an evaluation of **reject_on** (*expression_or_dist*) *property_expr* and of **sync_reject_on** (*expression_or_dist*) *property_expr*, there is an evaluation of the underlying *property_expr*. If during the evaluation, the abort condition becomes true, then the overall evaluation of the property results in false. Otherwise, the overall evaluation of the property is equal to the evaluation of the *property_expr*.

The operators **accept_on** and **reject_on** are evaluated at the granularity of the simulation time step like **disable iff**, but their abort condition is evaluated using sampled value as a regular Boolean expression in assertions. The operators **accept_on** and **reject_on** represent asynchronous resets.

The operators **sync_accept_on** and **sync_reject_on** are evaluated at the simulation time step when the clocking event happens, unlike **disable iff**, **accept_on**, and **reject_on**. Their abort condition is evaluated using sampled value as for **accept_on** and **reject_on**. The operators **sync_accept_on** and **sync_reject_on** represent synchronous resets.

The semantics of **accept_on** is similar to **disable iff**, except for the following differences:

— **accept_on** operates at the property level rather than the concurrent assertion level.

— **accept_on** uses sampled values.

— While a disable condition of a **disable iff** in a *property_spec* may cause an evaluation of the *property_spec* to be disabled, an abort condition of **accept_on** in a *property_expr* may cause the evaluation of the *property_expr* to be true.

The semantics of **reject_on**(*expression_or_dist*) *property_expr* is the same as **not**(**accept_on**(*expression_or_dist*) **not**(*property_expr*)).

The semantics of **sync_accept_on** is similar to **accept_on**, except that it evaluates only at the time steps when the clocking event happens.

The semantics of **sync_reject_on**(*expression_or_dist*) *property_expr* is the same as **not**(**sync_accept_on**(*expression_or_dist*) **not**(*property_expr*)).

Any nesting of abort operators **accept_on**, **reject_on**, **sync_accept_on**, and **sync_reject_on** is allowed.

For example, whenever `go` is high, followed by two occurrences of `get` being high, then `stop` cannot be high until after `put` is asserted twice (not necessarily consecutive).

```
    assert property (@(clk) go ##1 get[*2] |-> reject_on(stop) put[->2]);
```

In this example the `stop` is an asynchronous abort, its value is checked even between ticks of `clk`. The following is the synchronous version of the same example:

```
    assert property (@(clk) go ##1 get[*2] |-> sync_reject_on(stop) put[->2]);
```

Here `stop` is checked only at the `clk` ticks. The latter assertion can also be written as follows:

```
    assert property (@(clk) go ##1 get[*2] |-> !stop throughout put[->2]);
```

When the abort condition occurs at the same time step where the evaluation of the *property_expr* ends, the abort condition takes precedence. For example:

```
     property p; (accept_on(a) p1) and (reject_on(b) p2); endproperty
```

If `a` becomes true during the evaluation of `p1`, the first term is ignored in deciding the truth of `p`. On the other hand, if `b` becomes true during the evaluation of `p2` then `p` evaluates to false.

```
property p; (accept_on(a) p1) or (reject_on(b) p2); endproperty
```

If `a` becomes true during the evaluation of `p1` then `p` evaluates to true. On the other hand, if `b` becomes true during the evaluation of `p2`, then the second term is ignored in deciding the truth of `p`.

```
property p; not (accept_on(a) p1); endproperty
```

**not** inverts the effect of the abort operator. Therefore, if `a` becomes true while evaluating `p1`, property `p` evaluates to false.

Nested **accept_on**, **reject_on**, **sync_accept_on**, and **sync_reject_on** operators are evaluated in the lexical order (left to right). Therefore, if two nested operator conditions become true in the same time step during the evaluation of the argument property, then the outermost operator takes precedence. For example:

```
property p; accept_on(a) reject_on(b) p1; endproperty
```

If `a` becomes true in the same time step as `b` and during the evaluation of `p1`, then `p` succeeds in that time step. If `b` becomes true before `a` and during the evaluation of `p1`, then `p` fails.

The abort conditions may contain sampled value functions (see 16.9.3). When sampled value functions other than `$sampled` are used in the abort condition, the clock argument shall be explicitly specified. Abort conditions shall not contain any reference to local variables and the sequence methods `triggered` and `matched`.

### 16.12.15 Weak and strong operators

The property operators **s_nexttime**, **s_always**, **s_eventually**, **s_until**, **s_until_with**, and sequence operator **strong** are strong: they require that some terminating condition happen in the future, and this includes the requirement that the property clock ticks enough time to enable the condition to happen.

The property operators **nexttime**, **always**, **until**, **eventually**, **until_with**, and sequence operator **weak** are weak: they do not impose any requirement on the terminating condition, and do not require the clock to tick.

The concept of weak and strong operators is closely related to an important notion of safety properties. Safety properties have the characteristic that all their failures happen at a finite time. For example, the property **always** `a` is a safety property since it is violated only if after finitely many clock ticks there is a clock tick at which `a` is false, even if there are infinitely many clock ticks in the computation. To the contrary, a failure of the property **s_eventually** `a` on a computation with infinitely many clock ticks cannot be identified at a finite time; if it is violated, the value of `a` must be false at each of the infinitely many clock ticks.

### 16.12.16 Case

The *case* property statement is a multiway decision that tests whether a Boolean expression matches one of a number of other Boolean expressions and branches accordingly (see Syntax 16-19).

property_expr ::= *// from A.2.10*
  ...
  | **case (** expression_or_dist **)** property_case_item { property_case_item } **endcase**

...

property_case_item ::=
  expression_or_dist { **,** expression_or_dist } **:** property_expr **;**
  | **default** [ **:** ] property_expr **;**

*Syntax 16-19—Property statement case syntax (excerpt from Annex A)*

The *default* statement shall be optional. Use of multiple default statements in one property case statement shall be illegal.

A simple example of the use of the case property statement is the decoding of variable delay to produce a delay between the check of two signals as follows:

```
property p_delay(logic [1:0] delay);
   case (delay)
      2'd0    : a && b;
      2'd1    : a ##2 b;
      2'd2    : a ##4 b;
      2'd3    : a ##8 b;
      default: 0;        // cause a failure if delay has x or z values
   endcase
endproperty
```

During the linear search, if one of the case item expressions matches the case expression given in parentheses, then the property statement associated with that case item shall be evaluated, and the linear search shall terminate. If there is a default case item, it is ignored during this linear search. If all comparisons fail and the default item is given, then the default item property statement shall be executed. If the default property statement is not given and all of the comparisons fail, then none of the case item property statements shall be evaluated and the evaluation of the case property statement from that start point succeeds and returns true (vacuously).

The rules for comparing the case expression to the case item expressions are described in 12.5.

### 16.12.17 Recursive properties

SystemVerilog allows recursive properties. A named property is recursive if its declaration involves an instantiation of itself. Recursion provides a flexible framework for coding properties to serve as ongoing assumptions, obligations, or coverage monitors.

For example:

```
property prop_always(p);
   p and (1'b1 |=> prop_always(p));
endproperty
```

is a recursive property that says that the formal argument property `p` must hold at every cycle. This example is useful if the ongoing requirement that property `p` hold applies after a complicated triggering condition encoded in sequence `s`:

```
property p1(s,p);
   s |=> prop_always(p);
endproperty
```

As another example, the recursive property

```
property prop_weak_until(p,q);
   q or (p and (1'b1 |=> prop_weak_until(p,q)));
endproperty
```

says that formal argument property p must hold at every cycle up to, but not including, the first cycle at which formal argument property q holds. Formal argument property q is not required ever to hold, however. This example is useful if p must hold at every cycle after a complicated triggering condition encoded in sequence s, but the requirement on p is lifted by q:

```
property p2(s,p,q);
   s |=> prop_weak_until(p,q);
endproperty
```

More generally, several properties can be mutually recursive. For example:

```
property check_phase1;
   s1 |-> (phase1_prop and (1'b1 |=> check_phase2));
endproperty
property check_phase2;
   s2 |-> (phase2_prop and (1'b1 |=> check_phase1));
endproperty
```

There are four restrictions on recursive property declarations, as follows:

— *Restriction 1:* The negation operator **not** and strong operators **s_nexttime**, **s_eventually**, **s_always**, **s_until**, and **s_until_with** cannot be applied to any property expression that instantiates a recursive property. In particular, the negation of a recursive property cannot be asserted or used in defining another property.

  The following are examples of illegal property declarations that violate Restriction 1:

```
property illegal_recursion_1(p);
   not prop_always(not p);
endproperty

property illegal_recursion_2(p);
   p and (1'b1 |=> not illegal_recursion_2(p));
endproperty
```

  Furthermore, **not** cannot be applied to any property expression that instantiates a property that depends on a recursive property. The precise definition of dependency is given in F.7.

— *Restriction 2:* The operator **disable iff** cannot be used in the declaration of a recursive property. This restriction is consistent with the restriction that **disable iff** cannot be nested.

  The following is an example of an illegal property declaration that violates Restriction 2:

```
property illegal_recursion_3(p);
   disable iff (b)
   p and (1'b1 |=> illegal_recursion_3(p));
endproperty
```

  The intent of illegal_recursion_3 can be written legally as follows:

```
property legal_3(p);
   disable iff (b) prop_always(p);
endproperty
```

because legal_3 is not a recursive property.

— *Restriction 3:* If p is a recursive property, then, in the declaration of p, every instance of p must occur after a positive advance in time. In the case of mutually recursive properties, all recursive instances must occur after positive advances in time.

The following is an example of an illegal property declaration that violates Restriction 3:

```
property illegal_recursion_4(p);
   p and (1'b1 |-> illegal_recursion_4(p));
endproperty
```

If this form were legal, the recursion would be stuck in time, checking p over and over again at the same cycle.

— *Restriction 4:* For every recursive instance of property q in the declaration of property p, each actual argument expression e of the instance satisfies at least one of the following conditions:

- e is itself a formal argument of p.
- No formal argument of p appears in e.
- e is bound to a local variable formal argument of q.

For example:

```
property fibonacci1 (local input int a, b, n, int fib_sig);
   (n > 0)
   |->
   (
      (fib_sig == a)
      and
      (1'b1 |=> fibonacci1(b, a + b, n - 1, fib_sig))
   );
endproperty
```

is a legal declaration, but

```
property fibonacci2 (int a, b, n, fib_sig);
   (n > 0)
   |->
   (
      (fib_sig == a)
      and
      (1'b1 |=> fibonacci2(b, a + b, n - 1, fib_sig))
   );
endproperty
```

is not legal because, in the recursive instance fibonacci2(b, a+b, n-1, fib_sig), the actual argument expressions a+b, n-1 are not themselves formal arguments of fibonacci2, are not bound to local variable formal arguments, and yet formal arguments of fibonacci2 appear in these expressions.

The operators **accept_on**, **reject_on**, **sync_accept_on**, and **sync_reject_on** may be used inside a recursive property. For example, the following uses of **accept_on** and **reject_on** in a property are legal:

```
property p3(p, bit b, abort);
   (p and (1'b1 |=> p4(p, b, abort)));
endproperty

property p4(p, bit b, abort);
```

```
    accept_on(b) reject_on(abort) p3(p, b, abort);
endproperty
```

Recursive properties can represent complicated requirements, such as those associated with varying numbers of data beats, out-of-order completions, retries, etc. Following is an example of using a recursive property to check complicated conditions of this kind.

Suppose that write data must be checked according to the following conditions:

— Acknowledgment of a write request is indicated by the signal `write_request` together with `write_request_ack`. When a write request is acknowledged, it gets a 4-bit tag, indicated by signal `write_reqest_ack_tag`. The tag is used to distinguish data beats for multiple write transactions in flight at the same time.
— It is understood that distinct write transactions in flight at the same time must be given distinct tags. For simplicity, this condition is not a part of what is checked in this example.
— Each write transaction can have between 1 data beat and 16 data beats, and each data beat is 8 bits. There is a model of the expected write data that is available at acknowledgment of a write request. The model is a 128-bit vector. The most significant group of 8 bits represents the expected data for the first beat, the next group of 8 bits represents the expected data for the second beat (if there is a second beat), and so forth.
— Data transfer for a write transaction occurs after acknowledgment of the write request and, barring retry, ends with the last data beat. The data beats for a single write transaction occur in order.
— A data beat is indicated by the `data_valid` signal together with the signal `data_valid_tag` to determine the relevant write transaction. The signal data are valid with `data_valid` and carry the data for that beat. The data for each beat must be correct according to the model of the expected write data.
— The last data beat is indicated by signal `last_data_valid` together with `data_valid` and `data_valid_tag`. For simplicity, this example does not represent the number of data beats and does not check that `last_data_valid` is signaled at the correct beat.
— At any time after acknowledgment of the write request, but not later than the cycle after the last data beat, a write transaction can be forced to retry. Retry is indicated by the signal `retry` together with signal `retry_tag` to identify the relevant write transaction. If a write transaction is forced to retry, then its current data transfer is aborted, and the entire data transfer must be repeated. The transaction does not re-request, and its tag does not change.
— There is no limit on the number of times a write transaction can be forced to retry.
— A write transaction completes the cycle after the last data beat provided it is not forced to retry in that cycle.

The following is code to check these conditions:

```
property check_write;

    logic [0:127] expected_data;  // local variable to sample model data
    logic [3:0] tag;              // local variable to sample tag

    disable iff (reset)
    (
       write_request && write_request_ack,
       expected_data = model_data,
       tag = write_request_ack_tag
    )
    |=>
    check_write_data_beat(expected_data, tag, 4'h0);
```

```
endproperty

property check_write_data_beat
(
   local input logic [0:127] expected_data,
   local input logic [3:0]   tag, i
);
   (
      (data_valid && (data_valid_tag == tag))
      ||
      (retry && (retry_tag == tag))
   )[->1]
   |->
   (
      (
         (data_valid && (data_valid_tag == tag))
         |->
         (data == expected_data[i*8+:8])
      )
      and
      (
         if (retry && (retry_tag == tag))
         (
            1'b1 |=> check_write_data_beat(expected_data, tag, 4'h0)
         )
         else if (!last_data_valid)
         (
            1'b1 |=> check_write_data_beat(expected_data, tag, i+4'h1)
         )
         else
         (
            ##1 (retry && (retry_tag == tag))
            |=>
            check_write_data_beat(expected_data, tag, 4'h0)
         )
      )
   );

endproperty
```

### 16.12.18 Typed formal arguments in property declarations

The rules in 16.8.1 for typed formal arguments and their corresponding actual arguments apply to named properties, except as described next.

If a formal argument of a named property is typed, then the type shall be **property**, **sequence**, **event**, or one of the types allowed in 16.6. If the formal argument is of type **property**, then the corresponding actual argument shall be a *property_expr*, and each reference to the formal argument shall be in a place where a *property_expr* may be written.

For example, a Boolean expression or a *sequence_expr* may be passed as actual argument to a formal argument of type **property** because each is a *property_expr*. A formal argument of type **property** may not be referenced as the antecedent of `|->` or `|=>` (see 16.12.7), regardless of the corresponding actual argument, because a *property_expr* may not be written in that position.

### 16.12.19 Local variable formal arguments in property declarations

The rules in 16.8.2 for local variable formal arguments and their corresponding actual arguments apply to named properties, except as described next.

A local variable formal argument of a named property shall have direction **input**, either specified explicitly or inferred. It shall be illegal to declare a local variable formal argument of a named property with direction **inout** or **output**.

### 16.12.20 Property examples

The following examples illustrate the property forms:

```
property rule1;
   @(posedge clk) a |-> b ##1 c ##1 d;
endproperty
property rule2;
   @(clkev) disable iff (e) a |-> not(b ##1 c ##1 d);
endproperty
```

Property `rule2` negates the sequence `(b ##1 c ##1 d)` in the consequent of the implication. `clkev` specifies the clock for the property.

```
property rule3;
   @(posedge clk) a[*2] |-> ((##[1:3] c) or (d |=> e));
endproperty
```

Property `rule3` says that if `a` holds and `a` also held last cycle, then either `c` must hold at some point one to three cycles after the current cycle or, if `d` holds in the current cycle, then `e` must hold one cycle later.

```
property rule4;
   @(posedge clk) a[*2] |-> ((##[1:3] c) and (d |=> e));
endproperty
```

Property `rule4` says that if `a` holds and `a` also held last cycle, then `c` must hold at some point one to three cycles after the current cycle and, if `d` holds in the current cycle, then `e` must hold one cycle later.

```
property rule5;
   @(posedge clk)
   a ##1 (b || c)[->1] |->
      if (b)
         (##1 d |-> e)
      else // c
         f ;
endproperty
```

Property `rule5` has `a` followed by the next occurrence of either `b` or `c` as its antecedent. The consequent uses **if–else** to split cases on which of `b` or `c` is matched first.

```
property rule6(x,y);
   ##1 x |-> y;
endproperty
property rule5a;
   @(posedge clk)
   a ##1 (b || c)[->1] |->
      if (b)
```

```
          rule6(d,e)
       else // c
          f ;
endproperty
```

Property `rule5a` is equivalent to `rule5`, but it uses an instance of `rule6` as a property expression.

A property can optionally specify an event control for the clock. The clock derivation and resolution rules are described in 16.16.

A named property can be instantiated by referencing its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. Like sequence declarations, variables used within a property that are not formal arguments to the property are resolved hierarchically from the scope in which the property is declared.

Properties that use more than one clock are described in 16.13.

### 16.12.21 Finite-length versus infinite-length behavior

The formal semantics in F.5 defines whether a given property holds on a given behavior. How the outcome of this evaluation relates to the design depends on the behavior that was analyzed. In dynamic verification, only behaviors that are finite in length are considered. In such a case, SystemVerilog defines the following four levels of satisfaction of a property:

- Holds strongly
  - No bad states have been seen.
  - All future obligations have been met.
  - The property will hold on any extension of the path.
- Holds (but does not hold strongly)
  - No bad states have been seen.
  - All future obligations have been met.
  - The property may or may not hold on a given extension of the path.
- Pending
  - No bad states have been seen.
  - Future obligations have not been met.
  - The property may or may not hold on a given extension of the path.
- Fails
  - A bad state has been seen.
  - Future obligations may or may not have been met.
  - The property will not hold on any extension of the path.

### 16.12.22 Nondegeneracy

It is possible to define sequences that can never be matched. For example:

```
(1'b1) intersect(1'b1 ##1 1'b1)
```

It is also possible to define sequences that admit only empty matches. For example:

```
1'b1[*0]
```

A zero consecutive repetition means that there is no sample taken at any clock tick. Therefore, such a sequence can only match on an empty trace (as formally defined in F.4.3). A sequence may admit both empty and nonempty matches, for example, `a[*0:2]`. This sequence admits an empty match and up to two nonempty matches: `a` and `a[*2]`.

A sequence that admits no match or that admits only empty matches is called *degenerate*. A sequence that admits at least one nonempty match is called *nondegenerate*. A more precise definition of nondegeneracy is given in F.5.2 and F.5.5.

The following restrictions apply:

a) Any sequence that is used as a property shall be nondegenerate and shall not admit any empty match.
b) Any sequence that is used as the antecedent of an overlapping implication (`|->`) shall be nondegenerate.
c) Any sequence that is used as the antecedent of a nonoverlapping implication (`|=>`) shall admit at least one match. Such a sequence can admit only empty matches.

The reason for these restrictions is because the use of degenerate sequences in forbidden ways results in counterintuitive property semantics, especially when the property is combined with a **`disable iff`** clause.

## 16.13 Multiclock support

Multiclock sequences and properties can be specified as described in the following subclauses.

### 16.13.1 Multiclocked sequences

Multiclocked sequences are built by concatenating singly clocked subsequences using the single-delay concatenation operator ##1 or the zero-delay concatenation operator ##0. The single delay indicated by ##1 is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins. The zero delay indicated by ##0 is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest possibly overlapping tick of the second clock, where the second sequence begins.

*Example 1:*

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

A match of this sequence starts with a match of `sig0` at **`posedge`** `clk0`. Then ##1 moves the time to the nearest strictly subsequent **`posedge`** `clk1`, and the match of the sequence ends at that point with a match of `sig1`. If `clk0` and `clk1` are not identical, then the clocking event for the sequence changes after ##1. If `clk0` and `clk1` are identical, then the clocking event does not change after ##1, and the preceding sequence is equivalent to the singly clocked sequence

```
@(posedge clk0) sig0 ##1 sig1
```

*Example 2:*

```
@(posedge clk0) sig0 ##0 @(posedge clk1) sig1
```

A match of this sequence starts with a match of `sig0` at **`posedge`** `clk0`. Then ##0 moves the time to the nearest possibly overlapping **`posedge`** `clk1`, and the match of the sequence ends at that point with a match of `sig1`: if **`posedge`** `clk0` and **`posedge`** `clk1` happen simultaneously then the time does not move at ##0, otherwise, it behaves as ##1. If `clk0` and `clk1` are not identical, then the clocking event for the sequence changes after ##0. If `clk0` and `clk1` are identical, then the clocking event does not change after ##0, and the preceding sequence is equivalent to the following singly clocked sequence:

```
@(posedge clk0) sig0 ##0 sig1
```

which is equivalent to the following:

```
@(posedge clk0) sig0 && sig1
```

When concatenating differently clocked sequences, the maximal singly clocked subsequences are required to admit only nonempty matches. The term *maximal singly clocked subsequence* refers to the largest singly clocked sequence appearing in a multiclock sequence resulting from the application of the rewriting algorithm in F.4.1. Such a sequence cannot be enlarged by absorbing any surrounding operators and their arguments without changing the singly clocked sequence into a multiclock sequence or to a property.

Thus, if `s1`, `s2` are sequence expressions with no clocking events, then the multiclocked sequence

```
@(posedge clk1) s1 ##1 @(posedge clk2) s2
```

is legal only if neither `s1` nor `s2` can match the empty word. The clocking event `@(`**`posedge`**` clk1)` applies throughout the match of `s1`, while the clocking event `@(`**`posedge`**` clk2)` applies throughout the match of `s2`. Because the match of `s1` is nonempty, there is an end point of this match at **`posedge`** `clk1`. The ##1 synchronizes between this end point and the first occurrence of **`posedge`** `clk2` strictly after it. That occurrence of **`posedge`** `clk2` is the start point of the match of `s2`.

A multiclocked sequence has well-defined starting and ending clocking events and well-defined clock changes because of the restriction that maximal singly clocked subsequences not match the empty word. If `clk1` and `clk2` are not identical, then the sequence

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1[*0:1]
```

is illegal because of the possibility of an empty match of `sig1[*0:1]`, which would make ambiguous whether the ending clocking event is `@(`**`posedge`**` clk0)` or `@(`**`posedge`**` clk1)`.

Differently clocked or multiclocked sequence operands cannot be combined with any sequence operators other than ##1 and ##0. For example, if `clk1` and `clk2` are not identical, then the following are illegal:

```
@(posedge clk1) s1 ##2 @(posedge clk2) s2

@(posedge clk1) s1 intersect @(posedge clk2) s2
```

### 16.13.2 Multiclocked properties

A clock may be explicitly specified with any property. The property is multiclocked if some of its subproperties have a clock different from the property clock, or some of its subproperties are multiclocked sequences.

As in the case of singly clocked properties, the result of evaluating a multiclocked property is either true or false. Multiclocked sequences are themselves multiclocked properties. For example:

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

is a multiclocked property. If a multiclocked sequence is evaluated as a property starting at some point, the evaluation returns true if, and only if, there is a match of the multiclocked sequence beginning at that point.

The following example shows how to form a multiclocked property using Boolean property operators:

```
(@(posedge clk0) sig0) and (@(posedge clk1) sig1)
```

This is a multiclocked property, but it is not a multiclocked sequence. This property evaluates to true at a point if, and only if, the two sequences

```
@(posedge clk0) sig0
```

and

```
@(posedge clk1) sig1
```

both have matches beginning at the point.

The meaning of multiclocked nonoverlapping implication is similar to that of singly clocked nonoverlapping implication. For example, if `s0` and `s1` are sequences with no clocking event, then in

```
@(posedge clk0) s0 |=> @(posedge clk1) s1
```

`|=>` synchronizes between **`posedge`** `clk0` and **`posedge`** `clk1`. Starting at the point at which the implication is being evaluated, for each match of `s0` clocked by `clk0`, time is advanced from the end point of the match to the nearest strictly future occurrence of **`posedge`** `clk1`, and from that point there must exist a match of `s1` clocked by `clk1`.

The following example shows a combination of differently clocked properties using both implication and Boolean property operators:

```
@(posedge clk0) s0 |=> (@(posedge clk1) s1) and (@(posedge clk2) s2)
```

The multiclocked overlapping implication `|->` has the following meaning: at the end of the antecedent the nearest tick of the consequent clock is awaited. If the consequent clock happens at the end of the antecedent, the consequent is started checking immediately. Otherwise, the meaning of the multiclocked overlapping implication is the same as the meaning of the multiclock nonoverlapping implication.

For example, if `s0` and `s1` are sequences with no clocking events, then

```
@(posedge clk0) s0 |-> @(posedge clk1) s1
```

means the following: at each match of `s0` the nearest **`posedge`** `clk1` is awaited. If it happens immediately then `s1` is checked without delay, otherwise its check starts at the next **`posedge`** `clk1` as in case with `|=>`. In both cases the evaluation of `s1` is controlled by **`posedge`** `clk1`.

The semantics of multiclocked **`if`**/**`if-else`** operators is similar to the semantics of the overlapping implication. For example, if `s1` and `s2` are sequences with no clocking events, then

```
@(posedge clk0) if (b) @(posedge clk1) s1 else @(posedge clk2) s2
```

has the following meaning: the condition `b` is checked at **`posedge`** `clk0`. If `b` is true then `s1` is checked at the nearest, possibly overlapping **`posedge`** `clk1`, else `s2` is checked at the nearest non-strictly subsequent **`posedge`** `clk2`.

### 16.13.3 Clock flow

Throughout this subclause, *c* and *d* denote clocking event expressions and *v*, *w*, *x*, *y*, and *z* denote sequences with no clocking events.

Clock flow allows the scope of a clocking event to extend in a natural way through various parts of multiclocked sequences and properties and reduces the number of places at which the same clocking event must be specified.

Intuitively, clock flow provides that in a multiclocked sequence or property, the scope of a clocking event flows left to right across linear operators (e.g., repetition, concatenation, negation, implication, followed-by, and the **`nexttime`**, **`always`**, **`eventually`** operators) and distributes to the operands of branching operators (e.g., conjunction, disjunction, intersection, **`if-else`**, and the until operators) until it is replaced by a new clocking event.

For example:

```
@(c) x |=> @(c) y ##1 @(d) z
```

can be written more simply as

```
@(c) x |=> y ##1 @(d) z
```

because clock *c* is understood to flow across `|=>`.

Clock flow also makes the adjointness relationships between concatenation and implication clean for multiclocked properties:

```
@(c) x ##1 y |=> @(d) z
```

is equivalent to

```
@(c) x |=> y |=> @(d) z
```

and

```
@(c) x ##0 y |=> @(d) z
```

is equivalent to

```
@(c) x |-> y |=> @(d) z
```

The scope of a clocking event flows into parenthesized subexpressions and, if the subexpression is a sequence, also flows left to right across the parenthesized subexpression. However, the scope of a clocking event does not flow out of enclosing parentheses.

For example, in the following:

```
@(c) w ##1 (x ##1 @(d) y) |=> z
```

*w*, *x*, and *z* are clocked at *c*, and *y* is clocked at *d*. Clock *c* flows across `##1`, across the parenthesized subsequence `(x ##1 @(d) y)`, and across `|=>`. Clock *c* also flows into the parenthesized subsequence, but it does not flow through `@(d)`. Clock *d* does not flow out of its enclosing parentheses.

As another example, in the following:

```
@(c) v |=> (w ##1 @(d) x) and (y ##1 z)
```

*v*, *w*, *y*, and *z* are clocked at *c*, and *x* is clocked at *d*. Clock *c* flows across `|=>`, distributes to both operands of the **and** (which is a property conjunction due to the multiple clocking), and flows into each of the parenthesized subexpressions. Within `(w ##1 @(d) x)`, *c* flows across `##1` but does not flow through `@(d)`. Clock *d* does not flow out of its enclosing parentheses. Within `(y ##1 z)`, *c* flows across `##1`.

Similarly, the scope of a clocking event flows into an instance of a named property or sequence, regardless of whether method `triggered` or method `matched` is applied to the instance of the sequence. The scope of a clocking event flows left to right across an instance of a property or sequence. A clocking event in the declaration of a property or sequence does not flow out of an instance of that property or sequence.

The scope of a clocking event does not flow into the disable condition of **`disable iff`**.

Juxtaposing two clocking events nullifies the first of them; therefore, the following two-clocking-event statement:

```
@(d) @(c) x
```

is equivalent to the following:

```
@(c) x
```

because the flow of clock *d* is immediately overridden by clock *c*.

### 16.13.4 Examples

The following are examples of multiclock specifications:

```
sequence s1;
   a ##1 b; // unclocked sequence
endsequence
sequence s2;
   c ##1 d; // unclocked sequence
endsequence
```

a) Multiclock sequence

```
sequence mult_s;
   @(posedge clk) a ##1 @(posedge clk1) s1 ##1 @(posedge clk2) s2;
endsequence
```

b) Property with a multiclock sequence

```
property mult_p1;
   @(posedge clk) a ##1 @(posedge clk1) s1 ##1 @(posedge clk2) s2;
endproperty
```

c) Property with a named multiclock sequence

```
property mult_p2;
   mult_s;
endproperty
```

d) Property with multiclock implication

```
property mult_p3;
   @(posedge clk) a ##1 @(posedge clk1) s1 |=> @(posedge clk2) s2;
endproperty
```

e) Property with implication, where antecedent and consequent are named multiclocked sequences

```
property mult_p6;
   mult_s |=> mult_s;
endproperty
```

f) Property using clock flow and overlapped implication

```
property mult_p7;
   @(posedge clk) a ##1 b |-> c ##1 @(posedge clk1) d;
endproperty
```

Here, `a`, `b`, and `c` are clocked at **posedge** `clk`.

g) Property using clock flow and **if-else**

```
property mult_p8;
   @(posedge clk) a ##1 b |->
   if (c)
      (1 |=> @(posedge clk1) d)
   else
      e ##1 @(posedge clk2) f ;
endproperty
```

Here, `a`, `b`, `c`, `e`, and constant `1` are clocked at **posedge** `clk`.

### 16.13.5 Detecting and using end point of a sequence in multiclock context

Method `triggered` can be applied to detect the end point of a multiclocked sequence. Method `triggered` can also be applied to detect the end point of a sequence from within a multiclocked sequence. In both cases, the ending clock of the sequence instance to which `triggered` is applied shall be the same as the clock in the context where the application of method `triggered` appears.

To detect the end point of a sequence when the clock of the source sequence is different from the destination sequence, method `matched` on the source sequence is used. The end point of a sequence is reached whenever there is a match on its expression.

To detect the end point, the `matched` method may be applied to a named sequence instance, with or without arguments, an untyped formal argument, or a formal argument of type **sequence**, where such is allowed, as follows:

```
sequence_instance.matched
```

or

```
formal_argument_sequence.matched
```

`matched` is a method on a sequence that returns true (`1'b1`) or false (`1'b0`) . Unlike `triggered`, `matched` uses synchronization between the two clocks, by storing the result of the source sequence match until the arrival of the first destination clock tick after the match. The result of `matched` does not depend upon the starting point of the source sequence.

Like `triggered`, `matched` can be used on sequences that have formal arguments. An example is shown as follows:

```
sequence e1(a,b,c);
   @(posedge clk) $rose(a) ##1 b ##1 c ;
endsequence
sequence e2;
   @(posedge sysclk) reset ##1 inst ##1 e1(ready,proc1,proc2).matched [->1]
      ##1 branch_back;
endsequence
```

In this example, source sequence `e1` is evaluated at clock `clk`, while the destination sequence `e2` is evaluated at clock `sysclk`. In `e2`, the end point of the instance `e1(ready,proc1,proc2)` is tested to occur sometime after the occurrence of `inst`. Notice that method `matched` only tests for the end point of `e1(ready,proc1,proc2)` and has no bearing on the starting point of `e1(ready,proc1,proc2)`.

Local variables can be passed into an instance of a named sequence to which `matched` is applied. The same restrictions apply as in the case of `triggered`. Values of local variables sampled in an instance of a named sequence to which `matched` is applied will flow out under the same conditions as for `triggered`. See 16.10.

As with `triggered`, a sequence instance to which `matched` is applied can have multiple matches in a single cycle of the destination sequence clock. The multiple matches are treated semantically the same way as matching both disjuncts of an **`or`**. In other words, the thread evaluating the destination sequence will fork to account for such distinct local variable valuations.

### 16.13.6 Sequence methods

Methods `triggered` and `matched` are available to identify the end point of a sequence. The operand sequence shall be a named sequence instance, with or without arguments, an untyped formal argument, or a formal argument of type **`sequence`**, in the contexts where such arguments are legal. These methods are invoked using the following syntax:

```
sequence_instance.sequence_method
```

or

```
formal_argument_sequence.sequence_method
```

The results of these operations are true (`1'b1`) or false (`1'b0`) and do not depend upon the starting point of the match of their operand sequence. These methods can be invoked on sequences with formal arguments. The sampled values of these methods are defined as the current values (see 16.5.1).

The value of method `triggered` evaluates to true (`1'b1`) if the operand sequence has reached its end point at that particular point in time and false (`1'b0`) otherwise. The triggered status of the sequence is set in the Observed region and persists through the remainder of the time step. In addition to using this method in assertion statements, it may be used in **`wait`** statements (see 9.4.4) or Boolean expressions outside a sequence context. It shall be considered an error to invoke this method outside a sequence context on sequences that treat their formal arguments as local variables. A sequence treats its formal argument as a local variable if the formal argument is used as an lvalue in *operator_assignment* or *inc_or_dec_expression* in *sequence_match_item*. There shall be no circular dependencies between sequences induced by the use of `triggered`.

The method `matched` is used to detect the end point of one sequence (the source sequence) referenced in a multiclocked sequence (the destination sequence). It can only be used in sequence expressions. Unlike `triggered`, `matched` provides synchronization between two clocks by storing the result of the source sequence until the arrival of the first clock tick of the destination sequence after the match. The matched status of the sequence is set in the Observed region and persists until the Observed region following the arrival of the first clock tick of the destination sequence after the match.

It shall be considered an error to use the sequence method `matched` in sampled value functions (see 16.9.3).

An example of using the previous methods on a sequence is shown as follows:

```
sequence e1;
   @(posedge sysclk) $rose(a) ##1 b ##1 c;
endsequence

sequence e2;
   @(posedge sysclk) reset ##1 inst ##1 e1.triggered ##1 branch_back;
endsequence

sequence e3;
   @(posedge clk) reset1 ##1 e1.matched ##1 branch_back1;
endsequence

sequence e2_with_arg(sequence subseq);
   @(posedge sysclk) reset ##1 inst ##1 subseq.triggered ##1 branch_back;
endsequence

sequence e4;
   e2_with_arg(@(posedge sysclk) $rose(a) ##1 b ##1 c);
endsequence

program check;
   initial begin
      wait (e1.triggered || e2.triggered);
      if (e1.triggered)
         $display("e1 passed");
      if (e2.triggered)
         $display("e2 passed");
      L2: ...
   end
endprogram
```

In the preceding example, sequence `e2` tests for the end point of sequence `e1` using method `triggered` because both sequences use the same clock. The sequence `e3` tests for the end point of sequence `e1` using method `matched` because `e1` and `e3` use different clocks. The sequence `e4` is semantically equivalent to `e2` and shows an application of the sequence method `triggered` on a formal argument of type **`sequence`**. The **`initial`** procedure in the program waits for the end point of either `e1` or `e2`. When either `e1` or `e2` evaluates to true, the wait statement unblocks the initial process. The process then displays the sequence that caused it to unblock, and then continues to execute at the statement labeled `L2`.

The sequence on which a method is applied shall either be clocked or infer the clock from the context where it is used. The same rules are used to infer the clocking event as specified in 16.9.3 for sampled value functions.

If the default actual argument `$inferred_clock` is specified for a formal argument of a sequence, and an actual argument is not provided to the sequence instance to which a method is applied, the same rules as specified in 16.9.3 for sampled value functions are used to determine the inferred clocking event expression that is bound to that formal argument.

If a sequence with a method is passed as an actual argument to a checker instantiation, it is substituted in place of the corresponding formal argument. Such a sequence shall be clocked as if it were instantiated inside the checker.

If a sequence with a method is connected to a port of a module instantiation, it shall be clocked as if it were instantiated at the place of module instantiation. The same rule shall apply if a sequence with a method is connected to a port of an interface or program instantiation or passed as an actual argument to a function or task call.

The preceding rules for inferring the clocking event also apply to a sequence instantiated in an event expression.

The following examples illustrate how a clock is inferred by a sequence when a method is applied to it.

```
module mod_sva_checks;
   logic a, b, c, d;
   logic clk_a, clk_d, clk_e1, clk_e2;
   logic clk_c, clk_p;

   clocking cb_prog @(posedge clk_p); endclocking
   clocking cb_checker @(posedge clk_c); endclocking

   default clocking cb @(posedge clk_d); endclocking

   sequence e4;
      $rose(b) ##1 c;
   endsequence

   // e4 infers posedge clk_a as per clock flow rules
   a1: assert property (@(posedge clk_a) a |=> e4.triggered);

   sequence e5;
      // e4 will infer posedge clk_e1 as per clock flow rules
      // wherever e5 is instantiated (with/without a method)
      @(posedge clk_e1) a ##[1:3] e4.triggered ##1 c;
   endsequence

   // e4, used in e5, infers posedge clk_e1 from e5
   a2: assert property (@(posedge clk_a) a |=> e5.matched);

   sequence e6(f);
      @(posedge clk_e2) f;
   endsequence

   // e4 infers posedge clk_e2 as per clock flow rules
   a3: assert property (@(posedge clk_a) a |=> e6(e4.triggered));

   sequence e7;
      e4 ##1 e6(d);
   endsequence

   // Leading clock of e7 is posedge clk_a as per clock flow rules
   a4: assert property (@(posedge clk_a) a |=> e7.triggered);

   // Illegal use in a disable condition, e4 is not explicitly clocked
   a5_illegal: assert property (
      @(posedge clk_a) disable iff (e4.triggered) a |=> b);
   always @(posedge clk_a) begin
      // e4 infers default clocking cb and not posedge clk_a as there is
      // more than one event control in this procedure (16.14.6)
      @(e4);
      d = a;
   end

   program prog_e4;
      default clocking cb_prog;
      initial begin
         // e4 infers default clocking cb_prog
         wait (e4.triggered);
         $display("e4 passed");
      end
   endprogram : prog_e4

   checker check(input in1, input sequence s_f);
      default clocking cb_checker;
      always @(s_f)
         $display("sequence triggered");
      a4: assert property (a |=> in1);
   endchecker : check

   // e4 infers checker's default clocking cb_checker
   check c1(e4.triggered, e4);

   // e4 connected to port of a module instance infers default clocking cb
   mod_adder ai1(e4.triggered);

endmodule : mod_sva_checks
```

If a sequence admits an empty match, such empty matches shall not activate the `.triggered` or `.matched` methods (see 16.9.11).

More details about sequence methods can be found in 9.4.4, 16.9.11, and 16.13.5.

### 16.13.7 Local variable initialization assignments

For singly clocked sequences and properties, a local variable initialization assignment for an evaluation attempt of an instance of a named sequence or property is performed when the evaluation attempt begins. Such an evaluation attempt always begins in a time step in which there is a tick of the single governing clock.

For multiclock sequences and properties, a local variable initialization assignment for an evaluation attempt of an instance of a named sequence or property with a single semantic leading clock (see 16.16.1) shall be performed at the earliest tick of the semantic leading clock that is at or after the beginning of the evaluation attempt. If there are two or more distinct semantic leading clocks for an instance of a named property, then a separate copy of the local variable shall be created for each semantic leading clock. For each copy of the local variable, the initialization assignment shall be performed at the earliest tick of the corresponding semantic leading clock that is at or after the beginning of the evaluation attempt, and that copy of the local variable shall be used in the evaluation of the subproperty associated with the corresponding semantic leading clock.

For example, let

```
property p;
   logic v = e;
   (@(posedge clk1) (a == v)[*1:$] |-> b)
   and
   (@(posedge clk2) c[*1:$] |-> d == v)
   ;
endproperty
a1: assert property (@(posedge clk) f |=> p);
```

where `f` is of type **`logic`**. The instance of `p` in assertion `a1` has two semantic leading clocks, **`posedge`** `clk1` and **`posedge`** `clk2`. Separate copies of the local variable `v` are created for the two subproperties governed by these clocks. Let `t0` be a time step in which **`posedge`** `clk` occurs and in which the sampled value of `f` is true. According to the structure of `a1`, an evaluation attempt of the instance of `p` starts strictly after `t0`. Let `t1` be the earliest time step after `t0` in which **`posedge`** `clk1` occurs, and let `t2` be the earliest time step after `t0` in which **`posedge`** `clk2` occurs. Then a declaration assignment `v = e` is performed in `t1`, and the value is assigned to the copy of `v` associated with **`posedge`** `clk1`. This value is used in the evaluation of the subproperty `(a == v)[*1:$] |-> b`. Similarly, a declaration assignment `v = e` is performed in `t2`, and the value is assigned to the copy of `v` associated with **`posedge`** `clk2`. This value is used in the evaluation of the subproperty `c[*1:$] |-> d == v`.

An equivalent declaration of `p` that does not use local variable declaration assignments is as follows:

```
property p;
   logic v;
   (@(posedge clk1) (1, v = e) ##0 (a == v)[*1:$] |-> b)
   and
   (@(posedge clk2) (1, v = e) ##0 c[*1:$] |-> d == v)
   ;
endproperty
```

## 16.14 Concurrent assertions

A property on its own is never evaluated for checking an expression. It shall be used within an assertion statement (see 16.2) for this to occur.

A concurrent assertion statement may be specified in any of the following:

— An always procedure or initial procedure as a statement, wherever these procedures may appear (see 9.2)
— A module
— An interface
— A program
— A generate block
— A checker

---

concurrent_assertion_item ::= *// from A.2.10*
    [ block_identifier **:** ] concurrent_assertion_statement
    ...

procedural_assertion_statement ::= *// from A.6.10*
    concurrent_assertion_statement
  ...

concurrent_assertion_statement ::=
    assert_property_statement

| assume_property_statement
| cover_property_statement
| cover_sequence_statement
| restrict_property_statement

assert_property_statement::=
**assert property (** property_spec **)** action_block

assume_property_statement::=
**assume property (** property_spec **)** action_block

cover_property_statement::=
**cover property (** property_spec **)** statement_or_null

cover_sequence_statement::=
**cover sequence (** [clocking_event ] [ **disable iff (** expression_or_dist **)** ]
sequence_expr **)** statement_or_null

restrict_property_statement::=
**restrict property (** property_spec **) ;**

*Syntax 16-20—Concurrent assert construct syntax (excerpt from Annex A)*

The execution of assertion statements can be controlled using assertion control system tasks (see 20.12).

A concurrent assertion statement can be referenced by its optional name. A hierarchical name can be used consistent with the SystemVerilog naming conventions. When a name is not provided, a tool shall assign a name to the statement for the purpose of reporting. Unnamed assertions do not create a scope.

### 16.14.1 Assert statement

The **assert** statement is used to enforce a **property**. When the property for the **assert** statement is evaluated to be true, the pass statements of the action block are executed. When the property for the **assert** statement is evaluated to be false, the fail statements of the *action_block* are executed. When the property for the **assert** statement is evaluated to be disabled, no *action_block* statement is executed. The execution of pass and fail statements can be controlled by using assertion action control tasks. The assertion action control tasks are described in 20.12.

For example:

```
property abc(a, b, c);
   disable iff (a==2) @(posedge clk) not (b ##1 c);
endproperty
env_prop: assert property (abc(rst, in1, in2))
            $display("env_prop passed."); else $display("env_prop failed.");
```

When no action is needed, a null statement (i.e., `;` ) is specified. If no statement is specified for **else**, then `$error` is used as the statement when the assertion fails.

The *action_block* shall not include any concurrent **assert**, **assume**, or **cover** statement. The *action_block*, however, can contain immediate assertion statements.

The conventions regarding default severity (error) and the use of severity system tasks in concurrent assertion action blocks shall be the same as those specified for immediate assertions in 16.3.

The pass and fail statements of an assert statement are executed in the Reactive region. The regions of execution are explained in the scheduling semantics in Clause 4.

### 16.14.2 Assume statement

The purpose of the **assume** statement is to allow properties to be considered as assumptions for formal analysis as well as for dynamic simulation tools. When a property is assumed, the tools constrain the environment so that the property holds.

For formal analysis, there is no obligation to verify that the assumed properties hold. An assumed property can be considered as a hypothesis to prove the asserted properties.

For simulation, the environment must be constrained so that the properties that are assumed shall hold. Like an asserted property, an assumed property must be checked and reported if it fails to hold. When the property for the **assume** statement is evaluated to be true, the pass statements of the *action_block* are executed. If it evaluates to false, the fail statements of the *action_block* are executed. For example:

```
property abc(a, b, c);
   disable iff (c) @(posedge clk) a |=> b;
endproperty
env_prop:
   assume property (abc(req, gnt, rst)) else $error(”Assumption failed.”);
```

If the property has a disabled evaluation, neither the pass nor fail statements of the *action_block* are executed. The execution of pass and fail statements can be controlled by using assertion action control tasks. The assertion action control tasks are described in 20.12.

Additionally, for random simulation, biasing on the inputs provides a way to make random choices. An expression can be associated with biasing as follows:

```
expression dist { dist_list } ;  // from A.2.10
```

Distribution sets and the **dist** operator are explained in 18.5.4.

The biasing feature is useful when properties are considered as assumptions to drive random simulation. When a property with biasing is used within an **assert** or **cover** assertion statement, the **dist** operator is equivalent to **inside** operator, and the weight specification is ignored. For example:

```
a1:assume property ( @(posedge clk) req dist {0:=40, 1:=60} ) ;
property proto ;
   @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty
```

This is equivalent to the following:

```
a1_assertion:assert property ( @(posedge clk) req inside {0, 1} ) ;
property proto_assertion ;
   @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty
```

In the preceding example, signal `req` is specified with distribution in assumption `a1` and is converted to an equivalent assertion `a1_assertion`.

It should be noted that the properties that are assumed must hold in the same way with or without biasing. When using an assume statement for random simulation, the biasing simply provides a means to select values of free variables, according to the specified weights, when there is a choice of selection at a particular time.

Consider an example specifying a simple synchronous request and acknowledge protocol, where variable `req` can be raised at any time and must stay asserted until `ack` is asserted. In the next clock cycle, both `req` and `ack` must be deasserted.

Properties governing `req` are as follows:

```
property pr1;
   @(posedge clk) !reset_n |-> !req;          // when reset_n is asserted (0),
                                              // keep req 0
endproperty
property pr2;
   @(posedge clk) ack |=> !req;               // one cycle after ack, req
                                              // must be deasserted
endproperty
property pr3;
   @(posedge clk) req |-> req[*1:$] ##0 ack; // hold req asserted until
                                              // and including ack asserted
endproperty
```

Properties governing `ack` are as follows:

```
property pa1;
   @(posedge clk) !reset_n || !req |-> !ack;
endproperty
property pa2;
   @(posedge clk) ack |=> !ack;
endproperty
```

When verifying the behavior of a protocol controller that has to respond to requests on `req`, assertions `assert_ack1` and `assert_ack2` should be proven while assuming that statements `a1`, `assume_req1`, `assume_req2`, and `assume_req3` hold at all times.

```
a1:assume property (@(posedge clk) req dist {0:=40, 1:=60} );
assume_req1:assume property (pr1);
assume_req2:assume property (pr2);
assume_req3:assume property (pr3);

assert_ack1:assert property (pa1)
   else $error("ack asserted while req is still deasserted");
assert_ack2:assert property (pa2)
   else $error("ack is extended over more than one cycle");
```

### 16.14.3 Cover statement

There exist two categories of cover statements: **cover sequence** and **cover property**. The **cover sequence** statement specifies sequence coverage, while the **cover property** statement specifies property coverage. Both monitor behavioral aspects of the design for coverage. Tools shall collect coverage information and report the results at the end of simulation or on demand via an assertion API (refer to Clause 39). The difference between the two categories is that for sequence coverage, all matches per evaluation attempt are reported, whereas for property coverage the coverage count is incremented at most once per evaluation attempt. A cover statement may have an optional pass statement. The pass statement shall not include any concurrent **assert**, **assume**, or **cover** statement.

For property coverage, the statement appears as follows:

```
cover property ( property_spec ) statement_or_null
```

The results of this coverage statement for a property shall contain the following:

— Number of times attempted
— Number of times succeeded (maximum of one per attempt)
— Number of times succeeded because of vacuity

The pass statement specified in *statement_or_null* shall be executed once for each successful evaluation attempt of the underlying *property_spec*. The pass statement shall be executed in the Reactive region of the time step in which the corresponding evaluation attempt succeeds. The execution of *statement_or_null* can be controlled by using assertion action control tasks. The assertion action control tasks are described in 20.12.

The preceding coverage counters for success or vacuous success do not include disabled evaluations. The attempt counter includes the attempts that result in disabled evaluation. See 40.5.3 for details on obtaining assertion coverage results.

For sequence coverage, the statement appears as follows:

```
cover sequence (
   [clocking_event ] [ disable iff ( expression_or_dist ) ] sequence_expr )
      statement_or_null
```

Results of coverage for a sequence shall include the following:

— Number of times attempted
— Number of times matched (each attempt can generate multiple matches)

For a given attempt of the **cover sequence** statement, all matches of the *sequence_expr* that complete without the occurrence of the **disable iff** condition shall be counted, with multiplicity, toward the total number of times matched for the attempt. No other match shall be counted towards the total for the attempt. The pass statement specified in *statement_or_null* shall be executed, with multiplicity, for each match that is counted toward the total for the attempt. The pass statement shall be executed in the Reactive region of the time step in which the corresponding match completes. The execution of *statement_or_null* can be controlled by using assertion action control tasks. The assertion action control tasks are described in 20.12.

For a given attempt of the **cover sequence** statement, the total number of times matched for the attempt is equal to the number of times `increment_match_coverage()` is executed in the corresponding attempt of

```
assert property (
   [clocking_event] [ disable iff ( expression_or_dist ) ]
      sequence_expr |-> ( 1'b1, increment_match_coverage() ) );
```

For each execution of `increment_match_coverage()`, the pass statement of the cover sequence statement is executed in the Reactive region of the same time step.

### 16.14.4 Restrict statement

In formal verification, for the tool to converge on a proof of a property or to initialize the design to a specific state, it is often necessary to constrain the state space. For this purpose, the assertion statement **restrict property** is introduced. It has the same semantics as **assume property**, however, in contrast to that statement, the **restrict property** statement is not verified in simulation and has no action block.

The statement has the following form:

```
restrict property ( property_spec ) ;
```

There is no action block associated with the statement.

*Example:*

Suppose that when a control bit `ctr` has a value 0, an ALU performs an addition, and when it is 1, it performs a subtraction. It is required to formally verify that some behavior is correct when ALU does an addition (in another verification session it is possible to do the same for subtraction by changing the restriction). The behavior can thus be constrained using the statement:

```
restrict property (@(posedge clk) ctr == '0);
```

It does not mean that `ctr` cannot be 1 in any test case in the simulation; that is not an error.

### 16.14.5 Using concurrent assertion statements outside procedural code

A concurrent assertion statement can be used outside a procedural context. It can be used within a module, an interface, or a program. A concurrent assertion statement is an **assert**, an **assume**, a **cover**, or a **restrict** statement. Such a concurrent assertion statement uses the **always** semantics, meaning that it specifies that a new evaluation attempt of the underlying *property_spec* begins at every occurrence of its leading clock event.

The following two forms are equivalent:

```
assert property ( property_spec ) action_block

always assert property ( property_spec ) action_block ;
```

Similarly, the following two forms are equivalent:

```
cover property ( property_spec ) statement_or_null

always cover property ( property_spec ) statement_or_null
```

For example:

```
module top(input logic clk);
   logic a,b,c;
   property rule3;
      @(posedge clk) a |-> b ##1 c;
   endproperty
   a1: assert property (rule3);
   ...
endmodule
```

`rule3` is a property declared in module `top`. The assert statement `a1` starts checking the property from the beginning to the end of simulation. The property is always checked. Similarly,

```
module top(input logic clk);
   logic a,b,c;
   sequence seq3;
      @(posedge clk) b ##1 c;
   endsequence
   c1: cover property (seq3);
   ...
endmodule
```

The cover statement `c1` starts coverage of the sequence `seq3` from beginning to the end of simulation. The sequence is always monitored for coverage.

### 16.14.6 Embedding concurrent assertions in procedural code

A concurrent assertion statement can also be embedded in a procedural block. For example:

```
property rule;
   a ##1 b ##1 c;
endproperty

always @(posedge clk) begin
   <statements>
   assert property (rule);
end
```

The term *procedural concurrent assertion* is used to refer to any concurrent assertion statement (see 16.2) that appears in procedural code. Unlike an immediate assertion, a procedural concurrent assertion is not immediately evaluated when reached in procedural code. Instead, the assertion and the current values of all constant and automatic expressions appearing in its assertion arguments (see 16.14.6.1) are placed in a *procedural assertion queue* associated with the currently executing process. Each of the entries in this queue is said to be a *pending procedural assertion instance*. Since any given statement in a procedure may be executed multiple times (as in a loop for example), a particular procedural concurrent assertion may result in many pending procedural assertion instances within a single time step. A concurrent assertion statement that appears outside procedural code is referred to as a *static concurrent assertion statement*.

In the Observed region of each simulation time step, each pending procedural assertion instance that is currently present in a procedural assertion queue shall *mature*, which means it is confirmed for execution. When a pending procedural assertion instance matures, if the current time step is one that corresponds to that assertion instance's leading clocking event, an evaluation attempt of the assertion begins immediately within the Observed region. If the assertion's leading clocking event has not occurred in this time step, the matured instance shall be placed on the *matured assertion queue*, which will cause the assertion to begin an evaluation attempt upon the next clocking event that corresponds to the leading clocking event of the assertion.

If a *procedural assertion flush point* (see 16.14.6.2) is reached in a process, its procedural assertion queue is cleared. Any currently pending procedural assertion instances will not mature, unless again placed on the queue in the course of procedural execution.

If no clocking event is specified in a procedural concurrent assertion, the leading clocking event of the assertion shall be inferred from the procedural context, if possible. If no clock can be inferred from the procedural context, then the clocks shall be inferred from the default clocking (14.12), as if the assertion were instantiated immediately before the procedure.

A clock shall be inferred for the context of an always or initial procedure that satisfies the following requirements:

a) There is no blocking timing control in the procedure.
b) There is exactly one event control in the procedure.
c) One and only one event expression within the event control of the procedure satisfies both of the following conditions:
   1) The event expression consists solely of an event variable, solely of a clocking block event, or is of the form *edge_identifier expression1* [ **`iff`** *expression2* ] and is not a proper subexpression of an event expression of this form.
   2) If the event expression consists solely of an event variable or clocking block event, it does not appear anywhere else in the body of the procedure other than as a clocking event or within

assertion statements. If the event expression is of the form *edge_identifier expression1* [ **iff** *expression2* ], no term in expression1 appears anywhere else in the body of the procedure other than as a clocking event or within assertion statements.

If these requirements are satisfied, then the unique event expression from the third requirement shall be the clock inferred for the context of the procedure.

For example, in the following code fragment, the clocking event `@(posedge mclk)` is inferred as the clocking event of `r1_p1`, while `r1_p2` is clocked by `@(posedge scanclk)` as written:

```
property r1;
   q != d;
endproperty
always @(posedge mclk) begin
   q <= d1;
   r1_p1: assert property (r1);
   r1_p2: assert property (@(posedge scanclk)r1);
end
```

The resulting behavior of the preceding assertion `r1_p2` depends on the relative frequencies of `mclk` and `scanclk`. For example:

— If `scanclk` runs at twice the frequency of `mclk`, only every other posedge of `scanclk` will result in an evaluation of `r1_p2`. It is only queued when reached during procedural execution, which happens on a rising edge of `mclk`.
— If `mclk` runs at twice the frequency of `scanclk`, then by every posedge of `scanclk`, two pending procedural instances of `r1_p2` will mature. Thus every posedge of `scanclk` will see `r1_p2` evaluated and results reported twice.

Also see 17.4 for the context clock inference in checkers, and 17.5 for examples of clock inference in checker procedures.

Another, more complex example that is legal is as follows:

```
property r2;
   q != d;
endproperty

always_ff @(posedge clock iff reset == 0 or posedge reset) begin
   cnt <= reset ? 0 : cnt + 1;
   q <= $past(d1);
   r2_p: assert property (r2);
end
```

In the preceding example, the inferred clock is `posedge clock iff reset == 0`. The inferred clock is not `posedge clock` because `posedge clock` is a proper subexpression of `posedge clock iff reset == 0`.

In contrast, no clock is inferred for the context of the `always_ff` in the following:

```
property r3;
   q != d;
endproperty

always_ff @(clock iff reset == 0 or posedge reset) begin
   cnt <= reset ? 0 : cnt + 1;
```

```
      q <= $past(d1);               // no inferred clock
      r3_p: assert property (r3);   // no inferred clock
   end
```

The edge expression **posedge** `reset` cannot be inferred because `reset` is referenced within the procedure, and the expression `clock` **iff** `reset == 0` cannot be inferred because it does not have an edge identifier. In the absence of default clocking, the code above results in an error.

In the following example, no clock is inferred due to multiple event controls and delays in the always procedure.

```
   property r4;
      q != d;
   endproperty

   always @(posedge mclk) begin
      #10 q <= d1;                 // delay prevents clock inference
      @(negedge mclk)              // event control prevents clock inference
      #10 q1 <= !d1;
      r4_p: assert property (r4);  // no inferred clock
   end
```

### 16.14.6.1 Arguments to procedural concurrent assertions

A procedural concurrent assertion saves the value of its **const** expressions and automatic variables at the time the assertion evaluation attempt is added to the procedural assertion queue. This assertion evaluation attempt uses these saved values for the evaluation. For example:

```
   // Assume for this example that (posedge clk) will not occur at time 0
   always @(posedge clk) begin
      int i = 10;
      for (i=0; i<10; i++) begin
         a1: assert property (foo[i] && bar[i]);
         a2: assert property (foo[const'(i)] && bar[i]);
         a3: assert property (foo[const'(i)] && bar[const'(i)]);
      end
   end
```

In any given clock cycle, each of these assertions will result in 10 queued executions. Every execution of assertion `a1`, however, will be checking the value of `(foo[10] && bar[10])`, since the sampled value of `i` will always be 10, its final value from the previous execution of the procedure. In the case of `a2`, its executions will be checking `(foo[0] && bar[10])`, `(foo[1] && bar[10])`, ... `(foo[9] && bar[10])`. Assertion `a3`, since it has **const** casts on both uses of `i`, will be checking `(foo[0] && bar[0])`, `(foo[1] && bar[1])`, ... `(foo[9] && bar[9])`. So the preceding code fragment is logically equivalent (aside from instance names) to the following:

```
   default clocking @(posedge clk); endclocking
   generate for (genvar i=0; i<10; i++) begin
      a1: assert property (foo[10] && bar[10]);
      a2: assert property (foo[i] && bar[10]);
      a3: assert property (foo[i] && bar[i]);
   end
   endgenerate
```

Since automatic variables also have their immediate values preserved, in the following example, all three properties `a4`, `a5`, and `a6` are logically equivalent:

```
always @(posedge clk) begin
   // variable declared in for statement is automatic (see 12.7.1)
   for (int i=0; i<10; i++) begin
      a4: assert property (foo[i] && bar[i]);
      a5: assert property (foo[const'(i)] && bar[i]);
      a6: assert property (foo[const'(i)] && bar[const'(i)]);
   end
end
```

When a procedural concurrent assertion contains temporal expressions and has matured, the execution flow of the procedure no longer directly affects the matured instance in future time steps. In other words, the procedural execution only affects the activation of the assertion instance, not the completion of temporal expressions in the future. However, any constant values that were passed into the assertion instance due to constant or automatic variables will remain constant for the duration of that instance's evaluation. The following example illustrates this behavior:

```
wire w;
always @(posedge clk) begin : procedural_block_1
   if (my_activation_condition == 1) begin
      for (int i=0; i<2; i++) begin
         a7: assume property (foo[i] |=> bar[i] ##1 (w==1'b1));
      end
   end
end
```

During the time step when `my_activation_condition` is 1, two pending instances of `a7` will be placed on the procedural assertion queue, one for each value of `i`. Assume that they successfully mature, and `foo[0]` is true in the current time step. This means that on the next posedge of `clk`, regardless of the execution of `procedural_block_1` or the value of `my_activation_condition`, that matured instance of `a7` will be checking that `bar[0]` is true. The constant value of the automatic `i` from when the assertion was queued is still in effect, for this and any future clock cycles of this assertion evaluation. Then, one cycle later, the assertion will also be checking that the sampled value of `w` is `1'b1`.

The same rules that apply to procedural concurrent assertion arguments also apply to variables appearing in their action blocks. Thus, constant or automatic values may be used in action blocks as well as the assertion statements themselves, where they behave as inputs to the action block that shall not be modified. The following example illustrates this behavior:

```
// Assume for this example that (posedge clk) will not occur at time 0
always @(posedge clk) begin
   int i = 10;
   for (i=0; i<10; i++) begin
      a8: assert property (foo[const'(i)] && bar[i]) else
         $error("a8 failed for const i=%d and i=%d",
                const'(i), $sampled(i));
   end
end
```

Upon a failure, any instance of the preceding assertion will show the constant value of `i` (may be from 0 to 9) that was used in that instance for "`const i=`", while the string printed will always end in "`i=10`", since 10 will be the sampled value captured from the Preponed region.

When embedding procedural concurrent assertions in code using conditionals, it is important to remember that the current values of the conditionals in the procedure are used, rather than the sampled values. This contrasts with the assertion's expressions, where sampled values are used (see 16.5.1). The following example illustrates this situation:

```
// Assume a, b, c, and en are not automatic
always @(posedge clk) begin
   en = ...;
   if (en) begin
      a9: assert property p1(a,b,c);
   end
   if ($sampled(en)) begin
      a10: assert property p1(a,b,c);
   end
end
```

Assertion `a9` is queued on any time step when `en` becomes true, while `a10` is queued on any time step when the sampled value of `en` was true. Thus, assuming nothing else in the code modifies `en`, checks of `a10` will happen a time step later than checks on `a9`, even though both use the sampled values of `a`, `b`, and `c` on their respective time steps.

NOTE—This is an area of backwards-incompatibility between this standard and 17.13 of IEEE Std 1800-2005. In the 2005 definition, `en` would have been detected as the *inferred enabling condition* (a definition that no longer exists in this standard) of `a9` and always sampled, so `a9` and `a10` would have identical behavior.

### 16.14.6.2 Procedural assertion flush points

A process is defined to have reached a procedural assertion flush point if any of the following occur:

— The process, having been suspended earlier due to reaching an event control or **`wait`** statement, resumes execution.

— The process was declared by an **`always_comb`** or **`always_latch`**, and its execution is resumed due to a transition on one of its dependent signals.

— The outermost scope of the process is disabled by a **`disable`** statement (see 16.14.6.4).

The following example shows how procedural concurrent assertions inherently avoid multiple evaluations due to transitional combinational values in a single simulation time step:

```
assign not_a = !a;
default clocking @(posedge clk); endclocking
always_comb begin : b1
   // Probably better to not use consts in this example
   // ...but using them to illustrate effects of flushing method
   a1: assert property (const'(not_a) != const'(a));
end
```

When `a` changes in a time step during which a positive clock edge occurs, a simulator could evaluate assertion `a1` twice—once for the change in `a` and once for the change in `not_a` after the evaluation of the continuous assignment. The first execution of `a1`, which would have ended up reporting a failure, will be scheduled on the process's procedural assertion queue. When `not_a` changes, the procedural assertion queue is flushed due to the activation of `b1`, and a new pending instance of the procedural concurrent assertion will now be queued with the correct values, so no failure of `a1` will be reported.

The following example illustrates the behavior of procedural concurrent assertions in the presence of time delays:

```
default clocking @(posedge clk); endclocking
always @(a or b) begin : b1
   a2: assert property (a == b) r.success(0) else r.error(0, a, b);
```

```
        #1;
        a3: assert property (a == b) r.success(1) else r.error(1, a, b);
    end
```

In this case, due to the time delay in the middle of the procedure, an Observed region will always be reached after the queueing of a2 and before a flush point. Thus a2 will always mature. For a3, during time steps where either a or b changes after it has been queued, the assertion will always be flushed from the queue and never mature. In general, procedural concurrent assertions must be used carefully when mixed with time delays.

The following example illustrates a typical use of a procedural concurrent assertion statement with a **cover** rather than an **assert**:

```
    assign a = ...;
    assign b = ...;
    default clocking @(posedge clk); endclocking
    always_comb begin : b1
        ...
        c1: cover property (const'(b) != const'(a));
    end
```

In this example, the goal is to make sure some test is covering the case where a and b have different values at that point in the procedural code. Due to the arbitrary order of the assignments in the simulator, it might be the case that in a cycle where there is a positive clock edge and both variables are being assigned the same value, b1 executes while a has been assigned but b still holds its previous value. Thus c1 will be queued, but this is actually a glitch, and probably not a useful piece of coverage information. But, when b1 is executed the next time (after b has also been assigned its new value), that coverage point will be flushed, and when the coverage point matures, c2 will correctly not get reported as having been covered during that time step.

#### 16.14.6.3 Procedural concurrent assertions and glitches

One common concern with assertion execution is glitches, where the same assertion executes multiple times in a time step and reports undesired failures on temporary values that have not yet received their final values for the step. In general, procedural concurrent assertions are immune to glitches due to order of procedural execution due to the flushing mechanism, but are still subject to glitches caused by execution loops between regions.

For example, if code in the Reactive region modifies signals and causes another pass to the Active region to occur, this may create some glitching behavior, as the new passage in the Active region may requeue procedural concurrent assertions, and a second evaluation attempt may be added to the matured assertion queue. The following code illustrates this situation.

```
    always_comb begin : procedural_block_1
        if (en)
            foo = bar;
    end

    always_comb begin : procedural_block_2
        p1: assert property ( @(posedge clk) (const'(foo) == const'(bar)) );
    end
```

Suppose bar is assigned a new value elsewhere in the code at the posedge of the clock, and en is 1 so the assignment in procedural_block_1 takes place. Block procedural_block_2 may be executed twice in the Active region: once upon the initial change to bar, and once after the assignment that updates foo. Upon the first execution of procedural_block_2, a pending instance of p1 will be queued and would result in failure of the assertion if it matured. But this instance will be flushed upon the second execution of the procedural block before maturing, and thus there will be no glitch.

However, now suppose that in the same example, `en` is 0, and the assignment of the `bar` value to `foo` happens through VPI code in the Reactive region. In this case, the Observed region has already occurred, so `p1` has matured and executed, and reported the assertion failure due to `foo` and `bar` having different values. After the Reactive region, there will be another Active region in which `procedural_block_2` will be executed, and this time a newly queued instance of `p1` will pass. But this is too late to prevent the report of the failure earlier in the time step.

### 16.14.6.4 Disabling procedural concurrent assertions

The **`disable`** statement shall interact with procedural concurrent assertions as follows:

- A specific procedural concurrent assertion may be disabled. Any pending procedural instances of that assertion are cleared from the queue. Any pending procedural instances of other assertions remain in the queue.
- When a **`disable`** is applied to the outermost scope of a procedure that has a pending procedural assertion queue, in addition to normal disable activities (see 9.6.2), the pending procedural assertion queue is flushed and all pending assertion instances on the queue are cleared.

Once a procedural concurrent assertion evaluation attempt has matured, it shall not be impacted by any disable.

Disabling a task or a non-outermost scope of a procedure does not cause flushing of any pending procedural assertion instances.

The following example illustrates how user code can explicitly flush a pending procedural assertion instance. In this case, instances of `a1` only mature in time steps where `bad_val_ok` does not settle at a value of 1.

```
default clocking @(posedge clk); endclocking
always @(bad_val or bad_val_ok) begin : b1
   a1: assert property (bad_val) else $fatal(1, "Sorry");
   if (bad_val_ok) begin
      disable a1;
   end
end
```

The following example illustrates how user code can explicitly flush all pending procedural assertion instances on the procedural assertion queue of process `b2`:

```
default clocking @(posedge clk); endclocking
always @(a or b or c) begin : b2
   if (c == 8'hff) begin
      a2: assert property (a && b);
   end else begin
      a3: assert property (a || b);
   end
end

always @(clear_b2) begin : b3
   disable b2;
end
```

### 16.14.7 Inferred value functions

The following elaboration time system functions are available to query the inferred clocking event expression and disable expression:

— `$inferred_clock` returns the expression of the inferred clocking event.

— `$inferred_disable` returns the inferred disable expression.

The inferred clocking event expression is the current resolved event expression that can be used in a clocking event definition. It is obtained by applying clock flow rules to the point where `$inferred_clock` is called. If there is no current resolved event expression when `$inferred_clock` is encountered then an error shall be issued.

The inferred disable expression is the disable condition from the default disable declaration whose scope includes the call to `$inferred_disable` (see 16.15). If the call to `$inferred_disable` is not within the scope of any default disable declaration, then the call to `$inferred_disable` returns `1'b0` (false).

A call to an inferred expression function may only be used as the entire default value expression for a formal argument to a property or sequence declaration. A call to an inferred expression function shall not appear within the body expression of a property or sequence declaration. If a call to an inferred expression function is used as the entire default value expression for a formal argument to a property or sequence declaration, then it is replaced by the inferred expression as determined at the point where the property or sequence is instantiated. Therefore, if the property or sequence instance is the top-level property expression in an assertion statement, the event expression that is used to replace the default argument `$inferred_clock` is that as determined at the location of the assertion statement. If the property or sequence instance is not the top-level property expression in the assertion statement, then the event expression determined by clock flow rules up to the instance location in the property expression is used as the default value of the argument.

Consider the following example:

```
module m(logic a, b, c, d, rst1, clk1, clk2);

   logic rst;

   default clocking @(negedge clk1); endclocking
   default disable iff rst1;

   property p_triggers(start_event, end_event, form, clk = $inferred_clock,
                       rst = $inferred_disable);
      @clk disable iff (rst)
         (start_event ##0 end_event[->1]) |=> form;
   endproperty

   property p_multiclock(clkw, clkx = $inferred_clock, clky, w, x, y, z);
      @clkw w ##1 @clkx x |=> @clky y ##1 z;
   endproperty

   a1: assert property (p_triggers(a, b, c));
   a2: assert property (p_triggers(a, b, c, posedge clk1, 1'b0) );

   always @(posedge clk2 or posedge rst) begin
      if (rst) ... ;
      else begin
         a3: assert property (p_triggers(a, b, c));
         ...
      end
   end
```

```
      a4: assert property(p_multiclock(negedge clk2, , posedge clk1,
                          a, b, c, d) );

   endmodule
```

The preceding code is logically equivalent to the following:

```
   module m(logic a, b, c, d, rst1, clk1, clk2);

      logic rst;

      a1: assert property (@(negedge clk1) disable iff (rst1)
                              a ##0 b[->1] |=> c);

      a2: assert property (@(posedge clk1) disable iff (1'b0)
                              a ##0 b[->1] |=> c);

      always @(posedge clk2 or posedge rst) begin
         if (rst) ... ;
         else begin
            ...
         end
      end

      a3: assert property
         (
            @(posedge clk2) disable iff (rst1)
            (a ##0 b[->1]) |=> c
         );

      a4: assert property (@(negedge clk2) a ##1 @(negedge clk1) b |=>
            @(posedge clk1) c ##1 d);

   endmodule
```

In assertion `a1` the clock event is inferred from the default clocking, therefore `$inferred_clock` is **`negedge`** `clk1` for `a1`. In assertion `a2` the event expression **`posedge`** `clk1` is passed to the formal argument `clk` in the instance of property `p_triggers`. Therefore, the `$inferred_clock` is not used for `clk` in that instance. In assertion `a3` the clocking event is inferred from the event control of the always procedure, therefore `$inferred_clock` is **`posedge`** `clk2` for `a3`.

In assertion `a4`, as the property `p_multiclock` is instantiated in the **`assert property`** statement, `clkw` is replaced by the actual argument (**`negedge`** `clk2`), `clkx` by the default argument value `$inferred_clock`, which is the default clocking clock (**`negedge`** `clk1`) at the location of the property instance in the assertion. The third clock, `clky`, is replaced by the actual argument (**`posedge`** `clk1`) because it is explicitly specified.

The disable condition `rst1` is inferred for assertions `a1` and `a3` from the default disable statement. Assertion `a2` uses explicit reset value `1'b0` in which case the **`disable iff`** statement could be omitted altogether in the equivalent assertion.

### 16.14.8 Nonvacuous evaluations

An evaluation attempt of a property is either vacuous or nonvacuous. In particular, a vacuous success on all evaluation attempts may indicate a potential problem either in the design or in the formulation of the property. For example,

```
a |-> b
```

is evaluated as a vacuous success when a is false. In that case the evaluation is independent of the value of b; even though it is a successful evaluation, the property behavior is interpreted as not matching the user intent, meaning that an assertion of this property is not considered a pass or a failure.

For a general property, nonvacuous evaluation is defined recursively on the structure of the property as follows:

a) An evaluation attempt of a property that is a sequence is always nonvacuous.

b) An evaluation attempt of a property of the form **strong**(*sequence_expr*) is always nonvacuous.

c) An evaluation attempt of a property of the form **weak**(*sequence_expr*) is always nonvacuous.

d) An evaluation attempt of a property of the form **not** *property_expr* is nonvacuous if, and only if, the underlying evaluation attempt of property_expr is nonvacuous.

e) An evaluation attempt of a property of the form *property_expr1* **or** *property_expr2* is nonvacuous if, and only if, either the underlying evaluation attempt of *property_expr1* is nonvacuous or the underlying evaluation attempt of *property_expr2* is nonvacuous.

f) An evaluation attempt of a property of the form *property_expr1* **and** *property_expr2* is nonvacuous if, and only if, either the underlying evaluation attempt of *property_expr1* is nonvacuous or the underlying evaluation attempt of *property_expr2* is nonvacuous.

g) An evaluation attempt of a property of the form **if (** *expression_or_dist* **)** *property_expr1* is nonvacuous if, and only if, *expression_or_dist* evaluates to true and the underlying evaluation attempt of *property_expr1* is nonvacuous.

   An evaluation attempt of a property of the form **if** ( *expression_or_dist* ) *property_expr1* **else** *property_expr2* is nonvacuous if, and only if, either *expression_or_dist* evaluates to true and the underlying evaluation attempt of *property_expr1* is nonvacuous, or *expression_or_dist* evaluates to false and the underlying evaluation attempt of *property_expr2* is nonvacuous.

h) An evaluation attempt of a property of the form *sequence_expression* **|->** *property_expr* is nonvacuous if, and only if, there is an end point of the antecedent *sequence_expression* and the evaluation attempt of *property_expr* that starts at the end point is nonvacuous.

   An evaluation attempt of a property of the form *sequence_expression* **|=>** *property_expr* is nonvacuous if, and only if, there is a match point of the antecedent *sequence_expression* and the evaluation attempt of *property_expr* that starts at the clock event following the match point is nonvacuous.

i) An evaluation attempt of an instance of a property is nonvacuous if, and only if, the underlying evaluation attempt of the *property_expr* that results from substituting actual arguments for formal arguments is nonvacuous.

j) An evaluation attempt of a property of the form *sequence_expression* #-# *property_expr* is nonvacuous if, and only if, there is an end point of the antecedent *sequence_expression* and the evaluation attempt of *property_expr* that starts at the end point is nonvacuous.

k) An evaluation attempt of a property of the form *sequence_expression* #=# *property_expr* is nonvacuous if, and only if, there is a match point of the antecedent *sequence_expression* and the evaluation attempt of *property_expr* that starts at the clock event following the match point is nonvacuous.

l) An evaluation attempt of a property of the form **nexttime** *property_expr* is nonvacuous if, and only if, there is at least one more clock event, and in the evaluation attempt that starts in the next clock event, property_expr is nonvacuous.

m) An evaluation attempt of a property of the form **nexttime**[*constant_expression*] *property_expr* is nonvacuous if, and only if, there is at least *constant_expression* more clock events, and *property_expr* is nonvacuous in the evaluation attempt beginning at the last of the next *constant_expression* clock events.

n) An evaluation attempt of a property of the form **`s_nexttime`** *property_expr* is nonvacuous if, and only if, there is at least one more clock event, and in the evaluation attempt starting at the next clock event, *property_expr* is nonvacuous.

o) An evaluation attempt of a property of the form **`s_nexttime`** `[`*constant_expression*`]` *property_expr* is nonvacuous if, and only if, there is at least *constant_expression* more clock events, and *property_expr* is nonvacuous in the evaluation attempt beginning at the last of the next *constant_expression* clock events.

p) An evaluation attempt of a property of the form **`always`** *property_expr* is nonvacuous if, and only if, there is a clock event where the evaluation attempt of *property_expr* is nonvacuous, and *property_expr* does not fail in prior clock events.

q) An evaluation attempt of a property of the form **`always`** `[`*cycle_delay_const_range_expression*`]` *property_expr* is nonvacuous if, and only if, there is a clock event within the range specified by *cycle_delay_const_range_expression*, in which the evaluation attempt of *property_expr* is nonvacuous, and the *property_expr* does not fail in prior clock events within the range specified by *cycle_delay_const_range_expression*.

r) An evaluation attempt of a property of the form **`s_always`** `[`*constant_range*`]` *property_expr* is nonvacuous if, and only if, there is a clock event within the range specified by *constant_range*, in which the evaluation attempt of *property_expr* is nonvacuous, and *property_expr* does not fail in prior clock events within the range specified by *constant_range*.

s) An evaluation attempt of a property of the form **`s_eventually`** *property_expr* is nonvacuous if, and only if, there is a clock event in which the evaluation attempt of *property_expr* is nonvacuous, and the *property_expr* does not hold in prior clock events.

t) An evaluation attempt of a property of the form **`s_eventually`** `[`*cycle_delay_const_range_expression*`]` *property_expr* is nonvacuous if, and only if, there is a clock event within the range specified by *cycle_delay_const_range_expression*, in which the evaluation attempt of *property_expr* is nonvacuous, and *property_expr* does not hold in prior clock events within the range specified by *cycle_delay_const_range_expression*.

u) An evaluation attempt of a property of the form **`eventually`** `[`*constant_range*`]` *property_expr* is nonvacuous if, and only if, there is a clock event within the range specified by *constant_range*, in which the evaluation attempt of *property_expr* is nonvacuous, and *property_expr* does not hold in prior clock events within the range specified by *constant_range*.

v) An evaluation attempt of a property of the form *property_expr1* **`until`** *property_expr2* is nonvacuous if, and only if, there is a clock event in which either the evaluation attempt of *property_expr1* or the evaluation attempt of *property_expr2* is nonvacuous, *property_expr2* does not hold in prior clock events, and *property_expr1* holds in all prior clock events.

w) An evaluation attempt of a property of the form *property_expr1* **`s_until`** *property_expr2* is nonvacuous if, and only if, there is a clock event in which either the evaluation attempt of *property_expr1* or the evaluation attempt of *property_expr2* is nonvacuous, *property_expr2* does not hold in prior clock events, and *property_expr1* holds in all prior clock events.

x) An evaluation attempt of a property of the form *property_expr1* **`until_with`** *property_expr2* is nonvacuous if, and only if, there is a clock event in which the evaluation attempt of *property_expr1* is nonvacuous, *property_expr2* does not hold in prior clock events, and *property_expr1* holds in all prior clock events.

y) An evaluation attempt of a property of the form *property_expr1* **`s_until_with`** *property_expr2* is nonvacuous if, and only if, there is a clock event in which the evaluation attempt of *property_expr1* is nonvacuous, *property_expr2* does not hold in prior clock events, and *property_expr1* holds in all prior clock events.

z) An evaluation attempt of a property of the form *property_expr1* **`implies`** *property_expr2* is nonvacuous if, and only if, the underlying evaluation attempt of *property_expr1* is true and nonvacuous, and the underlying evaluation attempt of *property_expr2* is nonvacuous.

aa) An evaluation attempt of a property of the form *property_expr1* **`iff`** *property_expr2* is nonvacuous if, and only if, either the evaluation attempt of *property_expr1* is nonvacuous or the evaluation attempt of *property_expr2* is nonvacuous.

ab) An evaluation attempt of a property of the form **`accept_on`**`(`*expression_or_dist*`)` *property_expr* is nonvacuous if, and only if, the underlying evaluation attempt of *property_expr* is nonvacuous and *expression_or_dist* does not hold in any time step of that evaluation attempt.

ac) An evaluation attempt of a property of the form **`reject_on`**`(`*expression_or_dist*`)` *property_expr* is nonvacuous if, and only if, the underlying evaluation attempt of *property_expr* is nonvacuous and *expression_or_dist* does not hold in any time step of that evaluation attempt.

ad) An evaluation attempt of a property of the form **`sync_accept_on`**(*expression_or_dist*) property_expr is nonvacuous if, and only if, the underlying evaluation attempt of property_expr is nonvacuous and expression_or_dist does not hold in any clock event of that evaluation attempt.

ae) An evaluation attempt of a property of the form **`sync_reject_on`**(*expression_or_dist*) property_expr is nonvacuous if, and only if, the underlying evaluation attempt of property_expr is nonvacuous and expression_or_dist does not hold in any clock event of that evaluation attempt.

af) An evaluation attempt of a property of the form

```
case (expression_or_dist)
    expression_or_dist1 : property_stmt1
    ...
    expression_or_distn : property_stmtn
        [ default : property_stmtd ]
endcase
```

is nonvacuous if, and only if:

- For some index *i* such that 1 <= *i* <= *n*, (*expression_or_dist* === *expression_or_dist*$_i$), and
- For each index *j* such that 1 <= *j* < *i*, (*expression_or_dist* !== *expression_or_dist*$_j$), and
- The underlying evaluation attempt of *property_stmt*$_i$ is nonvacuous

or

- The default is present, and
- For each index *i* such that 1 <= *i* <= *n*, (*expression_or_dist* !== *expression_or_dist*$_i$), and
- The underlying evaluation attempt of *property_stmt*$_d$ is nonvacuous.

ag) An evaluation attempt of a property of the form **`disable iff`** `(`*expression_or_dist*`)` *property_expr* is nonvacuous if, and only if, the underlying evaluation attempt of *property_expr* is nonvacuous and *expression_or_dist* does not hold in any time step of that evaluation attempt.

An evaluation attempt of a property succeeds nonvacuously if, and only if, the property evaluates to true and the evaluation attempt is nonvacuous.

## 16.15 Disable iff resolution

```
module_or_generate_item_declaration ::=                          // from A.1.4
    ...
  | default clocking clocking_identifier ;
  | default disable iff expression_or_dist ;
```

*Syntax 16-21—Default clocking and default disable syntax (excerpt from Annex A)*

A **default disable iff** may be declared within a generate block or within a module, interface, or program declaration. It provides a default disable condition to all concurrent assertions in the scope and subscopes of the **default disable iff** declaration. Furthermore, the default extends to any nested module, interface, or program declarations, and to nested generate blocks. However, if a nested module, interface, or program declaration, or a generate block itself has a **default disable iff** declaration, then that **default disable iff** applies within the nested declaration or generate block and overrides any **default disable iff** from outside. Any signals referenced in the **disable iff** declaration that are resolved using scopes will be resolved from the scope of the declaration.

The effect of a **default disable iff** declaration is independent of the position of the declaration within that scope. More than one **default disable iff** declaration within the same module, interface, program declaration, or generate block shall be an error. The scope does not extend into any instances of modules, interfaces, or programs.

In the following example, module m1 declares rst1 to be the default disable condition, and there is no **default disable iff** declaration in the nested module m2. The default disable condition rst1 applies throughout the declaration of m1 and the nested declaration of m2. Therefore, the inferred disable condition of both assertions a1 and a2 is rst1.

```
module m1;
   bit clk, rst1;
   default disable iff rst1;
   a1: assert property (@(posedge clk) p1);     // property p1 is
                                                // defined elsewhere
   ...
   module m2;
      bit rst2;
      ...
      a2: assert property (@(posedge clk) p2);  // property p2 is
                                                // defined elsewhere
   endmodule
   ...
endmodule
```

If there is a **default disable iff** declaration in the nested module m2, then within m2 this default disable condition overrides the default disable condition declared in m1. Therefore, in the following example the inferred disable condition of a1 is rst1, but the inferred disable condition of a2 is rst2.

```
module m1;
   bit clk, rst1;
   default disable iff rst1;
   a1: assert property (@(posedge clk) p1);     // property p1 is
                                                // defined elsewhere
   ...
   module m2;
      bit rst2;
      default disable iff rst2;
      ...
      a2: assert property (@(posedge clk) p2);  // property p2 is
                                                // defined elsewhere
   endmodule
   ...
endmodule
```

The following rules apply for resolution of the disable condition:

a) If an assertion has a **`disable iff`** clause, then the disable condition specified in this clause shall be used and any **`default disable iff`** declaration ignored for this assertion.

b) If an assertion does not contain a **`disable iff`** clause, but the assertion is within the scope of a **`default disable iff`** declaration, then the disable condition for the assertion is inferred from the **`default disable iff`** declaration.

c) Otherwise, no inference is performed (this is equivalent to the inference of a `1'b0` disable condition).

Following are two example modules illustrating the application of these rules:

```
module examples_with_default (input logic a, b, clk, rst, rst1);
   default disable iff rst;
   property p1;
      disable iff (rst1) a |=> b;
   endproperty

   // Disable condition is rst1 - explicitly specified within a1
   a1 : assert property (@(posedge clk) disable iff (rst1) a |=> b);

   // Disable condition is rst1 - explicitly specified within p1
   a2 : assert property (@(posedge clk) p1);

   // Disable condition is rst - no explicit specification, inferred from
   // default disable iff declaration
   a3 : assert property (@(posedge clk) a |=> b);

   // Disable condition is 1'b0. This is the only way to
   // cancel the effect of default disable.
   a4 : assert property (@(posedge clk) disable iff (1'b0) a |=> b);
endmodule

module examples_without_default (input logic a, b, clk, rst);
   property p2;
      disable iff (rst) a |=> b;
   endproperty

   // Disable condition is rst - explicitly specified within a5
   a5 : assert property (@(posedge clk) disable iff (rst) a |=> b);

   // Disable condition is rst - explicitly specified within p2
   a6 : assert property (@ (posedge clk) p2);

   // No disable condition
   a7 : assert property (@ (posedge clk) a |=> b);

endmodule
```

## 16.16 Clock resolution

There are a number of ways to specify a clock for a property. They are as follows:

— Sequence instance with a clock, for example:

```
      sequence s2; @(posedge clk) a ##2 b; endsequence
      property p2; not s2; endproperty
      assert property (p2);
```

— Property, for example:

```
property p3; @(posedge clk) not (a ##2 b); endproperty
assert property (p3);
```

— Contextually inferred clock from a procedural block, for example:

```
always @(posedge clk) assert property (not (a ##2 b));
```

— A **clocking** block, for example:

```
clocking master_clk @(posedge clk);
   property p3; not (a ##2 b); endproperty
endclocking
assert property (master_clk.p3);
```

— Default clock, for example:

```
default clocking master_clk ;  // master clock as defined above
property p4; (a ##2 b); endproperty
assert property (p4);
```

In general, a clocking event applies throughout its scope except where superseded by an inner clocking event, as with clock flow in multiclocked sequences and properties. The following rules apply (the term *maximal property*, used in the rules below, is defined as the unique flattened property contained in the assertion statement and obtained by applying the rewriting algorithm in F.4.1):

a) In a module, interface, program, or checker with a default clocking event, a concurrent assertion statement that has no otherwise specified leading clocking event is treated as though the default clocking event had been written explicitly as the leading clocking event. The default clocking event does not apply to a sequence or property declaration except in the case that the declaration appears in a **clocking** block whose clocking event is the default.

b) The following rules apply within a **clocking** block:
   1) No explicit clocking event is allowed in any property or sequence declaration within the **clocking** block. All sequence and property declarations within the **clocking** block are treated as though the clocking event of the **clocking** block had been written explicitly as the leading clocking event.
   2) Multiclocked sequences and properties are not allowed within the **clocking** block.
   3) If a named sequence or property that is declared outside the **clocking** block is instantiated within the **clocking** block, the instance shall be singly clocked and its clocking event shall be identical to that of the **clocking** block.

c) A contextually inferred clocking event from a procedural block supersedes a default clocking event. The contextually inferred clocking event is treated as though it had been written as the leading clocking event of any concurrent assertion statement to which the inferred clock applies.

d) An explicitly specified leading clocking event in a concurrent assertion statement supersedes a default clocking event.

e) A multiclocked sequence or property can inherit the default clocking event as its leading clocking event. If a multiclocked property is the maximal property of a concurrent assertion statement, then the property shall have a unique semantic leading clock (see 16.16.1).

f) If a concurrent assertion statement has no explicit leading clocking event, there is no default clocking event, and no contextually inferred clocking event applies to the assertion statement, then the maximal property of the assertion statement shall be an instance of a sequence or property for which a unique leading clocking event is determined.

The following are two example modules illustrating the application of these rules with some legal and some illegal declarations, as indicated by the comments:

```
module examples_with_default (input logic a, b, c, clk);

   property q1;
      $rose(a) |-> ##[1:5] b;
   endproperty

   property q2;
      @(posedge clk) q1;
   endproperty

   default clocking posedge_clk @(posedge clk);
      property q3;
         $fell(c) |=> q1;
         // legal: q1 has no clocking event
      endproperty

      property q4;
         $fell(c) |=> q2;
         // legal: q2 has clocking event identical to that of
         // the clocking block
      endproperty

      sequence s1;
         @(posedge clk) b[*3];
            // illegal: explicit clocking event in clocking block
      endsequence
   endclocking

   property q5;
      @(negedge clk) b[*3] |=> !b;
   endproperty

   always @(negedge clk)
   begin
      a1: assert property ($fell(c) |=> q1);
         // legal: contextually inferred leading clocking event,
         // @(negedge clk)
      a2: assert property (posedge_clk.q4);
         // legal: will be queued (pending) on negedge clk, then
         // (if matured) checked at next posedge clk (see 16.14.6)
      a3: assert property ($fell(c) |=> q2);
         // illegal: multiclocked property with contextually
         // inferred leading clocking event
      a4: assert property (q5);
         // legal: contextually inferred leading clocking event,
         // @(negedge clk)
   end

   property q6;
      q1 and q5;
   endproperty

   a5: assert property (q6);
      // illegal: default leading clocking event, @(posedge clk),
      // but semantic leading clock is not unique
   a6: assert property ($fell(c) |=> q6);
      // legal: default leading clocking event, @(posedge clk),
      // is the unique semantic leading clock

   sequence s2;
      $rose(a) ##[1:5] b;
   endsequence

   c1: cover property (s2);
      // legal: default leading clocking event, @(posedge clk)
   c2: cover property (@(negedge clk) s2);
      // legal: explicit leading clocking event, @(negedge clk)

endmodule


module examples_without_default (input logic a, b, c, clk);

   property q1;
      $rose(a) |-> ##[1:5] b;
   endproperty

   property q5;
      @(negedge clk) b[*3] |=> !b;
   endproperty

   property q6;
      q1 and q5;
   endproperty

   a5: assert property (q6);
      // illegal: no leading clocking event
   a6: assert property ($fell(c) |=> q6);
      // illegal: no leading clocking event

   sequence s2;
      $rose(a) ##[1:5] b;
   endsequence

   c1: cover property (s2);
      // illegal: no leading clocking event
   c2: cover property (@(negedge clk) s2);
      // legal: explicit leading clocking event, @(negedge clk)

   sequence s3;
      @(negedge clk) s2;
   endsequence

   c3: cover property (s3);
      // legal: leading clocking event, @(negedge clk),
      // determined from declaration of s3
   c4: cover property (s3 ##1 b);
      // illegal: no default, inferred, or explicit leading
      // clocking event and maximal property is not an instance
```

```
endmodule
```

### 16.16.1 Semantic leading clocks for multiclocked sequences and properties

Throughout this subclause, *s*, $s_1$, and $s_2$ denote sequences without clocking events; *p*, $p_1$, and $p_2$ denote properties without clocking events; *m*, $m_1$, and $m_2$ denote multiclocked sequences, *q*, $q_1$, and $q_2$ denote multiclocked properties; and *c*, $c_1$, and $c_2$ denote nonidentical clocking event expressions.

This subclause defines a notion of the set of semantic leading clocks for a multiclocked sequence or property.

Some sequences and properties have no explicit leading clock event. Their initial clocking event is inherited from an outer clocking event according to the flow of clocking event scope. In this case, the semantic leading clock is said to be *inherited*. For example, in the property

```
@(c) s |=> p and @(c1) p1
```

the semantic leading clock of the subproperty *p* is inherited because the initial clock of *p* is the clock that flows across `|=>`.

A multiclocked sequence has a unique semantic leading clock, defined inductively as follows:

— The semantic leading clock of *s* is *inherited*.
— The semantic leading clock of `@(c)` *s* is *c*.
— If *inherited* is the semantic leading clock of *m*, then the semantic leading clock of `@(c)` *m* is *c*. Otherwise, the semantic leading clock of `@(c)` *m* is equal to the semantic leading clock of *m*.
— The semantic leading clock of (*m*) is equal to the semantic leading clock of *m*.
— The semantic leading clock of $m_1$ `##1` $m_2$ is equal to the semantic leading clock of $m_1$.
— The semantic leading clock of $m_1$ `##0` $m_2$ is equal to the semantic leading clock of $m_1$.

The set of semantic leading clocks of a multiclocked property is defined inductively as follows:

— The set of semantic leading clocks of **`strong`**`(`*m*`)` is {*c*}, where *c* is the unique semantic leading clock of *m*.
— The set of semantic leading clocks of **`weak`**`(`*m*`)` is {*c*}, where *c* is the unique semantic leading clock of *m*.
— The set of semantic leading clocks of *p* is {*inherited*}.
— If *inherited* is an element of the set of semantic leading clocks of *q*, then the set of semantic leading clocks of `@(c)` *q* is obtained from the set of semantic leading clocks of *q* by replacing *inherited* by *c*. Otherwise, the set of semantic leading clocks of `@(c)` *q* is equal to the set of semantic leading clocks of *q*.
— The set of semantic leading clocks of (*q*) is equal to the set of semantic leading clocks of *q*.
— The set of semantic leading clocks of **`not`** *q* is equal to the set of semantic leading clocks of *q*.
— The set of semantic leading clocks of $q_1$ **`and`** $q_2$ is the union of the set of semantic leading clocks of $q_1$ with the set of semantic leading clocks of $q_2$.
— The set of semantic leading clocks of $q_1$ **`or`** $q_2$ is the union of the set of semantic leading clocks of $q_1$ with the set of semantic leading clocks of $q_2$.
— The set of semantic leading clocks of *m* `|->` *p* is equal to the set of semantic leading clocks of *m*.
— The set of semantic leading clocks of *m* `|=>` *p* is equal to the set of semantic leading clocks of *m*.
— The set of semantic leading clocks of **`if`** (*b*) *q* is {*inherited*}.
— The set of semantic leading clocks of **`if`** (*b*) $q_1$ **`else`** $q_2$ is {*inherited*}.

- The set of semantic leading clocks of **`case`** ($b$) $b_1$`:` $q_1$ … $b_n$`:` $q_n$ [**`default:`** $q_d$] **`endcase`** is {*inherited*}.
- The set of semantic leading clocks of **`nexttime`** $q$ is {*inherited*}.
- The set of semantic leading clocks of **`always`** $q$ is {*inherited*}.
- The set of semantic leading clocks of **`s_eventually`** $q$ is {*inherited*}.
- The set of semantic leading clocks of $q_1$ **`until`** $q_2$ is {*inherited*}.
- The set of semantic leading clocks of $q_1$ **`until_with`** $q_2$ is {*inherited*}.
- The set of semantic leading clocks of **`accept_on`**`(`$b$`)` $q$ is the set of semantic leading clocks of $q$.
- The set of semantic leading clocks of **`reject_on`**`(`$b$`)` $q$ is the set of semantic leading clocks of $q$.
- The set of semantic leading clocks of **`sync_accept_on`**`(`$b$`)` $q$ is {*inherited*}.
- The set of semantic leading clocks of **`sync_reject_on`**`(`$b$`)` $q$ is {*inherited*}.
- The set of semantic leading clocks of a property instance is equal to the set of semantic leading clocks of the multiclocked property obtained from the body of its declaration by substituting in actual arguments.

For example, the multiclocked sequence

`@(`$c_1$`)` $s_1$ `##1 @(`$c_2$`)` $s_2$

has $c_1$ as its unique semantic leading clock, while the multiclocked property

**`not`** `(`$p_1$ **`and`** `(@(`$c_2$`)` $p_2$`)`

has {*inherited*, $c_2$} as its set of semantic leading clocks.

In the presence of an incoming outer clock, the inherited semantic leading clock is always understood to refer to the incoming outer clock. Therefore, the clocking of a property $q$ in the presence of incoming outer clock $c$ is equivalent to the clocking of the property `@(c)` $q$.

A multiclocked property has a unique semantic leading clock in cases where all its leading clocks are identical. Consider the following example:

```
wire clk1, clk2;
logic a, b;
...
assign clk2 = clk1;
a1: assert property (@(clk1) a and @(clk2) b); // Illegal
a2: assert property (@(clk1) a and @(clk1) b); // OK
always @(posedge clk1) begin
   a3: assert property(a and @(posedge clk2)); //Illegal
   a4: assert property(a and @(posedge clk1)); // OK
end
```

The assertions `a2` and `a4` are legal, while the assertions `a1` and `a3` are not. Though both clocks of `a1` have the same value, they are not identical. Therefore, `a1` does not have a unique semantic leading clock. The assertions `a3` and `a4` have `@(`**`posedge`** `clk1)` as their inferred clock. This clock is not identical to `@(`**`posedge`** `clk2)` therefore `a3` does not have a unique semantic leading clock.

## 16.17 Expect statement

The **expect** statement is a procedural blocking statement that allows waiting on a property evaluation. The syntax of the **expect** statement accepts a named property or a property declaration and is given in Syntax 16-22.

```
expect_property_statement ::=                                    // from A.2.10
    expect ( property_spec ) action_block
```

*Syntax 16-22—Expect statement syntax (excerpt from Annex A)*

The **expect** statement accepts the same syntax used to assert a property. An **expect** statement causes the executing process to block until the given property succeeds or fails. The statement following the **expect** is scheduled to execute after processing the Observed region in which the property completes its evaluation. When the property succeeds or fails, the process unblocks, and the property stops being evaluated (i.e., no property evaluation is started until that **expect** statement is executed again).

When executed, the **expect** statement starts a single thread of evaluation for the given property on the subsequent clocking event, that is, the first evaluation shall take place on the next clocking event. If the property fails at its clocking event, the optional **else** clause of the action block is executed. If the property succeeds, the optional pass statement of the action block is executed. The execution of pass and fail statements can be controlled by using assertion action control tasks. The assertion action control tasks are described in 20.12.

```
program tst;
   initial begin
      # 200ms;
      expect( @(posedge clk) a ##1 b ##1 c ) else $error( "expect failed" );
      ABC: ...
   end
endprogram
```

In the preceding example, the **expect** statement specifies a property that consists of the sequence `a ##1 b ##1 c`. The **expect** statement (second statement in the **initial** procedure of program `tst`) blocks until the sequence `a ##1 b ##1 c` is matched or is determined not to match. The property evaluation starts on the occurrence of the **posedge** `clk` event following the 200 ms delay. If the sequence is matched, the process is unblocked and continues to execute on the statement labeled `ABC`. If the sequence fails to match, then the **else** clause is executed, which in this case generates a run-time error. For the preceding **expect** statement to succeed, the sequence `a ##1 b ##1 c` must match starting on the occurrence of the **posedge** `clk` event immediately after time `200ms`. The sequence will not match if `a`, `b`, or `c` is evaluated to be false at the first, second, or third clocking event occurrence, respectively.

The **expect** statement can appear anywhere a **wait** statement (see 9.4.3) can appear. Because it is a blocking statement, the property can refer to automatic variables as well as static variables. For example, the task below waits between 1 and 10 clock ticks for the variable `data` to equal a particular value, which is specified by the automatic argument value. The second argument, `success`, is used to return the result of the **expect** statement: 1 for success and 0 for failure.

```
integer data;
...
task automatic wait_for( integer value, output bit success );
expect( @(posedge clk) ##[1:10] data == value ) success = 1;
   else success = 0;
endtask
```

```
initial begin
   bit ok;
   wait_for( 23, ok );  // wait for the value 23
   ...
end
```

## 16.18 Clocking blocks and concurrent assertions

If a variable used in a concurrent assertion is a **clocking** block variable, it will be sampled only in the **clocking** block.

*Examples:*

```
module A;
   logic a, clk;

   clocking cb_with_input @(posedge clk);
      input a;
      property p1;
         a;
      endproperty
   endclocking

   clocking cb_without_input @(posedge clk);
      property p1;
         a;
      endproperty
   endclocking

   property p1;
      @(posedge clk) a;
   endproperty

   property p2;
      @(posedge clk) cb_with_input.a;
   endproperty

   a1: assert property (p1);
   a2: assert property (cb_with_input.p1);
   a3: assert property (p2);
   a4: assert property (cb_without_input.p1);

endmodule
```

Figure 16-17 explains the behavior of all the assertions. In the preceding example, `a1`, `a2`, `a3`, and `a4` are equivalent.

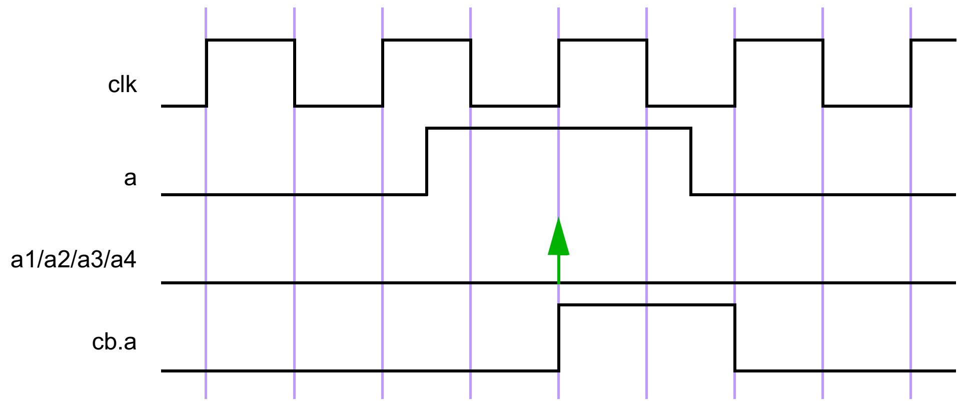


**Figure 16-17—Clocking blocks and concurrent assertion**

# 17. Checkers

## 17.1 Overview

Assertions provide building blocks to validate the behavior of the design. In many cases there is a need to group several assertions together into bigger blocks having a well-defined functionality. These verification blocks may also need to contain modeling code to compute values of auxiliary variables used in assertions or covergroup instances to be integrated with cover statements. The checker construct in SystemVerilog was specifically created to represent such verification blocks encapsulating assertions along with the modeling code. The intended use of checkers is to serve as verification library units, or as building blocks for creating abstract auxiliary models used in formal verification.

The modeling mechanism in checkers is similar to the modeling mechanism in modules and interfaces, though several limitations apply. For example, no nets can be declared and assigned in checkers. On the other hand, checkers allow nondeterministic modeling, which does not exist in modules and interfaces. Each variable declared in a checker may be either deterministic or random. Checker modeling is explained in 17.7. Random variables are useful to build abstract nondeterministic models for formal verification. Reasoning about nondeterministic models is sometimes much easier than reasoning about deterministic RTL models.

Deterministic variables allow a conventional (deterministic) modeling for assertions. Using random variables instead of regular variables in checkers has the advantage that the same checker may be used for both deterministic and nondeterministic cases.

## 17.2 Checker declaration

```
checker_declaration ::=                                                    // from A.1.2
    checker checker_identifier [ ( [ checker_port_list ] ) ] ;
        { { attribute_instance } checker_or_generate_item }
    endchecker [ : checker_identifier ]
checker_port_list ::=                                                      // from A.1.8
    checker_port_item {, checker_port_item}
checker_port_item ::=
    { attribute_instance } [ checker_port_direction ] property_formal_type formal_port_identifier
        {variable_dimension} [ = property_actual_arg ]
checker_port_direction ::=
    input | output
checker_or_generate_item ::=
    checker_or_generate_item_declaration
  | initial_construct
  | always_construct
  | final_construct
  | assertion_item
  | continuous_assign
  | checker_generate_item
checker_or_generate_item_declaration ::=
    [ rand ] data_declaration
  | function_declaration
  | checker_declaration
  | assertion_item_declaration
```

```
    | covergroup_declaration
    | genvar_declaration
    | clocking_declaration
    | default clocking clocking_identifier ;
    | default disable iff expression_or_dist ;
    | ;
checker_generate_item[6] ::=
      loop_generate_construct
    | conditional_generate_construct
    | generate_region
    | elaboration_system_task
checker_identifier ::=                                                // from A.9.3
      identifier
```

6) It shall be illegal for a checker_generate_item to include any item that would be illegal in a checker_declaration outside a checker_generate_item.

*Syntax 17-1—Checker declaration syntax (excerpt from Annex A)*

A checker may be declared in one of the following:

— A module
— An interface
— A program
— A checker
— A package
— A generate block
— A compilation unit scope

A checker is declared using the keyword **checker** followed by a name and optional formal argument list, and ending with the keyword **endchecker**.

The following elements from the scope enclosing the checker declaration shall not be referenced in a checker:

— Automatic variables and members or elements of dynamic variables (see 6.21).
— Elements of **fork-join**, **fork-join_any**, or **fork-join_none** blocks.

Action blocks of assertions within a checker will be referred to as *checker action blocks*, and the rest of the checker will be referred to as a *checker body*.

A checker body may contain the following elements:

— Declarations of **let** constructs, sequences, properties, and functions
— Deferred assertions (see 16.4)
— Concurrent assertions (see 16.14)
— Checker declarations
— Other checker instantiations
— Covergroup declarations and instances
— Checker variable declarations and assignments (see 17.7)
— **default clocking** and **default disable iff** declarations

— **initial**, **always_comb**, **always_latch**, **always_ff**, and **final** procedures (see 9.2)
— Generate blocks, containing any of the above elements

Modules, interfaces, programs, and packages shall not be declared inside checkers. Modules, interfaces, and programs shall not be instantiated inside checkers.

A formal argument of a checker may be optionally preceded by a direction qualifier: **input** or **output**. If no direction is specified explicitly then the direction of the previous argument shall be inferred. If the direction of the first checker argument is omitted, it shall default to **input**. An input checker formal argument shall not be modified by a checker.

The legal data types for checker formal arguments are those legal for a property (see 16.12). The type of an output argument shall not be of **untyped**, **sequence**, or **property**. If the type of a checker formal argument is omitted, it is inferred according to the following rules:

— If the argument has an explicit direction qualifier, it shall be an error to omit its type.
— Otherwise, if the argument is the first argument of the checker, it is assumed to be **input untyped**.
— Otherwise, the type of the previous formal argument is inferred as described for sequences and properties (see 16.8 and 16.12).

In a similar manner to sequences and properties, a checker declaration may specify a default value for each singular input port, as described in 16.8. A checker declaration may also specify an initial value for each singular output port using the same syntax as the default value specification for input arguments. Checker output port initialization has the same semantics as a variable initialization (see 6.8).

Following are examples of simple checkers:

*Example 1:*

```
// Simple checker containing concurrent assertions
checker my_check1 (logic test_sig, event clock);
   default clocking @clock; endclocking
   property p(logic sig);
      ...
   endproperty
   a1: assert property (p (test_sig));
   c1: cover property (!test_sig ##1 test_sig);
endchecker : my_check1
```

*Example 2:*

```
// Simple checker containing deferred assertions
checker my_check2 (logic a, b);
   a1: assert #0 ($onehot0({a, b});
   c1: cover  #0 (a == 0 && b == 0);
   c2: cover  #0 (a == 1);
   c3: cover  #0 (b == 1);
endchecker : my_check2
```

*Example 3:*

```
// Simple checker with output arguments
checker my_check3 (logic a, b, event clock, output bit failure, undef);
   default clocking @clock; endclocking
   a1: assert property ($onehot0({a, b}) failure = 1'b0; else failure = 1'b1;
   a2: assert property ($isunknown({a, b}) undef = 1'b0; else undef = 1'b1;
endchecker : my_check3
```

*Example 4:*

```
// Checker with default input and initialized output arguments
checker my_check4 (input logic in,
                   en = 1'b1, // default value
                   event clock,
                   output int ctr = 0); // initial value
   default clocking @clock; endclocking
   always_ff @clock
      if (en && in) ctr <= ctr + 1;
   a1: assert property (ctr < 5);
endchecker : my_check4
```

Type and data declarations within the checker are local to the checker scope and are static. Clock and **disable iff** contexts are inherited from the scope of the checker declaration (but see 17.4 for usage of context value functions for passing the instantiation context to the checker). For example:

```
module m;
   default clocking @clk1; endclocking
   default disable iff rst1;
   checker c1;
      // Inherits @clk1 and rst1
      ...
   endchecker : c1
   checker c2;
      // Explicitly redefines its default values
      default clocking @clk2; endclocking
      default disable iff rst2;
      ...
   endchecker : c2
   ...
endmodule : m
```

Variables used in a checker that are neither formal arguments to the checker nor internal variables of the checker are resolved according to the scoping rules from the scope in which the checker is declared.

## 17.3 Checker instantiation

---

concurrent_assertion_item ::= *// from* A.2.10
    ...
  | checker_instantiation

checker_instantiation ::= *// from* A.4.1.4
    ps_checker_identifier name_of_instance **(** [list_of_checker_port_connections] **)** **;**

list_of_checker_port_connections[29] ::=
    ordered_checker_port_connection { **,** ordered_checker_port_connection }
  | named_checker_port_connection { **,** named_checker_port_connection }

ordered_checker_port_connection ::= { attribute_instance } [ property_actual_arg ]

named_checker_port_connection ::=
    { attribute_instance } **.** formal_port_identifier [ **(** [ property_actual_arg ] **)** ]
  | { attribute_instance } **.***

ps_checker_identifier ::= *// from A.9.3*
    [ package_scope ] checker_identifier

29) The `.*` token shall appear at most once in a list of port connections.

*Syntax 17-2—Checker instantiation syntax (excerpt from Annex A)*

A checker may be instantiated wherever a concurrent assertion may appear (see 16.14) with the following exceptions:

— It shall be illegal to instantiate checkers in **`fork-join`**, **`fork-join_any`**, or **`fork-join_none`** blocks.
— It shall be illegal to instantiate a checker in a procedure of another checker.

A checker has different behavior depending on whether it is instantiated inside or outside procedural code. A checker instantiation in procedural code is referred to as a *procedural checker instance*. A checker instantiation outside procedural code is referred to as a *static checker instance*. See 16.14.6 for the corresponding definitions of procedural and static assertion statements.

When a checker is instantiated, actual arguments are passed to the checker. The mechanism for passing input arguments to a checker is similar to the mechanism for passing arguments to a property (see 16.12). The rewriting algorithm for checkers (see F.4.2) applies to checker input arguments. The rewriting algorithm substitutes actual arguments for references to the corresponding formal arguments in the body of the declaration of the checker. The rewriting algorithm does not itself account for name resolution and assumes that names have been resolved prior to the substitution of actual arguments. If the flattened checker is not legal, then the instance is not legal and there shall be an error.

The following restrictions apply:

— As in the case of sequences and properties, if `$` is an actual input argument to a checker instance, then the corresponding formal argument shall be untyped and each of its references either shall be an upper bound in a *cycle_delay_const_range_expression* or shall itself be an actual argument in an instance of a named sequence or property, or in a checker instance.
— If an actual input argument contains any subexpression that is a **`const`** cast or automatic value from procedural code, then the corresponding formal argument shall not be used in a continuous assignment or in the procedural code within the checker.

Checker actual output arguments shall be *variable_lvalue* or *net_lvalue*. The checker instantiation should be treated as if there were continuous assignments, executed in the Reactive region, of the checker's output formal arguments to their corresponding actual arguments.

Checker formal arguments may be connected to their actual arguments in ways similar to module ports (see 23.3.2):

— Positional connections by port order.
— Named port connections using fully explicit connections.
— Named port connections using implicit connections.
— Named port connections using a wildcard port name.

The following example illustrates a checker instantiation:

```
checker mutex (logic [31:0] sig, event clock, output bit failure);
   assert property (@clock $onehot0(sig))
      failure = 1'b0; else failure = 1'b1;
```

```
endchecker : mutex

module m(wire [31:0] bus, logic clk);
   logic res, scan;
   // ...
   mutex check_bus(bus, posedge clk, res);
   always @(posedge clk) scan <= res;
endmodule : m
```

On each rising edge of `clk` the bits of `bus` are checked for mutual exclusion and the result is assigned to `res` in the Reactive region. If `clk` is changed in the Active region, `scan` will capture the value of `res` generated on the previous rising edge of `clk`.

All contents of a checker instance other than static assertion statements are considered to exist during every time step, regardless of whether the checker is static or procedural. One copy of these contents exists for each instantiation. Procedural concurrent assertion statements in a checker shall be treated just like other procedural assertion statements as described in 16.14.6. However, static assertion statements within a checker are treated as if they appear at the checker's instantiation point. If the checker is instantiated inside some scope, any of its static assertions, both concurrent and deferred, are treated as if instantiated in this scope. Therefore, the following applies for static assertions within a checker:

— If the checker is static, the concurrent assertions are continually monitored, and begin execution on any time step matching their initial clock event. The deferred assertions are monitored whenever their expressions change.

— If the checker is procedural, all static concurrent assertions in the checker are added to the pending procedural assertion queue when the checker instantiation is reached in process execution, and then may mature or be flushed like any procedural concurrent assertion (see 16.14.6.2). Similarly all static deferred assertions in the checker are added to the pending deferred assertion report when the checker instantiation is reached in its process execution, and may mature or be flushed like any procedural deferred assertion (see 16.4.1).

— If the checker is statically instantiated inside another checker, any of its static assertions, concurrent or deferred, are treated as if instantiated in the parent checker, and thus will be treated as procedural assertions when an instantiation of its top-level ancestor in the checker hierarchy is visited in procedural code.

The following example illustrates this behavior:

```
checker c1(event clk, logic[7:0] a, b);
   logic [7:0] sum;
   always_ff @(clk) begin
      sum <= a + 1'b1;
      p0: assert property (sum < `MAX_SUM);
   end
   p1: assert property (@clk sum < `MAX_SUM);
   p2: assert property (@clk a != b);
   p3: assert #0 ($onehot(a));
endchecker

module m(input logic rst, clk, logic en, logic[7:0] in1, in2,
         in_array [20:0]);
   c1 check_outside(posedge clk, in1, in2);
   always @(posedge clk) begin
      automatic logic [7:0] v1=0;
      if (en) begin
         // v1 is automatic, so current procedural value is used
         c1 check_inside(posedge clk, in1, v1);
      end
      for (int i = 0; i < 4; i++) begin
         v1 = v1+5;
         if (i != 2) begin
            // v1 is automatic, so current procedural value is used
            c1 check_loop(posedge clk, in1, in_array[v1]);
         end
      end
   end
endmodule : m
```

In this example, there are three instantiations of `c1`: `check_outside`, `check_inside`, and `check_loop`. They have the following characteristics:

— `check_outside` is a static instantiation, while `check_inside` and `check_loop` are procedural.

— Each of the three instantiations has its own version of `sum`, which is updated at every positive clock edge, regardless of whether that instance was visited in procedural code. Even in the case of `check_loop`, there is only one instance of `sum`, and it will be updated using the sampled value of `in1`.

— Each of the three instantiations will queue an evaluation of `p0` at every posedge of the clock (according to the rules in 16.14.6), which will mature and report a violation during any time step when `sum` is not less than `MAX_SUM`, regardless of the behavior of the procedural code in module `m`.

— For checker instance `check_outside`, `p1` and `p2` are checked at every positive clock edge. For checker instance `check_inside`, `p1` and `p2` are queued to mature and be checked on any positive clock edge when `en` is true. For `check_loop`, three procedural instances of `p1` and `p2` are queued to mature on any positive clock edge. For `p1`, all three instances are identical, using the sampled value of `sum`; but for `p2`, the three instances compare the sampled value of `in1` to the sampled value of `in_array` indexed by constant `v1` values of 5, 10, and 20, respectively.

— For checker instance `check_outside`, `p3` is checked whenever `a` changes. In checker instances `check_inside` and `check_loop`, deferred assertion `p3` behaves as a procedural deferred assertion placed at the instantiation point of its checker.

## 17.4 Context inference

Context value functions (see 16.14.7) may be used as default values of formal arguments in a checker declaration. These functions enable adjusting the checker behavior depending on its instantiation context. For example:

```
// Context inference in a checker
checker check_in_context (logic test_sig,
                          event clock = $inferred_clock,
                          logic reset = $inferred_disable);
   property p(logic sig);
      ...
   endproperty
   a1: assert property (@clock disable iff (reset) p(test_sig));
   c1: cover property (@clock !reset throughout !test_sig ##1 test_sig);
endchecker : check_in_context

module m(logic rst);
   wire clk;
   logic a, en;
   wire b = a && en;
   // No context inference
   check_in_context my_check1(.test_sig(b), .clock(clk), .reset(rst));
```

```
      always @(posedge clk) begin
         a <= ...;
         if (en) begin
            ...
            // inferred from context:
            // .clock(posedge clk)
            // .reset(1'b0)
            check_in_context my_check2(a);
         end
         en <= ...;
      end
   endmodule : m
```

In the preceding example the default values of `clock` and `reset` in `check_in_context` are taken from the instantiation context. In the instantiation `my_check1` all formal arguments are provided explicitly. In the instantiation `my_check2` all optional arguments are passed their default value: the clock is inferred from the clock of the always procedure of the module `m`, the disable condition is inferred to be `1'b0`.

## 17.5 Checker procedures

The following procedures are allowed inside a checker body:

— **`initial`** procedure
— **`always`** procedure
— **`final`** procedure

An **`initial`** procedure in a checker body may contain **`let`** declarations, immediate, deferred, and concurrent assertions, and a procedural timing control statement using an event control only.

The following forms of always procedures are allowed in checkers: **`always_comb`**, **`always_latch`**, and **`always_ff`**. Checker always procedures may contain the following statements:

— Blocking assignments (see 10.4.1; **`always_comb`** and **`always_latch`** procedures only)
— Nonblocking assignments (see 10.4.2)
— Selection statements (see 12.4 and 12.5)
— Loop statements (see 12.7)
— Timing event control (see 9.4.2; **`always_ff`** procedure only)
— Subroutine calls (see Clause 13)
— Immediate, deferred, and concurrent assertions
— **`let`** declarations

Except for the variables used in the event control, all other expressions in **`always_ff`** procedures are sampled (see 16.5.1). It follows from this rule that the expressions in immediate and deferred assertions instantiated in this procedure are also sampled. Expressions in **`always_comb`** and **`always_latch`** procedures are not implicitly sampled and the assignments appearing in these procedures use the current values of their expressions. For example:

```
   checker check(logic a, b, c, clk, rst);
      logic x, y, z, v, t;

      assign x = a;           // current value of a

      always_ff @(posedge clk or negedge rst) // current values of clk and rst
      begin
         a1: assert (b);      // sampled value of b
         if (rst)             // current value of rst
            z <= b;           // sampled value of b
         else z <= !c;        // sampled value of c
      end

      always_comb begin
         a2: assert (b);      // current value of b
         if (a)               // current value of a
            v = b;            // current value of b
         else v = !b;         // current value of b
      end

      always_latch begin
         a3: assert (b);      // current value of b
         if (clk)             // current value of clk
            t <= b;           // current value of b
      end
      // ...
   endchecker : check
```

The following example illustrates clock inference for checker procedures, following the rules in 16.14.6.

```
checker clocking_example (logic sig1, sig2, default_clk, rst,
                          event e1, e2, e3 );

   bit local_sig;
   default clocking @(posedge default_clk); endclocking

   always_ff @(e1) begin: p1_block
      p1a: assert property (sig1 == sig2);
      p1b: assert property (@(e1) (sig1 == sig2));
   end
   always_ff @(e2 or e3) begin: p2_block
      local_sig <= rst;
      p2a: assert property (sig1 == sig2);
      p2b: assert property (@(e2) (sig1 == sig2));
   end
   always_ff @(rst or e3) begin: p3_block
      local_sig <= rst;
      p3a: assert property (sig1 == sig2);
      p3b: assert property (@(e3) (sig1 == sig2));
   end

   ...
endchecker
...

clocking_example c1 (s1, s2, default_clk, rst,
                     posedge clk1 or posedge clk2,
                     posedge clk1,
                     negedge rst);
```

In instance `c1` of `clocking_example`, the assertions will be clocked as follows:

— Assertion `p1a` will be clocked by **posedge** `default_clk`. This is because after the substitution of the actual argument **posedge** `clk1` **or posedge** `clk2` for the formal argument `e1`, it does not satisfy the clock inference conditions in 16.14.6, particularly condition (b). If clocking based on `e1` is desired, it must be done explicitly as in property `p1b`.

— Assertion p2a will be clocked by **posedge** clk1. This is because the event control of p2_block satisfies the conditions in 16.14.6, including condition (c 2), after the formal arguments are substituted with the actual arguments. Thus assertions p2a and p2b are equivalent.

— Assertion p3a will be clocked by **posedge** default_clk. This is because the event control of p3_block does not satisfy the conditions in 16.14.6, particularly condition (c 2). If clocking based on e3 is desired, it must be done explicitly as in property p3b.

A **final** procedure may be specified within a checker in the same manner as in a module (see 9.2.3). This allows for the checker to check conditions with immediate assertions or print out statistics at the end of simulation. The operation of the **final** procedure is independent of the instantiation context of the checker that contains it. It will be executed once at the end of simulation for every instantiation of that checker. There is no implied ordering in the execution of multiple **final** procedures. A **final** procedure within a checker may include any construct allowed in a non-checker **final** procedure.

## 17.6 Covergroups in checkers

One or more **covergroup** declarations or instances (see 19.3) are permitted within a checker. These declarations and instances shall not appear in any procedural block in the checker. A **covergroup** may reference any variable visible in its scope, including checker formal arguments and checker variables. However, it shall be an error if a formal argument referenced by a **covergroup** has a **const** actual argument. For example:

```
checker my_check(logic clk, active);
   bit active_d1 = 1'b0;

   always_ff @(posedge clk) begin
      active_d1 <= active;
   end

   covergroup cg_active @(posedge clk);
      cp_active : coverpoint active
      {
         bins idle = { 1'b0 };
         bins active = { 1'b1 };
      }
      cp_active_d1 : coverpoint active_d1
      {
         bins idle = { 1'b0 };
         bins active = { 1'b1 };
      }
      option.per_instance = 1;
   endgroup
   cg_active cg_active_1 = new();
endchecker : my_check
```

A covergroup may also be triggered by a procedural call to its sample() method (see 19.8). The following examples show how the sample() method may be called from a sequence match item to trigger a covergroup.

```
checker op_test (logic clk, vld_1, vld_2, logic [3:0] opcode);
   bit [3:0] opcode_d1;

   always_ff @(posedge clk) opcode_d1 <= opcode;

   covergroup cg_op;
      cp_op : coverpoint opcode_d1;
   endgroup: cg_op
   cg_op cg_op_1 = new();

   sequence op_accept;
      @(posedge clk) vld_1 ##1 (vld_2, cg_op_1.sample());
   endsequence
   cover property (op_accept);
endchecker
```

In this example, the coverpoint `cp_op` refers to the checker variable `opcode_d1` directly. It is triggered by a call to the default `sample()` method from a sequence match item. This function call occurs in the Reactive region, while nonblocking assignments to checker variables will occur in the Re-NBA region. As a result, the covergroup will sample the old value of the checker variable `opcode_d1`.

It is also possible to define a custom `sample()` method for a covergroup (see 19.8.1). The following is an example of this:

```
   checker op_test (logic clk, vld_1, vld_2, logic [3:0] opcode);
      bit [3:0] opcode_d1;

      always_ff @(posedge clk) opcode_d1 <= opcode;

      covergroup cg_op with function sample(bit [3:0] opcode_d1);
         cp_op : coverpoint opcode_d1;
      endgroup: cg_op
      cg_op cg_op_1 = new();

      sequence op_accept;
         @(posedge clk) vld_1 ##1 (vld_2, cg_op_1.sample(opcode_d1));
      endsequence
      cover property (op_accept);
   endchecker
```

In this example, a custom `sample()` method has been defined for the covergroup `cg_op`, and the coverpoint `cp_op` references the formal argument of the custom `sample()` method. This custom method will be called in the Reactive region upon a sequence match, but the sampled value of the sequential checker variable `opcode_d1` will be passed to the `sample()` function. As a result, the covergroup will sample the value from the Preponed region.

## 17.7 Checker variables

Variables may be defined in checkers, but defining nets in the checker body shall be illegal. All variables defined in a checker body shall have static lifetimes (see 17.2). The variables defined in the checker body are referred to as *checker variables*. The following example illustrates checker variable usage:

```
   checker counter_model(logic flag);
      bit [2:0] counter = '0;
      always_ff @$global_clock
         counter <= counter + 1'b1;
      assert property (@$global_clock counter == 0 |-> flag);
   endchecker : counter_model
```

Checker variables may have an optional **rand** qualifier. In this case, they are called *free variables*; free variables may behave nondeterministically.

Formal analysis tools shall take into account all possible values of the free checker variables imposed by the assumptions and assignments (see 17.7.1). Simulators shall assign random values to the free variables as explained in 17.7.2.

The following example shows how free variables can be used for modeling for formal verification:

```
checker observer_model(bit valid, reset);
   default clocking @$global_clock; endclocking
   rand bit flag;

   m1: assume property (reset |=> !flag);
   m2: assume property (!reset && flag |=> flag);
   m3: assume property ($rising_gclk(flag) |-> valid);
endchecker : observer_model
```

In this example, the following constraints are imposed on the free variable `flag`:

— If it is high, it remains high as long as there is no `reset`.

— If there is a `reset`, it becomes low at the next tick of the clock.

— It may rise only when `valid` is high.

Although the behavior of the free variable `flag` has been restricted by the assumptions `m1`, `m2`, and `m3`, it is still nondeterministic because it does not have to rise when `valid` is high. Figure 17-1 shows two possible legal behaviors of this variable given the same behaviors of `reset` and `valid`. Formal analysis tools shall take all possible legal behaviors of `flag` into account. Simulators shall assign random values to the variable `flag` as explained in 17.7.2.

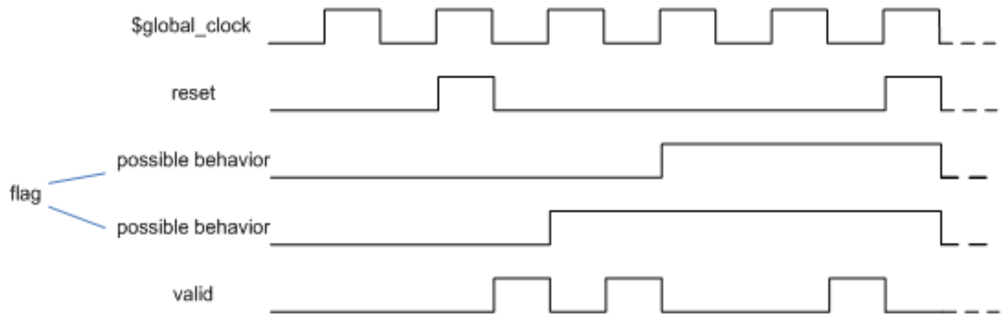


**Figure 17-1—Nondeterministic free checker variable**

The following example shows how free variables may be used to implement a nondeterministic choice:

```
// a may assume values 3 and 5 only
rand bit r;
let a = r ? 3'd3 : 3'd5;
```

A free variable declaration may have a **`const`** qualifier. If a constant free variable is initialized, it retains its initial value forever. An uninitialized constant free variable has a nondeterministic value at the initialization, and this value does not change. The following examples demonstrate the usage of constant free checker variables.

Formal analysis tools shall take into account any possible values of a constant free checker variable consistent with the imposed assumptions. Simulators shall assign a random constant value to a constant free variable as explained in 17.7.2.

*Examples:*

Reasoning about a representative bit:

```
checker reason_about_one_bit(bit [63:0] data1, bit [63:0] data2,
                             event clock);
   rand const bit [5:0] idx;
   a1: assert property (@clock data1[idx] == data2[idx]);
endchecker : reason_about_one_bit
```

In this example the assertion `a1` states that any fixed bit of `data1` has the same value as the corresponding bit of `data2`. Therefore, the checker `reason_about_one_bit` is equivalent in formal verification to the following checker (these two checkers are not equivalent in simulation):

```
checker reason_about_all_bit(bit [63:0] data1, bit [63:0] data2,
                             event clock);
   a1: assert property (@clock data1 == data2);
endchecker : reason_about_all_bit
```

The second realization of the checker compares two 64-bit values while the first one compares only 1-bit values, for every possible index. The first version may be more efficient for some formal tools.

Data integrity checking:

```
// If start_ev is asserted then the value of out_data at the next assertion
// of end_ev has to be equal to the current value of in_data at start_ev.
//
// It is assumed that in_data and out_data have the same size
checker data_legal(start_ev, end_ev, in_data, out_data);
   rand const bit [$bits(in_data)-1:0] mem_data;
   sequence transaction;
      start_ev && (in_data == mem_data) ##1 end_ev[->1];
   endsequence
   a1: assert property (@clock transaction |-> out_data == mem_data);
endchecker : data_legal
```

Since `mem_data` is a constant free variable, if `in_data` is equal to `mem_data` at the beginning of the transaction, then `mem_data` records that value and keeps it throughout the trace. In particular, at the end of the transaction, `mem_data` still holds that value and the assertion checks that it is equal to `out_data`. Moreover, `mem_data` was initialized with a nondeterministic value; it follows that for every value of `in_data`, there exists a computation in which `mem_data` is equal to that value of `in_data`, which in turn implies that the corresponding legality of data transfer through that transaction is being checked for formal verification. In simulation `mem_data` will be randomly initialized (see 17.7.2), and it will only be checked that if at the transaction beginning `in_data` equals to `mem_data` then at the transaction end `out_data` will have the same value as `in_data` at the beginning of the transaction.

The latter example may be rewritten for formal verification using local variables instead of constant free variables (see 16.10; these implementations are not equivalent in simulation):

```
// If start_ev is asserted then the value of in_data has to be
// equal to the value of out_data at the next assertion of end_ev
//
// It is assumed that in_data and out_data have the same size
checker data_legal_with_loc(start_ev, end_ev, in_data, out_data);
   sequence transaction (loc_var);
      (start_ev, loc_var = in_data) ##1 end_ev[->1];
   endsequence
   property data_legal;
      bit [$bits(in_data)-1:0] mem_data;
      transaction(mem_data) |-> out_data == mem_data;
   endproperty
   a1: assert property (@clock data_legal);
endchecker : data_legal_with_loc
```

There is a difference between a constant and a non-constant free variable: a constant free variable does not change its value, while a non-constant free variable can assume a new value any time. If a non-constant free variable has been initialized but is never assigned then it can assume any value at any time step in formal verification, or be randomized in subsequent time steps in simulation (see 17.7.2), except the first one where its value is defined by the initialization. Consider the following declaration:

```
   rand bit a = 1'b0, b;
```

The free variable a has initial value 0, but in other time steps its value may change. The free checker variable b may assume any value 0 or 1 at any time (in formal verification or randomized in simulation), as opposed to an uninitialized constant free checker variable, which keeps one specific value.

### 17.7.1 Checker variable assignments

Checker variables may be assigned using blocking and nonblocking procedural assignments, or non-procedural continuous assignments.

The following rules and restrictions apply:

- In **always_ff** procedures only nonblocking assignments are allowed.
- Referencing a checker variable using its hierarchical name in assignments (see 23.6) shall be illegal. For example:

```
checker check(...)
   bit a;
   ...
endchecker

module m(...)
   ...
   check my_check(...);
   ...
   wire x = my_check.a;    // Illegal
   bit y;
   ...
   always @(posedge clk) begin
      my_check.a = y;      // Illegal
      ...
   end
   ...
endmodule
```

- Continuous assignments and blocking procedural assignments to free checker variables shall be illegal.

```
checker check1(bit a, b, event clk, ...);
   rand bit x, y, z, v;
   ...
```

```
      assign x = a & b;        // Illegal
      always_comb
         y = a & b;            // Illegal
      always_ff @clk
         z <= a & b;           // OK
   endchecker : check1
```

— A checker variable may not be assigned in an **initial** procedure, but may be initialized in its declaration. For example:

```
   bit v;
   initial v = 1'b0;          // Illegal
   bit w = 1'b0;              // OK
```

— The right-hand side of a checker variable assignment may contain the sequence method `triggered` (see 16.13.6).

— The left-hand side of a nonblocking assignment may contain a free checker variable. The following example illustrates usage of free variable assignments.

```
   // Toggling variable:
   // a may have either 0101... or 1010... pattern
   rand bit a;
   always_ff @clk a <= !a;
```

### 17.7.2 Checker variable randomization with assumptions

Checker **assume** statements are used to describe assumptions that may be made about the values of variables. They may be used by simulators to constrain the random generation of free checker variable values or by formal tools to constrain the formal computation. As with normal **assume** statements, checker **assume** statements shall also be checked for violation during simulation.

Assume-based checker variable randomization is the process of periodically solving a set of properties appearing in **assume** statements (called an *assume set*) to find satisfying values for the free checker variables, and updating those variables with the newfound values. Unlike class-based constrained random generation, solving is triggered by any of the clock events of the properties in the assume set (called an *assume set clock event*) rather than by an explicit procedural call [e.g., there is no `randomize()` for checkers]. Once updated with solution values, free checker variables shall remain constant until the next assume set clock event or the end of the time step, whichever comes first.

All non-**const** free checker variables are treated as either active or inactive for assume-based randomization, in the same way as **rand** variables for class-based constrained random generation (see 17.9), but without an explicit control facility [such as `rand_mode()`]. All other variables (such as non-free checker variables and checker formals) are always treated as inactive. Any free checker variables that appear on the left-hand side of a checker variable assignment (see 17.7.1) are inactive; all other free checker variables are active. Free checker variables are active or inactive for each singular element of the variable. For example, a packed array or structure is active or inactive monolithically, whereas the elements of an unpacked array or structure are separately active or inactive.

All free checker variables, both **const** and non-**const**, active and inactive, are initialized with unconstrained random values unless explicitly initialized in their declaration.

Each checker instance has one and only one assume set, which may be empty. Like checker procedures and variables, checker assume sets are considered to exist at every time step, regardless of whether the checker instance is static or procedural (see 17.3).

The assume set of a checker instance is formed from the checker **assume** statements and child checker **assume** statements. Any of these **assume** statements that references a formal whose actual argument contains any subexpression that is a **const** cast or automatic value (see 17.3) is excluded from the assume set. This restriction allows a single copy of the assume set to exist for each instantiation that is valid for the entire simulation, as described in 17.3. Among the remaining **assume** statements, those that reference active free variables of the checker are included in the assume set. For example:

```
module my_mod();
   bit mclk, v1, v2;
   checker c1(bit fclk, bit a, bit b);
      default clocking @ (posedge fclk); endclocking
      checker c2(bit bclk, bit x, bit y);
         default clocking @ (posedge bclk); endclocking
         rand bit m, n;
         u1: assume property (f1(x,m));
         u2: assume property (f2(y,n));
      endchecker
      rand bit q, r;
      c2 B1(fclk, q+r, r);
      always_ff @ (posedge fclk)
         r <= a || q; // assignment makes r inactive
      u3: assume property (f3(a, q));
      u4: assume property (f4(b, r));
   endchecker
   ...
   c1 F1(mclk, v1, const'(v2));
endmodule
```

The assume set of `F1` consists of `F1.u3` and `F1.B1.u1`. The property `F1.B1.u1` is included because it references the formal `x`, whose actual expression `q+r` involves an active free checker variable. `F1.u4` is excluded because it references the formal `b`, which is associated with the **const** cast actual `v2`. `F1.B1.u2` is excluded because the only formal referenced is `y`, which is not associated with an active free variable actual (the actual `r` is inactive). However, checker instance `F1.B1` has its own assume set, which includes `u2` as well as `u1`; neither of those **assume** statements involve formals with **const** cast or automatic actuals.

When a solution attempt is made on an assume set, values shall be sought for all active checker variables such that, together with the inactive variables and state, none of the assumptions will fail in that time step. If a set of such values is found, the solution attempt is successful. Otherwise, any values may be chosen for the active variables and the solution attempt is unsuccessful. There is no requirement that a solution be found if it exists or that “dead end” states (states where no solution exists) be avoided. For example,

```
u_deadend: assume property (@(posedge clk) x |=> ##5 1'b0);
```

If the value 1 is chosen for `x`, the property would not fail in the current time step; however, it would inevitably fail six clock cycles later. Such an inevitable future failure is called a *dead end*. Despite the dead end, selecting 1 for `x` is considered a successful solution attempt.

Empty assume sets shall be considered to have an implicit assume set clock event in every time step before the Observed region. Active variables in checkers with empty assume sets are called *implicitly clocked* active free variables; those with nonempty assume sets are *explicitly clocked*. Implicitly clocked active variables may be updated with unconstrained random values at every time step. Once updated, the variables stay constant until the end of the time step.

Active variables that do not appear in any property in a nonempty assume set are unconstrained but explicitly clocked. They may be updated with random values at every assume set clock event.

When an implementation is about to begin the Observed region, it shall solve for all the active free checker variables using sampled values of all other variables. A sampled value of an active checker variable is defined as its current value (see 16.5.1). Note that checker procedures and properties execute in the Reactive and Observed regions (see 17.7.3), and so have the new values available.

When a solution attempt is unsuccessful, any resulting assumption failure(s) do not occur until an unsatisfied property is clocked and checked in the Observed region.

### 17.7.3 Scheduling semantics

Statements and constructs within a checker that are sensitive to changes (e.g., clocking events, continuous assignments) and all blocking statements are scheduled in the Reactive region (similarly to programs, see 24.3.1). The nonblocking assignments of checker variables schedule their updates in the Re-NBA region. The Re-NBA region is processed after the Reactive and Re-Inactive regions have been emptied of events (see 4.2). These scheduling rules make possible assignment of sequence end point values to checker variables. For example:

```
checker my_check(...);
   ...
   sequence s; ...; endsequence
   always_ff @clk a <= s.triggered;
endchecker
```

For every transition of signal `clk`, the simulator will update the variable `a` in the Re-NBA region with the value of `s.triggered` captured in the Reactive region. Had the checker captured the value of `s.triggered` in the Active region, `a` would always be assigned `1'b0`, since `s.triggered` is evaluated in the Observed region, and the preceding code would be meaningless.

If a free variable is referenced in several assumptions, and some of these assumptions are governed by a clock changing in the Active region, and others by a clock changing in the Reactive region, assertions and assumptions referencing this free variable can be executed twice in the same time step. This can result in an undesired behavior in simulation. For example:

```
checker check(bit clk1); // clk1 assigned in the Active region
   rand bit v, w;
   assign clk2 = clk1;
   m1: assume property (@clk1 !(v && w));
   m2: assume property (@clk2 v || w);
   a1: assert property (@clk1 v != w);
   // ...
endchecker : check
```

After `clk1` changes in the Active region, assumption `m1` and assertion `a1` are executed in the Observed region. Both free variables `v` and `w` may be assigned the value `0`, which causes assertion `a1` to fail. Then `clk2` changes in the Reactive region, and the simulator enters the Observed region again, where assumption `m2` is evaluated. As a result, new values are assigned to `v` and `w` (see 17.7.2), and these values can become `0` and `1`, respectively. The free variables `v`  and `w` then keep their values until the next tick of `clk1`.

Concurrent assertions have invariant scheduling semantics, whether present in checker code or design code.

## 17.8 Functions in checkers

The formal arguments and internal variables of functions used in checkers shall not be declared as free variables. However, free variables are allowed to be passed in as actual arguments to a function.

Expressions at the right-hand side of checker variable assignments are allowed to include function calls with the same restrictions that are imposed on function calls in concurrent assertions (see 16.6):

- Functions that appear in expressions shall not contain **output** or **ref** arguments (**const ref** is allowed).
- Functions shall be automatic (or preserve no state information) and have no side effects.

See an example of a function used in a checker in 17.9.

## 17.9 Complex checker example

The checkers in the following examples make sure that the expression is true in a window delimited by `start_event` and `end_event`. When `start_event` and `end_event` are Boolean, the checker may be implemented as shown in Example 1.

*Example 1:*

```
typedef enum { cover_none, cover_all } coverage_level;
checker assert_window1 (
   logic test_expr,      // Expression to be true in the window
   untyped start_event, // Window opens at the completion of the start_event
   untyped end_event,   // Window closes at the completion of the end_event
   event clock = $inferred_clock,
   logic reset = $inferred_disable,
   string error_msg = "violation",
   coverage_level clevel = cover_all  // This argument should be bound to an
                                      // elaboration time constant expression
);
   default clocking @clock; endclocking
   default disable iff reset;
   bit window = 1'b0, next_window = 1'b1;

   // Compute next value of window
   always_comb begin
      if (reset || window && end_event)
         next_window = 1'b0;
      else if (!window && start_event)
         next_window = 1'b1;
      else
         next_window = window;
   end

   always_ff @clock
      window <= next_window;

   property p_window;
      start_event && !window |=> test_expr[+] ##0 end_event;
   endproperty

   a_window: assert property (p_window) else $error(error_msg);

   generate if (clevel != cover_none) begin : cover_b
      cover_window_open: cover property (start_event && !window)
      $display("window_open covered");
      cover_window: cover property (
         start_event && !window
         ##1 (!end_event && window) [*]
         ##1 end_event && window
      ) $display("window covered");
   end : cover_b
   endgenerate
endchecker : assert_window1
```

If start_event and end_event may be arbitrary sequences, and not necessary Boolean values, the checker needs to be implemented differently, as shown in Example 2. This case requires a different implementation because the reset of the triggered status of a sequence does not create an event (see 9.4.4), and therefore a sequence triggered method should not be used in the right-hand side of a continuous assignment or of an assignment in an **always_comb** procedure.

*Example 2:*

```
typedef enum { cover_none, cover_all } coverage_level;
checker assert_window2 (
   logic test_expr,      // Expression to be true in the window
   sequence start_event, // Window opens at the completion of the start_event
   sequence end_event,   // Window closes at the completion of the end_event
   event clock = $inferred_clock,
   logic reset = $inferred_disable,
   string error_msg = "violation",
   coverage_level clevel = cover_all  // This argument should be bound to an
                                      // elaboration time constant expression
);
   default clocking @clock; endclocking
   default disable iff reset;
   bit window = 0;
   let start_flag = start_event.triggered;
   let end_flag = end_event.triggered;

   // Compute next value of window
   function bit next_window (bit win);
      if (reset || win && end_flag)
         return 1'b0;
      if (!win && start_flag)
         return 1'b1;
      return win;
   endfunction

   always_ff @clock
      window <= next_window(window);

   property p_window;
      start_flag && !window |=> test_expr[+] ##0 end_flag;
   endproperty

   a_window: assert property (p_window) else $error(error_msg);

   generate if (clevel != cover_none) begin : cover_b
      cover_window_open: cover property (start_flag && !window)
      $display("window_open covered");
      cover_window: cover property (
         start_flag && !window
         ##1 (!end_flag && window) [*]
         ##1 end_flag && window
         ) $display("window covered");
      end : cover_b
   endgenerate
endchecker : assert_window2
```

# 18. Constrained random value generation

## 18.1 General

This clause describes the following:

— Random variables
— Constraint blocks
— Randomization methods
— Disabling randomization
— Controlling constraints
— Scope variable randomization
— Seeding the random number generator (RNG)
— Random weighted case statements
— Random sequence generation

## 18.2 Overview

Constraint-driven test generation allows users to automatically generate tests for functional verification. Random testing can be more effective than a traditional, directed testing approach. By specifying constraints, one can easily create tests that can find hard-to-reach corner cases. SystemVerilog allows users to specify constraints in a compact, declarative way. The constraints are then processed by a solver that generates random values that meet the constraints.

The random constraints are typically specified on top of an object-oriented data abstraction that models the data to be randomized as objects that contain random variables and user-defined constraints. The constraints determine the legal values that can be assigned to the random variables. Objects are ideal for representing complex aggregate data types and protocols such as Ethernet packets.

Subclause 18.3 provides an overview of object-based randomization and constraint programming. The rest of this clause provides detailed information on random variables, constraint blocks, and the mechanisms used to manipulate them.

## 18.3 Concepts and usage

This subclause introduces the basic concepts and uses for generating random stimulus within objects. SystemVerilog uses an object-oriented method for assigning random values to the member variables of an object, subject to user-defined constraints. For example:

```
class Bus;
   rand bit[15:0] addr;
   rand bit[31:0] data;

   constraint word_align {addr[1:0] == 2'b0;}
endclass
```

The `Bus` class models a simplified bus with two random variables, `addr` and `data`, representing the address and data values on a bus. The `word_align` constraint declares that the random values for `addr` must be such that `addr` is word-aligned (the low-order 2 bits are 0).

The `randomize()` method is called to generate new random values for a bus object:

```
Bus bus = new;

repeat (50) begin
   if ( bus.randomize() == 1 )
      $display ("addr = %16h data = %h\n", bus.addr, bus.data);
   else
      $display ("Randomization failed.\n");
end
```

Calling `randomize()` causes new values to be selected for all of the random variables in an object so that all of the constraints are true (satisfied). In the preceding program test, a `bus` object is created and then randomized 50 times. The result of each randomization is checked for success. If the randomization succeeds, the new random values for `addr` and `data` are printed; if the randomization fails, an error message is printed. In this example, only the `addr` value is constrained, while the `data` value is unconstrained. Unconstrained variables are assigned any value in their declared range.

Constraint programming is a powerful method that lets users build generic, reusable objects that can later be extended or constrained to perform specific functions. The approach differs from both traditional procedural and object-oriented programming, as illustrated in this example that extends the `Bus` class:

```
typedef enum {low, mid, high} AddrType;

class MyBus extends Bus;
   rand AddrType atype;
   constraint addr_range
   {
      (atype == low ) -> addr inside {  [0 : 15] };
      (atype == mid ) -> addr inside { [16 : 127]};
      (atype == high) -> addr inside {[128 : 255]};
   }
endclass
```

The `MyBus` class inherits all of the random variables and constraints of the `Bus` class and adds a random variable called `atype` that is used to control the address range using another constraint. The `addr_range` constraint uses implication to select one of three range constraints depending on the random value of `atype`. When a `MyBus` object is randomized, values for `addr`, `data`, and `atype` are computed so that all of the constraints are satisfied. Using inheritance to build layered constraint systems enables the development of general-purpose models that can be constrained to perform application-specific functions.

Objects can be further constrained using the `randomize()` **`with`** construct, which declares additional constraints in-line with the call to `randomize()`:

```
task exercise_bus (MyBus bus);
   int res;

   // EXAMPLE 1: restrict to low addresses
   res = bus.randomize() with {atype == low;};

   // EXAMPLE 2: restrict to address between 10 and 20
   res = bus.randomize() with {10 <= addr && addr <= 20;};

   // EXAMPLE 3: restrict data values to powers-of-two
   res = bus.randomize() with {(data & (data - 1)) == 0;};
endtask
```

This example illustrates several important properties of constraints, as follows:

- — Constraints can be any SystemVerilog expression with variables and constants of integral type (e.g., **bit**, **reg**, **logic**, **integer**, **enum**, **packed struct**).
- — The constraint solver shall be able to handle a wide spectrum of equations, such as algebraic factoring, complex Boolean expressions, and mixed integer and bit expressions. In the previous example, the power-of-two constraint was expressed arithmetically. It could have also been defined with expressions using a shift operator; for example, `1 << n`, where `n` is a 5-bit random variable.
- — If a solution exists, the constraint solver shall find it. The solver can fail only when the problem is over-constrained and there is no combination of random values that satisfy the constraints.
- — Constraints interact bidirectionally. In this example, the value chosen for `addr` depends on `atype` and how it is constrained, and the value chosen for `atype` depends on `addr` and how it is constrained. All expression operators are treated bidirectionally, including the implication operator (`->`).
- — Constraints support only 2-state values. The 4-state values (`X` or `Z`) or 4-state operators (e.g., `===`, `!==` ) are illegal and shall result in an error.
- — For each active random variable of **enum** type, the solver shall select a value from the set of named constants defined by the corresponding **enum**. The solver shall not assign a value to a random variable of **enum** type that lies outside its associated named constant set, even if the value can be successfully cast to the enumerated type. Note that state variables of **enum** type may contain values outside the set of named **enum** constants, which may still allow a valid solution.

Sometimes it is desirable to disable constraints on random variables. For example, to deliberately generate an illegal address (nonword-aligned):

```
task exercise_illegal(MyBus bus, int cycles);
   int res;

   // Disable word alignment constraint.
   bus.word_align.constraint_mode(0);

   repeat (cycles) begin

   // CASE 1: restrict to small addresses.
   res = bus.randomize() with {addr[0] || addr[1];};
      ...
   end

   // Reenable word alignment constraint
   bus.word_align.constraint_mode(1);
endtask
```

The `constraint_mode()` method can be used to enable or disable any named constraint block in an object. In this example, the word-alignment constraint is disabled, and the object is then randomized with additional constraints forcing the low-order address bits to be nonzero (and thus unaligned).

The ability to enable or disable constraints allows users to design constraint hierarchies. In these hierarchies, the lowest level constraints can represent physical limits grouped by common properties into named constraint blocks, which can be independently enabled or disabled.

Similarly, the `rand_mode()` method can be used to enable or disable any random variable. When a random variable is disabled, it behaves in exactly the same way as other nonrandom variables.

Occasionally, it is desirable to perform operations immediately before or after randomization. That is accomplished via two built-in methods, `pre_randomize()` and `post_randomize()`, which are automatically called before and after randomization. These methods can be overridden with the desired functionality:

```
class XYPair;
   rand integer x, y;
endclass

class MyXYPair extends XYPair
   function void pre_randomize();
      super.pre_randomize();
      $display("Before randomize x=%0d, y=%0d", x, y);
   endfunction

   function void post_randomize();
      super.post_randomize();
      $display("After randomize x=%0d, y=%0d", x, y);
   endfunction
endclass
```

By default, `pre_randomize()` and `post_randomize()` call their overridden base class methods. When `pre_randomize()` or `post_randomize()` are overridden, care must be taken to invoke the base class's methods, unless the class is a base class (has no base class). Otherwise, the base class methods shall not be called.

The random stimulus generation capabilities and the object-oriented constraint-based verification methodology enable users to quickly develop tests that cover complex functionality and better assure design correctness.

## 18.4 Random variables

Class variables can be declared random using the **rand** and **randc** type-modifier keywords.

The syntax to declare a random variable in a class is as follows in Syntax 18-1.

```
class_property ::=                                                    // from A.1.9
     { property_qualifier } data_declaration
property_qualifier[8] ::=
       random_qualifier
     | class_item_qualifier
random_qualifier[8] ::=
       rand
     | randc
```

8) In any one declaration, only one of **protected** or **local** is allowed, only one of **rand** or **randc** is allowed, and **static** and/or **virtual** can appear only once.

*Syntax 18-1—Random variable declaration syntax (excerpt from Annex A)*

— The solver can randomize singular variables of any integral type.
— Arrays can be declared **rand** or **randc**, in which case all of their member elements are treated as **rand** or **randc**.

— Individual array elements can be constrained, in which case the index expression may include iterative constraint loop variables, constants, and state variables.

— Dynamic arrays, associative arrays and queues can be declared **rand** or **randc**. All of the elements in the array are randomized, overwriting any previous data.

— The size of a dynamic array or queue declared as **rand** or **randc** can also be constrained. In that case, the array shall be resized according to the size constraint, and then all the array elements shall be randomized. The array size constraint is declared using the `size` method. For example:

```
rand bit [7:0] len;
rand integer data[];
constraint db { data.size == len; }
```

The variable `len` is declared to be 8 bits wide. The randomizer computes a random value for the `len` variable in the 8-bit range of 0 to 255 and then randomizes the first `len` elements of the data array.

When a dynamic array is resized by `randomize()`, the resized array is initialized (see 7.5.1) with the original array. When a queue is resized by `randomize()`, elements are inserted or deleted (see 7.10.2.2 and 7.10.2.3) at the back (i.e., right side) of the queue as necessary to produce the new queue size; any new elements inserted take on the default value of the element type. That is, the resize grows or shrinks the array. This is significant for a dynamic array or queue of class handles. `randomize()` does not allocate any class objects. Up to the new size, existing class objects are retained and their content randomized. If the new size is greater than the original size, each of the additional elements has a **null** value requiring no randomization.

In resizing a dynamic array or queue by `randomize()` or **new**, the `rand_mode` of each retained element is preserved and the `rand_mode` of each new element is set to active.

If a dynamic array's size is not constrained, then the array shall not be resized and all the array elements shall be randomized.

— An object handle can be declared **rand**, in which case all of that object's variables and constraints are solved concurrently with the variables and constraints of the object that contains the handle. Randomization shall not modify the actual object handle. Object handles shall not be declared **randc**.

— An unpacked structure can be declared **rand**, in which case all of that structure's random members are solved concurrently using one of the rules listed in this subclause. Unpacked structures shall not be declared **randc**. A member of an unpacked structure can be made random by having a **rand** or **randc** modifier in the declaration of its type.

For example:

```
class packet;
typedef struct {
   randc int addr = 1 + constant;
   int crc;
   rand byte data [] = {1,2,3,4};
} header;
rand header h1;
endclass
packet p1=new;
```

— Unpacked unions shall not be declared **rand** or **randc**.

— Packed tagged unions shall not be declared **rand** or **randc**.

— A packed untagged union can be declared **rand** or **randc**, in which case that union is treated as an integral type. Members of packed untagged unions shall not have a random modifier.

— A packed structure can be declared **rand** or **randc**, in which case that structure is treated as an integral type. Members of packed structures shall not have a random modifier.

— If a **rand** variable of packed structure or packed untagged union type has a member of **enum** type, the rules in 18.3 restricting the random values of an **enum** variable shall not apply to that member.

For example:

```
typedef enum bit [1:0] { A=2'b00, B=2'b11 } ab_e;
typedef struct packed {
   ab_e ValidAB;
} VStructEnum;
typedef union packed {
   ab_e ValidAB;
} VUnionEnum;
```

When randomizing a variable of type `ab_e`, the solver can only select a random value of `2'b00` or `2'b11` (`A` or `B`, respectively). However, when randomizing a variable of type `VStructEnum` or `VUnionEnum`, the solver can select `2'b00`, `2'b01`, `2'b10` or `2'b11`.

### 18.4.1 Rand modifier

Variables declared with the **rand** keyword are standard random variables. Their values are uniformly distributed over their range. For example:

```
rand bit [7:0] y;
```

This is an 8-bit unsigned integer with a range of 0 to 255. If unconstrained, this variable shall be assigned any value in the range of 0 to 255 with equal probability. In this example, the probability of the same value repeating on successive calls to `randomize()` is 1/256.

### 18.4.2 Randc modifier

Variables declared with the **randc** keyword are random-cyclic variables that cycle through all the values in a random permutation of their declared range.

To understand **randc**, consider a 2-bit random variable `y`:

```
randc bit [1:0] y;
```

The variable `y` can take on the values 0, 1, 2, and 3 (range of 0 to 3). `randomize()` computes an initial random permutation of the range values of `y` and then returns those values in order on successive calls. After it returns the last element of a permutation, it repeats the process by computing a new random permutation.

The basic idea is that **randc** randomly iterates over all the values in the range and that no value is repeated within an iteration. When the iteration finishes, a new iteration automatically starts (see Figure 18-1).

The permutation sequence for any given **randc** variable is recomputed whenever the constraints change on that variable or when none of the remaining values in the permutation can satisfy the constraints. The permutation sequence shall contain only 2-state values.

To reduce memory requirements, implementations may impose a limit on the maximum size of a **randc** variable, but it shall be no less than 8 bits.

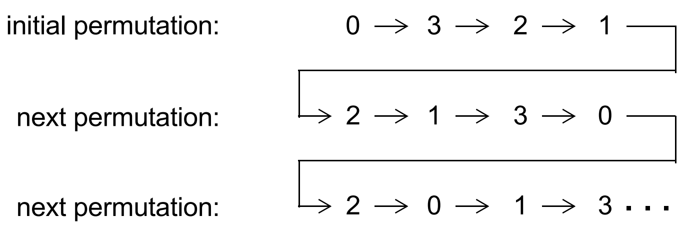


**Figure 18-1—Example of randc**

The semantics of random-cyclical variables requires that they be solved before other random variables. A set of constraints that includes both **`rand`** and **`randc`** variables shall be solved so that the **`randc`** variables are solved first, and this can sometimes cause `randomize()` to fail.

If a random variable is declared as **`static`**, the randc state of the variable shall also be static. Thus `randomize()` chooses the next cyclic value (from a single sequence) when the variable is randomized through any instance of the base class.

## 18.5 Constraint blocks

The values of random variables are determined using constraint expressions that are declared using constraint blocks. Constraint blocks are class members, like tasks, functions, and variables. Constraint block names shall be unique within a class.

The syntax to declare a constraint block is as follows in Syntax 18-2.

```
constraint_declaration ::=                                                    // from A.1.10
    [ static ] constraint constraint_identifier constraint_block
constraint_block ::= { { constraint_block_item } }
constraint_block_item ::=
    solve solve_before_list before solve_before_list ;
  | constraint_expression
solve_before_list ::= constraint_primary { , constraint_primary }
constraint_primary ::= [ implicit_class_handle . | class_scope ] hierarchical_identifier select
constraint_expression ::=
    [ soft ] expression_or_dist ;
  | expression -> constraint_set
  | if ( expression ) constraint_set [ else constraint_set ]
  | foreach ( ps_or_hierarchical_array_identifier [ loop_variables ] ) constraint_set
  | disable soft constraint_primary ;
constraint_set ::=
    constraint_expression
  | { { constraint_expression } }
dist_list ::= dist_item { , dist_item }
dist_item ::= value_range [ dist_weight ]
dist_weight ::=
    := expression
  | :/ expression
```

constraint_prototype ::= [constraint_prototype_qualifier] [ **static** ] **constraint** constraint_identifier **;**

constraint_prototype_qualifier ::= **extern** | **pure**

extern_constraint_declaration ::=
  [ **static** ] **constraint** class_scope constraint_identifier constraint_block

identifier_list ::= identifier { **,** identifier }

expression_or_dist ::= expression [ **dist** **{** dist_list **}** ]  *// from A.2.10*

loop_variables ::= [ index_variable_identifier ] { **,** [ index_variable_identifier ] }  *// from A.6.8*

*Syntax 18-2—Constraint syntax (excerpt from Annex A)*

The *constraint_identifier* is the name of the constraint block. This name can be used to enable or disable a constraint using the `constraint_mode()` method (see 18.9).

The *constraint_block* is a list of expression statements that restrict the range of a variable or define relations between variables. A *constraint_expression* is any SystemVerilog expression or one of the constraint-specific operators, **dist** and -> (see 18.5.4 and 18.5.6, respectively).

The declarative nature of constraints imposes the following restrictions on constraint expressions:

— Functions are allowed with certain limitations (see 18.5.12).
— Operators with side effects, such as **++** and **--**, are not allowed.
— **randc** variables cannot be specified in ordering constraints (see **solve**...**before** in 18.5.10).
— **dist** expressions cannot appear in other expressions.

### 18.5.1 External constraint blocks

Constraint blocks can be declared outside their enclosing class declaration if a *constraint prototype* appears in the enclosing class declaration. A constraint prototype specifies that the class shall have a constraint of the specified name, but does not specify a constraint block to implement that constraint. A constraint prototype can take either of two forms, as shown in the following example:

```
class C;
   rand int x;
   constraint proto1;             // implicit form
   extern constraint proto2;      // explicit form
endclass
```

For both forms the constraint can be completed by providing an *external constraint block* using the class scope resolution operator, as in the following example:

```
constraint C::proto1 { x inside {-4, 5, 7}; }
constraint C::proto2 { x >= 0; }
```

An external constraint block shall appear in the same scope as the corresponding class declaration and shall appear after the class declaration in that scope. If the explicit form of constraint prototype is used, it shall be an error if no corresponding external constraint block is provided. If the implicit form of prototype is used and there is no corresponding external constraint block, the constraint shall be treated as an empty constraint and a warning may be issued. An empty constraint is one that has no effect on randomization, equivalent to a constraint block containing the constant expression 1.

For either form, it shall be an error if more than one external constraint block is provided for any given prototype, and it shall be an error if a constraint block of the same name as a prototype appears in the same class declaration.

### 18.5.2 Constraint inheritance

Constraints follow the same general rules for inheritance as other class members. The `randomize()` method is virtual and therefore honors constraints of the object on which it was called, regardless of the data type of the object handle through which the method was called.

A derived class shall inherit all constraints from its superclass. Any constraint in a derived class having the same name as a constraint in its superclass shall replace the inherited constraint of that name. Any constraint in a derived class that does not have the same name as a constraint in the superclass shall be an additional constraint.

If a derived class has a constraint prototype with the same name as a constraint in its superclass, that constraint prototype shall replace the inherited constraint. Completion of the derived class's constraint prototype shall then follow the rules described in 18.5.1.

An abstract class (i.e., a class declared using the syntax **`virtual class`**, as described in 8.21) may contain *pure constraints*. A pure constraint is syntactically similar to a constraint prototype but uses the **`pure`** keyword, as in the following example:

```
virtual class D;
   pure constraint Test;
endclass
```

A pure constraint represents an obligation on any non-abstract derived class (i.e., a derived class that is not **`virtual`**) to provide a constraint of the same name. It shall be an error if a non-abstract class does not have an implementation of every pure constraint that it inherits. It shall be an error to declare a pure constraint in a non-abstract class.

It shall be an error if a class containing a pure constraint also has a constraint block, constraint prototype or external constraint block of the same name. However, any class (whether abstract or not) may contain a constraint block or constraint prototype of the same name as a pure constraint that the class inherits; such a constraint shall override the pure constraint, and shall be a non-pure constraint for the class and any class derived from it.

An abstract class that inherits a constraint from its superclass may have a pure constraint of the same name. In this case, the pure constraint in the derived virtual class shall replace the inherited constraint.

A constraint that overrides a pure constraint may be declared using a constraint block in the body of the overriding class, or may be declared using a constraint prototype and external constraint as described in 18.5.1.

### 18.5.3 Set membership

Constraints support integer value sets and the set membership operator (as defined in 11.4.13).

Absent any other constraints, all values (either single values or value ranges) have an equal probability of being chosen by the **`inside`** operator.

The negated form of the **`inside`** operator denotes that expression lies outside the set: `!(expression` **`inside`** `{ set })`.

For example:

```
rand integer x, y, z;
constraint c1 {x inside {3, 5, [9:15], [24:32], [y:2*y], z};}
```

```
    rand integer a, b, c;
    constraint c2 {a inside {b, c};}

    integer fives[4] = '{ 5, 10, 15, 20 };
    rand integer v;
    constraint c3 { v inside {fives}; }
```

In SystemVerilog, the **`inside`** operator is bidirectional; thus, the preceding second example is equivalent to `a == b || a == c`.

### 18.5.4 Distribution

In addition to set membership, constraints support sets of weighted values called *distributions*. Distributions have two properties: they are a relational test for set membership, and they specify a statistical distribution function for the results.

The syntax to define a distribution expression is as follows in Syntax 18-3.

```
constraint_expression ::=                                          // from A.1.10
    expression_or_dist ;
  ...
dist_list ::= dist_item { , dist_item }
dist_item ::= value_range [ dist_weight ]
dist_weight ::=
    := expression
  | :/ expression
expression_or_dist ::= expression [ dist { dist_list } ]           // from A.2.10
```

*Syntax 18-3—Constraint distribution syntax (excerpt from Annex A)*

The *expression* can be any integral SystemVerilog expression.

The distribution operator **`dist`** evaluates to true if the value of the expression is contained in the set; otherwise, it evaluates to false.

Absent any other constraints, the probability that the expression matches any value in the list is proportional to its specified weight. If there are constraints on some expressions that cause the distribution weights on these expressions to be not satisfiable, implementations are only required to satisfy the constraints. An exception to this rule is a weight of zero, which is treated as a constraint.

The distribution set is a comma-separated list of integral expressions and ranges. Optionally, each term in the list can have a weight, which is specified using the `:=` or `:/` operators. If no weight is specified for an item, the default weight is `:=` 1. The weight can be any integral SystemVerilog expression.

The `:=` operator assigns the specified weight to the item or, if the item is a range, to every value in the range.

The `:/` operator assigns the specified weight to the item or, if the item is a range, to the range as a whole. If there are `n` values in the range, the weight of each value is `range_weight / n`. For example:

```
    x dist {100 := 1, 200 := 2, 300 := 5}
```

means x is equal to 100, 200, or 300 with weighted ratio of 1-2-5. If an additional constraint is added that specifies that x cannot be 200,

```
x != 200;
x dist {100 := 1, 200 := 2, 300 := 5}
```

then x is equal to 100 or 300 with weighted ratio of 1-5.

It is easier to think about mixing ratios, such as 1-2-5, than the actual probabilities, because mixing ratios do not have to be normalized to 100%. Converting probabilities to mixing ratios is straightforward.

When weights are applied to ranges, they can be applied to each value in the range, or they can be applied to the range as a whole. For example:

```
x dist { [100:102] := 1, 200 := 2, 300 := 5}
```

means x is equal to 100, 101, 102, 200, or 300 with a weighted ratio of 1-1-1-2-5, and

```
x dist { [100:102] :/ 1, 200 := 2, 300 := 5}
```

means x is equal to one of 100, 101, 102, 200, or 300 with a weighted ratio of 1/3-1/3-1/3-2-5.

In general, distributions guarantee two properties: set membership and monotonic weighting. In other words, increasing a weight increases the likelihood of choosing those values.

Limitations are as follows:

— A **dist** operation shall not be applied to **randc** variables.

— A **dist** expression requires that the expression contain at least one **rand** variable.

### 18.5.5 Uniqueness constraints

A group of variables can be constrained using the **unique** constraint so that no two members of the group have the same value after randomization. The group of variables to be constrained shall be specified using a restricted form of the *open_range_list* syntax in which each item in the comma-separated list shall be one of the following:

— A scalar variable of integral type

— An unpacked array variable whose leaf element type is integral, or a slice of such a variable

---

constraint_expression ::= *// from A.1.10*
  ...
  | uniqueness_constraint **;**
uniqueness_constraint ::=
  **unique {** open_range_list[9] **}**

9) The *open_range_list* in a *uniqueness_constraint* shall contain only expressions that denote scalar or array variables, as described in 18.5.5.

---

*Syntax 18-4—Uniqueness constraint syntax (excerpt from Annex A)*

A *leaf element* of an unpacked array is found by descending through the array until an element is reached that is not of unpacked array type.

All members of the group of variables so specified (that is, any scalar variables, and all leaf elements of any arrays or slices) shall be of equivalent type. No **`randc`** variable shall appear in the group.

If the group of variables so specified contains fewer than two members, the constraint shall have no effect and shall not cause a constraint contradiction.

In the following example, variables `a[2]`, `a[3]`, `b`, and `excluded` will all contain different values after randomization. Because of the constraint `exclusion`, none of the variables `a[2]`, `a[3]`, and `b` will contain the value 5.

```
rand byte a[5];
rand byte b;
rand byte excluded;
constraint u { unique {b, a[2:3], excluded}; }
constraint exclusion { excluded == 5; }
```

### 18.5.6 Implication

Constraints provide two constructs for declaring conditional (predicated) relations: implication and **`if-else`**.

The implication operator ( `->` ) can be used to declare an expression that implies a constraint.

The syntax to define an implication constraint is as follows in Syntax 18-5.

```
constraint_expression ::=                                        // from A.1.10
    ...
  | expression -> constraint_set
```

*Syntax 18-5—Constraint implication syntax (excerpt from Annex A)*

The *expression* can be any integral SystemVerilog expression.

The Boolean equivalent of the implication operator `a -> b` is `(!a || b)`. This states that if the expression is true, then random numbers generated are constrained by the constraint (or constraint set). Otherwise, the random numbers generated are unconstrained.

The *constraint_set* represents any valid constraint or an unnamed constraint set. If the expression is true, all of the constraints in the constraint set shall also be satisfied.

For example:

```
mode == little -> len < 10;
mode == big -> len > 100;
```

In this example, the value of `mode` implies that the value of `len` shall be constrained to less than 10 (`mode == little`), greater than 100 (`mode == big`), or unconstrained (`mode != little` and `mode != big`).

In the example

```
rand bit [3:0] a, b;
constraint c { (a == 0) -> (b == 1); }
```

both `a` and `b` are 4 bits; therefore, there are 256 combinations of `a` and `b`. Constraint `c` says that `a == 0` implies that `b == 1`, thereby eliminating 15 combinations: {0,0}, {0,2}, … {0,15}. Therefore, the probability that `a == 0` is thus 1/(256-15) or 1/241.

### 18.5.7 if–else constraints

The **`if–else`** style constraints are also supported.

The syntax to define an **`if–else`** constraint is as follows in Syntax 18-6.

```
constraint_expression ::=                                        // from A.1.10
    ...
  | if ( expression ) constraint_set [ else constraint_set ]
```

*Syntax 18-6—If–else constraint syntax (excerpt from Annex A)*

The *expression* can be any integral SystemVerilog expression.

The *constraint_set* represents any valid constraint or an unnamed constraint set. If the expression is true, all of the constraints in the first constraint or constraint set shall be satisfied; otherwise, all of the constraints in the optional **`else`** constraint or constraint set shall be satisfied. Constraint sets can be used to group multiple constraints.

The **`if–else`** style constraint declarations are equivalent to implications

```
if (mode == little)
   len < 10;
else if (mode == big)
   len > 100;
```

which is equivalent to

```
mode == little -> len < 10 ;
mode == big -> len > 100 ;
```

In this example, the value of `mode` implies that the value of `len` is less than 10, greater than 100, or unconstrained.

Just like implication, **`if–else`** style constraints are bidirectional. In the preceding declaration, the value of `mode` constrains the value of `len`, and the value of `len` constrains the value of `mode`.

Because the **`else`** part of an **`if–else`** style constraint declaration is optional, there can be confusion when an **`else`** is omitted from a nested **`if`** sequence. This is resolved by always associating the **`else`** with the closest previous **`if`** that lacks an **`else`**. In the following example, the **`else`** goes with the inner **`if`**, as shown by indentation:

```
if (mode != big)
   if (mode == little)
      len < 10;
   else // the else applies to preceding if
      len > 100;
```

### 18.5.8 Iterative constraints

Iterative constraints allow arrayed variables to be constrained using loop variables and indexing expressions, or by using array reduction methods.

#### 18.5.8.1 foreach iterative constraints

The syntax to define a **foreach** iterative constraint is as follows in Syntax 18-7.

```
constraint_expression ::=                                                    // from A.1.10
    ...
  | foreach ( ps_or_hierarchical_array_identifier [ loop_variables ] ) constraint_set
loop_variables ::= [ index_variable_identifier ] { , [ index_variable_identifier ] }   // from A.6.8
```

*Syntax 18-7—Foreach iterative constraint syntax (excerpt from Annex A)*

The **foreach** construct specifies iteration over the elements of an array. Its argument is an identifier that designates any type of array (fixed-size, dynamic, associative, or queue) followed by a list of loop variables enclosed in square brackets. Each loop variable corresponds to one of the dimensions of the array.

For example:

```
class C;
   rand byte A[] ;

   constraint C1 { foreach ( A [ i ] ) A[i] inside {2,4,8,16}; }
   constraint C2 { foreach ( A [ j ] ) A[j] > 2 * j; }
endclass
```

`C1` constrains each element of the array `A` to be in the set [2,4,8,16]. `C2` constrains each element of the array `A` to be greater than twice its index.

The number of loop variables shall not exceed the number of dimensions of the array variable. The scope of each loop variable is the **foreach** constraint construct, including its *constraint_set*. The type of each loop variable is implicitly declared to be consistent with the type of array index. An empty loop variable indicates no iteration over that dimension of the array. As with default arguments, a list of commas at the end can be omitted; thus, **foreach**`( arr [ j ] )` is a shorthand for **foreach**`( arr [ j, , , , ] )`. It shall be an error for any loop variable to have the same identifier as the array.

The mapping of loop variables to array indices is determined by the dimension cardinality, as described in 20.7.

```
//      1  2  3           3    4       1   2     -> Dimension numbers
int A [2][3][4];    bit [3:0][2:1] B [5:1][4];

foreach( A [ i, j, k ] ) ...
foreach( B [ q, r, , s ] ) ...
```

The first **foreach** causes `i` to iterate from 0 to 1, `j` from 0 to 2, and `k` from 0 to 3. The second **foreach** causes `q` to iterate from 5 to 1, `r` from 0 to 3, and `s` from 2 to 1.

**foreach** iterative constraints can include predicates. For example:

```
class C;
   rand int A[] ;

   constraint c1 { A.size inside {[1:10]}; }
   constraint c2 { foreach ( A[ k ] ) (k < A.size - 1) -> A[k + 1] > A[k]; }
endclass
```

The first constraint, `c1`, constrains the size of the array `A` to be between 1 and 10. The second constraint, `c2`, constrains each array value to be greater than the preceding one, i.e., an array sorted in ascending order.

Within a **`foreach`**, predicate expressions involving only constants, state variables, object handle comparisons, loop variables, or the size of the array being iterated behave as guards against the creation of constraints, and not as logical relations. For example, the implication in constraint `c2` above involves only a loop variable and the size of the array being iterated; thus, it allows the creation of a constraint only when `k < A.size() - 1`, which in this case prevents an out-of-bounds access in the constraint. Guards are described in more detail in 18.5.13.

Index expressions can include loop variables, constants, and state variables. Invalid or out-of-bounds array indices are not automatically eliminated; users must explicitly exclude these indices using predicates.

The size method of a dynamic array or queue can be used to constrain the size of the array (see constraint `c1` above). If an array is constrained by both size constraints and iterative constraints, the size constraints are solved first and the iterative constraints next. As a result of this implicit ordering between size constraints and iterative constraints, the size method shall be treated as a state variable within the **`foreach`** block of the corresponding array. For example, the expression `A.size` is treated as a random variable in constraint `c1` and as a state variable in constraint `c2`. This implicit ordering can cause the solver to fail in some situations.

### 18.5.8.2 Array reduction iterative constraints

The array reduction methods can produce a single integral value from an unpacked array of integral values (see 7.12.3). In the context of a constraint, an array reduction method is treated as an expression iterated over each element of the array, joined by the relevant operand for each method. The result returns a single value of the same type as the array element type or, if specified, the type of the expression in the **`with`** clause. For example:

```
class C;
   rand bit [7:0] A[] ;
   constraint c1 { A.size == 5; }
   constraint c2 { A.sum() with (int'(item)) < 1000; }
endclass
```

The constraint `c2` will be interpreted as

```
( int'(A[0])+int'(A[1])+int'(A[2])+int'(A[3])+int'(A[4]) ) < 1000
```

### 18.5.9 Global constraints

When an object member of a class is declared **`rand`**, all of its constraints and random variables are randomized simultaneously along with the other class variables and constraints. Constraint expressions involving random variables from other objects are called *global constraints* (see Figure 18-2).

```
class A;            // leaf node
   rand bit [7:0] v;
endclass
```

```
class B extends A; // heap node
   rand A left;
   rand A right;

   constraint heapcond {left.v <= v; right.v > v;}
endclass
```

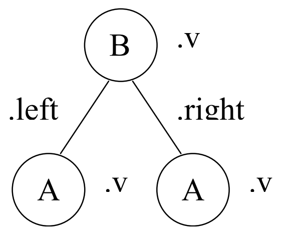


**Figure 18-2—Global constraints**

This example uses global constraints to define the legal values of an ordered binary tree. Class `A` represents a leaf node with an 8-bit value `v`. Class `B` extends class `A` and represents a heap node with value `v`, a left subtree, and a right subtree. Both subtrees are declared as **`rand`** in order to randomize them at the same time as other class variables. The constraint block named `heapcond` has two global constraints relating the left and right subtree values to the heap node value. When an instance of class `B` is randomized, the solver simultaneously solves for `B` and its left and right children, which in turn can be leaf nodes or more heap nodes.

The following rules determine which objects, variables, and constraints are to be randomized:

a) First, determine the set of objects that are to be randomized as a whole. Starting with the object that invoked the `randomize()` method, add all objects that are contained within it, are declared **`rand`**, and are active (see `rand_mode` in 18.8). The definition is recursive and includes all of the active random objects that can be reached from the starting object. The objects selected in this step are referred to as the *active random objects*.
b) Second, select all of the active constraints from the set of active random objects. These are the constraints that are applied to the problem.
c) Third, select all of the active random variables from the set of active random objects. These are the variables that are to be randomized. All other variable references are treated as state variables, whose current value is used as a constant.

### 18.5.10 Variable ordering

The solver shall assure that the random values are selected to give a uniform value distribution over legal value combinations (that is, all combinations of legal values have the same probability of being the solution). This important property guarantees that all legal value combinations are equally probable, which allows randomization to better explore the whole design space.

Sometimes, however, it is desirable to force certain combinations to occur more frequently. Consider the case where a 1-bit control variable `s` constrains a 32-bit data value `d`:

```
class B;
   rand bit s;
   rand bit [31:0] d;

   constraint c { s -> d == 0; }
endclass
```

The constraint `c` says "`s` implies `d` equals zero." Although this reads as if `s` determines `d`, in fact `s` and `d` are determined together. There are $1 + 2^{32}$ legal value combinations of `{s,d}`, but `s` is true only for one of them. Table 18-1 lists each of the legal value combinations and the probability of occurrence of each:

**Table 18-1—Unordered constraint c legal value probability**

| s | d | Probability |
|---|---|---|
| 1 | 'h00000000 | $1/(1 + 2^{32})$ |
| 0 | 'h00000000 | $1/(1 + 2^{32})$ |
| 0 | 'h00000001 | $1/(1 + 2^{32})$ |
| 0 | 'h00000002 | $1/(1 + 2^{32})$ |
| 0 | ... | |
| 0 | 'hfffffffe | $1/(1 + 2^{32})$ |
| 0 | 'hffffffff | $1/(1 + 2^{32})$ |

The constraints provide a mechanism for ordering variables so that `s` can be chosen independently of `d`. This mechanism defines a partial ordering on the evaluation of variables and is specified using the **`solve`** keyword.

```
class B;
   rand bit s;
   rand bit [31:0] d;
   constraint c { s -> d == 0; }
   constraint order { solve s before d; }
endclass
```

In this case, the order constraint instructs the solver to solve for `s` before solving for `d`. The effect is that `s` is now chosen 0 or 1 with 50/50% probability, and then `d` is chosen subject to the value of `s`. Adding this order constraint does not change the set of legal value combinations, but alters their probability of occurrence, as shown in Table 18-2:

**Table 18-2—Ordered constraint c legal value probability**

| s | d | Probability |
|---|---|---|
| 1 | 'h00000000 | 1/2 |
| 0 | 'h00000000 | $1/2 \times 1/2^{32}$ |
| 0 | 'h00000001 | $1/2 \times 1/2^{32}$ |
| 0 | 'h00000002 | $1/2 \times 1/2^{32}$ |
| 0 | ... | |
| 0 | 'hfffffffe | $1/2 \times 1/2^{32}$ |
| 0 | 'hffffffff | $1/2 \times 1/2^{32}$ |

Note that the probability of `d==0` is $1/(1 + 2^{32})$, near 0%, without the order constraint, and is $1/2 \times 1/2^{32}$, slightly over 50%, with the order constraint.

Variable ordering can be used to force selected corner cases to occur more frequently than they would otherwise. However, a "**solve**...**before**..." constraint does not change the solution space and, therefore, cannot cause the solver to fail.

The syntax to define variable order in a constraint block is as follows in Syntax 18-8.

```
constraint_block_item ::=                                                    // from A.1.10
    solve solve_before_list before solve_before_list ;
  | constraint_expression
solve_before_list ::= solve_before_primary { , solve_before_primary }
solve_before_primary ::= [ implicit_class_handle . | class_scope ] hierarchical_identifier select
```

*Syntax 18-8—Solve...before constraint ordering syntax (excerpt from Annex A)*

The following restrictions apply to variable ordering:

— Only random variables are allowed, that is, they shall be **rand**.

— **randc** variables are not allowed. **randc** variables are always solved before any other.

— The variables shall be integral values.

— A constraint block can contain both regular value constraints and ordering constraints.

— There shall be no circular dependencies in the ordering, such as "solve a before b" combined with "solve b before a."

— Variables that are not explicitly ordered shall be solved with the last set of ordered variables. These values are deferred until as late as possible to assure a good distribution of values.

— Variables that are partially ordered shall be solved with the latest set of ordered variables so that all ordering constraints are met. These values are deferred until as late as possible to assure a good distribution of values.

— Variables can be solved in an order that is not consistent with the ordering constraints, provided that the outcome is the same. An example situation where this might occur is as follows:

```
x == 0;
x < y;
solve y before x;
```

In this case, because `x` has only one possible assignment (0), `x` can be solved for before `y`. The constraint solver can use this flexibility to speed up the solving process.

## 18.5.11 Static constraint blocks

A constraint block can be defined as static by including the **static** keyword in its definition.

The syntax to declare a static constraint block is as follows in Syntax 18-9.

```
constraint_declaration ::=                                                   // from A.1.10
    [ static ] constraint constraint_identifier constraint_block
```

*Syntax 18-9—Static constraint syntax (excerpt from Annex A)*

If a constraint block is declared as **static**, then calls to `constraint_mode()` shall affect all instances of the specified constraint in all objects. Thus, if a static constraint is set to OFF, it is off for all instances of that particular class.

When a constraint is declared using a constraint prototype and an external constraint block, the **static** keyword shall be applied to both the constraint prototype and the external constraint block, or to neither. It shall be an error if one but not the other is qualified **static**. Similarly, a pure constraint may be qualified **static** but any overriding constraint must match the pure constraint's qualification or absence thereof.

### 18.5.12 Functions in constraints

Some properties are unwieldy or impossible to express in a single expression. For example, the natural way to compute the number of ones in a packed array uses a loop:

```
function int count_ones ( bit [9:0] w );
   for( count_ones = 0; w != 0; w = w >> 1 )
      count_ones += w & 1'b1;
endfunction
```

Such a function could be used to constrain other random variables to the number of 1 bits:

```
constraint C1  { length == count_ones( v ) ; }
```

Without the ability to call a function, this constraint requires the loop to be unrolled and expressed as a sum of the individual bits:

```
constraint C2
{
   length == ((v>>9)&1) + ((v>>8)&1) + ((v>>7)&1) + ((v>>6)&1) + ((v>>5)&1) +
             ((v>>4)&1) + ((v>>3)&1) + ((v>>2)&1) + ((v>>1)&1) + ((v>>0)&1);
}
```

Unlike the `count_ones` function, more complex properties, which require temporary state or unbounded loops, may be impossible to convert into a single expression. The ability to call functions, thus, enhances the expressive power of the constraint language and reduces the likelihood of errors. The two constraints, `C1` and `C2`, from above are not completely equivalent; `C2` is bidirectional (`length` can constrain `v` and vice versa), whereas `C1` is not.

To handle these common cases, SystemVerilog allows constraint expressions to include function calls, but it imposes certain semantic restrictions, as follows:

— Functions that appear in constraint expressions cannot contain **output** or **ref** arguments (**const ref** is allowed).
— Functions that appear in constraint expressions should be automatic (or preserve no state information) and have no side effects.
— Functions that appear in constraints cannot modify the constraints, for example, calling `rand_mode` or `constraint_mode` methods.
— Functions shall be called before constraints are solved, and their return values shall be treated as state variables.
— Random variables used as function arguments shall establish an implicit variable ordering or priority. Constraints that include only variables with higher priority are solved before other, lower priority constraints. Random variables solved as part of a higher priority set of constraints become state variables to the remaining set of constraints. For example:

```
class B;
   rand int x, y;
   constraint C { x <= F(y); }
   constraint D { y inside { 2, 4, 8 } ; }
endclass
```

forces `y` to be solved before `x`. Thus, constraint `D` is solved separately before constraint `C`, which uses the values of `y` and `F(y)` as state variables. In SystemVerilog, the behavior for variable ordering implied by function arguments differs from the behavior for ordering specified using the "**solve**...**before**..." constraint; function argument variable ordering subdivides the solution space thereby changing it. Because constraints on higher priority variables are solved without considering lower priority constraints at all, this subdivision can cause the overall constraints to fail. Within each prioritized set of constraints, cyclical (**randc**) variables are solved first.

— Circular dependencies created by the implicit variable ordering shall result in an error.
— Function calls in active constraints are executed an unspecified number of times (at least once) in an unspecified order.

### 18.5.13 Constraint guards

Constraint guards are predicate expressions that function as guards against the creation of constraints and not as logical relations to be satisfied by the solver. These predicate expressions are evaluated before the constraints are solved and are characterized by involving only the following items:

— Constants
— State variables
— Object handle comparisons (comparisons between two handles or a handle and the constant **null**)

In addition to these, iterative constraints (see 18.5.8) also consider loop variables and the size of the array being iterated as state variables.

Treating these predicate expressions as constraint guards prevents the solver from generating evaluation errors, thereby failing on some seemingly correct constraints. This enables users to write constraints that avoid errors due to nonexistent object handles or array indices out of bounds. For example, the sort constraint of the singly linked list, `SList`, shown below is intended to assign a random sequence of numbers that is sorted in ascending order. However, the constraint expression will fail on the last element when `next.n` results in an evaluation error due to a nonexistent handle.

```
class SList;
   rand int n;
   rand Slist next;

   constraint sort { n < next.n; }
endclass
```

This error condition can be avoided by writing a predicate expression to guard against that condition:

```
constraint sort { if( next != null ) n < next.n; }
```

In the preceding sort constraint, the **if** prevents the creation of a constraint when `next` == **null**, which in this case avoids accessing a nonexistent object. Both implication ( ->) and **if**…**else** can be used as guards.

Guard expressions can themselves include subexpressions that result in evaluation errors (e.g., null references), and they are also guarded from generating errors. This logical sifting is accomplished by evaluating predicate subexpressions using the following 4-state representation:

| | | | |
|---|---|---|---|
| — | `0` | `FALSE` | Subexpression evaluates to `FALSE`. |
| — | `1` | `TRUE` | Subexpression evaluates to `TRUE`. |
| — | `E` | `ERROR` | Subexpression causes an evaluation error. |
| — | `R` | `RANDOM` | Expression includes random variables and cannot be evaluated. |

Every subexpression within a predicate expression is evaluated to yield one of the previous four values. The subexpressions are evaluated in an arbitrary order, and the result of that evaluation plus the logical operation define the outcome in the alternate 4-state representation. A conjunction ( `&&` ), disjunction ( `||` ), or negation ( `!` ) of subexpressions can include some (perhaps all) guard subexpressions. The following rules specify the resulting value for the guard:

— Conjunction ( `&&` ): If any one of the subexpressions evaluates to `FALSE`, then the guard evaluates to `FALSE`. If any one subexpression evaluates to `ERROR`, then the guard evaluates to `ERROR`. Otherwise, the guard evaluates to `TRUE`.
  - If the guard evaluates to `FALSE`, then the constraint is eliminated.
  - If the guard evaluates to `TRUE`, then a (possibly conditional) constraint is generated.
  - If the guard evaluates to `ERROR`, then an error is generated and `randomize()` fails.

— Disjunction ( `||` ): If any one of the subexpressions evaluates to `TRUE`, then the guard evaluates to `TRUE`. If any one subexpression evaluates to `ERROR`, then the guard evaluates to `ERROR`. Otherwise, the guard evaluates to `FALSE`.
  - If the guard evaluates to `FALSE`, then a (possibly conditional) constraint is generated.
  - If the guard evaluates to `TRUE`, then an unconditional constraint is generated.
  - If the guard evaluates to `ERROR`, then an error is generated and `randomize()` fails.

— Negation ( `!` ): If the subexpression evaluates to `ERROR`, then the guard evaluates to `ERROR`. Otherwise, if the subexpression evaluates to `TRUE` or `FALSE`, then the guard evaluates to `FALSE` or `TRUE`, respectively.

These rules are codified by the truth tables shown in Figure 18-3.

| && | 0 | 1 | E | R |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | E | R |
| E | 0 | E | E | E |
| R | 0 | R | E | R |

Conjunction

| \|\| | 0 | 1 | E | R |
|---|---|---|---|---|
| 0 | 0 | 1 | E | R |
| 1 | 1 | 1 | 1 | 1 |
| E | E | 1 | E | E |
| R | R | 1 | E | R |

Disjunction

| ! | |
|---|---|
| 0 | 1 |
| 1 | 0 |
| E | E |
| R | R |

Negation

**Figure 18-3—Truth tables for conjunction, disjunction, and negation rules**

These rules are applied recursively until all subexpressions are evaluated. The final value of the evaluated predicate expression determines the outcome as follows:

— If the result is `TRUE`, then an unconditional constraint is generated.

— If the result is `FALSE`, then the constraint is eliminated and can generate no error.

— If the result is `ERROR`, then an unconditional error is generated and the constraint fails.

— If the final result of the evaluation is `RANDOM`, then a conditional constraint is generated.

When the final value is RANDOM, a traversal of the predicate expression tree is needed to collect all conditional guards that evaluate to RANDOM. When the final value is ERROR, a subsequent traversal of the expression tree is not required, allowing implementations to issue only one error.

*Example 1:*

```
class D;
   int x;
endclass

class C;
   rand int x, y;
   D a, b;
   constraint c1 { (x < y || a.x > b.x || a.x == 5 ) -> x+y == 10; }
endclass
```

In Example 1, the predicate subexpressions are (x < y), (a.x > b.x), and (a.x == 5), which are all connected by disjunction. Some possible cases are as follows:

— Case 1: a is non-**null**, b is **null**, a.x is 5.

  Because (a.x==5) is true, the fact that b.x generates an error does not result in an error.
  The unconditional constraint (x+y == 10) is generated.

— Case 2: a is **null**.

  This always results in error, irrespective of the other conditions.

— Case 3: a is non-**null**, b is non-**null**, a.x is 10, b.x is 20.

  All the guard subexpressions evaluate to FALSE.
  The conditional constraint (x<y) -> (x+y == 10) is generated.

*Example 2:*

```
class D;
   int x;
endclass

class C;
   rand int x, y;
   D a, b;
   constraint c1 { (x < y && a.x > b.x && a.x == 5 ) -> x+y == 10; }
endclass
```

In Example 2, the predicate subexpressions are (x < y), (a.x > b.x), and (a.x == 5), which are all connected by conjunction. Some possible cases are as follows:

— Case 1: a is non-**null**, b is **null**, a.x is 6.

  Because (a.x==5) is false, the fact that b.x generates an error does not result in an error.

  The constraint is eliminated.

— Case 2: a is **null**

  This always results in error, irrespective of the other conditions.

— Case 3: a is non-**null**, b is non-**null**, a.x is 5, b.x is 2.

  All the guard subexpressions evaluate to TRUE, producing constraint (x<y) -> (x+y == 10).

*Example 3:*

```
class D;
   int x;
endclass

class C;
   rand int x, y;
   D a, b;
   constraint c1 { (x < y && (a.x > b.x || a.x ==5)) -> x+y == 10; }
endclass
```

In Example 3, the predicate subexpressions are `(x < y)` and `(a.x > b.x || a.x == 5)`, which are connected by disjunction. Some possible cases are as follows:

- Case 1: `a` is non-**`null`**, `b` is **`null`**, `a.x` is 5.

  The guard expression evaluates to `(ERROR || a.x==5)`, which evaluates to `(ERROR || TRUE)`
  The guard subexpression evaluates to `TRUE`.
  The conditional constraint `(x<y) -> (x+y == 10)` is generated.
- Case 2: `a` is non-**`null`**, `b` is **`null`**, `a.x` is 8.

  The guard expression evaluates to `(ERROR || FALSE)` and generates an error.
- Case 3: `a` is **`null`**

  This always results in error, irrespective of the other conditions.
- Case 4: `a` is non-**`null`**, `b` is non-**`null`**, `a.x` is 5, `b.x` is 2.

  All the guard subexpressions evaluate to `TRUE`.
  The conditional constraint `(x<y) -> (x+y == 10)` is generated.

### 18.5.14 Soft constraints

The constraints described up to this point can be denoted as *hard constraints* because the solver must always satisfy them or result in a solver failure. In contrast, when there is no solution that satisfies all active hard constraints (if any) simultaneously with a constraint defined as *soft,* the solver shall discard that soft constraint and find a solution that satisfies the remaining constraints. If there are two or more soft constraint expressions that cannot be satisfied simultaneously, the soft constraints shall be discarded as specified in 18.5.14.1.

Soft constraints enable authors of generic verification blocks to provide complete working environments that are more easily extended because the constraint solver automatically disregards generic soft constraints overridden by subsequent, more specialized constraints. Soft constraints are often used to specify default values and distributions for random variables. For example, the author of a generic packet class might add a constraint to ensure packets of legal size are generated by default (absent any other constraints):

```
class Packet;
   rand bit mode;
   rand int length;
   constraint deflt {
      soft length inside {32,1024};
      soft mode -> length == 1024;
      // Note: soft mode -> {length == 1024;} is not legal syntax,
      // as soft must be followed by an expression
   }
endclass

Packet p = new();
p.randomize() with { length == 1512;}             // mode will randomize to 0
p.randomize() with { length == 1512; mode == 1;} // mode will randomize to 1
```

If the constraint expression `length inside {32,1024}` is not defined as soft, the call to `randomize()` will fail and require special attention. The failure might be resolved by explicitly turning off the constraint, which requires additional procedural code, or by using a new class to extend the base class and override the constraint with a new one, which significantly complicates the test. In contrast, a default value specified by a soft constraint is automatically overridden, which results in a simpler layered test.

### 18.5.14.1 Soft constraint priorities

Soft constraints only express a preference for one solution over another; they are discarded when they are contradicted by other more important constraints. Regular (hard) constraints must always be satisfied; they are never discarded and are thus considered to be of the same highest priority. Conversely, soft constraints may be overridden by hard constraints or other higher-priority soft constraints, therefore, a specific priority shall be associated with every soft constraint. The soft priorities are designed such that the last constraint specified by the user will prevail. Hence, constraints specified in subsequent layers of the verification environment are assigned higher priorities than those in the preceding layers.

The following rules determine the priorities of soft constraints:

- Constraints within the scope of the same construct—constraint block, class, or struct—are assigned a priority relative to their syntactic declaration order. Constraint expressions that appear later in the construct have higher priority.
- Constraints within external constraint blocks are assigned a priority relative to the declaration order of the constraint prototype (**extern** declaration) in the class. The priority depends on the prototype declaration and not on the out-of-body declaration. Constraint expressions in out-of-body constraint blocks whose prototypes appear later in the class have higher priority.
- Constraints in contained objects (**rand** class handles) have lower priority than all constraints in the container object (**class** or **struct**).
- Constraints in each contained object (**rand** class handle) are assigned a priority relative to the declaration order of their class handle. Constraints in objects whose handles appear later in the container object (**class** or **struct**) have higher priority. If the same object is contained multiple times, the constraints in the contained object shall have the priority of the highest priority object—the object whose handle is declared last.
- Constraints in a derived class shall have higher priority than all constraints in its superclasses.
- Constraints within in-line constraint blocks shall have higher priority than constraints in the class being randomized.
- Soft constraints within a **foreach** shall have a priority that is defined by iteration order; latter iterations shall have higher priority. If the relational operator is not defined for an index type of an associative array, the priority is implementation dependent.

The following example illustrates the preceding rules:

```
class B1;
   rand int x;
   constraint a { soft x > 10 ; soft x < 100 ; }
endclass            /* a1 */       /* a2 */

class D1 extends B1;
   constraint b { soft x inside {[5:9]} ; }
endclass                /* b1 */

class B2;
   rand int y;
   constraint c { soft y > 10 ; }
endclass            /* c1 */
```

```
class D2 extends B2;
   constraint d { soft y inside {[5:9]} ; }
   constraint e ;        /* d1 */
   rand D1 p1;
   rand B1 p2;
   rand D1 p3;
   constraint f { soft p1.x < p2.x ; }
endclass               /* f1 */

constraint D2::e { soft y > 100 ; }
                       /* e1 */

D2 d = new();
initial begin
   d.randomize() with { soft y inside {10,20,30} ; soft y < p1.x ; };
end                              /* i1 */              /* i2 */
```

The relative priority of the soft constraints (highest to lowest) in the preceding example is:

`i2→i1→f1→e1→d1→c1→p3.b1→p3.a2→p3.a1→p2.a2->p2.a1->p1.b1->p1.a2→p1.a1`

If handles `p1` and `p3` refer to the same object, the relative priority is:

`i2→i1→f1→e1→d1→c1→p3.b1→p3.a2→p3.a1→p2.a2→p2.a1`

Soft constraints can only be specified on random variables; they may not be specified for **`randc`** variables.

The constraint solver implements the following conceptual model:

— Consider two soft constraints `c1` and `c2`, such that `c1` has higher priority than `c2`.
  1) The constraint solver will first try to produce a solution satisfying both `c1` and `c2`.
  2) If it fails in (1) then it will try to produce a solution satisfying only `c1`.
  3) If it fails in (2) then it will try to produce a solution satisfying only `c2`.
  4) If it fails in (3) then it will discard both `c1` and `c2`.

— The constraint solver shall satisfy the following properties:
  - If a call to `randomize()` only involves soft constraints, the call can never fail.
  - If the soft constraints do not exhibit any contradictions, then the result is the same as if all constraints were declared hard.

### 18.5.14.2 Discarding soft constraints

A **`disable soft`** constraint on a random variable specifies that all lower priority soft constraints that reference the given random variable shall be discarded. For example:

```
class A;
   rand int x;
   constraint A1 { soft x == 3; }
   constraint A2 { disable soft x; } // discard soft constraints
   constraint A3 { soft x inside { 1, 2 }; }
endclass

initial begin
   A a = new();
   a.randomize();
end
```

The constraint A2 instructs the solver to discard all soft constraints of lower priority on random variable x, resulting in constraint A1 being discarded. Thus, the solver only has to satisfy the last constraint A3. As a result the previous example will result in random variable x taking on the values 1 and 2. Note that if the constraint A3 was omitted, then the variable would be unconstrained.

A **disable soft** constraint causes lower priority soft constraints to be discarded regardless of whether those constraints create a contradiction. This feature is very useful to extend the solution space beyond the default values specified by any preceding soft constraints. The following example illustrates this behavior:

```
class B;
   rand int x;
   constraint B1 { soft x == 5; }
   constraint B2 { disable soft x; soft x dist {5, 8};}
endclass

initial begin
   B b = new();
   b.randomize();
end
```

In this example, the **disable soft** constraint preceding the soft distribution in block B2 causes the lower priority constraint on variable x in block B1 to be discarded. Hence, the solver will assign to x the values 5 and 8 with equal distribution—the result of solving constraint: x **dist** {5,8}.

Contrast the behavior of the previous example when the disable soft constraint is omitted:

```
class B;
   rand int x;
   constraint B1 { soft x == 5; }
   constraint B3 { soft x dist {5, 8}; }
endclass

initial begin
   B b = new();
   b.randomize();
end
```

In this preceding, the soft distribution constraint in block B3 can be satisfied for the value of 5. Hence, the solver will assign x the value 5. In general, if it is desired for the distribution weights of a **soft dist** constraint to be satisfied regardless of the presence of lower priority soft constraints then those soft constraints should be discarded first.

## 18.6 Randomization methods

### 18.6.1 Randomize()

Variables in an object are randomized using the randomize() class method. Every class has a built-in randomize() virtual method, declared as follows:

```
virtual function int randomize();
```

The randomize() method is a virtual function that generates random values for all the active random variables in the object, subject to the active constraints.

The randomize() method returns 1 if it successfully sets all the random variables and objects to valid values; otherwise, it returns 0.

*Example:*

```
class SimpleSum;
   rand bit [7:0] x, y, z;
   constraint c {z == x + y;}
endclass
```

This class definition declares three random variables, `x`, `y`, and `z`. Calling the `randomize()` method shall randomize an instance of class `SimpleSum`:

```
SimpleSum p = new;
int success = p.randomize();
if (success == 1 ) ...
```

Checking the return status can be necessary because the actual value of state variables or addition of constraints in derived classes can render seemingly simple constraints unsatisfiable.

### 18.6.2 Pre_randomize() and post_randomize()

Every class contains `pre_randomize()` and `post_randomize()` methods, which are automatically called by `randomize()` before and after computing new random values.

The prototype for the `pre_randomize()` method is as follows:

```
function void pre_randomize();
```

The prototype for the `post_randomize()` method is as follows:

```
function void post_randomize();
```

When `obj.randomize()` is invoked, it first invokes `pre_randomize()` on `obj` and also all of its random object members that are enabled. After the new random values are computed and assigned, `randomize()` invokes `post_randomize()` on `obj` and also all of its random object members that are enabled.

Users can override the `pre_randomize()` in any class to perform initialization and set preconditions before the object is randomized. If the class is a derived class and no user-defined implementation of `pre_randomize()` exists, then `pre_randomize()` will automatically invoke `super.pre_randomize()`.

Users can override the `post_randomize()` in any class to perform cleanup, print diagnostics, and check post-conditions after the object is randomized. If the class is a derived class and no user-defined implementation of `post_randomize()` exists, then `post_randomize()` will automatically invoke `super.post_randomize()`.

If these methods are overridden, they shall call their associated base class methods; otherwise, their pre- and post-randomization processing steps shall be skipped.

The `pre_randomize()` and `post_randomize()` methods are not virtual. However, because they are automatically called by the `randomize()` method, which is virtual, they appear to behave as virtual methods.

### 18.6.3 Behavior of randomization methods

— Random variables declared as static are shared by all instances of the class in which they are declared. Each time the `randomize()` method is called, the variable is changed in every class instance.

— If `randomize()` fails, the constraints are infeasible, and the random variables retain their previous values.

— If `randomize()` fails, `post_randomize()` is not called.

— The `randomize()` method is built-in and cannot be overridden.

— The `randomize()` method implements object random stability. An object can be seeded by calling its `srandom()` method (see 18.13.3).

— The built-in methods `pre_randomize()` and `post_randomize()` are functions and cannot block.

## 18.7 In-line constraints—randomize() with

By using the `randomize()` **`with`** construct, users can declare in-line constraints at the point where the `randomize()` method is called. These additional constraints are applied along with the object constraints.

The syntax for `randomize()` **`with`** is as follows in Syntax 18-10.

```
inline_constraint _declaration ::=                                                    // not in Annex A
     class_variable_identifier . randomize [ ( [ variable_identifier_list | null ] ) ]
         with [ ( [ identifier_list ] ) ] constraint_block
randomize_call ::=                                                                    // from A.1.10
       randomize { attribute_instance }
          [ ( [ variable_identifier_list | null ] ) ]
          [ with [ ( [ identifier_list ] ) ] constraint_block ][38]
```

[38] In a randomize_call that is not a method call of an object of class type (i.e., a scope randomize), the optional parenthesized *identifier_list* after the keyword **with** shall be illegal, and the use of **null** shall be illegal.

*Syntax 18-10—Inline constraint syntax (excerpt from Annex A)*

The *class_variable_identifier* is the name of an instantiated object.

The unnamed *constraint_block* contains additional in-line constraints to be applied along with the object constraints declared in the class.

For example:

```
class SimpleSum;
   rand bit [7:0] x, y, z;
   constraint c {z == x + y;}
endclass

task InlineConstraintDemo(SimpleSum p);
   int success;
   success = p.randomize() with {x < y;};
endtask
```

This is the same example used before; however, `randomize()` **`with`** is used to introduce an additional constraint that `x < y`.

The `randomize()` **`with`** construct can be used anywhere an expression can appear. The constraint block following **`with`** can define all of the same constraint types and forms as would otherwise be declared in a class.

The `randomize()` **`with`** constraint block can also reference local variables and subroutine arguments, eliminating the need for mirroring a local state as member variables in the object class. When the constraint block is not preceded by the optional parenthesized *identifier_list*, the constraint block is considered to be *unrestricted*. The scope for resolution of variable names referenced in an unrestricted constraint block begins with the `randomize()` **`with`** object class; that is, the class of the object handle used in the method call to `randomize()`. Then, if a name fails to resolve within the `randomize()` **`with`** object class, the name is resolved normally starting in the scope containing the in-line constraint. Names qualified by **`this`** or **`super`** shall bind to the class of the object handle used in the call to the `randomize()` **`with`** method. Hence, it shall be an error if the qualified name fails to resolve within the `randomize()` **`with`** object class. Dotted names other than those qualified by **`this`** or **`super`** shall first be resolved in a downwards manner (see 23.3) starting in the scope of the `randomize()` **`with`** object class. If the dotted name does not resolve in the scope of the `randomize()` **`with`** object class, it shall be resolved following normal resolution rules in the scope containing the in-line constraint.

The **`local::`** qualifier (see 18.7.1) is used to bypass the scope of the (`randomize()` **`with`** object) class and begin the name resolution procedure in the (local) scope that contains the `randomize()` method call.

When the *constraint_block* is preceded by the optional parenthesized *identifier_list*, the constraint block is considered to be *restricted*. In a restricted constraint block, only variables whose name resolution begins with identifiers in the *identifier_list* shall resolve into the `randomize()` **`with`** object class; all other names shall resolve starting in the scope containing the `randomize()` method call. When the parenthesized *identifier_list* is present and the **`local::`** qualifier is used, the qualified name shall resolve starting in the scope containing the `randomize()` method call independent of whether the name is present in the *identifier_list*.

In the following example, the `randomize()` **`with`** class is `C1`.

```
class C1;
   rand integer x;
endclass

class C2;
   integer x;
   integer y;

   task doit(C1 f, integer x, integer z);
      int result;
      result = f.randomize() with {x < y + z;};
   endtask
endclass
```

In the `f.randomize()` **`with`** constraint block, `x` is a member of class `C1` and hides the `x` in class `C2`. It also hides the `x` argument in the `doit()` task. `y` is a member of `C2`. `z` is a local argument.

A restricted constraint block can be used to guarantee that local variable references will resolve into a local scope.

```
class C;
   rand integer x;
endclass

function int F(C obj, integer y);
   F = obj.randomize() with (x) { x < y; };
endfunction
```

In this example, only `x` is resolved into the object `obj` since only `x` is listed in the *identifier_list*. The reference to `y` will never bind into `obj` even if a later change adds a property named `y` into class `C`.

### 18.7.1 local:: scope resolution

The `randomize()` **`with`** constraint block can reference both class properties and variables local to the method call. Unqualified names in an unrestricted in-lined constraint block are then resolved by searching first in the scope of the `randomize()` **`with`** object class followed by a search of the scope containing the method call—the local scope. The **`local::`** qualifier modifies the resolution search order. When applied to an identifier within an in-line constraint, the **`local::`** qualifier bypasses the scope of the [`randomize()` **`with`** object] class and resolves the identifier in the local scope.

In the following example, the `randomize()` **`with`** class is `C`, and the local scope is the function `F()`:

```
class C;
   rand integer x;
endclass

function int F(C obj, integer x);
   F = obj.randomize() with { x < local::x; };
endfunction
```

In the unrestricted in-line constraint block of the `obj.randomize()` call, the unqualified name, `x`, binds to the property of class `C` (the scope of the object being randomized) while the qualified name **`local::`**`x` binds to the argument of the function `F()` (the local scope).

As a result of the preceding rules, the following apply:

- Names qualified only by **`this`** or **`super`** shall bind to the class of the object handle used in the `randomize()` **`with`** method call.
- Names qualified by **`local::`** shall bind to the scope containing the `randomize()` method call, including the special names **`this`** or **`super`** (i.e., **`local::this`**).
- The **`local::`** prefix may be used to qualify class scopes and type names.
- As it pertains to wildcard package imports, the syntactic form **`local::`**`a` shall be semantically identical to the unqualified name a declared in the local scope.
- Given a method call `obj.randomize()` **`with`**, the name **`local::`**`obj` shall bind to the scope of the `randomize()` **`with`** object class.

## 18.8 Disabling random variables with rand_mode()

The `rand_mode()` method can be used to control whether a random variable is active or inactive. When a random variable is inactive, it is treated the same as if it had not been declared **`rand`** or **`randc`**. Inactive variables are not randomized by the `randomize()` method, and their values are treated as state variables by the solver. All random variables are initially active.

The syntax for the `rand_mode()` method is as follows:

```
    task object[.random_variable]::rand_mode( bit on_off );
```

or

```
    function int object.random_variable::rand_mode();
```

The *object* is any expression that yields the object handle in which the random variable is defined.

The *random_variable* is the name of the random variable to which the operation is applied. If it is not specified (only allowed when called as a task), the action is applied to all random variables within the specified object.

When called as a task, the argument to the `rand_mode` method determines the operation to be performed as shown in Table 18-3.

**Table 18-3—rand_mode argument**

| Value | Meaning | Description |
|---|---|---|
| 0 | OFF | Sets the specified variables to inactive so that they are not randomized on subsequent calls to the `randomize()` method. |
| 1 | ON | Sets the specified variables to active so that they are randomized on subsequent calls to the `randomize()` method. |

For unpacked array variables, `random_variable` can specify individual elements using the corresponding index. Omitting the index results in all the elements of the array being affected by the call.

For unpacked structure variables, `random_variable` can specify individual members using the corresponding member. Omitting the member results in all the members of the structure being affected by the call.

If the random variable is an object handle, only the mode of the variable is changed, not the mode of random variables within that object (see global constraints in 18.5.9).

A compiler error shall be issued if the specified variable does not exist within the class hierarchy or it exists but is not declared as **`rand`** or **`randc`**.

When called as a function, `rand_mode()` returns the current active state of the specified random variable. It returns 1 if the variable is active (`ON`) and 0 if the variable is inactive (`OFF`).

The function form of `rand_mode()` only accepts singular variables; thus, if the specified variable is an unpacked array, a single element shall be selected via its index.

*Example:*

```
    class Packet;
       rand integer source_value, dest_value;
       ... other declarations
    endclass

    int ret;
    Packet packet_a = new;
    // Turn off all variables in object
    packet_a.rand_mode(0);

    // ... other code
    // Enable source_value
    packet_a.source_value.rand_mode(1);

    ret = packet_a.dest_value.rand_mode();
```

This example first disables all random variables in the object `packet_a` and then enables only the `source_value` variable. Finally, it sets the `ret` variable to the active status of variable `dest_value`.

The `rand_mode()` method is built-in and cannot be overridden.

If a random variable is declared as **`static`**, the rand_mode state of the variable shall also be static. For example, if `rand_mode()` is set to inactive, the random variable is inactive in all instances of the base class.

## 18.9 Controlling constraints with constraint_mode()

The `constraint_mode()` method can be used to control whether a constraint is active or inactive. When a constraint is inactive, it is not considered by the `randomize()` method. All constraints are initially active.

The syntax for the `constraint_mode()` method is as follows:

```
    task object[.constraint_identifier]::constraint_mode( bit on_off );
```

or

```
    function int object.constraint_identifier::constraint_mode();
```

The *object* is any expression that yields the object handle in which the constraint is defined.

The *constraint_identifier* is the name of the constraint block to which the operation is applied. The constraint name can be the name of any constraint block in the class hierarchy. If no constraint name is specified (only allowed when called as a task), the operation is applied to all constraints within the specified object.

When called as a task, the argument to the `constraint_mode` task method determines the operation to be performed as shown in Table 18-4.

**Table 18-4—constraint_mode argument**

| Value | Meaning | Description |
|---|---|---|
| 0 | OFF | Sets the specified constraint block to inactive so that it is not enforced by subsequent calls to the `randomize()` method. |
| 1 | ON | Sets the specified constraint block to active so that it is considered on subsequent calls to the `randomize()` method. |

A compiler error shall be issued if the specified constraint block does not exist within the class hierarchy.

When called as a function, `constraint_mode()` returns the current active state of the specified constraint block. It returns 1 if the constraint is active (`ON`) and 0 if the constraint is inactive (`OFF`).

*Example:*

```
class Packet;
   rand integer source_value;
   constraint filter1 { source_value > 2 * m; }
endclass

function integer toggle_rand( Packet p );
   if ( p.filter1.constraint_mode() )
      p.filter1.constraint_mode(0);
   else
      p.filter1.constraint_mode(1);

   toggle_rand = p.randomize();
endfunction
```

In this example, the `toggle_rand` function first checks the current active state of the constraint `filter1` in the specified `Packet` object `p`. If the constraint is active, the function deactivates it; if it is inactive, the function activates it. Finally, the function calls the `randomize()` method to generate a new random value for variable `source_value`.

The `constraint_mode()` method is built-in and cannot be overridden.

## 18.10 Dynamic constraint modification

There are several ways to dynamically modify randomization constraints, as follows:

— Implication and **`if–else`** style constraints allow declaration of predicated constraints.

— Constraint blocks can be made active or inactive using the `constraint_mode()` built-in method. Initially, all constraint blocks are active. Inactive constraints are ignored by the `randomize()` function.

— Random variables can be made active or inactive using the `rand_mode()` built-in method. Initially, all **`rand`** and **`randc`** variables are active. Inactive variables are not randomized by the `randomize()` method, and their values are treated as state variables by the solver.

— The weights in a **`dist`** constraint can be changed, affecting the probability that particular values in the set are chosen.

## 18.11 In-line random variable control

The `randomize()` method can be used to temporarily control the set of random and state variables within a class instance or object. When the `randomize()` method is called with no arguments, it behaves as described in the previous subclauses, that is, it assigns new values to all random variables in an object—those declared as **`rand`** or **`randc`**—so that all of the constraints are satisfied. When `randomize()` is called with arguments, those arguments designate the complete set of random variables within that object; all other variables in the object are considered state variables. For example, consider the following class and calls to `randomize()`:

```
class CA;
   rand byte x, y;
   byte v, w;

   constraint c1 { x < v && y > w );
endclass
```

```
CA a = new;

a.randomize();          // random variables: x, y  state variables: v, w
a.randomize( x );       // random variables: x     state variables: y, v, w
a.randomize( v, w );    // random variables: v, w  state variables: x, y
a.randomize( w, x );    // random variables: w, x  state variables: y, v
```

This mechanism controls the set of active random variables for the duration of the call to `randomize()`, which is conceptually equivalent to making a set of calls to the `rand_mode()` method to disable or enable the corresponding random variables. Calling `randomize()` with arguments allows changing the random mode of any class property, even those not declared as **rand** or **randc**. This mechanism, however, does not affect the cyclical random mode; it cannot change a nonrandom variable into a cyclical random variable (**randc**) and cannot change a cyclical random variable into a noncyclical random variable (change from **randc** to **rand**).

The scope of the arguments to the `randomize()` method is the object class. Arguments are limited to the names of properties of the calling object; expressions are not allowed. The random mode of local class members can only be changed when the call to `randomize()` has access to those properties, that is, within the scope of the class in which the local members are declared.

### 18.11.1 In-line constraint checker

Normally, calling the `randomize()` method of a class that has no random variables causes the method to behave as a checker. In other words, it assigns no random values and only returns a status: 1 if all constraints are satisfied and 0 otherwise. The in-line random variable control mechanism can also be used to force the `randomize()` method to behave as a checker.

The `randomize()` method accepts the special argument **null** to indicate no random variables for the duration of the call. In other words, all class members behave as state variables. This causes the `randomize()` method to behave as a checker instead of a generator. A checker evaluates all constraints and simply returns 1 if all constraints are satisfied and 0 otherwise. For example, if class `CA` defined previously executes the following call:

```
success = a.randomize( null );   // no random variables
```

then the solver considers all variables as state variables and only checks whether the constraint is satisfied, namely, that the relation (`x < v && y > w`) is true using the current values of `x`, `y`, `v`, and `w`.

## 18.12 Randomization of scope variables—std::randomize()

The built-in class `randomize()` method operates exclusively on class member variables. Using classes to model the data to be randomized is a powerful mechanism that enables the creation of generic, reusable objects containing random variables and constraints that can be later extended, inherited, constrained, overridden, enabled, disabled, and merged with or separated from other objects. The ease with which classes and their associated random variables and constraints can be manipulated makes classes an ideal vehicle for describing and manipulating random data and constraints. However, some less-demanding problems that do not require the full flexibility of classes can use a simpler mechanism to randomize data that do not belong to a class. The scope randomize function, `std::randomize()`, enables users to randomize data in the current scope without the need to define a class or instantiate a class object.

The syntax of the scope randomize function is as follows in Syntax 18-11.

scope_randomize ::=
    [ **std ::** ] **randomize (** [ variable_identifier_list ] **)** [ **with** constraint_block ]

*Syntax 18-11—Scope randomize function syntax (not in Annex A)*

The scope randomize function behaves exactly the same as a class randomize method, except that it operates on the variables of the current scope instead of class member variables. Arguments to this function specify the variables that are to be assigned random values, i.e., the random variables.

For example:

```
module stim;
   bit [15:0] addr;
   bit [31:0] data;

   function bit gen_stim();
      bit success, rd_wr;

      success = randomize( addr, data, rd_wr );  // call std::randomize
      return rd_wr ;
   endfunction

   ...
endmodule
```

The function `gen_stim` calls `std::randomize()` with three variables as arguments: `addr`, `data`, and `rd_wr`. Thus, `std::randomize()` assigns new random variables to the variables that are visible in the scope of the `gen_stim` function. In the preceding example, `addr` and `data` have module scope, whereas `rd_wr` has scope local to the function. The preceding example can also be written using a class:

```
class stimc;
   rand bit [15:0] addr;
   rand bit [31:0] data;
   rand bit rd_wr;
endclass

function bit gen_stim( stimc p );
   bit [15:0] addr;
   bit [31:0] data;
   bit success;
   success = p.randomize();
   addr = p.addr;
   data = p.data;
   return p.rd_wr;
endfunction
```

However, for this simple application, the scope randomize function leads to a straightforward implementation.

The scope randomize function returns 1 if it successfully sets all the random variables to valid values; otherwise, it returns 0. If the scope randomize function is called with no argument, it shall not change the value of any variable but instead it shall check its constraints. All constraint expressions in its constraint_block shall be evaluated, and if one or more of those expressions evaluates to false (0) then the randomize call shall return 0; otherwise it shall return 1.

### 18.12.1 Adding constraints to scope variables—std::randomize() with

The `std::randomize()` **with** form of the scope randomize function allows users to specify random constraints to be applied to the local scope variables. When specifying constraints, the arguments to the scope randomize function become random variables; all other variables are considered state variables.

```
task stimulus( int length );
   int a, b, c, success;

   success = std::randomize( a, b, c ) with { a < b ; a + b < length ; };
   ...
   success = std::randomize( a, b ) with { b - a > length ; };
   ...
endtask
```

The preceding task stimulus calls `std::randomize` twice resulting in two sets of random values for its local variables `a`, `b`, and `c`. In the first call, variables `a` and `b` are constrained so that variable `a` is less than `b` and their sum is less than the task argument length, which is designated as a state variable. In the second call, variables `a` and `b` are constrained so that their difference is greater than the state variable length.

## 18.13 Random number system functions and methods

### 18.13.1 $urandom

The system function `$urandom` provides a mechanism for generating pseudo-random numbers. The function returns a new 32-bit random number each time it is called. The number shall be unsigned.

The syntax for `$urandom` is as follows:

```
function int unsigned $urandom [ (int seed ) ] ;
```

The `seed` is an optional argument that determines the sequence of random numbers generated. The seed can be any integral expression. The random number generator (RNG) shall generate the same sequence of random numbers every time the same seed is used.

The RNG is deterministic. Each time the program executes, it cycles through the same random sequence. This sequence can be made nondeterministic by seeding the `$urandom` function with an extrinsic random variable, such as the time of day.

For example:

```
bit [64:1] addr;
bit [ 3:0] number;

addr[32:1] = $urandom( 254 );    // Initialize the generator,
                                 // get 32-bit random number
addr = {$urandom, $urandom };    // 64-bit random number
number = $urandom & 15;          // 4-bit random number
```

### 18.13.2 $urandom_range()

The `$urandom_range()` function returns an unsigned integer within a specified range.

The syntax for `$urandom_range()` is as follows:

```
function int unsigned $urandom_range( int unsigned maxval,
                                      int unsigned minval = 0 );
```

The function shall return an unsigned integer in the range of `maxval` ... `minval`.

*Example 1:*

```
val = $urandom_range(7,0);
```

If `minval` is omitted, the function shall return a value in the range of `maxval` ... `0`.

*Example 2:*

```
val = $urandom_range(7);
```

If `maxval` is less than `minval`, the arguments are automatically reversed so that the first argument is larger than the second argument.

*Example 3:*

```
val = $urandom_range(0,7);
```

All of the three previous examples produce a value in the range of 0 to 7, inclusive.

`$urandom_range()` is automatically thread stable (see 18.14.2).

### 18.13.3 srandom()

The `srandom()` method allows manually seeding the RNG of objects or threads. The RNG of a process can be seeded using the `srandom()` method of the process (see 9.7).

The prototype of the `srandom()` method is as follows:

```
function void srandom( int seed );
```

The `srandom()` method initializes an object's RNG using the value of the given seed.

### 18.13.4 get_randstate()

The `get_randstate()` method retrieves the current state of an object's RNG. The state of the RNG associated with a process is retrieved using the `get_randstate()` method of the process (see 9.7).

The prototype of the `get_randstate()` method is as follows:

```
function string get_randstate();
```

The `get_randstate()` method returns a copy of the internal state of the RNG associated with the given object.

The RNG state is a string of unspecified length and format. The length and contents of the string are implementation dependent.

### 18.13.5 set_randstate()

The `set_randstate()` method sets the state of an object's RNG. The state of the RNG associated with a process is set using the `set_randstate()` method of the process (see 9.7).

The prototype of the `set_randstate()` method is as follows:

```
function void set_randstate( string state );
```

The `set_randstate()` method copies the given state into the internal state of an object's RNG.

The RNG state is a string of unspecified length and format. Calling `set_randstate()` with a string value that was not obtained from `get_randstate()`, or from a different implementation of `get_randstate()`, is undefined.

## 18.14 Random stability

The RNG is localized to threads and objects. Because the sequence of random values returned by a thread or object is independent of the RNG in other threads or objects, this property is called *random stability*. Random stability applies to the following:

— The system randomization calls, `$urandom()` and `$urandom_range()`
— The `shuffle()` array manipulation method
— The procedural **`randcase`** and **`randsequence`** statements
— The object and process random seeding method, `srandom()`
— The object and scope randomization method, `randomize()`

Testbenches with this feature exhibit more stable RNG behavior in the face of small changes to the user code. Additionally, it enables more precise control over the generation of random values by manually seeding threads and objects.

### 18.14.1 Random stability properties

Random stability encompasses the following properties:

— *Initialization RNG*. Each module instance, interface instance, program instance, and package has an initialization RNG. Each initialization RNG is seeded with the default seed. The default seed is an implementation-dependent value. An initialization RNG shall be used in the creation of static processes and static initializers (see the following list items). Static processes are defined in Annex P.

— *Thread stability*. Each thread has an independent RNG for all randomization system calls invoked from that thread. When a new dynamic thread is created, its RNG is seeded with the next random value from its parent thread. This property is called *hierarchical seeding*. When a static process is created, its RNG is seeded with the next value from the initialization RNG of the module instance, interface instance, program instance, or package containing the thread declaration.

  Program and thread stability can be achieved as long as thread creation and random number generation are done in the same order as before. When adding new threads to an existing test, they can be added at the end of a code block in order to maintain random number stability of previously created work.

— *Object stability*. Each class instance (object) has an independent RNG for all randomization methods in the class. When an object is created using **`new`**, its RNG is seeded with the next random value from the thread that creates the object. When a class object is created by a static declaration initializer, there is no active thread; thus, the RNG of the created object is seeded with the next random value of the initialization RNG of the module instance, interface instance, program instance, or package in which the declaration occurred.

Object stability shall be preserved when object and thread creation and random number generation are done in the same order as before. In order to maintain random number stability, new objects, threads, and random numbers can be created after existing objects are created.

— *Manual seeding.* All noninitialization RNGs can be manually seeded. Combined with hierarchical seeding, this facility allows users to define the operation of a subsystem (hierarchy subtree) completely with a single seed at the root thread of the subsystem.

### 18.14.2 Thread stability

Random values returned from the `$urandom` and `$urandom_range` system calls, `std::randomize()` scope randomization method, and `shuffle()` array manipulation method are independent of thread execution order. Random values generated to select branches of the procedural **randcase** and **randsequence** statements are also independent of thread execution order. For example:

```
integer x, y, z;
fork        //set a seed at the start of a thread
   begin process::self.srandom(100); x = $urandom; end
            //set a seed during a thread
   begin y = $urandom; process::self.srandom(200); end
            // draw 2 values from the thread RNG
   begin z = $urandom + $urandom ; end
join
```

The preceding program fragment illustrates the following several properties:

— *Thread locality.* The values returned for x, y, and z are independent of the order of thread execution. This is an important property because it allows development of subsystems that are independent, controllable, and predictable.

— *Hierarchical seeding.* When a thread is created, its random state is initialized using the next random value from the parent thread as a seed. The three forked threads are all seeded from the parent thread.

Each thread is seeded with a unique value, determined solely by its parent. The root of a thread execution subtree determines the random seeding of its children. This allows entire subtrees to be moved and preserves their behavior by manually seeding their root thread.

### 18.14.3 Object stability

The `randomize()` method built into every class exhibits object stability. This is the property that calls to `randomize()` in one instance are independent of calls to `randomize()` in other instances and are independent of calls to other randomize functions.

For example:

```
class C1;
   rand integer x;
endclass

class C2;
   rand integer y;
endclass

initial begin
```

```
      C1 c1 = new();
      C2 c2 = new();
      integer z;
      void'(c1.randomize());
      // z = $random;
      void'(c2.randomize());
   end
```

- The values returned for `c1.x` and `c2.y` are independent of each other.
- The calls to `randomize()` are independent of the `$random` system call. If one uncomments the line `z = $random` above, there is no change in the values assigned to `c1.x` and `c2.y`.
- Each instance has a unique source of random values that can be seeded independently. That random seed is taken from the parent thread when the instance is created.
- Objects can be seeded at any time using the `srandom()` method.

```
class C3
   function new (integer seed);
      //set a new seed for this instance
      this.srandom(seed);
   endfunction
endclass
```

Once an object is created, there is no guarantee that the creating thread can change the object's random state before another thread accesses the object. Therefore, it is best that objects self-seed within their **new** method rather than externally.

An object's seed can be set from any thread. However, a thread's seed can only be set from within the thread itself.

## 18.15 Manually seeding randomize

Each object maintains its own internal RNG, which is used exclusively by its `randomize()` method. This allows objects to be randomized independently of each other and of calls to other system randomization functions. When an object is created, its RNG is seeded using the next value from the RNG of the thread that creates the object. This process is called *hierarchical object seeding*.

Sometimes it is desirable to manually seed an object's RNG using the `srandom()` method. This can be done either in a class method or external to the class definition:

An example of seeding the RNG internally, as a class method, is as follows:

```
class Packet;
   rand bit[15:0] header;
   ...
   function new (int seed);
      this.srandom(seed);
      ...
   endfunction
endclass
```

An example of seeding the RNG externally is as follows:

```
Packet p = new(200); // Create p with seed 200.
p.srandom(300);      // Re-seed p with seed 300.
```

Calling `srandom()` in an object's **`new`**`()` function assures the object's RNG is set with the new seed before any class member values are randomized.

## 18.16 Random weighted case—randcase

```
statement_item ::=                                                    // from A.6.4
    ...
  | randcase_statement
randcase_statement ::=                                                // from A.6.7
    randcase randcase_item { randcase_item } endcase
randcase_item ::= expression : statement_or_null
```

*Syntax 18-12—Randcase syntax (excerpt from Annex A)*

The keyword **`randcase`** introduces a **`case`** statement that randomly selects one of its branches. The *randcase_item expressions* are non-negative integral values that constitute the branch weights. An item's weight divided by the sum of all weights gives the probability of taking that branch. For example:

```
randcase
  3 : x = 1;
  1 : x = 2;
  4 : x = 3;
endcase
```

The sum of all weights is 8; therefore, the probability of taking the first branch is 0.375, the probability of taking the second is 0.125, and the probability of taking the third is 0.5.

If a branch specifies a zero weight, then that branch is not taken. If all *randcase_items* specify zero weights, then no branch is taken and a warning can be issued.

The randcase weights can be arbitrary expressions, not just constants. For example:

```
byte a, b;

randcase
  a + b : x = 1;
  a - b : x = 2;
  a ^ ~b : x = 3;
  12'h800 : x = 4;
endcase
```

The precision of each weight expression is self-determined. The sum of the weights is computed using standard addition semantics (maximum precision of all weights), where each summand is unsigned. Each weight expression is evaluated at most once (implementations can cache identical expressions) in an unspecified order. In the preceding example, the first three weight expressions are computed using 8-bit precision, and the fourth expression is computed using 12-bit precision. The resulting weights are added as unsigned values using 12-bit precision. The weight selection then uses unsigned 12-bit comparison.

Each call to **`randcase`** retrieves one random number in the range of 0 to the sum of the weights. The weights are then selected in declaration order: smaller random numbers correspond to the first (top) weight statements.