Recognized as an
American National Standard

IEEE P1364.1-2001
June 15, 2001

IEEE P1364.1 / D1.6
Draft Standard for Verilog® Register Transfer Level Synthesis

Prepared by the Verilog Synthesis Interoperability Working Group of the Design Automation Standards Committee

Sponsor

Design Automation Standards Committee of the IEEE Computer Society

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

ISBN XXXXXXXXXX

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying all patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Introduction

(This introduction is not a part of IEEE P1364.1, Draft Standard Register Transfer Level Subset Based on the Verilog® Hardware Description Language.)

This standard describes a standard syntax and semantics for Verilog HDL based RTL synthesis. It defines the subset of IEEE 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 the IEEE 1364 standard. 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 PAR was obtained 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:

Victor Berman
J. Bhasker (Chair)
David Bishop
Vassilios Gerousis
Don Hejna
Mike Quayle
Ambar Sarkar
Doug Smith
Yatin Trivedi
Rohit Vora

The Draft Std 1364.1 working group organization

Many individuals from many different organizations participated directly or indirectly in the standardization process. The main body of the IEEE P1364.1 working group is located in the United States.

The members of the IEEE P1364.1 working group had voting privileges, and all motions had to be approved by this group to be implemented. All task forces and subgroups focused on some specific areas, and their recommendations were eventually voted on by the IEEE P1364.1 working group.

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

Contents

Section 1

Overview

1.1 Scope

This standard defines a set of modeling rules for writing Verilog HDL descriptions. Adherence to these rules guarantee 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 is 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 section 1.2.1.
b) supports all pragmas defined in section 6
c) produces a netlist model that has the same functionality as the input model based on the conformance rules of section 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*).[1]

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 a construct, that is, 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. RTL synthesis does not expect to encounter 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**) and UPPER CASE letters to denote all other Verilog identifiers (such as SHIFTREG_A or N$657).
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.

1.5 Contents of this standard

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

1) **Overview**
   This section discusses the conventions used in this standard and its contents.
2) **References**
   This section contains bibliographic entries pertaining to this standard.
3) **Definitions**
   This section defines various terms used in this standard.
4) **Verification methodology**
   This section describes the guidelines for ensuring functtionality matches before and after synthesis.
5) **Modeling hardware elements**
   This section defines the styles for inferring special hardware elements.
6) **Pragmas**
   This section defines the pragmas that are part of this RTL synthesis subset.

[1]The wording of this paragraph is adapted from the *IEEE Standards Style Manual*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, c. 1992.

7) **Syntax**
This section describes the syntax of Verilog HDL supported for RTL synthesis.
8) **Annex A: Syntax summary**
Provides a summary of the syntax supported for synthesis.
9) **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.

Section 2

References

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

IEEE Std 1364-2001, IEEE Standard Verilog Language Reference Manual.

Section 3

Definitions

This section defines various terms used in this standard. Terms used within this standard but not defined in this section are assumed to be from the Language Reference Manual.

1) **don't care value**
The value **x** when used on the right-hand side of an assignment represents a don't care value.
2) **edge-sensitive storage device**
Any device mapped to by a synthesis tool that is edge-sensitive, for example, a flip-flop.
3) **event list**
Event list of an **always** statement.
4) **high-impedance value**
The value **z** represents a high impedance value.
5) **level-sensitive storage device**
Any device mapped to by a synthesis tool that is level-sensitive, for example, a latch.
6) **LRM**
The IEEE Verilog language reference manual, IEEE Std 1364-1995.

7) **meta-comment**
A Verilog comment (//) or (/* */) that is used to provide synthesis directives to a synthesis tool.
8) **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.
9) **RHS**
Right-hand side.
10) **RTL**
The register transfer level of modeling circuits in Verilog HDL.
11) **statically computable expression**
An expression whose value can be evaluated during compilation.
12) **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.
13) **user**
A person, system, process, or program that generates RTL Verilog HDL source code.
14) **vector**
A one-dimensional array.

Section 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.

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 element 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 storage device 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 shall be 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:

```
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. Immediately after the clock edge, the input values may be changed. 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 period 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 element, 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.

Section 5

Modeling hardware elements

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

The hardware inferences specified in this section 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 (this shall also include a net declaration assignment).

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

A variable assigned in an always statement cannot 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, variables that are assigned values using blocking assignments inside the always statement before being used by other expressions do not have to appear in the event list.

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

Example 5.1:

```
always @ (IN1 or IN2)
   OUT = IN1 + IN2;
// always statement models combinational logic.
```

Example 5.2:

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

Example 5.3:

```
always @ (IN)
   if (ENA)
      OUT = IN;
// 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 5.4:

```
always @ (IN1 or IN2 or SEL)
begin
   OUT = IN1; // Blocking assignment
   if (SEL)
      OUT <= 2; // Nonblocking assignment.
end
// Not supported, cannot mix blocking and nonblocking assignments in
// an always statement.
```

Example 5.5:

```
always @*      // Implicit event expression list 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 clock 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 5.6:

```
reg OUT;
. . .
always @ (posedge CLOCK)
   OUT <= IN;
// OUT is a positive edge triggered edge-sensitive storage device.
```

Example 5.7:

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

Example 5.8:

```
always @ (posedge CLOCK)
   if (RESET)
      OUT <= 1'b0;
   else
      OUT <= IN;
// OUT models a positive edge-triggered edge-sensitive storage device.
// The storage device may optionally have a synchronous reset.
```

Example 5.9:

```
always @ (posedge CLOCK)
   if (SET)
      OUT <= 1'b1;
   else
      OUT <= IN;
// OUT models a positive edge-triggered edge-sensitive storage device.
// The storage device may optionally have a synchronous reset.
```

Example 5.10:

```
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.
```

5.2.2.1 Edge-sensitive storage sevice 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 5.11:

```
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 5.12:

```
always @ (posedge CLOCK or posedge RESET)
   OUT <= IN;
// Not legal because the if statement is missing.
```

Example 5.13:

```
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 5.14:

```
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:

1) The variable is assigned a value in an always statement without edge events in its event list (combinational logic modeling style).
2) 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 prevent Verilog simulation race conditions.

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

Example 5.15:

```
always @ (ENABLE or D)
   if (ENABLE)
      Q <= D;
// A level-sensitive storage device is inferred for Q.
// If ENABLE is disasserted, Q will hold its value.
```

Example 5.16:

```
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. For a signal that has multiple drivers, if one of the drivers has an assignment of **z**, then all drivers shall have at least one assignment to **z**.

z assignments shall not propagate across variable assignments.

Example 5.17:

```
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;
   //  test2 is three-state when ena is low.
   assign test2 = (ena == 2'b10) ? 8'bz : test3;
   // Both drivers to test2 have an assignment of z.
endmodule
```

Example 5.18:

```
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 5.19:

```
always @ (IN)
begin
   TMP = 'bz;
   OUT = TMP; // OUT will never be driven by three state drivers.
end
```

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 section 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.

Section 6

Pragmas

A *pragma* is a generic term used to define a construct with no predefined language semantics that influences how a synthesis tool shall synthesize Verilog HDL code into a circuit. The following pragmas may appear with the Verilog HDL code.

i) Metacomments

ii) Attributes

6.1 Conditional compilation metacomments

Two metacomments provide for conditional compilation control.

a) // **rtl_synthesis off**
or
/* **rtl_synthesis off** */
b) // **rtl_synthesis on**
or
/* **rtl_synthesis on** */

A synthesis tool should ignore any metacomment (other than “rtl_synthesis on”) or any Verilog HDL construct after the “rtl_synthesis off” directive and before the “rtl_synthesis on” directive. The metacomments can appear in any combination of upper- and lower-case letters and can be inserted in any place where it is legal to insert a Verilog comment.

Note -- It is recommended that if a synthesis tool supports other pragmas to control the structure of the synthesized netlist, then the prefix “rtl_synthesis” be used in the metacomment.

6.2 Case decoding attributes

The following attributes shall be supported for decoding **case** statements and shall have the following interpretations:

a) **(* full_case *)**
This attribute shall communicate to the synthesis tool that the case choices specified are exhaustive.
b) **(* parallel_case *)**
This attribute shall communicate to the synthesis tool that the case choices specified are mutually exclusive.

These attributes shall have the special interpretation when specified with a case statement.

The full_case and parallel_case attributes may also appear as a single attribute instance, such as:

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

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)
   case (sel) (* full_case    *)
    3'b001:  out = op1;
    3'b010:  out = op2;
    3'b100:  out = op3;
   endcase
```

is equivalent to the much safer,

```
always @(sel) begin
   out = 'bx;
   case (sel)
    3'b001:  out = op1;
    3'b010:  out = op2;
    3'b100:  out = op3;
   endcase
end
```

Section 7

Syntax

Note: The subsections under this section are described using the same section hierarchy as described in the 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. Also see Section 6 on “Pragmas”.

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 Section 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 Section 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 } non_blocking 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 : expression 3

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
```

Attribute instances for a case statement shall be supported, and the only two attributes that are supported are “full_case” and “parallel_case”. See section 6.2.

7.2 Data types

7.2.1 Value set

Supported. See Section 5.5 on support for values x amd 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

```
Shall not be 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.

7.2.8 regs

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

7.2.9 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 ]
```

7.2.9.1 Operators and real numbers

Not supported.

7.2.9.2 Conversion

Not supported.

7.2.10 Arrays

Supported.

7.2.10.1 Net arrays

Supported.

7.2.10.2 reg and variable arrays

Supported.

7.2.10.3 Memories

Supported.

7.2.11 Parameters

7.2.11.1 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 ]
```

7.2.11.2 Local parameters - localparam

Supported.

7.2.11.3 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_descript
or
   = ( 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 ]
```

7.2.12 Name spaces

Supported.

7.3 Expressions

7.3.1 Operators

Supported. See also subsections 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 operators *, / and % shall only be supported when the second operand is a power of 2.

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 Section 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
```

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 section 5 shall be supported.

7.7.2.2 The non blocking procedural assignment

```
non-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
```

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 section 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 a **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 )
   |@ *
   |@ ( * )

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 section 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 section 5 shall be supported. Furthermore, event control shall appear only in the top-level statement (the statement with the always keyword) as described in Section 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

non-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
```

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
```

7.7.9.2 Always construct

```
always_construct ::= always statement
```

Section 5 describes how always statement can be used to model logic elements.

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 } ) ] ;
```

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.

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 } )
```

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_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 }
```

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

Ignored.

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 are ignored.

All system functions are not supported except for $signed and $unsigned.

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

Syntax summary

(informative)

This annex summarizes, using Backus-Naur Form (BNF), the syntax that is supported for RTL 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_descript
or
   = ( 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

non-blocking 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 } non_blocking 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 : expression 3

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

Functional Mismatches

(informative)

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)
   begin
      B = A;
   end
```

In this case, the synthesis tool is free to assign either 0 or 1 to register B as well, causing a potential functional mismatch.

B.2 Pragmas

Pragmas must be used wisely since they can affect how synthesis interprets certain constructs. For example, if a user specifies the parallel case directive but the case items are not independent, the behavior of the synthesis results may not match that of the RTL model.