SystemVerilog 3.0

Accellera's Extensions to Verilog®

Abstract: a set of extensions to the IEEE 1364-2001 Verilog Hardware Description Language to aid in the creation and verification of abstract architectural level models

SystemVerilog 3.0

Accellera’s Extensions to Verilog®

Abstract: a set of extensions to the IEEE 1364-2001 Verilog Hardware Description Language to aid in the creation and verification of abstract architectural level models

Approved by the Accellera Board of Directors on 3 June 2002.

Acknowledgements

This SystemVerilog 3.0 reference manual was developed by experts from many different fields, including design and verification engineers, Electronic Design Automation (EDA) companies, EDA vendors, and members of the IEEE 1364 Verilog standard working group. The primary contributors to the development of SystemVerilog 3.0 include:

Vassilios Gerousis, Chair
Dave Kelf, Co-chair
Stefen Boyd
Dennis Brophy
Kevin Cameron
Cliff Cummings
Simon Davidmann
Tom Fitzpatrick
Peter Flake
Harry Foster
Paul Graham
David Knapp
Adam Krolnik
Mike McNamara
Phil Moorby
Prakash Narian
Anders Nordstrom
Rajeev Ranjan
John Sanguinetti
David Smith
Alec Stanculescu
Stuart Sutherland
Bassam Tabbara
Andy Tsay

Stuart Sutherland served at the technical editor for this document. Stefen Boyd served as editor of the BNF annex.

Table of Contents

Section 1
Introduction to SystemVerilog

This document specifies the Accellera extensions for a higher level of abstraction for modeling and verification with the Verilog Hardware Description Language. These additions extend Verilog into the systems space and the verification space and was built on top of the work of the IEEE Verilog 2001 committee.

Throughout this document:

— "Verilog" or "Verilog-2001" refers to the IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language

— "SystemVerilog" refers to the Accellera extensions to the Verilog-2001 standard.

This document numbers the generations of Verilog as follows:

— **"Verilog 1.0"** is the IEEE Std. 1364-1995 Verilog standard, which is also called Verilog-1995

— **"Verilog 2.0"** is the IEEE Std. 1364-2001 Verilog standard, commonly called Verilog-2001; this generation of Verilog contains the first significant enhancements to Verilog since its release to the public in 1990

— **"SystemVerilog 3.0"** is Verilog-2001 plus an extensive set of high-level abstraction extensions, as defined in this document

The Accellera initiative to extend Verilog is an ongoing effort under the direction of the Accellera HDL+ Technical Subcommittee. This committee will continue to define additional enhancements to Verilog beyond SystemVerilog 3.0.

SystemVerilog 3.0 is built on top of Verilog 2001. SystemVerilog improves the productivity, readability, and reusability of Verilog based code. The language enhancements in SystemVerilog provide more concise hardware descriptions, while still providing an easy route with existing tools into current hardware implementation flows.

SystemVerilog adds several new constructs to Verilog-2001, including:

— C data types to provide better encapsulation and compactness of code
  — int, char, typedef, struct, union, enum
— Enhancements to existing Verilog constructs, to provide tighter specifications
  — Extensions to always blocks to include linting type features
  — Logic (0, 1, X, Z) and bit (0, 1) data types
  — Automatic/static specification on a per variable instance basis
  — Procedural break, continue, return
— Interfaces to encapsulate communication and facilitate "Communication Oriented" design
— Dynamic processes for modeling pipelines
— A $root top level hierarchy which can have global definitions

Section 2
Literal Values

2.1 Introduction (informative)

The lexical conventions for SystemVerilog literal values are extensions of those for Verilog. SystemVerilog adds literal time values, literal array values, literal structures and enhancements to literal strings.

2.2 Literal value syntax

```
time_literal ::=          // from Annex A.8.4
          unsigned_number time_unit
        | fixed_point_number time_unit
time_unit ::= s | ms | us | ns | ps | fs

number ::=          // from Annex A.8.7
          decimal_number
        | octal_number
        | binary_number
        | hex_number
        | real_number
decimal_number ::=
          unsigned_number
        | [ size ] decimal_base  unsigned_number
        | [ size ] decimal_base  x_digit { _ }
        | [ size ] decimal_base  z_digit { _ }
binary_number ::= [ size ] binary_base  binary_value
octal_number ::= [ size ] octal_base  octal_value
hex_number ::= [ size ] hex_base  hex_value
sign ::= + | -
size ::= non_zero_unsigned_number
non_zero_unsigned_number ::= non_zero_decimal_digit { _ | decimal_digit}
real_number ::=
          fixed_point_number
        | unsigned_number [ . unsigned_number ] exp [ sign ] unsigned_number
fixed_point_number ::= unsigned_number . unsigned_number
exp ::= e | E
unsigned_number[1] ::= decimal_digit { _ | decimal_digit }

string ::= " { Any_ASCII_Characters_except_new_line } "          // from Annex A.8.8
```

Syntax 2-1—Literal values (excerpt from Annex A)

2.3 Integer and logic literals

Literal integer and logic values can be sized or unsized, and follow the same rules for signedness, truncation and left-extending as Verilog-2001.

SystemVerilog adds the ability to specify unsized literal single bit values with a preceding apostrophe (**'**), but without the base specifier. All bits of the unsized value are set to the value of the specified bit.

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

2.4 Real literals

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

A cast can be used to convert literal **`real`** values to the **`shortreal`** type (e.g. `shortreal'(1.2)`). Casting is described in section 3.8.

2.5 Time literals

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

```
0.1ns
40ps
```

2.6 String literals

A string literal is enclosed in quotes and has its own data type. Non-printing and other special characters are preceded with a backslash. SystemVerilog adds the following special string characters:

\v vertical tab
\f form feed
\a bell
\x02 hex number

A string literal can be assigned to a character, or a packed array, as in Verilog-2001. If the size differs, it is right justified.

```
char c1 = "A" ; bit [7:0] d = "\n" ;
bit [0:11] [7:0] c2 = "hello world\n" ;
```

A string literal can be assigned to an unpacked array of characters, and a zero termination is added like in C. If the size differs, it is left justified.

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

Packed and unpacked arrays are discussed in section 4. The difference between string literals and array literals is discussed in section 2.7, which follows.

String literals can also be cast to a packed or unpacked array, which shall follow the same rules as assigning a literal string to a packed or unpacked array. Casting is discussed in section 3.8.

2.7 Array literals

Arrays literals are similar to C initializers, but with the replicate operator (**`{{}}`**) allowed.

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

The nesting of braces must follow the number of dimensions, unlike in C. However, replicate operators can be nested.

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

If the type is not given by the context, it must be specified with a cast.

```
typedef int [1:3] triple; // 3 integers packed together
b = triple'{0,1,2};
```

2.8 Structure literals

Structure literals are similar to C initializers. Structure literals must have a type, either from context or a cast.

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

Nested braces should reflect the structure. For example:

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

Note that the C alternative `{1, 1.0, 2, 2.0}` is not allowed.

Section 3
Data Types

3.1 Introduction (informative)

To provide for clear translation to and from C, SystemVerilog supports the C built-in types, with the meaning given by the implementation C compiler. However, to avoid the duplication of **`int`** and **`long`** without causing confusion, in SystemVerilog, **`int`** is 32 bits and **`longint`** is 64 bits. The C **`float`** type is called **`shortreal`** in SystemVerilog, so that it will not be confused with the Verilog-2001 **`real`** type.

Verilog-2001 has net data types, which may have 0, 1, X or Z, plus 7 strengths, giving 120 values. It also has variable data types such as **`reg`**, which have 4 values 0, 1, X, Z. These are not just different data types, they are used differently. SystemVerilog adds another 4-value data type, called **`logic`**. See section 3.3.1.

Verilog-2001 provides arbitrary fixed length arithmetic using **`reg`** data types. The **`reg`** type can have bits at X or Z, however, and so are less efficient than an array of bits, because the operator evaluation must check for X and Z, and twice as much data must be stored. SystemVerilog adds a **`bit`** type which can only have bits with 0 or 1 values. See section 3.3.1 on 2-state data types.

Automatic type conversions from a smaller number of bits to a larger number of bits involve zero extensions if unsigned or sign extensions if signed, and do not cause warning messages. Automatic truncation from a larger number of bits to a smaller number does cause a warning message. Automatic conversions between **`logic`** and **`bit`** do not cause warning messages. To convert a logic value to a bit, 1 converts to 1, anything else to 0.

User defined types are introduced by **`typedef`** and must be defined before they are used. Data types can also be parameters to modules or interfaces, making them like class templates in object-oriented programming. One routine can be written to reverse the order of elements in any array, which is impossible in C and in Verilog.

Structures and unions are complicated in C, because the tags have a separate name space. SystemVerilog follows the C syntax, but without the optional structure tags.

See also Section 4 on arrays.

3.2 Data type syntax

```
data_type ::=          // from Annex A.2.2.1
            integer_vector_type [ signing ] { packed_dimension } [ range ]
          | integer_atom_type [ signing ] { packed_dimension }
          | type_declaration_identifier
          | non_integer_type
          | struct { { struct_union_member } }
          | union { { struct_union_member } }
          | enum { enum_identifier [ = constant_expression ]
            { , enum_identifier [ = constant_expression ] } }
          | void
integer_type ::= integer_vector_type | integer_atom_type
integer_atom_type ::= byte | char | shortint | int | longint | integer
integer_vector_type ::= bit | logic | reg
non_integer_type ::= time | shortreal | real | realtime | $built-in
signing ::= [ signed ] | [ unsigned ]
simple_type ::= integer_type | non_integer_type | type_identifier
struct_union_member ::= data_type  list_of_variable_identifiers_or_assignments ;
```

Syntax 3-1—data types (excerpt from Annex A)

3.3 Integer data types

SystemVerilog offers several integer data types, representing a hybrid of both Verilog and C data types:

Table 3-1—Integer data types

`char`	2-state C data type, usually an 8 bit signed integer (ASCII) or a short int (Unicode)
`shortint`	2-state SystemVerilog data type, 16 bit signed integer
`int`	2-state SystemVerilog data type, 32 bit signed integer
`longint`	2-state SystemVerilog data type, 64 bit signed integer
`byte`	2-state SystemVerilog data type, 8 bit signed integer
`bit`	2-state SystemVerilog data type, user-defined vector size
`logic`	4-state SystemVerilog data type, user-defined vector size with different use rules from reg
`reg`	4-state Verilog-2001 data type, user-defined vector size
`integer`	4-state Verilog-2001 data type, at least 32 bit signed integer

3.3.1 2-state (two-value) and 4-state (four-value) data types

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

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

3.3.2 Signed and unsigned data types

Integer types use integer arithmetic and can be signed or unsigned. This affects the meaning of certain operators such as '<', etc.

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

The data types **char**, **byte**, **shortint**, **int**, **integer** and **longint** default to **signed**. The data types **bit**, **reg** and **logic** default to **unsigned**, as do arrays of these types.

Note that the **signed** keyword is part of Verilog-2001. The **unsigned** keyword is a reserved keyword in Verilog-2001, but is not utilized.

See also section 7, on operators and expressions.

3.4 Other basic data types

3.4.1 Time data types

Time is a special data type. It is a 64 bit integer of time steps. The default time step follows the rules of IEEE Verilog standard. The time step can be changed by the **timeprecision** declaration. It can also be changed by a **`timescale** directive.

The **timeprecision** declaration affects the local accuracy of delays.

```
module m;
   timeprecision 0.1ns;
   initial #10.11ns a = 1; // round to #10.1ns according to time precision
endmodule
```

The **timeunit** declaration is used to set the current time unit. When a literal time is expressed in SystemVerilog, it can be given with explicit time units, e.g. `12ns`. If no time units are specified, the literal number is multiplied by the current time unit. Time values are scaled to the time precision of the module, following the rules of Verilog-2001. An integer or real variable is cast to a time value by using the integer or real as a delay.

For example:

```
#10.11; // multiply by time unit and round according to time precision
```

See section 12.6 for more information on setting the time units and time precision.

3.4.2 Real and shortreal data types

The **real**[1] data type is from Verilog-2001, and is the same as a C **double**. The **shortreal** data type is a SystemVerilog data type, and is the same as a C **float**.

3.4.3 Void data type

The **void** data type represents non-existent data. This type can be specified as the return type of functions, indicating no return value.

[1] The real and shortreal types are represented as described by IEEE 734-1985, an IEEE standard for floating point numbers.

3.5 User-defined types

```
type_declaration ::=          // from Annex A.2.1.3
          typedef data_type type_declaration_identifier ;
        | typedef interface_identifier { [ constant_expression ] } . type_identifier
             type_declaration_identifier ;
```

Syntax 3-2—user-defined types (excerpt from Annex A)

The user can define a new type using **`typedef`**, as in C.

```
typedef int intP;
```

This can then be instantiated as:

```
intP a, b;
```

A type can be used before it is defined, provided it is first identified as a type by an empty typedef:

```
typedef foo;
foo f = 1;
typedef int foo;
```

Note that this does not apply to enumeration values, which must be defined before they are used.

If the type is defined within an interface, it must be re-defined locally before being used.

```
interface it;
   typedef int intP;
endinterface

it it1;
typedef it1.intP intP;
```

User-defined type names must be used for complex data types in casting (see section 3.7, below), and as parameters.

3.6 Enumerations

```
data_type ::=          // from Annex A.2.2.1
          ...
        | enum [ integer_type [ signing ] { packed_dimension } ]
             { enum_identifier [ = constant_expression ] { , enum_identifier [ = constant_expression ] } }
```

Syntax 3-3—enumerated types (excerpt from Annex A)

Enumerated data types provide the capability to abstractly declare strongly typed variables without either a data type or data value(s) and later add the required data type and value(s) for designs that require more definition. Enumerated data types also can be easily referenced or displayed using the enumerated names as opposed to the enumerated values.

In the absence of a data type declaration, the default data type shall be **int**. Any other data type used with enumerated types shall require an explicit data type declaration.

An enumerated type defines a set of named values. In the following example, “light1” and “light2” are defined to be variables of the anonymous (unnamed) enumerated int type that includes the three members: “red”, “yellow” and “green.”

```
enum {red, yellow, green} light1, light2; // anonymous int type
```

An enumerated name with x or z assignments assigned to an enum with no explicit data type declaration shall be a syntax error.

```
// Syntax error: IDLE=2’b00, XX=2’bx <ERROR>, S1=2’b01??, S2=2’b10??
enum {IDLE, XX=’x, S1=2’b01, S2=2’b10} state, next;
```

An **enum** declaration of a 4-state type, such as integer, that includes one or more names with x or z assignments shall be permitted.

```
// Correct: IDLE=2’b00, XX=2’bx, S1=2’b01, S2=2’b10
enum integer {IDLE, XX=’x, S1=2’b01, S2=2’b10} state, next;
```

An unassigned enumerated name that follows and enum name with x or z assignments shall be a syntax error.

```
// Syntax error: IDLE=2’b00, XX=2’bx, S1=??, S2=??
enum integer {IDLE, XX=’x, S1, S2} state, next;
```

The values can be cast to integer types, and increment from an initial value of 0. This can be overridden.

```
enum {bronze=3, silver, gold} medal; // silver=4, gold=5
```

The values can be set for some of the names and not set for other names. A name without a value is automatically assigned an increment of the value of the previous name.

```
// c is automatically assigned the increment-value of 8
enum {a=3, b=7, c} alphabet;
```

If an automatically incremented value is assigned elsewhere in the same enumeration, this shall be a syntax error.

```
// Syntax error: c and d are both assigned 8
enum {a=0, b=7, c, d=8} alphabet;
```

If the first name is not assigned a value, it is given the initial value of 0.

```
// a=0, b=7, c=8
enum {a, b=7, c} alphabet;
```

A sized constant can be used to set the size of the type. All sizes must be the same.

```
// silver=4’h4, gold=4’h5 (all are 4 bits wide)
enum {bronze=4’h3, silver, gold} medal4;
```

A type name can be given so that the same type can be used in many places.

```
typedef enum {NO, YES} boolean;
boolean myvar; // named type
```

Adding a constant range to the **enum** declaration can be used to set the size of the type. If any of the enum members are defined with a different sized constant, this shall be a syntax error.

```
// Error in the bronze and gold member declarations
enum [3:0] {bronze=5'h13, silver, gold=3'h5} medal4;

// Correct declaration - bronze and gold sizes are redundant
enum [3:0] {bronze=4'h13, silver, gold=4'h5} medal4;
```

The type is checked in assignments, arguments and relational operators (which check the values). Like C, there is no overloading of literals, so medal and medal4 cannot be defined in the same scope, since they contain the same names.

3.7 Structures and Unions

data_type ::=   // from Annex A.2.2.1
  ...
  | **struct {** { struct_union_member } **}**
  | **union {** { struct_union_member } **}**

struct_union_member ::= data_type list_of_variable_identifiers_or_assignments **;**

Syntax 3-4—structures and unions (excerpt from Annex A)

Structure and union declarations follow the C syntax, but without the optional structure tags before the '**{**'.

```
struct { bit[7:0] opcode; bit [23:0] addr; }IR; // anonymous structure defines
variable IR
   IR.opcode = 1; // set field in IR.
```

Some additional examples of declaring structure and unions are:

```
typedef struct {
            bit[7:0] opcode;
            bit [23:0] addr;
} instruction; // named structure type
instruction IR; // define variable

typedef union { int i; shortreal f; } num; // named union type
   num n;
n.f = 0.0; // set n in floating point format

typedef struct {
            bit isfloat;
            union { int i; shortreal f; } n; // anonymous type
} tagged; // named structure

tagged a[9:0]; // array of structures
```

A structure can be assigned as a whole, and passed to or from a function or task as a whole.

Section 2.8 discusses assigning initial values to a structure.

A packed structure consists of bit fields, which are packed together in memory without gaps. This means that they are easily converted to and from bit vectors. An unpacked structure has an implementation-dependent packing, normally matching the C compiler.

Like a packed array, a packed structure can be used as a whole with arithmetic and logical operators. The first member specified is the most significant. The structures are declared using the packed keyword, which can be followed by the signed or unsigned keywords, according to the desired arithmetic behavior, which defaults to unsigned:

```
struct packed signed {
   int a;
   shortint b;
   byte c;
   bit [7:0] d;
} pack1; // signed, 2-state

struct packed unsigned {
   time a;
   integer b;
   logic [31:0] c;
} pack2; // unsigned, 4-state
```

If any data type within a packed structure is masked, the whole structure is treated as masked. Any unmasked members are converted as if cast, i.e. an X will be read as 0 if it is in a member of type bit. One or more elements of the packed array may be selected, assuming an [n-1:0] numbering:

```
pack1 [15:8] // c
```

Non-integer data types, such as **`real`** and **`shortreal`**, are not allowed in packed structures or unions. Nor are unpacked arrays.

A packed structure can be used with a **`typedef`**.

```
typedef struct packed { // default unsigned
   bit [3:0] GFC;
   bit [7:0] VPI;
   bit [11:0] VCI;
   bit CLP;
   bit [3:0] PT ;
   bit [7:0] HEC;
   bit [47:0] [7:0] Payload;
   bit [2:0] filler;
} s_atmcell;
```

A packed union contains members that are packed structures or arrays of the same size. This ensures that you can read back a union member that was written as another member. If any member is 4-state, the whole union is 4-state. A packed union can also be used as a whole with arithmetic and logical operators, and its behavior is determined by the **`signed`** or **`unsigned`** keyword, the latter being the default.

For example, a union can be accessible with different access widths:

```
typedef union packed { // default unsigned
   s_atmcell acell;
   bit [423:0] bit_slice;
   bit [52:0][7:0] byte_slice;
} u_atmcell;

u_atmcell u1;
byte b; bit [3:0] nib;
b = u1.bit_slice[415:408]; // same as b = u1.byte_slice[51];
nib = u1.bit_slice [423:420]; // same as nib = u1.acell.GFC;
```

Note that writing one member and reading another is independent of the byte ordering of the machine, unlike a normal union of normal structures, which are C-compatible and have members in ascending address order.

3.8 Casting

```
primary ::=      // from Annex A.8.4
            ...
          | simple_type_or_number ' ( expression )
          | simple_type_or_number ' { expression { , expression } }
          | simple_type_or_number ' { expression { expression } }

simple_type_or_number ::= // from Annex A.2.2.1
            simple_type | number
simple_type ::= // from Annex A.2.2.1
            integer_type | non_integer_type | type_identifier
```

Syntax 3-5—casting (excerpt from Annex A)

A data type may be changed by using a cast (**'**) operation. The expression to be cast must be enclosed in parenthesis or within concatenation or replication braces.

```
int'(2.0 * 3.0)
shortint'{8'hFA,8'hCE}
```

A decimal number as a data type means a number of bits.

```
17'(x - 2)
```

The signedness can also be changed.

```
signed'(x)
```

A user-defined type can be used.

```
mytype'(foo)
```

When casting to a predefined type, the prefix of the cast must be the predefined type keyword. When casting to a user-defined type, the prefix of the cast must be the user-defined type identifier.

When a **`shortreal`** is converted to an **`int`**, its value is rounded as in Verilog. So the conversion can lose information. When a **`shortreal`** is converted to 32 bits, its bit pattern is preserved, which means it can be converted back to the same value without any loss of information. This technique can also be used for structures, where the **`$bits`** attribute gives the size of a structure in bits (the $bits system function is discussed in section 16.2):

```
typedef struct {
            bit isfloat;
            union { int i; shortreal f; } n; // anonymous type
} tagged; // named structure

typedef bit [$bits(tagged) - 1 : 0] tagbits; // tagged defined above

tagged a [7:0]; // unpacked array of structures
```

```
    tagbits t = tagbits'(a[3]); // convert structure to array of bits
    a[4] = tagged'(t); // convert array of bits back to structure
```

Note that the **`bit`** data type loses X values. If these are to be preserved, the logic type should be used instead.

The size of a union in bits is the size of its largest member. The size of a logic in bits is 1.

For compatibility, the Verilog functions **`$itor`**, **`$rtoi`**, **`$bitstoreal`**, **`$realtobits`**, **`$signed`**, **`$unsigned`** can also be used.

Section 4
Arrays

4.1 Introduction (informative)

In C, arrays are indexed from 0 by integers, or converted to pointers. Although the whole array can be initialized, each element must be read or written separately in procedural statements.

In Verilog-2001, arrays are indexed from left-bound to right-bound. If they are vectors, they can be assigned as a single unit, but not if they are arrays. Verilog-2001 allows multiple dimensions.

In Verilog-2001, all data types can be declared as arrays. The **reg**, **wire** and all other net types can also have a vector width declared. A dimension declared before the object name is referred to as the “vector width” dimension. The dimensions declared after the object name are referred to as the “array” dimensions.

```
reg [7:0] r1 [1:256];   // [7:0] is the vector width, [1:256] is the array size
```

SystemVerilog enhances array declarations in several ways.

4.2 Packed and unpacked arrays

SystemVerilog uses the term “***packed array***” to refer to the dimensions declared before the object name (what Verilog-2001 refers to as the vector width). The term “***unpacked array***” is used to refer to the dimensions declared after the object name.

```
bit [7:0] c1;         // packed array
real u [7:0];         // unpacked array
```

A packed array is a mechanism for subdividing a vector into subfields which can be conveniently accessed as array elements. Consequently, a packed array is guaranteed to be represented as a contiguous set of bits. An unpacked array may or may not be so represented. A packed array differs from an unpacked array in that, when a packed array appears as a primary, it is treated as a single vector.

If a packed array is declared as signed, then the array viewed as a single vector shall be signed. A part-select of a packed array shall be unsigned.

Packed arrays allow arbitrary length integer types, so a 48 bit integer can be made up of 48 bits. These integers can then be used for 48 bit arithmetic. The maximum size of a packed array may be limited, but shall be at least 65536 (2^{16}) bits.

Packed arrays can only be made of the single bit types: **bit**, **logic**, **reg**, **wire**, and the other net types. Unpacked arrays can be made up of any type.

Integer types with predefined widths cannot have packed array dimensions declared. These types are: **char**, **byte**, **shortint**, **int**, **longint**, and **integer**. An integer type with a predefined width can be treated as a single dimension packed array. The packed dimensions of these integer types shall be numbered down to 0, such that the right-most index is 0.

```
byte c2;    // same as bit [7:0] c2;
integer i1; // same as logic signed [31:0] i1;
```

The following operations can be performed on all arrays, packed or unpacked. The examples provided with these rules assume that A and B are arrays.

— Reading and writing the array, e.g., `A = B`

— Reading and writing a slice of the array, e.g., `A[i:j] = B[i:j]`

— Reading and writing a variable slice of the array, e.g., `A[x+:c] = B[y+:c]`

— Reading and writing an element of the array, e.g., `A[i] = B[i]`

The following operations can be performed on packed arrays, but not on unpacked arrays. The examples provided with these rules assume that A is an array.

— Assignment from an integer, e.g., `A = 8'b11111111;`

— Treatment as an integer in an expression, e.g., `(A + 3)`

When assigning to an unpacked array, the source and target must be arrays with the same number of unpacked dimensions, and the length of each dimension must be the same. Assignment to an unpacked array is done by assigning each element of the source unpacked array to the corresponding element of the target unpacked array. Note that an element of an unpacked array may be a packed array.

For the purposes of assignment, a packed array is treated as a vector. Any vector expression can be assigned to any packed array. The packed array bounds of the target packed array do not affect the assignment. A packed array cannot be assigned to an unpacked array.

4.3 Multiple dimensions

Like Verilog memories, the dimensions following the type set the packed size. The dimensions following the instance set the unpacked size.

```
bit [3:0] [7:0] joe [1:10]; // 10 entries of 4 bytes (packed into 32 bit int)
```

can be used as follows:

```
joe[9] = joe[8] + 1; // 4 byte add
joe[7][3:2] = joe[6][1:0]; // 2 byte copy
```

Note that the dimensions declared following the type and before the name (`[3:0][7:0]` in the preceding declaration) vary more rapidly than the dimensions following the name (`[1:10]` in the preceding declaration). When used, the first dimensions (`[3:0]`) follow the second dimensions (`[1:10]`).

In a list of dimensions, the right-most one varies most rapidly, as in C. However a packed dimension varies more rapidly than an unpacked one.

```
bit [1:10] foo1 [1:5];  // 1 to 10 varies most rapidly; compatible with
                           Verilog-2001 arrays
bit foo2 [1:5] [1:10];  // 1 to 10 varies most rapidly, compatible with C

bit [1:5] [1:10] foo3;  // 1 to 10 varies most rapidly

bit [1:5] [1:6] foo4 [1:7] [1:8];   // 1 to 6 varies most rapidly, followed by
                                       1 to 5, then 1 to 8 and then 1 to 7
```

Multiple packed dimensions can also be defined in stages with **`typedef`**.

```
typedef bit [1:5] bsix;
bsix [1:10] foo5; // 1 to 5 varies most rapidly
```

Multiple unpacked dimensions can also be defined in stages with **`typedef`**.

```
typedef bsix mem_type [0:3];  // array of four 'bsix' elements
mem_type bar [0:7];           // array of eight 'mem_type' elements
```

When the array is used with a smaller number of dimensions, these have to be the slowest varying ones.

```
bit [9:0] foo6;
foo5 = foo1[2]; // a 10 bit quantity.
```

As in Verilog-2001, a comma-separated list of array declarations can be made. All arrays in the list will have the same data type and the same packed array dimensions.

```
bit [7:0] [31:0] foo7 [1:5] [1:10], foo8 [0:255]; // two arrays declared
```

If an index expression is of a 4-state type, and the array is of a 4-state type, an **X** or Z in the index expression will cause a read to return **X**, and a write to issue a run-time warning. If an index expression is of a 4-state type, but the array is of a 2-state type, an **X** or Z in the index expression shall generate a run-time warning and be treated as **0**. If an index expression is out of bounds, a run-time warning may be generated.

Out of range index values shall be illegal for both reading from and writing to an array of 2-state variables, such as **int**. The result of an out of range index value is indeterminate. Implementations shall generate a warning if an out of range index occurs for a read or write operation.

4.4 Indexing and slicing of arrays

An expression can select part of a packed array, or any integer type, which is assumed to be numbered down to 0.

SystemVerilog uses the term “part select” to refer to a selection of one or more contiguous bits of a single dimension packed array. This is consistent with the usage of the term “part select” in Verilog.

```
reg [63:0] data;
reg [7:0] byte2;
byte2 = data[23:16]; // an 8-bit part select from data
```

SystemVerilog uses the term “slice” to refer to a selection of one or more contiguous elements of an array. Verilog only permits a single element of an array to be selected, and does not have a term for this selection.

An single element of a packed or unpacked array can be selected using an indexed name.

```
bit [3:0] [7:0] j;   // j is a packed array
byte k;
k = j[2]; // select a single 8-bit element from j
```

One or more contiguous elements can be selected using a slice name. A slice name of a packed array is a packed array. A slice name of an unpacked array is an unpacked array.

```
bit busA [7:0] [31:0] ;    // unpacked array of 8 32-bit vectors
int busB [1:0];            // unpacked array of 2 integers
busB = busA[7:6];          // select a slice from busA
```

The size of the part select or slice must be constant, but the position may be variable. The syntax of Verilog-2001 is used.

```
int i = bitvec[j +: k];    // k must be constant.
a = {(b[c -: d]), e};      // d must be constant
```

Slices of an array can only apply to one dimension, but other dimensions may have single index values in an expression.

4.5 Array querying functions

SystemVerilog provides new system functions to return information about an array. These are: **`$left`**, **`$right`**, **`$low`**, **`$high`**, **`$increment`**, **`$length`**, and **`$dimensions`**. These functions are described in section 16.3.

Section 5 Data Declarations

5.1 Introduction (informative)

There are several forms of data in SystemVerilog: literals (see section 2), parameters (see section 14), constants, variables, nets, and attributes (see section 6)

C constants are either literals, macros or enumerations. There is also a **`const`**, keyword but it is not enforced in C.

Verilog 2001 constants are literals, parameters, localparams and specparams. Verilog 2001 also has variables and nets. Variables must be written by procedural statements, and nets must be written by continuous assignments or ports.

SystemVerilog follows Verilog by requiring data to be declared before it is used, apart from implicit nets. The rules for implicit nets are the same as in Verilog-2001.

A variable can be static (storage allocated on instantiation and never de-allocated) or automatic (stack storage allocated on entry to a task, function or named block and de-allocated on exit). C has the keywords **`static`** and **`auto`**. SystemVerilog follows Verilog in respect of the static default storage class, with automatic tasks and functions, but allows **`static`** to override a default of **`automatic`** for a particular variable in such tasks and functions.

5.2 Data declaration syntax

```
data_declaration ::=          // from Annex A.2.1.3
          variable_declaration
        | constant_declaration
        | type_declaration
block_variable_declaration ::=
          [ lifetime ] data_type  list_of_variable_identifiers ;
        | lifetime data_type  list_of_variable_decl_assignments ;
variable_declaration ::=
        [ lifetime ] data_type  list_of_variable_identifiers_or_assignments ;
lifetime ::= static | automatic
```

Syntax 5-1—Data declaration syntax (excerpt from Annex A)

5.3 Constants

Constants are named data items which never change. There are three kinds of constants, declared with the keywords **`localparam`**, **`specparam`** and **`const`**, respectively. All three can be initialized with a literal.

```
localparam char colon1 = ":" ;
specparam int delay = 10 ; // specparams are used for specify blocks
const logic flag = 1 ;
```

A local parameter is a constant which is calculated at elaboration time, and can depend upon parameters or other local parameters at the top level or in the same module or interface.

A specify parameter is also calculated at elaboration time, but it may be modified by the PLI, and so cannot be used to set parameters or local parameters.

A constant declared with the const keyword is calculated after elaboration. This means that it can contain an expression with any hierarchical path name. This constant is like a variable which cannot be written.

```
const logic option = a.b.c ;
```

A constant expression contains literals and other named constants.

SystemVerilog enhancements to **`parameter`** constant declarations are presented in section 14. SystemVerilog does not change **`localparam`** and **`specparam`** constants declarations. A **`const`** form of constant differs from a **`localparam`** constant in that the **`localparam`** must be set during elaboration, whereas a **`const`** can be set during simulation, such as in an automatic task.

5.4 Variables

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

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

A variable can be declared with an initializer, which must be a constant expression.

```
int i = 0;
```

In Verilog-2001, an initialization value specified as part of the declaration is executed as if the assignment were made from an initial block, after simulation has started. Therefore, the initialization may cause an event on that variable at simulation time zero.

In SystemVerilog, setting the initial value of a static variable as part of the variable declaration shall occur before any **`initial`** or **`always`** blocks are started, and so does not generate an event. If an event is needed, an **`initial`** block should be used to assign the initial values.

5.5 Scope and lifetime

Any data declared outside a module, interface, task, or function, is global in scope (can be used anywhere after its declaration) and has a static lifetime (exists for the whole elaboration and simulation time).

SystemVerilog data declared inside a module or interface but outside a task, process or function is local in scope and static in lifetime (exists for the lifetime of the module or interface). This is roughly equivalent to C static data declared outside a function, which is local to a file.

Data declared in an automatic task, function or block has the lifetime of the call or activation and a local scope. This is roughly equivalent to a C automatic variable. Data declared in a dynamic process is also automatic.

Data declared in a static task, function or block defaults to a static lifetime and a local scope. If an initializer is used, the keyword **`static`** must be specified to make the code clearer.

Note that in SystemVerilog, data can be declared in unnamed blocks as well as in named blocks, but in the unnamed blocks a hierarchical name cannot be used to access it.

Verilog-2001 allows tasks and functions to be declared as **`automatic`**, making all storage within the task or function automatic. SystemVerilog allows specific data within a static task or function to be explicitly declared as **`automatic`**. Data declared as automatic has the lifetime of the call or block, and is initialized on each entry to the call or block.

SystemVerilog also allows data to be explicitly declared as **`static`**. Data declared to be **`static`** in an automatic task, function or in a process has a static lifetime and a scope local to the block. This is like C static data declared within a function.

```
module msl;
   int st0; // static
   initial begin
      int st1; //static
      static int st2; //static
      automatic int auto1; //automatic
   end
   task automatic t1();
      int auto2; //automatic
      static int st3; //static
      automatic int auto3; //automatic
   endtask
endmodule
```

Note that automatic variables cannot be used to trigger an event expression or be written with a nonblocking assignment.

See also section 10 on tasks and functions.

5.6 Nets, regs, and logic

A net can only be written by one or more continuous assignments, primitive outputs or through module ports. The resultant value of multiple drivers is determined by the resolution function of the net type. The value can be overridden by a **force** statement. If a net on one side of a port is driven by a variable on the other side, a continuous assignment is implied.

A **reg** variable can only be written by one or more procedural statements, including procedural (quasi-) continuous assignments. The last write determines the value. The **force** statement overrides the **assign** statement which overrides the normal assignments. A **reg** variable cannot be written through a port.

A **logic** variable can be written either by one continuous assignment or primitive output, or by one or more procedural statements. The last write determines the value. A **logic** variable can be written through a port. It shall be an error to have a continuous assignment and a procedural assignment write to the same **logic** variable, even through ports. The **assign** statement overrides normal procedural assignments to a **logic** variable, until deassigned.

Note the difference between a net declaration with assignment and a variable initialization:

```
wire w = vara & varb; // continuous assignment
reg r = consta & constb; // initial assignment
logic v = consta & constb; // initial assignment
```

Section 6 Attributes

6.1 Introduction (informative)

With Verilog-2001, users can add named attributes (properties) to Verilog objects, such as modules, instances, wires, etc. Attributes can also be specified on SystemVerilog interfaces. SystemVerilog also defines a default data type for attributes.

6.2 Attribute syntax for interfaces

```
interface_declaration ::=         // from Annex A.1.3
            { attribute_instance } interface interface_identifier [ parameter_port_list ]
            [ list_of_ports ] ; [unit] [precision] { interface_item }
            endinterface [: interface_identifier]
          | { attribute_instance } interface interface_identifier [ parameter_port_list ]
            [ list_of_port_declarations ] ; [unit] [precision] { non_port_interface_item }
            endinterface [: interface_identifier]
interface_item ::=        // from Annex A.1.6
            port_declaration
          | non_port_interface_item
attribute_instance ::= (* attr_spec { , attr_spec } *)        // from Annex A.9.1
attr_spec ::=
            attr_name = constant_expression
          | attr_name
attr_name ::= identifier
```

Syntax 6-1—Interface attribute syntax (excerpt from Annex A)

An example of defining an attribute for an interface declaration is:

```
(* interface_att = 10 *) interface bus1.... endinterface
```

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

The **`modport`** declaration can be preceded by an attribute instance, like any other interface item.

Section 7
Operators and Expressions

7.1 Introduction (informative)

The SystemVerilog operators are a combination of Verilog and C operators. In both languages, the type and size of the operands is fixed, and hence the operator is of a fixed type and size. The fixed type and size of operators is preserved in SystemVerilog. This allows efficient code generation.

Verilog does not have assignment operators or incrementor and decrementor operators. SystemVerilog includes the C assignment operators, such as **+=**, and the C incrementor and decrementor operators, **++** and **--**.

Verilog-2001 added signed nets and **reg** variables, and signed based literals. There is a difference in the rules for combining signed and unsigned integers between Verilog and C. SystemVerilog uses the Verilog-2001 rules.

7.2 Operator syntax

```
unary_operator ::=          // from Annex A.8.6
            + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
binary_operator ::=
            + | - | * | / | % | == | != | === | !== | && | || | **
          | < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | >>> | <<<
inc_or_dec_operator ::= ++ | --
unary_module_path_operator ::=
      ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~
binary_module_path_operator ::=
      == | != | && | || | & | | | ^ | ^~ | ~^
assignment_operator ::=
            = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>= | <<<= | >>>=
```

Syntax 7-1—Operator syntax (excerpt from Annex A)

7.3 Assignment, incrementor and decrementor operations

In addition to the simple assignment operator, **=**, SystemVerilog includes the C assignment operators and special bitwise assignment operators: **+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **<<=**, **>>=**, **<<<=**, and **>>>=**. Assignment operators may only be used with blocking assignments.

In SystemVerilog, an expression can include a blocking assignment, provided it does not have a timing control. Note that such an assignment must be enclosed in parentheses to avoid common mistakes such as using `a=b` for `a==b`, or `a|=b` for `a!=b`.

```
if ((a=b)) b = (a+=1);

a = (b = (c = 5));
```

SystemVerilog also includes the C incrementor and decrementor operators **++i**, **--i**, **i++**, and **i--** (provided there is no timing control). These can be used in expressions without parentheses. These increment and decrement operations behave as blocking assignments.

7.4 Operations on logic and bit types

When a binary operator has one operand of type **bit** and another of type **logic**, the result is of type **logic**. If one operand is of type **int** and the other of type **integer**, the result is of type **integer**.

The operators **!=** and **==** return an X if either operand contains an **X** or a **Z**, as in Verilog-2001. This is converted to a 0 if the result is converted to type **bit**, e.g. in an **if** statement.

The unary reduction operators (**& ~& | ~| ^ ~^**) can be applied to any integer expression (including packed arrays). The operators shall return a single value of type **logic** if the packed type is four valued, and of type **bit** if the packed type is two valued.

```
    int i;
    bit b = &i;
    integer j;
    logic c = &j;
```

7.5 Real operators

Operands of type **shortreal** have the same operation restrictions as Verilog **real** operands. The unary operators ++ and -- can have operands of type **real** and **shortreal** (the increment or decrement is by 1.0). The assignment operators **+=**, **-=**, ***=**, **/=** can also have operands of type **real** and **shortreal**.

If any operand is **real**, the result is **real**, except before the ? in the ternary operator. If no operand is **real** and any operand is **shortreal**, the result is **shortreal**.

Real operands can also be used in the following expressions:

```
    str.realval // structure or union member
    realarray[intval] // array element
```

7.6 Size

The number of bits of an expression is determined by the operands and the context, following the same rules as Verilog. In SystemVerilog, casting can be used to set the size context of an intermediate value.

With Verilog, some tools may issue a warning when the left and right hand sides of an assignment are different sizes. Using the SystemVerilog size casting, these warnings can be prevented.

7.7 Sign

The following unary operators give the signedness of the operand: **~ ++ -- + -**. All other operators shall follow the same rules as Verilog for performing signed and unsigned operations.

7.8 Operator precedence and associativity

Operator precedence and associativity is listed in table 7-2, below. The highest precedence is listed first.

Table 7-2—Operator precedence and associativity

`() [] .`	left
`Unary ! ~ ++ -- + - & ~& && | ~| || ^ ~^`	right
`**`	left

Table 7-2—Operator precedence and associativity (continued)

`* / %`	left
`+ -`	left
`<< >> <<< >>>`	left
`< <= > >=`	left
`== != === !==`	left
`&`	left
`^ ~^`	left
`|`	left
`&&`	left
`||`	left
`?:`	right
`= += -= *= /= %= &= ^= |= <<= >>= <<<= >>>=`	none
`{,}`	concatenation

Note that **`&`** is higher precedence than `^`, following the Verilog standard.

7.9 Concatenation

Braces (**{ }**) are used to show concatenation, as in Verilog. The concatenation is treated as a packed vector of **`bits`** (or **`logic`** if any operand is of type **`logic`**). It can be used on the left hand side of an assignment or in an expression.

```
logic log1, log2, log3;
{log1, log2, log3} = 3'b111;
{log1, log2, log3} = {1'b1, 1'b1, 1'b1}; // same effect as 3'b111
```

Software tools may generate a warning if the concatenation width on one side of an assignment is different than the expression on the other side. The following examples can give warning of size mismatch:

```
bit [1:0] packedbits = {32'b1,32'b1}; // right hand side is 64 bits
int i = {1'b1, 1'b1}; //right hand side is 2 bits
```

Note that braces are also used for initializers of structures or unpacked arrays. Unlike in C, the expressions must match element for element and the braces must match the structures and array dimensions. Each element must match the type being initialized, so the following do not give size warnings:

```
bit unpackedbits [1:0] = {1,1}; // no size warning, bit can be set to 1
int unpackedints [1:0] = {1'b1,1'b1}; //no size warning, int can be set to 1'b1
```

A concatenation of unsized values shall be illegal, as in Verilog. However, an array initializer can use unsized values within the braces, such as {1,1}.

The replication operator (also called a multiple concatenation) form of braces can also be used for initializers . For example, {3{1}} represents the initializer {1, 1, 1}.

Refer to sections 2.7 and 2.8 for more information on initializing arrays and structures .

Section 8
Procedural Statements and Control Flow

8.1 Introduction (informative)

Procedural statements are introduced by one of:

`initial` // do this statement once

`always`, **`always_comb`**, **`always_latch`**, **`always_ff`** // loop forever (see section 9 on processes)

`task` // do these statements whenever the task is called

`function` // do these statements whenever the function is called and return a value

SystemVerilog has the following types of control flow within a process

— Selection, loops and jumps

— Task and function calls

— Sequential and parallel blocks

— Timing control

Verilog procedural statements are in **`initial`** or **`always`** blocks, tasks or functions.

Verilog includes most of the statement types of C, except for `do`...`while`, `break`, `continue` and `goto`. Verilog has the **`repeat`** statement which C does not, and the **`disable`**. The use of the Verilog **`disable`** to carry out the functionality of break and continue requires the user to invent block names, and introduces the opportunity for error.

SystemVerilog adds C-like **`break`**, **`continue`** and **`return`** functionality, which do not require the use of block names.

Loops with a test at the end are sometimes useful to save duplication of the loop body. SystemVerilog adds a C-like **`do`**...**`while`** loop for this capability.

Verilog provides two overlapping methods for procedurally adding and removing drivers for variables: the force/release statements and the **`assign`**/**`deassign`** statements. The **`force`**/**`release`** statements can also be used to add or remove drivers for nets in addition to variables. A force statement targeting a variable that is currently the target of an assign will override that assign; however, once the force is released, the assign's effect again will be visible.

The keyword **`assign`** is much more commonly used for the somewhat similar, yet quite different purpose of defining permanent drivers of values to nets.

```
statement ::= [ block_identifier : ] statement_item          // from Annex A.6.4

statement_item ::=
          { attribute_instance } blocking_assignment ;
        | { attribute_instance } nonblocking_assignment ;
        | { attribute_instance } procedural_continuous_assignments ;
        | { attribute_instance } case_statement
        | { attribute_instance } conditional_statement
        | { attribute_instance } transition_to_state statement_or_null
        | { attribute_instance } inc_or_dec_expression
        | { attribute_instance } function_call        /* must be void function */
        | { attribute_instance } disable_statement
        | { attribute_instance } event_trigger
        | { attribute_instance } loop_statement
        | { attribute_instance } jump_statement
        | { attribute_instance } par_block
        | { attribute_instance } procedural_timing_control_statement
        | { attribute_instance } seq_block
        | { attribute_instance } system_task_enable
        | { attribute_instance } task_enable
        | { attribute_instance } wait_statement
        | { attribute_instance } process statement
        | { attribute_instance } proc_assertion

statement_or_null ::=
          statement
        | { attribute_instance } ;

procedural_timing_control_statement ::=
          delay_or_event_control  statement_or_null
```

Syntax 8-1—statement syntax (excerpt from Annex A)

8.2 Blocking and nonblocking assignments

```
blocking_assignment ::=          // from Annex A.6.2
          variable_lvalue = delay_or_event_control expression
        | operator_assignment

operator_assignment ::= variable_lvalue  assignment_operator  expression

assignment_operator ::=
          = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>= | <<<= | >>>=

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

Syntax 8-2—blocking and nonblocking assignment syntax (excerpt from Annex A)

The following assignments are allowed in both Verilog-2001 and SystemVerilog:

```
#1 r = a;
r = #1 a;
r <= #1 a;
r <= a;
```

```
@c r = a;
r = @c a;
r <= @c a;
```

SystemVerilog also allows a time unit to specified in the assignment statement, as follows:

```
#1ns r = a;
r = #1ns a;
r <= #1ns a;
```

It shall be illegal to make nonblocking assignments to automatic variables.

The size of the left-hand side of an assignment forms the context for the right hand side expression. If the left-hand side is smaller than the right hand side, and information may be lost, a warning can be given.

8.3 Selection statements

conditional_statement ::= // from Annex A.6.6
  [unique_priority] **if (** expression **)** statement_or_null [**else** statement_or_null]
  | if_else_if_statement

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

case_statement ::= // from Annex A.6.7
  [unique_priority] **case (** expression **)** case_item { case_item } **endcase**
  | [unique_priority] **casez (** expression **)** case_item { case_item } **endcase**
  | [unique_priority] **casex (** expression **)** case_item { case_item } **endcase**

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

unique_priority ::= **unique** | **priority**

Syntax 8-3—Selection statement syntax (excerpt from Annex A)

In Verilog, an `if (`expression`)` is evaluated as a boolean, so that if the result of the expression is 0 or X, the test is considered false.

SystemVerilog adds the keywords **`unique`** and **`priority`**, which can be used before an **`if`**. If either keyword is used, it shall be a run-time warning for no condition to match unless there is an explicit **`else`**. For example:

```
unique if((a==0) || (a==1)) $display("0 or 1");
else if (a == 2) $display("2");
else if (a == 4) $display("4"); // values 3,5,6,7 will cause a warning

priority if(a[2:1]==0) $display("0 or 1");
else if (a[2] == 0) $display("2 or 3");
else $display("4 to 7"); //covers all other possible values, so no warning
```

A **`unique if`** indicates that there should not be any overlap in a series of **`if`**...**`else`**...**`if`** conditions, allowing the expressions to be evaluated in parallel. A software tool shall issue an error if it determines that there is a potential overlap in the conditions.

A **`priority if`** indicates that a series of `if`...**`else`**...**`if`** conditions shall be evaluated in the order listed. In the preceding example, if the variable ‘`a`’ had a value of 0, it would satisfy both the first and second conditions, requiring priority logic.

In Verilog, there are three types of case statements, introduced by **`case`**, **`casez`** and **`casex`**. With SystemVerilog, each of these can be qualified by **`priority`** or **`unique`**. A **`priority case`** shall act on the first match only. A **`unique case`** shall guarantee no overlapping case values, allowing the case items to be evaluated in parallel. If the case is qualified as **`priority`** or **`unique`**, the simulator shall issue a warning message if an unexpected case value is found. By specifying **`unique`** or **`priority`**, it is not necessary to code a **`default`** case to trap unexpected case values. For example:

```
bit[2:0] a;
unique case(a) // values 3,5,6,7 will cause a run-time warning
      0,1: $display("0 or 1");
      2: $display("2");
      4: $display("4");
endcase

priority casez(a)
      2’b00?: $display("0 or 1");
      2’b0??: $display("2 or 3");
      default: $display("4 to 7");
endcase
```

The **`unique`** and **`priority`** keywords shall determine the simulation behavior. It is recommended that synthesis follow simulation behavior where possible. Attributes may also be used to determine synthesis behavior.

8.4 Loop statements

```
loop_statement ::=          // from Annex A.6.8
          forever statement
        | repeat ( expression ) statement_or_null
        | while ( expression ) statement_or_null
        | for ( variable_decl_or_assignment ; expression ; variable_assignment ) statement_or_null
        | do statement while ( expression )
variable_decl_or_assignment ::=
          data_type  list_of_variable_identifiers_or_assignments ;
        | variable_assignment
```

Syntax 8-4—Loop statement syntax (excerpt from Annex A)

Verilog provides **`for`**, **`while`**, **`repeat`** and **`forever`** loops. SystemVerilog adds a **`do`**...**`while`** loop.

```
do statement while(condition) // as C
```

The condition can be any expression which can be treated as a boolean. It is evaluated after the statement.

In Verilog, the variable used to control a **`for`** loop must be declared prior to the loop. If loops in two or more parallel procedures use the same loop control variable, there is a potential of one loop modifying the variable while other loops are still using it.

SystemVerilog adds the ability to declare the **`for`** loop control variable within the **`for`** loop. This creates a local variable within the loop. Other parallel loops cannot inadvertently affect the loop control variable. For example:

```
module foo;

   initial begin
      for (int i = 0; i <= 255; i++)
         ...
   end

   initial begin
      loop2: for (int i = 15; i >= 0; i--)
         ...
   end
endmodule
```

The local variable declared within a **for** loop can be referenced hierarchically by adding a statement label before the **for** loop (see section 8.6).

8.5 Jump statements

jump_statement ::=   // from Annex A.6.5
    return [expression] **;**
  | **break ;**
  | **continue ;**

Syntax 8-5—Jump statement syntax (excerpt from Annex A)

SystemVerilog adds the C jump statements **break**, **continue** and **return**.

```
break    // out of loop as C
continue // skip to end of loop as C
return expression    // exit from a function
return   // exit from a task or void function
```

The **continue** and **break** statements can only be used in a loop. The **continue** statement jumps to the end of the loop and executes the loop control if present. The **break** statement jumps out of the loop.

The **return** statement can only be used in a task or function. In a function returning a value, the return must have an expression of the correct type.

Note that SystemVerilog does not include the C **goto** statement.

8.6 Named blocks and statement labels

par_block ::=   // from Annex A.6.3
    fork [**:** block_identifier] { block_item_declaration } { statement } **join** [**:** block_identifier]

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

statement ::= [block_identifier **:**] statement_item

Syntax 8-6—Blocks and labels syntax (excerpt from Annex A)

Verilog allows a **begin**...**end** or **fork**...**join** statement block to be named. A named block is used to identify the entire statement block. A named block creates a new hierarchy scope. The block name is specified after the **begin** or **fork** keyword, preceded by a colon. For example:

```
begin : blockA    // Verilog-2001 named block
   ...
end
```

SystemVerilog allows a matching block name to be specified after the block **end** or **join** keyword, preceded by a colon. This can help document which **end** or **join** is associated with which **begin** or **fork** when there are nested blocks. A name at the end of the block is not required. It shall be an error if the name at the end is different than the block name at the beginning.

```
begin: blockB     // block name after the begin or fork
   ...
end: blockB
```

SystemVerilog allows a label to be specified before any statement, as in C. A statement label is used to identify a single statement. A statement label does not create a hierarchy scope. The label name is specified before the statement, followed by a colon.

```
labelA: statement
```

A **begin**...**end** or **fork**...**join** block is considered a statement, and can have a statement label before the block. This is not the same as a block name, however, because it does not create a hierarchy scope.

```
labelB: fork   // label before the begin or fork
   ...
join : labelB
```

It shall be illegal to have both a label before a **begin** or **fork** and a block name after the **begin** or **fork**. A label cannot appear before the **end** or **join**, as these keywords do not form a statement.

A statement with a label can be disabled using a **disable** statement. Disabling a statement shall have the same behavior as disabling a named block.

8.7 Processes

Each **initial** and **always** block is a process. Each branch of a **fork** within such a block is also a process. These are static processes and they can be explicitly named with a statement label as shown above.

A dynamic process can be created using the **process** keyword. This forks off a statement without waiting for completion.

```
process statement
```

See Section 9 for more information about processes.

8.8 Disable

SystemVerilog has **break** and **continue** for a clean way to break out of or continue the execution of loops. The Verilog-2001 disable can also be used to break out of or continue a loop, but is more awkward than using **break** or **continue**. The **disable** is also allowed to disable a named block, which does not contain the **disable** statement. If the block is currently executing, this causes control to jump to the statement immediately after the block. If the block is a loop body, it acts like a **continue**. If the block is not currently executing, the **disable** has no effect. The **disable**, **break** and **continue** statements shall not affect any nonblocking assignments which have been started.

It shall be illegal to disable a function, because the return value would be uncertain. However, a function may disable its calling block.

SystemVerilog has **`return`** from a task, but **`disable`** is also supported. If **`disable`** is applied to a named task, all current executions of the task are disabled.

```
module ...
always always1: begin ... t1: task1( ); ... end
...
endmodule

always begin
   ...
   disable u1.always1.t1; // exit task1, which was called from always1 (static)
end
```

8.9 Event control

```
delay_or_event_control ::=          // from Annex A.6.5
        delay_control
      | event_control
      | repeat ( expression ) event_control
delay_control ::=
        # delay_value
      | # ( mintypmax_expression )
event_control ::=
        @ event_identifier
      | @ ( event_expression )
      | @*
      | @ (*)
event_expression ::=
        expression [ iff expression ]
      | hierarchical_identifier [ iff expression ]
      | [ edge ] expression [ iff expression ]
      | event_expression or event_expression
      | event_expression , event_expression
edge ::= posedge | negedge | changed
```

Syntax 8-7—Delay and event control syntax (excerpt from Annex A)

Any change in a variable or net can be detected using the **@** event control, as in Verilog. If the expression evaluates to a result of more than one bit, a change on any of the bits of the result (including an x to z change) will trigger the event control.

SystemVerilog adds an **`iff`** qualifier to the **`@`** event control.

```
module latch (output logic [31:0] y, input [31:0] a, input enable);
   always @(a iff enable == 1)
      y <= a; //latch is in transparent mode
endmodule
```

The event expression only triggers if the expression after the **`iff`** is true, in this case when `enable` is equal to 1. Note that such an expression is evaluated when `clk` changes, and not when `enable` changes. Also note that **`iff`** has precedence over **`or`**. This can be made clearer by the use of parentheses.

If a variable or net is not of type **`logic`**, then **`posedge`** and **`negedge`** refer to transitions from 0 and to 0 respectively. If the variable or net is a packed array or structure, it is zero if all elements are 0.

SystemVerilog also allows the **@** event control to explicitly state any change, using the **`changed`** keyword.

```
@(myvar)          // triggers on any change to myvar

@(changed myvar)  // triggers on any change to myvar
```

The **`@(changed`** expression**`)`** differs from **`@(`**expression**`)`** in that the **`changed`** keyword explicitly defines that the event control only triggers on a change of the result of the expression. In certain types of expressions, **`@(`**expression**`)`** can trigger on changes to operands of the expression that do not affect the result.

SystemVerilog allows assignment expressions to be used in an event control, e.g. `@((a = b + c))`. The event control shall only be sensitive to changes in the result of the expression on the right-hand side of the assignment. It shall not be sensitive to changes on the left-hand side expression.

8.10 Procedural assign and deassign removal

SystemVerilog currently supports the procedural **`assign`** and **`deassign`** statements. However, these statements may be removed from future versions of the language. See section 18.3.

Section 9
Processes

9.1 Introduction (informative)

Verilog-2001 has **always** and **initial** blocks which define static processes.

In an **always** block which is used to model combinational logic, forgetting an **else** leads to an unintended latch. To avoid this mistake, SystemVerilog adds specialized **always_comb** and **always_latch** blocks, which indicate design intent to simulation, synthesis and formal verification tools. SystemVerilog also adds an always_ff block to indicate sequential logic.

In systems modeling, one of the key limitations of Verilog is the inability to create processes dynamically, as happens in an operating system. Verilog has the **fork** .. **join** construct, but this still imposes a static limit.

SystemVerilog has both static processes, introduced by **always**, **initial** or **fork**, and dynamic processes introduced by **process**.

SystemVerilog creates a thread of execution for each **initial** or **always** block, for each parallel statement in a **fork**...**join** block and for each dynamic process. Each continuous assignment may also be considered its own thread. Execution of each thread may be interrupted between statements at a semicolon, but a single statement (not a block) containing no user task or function call is uninterrupted. This allows atomic test-and-set using assignment operators in an if statement.

9.2 Level sensitive logic

SystemVerilog provides a special **always_comb** procedure for modeling combinational logic behavior. For example:

```
always_comb
   a = b & c;

always_comb
   d <= #1ns b & c;
```

The **always_comb** procedure provides functionality that is different than a normal always procedure:

— There is an inferred sensitivity list that includes every variable read by the procedure.

— The variables written on the left-hand side of assignments may not be written to by any other process.

— The procedure is automatically triggered once at time zero, after all **initial** and **always** blocks have been started, so that the outputs of the procedure are consistent with the inputs.

The SystemVerilog **always_comb** procedure differs from the Verilog-2001 **always @*** in the following ways:

— **always_comb** automatically executes once at time zero, whereas **always @*** waits until a change occurs on a signal in the inferred sensitivity list.

— **always_comb** is sensitive to changes within the contents of a function, whereas **always @*** is only sensitive to changes to the arguments of a function.

— Variables on the left-hand side of assignments within an **always_comb** procedure may not be written to by any other processes, whereas **always @*** permits multiple processes to write to the same variable.

Software tools can perform additional checks to warn if the behavior within an **always_comb** procedure does not represent combinational logic, such as if latched behavior may be inferred.

9.3 Latch sensitive logic

SystemVerilog also provides a special **`always_latch`** procedure for modeling latched logic behavior. For example:

```
always_latch
   if(ck) q <= d;
```

The **`always_latch`** procedure differs from a normal **`always`** procedure in the following ways:

— There is an inferred sensitivity list that includes every variable read by the procedure.

— The variables written on the left-hand side of assignments may not be written to by any other process.

— The procedure is automatically triggered once at time zero, after all **`initial`** and **`always`** blocks have been started, so that the outputs of the procedure are consistent with the inputs.

Software tools may perform additional checks to warn if the behavior within an always_latch procedure does not represent latched logic.

9.4 Edge sensitive logic

The SystemVerilog **`always_ff`** procedure can be used to model synthesizable sequential logic behavior. For example:

```
always_ff @(posedge clock iff reset == 0 or posedge reset) begin
   r1 <= reset ? 0 : r2 + 1;
   ...
end
```

The **`always_ff`** block imposes the restriction that only one event control is allowed. Software tools may perform additional checks to warn if the behavior within an **`always_ff`** procedure does not represent sequential logic.

9.5 Continuous assignments

In Verilog, continuous assignments can only drive nets, and not variables.

SystemVerilog removes this restriction, and permits continuous assignments to drive nets, **`logic`** variables, and any other type of variables, except **`reg`** variables. Nets can be driven by multiple continuous assignments, or a mixture of primitives and continuous assignments. **`logic`** variables and other data types can only be driven by one continuous assignment or one primitive output. It shall be an error for a variable driven by a continuous assignment or primitive output to have an initializer in the declaration or any procedural assignment.

9.6 Dynamic processes

The SystemVerilog dynamic process adds capability that behaves like a **`fork`** without a **`join`**. A dynamic process is started as a separate thread, and execution of the current procedure or task continues while the process is executing. The process does not block the flow of execution of statements within the procedure or task. Dynamic processes allow the creation of multi-threaded processes, as opposed to multiple procedures, which are static parallel processes.

A dynamic process shall be created by the **`process`** keyword, which is used as follows:

```
process statement
```

For example, the following task initiates an endless loop and returns immediately to the caller. The task can be launched any number of times to display a selected location at every strobe.

```
task monitorMem(input int address);
   process forever @strobe $display("address %h data %h", mem[address] );
endtask
```

The following example illustrates using a dynamic process to model a pipeline.

```
// pipeline module
module p(input clk, flush, input int x_in, y_in, z_in);
   parameter int latency = 6, throughput = 2;
   int z_out;
   int processes = 0;

   always begin
      while (!flush) begin
      process begin
         int v2, v3, v4, v5; // lifetime matches process
         processes++;
         v2 = x_in + y_in;
         v3 = x_in - z_in;
         v4 = v2 * v3;
         v5 = v4 * x_in;
         repeat(latency) @ (posedge clk);
         z_out <= v5;
         processes--;
      end
      repeat(throughput) @(posedge clk);
   end
   wait(processes == 0); //wait for flush
   end
endmodule
```

In the proceeding example, the **`while`** loop contains a delay of two clock cycles, from the **`repeat`** statement, and this determines the pipeline throughput. Each iteration spawns a process which lasts six clock cycles, the latency of the pipeline. The variable `processes` keeps a count of the number of currently active processes. The pipeline flush is not complete until this count has fallen to zero.

SystemVerilog 3.0 does not provide a mechanism to disable a process once it has been started, but all instances of a named block within a dynamic process can be disabled by referring to a named block.

9.7 Process execution threads

SystemVerilog creates a thread of execution for:

— Each **`initial`** block

— Each **`always`** block

— Each parallel statement in a **`fork`**...**`join`** statement group

— Each dynamic process

Each continuous assignment may also be considered its own thread.

Execution of each thread can be interrupted between statements at a semicolon, but a single statement (not a block) containing no user task or function call shall not be interrupted. This allows atomic test-and-set using assignment operators in an **`if`** statement.

Section 10 Tasks and Functions

10.1 Introduction (informative)

Verilog-2001 has static and automatic tasks and functions. Static tasks and functions share the same storage space for all calls to the tasks or function within a module instance. Automatic tasks and function allocate unique, stacked storage for each instance.

SystemVerilog adds the ability to declare automatic variables within static tasks and functions, and static variables within automatic tasks and functions.

SystemVerilog also adds:

— More capabilities for declaring task and function ports

— Function output and inout ports

— Void functions

— Multiple statements in a task or function without requiring a **`begin`**...**`end`** or **`fork`**...**`join`** block

— Returning from a task or function before reaching the end of the task or function

10.2 Tasks

```
task_declaration ::=          // from Annex A.2.7
          task [ automatic ] [ interface_identifier . ] task_identifier ;
          { task_item_declaration }
          { statement }
          endtask [ : task_identifier ]
        | task [ automatic ] [ interface_identifier . ] task_identifier ( task_port_list ) ;
          { block_item_declaration }
          { statement }
          endtask [ : task_identifier ]

task_item_declaration ::=
          block_item_declaration
        | { attribute_instance } input_declaration ;
        | { attribute_instance } output_declaration ;
        | { attribute_instance } inout_declaration ;

task_port_list ::= task_port_item { , task_port_item }

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

task_prototype ::=
      task ( { attribute_instance } task_proto_formal
          { , { attribute_instance } task_proto_formal } )

named_task_proto ::= task task_identifier ( task_proto_formal { , task_proto_formal } )

task_proto_formal ::=
          input data_type [ variable_declaration_identifier ]
        | inout data_type [ variable_declaration_identifier ]
        | output data_type [ variable_declaration_identifier ]
```

Syntax 10-1—Task syntax (excerpt from Annex A)

A Verilog task declaration either has the formal arguments in parentheses (like ANSI C) or in declarations and directions.

```
task mytask1 (output int x, input logic y);
   ...
endtask

task mytask2;
   output x;
   input y;
   int x;
   logic y;
   ...
endtask
```

Each formal argument has one of the following directions:

```
input   // copy value in at beginning
output  // copy value out at end
inout   // copy in at beginning and out at end
```

With SystemVerilog, there is a default direction of **`input`** if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments **`a`** and **`b`** default to inputs, and **`u`** and **`v`** are both outputs.

```
task mytask3(a, b, output logic [15:0] u, v);
   ...
endtask
```

Each formal argument also has a data type which can be explicitly declared or can inherit a default type. The default type in SystemVerilog is **`logic`**, which is compatible with Verilog. SystemVerilog allows an array to be specified as a formal argument to a task. For example:

```
// the resultant declaration of b is input [3:0][7:0] b[3:0]
task mytask4(input [3:0][7:0] a, b[3:0], output [3:0][7:0] y[1:0]);
   ...
endtask
```

Verilog-2001 allows tasks to be declared as **`automatic`**, so that all formal arguments and local variables are stored on the stack. SystemVerilog extends this capability by allowing specific formal arguments and local variables to be declared as **`automatic`** within a static task, or by declaring specific formal arguments and local variables as **`static`** within an automatic task.

With SystemVerilog, multiple statements can be written between the task declaration and **`endtask`**, which means that the **`begin`** .... **`end`** can be omitted. If **`begin`** .... **`end`** is omitted, statements are executed sequentially, the same as if they were enclosed in a **`begin`** .... **`end`** group. It shall also be legal to have no statements at all.

In Verilog, a task exits when the endtask is reached. With SystemVerilog, the **`return`** statement can be used to exit the task before the **`endtask`** keyword.

10.3 Functions

function_declaration ::= // from Annex A.2.6
  function [**automatic**] [signing] [range_or_type]
  [interface_identifier **.**] function_identifier **;**
  { function_item_declaration }
  { function_statement }
  endfunction [**:** function_identifier]
  | **function** [**automatic**] [signing] [range_or_type]
  [interface_identifier **.**] function_identifier **(** function_port_list **) ;**
  { block_item_declaration }
  { function_statement }
  endfunction [**:** function_identifier]

function_item_declaration ::=
  block_item_declaration
  | { attribute_instance } input_declaration **;**
  | { attribute_instance } output_declaration **;**
  | { attribute_instance } inout_declaration **;**

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

function_port_list ::= function_port_item { **,** function_port_item }

function_prototype ::= **function** data_type **(** list_of_function_proto_formals **)**

named_function_proto::= **function** data_type function_identifier **(** list_of_function_proto_formals **)**

list_of_function_proto_formals ::=
  [{ attribute_instance } function_proto_formal { **,** { attribute_instance }
  function_proto_formal }]

function_proto_formal ::=
  input data_type [variable_declaration_identifier]
  | **inout** data_type [variable_declaration_identifier]
  | **output** data_type [variable_declaration_identifier]
  | variable_declaration_identifier

range_or_type ::=
  { packed_dimension } range
  | data_type

Syntax 10-2—Function syntax (excerpt from Annex A)

A Verilog function declaration either has the formal arguments in parentheses (like ANSI C) or in declarations and directions:

```
function logic [15:0] myfunc1(int x, int y);
   ...
endfunction

function logic [15:0] myfunc2;
   input int x;
   input int y;
   ...
endfunction
```

SystemVerilog extends Verilog functions to allow the formal arguments to be inputs or outputs. Function arguments are all passed by value, i.e. copied.

```
input   // copy value in at beginning

output  // copy value out at end

inout   // copy in at beginning and out at end
```

Function declarations default to the formal direction **`input`** if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments **`a`** and **`b`** default to inputs, and **`u`** and **`v`** are both outputs:

```
function logic [15:0] myfunc3(int a, int b, output logic [15:0] u, v);
   ...
endfunction
```

Each formal argument has a data type which can be explicitly declared or can inherit a default type. The default type in SystemVerilog is **`logic`**, which is compatible with Verilog. SystemVerilog allows an array to be specified as a formal argument to a function, for example:

```
function [3:0][7:0] myfunc4(input [3:0][7:0] a, b[3:0]);
   ...
endfunction
```

In Verilog, functions must return values. The return value is specified by assigning a value to the name of the function.

```
function [15:0] myfunc1 (input foo);
   myfunc1 = 16'hbeef; //return value is assigned to function name
endfunction
```

SystemVerilog allows functions to be declared as type **`void`**, which do not have a return value. For non-void functions, a value can be returned by assigning the function name to a value, as in Verilog, or by using **`return`** with a value. The **`return`** statement shall override any value assigned to the function name. When the return statement is used, non-void functions must specify an expression with the return.

```
function [15:0] myfunc2 (input foo);
   return 16'hbeef; //return value is specified using return statement
endfunction
```

In SystemVerilog, a function return can be a structure or union. In this case, a hierarchical name used inside the function and beginning with the function name is interpreted as a member of the return value. If the function name is used outside the function, the name indicates the scope of the whole function. If the function name is used within a hierarchical name, it also indicates the scope of the whole function.

Function calls are expressions unless of type **`void`**, which are statements:

```
a = b + myfunc1(c, d); //call myfunc1 (defined above) as an expression

myprint(a); //call myprint (defined below) as a statement

function void myprint (int a);
   ...
endfunction
```

With SystemVerilog, a non-void function call can also be used as a statement, but this can result in a warning message.

SystemVerilog allows multiple statements to be written between the function header and **`endfunction`**, which means that the **`begin`**...`end` can be omitted. If **`begin`**...**`end`** is omitted, statements are executed sequentially, as if they were enclosed in a **`begin`**...**`end`** group. It is also legal to have no statements at all, in which case the function returns the current value of the implicit variable that has the same name as the function.

Section 11 Assertions

11.1 Introduction (informative)

An assertion is a statement that a property must be true. There are two kinds of assertions: concurrent assertions which state that the property must be always be true, e.g. throughout a simulation, and procedural assertions which are incorporated in procedural code and apply only for a limited time or under limited conditions.

There are various applications of assertions. They can be included in the design, to document the assumptions made by the designer and to facilitate “white box” testing. They can be outside the design, either in a testbench to check the response of the design to the stimulus, or to control a tool such as a stimulus generator or a model checker.

Concurrent assertions can be coded as modules in a library, but this limits the complexity of the property that can be expressed easily. It is more difficult to code procedural assertions as a library of tasks in Verilog, because events cannot be arguments, each assertion may need static data, and tasks block.

11.2 Procedural assertions

```
proc_assertion ::=          // from Annex A.6.10
            immediate_assert
          | strobed_assert
          | clocked_immediate_assert
          | clocked_strobed_assert
immediate_assert ::= assert ( expression )
            statement_or_null
            [ else statement_or_null ]
strobed_assert ::= assert_strobe ( expression )
            restricted_statement_or_null
            [ else restricted_statement_or_null ]
clocked_immediate_assert ::= assert ( expr_sequence ) step_control
            statement_or_null
            [ else statement_or_null ]
clocked_strobed_assert ::= assert_strobe ( expr_sequence ) step_control
            restricted_statement_or_null
            [ else restricted_statement_or_null ]
restricted_statement_or_null ::=
            restricted_statement
          | { attribute_instance } ;
restricted_statement ::=
            [ block_identifier : ] restricted_statement_item
restricted_statement_item ::=
            { attribute_instance } proc_assertion
          | { attribute_instance } system_task_enable
          | { attribute_instance } delay_or_event_control statement
          | { attribute_instance } restricted_seq_block
restricted_seq_block ::= begin [ : block_identifier ] { block_item_declaration }{ restricted_statement }
            end [ : block_identifier ]
expr_sequence ::=
            expression
          | [ constant_expression ]
          | range
          | expr_sequence ; expr_sequence
          | expr_sequence * [ constant_expression ]
          | expr_sequence * range
          | ( expr_sequence )
step_control ::=
            @@ event_identifier
          | @@ ( event_expression )
```

Syntax 11-1—Assertion syntax (excerpt from Annex A)

SystemVerilog provides four kinds of procedural assertions, which allow the user to test boolean expressions or sequences of boolean expressions, and perform some action based on whether the expression or sequence is true or false. Immediate assertions test the value of a boolean expression at the time the statement is executed, and may be used in always and initial blocks, tasks and functions. Strobed assertions schedule the evaluation of the expression to be delayed until the end of the current timeslice, to allow for glitches to settle. Strobed assertions may be used in **`initial`** and **`always`** blocks and tasks, but not in functions, since functions must return immediately.

To test sequences of expressions, it is necessary to specify a sampling clock event on which to test each element of the sequence. Therefore, Clocked Immediate and Clocked Strobed assertions are added to allow progressive evaluation of sequences of expressions. Since these clocked assertions, by definition, take time, they cannot be used in functions. Clocked immediate assertions evaluate each expression in the sequence when the clock event triggers, and clocked strobed assertions evaluate each expression at the end of the timeslice at which the event triggers.

11.3 Immediate assertions

The immediate assert statement is a test of an expression performed when the statement is executed in the procedural code. The expression is treated as a condition like in an if statement.

[identifier **:**] **assert (** expression **)** [pass_statement] [**else** fail_statement]

The pass statement is executed if the assertion succeeds, i.e. the expression evaluates to true. As with the if statement, if the expression evaluates to `'X'`, `'Z'` or `'0'`, then the assertion fails. The pass statement may, for example, record the number of successes for a coverage log, but may be omitted altogether. If the pass statement is omitted, then no action is taken if the assert expression is true. The fail statement is executed if the assertion fails (i.e. the expression does not evaluate to a known, non-zero value) and can be omitted. The optional assertion label (identifier and colon) creates a notional named block around the assertion statement (or any other SystemVerilog statement) and can be displayed using the `%m` format code.

```
assert_foo : assert (foo) $display("%m passed"); else $display("%m failed");
```

Since the assertion is a statement that something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is “error”. Other severity levels may be specified by including one of the following severity system tasks in the fail statement.

— **`$fatal`** is a run-time Fatal, which terminates the simulation with an error code. The first argument passed to $fatal shall be consistent with the argument to $finish.

— **`$error`** is a Run-time Error.

— **`$warning`** is a Run-time Warning, which can be suppressed in a tool-specific manner.

— **`$info`** indicates that the assertion failure carries no specific severity.

The syntax for these system tasks is shown in section 16.4.

All of these severity system tasks shall print a tool-specific message indicating the severity of the failure, and specific information about the specific failure, which shall include the following information:

— The file name and line number of the assertion statement,

— The hierarchical name of the assertion, if it is labeled, or the scope of the assertion if it is not labeled.

For simulation tools, these tasks shall also include the simulation run-time at which the severity system task is called.

Each system task can also include additional user-specified information using the same format as the Verilog **`$display`**.

If more than one of these system tasks is included in the **`else`** clause, then each shall be executed as specified.

If an assertion fails and no **`else`** clause is specified, the tool shall, by default, call **`$error`**, unless a tool-specific command-line option is enabled to suppress the failure.

If the severity system task is executed at a time other than when the assertion fails, the actual failure time of the assertion can be recorded and displayed programmatically. For example:

```
time t;

always @(posedge clk)
   if(state == REQ)
      assert(req1 || req2)
      else begin
         t = $time;
         #5 $error("assert failed at time %0t",t);
      end
```

If the assertion fails at time 10, the error message will be printed at time 15, but the user-defined string printed will be “assert failed at time 10”.

The display of messages of warning and info types can be controlled by a tool-specific command-line option.

Since the fail statement, like the pass statement, is any legal SystemVerilog procedural statement, it can also be used to signal a failure to another part of the testbench.

```
assert (myfunc(a,b)) count1 = count + 1; else ->event1;
assert (y == 0); else flag = 1;
```

The assert statement serves as guidance to non-simulation tools that the condition should be true. The second statement above is equivalent to:

```
if ( y!=0) begin flag = 1; end
```

11.4 Strobed assertions

If an immediate assertion is in code triggered by a timing control that happens at the same time as a blocking assignment to the data being tested, there is a risk of the wrong value being sampled. For example:

```
always @(posedge clock) a = a + 1; // blocking assignment
always @(posedge clock) begin
   ....
   assert (a < b);
end
```

This can be solved by using a strobed assertion, which waits in the background until the end of the time slot, like the Verilog **$strobe** system task.

```
always @(posedge clock) begin
   ....
   cas:assert_strobe (a < b);
end
```

Strobed assertions can have pass or fail statements like immediate assertions. However, the statements are restricted to another assertion statement, a system task call, a statement preceded by a delay control or an event control, or sequential block containing them. This is because the statement happens after the assertion is evaluated, at the end of the time slot, and hence must not create more events at that time slot or change values. Statements which cause additional events to occur at the current time shall be an error.

The example below illustrates the effect of blocking and nonblocking assignments on immediate and strobed assertions. The immediate assertions are like **$display** statements and the strobed assertions are like **$strobe** statements.

```
module test;
reg [3:0] a=0; c=0, d=0;
reg clk = 0;
wire b;

initial begin
   #10 clk = 1;
   forever #5 clk = !clk; // posedge clk at 10,20,30,40...
end

assign b = a+1;

always @(posedge clk) begin
   a1: assert(c<3); // fails at time 40
   c = c+1;
   a2: assert(c<3); // fails at time 30
   a <= a+1;
   a3: assert(a<3); // fails at time 40
   a4: assert(b<3); // fails at time 40
   a5: assert_strobe(a<3); // fails at time 30
   a6: assert_strobe(b<3); // fails at time 30
end

always @(a) begin // models transient behavior on comb. nets
   d = a+2; // spikes to 2 at 0, 3 at 10, 4 at 20
   assert(d<3); // fails at time 10
   d = d-1; // settles to 1 at 0, 2 at 10, 3 at 20
   assert(d<3); // fails at time 20
end

always @(d) assert_strobe (d<3); // fails at time 20

endmodule
```

11.5 Sequential assertions

In addition to assertions about single expressions, it is often useful to assert sequences of expressions over time. One way of doing this is to use nested immediate assertions, where each subsequent assertion is the pass statement of the previous assertion.

```
always @(posedge clk or negedge rst)
   if(state == REQ)
      a7: assert(req1) // no semicolon
      @(posedge clk) assert(gnt)
      @(posedge clk) assert(!req1);
```

The above example verifies the sequence that, if state is equal to `REQ`, the `req1` signal must be true immediately, then on the next posedge `clk`, `gnt` must be true and on the following posedge `clk`, `req` must be false. Note that the assertion statement itself is nonblocking, so the sequence in assertion `a7` is equivalent to:

```
always @(posedge clk or negedge rst)
   if(state == REQ)
   a8: assert(req1)
   process
      @(posedge clk) assert (gnt)
         @(posedge clk) assert(!req1);
```

To simplify this complex nested assertion, a *sequential regular expression* is used in the assert statement.

Sequential regular expressions require a *step control event expression* to specify the timing between evaluations of each element in the regular expression. Using a sequential regular expression, the assertion `a8` could be rewritten as:

```
always @(posedge clk or negedge rst)
if(state == REQ)
   a9: assert(req1;gnt;!req1) @@(posedge clk);
   // note the @@ token to distinguish the step control from the pass statement
```

A sequential regular expression is a semicolon-delimited list of expressions. The first expression in the list is evaluated immediately when the assert statement is executed. The other subsequent expressions are evaluated one at a time on successive occurrences of the step control event expression. In assertion `a9` above, `req1` is evaluated immediately when the assert statement is executed, just as for an immediate assertion, then `gnt` is evaluated on the next posedge `clk` event, and so on.

The '**@@**' token is introduced to distinguish the step control from an ordinary event control at the start of the pass statement. Consider the following:

```
always @(posedge clock or negedge rst)
   if(state == REQ)
      a10: assert (req1)
      @(posedge clk) // This is an event control in the pass statement
         $display("Hello at time %t", $time);
```

In this example, the "`@(posedge clock)`" in the pass statement causes the display action to occur on the next posedge of clock after the assertion succeeds. Therefore, a new token is required to distinguish the assertion sequence step control from the pass statement.

Note that, since the first expression is evaluated immediately, assertion `a9` above is equivalent to:

```
always @(posedge clk or negedge rst)
   if(state == REQ)
      assert(req1)
         assert(1;gnt;!req1) @@(posedge clk);
```

The sequence notation "`(1;<expression_or_sequence>)`" is a convenient shorthand, indicating that the `<expression_or_sequence>` is to be evaluated on the next occurrence of the step control event. This is because the expression '`1`' is evaluated immediately and is always true.

Sequential assertions using the **`assert`** keyword are called *clocked immediate assertions*, since the expressions are evaluated as with immediate assertions. Similarly, clocked strobed assertions may be written using the **`assert_strobe`** keyword, in which each expression in the sequence is evaluated either at the end of the timeslice in which the assertion is executed or in which the step control event occurs. The pass and fail statements of clocked strobed assertions have the same restrictions as strobed assertions.

Specifying an explicit step control for a sequence makes it possible to use clocked assertions in combinational always blocks.

```
always @(foo,bar)
   assert_strobe (a;b;c) @@(posedge clk);
   // look for a when foo or bar changes, then look for b on next posedge clk
```

Since it is common for combinational always blocks to be executed multiple times in a single timestep as the signals in the event trigger expression settle, it is common to use strobed assertions in combinational always blocks. Immediate assertions are commonly used in clocked always blocks.

Note that to avoid races, the variables read in clocked immediate assertions should be written by nonblocking assignments. Expressions in clocked strobed assertions are always sampled at the end of the timestep, so no race conditions should occur.

An assertion could be executed twice in the same timestep via a zero-delay loop or a combinational always block, for example. If a clocked immediate assertion is executed more than once at the same timestep, the first expression in the sequence will be re-evaluated. If a clocked strobed assertion is executed more than once at the same timestep, the first expression in the sequence will be evaluated once at the end of the timestep.

An assertion shall only spawn a single process to evaluate the next expression in the sequence at the next step control event. If the step control event occurs multiple times at the same timestep, then in a clocked immediate assertion the current expression in the sequence shall be re-evaluated. In a clocked strobed assertion, the current expression will still be evaluated only once at the end of the timestep. The next expression in the sequence shall not be evaluated until the step control occurs in a later timestep.

As mentioned above, the execution of a sequential assertion spawns a process that monitors each event in the sequence when the step control event occurs. If the sequential assert statement is executed again before the sequence spawned by the original execution has expired, then a new process shall be spawned that looks for the sequence starting at the current timestep. It is therefore possible to have multiple processes in-flight, each monitoring the same sequence, but offset in time. It is possible for these multiple processes to be satisfied by the same sequential behavior, even though the processes are offset in time. In such a case, both processes shall terminate at the same timestep, in which both sequences are satisfied. Consider:

```
module top;
   reg clk = 0;
   reg a,b,c;

   initial begin
      #10 clk = 1;
      forever begin
         clk = 0;
         clk = 1; // 2 posedges clk at 10,20,30,40...
         #5 clk = 0;
         #5 clk = 1;
      end
   end

   always @(posedge clk)
      assert(a;b;c) @@(posedge clk);
      // 'a' is evaluated only once at 10, 'b' once at 20, 'c' once at 30
```

Note that the step control expression may be any valid event expression in SystemVerilog. The following assertions all use valid step control expressions:

```
bit clk;
event ev1;

always @(posedge clk or negedge reset) begin
   assert (a;b;c) @@(negedge clk); // sequence sampled on negedge clk
   assert (a;b;c) @@(clk); // sequence sampled on any edge of clk
   assert (a;b;c) @@(ev1); // sequence sampled when event ev1 fires
   a11: assert(a;b;c) @@(posedge clk iff !rst);
      // sequence sampled on posedge clk if rst == 0
end
```

Note the use of the `iff` operator in assertion a11 above. In effect, this allows a “gated clock” to control the assertion without the user having to declare the gated clock explicitly (see section 8.9). Because this could have significant impact on the ability of Formal Verification tools to evaluate the assertion successfully, it is recommended that this construct be used only for simulation.

This flexibility also allows nested assertions to use different clocks:

```
always @(posedge clk) begin
```

```
      assert (a;b) @@(posedge clk) // on posedge clk
      assert (1;c;d) @@(negedge clk); // look for c and d on negedge clk
      assert (e;f) @@(posedge clk2)
      assert (1;g;h) @@(ev1);
   end
```

11.6 More expression sequences

A number of steps can be skipped either by writing expressions which are always true:

```
assert (a;1;1;c) @@(posedge clk); // two steps between a and c
```

or by using the notation [n] to count the number of steps:

```
assert (a;[2];c) @@(posedge clk); // two steps between a and c
assert (a;[1];[1];c) @@(posedge clk); // two steps between a and c
```

Note that in [n], the n must be a non-negative literal or a constant expression. [0] has no effect. The number of steps to be skipped may also be expressed using [min:max], where the minimum number of steps must be greater than or equal to zero. Both min and max must be a literal or constant expression.

```
assert (a;[0:10];b) @@(posedge clk);
// b occurs between the next and 11th clock edges, inclusive
```

If an expression must be repeated a defined number of times, this can be expressed with a trailing *[n]. If it can be repeated a minimum or maximum number of times, this can be expressed with a trailing *[min:max]. These repetition counts must also be literals or constant expressions.

```
assert ((a; b)*[5]) @@(posedge clk); // a;b;a;b;a;b;a;b;a;b
assert ((a*[0:3];b;c)) @@(posedge clk); // equivalent to
                        //    (b;c) or (a;b;c) or (a;a;b;c) or (a;a;a;b;c).
```

This means that a sequence `a;ab;a;b;c;` will pass. The expression sequence is not equivalent to `((a && !b)* [0:3];b;c)`, which would fail the same sequence.

The rules for specifying repeat counts are summarized as:

— Each form of repeat count specifies a minimum and maximum number of occurrences

— expr*[n:m], where n is the minimum, m is the maximum

— expr*[n], same as expr*[n:n]

— [n], same as 1*[n:n]

— The sum of the minimum repeat counts for all terms in a sequence must be greater than 0

— The sequence as a whole cannot be empty

— The last term in a sequence shall not have a min:max range of repetition. If it does, it shall be an error.

11.7 Aborting assertions externally

A named assertion can be disabled like any other named SystemVerilog block. If this is done before the expression sequence has finished, it means that neither the pass statement nor the fail statement shall be executed.

```
disable cas;
```

Note that if the **`disable`** is applied at the same simulation time step as the last clock step of a sequence, there is a race in the case of an immediate assertion, but a strobed assertion is always disabled.

If the pass or fail statement is executing when the disable is executed, the statement shall be disabled, just as if the statement were in another named block that gets disabled.

If a sequential assertion has been executed multiple times before the sequence has expired, then all instances of the assertion shall be disabled when the assertion is disabled.

11.8 Controlling assertions

System tasks are provided to limit assertion checking to part of the design and part of the simulation time.

The **`$assertoff`** system task stops the checking of all specified assertions. When these assertions are encountered before a subsequent **`$asserton`**, the assert statement shall be ignored. Neither the pass statement nor the fail statement shall be executed. An assertion that is already executing, including execution of the pass or fail statement, is not affected by **`$assertoff`**.

The **`$assertkill`** system task disables all specified assertions and prevents them from executing until a subsequent **`$asserton`**. As with disable, the checking of the sequence is aborted, and neither the pass nor fail statement is executed.

The **`$asserton`** system task re-enables the execution of all specified assertions.

The assertion control system tasks may be used with or without arguments. When invoked with no arguments, the system task refers to all assertions throughout the model. Refer to section 16.5 for the syntax of these system tasks.

Assertions are on by default until turned off. When an assertion control task is specified with arguments, the first argument indicates how many levels of the hierarchy below each specified module instance to turn on or off. Subsequent arguments specify which scopes of the model in which to control assertions. These arguments can specify entire modules or individual named assertions within a module. Setting the first argument to 0 causes all assertions in the specified module and in all module instances below the specified module to be affected. The argument 0 applies only to subsequent arguments which specify module instances, and not to individual assertions.

11.9 System functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is "one-hot". The following system functions are included to facilitate such common assertion functionality:

— **`$onehot`** (<expression>) returns true if only one and only one bit of expression is high.

— **`$onehot0`**(<expression>) returns true if at most one bit of expression is low.

— **`$inset`** (<expression>, <expression> {, <expression> }) returns true if the first expression is equal to at least one of the subsequent expression arguments.

— **`$insetz`**(<expression>,<expression> {, <expression> }) returns true if the first expression is equal to at least other expression argument. Comparison is performed using casez semantics, so '`z`' or '`?`' bits are treated as don't-cares.

— **`$isunknown`**(<expression>) returns true if any bit of the expression is '`x`'. This is equivalent to **^**<expression> `=== 'bx`.

All of the above system functions have a return type of bit. A return value of `1'b1` indicates true, and a return value of `1'b0` indicates false.

Section 12 Hierarchy

12.1 Introduction (informative)

Verilog has a simple organization. All data, functions and tasks are in modules except for system tasks and functions, which are global, and may be defined in the PLI. A Verilog module can contain instances of other modules. Any uninstantiated module is at the top level. This does not apply to libraries, which therefore have a different status and a different procedure for analyzing them. A hierarchical name can be used to specify any named object from anywhere in the instance hierarchy. The module hierarchy is often arbitrary and a lot of effort is spent in maintaining port lists.

In Verilog, only net, **`reg`**, **`integer`** and **`time`** data types can be passed through module ports.

SystemVerilog adds many enhancements for representing design hierarchy:

— A global declaration space, visible to all modules at all levels of hierarchy

— Nested module declarations, to aid in representing self-contained models and libraries

— Relaxed rules on port declarations

— Simplified named port connections, using `.name`

— Implicit port connections, using **`.*`**

— Time unit and time precision specifications bound to modules

— A concept of interfaces to bundle connections between modules (presented in section 13)

An important enhancement in SystemVerilog is the ability to pass any data type through module ports, including nets, and all variable types including reals, arrays, and structures.

12.2 The $root top level

In SystemVerilog there is a top level called $root, which is the whole source text. This allows declarations outside any named modules or interfaces, unlike Verilog.

SystemVerilog requires an elaboration phase. All modules and interfaces must be parsed before elaboration. The order of elaboration shall be: First, look for explicit instantiations in $root. If none, then look for implicit instantiations (i.e. uninstantiated modules). Next, traverse non-generate instantiations depth-first, in source order. Finally, execute generate blocks depth-first, in source order.

The source text can include the declaration and use of modules and interfaces. Modules can include the declaration and use of other modules and interfaces. Interfaces can include the declaration and use of other interfaces. A module or interface need not be declared before it is used in text order.

A module can be explicitly instantiated in the $root top-level. All uninstantiated modules become implicitly instantiated within the top level, which is compatible with Verilog.

The following paragraphs compare the $root top level and modules.

The $root top level:

— has a single occurrence

— can be distributed across any number of files

— variable and net definitions are in a global name space and can be accessed throughout the hierarchy

— task and function definitions are in a global name space and can be accessed throughout the hierarchy

— shall not contain **`initial`** or **`always`** procedures

— shall contain procedural statements, which will be executed one time, as if in an **`initial`** procedure

Modules:

— can have any number of module definitions

— can have any number of module instances, which create new levels of hierarchy

— can be distributed across any number of files, and can be defined in any order

— variable and net definitions are in the module instance name space and are local to that scope

— task and function definitions are in the module instance name space and are local to that scope

— can contain any number of **`initial`** and **`always`** procedures

— shall not contain procedural statements that are not within an **`initial`** procedure, **`always`** procedure, task, or function

When an identifier is referenced within a scope, SystemVerilog follows the Verilog name search rules, and then searches in the $root global name space. An identifier in the global name space can be explicitly selected by pre-pending **`$root.`** to the identifier name. For example, a global variable named `system_reset` can be explicitly referenced from any level of hierarchy using `$root.system_reset`.

The $root space can be used to model abstract functionality without modules. The following example illustrates using the $root space with just declarations, statements and functions.

```
typedef int myint;

function void main ();
   myint i,j,k;
   $display ("entering main...");
   left (k);
   right (i,j,k);
   $display ("ending... i=%0d, j=%0d, k=%0d", i, j, k);
endfunction

function void left (output myint k);
   k = 34;
   $display ("entering left");
endfunction

function void right (output myint i, j, input myint k);
   $display ("entering right");
   i = k/2;
   j = k+i;
endfunction

main();
```

12.3 Module declarations

```
module_declaration ::=          // from Annex A.1.3
          { attribute_instance } module_keyword  module_identifier [ parameter_port_list ]
          [ list_of_ports ] ; [unit] [precision] { module_item }
          endmodule
        | { attribute_instance } module_keyword  module_identifier [ parameter_port_list ]
          [ list_of_port_declarations ] ; [unit] [precision] { non_port_module_item }
          endmodule

module_keyword ::= module | macromodule          // from Annex A.1.3

timeunits_declaration ::=          // from Annex A.1.3
          timeunit time_literal ;
        | timeprecision time_literal ;
        | timeunit time_literal ;
          timeprecision time_literal ;
        | timeprecision time_literal ;
          timeunit time_literal ;

module_or_generate_item_declaration ::=          // from Annex A.1.5
          net_declaration
        | data_declaration
        | event_declaration
        | genvar_declaration
        | task_declaration
        | function_declaration

module_item ::=          // from Annex A.1.5
          port_declaration ;
        | non_port_module_item

non_port_module_item ::=          // from Annex A.1.5
          { attribute_instance } generated_module_instantiation
        | { attribute_instance } local_parameter_declaration
        | { attribute_instance } module_or_generate_item
        | { attribute_instance } parameter_declaration ;
        | { attribute_instance } specify_block
        | { attribute_instance } specparam_declaration
        | module_declaration

module_or_generate_item ::=          // from Annex A.1.5
          { 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
        | { attribute_instance } combinational_statement
        | { attribute_instance } latch_statement
        | { attribute_instance } ff_statement
        | module_common_item

module_common_item ::=          // from Annex A.1.5
          { attribute_instance } module_or_generate_item_declaration
        | { attribute_instance } interface_instantiation
```

Syntax 12-1—Module declaration syntax (excerpt from Annex A)

In Verilog, a module must be declared apart from other modules, and can only be instantiated within another module. A module declaration may appear after it is instantiated in the source text.

SystemVerilog adds the capability to nest module declarations, and to instantiate modules in the $root top-level space, outside of other modules.

```
module m1(...); ... endmodule

module m2(...); ... endmodule

module m3(...);

   m1 i1(...); // instantiates the local m1 declared below
   m2 i4(...); // instantiates m2 - no local declaration
   module m1(...); ... endmodule // nested module declaration,
                                 // m1 module name is in m3's name space
endmodule

m1 i2(...); // module instance in the $root space,
            // instantiates the module m1 that is not nested in another module
```

12.4 Nested modules

A module can be declared within another module. The outer name space is visible to the inner module, so that any name declared there can be used, unless hidden by a local name, provided the module is declared and instantiated in the same scope.

One purpose of nesting modules is to show the logical partitioning of a module without using ports. Names that are global are in the outermost scope, and names that are only used locally can be limited to local modules.

```
// This example shows a D-type flip-flop made of NAND gates
module dff_flat(input d, ck, pr, clr, output q, nq);
wire q1, nq1, q2, nq2;

    nand g1b (nq1, d, clr, q1);
    nand g1a (q1, ck, nq2, nq1);

    nand g2b (nq2, ck, clr, q2);
    nand g2a (q2, nq1, pr, nq2);

    nand g3a (q, nq2, clr, nq);
    nand g3b (nq, q1, pr, q);
endmodule


// This example shows how the flip-flop can be structured into 3 RS latches.
module dff_nested(input d, ck, pr, clr, output q, nq);
wire q1, nq1, nq2;

    module ff1;
        nand g1b (nq1, d, clr, q1);
        nand g1a (q1, ck, nq2, nq1);
    endmodule
    ff1 i1;

    module ff2;
```

```
            wire q2; // This wire can be encapsulated in ff2
            nand g2b (nq2, ck, clr, q2);
            nand g2a (q2, nq1, pr, nq2);
        endmodule
        ff2 i2;

        module ff3;
            nand g3a (q, nq2, clr, nq);
            nand g3b (nq, q1, pr, q);
        endmodule
        ff3 i3;
    endmodule
```

The nested module declarations can also be used to create a library of modules that is local to part of a design.

```
    module part1(....);
       module and2(input a; input b; output z);
       ....
       endmodule
       module or2(input a; input b; output z);
       ....
       endmodule
       ....
       and2 u1(....), u2(....), u3(....);
       .....
    endmodule
```

This allows the same module name, e.g. `and2`, to occur in different parts of the design and represent different modules. Note that an alternative way of handling this problem is to use configurations.

12.5 Port declarations

inout_declaration ::= **inout** [port_type] list_of_port_identifiers // from Annex A.2.1.2

input_declaration ::= **input** [port_type] list_of_port_identifiers // from Annex A.2.1.2

output_declaration ::= // from Annex A.2.1.2
  output [port_type] list_of_port_identifiers
  | output data_type list_of_variable_port_identifiers

interface_port_declaration ::= // from Annex A.2.1.2
  interface list_of_interface_identifiers
  | **interface .** modport_identifier list_of_interface_identifiers
  | **identifier** list_of_interface_identifiers
  | **identifier .** modport_identifier list_of_interface_identifiers

port_type ::= // from Annex A.2.2.1
  data_type { packed_dimension }
  | net_type [signing] { packed_dimension }
  | **trireg** [signing] { packed_dimension }
  | **event**
  | [signing] { packed_dimension } range

signing ::= [**signed**] | [**unsigned**] // from Annex A.2.2.1

Syntax 12-2—Port declaration syntax (excerpt from Annex A)

With SystemVerilog, a port can be a declaration of a net, an interface, an event, or a variable of any type, including an array, a structure or a union.

```
typedef struct {
            bit isfloat;
            union { int i; shortreal f; } n;
} tagged; // named structure

module mh1 (input int in1, input shortreal in2, output tagged out);
   ...
endmodule
```

For the first port, if neither a type nor a direction is specified, then it shall be assumed to be a member of a port list, and any port direction or type declarations must be declared after the port list. This is compatible with the Verilog-1995 syntax. If the first port type but no direction is specified, then the port direction shall default to **`inout`**. If the first port direction but no type is specified, then the port type shall default to **`wire`**. This default type can be changed using the **`` `default_nettype ``** compiler directive, as in Verilog.

```
// Any declarations must follow the port list, because first port does not
// have either a direction or type specified; Port directions default to inout
module mh4(x, y);
   int x;
   char y;
   ...
endmodule
```

For subsequent ports in the port list, if the type and direction are omitted, then both are inherited from the previous port. If only the direction is omitted, then it is inherited from the previous port. If only the type is omitted, it shall default to **`wire`**. This default type can be changed using the **`` `default_nettype ``** compiler directive, as in Verilog.

```
// second port inherits its direction and type from previous port
module mh3 (input char a, b);
   ...
endmodule
```

A software tool can use the port direction to check against writing to an input port or not writing to an output port.

Ports which are of a net type can have multiple drivers, which are resolved according to the net's resolution function. A driver can be an **`output`** port of an instantiation, or a continuous assignment.

If the port is of type **`logic`** or any other variable data type, then the port has the value of the last assignment to it. If the port is an **`inout`**, then these assignments can be inside or outside the module. If the port is an **`output`**, then these assignments shall only be inside the module. This provides a way to model a port which is meant to be a single driver.

12.6 Time unit and precision

The time unit can be set by the **`timeunit`** keyword to a time which must be a power of 10 units. For example:

```
timeunit 100ps;
```

The time unit is determined as follows:

1) If a **timeunit** has been specified in the current module, then the time unit is set to module's time units.

2) Else, if the module definition is nested, then the time unit is inherited from the enclosing module.

3) Else, if a **‘timescale** directive has been specified, then the time unit is set to the units of last **‘timescale** directive.

4) Else, if the **$root** top level has a time unit, then the time unit set to the time units of the root module.

5) Else, the simulator’s default time units are used.

The simulator’s default time units follow the rules of Verilog.

The time precision is set by the `timeprecision` keyword to a time which must be a power of 10 units e.g.

```
timeprecision 100fs;
```

If the `timeprecision` is not specified, then the precision is determined following the same precedence as with time units.

It is an error to set a precision larger than the current unit.

The `timeunit` and `timeprecision` keywords shall precede any other item in the top level, module, or interface, because the other items can contain delays and therefore can be dependent on the time unit.

12.7 Module instances

```
module_instantiation ::=          // from Annex A.4.1.1
        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 | data_type
named_parameter_assignment ::=
        . parameter_identifier ( [ expression ] )
      | . parameter_identifier ( [ data_type ] )
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 }
      | dot_named_port_connection { , dot_named_port_connection }
      | { named_port_connection , } dot_star_port_connection { , named_port_connection }
ordered_port_connection ::= { attribute_instance } [ expression ]
named_port_connection ::= { attribute_instance } .port_identifier ( [ expression ] )
dot_named_port_connection ::=
        { attribute_instance } .port_identifier
      | named_port_connection
dot_star_port_connection ::= { attribute_instance } .*
```

Syntax 12-3—Module instance syntax (excerpt from Annex A)

A module can be used (instantiated) in two ways, hierarchical or top level. Hierarchical instantiation allows more than one instance of the same type. The module name can be a module previously declared or one declared later. Actual parameters can be named or ordered. Port connections can be named, ordered or implicitly connected. They can be nets, variables, or other kinds of interfaces, events, or expressions. See below for the connection rules.

Consider an ALU accumulator (`alu_accum`) example module that includes instantiations of an ALU module, an accumulator register (`accum`) module and a sign-extension (`xtend`) module. The module headers for the three instantiated modules are shown in the following example code.

```
module alu (
   output reg [7:0] alu_out,
   output reg zero,
   input [7:0] ain, bin,
   input [2:0] opcode);
   // RTL code for the alu module
endmodule

module accum (
   output reg [7:0] dataout,
   input [7:0] datain,
   input clk, rst_n);
   // RTL code for the accumulator module
endmodule

module xtend (
   output reg [7:0] dout,
   input din,
   input clk, rst_n);
   // RTL code for the sign-extension module
endmodule
```

12.7.1 Instantiation using positional port connections

Verilog has always permitted instantiation of modules using positional port connections, as shown in the `alu_accum1` module example, below.

```
module alu_accum1 (
   output [15:0] dataout,
   input [7:0] ain, bin,
   input [2:0] opcode,
   input clk, rst_n);
   wire [7:0] alu_out;

   alu alu (alu_out, , ain, bin, opcode);
   accum accum (dataout[7:0], alu_out, clk, rst_n);
   xtend xtend (dataout[15:8], alu_out[7], clk, rst_n);
endmodule
```

As long as the connecting variables are ordered correctly and are the same size as the instance-ports that they are connected to, there will be no warnings and the simulation will work as expected.

12.7.2 Instantiation using named port connections

Verilog has always permitted instantiation of modules using named port connections as shown in the `alu_accum2` module example.

```
module alu_accum2 (
   output [15:0] dataout,
   input [7:0] ain, bin,
   input [2:0] opcode,
      input clk, rst_n);
      wire [7:0] alu_out;

      alu   alu   (.alu_out(alu_out), .zero(),
                   .ain(ain), .bin(bin), .opcode(opcode));
      accum accum (.dataout(dataout[7:0]), .datain(alu_out),
                   .clk(clk), .rst_n(rst_n));
      xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]),
                   .clk(clk), .rst_n(rst_n));
   endmodule
```

Named port connections do not have to be ordered the same as the ports of the instantiated module. The variables connected to the instance ports must be the same size or a port-size mismatch warning will be reported.

12.7.3 Instantiation using implicit .name port connections

SystemVerilog adds the capability to implicitly instantiate ports using a .name syntax if the instance-port name and size match the connecting variable-port name and size. This enhancement eliminates the requirement to list a port name twice when the port name and signal name are the same, while still listing all of the ports of the instantiated module for documentation purposes.

In the following `alu_accum3` example, all of the ports of the instantiated alu module match the names of the variables connected to the ports, except for the unconnected zero port, which is listed using a named port connection, showing that the port is unconnected. Implicit .name port connections are made for all name and size matching connections on the instantiated module.

In the same `alu_accum3` example, the `accum` module has an 8-bit port called `dataout` that is connected to a 16-bit bus called `dataout`. Because the internal and external sizes of `dataout` do not match, the port must be connected by name, showing which bits of the 16-bit bus are connected to the 8-bit port. The `datain` port on the `accum` is connected to a bus by a different name (`alu_out`), so this port is also connected by name. The `clk` and `rst_n` ports are connected using implicit .name port connections. Also in the same `alu_accum3` example, the `xtend` module has an 8-bit output port called `dout` and a 1- bit input port called `din`. Since neither of these port names match the names (or sizes) of the connecting variables, both are connected by name. The `clk` and `rst_n` ports are connected using implicit .name port connections.

```
   module alu_accum3 (
      output [15:0] dataout,
      input [7:0] ain, bin,
      input [2:0] opcode,
      input clk, rst_n);
      wire [7:0] alu_out;

      alu   alu   (.alu_out, .zero(), .ain, .bin, .opcode);
      accum accum (.dataout(dataout[7:0]), .datain(alu_out), .clk, .rst_n);
      xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .clk, .rst_n);
   endmodule
```

Implicit .name port connections do not have to be ordered the same as the ports of the instantiated module.

The following rules apply to implicit .name port connections:

— For an implicit .name port connection to be legal, the connecting variable name must match the port name of the instantiated module.

— For an implicit .name port connection to be legal, the connecting variable size must match the port size of the instantiated module.

— For an implicit .name port connection to be legal, the connecting variable data type must be compatible to the port data type of the instantiated module. See section 12.7.5 for a description of compatible data types for implicit port connections.

— Implicit .name port connections cannot be used in the same instantiation with positional port connections.

— Implicit .name port connections may be used in the same instantiation with named port connections.

— Implicit .name port connections cannot be used in the same instantiation with implicit .* port connections.

— The order of the implicit .name port connections does not have to match the port-order of the instantiated module.

— All connecting variables must be explicitly declared, either as a port in the parent module (following the rules of Verilog-2001) or as an explicit net or variable of one or more bits.

12.7.4 Instantiation using implicit .* port connections

SystemVerilog adds the capability to implicitly instantiate ports using a `.*` syntax for all ports where the instance-port name and size match the connecting variable-port name and size. This enhancement eliminates the requirement to list any port where the name and size of the connecting variable match the name and size of the instance port. This implicit port connection style is used to indicate that all port names and sizes match the connections where emphasis is placed only on the exception ports. The implicit `.*` port connection syntax can greatly facilitate rapid block-level testbench generation where all of the testbench variables are chosen to match the instantiated module port names and sizes.

In the following `alu_accum4` example, all of the ports of the instantiated alu module match the names of the variables connected to the ports, except for the unconnected zero port, which is listed using a named port connection, showing that the port is unconnected. The implicit `.*` port connection syntax connects all other ports on the instantiated module.

In the same `alu_accum4` example, the `accum` module has an 8-bit port called `dataout` that is connected to a 16-bit bus called `dataout`. Because the internal and external sizes of `dataout` do not match, the port must be connected by name, showing which bits of the 16-bit bus are connected to the 8-bit port. The `datain` port on the `accum` is connected to a bus by a different name (`alu_out`), so this port is also connected by name. The `clk` and `rst_n` ports are connected using implicit `.*` port connections. Also in the same `alu_accum4` example, the `xtend` module has an 8-bit output port called `dout` and a 1- bit input port called `din`. Since neither of these port names match the names (or sizes) of the connecting variables, both are connected by name. The `clk` and `rst_n` ports are connected using implicit `.*` port connections.

```
module alu_accum4 (
   output [15:0] dataout,
   input [7:0] ain, bin,
   input [2:0] opcode,
   input clk, rst_n);
   wire [7:0] alu_out;

   alu   alu   (.*, .zero());
   accum accum (.*, .dataout(dataout[7:0]), .datain(alu_out));
   xtend xtend (.*, .dout(dataout[15:8]), .din(alu_out[7]));
endmodule
```

The following rules apply to implicit `.*` port connections:

— For an implicit `.*` port connection to be legal, all implicitly connected ports must have a connecting variable name to match the port name of the instantiated module.

— For an implicit `.*` port connection to be legal, all implicitly connected ports must have a connecting variable size to match the port size of the instantiated module.

— For an implicit `.*` port connection to be legal, the connecting variable data type must be compatible to the port data type of the instantiated module. See section 12.7.5 for a description of compatible data types for implicit port connections.

— Implicit `.*` port connections cannot be used in the same instantiation with positional port connections.

— Implicit **.*** port connections may be used in the same instantiation with named port connections.

— Implicit **.*** port connections cannot be used in the same instantiation with implicit .name port connections.

— If implicit **.*** port connections are used in an instantiation, all unconnected ports must be shown using named port connections.

— When the implicit **.*** port connection is mixed in the same instantiation with named port connections, the implicit **.*** port connection token can be placed anywhere in the port list.

— All connecting variables must be explicitly declared, either as a port in the parent module (following the rules of Verilog-2001) or as an explicit net or variable of one or more bits.

Modules may be instantiated into the same parent module using any combination of legal positional, named, implicit .name connected and implicit **.*** connected instances as shown in `alu_accum5` example.

```
module alu_accum5 (
   output [15:0] dataout,
   input [7:0] ain, bin,
   input [2:0] opcode,
   input clk, rst_n);
   wire [7:0] alu_out;

   // mixture of named port connections and
   // implicit .name port connections
   alu   alu   (.ain(ain), .bin(bin), .alu_out, .zero(), .opcode);

   // positional port connections
   accum accum (dataout[7:0], alu_out, clk, rst_n);

   // mixture of named port connections and implicit .* port connections
   xtend xtend (.dout(dataout[15:8]), .*, .din(alu_out[7]));
endmodule
```

12.7.5 Compatible data types for implicit port connections

Implicit port connections are permitted between any two data types that are allowed by SystemVerilog port connection rules, as long as the SystemVerilog simulator is not required to report a warning about the connection. Any SystemVerilog instantiation that would cause a warning to be issued must be connected by name if other ports of the instance are instantiated using an implicit port connection style.

If, for example, a top-level module connects a signal named `net1` of any data type to an instantiated submodule with a port also named `net1` of same data type, SystemVerilog will run this simulation without warning, because the data types are the same across ports. It is legal to make this type of connection using an implicit port connection style.

If, for example, a top-level module connects a signal named `net2` of type **`wire`** to an instantiated submodule with a port also named `net2` of type **`reg`**, Verilog simulators run this simulation without warning, because the data types are compatible across ports. It is legal to make this type of connection using an implicit port connection style.

If, for example, a top-level module connects a signal named `net3` of type **`tri1`** to an instantiated submodule with a port named `net3` of type **`tri0`**, Verilog simulators issue a warning and the top-level data type (**`tri1`**) is used during simulation, as described in the IEEE Verilog-2001 Standard. It is legal to make this type of connection using named port connections but it shall be a syntax error to make this connection using an implicit port connection style. Any port connection that results in a required warning message shall not be permitted to be instantiated using an implicit port connection style.

A top-level module shall not implicitly connect a signal of any data type to a port by the same name of another data type if connecting the data types is illegal as defined by this SystemVerilog standard.

12.8 Port connection rules

If a port declaration has a variable data type such as **`logic`**, then its direction controls how it can be connected, as follows:

— An **`input`** can be connected to any expression of a compatible data type. If unconnected, it has the initial value corresponding to the data type.

— An **`output`** can be connected to a variable (or a concatenation) of a compatible data type, and has shared variable behavior if multiple outputs are connected (last write wins); An **`output logic`** can be connected to a net (to provide a resolution function in the case of multiple drivers).

— An **`inout`** can be connected to a variable (or a concatenation) of the same data type.

If a port declaration has a **`wire`** type (which is the default), or any other net type, then its direction controls how it can be connected as follows:

— An **`input`** can be connected to any expression of a compatible data type. If unconnected, it has the value `'z`.

— An **`output`** can be connected to a net type (or a concatenation of net types) or left unconnected, but not to a **`logic`** variable.

— An **`inout`** can be connected to a net type (or a concatenation of net types) or left unconnected, but not to a **`logic`** variable.

Note that where the data types differ between the port declaration and connection, an initial value change event may be caused at time zero.

If a port declaration has a generic **`interface`** type, then it can be connected to an interface of any type. If a port declaration has a named interface type, then it must be connected to a generic interface or an interface of the same type.

A mismatch between vector width across a port connection is resolved as follows:

— If the port is a net vector, then the Verilog connection rules for nets are followed.

— If the port is an **`inout`** port variable, then a port connection must have the same size and representation on both sides of the port. It shall be an error if there is a mismatch.

— If the port is an **`input`** or an **`output`** variable, then the Verilog assignment rules are followed.

For an unpacked array port, the port and the array connected to the port must have the same number of unpacked dimensions, and each dimension of the port must have the same size as the corresponding dimension of the array being connected.

If the size and type of the port connection match the size and type of a single instance port, the connection shall be made to each instance in the array.

If the port connection is an unpacked array, the unpacked array dimensions of each port connection shall be compared with the dimensions of the instance array. If they match exactly in size, each element of the port connection shall be matched to the port left index to left index, right index to right index. If they do not match it shall be considered an error.

If the port connection is a packed array, each instance shall get a part-select of the port connection, starting with all right-hand indices to match the right most part-select, and iterating through the right most dimension first. Too many or too few bits to connect all the instances shall be considered an error.

12.9 Name spaces

There is one name space hierarchy in SystemVerilog. A type name may be not be the same as an instance name. Type names include modules, interfaces, and data types. Instance names include tasks, functions, procedures, variables, constants and labels as well as module and interface instances.

Pre-defined (built-in) names begin with **`$`**. For example **`$root`** is the name of the top level of the hierarchy.

Names are initially global. A new scope is defined by:

— a module or interface

— a task or function

— a sequential or parallel block

— a structure or union

Tasks and function definitions cannot be nested within themselves, but can be defined in modules or interfaces. The declaration in the closest enclosing scope is matched.

12.10 Hierarchical names

Hierarchical names are also called nested identifiers. They consist of instance names separated by periods, where an instance name may be an array element.

```
$root.mymodule.u1 // absolute name
u1.struct1.field1 // u1 must be visible locally or above, including globally
adder1[5].sum
```

Nested identifiers can be read (in expressions), written (in assignments or task/function calls) or triggered off (in event expressions). They can also be used as type, task or function names.

Section 13 Interfaces

13.1 Introduction (informative)

The communication between blocks of a digital system is a critical area that can affect everything from ease of RTL coding, to hardware-software partitioning to performance analysis to bus implementation choices and protocol checking. The interface construct in SystemVerilog was created specifically to encapsulate the communication between blocks, allowing a smooth migration from abstract system-level design through successive refinement down to lower-level register-transfer and structural views of the design. By encapsulating the communication between blocks, the interface construct also facilitates design re-use. The inclusion of interface capabilities is one of the major advantages of SystemVerilog.

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

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

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

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

The methods can be abstract, i.e. defined in one module and called in another, using the export and import constructs. This could be coded using hierarchical path names, but this would impede re-use because the names would be design-specific. A better way is to declare the task and function names in the interface, and to use local hierarchical names from the interface instance for both definition and call. Broadcast communication is modeled by **`forkjoin`** tasks, which can be defined in more than one module and executed concurrently.

13.2 Interface syntax

```
modport_declaration ::= modport list_of_modport_identifiers ;          // from Annex A.2.9
list_of_modport_identifiers ::= modport_item { , modport_item }
modport_item ::= modport_identifier ( modport_port { , modport_port } )
modport_port ::=          // from Annex A.2.9
                input [port_type] port_identifier
              | output [port_type] port_identifier
              | inout [port_type] port_identifier
              | interface_identifier . port_identifier
              | import_export task named_task_proto
              | import_export function named_fn_proto
              | import_export task_or_function_identifier { , task_or_function_identifier }
import_export ::= import | export
interface_port_declaration ::=          // from Annex A.2.1.2
                interface list_of_interface_identifiers
              | interface . modport_identifier  list_of_interface_identifiers
              | identifier list_of_interface_identifiers
              | identifier . modport_identifier  list_of_interface_identifiers
interface_or_generate_item ::=          // from Annex A.1.6
                { attribute_instance } continuous_assign
              | { attribute_instance } initial_construct
              | { attribute_instance } always_construct
              | { attribute_instance } combinational_statement
              | { attribute_instance } latch_statement
              | { attribute_instance } ff_statement
              | { attribute_instance } local_parameter_declaration
              | { attribute_instance } parameter_declaration ;
              | module_common_item
              | { attribute_instance } modport_declaration
interface_item ::=          // from Annex A.1.6
                port_declaration
              | non_port_interface_item
non_port_interface_item ::=          // from Annex A.1.6
              { attribute_instance } generated_interface_instantiation
              | { attribute_instance } local_parameter_declaration
              | { attribute_instance } parameter_declaration ;
              | { attribute_instance } specparam_declaration
              | interface_or_generate_item
              | interface_declaration
interface_instantiation ::=          // from Annex A.4.1.2
                interface_identifier [ parameter_value_assignment ] module_instance { , module_instance } ;
```

Syntax 13-1—Interface syntax (excerpt from Annex A)

The interface construct provides a new hierarchical structure. It can contain smaller interfaces and can be passed through ports.

The aim of interfaces is to encapsulate communication. At the lower level, this means bundling variables and wires in interfaces, and bundling ports with directions in modports. The modules can be made generic so that the interfaces can be changed. The following examples show these features. At a higher level of abstraction, communication can be done by tasks and functions. Interfaces can include task and function definitions, or just task and function prototypes with the definition in one module (server/slave) and the call in another (client/ master).

An interface is declared as follows:

```
interface <identifier>; <interface_items> endinterface [: <name> <identifier>]
```

An interface can be instantiated hierarchically like a module with or without ports. For example:

```
myinterface #(100) scalar1, vector[9:0];
```

Interfaces can be declared and instantiated in modules (either flat or hierarchical) but modules can neither be declared nor instantiated in interfaces.

The simplest use of an interface is to bundle wires, as is illustrated in the examples below.

13.2.1 Example without using interfaces

This example shows a simple bus implemented without interfaces. Note that the logic type can replace wire and reg if no resolution of multiple drivers is needed.

```
module memMod( input    bit req,
                        bit clk,
                        bit start,
                        logic[1:0] mode,
                        logic[7:0] addr,
               inout    logic[7:0] data,
               output   bit gnt,
                        bit rdy );
   logic avail;

   ...
endmodule

module cpuMod(
   input    bit clk,
            bit gnt,
            bit rdy,
   inout    logic [7:0] data,
   output   bit req,
            bit start,
            logic[7:0] addr,
            logic[1:0] mode );
   ...
endmodule

module top;
   logic req, gnt, start, rdy; // req is logic not bit here
   logic clk = 0;
   logic [1:0] mode;
   logic [7:0] addr, data;

   memMod mem(req, clk, start, mode, addr, data, gnt, rdy);
   cpuMod cpu(clk, gnt, rdy, data, req, start, addr, mode);

endmodule
```

13.2.2 Interface example using a named bundle

The simplest form of a SystemVerilog interface is a bundled collection of variables or nets. When an interface is used as a port, the variables and nets in it are assumed to be **inout** ports. The following interface example shows the basic syntax for defining, instantiating and connecting an interface. Usage of the SystemVerilog interface capability can significantly reduce the amount of code required to model port connections.

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

module memMod(simple_bus a, // Use the simple_bus interface
              input bit clk);
   logic avail;
   // a.req is the req signal in the 'simple_bus' interface
   always @(posedge clk) a.gnt <= a.req & avail;
endmodule

module cpuMod(simple_bus b, input bit clk);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf; // Instantiate the interface

   memMod mem(sb_intf, clk); // Connect the interface to the module instance
   cpuMod cpu(.b(sb_intf), .clk(clk)); // Either by position or by name

endmodule
```

In the preceding example, if the same identifier, sb_intf, had been used to name the simple_bus interface in the memMod and cpuMod module headers, then implicit port declarations also could have been used to instantiate the memMod and cpuMod modules into the top module, as shown below.

```
module memMod (simple_bus sb_intf, input bit clk);
   ...
endmodule

module cpuMod (simple_bus sb_intf, input bit clk);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf;

   memMod mem (.*);  // implicit port connections
   cpuMod cpu (.*);  // implicit port connections

endmodule
```

13.2.3 Interface example using a generic bundle

A module header can be created with an unspecified interface instantiation as a place-holder for an interface to be selected when the module itself is instantiated. The unspecified interface is referred to as a "generic" interface port. The following interface example shows how to specify a generic interface port in a module definition.

```
// memMod and cpuMod can use any interface
module memMod (interface a, input bit clk);
   ...
endmodule

module cpuMod(interface b, input bit clk);
   ...
endmodule

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

module top;
   logic clk = 0;

   simple_bus sb_intf; // Instantiate the interface

   // Connect the sb_intf instance of the simple_bus
   // interface to the generic interfaces of the
   // memMod and cpuMod modules
   memMod mem (.a(sb_intf), .clk(clk));
   cpuMod cpu (.b(sb_intf), .clk(clk));

endmodule
```

An implicit port cannot be used to connect to a generic interface. A named port must be used to connect to a generic interface, as shown below.

```
module memMod (interface a, input bit clk);
   ...
endmodule

module cpuMod (interface b, input bit clk);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf;

   memMod mem (.*, .a(sb_intf)); // partial implicit port connections
   cpuMod cpu (.*, .b(sb_intf)); // partial implicit port connections

endmodule
```

13.3 Ports in interfaces

One limitation of simple interfaces is that the nets and variables declared within the interface are only used to connect to a port with the same nets and variables. To share an external net or variable, one that makes a connection from outside of the interface as well as forming a common connection to all module ports that instantiate the interface, an interface port declaration is required. The difference between nets or variables in the interface port list and other nets or variables within the interface is that only those in the port list can be connected externally by name or position when the interface is instantiated.

```
interface i1 (input a, output b, inout c);
   wire d;
endinterface
```

The wires a, b and c can be individually connected to the interface and thus shared with other interfaces.

The following example shows how to specify an interface with inputs, allowing a wire to be shared between two instances of the interface.

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

module memMod(simple_bus a); // Uses just the interface
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
      a.gnt <= a.req & avail; // a.req is in the 'simple_bus' interface
endmodule

module cpuMod(simple_bus b);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf1(clk); // Instantiate the interface
   simple_bus sb_intf2(clk); // Instantiate the interface

   memMod mem1(.a(sb_intf1)); // Connect bus 1 to memory 1
   cpuMod cpu1(.b(sb_intf1));
   memMod mem2(.a(sb_intf2)); // Connect bus 2 to memory 2
   cpuMod cpu2(.b(sb_intf2));

endmodule
```

Note: Because the instantiated interface names do not match the interface names used in the memMod and cpuMod modules, implicit port connections cannot be used for this example.

13.4 Modports

To bundle module ports, there are **`modport`** lists with directions declared within the interface. The keyword **`modport`** indicates that the directions are declared as if inside the module.

```
interface i2;
   wire a, b, c, d;
   modport master (input a, b, output c, d);
   modport slave (output a, b, input c, d);
endinterface
```

The **`modport`** list name (master or slave) can be specified in the module header, where the **`modport`** name acts as a direction and the interface name as a type.

```
module m (i2.master i);
   ...
endmodule

module s (i2.slave i);
   ...
endmodule

module top;
   i2 i;

   m u1(.i(i));
   s u2(.i(i));
endmodule
```

The **`modport`** list name (master or slave) can also be specified in the port connection with the module instance, where the **`modport`** name is hierarchical from the interface instance.

```
module m (i2 i);
   ...
endmodule

module s (i2 i);
   ...
endmodule

module top;
   i2 i;

   m u1(.i(i.master));
   s u2(.i(i.master));
endmodule
```

The syntax of `interface_name.modport_name  instance_name` is really a hierarchical type followed by an instance. Note that this can be generalized to any interface with a given **`modport`** name by writing **`interface`**`.modport_name instance_name`.

In a hierarchical interface, the directions in a **`modport`** declaration can themselves be **`modport`** plus name.

```
interface i1;
   interface i3;
      wire a, b, c, d;
      modport master (input a, b, output c, d);
      modport slave (output a, b, input c, d);
   endinterface
   i3 ch1, ch2;
   modport master2 (ch1.master, ch2.master);
endinterface
```

Note that if no **modport** is specified in the module header or in the port connection, then all the wires and variables in the interface are accessible with direction **inout**, as in the examples above.

13.4.1 An example of a named port bundle

This interface example shows how to use modports to control signal directions as in port declarations. It uses the modport name in the module definition.

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

   modport slave (input req, addr, mode, start, clk,
                  output gnt, rdy,
                  inout data);

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout data);

endinterface: simple_bus

module memMod (simple_bus.slave a); // interface name and modport name
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
      a.gnt <= a.req & avail; // the gnt and req signal in the interface
endmodule

module cpuMod (simple_bus.master b);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   initial repeat(10) #10 clk++;

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

13.4.2 An example of connecting a port bundle

This interface example shows how to use modports to control signal directions. It uses the modport name in the module instantiation.

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

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout data);

endinterface: simple_bus

module memMod(simple_bus a); // Uses just the interface name
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
      a.gnt <= a.req & avail; // the gnt and req signal in the interface
endmodule

module cpuMod(simple_bus b);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   initial repeat(10) #10 clk++;

   memMod mem(sb_intf.slave); // Connect the modport to the module instance
   cpuMod cpu(sb_intf.master);
endmodule
```

13.4.3 An example of connecting a port bundle to a generic interface

This interface example shows how to use modports to control signal directions. It shows the use of the interface keyword in the module definition. The actual interface and modport are specified in the module instantiation.

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

   modport slave (input req, addr, mode, start, clk,
                  output gnt, rdy,
                  inout data);

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout data);

endinterface: simple_bus

module memMod(interface a); // Uses just the interface
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
```

```
      a.gnt <= a.req & avail; // the gnt and req signal in the interface
endmodule

module cpuMod(interface b);
   ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   memMod mem(sb_intf.slave); // Connect the modport to the module instance
   cpuMod cpu(sb_intf.master);
endmodule
```

13.5 Tasks and functions in interfaces

Tasks and functions may be defined within an interface, or they may be defined within one or more of the modules connected. This allows a more abstract level of modeling. For example "read" and "write" can be defined as tasks, without reference to any wires, and the master module can merely call these tasks. In a **`modport`** these tasks are declared as **`import`** tasks.

If the tasks or functions are defined in a module, using a hierarchical name, they must also be declared as **`extern`** in the interface, or as **`export`** in a **`modport`**.

Tasks (not functions) may be defined in a module that is instantiated twice, e.g. two memories driven from the same CPU. Such multiple task definitions are allowed by a **`forkjoin extern`** declaration in the interface.

13.5.1 An example of using tasks in an interface

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

   task masterRead(input logic[7:0] raddr); // masterRead method
      // ...
   endtask: masterRead

   task slaveRead; // slaveRead method
      // ...
   endtask: slaveRead

endinterface: simple_bus

module memMod(interface a); // Uses any interface
   logic avail;

   always @(posedge a.clk) // the clk signal from the interface
      a.gnt <= a.req & avail // the gnt and req signals in the interface

   always @(a.start)
      a.slaveRead;
endmodule
```

```
module cpuMod(interface b);
   enum {read, write} instr;
   logic [7:0] raddr;

   always @(posedge b.clk)
      if (instr == read)
         b.masterRead(raddr); // call the Interface method
      ...
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   memMod mem(sb_intf.slave); // only has access to the slaveRead task
   cpuMod cpu(sb_intf.master); // only has access to the masterRead task
endmodule
```

A function prototype specifies the types and directions of the arguments and the return value of a function which is defined elsewhere. Similarly, a task prototype specifies the types and directions of the arguments of a task which is defined elsewhere. In a modport, the import and export constructs can either use task or function prototypes or use just the identifiers.

13.5.2 An example of using tasks in modports

This interface example shows how to use modports to control signal directions and task access in a full read/write interface.

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

   modport slave (input req, addr, mode, start, clk,
                  output gnt, rdy,
                  inout data,
                  import task slaveRead(),
                         task slaveWrite());
            // import into module that uses the modport

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout data,
                  import task masterRead(input logic[7:0] raddr),
                         task masterWrite(input logic[7:0] waddr));
            // import requires the full task prototype

   task masterRead(input logic[7:0] raddr); // masterRead method
      // ...
   endtask

   task slaveRead; // slaveRead method
      // ...
   endtask

   task masterWrite(input logic[7:0] waddr);
      //...
      endtask

      task slaveWrite;
         //...
      endtask

   endinterface: simple_bus

   module memMod(interface a); // Uses just the interface
      logic avail;

      always @(posedge a.clk) // the clk signal from the interface
         b.gnt <= b.req & avail; // the gnt and req signals in the interface

      always @(a.start)
      if (a.mode[0] == 1'b0)
         a.slaveRead;
      else
         a.slaveWrite;
   endmodule

   module cpuMod(interface b);
      enum {read, write} instr = $rand();
      logic [7:0] raddr = $rand();

      always @(posedge b.clk)
         if (instr == read)
            b.masterRead(raddr); // call the Interface method
         // ...
         else
            b.masterWrite(raddr);
   endmodule

   module omniMod(interface b);
      //...
   endmodule: omniMod

   module top;
      logic clk = 0;

      simple_bus sb_intf(clk); // Instantiate the interface

      memMod mem(sb_intf.slave); // only has access to the slaveRead task
      cpuMod cpu(sb_intf.master); // only has access to the masterRead task
      omniMod omni(sb_intf); // has access to all master and slave tasks
   endmodule
```

13.5.3 An example of exporting tasks and functions

This interface example shows how to define tasks in one module and call them in another, using modports to control task access.

```
   interface simple_bus (input bit clk); // Define the interface
      logic req, gnt;
      logic [7:0] addr, data;
      logic [1:0] mode;
      logic start, rdy;
   modport slave( input req, addr, mode, start, clk,
                  output gnt, rdy,
                  inout data,
                  export task Read(),
                         task Write());
           // export from module that uses the modport

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout data,
                  import task Read(input logic[7:0] raddr),
                         task Write(input logic[7:0] waddr));
           // import requires the full task prototype

endinterface: simple_bus

module memMod(interface a); // Uses just the interface keyword
   logic avail;

   task a.Read; // Read method
      avail = 0;
      ...
      avail = 1;
   endtask

   task a.Write;
      avail = 0;
      ...
      avail = 1;
   endtask
endmodule

module cpuMod(interface b);
   enum {read, write} instr;
   logic [7:0] raddr;

   always @(posedge b.clk)
      if (instr == read)
         b.Read(raddr); // call the slave method via the interface
         ...
      else
         b.Write(raddr);
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   memMod mem(sb_intf.slave); // exports the Read and Write tasks
   cpuMod cpu(sb_intf.master); // imports the Read and Write tasks
endmodule
```

13.5.4 An example of multiple task exports

It is normally an error for more than one module to export the same task name. However, several instances of the same modport type may be connected to an interface, such as memory modules in the previous example. So that these can still export their read and write tasks, the tasks must be declared in the interface using the

extern forkjoin keywords. Normally, only one module responds to the task call, e.g. the one containing the appropriate address. Only then should the task write to the result variables. Note multiple export of functions is not allowed, because they must always write to the result.

This interface example shows how to define tasks in more than one module and call them in another using **extern forkjoin**. The multiple task export mechanism can also be used to count the instances of a particular modport that are connected to each interface instance.

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;
   int slaves;
   // tasks executed concurrently as a fork/join block
   extern forkjoin task countSlaves( );
   extern forkjoin task Read(input logic[7:0] raddr);
   extern forkjoin task Write(input logic[7:0] waddr);

   modport slave( input req, addr, mode, start, clk,
                  output gnt, rdy,
                  inout data,
                  export task Read(),
                         task Write());
         // export from module that uses the modport

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout data,
                  import task Read(input logic[7:0] raddr),
                         task Write(input logic[7:0] waddr));
         // import requires the full task prototype

   initial begin
      slaves = 0;
      countSlaves;
      $display ("number of slaves = %d", slaves);
   end

endinterface: simple_bus

module memMod(interface a); // Uses just the interface keyword
   logic avail;

   task a.countSlaves;
      a.slaves++;
   endtask

   task a.Read; // Read method
      avail = 0;
      ...
      avail = 1;
   endtask

   task a.Write;
      avail = 0;
      ...
      avail = 1;
   endtask
```

```
endmodule

module cpuMod(interface b);
   enum {read, write} instr;
   logic [7:0] raddr;

   always @(posedge b.clk)
      if (instr == read)
         b.Read(raddr); // call the slave method via the interface
         // ...
   else
      b.Write(raddr);
endmodule

module top;
   logic clk = 0;

   simple_bus sb_intf(clk); // Instantiate the interface

   memMod mem1(sb_intf.slave); //exports the countSlaves, Read and Write tasks
   memMod mem2(sb_intf.slave); //exports the countSlaves, Read and Write tasks
   cpuMod cpu(sb_intf.master); //imports the Read and Write tasks
endmodule
```

13.6 Parameterized interfaces

Interface definitions can take advantage of parameters and parameter redefinition, in the same manner as module definitions. This example shows how to use parameters in interface definitions.

```
interface simple_bus #(parameter AWIDTH = 8, DWIDTH = 8;)
                      (input bit clk); // Define the interface
   logic req, gnt;
   logic [AWIDTH-1:0] addr;
   logic [DWIDTH-1:0] data;
   logic [1:0] mode;
   logic start, rdy;

   modport slave( input req, addr, mode, start, clk,
                  output gnt, rdy,
                  inout data,
                  import task slaveRead(),
                         task slaveWrite());
         // import into module that uses the modport

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout data,
                  import task masterRead(input logic[AWIDTH-1:0] raddr),
                         task masterWrite(input logic[AWIDTH-1:0] waddr));
         // import requires the full task prototype

   task masterRead(input logic[AWIDTH-1:0] raddr); // masterRead method
      ...
   endtask

   task slaveRead; // slaveRead method
      ...
   endtask
```

```
      task masterWrite(input logic[AWIDTH-1:0] waddr);
         ...
      endtask

      task slaveWrite;
         ...
      endtask

   endinterface: simple_bus

   module memMod(interface a); // Uses just the interface keyword
      logic avail;

      always @(posedge b.clk) // the clk signal from the interface
         a.gnt <= a.req & avail; //the gnt and req signals in the interface

      always @(b.start)
         if (a.mode[0] == 1'b0)
            a.slaveRead;
         else
            a.slaveWrite;
   endmodule

   module cpuMod(interface b);
      enum {read, write} instr;
      logic [7:0] raddr;

      always @(posedge b.clk)
         if (instr == read)
            b.masterRead(raddr); // call the Interface method
            // ...
         else
            b.masterWrite(raddr);
   endmodule

   module top;

      logic clk = 0;

      simple_bus sb_intf(clk); // Instantiate default interface
      simple_bus #(.DWIDTH(16)) wide_intf(clk); // Interface with 16-bit data

      initial repeat(10) #10 clk++;

      memMod mem(sb_intf.slave); // only has access to the slaveRead task
      cpuMod cpu(sb_intf.master); // only has access to the masterRead task

      memMod memW(wide_intf.slave); // 16-bit wide memory
      cpuMod cpuW(wide_intf.master); // 16-bit wide cpu
   endmodule
```

13.7 Access without Ports

In addition to interfaces being used to connect two or more modules, the interface object/method paradigm allows for interfaces to be instantiated directly as static data objects within a module. If the methods are used to access internal state information about the interface, then these methods may be called from different points in the design to share information.

```
interface intf_mutex;

   task lock ();
      ...
   endtask

   function unlock();
      ...
   endfunction
endinterface

function int f(input int i);
   return(i); // just returns arg
endfunction

function int g(input int i);
   return(i); // just returns arg
endfunction

module mod1(input int in, output int out);

   intf_mutex mutex;

   always begin
      #10 mutex.lock();
      @(in) out = f(in);
      mutex.unlock;
   end

   always begin
      #10 mutex.lock();
      @(in) out = g(in);
      mutex.unlock;
   end
endmodule
```

Section 14 Parameters

14.1 Introduction (informative)

Verilog-2001 provides three constructs for defining compile time constants: the **`parameter`**, **`localparam`** and **`specparam`** statements.

The language provides four methods for setting the value of parameter constants in a design. Each parameter must be assigned a default value when declared. The default value of a parameter of an instantiated module can be overridden in each instance of the module using one of the following:

— Implicit in-line parameter redefinition (e.g. `foo #(value, value) u1 (...);`)

— Explicit in-line parameter redefinition (e.g. `foo #(.name(value), .name(value)) u1 (...);`)

— **`defparam`** statements, using hierarchical path names to redefine each parameter

14.1.1 Defparam removal

The **`defparam`** statement may be removed from future versions of the language. See section 18.2.

14.2 Parameter declaration syntax

```
local_parameter_declaration ::=          // from Annex A.2.1.1
        localparam [ signing ] { packed_dimension } [ range ] list_of_param_assignments ;
      | localparam data_type  list_of_param_assignments ;
parameter_declaration ::=
        parameter [ signing ] { packed_dimension } [ range ] list_of_param_assignments
      | parameter data_type  list_of_param_assignments
      | parameter type  list_of_type_assignments
specparam_declaration ::=
        specparam [ range ] list_of_specparam_assignments ;
list_of_param_assignments ::= param_assignment { , param_assignment }          // from Annex A.2.3
list_of_specparam_assignments ::= specparam_assignment { , specparam_assignment }
list_of_type_assignments ::= type_assignment { , type_assignment }
param_assignment ::= parameter_identifier = constant_param_expression          // from Annex A.2.4
specparam_assignment ::=
        specparam_identifier = constant_mintypmax_expression
      | pulse_control_specparam
type_assignment ::= type_identifier = data_type
```

Syntax 14-1—Parameter declaration syntax (excerpt from Annex A)

A module or an interface can have parameters, which are set during elaboration and are constant during simulation. They are defined with data types and default values. With SystemVerilog, if no data type is supplied, parameters default to type **`logic`** of arbitrary size for Verilog-2001 compatibility and interoperability.

SystemVerilog adds the ability for a parameter to also specify a data type, allowing modules or instances to have data whose type is set for each instance.

```
module ma   #( parameter p1 = 1; parameter type p2 = shortint; )
            (input logic [p1:0] i, output logic [p1:0] o);
   p2 j = 0; // type of j is set by a parameter, which is shortint unless
redefined
   always @(i) begin
      o = i;
      j++;
   end
endmodule

module mb;
   logic [3:0] i,o;
   ma #(.p1(3), .p2(int)) u1(i,o); //redefines p2 to a type of int
endmodule
```

Section 15
Configuration libraries

15.1 Introduction (informative)

Verilog-2001 provides the ability to specify design configurations, which specify the binding information of module instances to specific Verilog HDL source code. Configurations utilize ***libraries***. A library is a collection of modules, primitives and other configurations. Separate ***library map files*** specify the source code location for the cells contained within the libraries. The names of the library map files is typically specified as invocation options to simulators or other software tools reading in Verilog source code.

SystemVerilog adds support for interfaces to Verilog configurations. SystemVerilog also provides an alternate method for specifying the names of library map files.

15.2 Libraries

A library is a named collection of cells. A cell is a module, macromodule, primitive, interface, or configuration. A configuration is a specification of which source files bind to each instance in the design.

15.3 Library map files

Verilog 2001 specifies that library declarations, include statements, and config declarations are normally in a mapping file that is read first by a simulator or other software tool. SystemVerilog does not require a special library map file. Instead, the mapping information can be specified in the **$root** top level.

Section 16
System tasks and system functions

16.1 Introduction (informative)

SystemVerilog adds several system tasks and system functions.

16.2 Expression size system function

```
size_function ::= // not in Annex A
          $bits ( expression )
```

Syntax 16-1—Size function syntax (not in Annex A)

The **`$bits`** system function returns the number of bits required to hold a value. A 4 state value counts as one bit. Given the declaration:

```
logic [31:0] foo;
```

Then `$bits(foo)` will return 32, even if a software tool uses more than 32-bits of storage to represent the 4-state values.

16.3 Array querying system functions

```
array_query_functions ::= // not in Annex A
          array_dimension_function ( array_identifier , dimension_expression )
        | $dimensions ( array_identifier )
array_dimension_function ::=
          $left
        | $right
        | $low
        | $high
        | $increment
        | $length
dimension_expression ::= expression
```

Syntax 16-2—Array querying function syntax (not in Annex A)

SystemVerilog provides new system functions to return information about an array

— **`$left`** shall return the left bound (msb) of the dimension

— **`$right`** shall return the right bound (lsb) of the dimension

— **`$low`** shall return the minimum of **`$left`** and **`$right`** of the dimension

— **`$high`** shall return the maximum of **`$left`** and **`$right`** of the dimension

— **`$increment`** shall return 1 if **`$left`** is greater than or equal to **`$right`**, and -1 if **`$left`** is less than **`$right`**

— **`$length`** shall return the number of elements in the dimension, which is equivalent to **`$high`** - **`$low`** + 1

— **`$dimensions`** shall return the number of dimensions in the array, or 0 for a scalar object

The dimensions of an array shall be numbered as follows: The slowest varying dimension (packed or unpacked) is dimension 1. Successively faster varying dimensions have sequentially higher dimension numbers. For instance:

```
//     Dimension numbers
//    3    4       1    2
reg [3:0][2:1] n [1:5][2:8];
```

For an integer or bit type, only dimension 1 is defined. For an integer N declared without a range specifier, its bounds are assumed to be `[$bits(N)-1:0]`.

If an out-of-range dimension is specified, these functions shall return a logic X.

16.4 Assertion severity system tasks

assert_severity_tasks ::= // not in Annex A
    fatal_message_task
    | nonfatal_message_task

fatal_message_task ::=
    $fatal ;
    | **$fatal (** finish_number [**,** message_argument { **,** message_argument] } **) ;**

nonfatal_message_task ::=
    severity_task **;**
    | severity_task **(** [message_argument { **,** message_argument] } **) ;**

severity_task ::= **$error** | **$warning** | **$info**

finish_number ::= **0** | **1** | **2**

message_argument ::= string | expression

Syntax 16-3—Assertion severity system task syntax (not in Annex A)

SystemVerilog assertions have a severity level associated with any assertion failures detected. By default, the severity of an assertion failure is “error”. The severity levels can be specified by including one of the following severity system tasks in the assertion fail statement:

— **`$fatal`** shall generate a run-time fatal assertion error, which terminates the simulation with an error code. The first argument passed to **`$fatal`** shall be consistent with the corresponding argument to the Verilog **`$finish`** system task, which sets the level of diagnostic information reported by the tool.

— **`$error`** shall be a run-time error.

— **`$warning`** shall be a run-time warning, which can be suppressed in a tool-specific manner.

— **`$info`** shall indicate that the assertion failure carries no specific severity.

All of these severity system tasks shall print a tool-specific message, indicating the severity of the failure, and specific information about the failure, which shall include the following information:

— The file name and line number of the assertion statement,

— The hierarchical name of the assertion, if it is labeled, or the scope of the assertion if it is not labeled.

For simulation tools, these tasks shall also report the simulation run-time at which the severity system task is called.

Each of the severity tasks can include optional user-defined information to be reported. The <user-defined_message> shall use the same syntax as the Verilog `$display` system task, and can include any number of arguments.

16.5 Assertion control system tasks

```
assert_control_tasks ::= // not in Annex A
          assert_task ;
        | assert_task ( levels [ , list_of_modules_or_assertions ] ) ;
assert_task ::=
          $asserton
        | $assertoff
        | $assertkill
list_of_modules_or_assertions ::=
          module_or_assertion { , module_or_assertion }
module_or_assertion ::=
          module_identifier
        | assertion_identifier
        | hierarchical_identifier
```

Syntax 16-4—Assertion control syntax (not in Annex A)

SystemVerilog provides three system tasks to control assertions.

— `$assertoff` shall stop the checking of all specified assertions until a subsequent $asserton. An assertion that is already executing, including execution of the pass or fail statement, is not affected

— `$assertkill` shall abort execution of any currently executing specified assertions and then stop the checking of all specified assertions until a subsequent $asserton.

— `$asserton` shall re-enable the execution of all specified assertions

16.6 Assertion system functions

```
assert_boolean_functions ::= // not in Annex A
          assert_function ( expression ) ;
        | $insetz ( expression, expression [ { , expression } ] ) ;
assert_function ::=
          $onehot
        | $onehot0
        | $inset
        | $isunknown
```

Syntax 16-5—Assertion system function syntax (not in Annex A)

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is “one-hot”. The following system functions are included to facilitate such common assertion functionality:

— **`$onehot`** returns true if one and only one bit of expression is high.

— **`$onehot0`** returns true if at most one bit of expression is low.

— **`$inset`** returns true if the first expression is equal to at least one of the subsequent expression arguments.

— **`$insetz`** returns true if the first expression is equal to at least one other expression argument. Comparison is performed using **`casez`** semantics, so `Z` or `?` bits are treated as don’t-cares.

— **`$isunknown`** returns true if any bit of the expression is `X`. This is equivalent to `^expression === 'bx`.

All of the above system functions shall have a return type of **`bit`**. A return value of `1'b1` shall indicate true, and a return value of `1'b0` shall indicate false.

Section 17
Compiler Directives

17.1 Introduction (informative)

Verilog provides the **`define** text substitution macro compiler directive. A macro can contain arguments, whose values can be set for each instance of the macro. For example:

```
`define NAND(dval) nand #(dval)

`NAND(3)       i1 (y, a, b); //`NAND(3) macro substitutes with: nand #(3)

`NAND(3:4:5)   i2 (o, c, d); //`NAND(3:4:5) macro substitutes with: nand
#(3:4:5)
```

SystemVerilog enhances the capabilities of the **`define** compiler directive to support strings as macro arguments

17.2 ‘define macros

In SystemVerilog, the **`define** macro text can include a backslash (\) at the end of a line to show continuation on the next line.

The macro text can also include an isolated quote, which must be preceded by a back tick, **`"**. This allows macro arguments to be included in strings. If the strings are to contain **\\"**, the macro text should be written **`\\`"**. Otherwise, the backslash will be treated as the start of an escaped identifier.

The macro text can also include a double back tick, **``**, to allow identifiers to be constructed from arguments, e.g.

```
`define foo(f) f``_suffix
```

This expands:

```
foo(bar)
```

to:

```
bar_suffix
```

Note that there must be no space before the parenthesis. Otherwise, it is treated as macro text.

The ‘**include** directive can be followed by a macro, instead of a literal string:

```
`define f1 "/home/foo/myfile"
`include `f1
```

Section 18
Features under consideration for removal from SystemVerilog

18.1 Introduction (informative)

Certain Verilog language features can be simulation inefficient, easily abused, and the source of design problems. These features are being considered for removal from the SystemVerilog language, if there is an alternate method for these features.

The Verilog language features that have been identified in this standard as ones which can be removed from Verilog are **`defparam`** and procedural **`assign`**/**`deassign`**.

18.2 Defparam statements

The SystemVerilog committee has determined, based on the solicitation of input from tool implementers and tools users, that the **`defparam`** method of specifying the value of a parameter can be a source of design errors, and can be an impediment to tool implementation. The **`defparam`** statement does not provide a capability that can not be done by another method, which avoids these problems. Therefore, the committee has placed the **`defparam`** statement on a deprecation list. This means is that a future revision of the Verilog standard may not require support for this feature. This current standard still requires tools to support the **`defparam`** statement. However, users are strongly encouraged to migrate their code to use one of the alternate methods of parameter redefinition.

Prior to the acceptance of the Verilog-2001 Standard, it was common practice to change one or more parameters of instantiated modules using a separate defparam statement. Defparam statements can be a source of tool complexity and design problems.

A **`defparam`** statement can precede the instance to be modified, can follow the instance to be modified, can be at the end of the file that contains the instance to be modified, can be in a separate file from the instance to be modified, can modify parameters hierarchically that in turn must again be passed to other **`defparam`** statements to modify, and can modify the same parameter from two different **`defparam`** statements (with undefined results). Due to the many ways that a **`defparam`** can modify parameters, a Verilog compiler cannot insure the final parameter values for an instance until after all of the design files are compiled.

Prior to Verilog-2001, the only other method available to change the values of parameters on instantiated modules was to use implicit in-line parameter redefinition. This method uses `#(parameter_value)` as part of the module instantiation. Implicit in-line parameter redefinition syntax requires that all parameters up to and including the parameter to be changed must be placed in the correct order, and must be assigned values.

Verilog-2001 introduced explicit in-line parameter redefinition, in the form `#(.parameter_name(value))`, as part of the module instantiation. This method gives the capability to pass parameters by name in the instantiation, which supplies all of the necessary parameter information to the model in the instantiation itself.

The practice of using **`defparam`** statements is highly discouraged. Engineers are encouraged to take advantage of the Verilog-2001 explicit in-line parameter redefinition capability.

See section 14 for more details on parameters.

18.3 Procedural assign and deassign statements

The SystemVerilog committee has determined, based on the solicitation of input from tool implementers and tools users, that the procedural **`assign`** and **`deassign`** statements can be a source of design errors, and can be an impediment to tool implementation. The procedural **`assign`**/**`deassign`** statements do not provide a capability that can not be done by another method, which avoids these problems. Therefore, the committee has placed the procedural **`assign`**/**`deassign`** statements on a deprecation list. This means that a future revision of the Verilog standard may not require support for theses statements. This current standard still requires tools to support the procedural **`assign`**/**`deassign`** statements. However, users are strongly encouraged to migrate their code to use one of the alternate methods of procedural or continuous assignments.

Verilog has two forms of the **`assign`** statement:

— Continuous assignments, placed outside of any procedures

— Procedural continuous assignments, placed within a procedure

Continuous assignment statements are a separate process that are active throughout simulation. The continuous assignment statement accurately represents combinational logic at an RTL level of modeling, and is frequently used.

Procedural continuous assignment statements become active when the **`assign`** statement is executed in the procedure. The process can be de-activated using a **`deassign`** statement. The procedural **`assign`**/**`deassign`** statements are seldom needed to model hardware behavior. In the unusual circumstances where the behavior of procedural continuous assignments are required, the same behavior can be modeled using the procedural force and release statements.

The fact that the **`assign`** statement to be used both outside and inside a procedure can cause confusion and errors in Verilog models. The practice of using the **`assign`** and **`deassign`** statements inside of procedural blocks is highly discouraged.

See section 8 for more information on procedural assignments.

Annex A
Formal Syntax

(Normative)

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

— Keywords and punctuation are in **bold** text.

— Syntactic categories are named in non-bold text.

— A vertical bar (|) separates alternatives.

— Square brackets ([ ]) enclose optional items.

— Braces ({ }) enclose items which may be repeated zero or more times.

The full syntax and semantics of Verilog and SystemVerilog are not described solely using BNF. The normative text description contained within the chapters of the IEEE 1364-2001 Verilog standard and this SystemVerilog document provide additional details on the syntax and semantics described in this BNF.

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 ::= [timeunits_declaration] { description }

description ::=
  module_declaration
  | udp_declaration
  | module_root_item
  | statement

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

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

interface_declaration ::=
  { attribute_instance } **interface** interface_identifier [parameter_port_list]
    [list_of_ports] **;** [timeunits_declaration] { interface_item }
  endinterface [**:** interface_identifier]
  | { attribute_instance } **interface** interface_identifier [parameter_port_list]
    [list_of_port_declarations] **;** [timeunits_declaration] { non_port_interface_item }
  endinterface [**:** interface_identifier]

timeunits_declaration ::=
  timeunit time_literal ;
  | **timeprecision** time_literal ;
  | **timeunit** time_literal ;
  timeprecision time_literal ;
  | **timeprecision** time_literal ;
  timeunit time_literal ;

A.1.4 Module parameters and ports

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
  | { attribute_instance } interface_port_declaration

A.1.5 Module items

```
module_common_item ::=
          { attribute_instance } module_or_generate_item_declaration
        | { attribute_instance } interface_instantiation

module_item ::=
          port_declaration ;
        | non_port_module_item

module_or_generate_item ::=
          { 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
        | { attribute_instance } combinational_statement
        | { attribute_instance } latch_statement
        | { attribute_instance } ff_statement
        | module_common_item

module_root_item ::=
          { attribute_instance } module_instantiation
        | { attribute_instance } local_parameter_declaration
        | interface_declaration
        | module_common_item

module_or_generate_item_declaration ::=
          net_declaration
        | data_declaration
        | event_declaration
        | genvar_declaration
        | task_declaration
        | function_declaration

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

parameter_override ::= defparam list_of_param_assignments ;
```

A.1.6 Interface items

```
interface_or_generate_item ::=
          { attribute_instance } continuous_assign
        | { attribute_instance } initial_construct
        | { attribute_instance } always_construct
        | { attribute_instance } combinational_statement
        | { attribute_instance } latch_statement
        | { attribute_instance } ff_statement
        | { attribute_instance } local_parameter_declaration
        | { attribute_instance } parameter_declaration ;
        | module_common_item
```

| { attribute_instance } modport_declaration

interface_item ::=
port_declaration
| non_port_interface_item

non_port_interface_item ::=
{ attribute_instance } generated_interface_instantiation
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**
| { attribute_instance } specparam_declaration
| interface_or_generate_item
| interface_declaration

A.2 Declarations

A.2.1 Declaration types

A.2.1.1 Module parameter declarations

local_parameter_declaration ::=
localparam [signing] { packed_dimension } [range] list_of_param_assignments **;**
| **localparam** data_type list_of_param_assignments **;**

parameter_declaration ::=
parameter [signing] { packed_dimension } [range] list_of_param_assignments
| **parameter** data_type list_of_param_assignments
| **parameter** type list_of_type_assignments

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

A.2.1.2 Port declarations

inout_declaration ::= **inout** [port_type] list_of_port_identifiers

input_declaration ::= **input** [port_type] list_of_port_identifiers

output_declaration ::=
output [port_type] list_of_port_identifiers
| output data_type list_of_variable_port_identifiers

interface_port_declaration ::=
interface list_of_interface_identifiers
| **interface .** modport_identifier list_of_interface_identifiers
| **identifier** list_of_interface_identifiers
| **identifier .** modport_identifier list_of_interface_identifiers

A.2.1.3 Type declarations

block_data_declaration ::=
block_variable_declaration
| constant_declaration
| type_declaration

constant_declaration ::= **const** data_type const_assignment **;**

data_declaration ::=
variable_declaration
| constant_declaration
| type_declaration

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

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

net_declaration ::=
  net_type [signing]
    [delay3] list_of_net_identifiers **;**
  | net_type [drive_strength] [signing]
    [delay3] list_of_net_decl_assignments **;**
  | net_type [vectored | scalared] [signing]
    { packed_dimension } range [delay3] list_of_net_identifiers **;**
  | net_type [drive_strength] [vectored | scalared] [signing]
    { packed_dimension } range [delay3] list_of_net_decl_assignments **;**
  | **trireg** [charge_strength] [signing]
    [delay3] list_of_net_identifiers **;**
  | **trireg** [drive_strength] [signing]
    [delay3] list_of_net_decl_assignments **;**
  | **trireg** [charge_strength] [vectored | scalared] [signing]
    { packed_dimension } range [delay3] list_of_net_identifiers **;**
  | **trireg** [drive_strength] [vectored | scalared] [signing]
    { packed_dimension } range [delay3] list_of_net_decl_assignments **;**

type_declaration ::=
  typedef data_type type_declaration_identifier **;**
  | **typedef** interface_identifier { [constant_expression] } **.** type_identifier
    type_declaration_identifier **;**

block_variable_declaration ::=
  [lifetime] data_type list_of_variable_identifiers **;**
  | lifetime data_type list_of_variable_decl_assignments **;**

variable_declaration ::=
  [lifetime] data_type list_of_variable_identifiers_or_assignments **;**

lifetime ::= **static** | **automatic**

A.2.2 Declaration data types

A.2.2.1 Net and variable types

data_type ::=
  integer_vector_type [signing] { packed_dimension } [range]
  | integer_atom_type [signing] { packed_dimension }
  | type_declaration_identifier
  | non_integer_type
  | **struct** [**packed**] [signing] **{** { struct_union_member } **}**
  | **union** [**packed**] [signing] **{** { struct_union_member } **}**
  | **enum** [integer_type [signing] { packed_dimension }]
    { enum_identifier [**=** constant_expression] { **,** enum_identifier [**=** constant_expression] } **}**
  | **void**

integer_type ::= integer_vector_type | integer_atom_type

integer_atom_type ::= **byte** | **char** | **shortint** | **int** | **longint** | **integer**

integer_vector_type ::= **bit** | **logic** | **reg**

non_integer_type ::= **time** | **shortreal** | **real** | **realtime** | $built-in

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

port_type ::=
  data_type { packed_dimension }
  | net_type [signing] { packed_dimension }
  | **trireg** [signing] { packed_dimension }
  | **event**
  | [signing] { packed_dimension } range

signing ::= [**signed**] | [**unsigned**]

simple_type_or_number ::= simple_type | number

simple_type ::= integer_type | non_integer_type | type_identifier

struct_union_member ::= data_type list_of_variable_identifiers_or_assignments **;**

A.2.2.2 Strengths

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

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

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

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

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 [unpacked_dimension { unpacked_dimension }]
  { **,** event_identifier [unpacked_dimension { unpacked_dimension }] }

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

list_of_interface_identifiers ::= interface_identifier { unpacked_dimension }
  { **,** interface_identifier { unpacked_dimension } }

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

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

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

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

list_of_udp_port_identifiers ::= port_identifier { **,** port_identifier }

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

list_of_type_assignments ::= type_assignment { **,** type_assignment }

list_of_variable_decl_assignments ::= variable_decl_assign_identifier { **,** variable_decl_assign_identifier }

list_of_variable_identifiers ::= variable_declaration_identifier { **,** variable_declaration_identifier }

list_of_variable_identifiers_or_assignments ::=
  list_of_variable_decl_assignments
  | list_of_variable_identifiers

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

A.2.4 Declaration assignments

const_assignment ::= const_identifier **=** constant_expression

net_decl_assignment ::= net_identifier **=** expression

param_assignment ::= parameter_identifier **=** constant_param_expression

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

type_assignment ::= type_identifier **=** data_type

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

error_limit_value ::= limit_value

reject_limit_value ::= limit_value

limit_value ::= constant_mintypmax_expression

A.2.5 Declaration ranges

unpacked_dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**

packed_dimension ::= **[** dimension_constant_expression **:** dimension_constant_expression **]**

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

A.2.6 Function declarations

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

function_item_declaration ::=
    block_item_declaration
  | { attribute_instance } input_declaration **;**
  | { attribute_instance } output_declaration **;**
  | { attribute_instance } inout_declaration **;**

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

function_port_list ::= function_port_item { **,** function_port_item }

function_prototype ::= **function** data_type **(** list_of_function_proto_formals **)**

named_function_proto::= **function** data_type function_identifier **(** list_of_function_proto_formals **)**

list_of_function_proto_formals ::=
    [{ attribute_instance } function_proto_formal { **,** { attribute_instance } function_proto_formal }]

function_proto_formal ::=
    input data_type [variable_declaration_identifier]
  | **inout** data_type [variable_declaration_identifier]

| **output** data_type [variable_declaration_identifier]
| variable_declaration_identifier

range_or_type ::=
{ packed_dimension } range
| data_type

A.2.7 Task declarations

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

task_item_declaration ::=
block_item_declaration
| { attribute_instance } input_declaration **;**
| { attribute_instance } output_declaration **;**
| { attribute_instance } inout_declaration **;**

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

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

task_prototype ::=
task ({ attribute_instance } task_proto_formal { **,** { attribute_instance } task_proto_formal } **)**

named_task_proto ::= **task** task_identifier **(** task_proto_formal { **,** task_proto_formal } **)**

task_proto_formal ::=
input data_type [variable_declaration_identifier]
| **inout** data_type [variable_declaration_identifier]
| **output** data_type [variable_declaration_identifier]

A.2.8 Block item declarations

block_item_declaration ::=
{ attribute_instance } block_data_declaration
| { attribute_instance } event_declaration
| { attribute_instance } local_parameter_declaration
| { attribute_instance } parameter_declaration **;**

A.2.9 Interface declarations

modport_declaration ::= modport list_of_modport_identifiers **;**

list_of_modport_identifiers ::= modport_item { **,** modport_item }

modport_item ::= modport_identifier **(** modport_port { **,** modport_port } **)**

modport_port ::=
input [port_type] port_identifier
| **output** [port_type] port_identifier
| **inout** [port_type] port_identifier
| interface_identifier **.** port_identifier
| import_export **task** named_task_proto
| import_export **function** named_fn_proto

| import_export task_or_function_identifier { **,** task_or_function_identifier }

import_export ::= **import** | **export**

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 [delay2] 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, interface and generated instantiation

A.4.1 Instantiation

A.4.1.1 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 | data_type

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

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 }
  | dot_named_port_connection { **,** dot_named_port_connection }
  | { named_port_connection **,** } dot_star_port_connection { **,** named_port_connection }

ordered_port_connection ::= { attribute_instance } [expression]

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

dot_named_port_connection ::=
    { attribute_instance } **.**port_identifier
  | named_port_connection

dot_star_port_connection ::= { attribute_instance } **.***

A.4.1.2 Interface instantiation

interface_instantiation ::=
    interface_identifier [parameter_value_assignment] module_instance { **,** module_instance } **;**

A.4.2 Generated instantiation

A.4.2.1 Generated module instantiation

generated_module_instantiation ::= **generate** { generate_module_item } **endgenerate**

generate_module_item_or_null ::= generate_module_item | **;**

generate_module_item ::=
    generate_module_conditional_statement
  | generate_module_case_statement
  | generate_module_loop_statement
  | [generate_block_identifier :] generate_module_block
  | module_or_generate_item

generate_module_conditional_statement ::=
    if (constant_expression) generate_module_item_or_null [**else** generate_module_item_or_null]

generate_module_case_statement ::=
    case (constant_expression **)** genvar_module_case_item { genvar_module_case_item }**endcase**

genvar_module_case_item ::=
    constant_expression { **,** constant_expression } **:** generate_module_item_or_null
  | **default** [**:**] generate_module_item_or_null

generate_module_loop_statement ::=
    for **(** genvar_decl_assignment **;** constant_expression **;** genvar_assignment **)**
        generate_module_named_block

genvar_assignment ::=
    genvar_identifier **=** constant_expression
  | genvar_identifier assignment_operator constant_expression
  | inc_or_dec_operator genvar_identifier
  | genvar_identifier inc_or_dec_operator

genvar_decl_assignment ::=
    [**genvar**] genvar_identifier **=** constant_expression

generate_module_named_block ::=
    begin : generate_block_identifier { generate_module_item } **end** [: generate_block_identifier]
  | generate_block_identifier **:** generate_module_block

generate_module_block ::=
    begin [**:** generate_block_identifier] { generate_module_item } **end** [**:** generate_block_identifier]

A.4.2.2 Generated interface instantiation

generated_interface_instantiation ::= **generate** { generate_interface_item } **endgenerate**

generate_interface_item_or_null ::= generate_interface_item | **;**

generate_interface_item ::=
    generate_interface_conditional_statement
  | generate_interface_case_statement
  | generate_interface_loop_statement
  | [generate_block_identifier :] generate_interface_block
  | interface_or_generate_item

generate_interface_conditional_statement ::=
    if (constant_expression) generate_interface_item_or_null [**else** generate_interface_item_or_null]

generate_interface_case_statement ::=
    case (constant_expression) genvar_interface_case_item { genvar_interface_case_item } **endcase**

genvar_interface_case_item ::=
    constant_expression { **,** constant_expression } **:** generate_interface_item_or_null
  | **default** [**:**] generate_interface_item_or_null

generate_interface_loop_statement ::=
    for (genvar_decl_assignment **;** constant_expression **;** genvar_assignment **)**
        generate_interface_named_block

generate_interface_named_block ::=
    begin : generate_block_identifier { generate_interface_item } **end** [: generate_block_identifier]
  | generate_block_identifier **:** generate_interface_block

generate_interface_block ::=
    begin [**:** generate_block_identifier]
    { generate_interface_item }
    end [**:** generate_block_identifier]

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_udp_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] { range } **(** 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

combinational_statement ::= always_comb statement

latch_statement ::= always_latch statement

ff_statement ::= always_ff statement

blocking_assignment ::=
    variable_lvalue **=** delay_or_event_control expression
  | operator_assignment

operator_assignment ::= variable_lvalue assignment_operator expression

assignment_operator ::=
    = | **+=** | **-=** | ***=** | **/=** | **%=** | **&=** | **|=** | **^=** | **<<=** | **>>=** | **<<<=** | **>>>=**

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

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

function_blocking_assignment ::= variable_lvalue **=** expression

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

variable_assignment ::= variable_lvalue **=** expression

A.6.3 Parallel and sequential blocks

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

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

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

A.6.4 Statements

statement ::= [block_identifier **:**] statement_item

statement_item ::=
    { attribute_instance } blocking_assignment **;**
  | { attribute_instance } nonblocking_assignment **;**
  | { attribute_instance } procedural_continuous_assignments **;**
  | { attribute_instance } case_statement
  | { attribute_instance } conditional_statement
  | { attribute_instance } inc_or_dec_expression
  | { attribute_instance } function_call[7]
  | { attribute_instance } disable_statement
  | { attribute_instance } event_trigger
  | { attribute_instance } loop_statement

| { attribute_instance } jump_statement
| { attribute_instance } par_block
| { attribute_instance } procedural_timing_control_statement
| { attribute_instance } seq_block
| { attribute_instance } system_task_enable
| { attribute_instance } task_enable
| { attribute_instance } wait_statement
| { attribute_instance } process statement
| { attribute_instance } proc_assertion

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

function_statement ::= [block_identifier **:**] function_statement_item

function_statement_item ::=
{ attribute_instance } function_blocking_assignment **;**
| { attribute_instance } function_case_statement
| { attribute_instance } function_conditional_statement
| { attribute_instance } inc_or_dec_expression
| { attribute_instance } function_call[7]
| { attribute_instance } function_loop_statement
| { attribute_instance } jump_statement
| { attribute_instance } function_seq_block
| { attribute_instance } disable_statement
| { attribute_instance } system_task_enable

A.6.5 Timing control statements

procedural_timing_control_statement ::=
delay_or_event_control statement_or_null

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

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

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

event_expression ::=
expression [**iff** expression]
| hierarchical_identifier [**iff** expression]
| [edge] expression [**iff** expression]
| event_expression **or** event_expression
| event_expression **,** event_expression

edge ::= **posedge** | **negedge** | **changed**

jump_statement ::=
return [expression] **;**
| **break ;**
| **continue ;**

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

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

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

A.6.6 Conditional statements

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

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

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

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

unique_priority ::= **unique** | **priority**

A.6.7 Case statements

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

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

function_case_statement ::=
    [unique_priority] **case (** expression **)** function_case_item { function_case_item } **endcase**
  | [unique_priority] **casez (** expression **)** function_case_item { function_case_item } **endcase**
  | [unique_priority] **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_or_null
  | **while (** expression **)** function_statement_or_null
  | **for (** variable_decl_or_assignment **;** expression **;** variable_assignment **)**
        function_statement_or_null
  | **do** function_statement **while (** expression **)**

loop_statement ::=
    forever statement
  | **repeat (** expression **)** statement_or_null

| **while (** expression **)** statement_or_null
| **for (** variable_decl_or_assignment **;** expression **;** variable_assignment **)** statement_or_null
| **do** statement **while (** expression **)**

variable_decl_or_assignment ::=
data_type list_of_variable_identifiers_or_assignments **;**
| variable_assignment

A.6.9 Task enable statements

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

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

A.6.10 Assertion statements

proc_assertion ::=
immediate_assert
| strobed_assert
| clocked_immediate_assert
| clocked_strobed_assert

immediate_assert ::= **assert (** expression **)**
statement_or_null
[**else** statement_or_null]

strobed_assert ::= **assert_strobe (** expression **)**
restricted_statement_or_null
[**else** restricted_statement_or_null]

clocked_immediate_assert ::= **assert (** expr_sequence **)** step_control
statement_or_null
[**else** statement_or_null]

clocked_strobed_assert ::= **assert_strobe (** expr_sequence **)** step_control
restricted_statement_or_null
[**else** restricted_statement_or_null]

restricted_statement_or_null ::=
restricted_statement
| { attribute_instance } **;**

restricted_statement ::=
[block_identifier **:**] restricted_statement_item

restricted_statement_item ::=
{ attribute_instance } proc_assertion
| { attribute_instance } system_task_enable
| { attribute_instance } delay_or_event_control statement
| { attribute_instance } restricted_seq_block

restricted_seq_block ::= **begin** [**:** block_identifier] { block_item_declaration }{ restricted_statement }
end [**:** block_identifier]

expr_sequence ::=
expression
| [constant_expression]
| range
| expr_sequence **;** expr_sequence
| expr_sequence * [constant_expression]
| expr_sequence * range
| **(** expr_sequence **)**

step_control ::=
@@ event_identifier

| **@@ (** event_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_outputs **;**
| **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_descriptor1 ::= **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

inc_or_dec_expression ::=
  inc_or_dec_operator variable_lvalue
  | variable_lvalue inc_or_dec_operator

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

constant_base_expression ::= constant_expression

constant_expression ::=
  constant_primary
  | unary_operator { attribute_instance } constant_primary
  | constant_expression binary_operator { attribute_instance } constant_expression
  | constant_expression **?** { attribute_instance } constant_expression **:** constant_expression
  | string

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

constant_param_expression ::=
  constant_expression
  | data_type

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
  | { attribute_instance } inc_or_dec_expression
  | **(** operator_assignment **)**
  | 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
  | time_literal
  | **’0** | **’1** | **’z** | **’Z** | **’x** | **’X**

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 **)**
| **{** expression { , expression } **}**
| **{** expression **{** expression **} }**
| simple_type_or_number **’ (** expression **)**
| simple_type_or_number **’ {** expression { , expression } **}**
| simple_type_or_number **’ {** expression **{** expression **} }**
| time_literal
| **’0** | **’1** | **’z** | **’Z** | **’x** | **’X**

time_literal ::=
unsigned_number time_unit
| fixed_point_number time_unit

time_unit ::= **s** | **ms** | **us** | **ns** | **ps** | **fs**

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 **]**
| hierarchical_net_identifier **(** [constant_expression { **,** constant_expression }] **)**
| net_concatenation

variable_lvalue ::=
variable_lvalue_item [inc_or_dec_operator]
| hierarchical_variable_identifier **(** [constant_expression { **,** constant_expression }] **)**

variable_lvalue_item ::=
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 ::=
+ | - | * | / | % | == | != | === | !== | && | || | **

| < | <= | > | >= | **&** | | | ^ | ^~ | ~^ | >> | << | >>> | <<<

inc_or_dec_operator ::= **++** | **--**

unary_module_path_operator ::=

! | ~ | **&** | **~&** | | | ~| | ^ | ~^ | ^~

binary_module_path_operator ::=

== | **!=** | **&&** | **||** | **&** | | | ^ | ^~ | ~^

A.8.7 Numbers

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

decimal_number ::=
unsigned_number
| [size] decimal_base unsigned_number
| [size] decimal_base x_digit { **_** }
| [size] decimal_base z_digit { **_** }

binary_number ::= [size] binary_base binary_value

octal_number ::= [size] octal_base octal_value

hex_number ::= [size] hex_base hex_value

sign ::= **+** | **-**

size ::= non_zero_unsigned_number

non_zero_unsigned_number[1] ::= non_zero_decimal_digit { **_** | decimal_digit}

real_number[1] ::=
fixed_point_number
| unsigned_number [**.** unsigned_number] exp [sign] unsigned_number

fixed_point_number[1] ::= unsigned_number **.** unsigned_number

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

unsigned_number[1] ::= decimal_digit { **_** | decimal_digit }

binary_value[1] ::= binary_digit { **_** | binary_digit }

octal_value[1] ::= octal_digit { **_** | octal_digit }

hex_value[1] ::= hex_digit { **_** | hex_digit }

decimal_base[1] ::= **'**[**s**|**S**]**d** | **'**[**s**|**S**]**D**

binary_base[1] ::= **'**[**s**|**S**]**b** | **'**[**s**|**S**]**B**

octal_base[1] ::= **'**[**s**|**S**]**o** | **'**[**s**|**S**]**O**

hex_base[1] ::= **'**[**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

const_identifier ::= identifier

enum_identifier ::= identifier

escaped_arrayed_identifier ::= escaped_identifier **[** range **]**

escaped_hierarchical_identifier[4] ::=
    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[8]

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

interface_identifier ::= identifier

inout_port_identifier ::= identifier

input_port_identifier ::= identifier

instance_identifier ::= identifier

library_identifier ::= identifier

memory_identifier ::= identifier

modport_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[3] ::= simple_hierarchical_branch [**.**escaped_identifier]

simple_identifier[2] ::= [**a-zA-Z_**] { [**a-zA-Z0-9_$**] }

specparam_identifier ::= identifier

state_identifier ::= identifier

system_function_identifier[5] ::= **$**[a-zA-Z0-9_$]{ [a-zA-Z0-9_$] }

system_task_identifier[5] ::= $[**a-zA-Z0-9_$**]{ [**a-zA-Z0-9_$**] }

task_or_function_identifier ::= task_identifier | function_identifier

task_identifier ::= identifier

terminal_identifier ::= identifier

text_macro_identifier ::= simple_identifier

topmodule_identifier ::= identifier

type_declaration_identifier ::= type_identifier { packed_dimension }

type_identifier ::= identifier

udp_identifier ::= identifier

udp_instance_identifier ::= arrayed_identifier

variable_decl_assign_identifier ::= variable_identifier { unpacked_dimension } [**=** constant_expression]

variable_declaration_identifier ::= variable_identifier { unpacked_dimension }

variable_identifier ::= identifier

A.9.4 Identifier branches

simple_hierarchical_branch[3] ::=
    simple_identifier { **[** unsigned_number **]** } [{ **.** simple_identifier { **[** unsigned_number **]** } }]

escaped_hierarchical_branch[4] ::=
    escaped_identifier { **[** unsigned_number **]** } [{ **.** escaped_identifier { **[** unsigned_number **]** } }]

A.9.5 White space

white_space ::= space | tab | newline | eof[6]

NOTES

1) Embedded spaces are illegal.

2) A simple_identifier and arrayed_reference shall start with an alpha or underscore (_) character, shall have at least one character, and shall not have any spaces.

3) The period (.) in simple_hierarchical_identifier and simple_hierarchical_branch shall not be preceded or followed by white_space.

4) The period in escaped_hierarchical_identifier and escaped_hierarchical_branch shall be preceded by white_space, but shall not be followed by white_space.

5) The $ character in a system_function_identifier or system_task_identifier shall not be followed by white_space. A system_function_identifier or system_task_identifier shall not be escaped.

6) End of file.

7) Must be a void function

8) Hierarchy is not allowed

Annex B
Keywords

SystemVerilog reserves the following keywords:

always	endtable	nand	small
always_comb†	endtask	negedge	specify
always_ff†	**endtransition**†	nmos	specparam
always_latch†	**enum**†	nor	**static**†
and	event	noshowcancelled	strong0
assert†	**export**†	not	strong1
assert_strobe†	**extern**†	notif0	**struct**†
assign	for	notif1	supply0
automatic	force	or	supply1
begin	forever	output	table
bit†	fork	**packed**†	task
break†	**forkjoin**†	parameter	time
buf	function	pmos	**timeprecision**†
bufif0	generate	posedge	**timeunit**†
bufif1	genvar	primitive	tran
byte†	highz0	**process**†	tranif0
case	highz1	**priority**†	tranif1
casex	if	pull0	**transition**†
casez	**iff**†	pull1	tri
cell	ifnone	pulldown	tri0
changed†	**import**†	pullup	tri1
char†	incdir	pulsestyle_oneven	triand
cmos	include	t	trior
config	initial	pulsestyle_ondete	trireg
const†	inout	ct	**type**†
continue†	input	rcmos	**typedef**†
deassign	instance	real	**union**†
default	**int**†	realtime	**unique**†
defparam	integer	reg	unsigned
design	**interface**†	release	use
disable	join	repeat	vectored
do†	large	return	**void**†
else	liblist	rnmos	wait
end	library	rpmos	wand
endcase	localparam	rtran	weak0
endconfig	**logic**†	rtranif0	weak1
endfunction	**longint**†	rtranif1	while
endgenerate	**longreal**†	scalared	wire
endinterface†	macromodule	**shortint**†	wor
endmodule	medium	**shortreal**†	xnor
endprimitive	**modport**†	showcancelled	xor
endspecify	module	signed	

† keywords not in the IEEE 1364 Verilog-2001 standard

Annex C Glossary

(Informative)

Assertion — An assertion is a statement that a certain property must be true. For example, that a read_request must always be followed by a read_grant within 2 clock cycles. Assertions allow for automated checking that the specified property is true, and can generate automatic error messages if the property is not true. SystemVerilog provides special assertion constructs, which are discussed in Section 11.

Elaboration — Elaboration is the process of binding together the components that make up a design. These components can include module instances, primitive instances, interfaces, and the top-level of the design hierarchy. SystemVerilog requires a specific order of elaboration, which is presented in Section 12.2.

Enumerated type — Enumerated data types provide the capability to declare a variable which can have one of a set of named values. The numerical equivalents of these values may be specified. Enumerated types can be easily referenced or displayed using the enumerated names, as opposed to the enumerated values. Section 3.6 discusses enumerated types.

Interface — An interface encapsulates the communication between blocks of a design, allowing a smooth migration from abstract system-level design through successive refinement down to lower-level register-transfer and structural views of the design. By encapsulating the communication between blocks, the interface construct also facilitates design re-use. The inclusion of interface capabilities is one of the major advantages of SystemVerilog. Interfaces are covered in Section 13.

LRM — LRM is an abbreviation for Language Reference Manual. “SystemVerilog LRM” refers to this document. “Verilog LRM” refers to the IEEE manual “1364-2001 IEEE Standard for Verilog Hardware Description Language 2001”. See Annex D for information about this manual.

Packed array — Packed array refers to an array where the dimensions are declared before an object name. Packed arrays can have any number of dimensions. A one-dimensional packed array is the same as a vector width declaration in Verilog. Packed arrays provide a mechanism for subdividing a vector into subfields, which can be conveniently accessed as array elements. A packed array differs from an unpacked array, in that the whole array is treated as a single vector for arithmetic operations. Packed arrays are discussed in detail in Section 4.

Process — A process is a thread of one or more programming statements which can be executed independently of other programming statements. Each initial procedure, always procedure and continuous assignment statement in Verilog is a separate process. These are static processes. That is, each time the process starts running, there is an end to the process. SystemVerilog adds specialized always procedures, which are also static processes, and dynamic processes, introduced by the process keyword. When dynamic processes are started, they can run without ending. Processes are presented in Section 9.

SystemVerilog — SystemVerilog refers to the Accellera standard for a set of abstract modeling and verification extensions to the IEEE 1364-2001 Verilog standard. The many features of the SystemVerilog standard are presented in this document.

Unpacked array — Unpacked array refers to an array where the dimensions are declared after an object name. Unpacked arrays are the same as arrays in Verilog, and can have any number of dimensions. An unpacked array differs from a packed array, in that the whole array cannot be used for arithmetic operations. Each element must be treated separately. Unpacked arrays are discussed in Section 4.

Verilog — Verilog refers to the IEEE 1364-2001 Verilog Hardware Description Language (HDL), commonly called Verilog-2001. This language is documented in the IEEE manual “1364-2001 IEEE Standard for Verilog Hardware Description Language 2001”. See Annex D for information about this manual.

Annex D
Bibliography

(Informative)

[B1] IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic 1985. ISBN 1-5593-7653-8. IEEE Product No. SH10116-TBR.

[B2] IEEE Std. 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog¨ Hardware Description Language 1995. ISBN 0-7381-3065-6. IEEE Product No. WE94418-TBR.

[B3] IEEE Std. 1364-2001, IEEE Standard for Verilog Hardware Description Language 2001. ISBN 0-7381-2827-9. IEEE Product No. SH94921-TBR.

Index

accellera