IEEE Standards

IEEE Std 1364.1™-2002

1364.1™

IEEE Standard for Verilog® Register Transfer Level Synthesis

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

IEEE

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

18 December 2002

Print: SH95068
PDF: SS95068

IEEE Standard for Verilog® Register Transfer Level Synthesis

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 10 December 2002

IEEE-SA Standards Board

Abstract: Standard syntax and semantics for Verilog® HDL-based RTL synthesis are described in this standard.
Keywords: hardware description language, HDL, RTL, synthesis, Verilog®

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

Print: ISBN 0-7381-3501-1 SH95068
PDF: ISBN 0-7381-3502-X SS95068

Introduction

(This introduction is not part of IEEE Std 1364.1-2002, IEEE Standard for Verilog® Register Transfer Level Synthesis.)

This standard describes a standard syntax and semantics for Verilog® HDL-based RTL synthesis. It defines the subset of IEEE Std 1364-2001 (Verilog HDL) that is suitable for RTL synthesis and defines the semantics of that subset for the synthesis domain.

The purpose of this standard is to define a syntax and semantics that can be used in common by all compliant RTL synthesis tools to achieve uniformity of results in a similar manner to which simulation and analysis tools use IEEE Std 1364-2001. This will allow users of synthesis tools to produce well-defined designs whose functional characteristics are independent of a particular synthesis implementation by making their designs compliant with this standard.

The standard is intended for use by logic designers and electronic engineers.

Initial work on this standard started as a RTL synthesis subset working group under Open Verilog International (OVI). After OVI approved of the draft 1.0 with an overwhelming affirmative response, an IEEE Project Authorization Request (PAR) was obtained in July 1998 to clear its way for IEEE standardization. Most of the members of the original group continued to be part of the Pilot Group under P1364.1 to lead the technical work. The active members at the time of OVI draft 1.0 publication were as follows:

J. Bhasker, *Chair*

Victor Berman	Don Hejna	Doug Smith
David Bishop	Mike Quayle	Yatin Trivedi
Vassilios Gerousis	Ambar Sarkar	Rohit Vora

An approved draft D1.4 was ready by April 1999, thanks very much to the efforts of the following task leaders:

David Bishop (Web Admin.)	Don Hejna (Syntax)	Doug Smith (Pragmas)
Ken Coffman (Semantics)		Yatin Trivedi (Editor)

When the working group was ready to initiate the standardization process, it was decided to postpone the process for the following reasons:

a) The synthesis subset draft was based on Verilog IEEE Std 1364-1995.

b) A new updated Verilog language was imminent.

c) The new Verilog language contained many new synthesizable constructs.

It wasn't until early 2001 that Verilog IEEE Std 1364-2001 was finalized. The working group restarted their work by first looking at the synthesizability aspects of the new features in the language. Thereafter, RAM/ROM modeling features and new attributes syntax were introduced into the draft standard.

Many individuals from many different organizations participated directly or indirectly in the standardization process. A majority of the working group meetings were held via teleconferences with continued discussions on the working group reflector.

Participants

At the time this standard was completed, the working group had the following membership:

J. Bhasker, *Chair*

Ed Arthur
Victor Berman
David Bishop (Web Admin.)
Stefen Boyd
Shalom Bresticker
Mike Ciletti
Ken Coffman
Ben Cohen
Cliff Cummings
Mark Curry
Tom Fitzpatrick
Krishna Garlapati
Vassilios Gerousis
Paul Graham
Don Hejna
Jim Hudson
Muzaffer Kal
Apurva Kalia
Tommy Kelly
Erich Marschner
Mike McNamara
Paul Menchini
Gilbert Nguyen
Sashi Obilisetty
Mike Quayle
Alain Raynaud
Ambar Sarkar
Steven Schulz
Doug Smith
Daryl Stewart
Jenjen Tiao
Praveen Tiwari
Yatin Trivedi
Alain Vachoux
Eugenio Villar
Rohit Vora
Joe Wetstein

The following members of the balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Peter Ashenden
Stephen Bailey
Victor Barinov
J. Bhasker
Stefen Boyd
Dennis Brophy
Kai Moon “Charles” Chow
Keith Chow
S. M. Claassen
Ken Coffman
Ben Cohen
Cliff Cummings
Brian Dalio
George Economakos
John Eldon
Peter Flake
Bill Galloway
Steve Golson
Jonathan Greenlaw
Ian Andrew Guyler
William Hanna
Downing Hopkins
Mitsuaki Ishikawa
Masamichi Kawarabayashi
Muzaffer Kal
Jake Karrfalt
Jim Lewis
Gregory Luri
James Markevitch
Gregory Maston
Michael McKinney
Mike McNamara
Paul Menchini
Yinghua Min
J. Manuel Moreno
Fumiaki Nagao
Karen Pieper
Mike Ricchetti
Francesco Sforza
David R. Smith
Stuart Sutherland
Atsushi Takahara
Jenjen Tiao
Srinivasa Vemuru
Scott Wecht
John Williams
Mark Zwolinski

When the IEEE-SA Standards Board approved this standard on 10 December 2002, it had the following membership:

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Alan Cookson, *NIST Representative*
Satish K. Aggarwal, *NRC Representative*

Andrew Ickowicz
IEEE Standards Project Editor

Verilog® is a registered trademark of Cadence Design Systems, Inc.

Contents

IEEE Standard for Verilog® Register Transfer Level Synthesis

1. Overview

1.1 Scope

This standard defines a set of modeling rules for writing Verilog® HDL descriptions for synthesis. Adherence to these rules guarantees the interoperability of Verilog HDL descriptions between register-transfer level synthesis tools that comply to this standard. The standard defines how the semantics of Verilog HDL are used, for example, to describe level- and edge-sensitive logic. It also describes the syntax of the language with reference to what shall be supported and what shall not be supported for interoperability.

Use of this standard will enhance the portability of Verilog-HDL-based designs across synthesis tools conforming to this standard. In addition, it will minimize the potential for functional mismatch that may occur between the RTL model and the synthesized netlist.

1.2 Compliance to this standard

1.2.1 Model compliance

A Verilog HDL model shall be considered compliant to this standard if the model:

a) uses only constructs described as supported or ignored in this standard, and
b) adheres to the semantics defined in this standard.

1.2.2 Tool compliance

A synthesis tool shall be considered compliant to this standard if it:

a) accepts all models that adhere to the model compliance definition in 1.2.1.
b) supports all pragmas defined in Clause 6.
c) produces a netlist model that has the same functionality as the input model based on the conformance rules of Clause 4.

NOTE—A compliant synthesis tool may have more features than those required by this standard. A synthesis tool may introduce additional guidelines for writing Verilog HDL models that may produce more efficient logic, or other mechanisms for controlling how a particular description is best mapped to a particular library.

1.3 Terminology

The word *shall* indicates 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 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* indicates a course of action permissible within the limits of the standard (*may* equals *is permitted*).

A synthesis tool is said to *accept* a Verilog construct if it allows that construct to be legal input. The construct is said to *interpret* the construct (or to provide an *interpretation* of the construct) by producing logic that represents the construct. A synthesis tool shall not be required to provide an interpretation for every construct that it accepts, but only for those for which an interpretation is specified by this standard.

The Verilog HDL constructs in this standard are categorized as:

— **Supported:** RTL synthesis shall interpret and map the construct to hardware.
— **Ignored:** RTL synthesis shall ignore the construct and shall not map that construct to hardware. Encountering the construct shall not cause synthesis to fail, but may cause a functional mismatch between the RTL model and the synthesized netlist. The mechanism, if any, by which a RTL synthesis notifies the user of such constructs is not defined. It is acceptable for a not supported construct to be part of an ignored construct.
— **Not supported:** RTL synthesis shall not support the construct. An RTL synthesis tool shall fail upon encountering the construct, and the failure mode shall be undefined.

1.4 Conventions

This standard uses the following conventions:

a) The body of the text of this standard uses **boldface** font to denote Verilog reserved words (such as **if**).
b) The text of the Verilog examples and code fragments is represented in a `fixed-width` font.
c) Syntax text that is ~~struck-through~~ refers to syntax that is not supported.
d) Syntax text that is underlined refers to syntax that is ignored.
e) “<“ and “>” are used to represent text in one of several different, but specific forms.
f) Any paragraph starting with “NOTE—” is informative and not part of the standard.
g) In the PDF version of this standard, colors are used in Clause 7 and Annex A. Supported reserved words are in red **boldface** font. Blue ~~struck-through~~ are unsupported constructs, and blue underlined are ignored constructs.

1.5 Contents of this standard

A synopsis of the clauses and annexes is presented as a quick reference. There are seven clauses and two annexes. All the clauses are the normative parts of this standard, while all the annexes are the informative part of the standard.

a) **Clause 1—Overview:** This clause discusses the conventions used in this standard and its contents.
b) **Clause 2—References:** This clause contains bibliographic entries pertaining to this standard.
c) **Clause 3—Definitions:** This clause defines various terms used in this standard.
d) **Clause 4—Verification methodology:** This clause describes the guidelines for ensuring functionality matches before and after synthesis.
e) **Clause 5—Modeling hardware elements:** This clause defines the styles for inferring special hardware elements.

f) **Clause 6—Pragmas:** This clause defines the pragmas that are part of this RTL synthesis subset.
g) **Clause 7—Syntax:** This clause describes the syntax of Verilog HDL supported for RTL synthesis.
h) **Annex A—Syntax summary:** This informative annex provides a summary of the syntax supported for synthesis.
i) **Annex B—Functional mismatches:** This informative annex describes some cases where a potential exists for functional mismatch to occur between the RTL model and the synthesized netlist.

1.6 Examples

All examples that appear in this document under *"Example:"* are for the sole purpose of demonstrating the syntax and semantics of Verilog HDL for synthesis. It is not the intent of this clause to demonstrate, recommend, or emphasize coding styles that are more (or less efficient) in generating synthesizable hardware. In addition, it is not the intent of this standard to present examples that represent a compliance test suite, or a performance benchmark, even though these examples are compliant to this standard.

2. References

This standard shall be used in conjunction with the following publication. When the following standards are superseded by an approved revision, the revision shall apply.

IEEE Std 1364™-2001, IEEE Standard Verilog Language Reference Manual.[1, 2]

3. Definitions

This clause defines various terms used in this standard. Terms used within this standard, but not defined in this clause, are assumed to be from IEEE Std 1364-2001[3].

3.1 asynchronous: Data that changes value independent of the clock edge.

3.2 combinational logic: Logic that does not have any storage device, either edge-sensitive or level-sensitive.

3.3 don't care value: The value **x** when used on the right-hand side of an assignment represents a don't care value.

3.4 edge-sensitive storage device: Any device mapped to by a synthesis tool that is edge-sensitive to a clock, for example, a flip-flop.

3.5 event list: Event list of an **always** statement.

3.6 high-impedance value: The value **z** represents a high-impedance value.

3.7 level-sensitive storage device: Any device mapped to by a synthesis tool that is level-sensitive to a clock; for example, a latch.

3.8 LRM: The IEEE Standard Verilog Language Reference Manual, IEEE Std 1364-2001.

---

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (http://standards.ieee.org/).
[2]The IEEE standards referred to in Clause 2 are trademarks belonging to the Institute of Electrical and Electronics Engineers, Inc.
[3]Information on references can be found in Clause 2 of this standard.

3.9 meta-comment: A Verilog comment (//) or (/* */) that is used to provide synthesis directives to a synthesis tool.

3.10 metalogical: A metalogical value is either an **x** or a **z**.

3.11 pragma: A generic term used to define a construct with no predefined language semantics that influences how a synthesis tool shall synthesize Verilog code into a circuit.

3.12 RHS: Right-hand side.

3.13 RTL: The register transfer level of modeling circuits in Verilog HDL.

3.14 sequential logic: Logic that includes any kind of storage device, either level-sensitive or edge-sensitive.

3.15 statically computable expression: An expression whose value can be evaluated during compilation.

3.16 synchronous: Data that changes only on a clock edge.

3.17 synthesis tool: Any system, process, or program that interprets register transfer level Verilog HDL source code as a description of an electronic circuit and derives a netlist description of that circuit.

3.18 timeout clause: Delays specified in an assignment statement, inter-assignment or intra-assignment.

3.19 transient delay: Propagation delay. Delays through multiple paths of logic each with its own propagation delay.

3.20 user: A person, system, process, or program that generates RTL Verilog HDL source code.

3.21 vector: A one-dimensional array.

4. Verification methodology

Synthesized results may be broadly classified as either combinational or sequential. Sequential logic has some form of internal storage (level-sensitive storage device, register, memory) that is involved in an output expression. Combinational logic has no such storage—the outputs are a pure function of the inputs with no internal loops.

The process of verifying synthesis results consists of applying identical inputs to both the original model and synthesized models and then comparing their outputs to ensure that they are equivalent. Equivalent in this context means that a synthesis tool shall provide an unambiguous definition of equivalence for values on input, output, and bidirectional ports. This also implies that the port list of the synthesized result must be the same as the original model—ports cannot be added or deleted during synthesis. Since synthesis in general does not recognize all the same delays as simulators, the outputs cannot be compared at every simulation time step. Rather, they can only be compared at specific points, when all transient delays have settled and all active timeout clauses have been exceeded. If the outputs match at the compared ports, the synthesis tool shall be compliant. There is no matching requirement placed on any internal nodes unless the *keep* attribute (see 6.1.4) is specified for such a node, in which case matching shall be ensured for that node.

Input stimulus shall comply to the following criteria:

a) Input data does not contain "unknowns" or other metalogical values.
b) For sequential verification, input data must change far enough in advance of sensing times for transient delays to have settled.
c) Clock and/or input data transitions must be delayed until after asynchronous set/reset signals have been released. The delay must be long enough to avoid a clock and/or data setup/hold time violation.
d) For edge-sensitive based designs, primary inputs of the design must change far enough in advance for the edge-sensitive storage device input data to respect the setup times with respect to the active clock edge. Also, the input data must remain stable for long enough to respect the hold times with respect to the active clock edge.
e) For level-sensitive based designs, primary inputs of the design must change far enough in advance for the level-sensitive storage device input data to respect the setup times. Also, the input data must remain stable for long enough to respect the hold times.

NOTE—A synthesis tool may define metalogical values appearing on primary outputs in one model as equivalent to logical values in the other model. For this reason, the input stimulus may need to reset internal storage devices to specific logical values before the outputs of both models are compared for logical values.

4.1 Combinational logic verification

To verify a combinational logic design or part of a design, the input stimulus shall be applied first. Sufficient time shall be provided for the design to settle, and then the outputs examined. Typically, this is done in a loop, so the outputs may be examined just before the next set of inputs is applied, that is, when all outputs have settled. Each iteration of the loop shall include enough delay so that the transient delays and timeout clause delays have been exceeded. A model is not in compliance with this standard if it is possible for combinational outputs to never reach a steady state (i.e., oscillatory behavior).

Example 1:

```
always @* a = #5 ~a;
// Example is not compliant with this standard because it
// exhibits oscillatory behavior.
```

4.2 Sequential logic verification

The general scheme of applying inputs periodically and then checking the outputs just before the next set of inputs is applied shall be repeated. Sequential designs are either edge-sensitive (typically consisting of edge-sensitive storage devices) or level-sensitive (typically consisting of level-sensitive storage devices).

The verification of designs containing edge-sensitive or level-sensitive components are as follows:

a) **Edge-sensitive models:** The same sequence of tasks shall be performed during verification: change the inputs, compute the results, check the outputs. However, for sequential verification these tasks shall be synchronized with a clock. The checking portion of the verification shall be performed just before the active clock edge. The input values may be changed after the clock edge and after sufficient time has elapsed to ensure that no hold time violations will occur. The circuit then has the entire rest of the clock period to compute the new results before they are latched at the next clock edge. The period of the clock generated by the stimulus shall be sufficient enough to allow the input and output signals to settle. When asynchronous data is assigned, the asynchronous data shall not change during the period in which the asynchronous control (the condition under which the data is assigned) is active.

b) **Level-sensitive models:** These designs are generally less predictable than edge sensitive models due to the asynchronous nature of the signal interactions. Verification of synthesized results depends on the application. With level-sensitive storage elements, a general rule is that data inputs should be stable before enables go inactive (i.e. latch) and checking of outputs is best done after enables are inactive (i.e. latched) and combinational delays have settled. A level-sensitive model in which it is possible, in the absence of further changes to the inputs of the model, for one or more internal values or outputs of the model never to reach a steady state (oscillatory behavior) is not in compliance with this standard.

5. Modeling hardware elements

This clause describes styles for modeling various hardware elements such as edge-sensitive storage devices, level-sensitive storage devices and three-state drivers.

The hardware inferences specified in this clause do not take into account any optimizations or transformations. This standard does not specify or limit optimization. A specific tool may perform optimization and not generate the suggested hardware inferences or may generate a different set of hardware inferences. This shall *not* be taken as a violation of this standard provided the synthesized netlist has the same functionality as the input model.

5.1 Modeling combinational logic

Combinational logic shall be modeled using a continuous assignment or a net declaration assignment or an always statement.

When using an always statement, the event list shall not contain an edge event (**posedge** or **negedge**). The event list does not affect the synthesized netlist. However, it may be necessary to include in the event list all the variables read in the always statement to avoid mismatches between simulation and synthesized logic.

A variable assigned in an always statement shall not be assigned using both a blocking assignment (=) and a nonblocking assignment (<=) in the same always statement.

The event list for a combinational logic model shall not contain the reserved words **posedge** or **negedge**. Not all variables that appear in the right hand side of an assignment are required to appear in the event list. For example, a variable does not have to appear in the event list of an always statement if it is assigned a value with a blocking assignment before being used in subsequent expressions within the same always statement.

The event list may be the implicit event expression list (@(*), @*).

Example 2:

```
always @ (in1 or in2)
   out = in1 + in2;
// always statement models combinational logic.
```

Example 3:

```
always @ (posedge a or b)
// Not supported; does not model combinational logic.
...
```


Example 4:

```
always @ (in)
   if (ena)
      out = in;
   else
      out = 1'b1;
// Supported, but simulation mismatch might occur.
// To assure the simulation will match the synthesized logic, add ena
// to the event list so the event list reads: always @ (in or ena)
```

Example 5:

```
always @ (in1 or in2 or sel)
begin
   out = in1; // Blocking assignment
   if (sel)
      out <= in2; // Nonblocking assignment.
end
// Not supported, cannot mix blocking and nonblocking assignments in
// an always statement.
```

Example 6:

```
always @*   // Implicit event expression yields combinational logic
begin
   tmp1 = a & b;
   tmp2 = c & d;
   z = tmp1 | tmp2;
end
```

5.2 Modeling edge-sensitive sequential logic

Sequential logic shall be modeled using an always statement that has one or more edge events in the event list.

5.2.1 Edge events

The reserved words **posedge** or **negedge** shall be used to specify edge events in the event list of the always statement.

5.2.1.1 Positive edge

The following represents a positive edge expression in an always statement:

```
always @ (posedge <clock_name>)
...
```

5.2.1.2 Negative edge

The following represents a negative edge expression in an always statement.

```
always @ (negedge <clock_name>)
...
```

5.2.2 Modeling edge-sensitive storage devices

An edge-sensitive storage device shall be modeled for a variable that is assigned a value in an always statement that has exactly one edge event in the event list. The edge event specified shall represent the clock edge condition under which the storage device stores the value.

Nonblocking procedural assignments should be used for variables that model edge-sensitive storage devices. Nonblocking assignments are recommended to avoid Verilog simulation race conditions.

Blocking procedural assignments may be used for variables that are temporarily assigned and used within an always statement.

Multiple event lists in an always statement shall not be supported.

Example 7:

```
reg out;
. . .
always @ (posedge clock)
   out <= in;
// out is a positive edge triggered edge-sensitive storage device.
```

Example 8:

```
reg [3:0] out;
. . .
always @ (negedge clock)
   out <= in;
// out models four negative edge-triggered
// edge-sensitive storage devices.
```

Example 9:

```
always @ (posedge clock)
   if (reset)
      out <= 1'b0;
   else
      out <= in;
// out models a positive edge-sensitive storage
// device with optionally a synchronous reset.
```


Example 10:

```
always @ (posedge clock)
   if (set)
      out <= 1'b1;
   else
      out <= in;
// out models a positive edge-sensitive storage
// device with optionally a synchronous set.
```

Example 11:

```
always @ (posedge clock)
begin
   out <= 0;
   @(posedge clock);
   out <= 1;
   @(posedge clock);
   out <= 1;
end
// Not legal; multiple event lists are not supported within an
// always statement.
```

NOTE—No specific style is required to infer edge-sensitive storage device with synchronous set/reset. A synthesis tool may optionally choose to or not to infer such a storage device. See the *sync_set_reset* attribute on how it can be used to infer a device with synchronous set/reset.

5.2.2.1 Edge-sensitive storage device modeling with asynchronous set-reset

An edge-sensitive storage device with an asynchronous set and/or asynchronous reset is modeled using an always statement whose event list contains edge events representing the clock and asynchronous control variables. Level-sensitive events shall not be allowed in the event list of an edge-sensitive storage device model.

Furthermore, the always statement shall contain an if statement to model the first asynchronous control and optional nested else if statements to model additional asynchronous controls. A final else statement, which specifies the synchronous logic portion of the always block, shall be controlled by the edge control variable not listed in the if and else if statements. The always statement shall be of the form:

```
always @ (posedge <condA> or negedge <condB> or negedge <condC> or ...
         posedge <Clock>)
   // Any sequence of edge events can be in event list.
   if (<condA>)  // Positive polarity since posedge <condA>.
   // ... <asynchronous logic>
   else if (~ <condB>) // Negative polarity since negedge <condB>.
   // ... <asynchronous logic>
   else if (~ <condC>)
   // ... <asynchronous logic>
   else  // Implicit posedge <Clock>.
   // ... <synchronous logic>
```

For every asynchronous control, there is an if statement that precedes the clock branch. The asynchronous set and or reset logic will therefore have higher priority than the clock edge.

The “final else” statement is determined as follows. If there are N edge events in the event list, the “else” following (N–1) if’s, at the same level as the top-level if statement, determines the “final else.” The final else statement specifies the synchronous logic part of the design.

Example 12:

```
always @ (posedge clock or posedge set)
   if (set)
      out <= 1'b1;
   else
      out <= din;
//  out is an edge-sensitive storage device with an asynchronous set.
```

Example 13:

```
always @ (posedge clock or posedge reset)
   out <= in;
// Not legal because the if statement is missing.
```

Example 14:

```
always @ (posedge clock or negedge clear)
   if (clear)  // This term should be inverted (!clear) to match
               // the polarity of the edge event.
      out <= 0;
   else
      out <= in;
// Not legal; if condition does not match the polarity of
// the edge event.
```

Example 15:

```
always @ (posedge clock or negedge clear)
   if (~ clear)
      out <= 0;
   else if (ping)          // Synchronous logic starts with this if.
      out <= in;
   else if (pong)
      out <= 8'hFF;
   else
      out <= pdata;
// Synchronous logic starts after first else.
```

5.3 Modeling level-sensitive storage devices

A level-sensitive storage device may be modeled for a variable when all the following apply:

a) The variable is assigned a value in an always statement without edge events in its event list (combinational logic modeling style).
b) There are executions of the always statement in which there is no explicit assignment to the variable.

The event list of the always statement should list all variables read within the always statement.

Nonblocking procedural assignments should be used for variables that model level-sensitive storage devices. This is to prevent Verilog simulation race conditions.

Blocking assignments may be used for intermediate variables that are temporarily assigned and used only in the same always statement.

Example 16:

```
always @ (enable or d)
   if (enable)
      q <= d;
// A level-sensitive storage device is inferred for q.
// If enable is deasserted, q will hold its value.
```

Example 17:

```
always @ (enable or d)
   if (enable)
      q <= d;
   else
      q <= 'b0;

// A latch is not inferred because the assignment to q is complete,
// i.e., q is assigned on every execution of the always statement.
```

5.4 Modeling three-state drivers

Three-state logic shall be modeled when a variable is assigned the value **z**. The assignment of **z** can be conditional or unconditional. If any driver of a signal contains an assignment to the value **z**, then all the drivers shall contain such an assignment.

z assignments shall not propagate across variable assignments (including implicit assignments, such as those which occur with module instantiations).

Example 18:

```
module ztest (test2, test1, test3, ena);
   input [0:1] ena;
   input [7:0] test1, test3;
   output [7:0] test2;
   wire [7:0] test2;

   assign test2 = (ena == 2'b01) ? test1 : 8'bz;
   assign test2 = (ena == 2'b10) ? test3 : 8'bz;
   // test2 is three-state when ena is 2'b00 or 2'b11.
endmodule
```

Example 19:

```
module ztest;
   wire test1, test2, test3;
   input test2;
   output test3;
   assign test1 = 1'bz;
   assign test3 = test1 & test2; // test3 will never receive
                                 // a z assignment.
endmodule
```

Example 20:

```
always @ (in)
begin
   tmp = 'bz;
   out = tmp;  // out shall not be driven by three state drivers
               // because the value 'bz does not propagate across the
               // variable assignment.
end
```

Example 21:

```
always @ (q or enb)
   if (!enb)
      out <= 'bz;
   else
      out <= q;
// out is a three-state driver.
```

Example 22:

```
// Three-state driver with non-registered enable:
always @(posedge clock)
   q <= din;

assign out = enb ? q : 1'bz;
// Generates one edge-sensitive storage device with a
// three-state driver on the output.
```

Example 23:

```
// Three-state driver with registered enable:
always @(posedge clock)
   if (!enb)
      out <= 1'bz;
   else
      out <= din;
// Generates two edge-sensitive storage devices, one for din, and
// one for enb, with a three-state driver on the output of the first
// storage device, controlled by the output of the second
// storage device.
```


5.5 Support for values x and z

The value **x** may be used as a primary on the RHS of an assignment to indicate a don't care value for synthesis.

The value **x** may be used in case item expressions (may be mixed with other expressions, such as 4'b01x0) in a casex statement to imply a don't care value for synthesis.

The value **x** shall not be used with any operators or mixed with other expressions.

The value **z** may be used as a primary on the RHS of an assignment to infer a three-state driver as described in 5.4.

The value **z** (or **?**) may be used in case item expressions (may be mixed with other expressions, such as 4'bz1z0) for casex and casez statements to imply a don't care value for synthesis.

The value **z** shall not be used with any operators or mixed with other expressions.

5.6 Modeling read-only memories (ROM)

An asynchronous ROM shall be modeled as combinational logic using one of the following styles:

a) One-dimensional array with data in case statement (see 5.6.1).
b) Two-dimensional array with data in initial statement (see 5.6.2).
c) Two-dimensional array with data in text file (see 5.6.3).

The *rom_block* attribute shall be used to identify the variable that models the ROM. If the *logic_block* attribute is used, then it shall imply that no ROM is to be inferred, and combinational logic be used instead.

NOTES

1—In the absence of either a rom_block or a logic_block attribute, a synthesis tool may opt to implement either as random logic or as a ROM.

2—The standard does not define how or in what form the ROM values are to be saved after synthesis when the *rom_block* attribute is used.

3—In each of the three cases above, there may be a simulation mismatch at time 0 if the ROM initialization does not occur prior to reading the ROM values.

5.6.1 One-dimensional array with data in case statement

In this style, the data values of a ROM shall be defined within a case statement. All the values of the ROM shall be defined within the **case** statement. The value assigned to each ROM address shall be a static expression (a static expression is one that can be evaluated at compile time).

The variable attributed with the *rom_block* attribute models the ROM. The address of the ROM shall be the same as the case expression. The ROM variable is the data. The case statement may contain other assignments or statements that may or may not affect the ROM variable. However all assignments to the ROM variable shall be done within only one case statement. In addition, the ROM variable must be assigned for all possible values of the case expression (ROM address).

Example 24:

```
module rom_case(
      (* synthesis, rom_block = "ROM_CELLXYZ01" *)
      output reg [3:0] z,
      input wire [2:0] a); // Address - 8 deep memory.

   always @*      // @(a)
      case (a)
         3'b000: z = 4'b1011;
         3'b001: z = 4'b0001;
         3'b100: z = 4'b0011;
         3'b110: z = 4'b0010;
         3'b111: z = 4'b1110;
         default: z = 4'b0000;
      endcase
endmodule // rom_case
// z is the ROM, and its address size is determined by a.
```

5.6.2 Two-dimensional array with data in initial statement

A Verilog memory (two-dimensional reg array) attributed as a *rom_block*, decorated with the attribute *rom_block*, shall be used to model a ROM. The address size and data size of the ROM shall be as specified in the declaration of the memory.

In addition, the values of the ROM shall be assigned using an initial statement. Uninitialized values shall have an implicit don't care assignment. The initial statement shall not be restricted to contain only assignment statements. It may contain other synthesizable statements, such as for loop statements, if and case statements, with the only restriction that the assignments to the ROM, which include data and address, shall be statically computable.

Such a memory shall only be read from other procedural blocks. It is an error to write to such a memory from any other procedural block other than the initial statement in which it is initialized.

The initial statement shall be supported when either of the attributes *logic_block* or *rom_block* is used.

Example 25:

```
module rom_2dimarray_initial (
      output wire [3:0] z,
      input wire [2:0] a); // address- 8 deep memory
// Declare a memory rom of 8 4-bit registers. The indices are 0 to 7:
   (* synthesis, rom_block = "ROM_CELL XYZ01" *) reg [3:0] rom[0:7];
   // (* synthesis, logic_block *) reg [3:0] rom [0:7];

   initial begin
      rom[0] = 4'b1011;
      rom[1] = 4'b0001;
      rom[2] = 4'b0011;
      rom[3] = 4'b0010;
      rom[4] = 4'b1110;
      rom[5] = 4'b0111;
      rom[6] = 4'b0101;
      rom[7] = 4'b0100;
```



```
    end

    assign Z = rom[a];
endmodule
```

NOTE—If combinational logic is desired instead of a ROM, specify the attribute *logic_block* instead of the attribute *rom_block*.

5.6.3 Using two-dimensional array with data in text file

The modeling of the ROM shall be identical to that in 5.6.2 except that the ROM is initialized from a text file using the system tasks $readmemb and $readmemh.

NOTE—The name and format of the file are identified by the system tasks $readmemb or $readmemh.

Example 26:

```
module rom_2dimarray_initial_readmem (
      output wire [3:0] z,
      input wire [2:0] a);
      // Declare a memory rom of 8 4-bit registers.
      // The indices are 0 to 7:
   (* synthesis, rom_block = "ROM_CELL XYZ01" *) reg [3:0] rom[0:7];

   initial $readmemb("rom.data", rom);

   assign z = rom[a];
endmodule

// Example of content “rom.data” file:
   // file: /user/name/project/design/rom/rom.data
   // date : Jan 08, 02
   1011     // addr=0
   1000     // addr=1
   0000     // addr=2
   1000     // addr=3
   0010     // addr=4
   0101     // addr=5
   1111     // addr=6
   1001     // addr=7
```

NOTE—This style can lead to simulation/synthesis mismatch if the content of data file changes after synthesis.

5.7 Modeling random access memories (RAM)

A RAM shall be modeled using a Verilog memory (a two-dimensional reg array) that has the attribute *ram_block* associated with it. A RAM element may either be modeled as an edge-sensitive storage element or as a level-sensitive storage element. A RAM data value may be read synchronously or asynchronously.

Example 27:

```
// A RAM element is an edge-sensitive storage element:
   module ram_test(
      output wire [7:0] q,
      input wire [7:0] d,
      input wire [6:0] a,
      input wire clk, we);
   (* synthesis, ram_block *) reg [7:0] mem [127:0];

   always @(posedge clk) if (we) mem[a] <= d;

   assign q = mem[a];
endmodule
```

Example 28:

```
// A RAM element is a level-sensitive storage element:
module ramlatch (
      output wire [7:0] q, // output
      input  wire [7:0] d, // data input
      input  wire [6:0] a, // address
      input  wire       we); // clock and write enable
   // Memory 128 deep, 8 wide:
   (* synthesis, ram_block *) reg [7:0] mem [127:0];

   always @* if (we) mem[a] <= d;

   assign q = mem[a];
endmodule
```

NOTES

1—If latch or register logic is desired instead of a RAM, use the attribute *logic_block* instead of the attribute *ram_block*.

2—In the absence of either a *ram_block* or a *logic_block* attribute, a synthesis tool may implement memory as random logic or as a RAM.

6. Pragmas

A *pragma* is a generic term used to define a construct with no predefined language semantics that influences how a synthesis tool should synthesize Verilog HDL code into a circuit. The only standard pragma style that shall appear with the Verilog HDL code is a Verilog attribute instance.

6.1 Synthesis attributes

NOTES

1—An attribute instance, as defined by the Verilog standard, is a set of one or more comma separated attributes, with or without assignment to the attribute, enclosed within the reserved (* and *) Verilog tokens.

2—Per the Verilog standard, "An attribute instance can appear in the Verilog 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 Verilog function name in an expression."

If a synthesis tool supports pragmas to control the structure of the synthesized netlist or to give direction to the synthesis tool, attributes shall be used to convey the required information. The first attribute within the attribute instance shall be *synthesis* followed by a comma separated list of synthesis-related attributes. Here is the template for specifying such an attribute.

```
(* synthesis, <attribute=value_or_optional_value>
              { , <attribute=value_or_optional_value> } *)
```

The attribute *synthesis* shall be listed as the first attribute in an attribute instance.

NOTE—By placing the *synthesis* attribute first, a synthesis tool can more easily parse the attribute instance to determine if the rest of the attributes in the attribute instance are intended for the synthesis tool or for a different tool.

If the attribute has an <optional_value>, such an attribute may be disabled (turned off) by providing a value of 0. If an attribute has a non-zero value (including a string value), it shall be interpreted as an enabled attribute. Additional semantics for a non-zero value are not defined by this standard. If no value is provided, then the attribute is enabled (as if the value is non-zero). The <optional_value>, if provided, shall be a constant expression.

If the attribute has a <value>, then a value shall be required for this attribute.

The following is the list of synthesis attributes that shall be supported as part of this standard and their functionality is described in the remainder of this clause. Additional vendor-specific attributes and attribute values may exist.

```
(* synthesis, async_set_reset [="signal_name1, signal_name2, ..."] *)
(* synthesis, black_box             [ =<optional_value> ] *)
(* synthesis, combinational         [ =<optional_value> ] *)
(* synthesis, fsm_state             [ =<encoding_scheme> ] *)
(* synthesis, full_case             [ = <optional_value> ] *)
(* synthesis, implementation        = "<value>" *)
(* synthesis, keep                  [ =<optional_value> ] *)
(* synthesis, label                 = "name" *)
(* synthesis, logic_block           [ = <optional_value> ] *)
(* synthesis, op_sharing            [ = <optional_value> ] *)
(* synthesis, parallel_case         [ = <optional_value> ] *)
(* synthesis, ram_block             [ = <optional_value> ] *)
(* synthesis, rom_block             [ = <optional_value> ] *)
(* synthesis, sync_set_reset  [="signal_name1, signal_name2, ..."] *)
(* synthesis, probe_port            [ = <optional_value> ] *)
```

Multiple comma separated synthesis attributes may be added to the same attribute instance without repeating the keyword *synthesis* before each additional attribute.

Example 29:

```
(* synthesis, full_case, parallel_case *)
case (state)
   ...
endcase
```

NOTES

1—The use of the *full_case* and *parallel_case* attributes is generally not recommended.

2—The LRM also allows multiple attribute instances to be placed before legal, attribute-prefixed statements.

Example 30:

```
(* synthesis, full_case *)
(* synthesis, parallel_case *)
case (state)
   ...
endcase
```

Only synthesis attributes shall be placed in an (single) attribute instance with other synthesis attributes. Non-synthesis attribute instances may be placed along with synthesis attribute instances before legal attribute prefixed statements and no predetermined placement-order of mixed synthesis and non-synthesis attribute instances shall be imposed by this standard.

NOTES

1—It is recommended that if a synthesis tool supports attributes other than those listed as part of this standard, then the syntax for specifying such an attribute be identical with the format described in this clause.

2—It is recommended that a synthesis tool not use the *synthesis* attribute in any other form or meaning other than its intended use as described in this standard.

6.1.1 Case decoding attributes

The following attributes shall be supported for decoding case statements.

6.1.1.1 Full case attribute

Its syntax is:

```
(* synthesis, full_case [ = <optional_value> ] *)
```

This attribute shall inform the synthesis tool that for all unspecified case choices, the outputs assigned within the case statement may be treated as synthesis don't-care assignments.

NOTES

1—This synthesis attribute provides different information to the synthesis tool than is known by the simulation tool and can cause a pre-synthesis simulation to differ with a post-synthesis simulation.

2—This synthesis attribute does not remove all latches that could be inferred by a Verilog case statement. If one or more outputs are assigned by the specified case items, but not all outputs are assigned by all of the specified case items, a latch will be inferred even if the *full_case* attribute has been added to the case statement.

3—Adding a default statement to a case statement nullifies the effect of the *full_case* attribute.

4—The use of the *full_case* synthesis attribute is generally discouraged.

6.1.1.2 Parallel case attribute

Its syntax is:

```
(* synthesis, parallel_case [ = <optional_value> ] *)
```

This attribute shall inform the synthesis tool that all case items are to be tested, even if more than one case item could potentially match the case expression.

NOTES

1—This synthesis attribute provides different information to the synthesis tool than is known by the simulation tool and can cause a pre-synthesis simulation to differ with a post-synthesis simulation.

2—Verilog case statements can have overlapping case items (a case expression could match more than one case item), and the first case item that matches the case expression will cause the statement for that case item to be executed and an implied break insures that no other case item will be tested against the case expression for the current pass through the case statement. The Verilog statement for the matched case item is the only Verilog code that will be executed during the current pass of the case statement.

3—The *parallel_case* attribute directs the synthesis tool to test each and every case item in the case statement every time the case statement is executed. This attribute causes the synthesis tool to remove any priority that might be assigned to the case statement by testing every case item, even if more than one case item matches the case expression. This behavior differs from the behavior of standard Verilog simulation.

4—The *parallel_case* attribute is commonly used to remove priority encoders from the gate-level implementation of an RTL case statement. Unfortunately, the RTL case statement may still simulate like a priority encoder, causing a mismatch between pre-synthesis and post-synthesis simulations.

5—Adding a default statement to a case statement does not nullify the effect of the *parallel_case* attribute.

6—The use of the *parallel_case* synthesis attribute is generally discouraged. One exception is the careful implementation of a one-hot Verilog state machine design.

6.1.1.3 Using both attributes

The syntax is:

```
(* synthesis, full_case, parallel_case *)
```

The *full_case* and *parallel_case* attributes may also appear as a single attribute instance, as shown above. The order in which they appear shall not be of importance.

NOTE—Strictly speaking, *full_case* should not be needed by any tool. It's purpose is to communicate to the tool some information which is also available from alternative modeling styles. The risk is that the user could be wrong about the 'fullness' of the case, and, if so, the results will not match simulation. For example,

```
always @(sel)
   (* synthesis, full_case *) case (sel)
      2'b01: out = op1;
      2'b10: out = op2;
      2'b11: out = op3;
   endcase
```

is synthesis-equivalent to the much safer:

```
always @(sel) begin
   out = 'bx;
   case (sel)
      2'b01: out = op1;
      2'b10: out = op2;
      2'b11: out = op3;
   endcase
end
```

6.1.2 RAM/ROM inference attributes

6.1.2.1 RAM attribute

The attribute described shall be supported to assist in the selection of the style of an inferred RAM device.

The syntax is:

```
(* synthesis, ram_block [ = <optional_value> ] *)
```

6.1.2.2 ROM attribute

The attribute described shall be supported to assist in the selection of the style of an inferred ROM device.

The syntax is:

```
(* synthesis, rom_block [ = <optional_value> ] *)
```

6.1.2.3 Logic block attribute

The attribute described shall be supported to assist in inferring discrete logic for a particular RTL coding style as opposed to a ROM or a RAM.

The syntax is:

```
(* synthesis, logic_block [ = <optional_value> ] *)
```

NOTE—Examples of these attributes appear in 5.6 and 5.7.

6.1.3 FSM attributes

These attributes apply to finite state machine (FSM) extraction. FSM extraction is the process of extracting a state transition table from an RTL model where the hardware advances from state to state at a clock edge. In such a case, it may be necessary to guide the synthesis tool in identifying the state register explicitly and to provide a mechanism to override the default encoding if necessary.

If a synthesis tool supports FSM extraction, then the following attribute shall also be supported.

```
(* synthesis, fsm_state [=<encoding_scheme>] *) // Applies to a reg.
```

The attribute when applied to a reg identifies the reg as the state vector.

The *encoding_scheme* is optional. If no encoding is specified, the default encoding as specified in the model is used. The value of *encoding_scheme* is not defined by this standard.

NOTE—Use of encoding scheme may cause simulation mismatches.

Example 31:

```
(* synthesis, fsm_state *) reg [4:0] next_state;
// Default encoding is used and next_state is the state vector.
(* synthesis, fsm_state = "onehot" *) reg [7:0] rst_state;
// "onehot" encoding is used and rst_state is the state vector.
```


6.1.4 Miscellaneous attributes

6.1.4.1 Asynchronous set reset attribute

The syntax is:

```
(* synthesis, async_set_reset [ = "signal_name1, signal_name2, ..." ] *)
```

This attribute shall apply to an always block that infers level-sensitive storage devices. If no level-sensitive storage devices are inferred for the block, a warning shall be issued.

This attribute shall also apply to a module in which case, it shall apply to all always blocks in that module. If no level-sensitive storage devices are inferred for the block, a warning shall be issued.

The presence of the attribute shall cause the set/reset logic to be applied directly to the set/reset terminals of a level-sensitive storage device if such a device is available in the technology library.

NOTE—Definitions: Set logic—the logic that sets the output of storage device to 1; reset logic—the logic that sets the output of storage device to 0.

When no signal names are specified in the attribute instance, both set and reset logic signals shall be applied directly to the set/reset terminals of a level-sensitive storage device.

When signal names are specified, only the specified signals shall be connected to the set/reset terminals (others are connected through the data input of the level-sensitive storage device).

Example 32:

```
(* synthesis, async_set_reset = “set” *)
always @(*)
   if (reset)
      qlatch <= 0;
   else if (set)
      qlatch <= 1;
   else if (enable)
      qlatch <= data;
// reset and enable logic connect through the data input.
```

6.1.4.2 Black box attribute

The syntax is:

```
(* synthesis, black_box [ = <optional_value>] *)
```

This attribute shall apply to a module instance or to a module in which case the attribute shall apply to all its module instances.

Only the module’s interface shall be defined for synthesis. The module itself may be empty or may contain non-synthesizable statements. It may also refer to an external implementation, for example, in an EDIF file. Such a black box shall not be optimized during synthesis.

Example 33:

```
(* synthesis, black_box *)
module add2 (dataa, datab, cin, result, cout, overflow);
   input [7:0] dataa;
   input [7:0] datab;
   input cin;
   output [7:0] result;
   output cout;
   output overflow;
endmodule
```

Example 34:

```
(* synthesis, black_box *)
   (* // Following are non-standard synthesis attributes:
   LPM_WIDTH = 8,
   LPM_DIRECTION = "ADD",
   ONE_INPUT_IS_CONSTANT = "NO",
   LPM_HINT = "SPEED",
   LPM_TYPE = "LPM_ADD_SUB"
*)
module add2 (
   input [7:0] dataa,
   input [7:0] datab,
   input cin,
   output [7:0] result,
   output cout,
   output overflow);
endmodule
```

6.1.4.3 Combinational attribute

The syntax is:

```
(* synthesis, combinational [ =<optional_value>] *)
```

This attribute shall be applied to an always block or to a module in which case, it shall apply to all always blocks in that module.

The attribute indicates that the logic generated from the always block shall be combinational. It shall be an error if it is not so.

Example 35:

```
(* synthesis, combinational *)
always @(*)
   if (reset)
      q = 0;
   else
      q = d;
```


6.1.4.4 Implementation attribute

The syntax is:

```
(* synthesis, implementation = "<value>" *)
```

This attribute shall apply only to an operator.

The "value" is not defined by the standard. Examples of "value" are "cla" for +, "wallace" for *.

Example 36:

```
assign x = a + (* synthesis, implementation = "ripple" *) b;
```

NOTE—The implementation is only a recommendation to the synthesis tool.

6.1.4.5 Keep attribute

The syntax is:

```
(* synthesis, keep [ =<optional_value> ] *)
```

This attribute shall apply to a net, reg or a module instance or to a module.

With the presence of this attribute on an instance or module, the instance or module shall be preserved, and not deleted nor replicated, even if the outputs of the module are not connected. The internals of the instance or the module shall not be subject to optimization.

Similarly, a net with such an attribute shall be preserved.

If a reg has a keep attribute and an *fsm_state* attribute, the *fsm_state* attribute shall be ignored. This attribute does not apply if the reg with the *fsm_state* attribute, has not been inferred as an edge-sensitive storage device.

Example 37:

```
(* synthesis, keep *) wire [2:0] opcode;

(* synthesis, keep *) add2 a1 (.dataa(da), .datab(db), .cin(carry),
   .result(), .cout(), .overflow(nextstage));
(* synthesis, keep *) reg [3:0] count_state;
(* synthesis, keep *) wire [7:0] outa; // default keep is keep = 1.

(* synthesis, keep *) reg [7:0] b;

(* synthesis, keep = 1 *) my_design my_design1 (out1, in1, clkin);
// Preserve the instance and its subelements from optimization.

(* synthesis, keep = 0 *) my_design my_design2 (out, in, clkin);
// This instance may be optimized away.
```

Example 38:

```
(* synthesis, keep *)
module count (reset, clk, counter, flag);
   . . .
   always @(posedge clk)
      if (reset) begin
         counter <= 0;
         flag <= FALSE;
      end
      else
         counter <= counter + 1;
         flag <= counter > 10 ? TRUE : FALSE;
      end
   end
endmodule
// All instances of module count is preserved.
```

NOTE—Objects connected to a *keep* net do not need to be kept unless the objects have a *keep* attribute on them. A warning may be issued by a synthesis tool for a *keep* net that has no objects connected to it.

6.1.4.6 Label attribute

The syntax is:

```
(* synthesis, label = "name" *)
```

This attribute shall apply to any item that can be attributed.

This attribute shall assign a name to the attributed item. By doing so, other attributes or tool-specific attributes can be used to reference such an item.

Example 39:

```
(* synthesis, label = "incrementor1" *) counter = counter + 1;

a = b * (* synthesis, label = "mult1" *) c
      * (* synthesis, label = "mult2" *) d;
```

NOTE—The use of a label attribute is not defined by the standard. The attribute provides a standard way to label a sentence or an item.

6.1.4.7 Operator sharing attribute

The syntax is:

```
(* synthesis, op_sharing [=<optional_value>] *)
```

This attribute shall apply to a module.

The presence of the attribute enables operator sharing to be performed in that module (and all its instances).

NOTES

1—Operator sharing technique allows the use of one arithmetic logic unit to perform same or different operations that are mutually exclusive.

2—The presence of the attribute does not guarantee that operator sharing will take place; it is only enabled. Sharing occurs based on design cost specifications.

3—Sharing may be done across always blocks.

4—In the absence of the attribute, a synthesis tool may still perform sharing.

Example 40:

```
(* synthesis, op_sharing = 1 *)
module ALU (
      input [3:0] a, b,
      input [1:0] op_code,
      output [3:0] alu_out);
   always @(*)
      case (op_code)
         ADD: alu_out = a + b;
         SUB: alu_out = a - b;
         GT : alu_out = a > b;
         default : alu_out = 4'bz;
      endcase
endmodule
```

6.1.4.8 Synchronous set reset attribute

The syntax is:

```
(* synthesis, sync_set_reset [= "signal_name1, signal_name2, ..."] *)
```

This attribute shall apply to an always block that infers edge-sensitive storage devices. If no edge-sensitive storage device is inferred in the block, a warning shall be issued.

This attribute shall also apply to a module in which case, it shall apply to all always blocks in that module. If no edge-sensitive storage devices are inferred for the block, a warning shall be issued.

The presence of the attribute shall cause the set/reset logic to be applied directly to the set/reset terminals of an edge-sensitive storage device if such a device is available in the technology library.

It is an error if the attribute is applied to an asynchronous set or reset signal.

NOTE—Definitions: Set logic—the logic that sets the output of storage device to 1; reset logic—the logic that sets the output of storage device to 0.

When no signal names are present, both set and reset logic signals shall be applied directly to the set/reset terminals of an edge-sensitive storage device.

When signal names are present, only the specified signals shall be connected to the set/reset terminals (others are connected through the data input of the edge-sensitive storage device).

Example 41:

```
(* synthesis, sync_set_reset *)
always @(posedge clk)
   if (rst)
      q <= 0;
   else if (set)
      q <= 1;
   else
      q <= d;
```

6.1.4.9 Probe port attribute

The syntax is:

```
(* synthesis, probe_port [ = <optional_value> ] *)
```

This attribute shall apply to a net or a reg. The net or reg shall only be a single bit or a 1-dimensional array.

The presence of this attribute preserves the net or the reg for probing and shall cause it to appear as an output port (a probe port) in the module it appears. If a module with a probe port is instantiated in another module, a new probe port shall also be created (one for each instance) in the parent module.

If an object with the *probe_port* attribute is optimized out, that object shall not be mapped onto a port, unless the object has an additional *keep* attribute on it. The appearance or omission of a *probe_port* as a result of optimization may be reported by the synthesis tool.

The name of the *probe_port* is not specified by this standard (may be determined by the synthesis tool). All newly created probe ports shall appear in the synthesized netlist at the end of the module port list. The order of the probe ports itself is not specified by this standard.

Example 42:

```
(* synthesis, probe_port *) reg [3:0] current_state;

(* synthesis, probe_port = 1 *) wire q0, q1, q2;
```

Example 43:

```
module ff (q, d, clk, rst);
   parameter WIDTH = 1;
   parameter PROBE_PORT = 1; // 1 => ON, 0 => OFF
   output [WIDTH-1:0] q; // output
   input [WIDTH-1:0] d; // data input
   input clk; // clock
   input rst; // reset, active hi
   reg [WIDTH-1:0] q; // FF output

   (* synthesis, keep, // Do not remove in optimization.
      probe_port = PROBE_PORT *) // Bring to a test port.
   wire [WIDTH-1:0] qbar; // Test point.
   assign qbar = ~q; // Equation for test point.

   always @(posedge clk or posedge rst)
```

```
      if (rst) q <= {WIDTH{1'b0}};
      else q <= d;
endmodule //ff

module top (q, d, clk, rst);
   parameter WIDTH = 2;
   parameter WIDTH_ONE = 1;
   parameter PROBE_PORT_ON = 1;
   parameter PROBE_PORT_OFF = 0;

   output [WIDTH-1:0] q;   // output
   input [WIDTH-1:0] d;    // data input
   input clk;              // clock
   input rst;              // reset, active hi

   // ff #(.WIDTH (1),
   // .PROBE_PORT (1))
   ff #(WIDTH_ONE, PROBE_PORT_ON) // Bring probe port out.
      ff_1 (
         // Outputs
         .q (q[0]),
         // Inputs
         .d (d[0]),
         .clk (clk),
         .rst (rst));

   // ff #(.WIDTH (1),
   // .PROBE_PORT (0))
   ff #(WIDTH_ONE, PROBE_PORT_OFF) // Do NOT bring probe port out.
      ff_2 (
         // Outputs
         .q (q[1]),
         // Inputs
         .d (d[1]),
         .clk (clk),
         .rst (rst));
endmodule // top
```

NOTES

1—This attribute is needed for the verification of gate-level model designs at the "grey-box" level where internal signals may be needed for triggering of events in a verifier (example, the occurrence of a simulation push/pop of a fifo). It may also be needed for hardware debugging when a difficult bug occurs.

2—Since this attribute creates additional ports in the synthesized logic, testbench reuse and verification (see Clause 4) may be an issue.

6.2 Compiler directives and implicit-synthesis defined macros

A synthesis tool shall define a Verilog macro definition for the macro named *SYNTHESIS* before reading any Verilog synthesis source files. This is equivalent to adding the following macro definition to the front of a Verilog input stream:

```
`define SYNTHESIS
```

NOTE—This macro definition makes it possible for Verilog users to add conditionally compiled code to their design that will be read and interpreted by synthesis tools but that by default will be ignored by simulators (unless the Verilog simulation input stream also defines the *SYNTHESIS* text macro).

Example 44:

```
module ram (q, d, a, clk, we);
   output [7:0] q;
   input  [7:0] d;
   input  [6:0] a;
   input  clk, we;

   `ifndef SYNTHESIS
      // RTL model of a ram device for pre-synthesis simulation
      reg [7:0] mem [127:0];
      always @(posedge clk) if (we) mem[a] <= d;

      assign q = mem[a];
   `else
      // Instantiation of an actual ram block for synthesis
      xram ram1 (.dout(q), .din(d), .addr(a), .ck(clk), .we(we));
   `endif
endmodule
```

NOTE—The use of the above conditional compilation capability removes the need to use the deprecated *translate_off/translate_on* synthesis pragmas.

6.3 Deprecated features

Current common practices (prior to this standard) of using meta-comments and *translate_off/translate_on* pragmas shall not be supported by this standard.

6.3.1 Meta-comments deprecated

Prior to the acceptance of the Verilog IEEE Std 1364-2001, it was common practice to include synthesis pragmas embedded within a comment, for example: // *synthesis full_case*. The practice of embedding pragmas into a comment meant that any synthesis tool that accepted such pragmas was required to partially or fully parse all comments within a Verilog RTL design just to determine if the comment contained a pragma for the synthesis tool.

The Verilog standard introduced attributes to discourage the practice of putting pragmas into comments and to replace them with a set of tokens (attribute delimiters) that could then be parsed for tool-specific information.

The practice of putting pragmas into comments is highly discouraged and deprecated for this standard.

6.3.2 "translate_off/translate_on" pragmas deprecated

Prior to this standard, it was common practice to include *translate_off/translate_on* pragmas to instruct the synthesis tool to ignore a block of Verilog code enclosed by these pragmas.

The practice of a synthesis tool ignoring Verilog source code by enclosing the code within *translate_off/translate_on* pragmas is highly discouraged and deprecated for this standard. Users are encouraged to take advantage of *SYNTHESIS* macro definition and ***`ifdef*** *SYNTHESIS* and ***`ifndef*** *SYNTHESIS* compiler directives (see 6.2) to exclude blocks of Verilog code from being read and compiled by synthesis tools.

7. Syntax

NOTE—The subclauses within this clause are described using the same section hierarchy as described in the IEEE Std 1364-2001 LRM. This enables cross-referencing between the two standards to be much easier.

7.1 Lexical conventions

7.1.1 Lexical tokens

Supported.

7.1.2 White space

Supported.

7.1.3 Comments

Supported.

7.1.4 Operators

Supported.

7.1.5 Numbers

number ::=
    decimal_number
  | octal_number
  | binary_number
  | hex_number
  ~~| real_number~~

~~real_number~~ ::=
    unsigned_number **.** unsigned_number
  | unsigned_number [**.** unsigned_number] exp [sign] unsigned_number

~~exp~~ ::= **e** | **E**

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 ::= non_zero_decimal_digit { **_** | decimal_digit } unsigned_number ::= decimal_digit { **_** | decimal_digit }

binary_value ::= binary_digit { **_** | binary_digit }

octal_value ::= octal_digit { **_** | octal_digit }

hex_value ::= hex_digit { **_** | hex_digit }

decimal_base ::= **’**[**s**|**S**]**d** | **’**[**s**|**S**]**D**

binary_base ::= **’**[**s**|**S**]**b** | **’**[**s**|**S**]**B**

octal_base ::= **’**[**s**|**S**]**o** | **’**[**s**|**S**]**O**

hex_base ::= **’**[**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** | **?**

7.1.5.1 Integer constants

Supported. See 5.5 on usage of x_digit and z_digit.

7.1.5.2 Real constants

Not supported.

7.1.5.3 Conversion

Not supported.

7.1.6 Strings

Supported.

7.1.6.1 String variable declaration

Supported.

7.1.6.2 String manipulation

Supported.

7.1.6.3 Special characters in strings

Supported.

7.1.7 Identifiers, keywords, and system names

Simple identifiers are supported.

7.1.7.1 Escaped identifiers

Supported.

7.1.7.2 Generated identifiers

Not supported.

7.1.7.3 Keywords

Supported.

7.1.7.4 System tasks and functions

system_task_enable ::=
    system_task_identifier [**(** expression { **,** expression } **)**] **;**

~~system_function_call~~ ::= system_function_identifier
    [**(** expression { **,** expression } **)**]

~~system_function_identifier~~ ::= **$**[**a-zA-Z0-9_$**]{[**a-zA-Z0-9_$**]}

system_task_identifier ::= **$**[**a-zA-Z0-9_$**]{[**a-zA-Z0-9_$**]}

System task enable shall be ignored. System function call shall not be supported.

7.1.7.5 Compiler directives

Supported. See 7.17 for more detail.

7.1.8 Attributes

attribute_instance ::= **(*** attr_spec { **,** attr_spec } ***)**

attr_spec ::=
    attr_name **=** constant_expression
    | attr_name

attr_name ::= identifier

module_declaration ::=
    { attribute_instance } module_keyword module_identifier [module_parameter_port_list]
        [list_of_ports] **;** { module_item }
    endmodule
    | { attribute_instance } module_keyword module_identifier [module_parameter_port_list]
        [list_of_port_declarations] **;** { non_port_module_item }
    endmodule

```
port_declaration ::=
          { attribute_instance } inout_declaration
        | { attribute_instance } input_declaration
        | { attribute_instance } output_declaration

module_item ::=
          module_or_generate_item
        | port_declaration ;
        | { attribute_instance } generated_instantiation
        | { attribute_instance } local_parameter_declaration
        | { attribute_instance } parameter_declaration ;
        | { attribute_instance } specify_block
        | { attribute_instance } specparam_declaration

module_or_generate_item ::=
          { attribute_instance } module_or_generate_item_declaration
        | { attribute_instance } parameter_override
        | { attribute_instance } continuous_assign
        | { attribute_instance } gate_instantiation
        | { attribute_instance} ~~udp_instantiation~~
        | { attribute_instance } module_instantiation
        | { attribute_instance } initial_construct
        | { attribute_instance } always_construct

non_port_module_item ::=
          { attribute_instance } generated_instantiation
        | { attribute_instance } local_parameter_declaration
        | { attribute_instance } module_or_generate_item
        | { attribute_instance } parameter_declaration ;
        | { attribute_instance } specify_block
        | { attribute_instance } specparam_declaration

function_port_list ::=
          { attribute_instance } tf_input_declaration { , { attribute_instance } tf_input_declaration }

task_item_declaration ::=
          block_item_declaration
        | { attribute_instance } tf_input_declaration ;
        | { attribute_instance } tf_output_declaration ;
        | { attribute_instance } tf_inout_declaration ;

task_port_item ::=
          { attribute_instance } tf_input_declaration
        | { attribute_instance } tf_output_declaration
        | { attribute_instance } tf_inout_declaration

block_item_declaration ::=
          { attribute_instance } block_reg_declaration
        ~~| { attribute_instance } event_declaration~~
        | { attribute_instance } integer_declaration
        | { attribute_instance } local_parameter_declaration
        | { attribute_instance } parameter_declaration ;
        ~~| { attribute_instance } real_declaration~~
        ~~| { attribute_instance } realtime_declaration~~
        ~~| { attribute_instance } time_declaration~~

ordered_port_connection ::= { attribute_instance } [ expression ]
```


named_port_connection ::= { attribute_instance } **.** port_identifier **(** [expression] **)**

~~udp_declaration~~ ::=
{ attribute_instance } **primitive** udp_identifier **(** udp_port_list **) ;**
udp_port_declaration { udp_port_declaration }
udp_body
endprimitive
| { attribute_instance } **primitive** udp_identifier **(** udp_declaration_port_list **) ;**
udp_body
endprimitive

~~udp_output_declaration~~ ::=
{ attribute_instance } **output** port_identifier
| { attribute_instance } **output reg** port_identifier [**=** constant_expression]

~~udp_input_declaration~~ ::={ attribute_instance } **input** list_of_port_identifiers

~~udp_reg_declaration~~ ::= { attribute_instance } **reg** variable_identifier

function_statement_or_null ::=
function_statement
| { attribute_instance } **;**

statement ::=
{ attribute_instance } blocking_assignment **;**
| { attribute_instance } case_statement
| { attribute_instance } conditional_statement
| { attribute_instance } disable_statement
~~| { attribute_instance } event_trigger~~
| { attribute_instance } loop_statement
| { attribute_instance } nonblocking assignment **;**
~~| { attribute_instance } par_block~~
~~| { attribute_instance } procedural_continuous_assignments ;~~
| { attribute_instance } procedural_timing_control_statement
| { attribute_instance } seq_block
| { attribute_instance } system_task_enable
| { attribute_instance } task_enable
~~| { attribute_instance } wait_statement~~

statement_or_null ::=
statement
| { attribute_instance } **;**

function_statement ::=
{ attribute_instance } function_blocking_assignment **;**
| { attribute_instance } function_case_statement
| { attribute_instance } function_conditional_statement
| { attribute_instance } function_loop_statement
| { attribute_instance } function_seq_block
| { attribute_instance } disable_statement
| { attribute_instance } system_task_enable

constant_function_call ::= function_identifier { attribute_instance }
(constant_expression { **,** constant_expression } **)**

function_call ::= hierarchical_function_identifier { attribute_instance }
(expression { **,** expression } **)**

genvar_function_call ::= genvar_function_identifier { attribute_instance }
    (constant_expression { **,** constant_expression } **)**

conditional_expression ::= expression1 **?** { attribute_instance }expression2 **:** expression3

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
    | string

expression ::=
    primary
    | unary_operator { attribute_instance } primary
    | expression binary_operator { attribute_instance } expression
    | conditional_expression
    | string

module_path_conditional_expression ::=
    module_path_expression **?** { attribute_instance }
        module_path_expression **:** module_path_expression

module_path_expression ::=
    module_path_primary
    | unary_module_path_operator { attribute_instance } module_path_primary
    | module_path_expression binary_module_path_operator
        { attribute_instance } module_path_expression
    | module_path_conditional_expression

The set of predefined attributes that shall be supported are described in 6.1.

7.2 Data types

7.2.1 Value set

Supported. See 5.5 on support for values ***x*** *and* ***z****.*

7.2.2 Nets and variables

7.2.2.1 Net declarations

net_declaration ::=
    net_type [**signed**] [delay3] list_of_net_identifiers **;**
    | net_type [drive_strength] [**signed**] [delay3]
        list_of_net_decl_assignments **;**
    | net_type [**vectored** | **scalared**] [**signed**] range [delay3]
        list_of_net_identifiers **;**
    | net_type [drive_strength] [**vectored** | **scalared**] [**signed**] range
        [delay3] list_of_net_decl_assignments **;**
    ~~| **trireg** [charge_strength] [**signed**] [delay3] list_of_net_identifiers **;**~~
    ~~| **trireg** [drive_strength] [**signed**] [delay3]~~
        ~~list_of_net_decl_assignments **;**~~
    ~~| **trireg** [charge_strength] [**vectored** | **scalared**] [**signed**] range~~
        ~~[delay3] list_of_net_identifiers **;**~~
    ~~| **trireg** [drive_strength] [**vectored** | **scalared**] [**signed**] range~~
        ~~[delay3] list_of_net_decl_assignments **;**~~

net_type ::=
    supply0 | **supply1**
  | **tri** | **triand** | **trior** | ~~**tri0**~~ | ~~**tri1**~~
  | **wire** | **wand** | **wor**

drive_strength ::=
    (strength0 **,** strength1 **)**
  | **(** strength1 **,** strength0 **)**
  | **(** strength0 **, highz1)**
  | **(** strength1 **, highz0)**
  | **(highz1 ,** strength0 **)**
  | **(highz0 ,** strength1 **)**

strength0 ::= **supply0** | **strong0** | **pull0** | **weak0**

strength1 ::= **supply1** | **strong1** | **pull1** | **weak1**

charge_strength ::= **(small)** | **(medium)** | **(large)**

delay3 ::= **#** delay_value | **# (** delay_value [**,** delay_value [**,** delay_value]] **)**

delay2 ::= **#** delay_value | **# (** delay_value [**,** delay_value] **)**

delay_value ::=
    unsigned_number
  | parameter_identifier
  | specparam_identifier
  | mintypmax_expression

list_of_net_decl_assignments ::= net_decl_assignment { **,** net_decl_assignment }

list_of_net_identifiers ::= net_identifier [dimension { dimension }]
  { **,** net_identifier [dimension { dimension }] }

net_decl_assignment ::= net_identifier **=** expression

dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**

range ::= **[** msb_constant_expression **:** lsb_constant_expression **]**

7.2.2.2 Variable declarations

integer_declaration ::= **integer** list_of_variable_identifiers **;**

~~real_declaration~~ ::= **real** list_of_real_identifiers **;**

~~realtime_declaration~~ ::= **realtime** list_of_real_identifiers **;**

reg_declaration ::= **reg** [**signed**] [range] list_of_variable_identifiers **;**

~~time_declaration~~ ::= **time** list_of_variable_identifiers **;**

~~real_type~~ ::=
    real_identifier [**=** constant_expression]
  | real_identifier dimension { dimension }

variable_type ::=
  variable_identifier [= constant_expression]
  | variable_identifier dimension { dimension }

~~list_of_real_identifiers~~ ::= real_type { **,** real_type }

list_of_variable_identifiers ::= variable_type { **,** variable_type }

dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**

range ::= **[** msb_constant_expression **:** lsb_constant_expression **]**

7.2.3 Vectors

Supported.

7.2.4 Strengths

7.2.4.1 Charge strength

Ignored.

7.2.4.2 Drive strength

Ignored.

7.2.5 Implicit declarations

Supported.

7.2.6 Net initialization

Not supported.

7.2.7 Net types

7.2.7.1 Wire and tri nets

Supported.

7.2.7.2 Wired nets

Supported.

7.2.7.3 Trireg net

Not supported.

7.2.7.4 Tri0 and tri1 nets

Not supported.

7.2.7.5 Supply nets

Supported.

7.2.7.6 regs

Supported. See Clause 5 on how edge-sensitive and level-sensitive storage devices are inferred.

7.2.8 Integers, reals, times and realtimes

integer_declaration ::= **integer** list_of_variable_identifiers **;**

~~real_declaration~~ ::= **real** list_of_real_identifiers **;**

~~realtime_declaration~~ ::= **realtime** list_of_real_identifiers **;**

~~time_declaration~~ ::= **time** list_of_variable_identifiers **;**

~~real_type~~ ::=
    real_identifier [**=** constant_expression]
  | real_identifier dimension { dimension }

variable_type ::=
    variable_identifier [= constant_expression]
  | variable_identifier dimension { dimension }

~~list_of_real_identifiers~~ ::= real_type { **,** real_type }

list_of_variable_identifiers ::= variable_type { **,** variable_type }

dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**

7.2.8.1 Operators and real numbers

Not supported.

7.2.8.2 Conversion

Not supported.

7.2.9 Arrays

Supported.

7.2.9.1 Net arrays

Supported.

7.2.9.2 reg and variable arrays

Supported.

7.2.9.3 Memories

Supported.

7.2.10 Parameters

7.2.10.1 Module parameters

local_parameter_declaration ::=
    localparam [**signed**] [range] list_of_param_assignments **;**
    | **localparam integer** list_of_param_assignments **;**
    ~~| **localparam real** list_of_param_assignments ;~~
    ~~| **localparam realtime** list_of_param_assignments ;~~
    ~~| **localparam time** list_of_param_assignments ;~~

parameter_declaration ::=
    parameter [**signed**] [range] list_of_param_assignments
    | **parameter integer** list_of_param_assignments
    ~~| **parameter real** list_of_param_assignments~~
    ~~| **parameter realtime** list_of_param_assignments~~
    ~~| **parameter time** list_of_param_assignments~~

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

param_assignment ::= parameter_identifier **=** constant_expression

range ::= **[** msb_constant_expression **:** lsb_constant_expression **]**

7.2.10.2 Local parameters—localparam

Supported.

7.2.10.3 Specify parameters

specparam_declaration ::= **specparam** [range] list_of_specparam_assignments **;**

list_of_specparam_assignments ::=
    specparam_assignment { **,** specparam_assignment }

specparam_assignment ::=
    specparam_identifier **=** constant_mintypmax_expression
    | pulse_control_specparam

pulse_control_specparam ::=
    PATHPULSE$ = (reject_limit_value [**,** error_limit_value] **) ;**
    | **PATHPULSE$**specify_input_terminal_descriptor**$**specify_output_terminal_descriptor
        = (reject_limit_value [**,** error_limit_value] **) ;**

error_limit_value ::= limit_value

reject_limit_value ::= limit_value

limit_value ::= constant_mintypmax_expression

range ::= **[** msb_constant_expression **:** lsb_constant_expression **]**

7.2.11 Name spaces

Supported.

7.3 Expressions

7.3.1 Operators

Supported. See also subclauses that follow.

7.3.1.1 Operators with real operands

Not supported.

7.3.1.2 Binary operator precedence

Supported.

7.3.1.3 Using integer numbers in expressions

Supported.

7.3.1.4 Expression evaluation order

Supported.

7.3.1.5 Arithmetic operators

The power operator (**) shall be supported only when both operands are constants or if the first operand is 2.

7.3.1.6 Arithmetic expressions with regs and integers

Supported.

7.3.1.7 Relational operators

Supported.

7.3.1.8 Equality operators

The case equality operators === and !== shall not be supported.

7.3.1.9 Logical operators

Supported.

7.3.1.10 Bit-wise operators

Supported.

7.3.1.11 Reduction operators

Supported.

7.3.1.12 Shift operators

Supported.

7.3.1.13 Conditional operator

Supported.

7.3.1.14 Concatenations

Supported.

7.3.1.15 Event or

Supported.

7.3.2 Operands

7.3.2.1 Vector bit-select and part-select addressing

Supported.

7.3.2.2 Array and memory addressing

Supported.

7.3.2.3 Strings

Supported.

7.3.3 Minimum, typical, and maximum delay expressions

```
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
    | string

constant_mintypmax_expression ::=
      constant_expression
    | constant_expression : constant_expression : constant_expression

expression ::=
      primary
    | unary_operator { attribute_instance } primary
    | expression binary_operator { attribute_instance } expression
    | conditional_expression
    | string

mintypmax_expression ::=
      expression
    | expression : expression : expression

constant_primary ::=
      constant_concatenation
    | constant_function_call
    | ( constant_mintypmax_expression )
    | constant_multiple_concatenation
```


| genvar_identifier
| number
| parameter_identifier
| specparam_identifier

primary ::=
number
| hierarchical_identifier
| hierarchical_identifier **[** expression **]** { **[** expression **]** }
| hierarchical_identifier **[** expression **]** { **[** expression **]** }
[range_expression **]**
| hierarchical_identifier **[** range_expression **]**
| concatenation
| multiple_concatenation
| function_call
| system_function_call
| constant_function_call
| **(** mintypmax_expression **)**

7.3.4 Expression bit lengths

Supported.

7.3.5 Signed expressions

Supported. See 5.5 for handling of ***x*** *and* ***z*** *values.*

7.4 Assignments

7.4.1 Continuous assignments

net_declaration ::=
net_type [**signed**] [delay3] list_of_net_identifiers **;**
| net_type [drive_strength] [**signed**] [delay3]
list_of_net_decl_assignments **;**
| net_type [**vectored** | **scalared**] [**signed**] range [delay3]
list_of_net_identifiers **;**
| net_type [drive_strength] [**vectored** | **scalared**] [**signed**] range
[delay3] list_of_net_decl_assignments **;**
~~| **trireg** [charge_strength] [**signed**] [delay3] list_of_net_identifiers **;**~~
~~| **trireg** [drive_strength] [**signed**] [delay3]~~
~~list_of_net_decl_assignments **;**~~
~~| **trireg** [charge_strength] [**vectored** | **scalared**] [**signed**] range~~
~~[delay3] list_of_net_identifiers **;**~~
~~| **trireg** [drive_strength] [**vectored** | **scalared** } [**signed**] range~~
~~[delay3] list_of_net_decl_assignments **;**~~

list_of_net_decl_assignments ::= net_decl_assignment { **,** net_decl_assignment }

net_decl_assignment ::= net_identifier **=** expression

continuous_assign ::= **assign** [drive_strength] [delay3] list_of_net_assignments **;**

list_of_net_assignments ::= net_assignment { **,** net_assignment }

net_assignment ::= net_lvalue **=** expression

7.4.1.1 The net declaration assignment

Supported.

7.4.1.2 The continuous assignment statement

Supported.

7.4.1.3 Delays

Ignored.

7.4.1.4 Strengths

Ignored.

7.4.2 Procedural assignments

Supported.

7.4.2.1 Variable declaration assignment

Ignored.

7.4.2.2 Variable declaration syntax

integer_declaration ::= **integer** list_of_variable_identifiers **;**

~~real_declaration~~ ::= **real** list_of_real_identifiers **;**

~~realtime_declaration~~ ::= **realtime** list_of_real_identifiers **;**

reg_declaration ::= **reg** [**signed**] [range] list_of_variable_identifiers **;**

~~time_declaration~~ ::= **time** list_of_variable_identifiers **;**

~~real_type~~ ::=
    real_identifier [**=** constant_expression]
    | real_identifier dimension { dimension }

variable_type ::=
    variable_identifier [= constant_expression]
    | variable_identifier dimension { dimension }

~~list_of_real_identifiers~~ ::= real_type { **,** real_type }

list_of_variable_identifiers ::= variable_type { **,** variable_type }

7.5 Gate and switch level modeling

7.5.1 Gate and switch declaration syntax

gate_instantiation ::=
cmos_switchtype [delay3] cmos_switch_instance { , cmos_switch_instance } ;
| enable_gatetype [drive_strength] [delay3] enable_gate_instance
{ , enable_gate_instance} ;
| mos_switchtype [delay3] mos_switch_instance { , mos_switch_instance } ;
| n_input_gatetype [drive_strength] [delay2] n_input_gate_instance
{ , n_input_gate_instance } ;
| n_output_gatetype [drive_strength] [delay2] n_output_gate_instance
{ , n_output_gate_instance } ;
| pass_en_switchtype [delay3] pass_enable_switch_instance
{ , pass_enable_switch_instance } ;
| pass_switchtype pass_switch_instance { , pass_switch_instance } ;
| **pulldown** [pulldown_strength] pull_gate_instance { , pull_gate_instance } ;
| **pullup** [pullup_strength] pull_gate_instance { , pull_gate_instance } ;

cmos_switch_instance ::= [name_of_gate_instance] (output_terminal ,
input_terminal , ncontrol_terminal , pcontrol_terminal)

enable_gate_instance ::= [name_of_gate_instance] (output_terminal ,
input_terminal , enable_terminal)

mos_switch_instance ::= [name_of_gate_instance] (output_terminal ,
input_terminal , enable_terminal)

n_input_gate_instance ::= [name_of_gate_instance] (output_terminal ,
input_terminal { , input_terminal })

n_output_gate_instance ::= [name_of_gate_instance] (output_terminal { ,
output_terminal } , input_terminal)

pass_switch_instance ::= [name_of_gate_instance] (inout_terminal ,
inout_terminal)

pass_enable_switch_instance ::= [name_of_gate_instance] (inout_terminal ,
inout_terminal , enable_terminal)

pull_gate_instance ::= [name_of_gate_instance] (output_terminal)

name_of_gate_instance ::= gate_instance_identifier [range]

pulldown_strength ::=
(strength0 , strength1)
| (strength1 , strength0)
| (strength0)

pullup_strength ::=
(strength0 , strength1)
| (strength1 , strength0)
| (strength1)

enable_terminal ::= expression

inout_terminal ::= net_lvalue

input_terminal ::= expression

~~ncontrol_terminal~~ ::= expression

output_terminal ::= net_lvalue

~~pcontrol_terminal~~ ::= expression

~~cmos_switchtype~~ ::= **cmos** | **rcmos**

enable_gatetype ::= **bufif0** | **bufif1** | **notif0** | **notif1**

~~mos_switchtype~~ ::= **nmos** | **pmos** | **rnmos** | **rpmos**

n_input_gatetype ::= **and** | **nand** | **or** | **nor** | **xor** | **xnor**

n_output_gatetype ::= **buf** | **not**

~~pass_en_switchtype~~ ::= **tranif0** | **tranif1** | **rtranif1** | **rtranif0**

~~pass_switchtype~~ ::= **tran** | **rtran**

7.5.1.1 The gate type specification

The pull gates, MOS switches, and the bidirectional switches shall not be supported.

7.5.1.2 The drive strength specification

Ignored.

7.5.1.3 The delay specification

Ignored.

7.5.1.4 The primitive instance identifier

Supported.

7.5.1.5 The range specification

Supported.

7.5.1.6 Primitive instance connection list

Supported.

7.5.2 and, nand, nor, or, xor, and xnor gates

Supported.

7.5.3 buf and not gates

Supported.

7.5.4 buif1, bufif0, notif1, and notif0 gates

Supported.

7.5.5 MOS switches

Not supported.

7.5.6 Bidirectional pass switches

Not supported.

7.5.7 CMOS switches

Not supported.

7.5.8 pullup and pulldown sources

Not supported.

7.5.9 Logic strength modeling

Ignored.

7.5.10 Strengths and values of combined signals

Ignored.

7.5.11 Strength reduction by nonresistive devices

Ignored.

7.5.12 Strength reduction by resistive devices

Ignored.

7.5.13 Strengths of net types

Ignored.

7.5.14 Gate and net delays

Ignored.

7.5.14.1 min:typ:max delays

Ignored.

7.5.14.2 trireg net charge decay

Ignored.

7.6 User-defined primitives (UDPs)

Not supported.

7.7 Behavioral modeling

7.7.1 Behavioral model overview

Supported.

7.7.2 Procedural assignments

Supported.

7.7.2.1 Blocking procedural assignments

```
blocking_assignment ::=
    variable_lvalue = [ delay_or_event_control ] expression

delay_control ::=
      # delay_value
    | # ( mintypmax_expression )

delay_or_event_control ::=
      delay_control
    | event_control
    | repeat ( expression ) event_control

event_control ::=
      @ event_identifier
    | @ ( event_expression )
    | @ *
    | @ ( * )

event_expression ::=
      expression
    | hierarchical_identifier
    | posedge expression
    | negedge expression
    | event_expression or event_expression
    | event_expression , event_expression

variable_lvalue ::=
      hierarchical_variable_identifier
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
    | hierarchical_variable_identifier [ expression ] { [ expression ] }
        [ range_expression ]
    | hierarchical_variable_identifier [ range_expression ]
    | variable_concatenation
```

Note: the "| repeat (expression) event_control" alternative and the "expression" alternative of event_expression are struck through on the page.

A variable shall not be assigned using a blocking assignment and a non-blocking assignment in the same module.

Only those event expressions used in modeling hardware elements as shown in Clause 5 shall be supported.

7.7.2.2 The non blocking procedural assignment

```
nonblocking assignment ::=
    variable_lvalue <= [ delay_or_event_control ] expression
```



```
delay_control ::=
        # delay_value
      | # ( mintypmax_expression )

delay_or_event_control ::=
        delay_control
      | event_control
      | repeat ( expression ) event_control

event_control ::=
        @ event_identifier
      | @ ( event_expression )
      | @ *
      | @ ( * )

event_expression ::=
        expression
      | hierarchical_identifier
      | posedge expression
      | negedge expression
      | event_expression or event_expression
      | event_expression , event_expression

variable_lvalue ::=
        hierarchical_variable_identifier
      | hierarchical_variable_identifier [ expression ] { [ expression ] }
      | hierarchical_variable_identifier [ expression ] { [ expression ] }
          [ range_expression ]
      | hierarchical_variable_identifier [ range_expression ]
      | variable_concatenation
```

A variable shall not be assigned using a blocking assignment and a non-blocking assignment in the same module.

Only those event expressions used in modeling hardware elements as shown in Clause 5 shall be supported.

7.7.3 Procedural continuous assignments

```
net_assignment ::= net_lvalue = expression

procedural_continuous_assignments ::=
        assign variable_assignment ;
      | deassign variable_lvalue ;
      | force variable_assignment ;
      | force net_assignment ;
      | release variable_lvalue ;
      | release net_lvalue ;

variable_assignment := variable_lvalue = expression

net_lvalue ::=
        hierarchical_net_identifier
      | hierarchical_net_identifier [ constant_expression ] { [ constant_expression ] }
      | hierarchical_net_identifier [ constant_expression ] { [ constant_expression ] }
          [ constant_range_expression ]
      | hierarchical_net_identifier [ constant_range_expression ]
      | net_concatenation
```

```
variable_lvalue ::=
        hierarchical_variable_identifier
      | hierarchical_variable_identifier [ expression ] { [ expression ] }
      | hierarchical_variable_identifier [ expression ] { [ expression ] }
          [ range_expression ]
      | hierarchical_variable_identifier [ range_expression ]
      | variable_concatenation
```

7.7.3.1 The assign and deassign procedural statements

Not supported.

7.7.3.2 The force and release procedural statements

Not supported.

7.7.4 Conditional statement

```
conditional_statement ::=
        if ( expression ) statement_or_null [ else statement_or_null ]
      | if_else_if_statement

function_conditional_statement ::=
        if ( expression ) function_statement_or_null
          [ else function_statement_or_null ]
      | function_if_else_if_statement
```

7.7.4.1 If-else-if construct

```
if_else_if_statement ::=
        if ( expression ) statement_or_null
          { else if ( expression ) statement_or_null }
          [ else statement_or_null ]

function_if_else_if_statement ::=
        if ( expression ) function_statement_or_null
          { else if ( expression ) function_statement_or_null }
          [ else function_statement_or_null ]
```

7.7.5 Case statement

```
case_statement ::=
        case ( expression ) case_item { case_item } endcase
      | casez ( expression ) case_item { case_item } endcase
      | casex ( expression ) case_item { case_item } endcase

case_item ::=
        expression { , expression } : statement_or_null
      | default [ : ] statement_or_null

function_case_statement ::=
        case ( expression ) function_case_item { function_case_item } endcase
      | casez ( expression ) function_case_item { function_case_item } endcase
      | casex ( expression ) function_case_item { function_case_item } endcase
```


function_case_item ::=
expression { **,** expression } **:** function_statement_or_null
| **default** [**:**] function_statement_or_null

7.7.5.1 Case statement with don't-cares

Case expression in a casex statement shall not have an ***x*** *or a* ***z*** *(or* ***?****) value.*

Case expression in a casez statement shall not have a ***?*** *or* ***z****.*

7.7.5.2 Constant expression in case statement

Supported.

7.7.6 Looping statements

function_loop_statement ::=
~~**forever** function_statement~~
~~| **repeat (** expression **)** function_statement~~
~~| **while (** expression **)** function_statement~~
| **for (** variable_assignment **;** expression **;** variable_assignment **)**
function_statement

loop_statement ::=
~~**forever** statement~~
~~| **repeat (** expression **)** statement~~
~~| **while (** expression **)** statement~~
| **for (** variable_assignment **;** expression **;** variable_assignment **)** statement

Loop bounds shall be statically computable for a for loop.

7.7.7 Procedural timing controls

delay_control ::=
delay_value
| **#** **(** mintypmax_expression **)**

delay_or_event_control ::=
delay_control
| event_control
~~| **repeat (** expression **)** event_control~~

event_control ::=
@ event_identifier
| **@** **(** event_expression **)**
| **@** *****
| **@** **(*)**

The event control, including the implicit form, shall only be supported at the topmost statement in an always construct.

event_expression ::=
~~expression~~
| hierarchical_identifier
| **posedge** expression
| **negedge** expression
| event_expression **or** event_expression
| event_expression **,** event_expression

Only those event expressions used in modeling hardware elements as shown in Clause 5 shall be supported.

7.7.7.1 Delay control

Delay control may appear with inner statements (statements within the top-level statement (the statement with the always keyword)) but shall be ignored. Delay control shall not be allowed in the top level statement.

7.7.7.2 Event control

Only those event expressions used in modeling hardware elements as shown in Clause 5 shall be supported. Furthermore, event control shall appear only in the top-level statement (the statement with the always keyword) as described in Clause 5. Event control shall not be allowed in inner statements.

7.7.7.3 Named events

~~event_declaration~~ ::= **event** list_of_event_identifiers **;**

~~list_of_event_identifiers~~ ::= event_identifier [dimension { dimension }]
{ **,** event_identifier [dimension { dimension }] }

dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**

~~event_trigger~~ ::=
-> hierarchical_event_identifier **;**

7.7.7.4 Event or operator

Supported.

7.7.7.5 Implicit event_expression list

Supported.

7.7.7.6 Level-sensitive event control

~~wait_statement~~ ::=
wait (expression **)** statement_or_null

7.7.7.7 Intra-assignment timing controls

blocking_assignment ::=
variable_lvalue **=** [delay_or_event_control] expression

nonblocking_assignment ::=
variable_lvalue **<=** [delay_or_event_control] expression

delay_control ::=
    # delay_value
    | # (mintypmax_expression)

delay_or_event_control ::=
    delay_control
    | event_control
    ~~| **repeat** (expression) event_control~~

event_control ::=
    @ event_identifier
    | @ (event_expression)
    | @ *
    | @ (*)

The event control, including the implicit form, shall only be supported at the topmost statement in an always construct.

event_expression ::=
    ~~expression~~
    | hierarchical_identifier
    | **posedge** expression
    | **negedge** expression
    | event_expression **or** event_expression
    | event_expression **,** event_expression

Only those event expressions used in modeling hardware elements as shown in Clause 5 shall be supported.

7.7.8 Block statements

7.7.8.1 Sequential blocks

function_seq_block ::= **begin** [**:** block_identifier
    { block_item_declaration }] { function_statement } **end**

seq_block ::= **begin** [**:** block_identifier
    { block_item_declaration }] { statement } **end**

block_item_declaration ::=
    { attribute_instance } block_reg_declaration
    ~~| { attribute_instance } event_declaration~~
    | { attribute_instance } integer_declaration
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration **;**
    ~~| { attribute_instance } real_declaration~~
    ~~| { attribute_instance } realtime_declaration~~
    ~~| { attribute_instance } time_declaration~~

7.7.8.2 Parallel blocks

~~par_block~~ ::= **fork** [**:** block_identifier
    { block_item_declaration }] { statement } **join**

block_item_declaration ::=
{ attribute_instance } block_reg_declaration
~~| { attribute_instance } event_declaration~~
| { attribute_instance } integer_declaration
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
~~| { attribute_instance } real_declaration~~
~~| { attribute_instance } realtime_declaration~~
~~| { attribute_instance } time_declaration~~

7.7.8.3 Block names

Supported.

7.7.8.4 Start and finish times

Ignored.

7.7.9 Structured procedures

7.7.9.1 Initial construct

initial_construct ::= **initial** statement

The initial statement shall be supported only for ROM modeling as described in 5.6.2. It shall be ignored in all other contexts.

7.7.9.2 Always construct

always_construct ::= **always** statement

Clause 5 describes how the always construct can be used to model logic elements. Event control shall only be supported in the top-level statement.

7.8 Tasks and functions

7.8.1 Distinctions between tasks and functions

Supported.

7.8.2 Tasks and task enabling

7.8.2.1 Task declarations

task_declaration ::=
task [**automatic**] task_identifier **;**
{ task_item_declaration }
statement
endtask
| **task** [**automatic**] task_identifier **(** task_port_list **) ;**
{ block_item_declaration }
statement
endtask

task_item_declaration ::=
block_item_declaration
| { attribute_instance } tf_input_declaration **;**
| { attribute_instance } tf_output_declaration **;**
| { attribute_instance } tf_inout_declaration **;**

task_port_list ::= task_port_item { **,** task_port_item }

task_port_item ::=
{ attribute_instance } tf_input_declaration
| { attribute_instance } tf_output_declaration
| { attribute_instance } tf_inout_declaration

tf_input_declaration ::=
input [**reg**] [**signed**] [range] list_of_port_identifiers
| **input** [task_port_type] list_of_port_identifiers

tf_output_declaration ::=
output [**reg**] [**signed**] [range] list_of_port_identifiers
| **output** [task_port_type] list_of_port_identifiers

tf_inout_declaration ::=
inout [**reg**] [**signed**] [range] list_of_port_identifiers
| **inout** [task_port_type] list_of_port_identifiers

task_port_type ::=
~~**time**~~ | ~~**real**~~ | ~~**realtime**~~ | **integer**

block_item_declaration ::=
{ attribute_instance } block_reg_declaration
~~| { attribute_instance } event_declaration~~
| { attribute_instance } integer_declaration
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
~~| { attribute_instance } real_declaration~~
~~| { attribute_instance } realtime_declaration~~
~~| { attribute_instance } time_declaration~~

block_reg_declaration ::= **reg** [**signed**] [range]
list_of_block_variable_identifiers ;

list_of_block_variable_identifiers ::=
block_variable_type { **,** block_variable_type }

block_variable_type ::=
variable_identifier
| variable_identifier dimension { dimension }

Use of variables (both reading the value of and writing a value to) that are defined outside a task declaration but within the enclosing module declaration shall be supported.

*The keyword **automatic** is not optional.*

7.8.2.2 Task enabling and argument passing

task_enable ::=
hierarchical_task_identifier [**(** expression { **,** expression } **)**] **;**

Recursion with a static bound shall be supported.

7.8.2.3 Task memory usage and concurrent activation

Supported.

7.8.3 Functions and function calling

7.8.3.1 Function declarations

function_declaration ::=
function [**automatic**] [**signed**] [range_or_type] function_identifier **;**
function_item_declaration { function_item_declaration }
function_statement
endfunction
| **function** [**automatic**] [**signed**] [range_or_type] function_identifier
(function_port_list) **;**
block_item_declaration { block_item_declaration }
function_statement
endfunction

function_item_declaration ::=
block_item_declaration
| tf_input_declaration **;**

function_port_list ::=
{ attribute_instance } tf_input_declaration { **,** { attribute_instance } tf_input_declaration }

tf_input_declaration ::=
input [**reg**] [**signed**] [range] list_of_port_identifiers
| **input** [task_port_type] list_of_port_identifiers

range_or_type ::= range | **integer** | ~~**real**~~ | ~~**realtime**~~ | ~~**time**~~

block_item_declaration ::=
{ attribute_instance } block_reg_declaration
~~| { attribute_instance } event_declaration~~
| { attribute_instance } integer_declaration
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
~~| { attribute_instance } real_declaration~~
~~| { attribute_instance } realtime_declaration~~
~~| { attribute_instance } time_declaration~~

block_reg_declaration ::= **reg** [**signed**] [range]
list_of_block_variable_identifiers ;

list_of_block_variable_identifiers ::=
block_variable_type { **,** block_variable_type }

block_variable_type ::=
    variable_identifier
    | variable_identifier dimension { dimension }

Number of parameters shall match number of arguments in call.

*The keyword **automatic** is not optional.*

Use of variables (both reading the value of and writing a value to) that are defined outside a function declaration but within the enclosing module declaration shall be supported.

7.8.3.2 Returning a value from a function

Supported.

7.8.3.3 Calling a function

function_call ::= hierarchical_function_identifier { attribute_instance }
    (expression { **,** expression } **)**

Recursion with a static bound shall be supported.

7.8.3.4 Function rules

Supported.

7.8.3.5 Use of constant functions

Supported.

7.9 Disabling of named blocks and tasks

disable_statement ::=
    ~~**disable** hierarchical_task_identifier **;**~~
    | **disable** hierarchical_block_identifier **;**

The block identifier shall be that of the enclosing block. Disable of any other blocks shall not be supported.

7.10 Hierarchical structures

7.10.1 Modules

module_declaration ::=
    { attribute_instance } module_keyword module_identifier [module_parameter_port_list]
        [list_of_ports] **;** { module_item }
    endmodule
    | { attribute_instance } module_keyword module_identifier [module_parameter_port_list]
        [list_of_port_declarations] **;** { non_port_module_item }
    endmodule

module_keyword ::= **module** | ~~**macromodule**~~

module_parameter_port_list ::= **# (** parameter_declaration { **,** parameter_declaration } **)**

```
list_of_ports ::= ( port { , port } )

list_of_port_declarations ::=
        ( port_declaration { , port_declaration } )
        | ()

port ::=
        [ port_expression ]
        | . port_identifier ( [ port_expression ] )

port_expression ::=
        port_reference
        | { port_reference { , port_reference} }

port_reference ::= port_identifier
        port_identifier [ constant_expression ]
        | port_identifier [ range_expression ]

port_declaration ::=
        { attribute_instance } inout_declaration
        | { attribute_instance } input_declaration
        | { attribute_instance } output_declaration

module_item ::=
        module_or_generate_item
        | port_declaration ;
        | { attribute_instance } generated_instantiation
        | { attribute_instance } local_parameter_declaration
        | { attribute_instance } parameter_declaration ;
        | { attribute_instance } specify_block
        | { attribute_instance } specparam_declaration

module_or_generate_item ::=
        { attribute_instance } module_or_generate_item_declaration
        | { attribute_instance } parameter_override
        | { attribute_instance } continuous_assign
        | { attribute_instance } gate_instantiation
        | { attribute_instance} udp_instantiation
        | { attribute_instance } module_instantiation
        | { attribute_instance } initial_construct
        | { attribute_instance } always_construct

module_or_generate_item_declaration ::=
        net_declaration
        | reg_declaration
        | integer_declaration
        | real_declaration
        | time_declaration
        | realtime_declaration
        | event_declaration
        | genvar_declaration
        | task_declaration
        | function_declaration
```


non_port_module_item ::=
{ attribute_instance } generated_instantiation
| { attribute_instance } local_parameter_declaration
| { attribute_instance } module_or_generate_item
| { attribute_instance } parameter_declaration **;**
| { attribute_instance } specify_block
| { attribute_instance } specparam_declaration

~~parameter_override~~ ::= **defparam** list_of_param_assignments **;**

7.10.1.1 Top-level modules

Supported.

7.10.1.2 Module instantiation

module_instantiation ::=
module_identifier [parameter_value_assignment] module_instance { **,**
module_instance } **;**

parameter_value_assignment ::= **# (** list_of_parameter_assignments **)**

list_of_parameter_assignments ::=
ordered_parameter_assignment { **,** ordered_parameter_assignment }
| named_parameter_assignment { **,** named_parameter_assignment }

ordered_parameter_assignment ::= expression

named_parameter_assignment ::= **.** parameter_identifier **(** [expression] **)**

module_instance ::= name_of_instance **(** [list_of_port_connections] **)**

name_of_instance ::= module_instance_identifier [range]

list_of_port_connections ::=
ordered_port_connection { **,** ordered_port_connection }
| named_port_connection { **,** named_port_connection }

ordered_port_connection ::= { attribute_instance } [expression]

named_port_connection ::= { attribute_instance } **.** port_identifier **(** [expression] **)**

7.10.1.3 Generated instantiation

module_item ::=
module_or_generate_item
| port_declaration **;**
| { attribute_instance } generated_instantiation
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
| { attribute_instance } specify_block
| { attribute_instance } specparam_declaration

module_or_generate_item ::=
{ attribute_instance } module_or_generate_item_declaration
| { attribute_instance } parameter_override
| { attribute_instance } continuous_assign
| { attribute_instance } gate_instantiation
| { attribute_instance} ~~udp_instantiation~~
| { attribute_instance } module_instantiation
| { attribute_instance } initial_construct
| { attribute_instance } always_construct

module_or_generate_item_declaration ::=
net_declaration
| reg_declaration
| integer_declaration
~~| real_declaration~~
~~| time_declaration~~
~~| realtime_declaration~~
~~| event_declaration~~
| genvar_declaration
| task_declaration
| function_declaration

generated_instantiation ::= **generate** { generate_item } **endgenerate**

generate_item_or_null ::= generate_item | **;**

generate_item ::=
generate_conditional_statement
| generate_case_statement
| generate_loop_statement
| generate_block
| module_or_generate_item

generate_conditional_statement ::=
if (constant_expression **)** generate_item_or_null
[**else** generate_item_or_null]

generate_case_statement ::= **case (** constant_expression **)**
genvar_case_item { genvar_case_item } **endcase**

genvar_case_item ::= constant_expression { **,** constant_expression } **;**
generate_item_or_null | **default** [**:**] generate_item_or_null

generate_loop_statement ::=
for (genvar_assignment **;** constant_expression **;** genvar_assignment **)**
begin : generate_block_identifier { generate_item } **end**

genvar_assignment ::= genvar_identifier **=** constant_expression

generate_block ::= **begin** [**:** generate_block_identifier] { generate_item } **end**

genvar_assignment ::= genvar_identifier = constant_expression

generate_block ::=
begin [**:** generate_block_identifier] { generate_item } **end**

genvar_declaration ::= **genvar** list_of_genvar_identifiers **;**

list_of_genvar_identifiers ::= genvar_identifier { **,** genvar_identifier }

7.10.2 Overriding module parameter values

7.10.2.1 defparam statement

Not supported.

7.10.2.2 Module instance parameter value assignment

Supported.

7.10.2.3 Parameter dependence

Supported.

7.10.3 Ports

7.10.3.1 Port definition

list_of_ports ::= **(** port { **,** port } **)**

list_of_port_declarations ::=
    (port_declaration { **,** port_declaration } **)**
    | **()**

port ::=
    [port_expression]
    | **.** port_identifier **(** [port_expression] **)**

port_expression ::=
    port_reference
    | **{** port_reference { **,** port_reference} **}**

port_reference ::= port_identifier
    port_identifier **[** constant_expression **]**
    | port_identifier **[** range_expression **]**

port_declaration ::=
    { attribute_instance } inout_declaration
    | { attribute_instance } input_declaration
    | { attribute_instance } output_declaration

Input ports shall not be assigned values.

If an output identifier is also declared as a reg, the range and indices shall be identical in both the declarations.

7.10.3.2 List of ports

Supported.

7.10.3.3 Port declarations

inout_declaration ::=
    inout [net_type] [**signed**] [range] list_of_port_identifiers

input_declaration ::=
  input [net_type] [signed] [range] list_of_port_identifiers

output_declaration ::=
  output [net_type] [signed] [range] list_of_port_identifiers
  | **output** [**reg**] [**signed**] [range] list_of_port_identifiers
  | **output reg** [**signed**] [range] list_of_variable_port_identifiers
  | **output** [output_variable_type] list_of_port_identifiers
  | **output** output_variable_type list_of_variable_port_identifiers

list_of_port_identifiers ::= port_identifier { **,** port_identifier }

A port of an integer type shall be treated as a 32 bit signed type.

7.10.3.4 List of ports declarations

Supported.

7.10.3.5 Connecting module instance ports by ordered list

Supported.

7.10.3.6 Connecting module instance ports by name

Supported.

7.10.3.7 Real numbers in port connections

Not supported.

7.10.3.8 Connecting dissimilar ports

Supported.

7.10.3.9 Port connection rules

Supported.

7.10.3.10 Net types resulting from dissimilar port connections

Ignored.

7.10.3.11 Connecting signed values via ports

Supported.

7.10.4 Hierarchical names

escaped_hierarchical_identifier ::=
  escaped_hierarchical_branch ~~{ .simple_hierarchical_branch |
  .escaped_hierarchical_branch }~~

escaped_identifier ::= **** {Any_ASCII_character_except_white_space} white_space

hierarchical_identifier ::=
simple_hierarchical_identifier
| escaped_hierarchical_identifier

simple_hierarchical_identifier ::=
simple_hierarchical_branch [.escaped_identifier]

simple_identifier ::= [a-zA-Z_] { [a-zA-Z0-9_$] }

simple_hierarchical_branch ::=
simple_identifier [[unsigned_number]]
[{ .simple_identifier [[unsigned_number]] }]

escaped_hierarchical_branch ::=
escaped_identifier [[unsigned_number]]
[{ .escaped_identifier [[unsigned_number]] }]

white_space ::= space | tab | newline | eof

7.10.5 Upwards name referencing

upward_name_reference ::= module_identifier.item_name

item_name ::=
function_identifier
| block_identifier
| net_identifier
| parameter_identifier
| port_identifier
| task_identifier
| variable_identifier

7.10.6 Scope rules

Supported.

7.11 Configuring the contents of a design

7.11.1 Introduction

Supported.

7.11.1.1 Library notation

Supported.

7.11.1.2 Basic configuration elements

Supported.

7.11.2 Libraries

Supported.

7.11.2.1 Specifying libraries—the library map file

library_text := { library_descriptions }

library_descriptions ::=
library_declaration
| include_statement
| config_declaration

library_declaration ::=
library library_identifier file_path_spec [{ **,** file_path_spec }]
[**-incdir** file_path_spec [{ **,** file_path_spec }] ;

file_path_spec ::= file_path

include_statement ::= **include** < file_path_spec > ;

7.11.2.2 Using multiple library mapping files

include_statement ::= **include** < file_path_spec > ;

7.11.2.3 Mapping source files to libraries

Supported.

7.11.3 Configurations

Supported.

7.11.3.1 Basic configuration syntax

config_declaration ::=
config config_identifier ;
design_statement
{ config_rule_statement }
endconfig

design_statement ::= **design** { [library_identifier **.**] cell_identifier } ;

config_rule_statement ::=
default_clause liblist_clause
| inst_clause liblist_clause
| inst_clause use_clause
| cell_clause liblist_clause
| cell_clause use_clause

default_clause ::= **default**

inst_clause ::= **instance** inst_name

inst_name ::= topmodule_identifier{**.**instance_identifier}

cell_clause ::= **cell** [library_identifier**.**]cell_identifier

liblist_clause ::= **liblist** [{ library_identifier }]

use_clause ::= **use** [library_identifier **.**] cell_identifier [**:config**]

7.11.3.2 Hierarchical configurations

Supported.

7.11.4 Using libraries and configs

Supported.

7.11.5 Configuration examples

Supported.

7.11.6 Displaying library binding information

Ignored.

7.11.7 Library mapping examples

Supported.

7.12 Specify blocks

Ignored.

7.13 Timing checks

Ignored.

7.14 Backannotation using the standard delay format

Ignored.

7.15 System tasks and functions

All system tasks shall be ignored.

The system functions $signed and $unsigned shall be supported. All other system functions are not supported.

7.16 Value change dump (VCD) files

All VCD system tasks are ignored.

7.17 Compiler directives

7.17.1 ‘celldefine and ‘endcelldefine

Ignored.

7.17.2 ‘default_nettype

Supported.

7.17.3 ‘define and ‘undef

Supported. For the define directive, parameterized, multiline and nested directives shall also be supported.

7.17.4 ‘ifdef, ‘else, ‘elsif, ‘endif, ‘ifndef

Supported.

7.17.5 ‘include

Supported.

7.17.6 ‘resetall

Ignored.

7.17.7 ‘line

Ignored.

7.17.8 ‘timescale

Ignored.

7.17.9 ‘unconnected_drive and ‘nounconnected_drive

Ignored.

7.18 PLI

PLI task calls shall be ignored.

PLI function calls shall not be supported.

Annex A

(informative)

Syntax summary

This annex summarizes, using Backus-Naur Form (BNF), the syntax that is supported for RTL synthesis.

NOTE—The BNF presented here is the complete Verilog BNF that identifies the supported, ignored, and unsupported constructs for synthesis.

A.1 Source text

A.1.1 Library source text

library_text := { library_descriptions }

library_descriptions ::=
    library_declaration
  | include_statement
  | config_declaration

library_declaration ::=
    library library_identifier file_path_spec [{ **,** file_path_spec }]
      [**-incdir** file_path_spec [{ **,** file_path_spec }] ;

file_path_spec ::= file_path

include_statement ::= **include** < file_path_spec > ;

A.1.2 Configuration source text

config_declaration ::=
    config config_identifier ;
      design_statement
      { config_rule_statement }
    endconfig

design_statement ::= **design** { [library_identifier **.**] cell_identifier } ;

config_rule_statement ::=
    default_clause liblist_clause
  | inst_clause liblist_clause
  | inst_clause use_clause
  | cell_clause liblist_clause
  | cell_clause use_clause

default_clause ::= **default**

inst_clause ::= **instance** inst_name

inst_name ::= topmodule_identifier{**.**instance_identifier} cell_clause ::= **cell** [library_identifier**.**]cell_identifier

liblist_clause ::= **liblist** [{ library_identifier }]

use_clause ::= **use** [library_identifier **.**] cell_identifier [**:config**]

A.1.3 Module and primitive source text

source_text ::= {description}

description ::=
module_declaration
~~| udp_declaration~~

module_declaration ::=
{ attribute_instance } module_keyword module_identifier [module_parameter_port_list]
[list_of_ports] **;** { module_item }
endmodule
| { attribute_instance } module_keyword module_identifier [module_parameter_port_list]
[list_of_port_declarations] **;** { non_port_module_item }
endmodule

module_keyword ::= **module** | ~~**macromodule**~~

A.1.4 Module parameters and ports

module_parameter_port_list ::= **# (** parameter_declaration { **,** parameter_declaration } **)**

list_of_ports ::= **(** port { **,** port } **)**

list_of_port_declarations ::=
(port_declaration { **,** port_declaration } **)**
| **()**

port ::=
[port_expression]
| **.** port_identifier **(** [port_expression] **)**

port_expression ::=
port_reference
| **{** port_reference { **,** port_reference} **}**

port_reference ::= port_identifier
port_identifier **[** constant_expression **]**
| port_identifier **[** range_expression **]**

port_declaration ::=
{ attribute_instance } inout_declaration
| { attribute_instance } input_declaration
| { attribute_instance } output_declaration

A.1.5 Module items

module_item ::=
module_or_generate_item
| port_declaration **;**
| { attribute_instance } generated_instantiation
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
| { attribute_instance } specify_block
| { attribute_instance } specparam_declaration

module_or_generate_item ::=
{ attribute_instance } module_or_generate_item_declaration
| { attribute_instance } parameter_override
| { attribute_instance } continuous_assign
| { attribute_instance } gate_instantiation
| { attribute_instance} ~~udp_instantiation~~
| { attribute_instance } module_instantiation
| { attribute_instance } initial_construct
| { attribute_instance } always_construct

module_or_generate_item_declaration ::=
net_declaration
| reg_declaration
| integer_declaration
~~| real_declaration~~
~~| time_declaration~~
~~| realtime_declaration~~
~~| event_declaration~~
| genvar_declaration
| task_declaration
| function_declaration

non_port_module_item ::=
{ attribute_instance } generated_instantiation
| { attribute_instance } local_parameter_declaration
| { attribute_instance } module_or_generate_item
| { attribute_instance } parameter_declaration **;**
| { attribute_instance } specify_block
| { attribute_instance } specparam_declaration

~~parameter_override~~ ::= **defparam** list_of_param_assignments **;**

A.2 Declarations

A.2.1 Declaration types

A.2.1.1 Module parameter declarations

local_parameter_declaration ::=
localparam [**signed**] [range] list_of_param_assignments **;**
| **localparam integer** list_of_param_assignments **;**
~~| **localparam real** list_of_param_assignments **;**~~
~~| **localparam realtime** list_of_param_assignments **;**~~
~~| **localparam time** list_of_param_assignments **;**~~

parameter_declaration ::=
  parameter [**signed**] [range] list_of_param_assignments
  | **parameter integer** list_of_param_assignments
  ~~| **parameter real** list_of_param_assignments~~
  ~~| **parameter realtime** list_of_param_assignments~~
  ~~| **parameter time** list_of_param_assignments~~

specparam_declaration ::= **specparam** [range] list_of_specparam_assignments **;**

A.2.1.2 Port declarations

inout_declaration ::=
  inout [net_type] [**signed**] [range] list_of_port_identifiers

input_declaration ::=
  input [net_type] [signed] [range] list_of_port_identifiers

output_declaration ::=
  output [net_type] [signed] [range] list_of_port_identifiers
  | **output** [**reg**] [**signed**] [range] list_of_port_identifiers
  | **output reg** [**signed**] [range] list_of_variable_port_identifiers
  | **output** [output_variable_type] list_of_port_identifiers
  | **output** output_variable_type list_of_variable_port_identifiers

A.2.1.3 Type declarations

~~event_declaration~~ ::= **event** list_of_event_identifiers **;**

genvar_declaration ::= **genvar** list_of_genvar_identifiers **;**

integer_declaration ::= **integer** list_of_variable_identifiers **;**

net_declaration ::=
  net_type [**signed**] [delay3] list_of_net_identifiers **;**
  | net_type [drive_strength] [**signed**] [delay3]
    list_of_net_decl_assignments **;**
  | net_type [**vectored** | **scalared**] [**signed**] range [delay3]
    list_of_net_identifiers **;**
  | net_type [drive_strength] [**vectored** | **scalared**] [**signed**] range
    [delay3] list_of_net_decl_assignments **;**
  ~~| **trireg** [charge_strength] [**signed**] [delay3] list_of_net_identifiers **;**~~
  ~~| **trireg** [drive_strength] [**signed**] [delay3]~~
    ~~list_of_net_decl_assignments **;**~~
  ~~| **trireg** [charge_strength] [**vectored** | **scalared**] [**signed**] range~~
    ~~[delay3] list_of_net_identifiers **;**~~
  ~~| **trireg** [drive_strength] [**vectored** | **scalared** } [**signed**] range~~
    ~~[delay3] list_of_net_decl_assignments **;**~~

~~real_declaration~~ ::= **real** list_of_real_identifiers **;**

~~realtime_declaration~~ ::= **realtime** list_of_real_identifiers **;**

reg_declaration ::= **reg** [**signed**] [range] list_of_variable_identifiers **;**

~~time_declaration~~ ::= **time** list_of_variable_identifiers **;**

A.2.2 Declaration data types

A.2.2.1 Net and variable types

net_type ::=
 supply0 | **supply1**
 | **tri** | **triand** | **trior** | ~~**tri0**~~ | ~~**tri1**~~
 | **wire** | **wand** | **wor**

output_variable_type ::= **integer** | **time**

~~real_type~~ ::=
 real_identifier [**=** constant_expression]
 | real_identifier dimension { dimension }

variable_type ::=
 variable_identifier [= constant_expression]
 | variable_identifier dimension { dimension }

A.2.2.2 Strengths

drive_strength ::=
 (strength0 **,** strength1 **)**
 | **(** strength1 **,** strength0 **)**
 | **(** strength0 **, highz1)**
 | **(** strength1 **, highz0)**
 | **(highz1 ,** strength0 **)**
 | **(highz0 ,** strength1 **)**

strength0 ::= **supply0** | **strong0** | **pull0** | **weak0**

strength1 ::= **supply1** | **strong1** | **pull1** | **weak1**

charge_strength ::= **(small)** | **(medium)** | **(large)**

A.2.2.3 Delays

delay3 ::= **#** delay_value | **# (** delay_value [**,** delay_value [**,** delay_value]] **)**

delay2 ::= **#** delay_value | **# (** delay_value [**,** delay_value] **)**

delay_value ::=
 unsigned_number
 | parameter_identifier
 | specparam_identifier
 | mintypmax_expression

A.2.3 Declaration lists

~~list_of_event_identifiers~~ ::= event_identifier [dimension { dimension }]
 { **,** event_identifier [dimension { dimension }] }

list_of_genvar_identifiers ::= genvar_identifier { **,** genvar_identifier }

list_of_net_decl_assignments ::= net_decl_assignment { **,** net_decl_assignment }

list_of_net_identifiers ::= net_identifier [dimension { dimension }]
{ **,** net_identifier [dimension { dimension }] }

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

list_of_port_identifiers ::= port_identifier { **,** port_identifier }

~~list_of_real_identifiers~~ ::= real_type { **,** real_type }

list_of_specparam_assignments ::=
specparam_assignment { **,** specparam_assignment }

list_of_variable_identifiers ::= variable_type { **,** variable_type }

list_of_variable_port_identifiers ::= port_identifier
[**=** constant_expression] { **,** port_identifier [**=** constant_expression] }

A.2.4 Declaration assignments

net_decl_assignment ::= `net`_identifier **=** expression

param_assignment ::= parameter_identifier **=** constant_expression

specparam_assignment ::=
specparam_identifier **=** constant_mintypmax_expression
| pulse_control_specparam

pulse_control_specparam ::=
PATHPULSE$ = (reject_limit_value [**,** error_limit_value] **) ;**
| **PATHPULSE$**specify_input_terminal_descriptor**$**specify_output_terminal_descriptor
= (reject_limit_value [**,** error_limit_value] **) ;**

error_limit_value ::= limit_value

reject_limit_value ::= limit_value

limit_value ::= constant_mintypmax_expression

A.2.5 Declaration ranges

dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**

range ::= **[** msb_constant_expression **:** lsb_constant_expression **]**

A.2.6 Function declarations

function_declaration ::=
function [**automatic**] [**signed**] [range_or_type] function_identifier **;**
function_item_declaration { function_item_declaration }
function_statement
endfunction
| **function** [**automatic**] [**signed**] [range_or_type] function_identifier
(function_port_list) **;**
block_item_declaration { block_item_declaration }
function_statement
endfunction

function_item_declaration ::=
    block_item_declaration
    | tf_input_declaration **;**

function_port_list ::=
    { attribute_instance } tf_input_declaration { **,** { attribute_instance } tf_input_declaration }

range_or_type ::= range | **integer** | ~~**real**~~ | ~~**realtime**~~ | ~~**time**~~

A.2.7 Task declarations

task_declaration ::=
    task [**automatic**] task_identifier **;**
        { task_item_declaration }
        statement
    endtask
    | **task** [**automatic**] task_identifier **(** task_port_list **) ;**
        { block_item_declaration }
        statement
    endtask

task_item_declaration ::=
    block_item_declaration
    | { attribute_instance } tf_input_declaration **;**
    | { attribute_instance } tf_output_declaration **;**
    | { attribute_instance } tf_inout_declaration **;**

task_port_list ::= task_port_item { **,** task_port_item }

task_port_item ::=
    { attribute_instance } tf_input_declaration
    | { attribute_instance } tf_output_declaration
    | { attribute_instance } tf_inout_declaration

tf_input_declaration ::=
    input [**reg**] [**signed**] [range] list_of_port_identifiers
    | **input** [task_port_type] list_of_port_identifiers

tf_output_declaration ::=
    output [**reg**] [**signed**] [range] list_of_port_identifiers
    | **output** [task_port_type] list_of_port_identifiers

tf_inout_declaration ::=
    inout [**reg**] [**signed**] [range] list_of_port_identifiers
    | **inout** [task_port_type] list_of_port_identifiers

task_port_type ::=
    ~~**time**~~ | ~~**real**~~ | ~~**realtime**~~ | **integer**

A.2.8 Block item declarations

block_item_declaration ::=
{ attribute_instance } block_reg_declaration
~~| { attribute_instance } event_declaration~~
| { attribute_instance } integer_declaration
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
~~| { attribute_instance } real_declaration~~
~~| { attribute_instance } realtime_declaration~~
~~| { attribute_instance } time_declaration~~

block_reg_declaration ::= **reg** [**signed**] [range]
list_of_block_variable_identifiers ;

list_of_block_variable_identifiers ::=
block_variable_type { **,** block_variable_type }

block_variable_type ::=
variable_identifier
| variable_identifier dimension { dimension }

A.3 Primitive instances

A.3.1 Primitive instantiation and instances

gate_instantiation ::=
~~cmos_switchtype [delay3] cmos_switch_instance { , cmos_switch_instance } ;~~
| enable_gatetype [drive_strength] [delay3] enable_gate_instance
{ **,** enable_gate_instance} **;**
~~| mos_switchtype [delay3] mos_switch_instance { , mos_switch_instance } ;~~
| n_input_gatetype [drive_strength] [delay2] n_input_gate_instance
{ **,** n_input_gate_instance } **;**
| n_output_gatetype [drive_strength] [delay2] n_output_gate_instance
{ **,** n_output_gate_instance } **;**
~~| pass_en_switchtype [delay3] pass_enable_switch_instance~~
~~{ , pass_enable_switch_instance } ;~~
~~| pass_switchtype pass_switch_instance { , pass_switch_instance } ;~~
~~| **pulldown** [pulldown_strength] pull_gate_instance { , pull_gate_instance } ;~~
~~| **pullup** [pullup_strength] pull_gate_instance { , pull_gate_instance } ;~~

~~cmos_switch_instance~~ ::= [name_of_gate_instance] **(** output_terminal **,**
input_terminal **,** ncontrol_terminal **,** pcontrol_terminal **)**

enable_gate_instance ::= [name_of_gate_instance] **(** output_terminal **,**
input_terminal **,** enable_terminal **)**

~~mos_switch_instance~~ ::= [name_of_gate_instance] **(** output_terminal **,**
input_terminal **,** enable_terminal **)**

n_input_gate_instance ::= [name_of_gate_instance] **(** output_terminal **,**
input_terminal { **,** input_terminal } **)**

n_output_gate_instance ::= [name_of_gate_instance] **(** output_terminal { **,**
output_terminal } **,** input_terminal **)**

~~pass_switch_instance~~ ::= [name_of_gate_instance] **(** inout_terminal **,**
inout_terminal **)**

~~pass_enable_switch_instance~~ ::= [name_of_gate_instance] **(** inout_terminal **,**
inout_terminal **,** enable_terminal **)**

~~pull_gate_instance~~ ::= [name_of_gate_instance] **(** output_terminal **)**

name_of_gate_instance ::= gate_instance_identifier [range]

A.3.2 Primitive strengths

~~pulldown_strength~~ ::=
(strength0 **,** strength1 **)**
| **(** strength1 **,** strength0 **)**
| **(** strength0 **)**

~~pullup_strength~~ ::=
(strength0 **,** strength1 **)**
| **(** strength1 **,** strength0 **)**
| **(** strength1 **)**

A.3.3 Primitive terminals

enable_terminal ::= expression

inout_terminal ::= net_lvalue

input_terminal ::= expression

~~ncontrol_terminal~~ ::= expression

output_terminal ::= net_lvalue

~~pcontrol_terminal~~ ::= expression

A.3.4 Primitive gate and switch types

~~cmos_switchtype~~ ::= **cmos** | **rcmos**

enable_gatetype ::= **bufif0** | **bufif1** | **notif0** | **notif1**

~~mos_switchtype~~ ::= **nmos** | **pmos** | **rnmos** | **rpmos**

n_input_gatetype ::= **and** | **nand** | **or** | **nor** | **xor** | **xnor**

n_output_gatetype ::= **buf** | **not**

~~pass_en_switchtype~~ ::= **tranif0** | **tranif1** | **rtranif1** | **rtranif0**

~~pass_switchtype~~ ::= **tran** | **rtran**

A.4 Module and generated instantiation

A.4.1 Module instantiation

module_instantiation ::=
    module_identifier [parameter_value_assignment] module_instance { **,**
      module_instance } **;**

parameter_value_assignment ::= **#** **(** list_of_parameter_assignments **)**

list_of_parameter_assignments ::=
    ordered_parameter_assignment { **,** ordered_parameter_assignment }
  | named_parameter_assignment { **,** named_parameter_assignment }

ordered_parameter_assignment ::= expression

named_parameter_assignment ::= **.** parameter_identifier **(** [expression] **)**

module_instance ::= name_of_instance **(** [list_of_port_connections] **)**

name_of_instance ::= module_instance_identifier [range]

list_of_port_connections ::=
    ordered_port_connection { **,** ordered_port_connection }
  | named_port_connection { **,** named_port_connection }

ordered_port_connection ::= { attribute_instance } [expression]

named_port_connection ::= { attribute_instance } **.** port_identifier **(** [expression] **)**

A.4.2 Generated instantiation

generated_instantiation ::= **generate** { generate_item } **endgenerate**

generate_item_or_null ::= generate_item | **;**

generate_item ::=
    generate_conditional_statement
  | generate_case_statement
  | generate_loop_statement
  | generate_block
  | module_or_generate_item

generate_conditional_statement ::=
    if **(** constant_expression **)** generate_item_or_null
      [**else** generate_item_or_null]

generate_case_statement ::= **case** **(** constant_expression **)**
    genvar_case_item { genvar_case_item } **endcase**

genvar_case_item ::= constant_expression { **,** constant_expression } **;**
    generate_item_or_null | **default** [**:**] generate_item_or_null

generate_loop_statement ::=
    for **(** genvar_assignment **;** constant_expression **;** genvar_assignment **)**
      begin **:** generate_block_identifier { generate_item } **end**

genvar_assignment ::= genvar_identifier **=** constant_expression

generate_block ::= **begin** [**:** generate_block_identifier] { generate_item } **end**

A.5 UDP declaration and instantiation

A.5.1 UDP declaration

~~udp_declaration~~ ::=
{ attribute_instance } **primitive** udp_identifier **(** udp_port_list **) ;**
udp_port_declaration { udp_port_declaration }
udp_body
endprimitive
| { attribute_instance } **primitive** udp_identifier **(** udp_declaration_port_list **) ;**
udp_body
endprimitive

A.5.2 UDP ports

~~udp_port_list~~ ::=
output_port_identifier **,** input_port_identifier { **,** input_port_identifier }

~~udp_declaration_port_list~~ ::=
udp_output_declaration **,** udp_input_declaration { **,** udp_input_declaration }

~~udp_port_declaration~~ ::=
udp_output_declaration **;**
| udp_input_declaration **;**
| udp_ reg_declaration **;**

~~udp_output_declaration~~ ::=
{ attribute_instance } **output** port_identifier
| { attribute_instance } **output reg** port_identifier [**=** constant_expression]

~~udp_input_declaration~~ ::={ attribute_instance } **input** list_of_port_identifiers

~~udp_reg_declaration~~ ::= { attribute_instance } **reg** variable_identifier

A.5.3 UDP body

~~udp_body~~ ::= combinational_body | sequential_body

~~combinational_body~~ ::= **table** combinational_entry { combinational_entry } **endtable**

~~combinational_entry~~ ::= level_input_list **:** output_symbol **;**

~~sequential_body~~ ::= [udp_initial_statement] **table** sequential_entry
{ sequential_entry } **endtable**

~~udp_initial_statement~~ ::= **initial** output_port_identifier **=** init_val **;**

~~init_val~~ ::= **1 b0** | **1 b1** | **1 bx** | **1 bX** | **1 B0** | **1 B1** | **1 Bx** | **1 BX** | **1** | **0**

~~sequential_entry~~ ::= seq_input_list **:** current_state **:** next_state **;**

~~seq_input_list~~ ::= level_input_list | edge_input_list

~~level_input_list~~ ::= level_symbol { level_symbol }

~~edge_input_list~~ ::= { level_symbol } edge_indicator { level_symbol }

~~edge_indicator~~ ::= **(** level_symbol level_symbol **)** | edge_symbol

~~current_state~~ ::= level_symbol

~~next_state~~ ::= output_symbol | **-**

~~output_symbol~~ ::= **0** | **1** | **x** | **X**

~~level_symbol~~ ::= **0** | **1** | **x** | **X** | **?** | **b** | **B**

~~edge_symbol~~ ::= **r** | **R** | **f** | **F** | **p** | **P** | **n** | **N** | *****

A.5.4 UDP instantiation

~~udp_instantiation~~ ::= udp_identifier [drive_strength] [delay2]
udp_instance { **,** udp_instance } **;**

~~udp_instance~~ ::= [name_of_udp_instance] **(** output_terminal **,** input_terminal
{ **,** input_terminal } **)**

~~name_of_udp_instance~~ ::= udp_instance_identifier [range]

A.6 Behavioral statements

A.6.1 Continuous assignment statements

continuous_assign ::= **assign** [drive_strength] [delay3] list_of_net_assignments **;**

list_of_net_assignments ::= net_assignment { **,** net_assignment }

net_assignment ::= net_lvalue **=** expression

A.6.2 Procedural blocks and assignments

initial_construct ::= **initial** statement

always_construct ::= **always** statement

blocking_assignment ::=
variable_lvalue **=** [delay_or_event_control] expression

nonblocking_assignment ::=
variable_lvalue **<=** [delay_or_event_control] expression

~~procedural_continuous_assignments~~ ::=
 assign variable_assignment **;**
 | **deassign** variable_lvalue **;**
 | **force** variable_assignment **;**
 | **force** net_assignment **;**
 | **release** variable_lvalue **;**
 | **release** net_lvalue **;**

function_blocking_assignment ::= variable_lvalue **=** expression

function_statement_or_null ::=
 function_statement
 | { attribute_instance } **;**

A.6.3 Parallel and sequential blocks

function_seq_block ::= **begin** [**:** block_identifier
 { block_item_declaration }] { function_statement } **end**

variable_assignment := variable_lvalue **=** expression

~~par_block~~ ::= **fork** [**:** block_identifier
 { block_item_declaration }] { statement } **join**

seq_block ::= **begin** [**:** block_identifier
 { block_item_declaration }] { statement } **end**

A.6.4 Statements

statement ::=
 { attribute_instance } blocking_assignment **;**
 | { attribute_instance } case_statement
 | { attribute_instance } conditional_statement
 | { attribute_instance } disable_statement
 ~~| { attribute_instance } event_trigger~~
 | { attribute_instance } loop_statement
 | { attribute_instance } nonblocking assignment **;**
 ~~| { attribute_instance } par_block~~
 ~~| { attribute_instance } procedural_continuous_assignments ;~~
 | { attribute_instance } procedural_timing_control_statement
 | { attribute_instance } seq_block
 | { attribute_instance } system_task_enable
 | { attribute_instance } task_enable
 ~~| { attribute_instance } wait_statement~~

statement_or_null ::=
 statement
 | { attribute_instance } **;**

function_statement ::=
{ attribute_instance } function_blocking_assignment ;
| { attribute_instance } function_case_statement
| { attribute_instance } function_conditional_statement
| { attribute_instance } function_loop_statement
| { attribute_instance } function_seq_block
| { attribute_instance } disable_statement
| { attribute_instance } system_task_enable

A.6.5 Timing control statements

delay_control ::=
delay_value
| # (mintypmax_expression)

delay_or_event_control ::=
delay_control
| event_control
~~| **repeat** (expression) event_control~~

disable_statement ::=
~~**disable** hierarchical_task_identifier ;~~
| **disable** hierarchical_block_identifier ;

event_control ::=
@ event_identifier
| @ (event_expression)
| @ *
| @ (*)

~~event_trigger~~ ::=
-> hierarchical_event_identifier ;

event_expression ::=
~~expression~~
| hierarchical_identifier
| **posedge** expression
| **negedge** expression
| event_expression **or** event_expression
| event_expression , event_expression

procedural_timing_control_statement ::=
delay_or_event_control statement_or_null

~~wait_statement~~ ::=
wait (expression) statement_or_null

A.6.6 Conditional statements

conditional_statement ::=
if (expression) statement_or_null [**else** statement_or_null]
| if_else_if_statement

if_else_if_statement ::=
 if (expression **)** statement_or_null
 { **else if (** expression **)** statement_or_null }
 [**else** statement_or_null]

function_conditional_statement ::=
 if (expression **)** function_statement_or_null
 [**else** function_statement_or_null]
 | function_if_else_if_statement

function_if_else_if_statement ::=
 if (expression **)** function_statement_or_null
 { **else if (** expression **)** function_statement_or_null }
 [**else** function_statement_or_null]

A.6.7 Case statements

case_statement ::=
 case (expression **)** case_item { case_item } **endcase**
 | **casez (** expression **)** case_item { case_item } **endcase**
 | **casex (** expression **)** case_item { case_item } **endcase**

case_item ::=
 expression { **,** expression } **:** statement_or_null
 | **default** [**:**] statement_or_null

function_case_statement ::=
 case (expression **)** function_case_item { function_case_item } **endcase**
 | **casez (** expression **)** function_case_item { function_case_item } **endcase**
 | **casex (** expression **)** function_case_item { function_case_item } **endcase**

function_case_item ::=
 expression { **,** expression } **:** function_statement_or_null
 | **default** [**:**] function_statement_or_null

A.6.8 Looping statements

function_loop_statement ::=
 ~~**forever** function_statement~~
 ~~| **repeat (** expression **)** function_statement~~
 ~~| **while (** expression **)** function_statement~~
 | **for (** variable_assignment **;** expression **;** variable_assignment **)**
 function_statement

loop_statement ::=
 ~~**forever** statement~~
 ~~| **repeat (** expression **)** statement~~
 ~~| **while (** expression **)** statement~~
 | **for (** variable_assignment **;** expression **;** variable_assignment **)** statement

A.6.9 Task enable statements

system_task_enable ::=
 system_task_identifier [**(** expression { **,** expression } **)**] **;**

task_enable ::=
    hierarchical_task_identifier [(expression { , expression })] ;

A.7 Specify section

A.7.1 Specify block declaration

specify_block ::= **specify** { specify_item } **endspecify**

specify_item ::=
    specparam_declaration
    | pulsestyle_declaration
    | showcancelled_declaration
    | path_declaration
    | system_timing_check

pulsestyle_declaration ::=
    pulsestyle_onevent list_of_path_output ;
    | **pulsestyle_ondetect** list_of_path_outputs ;

showcancelled_declaration ::=
    showcancelled list_of_path_outputs ;
    | **noshowcancelled** list_of_path_outputs ;

A.7.2 Specify path declarations

path_declaration ::=
    simple_path_declaration ;
    | edge_sensitive_path_declaration ;
    | state_dependent_path_declaration ;

simple_path_declaration ::=
    parallel_path_description = path_delay_value
    | full_path_description = path_delay_value

parallel_path_description ::=
    (specify_input_terminal_descriptor [polarity_operator] =>
      specify_output_terminal_descriptor)

full_path_description ::=
    (list_of_path_inputs [polarity_operator] *> list_of_path_outputs)

list_of_path_inputs ::=
    specify_input_terminal_descriptor { , specify_input_terminal_descriptor }

list_of_path_outputs ::=
    specify_output_terminal_descriptor { , specify_output_terminal_descriptor }

A.7.3 Specify block terminals

specify_input_terminal_descriptor ::=
    input_identifier
    | input_identifier [constant_expression]
    | input_identifier [range_expression]

specify_output_terminal_descriptor ::=
output_identifier
| output_identifier [constant_expression]
| output_identifier [range_expression]

input_identifier ::= input_port_identifier | inout_port_identifier

output_identifier ::= output_port_identifier | inout_port_identifier

A.7.4 Specify path delays

path_delay_value ::=
list_of_path_delay_expressions
| (list_of_path_delay_expressions)

list_of_path_delay_expressions ::=
t_path_delay_expression
| trise_path_delay_expression , tfall_path_delay_expression
| trise_path_delay_expression , tfall_path_delay_expression ,
tz_path_delay_expression
| t01_path_delay_expression , t10_path_delay_expression ,
t0z_path_delay_expression ,
tz1_path_delay_expression , t1z_path_delay_expression ,
tz0_path_delay_expression
| t01_path_delay_expression , t10_path_delay_expression ,
t0z_path_delay_expression ,
tz1_path_delay_expression , t1z_path_delay_expression ,
tz0_path_delay_expression ,
t0x_path_delay_expression , tx1_path_delay_expression ,
t1x_path_delay_expression ,
tx0_path_delay_expression , txz_path_delay_expression ,
tzx_path_delay_expression

t_path_delay_expression ::= path_delay_expression

trise_path_delay_expression ::= path_delay_expression

tfall_path_delay_expression ::= path_delay_expression

tz_path_delay_expression ::= path_delay_expression

t01_path_delay_expression ::= path_delay_expression

t10_path_delay_expression ::= path_delay_expression

t0z_path_delay_expression ::= path_delay_expression

tz1_path_delay_expression ::= path_delay_expression

t1z_path_delay_expression ::= path_delay_expression

tz0_path_delay_expression ::= path_delay_expression

t0x_path_delay_expression ::= path_delay_expression

tx1_path_delay_expression ::= path_delay_expression t1x_path_delay_expression ::= path_delay_expression

tx0_path_delay_expression ::= path_delay_expression

txz_path_delay_expression ::= path_delay_expression

tzx_path_delay_expression ::= path_delay_expression

path_delay_expression ::= constant_mintypmax_expression

edge_sensitive_path_declaration ::=
parallel_edge_sensitive_path_description = path_delay_value
| full_edge_sensitive_path_description = path_delay_value

parallel_edge_sensitive_path_description ::=
([edge_identifier] specify_input_terminal_descriptor =>
specify_output_terminal_descriptor [polarity_operator] : data_source_expression)

full_edge_sensitive_path_description ::=
([edge_identifier] list_of_path_inputs *>
list_of_path_outputs [polarity_operator] : data_source_expression)

data_source_expression ::= expression

edge_identifier ::= **posedge** | **negedge**

state_dependent_path_declaration ::=
if (module_path_expression) simple_path_declaration
| **if** (module_path_expression) edge_sensitive_path_declaration
| **ifnone** simple_path_declaration

polarity_operator ::= + | -

A.7.5 System timing checks

A.7.5.1 System timing check commands

system_timing_check ::=
$setup_timing_check
| $hold_timing_check
| $setuphold_timing_check
| $recovery_timing_check
| $removal_timing_check
| $recrem_timing_check
| $skew_timing_check
| $timeskew_timing_check
| $fullskew_timing_check
| $period_timing_check
| $width_timing_check
| $nochange_timing_check

$setup_timing_check ::=
$setup (data_event , reference_event , timing_check_limit [, [notify_reg]]) ;

$hold_timing_check ::=
$hold (reference_event , data_event , timing_check_limit [, [notify_reg]]) ;

$setuphold_timing_check ::=
    $setuphold (reference_event **,** data_event **,** timing_check_limit **,** timing_check_limit
        [**,** [notify_reg] [**,** [stamptime_condition] [**,** [checktime_condition]
        [**,** [delayed_reference] [**,** [delayed_data]]]]]] **) ;**

$recovery_timing_check ::=
    $recovery (reference_event **,** data_event **,** timing_check_limit [**,** [notify_reg]] **) ;**

$removal_timing_check ::=
    $removal (reference_event **,** data_event **,** timing_check_limit [**,** [notify_reg]] **) ;**

$recrem_timing_check ::=
    $recrem (reference_event **,** data_event **,** timing_check_limit **,** timing_check_limit
        [**,** [notify_reg] [**,** [stamptime_condition] [**,** [checktime_condition]
        [**,** [delayed_reference] [**,** [delayed_data]]]]]] **) ;**

$skew_timing_check ::=
    $skew (reference_event **,** data_event **,** timing_check_limit [**,** [notify_reg]] **) ;**

$timeskew_timing_check ::=
    $timeskew (reference_event **,** data_event **,** timing_check_limit
        [**,** [notify_reg] [**,** [event_based_flag] [**,** [remain_active_flag]]]] **) ;**

$fullskew_timing_check ::=
    $fullskew (reference_event **,** data_event **,** timing_check_limit **,** timing_check_limit
        [**,** [notify_reg] [**,** [event_based_flag] [**,** [remain_active_flag]]]] **) ;**

$period_timing_check ::=
    $period (controlled_reference_event **,** timing_check_limit [**,** [notify_reg]] **) ;**

$width_timing_check ::=
    $width (controlled_reference_event **,** timing_check_limit **,** threshold [**,** [notify_reg]] **) ;**

$nochange_timing_check ::=
    $nochange (reference_event **,** data_event **,** start_edge_offset **,**
        end_edge_offset [**,** [notify_reg]] **) ;**

A.7.5.2 System timing check command arguments

checktime_condition ::= mintypmax_expression

controlled_reference_event ::= controlled_timing_check_event

data_event ::= timing_check_event

delayed_data ::=
    terminal_identifier
    | terminal_identifier **[** constant_mintypmax_expression **]**

delayed_reference ::=
    terminal_identifier
    | terminal_identifier **[** constant_mintypmax_expression **]**

end_edge_offset ::= mintypmax_expression

event_based_flag ::= constant_expression

notify_reg ::= variable_identifier

reference_event ::= timing_check_event

remain_active_flag ::= constant_mintypmax_expression

stamptime_condition ::= mintypmax_expression

start_edge_offset ::= mintypmax_expression

threshold ::= constant_expression

timing_check_limit ::= expression

A.7.5.3 System timing check event definitions

timing_check_event ::=
    [timing_check_event_control] specify_terminal_descriptor [**&&&** timing_check_condition]

controlled_timing_check_event ::=
    timing_check_event_control specify_terminal_descriptor [**&&&** timing_check_condition]

timing_check_event_control ::=
    posedge
    | **negedge**
    | edge_control_specifier

specify_terminal_descriptor ::=
    specify_input_terminal_descriptor
    | specify_output_terminal_descriptor

edge_control_specifier ::= **edge** [edge_descriptor [**,** edge_descriptor]]

edge_descriptor ::=
    01
    | **10**
    | z_or_x zero_or_one
    | zero_or_one z_or_x

zero_or_one ::= 0 | 1

z_or_x ::= **x** | **X** | **z** | **Z**

timing_check_condition ::=
    scalar_timing_check_condition
    | **(** scalar_timing_check_condition **)**

scalar_timing_check_condition ::=
    expression
    | **~** expression
    | expression **==** scalar_constant
    | expression **===** scalar_constant
    | expression **!=** scalar_constant
    | expression **!==** scalar_constant

scalar_constant ::=
  1 b0 | **1 b1** | **1 B0** | **1 B1** | **b0** | **b1** | **B0** | **B1** | **1** | **0**

A.8 Expressions

A.8.1 Concatenations

concatenation ::= **{** expression { **,** expression } **}**

constant_concatenation ::= **{** constant_expression { **,** constant_expression } **}**

constant_multiple_concatenation ::= **{** constant_expression constant_concatenation **}**

module_path_concatenation ::= **{** module_path_expression { **,** module_path_expression } **}**

module_path_multiple_concatenation ::= **{** constant_expression module_path_concatenation **}**

multiple_concatenation ::= **{** constant_expression concatenation **}**

net_concatenation ::= **{** net_concatenation_value { **,** net_concatenation_value } **}**

net_concatenation_value ::=
hierarchical_net_identifier
| hierarchical_net_identifier **[** expression **]** { **[** expression **]** }
| hierarchical_net_identifier **[** expression **]** { **[** expression **]** } **[** range_expression **]**
| hierarchical_net_identifier **[** range_expression **]**
| net_concatenation

variable_concatenation ::= { variable_concatenation_value { **,** variable_concatenation_value } }

variable_concatenation_value ::=
hierarchical_variable_identifier
| hierarchical_variable_identifier **[** expression **]** { **[** expression **]** }
| hierarchical_variable_identifier **[** expression **]** { **[** expression **]** } **[** range_expression **]**
| hierarchical_variable_identifier **[** range_expression **]**
| variable_concatenation

A.8.2 Function calls

constant_function_call ::= function_identifier { attribute_instance }
(constant_expression { **,** constant_expression } **)**

function_call ::= hierarchical_function_identifier { attribute_instance }
(expression { **,** expression } **)**

genvar_function_call ::= genvar_function_identifier { attribute_instance }
(constant_expression { **,** constant_expression } **)**

~~system_function_call~~ ::= system_function_identifier
[**(** expression { **,** expression } **)**]

A.8.3 Expressions

base_expression ::= expression

conditional_expression ::= expression1 **?** { attribute_instance }expression2 **:** expression3

constant_base_expression ::= constant_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
  | string

constant_mintypmax_expression ::=
  constant_expression
  | constant_expression **:** constant_expression **:** constant_expression

constant_range_expression ::=
  constant_expression
  | msb_constant_expression **:** lsb_constant_expression
  | constant_base_expression **+:** width_constant_expression
  | constant_base_expression **-:** width_constant_expression

dimension_constant_expression ::= constant_expression

expression1 ::= expression

expression2 ::= expression

expression3 ::= expression

expression ::=
  primary
  | unary_operator { attribute_instance } primary
  | expression binary_operator { attribute_instance } expression
  | conditional_expression
  | string

lsb_constant_expression ::= constant_expression

mintypmax_expression ::=
  expression
  | expression **:** expression **:** expression

module_path_conditional_expression ::=
  module_path_expression **?** { attribute_instance }
    module_path_expression **:** module_path_expression

module_path_expression ::=
  module_path_primary
  | unary_module_path_operator { attribute_instance } module_path_primary
  | module_path_expression binary_module_path_operator
    { attribute_instance } module_path_expression
  | module_path_conditional_expression

module_path_mintypmax_expression ::=
  module_path_expression
  | module_path_expression : module_path_expression : module_path_expression

msb_constant_expression ::= constant_expression

range_expression ::=
  expression
  | msb_constant_expression **:** lsb_constant_expression
  | base_expression **+:** width_constant_expression
  | base_expression **-:** width_constant_expression

width_constant_expression ::= constant_expression

A.8.4 Primaries

constant_primary ::=
  constant_concatenation
  | constant_function_call
  | **(** constant_mintypmax_expression **)**
  | constant_multiple_concatenation
  | genvar_identifier
  | number
  | parameter_identifier
  | specparam_identifier

module_path_primary ::=
  number
  | identifier
  | module_path_concatenation
  | module_path_multiple_concatenation
  | function_call
  | system_function_call
  | constant_function_call
  | (module_path_mintypmax_expression)

primary ::=
  number
  | hierarchical_identifier
  | hierarchical_identifier **[** expression **]** { **[** expression **]** }
  | hierarchical_identifier **[** expression **]** { **[** expression **]** }
    [range_expression **]**
  | hierarchical_identifier **[** range_expression **]**
  | concatenation
  | multiple_concatenation
  | function_call
  | system_function_call
  | constant_function_call
  | **(** mintypmax_expression **)**

A.8.5 Expression left-side values

net_lvalue ::=
  hierarchical_net_identifier
  | hierarchical_net_identifier **[** constant_expression **]** { **[** constant_expression **]** }
  | hierarchical_net_identifier **[** constant_expression **]** { **[** constant_expression **]** }
    [constant_range_expression **]**
  | hierarchical_net_identifier **[** constant_range_expression **]**
  | net_concatenation

variable_lvalue ::=
hierarchical_variable_identifier
| hierarchical_variable_identifier [expression] { [expression] }
| hierarchical_variable_identifier [expression] { [expression] }
[range_expression]
| hierarchical_variable_identifier [range_expression]
| variable_concatenation

A.8.6 Operators

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

binary_operator ::=
+ | - | * | / | % | == | != | ~~===~~ | ~~!==~~ | && | || | **
| < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | >>> | <<<

unary_module_path_operator ::=
! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~

binary_module_path_operator ::=
== | != | && | || | & | | | ^ | ^~ | ~^

A.8.7 Numbers

number ::=
decimal_number
| octal_number
| binary_number
| hex_number
~~| real_number~~

~~real_number~~ ::=
unsigned_number . unsigned_number
| unsigned_number [. unsigned_number] exp [sign] unsigned_number

~~exp~~ ::= e | E

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 ::= non_zero_decimal_digit { _ | decimal_digit } unsigned_number ::= decimal_digit { **_** | decimal_digit }

binary_value ::= binary_digit { **_** | binary_digit }

octal_value ::= octal_digit { **_** | octal_digit }

hex_value ::= hex_digit { **_** | hex_digit }

decimal_base ::= **'**[**s**|**S**]**d** | **'**[**s**|**S**]**D**

binary_base ::= **'**[**s**|**S**]**b** | **'**[**s**|**S**]**B**

octal_base ::= **'**[**s**|**S**]**o** | **'**[**s**|**S**]**O**

hex_base ::= **'**[**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** | **?**

A.8.8 Strings

string ::= **"** { Any_ASCII_Characters_except_new_line } **"**

A.9 General

A.9.1 Attributes

attribute_instance ::= **(*** attr_spec { **,** attr_spec } ***)**

attr_spec ::=
attr_name **=** constant_expression
| attr_name

attr_name ::= identifier

A.9.2 Comments

comment ::=
one_line_comment
| block_comment

one_line_comment ::= **//** comment_text \n block_comment ::= /* comment_text */

comment_text ::= { Any_ASCII_character }

A.9.3 Identifiers

arrayed_identifier ::=
simple_arrayed_identifier
| escaped_arrayed_identifier

block_identifier ::= identifier

cell_identifier ::= identifier

config_identifier ::= identifier

escaped_arrayed_identifier ::= escaped_identifier [range]

escaped_hierarchical_identifier ::=
escaped_hierarchical_branch ~~{ .simple_hierarchical_branch | .escaped_hierarchical_branch }~~

escaped_identifier ::= \ {Any_ASCII_character_except_white_space} white_space

event_identifier ::= identifier

function_identifier ::= identifier

gate_instance_identifier ::= arrayed_identifier

generate_block_identifier ::= identifier

genvar_function_identifier ::= identifier /* hierarchy disallowed */

genvar_identifier ::= identifier

~~hierarchical_block_identifier~~ ::= hierarchical_identifier

~~hierarchical_event_identifier~~ ::= hierarchical_identifier

~~hierarchical_function_identifier~~ ::= hierarchical_identifier

hierarchical_identifier ::=
simple_hierarchical_identifier
| escaped_hierarchical_identifier

~~hierarchical_net_identifier~~ ::= hierarchical_identifier

~~hierarchical_variable_identifier~~ ::= hierarchical_identifier

~~hierarchical_task_identifier~~ ::= hierarchical_identifier

identifier ::=
simple_identifier
| escaped_identifier

inout_port_identifier ::= identifier

input_port_identifier ::= identifier

instance_identifier ::= identifier

library_identifier ::= identifier

memory_identifier ::= identifier

module_identifier ::= identifier

module_instance_identifier ::= arrayed_identifier

net_identifier ::= identifier

output_port_identifier ::= identifier

parameter_identifier ::= identifier

port_identifier ::= identifier

real_identifier ::= identifier

simple_arrayed_identifier ::= simple_identifier [range]

simple_hierarchical_identifier ::=
simple_hierarchical_branch ~~[.escaped_identifier]~~

simple_identifier ::= [**a-zA-Z_**] { [**a-zA-Z0-9_$**] }

specparam_identifier ::= identifier

~~system_function_identifier~~ ::= **$**[**a-zA-Z0-9_$**]{[**a-zA-Z0-9_$**]}

system_task_identifier ::= **$**[**a-zA-Z0-9_$**]{[**a-zA-Z0-9_$**]}

task_identifier ::= identifier

terminal_identifier ::= identifier

text_macro_identifier ::= simple_identifier

topmodule_identifier ::= identifier

udp_identifier ::= identifier

udp_instance_identifier ::= arrayed_identifier

variable_identifier ::= identifier

A.9.4 Identifier branches

simple_hierarchical_branch ::=
simple_identifier ~~[[unsigned_number]]~~
~~[{ .simple_identifier [[unsigned_number]] }]~~

escaped_hierarchical_branch ::=
escaped_identifier ~~[[unsigned_number]]~~
~~[{ .escaped_identifier [[unsigned_number]] }]~~

A.9.5 White space

white_space ::= space | tab | newline | eof

Annex B

(informative)

Functional mismatches

This annex describes certain situations where functional differences may arise between the RTL model and its synthesized netlist.

B.1 Non-deterministic behavior

The Verilog language has some inherent sources of non-determinism. In such cases, there is a potential for a functional mismatch. For example, statements without time-control constructs (# and @ expression constructs) in behavioral blocks do not have to be executed as one event. This allows for interleaving the execution of always statements in any order. In the following example, the behavior can be interpreted such that it is free to assign a value of either 0 or 1 to register *b*:

```
always @(posedge clock) begin
   a = 0;
   a = 1;
end

always @(posedge clock)
   b = a;
```

In this case, the synthesis tool is free to assign either 0 or 1 to register *b* as well, causing a potential functional mismatch.

When Verilog non-deterministic behavior permits different results from two different Verilog simulators that are both IEEE-compliant, this is known as a race condition. There are common Verilog guidelines that help to insure that a simulation race condition will not happen. This annex does not describe these common guidelines.

Most Verilog RTL models that include race conditions can cause a mismatch between pre-synthesis and post-synthesis simulations and should be avoided.

B.2 Pragmas

Pragmas must be used wisely since they can affect how synthesis interprets certain constructs. A pragma that directs the synthesis tool to do something different than what the simulator does should either be avoided or used with great caution. Some problematic synthesis pragmas are described in the following subclauses.

B.2.1 Full case attribute

The *full_case* attribute directs the synthesis tool to treat all undefined case items as synthesis don't care cases. Verilog simulators ignore undefined case items.

In the following decoder example, the simulator correctly sets the decoder outputs to 0 when the enable signal is low. Because the pragma *(* synthesis, full_case *)* has been added to this model, the synthesis tool recognizes that only four of eight possible case items have been defined and treats all other cases as don't care cases. With the don't care cases defined, the synthesis tool recognizes that the output is a don't care whenever the enable is low; therefore, the enable is a don't care so the enable is optimized out of the design, changing the functionality of the design. This will cause a mismatch between the pre-synthesis and post-synthesis simulations.

```
module decode4_fc (output reg [3:0] y, input [1:0] a, input en);
   // en is the enable signal.
   always @* begin
      y = 4'b0;
      (* synthesis, full_case *)
      case ({en, a})
         3'b1_00: y[a] = 1'b1;
         3'b1_01: y[a] = 1'b1;
         3'b1_10: y[a] = 1'b1;
         3'b1_11: y[a] = 1'b1;
      endcase
   end
endmodule
```

B.2.2 Parallel case attribute

The *parallel_case* attribute directs the synthesis tool to test each case item every time the case statement is executed. Verilog simulators only test case items until there is a match between the case expression and a case item. Once a case item is matched to the case expression, the case item statement is executed and an implied break causes the simulator to ignore the remaining case items.

In the following enable-decoder example, if the *en = 2'b11*, the simulation will execute the first case item statement, skipping the second, while the synthesis tool will execute the first two case item statements. This will cause a mismatch between the pre-synthesis and post-synthesis simulations.

```
module endec_pc (
      output reg       en_mem, en_cpu, en_io,
      input      [1:0] en);
   always @* begin
      en_mem=1'b0;
      en_cpu=1'b0;
      en_io =1'b0;
      (* synthesis, parallel_case *)
      casez (en)
         2'b1?:   en_mem=1'b1;
         2'b?1:   en_cpu=1'b1;
         default: en_io =1'b1;
      endcase
   end
endmodule
```

B.3 Using `ifdef

Conditionally compiling or omitting compilation of Verilog source code based on synthesis and simulation should be used with extreme caution.

In the following model, if the *SYNTHESIS* macro definition is not defined, the memory will be modeled with synthesizable RTL code. If the *SYNTHESIS* macro is defined, a vendor *xram* device will be instantiated into the design.

```
module ram_ifdef (
   output [7:0] q,
   input  [7:0] d,
   input  [6:0] a,
   input        clk, we);
   `ifndef SYNTHESIS
      // RTL model of a ram device for pre-synthesis simulation
      reg [7:0] mem [0:127];
      always @(posedge clk) if (we) mem[a] <= d;

      assign q = mem[a];
   `else
      xram ram1 (.dout(q), .din(d), .addr(a), .ck(clk), .we(we));
   `endif
endmodule

// Vendor ram model
module xram (
   output [7:0] dout,
   input  [7:0] din,
   input  [6:0] addr,
   input        ck, we);
   // Vendor ram model implementation
endmodule
```

Although selecting between a modeled RTL core device and an instantiated core device is one reason that conditional compilation would be used, it is the responsibility of the user to insure that there are no simulation-functional differences in the models that would cause a mismatch between pre-synthesis and post-synthesis simulations.

In the following model, the conditionally compiled code will clearly cause a mismatch between pre-synthesis and post-synthesis simulations. When the *SYNTHESIS* macro is defined, the *y* output is set equal to the bitwise-or of the *a* and *b* inputs. When the *SYNTHESIS* macro is not defined, the *y* output is set equal to the bitwise-and of the *a* and *b* inputs.

```
module and2_ifdef (output y, input [3:0] a, b);
   `ifdef SYNTHESIS
      assign y = a | b;
   `else
      assign y = a & b;
   `endif
endmodule
```

B.4 Incomplete sensitivity list

An incomplete sensitivity list on a combinational always block will typically cause a mismatch between pre-synthesis and post-synthesis simulations.

The following model is coded correctly to model a two-input and gate. Both and-gate inputs are listed in the combinational sensitivity list. There will be no mismatch between pre-synthesis and post-synthesis simulations using this model.

```
module myand1b (output reg y, input a, b);
   always @(a or b)
      y = a & b;
endmodule
```

In the following model, the *b* input is missing from the combinational sensitivity list. This model typically synthesizes to a two-input and gate but does not simulate correctly whenever the *b* input changes. This will cause a mismatch between pre-synthesis and post-synthesis simulations.

```
module myand1c (output reg y, input a, b);
   always @(a)
      y = a & b;
endmodule
```

In the following model, both inputs are missing from the combinational sensitivity list. This model typically synthesizes to a two-input and gate but does not simulate correctly. If this model is simulated, the simulator will hang as it loops in zero time, continuously executing the statement, *y = a & b*. This will cause a mismatch between pre-synthesis and post-synthesis simulations (the pre-synthesis simulation hangs).

```
module myand1d (output reg y, input a, b);
   always
      y = a & b;
endmodule
```

The @* combinational sensitivity list feature can be used to reduce redundant typing and to greatly reduce the number of errors that are introduced by coding incomplete sensitivity lists.

```
module myand1a (output reg y, input a, b);
   always @*
      y = a & b;
endmodule
```

B.5 Assignment statements mis-ordered

If assignment statements are mis-ordered in a combinational always block, incorrect logic may be generated with warnings.

The following model is coded correctly to model an and-or gate where the *a* and *b* inputs are and'ed together, and the result is or'ed with the *c* input. There will be no mismatch between pre-synthesis and post-synthesis simulations using this model.

```
module andor1a (output reg y, input a, b, c);
   reg tmp;

   always @* begin
      tmp = a & b;
      y   = tmp | c;
   end
endmodule
```

The following model is coded incorrectly to model an and-or gate where the *a* and *b* inputs are and'ed together, and the result is or'ed with the *c* input. The pre-synthesis simulation will not correctly update the or'ed output *y* after changes on the *a* and *b* inputs. There will be a mismatch between pre-synthesis and post-synthesis simulations using this model.

```
module andor1b (output reg y, input a, b, c);
   reg tmp;

   always @* begin
      y   = tmp | c;
      tmp = a & b;
   end
endmodule
```

B.6 Flip-flop with both asynchronous reset and asynchronous set

There is a small problem with the pre-synthesis model of a flip-flop with both asynchronous reset and asynchronous preset signals. The correct synthesizable model for this type of flip-flop is shown below.

```
module dffaras (output reg q, input d, clk, rst_n, set_n);
   always @(posedge clk or negedge rst_n or negedge set_n)
      if      (!rst_n) q <= 1'b0;
      else if (!set_n) q <= 1'b1;
      else             q <= d;
endmodule
```

The problem occurs when both reset and preset are asserted at the same time and reset is removed first. When reset is removed (**posedge** *rst_n*), the always block is not activated. This means that the output will continue to drive the reset output to '0' until the next rising clock edge. A real flip-flop of this type would immediately drive the output to '1' because the *set_n* signal is an asynchronous preset. This potentially could cause a mismatch between pre-synthesis and post-synthesis simulations using this model.

It should be noted that it is rare to design flip-flops with both asynchronous set and asynchronous reset, it is even more rare to use this type of flip-flop in a design where both reset and preset are permitted to be asserted at the same time and even more rare to allow reset to be removed before the preset is removed. It is estimated that fewer than 1% of all designs would ever be subject to this mismatch.

For the rare designs that do require assertion of both reset and preset and must permit removal of reset first, the following conditionally compiled code can be added to the above simulation model to correct the simulation problem. Note that this is only a rare simulation problem, not a synthesis problem.

```
`ifndef SYNTHESIS
   always @(rst_n or set_n)
      if (rst_n && !set_n) force q = 1'b1;
      else                 release q;
`endif
```

B.7 Functions

In general, synthesis tools always synthesize Verilog functions to combinational logic, even if the simulation behaves like a latch. The following is a correct model for a simple D-latch.

```
module latch1a (output reg y, input d, en);
   always @*
      if (en) y <= d;
endmodule
```

If the latching code is placed into a Verilog function, as shown in the following model, the simulation still behaves like a latch but synthesis tools generally infer combinational logic causing a mismatch between pre-synthesis and post-synthesis simulations.

```
module latch1b (output reg y, input d, en);
   always @*
      y <= lat(d, en);

   function lat (input d, en);
      if (en) lat = d;
   endfunction
endmodule
```

Verilog functions should be used with caution since there is no warning from a synthesis tool that latch-behavior coding will be synthesized to combinational logic.

B.8 Casex

The Verilog casex statement treats all **z**, **x**, and **?** bits as don't cares, whether they appear in the case expression or in the case item being tested. In the following model, if the *en* (enable) goes unknown during simulation, the *en_mem* output will be driven high. In a synthesized gate-level model, the outputs would most likely become unknown, thus indicating a design problem. This is a mismatch between pre-synthesis and post-synthesis simulations.

```
module endec_x (output reg en_mem, en_cpu, en_io, input [1:0] en);
   always @* begin
      en_mem = 1'b0;
      en_cpu = 1'b0;
      en_io = 1'b0;
      casex (en)
         2'b1?:   en_mem = 1'b1;
         2'b01:   en_cpu = 1'b1;
         default: en_io = 1'b1;
      endcase
   end
endmodule
```

It is too easy for a pre-synthesis simulation to have startup problems that cause signals to go unknown and to be treated as a don't care by the casex statement. For this reason, in general, the casex statement should be avoided.

B.9 Casez

The Verilog casez statement treats all **z** and **?** bits as don't cares, whether they appear in the case expression or in the case item being tested. In the following model, if both *en* (enable) bits go high during simulation, the *en_mem* output will be driven high. In a synthesized gate-level model, the outputs would most likely become unknown, thus indicating a design problem. This is a mismatch between pre-synthesis and post-synthesis simulations.


```
module endec_z (output reg en_mem, en_cpu, en_io, input [1:0] en);
   always @* begin
      en_mem = 1'b0;
      en_cpu = 1'b0;
      en_io = 1'b0;
      casez (en)
         2'b1?:   en_mem = 1'b1;
         2'b01:   en_cpu = 1'b1;
         default: en_io = 1'b1;
      endcase
   end
endmodule
```

It is unlikely (but not impossible) that a pre-synthesis simulation would experience stray high impedance values on most design signals. For this reason, in general, the casez statement is safe to use.

B.10 Making x assignments

Making a Verilog x-assignment to a signal tells the simulator to treat the signal as having an unknown value and tells the synthesis tool to treat the signal as a don't care. The synthesis tool will build a gate-level design using optimized gates that will not drive an unknown output on the signal. This means there is a mismatch between pre-synthesis and post-synthesis simulations for all x-assigned signals.

In the following 3-to-1 multiplexer model, the output is initialized to an **x** value and then updated based on the value of the *sel* signal. This design assumes that *sel* should never be equal to 2'b11. If the pre-synthesis simulation permits the *sel* signal to briefly pass through the 2'b11 pattern, the simulation will drive **x** to the output until the *sel* signal take on a valid select-pattern.

```
module mux3_x (output reg y, input [2:0] a, input [1:0] sel);
   always @* begin
      y = 1'bx; // synthesis "don't-care"
      case (sel)
         2'b00: y = a[0];
         2'b01: y = a[1];
         2'b10: y = a[2];
      endcase
   end
endmodule
```

The **x** value can be useful to help find bugs in the design during pre-synthesis simulations. It can also help direct the synthesis tool to optimize the design based on a don't care assignment. If the pre-synthesis simulation tests the output while it is unknown and if that unknown output is compared to a post-synthesis simulation, there will be a mismatch.

In general, the unknown output is short (unless there is a real design problem) and the output will be tested closer to a clock edge when the signal has had time to propagate to a known and correct value. Designers should just recognize that there is potential for a mismatch between pre-synthesis and post-synthesis simulations using this technique.

B.11 Assignments in variable declarations

A variable may be initialized in its declaration. Making assignments in the declaration forces the signal to a known value for pre-synthesis simulations. In general, no such initialization occurs in an actual gate level design. This can cause a mismatch between pre-synthesis and post-synthesis simulations and in general should be avoided.

```
// module dff_init (output reg q = 1'b0, input d, clk, rst_n);
module dff_init (q, d, clk, rst_n);
   output q;
   input  d, clk, rst_n;
   reg    q = 1'b0;

   always @(posedge clk or negedge rst_n)
      if (!rst_n) q <= 1'b0;
      else        q <= d;
endmodule
```

B.12 Timing delays

Synthesis tools ignore time delay in a model. Adding time delays to a Verilog pre-synthesis simulation can cause a mismatch between pre-synthesis and post-synthesis simulations and in general should be avoided.

In the following delay-line model, the latch enable output is delayed in a pre-synthesis simulation but the delay will be removed from the post-synthesis implementation, potentially causing a delayed latch signal to be enabled too soon.

```
`timescale 1ns/1ns
module delay1 (output reg latchendly, input latchen);
   always @*
      latchendly <= #25 latchen;
endmodule
```

Adding delay elements to a synthesized model typically requires instantiation of the delay element in the pre-synthesis RTL model.

