

5/31/2003

## **Proposal for Extending Verilog Data Types**

This proposal has been prepared by Cadence Design Systems, Inc. for consideration by the IEEE 1364 working group for inclusion in the next revision of the IEEE 1364 standard.

## Contents

1	Overview.....	1
1.1	Scope .....	1
1.2	Purpose .....	1
1.3	Conventions .....	2
2	Types .....	3
2.1	Primitive types .....	3
2.2	Type declarations .....	4
2.3	User-defined types .....	6
2.4	Vector types .....	7
2.5	Array types .....	12
2.6	Structure types .....	13
2.7	Union types .....	14
2.8	Handle types .....	15
2.9	Incomplete types .....	16
2.10	Predefined type names .....	16
2.11	Type compatibility .....	17
3	Objects .....	19
3.1	Data object declarations .....	19
3.2	Variables .....	20
3.3	Nets .....	21
3.4	Parameters .....	26
3.5	Ports .....	28
3.6	Task and function arguments .....	34
4	Expressions .....	35
4.1	Boolean literals .....	35
4.2	Vector literals .....	35

4.3	Concatenations .....	35
4.3	Operators .....	36
5	Assignment .....	38
5.1	Assignment compatibility .....	38
5.2	Value conversions.....	39
6	Dynamic objects .....	40
6.1	Memory allocation .....	40
6.2	Null values .....	41
6.3	Initialization .....	41
6.4	Operators .....	41
Annex A:	Syntax summary .....	47
Annex B:	Detailed examples.....	52
Annex C:	Possible extensions .....	57
Annex D:	Rationale .....	58
Annex E:	References .....	60



# 1 Overview

## 1.1 Scope

- [1] This proposal extends Verilog 1364-2001 data types and objects. The existing concept of variable and net objects is enhanced to create a type system that is orthogonal to the simulation semantic of the object itself. The type system is extended beyond that of the current language definition by adding user-defined types for enumerations, multi-dimensional arrays, structures, and dynamically allocated objects. These type extensions allow variables, nets, parameters, ports, and arguments to tasks and functions to be of any type. The operators and interconnect behavior for all types and object kinds are specified.

## 1.2 Purpose

- [2] The proposed extensions support the modeling of system-level components and refinement from system-level to implementation. In order to achieve a higher level of abstraction in Verilog models, user-defined types are proposed. These types allow multi-dimensional arrays and structure declarations. Composite objects can be created that allow a single name to refer to multiple objects. These objects can be manipulated collectively and passed between modules in system-level models; this simplifies both the model and port interconnect.
- [3] In order to allow composite objects to be treated both as variables and nets, this proposal makes a distinction between the kind of an object and the type of an object. The *kind* of an object establishes its simulation semantics. The *type* of an object specifies the set of values that the object can hold. In IEEE 1364-2001, the standard was modified to consistently refer to variables rather than registers. This proposal continues this cleanup. It uses *net* to refer to an object kind that has the semantics of a **net**, as defined in IEEE 1364-2001, and *variable* as an object kind that has the semantics of a **reg**. The declarations of nets and variables have been extended to allow an optional type qualifier so that the value can be any pre-defined or user-defined type.
- [4] In the past, extensions to the type system, such as **integer**, **real**, and the Verilog-AMS **wreal**, have been added to make models easier to express, but they were limited to extending the value set held by a variable or a net, not both. As a result, each new type added to the language required a new keyword as a variable, and if desired, a new keyword as a net. This can be seen in the current definition of a **real** type for registers and a **wreal** type for wires. There has been no similar extension to date for the **integer** type to create a wire that passes an integer value; instead a wire must be declared with the same width as the implementation's **integer** type (typically [31:0]). the integer is divided into 32 individual bits that are each passed across individual nets. Adding such an integer-valued net type would require a new predefined type of net, for instance **winteger** might be used. Continuing this method of type definition would extend the keywords linearly as the type system required growth. This linear expansion of keywords for every new type is not a sustainable way to extend the language or type system over the long term.
- [5] The lack of a user-defined type also severely limits system-level design and creation of data structures. In system-level design, it is desirable to manipulate an entire data structure that may represent a network packet or an instruction as a single object. The language standardization cannot anticipate all required types for system-level designs and therefore must include user-defined types. A similar argument can be made for allowing users to create types and functions/tasks that implement traditional data structures. It is tempting to define pre-defined types for such items as lists, queues, and stacks just as Verilog '87 pre-defined 4-valued logic and primitive gates. However, in hardware description languages, these data type constructs may represent real hardware, and the operations on them may be specific to the hardware (for example, handling of overflow, underflow, statistics gathering, or performance monitoring). In a system-level description it is not possible to anticipate all the uses, behavioral idiosyncrasies, or implementations of the underlying structure, therefore allowing the user to model them explicitly is required.<sup>1</sup>

- [6] The creation of a robust type system has the additional advantage that future additions of types can be made and easily applied to all object kinds. This may be necessary for interaction with other languages such as 'C', assertion languages, or hardware verification languages. The creation of objects kinds of that type will already be defined. In the current Verilog type system, a new predefined type is always added as either a variable (as in **integer**) or as a net (as in the AMS **wreal**). Adding a new type that can exist as both a variable and a net cannot be done today without two new keywords and detailed explanation of how they interact.
- [7] The types and keywords **reg** and **wire** are preserved in a manner that is backward compatible with the IEEE 1364-2001 Verilog standard.
- [8] The existing variable types **real**, **integer**, **time**, and **event** are redefined in terms of the new type system in a 100% backward compatible fashion but in such a way as to allow their values to be placed on a net.
- [9] A new two-valued logic type, **bool** is introduced. This type eliminates the need to model the 'x' and 'z' states and can create simpler models. Simulation vendors may be able to take advantage of the **bool** type to optimize both performance and capacity of designs.
- [10] A new type, **handle**, is introduced. Handles are used to reference dynamically allocated objects. Dynamically allocated objects are useful for writing system-level models and testbenches. Objects that are referenced by handles are explicitly allocated through the **new** operator. Deallocation is automatic through garbage collection.
- [11] The type system is extended to allow dynamically allocated objects that can interoperate with the static objects in a design. The dynamic objects can be used to model data structures of indeterminate length or size such as lists, queues, stacks, and sparse memories.
- [12] A new wire type, **wone**, is introduced to provide a means of restricting a net to a single driver.
- [13] The remainder of this donation is organized into sections that define data type declarations, data object declarations, expressions, and assignment.
- [14] In introducing the concept of a user-defined type, this donation makes substantial changes to IEEE 1364-2001 Section 3. Because of the pervasive nature of the changes, this donation makes a complete proposal rather than simply suggesting a set of changes to the IEEE standard. The standard section on "Data types" has been split into two sections, one on "Types" and another on "Objects". In cases where a subsection of the standard is substantially unchanged, such as the subsection on "Strengths", this donation simply makes a reference of the form "IEEE 1364-2001 *section number*" rather than repeat the content.

### 1.3 Conventions

- [15] This document attempts to follow the conventions of an IEEE-draft standard as available at <http://standards.ieee.org/resources/development/writing/templates.html> and as used in the IEEE Standard 1364-2001 for the Verilog hardware description language.
- [16] Some sections may contain a "Usage" section. Usage sections contain helpful material written in the style of a user guide. Usage discussion is not intended for inclusion in an IEEE standard as normative text.

## 2 Types

- [17] A type is a set of values and a corresponding set of operations.
- [18] The Verilog types include a small number of *primitive types* that serve as a basis for the value system. The values of a primitive type cannot be defined in terms of other values or types.
- [19] The type system also includes types that have values that are constructed from other values. Such types are called *user-defined types* because a Verilog description must include an explicit specification of how the values are constructed. User-defined types can be constructed from primitive types and from other user-defined types in arbitrarily complex arrangements.
- [20] A user-defined type can be described directly in an object declaration. Alternatively, a user-defined type can be described in a type declaration that gives the type a name that can then be used in other declarations.
- [21] Type compatibility is determined by the structure of the values, rather than by the names, if any, of the types involved. Objects and values that share a common value topology are considered to be of the same type. Objects and values that have different value topologies are not of the same type. These concepts are described in more detail in 2.11.

### Usage:

- [22] Verilog types include predefined types as well as user-defined types that are constructed from other types. The simplest use for a type is to define the sort of values that an object holds. For example, in the following object declaration, the net 'realnum' is declared to take on values of the primitive type **real**:

```
wire real realnum;
```

- [23] You can also use a type to construct other types. For example, in order to define a type that contains multiple elements, you must specify the types of its elements. In the following array type declaration, **integer** indicates that type `int_array_type` is an array of integers:

```
typedef integer int_array_type [1:10];
```

- [24] This array type now has a name that you can use to type objects or to construct other types. The following declaration defines a 100-element array of 'int\_array\_type' elements:

```
reg int_array_type my_array_of_arrays [1:100];
```

- [25] The type of 'my\_array\_of\_arrays' has no name because the type is described directly in the object declaration rather than in a type declaration. However, the type of 'my\_array\_of\_arrays' is compatible with that of 'my\_other\_array\_of\_arrays', below, because they are both 100x10 arrays of integers.

```
reg integer my_other_array_of_arrays [1:100] [1:10];
```

### 2.1 Primitive types

- [26] A primitive type is a predefined type that has values that cannot be described in terms of other values or types. A primitive type has no equivalent type declaration. Primitive type names are keywords.
- [27] The primitive types are **bool**, **logic**, and **real**.

```
primitive_type_specifier ::=
    scalar_logic_type | real

scalar_logic_type ::=
    bool | logic
```

- [28] The **bool** and **logic** types belong to a special class of primitive types called *scalar logic types*. These types define two-state and four-state logic systems, respectively. Each state value is 1 bit wide. The values of a composite type have an underlying logic representation in either the two-state system or the four-state system. In addition to being used as a general type, a scalar logic type can be used to specify an underlying logic representation for a given object or type.

### 2.1.1 Bool

- [29] The **bool** type represents a two-state logic system. The value set consists of two basic values:

- 0 – represents a logic zero or a false condition
- 1 – represents a logic one or a true condition

- [30] The values 0 and 1 are logical complements of one another.

### 2.1.2 Logic

- [31] The **logic** type represents a four-state logic system. The value set consists of four basic values:

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

- [32] The values 0 and 1 are logical complements of one another.

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

- [34] The language includes *strength* information in addition to the basic value information for nets of type **logic**. This is described in detail in IEEE 1364-2001 7.

### 2.1.3 Real

- [35] The values of type **real** are 64-bit floating point numbers.

## 2.2 Type declarations

- [36] A type declaration introduces a new name for a data type. Once a type name has been introduced, the type name can be used anywhere that the type name is visible and a type specifier is expected. A type name shall not be used before it is declared.

```

description ::=
    | data_type_declaration
module_or_generate_item ::=
    | data_type_declaration
block_item_declaration ::=
    | data_type_declaration
type_identifier ::= identifier
data_type_declaration ::=
    incomplete_data_type_declaration
    | complete_data_type_declaration
complete_data_type_declaration ::=
    typedef type_specifier list_of_type_declaration_names ;
list_of_type_declaration_names ::=
    type_declaration_name { , type_declaration_name }
type_declaration_name ::=
    identifier [ array_type_specifier ]
type_specifier ::=
    type_name
    | predefined_type_name
    | primitive_type_specifier
    | user_defined_type_specifier
type_name ::=
    [ scalar_logic_type ] type_identifier

```

- [37] A type declaration can appear outside of a module declaration, or in any location in which an object declaration is allowed. If a type name is declared inside a block, then the type name shall be visible only within that block. If a type name is declared directly inside a module then the type name shall be visible only within that module. If a type name is declared outside of a module, then that type declaration shall be treated lexically in the order in which it is encountered in the description; that is, the type name shall be visible in any module that follows the type declaration in the description.
- [38] A type declaration does not create a new type; it merely adds new names for existing types. The type itself is defined by the form of its values and the associated operators. It is an error if the same type name is introduced in two or more type declarations that are visible in the same location and the corresponding types are not identical in the topology of their values.
- [39] A complete type declaration associates type names with types. A type can be associated by naming a previously declared type or a primitive type. Alternatively, the associated type can be defined in the declaration itself by using a type specifier to describe a user-defined type. Depending on the nature of the associated type, the type specifier appears to the left or to the right of the type identifier.
- [40] A type declaration does not necessarily specify the types associated with the type names. An *incomplete type declaration* simply introduces one or more type names. The association of an incomplete type name with a specific set of values occurs in a subsequent *complete type declaration*. This coupled use of incomplete and complete type declarations is important for declaring types that are mutually dependent or recursive.

## 2.3 User-defined types

- [41] A user-defined type is a type that is defined in terms of another type. User-defined types can be used to define values that are composed of other values, to define values that denote dynamically allocated objects, and to define alternative uses for the same object storage.

```

user_defined_type_specifier ::=
    handle_type_specifier
  | union_type_specifier
  | structure_type_specifier
  | vector_type_specifier

```

- [42] A type that defines a reference to a certain type of dynamically allocated object is a **handle** type.
- [43] A type that defines different uses for the same object storage is a **union** type.
- [44] A type that has values that are composed of other values, called *elements*, is a *composite type*. Elements that have names associated with them are called *members*. Depending on the nature of the composite type, its elements can be accessed by position using a numeric value called an *index*, or by member name.
- [45] Each Verilog composite type is either a *vector type* or an *aggregate type*. Both vector and aggregate types define values as sets of elements. The essential difference is that the elements of a vector type have a predefined ordering relation between them, while those of an aggregate type do not.
- [46] A vector type is a multiple-bit type representing a vector as defined in IEEE 1364-2001 3.3.1. The elements are logic values and their order is significant. Predefined arithmetic and logical operations may be performed on values of a vector type. Different forms of vector types provide convenient abstractions, such as the ability to reference elements by name rather than position. Regardless of its form, a vector type still represents a bit vector, and its values can be manipulated as such. Vector types conveniently represent hardware abstractions such as the bit-level layout of an integer.
- [47] There are three basic kinds of vector types. An indexed vector type defines a vector with elements that are referenced by position. A tagged vector type defines a vector type with elements that can be referenced both by position and name. An enumerated vector type associates a vector type with a special set of named values called *enumeration constants*.
- [48] An object or value of an aggregate type is simply a set of elements. There is no predefined relationship between the elements, and no meaning is implied by their order. Predefined arithmetic and logical operations cannot be performed on an aggregate value as a whole. Aggregate types are like the composite types in many general-purpose programming languages.
- [49] There are two basic kinds of aggregate types. An array type defines a set of elements that are referenced by position. A structure type defines a set of elements that are referenced by name.
- [50] Vector types are inherently homogeneous, while aggregate types can be homogeneous or heterogeneous. Since vector types are all bit vectors, they have a common topology and are all assignment compatible with one another. Aggregate types may define different topologies, and are not necessarily assignment compatible with one another. Vector and aggregate types are not assignment compatible with one another.

### Usage:

- [51] Array and structure types are provided to create data structures for which the underlying mapping onto logic vectors is not yet specified. These composite types can contain elements of any type. Unlike vector types, arrays and records can contain elements of type **real** and **handle**.

- [52] To illustrate the difference between vector types and aggregate types, consider the code fragments below.

```

reg logic [7:0] logic_vector, logic_vector2;

reg integer integer_array1 [1:10], integer_array2 [1:10];

...
logic_vector1 = ~logic_vector2           // legal bit-wise negation
integer_array1[1] = ~integer_array[1]; // legal also
integer_array1 = ~integer_array2       // error

```

- [53] The logic vector and integer array variables each have elements that can be accessed through an index; however, you can use variable 'logic\_vector' as an operand to a predefined numeric or logical operator; but you cannot use these predefined operators on 'integer\_array' as a whole.
- [54] Arrays, records, and unions of other types are intended for use in system-level models and/or test benches. For example, a structure or union may be used to represent an object such as a network packet, video frame, or software instruction. It may be more practical at a system-level to represent these objects as having dynamically allocated content and to express activity on these objects in terms of events, rather than relying on statically allocated storage and assignment semantics for sensitivity.

## 2.4 Vector types

- [55] A vector type defines a set of logic bits that have a predefined relationship to one another. Vector objects and expressions shall obey laws of arithmetic modulo 2 to the power of  $n$  ( $2^n$ ), where  $n$  is the number of bits in the vector. Implementations do not have to detect overflow of integer operations.
- [56] The different kinds of vector types provide different ways of abstracting the data, but all declare a contiguous vector of a single underlying scalar logic type. These vectors remain completely assignment and operator compatible with one another.
- [57] There are three kinds of vector types: indexed vectors, tagged vectors, and enumerated vectors. Elements of a vector type can always be referenced by numeric position. Additionally, tagged vector elements can also be referenced by name, and enumerated vector objects can be manipulated with a special set of named values. Regardless of these additional capabilities, objects of a vector type can always be manipulated as logic vectors.

```

vector_type_specifier ::=
    indexed_vector_type_specifier
    | tagged_vector_type_specifier
    | enumerated_vector_type_specifier

vector_ranges ::=
    vector_range { vector_range }

vector_range ::=
    [ signed ] range

```

- [58] A vector type is characterized by its width, by its ranges, by its signed or unsigned properties, and by its underlying logic representation.
- [59] The *width* of a vector type is the number of logic elements in its values. This number is distinct from the storage required for the value itself. A four-state logic vector takes two storage bits for each logic element, and therefore requires  $2 * width$  bits to store all possible values. An implementation may set a limit on the maximum width of a vector, but the limit shall be at least  $65536$  ( $2^{16}$ ) bits.

- [60] The ranges of a vector give addresses to the individual bits in a multi-bit object or value. The most significant bit specified by the *msb* constant expression is the left-hand value in the range, and the least significant bit specified by the *lsb* constant expression is the righthand value in the range.
- [61] Both the *msb* constant expression and the *lsb* constant expression shall be constant expressions. The *msb* and *lsb* constant expressions can be any value – positive, negative, or zero. The *lsb* constant expression can be a greater, equal, or lesser value than *msb* constant expression.
- [62] Multiple vector ranges of a vector type are evaluated such that the right-most specified range is the least significant. An index in the right-most range accesses a bit, while an index in any of the remaining ranges accesses a subset of the vector bits. The subset of bits that are accessed by a given vector range contains the bits or subsets of bits accessed by ranges to the right.
- [63] Each range of a vector type can individually have the property of being **signed**. In the absence of this keyword, the set of bits that are accessed by that vector range shall be considered unsigned.
- [64] The scalar logic type can be omitted from a vector type definition, thereby allowing subsequent type or object declarations to determine the scalar logic type. If the scalar logic type is not specified, then the vector shall have an *undetermined logic type*. Eventually, the underlying scalar logic type is determined by a type or object declaration.
- [65] Because all of the elements of a vector type must have the same scalar logic type, the scalar logic type is specified at the level of the object declaration or the type specifier that is used in the object declaration. It shall be an error to create a vector type or object that is determined to have a mix of scalar logic types for its underlying elements.
- [66] The scalar logic type can be specified by explicitly naming the logic type as a part of the type specifier in a type or object declaration. If the type specifier in an object declaration denotes a vector type with an undetermined scalar logic type, then the scalar logic type of the vector object shall be **logic**.

### Usage:

- [67] This undetermined scalar logic type capability is provided to simplify design refinement from abstract two-state implementation to more precise four-state implementation. A single type name with an undetermined scalar logic type can be created. This declaration allows different models at different levels of detail to declare either two-state or four-state versions of the object by sharing the type name and simply changing the object declaration, thus eliminating the need to duplicate the type declaration in a **bool** or **logic** version and change the declarations of the objects.
- [68] The declarations below show different ways to specify the underlying logic type of a vector type.

```
// A 32-bit integer with an undetermined logic type
typedef [31:0] generic_integer;

// 2-state and 4-state versions of our integer type
typedef bool generic_integer integer_2state;
typedef logic generic_integer integer_4state;

reg bool generic_integer my_2state;
reg integer_2state another_2state;
reg bool [31:0] yet_another_2state;

reg logic generic_integer my_4state;
reg integer_4state another_4state;
reg logic [31:0] yet_another_4state;
```

- [69] Vector and primitive logic types are the types on which all arithmetic and logical operators are defined. All vector types are completely assignment and operator compatible. For instance, the

value of an enumerated vector object can be assigned to a tagged vector object. Vector types also remain compatible across object kind, so a variable of a vector type can be used to drive a net of any other vector type. Extension or truncation of values of different widths is defined exactly the same way as it is for vectors in IEEE1364-2001 today. Using this form of abstraction, a model that chooses to represent internal data as a structure can drive that data onto a vector port or wire without any sort of value conversion.

### 2.4.1 Indexed vector types

An indexed vector type is the most basic form of vector type. It simply defines a logic vector of elements that are accessed by one or more index positions, with no additional capabilities like selection by name.

```
indexed_vector_type_specifier ::=
    [ type_specifier ] vector_ranges
```

- [70] The optional type specifier defines the vector element type. If the vector element type is not specified then the vector has an undetermined logic type. In this case the element type is one of the scalar logic types, as determined by a subsequent type declaration or object declaration. If an element type is specified, then that element type shall itself be either another vector type, or a scalar logic type. If the vector element type is itself a vector type, then the indexed vector type is equivalent to a multi-dimensional indexed vector type in which the element vector ranges follow the indexed vector ranges.

#### Usage:

- [71] An indexed vector type is simply a conventional Verilog vector that has been extended to support multiple dimensions.
- [72] In the following example, the declaration of type 'twowords' defines a two-dimensional logic vector:

```
typedef bool [1:0] signed [31:0] twowords;
reg twowords tw;
```

- [73] The variable 'tw' has values of type 'twowords', which means that
- tw is unsigned
  - tw[1] is a signed 32-bit two-state vector
  - tw[1][31] is the sign bit of tw[1]
- [74] The power of multiple ranges and an individual **signed** specification on each range is best demonstrated by example. See a detailed example in B.1.

### 2.4.2 Tagged vector types

- [75] A tagged vector type provides an alternate view of a vector by allowing the elements to be manipulated both by name and by index. The names for individual members of the structure are equivalent to bit or part selects of the underlying vector.

```

tagged_vector_type_specifier ::=
    [ scalar_logic_type ] vect [ vector_range ]
    { member_declarations }

member_declarations ::=
    member_declaration { member_declaration }

member_declaration ::=
    type_specifier list_of_type_declaration_names ;

```

- [76] If a vector range is given, then that range determines the bounds of the vector, and the sum of the widths of the elements in the vector members shall equal the width of this range. If a range is not specified then the range is implicitly defined to be [width-1:0], where the width is the sum of the widths of the members.
- [77] The members of a tagged vector are accessed by using the dot operator (.). The precedence is the same as when the dot operator is used in hierarchical pathnames.
- [78] For assignment purposes, the topology of a vector structure is that of the underlying vector array type.

### Usage:

- [79] This example defines a tagged vector for the implementation of a packet type. Because the elements of a tagged vector have the same underlying logic representation, the specification of the logic type is made at the level of the packet type itself rather than at the level of its members.

```

typedef enum {
    PACKET_OK, PACKET_ERROR
} status_t;

typedef bool vect {
    [7:0]    src_port; // number of the source port
    [7:0]    dest_port; // number of the destination port
    status_t status; // status bit
    [127:0] payload; // packet payload
} packet_t;

// Objects of this type are 2-state vectors
// with an implicit width of [144:0]

reg packet_t rp;
wire packet_t wp;

```

- [80] You can find a different version of this type in B.2. This other version builds type packet\_t from other type declarations.

### 2.4.3 Enumeration Types

- [81] Enumeration types provide abstraction by defining a means to create named constants with specific bit patterns. Literals of enumeration types are simply uses of the enumeration constant identifiers. Since enumerations are vector types, any vector literal value can also be used.

```

enumerated_vector_type_specifier ::=
    [ scalar_logic_type ] enum [ vector_range ]
    { enum_constant_list }

enum_constant_list ::=
    enum_constant_decl { , enum_constant_decl }

enum_constant_decl ::=
    identifier [ = constant_expression ]

```

- [82] If an enumeration constant declaration includes a constant expression, then that expression defines the *enumeration value encoding* for that enumeration constant. Thereafter, use of the enumeration constant shall be equivalent to specifying the constant expression itself. The constant expression shall be appropriate for a vector type.
- [83] If an enumeration constant declaration does not have a value encoding, then default rules shall determine the value of the constant. If the constant is the first one, then its value shall be `b0; otherwise the value shall be that of the previous constant plus one. It shall be an error if the previous value contains x or z values in this case. In the presence of mixed explicit and implicit encodings, it is possible for the representation of the next implicit value to exceed the width of the underlying vector type. If this occurs, then the value of the constant shall wrap to `b0.
- [84] It shall be an error if more than one enumeration constant is determined to have the same value encoding, whether the encodings are explicit or not. Furthermore, it shall be an error if a type or object declaration specifies that the underlying scalar logic type is **bool** and the enumeration type has an explicit encoding containing x or z values.
- [85] If a scalar logic type is specified, then that type shall determine the scalar logic representation of the vector type.
- [86] If a vector range is given, then that range shall define the bounds of the underlying vector. Otherwise, the bounds shall be determined by the enumeration constant declarations themselves. In this case, it shall be an error if an enumeration value encoding is specified using an unsized literal, or if all explicit enumeration value encodings are not of the same size.
- [87] When determining implicit enumeration bounds, if enumeration value encodings are present then the range shall be set to be [width-1:0], where width is chosen to be the width of any and all enumeration value encodings given. If no enumeration value encodings are given, then the range shall be set to be [(ceil(log2(n)) - 1):0] where  $n$  is the number of enumeration constants (i.e. the number of bits it takes to binary encode all given enumeration constants).

### Usage:

- [88] Enumeration value checking is performed only in order to determine a proper encoding of the enumeration values. Since objects of an enumeration type are simply net or variable objects of the appropriate vector width, no runtime checking of value assignments is mandated. An implementation may choose to perform optional range checking to ensure that assigned values correspond to one of the enumeration constants.
- [89] Some examples of enumeration types follow.

```

typedef enum { GREEN, YELLOW, RED } color;
// scalar_logic_type is undetermined
// range is [1:0]
// encoding is GREEN = 2'b00, YELLOW=2'b01, RED=2'b10

// An object declaration can determine the scalar logic type

```

```

reg bool color rcb;    // A 2-state valued register of color
wire logic color wcl; // A 4-state valued wire of color

// vector array type declaration
typedef logic enum {
    BASE_MEM      = 'h0000,
    END_MEM       = 'h10FF,
    BASE_PCI      = 'h1100,
    END_PCI       = 'h110F,
    BASE_CPU      = 'h1200,
    END_CPU       = 'h120F,
    BAD_ADDR      = 'hXXXX
} mem_map;
// scalar logic type is logic
// range is [15:0]

```

- [90] The following enumeration type declaration contains multiple errors:

```

typedef bool enum [1:0] bit {
    ZERO = 'b0,    // OK
    ONE  = 4'b1,   // ERROR: Initializer too wide
    TWO  = 2'1X,   // ERROR: X value in 'bool'
    NULL = 2'b0    // ERROR: duplicates 0 value
} enum_error;

```

## 2.5 Array types

- [91] An array type is a composite type that defines a homogenous collection of values. An individual element is accessed through its position in the array, as given by one or more numeric indices.
- [92] An array can be used to group elements into multi-dimensional objects. An array type specifier shall include the element address range for each dimension of the array. The expression(s) that specify the indices of the array shall be constant expressions. The value of the constant expression can be a positive integer, a negative integer, or zero. There is no sign extension property associated with an array dimension.

```

array_type_specifier ::=
    dimension { dimension }

```

- [93] Unlike most type specifiers, the dimensions of an array type are specified to the right of the type or object name. The element type, which is specified to the left of the type or object name, can be any kind of type. Specification of the scalar logic type cannot be deferred; it shall be an error if the element type has an undetermined logic type.
- [94] There are no predefined arithmetic or logical operators for array types; however, if the element type is a vector type then the predefined arithmetic and logical operators can be applied to individual elements of the array.
- [95] An implementation may limit the maximum size of an array, but the limit shall be at least  $16777216$  ( $2^{24}$ ).

### Usage:

- [96] Array types can have any element type. This example shows an array of structures:

```

// declare the full structure type
typedef struct {

```

```

        int opcode;
        int op1;
        int op2;
    } instruction_s;

    // declare type name for an array of 32 instructions
    typedef instruction_s instruction_a [31:0];

    // declare an instruction array object
    reg instruction_a instructions;

```

- [97] Arrays are syntactically distinct from indexed vectors because their ranges are specified to right of the type or object identifier rather than to the left. The following example shows the difference in their declarations:

```

    // 64-bit wide vector
    typedef bool [1:0] [31:0] doubleword;

    // 2-element array of 32-bit vectors
    typedef bool [31:0] twowords [1:0];

```

- [98] Arrays and vectors are not assignment compatible.

## 2.6 Structure types

- [99] A structure type is a composite type that defines a collection of values with names. The elements of a structure can be of different types and sizes.

```

structure_type_specifier ::=
    struct { member_declarations }

```

- [100] An individual element of a structure is called a *member*. A member is accessed through its name by using the dot operator (.). The precedence is the same as when the dot operator is used in hierarchical pathnames.

The type of a structure member can be any kind of type; however it shall be an error if that type has an undetermined logic type.

The only predefined operators on structures are identity (===, !==) and assignment (=, <=). All other operators must be modeled through user-defined tasks and functions.

### Usage:

- [101] Structures can be used to bundle elements of any type. Because they can include **handle** types, structures can be used to create dynamically allocated data structures. The following example shows how a doubly linked list of dynamically allocated objects might be represented:

```

module dlist ();

    typedef queue_t; // incomplete type declaration for queue_t type
    typedef payload_t; // incomplete type declaration for payload
    typedef handle payload_t payload_h; // a handle to the payload

    // complete typedef for the queue_t type
    typedef struct {
        payload_h payload; // handle to the payload
        handle queue_t prev, next;
        // prev and next handles for queue type
    }

```

```

    } queue_t;
endmodule

```

- [102] Although structures and tagged vectors are similar in that they both define member elements; they are different kinds of types, and they are not assignment compatible.

## 2.7 Union types

- [103] Union types define multiple names for the same storage locations. Unions allow abstract modeling of objects with a variable layout depending on the content of some of the data.

```

union_type_specifier ::=
    union { member_declarations }

```

- [104] All members of a union occupy the same physical location in memory, therefore each member of a union provides a different view of the value of the union. A particular union member is selected by prefixing the member name with the dot operator (.). The precedence is the same as when the dot operator is used in hierarchical pathnames.

- [105] The members of a union shall satisfy one of the following type requirements:

1. All members have the same primitive type.
2. All members are logic vectors with the same width and scalar logic type.
3. All members have identical array types.
4. All members have identical structure types.

- [106] The nature of the member types determines how a union can be used. If all of the members of a union are vector types then that union is a *union of vector types*, and that union can be used in any context in which a vector type is allowed. Unions that do not meet this criteria cannot be used in a context that requires a vector type.

### Usage:

- [107] Unions are typically used in situations where they are embedded in a structure that contains an element to indicate which member of the union is currently in use. The following example shows this sort of use of a union:

```

typedef bool enum { one_long, two_shorts } word_kind;

typedef struct {
    word_kind kind;
    union {
        logic [31:0]    long_word;
        logic [1:0][15:0] short_words;
    } u;
} word_type;

reg word_type word;

```

- [108] The value of member “word.kind” indicates whether member “u” is manipulated as “word.u.long\_words” or “word.u.short\_words”.

- [109] A detailed example in B.3 shows how to use a union of vector types to define alternate bit field layouts.

## 2.8 Handle types

- [110] A **handle** is a descriptor that contains information about a dynamically allocated object. A handle contains information on the type of the referenced object, the fanout of the object (if any) for scheduling when the object is modified, and usage information to allow for garbage collection of memory. The representation of a **handle** value is implementation defined.
- [111] There are three forms of handle specifiers. The first two forms define a handle to a data object. The third form defines a handle to an event object.

```

handle_type_specifier ::=
    handle_type_specifier
    | handle *
    | handle event

```

- [112] Handles to data objects reference objects that have values. A *typed handle* references a specific type of data object; its declaration includes a type specifier that determines the type of that object. If the asterisk character is used instead of a type specifier, then the handle is an *untyped handle*. An untyped handle can denote any type of data object.
- [113] An event handle references a named event, which does not have a value. Event handles can be assigned to one another, but the event objects themselves cannot be assigned.
- [114] Section 6 describes dynamic memory allocation and operations on handles in detail.

### Usage:

```

typedef handle integer i_h; // type for a handle to an integer
reg i_h r_i; // variable of the integer handle type
reg handle integer ii_h; // variable that is a handle
// to an integer (without typedef)

typedef list_s;
typedef handle list_s list_h; // type for handle to a list_s type;

typedef struct {
    logic [31:0] addr;
    logic [31:0] data;
    handle event ready;
    list_h next; // handle to another object of this type
} list_s;

wire list_h w_1; // wire holding a handle to a list_s object

// Two handles to the same event
handle event data_ready1 = new;
handle event data_ready2 = data_ready1;

```

## 2.9 Incomplete types

- [115] An incomplete type declaration introduces one or more type names without specifying the nature of the types. The primary use of an incomplete type declaration is to define types that are mutually dependent or recursive in nature.

```
incomplete_data_type_declaration ::=
    typedef list_of_incomplete_type_identifiers ;

list_of_incomplete_type_identifiers ::=
    type_identifier { , type_identifier }
```

- [116] Each incomplete type identifier shall have a corresponding complete type declaration with the same identifier. If the incomplete type is declared outside of a module, then the corresponding complete type declaration shall occur before the next module in the description. If the incomplete type is declared in a block (including a task or function), then the corresponding complete type declaration shall occur in the same block. If the incomplete type is declared directly in a module or generate, then the corresponding complete type declaration shall occur in the same module or generate.
- [117] There is no relationship between the identifiers in an incomplete type declaration. Each type identifier can have a completely different type associated with it. Multiple incomplete type declarations for the same identifier can occur before or after the complete type definition.

### Usage

- [118] The following example shows a typical use of an incomplete type declaration. This example defines a type for representing a simple linked list of integers. The type of a node in the linked list is recursive because a list node contains a handle to the next list node. An incomplete type declaration introduces the name of the list node type, which is then used to declare a type name for a handle to a list node, which in turn is used to declare the 'next' element of a list node in the complete type declaration.

```
// An incomplete type declaration for the list node
typedef list_node_type;

// A complete type declaration for a list node handle
typedef handle list_node_type list_handle_type;

// The complete type declaration for the list node
typedef struct {
    integer          data;
    list_handle_type next;
} list_node_type;
```

## 2.10 Predefined type names

- [119] Some important common types have names that are predefined by the standard. These types are not primitive types because their values are described in terms of other types. These predefined type names shall be implicitly declared prior to any user-defined source code.

```
predefined_type_name ::=
    integer | time | realtime | long | int | shortint | char
```

- [120] The following are implicit type declarations for the IEEE 1364-2001 types **integer**, **time**, and **realtime**:

```
typedef logic signed [implementation_defined:0] integer;
typedef logic [63:0] time;
typedef real realtime;
```

- [121] An **integer** is a general-purpose type that is used for manipulating quantities that are not regarded as hardware registers. Integer values are signed quantities with the least significant bit being zero. An implementation may limit the width of values of type **integer**, but the values shall be at least 32 bits wide.
- [122] Type **time** is used for storing and manipulating simulation time quantities in situations where timing checks are required and for diagnostics and debugging purposes. This data type is typically used in conjunction with the **\$time** system function (see IEEE 1364-2001 Section 17). Time values are unsigned quantities with the least significant bit being zero. An implementation may limit the width of values of type **time**, but the values shall be at least 64 bits wide.
- [123] Type **realtime** is simply a synonym for type **real**; these two type names can be used interchangeably.
- [124] The following implicit type declarations are defined for types with a two-state scalar logic type:

```
typedef bool signed [implementation_defined:0] long;
typedef bool signed [implementation_defined:0] int;
typedef bool signed [15:0] shortint;
typedef bool signed [7:0] char;
```

- [125] Type **int** is equivalent to type **integer**, except for its scalar logic type; values of type **int** have a two-state representation. The widths of type **long** and **int** are implementation-defined; however, the width of each shall be at least 32 bits. The widths of **long** and **int** are not necessarily the same. The width of **int** shall be no greater than the width of **long**.

## Usage

- [126] The following examples show use of the predefined type names:

```
integer a;           // a variable to store an integer value
time last_chng;     // a variable to store a time value
real float;         // a variable to store a real value
realtime rtime;    // a variable to store time as a real value

long update_count; // a variable to store a 2-state integer value
int loop_index;    // a variable to store a 2-state integer value
short_int num_states; // a variable to store a 2-state 16-bit value
char ch;           // a variable to store a character value
```

## 2.11 Type compatibility

- [127] When constructs such as objects, ports, and expressions interact in a design, it is often necessary to determine that the types of the constructs are compatible; that is, that there is relationship between the values of the respective types that makes the interaction both safe and meaningful.
- [128] Type compatibility is determined by topology rather than by type name. Different contexts require different kinds of type compatibility.

- [129] The strictest form of compatibility requires that two types be identical. Two types are considered to be the same or identical if they are *topologically identical* – that is, if every aspect of the two types, except for type names, is the same. Two types are topologically identical one of the following statements is true:
1. Both types are the same primitive type.
  2. Both types are handles to events.
  3. Both types are untyped handle types.
  4. Both types are typed handle types, and the handle object types are topologically identical.
  5. Both types are indexed vector types with the same scalar logic type and the same number of vector ranges, and corresponding ranges have the same msb, lsb, and signed property.
  6. Both types are tagged vector types with the same scalar logic type, the same msb, the same lsb, the same signed property, the same number of members, and members at corresponding positions in the lists of member declarations have the same name and type.
  7. Both types are enumerated vector types with the same scalar logic type, the same msb, the same lsb, the same signed property, the same number of enumeration constants, and constants at corresponding positions in the lists of constant declarations have the same name and value.
  8. Both types are array types with the same element type and dimensionality, and corresponding dimensions of the array types have the same msb and the same lsb.
  9. Both types are structure types with the same number of members, and members at corresponding positions in the lists of member declarations have the same name and type.
  10. Both types are union types with the same number of members, and members at corresponding positions in the lists of member declarations have the same name and type.
- [130] Assignment type compatibility is discussed in 5.
- [131] Type compatibility at a port connection is discussed in 3.5.3.3.

### 3 Objects

- [132] An object is a named design element that is used to describe structure or behavior. A *data object* has a value associated with it. Verilog data objects are designed to represent the data storage elements and transmission elements found in digital hardware systems. Other kinds of objects, such as named events, do not have values associated with them, but provide important capabilities through their execution properties.
- [133] There are many basic kinds of data objects: parameters, nets, ports, variables, and arguments. These kinds of objects differ in the way that they are assigned and hold values. They also represent different hardware structures.
- [134] Verilog parameters represent constants. Typical uses of parameters are to specify delays and width of variables.
- [135] Nets and ports represent connections between structural components and behavioral blocks.
- [136] Assignments to a variable are made by procedural assignments (see IEEE 1364-2001 6.2 and 9.2). Since a variable holds a value between assignments, it can be used to model hardware registers. Edge-sensitive (i.e. flip-flops) and level-sensitive (i.e. RS and transparent latches) storage elements can be modeled. A variable need not represent a hardware storage element since it can also be used to represent combinatorial logic. In addition to modeling hardware, variables can be used for general purposes, such as counting the number of times that a particular net changes value.

#### 3.1 Data object declarations

- [137] A data object is generally introduced by an object declaration that gives the object a name and establishes the semantics by which values are associated with that object. Data objects that share the same characteristics can be declared in the same declaration statement. An object declaration shall not declare a name that is already declared by a net, parameter, variable, or type declaration (see IEEE 1364-2001 3.12).
- [138] Every object has a type that determines a set of possible values for that object. The type of an object can be explicitly stated in the object declaration, or in the absence of a type specifier, the type shall be inferred to be type **logic**.
- [139] In the absence of an explicit declaration, an implicit net with default characteristics can be assumed under the conditions described in 3.3.

#### Usage:

- [140] The data object declarations below declare wire nets 'w1' and 'w2' of the primitive four-state type **logic**. The type of 'w1' is inferred, and the type of 'w2' is explicitly stated.

```
wire w1;
wire logic w2;
```

- [141] You can declare more than one object in a single declaration:

```
wire w1, w2;           // declares two wires
reg [4:0] x, y, z;    // declares three 5-bit regs
```

- [142] Because **logic** is the default scalar logic type, you must be explicit in order to declare a two-state variable:

```

reg bool has_error;    // a Boolean status variable
wire bool [7:0] byte;  // a two-state, 8-bit wire
reg int my_int;        // int is a predefined 2-state integer type

```

- [143] You can define a type directly in an object declaration. The object declaration below introduces two enumeration constants for the values of variable 'process\_status':

```

reg bool enum { suspended, active } process_status = suspended;

```

## 3.2 Variables

*NOTE: This section updates IEEE 1364-2001 clauses 3.2.2, 3.8, and 3.9 to take into account the new data types.*

- [144] A *variable* is an abstraction of a data storage element. A variable shall store a value from one assignment to the next. An assignment in a procedure changes the value in the data storage element.
- [145] There are two basic forms of variable declaration. The first form includes the keyword **reg** but no type specifier to the left of the variable identifiers; this form can be used to declare scalar or array variables of a four-state logic type. The second form has an optional keyword **reg**, but includes a type specifier to the left of the variable identifiers. The second form can be used to declare a variable of any type.

```

variable_declaration ::=
    reg list_of_variable_definitions ;
  | [ reg ] type_specifier list_of_variable_definitions ;

list_of_variable_definitions ::=
    variable_definition { , variable_definition }

variable_definition ::=
    variable_identifier [ array_specifier ] [ = expression ]

```

- [146] A variable shall be initialized according to its type. The initialization value for **logic** variables shall be the unknown value, x. The initialization value for **bool** variables shall be a logical zero, 0. The initialization value for **real** and **realtime** variables shall be 0.0. The initialization value for **handle** variables shall be **null**. Variables of a composite type shall be initialized by recursively initializing each element according to its element type.
- [147] If a variable declaration assignment is used (see IEEE 1364-2001 6.2.1), the variable shall take this value as if the assignment occurred in a blocking assignment in an initial construct.

### Usage

- [148] When declaring a variable, you can use the keyword **reg**, or you can specify a type, or you can do both. The following three declarations show equivalent ways to declare a four-state bit variable:

```

reg logic r1;    // explicitly specify everything
reg r2;          // logic is the default
logic r3;        // reg is optional

```

- [149] If you omit the scalar logic type, the scalar logic type defaults to **logic**:

```

reg scalar_reg;    // a 4-state logic bit
reg [7:0] vector_reg; // a 4-state 8-bit logic vector

```

```
reg array_reg [7:0]; // a 4-state 8-bit logic array
```

[150] You can also specify the scalar logic type:

```
reg bool [7:0] logic2state; // a 2-state 8-bit logic vector
reg logic [7:0] logic4state; // a 4-state 8-bit logic vector
```

[151] The following declarations show two different ways to declare a variable with a user-defined type. You can describe the type directly in the variable declaration, or you can use a previously defined type name:

```
bool enum { clear, warning, error } status1 = clear;

typedef bool enum { clear, warning, error } status_type;
status_type status2 = clear;
```

### 3.3 Nets

*NOTE: This section updates IEEE 1364-2001 clauses 3.2.1, 3.6, and 3.7 to take into account the new data types and to introduce the new net type **wone**.*

[152] A *net* represents a physical connection between structural entities, such as gates. A net shall not store a value, except for the **trireg** net. Instead, its value shall be determined by its drivers, such as a continuous assignment or a gate (see IEEE 1364-2001 Sections 6 and 7 for definitions of these constructs). If the type of a net is a composite type then each element of the net is itself a net, and can be driven independently from the other elements of the net.

[153] The declaration of one or more nets is introduced by a *net type* that determines how many drivers each net can have and how the values associated with each net is computed. Net types are discussed in more detail in 3.3.2. There are several different forms of net declaration because many aspects of the net declaration are optional.

[154] The **vectored** and **scalared** keywords are optional advisory keywords. If these keywords are implemented, then certain operations on vectors may be restricted. If the keyword **vectored** is used, bit and part-selects and strength specifications may not be permitted, and the PLI may consider the object *unexpanded*. If the keyword **scalared** is used, bit and part selects of the object shall be permitted, and the PLI shall consider the object expanded. It shall be an error to specify one of these keywords if the type of the net is not a vector type.

[155] If a type specifier is not given in the net declaration, then the default type of the net shall be **logic**. If the net is a vector and it has an undetermined logic type then the scalar logic type shall be **logic**.

[156] Strength in a net declaration is discussed in IEEE 1364-2001 3.4. Strength specifications in a net declaration shall apply only to nets or elements of nets that are of a logical type (a scalar logic type or a vector type).

```

net_declaration ::=
  net_type [ signed ] [ delay3 ] list_of_net_definitions ;
  | net_type [ vector | scalared ] type_specifier
    [ delay3 ] list_of_net_definitions ;
  | net_type [ drive_strength ] signed
    [ delay3 ] list_of_net_decl_assignments ;
  | net_type [ drive_strength ] [ vector | scalared ] type_specifier
    [ delay3 ] list_of_net_decl_assignments ;
  | trireg [ charge_strength ] [ signed ]
    [ delay3 ] list_of_net_definitions ;
  | trireg [ charge_strength ] [ vector | scalared ] type_specifier
    [ delay3 ] list_of_net_definitions ;

net_type ::=
  supply0 | supply1 | tri | triand | trior | trireg
  | tri0 | tri1 | wire | wand | wor | wone

list_of_net_definitions ::=
  net_definition { , net_definition }

net_definition ::=
  net_identifier [ array_type_specifier ]

list_of_net_decl_assignments ::=
  net_decl_assignment { , net_decl_assignment }

net_decl_assignment ::=
  net_definition = expression

```

[157] If no driver is connected to a net, the value of the net shall be the undriven state as determined by its type, unless the net is a trireg. In the case of a trireg, the net shall hold the previously driven value. The undriven state for a **logic** net shall be high impedance (z). The undriven state for a **bool** net shall be zero (0). The undriven state for a **handle** net shall be the **null** value. The undriven state for a **real** net shall be 0.0.

[158] In the absence of an explicit declaration, an implicit net shall be assumed in the following circumstances:

1. If an identifier is used in a port expression, then an implicit net of type **logic** and the default net type shall be assumed, with the vector width of the port expression declaration. See IEEE 1364-2001 12.3.3 for a discussion of port expression declarations.
2. If an identifier is used in the terminal list of a primitive instance or a module instance, and that identifier has not been explicitly declared previously in one of the declaration statements of the instantiating module, then an implicit scalar net of type **logic** and the default net type shall be assumed. See IEEE 1364-2001 Section 19 for a discussion of control of the net type for implicitly declared nets with the ``default_nettype` compiler directive.

### Usage:

[159] The following are examples of logic net declarations:

```

wire w;           // logic is the default
wire logic w;    // explicitly specify logic

typedef logic mvl4; // scalar type equivalent to logic
triand mvl4 tm;   // exactly equivalent to next line
triand tm;

wire [31:0] wi;   // 32-bit unsigned logic vector
wire signed [31:0] ws; // 32-bit signed logic vector
wire integer wint; // 32-bit signed logic vector

```

```

wone [4:0] x, y, z;      // three one-driver, 5-bit wires

wand w;                // scalar net of net type wand
tri [15:0] busa;       // three-state 16-bit bus
trireg (small) storeit; // charge storage node of strength small

```

[160] The following are examples of net declarations of other types:

```

wone real realnet;      // A one-driver real net
wire integer arrnet [63:0]; // Array of 32-bit integers

// A one-driver handle net
typedef list_node;
typedef handle list_node list;
typedef struct { int elem; list next; } list_node;
wone list list_head;

```

### 3.3.1 Net initialization

[161] The default initialization value for a net shall be the undriven state as defined in 3.3. Nets with drivers shall assume the output value of their drivers. The **trireg** net is an exception; the **trireg** net shall default to the value **x**, with the strength specified in the net declaration (**small**, **medium**, or **large**).

### 3.3.2 Net types

[162] There are several different kinds of nets, as shown in Table 1.

**Table 1 – Net types**

<b>wire</b>	<b>tri</b>	<b>tri0</b>	<b>supply0</b>
<b>wand</b>	<b>triand</b>	<b>tri1</b>	<b>supply1</b>
<b>wor</b>	<b>trior</b>	<b>trireg</b>	<b>wone</b>

[163] The net type of a net determines how values are associated with the net. Certain net types impose restrictions on the type of a net. The net types **wire**, **tri**, and **wone** can apply to nets of any datatype. The remaining net types are meaningful only for nets of a vector type or a scalar logic type. Table 2 shows which pairings of net types and net datatypes are valid; the “other” column in this table refers to the following kinds of datatypes: real, handle, array, struct, and union. It shall be an error if a net declaration includes a type specifier and that type is not a valid datatype for the specified net type.

Table 2 – Valid net type and net datatype combinations

Net Type	Net Datatype		
	logic	bool	other
wire	valid	valid	valid
wone	valid	valid	valid
tri	valid	valid	valid
supply0	valid	valid	error
supply1	valid	valid	error
wand	valid	valid	error
wor	valid	valid	error
triand	valid	valid	error
trior	valid	valid	error
tri0	valid	error	error
tri1	valid	error	error
trireg	valid	error	error

[164]

### 3.3.3 One-driver nets

[165] The net type **wone** represents single-driver nets. A net of net type **wone** shall have at most one driver. An error shall be raised if a **wone** net has more than one driver.

#### Usage

[166] The following design shows a valid net of net type **wone**. Net 'one\_driver' has just one driver.

```

module top;
    wone one_driver;
    bottom b (one_driver);
endmodule

module bottom ( inout p );
    buf (p, 1'b1);
    initial $monitor($stime,,"p is %b", p);
endmodule

```

- [167] If, however, you defined module 'top' as follows, you would get an error because net 'one\_driver' has two drivers, even though both drivers have the same value and there is no logic conflict.

```

module top;
    wone one_driver;
    assign one_driver = 1'b1;
    bottom b (one_driver);
endmodule

```

### 3.3.4 Logical Nets

- [168] A logical net can be either a two-state net or a four-state net. A two-state net has a two-state value associated with it, and a four-state net has a four-state value associated with it.
- [169] Under certain circumstances, a two-state net can be driven by a four-state driver (for example, if the net is connected to a primitive output). In such cases, the driving value is implicitly converted to a two-state value as defined in 5.2 before being placed on the net.
- [170] A connection of a four-state port to a two-state net cannot be collapsed.
- [171] Logic conflicts from multiple sources of the same strength on a four-state net are resolved as defined in IEEE 1364-2001 3.7.
- [172] Logic conflicts from multiple sources on a two-state net are resolved according to the net type as defined in Tables 3 and 4.

**Table 3 – Truth table for two-state wire/tri/wor/trior net**

wire/tri/wor/trior	0	1
0	0	1
1	1	1

**Table 4 – Truth table for two-state wand/triand net**

wand/triand	0	1
0	0	0
1	0	1

### 3.3.5 Real Nets

- [173] Nets of type **real** have a real value associated with them. There shall be at most one driver for a net of type real, and the value of the net shall be the value of that driver. The presence of multiple drivers shall result in an error.

- [174] Strength specifications do not apply to nets of type **real**.

### 3.3.6 Handle Nets

- [175] Nets of type **handle** have a handle value associated with them. There shall be at most one unique non-null value among the drivers of a handle net. A violation shall result in a runtime error, and the value of the net in error shall be the null value. If all of the drivers of a handle net are null, then the value of the net shall be the null value; otherwise, the value of the net shall be the non-null value.
- [176] Strength specifications do not apply to nets of type **handle**.

## 3.4 Parameters

- [177] Verilog parameters represent constants. There are two types of parameters: module parameters and specify parameters.

### 3.4.1 Module parameters

*NOTE: This section updates IEEE 1364-2001 clauses 3.11.1 and 12.2 to take into account the new datatypes.*

- [178] Module parameters are constants that can be modified at compilation time to have values that are different from those specified in the declaration assignment. This allows customization of module instances.
- [179] There are two different ways that module parameters can be defined. The first is in a *module\_parameter\_port\_list*, and the second is as a *module\_item* (see IEEE 1364-2001 12.1). A module declaration can contain parameter definitions of either or both kinds, or no parameter definitions at all. If any parameter assignments appear in a *module\_parameter\_port\_list*, then any parameter assignments that appear in the module become local parameters and shall not be overridden by any method.
- [180] A module parameter declaration shall contain a comma-separated list of assignments, where the right hand side of the assignment shall be a constant expression; that is, an expression containing only constant numbers and previously defined parameters (see IEEE 1364-2001 Section 4).

```

local_parameter_declaration ::=
    localparam [ signed ] list_of_param_assignments ;
    | localparam type_specifier list_of_typed_param_assignments ;

parameter_declaration ::=
    parameter [ signed ] list_of_param_assignments ;
    | parameter type_specifier list_of_typed_param_assignments ;

list_of_param_assignments ::=
    param_assignment { , param_assignment }

param_assignment ::=
    parameter_identifier = constant_expression

list_of_typed_param_assignments ::=
    typed_param_assignment { , typed_param_assignment }

typed_param_assignment ::=
    parameter_identifier [ array_type_specifier ]
    = constant_expression

```

- [181] A module parameter can be modified with the **defparam** statement or with a parameter value assignment in the module instance statement. A module parameter value shall not be modified at runtime.

[182] A module parameter declaration can include a type specifier or an optional **signed** specification. The type of a module parameter shall be in accordance with the following rules:

1. The type of a parameter that has no type specifier and no **signed** specification in its declaration shall default to the type of the final value assigned to the parameter, after any parameter value overrides have been applied.
2. A parameter that is declared with a **signed** specification shall be a one-dimensional signed vector with a range that is determined by the final value assigned to the parameter, after any parameter overrides have been applied. A value that is assigned to such a parameter shall be assignment compatible with a vector type, and the value shall be converted according to the rules in 5.2 if necessary.
3. If the final value applied to a parameter is sized, and that value determines the vector range of the parameter, then the parameter shall have an implied range with an *lsb* equal to 0 and an *msb* equal to one less than the size of the final value.
4. If the final value applied to a parameter is unsized, and that value determines the vector range of the parameter, then the parameter shall have an implied range with an *lsb* equal to 0 and an *msb* equal to an implementation-dependent value of at least 31.
5. A parameter that has a type specifier other than a scalar logic type shall have the type specified in its declaration. A value that is assigned to such a parameter shall be assignment compatible with its type, and the value shall be converted according to the rules in 5.2 if necessary.
6. A parameter that is declared with a type specifier that is a scalar logic type shall have that determined scalar logic type. A value that is assigned to such a parameter shall be assignment compatible with a vector type, and the value shall be converted according to the rules in 5.2 if necessary. If, after any parameter overrides have been applied, the final value assigned to the parameter is sized and the size is one, then the parameter is a scalar; otherwise the parameter is a one-dimensional vector with a range that is determined by the final value.
7. A parameter that has an undetermined logic type shall have the scalar logic type of the final value assigned to the parameter, after any parameter value overrides have been applied. If the final value has an undetermined scalar logic type then the scalar logic type of the parameter shall be **logic**.
8. The type of a parameter shall not be a handle type, nor can the type of a parameter have an element of a handle type. A handle value shall not be assigned to a parameter or an element of a parameter.

#### Usage:

[183] The following declarations show different parameter declarations and what those declarations imply about a parameter's type and value:

```

parameter msb = 7;           // defines msb as a constant value 7
parameter e = 25, f = 9;    // defines two constant numbers
parameter r = 5.7;         // declares r as a real parameter
parameter byte_size = 8,
       byte_mask = byte_size - 1;
parameter average_delay = (r + f) / 2;
parameter signed [3:0] mux_selector = 0;
parameter real r1 = 3.5e17;
parameter p1 = 13'h7e;

parameter [31:0] dec_const = 1'b1; // value converted to 32 bits,
                                   // scalar logic type is 4-state

```

```
parameter bool newconst = 3'h4; // vector with implied range [2:0]
                                // scalar logic type is 2-state

parameter intconst = 4; // implied range of at least [31:0]
                        // scalar logic type is 4-state

parameter bool boolconst = 4; // implied range of at least [31:0]
                                // scalar logic type is 2-state

parameter bool print_warnings = 1'b0; // 2-state scalar

parameter logic initial_state = 1'b0; // 4-state scalar
```

### 3.4.2 Local parameters

[184] Local parameters are described in IEEE 1364-2001 3.11.2.

### 3.4.3 Specify parameters

[185] Specify parameters are described in IEEE 1364-2001 3.11.3.

## 3.5 Ports

[186] Ports provide a means of interconnecting a hardware description consisting of modules, primitives, and macromodules. For example, module A can instantiate module B, using port connections appropriate to module A. These port names can differ from the names of the internal nets and variables specified in the definition of module B.

[187] The syntax and semantics of module port lists are described in IEEE 1364-2001 12.3.1 and 12.3.2.

[188] If a port expression in a module port list is an expression other than a simple identifier, then each port reference in that expression shall denote a port of a scalar logic type or a vector type.

### 3.5.1 Port declarations

*NOTE: This section updates IEEE 1364-2001 clause 12.3.3 to take into account the new data types and to introduce reference ports.*

[189] Each port expression in the list of ports for the module declaration shall also be declared in the body of the module as one of the following port declarations: **input**, **output**, **inout** (bidirectional), or **ref** (reference). This is in addition to any other object declaration for a particular port (for example, a **reg** or **wire** declaration).

```

inout_declaration ::=
    inout [ net_type ] [ type_specifier ] list_of_port_definitions

input_declaration ::=
    input [ net_type ] [ type_specifier ] list_of_port_definitions

output_declaration ::=
    output [ net_type ] [ type_specifier ] list_of_port_definitions
  | output [reg] [ net_type ] [ type_specifier ]
    list_of_variable_port_definitions

reference_declaration ::=
    ref [ type_specifier ] list_of_port_definitions

list_of_port_definitions ::=
    port_definition { , port_definition }

port_definition ::=
    port_identifier [ array_type_specifier ]

list_of_variable_port_definitions ::=
    variable_port_definition { , variable_port_definition }

variable_port_definition ::=
    port_identifier [ array_type_specifier ] [ = constant_expression ]

```

- [190] If an input, output, or bidirectional port declaration does not include a type specifier or a net type, then the port can be again declared in a net or variable declaration. In this case, the types described in the two declarations shall be identical, as defined in 2.11.
- [191] If a port declaration includes a type specifier or a net type, then the port is considered completely declared, and it shall be an error for the port to be declared again as a variable or net declaration. Because of this, all other aspects of the port shall be declared in such a port declaration.
- [192] A reference port is completely declared in the list of port declarations of the module; a reference port cannot appear in a port expression, nor can a reference port be redeclared.
- [193] A signed attribute can be attached to one or more vector ranges of a port declaration, or to one or more vector ranges of the corresponding net or variable declaration, or to one or more vector ranges of both. If either the port declaration or the net or variable declaration declares a given vector range as signed, then the corresponding vector range of the other shall also be considered signed.
- [194] Implicit nets shall be considered unsigned. Nets connected to ports without an explicit net declaration shall be considered unsigned, unless the port is declared as signed.
- [195] Implementations may limit the maximum number of ports in a module definition, but the limit shall be at least 256.

### 3.5.2 Reference ports

- [196] A reference port provides a simple identifier for accessing a variable or event that is declared elsewhere. Use of a reference port name is equivalent to a direct reference to the connected item itself.
- [197] A reference port that is declared with the keyword **event** is a named event. A reference event port shall be connected to a named event or to another reference event port.
- [198] The other form of reference port is a reference variable port. If a type specifier is not given, the type of the reference port is **logic**. A reference port that is a variable shall be connected to a variable, including another

reference variable port, or to an element of a variable. Two types in a reference port connection are compatible only if the types are identical.

### Usage:

- [199] You can think of a reference port as an abbreviated form of a hierarchical reference to a variable or an event. In the code fragments below, module 'bottom' has a reference port, 'raise\_error, that can be used to directly read and modify the variable 'error\_status' in module 'top'.

```

module top;

    bool error_status = 1'b0;
    wire w;

    bottom b1 (w, error_status);

    always @(error_status) ...

endmodule

module bottom ( output outport, ref bool raise_error );
    always
    begin
        ...
        raise_error = 0'b1;
    end
endmodule

```

### 3.5.3 Input, output, and inout port connections

*NOTE: This section updates IEEE 1364-2001 clauses 12.3.8, 12.3.9, and 12.3.10 to take into account the new data types and the new net type **wone**.*

- [200] A port of a module can be viewed as providing a link or connection between two items (nets, variables, expressions, etc.), one internal to the module instance and one external to the module instance.
- [201] The item receiving the value through the port (the internal item for inputs, the external item for outputs) shall be a structural net expression. The item that provides the value can be any expression.
- [202] A *structural net expression* is one of the following kinds of expression:
1. A net
  2. A constant bit-select of a vector net
  3. A part-select of a vector net
  4. A net of a primitive type
  5. A member of a composite net
  6. An element of an array net
  7. A concatenation of structural net expressions, each of which has a vector or scalar logic type

- [203] A port that is declared as input but used as an output or inout may be coerced to inout. Similarly, a port that is declared as output but used as input or inout may be coerced to inout. In either case, if the port is not coerced to inout, a warning shall be issued.

### 3.5.3.1 Port connection rules

- [204] The following rules apply to connections to input, output, and inout ports:
1. An input or inout port shall be a net.
  2. Each port connection shall be a continuous assignment of source to sink, where one connected item shall be a signal source the other shall be a signal sink. The assignment shall be a continuous assignment from source to sink for input or output ports. The assignment is a nonstrength reducing transistor connection for inout ports. Only nets or structural net expressions shall be the sinks in the assignment.
  3. External variables and expressions other than structural net expressions shall not be connected to the output or inout ports of a module.

### 3.5.3.2 Net types resulting from dissimilar port connections

- [205] When different net types are connected through a module port, the nets on both sides of the port can take on the same net type. The resulting net type can be determined as shown in Table 6. In the table, *external net* means the net specified in the module instantiation, and *internal net* means the net specified in the module definition. The net whose type is changed is said to be the *dominated net*. It is permissible to merge the dominating and dominated nets into a single net, whose type shall be that of the dominating net. The resulting net is called the *simulated net*, and the dominated net is called a *collapsed net*.
- [206] The simulated net shall take the delay specified for the dominating net. If the dominating net is of the net type **triereg**, any strength value specified for the triereg net shall apply to the simulated net.

**Table 6 – Net types resulting from dissimilar port connections**

Internal Net	External Net								
	wire tri	wone	wand triand	wor trior	trireg	tri0	tri1	supply0	supply1
wire tri	ext	ext	ext	ext	ext	ext	ext	ext	ext
wone	int	ext	int warn	int warn	int warn	int warn	int warn	ext	ext
wand triand	int	ext warn	ext	ext warn	ext warn	ext warn	ext warn	ext	ext
wor trior	int	ext warn	ext warn	ext warn	ext warn	ext warn	ext warn	ext	ext
trireg	int	ext warn	ext warn	ext warn	ext	ext	ext	ext	ext
tri0	int	ext warn	ext warn	ext warn	int	ext	ext warn	ext	ext
tri1	int	ext warn	ext warn	ext warn	int	ext warn	ext	ext	ext
supply0	int	int	int	int	int	int	int	ext	ext warn
supply1	int	int	int	int	int	int	int	ext warn	ext

**KEY:**  
ext = The external net type is used  
int = The internal net type is used  
warn = A warning is issued

### 3.5.3.3 Type compatibility at a port connection

[207] Type compatibility at a port connection is determined by the topology of the types, rather than by the names, if any, of the types involved.

[208] Two types shall be considered compatible for a port connection of an input, output, or inout port of one of the following statements is true:

1. Both types are the same primitive type.
2. Both types are scalar logic types.

3. Both types are vector types.
4. One type is a scalar logic type and the other type is a vector type.
5. Both types are handle types, and the handle object types are identical.
6. Both types are array types with the same element type and dimensionality, and corresponding dimensions have the same width.
7. Both types are structure types with the same number of members, and members at corresponding positions in the lists of member declarations have the same type.
8. Both types are a union of vector types with the same width and scalar logic type.

### Usage:

[209] Conceptually, two types are compatible in a connection to an input, output, or inout port if they are logic or vector ports, or if their values have the same size, shape, element layout, and primitive qualities. Names do not matter, nor does the signed property. Width matters, but the values of the range bounds themselves do not matter.

[210] The following pairs of statements declare objects that are compatible for connection:

```

inout bool [3:0] vector1;
inout logic signed [7:0] vector2;

inout bool [7:0] logic1;
inout logic logic2;

inout [7:0] array1 [127:0];
inout [7:0] array2 [1:128];

inout struct { int elem1; real elem2; } struct1;
inout struct { int val1; real val2; } struct2;

inout enum { red, blue, yellow } primary_color1;
inout [1:0] primary_color2;

```

[211] The following pairs of statements declare objects that are not compatible for connection:

```

// Different primitive types, and both are not logic
inout integer number1;
inout real number2;

// Corresponding dimensions have different widths
inout array1 [127:0] [7:0];
inout array2 [1:8] [1:128];

// Corresponding element types are not the same
inout struct { int elem1; real elem2; } struct1;
inout struct { real elem1; int elem2; } struct2;

```

### 3.5.3.4 Connecting signed values through ports

[212] The signed property of a vector range shall not cross hierarchy. In order to have the signed property apply throughout the hierarchy, the signed property must be applied to the object declarations (either directly through the signed **keyword** or indirectly through a type specifier) at each of the different levels of

hierarchy. Any expression on a port shall be treated as any other expression in an assignment. The expression shall be typed, sized, evaluated and the resulting value assigned to the object on the other side of the port using the same rules as an assignment.

### 3.6 Tasks and function arguments

- [213] Tasks and functions are described in IEEE 1364-2001 Section 10.
- [214] Task and function argument declarations have optional type specifiers that determine the types of the arguments. In the absence of a type specifier, type **logic** is inferred.
- [215] A function declaration has an optional type specifier to define the type of its return value. If none is specified then the function returns a 1-bit value of type **logic**.

```
tf_input_declaration ::=
    input [ reg ] [ type_specifier ] list_of_port_definitions

tf_output_declaration ::=
    output [ reg ] [ type_specifier ] list_of_port_definitions

tf_inout_declaration ::=
    inout [ reg ] [ type_specifier ] list_of_port_definitions

function_declaration ::=
    function [ automatic ] [ type_specifier ] function_identifier;
    function_item_declaration { function_item_declaration }
    function_statement
    endfunction
```

- [216] The expressions in a task enable statement shall be assignment compatible with the corresponding task arguments, as defined in 5.1. Similarly, the expressions in a function call shall be assignment compatible with the corresponding function arguments, as defined in 5.1.

## 4 Expressions

[217] Expressions are described in IEEE 1364-2001 Section 4.

### 4.1 Boolean literals

[218] Literal values for **bool** type objects are binary literals as described in IEEE 1364-2001 2.5.

### 4.2 Vector literals

[219] Vector literals provide flexibility in specifying literal values for use with vector objects and operators.

```

vector_literal ::=
    size' { vector_literal_constructors }
  | type_identifier' { vector_literal_constructors }

vector_literal_constructors ::=
    number { , number }
  | number { , number } default binary_number
  | default binary_number

```

[220] The size of a vector literal is determined by the specified size or type. The type identifier, if specified, shall denote a vector type. A number in a vector literal shall not be a real number.

[221] The final optional **default** indicates that all remaining elements take on the value specified. The default value must be a scalar **bool** or **logic** value.

#### Usage:

[222] Examples of vector literals include:

```

1'b1           => 1
5'bx          => xxxxx
16' { 4'hf, 4'h0, default 1'b1 } => 1111_0000_1111_1111
integer' { -1 }

```

[223] The use of a type identifier in front of a composite expression indicates that the size of the expression should be taken from that type. This allows composite expressions to be expressed with respect to the size of the type so if the type is redefined the literal automatically assumes the new size. Combined with the use of **default** this provides a powerful , flexible initialization mechanism.

### 4.3 Concatenations

[224] A concatenation is the joining together of bits or elements resulting from two or more expressions. Concatenations can be used to construct expressions with the topology of a vector, array, or structure type.

- [225] The characters that enclose the concatenation and separate the expressions within the concatenation reflect topology of the concatenation. A vector concatenation is delimited by square brackets, and the concatenation expressions are separated by commas. An array concatenation is delimited by square brackets, and the concatenation expressions are separated by semicolons. A structure concatenation is delimited by curly braces, and the concatenation expressions are separated by semicolons.

```

constant_primary ::=
    ...
    | constant_array_concatenation
    | constant_structure_concatenation

primary ::=
    ...
    | array_concatenation
    | structure_concatenation

constant_array_concatenation ::=
    [ type_identifier ]
    '[ constant_expression { ; constant_expression } ]

constant_structure_concatenation ::=
    [ type_identifier ]
    '{ constant_expression { ; constant_expression } }

array_concatenation ::=
    [ type_identifier ] '[ expression { ; expression } ]

structure_concatenation ::=
    [ type_identifier ] '{ expression { ; expression } }

```

- [226] The optional type identifier can be used to specify the topology of the concatenation if the element expressions are ambiguous. For example, the type identifier can be used to indicate that the scalar logic type of the element expressions is two-state. It shall be an error if one of the expressions in the concatenation is not compatible with the corresponding element type.

### Usage:

- [227] The following example shows how to use array and structure concatenations to connect a port that has a structure type with a nested array:

```

typedef struct {
    real r;
    logic a[2:0];
} struct_s;

module bottom ( input struct_s s );
endmodule

module top;
    real t_r;
    wire x, y, z;

    bottom b ( '{ t_r ; '[ x; y; z ] } );

endmodule

```

## 4.4 Operators

- [228] Operators are described in IEEE 1364-2001 4.1.

- [229] Operators for manipulating dynamically allocated objects are described in **Error! Reference source not found.**
- [230] Table 7 shows operator precedence. The unary \* operator is included with the other unary operators. The top row indicates the relative precedence of array subscripts and member selection.

**Table 7 – Operator precedence**

() [] . ->	Highest precedence
+ - ! * ~ (unary)	
**	
* / % (binary)	
+ - (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& ~&	
^ ^~ ~^	
~	
&&	
?: (conditional operator)	Lowest precedence

## 5 Assignment

- [231] The assignment is the basic mechanism for placing values into nets and variables. The two basic forms of assignment are described in IEEE 1364-2001 Section 6.

### 5.1 Assignment compatibility

- [232] The value on the right-hand side of an assignment shall have a type that is assignment compatible with the left-hand side.

- [233] For a composite type, the topology of the type is central to determining its assignment compatibility. The topology refers to the “shape” of a type. This places no restrictions on how an implementation actually allocates the object; re-ordering, padding, or other optimizations are not restricted or required. Topology refers instead to a conceptual layout that begins with the first element declared in the type and proceeds through the elements sequentially. If a type is an array, structure, or union, then the topology is defined by recursion into the definition of the element or member, until primitives, handles, or vectors are found. The correspondence between elements in the conceptual layout determines which rvalues are transferred to which lvalues, and whether or not a conversion is required for a given element.

- [234] In the following cases the types are assignment compatible:

1. Both types are the same primitive type. No conversion is required.
2. Both types are handles to events. No conversion is required.
3. Both types are untyped handles. No conversion is required.
4. Both types are typed handles, and the handle object types are identical. No conversion is required.
5. Real types and logic types (scalar or vector) are assignment compatible. Values shall be converted as described in IEEE 1364-2001 3.9.
6. All vector types and scalar logic types are completely assignment compatible with one other. The IEEE 1364-2001 rules for extending or truncating assignments to variables and nets apply in all cases. If the right-hand side has a four-state logic representation and the left-hand side has a two-state logic representation then each four-state value shall be converted to the corresponding two-state value according to the rules in 3.3.4.
7. Both types are array types with the same kind of element type and dimensionality, and corresponding dimensions have the same width, and the element types are assignment compatible. The need for conversion is determined on an element-by-element basis in a top-down fashion.
8. Both types are structure types with the same number of members, and members at corresponding positions in the lists of member declarations have the same kind of type, and corresponding member types are assignment compatible. The need for conversion is determined on an element-by-element basis in a top-down fashion.
9. Both types are a union of vector types with the same width and scalar logic type. No conversion is required.

## 5.2 Value conversions

- [235] An assignment may involve a conversion of a value from the right-hand side.
- [236] IEEE 1364-2001 3.9.2 describes conversions between real numbers and integers.
- [237] Table 6 shows the conversion of a four-state value to a two-state value.

**Table 6 – Four-state to two-state value conversion**

<b>Four-state</b>	<b>Two-state</b>
<b>0</b>	<b>0</b>
<b>1</b>	<b>1</b>
<b>x</b>	<b>0</b>
<b>z</b>	<b>0</b>

## 6 Dynamic objects

This section describes the operations for allocating and accessing dynamically allocated objects.

### 6.1 Memory Allocation

- [238] Memory for an object described by a handle is allocated by calling the predefined function **new**. The **new** function takes a single optional argument that specifies the initial value to be assigned to the object if it is a data object. The **new** function returns an untyped handle.
- [239] If the allocated object is a data object then the optional argument, if provided, must be an expression that is used as the initialization value of the object. If no argument is passed to **new** then the allocated object takes on the default value for the type of the object. If the allocated object is a named event then the **new** function shall not be called with the optional argument.
- [240] The type of object allocated is determined first by the context in which the call occurs. Contexts in which the type is determined in this way include: assignment to a typed handle object in any assignment statement, default value specification, task/function argument, parameter association, or port association. In each of these cases the allocated object takes on the type and topology of the object to which the handle is being assigned or associated. If **new** is used in one of these contexts and a default value is provided, then the topology is first computed by context and then the value is assigned. The default value must be assignment compatible with the allocated object.
- [241] If the topology of the object allocated is not determined in one of the contexts above then the topology of the argument to **new** is used to determine the topology of the allocated object. This occurs primarily when an allocation is occurring directly to an untyped handle.

#### Usage:

- [242] Some handle types and objects:

```
typedef list_s;
typedef handle list_s list_h;
typedef struct {
    logic [31:0] addr;
    logic [31:0] data;
    list_h      next // A handle to another object of this type
} list_s;

// object allocated with default value
list_h l_h = new;

// default value provided
list_h ll_h = new( '{ 32'b0; 32'bz; null } );
```

- [243] There is no restriction on creating handles to handle types:

```
typedef signed [63:0] logic long_int;
typedef handle long_h long_h_h;

// create a handle type to a handle type
handle long_int long_h;
long_h_h l_hh;

l_hh = new; // allocate a handle to a long handle
```

```
*l_h = new;    // allocate a handle to a long_int
**l_h = 64'b6 // assign a value to the dynamic object
```

## 6.2 Null values

- [244] The special literal value **null** represents the value of a handle that does not refer to any object and can be used in assignment to any handle to indicate that it does not refer to any object.
- [245] The **null** value can also be used with equality and inequality operators to test for a handle that refers to no object. The value **null** is interpreted as a boolean false if used in a conditional context.

## 6.3 Initialization

- [246] Handles are initialized to the value **null** by default.

### Usage:

```
// initialized to null by default
list_h head;

// explicit initialization to null is allowed for the careful
list_h tail = null;

// default values of parameters of new are applied
list_h element = new;

// explicit parameters to new
list_h element_2 = new('{ 32'b1 ; 32'b0 ; null });
```

## 6.4 Operators

- [247] The following sections describe operations involving handle objects and expressions.

### 6.4.1 Dereferencing

- [248] The unary **\*\***, binary **.**, and binary **->** operators provide access to a dynamically allocated object.
- [249] The unary **\*\*** operator dereferences a handle for access to the contents of the dynamic object. The unary **\*\*** operator has the same precedence as other unary operators.
- [250] In the case of composite types it is often desirable to first dereference a handle and then refer to a member of the resulting structure. Due to the relative precedence of **.** and **\*\***, parentheses in the expression are required. The **->** operator is a shorthand for a dereference operation followed by member selection. This operator can be used for struct, union, and tagged vector members.
- [251] It shall be an error to perform a dereference of a handle with the value **null**.
- [252] It shall be illegal to dereference an untyped handle.

### Usage:

- [253] In the case of composite types it is often desirable to first dereference a handle and then refer to a given element in the structure or array as in the example below:

```

typedef handle integer integer_h; // handle type to an integer

integer i;
integer_h i_h;

begin
    i_h = new('b0); // allocate an integer and initialize to zero
    i = *i_h; // assign the value zero to the integer i
end
handle list_s l_h = new;

...
(*l_h).addr
...

```

- [254] Due to the relative precedence of “.” and “\*”, the parenthesis in the expression (\*l\_h).addr are required. Because this is such a common operation, the C-like ‘->’ operator is also introduced to indicate a dereference operation followed by member selection. This operator can be used for struct, union, and tagged vector members. The shorthand for the expression above is:

```
l_h->addr;
```

- [255] Note that this uses the same symbol as the event triggering operator. Since an event trigger is a statement and this is an operator, there is no syntactic ambiguity.
- [256] There is no equivalent shortcut provided for vector or array indexing however note that the same precedence issue exists and therefore parenthesis are required to first dereference a handle to a vector or array and then index the object.

```

typedef signed [63:0] logic long_int;
handle long_int long_h;
long_h l_h;
long_int l_i;
logic l;

l_h = new;

...
l = (*l_h)[63] ) ) // parenthesis are required here
l_i = *l_h; // assigns the entire value to l_i
...

```

- [257] A continuing example shows an array of handles where the parenthesis are not required.

```

// Array of 64 handles to long integers
typedef long_h long_a [63:0];
long_a l_a;
long_i l_i;

// contents of 63rd element is assigned to l_I
l_i = *l_a[63];

```

#### 6.4.2 Sensitivity

- [258] If an event control refers to a handle object, then the event control refers only to the value of the handle, not the object referred to. In order to be sensitive to changes in the value of the object referred to a dereference of the handle must be used. Handle dereferences in sensitivity may also refer to members or indexed elements.

- [259] When an event control or wait statement contains a dereference of an object the object referred to is computed once when the event control or wait statement is executed. If the handle changes while waiting the object that is being waited on is unaffected. If all handles to an object have been set to **null** and a process is waiting on the object, then the object may be deallocated implicitly and the event control or wait will never unblock.

**Usage:**

- [260] To wait on the handle changing value:

```
handle list_h head;
if (head == null)
    @(head) // waits until head becomes non-null
```

- [261] To wait on a member of the structure (or array if subscripting is used):

```
if (head != null)
    @(head->data) // waits until the data field changes
```

- [262] To wait for a change in any field of the structure (or vector/array if the object is an vector/array):

```
if (head != null)
    @(*head) // waits until any element of head changes
```

An untyped handle shall not be used in an event control or wait statement.

### 6.4.3 Assignment of handles

- [263] Assignment between handles creates multiple references to the same object. Two handles are assignment compatible if and only if the types to which they refer are topologically identical, or if one or both of the types is an untyped handle.
- [264] If the left hand side of an assignment is an untyped handle then any typed or untyped handle can occur on the right hand side of the assignment. When an assignment occurs to an untyped handle, the untyped handle must store the topology of the object now referred to by the untyped handle. In this way untyped handles always internally store the topology of the object to which they refer.
- [265] If the right hand side of an assignment is an untyped handle and the left hand side is a typed handle, then the topology of the object referred to by the untyped handle must be topologically identical to the type referred to on the left-hand side of the assignment.
- [266] Untyped handles are always assignment compatible.
- [267] Any handle can also be assigned the literal value **null**. Assignment to **null** indicates that the handle no longer refers to any object. In this case an untyped handle has no topology stored. Assignment to a typed handle from an untyped handle with a **null** value is allowed.

**Usage:**

- [268] Some legal handle assignments:

```

list_h l_h = new;           // handle to a list_s
list_h ll_h = new;        // second handle to a list_s

integer_h i_h = new;      // handle to an integer object

handle u_h, uu_h;        // two untyped handle objects

// example 1
l_h = ll_h;              // handles refer to the same type

// example 2
u_h = l_h;               // typed handle can be assigned to untyped handle

// example 3
u_h = uu_h;              // two untyped handles can always be assigned

// example 4
begin
    u_h = l_h;           // can always assign to an untyped handle
    ll_h = u_h;          // stored type in u_h is identical to ll_h
end

```

[269] Some illegal handle assignments:

```

// example 5
i_h = l_h;              // ERROR: handles refer to different types

// example 6
begin
    u_h = l_h;           // LEGAL: can always assign to an untyped handle
    i_h = u_h;           // ERROR: stored type in u_h is different
                        // than integer handle
end

```

#### 6.4.4 Comparison of handles

[270] All of the logical equality, logical inequality, case equality and case inequality comparison operators are defined on **handle** objects. Equality returns 1'b1 if the two handles refer to the same object or if both handles are **null**, otherwise 1'b0. Both typed and untyped handles can be compared. Additionally a handle can be compared against the literal value **null**. A handle is equal to **null** if it refers to no object.

#### 6.4.5 \$htest system function

[271] In order to safely assign from an untyped handle that could potentially refer to multiple different types at runtime, the \$htest system function is introduced. The \$htest system function takes two handles as arguments. If the two handles are assignment compatible, it returns the value 1'b1. If the two handles are not assignment compatible, then it returns the value 1'b0.

#### Usage:

```

handle u_h;
handle list_s l_h;
handle integer i_h;

if ($hassign(i_h, u_h))
begin
    // deal with this value as an integer
end

```

```
        i_h = u_h; // this assignment is guaranteed type compatible
        ...
    end
    else if ($hassign(l_h, u_h))
    begin
        // deal with this value as a list
        l_h = u_h;
        ...
    end
```

#### 6.4.6 Implicit deallocation

[272] All dynamically allocated objects may be implicitly deallocated when they are no longer referenced by any handles. The algorithm for or timing of implicit deallocation is implementation defined.



## Annex A: Syntax summary

This annex provides a summary of the syntax for the Verilog type extensions that are proposed in this document. Some of the productions extend the IEEE 1364-2001 productions to include data type declarations as well as a new kind of net and a new kind of port. Ellipses in extended productions represent the rule as currently defined in the standard.

Some productions make use of the IEEE 1364-2001 productions for `identifier`, `net_identifier`, `variable_identifier`, `parameter_identifier`, `port_identifier`, `range`, `dimension`, `expression`, `constant_expression`, `drive_strength`, `charge_strength`, and `delay3`; no new syntax or semantics are intended for these constructs.

### VERILOG 2001 GRAMMAR EXTENSIONS

```

description ::=
    ...
    | data_type_declaration

module_or_generate_item ::=
    ...
    | data_type_declaration

block_item_declaration ::=
    ...
    | data_type_declaration

port_declaration ::=
    ...
    | { attribute_instance } shared_variable_declaration

net_type ::=
    ...
    | wone

constant_primary ::=
    ...
    | constant_array_concatenation
    | constant_structure_concatenation

primary ::=
    ...
    | array_concatenation
    | structure_concatenation

```

### IDENTIFIERS

```

type_identifier ::= identifier

```

### DATA TYPE DECLARATIONS

```

data_type_declaration ::=
    incomplete_data_type_declaration
    | complete_data_type_declaration

```

### INCOMPLETE TYPE DECLARATIONS

```

incomplete_data_type_declaration ::=
    typedef list_of_incomplete_type_identifiers ;

list_of_incomplete_type_identifiers ::=

```

```
type_identifier { , type_identifier }
```

#### COMPLETE TYPE DECLARATIONS

```
complete_data_type_declaration ::=
    typedef type_specifier list_of_type_declaration_names ;

list_of_type_declaration_names ::=
    type_declaration_name { , type_declaration_name }

type_declaration_name ::=
    identifier [ array_type_specifier ]

type_specifier ::=
    type_name
    | predefined_type_name
    | primitive_type_specifier
    | user_defined_type_specifier

type_name ::=
    [ scalar_logic_type ] type_identifier
```

#### PRIMITIVE TYPES

```
primitive_type_specifier ::=
    scalar_logic_type | real

scalar_logic_type ::=
    bool | logic
```

#### PREDEFINED TYPE NAMES

```
predefined_type_name ::=
    integer | time | realtime | long | int | shortint | char
```

#### USER-DEFINED TYPES

```
user_defined_type_specifier ::=
    handle_type_specifier
    | union_type_specifier
    | structure_type_specifier
    | vector_type_specifier
```

#### HANDLE TYPES

```
handle_type_specifier ::=
    handle type_specifier
    | handle *
    | handle event
```

#### UNION TYPES

```
union_type_specifier ::=
    union { member_declarations }
```

#### STRUCTURE TYPES

```
structure_type_specifier ::=
    struct { member_declarations }
```

**ARRAY TYPES**

```
array_type_specifier ::=
    dimension { dimension }
```

**VECTOR TYPES**

```
vector_type_specifier ::=
    indexed_vector_type_specifier
    | tagged_vector_type_specifier
    | enumerated_vector_type_specifier
```

```
vector_ranges ::=
    vector_range { vector_range }
```

```
vector_range ::=
    [ signed ] range
```

```
indexed_vector_type_specifier ::=
    [ type_specifier ] vector_ranges
```

```
tagged_vector_type_specifier ::=
    [ scalar_logic_type ] vect [ vector_ranges ]
    { member_declarations }
```

```
enumerated_vector_type_specifier ::=
    [ scalar_logic_type ] enum [ vector_range ]
    { enum_constant_list }
```

```
enum_constant_list ::=
    enum_constant_decl { , enum_constant_decl }
```

```
enum_constant_decl ::=
    identifier [ = constant_expression ]
```

**MEMBER DECLARATIONS**

```
member_declarations ::=
    member_declaration { member_declaration }
```

```
member_declaration ::=
    type_specifier list_of_type_declaration_names ;
```

**NET DECLARATIONS**

```
net_declaration ::=
    net_type [ signed ] [ delay3 ] list_of_net_definitions ;
    | net_type [ vector | scalared ] type_specifier
      [ delay3 ] list_of_net_definitions ;
    | net_type [ drive_strength ] signed
      [ delay3 ] list_of_net_decl_assignments ;
    | net_type [ drive_strength ] [ vector | scalared ] type_specifier
      [ delay3 ] list_of_net_decl_assignments ;
    | trireg [ charge_strength ] [ signed ]
      [ delay3 ] list_of_net_definitions ;
    | trireg [ charge_strength ] [ vector | scalared ] type_specifier
      [ delay3 ] list_of_net_definitions ;
```

```
list_of_net_definitions ::=
    net_definition { , net_definition }
```

```

net_definition ::=
    net_identifier [ array_type_specifier ]

list_of_net_decl_assignments ::=
    net_decl_assignment { , net_decl_assignment }

net_decl_assignment ::=
    net_definition = expression

```

**VARIABLE DECLARATIONS**

```

variable_declaration ::=
    reg list_of_variable_definitions ;
    | [ reg ] type_specifier list_of_variable_definitions ;

list_of_variable_definitions ::=
    variable_definition { , variable_definition }

variable_definition ::=
    variable_identifier [ array_specifier ] [ = expression ]

```

**PARAMETER DECLARATIONS**

```

local_parameter_declaration ::=
    localparam [ signed ] list_of_param_assignments ;
    | localparam type_specifier list_of_typed_param_assignments ;

parameter_declaration ::=
    parameter [ signed ] list_of_param_assignments ;
    | parameter type_specifier list_of_typed_param_assignments ;

list_of_param_assignments ::=
    param_assignment { , param_assignment }

param_assignment ::=
    parameter_identifier = constant_expression

list_of_typed_param_assignments ::=
    typed_param_assignment { , typed_param_assignment }

typed_param_assignment ::=
    parameter_identifier [ array_type_specifier ] = constant_expression

```

**PORT DECLARATIONS**

```

inout_declaration ::=
    inout [ net_type ] [ type_specifier ] list_of_port_definitions

input_declaration ::=
    input [ net_type ] [ type_specifier ] list_of_port_definitions

output_declaration ::=
    output [ net_type ] [ type_specifier ] list_of_port_definitions
    | output [ reg ] [ net_type ] [ type_specifier ]
      list_of_variable_port_definitions

reference_port_declaration ::=
    ref [ type_specifier ] list_of_port_definitions
    | ref event list_of_port_identifiers

list_of_port_definitions ::=

```

```

    port_definition { , port_definition }

port_definition ::=
    port_identifier [ array_type_specifier ]

list_of_variable_port_definitions ::=
    variable_port_definition { , variable_port_definition }

variable_port_definition :=
    port_identifier [ array_type_specifier ] [ = constant_expression ]

list_of_event_identifiers ::=
    port_identifier { , port_identifier }

```

**TASK AND FUNCTION ARGUMENTS**

```

tf_input_declaration ::=
    input [ reg ] [ type_specifier ] list_of_port_definitions

tf_output_declaration ::=
    output [ reg ] [ type_specifier ] list_of_port_definitions

tf_inout_declaration ::=
    inout [ reg ] [ type_specifier ] list_of_port_definitions

function_declaration ::=
    function [ automatic ] [ type_specifier ] function_identifier ;
    function_item_declaration { function_item_declaration }
    function_statement
    endfunction

```

**CONCATENATIONS**

```

constant_array_concatenation ::=
    [ type_identifier ]
    '[' constant_expression { ; constant_expression } ]

constant_structure_concatenation ::=
    [ type_identifier ]
    '{ constant_expression { ; constant_expression } }

array_concatenation ::=
    [ type_identifier ] '[' expression { ; expression } ]

structure_concatenation ::=
    [ type_identifier ] '{ expression { ; expression } }

```

**VECTOR LITERALS**

```

vector_literal ::=
    size '{ vector_literal_constructors }
    | type_identifier '{ vector_literal_constructors }

vector_literal_constructors ::=
    number { , number }
    | number { , number } default binary_number
    | default binary_number

```

## Annex B: Detailed Examples

### B.1 Multiple ranges with different sign properties

- [273] The power of multiple ranges and an individual **signed** specification on each range is best demonstrated by example. First, define names for vector types that represents a short integer and a longer integer:

```
typedef logic signed [15:0] shortinteger;
typedef logic signed [31:0] longerinteger;
```

- [274] Now, define a name for a third vector type that holds many of these, perhaps to model a bus capable of carrying two of these objects in parallel. Note that the [1:0] dimension is not **signed**.

```
typedef shortinteger [1:0] sintbus_t;
```

- [275] If we create some simple assignments we can see that the fact that the sintbus\_t type preserves the fact that that sub-range of shortinteger is signed becomes very important.

```
reg shortinteger sint; // Our 16-bit signed integer
reg longerinteger lint; // Our 32-bit signed integer
reg sintbus_t sintbus; // Our 32-bit wide vector of 2 shortint
reg [63:0] hugereg; // An even wider unsigned register

lint = sint;
// sign extends the value of sint into lint

lint = sintbus[0]
// sign extends the value of sintbus[0] into lint

hugereg = sintbus;
// does not extend the sign of sintbus into hugereg
// because subscript used is not a signed range
```

### B.2 Using vector type declarations to define a packet type

- [276] This example defines a tagged vector for the implementation of a packet type. Because the elements of a tagged vector have the same scalar logic type, the scalar logic type is specified at the level of the packet type itself rather than at the level of its members.

```
// Structure with elements of undetermined logic type
typedef [7:0] u_port_no_t;
typedef [127:0] u_payload_t;

typedef enum {
    PACKET_OK, PACKET_ERROR
} u_status_e;

typedef vect {
    u_port_no_t src_port, dest_port; // note element list
    u_status_e status;
    u_payload_t payload;
} u_packet_t;

// object declarations determining type of all elements
// A 2-state variable kind packet
reg bool u_packet_t bool_packet;

// A 4-state net kind packet
```

```
wire logic u_packet_t logic_packet;
```

### B.3 Using a union of vectors for alternate bit field layouts

[277] The most powerful use of a union of vector types is to express alternate bit field layouts of data in a vector. The following example is taken from the 32-bit Sparc instruction set:

```
// Typedef for total width of 32-bit instruction
typedef bool [31:0] sparc_vector_t;

typedef [4:0] reg_no_t; // register numbers are 5-bits

// Typedef for the first 19 bits of an instruction
// These are common in many Sparc instructions
typedef vect [31:13] {
    [1:0] op1; // first half of opcode
    reg_no_t rd; // destination register
    [5:0] op2; // second half of opcodes
    reg_no_t rs1; // first source register
    [0:0] is_immed; // bit indicating type of second operand
} opcode_dest_src_t;

// The bottom 13 bits can be either another register
// or an immediate 13-bit operand
// (Note the use of an embedded tagged vector without a typedef)
typedef union {
    vect {
        [12:5] ignore; // these bits are unused
        reg_no_t rs2; // second source register
    } s;
    [12:0] immed;
} op2_u;

// Declare the actual full instruction
// (Note the use of a vector range to size this vector)
typedef vect [31:0] {
    opcode_dest_src_s ods;
    op2_u op2;
} sparc_instr_s;

// Code extracting the second operand as a reg would look like
sparc_instr_s instr;
reg_no_t r;

if (instr.ods.is_immed == 1'b0) // operand is not immediate
    r = instr.op2.s.rs2;
```

### B.4 Different kinds of variable declarations

[278] The following declarations illustrate different ways to declare variables, from typical simple declarations to more complicated declarations involving user-defined types.

```
// Declare a scalar 2-state 1-bit variable
reg bool b2; // explicitly specify everything
bool b1; // reg is optional

// Declare a vector array of 32 4-state values
reg logic signed [31:0] v1; // explicitly specify everything
reg signed [31:0] v2; // logic is the default
reg integer v2; // 'integer' is a predefined type name
```

```

integer v2;                // reg is optional

// Create vector array type for doubleword,
// of an undetermined scalar logic type
typedef [1:0] [31:0] doubleword_t;

// Declare some 2-valued doubleword variables
reg bool doubleword_t dwb1;    // 2-valued double word variable
bool doubleword_t dwb2;        // reg is optional

// Declare some 4-valued doubleword variables
reg logic doubleword_t dwl1;   // 4-valued double word
doubleword_t dwl2;            // reg is optional, logic is default

// Create data structures for a queue
typedef queue_s;                // incomplete type for queue
typedef handle queue_s queue_h; // handle to incomplete type
typedef struct {
    handle payload;    // handle to any payload data
    queue_h prev;     // previous element
    queue_h next;     // next element
} queue_s;

reg queue_h head, tail; // declare head and tail handles for queue

```

## B.5 Infinite length FIFO

[279] This example shows a module that takes in a packet structure on its ports and then creates an infinite length FIFO internally.

```

typedef [7:0] port_no_t;
typedef [63:0] payload_t;

typedef bool vect {
    port_no_t src;
    port_no_t dest;
    payload_t payload;
} packet_t;

module packet_fifo
    (ref event in_ready,    // packet is ready
     input packet_t in_packet, // incoming packet
     ref event out_wanted, // produce an output
     output reg packet_t out_packet, // outbound packet
     output event out_avail); // output ready

/*
 * Create an infinite length fifo where objects come in the 'back'
 * of the fifo, and go out the 'front'
 *
 * Use a double linked list holding handles to
 * - the packet
 * - the next element toward the front of the list
 * - the next element toward the back of the list
 */
typedef struct {
    handle packet_t p_packet;
    handle packet_queue_t p_front, p_back;
} packet_queue_t;

/*
 * Declare the front and back of the queue,

```

```

    * implicitly initialize to null
    */
    handle packet_queue_t front, back;

    /*
    * When in_ready is triggered put a new element
    * on the head of the queue.
    */
    always @(in_ready)
    begin: in_block

        /*
        * Allocate a new packet to hold this entry and
        * initialize it to the incoming packet
        */
        handle packet_t tmp_packet = new(in_packet);

        /*
        * Allocate a new queue entry, initialized with this packet
        */
        handle packet_queue_t tmp_queue =
            new( '{ tmp_packet, null, null } ');

        /*
        * Link this packet into the queue
        *
        * Note this must be done carefully because someone might be
        * waiting on the queue 'front' below. 'front' should be the
        * last thing we assign to avoid a potential race
        */
        if (back)
        begin
            back->p_back = tmp_queue;
            tmp_queue->p_front = back;
        end

        back = tmp_queue;

        if (!front)
            front = tmp_queue;
    end

    /*
    * When an out_wanted event is triggered, then take the front
    * element and put it on the output of the fifo.
    * then trigger out_avail.
    *
    * (Note multiple out_wanted requests could be missed in this
    * simple example while waiting for an empty queue)
    */
    always @(out_wanted)
    begin

        if (!front) // nothing in the fifo, block until something is
            @(front) // wait for the handle to be non-null

        /*
        * Place the value at the front of the queue on the output
        */
        out_packet = *(front->p_packet);

        /*
        * Clear the handle to this packet
        * (so it can be deallocated)

```

```
    */
    front->p_packet = null;

    /*
    * Move the front pointer back one element if there is one
    */
    if (front->p_back)
    begin
        front = front->p_back;

        /*
        * Unlink the old front
        */
        front->p_front->p_back = null;
        front->p_front = null;
    end
    else // queue is now empty
    begin
        front->p_back = null;
        front = null;
        back = null;
    end

    /*
    *Go ahead and trigger the output
    */
    -> out_avail;

end
endmodule
```

## **Annex C: Possible Extensions**

[280] This annex briefly describes possible extensions to Verilog that are related to extensions in this proposal.

### **C.1 Associative arrays**

[281] An associative array is a dynamically sized array with indices that can be of any type. Element addresses are not restricted to integral expressions; element access is achieved through a lookup operation rather than relative position. Associative arrays would be useful for implementing hashes, content addressable memories (CAMs), testbench scoreboards, etc.

### **C.2 Untypedef**

[282] An “untypedef” capability could be added to control the visibility of type names by removing a type name from a name space.

### **C.3 Passing arguments by reference**

[283] Verilog task arguments are passed by value. An extension to Verilog task arguments would allow the arguments to be passed by reference as well. Arguments passed by reference would be treated as references to the variables in the task enabling statement. Operations like assignment could then affect the enabling context before the task returns.

### **C.4 Modules as ports**

A powerful extension to Verilog would be to allow a module to have ports of a module type. This extension would allow you to bundle different kinds of objects, such as nets and variables, into a single entity. In addition to allowing you to refer to a group of objects by a single name, ports that are modules would be capable of encapsulating functionality along with connectivity.

### **C.5 Timing for objects of a user-defined type**

[284] Verilog specify blocks and SDF annotation apply to logical objects – objects that have a scalar logic type or a vector type. Verilog timing could be extended to support objects with user-defined types, such as a handles and structures.

## Annex D: Rationale

### D.1 Naming the two-state type

- [285] The **bool** keyword was selected instead of ‘bit’ for numerous reasons. It is believed that fewer existing designs will have a conflict with the **bool** keyword. It also makes it simpler in this specification to say things like “The **bool** type can be represented in a single bit whereas the **logic** type must be represented in at least 2 bits”. If **bit** was used, this would become “The **bit** type can be represented in a single bit whereas the **logic** type must be represented in at least 2 bits.”

### D.2 “Address of” operator

- [286] Note that no “address of” operator has been proposed in this donation. This means that handles can only point to explicitly allocated objects. All memory allocation for objects created by the **new** operator is implicitly deallocated through implementation dependant garbage collection. The specific time at which garbage collection occurs is also implementation dependant.

### D.3 Type name visibility

- [287] It is an error if the same name is used in two or more typedef declarations that are visible in the same location and are not identical in the topology of the value they create. This rather convoluted rule is intended to handle the case where a ``include` makes a typedef visible multiple times in the same source stream and they are all identical. This is intended to be allowed. Overriding a typedef name by subsequent definition as can be done with macro definitions is not intended.

### D.4 Vector types

- [288] The existing IEEE 1364-2001 **reg** and **wire** types are both vector types. All vector types proposed in this donation can be represented in these types with the exception that the signedness of individual ranges might not be preserved and **bool** objects would become 4-state. This, in theory, makes it possible to write a translator from a system-level description that uses only vector types to IEEE 1364-2001 for processing in tools that do not handle the new language extensions.

### D.5 Associating the signed property with a type

- [289] Allowing a list of ranges and including the optional **signed** specification as a part of the range is a subtle extension of IEEE 1364-2001. In IEE 1364-2001 only a single range is allowed on vector declarations, and the signedness is always associated with the object, not the type. This change is 100% backward compatible in that the syntax is a superset of the 1364 syntax and an object with a single signed or unsigned range will have exactly the same semantics. However, it significantly extends the power of declarations by allowing vectors with multiple ranges and by allowing each one of those ranges to independently be signed or unsigned. Examples of this technique and its advantages are given in section 2.4.1.

### D.6 Enumeration type checking

- [290] The enumeration constant encoding scheme allows discontinuous ranges to be created. It also allows encodings that are not strictly increasing. These forms of ranges are explicitly allowed to provide maximum flexibility, however they create a situation where runtime range checking of enumeration values would be computationally expensive therefore runtime checking is not required. An implementation may choose to perform this checking or choose to perform it under user control.

- [291] This can already be accomplished in IEEE 1364-2001 using ``define`, but enumerations have the advantage of enforcing elaboration-time checking of the underlying element type, range, and duplication of values. They also provide the additional descriptive capability of declaring objects of these types to indicate that they are intended to only hold values of this type. Enumeration type declarations always define a vector type. There is no aggregate representation of an enumeration.

## **D.7 Restrictions on unions**

- [292] Member types in Verilog unions are more restricted than member types in more general programming languages. Since Verilog objects are not simply values (they can have fanout), having objects of different types and sizes in a single storage location is problematic; an implementation cannot determine which member of a union is in use at a given time.

## **Annex E: References**

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

IEEE Std 1497-1999, IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process

---