

SystemVerilog 3.0

Accellera's Extensions to Verilog[®]

Editor's note:

Draft 5 reflects all changes approved in the HDL++ meetings 16 through 19 (through March 18, 2002)

Legend:

—~~red strike through text~~ indicates text to be deleted

— **blue text** indicates text that was added

— **red text** indicates editor notes or things we need to consider at a future meeting

— All strike through text in draft 4 has been deleted. Any strike through text in this draft are for text in draft 4 that has been approved for deletion.

— All new (in blue) text in draft 4 has been changed to regular text. Any text in blue in this draft is new text added since draft 4.

Sponsor

Accellera

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

Copyright © 2002 by Accellera
1370 Trancas Street #163
Napa, CA 94558
Phone: (707) 251-9977
Fax: (707) 251-9877

This is an unapproved draft of a proposed Accellera Standard, subject to change. Use of information contained in the unapproved draft is at your own risk.

Do not copy, fax, reproduce, or distribute without written permission.

Table of Contents

Section 1	Introduction to SystemVerilog	1
Section 2	Lexical Conventions Literal Values.....	2
2.1	Introduction (informative)	2
2.2	Literal value syntax.....	2
2.3	Integer and logic literals	2
2.4	Real literals	2
2.5	Time literals	3
2.6	String literals.....	3
2.7	Array literals	3
2.8	Structure literals.....	3
Section 3	Data Types.....	5
3.1	Introduction (informative)	5
3.2	Data type syntax.....	6
3.3	Integer data types	6
3.4	Other basic data types	7
3.5	User-defined types	8
3.6	Enumerations	8
3.7	Structures and Unions	9
3.8	Casting	9
Section 4	Arrays.....	11
4.1	Introduction (informative)	11
4.2	Packed and unpacked arrays	11
4.3	Multiple dimensions	11
4.4	Part selects (slices).....	12
Section 5	Data Declarations	14
5.1	Introduction (informative)	14
5.2	Data declaration syntax.....	14
5.3	Constants.....	15
5.4	Variables	15
5.5	Scope and lifetime	15
5.6	Nets, regs, and logic.....	16
Section 6	Attributes.....	17
6.1	Introduction (informative)	17
6.2	Attribute syntax for interfaces	17
Section 7	Operators and Expressions.....	18
7.1	Introduction (informative)	18
7.2	Operator syntax.....	19
7.3	Assignment, incrementor and decrementor operations.....	19
7.4	Operations on logic and bit types	20
7.5	Real operators	20
7.6	Size.....	21
7.7	Sign	21
7.8	Operator precedence and associativity	21
7.9	Concatenation	22

Section 8	Procedural Statements and Control Flow	23
8.1	Introduction (informative)	23
8.2	Blocking and nonblocking assignments	24
8.3	Selection statements	25
8.4	Transition statements	26
8.5	Loops statements	26
8.6	Jump statements	26
8.7	Named blocks and statement labels	27
8.8	Processes	28
8.9	Disable	28
8.10	Delay and Event control	29
Section 9	State Machines	31
9.1	Introduction (informative)	31
9.2	State machine constructs	31
9.3	State declarations	33
9.4	Transition statements	34
9.5	Hierarchical and concurrent state machines	36
Section 10	Processes	39
10.1	Introduction (informative)	39
10.2	Level sensitive logic	39
10.3	Latch sensitive logic	40
10.4	Edge sensitive logic	40
10.5	Continuous assignments	40
10.6	Static Process labels	41
10.7	Dynamic processes	41
Section 11	Tasks and Functions	43
11.1	Introduction (informative)	43
11.2	Tasks	43
11.3	Functions	45
Section 12	Hierarchy	47
12.1	Introduction (informative)	47
12.2	The \$root top level	47
12.3	Module declarations	49
12.4	Nested modules	50
12.5	Port declarations	51
12.6	Time unit and precision	52
12.7	Module instances	53
12.8	Port connection rules	53
12.9	Name spaces	54
12.10	Hierarchical names	55
Section 13	Interfaces	56
13.1	Introduction (informative)	56
13.2	Interface syntax	57
13.3	Ports in interfaces	60
13.4	Modports	61
13.5	Tasks and Functions in Interfaces	65
13.6	Parameterized interfaces	70
13.7	Access without Ports	71

Section 14 Parameters	73
14.1 Introduction (informative)	73
14.2 Syntax	73
Section 15 Configuration libraries	75
15.1 Introduction (informative)	75
15.2 Libraries	75
15.3 Library map files	75
Section 16 System tasks and system functions	77
16.1 Introduction (informative)	77
16.2 The \$bits system function	77
Section 17 Compiler Directives.....	78
17.1 Introduction (informative)	78
17.2 `define macros.....	78
Section 18 Assertions	79
Section 19 Recommended items for deprecation	81
Annex A Formal Syntax.....	83
Annex B Keywords.....	93
Annex C Glossary	95
Annex D Bibliography.....	97

Section 1

Introduction to SystemVerilog

This document specifies the Accellera extensions for a higher level of abstraction for modeling and verification with the Verilog Hardware Description Language. Much of the syntax and semantics in these extensions are part of the SUPERLOG Extended Synthesis Subset (ESS) donation made to Accellera by Co-Design Automation, Inc. and proven with their products. SUPERLOG was developed by Peter Flake and Simon Davidmann to extend Verilog into the systems space and the verification space and was built on top of the work of the IEEE Verilog 2001 committee.

Throughout this document:

- “Verilog” or “Verilog-2001” refers to the IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language
- “SystemVerilog” refers to the Accellera extensions to the Verilog-2001 standard.

This document numbers the generations of Verilog as follows:

- **“Verilog 0.0”** is the original Verilog language, first developed by Gateway Design Automation in 1984
- **“Verilog 1.0”** is the Open Verilog International (OVI) public version of Verilog released in 1990, which was standardized by the IEEE in 1995 as IEEE Std. 1364-1995
- **“Verilog 2.0”** is the IEEE Std. 1364-2001 Verilog standard, commonly called Verilog-2001; this generation of Verilog contains the first significant enhancements to Verilog since its release to the public in 1990
- **“SystemVerilog 3.0”** is Verilog-2001 plus an extensive set of high-level abstraction extensions, as defined in this document

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

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

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

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

Section 2

~~Lexical Conventions~~ Literal Values

per Stu's e-mail
25 Mar 2002

2.1 Introduction (informative)

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

2.2 Literal value syntax

[BNF excerpt to be inserted after BNF is approved]

Syntax 2-1—Literal values

2.3 Integer and logic literals

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

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

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

2.4 Real literals

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

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

2.5 Time literals

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

```
0.1ns
40ps
```

2.6 String literals

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

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

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

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

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

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

Packed and unpacked arrays are discussed in section 4.

2.7 Array literals

Arrays literals are similar to C initializers, but with the repeat operator ({ { } }) allowed

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

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

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

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

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

2.8 Structure literals

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

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

Nested braces should reflect the structure, for example:

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

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

Section 3 Data Types

3.1 Introduction (informative)

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

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

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

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

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

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

See also Section 4 on arrays.

3.2 Data type syntax

[BNF excerpt to be inserted after BNF is approved]

Syntax 3-2—data types

3.3 Integer data types

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

Table 3-1—Integer data types

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

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

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

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

3.3.2 Signed and unsigned data types

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

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

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

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

See also section 7, on operators and expressions.

3.4 Other basic data types

3.4.1 Time data types

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

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

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

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

For example

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

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

3.4.2 Real and shortreal data types

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

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

3.4.3 Void data type

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

3.5 User-defined types

[BNF excerpt to be inserted after BNF is approved]

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

```
typedef int intP;
```

This can then be instantiated:

```
intP a, b;
```

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

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

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

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

```
interface it;  
    typedef int intP;  
endinterface  
  
it it1;  
typedef it1.intP intP;
```

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

3.6 Enumerations

An enumerated type has one of a set of named values.

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

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

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

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

```
enum {bronze=4'h3, silver, gold} medal4; // 4 bits wide
```

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

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

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

3.7 Structures and Unions

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

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

Named structure types must always use **typedef**, as there is no equivalent of the C **struct** adjective, such as 'struct instruction IR;'. Some additional examples of declaring structure and unions are:

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

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

typedef struct {
    bit isfloat;
    union { int i; shortreal f; } n; // anonymous union type
} tagged; // named structure
    tagged a[9:0]; // array of them
```

per Peter's
e-mail
11 Mar 2002

A structure can be assigned as a whole, and passed to or from a function or task as a whole. Note that it is inefficient to copy large structures. ~~A structure can contain arrays, but a union cannot contain an array of variable size.~~

per Peter's
e-mail
11 Mar 2002

Section 2.8 discusses assigning initial values to a structure.

3.8 Casting

A data type may be changed by using a cast (') operation.

```
int'(2.0 * 3.0)
```

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

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

The signedness can also be changed.

signed' (x)

A user-defined type can be used.

mytype'(foo)

A complex data type cannot be used. It must be defined with a **typedef**.

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

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

typedef bit [$bits(tagged) - 1 : 0] tagbits; // tagged defined above
tagbits t = tagbits'(a[3]); // convert structure to array of bits
a[4] = tagged't; // convert array of bits back to structure
```

per Stu's e-mail
25 Mar 2002

Editor's note: Are parenthesis required around the casted value? The preceding line does not have them, but all other examples do..

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

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

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

Section 4 Arrays

4.1 Introduction (informative)

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

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

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

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

SystemVerilog enhances array declarations in several ways.

4.2 Packed and unpacked arrays

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

Dense arrays of small data types such as bits can be stored packed (8 bits to a byte) or unpacked (1 bit to a word). This choice affects the efficiency of operations such as addition of bit vectors or selection of individual bits, and is similar to the Verilog-2001 notions of **vectorof** and **scalared**. Assignments and arithmetic operations are allowed for packed arrays but not for unpacked. The **vectorof** and **scalared** modifiers shall behave as defined in the IEEE Verilog standard. They may be used by software implementations to optimize performance.

Packed arrays can only be made of the single bit types: **bit**, **logic**, **reg**, **wire**, and the other net types. The dimensions are written to the left of the variable for a packed array, and to the right for an unpacked array.

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

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

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

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

4.3 Multiple dimensions

Like Verilog memories, the dimensions following the type set the packed size. The dimensions following the instance set the unpacked size, where access must be by index.

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

can be used as follows:

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

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

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

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

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

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

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

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

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

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

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

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

per Stu's e-mail
25 Mar 2002

If an index expression is of ~~type logic vector a 4-state type~~, and the array is of ~~type logic vector a 4-state type~~, an **X** or **Z** in the index expression will cause a read to return **X**, and a write to issue a run-time warning. If an index expression is of ~~type logic vector a 4-state type~~, but the array is ~~not of type logic a 2-state type~~, an **X** or **Z** in the index expression ~~will shall~~ generate a run-time warning and be treated as **0**. If an index expression is out of bounds, a run-time warning may be generated. ~~This check can be turned off for efficiency.~~

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

4.4 Part selects (slices)

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

```
int j = 0;
shortint msh = j[31:16];
```

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

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

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

Section 5

Data Declarations

5.1 Introduction (informative)

There are several forms of data in SystemVerilog: macros (see section 17), literals (see section 2), parameters (see section 14), constants, variables, nets, [and](#) attributes (see section 6)

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

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

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

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

5.2 Data declaration syntax

[BNF excerpt to be inserted after BNF is approved]

Syntax 5-3—Data declaration syntax

5.3 Constants

Constants are named data items which never change. SystemVerilog allows any data type to be declared as constant, with the **const** keyword.

```
const char colon = ":" ;
```

A constant expression contains literals and other named constants.

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

per Peter's
e-mail
11 Mar 2002

Editor's note: Suggest integrating contents of section 14 here, and deleting section 14.

5.4 Variables

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

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

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

```
int i = 0;
```

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

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

5.5 Scope and lifetime

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

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

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

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

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

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

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

```

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

```

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

See also section 11 on tasks and functions.

5.6 Net declarations

per mtg. 19
18 Mar 2002

[BNF excerpt to be inserted after BNF is approved]

Syntax 5-4—Net declaration syntax

5.6 Nets, regs, and logic

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

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

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

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

```

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

```

Section 6 Attributes

6.1 Introduction (informative)

With Verilog-2001, users can add named attributes (properties) to Verilog objects, such as modules, instances, wires, etc. The SystemVerilog extends the attribute syntax to support interfaces. The SystemVerilog also defines a default data type for attributes.

6.2 Attribute syntax for interfaces

[BNF excerpt to be inserted after BNF is approved]

Syntax 6-5—Interface attribute syntax

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

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

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

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

per Peter's e-mail
after mtg. 19
19 Mar 2002

Section 7

Operators and Expressions

7.1 Introduction (informative)

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

per Stu's e-mail
25 Mar 2002

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

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

Editor's note: Suggest the preceding paragraph be moved to normative section 7.7.

~~The fixed type and size of operators is preserved in SystemVerilog.~~

~~In Verilog-2001, the size is the maximum of the operands and the context, with unsigned winning whether it is bigger or not. This is likely to give unexpected behavior — adding a carry bit can make a signed result unsigned. An example of this in Verilog is:~~

per Stu's e-mail
25 Mar 2002

```
parameter p1 = -10 + 'h1; // this is unsigned
wire {63:0} w1 = p1;
wire {31:0} w2 = p1;
```


7.2 Operator syntax

[BNF excerpt to be inserted after BNF is approved]

Syntax 7-6—Operator syntax

7.3 Assignment, incrementor and decrementor operations

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

per Stu's e-mail
25 Mar 2002

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

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

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

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

7.4 Operations on logic and bit types

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

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

per Stu's e-mail
25 Mar 2002

~~The expression **bit(1bX ? 1b0 : 1b1)** returns 0 not 1 because the X on the selector produces an X on the ternary operator output, which is converted to 0 by the cast to bit.~~

~~The operators **||** and **&&** provide 'short-circuit' evaluation as follows:~~

```
a() || b();           // b is not evaluated if a returns 1.
a() && b();           // b is not evaluated if a returns 0;
```

~~Note that b is evaluated if a returns X.~~

~~To avoid short-circuit evaluation use a bitwise operator~~

```
a() != 0 || b() != 0; // b is always evaluated
```

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

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

7.5 Real operators

~~Real and shortreal operands are allowed with the following unary operators (the increment is by 1.0):~~

```
++
--
```

per Stu's e-mail
25 Mar 2002

~~Real and shortreal operands are allowed with the following binary and ternary operators:~~

```
*/
>> <<
&& ||
== !=
?:
```

Operands of type **shortreal** have the same operation restrictions as Verilog **real** operands. The unary operators **++** and **--** may have operands of type **real** and **shortreal** (the increment is by 1.0).

Editor's note: Preceding change made to emphasize what is different in SystemVerilog.
QUESTION: What assignment operators are legal/illegal with real operands?

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

Real operands can also be used in the following expressions:

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

7.6 Size

The number of bits of an expression is determined by the operands and the context, following the same rules as in Verilog-2001. In SystemVerilog, casting can be used to set the size context of an intermediate value.

per Stu's e-mail
25 Mar 2002

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

7.7 Sign

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

7.8 Operator precedence and associativity

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

Table 7-2—Operator precedence and associativity

() [] -> .	left
Unary ! ~ ++ -- + - & ~& && ~ ^ ~^	right
**	left
* / %	left
+ -	left
<< >> <<< >>>	left
< <= > >=	left
== != === !==	left
&	left
^ ~^	left
	left
&&	left
	left
? :	right
= += -= *= /= %= &= ^= = <<= >>= <<<= >>>=	none
{ , }	concatenation

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

7.9 Concatenation

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

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

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

per Stu's e-mail
25 Mar 2002

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

Editor's note: The concatenation {1,1} is illegal in Verilog, which only permits sized literals in concatenation. If unsized literals are allowed in SystemVerilog, this new feature needs to be described.

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

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

Multiple concatenation can be used for initializers as well e.g. {3{1}} for {1, 1, 1}.

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

Section 8

Procedural Statements and Control Flow

8.1 Introduction (informative)

Procedural statements are introduced by one of:

```
initial // do this statement once  
always, always_comb, always_latch, always_ff // loop forever (see section 10 on processes)  
task // do these statements whenever the task is called  
function // do these statements whenever the function is called and return a value
```

SystemVerilog has the following types of control flow within a process

- Selection, loops and jumps
- Task and function calls
- Sequential and parallel blocks
- Timing control

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

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

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

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

per Stu's e-mail
28 Mar 2002

[BNF excerpt to be inserted after BNF is approved]

Syntax 8-7—Statement syntax

8.2 Blocking and nonblocking assignments

[BNF excerpt to be inserted after BNF is approved]

Syntax 8-8—Blocking and nonblocking assignment syntax

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

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

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

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

It shall be illegal to make nonblocking assignments to automatic variables ~~are not allowed~~.

per Stu's e-mail
28 Mar 2002

The size of the left-hand side of an assignment forms the context for the right hand side expression. If the left-hand side is smaller than the right hand side, and information may be lost, a warning ~~is can be~~ given. ~~Non-blocking assignments to automatic variables are not allowed.~~

per Stu's e-mail
28 Mar 2002

8.3 Selection statements

[BNF excerpt to be inserted after BNF is approved]

Syntax 8-9—Selection statement syntax

```
if (condition) statement // as Verilog or C
if (condition) statement else statement
```

per Stu's e-mail
28 Mar 2002

The condition is evaluated as a boolean, so that 0 or X or In Verilog, an **if** (expression) is evaluated as a boolean, so that if the result of the expression is 0 or X, the test is considered false. With SystemVerilog, **null** or **void** or **{}** are also false, and any other values are true.

SystemVerilog adds the keywords **unique** and **priority**, which can be used before an **if**. In the case of an **else...if**, the **unique** indicates that the conditions do not overlap. If either keyword is used it is shall be a run-time warning for no condition to match unless there is an explicit **else**. For example:

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

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

In the case of the **unique if**, there is no overlap in the conditions, allowing the expressions to be evaluated in parallel. With the **priority if**, if the variable 'a' in the preceding example had a value of 0, it would satisfy both the first and second conditions, requiring priority logic.

per Stu's e-mail
28 Mar 2002

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

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

In Verilog-2001, there are three types of case statements, introduced by **case**, **casez** and **casex**. With SystemVerilog, each of these can be qualified by **priority** or **unique**. A **priority case** shall act on the first match only. A **unique case** shall guarantee no overlapping case values, allowing the case items to be evaluated in parallel. If the case is qualified as **priority** or **unique**, the simulator shall issue a warning message if an unexpected case value is found. The user does not need By specifying **unique** or **priority**, it is not necessary to code a **default** case to trap unexpected case values.

per Stu's e-mail
28 Mar 2002

For example

```

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

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

```

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

8.4 Transition statements

The transition statement is used in state machine modeling, and discussed in section 9 on state machines.

8.5 Loops statements

[BNF excerpt to be inserted after BNF is approved]

Syntax 8-10—Loop statement syntax

Verilog-2001 provides **for**, **while**, **repeat** and **forever** loops. SystemVerilog adds a **do...while** loop:

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

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

8.6 Jump statements

[BNF excerpt to be inserted after BNF is approved]

Syntax 8-11—Jump statement syntax

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

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

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

8.7 Named blocks and statement labels

[BNF excerpt to be inserted after BNF is approved]

Syntax 8-12—Blocks and labels syntax

Verilog-2001 allows a **begin...end** or **fork...join** statement block to be named. A named block is used to identify the entire statement block. A named block creates a new hierarchy scope. The block name is specified after the **begin** or **fork** keyword, preceded by a colon. For example:

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

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

per Stu's e-mail
28 Mar 2002

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

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

```
labelA: statement
```

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

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

A label at the end of the block is not required. It **is shall be** an error if the label at the end is different **than the label at the beginning**.

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

8.8 Processes

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

A new (dynamic) process can be created by the **process** keyword. This forks off a statement without waiting for completion:

```
process statement
```

See Section 10 for more information about processes.

8.9 Disable

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

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

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

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

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

8.10 ~~Delay and~~ Event control

per Stu's e-mail
28 Mar 2002

[BNF excerpt to be inserted after BNF is approved]

Syntax 8-13—*Delay and event control syntax*

SystemVerilog has ~~#time or #(time_expression)~~ as a delay control, like Verilog-2001.

per Stu's e-mail
28 Mar 2002

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

SystemVerilog adds ~~the following enhancement:~~ an **iff** qualifier to the @ event control.

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

The event expression only triggers if the expression after the **iff** is true, in this case when `rst == 0`. Note that such an expression is evaluated when `clk` changes not when `rst` changes. Also note that **iff** has precedence over **or**. This can be made clearer by the use of parentheses.

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

per Stu's e-mail
28 Mar 2002

~~Any change in a variable or net can be detected using the @ event control, as in Verilog-2001. For more clarity,~~ SystemVerilog also allows the @ event control to explicitly state any change, using the **changed** keyword:

per Stu's e-mail
28 Mar 2002

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

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

~~If the expression evaluates to a result of more than one bit, a change on any of the bits of the result (including an x to z change) will trigger the event control.~~ The @(**changed** expression) differs from @(expression) in that the **changed** keyword explicitly defines that the event control only triggers on a change of the result of the expression. In certain types of expressions, @(expression) ~~may~~ can trigger on changes to operands of the expression that do not affect the result.

per Stu's e-mail
28 Mar 2002

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

Section 9 State Machines

Editor's note: This section not yet indexed.

9.1 Introduction (informative)

Control logic, whether for a data path or an interface, is often specified as a Finite State Machine (FSM). Often this specification is in the form of a 'bubble' diagram, where circles represent states and arcs represent transitions. An alternative convention is to have vertical lines representing states and horizontal lines representing transitions. State machine capture tools often add hierarchy and concurrency to the basic FSM model.

Transitions are usually annotated with an input condition, for a synchronous FSM, or event, for an asynchronous one. In addition, an FSM may be active, i.e. it sets outputs. These outputs may be combinational functions of inputs (Mealy model) or just depend on the state (Moore model). These names have also been applied to procedural operations associated with a transition (Mealy) or a state (Moore).

To avoid timing hazards, inputs should not change at the same time as state. For an asynchronous FSM this normally means that input events should arrive well separated. For a synchronous FSM it means that other inputs should not change at the same time as the clock. Where two machines communicate, this in turn means that the outputs should not change until the inputs of both machines have been read. Nonblocking assignments are often used for this. Alternatively, delays can be put in the communication paths.

Verilog-2001 has no specific support for state machines. There are three modeling styles which are often used: case statements, named events, and implicit.

SystemVerilog adds several constructs specific to modeling state machines. These include named states and transitions, with dedicated syntax to distinguish them, and a semantics such that each statement is executed in a state or transition which can be determined by static analysis. The state is accessible to combinational logic.

The benefits of the SystemVerilog FSM constructs are readability for the user, including hierarchy and concurrency, ease of analysis for tools, and a better trade-off between debug and performance. The style should also support the software implementation of a state machine without any timing and without the overhead of extra events.

9.2 State machine constructs

The SystemVerilog state machine definition adds a new declaration type and two new statement types. The states are declared like an enumeration with a timing control. Note that S is used to represent both currentState and nextState:

```
state {S0, S1, S2} S @(posedge clock);
```

Editor's note: Per meeting 14, the keyword "state" needs to be changed to something else. It is a commonly used name in existing models.

SystemVerilog provides a transition statement, which can be used in a procedural block, such as an **always_comb** block. As well as the state declaration, SystemVerilog uses the **transition** statement in procedural code. The **transition** statement contains the individual state transitions shown by ->> :

```
transition (S)
  S0: if (serial == 1) ->>S2;
  S2: if (serial == 0) ->> S1; else ->> S0;
  S1: ->> S0;
endtransition
```

A complete example, referred to as the Wang and Edsall² example, is shown below. This Mealy style FSM has three states, a single input, and a single output. Its state structure is:

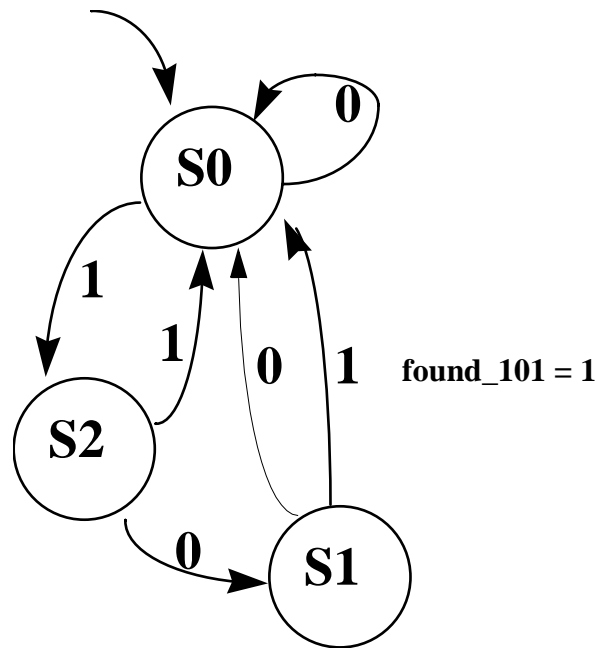


Figure 9-1—Simple state machine example

The three examples that follow illustrate some ways the state declaration and transitions can be used.

The following code illustrates how this state machine can be modeled using SystemVerilog. In this example the reset is synchronous, because the time control on the state declaration specifies transitions occur on the clock edge.

```
// SystemVerilog version of Wang & Edsall example, synchronous reset
module FSM1(output found_101, input serial, input clk, input reset);
  state {S0, S1, S2} S @(posedge clk);
  always_comb begin
    found_101 = 0;
    if (reset) ->> S.S0;
    else transition (S)
      S0: if (serial == 1) ->> S2;
      S2: if (serial == 0) ->> S1; else ->> S0;
      S1: ->> S0 if (serial == 1) found_101 = 1;
    endtransition
  end
endmodule
```

Note that the timing control in the example above can include a test for a condition, e.g. `@(posedge clock iff !reset)`, showing that states change on the rising edge of the clock only if reset is 0.

Editor's note: poor usage of iff, as discussed in meeting 14.

² T-H. Wang, T. Edsall "Practical FSM Analysis for Verilog", Proceedings, IVC 98, pp. 52-58.

The following example, the transition is instantaneous, because there is no timing control with the state declaration. Instead, the timing control is specified on the always procedure, with an asynchronous reset.

```
// SystemVerilog version of Wang & Edsall example, asynchronous reset
module FSM2(output logic found_101, input serial, input clock, input reset);
  state {S0, S1, S2} S; // transition is instantaneous
  always @(posedge clock or posedge reset) begin // asynchronous reset
    if (reset) ->> S.S0;
    else transition (S)
      S0: if (serial == 1) ->> S2;
      S2: if (serial == 0) ->> S1; else ->> S0;
      S1: ->> S0;
    endtransition
  end
  always_comb
    if (S.S1 && serial == 1) found_101 = 1;
    else found_101 = 0;
endmodule
```

In the next example, the transition is nonblocking, with a delay. The reset is asynchronous.

```
// SystemVerilog version of Wang & Edsall example, delayed transition
module FSM3(output logic found_101, input serial, input clock, input reset);
  state {S0, S1, S2} S #1ns; // transition is nonblocking with delay
  always @(posedge clock or posedge reset) begin // asynchronous reset
    if (reset) ->> S.S0;
    else transition (S)
      S0: if (serial == 1) ->> S2;
      S2: if (serial == 0) ->> S1; else ->> S0;
      S1: ->> S0;
    endtransition
  end
  always_comb
    if (S.S1 && serial == 1) found_101 = 1;
    else found_101 = 0;
endmodule
```

9.3 State declarations

A SystemVerilog state declaration is syntactically based on an enumerated type:

[BNF excerpt to be inserted after BNF is approved]

Syntax 9-14—State declaration syntax

Each state name shall be unique to the state declaration, so that if there are several state machines, the state names can overlap. Each state name can be preceded by a list of sub-states to represent a hierarchical state machine. The optional control expression is used to modify the nonblocking assignment to the state variable as described below.

The initial state is the first state listed. This is because the syntax <state variable> '=' <initial state> would be anomalous, since state variables cannot be assigned.

The state variable can be read as a string. for display purposes:

```
$display (" state = %s", S);
```

The semantics differs both from an enumerated type and from a structure. Firstly the state name is not a constant but can be read as a variable that has a bit value which is 1 if the machine is in that state and a 0 if not, so it can be tested in conditions e.g. `if (S.S2 || S.S0)`.

Secondly the state variable cannot be assigned directly, but change to a new state is marked by a `->>` operator. This updates the state machine to the new state as if an assignment were used. The optional timing control makes this assignment nonblocking with the specified delay. If more than one assignment is applied to happen at the same simulation time, the last one wins (there may be a race). This operator makes it easy to distinguish state changes from other assignments.

Editor's note: Per meetings 12-15, there are several open issues:

- Need a method of specifying the state values. Syntax proposed is: `state {S1=3'b001, S2=3'b010, S3=3'b100} S;`
- Need to provide ability to do bit selects of state value, in order to model 1-hot encoding and such
- Need to describe and provide examples of synchronous and asynchronous resets

9.4 Transition statements

[BNF excerpt to be inserted after BNF is approved]

[need to add default statement]

Syntax 9-15—State transition statement syntax

Each change of state shall be written as a transition_to_state. This can be unconditional, in which case the machine must be specified, or it can be in a transition statement, which also allows the transition to be named. Thus, each arc of the state diagram (represented by `->>`) can be labeled (e.g. `S0_S2`) as well as each node, as shown in the modified version of the Wang & Edsall example:

```
// SystemVerilog version of Wang & Edsall example with named transitions
```



```

module FSM5(output logic found_101, input serial, input clock, input reset);
    state {S0, S1, S2} S @ (posedge clock);

    always_comb begin
        found_101 = 0;
        if (reset) ->> S.S0;
        else transition (S)
            S0:if (serial == 1) S0_S2 ->>S2;
            S2:if (serial == 0) S2_S1 ->> S1; else S2_S0 ->> S0;
            S1: S1_S0 ->> S0 if (serial == 1) found_101 = 1;
        endtransition
    end
endmodule

```

The optional transition name is conceptually rather like an event name, but it differs in that it is not declared and must be unique in the state machine. A transition name may be used in an event expression elsewhere in the module e.g. @S0_S2. The timing of such a transition is the same as the event that triggered it e.g. @(posedge clk).

The state conditions are compared with the state variable, like the case conditions in a case statement. During execution, if the transition statement does not have a state matching the current state a run time warning occurs. If more than one state condition is specified, or the **default** keyword is used, the following transition cannot be given a name because there is more than one transition implied. The unconditional transition ->>A.B is an abbreviation for **transition(A) default: ->>B; endtransition**.

Transition names and entry and exit of states can be used like other event expressions to trigger other **always** blocks or for timing checks. The transition is written with the state machine name e.g.

```
@S.S0_S2.
```

The change to a state can be written:

```
@(posedge S.S1)
```

and the change from a state can be written:

```
@(negedge S.S1)
```

Similarly any state change can be written:

```
@(S)
```

These state change event expressions trigger after the state has changed, i.e. later than the execution of the transition statement.

A simulator can check that the first transition to be executed is unconditional or contains a **default**, to model reset from an unreachable or unknown state.

The following example illustrates using **fork** and **join** with transitions on implicit named events, so that all transitions must have fired, in any order, to continue past the join.

```

initial begin
    fork
        @S.S0_S2;
        @S.S2_S1;
        @S.S2_S0;
    join
end

```

The state of state variables can be randomly initialized using an initial block with transitions determined by a random number.

9.5 Hierarchical and concurrent state machines

For large state machines it is convenient to structure the state space into states and sub-states. The states can be *or-states* which are comma separated, or *and-states*, e.g.

```
state {STA, {s1, s2, s3} STB, {{s4, s5, s6} c1 and {s7, s8} c2} STC} hsm1 @
clock;
```

This example can be illustrated as follows:

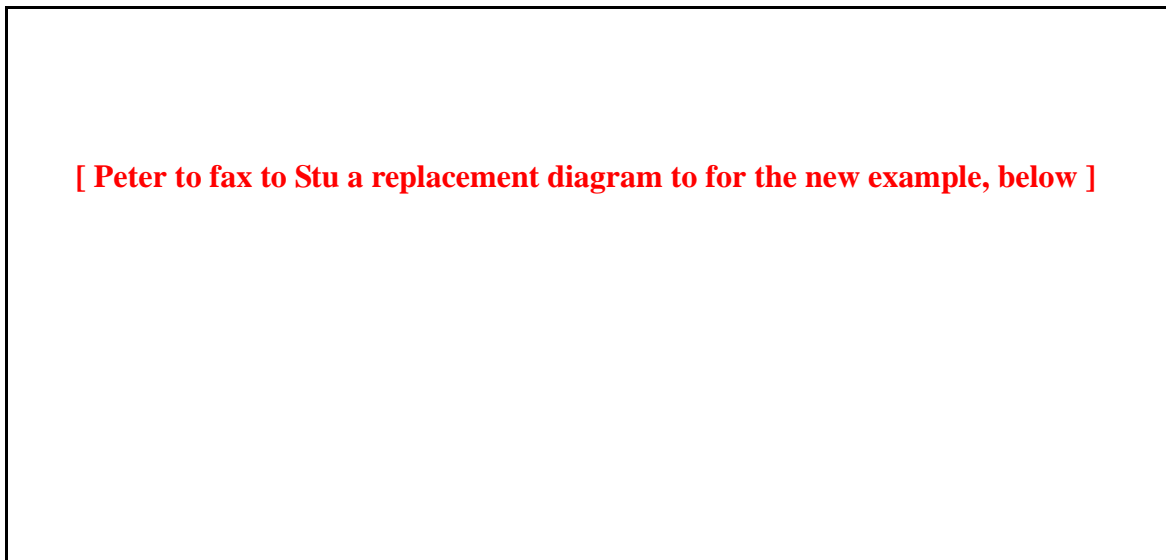


Figure 9-2—Hierarchical state machine

Only one of a set of **or** states can be active at any time. But all **and** states are active or inactive simultaneously. Therefore, s4 and s7 can be active together.

Transitions are allowed from any state to any state, from any sub-state to any other state, and from one sub-state to another in the same state:

```
STA:->> STB;
s1:if(input1) ->> STC; else ->> s2;
```

Other transitions to sub-states are not allowed. When a state contains **or** sub-states, entering the state implies entering the first sub-state. So ->> **STB** enters **s1** as well. The truth value of the state name is the logical or of the truth values of the sub-states.

When a state contains **and** sub-states, entering the state implies entering all the sub-states, as shown by the two arrows in STC.

Two sub-states within **and** states can share a transition:

```
s5 and s7:T2 ->> s6 and s8;
```

The following example illustrates a more complex state machine of a two-player Reflex game³, with reset

logic and transition statements within a **begin...end** block.

```
// SystemVerilog description of two-player Reflex game. Note that this is
// an abstract model with events for input and enumeration for output

module REFLEX ( input  event ready, stop, go, coin,
                output enum {blue, yellow, red, green, flashing} light) ;

    event timer;

    always #10 ->timer;

    state {
        game_off,
        {
            {wait_ready, wait_go, wait_stop, done} player1 and
            {idle, waiting} player2
        } game_on
    } game;

    always begin: normal

        // Either player may assert coin to start the game, status light turns blue
        @(coin) transition (game) game_off: ->> game_on light = blue; endtransition

        // When player 1 presses ready, the status light turns yellow
        @(ready) transition (game)
            wait_ready and idle: start->> wait_go and waiting light = yellow;
        endtransition

        // When player 2 presses go the status light turns green
        fork : fork1
            @(go) disable fork1;
            repeat ('L) @timer disable fork1; // if player 2 does not press go in time
            @(stop) begin // stop before go
                transition (game)
                    wait_go: ->> game_off light = flashing;
                endtransition
                disable normal;
            end
        join
        transition (game)
            wait_go and waiting: end2->> wait_stop and idle light = green;
        endtransition

        // Player 1 presses stop within L time units
        fork : fork2
            begin @(stop) transition (game)
                wait_stop: ->> game_off light = red;
            endtransition disable fork2; end
            begin repeat ('L) @(timer) transition (game)
                wait_stop: ->> game_off light = flashing;
            endtransition disable fork2; end
        join
```

³ “Hierarchical Finite State Machines with Multiple Concurrency Models”, Alain Girault, Bilung Lee, Edward A Lee, IEEE Transactions on Computer Aided Design, Vol 18, No 6, June 1999, pp. 742.

```
end: normal

always @light $display("%t %s", $realtime, light);

endmodule
```

Section 10 Processes

10.1 Introduction (informative)

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

~~Verilog-2001 has a continuous assignment to describe combinational logic. However, it is missing a case statement for multiplexers and there is no name for the driving value if the wire has multiple drivers.~~

per Peter's
e-mail
11 Mar 2002

In an **always** block which is used to model combinational logic, forgetting an **else** leads to an unintended latch. To avoid this mistake, SystemVerilog adds specialized **always_comb** and **always_latch** blocks ~~are provided~~, which indicate the design intent to simulation, synthesis and formal verification tools. SystemVerilog also adds an **always_ff** block to indicate sequential logic. ~~These blocks treat functions calls as part of the block, and therefore the sensitivity list and restrictions are extended into the functions called. Note that this does not apply to tasks.~~

per Stu's
e-mail
28 Mar 2002

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

per Stu's
e-mail
28 Mar 2002

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

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

per Peter's e-
mail re. mtg. 19
19 Mar 2002

Editor's note: Some of the above paragraph seems like it should be in the normative text, as well. Where?

~~When dynamic processes are created, there must be an identification system for subsequently deleting them. This is like the process id in Unix. The process identifier is also used to refer to variables within that process.~~

per Stu's
e-mail
28 Mar 2002

10.2 Static process labels

Editor's note: Draft 4 section 10.2 is now 10.6, per Stu's e-mail 28 Mar 2002

10.2 Level sensitive logic

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

```
always_comb
  a = b & c;

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

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

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

- The variables written on the left-hand side of assignments may not be written to by any other process.
- The procedure is automatically triggered once at time zero, after all **initial** and **always** blocks have been started, so that the outputs of the procedure are consistent with the inputs.

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

- **always_comb** automatically executes once at time zero, whereas **always @*** waits until a change occurs on a signal in the inferred sensitivity list.
- **always_comb** is sensitive to changes within the contents of a function, whereas **always @*** is only sensitive to changes to the arguments of a function.
- Variables on the left-hand side of assignments within an **always_comb** procedure may not be written to by any other processes, whereas **always @*** permits multiple processes to write to the same variable.

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

10.3 Latch sensitive logic

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

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

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

- There is an inferred sensitivity list that includes every variable read by the procedure.
- The variables written on the left-hand side of assignments may not be written to by any other process.
- The procedure is automatically triggered once at time zero, after all **initial** and **always** blocks have been started, so that the outputs of the procedure are consistent with the inputs.

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

10.4 Edge sensitive logic

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

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

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

10.5 Continuous assignments

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

SystemVerilog removes this restriction, and permits continuous assignments to drive nets, logic vari-

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

10.6 Static Process labels

per Stu's
e-mail
28 Mar 2002

In SystemVerilog, ~~users can name static~~ processes can be named by labeling the outermost statement. A label before the process statement serves as an identifier of the process. This process identifier can be used to refer to variables within that process.

```
module processes1;
  int b;
  always
    # foo: begin //the begin...end statement group is named foo
      #10ns b = 0;
      #10ns b = 1;
    end
endmodule
```

per Stu's
e-mail
28 Mar 2002

Within a parallel block such as **fork ... join**, each statement is a different process, and different from the outer block. In the following example, a, b, c are different processes:

```
always
  a: fork //the fork...join statement group is named a
    b: task1();
    c: task2();
  join
```

per Stu's
e-mail
28 Mar 2002

10.7 Dynamic processes

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

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

process statement

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

per Peter's
e-mail
11 Mar 2002

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

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

```
// pipeline module
```

```

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

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

```

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

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

per Peter's
e-mail
27 Mar 2002

Section 11

Tasks and Functions

Editor's note: Except as noted in a margin, changes in this draft are non-technical clean-up by Stu, for clarity.

11.1 Introduction (informative)

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

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

SystemVerilog also adds:

- More capabilities for declaring task and function ports
- Function output and inout ports
- Void functions
- Multiple statements in a task or function without requiring a **begin...end** or **fork...join** block
- Returning from a task or function before reaching the end of the task or function

11.2 Tasks

[BNF excerpt to be inserted after BNF is approved]

Syntax 11-16—Task syntax

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

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

per mtg. 20
25 Mar 2002

```

    ...
endtask

task mytask2;
    output x;
    input y;
    int x;
    string y;
    ...
endtask

```

Each formal argument has one of the following directions:

```

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

```

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

```

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

```

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

```

task mytask4(input [3:0][7:0] a, b, output [3:0][7:0] y);
    ...
endtask

```

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

With SystemVerilog, multiple statements can be written between the task declaration and **endtask**, which means that the **begin end** can be omitted. If **begin end** is omitted, statements are executed sequentially, the same as if they were enclosed in a **begin end** group.

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

11.3 Functions

[BNF excerpt to be inserted after BNF is approved]

Syntax 11-17—Function syntax

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

```
function logic [15:0] myfunc1(int x, int y);  
    ...  
endfunction  
  
function logic [15:0] myfunc2;  
    input int x;  
    input int y;  
    ...  
endfunction
```

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

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

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

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

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

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

```
...
endfunction
```

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

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

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

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

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

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

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

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

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

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

Section 12 Hierarchy

Editor's note: Except as noted in a margin, changes in this draft are non-technical clean-up by Stu, for clarity.

12.1 Introduction (informative)

Verilog-2001 has a simple organization. All data, functions and tasks are in modules except for system tasks and functions, which are global, and may be defined in the PLI. A Verilog module can contain instances of other modules. Any uninstantiated module is at the top level. This does not apply to libraries, which therefore have a different status and a different procedure for analyzing them. A hierarchical name can be used to specify any named object from anywhere in the instance hierarchy. The module hierarchy is often arbitrary and a lot of effort is spent in maintaining port lists.

In Verilog-2001, only `net`, `reg`, `integer` and `time` data types can be passed through module ports.

SystemVerilog adds many enhancements for representing design hierarchy:

- A global declaration space, visible to all modules at all levels of hierarchy
- Nested module declarations, to aid in representing self-contained models and libraries
- Relaxed rules on port declarations
- Simplified named port connections
- Implicit port connections, using `.*`
- Time unit and time precision specifications bound to modules
- A concept of interfaces to bundle connections between modules (presented in section 13)

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

12.2 The \$root top level

In SystemVerilog there is a top level called `$root`, which is the whole source text. This allows declarations outside any named modules or interfaces, unlike Verilog-2001.

SystemVerilog requires an elaboration phase. All modules and interfaces must be parsed before elaboration and the order of elaboration must be defined.

Editor's note: Where is "the order of elaboration" defined?

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

~~If there is no explicit top level instantiation, then~~ A module can be explicitly instantiated in the \$root top-level. All uninstantiated modules become implicitly instantiated within the top level, ~~This~~ which is compatible with Verilog-2001.

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

The \$root top level:

- has a single occurrence

- can be distributed across any number of files
- variable and net definitions are in a global name space and can be accessed throughout the hierarchy
- task and function definitions are in a global name space and can be accessed throughout the hierarchy
- may not contain **initial** or **always** procedures
- may contain procedural statements, which will be executed one time, as if in an **initial** procedure

Modules:

- can have any number of module definitions
- can have any number of module instances, which create new levels of hierarchy
- can be distributed across any number of files, and can be defined in any order
- variable and net definitions are in the module instance name space and are local to that scope
- task and function definitions are in the module instance name space and are local to that scope
- may contain any number of **initial** and **always** procedures
- may not contain procedural statements that are not within an **initial** procedure, **always** procedure, task, or function

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

per Peter's
e-mail
20 Mar 2002

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

per Peter's
e-mail
19 Mar 2002

```
typedef int myint;

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

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

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

main();
```

12.3 Module declarations

[BNF excerpt to be inserted after BNF is approved]

Syntax 12-18—Module declaration syntax

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

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

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

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

module m3(...);

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

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

12.4 Nested modules

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

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

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

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

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

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

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

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

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

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

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

```
module part1(...);
    module and2(input a; input b; output z);
    ....
endmodule
module or2(input a; input b; output z);
    ....
```



```
    endmodule
    ....
    and2 u1(....), u2(....), u3(....);
    ....
endmodule
```

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

Editor's note: Open issue from meeting 11: Need to discuss if an extern module declaration should be added. Per meeting 13, Kevin is to write a proposal.

12.5 Port declarations

[BNF excerpt to be inserted after BNF is approved]

Syntax 12-19—Port declaration syntax

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

Editor's note: Please verify what I added to the types of port declarations.

```
module mhl (input wire w0, output wire w1);
    assign w1 = w0;
endmodule
```

If no direction is specified, the port type defaults to an interface. If a direction is specified, the port type defaults to a **wire**.

~~A port may also have its own name specified, and an expression of nets or variables. Such a port name does not share the name space with the other module items. Names in the expression must be declared later, or be nets or variables within interfaces instantiated in the module.~~

per mtg. 20
25 Mar 2002

```
module mh (input in(w[0]), output out(w[1]));
    wire [1:0] w;
    always_comb w[1] = w[0];
endmodule
```

If no type or direction is given to a port, it inherits them from the last specified type and direction.

```
module mh3 (input char a, b);
    ...
endmodule
```

If there is no previous type or direction specified in the port list, then the ports are treated as port expressions and the types must be declared later (as in Verilog).

Editor's note: The above seems to conflict with two paragraphs before, which says if no direction is specified, the default is an interface.

```
module mh4(x, y);
    int x;
    char y;
endmodule
```

~~Note that~~ A software tool can use the port direction ~~provides an optional~~ to check against writing to an input port or not writing to an output port.

~~10.6 Port types and directions~~

Editor's note: The previous section discussed default port types and directions, so I deleted the subsection title

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

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

12.6 Time unit and precision

The time unit can be set by the **timeunit** keyword to a time which must be a power of 10 units e.g.

```
timeunit 100ps;
```

The time unit is determined by:

- 1) If a **timeunit** has been specified in the current module, then the time unit is set to module's time units.
- 2) Else, if the module definition is nested, then the time unit is inherited from the enclosing module.

- 3) Else, if a **'timescale** directive has been specified, then the time unit is set to the units of last **'timescale** directive.
- 4) Else, if the **\$root** top level has a time unit, then the time unit set to the time units of the root module.
- 5) Else, the simulator's default time units are used.

The simulator's default time units follow the rules of ~~the IEEE~~ Verilog ~~standard~~.

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

```
timeprecision 100fs;
```

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

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

12.7 Module instances

[BNF excerpt to be inserted after BNF is approved]

Syntax 12-20—Module instance syntax

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

Editor's note: Open issue from meeting 20: Cliff to send wording for **.*** and **.name** port connections.

12.8 Port connection rules

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

as follows:

- An **input** can be connected to any expression of a compatible data type. If unconnected, it has the initial value corresponding to the data type
- An **output** can be connected to a variable (or a concatenation) of a compatible data type, and has shared variable behavior if multiple outputs are connected (last write wins); An **output logic** can be connected to a net (to provide a resolution function in the case of multiple drivers)
- An **inout** can be connected to a variable (or a concatenation) of the same data type

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

- An **input** can be connected to any expression of a compatible data type. If unconnected, it has the value **'z'**
- An **output** can be connected to a **wire net type** (or a concatenation of net types) or left unconnected, but not to a **logic** variable
- An **inout** can be connected to a **wire net type** (or a concatenation of net types) or left unconnected, but not to a **logic** variable

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

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

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

- If the port is a net vector, then the Verilog connection rules for nets are followed.
- If the port is an **inout** port variable, then a port connection must have the same size and representation on both sides of the port. It shall be an error if there is a mismatch.
- If the port is an **input** or an **output** variable, then the Verilog assignment rules are followed.

12.9 Name spaces

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

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

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

- a module or interface
- a task or function
- a sequential or parallel block
- a structure or union

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

12.10 Hierarchical names

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

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

Nested identifiers can be read (in expressions), written (in assignments or task/function calls) or triggered off (in event expressions). They can also be used as type, task or function names. ~~See section 13 on interfaces.~~

Section 13

Interfaces

Editor's note: This section not yet indexed.

13.1 Introduction (informative)

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

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

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

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

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

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

13.2 Interface syntax

[BNF excerpt to be inserted after BNF is approved]

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

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

per Peter's
e-mail
11 Mar 2002

An interface is declared as follows:

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

per Peter's
e-mail
11 Mar 2002

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

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

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

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

13.2.1 Example without using interfaces

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

```
module memMod( input    bit req,  
               bit clk,  
               bit start,
```

```

        logic[1:0] mode,
        logic[7:0] addr,
    inout    logic[7:0] data,
    output    bit gnt,
            bit rdy );

    logic avail;

    ...
endmodule

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

    ...
endmodule

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

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

endmodule

```

13.2.2 Interface example using a named bundle

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

```

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

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

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

```



```

module top;
    logic clk = 0;

    simple_bus sb_intf; // Instantiate the interface

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

endmodule

```

Editor's note: The following paragraph and example is per Cliff's e-mail of 12/16/01, and is pending approval of the .* implicit port connection syntax.

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

```

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

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

module top;
    logic clk = 0;

    simple_bus sb_intf;

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

endmodule

```

13.2.3 Interface example using a generic bundle

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

```

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

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

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

```

```

endinterface: simple_bus

module top;
    logic clk = 0;

    simple_bus sb_intf; // Instantiate the interface

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

endmodule

```

Editor's note: The following paragraph and example is per Cliff's e-mail of 12/16/01, and is pending approval of the .* implicit port connection syntax.

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

```

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

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

module top;
    logic clk = 0;

    simple_bus sb_intf;

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

endmodule

```

13.3 Ports in interfaces

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

```

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

```

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

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

```

interface simple_bus (input bit clk); // Define the interface

```

```

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

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

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

module cpuMod(simple_bus b);
    ...
endmodule

module top;
    logic clk = 0;

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

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

endmodule

```

Editor's note: The following paragraph is per Cliff's e-mail of 12/16/01, and is pending approval of the .* implicit port connection syntax.

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

13.4 Modports

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

```

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

```

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

```

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

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

```

```

module top;
    i2 i;

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

```

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

```

module m (i2 i);
    ...
endmodule

module s (i2 i);
    ...
endmodule

module top;
    i2 i;

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

```

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

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

```

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

```

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

13.4.1 An example of a named port bundle

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

```

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

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

```

```

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

endinterface: simple_bus

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

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

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

module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

    initial repeat(10) #10 clk++;

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

endmodule

```

13.4.2 An example of connecting a port bundle

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

```

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

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

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

endinterface: simple_bus

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

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

module cpuMod(simple_bus b);

```

```

    ...
endmodule

module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

    initial repeat(10) #10 clk++;

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

endmodule

```

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

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

```

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

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

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

endinterface: simple_bus

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

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

module cpuMod(interface b);
    ...
endmodule

module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

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

endmodule

```

13.5 Tasks and Functions in Interfaces

Tasks and functions may be defined within an interface, or they may be defined within one or more of the modules connected. This allows a more abstract level of modeling. For example “read” and “write” can be defined as tasks, without reference to any wires, and the master module can merely call these tasks. In a **modport** these tasks are declared as **import** tasks.

If the tasks or functions are defined in a module, using a hierarchical name, they must also be declared as **extern** in the interface, or as **export** in a **modport**.

Tasks (not functions) may be defined in a module that is instantiated twice, e.g. two memories driven from the same CPU. Such multiple task definitions are allowed by a **forkjoin extern** declaration in the interface.

13.5.1 An example of using tasks in an interface

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

    task masterRead(input logic[7:0] raddr); // masterRead method
        // ...
    endtask: masterRead

    task slaveRead; // slaveRead method
        // ...
    endtask: slaveRead

endinterface: simple_bus

module memMod(interface a); // Uses any interface
    logic avail;

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

    always @(a.start)
        a.slaveRead;
endmodule

module cpuMod(interface b);
    enum {read, write} instr;
    logic [7:0] raddr;

    always @(posedge b.clk)
        if (instr == read)
            b.masterRead(raddr); // call the Interface method
        ...
endmodule

module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

    memMod mem(sb_intf.slave); // only has access to the slaveRead task
    cpuMod cpu(sb_intf.master); // only has access to the masterRead task
endmodule
```

```
endmodule
```

13.5.2 An example of using tasks in modports

This interface example shows how to use modports to control signal directions and task access in a full read/write interface.

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

    modport slave (input req, addr, mode, start, clk,
                  output gnt, rdy,
                  inout data,
                  import task slaveRead(),
                  task slaveWrite());
    // import into module that uses the modport

    modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout data,
                  import task masterRead(input logic[7:0] raddr),
                  task masterWrite(input logic[7:0] waddr));
    // import requires the full task prototype

    task masterRead(input logic[7:0] raddr); // masterRead method
    // ...
    endtask

    task slaveRead; // slaveRead method
    // ...
    endtask

    task masterWrite(input logic[7:0] waddr);
    //...
    endtask

    task slaveWrite;
    //...
    endtask

endinterface: simple_bus

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

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

    always @(a.start)
        if (a.mode[0] == 1'b0)
            a.slaveRead;
        else
            a.slaveWrite;
endmodule
```



```

module cpuMod(interface b);
    enum {read, write} instr = $rand();
    logic [7:0] raddr = $rand();

    always @(posedge b.clk)
        if (instr == read)
            b.masterRead(raddr); // call the Interface method
        // ...
        else
            b.masterWrite(raddr);

endmodule

module omniMod(interface b);
    //...
endmodule: omniMod

module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

    memMod mem(sb_intf.slave); // only has access to the slaveRead task
    cpuMod cpu(sb_intf.master); // only has access to the masterRead task
    omniMod omni(sb_intf); // has access to all master and slave tasks

endmodule

```

13.5.3 An example of exporting tasks and functions

This interface example shows how to define tasks in one module and call them in another using modports to control task access.

```

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

    modport slave( input req, addr, mode, start, clk,
                  output gnt, rdy,
                  inout data,
                  export task Read(),
                  task Write());
        // export from module that uses the modport

    modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout data,
                  import task Read(input logic[7:0] raddr),
                  task Write(input logic[7:0] waddr));
        // import requires the full task prototype

endinterface: simple_bus

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

```

```

    task a.Read; // Read method
        avail = 0;
        ...
        avail = 1;
    endtask

    task a.Write;
        avail = 0;
        ...
        avail = 1;
    endtask

endmodule

module cpuMod(interface b);
    enum {read, write} instr;
    logic [7:0] raddr;

    always @(posedge b.clk)
        if (instr == read)
            b.Read(raddr); // call the slave method via the interface
            ...
        else
            b.Write(raddr);

endmodule

module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

    memMod mem(sb_intf.slave); // exports the Read and Write tasks
    cpuMod cpu(sb_intf.master); // imports the Read and Write tasks

endmodule

```

13.5.4 An example of multiple task exports

It is normally an error for more than one module to export the same task name. However, several instances of the same modport type may be connected to an interface, such as memory modules in the previous example. So that these can still export their read and write tasks, the tasks must be declared in the interface using the **extern forkjoin** keywords. Normally only one module responds to the task call, e.g. the one containing the appropriate address. Only then should the task write to the result variables. Note multiple export of functions is not allowed because they must always write to the result.

This interface example shows how to define tasks in more than one module and call them in another using **extern forkjoin**. The multiple task export mechanism can also be used to count the instances of a particular modport that are connected to each interface instance.

```

interface simple_bus (input bit clk); // Define the interface
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;
    int slaves;
    // tasks executed concurrently as a fork/join block

```

```

extern forkjoin task countSlaves( );
extern forkjoin task Read(input logic[7:0] raddr);
extern forkjoin task Write(input logic[7:0] waddr);

modport slave( input req, addr, mode, start, clk,
               output gnt, rdy,
               inout data,
               export task Read(),
                   task Write());
    // export from module that uses the modport

modport master(input gnt, rdy, clk,
               output req, addr, mode, start,
               inout data,
               import task Read(input logic[7:0] raddr),
                   task Write(input logic[7:0] waddr));
    // import requires the full task prototype

initial begin
    slaves = 0;
    countSlaves;
    $display ("number of slaves = %d", slaves);
end

endinterface: simple_bus

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

    task a.countSlaves;
        a.slaves++;
    endtask

    task a.Read; // Read method
        avail = 0;
        ...
        avail = 1;
    endtask

    task a.Write;
        avail = 0;
        ...
        avail = 1;
    endtask

endmodule

module cpuMod(interface b);
    enum {read, write} instr;
    logic [7:0] raddr;

    always @(posedge b.clk)
        if (instr == read)
            b.Read(raddr); // call the slave method via the interface
            // ...
        else
            b.Write(raddr);

endmodule

```

```

module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

    memMod mem1(sb_intf.slave); //exports the countSlaves, Read and Write tasks
    memMod mem2(sb_intf.slave); //exports the countSlaves, Read and Write tasks
    cpuMod cpu(sb_intf.master); //imports the Read and Write tasks

endmodule

```

13.6 Parameterized interfaces

Interface definitions can take advantage of parameters and parameter redefinition, in the same manner as module definitions. This example shows how to use parameters in interface definitions.

```

interface simple_bus #(parameter AWIDTH = 8, DWIDTH = 8;)
    (input bit clk); // Define the interface

    logic req, gnt;
    logic [AWIDTH-1:0] addr;
    logic [DWIDTH-1:0] data;
    logic [1:0] mode;
    logic start, rdy;

    modport slave( input req, addr, mode, start, clk,
                  output gnt, rdy,
                  inout data,
                  import task slaveRead(),
                      task slaveWrite());
    // import into module that uses the modport

    modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  inout data,
                  import task masterRead(input logic[AWIDTH-1:0] raddr),
                      task masterWrite(input logic[AWIDTH-1:0] waddr));
    // import requires the full task prototype

    task masterRead(input logic[AWIDTH-1:0] raddr); // masterRead method
    ...
endtask

    task slaveRead; // slaveRead method
    ...
endtask

    task masterWrite(input logic[AWIDTH-1:0] waddr);
    ...
endtask

    task slaveWrite;
    ...
endtask

endinterface: simple_bus

module memMod(interface a); // Uses just the interface keyword

```

```

    logic avail;

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

    always @(b.start)
        if (a.mode[0] == 1'b0)
            a.slaveRead;
        else
            a.slaveWrite;
    endmodule

module cpuMod(interface b);
    enum {read, write} instr;
    logic [7:0] raddr;

    always @(posedge b.clk)
        if (instr == read)
            b.masterRead(raddr); // call the Interface method
            // ...
        else
            b.masterWrite(raddr);

    endmodule

module top;

    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate default interface
    simple_bus #(DWIDTH(16)) wide_intf(clk); // Interface with 16-bit data

    initial repeat(10) #10 clk++;

    memMod mem(sb_intf.slave); // only has access to the slaveRead task
    cpuMod cpu(sb_intf.master); // only has access to the masterRead task

    memMod memW(wide_intf.slave); // 16-bit wide memory
    cpuMod cpuW(wide_intf.master); // 16-bit wide cpu

    endmodule

```

13.7 Access without Ports

In addition to interfaces being used to connect two or more modules, the interface object/method paradigm allows for interfaces to be instantiated directly as static data objects within a module. If the methods are used to access internal state information about the interface, then these methods may be called from different points in the design to share information.

This example is equivalent to the SystemC example of “portless channel access”.

per mtg. 19

```

interface intf_mutex;

    task lock ();
        ...
    endtask

    function unlock();

```

```
    ...
    endfunction
endinterface

function int f(input int i);
    return(i); // just returns arg
endfunction

function int g(input int i);
    return(i); // just returns arg
endfunction

module mod1(input int in, output int out);

    intf_mutex mutex;

    always begin
        #10 mutex.lock();
        @(in) out = f(in);
        mutex.unlock;
    end

    always begin
        #10 mutex.lock();
        @(in) out = g(in);
        mutex.unlock;
    end
end
```

endmodule

Section 14 Parameters

Editor's note: Except as noted in a margin, changes in this draft are non-technical clean-up by Stu, for clarity.

14.1 Introduction (informative)

Verilog-2001 has parameters, [localparams](#) and [specparams](#), which are typically used either for controlling the dimensions of arrays, or for controlling delays. The parameter values can be set in three ways. They must be given a default value when declared. ~~This~~ The default value of parameters can be overridden by the instantiation of the module, and this in turn can be overridden by a **defparam** statement. The latter can have a hierarchical name so that it does not need to be in the same module as the instantiation.

SystemVerilog extends the Verilog **parameter** to include **type**. This provides polymorphism for modules and interfaces.

14.2 Syntax

[BNF excerpt to be inserted after BNF is approved]

Syntax 14-21—Parameter declaration syntax

A module or an interface can have parameters, which are set during elaboration and are constant during simulation. They are defined with data types and default values. With SystemVerilog, if no data type is supplied, parameters default to type **logic** of arbitrary size for Verilog-2001 compatibility and interoperability.

SystemVerilog adds the ability for a parameter to also specify a data type, allowing modules or instances to have data whose type is set for each instance.

```
module ma    #( parameter p1 = 1; parameter type p2 = shortint; )
              (input logic [p1:0] i, output logic [p1:0] o);
    p2 j = 0; // type of j is set by a parameter, which is shortint unless
redefined
    always @(i) begin
        o = i;
        j++;
    end
endmodule
```

```
module mb;
  logic [3:0] i,o;
  ma #(.p1(3), .p2(int)) u1(i,o); //redefines p2 to a type of int
endmodule
```


Section 15

Configuration libraries

15.1 Introduction (informative)

Verilog-2001 provides the ability to specify design configurations, which specify the binding information of module instances to specific Verilog HDL source code. Configurations utilize *libraries*. A library is a collection of modules, primitives and other configurations. Separate *library map files* specify the source code location for the cells contained within the libraries. The names of the library map files is typically specified as invocation options to simulators or other software tools reading in Verilog source code.

SystemVerilog adds support for interfaces to Verilog configurations. SystemVerilog also provides an alternate method for specifying the names of library map files.

15.2 Libraries

A library is a named collection of cells. A cell is a module, macromodule, primitive, interface, or configuration. A configuration is a specification of which source files bind to each instance in the design.

15.3 Library map files

Verilog 2001 specifies that library declarations, include statements, and config declarations are normally in a mapping file that is read first by a simulator or other software tool. SystemVerilog does not require a special library map file. *Instead*, the mapping information can be specified in the **\$root** top level.

Section 16

System tasks and system functions

Editor's note: Except as noted in a margin, changes in this draft are non-technical clean-up by Stu, for clarity.

16.1 Introduction (informative)

SystemVerilog adds a system function to determine the bit size of a value.

16.2 The \$bits system function

The \$bits system function returns the number of bits required to hold a value. A 4 state value counts as one bit, so \$bits on an integer will return 32. Given the declaration `logic [31:0] foo;`, `$bits(foo)` will return 32, even if a tool may require more than 32-bits of storage to represent the 4-state values.

Section 17

Compiler Directives

Editor's note: Except as noted in a margin, changes in this draft are non-technical clean-up by Stu, for clarity.

17.1 Introduction (informative)

Verilog-2001 provides the ``define` text substitution macro compiler directive. A macro can contain arguments, whose values can be set for each instance of the macro. For example:

```
`define NAND(dval) nand #(dval)

`NAND(3)      i1 (y, a, b); //`NAND(3) macro substitutes with: nand #(3)

`NAND(3:4:5)  i2 (o, c, d); //`NAND(3:4:5) macro substitutes with: nand
#(3:4:5)
```

SystemVerilog enhances the capabilities of the ``define` compiler directive to support strings as macro arguments

17.2 ``define` macros

In SystemVerilog, the ``define` macro text can include a backslash (\) at the end of a line to show continuation on the next line.

The macro text can also include an isolated quote, which must be preceded by a back tick, ``"`. This allows macro arguments to be included in strings. If the strings are to contain `\`, the macro text should be written ``\"`. Otherwise, the backslash will be treated as the start of an escaped identifier.

The macro text can also include a double back tick, `````, to allow identifiers to be constructed from arguments, e.g.

```
`define foo(f) f``_suffix
```

This expands:

```
foo(bar)
```

to:

```
bar_suffix
```

Note that there must be no space before the parenthesis. Otherwise it is treated as macro text.

The ``include` directive can be followed by a macro instead of a literal string:

```
`define f1 "/home/foo/myfile"
`include `f1
```

Section 18

Assertions

Editor's note: This section is currently in a separate document, and will be added once approved.

~~Section 19~~

~~Recommended items for deprecation~~

Editor's note: This section deferred to SystemVerilog 3.1, per meeting 19.

Annex A

Formal Syntax

(Normative)

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

- Keywords and punctuation are in **bold** text.
- Syntactic categories are named in non-bold text.
- Italics do not form part of the category, so *type_identifier* has the same syntax as identifier
- A vertical bar (|) separates alternatives.
- Square brackets ([]) enclose optional items.
- Braces ({ }) enclose items which may be repeated zero or more times.

Editor's note: The following BNF is a subset of the IEEE 1364 Verilog-2001 BNF, and reflects what is different between SystemVerilog and Verilog-2001. A complete BNF is being created, and will replace this section once approved.

A.1 Source text

source_text ::= [unit] [precision] {declaration_or_statement}

declaration_or_statement ::=
 library_declaration
 | include_statement
 | config_declaration
 | module_declaration
 | interface_declaration
 | task_declaration
 | function_declaration
 | udp_declaration
 | module_instantiation
 | interface_instantiation
 | event_declaration
 | net_declaration
 | data_declaration
 | statement

library_declaration ::=
 library library_identifier file_path_spec ;

include_statement ::=
 include file_path_spec ;

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

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

```

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

cell_clause ::= cell library_identifier . cell_identifier
inst_clause ::= instance name
liblist_clause ::= liblist { library_identifier }
use_clause ::= use [library_identifier.] cell_identifier
module_declaration ::=
    module_keyword identifier [parameter_port_list]
        [( port_list )] ; [unit] [precision]
        { module_item } endmodule
    | module_keyword identifier [parameter_port_list]
        [( port_decls )] ; [unit] [precision]
        { non_port_module_item } endmodule

module_keyword ::= module | macromodule
port_list ::= port { , port }
port ::=
    port_expression
    | . identifier ( [port_expression] )
port_decls ::= port_declaration { , port_declaration }
port_declaration ::=
    attribute_instance port_declaration
    | direction [port_type] variables
    | interface variables
    | interface . identifier variables
    | identifier variables
    | identifier . identifier variables
    | direction . identifier ( port_expression )

direction ::= input | output | inout
port_type ::=
    data_type { const_range }
    | net_type [signing] { packed_dimension }
    | event

port_expression ::=
    indexed_identifier
    | { indexed_identifier { , indexed_identifier } }

module_item ::=
    io_declaration
    | non_port_module_item
non_port_module_item ::=
    attribute_instance non_port_module_item
    | module_or_generate_item
    | module_declaration
    | interface_declaration
    | parameter_declaration
    | specparam_declaration
    | specify_block

```

```

module_or_generate_item ::=
    event_declaration
    | module_instantiation
    | primitive_instantiation
    | interface_or_generate_item
    | generated_instantiation
interface_declaration ::=
    interface identifier [parameter_port_list] ; [ ( port_list ) ] ; [unit][precision]
        { interface_item }
    endinterface [: identifier]
    | interface identifier [parameter_port_list] ;
        [ ( port_decls ) ] ; [unit][precision]
        { non_port_interface_item }
    endinterface [: identifier]
interface_item ::=
    io_declaration
    | non_port_interface_item
non_port_interface_item ::=
    attribute_instance non_port_interface_item
    | interface_declaration
    | parameter_declaration
    | specparam_declaration
    | interface_or_generate_item
    | generated_instantiation
interface_or_generate_item ::=
    event_declaration
    | net_declaration
    | interface_instantiation
    | data_declaration
    | task_declaration
    | function_declaration
    | modport_declaration
    | initial_statement
    | always_statement
    | continuous_assign
parameter_port_list ::=
    # ( parameter_declaration { parameter_declaration } )
unit ::= [ timeunit [time_literal] ; ]
precision ::= [ timeprecision [time_literal] ; ]
modport_declaration ::= modport modport_item { , modport_item } ;
modport_item ::= identifier ( modport_port { , modport_port } )
modport_port ::=
    [direction] [port_type] identifier
    | identifier . identifier

```

A.2 Data, Event and Net declarations

```

parameter_declaration ::=
    parameter [data_type] initial_assignments ;
    | parameter [signing] {packed_dimension} initial_assignments ;
    | parameter type type_assignments ;

```

```

specparam_declaration ::=
    param [data_type] initial_assignments ;
    | param [signing] {packed_dimension} initial_assignments ;
param ::= localparam | specparam
net_declaration ::=
    net_type [strength] [vector_or_scalar] [signing] {packed_dimension}
        [delay_values] vars_or_assigns ;
event_declaration ::= event variables ;
net_type ::= wire | wand | wor | supply0 | supply1 | tri | tri0 | tri1 | triand | trior | triereg
strength ::=
    ( strength0 , strength1 )
    | ( strength1 , strength0 )
    | ( charge_strength )
strength0 ::= supply0 | strong0 | pull0 | weak0 | highz0
strength1 ::= supply1 | strong1 | pull1 | weak1 | highz1
charge_strength ::= large | medium | small
vector_or_scalar ::= vectored | scalared
delay_values ::=
    # delay_value
    | # ( delay_value [, delay_value [, delay_value]] )
initial_assignments ::= initial_assignment { , initial_assignment }
initial_assignment ::= identifier = expression
type_assignments ::= type_assignment { , type_assignment }
type_assignment ::= identifier = data_type
data_declaration ::=
    variable_declaration
    | constant_declaration
    | type_declaration
    | state_declaration
variable_declaration ::= [ lifetime ] data_type vars_or_assigns ;
lifetime ::= static | automatic
constant_declaration ::= const data_type initial_assignments ;
type_declaration ::=
    typedef data_type variable ;
    | typedef identifier {[constant_expression]} . type_identifier variable ;
state_declaration ::= state { state_list } identifier [ delay_or_event ] ;
state_list ::= and_states | or_states
and_states ::= state and state { and state }
or_states ::= state , state { , state }
state ::= [ { state_list } ] identifier
vars_or_assigns ::= var_or_assign { , var_or_assign }
var_or_assign ::= variable [ = constant_expression ]
variables ::= variable { , variable }
variable ::= identifier { unpacked_dimension }
unpacked_dimension ::= packed_dimension
simple_type ::= integer_type | non_integer_type | type_identifier

```

A.3 Tasks and Functions

```

task_declaration ::=
    task [automatic] tf_name ;
        { task_item_declaration } { statement }
    endtask [: identifier]
| task [automatic] tf_name ( task_formals ) ;
        { data_declaration } { statement }
    endtask [: identifier]

task_item_declaration ::=
    direction [ data_type ] variables ;
| data_declaration

task_formals ::= [ task_formal { , task_formal } ]

task_formal ::=
    [port] [direction] [data_type] variable
| port event variable

task_prototype ::= task ( [ task_proto_formal { , task_proto_formal } )
named_task_proto ::= task identifier ( [ task_proto_formal { , task_proto_formal } )
task_proto_formal ::=
    [port] direction data_type [variable]
| port event variable

function_declaration ::=
    function [automatic] data_type tf_name ;
        { fn_item_declaration }
        { statement }
    endfunction [: identifier : ]
| function [automatic] data_type tf_name ( fn_formals ) ;
        { data_declaration }
        { statement }
    endfunction [: identifier : ]

fn_item_declaration ::=
    [direction] [ data_type ] variables ;
| data_declaration

tf_name ::= identifier [ . identifier ]

fn_formals ::= [ fn_formal { , fn_formal } ]

fn_formal ::= [direction] [data_type] variable

fn_prototype ::= function data_type ( fn_proto_formals )
named_fn_proto ::= function data_type identifier ( fn_proto_formals )
fn_proto_formals ::= [ fn_proto_formal { , fn_proto_formal } ]
fn_proto_formal ::=
    [direction] data_type [variable]
| variable

```

A.4 Data Types

```

data_type ::=
    integer_type [signing] {packed_dimension}
| type_identifier {packed_dimension}
| non_integer_type

```

```

    | struct { { struct_union_member } }
    | union { { struct_union_member } }
    | enum { enum_member }
    | void

integer_type ::= bit | logic | reg | byte | char | shortint | int | longint | integer
non_integer_type ::= time | shortreal | real | $built-in
signing ::= [ signed ] | [ unsigned ]
packed_dimension ::= [ constant_expression : constant_expression ]
struct_union_member ::= data_type variables ;
enum_member ::=
    identifier
    | identifier = constant_expression

```

A.5 Instantiations

```

generated_instantiation ::= generate { generate_item } endgenerate
generate_item_or_null ::= generate_item | ;
generate_item ::=
    generate_conditional_statement
    | generate_case_statement
    | generate_loop_statement
    | generate_block
    | module_or_generate_item /* if in module */
    | interface_or_generate_item /* if in interface */
generate_conditional_statement ::=
    if ( constant_expression ) generate_item_or_null
    [ else generate_item_or_null ]
generate_case_statement ::=
    case ( constant_expression )
        generate_case_item { generate_case_item }
    endcase
generate_case_item ::=
    constant_expression { , constant_expression : generate_item_or_null }
generate_loop_statement ::=
    for ( genvar_decl_or_assign ; expression ; genvar_expr_or_assign )
        generate_named_block
genvar_decl_or_assign ::= [ genvar ] identifier = constant_expression
genvar_expr_or_assign ::= unary_expression | operator_assignment
generate_named_block ::=
    begin : identifier { generate_item } end
    | identifier : generate_block
generate_block ::= begin [ : identifier ] { generate_item } end
module_instantiation ::=
    module_identifier [ parameter_values ] named_instance { , named_instance } ;
interface_instantiation ::=
    interface_identifier [ parameter_values ] named_instance { , named_instance } ;
udp_instantiation ::=
    identifier [strength] [delay_values] primitive_instance { ; primitive_instance } ;

```

```

gate_instantiation ::=
    gate [strength] [delay_values] primitive_instance { ; primitive_instance } ;

gate ::=
    and | nand | or | nor | xor | xnor | buf* | not | bufif0 | bufif1 | notif0 | notif1
    | cmos | remos | nmos | rnmos | pmos | rpmos
    | tranif0 | rtranif0 | tranif1 | rtranif1 | tran | rtran

parameter_values ::=
    # ( expression { , expression } )
    | # ( named_param_val { , named_param_val } )

named_param_val ::= . identifier ( expression )

named_instance ::= identifier { [ expression : expression ] } [ ( port_connections ) ]

primitive_instance ::= [ identifier ] { [ expression : expression ] } [ ( port_connections ) ]

port_connections ::=
    [ expression ] { , [expression] }
    | named_port_connection { , named_port_connection }

named_port_connection ::= . identifier ( expression )

```

A.6 Procedural Statements

```

initial_statement ::= initial statement
always_statement ::= always statement
combinational_statement ::= always_comb statement
latch_statement ::= always_latch statement
ff_statement ::= always_ff statement
statement_or_null ::= statement | ;
statement ::=
    blocking_assignment ;
    | non_blocking_assignment ;
    | selection
    | loop
    | jump
    | delay_control statement_or_null
    | event_control statement_or_null
    | wait ( expression ) statement_or_null
    | process statement
    | disable name ;
    | sequential_block
    | parallel_block
    | -> event_name ;
    | transition_to_state statement_or_null
    | expression ;
    | proc_continuous_assign ;
    | identifier : statement

selection ::=
    [ up ] if ( expression ) statement_or_null
    [ else statement_or_null ]
    | [ up ] case ( expression ) case_item { case_item } endcase
    | transition ( name ) transition { transition } endtransition

up ::= unique | priority

```

```

case ::= case | casez | casex
loop ::=
    forever statement
    | repeat ( expression ) statement_or_null
    | while ( expression ) statement_or_null
    | for ( [declare_or_assign] ; [expression] ; [expression_or_assign] ) statement_or_null
    | do statement while ( expression )
jump ::=
    return [ expression ] ;
    | break ;
    | continue ;
declare_or_assign ::=
    lvalue = expression
    | data_type identifier = expression
declare_or_exp ::=
    unary_expression
    | data_type identifier
blocking_assignment ::=
    operator_assignment
    | lvalue = delay_or_event expression
operator_assignment ::= lvalue assignment_operator expression
assignment_operator ::= = | *= | /= | %= | += | -= | <<= | >>= | &= | ^= | |=
non_blocking_assignment ::= lvalue <= [ delay_or_event ] expression
delay_or_event ::=
    delay_control
    | [ repeat ( expression ) ] event_control
case_item ::=
    expression { , expression } : statement_or_null
    | default [ : ] statement_or_null
transition ::=
    state_conditions : statement_or_null
    | default [ : ] statement_or_null
state_conditions ::= state_condition { , state_condition }
state_condition ::= state_identifier { and state_identifier }
sequential_block ::= begin [ : identifier ] { statement } end [ : identifier]
parallel_block ::= fork [ : identifier ] { statement } join [ : identifier]
transition_to_state ::=
    ->> machine_name . state_identifier
    | ->> machine_name . ( state_condition )
    | [transition_identifier] ->> state_condition
proc_continuous_assign ::=
    assign name_or_names = expression
    | deassign name
    | force name_or_names = expression
    | release name

```

A.7 Names

```
name_or_names ::= name | { name { , name } }
```



```
name ::= indexed_identifier { . indexed_identifier }
indexed_identifier ::= identifier [ [ constant_expression [ : constant_expression ] ] ]
```

A.8 Delay and Event Controls

```
delay_control ::=
    # number
    | # name
    | # ( expression )
event_control ::=
    @ name
    | @ ( event_expressions )
    | @*
event_expressions ::=
    event_expression { or event_expression }
    | event_expression { , event_expression }
event_expression ::=
    [ edge ] expression [ iff expression ]
    | ( [ edge ] expression [ iff expression ] )
edge ::= posedge | negedge | changed
```

A.9 Expressions

```
expression ::=
    unary_expression
    | expression binary_operator expression
    | expression ? expression : expression
    | ( operator_assignment )
unary_expression ::= [unary_operator] lvalue
unary_operator ::= + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~ | bump_operator
lvalue ::= postfix_expression
postfix_expression ::=
    primary [ bump_operator ]
    | postfix_expression . identifier
    | postfix_expression -> identifier
    | postfix_expression [ expression ]
    | postfix_expression [ expression : expression ]
    | postfix_expression ( [ expression { , expression } ] )
    | postfix_expression bump_operator
bump_operator ::= ++ | --
binary_operator ::=
    + | - | * | / | % | == | != | === | !== | < | <= | > | >= | << | >> | <<< | >>> | && | || | & | | | ^ | ~^ | ^~
primary ::=
    identifier
    | literal
    | data_type
    | ( expression )
    | { expression { , expression } }
    | { expression { expression } }
    | simple_type '(expression )
```

```
| simple_type '{ expression { , expression } }
| simple_type '{ expression { expression } }
```

A.10 Literals

literal ::=

```
    string_literal
  | number
  | time_literal
  | '0' | '1' | 'z' | 'Z' | 'x' | 'X'
```

string_literal ::= " value "

number ::=

```
    integer
  | [integer] ' base value
  | real
```

base ::= 'b | 'd | 'h | 'o | 'sb | 'sd | 'sh | 'so

time_literal ::= integer [. integer] time_unit

time_unit ::= s | ms | us | ns | ps | fs

Annex B

Keywords

SystemVerilog reserves the following keywords:

always	endtable	module	small
always_comb [†]	endtask	nand	specify
always_ff [†]	endtransition [†]	negedge	specparam
always_latch [†]	enum [†]	nmos	state [†]
and	event	nor	static [†]
assign	export [†]	noshowcancelled	strong0
automatic	extern [†]	not	strong1
begin	for	notif0	struct [†]
bit [†]	force	notif1	supply0
break [†]	forever	or	supply1
buf	fork	output	table
bufif0	forkjoin [†]	packed [†]	task
bufif1	function	parameter	time
byte [†]	generate	pmos	timeprecision [†]
case	genvar	posedge	timeunit [†]
casex	highz0	primitive	tran
casez	highz1	process [†]	tranif0
cell	if	priority [†]	tranif1
changed [†]	iff [†]	pull0	transition [†]
char [†]	ifnone	pull1	tri
class [†]	import [†]	pulldown	tri0
cmos	incdir	pullup	tril
config	include	pulsetyle_oneevent	triand
const [†]	initial	pulsetyle_ondetect	trior
continue [†]	inout	rcmos	trireg
deassign	input	real	type [†]
default	instance	realtime	typedef [†]
defparam	int [†]	reg	union [†]
design	integer	release	unique [†]
disable	interface [†]	repeat	unsigned
do [†]	join	return	use
else	large	rnmos	vectored
end	liblist	rpmos	wait
endcase	library	rtran	wand
endconfig	localparam	rtranif0	weak0
endfunction	logic [†]	rtranif1	weak1
endgenerate	longint [†]	scalared	while
endinterface [†]	longreal [†]	shortint [†]	wire
endmodule	macromodule	shortreal [†]	wor
endprimitive	medium	showcancelled	xnor
endspecify	modport [†]	signed	xor

[†] keywords not in the Verilog-2001 standard

Annex C

Glossary

Editor's note: Stu to provide a list of terms for the glossary. Peter to write the definitions of the terms.

Annex D

Bibliography

Editor's note: This annex to be completed before the final draft.

Index

Symbols

\$bits 10, 77
\$root 47, 48
%= operator 19
&= operator 19
*= operator 19
++ operator 20
+= operator 19
/= operator 19
<<<= operator 19
<<= operator 19
-- operator 20
-= operator 19
>>= operator 19
>>>= operator 19
\ line continuation 78
\a bell 3
\f form feed 3
\v vertical tab 3
\x02 hex number 3
^= operator 19
'timescale 7
|= operator 19
" isolated quote 78
" double back tick 78
'define 78
'timescale 53
' cast operator 9

Numerics

2-state types 7
4-state types 7

A

always @* 40
always_comb 39, 40
always_ff 40
always_latch 39, 40
array literals 3
array part selects 12
array slices 12
arrays 11
assign 16
assignment operators 18, 19
assignments in expressions 19
attributes 17
automatic 14, 15, 16, 43
automatic tasks 44

B

bell 3
bit 5, 6, 7

block name 27, 28
blocking assignments 24
break 23, 27, 28
byte 6, 7

C

casting 9
changed 29
char 6, 7
combinational logic 39
concatenation 22
configurations 75
const 14, 15
constants 15
continue 23, 27, 28
continuous assignment 40

D

data declarations 14
data types 5
decrementor operator 18, 20
defparam 73
disable 28
do...while loop 23, 26
double 7
dynamic processes 28, 39, 41

E

elaboration 47
enum 8, 9
enumerated types 8

F

float 7
force 16
fork...join processes 41
form feed 3
functions 45

G

goto 27

H

hierarchical names 55

I

iff 29
incrementor operator 18, 20
int 5, 6, 7
integer 6, 7
integer literals 2
interface 17
introduction to SystemVerilog 1

L

labels 27, 28

latched logic 40
libraries 75
library map files 75
literal values 2
localparam 73
logic 5, 6, 7, 16
longint 5, 6, 7

M

multiple dimension arrays 11

N

name space 54
named blocks 27
nested identifiers 55
nested modules 49, 50
nonblocking assignments 24

O

operator associativity 21
operator precedence 21
overview of SystemVerilog 1

P

packed arrays 11, 12, 20
parameter 73
part selects 12
port declarations 51, 52, 53, 54
precedence 21
priority 25, 26
process 28, 39, 41
process labels 41

R

real 2, 5, 7, 20
real literals 2
reg 5, 6, 7
return 23, 27, 28, 44, 46

S

scalared 11
sequential logic 40
shortint 6, 7
shortreal 2, 5, 7, 20
signed types 7
slices 12
specparam 73
statement labels 27
static 14, 15, 16, 43
static processes 28, 39
static tasks 44
string literals 3
struct 9
structure literals 3
structures 9
SystemVerilog, overview 1

SystemVerilog, version numbers 1

T

tasks 43
time literals 3
time unit 3
timeprecision 7, 53
timeunit 7, 52
top level 47
typedef 5, 8

U

union 9
unions 9
unique 25, 26
unpacked arrays 11, 12
unsigned types 7
unsized literals 2
user-defined types 8

V

variable initialization 15
vectored 11
Verilog, version numbers 1
version numbers 1
vertical tab 3
void 8
void functions 43, 46

W

while 23, 26